TURING 图灵程序设计丛书 Web开发系列

Apress®

PHP Objects, Patterns, and Practice Third Edition

深入PHP
面向对象、模式与实践
（第3版）

[美] Matt Zandstra 著
陈 浩 吴孙滨
胡 丹 李 静 译
陈 浩 审校

- 雅虎公司PHP专家力作
- PHP模块化编程必读
- 完美结合面向对象思想

人民邮电出版社
POSTS & TELECOM PRESS

Matt Zandstra

从事互联网应用开发十余年，目前是雅虎公司工程师，从事核心模板管理系统的开发。他是一位多产的技术作家，除本书外另著有*Sams Teach Yourself PHP in 24 Hours*等书，他还为*Linux Magazine*、*php|architect*杂志、IBM DeveloperWorks、Zend.com和bdz-consult.com写过许多PHP方面的文章。

"本书正是为PHP企业级开发人员和软件工程师量身打造的经典著作，每一位想提高技能的PHP开发人员都应阅读。"

——Ken Guest，爱尔兰PHP用户组主席，PEAR QA团队和PEAR组的成员

"这是我读过的PHP书中最好的一本，它介绍了面向对象编程的方方面面。我自学PHP已经5年了，遇到的大多数问题都能在这本书中找到答案。强烈推荐那些有一定PHP经验并想更上一层楼的程序员阅读。"

——Amazon.com读者评论

PHP Objects, Patterns, and Practice Third Edition

深入PHP
面向对象、模式与实践（第3版）

本书全面深入地剖析了面向对象的PHP编程与设计。书中首先介绍了PHP的对象特性（包括抽象类、反射、接口和错误处理等）及可帮助开发人员了解类、对象和方法的对象工具，然后介绍了设计模式，阐述了模式的概念，展示了如何在PHP中实现一些关键的模式，并用专门的章节介绍了企业模式和数据库模式。最后，本书围绕PHP应用程序开发，详细介绍了一批极为实用的辅助开发工具，讨论了具有普遍意义的最佳开发实践。另外，这一版中还新增了闭包、命名空间、持续集成等内容。

本书适合每位PHP开发人员进阶参考，可帮助他们掌握PHP面向对象设计和开发的精髓，并最终跻身高端PHP开发人员之列。

Apress®

图灵网站：www.turingbook.com 热线：(010)51095186转604
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议 计算机/网络技术/PHP

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-25624-9

定价：69.00元

TURING 图灵程序设计丛书 Web开发系列

PHP Objects, Patterns, and Practice Third Edition

深入PHP
面向对象、模式与实践

（第3版）

[美] Matt Zandstra 著

陈 浩 吴孙滨
胡 丹 李 静 译

陈 浩 审校

人民邮电出版社

北京

图书在版编目（CIP）数据

深入PHP ：面向对象、模式与实践 ：第3版 / （美）
赞德斯彻（Zandstra,M.）著 ；陈浩等译. -- 北京 ：人
民邮电出版社，2011.7
（图灵程序设计丛书）
书名原文：PHP Objects, Patterns, and Practice,
Third Edition
ISBN 978-7-115-25624-9

Ⅰ. ①深… Ⅱ. ①赞… ②陈… Ⅲ. ①PHP语言—程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2011)第106171号

内 容 提 要

本书是PHP专家经典力作的最新版本。书中主要介绍了如何使用面向对象技术和设计模式编写稳定的、可维护的代码，如何使用Subversion管理多个开发人员，如何使用Phing和PEAR进行构建和安装，以及将构建和测试过程自动化的策略，包括持续集成。

本书适合中高级PHP程序员阅读。

图灵程序设计丛书

深入PHP：面向对象、模式与实践（第3版）

◆ 著　　[美] Matt Zandstra
　译　　陈　浩　吴孙滨　胡　丹　李　静
　审　校　陈　浩
　责任编辑　卢秀丽
　执行编辑　毛倩倩

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
　邮编　100061　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
　印张：29
　字数：685千字　2011年7月第1版
　印数：1－3 000册　2011年7月北京第1次印刷

著作权合同登记号　图字：01-2011-0613号

ISBN 978-7-115-25624-9

定价：69.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版权声明

Original English language edition, entitled *PHP Objects, Patterns, and Practice, Third Edition* by Matt Zandstra, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

Copyright © 2010 by Matt Zandstra. Simplified Chinese-language edition copyright © 2011 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

译　者　序

译者从接触PHP开发到现在接近10年，参与了不少大型互联网应用的开发。在此过程中，有几个问题是译者一直在关注的。

中国互联网的发展如日中天，全国互联网用户数量增长很快（已达3.5亿人）。当前较为流行的站点往往每天要面对至少上百万的访问者，而且需要提供越来越复杂的服务。

作为研发者的我们，应该如何设计站点，让站点能满足当前阶段的业务需求（访问量和服务），并且保证网站有良好的架构设计，方便未来的后续开发和维护？我们应该如何解决代码的“耦合”问题，如何编写出易重用的代码，甚至达到理想化的软件层面的“美学”境界（可能是程序员的终极目标）？

如何建立一套行之有效的团队协作机制，使研发团队日常的开发标准化，提高团队沟通和合作的效率？是否有一些放之四海而皆准的准则或通用的工具可以利用？

正所谓“水涨船高”，PHP这种为互联网而生的编程语言这些年来也一直在进步。如何充分利用PHP中的新特性，使日常的开发更为高效？

读者可以从本书中找到上述问题的部分答案。因为本书不仅仅是一本PHP图书，更是一本鼓励和促进读者超越代码层次，对整个站点的“设计”进行思考的书。

条条大路通罗马。要设计出漂亮又实用的站点架构，也许每个人都有自己的体会和经验，但就PHP站点而言，从根本上不外乎“面向过程”和“面向对象”两种比较底层的开发思路。对于面向过程的开发方式，每个人或每个团队都有自己的开发习惯，不太容易形成系统，所以这么多年来虽然有不少优秀的面向过程的PHP程序面世，但没能形成任何可供共享的“框架”或“思想”，而面向对象的开发方式则积累了大量公认的模式和框架。从这个角度来说，建议读者朋友多花时间在面向对象设计上。本书就“面向对象”在PHP中的实现做了非常深入的介绍，涵盖了基础、高级知识以及设计模式的实现。

从学习角度来说，译者推荐PHP程序员沿这样的学习路线前进：PHP基础入门（语法、常用函数和扩展等）→面向对象的PHP（本书就是很好的学习资料）→网站软件架构设计（设计模式、框架等）→网站物理层次架构设计（分布式计算、存储、负载均衡、高可用性等）。在现实当中，上述这些阶段和工作其实是交织在一起的。另外，程序设计之外的很多工具和实践经验，也极其重要。因为个人的能力有限，个人的价值必须在团队和企业当中得到体现。本书介绍的很多工具

（如版本控制工具、文档工具和应用构建工具等）都可以促进团队协作。

最后，真诚邀请读者朋友通过本书来探索PHP。不论是刚入门的PHP开发人员，还是有多年经验的同行，相信看完本书之后，你都会有所收获。当然，一个真正的程序员，他的思想永远是超越具体语言和工具的。期待有一天，读者朋友可以在了解书中这些知识的基础上，总结出属于自己的更加优秀的开发框架甚至是设计思想，开发出在互联网上受到用户欢迎的优秀站点。

本书在PHPEye开源社区设有读者反馈版块，网址为http://bbs.phpeye.com/forum-16-1.html。译者将通过这个版块为读者提供勘误和答疑。

陈浩（Haohappy）
2010年3月于上海

第3版简介

我最初有写这本书的想法时，就感觉自己是在逆流而行。由于语言的限制，PHP的很多模式实现更像是美化了的变通方法。当今，我们也许很难跟得上PHP对象、设计和项目实践的革新步伐。

如果说这是个问题，那么它正是你所期望的，特别是当你有工具在手可以驾驭它们呈现的风险和机会时。

PHP不断地实现面向对象开发人员所期望的功能。这一版中涉及了命名空间、延迟静态绑定（late static binding）、匿名函数和闭包等，此时你不必担心还不明白这些，本书将会一一讲述。PHP是一门活跃的语言，一直在不断地演进以满足用户的需求。

这给开发人员提出了一些有趣的挑战，尤其是他们在期望稳定的代码库和期望利用每次新发布的功能之间所存在的两难选择。有了一套良好的，特别是可自动运行的测试工具来协作，以及容易安装的系统，你就可以改善代码设计、使用新特性，而且可以十分确定不会搞坏什么。

我希望，这就是本书的意义。我想探讨在PHP语言和更广泛的面向对象设计领域中是什么令人振奋，同时也想谈及一些工具和实践（使用它们，你可以在对项目作出任何改变的时候防止出现一堆潜在的错误）。

除了PHP的新特性，这一版还介绍了使用Selenium的网络测试和持续集成服务器（这款工具可以运行测试、构建系统并对项目应用诊断工具）。

Web应用程序有多么真实？当然，它是存储在计算机中的代码行、位，同时也存在于服务器上的执行程序。其实，对于开发人员，应用程序首先存在于他的想象中。它是一种由能或多或少、优雅地互锁在一起的部分组合起来的结构。如果我们幸运，它就会被实现和部署，并在某个人使用它的瞬间变得鲜活起来。那儿，就是那儿，就是编码生活的魅力所在。

这就是本书的真正意义。本书会介绍如何将头脑中的点子变成现实，在这个过程中你会找到乐趣。本书还会介绍如何将你的想象力系统化，以及用代码实现该系统以后获得的满足感，以及系统可以真正工作时获得的另一份满足感。测试会让你随心所欲地去冒险，而想象力会激励你去冒险。正是在那一刻，你写出的东西将在他人眼里变得真实。

致　　谢

如果突然想写一本书（就像我在布莱顿一家咖啡厅喝咖啡时突然有了这个念头），可能你只想到了该写什么主题，那一刹那的冲动很容易让你忘记写书是多么不容易。我很快就重新认识到写书是非常辛苦的工作，并且不是光靠一个人单打独斗就可以完成的。在编写本书的每个阶段，我都得到了很多帮助。

实际上，我的致谢首先要追溯到还没有本书构想的阶段。本书的萌芽来自于我在布莱顿为某个技术组织做的一次报告，该组织名为Skillswap（www.skillswap.org），由Andy Budd管理。正是由于Andy邀请我做报告，我才开始有了总结PHP面向对象的想法。在此特别感谢Andy，我该请他喝上一杯。

很偶然地，我在那次会议上遇到了Apress出版社的一位作者Jessey White-Cinis，是她牵线让我与Apress出版社的Martin Streicher编辑建立了联系，而后者则正式邀请我为Apress写一本书。

谢谢Jessey和Martin，他们让本书的出版迈出了第一步。

写作中我面临了非常紧的时间期限，并曾因与家人搬到了新大陆而想到放弃，而那时Apress出版社的团队为我提供了很多帮助。

谢谢Steven Metsker，是他允许我将其在*Building Parsers in Java*一书中的解析器API代码改写成简化的PHP版本并放入本书。

写书期间，我一直没有太多时间照顾家庭。在此谢谢我的妻子Louise以及我们的孩子Holly和Jake。我一直都很想念他们。

自从本书第1版发行后，我收到了很多热情洋溢并富有建设性的读者反馈。我非常抱歉没能够一一回复，希望在此对你们一并表达我的谢意。

我在写第1版时常听John Peel[1]主持的节目。John 40年来一直在抨击烦乏无味粗制滥造的音乐作品，捍卫原创音乐，推崇兼收并蓄的风格。John于2004年10月突然去世，令无数喜欢他的听众怅然若失。他对很多人的生活产生了巨大影响，在此表示敬意。

① 他是BBC电台的资深DJ。——译者注

目　录

第一部分　介　　绍

第 1 章　PHP：设计与管理 ············ 2

1.1　问题 ············ 2
1.2　PHP 和其他语言 ············ 3
1.3　关于本书 ············ 5
1.3.1　对象 ············ 5
1.3.2　模式 ············ 5
1.3.3　实践 ············ 6
1.3.4　第 3 版新增内容 ············ 7
1.4　小结 ············ 7

第二部分　对　　象

第 2 章　PHP 与对象 ············ 10

2.1　PHP 对象的偶然成功 ············ 10
2.1.1　最初：PHP/FI ············ 10
2.1.2　语法糖：PHP 3 ············ 10
2.1.3　一场静悄悄的革命：PHP 4 ············ 11
2.1.4　拥抱改变：PHP 5 ············ 12
2.2　走向未来：PHP 6 ············ 13
2.3　拥护和疑虑：关于对象的争辩 ············ 13
2.4　小结 ············ 14

第 3 章　对象基础 ············ 15

3.1　类和对象 ············ 15
3.1.1　编写第一个类 ············ 15
3.1.2　第一个对象（或两个） ············ 16
3.2　设置类中的属性 ············ 17
3.3　使用方法 ············ 19
3.4　参数和类型 ············ 21
3.4.1　基本类型 ············ 22
3.4.2　获得提示：对象类型 ············ 24
3.5　继承 ············ 26
3.5.1　继承问题 ············ 26
3.5.2　使用继承 ············ 30
3.5.3　public、private、protected：管理类的访问 ············ 35
3.6　小结 ············ 39

第 4 章　高级特性 ············ 40

4.1　静态方法和属性 ············ 40
4.2　常量属性 ············ 43
4.3　抽象类 ············ 44
4.4　接口 ············ 46
4.5　延迟静态绑定：static 关键字 ············ 47
4.6　错误处理 ············ 50
4.7　Final 类和方法 ············ 56
4.8　使用拦截器 ············ 57
4.9　析构方法 ············ 61
4.10　使用__clone()复制对象 ············ 62
4.11　定义对象的字符串值 ············ 64
4.12　回调、匿名函数和闭包 ············ 65
4.13　小结 ············ 69

第 5 章　对象工具 ············ 70

5.1　PHP 和包 ············ 70

5.1.1 PHP包和命名空间 …… 70
5.1.2 自动加载 …… 78
5.2 类函数和对象函数 …… 79
5.2.1 查找类 …… 80
5.2.2 了解对象或类 …… 81
5.2.3 了解类中的方法 …… 82
5.2.4 了解类属性 …… 83
5.2.5 了解继承 …… 83
5.2.6 方法调用 …… 84
5.3 反射API …… 85
5.3.1 入门 …… 85
5.3.2 开始行动 …… 86
5.3.3 检查类 …… 88
5.3.4 检查方法 …… 89
5.3.5 检查方法参数 …… 91
5.3.6 使用反射API …… 92
5.4 小结 …… 95

第6章 对象与设计 …… 96

6.1 代码设计的定义 …… 96
6.2 面向对象设计和过程式编程 …… 97
6.2.1 职责 …… 100
6.2.2 内聚 …… 100
6.2.3 耦合 …… 101
6.2.4 正交 …… 101
6.3 选择类 …… 101
6.4 多态 …… 102
6.5 封装 …… 104
6.6 忘记细节 …… 105
6.7 4个方向标 …… 105
6.7.1 代码重复 …… 106
6.7.2 类知道的太多 …… 106
6.7.3 万能的类 …… 106
6.7.4 条件语句 …… 106
6.8 UML …… 106
6.8.1 类图 …… 107
6.8.2 时序图 …… 111
6.9 小结 …… 112

第三部分 模 式

第7章 什么是设计模式？为何使用它们 …… 114

7.1 什么是设计模式 …… 114
7.2 设计模式概览 …… 116
7.2.1 命名 …… 116
7.2.2 问题 …… 116
7.2.3 解决方案 …… 117
7.2.4 效果 …… 117
7.3 《设计模式》格式 …… 117
7.4 为什么使用设计模式 …… 118
7.4.1 一个设计模式定义了一个问题 …… 118
7.4.2 一个设计模式定义了一个解决方案 …… 118
7.4.3 设计模式是语言无关的 …… 118
7.4.4 模式定义了一组词汇 …… 118
7.4.5 模式是经过测试的 …… 119
7.4.6 模式是为协作而设计的 …… 119
7.4.7 设计模式促进良好设计 …… 119
7.5 PHP与设计模式 …… 120
7.6 小结 …… 120

第8章 模式原则 …… 121

8.1 模式的启示 …… 121
8.2 组合与继承 …… 122
8.2.1 问题 …… 122
8.2.2 使用组合 …… 124
8.3 解耦 …… 127
8.3.1 问题 …… 127
8.3.2 降低耦合 …… 128
8.4 针对接口编程，而不是针对实现编程 …… 130
8.5 变化的概念 …… 131
8.6 父子关系 …… 132
8.7 模式 …… 132
8.7.1 用于生成对象的模式 …… 133
8.7.2 用于组织对象和类的模式 …… 133
8.7.3 面向任务的模式 …… 133

8.7.4 企业模式 ······ 133
8.7.5 数据库模式 ······ 133
8.8 小结 ······ 133

第 9 章 生成对象 ······ 134

9.1 生成对象的问题和解决方法 ······ 134
9.2 单例模式 ······ 138
9.2.1 问题 ······ 138
9.2.2 实现 ······ 139
9.2.3 结果 ······ 141
9.3 工厂方法模式 ······ 141
9.3.1 问题 ······ 141
9.3.2 实现 ······ 143
9.3.3 结果 ······ 145
9.4 抽象工厂模式 ······ 146
9.4.1 问题 ······ 146
9.4.2 实现 ······ 147
9.4.3 结果 ······ 148
9.4.4 原型模式 ······ 150
9.4.5 问题 ······ 150
9.4.6 实现 ······ 151
9.5 某些模式的骗术 ······ 153
9.6 小结 ······ 154

第 10 章 让面向对象编程更加灵活的模式 ······ 155

10.1 构造可灵活创建对象的类 ······ 155
10.2 组合模式 ······ 155
10.2.1 问题 ······ 156
10.2.2 实现 ······ 158
10.2.3 效果 ······ 161
10.2.4 组合模式小结 ······ 164
10.3 装饰模式 ······ 165
10.3.1 问题 ······ 165
10.3.2 实现 ······ 167
10.3.3 效果 ······ 170
10.4 外观模式 ······ 170
10.4.1 问题 ······ 170
10.4.2 实现 ······ 172
10.4.3 效果 ······ 172
10.5 小结 ······ 173

第 11 章 执行及描述任务 ······ 174

11.1 解释器模式 ······ 174
11.1.1 问题 ······ 174
11.1.2 实现 ······ 175
11.1.3 解释器的问题 ······ 182
11.2 策略模式 ······ 183
11.2.1 问题 ······ 183
11.2.2 实现 ······ 184
11.3 观察者模式 ······ 187
11.4 访问者模式 ······ 194
11.4.1 问题 ······ 194
11.4.2 实现 ······ 195
11.4.3 访问者模式的问题 ······ 199
11.5 命令模式 ······ 200
11.5.1 问题 ······ 200
11.5.2 实现 ······ 200
11.6 小结 ······ 204

第 12 章 企业模式 ······ 205

12.1 架构概述 ······ 205
12.1.1 模式 ······ 205
12.1.2 应用程序和层 ······ 206
12.2 企业架构之外的基础模式 ······ 208
12.2.1 注册表 ······ 208
12.2.2 实现 ······ 209
12.3 表现层 ······ 217
12.3.1 前端控制器 ······ 217
12.3.2 应用控制器 ······ 227
12.3.3 页面控制器 ······ 237
12.3.4 模板视图和视图助手 ······ 242
12.4 业务逻辑层 ······ 245
12.4.1 事务脚本 ······ 245
12.4.2 领域模型 ······ 249
12.5 小结 ······ 252

第 13 章 数据库模式 ······ 253

13.1 数据层 ······ 253

13.2 数据映射器……253
13.2.1 问题……254
13.2.2 实现……254
13.2.3 效果……265
13.3 标识映射……266
13.3.1 问题……266
13.3.2 实现……266
13.3.3 效果……269
13.4 工作单元……269
13.4.1 问题……269
13.4.2 实现……270
13.4.3 效果……274
13.4.4 延迟加载……274
13.4.5 问题……274
13.4.6 实现……274
13.4.7 效果……276
13.5 领域对象工厂……276
13.5.1 问题……276
13.5.2 实现……277
13.5.3 效果……278
13.6 标识对象……279
13.6.1 问题……279
13.6.2 实现……280
13.6.3 效果……285
13.7 选择工厂和更新工厂模式……285
13.7.1 问题……285
13.7.2 实现……285
13.7.3 效果……288
13.8 数据映射器中剩下些什么……289
13.9 小结……291

第四部分 实 践

第 14 章 良好和糟糕的实践……294

14.1 超越代码……294
14.2 借一个轮子……295
14.3 合作愉快……296
14.4 为你的代码插上双翼……297
14.5 文档……297
14.6 测试……298
14.7 持续集成……299
14.8 小结……300

第 15 章 PEAR 和 Pyrus……301

15.1 什么是 PEAR……301
15.2 了解 Pyrus……302
15.3 安装 PEAR 包……303
15.4 使用 PEAR 包……306
15.5 创建自己的 PEAR 包……311
15.5.1 package.xml……311
15.5.2 package.xml 的组成……312
15.5.3 contents 元素……313
15.5.4 依赖……316
15.5.5 使用 phprelease 进行灵活的自定义安装……317
15.5.6 准备发布包……318
15.5.7 创建自己的 PEAR 频道……318
15.6 小结……323

第 16 章 用 phpDocumentor 生成文档……324

16.1 为什么要使用文档……324
16.2 安装……325
16.3 生成文档……325
16.4 DocBlock 注释……327
16.5 类的文档……328
16.6 文件的文档……330
16.7 属性的文档……330
16.8 方法的文档……331
16.9 在文档中创建链接……333
16.10 小结……335

第 17 章 使用 Subversion 进行版本控制……336

17.1 为什么要使用版本控制……336
17.2 获得 Subversion……337
17.3 配置 Subversion 代码库……338
17.4 开始项目……339
17.5 更新和提交……341
17.6 增加和删除文件及目录……344
17.6.1 添加文件……344

17.6.2 删除文件 ······ 345
17.6.3 添加目录 ······ 345
17.6.4 删除目录 ······ 345
17.7 标记和导出项目 ······ 345
17.7.1 标记项目 ······ 345
17.7.2 导出项目 ······ 346
17.8 创建项目分支 ······ 347
17.9 小结 ······ 350

第 18 章 使用 PHPUnit 进行测试 ······ 351
18.1 功能测试与单元测试 ······ 351
18.2 手工测试 ······ 352
18.3 引入 PHPUnit ······ 354
18.3.1 创建测试用例 ······ 354
18.3.2 断言方法 ······ 355
18.3.3 测试异常 ······ 356
18.3.4 运行测试套件 ······ 357
18.3.5 约束 ······ 358
18.3.6 模拟与桩 ······ 360
18.3.7 失败是成功之母 ······ 362
18.4 编写 Web 测试 ······ 365
18.4.1 为测试重构 Web 应用 ······ 366
18.4.2 简单的 Web 测试 ······ 368
18.4.3 Selenium ······ 370
18.5 警告 ······ 374
18.6 小结 ······ 376

第 19 章 用 Phing 实现项目的自动构建 ······ 377
19.1 什么是 Phing ······ 377
19.2 获取和安装 Phing ······ 378
19.3 编写 build 文档 ······ 378
19.3.1 目标 ······ 380
19.3.2 属性 ······ 382
19.3.3 类型 ······ 387
19.3.4 任务 ······ 391
19.4 小结 ······ 394

第五部分 结 论

第 20 章 持续集成 ······ 396
20.1 什么是持续集成 ······ 396
20.2 CruiseControl 和 phpUnderControl ······ 404
20.2.1 安装 CruiseControl ······ 404
20.2.2 安装 phpUnderControl ······ 406
20.2.3 安装项目 ······ 408
20.3 小结 ······ 418

第 21 章 对象、模式与实践 ······ 419
21.1 对象 ······ 419
21.1.1 选择 ······ 420
21.1.2 封装和委托 ······ 420
21.1.3 解耦 ······ 420
21.1.4 复用性 ······ 421
21.1.5 美学 ······ 421
21.2 模式 ······ 421
21.2.1 模式给我们带来了什么 ······ 422
21.2.2 模式和设计原则 ······ 422
21.3 实践 ······ 424
21.3.1 测试 ······ 424
21.3.2 文档 ······ 424
21.3.3 版本控制 ······ 425
21.3.4 自动构建 ······ 425
21.3.5 持续集成 ······ 425
21.3.6 我们还遗漏了什么 ······ 425
21.4 小结 ······ 426

第六部分 附 录

附录 A 参考文献 ······ 428
附录 B 简单的解析器 ······ 431

Part 1

第一部分

介　　绍

本部分内容

■ 第 1 章　PHP：设计与管理

第1章

PHP：设计与管理

2004年初PHP 5发布，它带来的新特性中最重要的是加强了对面向对象编程（OOP）的支持。它使PHP社区开始对"对象"和"设计"产生很大兴趣。事实上，这一趋势从PHP 4支持面向对象编程就已经开始了，而PHP 5使这个过程向前跨跃了一大步。

本章介绍对象编码技术可以满足我们的一些需求。我们将十分简要地总结在Java中模式的发展和相关的实践，然后看看在PHP中类似的过程。

我们还会在本章简单介绍全书涉及的主题。

本章包括以下内容。

- 灾难的演变：项目变糟糕。
- 设计与PHP：面向对象设计技术如何深入影响了PHP社区。
- 本书：对象、模式与实践。

1.1 问题

我们所面对的问题在于PHP太易于使用了！它吸引你马上试验你的想法，并给出令人满意的结果。你可能会直接在Web页面上书写大量的代码，因为PHP本身就有这样的功能；你也可能会在文件中添加工具函数（如数据库访问代码），然后把文件放到页面中去，不知不觉中写出来的Web应用程序也可以正常运行。

你将会陶醉于这条毁灭之路。当然，你并不会意识到这一点，因为你的站点看起来是那么奇妙。站点运行良好，客户也很开心，站点的用户也愿意为之付费。

然而当开始一个新的开发阶段时，再去看以前的代码，问题就出现了。现在你有了一个更大的开发团队、更多的用户和更充足的预算。在没有任何预警的情况下，事态开始恶化，就像你的项目中了毒一样。

新加入的程序员开始费力地理解你的代码，虽然对你而言是件几分钟就能搞定的事，即便代码有些绕。他将会花费超出你想象的时间来成为一个合格的团队成员。

要做一个简单的改变，你发现必须更新20多个页面。这样原本估计只要一天就可完成的工作，实际上却用了3天。

一个程序员保存了他对某个文件的修改，而实际上你在较早时候曾对这个文件做了大量修

改，于是你修改过的内容全部作废。这个损失在3天内都没有被发现，因为你有自己本地的副本，你一直在自己的代码上工作。于是你需要整整一天的时间来理清头绪，还要让第三个用到这份文件的程序员也按兵不动。

由于站点受欢迎，访问量增加，你需要将代码迁移到新的服务器上。这需要手动安装，而你发现文件路径、数据库名和密码已经被硬编码（hard-coded）到许多源文件中。因为你不想重写迁移到新服务器所修改的代码，所以迁移过程中必须暂停开发工作。另外，因为某人自作聪明地使用了Apache的URL重写模块ModRewrite，网站现在需要这个模块才能正常运转。以上种种原因导致原计划需要的2小时变成了8小时。

最后你开始第二个阶段，并且头一天半一切正常。在你就要下班离开办公室时出现了第一个bug报告，几分钟后客户打电话过来抱怨。他的报告和第一个bug报告很相似，但是经过更加详细的调查，却发现这是另一个引起类似行为的bug。你开始回忆在本阶段开始时的简单修改导致对整个项目其余部分所做的大量修改。

这时你认识到有些修改还不彻底。可能出错的原因是有的在最初被忽略了，或是出问题的文件在合并时被覆盖了。你急忙进行必要的修改来解决问题。你脚不沾地地忙于测试所做的修改，但只是简单的复制和粘贴，应该不会有什么问题了吧？

第二天早上你到达办公室后发现，购物车模块一整晚都不能运行了。你最后一分钟的修改漏掉了一个上引号，所以代码就不能用了。自然，在你酣睡期间，其他时区的顾客并没有睡觉，他们本来是有可能到你的网店上来消费的。你解决了这个问题，安抚了客户，并且集合队伍开始另一天的救火行动。

以上这个每天都会发生的代码人员的故事可能让你觉得有点夸张，但是我看到现实中这些事情一而再地发生。许多PHP项目都是这样从一个小项目开始慢慢变成令人恐惧的怪兽。

由于表现层中也包含应用程序逻辑，随着数据库查询、用户验证、表单处理等工作的进行代码被从一个页面复制到另一个页面，重复现象很早就开始在代码中蔓延。每当改动涉及这些代码块之一，那么代码块出现的每个地方都要改动，否则将会出现bug。

缺少文档使得代码难以阅读，缺少测试则让不明显的错误直到项目部署也未被发现。客户惯于改变需求的天性常常使代码从它原来的用途变为完成根本不适合或不在计划中的任务。这样的代码逐渐成为混杂的一团，所以很难通过替换或重写部分代码来实现新的目标。

当然，如果你是一个PHP技术顾问，这并不是坏消息。评估和修复一个这样的系统的费用足够你买6个月或更长时间昂贵的咖啡和DVD。但若严肃看待，此类问题通常意味着一个业务的成功或失败。

1.2 PHP和其他语言

PHP的流行意味着它的使用范围很早就经过严格测试。就像我们在下一章中看到的那样，PHP最初是用于管理个人主页的宏指令集合。随着PHP 3和PHP 4的先后出现，PHP语言很快成为了大型企业级站点成功的支持力量。尽管如此，PHP早期的遗留思想直到现在还影响着脚本设计和项目管理。PHP有时会被误解为一种属于计算机业余爱好者的语言，最适合用于表现层，其实

这样说是不公平的，这是一种偏见。

2000年前后，新思潮开始风行于很多其他编程语言社区。面向对象设计使Java社区感到兴奋。你会认为这是一件很自然的事情，因为Java就是一种面向对象的语言。Java提供一种“颗粒”单元（即对象），易于组织开发。但是，使用类和对象并不意味着这就是一种特殊的设计方式。

在20世纪70年代，设计模式（Design Pattern）作为一种描述和解决问题的方案第一次被提出。实际上这种想法源自建筑学，而不是计算机科学。在20世纪90年代初，面向对象编程使用了相同的技术来命名和描述软件设计问题。由“四人组”（Gang of Four）著的在1995年出版的*Design Patterns: Elements of Reusable Object-Oriented Software*是里程碑式的设计模式的图书，而且至今仍是不可替代的。该书中介绍的模式是此领域的任何人开始时所必需的第一步，因此本书的大部分模式也都从中选取。

Java语言本身在其API中使用了很多核心模式，但是设计模式到20世纪90年代末期才引起开发社区的密切关注。有关模式的书很快占据了各大书店的计算机书架，并且有人已经开始在邮件列表和论坛上对模式展开激烈讨论。

无论你认为模式是沟通专业知识的强大方式，还是一时的流行性事件（通过本书的书名，你可能已经猜到我要表达的观点），很难否定的是模式对于软件设计方法的关注和强调是很有意义的。

模式相关的话题也渐渐引起关注，其中之一是由Kent Beck倡导的XP（eXtreme Programming，极限编程）。XP是一种针对项目的方法，鼓励灵活的、面向对象设计的、高度集中的计划和执行方法。

XP的主要原则是坚决主张测试是项目成功的关键。测试应该是自动化的、经常进行的，并且最好是在目标代码编写前就设计好的。

XP同时认为项目应该被划分为小（非常小）的迭代任务。要一直仔细检查代码和需求。架构和设计应该是共享和不变的版本，而代码则可以频繁地进行修改。

如果说XP思想为设计发展插上了激进的翅膀，那么要了解软件设计发展相对缓和的趋势，可以看看Andrew Hunt 和 David Thomas著的2000年出版的*The Pragmatic Programmer*[①]，它是我读过的关于编程的最好的书之一。

XP曾被一些人认为是幼稚的想法，但是它的成长经过了20年来最高水准的面向对象实践的考验，XP原则也已经被广泛使用。特别是代码修订，即通常所说的重构，被作为模式的强大辅助手段。重构从20世纪80年代开始发展，但是在Martin Fowler的*Refactoring*：*Improving the Design of Existing Code*[②]一书中才对重构进行了分类和定义，此书已于1999年出版。

随着XP和设计模式的流行，测试也成为一个热点。强大的测试工具JUnit的发布则使自动测试更加容易，而JUnit也成为Java程序员武器库中必不可少的一件利器。Kent Beck和Erich Gamma发表的“Test Infected: Programmers Love Writing Tests”（http://junit.sourceforge.net/doc/testinfected/

① 本书中文版为《程序员修炼之道》，由中国电子工业出版社出版。——译者注

② 本书中文版为《重构：改善既有代码的设计》，由人民邮电出版社出版。——译者注

testing.htm）是一篇非常好的介绍测试的文章，并且至今仍有很大影响力。

PHP 4也在这个时期发布，它在效率上有很大提升，更重要的是强化了对对象的支持。这个改进使完全进行面向对象开发变得可能。程序员们都喜欢这个新特性，这让Zend公司的创始人Zeev Suraski和Andi Gutmans（他们和PHP创始人Rasmus Lerdorf一起参与PHP内核的开发）有点吃惊。正如我们在下一章将看到的，PHP中的对象支持并不完美，但只要遵守一定的原则，在使用语法时细心一点，你就可以真正使用PHP进行面向对象开发。

然而，本章开头描述的设计灾难仍然很常见。在PHP中，“设计”的身影似乎绝迹，但显然现在越来越多的人开始关注设计。2001年，Leon Atkinson写了一点东西来介绍PHP和模式，而Harry Fuecks则在2002年开始运行网站www.phppatterns.com（现在似乎停止更新了）。基于设计模式的框架项目（如BinaryCloud）开始面世，还出现了一些自动测试和文档工具。

在2003年，PHP 5的第一个beta版本发布，这确保了PHP成为一种面向对象编程语言。Zend Engine 2提供了非常棒的对象支持。同样重要的是，这个版本的发布说明了对象和面向对象设计现在已经成为PHP语言的核心部分。

多年以来，PHP 5一直在演变改进，例如引入了命名空间[①]和闭包等重要的新特性。在此期间，PHP 5已经成为服务器端Web编程的最佳之选。

1.3 关于本书

本书不尝试开拓面向对象设计的新领域，而是打算站在巨人的肩膀之上，介绍在PHP中如何应用良好的设计原则，实现一些关键的模式（特别是在四人组著的经典图书*Design Patterns*中介绍的模式）。然后，超越具体的代码实现，介绍那些有助于项目成功的工具和技术。除了本章和本书最后的简短总结之外，本书分为3个部分：对象、模式与实践。

1.3.1 对象

在第二部分中，我们首先看看PHP和对象的历史，从对PHP 3的回顾一直到PHP 5的核心特性。

可能你是一个经验丰富的、成功的PHP程序员，但不了解面向对象开发。因此，本书会从头解释对象、类和继承等。在第二部分中，也会介绍PHP 5带来的一些面向对象的新特性。

打下基础之后，我们开始深入讨论主题，研究PHP中较高级的面向对象特性。我们也会用一章内容介绍PHP提供的帮助处理对象和类的工具。

但这还不够，你还要知道如何声明类，如何使用类来实例化对象。你要先规划好系统中的组成单元，并决定它们之间如何进行交互。学习这些比学习对象工具和语法要困难得多。最后，我们将介绍用PHP进行面向对象设计。

1.3.2 模式

“模式”描述了软件设计中的问题，并提供解决方案的核心部分。我们所说的“解决方案”只是描述一个可以用来解决问题的方式，并不像代码参考手册那样直接提供代码让你来复制粘贴

① namespace，PHP 5.3中已经提前引入。——译者注

（尽管参考手册对于程序员来说是很有用的资料）。介绍模式时我们可能会提供一个示例实现，但示例的重要性不如模式本身所阐述的概念。

第三部分首先给出设计模式的定义，并描述其结构。我们也会介绍某些模式流行的原因。

模式通常都遵循一定的核心设计原则。理解这些核心原则有助于分析一个模式的设计目的，也可应用于其他所有编程。我将介绍一些设计原则，也将介绍UML（Unified Modeling Language，统一建模语言），它是一种与平台无关的描述类及其之间关系的方法。

本书不想罗列出各种模式，只介绍最知名和实用的设计模式。 对于每种设计模式，都描述问题，分析解决方案并给出PHP实现的代码示例。

1.3.3 实践

如果没有妥善管理，那么即使有良好的架构，项目也可能失败。在第四部分中，我们将介绍帮助搭建开发框架的工具，以便确保项目的成功。如果本书其他部分是关于设计与编程的实践，那么第四部分就是关于管理代码的实践。我们介绍的工具可以构成一个项目的支持架构，可以帮助我们在错误发生时跟踪bug，促进程序员间的交流，使代码易于安装、更加明晰。

我们之前提到过自动测试的威力。在第四部分中将首先介绍测试领域中所存在的问题及相应的解决方案。

很多程序员习惯于依靠自己来完成一切功能。PHP社区维护着PEAR——一个质量经过严格控制的代码库，可以方便地用在项目中。我们将比较自己实现功能和直接利用PEAR软件包这两种不同做法。

我将介绍PEAR的安装机制，你会发现通过命令行安装一个PEAR包是非常简单的。作为相对独立的软件包，这套机制也可以用于自动安装你自己的代码。我将向你演示如何实现它。

写文档是一件痛苦的事情，特别是当项目完成日期接近时，文档和测试最可能被放弃。但我认为这么做很可能是错误的，所以向你介绍PHPDocumentor，它是一个帮助你把代码中的注释转化为HTML文件（用于描述API中的每个元素）的工具。

本书中几乎所有工具或技术均直接用PHP来描述或实现。唯一的例外是Subversion。Subversion是一个软件版本控制系统，可以让很多程序员在同一个代码库上合作，而不会发生冲突。Subversion允许你在项目开发的任何阶段获取项目的快照，看到是谁对代码做了修改，并可以把项目分成几个可合并的分支。总有一天Subversion会挽救一个项目。

有两个事实无法回避：bug总是在代码中相同的区域重复出现，这有时让我们的工作好像时不时经历相同的场景，对于bug总有似曾相识的感觉；通常改进代码比修正bug带来更糟的后果。自动测试可以解决这两个问题，可以为系统提供早期的预警。我将介绍PHPUnit，它是一个由xUnit移植而来的强大的测试工具。xUnit原本是用Smalltalk语言设计的，但后来衍生出了多个语言版本，最知名的是Java的JUnit。我将介绍PHPUnit的特性及使用PHPUnit的好处，也将讨论使用测试要付出的代价。

PEAR提供了一个构建工具来安装软件包。对于安装封闭和独立的软件包，使用PEAR是非常理想的。但对于一个完整的PHP项目来说，需要更多的灵活性，因为整个网站要庞大和复杂得多。

我们可能需要安装文件到非标准的路径，或者搭建数据库，或者修改服务器配置等。简而言之，完整的应用程序需要在安装过程中完成很多工作。Phing是一个可以依赖的选择，它是Java工具Ant的PHP移植版本。Phing和Ant会解析构建文件，然后按你告诉的方式处理源文件。这通常意味着从一个源代码目录中复制代码到几个不同的目的路径中。但是如果你需要更加复杂的功能，Phing也可以很容易地进行扩展，从而满足要求。

测试和构建是很重要的，但你必须安装和运行测试，而且为了从中受益要持续这样做。如果构建和测试不能自动进行，就很容易懈怠，并一切都听之任之。在讨论持续集成的时候我们会介绍一些需要集成使用的工具和技术，通过它们可以实现构建和测试的自动化。

1.3.4 第 3 版新增内容

PHP是一种活跃的语言，因而它一直在接受来自开发者的改进建议，也在不断发展。这个新版本经过了全面审校，涵盖了这门语言新的变化和各种可能性。比如说，这一版中讨论了闭包等新特性。第2版中曾讨论过一个实验性的命名空间特性，但PHP 5.3用新的命名空间特性取代了它。当然，我也更新了这个版本来介绍这个新特性。

我更新了关于版本控制的章节，用介绍Subversion的内容代替了CVS。因为自本书第1版出版以来，版本控制工具也有了新的发展。这一版也增加了讲述持续集成的一章内容，掌握其中讨论的方法和工具能够让开发人员自动化并监控自己的构建和测试工作。

1.4 小结

这是一本关于面向对象设计与编程的书，它也介绍了管理PHP代码库（从协作开发到部署）的工具。

这是从两个不同但互补的角度来阐述同一个问题。我们的目标是构建实现既定目标的系统，并且它们易于进行协作开发。

本书的第二个目标（易于进行协作开发）是展示软件系统的美学。作为程序员，我们创造了拥有“形状”和“行为”的机器，我们一生中投入很多天，每天投入很多个小时来开发程序，使这些机器的“形状”诞生。我们希望我们创造的单个类或对象、软件组件直至最终产品，能够构成一个优雅的整体。版本控制、测试、文档和项目构建工具不仅让我们能更好地实现这个目标，还是我们想要创造的“形状”的一部分。我们想要干净和聪明的代码，也想要一份经过良好设计的代码库（不论是对其他程序员还是用户来说）。分享、阅读和部署项目的机制和代码本身一样重要。

Part 2 第二部分

对　　象

本部分内容

- 第 2 章　PHP 与对象
- 第 3 章　对象基础
- 第 4 章　高级特性
- 第 5 章　对象工具
- 第 6 章　对象与设计

第2章

PHP与对象

2

对象一开始并不是PHP项目的关键，它是PHP创建者们后来才想到要引进的。引进对象的想法被证明是非常正确的。本章总结了PHP面向对象特性的发展过程，逐步介绍对象的概念。

本章包括以下内容。

- PHP/FI 2.0：我们所不知道的PHP。
- PHP 3：对象的首次出现。
- PHP 4：面向对象编程的发展。
- PHP 5：对象成为PHP的核心部分。
- PHP 6：PHP的未来。

2.1 PHP 对象的偶然成功

很多面向对象风格的PHP类库和程序在广泛流传，说明了对象对于PHP的重要性。PHP 5对于面向对象开发的加强，看上去就像一个自然和不可避免的结果。实际上，事实并非如此。

2.1.1 最初：PHP/FI

正如我们今天知道的那样，PHP起源于由Rasmus Lerdorf用Perl开发的两个工具。PHP是Personal Homepage Tool的简写，意为“个人主页工具”，而FI即Form Interpreter，意为表单解释器。结合这两种工具形成的宏命令可用来发送SQL语句到数据库、处理表单和流控制。

这两个工具用C语言重写后组合成了PHP/FI 2.0。这个时期的PHP看上去和现在的语法不太一样，但本质上没有太大的不同。它支持变量、关联数组和函数。当然，那时还没有对象。

2.1.2 语法糖[①]：PHP 3

事实上，在PHP 3的计划阶段，对象并不在安排之中。和今天的PHP一样，PHP 3的主要架构设计师也是Zeev Suraski和Andi Gutmans。PHP 3是由PHP/FI 2.0完全重写成的，但是对象并不是新语法中必需的部分。

① Syntactic Sugar，指高级编程语言提供的语法上的便利，以方便程序编写。——译者注

据Zeev Suraski回忆，PHP对类的支持实际上是后来才加上的（准确地说是1997年8月27日）。那时的类和对象实际上是定义和存取关联数组的另一种方式。

当然，加入了方法和继承使得类比关联数组更加强大，但那时对于类的操作仍然存在很多局限性，比如不能访问父类中被覆盖的方法（如果你不明白这里的意思也不用担心，我们将会在后面解释）。另一个不足之处是对象在PHP脚本中被传递的方式并非最佳，我们马上就会讲到。

在这个时期对象只是个边缘话题，在官方文档中也没有提及这方面的内容。PHP手册只用了一句话和一段代码示例来介绍对象，而且该示例没有介绍对象的继承和属性等。

2.1.3 一场静悄悄的革命：PHP 4

PHP 4可算是夯实PHP基础的另一个阶段，因为此时PHP大部分核心代码都被重写了。Zend引擎（由Zeev和Andi而得名）开始为PHP提供强大动力。Zend引擎是PHP的主要组成部分之一。任何你可能调用到的PHP函数都是高级扩展层的一部分，它们一般都是根据所完成的工作来命名的，比如和数据库API对话或者转换字符串。Zend引擎管理内存，将控制交由其他组件，并将我们所熟识的每天使用的PHP语法转换为可执行的字节码（bytecode），而且类也是由Zend引擎引入PHP的核心特性。

客观地说，在PHP 4中，我们可以在子类中覆盖并访问父类的方法，这是一个很大的进步。

然而主要的不足依然存在。把对象赋值给变量，传递对象到函数，或从方法返回对象，结果都会创建对象的副本。所以这样的赋值：

```
$my_obj = new User('bob');
$other = $my_obj;
```

将导致同时存在两个User对象，而不是同一个User对象的两个引用。在大多数面向对象语言中，我们一直都是按引用传递对象，而不是像这里这样按值传递。也就是说，只传递指向对象的引用，而不是对象的副本。默认按值传送的行为导致了许多不明显的bug，因为程序员可能会无意识地在脚本的某个部分修改了对象，而希望在其他的引用上看到改变（实际上在其他引用上却没有任何改变）。在本书中，我们将看到许多对同一对象进行多次引用的例子。

幸运的是，PHP 4中存在一种强制按引用传递的方式，但是它需要使用一种看起来很笨的语法。

按引用赋值：

```
$other =& $my_obj;
// $other和$my_obj指向相同对象
```

按引用传递：

```
function setSchool( & $school ) {
    // $school现在是被传递对象的引用而不是副本
}
```

按引用返回：

```
function & getSchool( ) {
    // 返回一个引用而不是副本
```

```
        return $this->school;
    }
```

尽管这样也可以正常工作，但是程序员很容易忘记使用&符号，容易使bug蔓延到面向对象的代码中。而且这样的代码不会报错，加上导致错误的代码看上去好像还很合理，所以很难追踪到bug。

这时PHP手册中不仅包括语法，还特别增加了对象方面的内容，并且面向对象的代码开始成为主流。对象在PHP中并非没有引起争论（显然现在还是这样），像"我需要对象吗"这样的论调在邮件列表中是很常见的。Zend站点也将支持面向对象程序设计和其他持反对意见的文章放在了一起。

尽管按引用传递的观点存在争论，许多程序员还是在他们的代码中加入了&符号。面向对象的PHP逐渐流行。就像Zeev Suraski在DevX.com上写的一篇文章（http://www.devx.com/webdev/Article/10007/0/ page/1）中所说的那样：

> PHP历史上最大的一个进步就是对于面向对象编程的支持（尽管功能不多，还存在一些问题和限制），并且面向对象已成为最流行的开发方式。这个趋势，是我们没有预料到的。目前我们没有为PHP设计充足的面向对象开发功能，所以在这方面PHP显得不那么让人满意。现在PHP中的对象与其他面向对象语言中真正的对象不同，它更像是关联数组。

像第1章所说的那样，很多站点和在线文章开始对面向对象设计产生浓厚的兴趣。PHP的官方代码库PEAR本身就使用了面向对象设计。在扩展PHP功能的PEAR代码包中，我们可以找到一些面向对象设计模式的最佳示例。

也许有人会认为PHP是迫于不可避免的外部压力才支持面向对象编程的。值得注意的是，虽然面向对象编程在20世纪60年代就已经出现，但是从20世纪90年代中期才开始发展。最为流行的Java直到1995年才发布。面向过程的语言C的扩展版C++则在1979年才开始出现。经过长时间的发展，C++在20世纪90年代才有了真正的飞跃。Perl 5在1994年发布，新的变革使得这个曾经是面向过程的语言也支持了面向对象（虽然有些人认为Perl对面向对象的支持是后来才想到的）。作为一个面向过程的小型语言，PHP迅速地实现了对面向对象的支持，这显示了PHP对用户需求的切实考虑。

2.1.4　拥抱改变：PHP 5

PHP 5明确表示支持对象和面向对象程序设计。这并不是说对象现在是PHP的唯一工作方式（本书也没有这样说），但是现在对象被认为是开发企业级系统的强大助力和重要方法，并且PHP的核心设计也完全支持对象。

对象已经从事后的想法变为PHP语言的驱动力。可能最重要的改变是默认按引用传递的行为代替了问题多多的对象复制。但这只是一个开始。在本书中，尤其是这一部分，我们将会遇到更多扩展和增强PHP对象支持的更多变化，包括参数指示（argument hinting）、私有（private）和保护（protected）类型的方法及属性、static关键字、命名空间、异常等。

PHP是一个支持面向对象开发的语言，而不是一种纯面向对象的语言。但PHP对于对象的支持程度，已经足以实现本书中介绍的所有面向对象设计的目标。

2.2 走向未来：PHP 6

写到这里的时候，距离PHP 6的发布还有相当长的一段时间，不过它正处于积极开发中。它将构建在全新的Zend Engine（ZE3）基础之上，将提供对Unicode字符串处理的内在支持，从而更好地支持国际化。这意味着你在使用任何PHP字符串函数时，都不必再担心它们是否和当前的字符集匹配了。在过去，开发人员为实现很多常见的功能，必须面对处理多字节的难题，不仅效率低，而且不可靠。国际化变得越来越重要，这个核心特性也迅速成为了任何编程语言中必不可少的一部分。

从某些方面上说，未来已经提前到来了。有一个被规划在PHP 6中的特性，已经在PHP 5（从PHP 5.3起）中得到了支持：命名空间。通过命名空间创建类和函数的命名作用域（naming scope），在扩展系统或者包含类库时，你就可以在不同类库中使用名称相同的类，而不会产生冲突。命名空间还可以把你从使用复杂却必需的命名规范中拯救出来[①]：

```
class megaquiz_util_Conf {
}
```

像这样的类名可以防止包之间的冲突，但是会使代码显得不够简洁精练。

在写作本书时，提示返回类型的特性再次被列入PHP 6的开发计划。这样你就可以在方法或函数的声明中声明它所返回的对象类型了。这一承诺会被PHP引擎强制实现。提示返回类型将进一步改进PHP对模式原则（例如“针对接口编程，而不是针对实现编程”）的支持，我希望能在本书重印或再版时向大家介绍这一特性。

2.3 拥护和疑虑：关于对象的争辩

对象和面向对象设计引起了两方阵营的热烈争辩。很多优秀的程序员这几年来开发了很多优秀的程序而并没有使用对象，而PHP将仍然是一个重要的面向过程的Web编程平台。

本书很自然地会偏向于面向对象，这代表我对面向对象的偏爱。因为本书就是关注对象和面向对象设计的，不可避免地强调面向对象。但要注意，本书绝对不是要说明对象才是PHP中唯一可以开发出成功代码的途径。

我们一开始就应该记住那个著名的Perl格言：做一件事有很多种方法（There's more than one way to do it）。对于小脚本来说尤其如此，很多时候让代码快速建立并执行比搭建一个良好的易于扩展的架构更为重要。在极限编程领域，我们将对这样项目的测试称为spike。

代码是个很灵活的东西。写代码的技巧在于：搞清楚什么时候你一时的想法会成为另一个更大的开发的源头，在确定设计方案之前暂停大量的代码编写工作。你可能已经决定使用面向对象

① 这种命名方式被很多PHP框架所采用，如Zend Framework、CakePHP等。另外，PEAR也使用这种命名方式。——译者注

的设计方案，但目前市面上大量的PHP书都只提供面向过程设计的示例，而本书尝试提供一些用对象来设计的想法。我希望这是一个很好的开始。

2.4 小结

本章介绍了PHP语言中对象的背景。未来，PHP将和面向对象设计紧密相连。在接下来的几章中，我将介绍目前PHP对于对象的支持及一些与设计相关的主题。

第3章

对象基础

3

对象（object）和类（class）是本书的核心。随着PHP 5的发布，它们也成为了PHP的核心内容。在本章中，通过介绍PHP的核心面向对象特性奠定一些基础知识，以便后面深入分析对象与设计。

PHP 5在面向对象支持方面有了很大的进展，因此如果你已经熟悉了PHP 4，就可以在PHP 5中发现一些新东西。如果你是初次接触面向对象程序设计，则需要仔细阅读本章。

本章包括如下内容。

- 类和对象：声明类及实例化对象。
- 构造方法：自动加载对象。
- 基本数据类型和类的类型：为什么类型很重要。
- 继承（inheritance）：为什么需要继承以及如何使用继承。
- 可见性（visibility）：整合对象接口并保护类中的方法和属性不受干涉。

3.1 类和对象

理解面向对象程序设计的第一个障碍是类和对象之间奇妙的关系。对于许多人来说，这是他们第一次接触这种关系，是和面向对象产生的第一缕火花，所以我们不会跳过基础知识。

3.1.1 编写第一个类

我们经常用对象来描述类，有趣的是也经常用类来描述对象。这个循环使面向对象编程的第一步很难继续下去。因为类定义对象，所以我们首先来定义一个类。

简单地说，类是用于生成对象的代码模板。我们使用class关键字和一个任意的类名来声明类。类名可以是任意数字和字母的组合，但是不能以数字开头。和一个类关联的代码必须用大括号括起来。据此我们创建一个类：

```
class ShopProduct {
    // 类体
}
```

例中的ShopProduct类已经是一个合法的类。虽然它没有任何用处，但是我们已经完成了一

些非常重要的事情。我们定义了一个类型，即创造了一种可以在脚本中使用的数据。读完本章后，你就会明白它的作用。

3.1.2　第一个对象（或两个）

如果类是生成对象的模板，那么对象是根据类中定义的模板所构造的数据。对象可以被说成是类的“实例”，它是由类定义的数据类型。

我们使用ShopProduct类作为生成ShopProduct对象的模板。这样做需要使用new操作符。new操作符和类名一起使用，如下所示：

```
$product1 = new ShopProduct();
$product2 = new ShopProduct();
```

new操作符和作为它唯一操作数的类名一起被调用，并生成类的实例。在本例中，它生成了一个ShopProduct对象。

我们使用了ShopProduct类作为生成两个ShopProduct对象的模板。虽然它们的功能相同（都为空），但是$product1和$product2是由同一个类生成的相同类型的不同对象。

如果你还不明白，那么打个比方：把类当做生产塑料鸭子的机器的一个铸模，那么对象就是这台机器生产的鸭子。物品的类型由把它们压制出来的模具决定。它们看起来都是一样的，但是它们是不同的个体。也就是说，它们是同一类型的不同实例。这些鸭子可能有它们自己的序列号来说明它们的身份，在PHP脚本中创建的每个对象也有唯一的身份（在对象的生命周期中唯一），也就是说，PHP会在一个进程内重复使用这些身份（或标识符，identifier）来访问这些对象。可以通过输出$product1和$product2对象来证明这一点：

```
var_dump($product1);
var_dump($product2);
```

执行这两个函数后，得到的输出如下：

```
object(ShopProduct)#1 (0) {
}
object(ShopProduct)#2 (0) {
}
```

注解　在PHP 4和PHP 5（直到版本5.1）中，直接输出一个对象，得到的是包含对象ID的字符串。从PHP 5.2起，PHP不再支持这个功能，把对象当做字符串处理将会出错，除非在对象的类中定义了__toString()方法。在本章后面会用到这个方法，并且在第4章中也会介绍__toString()。

通过传递对象给var_dump()，我们获得了它所包含的有用信息——每个对象的内部标志符（#号后面的数字）。

要使对象更加有趣，可以修改ShopProduct类以支持特定的数据字段——属性（property）。

3.2 设置类中的属性

类可以定义被称为属性的特定变量。属性也被称为成员变量，用来存放对象之间互不相同的数据。例如，在ShopProduct对象的例子中，我们可以用类属性来指定商品名称和价格。

类的属性和标准的变量很相似，不过必须在声明和赋值前加一个代表可见性的关键字（keyword）。这个关键字可以是public、protected或private，它决定了属性的作用域。

注解 变量作用域是指变量在函数或类中有意义的区域（它和类方法的作用域概念相同，这将在本章后面讨论）。因此，在函数内定义的变量存在于局部作用域（local scope）中，在函数之外定义的变量则存在于全局作用域（global scope）中。通常来说，我们无法从外部访问定义于内部作用域的变量。因此，如果将变量定义在函数内部，就不能从函数外部访问它。但对象则要宽松些，某些对象中的变量有时可以从其他地方访问。我们将会看到，哪些变量可以被访问以及可以从哪里访问是由public、protected和private关键字决定的。

本章稍后将回到这些关键字和可见性的问题上。现在，让我们用public关键字来声明属性：

```
class ShopProduct {
    public $title               = "default product";
    public $producerMainName    = "main name";
    public $producerFirstName   = "first name";
    public $price               = 0;
}
```

如上所示，我们设置了4个属性，并且给每个属性赋了初始值。实例化ShopProduct类得到的任何对象都将加载默认数据。每个属性声明中的public关键字确保我们可以从对象之外访问属性。

注解 可见性关键字public、private和protected是在PHP 5中引入的。如果在PHP 4下运行，这些例子将无法正常工作。在PHP 4中，所有的属性都用var关键字声明，它的效果和使用public一样。因为考虑到向后兼容，PHP 5中保留了对var的支持，但会将var自动转换为public。

我们可以使用->字符连接对象变量和属性名来访问属性变量，如下所示：

```
$product1 = new ShopProduct();
print $product1->title;
```

```
default product
```

因为属性被定义为public，所以可以给属性赋值（也可以读取属性），替换其在类中设置的默认值。

```
$product1 = new ShopProduct();
$product2 = new ShopProduct();
$product1->title="My Antonia";
$product2->title="Catch 22";
```

通过在ShopProduct类中声明和设置$title属性，可以确保所有的ShopProduct对象在初次创建时都有此属性。这就意味着使用这个类的代码在此假定上可以使用这个ShopProduct对象。因为可以重设这一属性，所以不同对象的$title值都可不同。

注解　*调用类、函数或方法的代码通常被称为该类、函数或方法的客户端（client）或者客户端代码（client code），在后面的章节中将会频繁看到这个术语。*

事实上，PHP并没有强制所有的属性都要在类中声明。我们可以动态地增加属性到对象，如下所示：

```
$product1->arbitraryAddition = "treehouse";
```

但这种给对象赋属性的方法在面向对象编程中并不是一个良好的做法，我们基本上不使用。

为什么说动态地设置属性是一个坏习惯？因为你在创建类的时候就已经定义了数据类型，说明这个类（以及由类实例化的对象）是由特定的字段和函数集合组成。如果ShopProduct类定义了一个$title属性，那么之后任何和ShopProduct对象一起工作的代码都假设$title属性是可用的，但动态新增的属性则没有这种保证。

此时我们的对象依然很笨拙。当需要使用对象的属性时，必须在对象之外才能完成。我们要直接获取或设置属性信息，在多个对象中设置多个属性将变得烦琐：

```
$product1 = new ShopProduct();
$product1->title = "My Antonia";
$product1->producerMainName  = "Cather";
$product1->producerFirstName = "Willa";
$product1->price = 5.99;
```

我们再次使用ShopProduct类，覆写（override）所有的默认属性值，直到设置了所有的产品信息。设置完数据后，就可以访问它：

```
print "author: {$product1->producerFirstName} "
          ."{$product1->producerMainName}\n";
```

执行后输出

```
author: Willa Cather
```

用这种方式设置属性值会有很多问题。因为PHP允许动态设置属性，所以如果拼错或忘记属性名时并不会得到警告。例如，错误地把这一行

```
$product1->producerMainName  = "Cather";
```

写作

```
$product1->producerSecondName  = "Cather";
```

因为PHP引擎认为这行代码完全合法，所以不会发出警告。当我们开始输出作者名字时，将会得到意外的结果。

另外一个问题是类太过松散。我们没有被强制设置标题、价格或产品名，客户端代码可以确定的是这些属性存在，但是面对的可能是默认值也可能不是。理想情况下，我们希望实例化ShopProduct对象时设置有意义的属性值。

最后，我们不得不重复做一些经常要做的事情。输出完整的作者名就是一个烦人的过程：

```
print "author: {$product1->producerFirstName} "
          ."{$product1->producerMainName}\n";
```

如果可以让对象代替我们来处理这件苦差事该多好。

如果能够让ShopProduct对象具有在内部处理属性数据的能力，则所有问题将迎刃而解。

3.3 使用方法

属性可以让对象存储数据，类方法则可以让对象执行任务。方法是在类中声明的特殊函数。方法声明类似于函数声明。function关键字在方法名之前，方法名之后圆括号中的是可选的参数列表。方法体用大括号括起来：

```
public function myMethod( $argument, $another ) {
    // ...
}
```

和函数不同的是，方法必须在类中声明。它们也可以接受限定词，包括可见性关键字。和属性一样，方法可以被声明为public、protected或private。将方法声明为public，就可以在当前对象之外调用该方法。如果在方法声明中省略了可见性关键字，那么方法将被隐式声明为public。本章稍后将详细介绍方法的修饰词。

注解 PHP 4不能识别方法或属性的可见性关键字。增加public、protected或private到方法声明中将会使程序出错。在PHP 4中，所有的方法都是public类型。

在大多数情况下，我们可以使用->连接对象变量和方法名来调用方法。调用方法时必须使用一对圆括号（即使没有传递任何参数给该方法，这类似于函数调用）。

```
class ShopProduct {
    public $title               = "default product";
    public $producerMainName    = "main name";
    public $producerFirstName   = "first name";
    public $price               = 0;


    function getProducer() {
        return "{$this->producerFirstName}".
           " {$this->producerMainName}";
```

```
    }
}

$product1 = new ShopProduct();
$product1->title = "My Antonia";
$product1->producerMainName  = "Cather";
$product1->producerFirstName = "Willa";
$product1->price = 5.99;


print "author: {$product1->getProducer()}\n";
```

输出如下所示：

```
author: Willa Cather
```

我们向ShopProduct类中增加了getProducer()方法。注意声明方法时没有指定可见性关键字，即getProducer()默认是一个public方法，可以在类之外被调用。

我们在该方法中使用了一个特性。$this伪变量（pseudo-variable）把类指向一个对象实例。如果觉得这个概念难以理解，尝试用“当前实例”替换$this，即把语句

```
$this->producerFirstName
```

替换为

```
当前实例的$producerFirstName属性
```

因此getProducer()合并并返回$producerFirstName和$producerMainName的值，这样我们可以很方便地得到完整的商品产家名称。

这样做对类有一定改进，但我们仍然需要做大量重复性的工作。我们依靠客户端程序员来改变ShopProduct对象的属性。这样做有两方面的问题：首先，初始化ShopProduct对象需要5行代码，过于麻烦；其次，我们无法保证在初始化ShopProduct对象时每个属性都被设置。因此，我们需要一个当类实例化对象时可被自动调用的类方法。

创建构造方法

创建对象时，构造方法［constructor method，也称为构造器（constructor）］会被自动调用。构造方法可以用来确保必要的属性被设置，并完成任何需要准备的工作。在PHP 5之前的版本中，构造方法使用和所在类相同的名字，因此ShopProduct类可以使用ShopProduct()方法作为它的构造方法。在PHP 5中，虽然这种方式依然有效，但应该将构造方法命名为__construct()，注意方法名以两个下划线字符开头。以后我们将看到PHP类中其他许多特殊方法也以两个下划线开头。下面为ShopProduct类定义一个构造方法：

```
class ShopProduct {
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price = 0;
```

```
    function __construct( $title,
                          $firstName, $mainName, $price ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
    }

    function getProducer() {
        return "{$this->producerFirstName}".
               " {$this->producerMainName}";
    }

}
```

在上例中我们把之前的初始化功能集成到类中，以减少代码中的重复。当使用new操作符创建对象时，__construct()方法会被调用。

```
$product1 = new ShopProduct( "My Antonia", "Willa", "Cather", 5.99 );
print "author: {$product1->getProducer()}\n";
```

这将输出：

```
author: Willa Cather
```

任何给定的参数都将被传递给构造方法。因此在我们的例子中，传递了商品名称、生产者姓、名和商品价格到构造方法。构造方法使用伪变量$this给对象的每个属性赋值。

注解 PHP 4不会把__construct()方法当做构造方法。如果你使用的是PHP 4，可以通过声明一个和类的名称相同的方法来创建一个构造方法，即如果类名为ShopProduct，则可以使用名称为ShopProduct()的方法声明构造方法。

PHP仍然支持这种命名方案，但是除非是为了向下兼容，否则命名构造方法时最好还是使用__construct()。

ShopProduct对象现在更易于实例化，使用起来也更安全。实例化和设置只需一条语句就可以完成。任何使用ShopProduct对象的代码都可以相信所有的属性皆被初始化了。

这种可预测性是面向对象编程的重要部分。你所设计的类应该能让对象的使用者确定它们的特性。同理，使用对象时也应该确定它的类型。在下一节中，我们将介绍一种机制，即在声明方法时可以强制规定参数的对象类型。

3.4 参数和类型

数据类型决定了PHP代码中处理数据的方式。例如，使用字符串类型来显示字符数据并用字符串函数来处理这些数据，在数学表达式中使用整型，在测试表达式中使用布尔型，等等。这些

类型称为“基本类型”（primitive type）。从更高层次来说，每个类都定义了一种数据类型。因此，ShopProduct对象属于基本类型，但同时也属于ShopProduct类这一类型。在本节中，我们将看到类方法的两种类型。

在PHP中，定义方法和函数时不需要指定参数的数据类型。这既是一种麻烦，也是一种便利。参数可以是任何类型给我们提供了很大的灵活性。你可以创建灵活响应不同数据类型的方法，根据不同的环境调整其功能。但当类方法希望一个参数必须是某种数据类型（而不是其他类型）时，这种灵活性可能会引起代码的歧义。

3.4.1　基本类型

PHP是一种弱类型语言，即变量不需要声明为特定的数据类型。变量$number可以在相同的作用域中存放值2和字符串"two"。在强类型语言（如C或Java）中，必须在变量赋值前声明变量的类型，并且变量的值必须是规定的类型。

这并不是说PHP没有类型的概念。每个赋给变量的值都有一种类型。可以使用PHP的类型检查函数来确定变量的值的类型。表3-1列出了PHP中认可的基本类型以及它们所对应的测试函数。每个函数接受一个变量或值，如果参数是相应的类型，则返回true。

表3-1　基本类型和PHP的类型检查函数

类型检查函数	类　型	描　述
is_bool()	布尔型	值为true或false
is_integer()	整型	整数
is_double()	双精度型	浮点数（有小数点的数字）
is_string()	字符串	字符数据
is_object()	对象	对象
is_array()	数组	数组
is_resource()	资源	用于识别和处理外部资源（如数据库或文件）的句柄
is_null()	NULL	未分配的值

处理方法和函数的参数时，检查变量的类型尤其重要。

基本数据类型：一个例子

你需要时刻留意代码中的数据类型。下面看一个可能遇到的问题，它是你可能会遇到的数据类型相关的许多问题之一。

假设你要从XML文件中提取配置参数。XML元素<resolvedomains>告诉应用程序是否应该将IP地址解析为域名。这是一个有用的过程，但是会花费比较多的时间。下面是XML的一部分：

```
<settings>
    <resolvedomains>false</resolvedomains>
</settings>
```

应用程序提取字符串"false"作为特征值传递给outputAddresses()方法，这个方法用于显

示IP地址数据。下面是outputAddresses():

```
class AddressManager {
    private $addresses = array( "209.131.36.159", "74.125.19.106" );


    function outputAddresses( $resolve ) {
        foreach ( $this->addresses as $address ) {
            print $address;
            if ( $resolve ) {
                print " (".gethostbyaddr( $address ).")";
            }
            print "\n";
        }
    }
}
```

如你所见，outputAddresses()方法循环遍历IP地址数组，输出每个IP。如果$resolve参数变量为true，该方法同时输出IP地址和域名。

我们可能会这样调用outputAddresses()方法:

```
$settings = simplexml_load_file("settings.xml");
$manager = new AddressManager();
$manager->outputAddresses( (string)$settings->resolvedomains );
```

这个代码段使用了SimpleXML API（在PHP 5中新增）来获取resolvedomains元素的值。在本例中，我们知道这个值是元素文本"false"，并且将其转换为SimpleXML文档所需要的字符串。

代码不会按照我们期望的那样执行。在传递字符串"false"到outputAddresses()方法时，我们误解了方法对参数处理的隐含假设。该方法接受一个布尔值（true或false）。事实上，字符串"false"在测试中解析为true。这是因为PHP在测试变量时会转换一个非空字符串值为布尔值true。所以

```
if ( "false" ) {
    // ...
}
```

等效于

```
if ( true ) {
    // ...
}
```

有许多种方法来改正这个错误。

你可以使outputAddresses()方法更宽松，让它可以识别字符串并用一些基本规则将字符串转换为相等的布尔型。

```
// 类AddressManager...
    function outputAddresses( $resolve ) {
        if ( is_string( $resolve ) ) {
            $resolve =
                ( preg_match("/false|no|off/i", $resolve ) )?
```

```
                false:true;
        }
        // ...
    }
```

你也可以不改动outputAddresses()方法，但添加一个注释，它明确说明$resolve参数应该是一个布尔值。这种方法让程序员更容易理解代码。

```
/**
 * 输出地址列表
 * 如果$resolve为true，则每个地址都会被解析为域名
 * @param $resolve Boolean是否解析地址?
 */
function outputAddresses( $resolve ) {
    // ...
}
```

最后，你也可以让outputAddresses()严格检查$resolve参数中的数据类型。

```
function outputAddresses( $resolve ) {
    if ( ! is_bool( $resolve ) ) {
        die( "outputAddress() requires a Boolean argument\n" );
    }
    //...
}
```

这种方法强制性要求客户端代码提供正确的数据类型给$resolve参数。把字符串参数转化成布尔类型对于客户端代码来说更加友好，但是这可能会引起其他问题。在提供转换方法的情况下，我们需要猜测客户端代码的环境和意图。另一方面，通过强制规定参数为布尔型数据类型，客户端代码需要决定是否将字符串映射为布尔值以及哪个单词映射为哪个值。这样，outputAddresses()方法可以专注于执行它的任务。在面向对象开发中，“专注特定任务，忽略外界上下文”是一个重要的设计原则，我们将在本书中经常提到这一点。

事实上，采用哪种处理参数类型的策略，取决于任何潜在bug的严重程度。通常PHP会根据语境自动转换大多数基本数据类型。例如，在数学表达式中，字符串中的数字被转换成相等的整型或浮点型。因此你的代码多半会自然地容忍类型错误。但如果要求方法参数必须是一个数组，就需要更加小心。传递一个非数组值到PHP数组函数不会产生有用的结果，还可能会在方法中引起一连串的错误。

因此，你需要在检测类型、转换类型和依赖良好清晰的文档（无论决定用哪一种，都应该提供文档）之间仔细权衡。

无论你如何解决这类问题，都要认真思考一件事情：类型处理。PHP是一种弱类型的语言，这使得这件事更加重要。我们不能依靠编译器来防止类型相关的bug，必须考虑到当非法数据类型的参数传递给方法时，会产生怎样的后果。我们不能完全信任客户端程序员，应该始终考虑如何在方法中处理他们引入的无用信息。

3.4.2　获得提示：对象类型

正如参数变量可以包含任何基本类型的数据，参数默认情况下也可以包含任何类型的对象。

这种灵活性有它的好处，但是在方法定义中可能会出现一些问题。

假设有一个设计为接受ShopProduct对象的方法：

```
class ShopProductWriter {
    public function write( $shopProduct ) {
        $str  = "{$shopProduct->title}: " .
                $shopProduct->getProducer() .
                " ({$shopProduct->price})\n";
        print $str;
    }
}
```

可以这样测试这个类：

```
$product1 = new ShopProduct( "My Antonia", "Willa", "Cather", 5.99 );
$writer = new ShopProductWriter();
$writer->write( $product1 );
```

输出

```
My Antonia: Willa Cather (5.99)
```

ShopProductWriter类只包括一个方法：write()。write()方法接受一个ShopProduct对象，并用它的属性和方法构造和打印一个摘要字符串。我们把参数变量命名为$shopProduct，说明该方法希望接受一个ShopProduct对象，但是我们并没有强制要求这一点。也就是说，我们可能接收到非预期的对象或基本类型，但在实际处理$ShopProduct之前不会知道具体是什么。而那时代码则可能已根据预期（接收到了真正的ShopProduct对象）执行了相应操作。

注解 你也许想知道为什么我们不直接增加write()方法到ShopProduct类，而是另外设计了一个ShopProductWriter类。简单来说，理由是"划分责任区"——ShopProduct类负责管理产品数据，ShopProductWriter类则负责写入数据。随着进一步阅读，你会看到这样分工有一定的好处。

为了解决这个问题（没有强制要求参数类型），PHP 5引入了类的类型提示（type hint）。要增加一个方法参数的类型提示，只需简单地将类名放在需要约束的方法参数之前。我们可以这样修改write()方法：

```
public function write( ShopProduct $shopProduct ) {
    // ...
}
```

现在write()方法只接受包含ShopProduct对象的$shopProduct参数。让我们尝试用一个错误的对象来调用write()：

```
class Wrong { }
$writer = new ShopProductWriter();
$writer->write( new Wrong() );
```

因为write()方法包含类型提示，给它传递Wrong对象将会产生严重错误。

```
PHP Catchable fatal error:  Argument 1 passed to ShopProductWriter::write() must be
 an instance of ShopProduct, instance of Wrong given ...
```

有了参数的类型提示，我们不再需要在使用参数前对其进行类型检查。对于客户端程序员来说，方法的定义更加清晰易懂，一眼就能看明白该方法对于参数的要求，不用再担心一些由对象类型错误而引起的bug，因为提示是严格执行的。

虽然自动类型检查是防止bug的极佳途径，但我们要知道类型提示是在运行时才生效的。也就是说，类提示只有在错误的对象被传递给方法时才会报错。如果对write()的调用隐藏在条件子句中（例如只在圣诞节早上才运行），那么如果没有仔细检查代码的话，很难发现错误。

类型提示不能用于强制规定参数为某种基本数据类型，比如字符串和整型。如果要处理基本数据类型，在方法体中要使用is_int()这样的类型检查函数。但你可以强制规定使用数组作为参数。这被称为数组提示：

```
function setArray( array $storearray ) {
    $this->array = $storearray;
}
```

在PHP 5.1中加入了对数组提示的支持，而后来的版本还新增了对null默认值的参数提示，即可以指定参数为一个特定类型或null值。如下所示：

```
function setWriter( ObjectWriter $objwriter=null ) {
    $this->writer = $objwriter;
}
```

到目前为止，我们把数据类型和类看成是同义词，但实际上两者有一个关键的不同之处：定义一个类也就定义了一个类型，但是一个类型可以用于描述一个家族的众多类。这种将不同的类放在同一个类型之下的机制被称为继承，这将在下一节讨论。

3.5 继承

继承是从一个基类得到一个或多个派生类的机制。

继承自另一个类的类被称为该类的子类。这种关系通常用父亲和孩子来比喻。子类将继承父类的特性，这些特性由属性和方法组成。子类可以增加父类（也称为超类，superclass）之外的新功能，因此子类也被称为父类的“扩展”。

在深入学习继承的语法之前，我们先了解一下它可以帮助我们解决什么问题。

3.5.1 继承问题

再次查看ShopProduct类。此时，它已经有了一定的功能，可以处理各种各样的产品。

```
$product1 = new ShopProduct( "My Antonia", "Willa", "Cather", 5.99 );
$product2 = new ShopProduct( "Exile on Coldharbour Lane",
                             "The", "Alabama 3", 10.99 );
print "author: ".$product1->getProducer()."\n";
```

```
print "artist: ".$product2->getProducer()."\n";
```

输出如下：

```
author: Willa Cather
artist: The Alabama 3
```


不论对于书还是CD，将生产者名称分为两个部分都是可行的。我们想按照“Alabama 3”和“Cather”而不是“The”和“Willa”排序。偷懒是一种很好的设计策略，因此此时无需担心多种产品使用ShopProduct的情况。

但如果在本例中增加新的需求，那么事情立刻变得复杂起来。例如，需要显示图书和CD的特定数据。对于CD，必须存放总的播放时间；对于图书，则必须存放总页数。可能还会有其他的差别，但这些足以说明问题。

怎么扩展我们的例子来适应这些变化呢？有两种选择：第一，将所有的数据都抛给ShopProduct类；第二，将ShopProduct拆分为两个单独的类。

先看第一种方法。在这里，我们把与CD和书籍相关的数据放到一个单独的类中：

```
class ShopProduct {
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    function __construct(   $title, $firstName,
                            $mainName, $price,
                            $numPages=0, $playLength=0 ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }

    function getNumberOfPages() {
        return $this->numPages;
    }

    function getPlayLength() {
        return $this->playLength;
    }

    function getProducer() {
        return "{$this->producerFirstName}".
               " {$this->producerMainName}";
    }
}
```

我们提供了访问$numPages和$playLength属性的方法，以说明如何满足CD和书这两种分支对象的使用要求，但从这个类实例化得到的对象将包含多余的方法，并且对于CD来说，创建一个实例时必须使用不必要的构造方法参数：CD将存放和书总页数相关的信息和功能，而图书也包含表示播放长度的数据。现在这些属性和方法可能可以共存，但是增加更多的产品类型时会发生什么呢？每个类型有自己的方法，那么要为每个类型增加更多的方法吗？如果这么做，我们的类将会日渐复杂并且难以管理。

因此强行将不同类的字段合并到一个类中，会导致对象臃肿，产生冗余的属性和方法。

问题还不仅在于数据，类的功能也是一个问题。假设有一个用于描述产品的类方法。销售部门要求发票上提供一行简明的摘要信息：对于CD，需要包含CD的播放时间；对于图书，需要包含图书的页数。那么，我们将不得不为每种类型提供不同的实现，同时也需要使用特征值来说明当前产品的类型，以决定发票上摘要信息的格式。下面是一个例子：

```
function getSummaryLine() {
    $base  = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    if ( $this->type == 'book' ) {
        $base .= ": page count - {$this->numPages}";
    } else if ( $this->type == 'cd' ) {
        $base .= ": playing time - {$this->playLength}";
    }
    return $base;
}
```

为了设置$type属性，我们可以检测构造方法接收的$numPages参数。显然ShopProduct类变得过于复杂。如果我们的发票格式有了更多的变化，或者增加了新的产品类型，这些功能的差别将变得难以管理。也许我们应该尝试这个问题的第二种解决途径。

上面的解决方案里ShopProduct就像将两个类放在一个类中一样。现在我们创建两种类型（即两个类），而不是一种，如下所示：

```
class CdProduct {
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    function __construct(   $title, $firstName,
                            $mainName, $price,
                            $playLength ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->playLength        = $playLength;

    }
```

```
    function getPlayLength() {
        return $this->playLength;
    }

    function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": playing time - {$this->playLength}";
        return $base;
    }

    function getProducer() {
        return "{$this->producerFirstName}".
               " {$this->producerMainName}";
    }
}

class BookProduct {
    public $numPages;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    function __construct(   $title, $firstName,
                            $mainName, $price,
                            $numPages ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
    }

    function getNumberOfPages() {
        return $this->numPages;
    }

    function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": page count - {$this->numPages}";
        return $base;
    }

    function getProducer() {
        return "{$this->producerFirstName}".
               " {$this->producerMainName}";
    }
}
```

这时我们解决了“复杂性”问题，但是我们付出了代价。现在不需要测试特征值，就可以为每种产品创建getSummaryLine()方法。类也不再包含和它不相关的字段或方法。

这样做的代价是代码重复。每个类中的getProducer()方法完全相同。每个构造方法用同样的方法设置了许多相同的属性。这是令人不愉快的地方，我们需要想办法来避免它。

如果需要让getProducer()方法在每个类中的行为相同，在一个类中所作的任何改变也需要在其他类中出现。稍不小心，我们的类便将无法即时同步。

即使我们确信可以不停地复制、粘贴，仍然存在问题。我们现在有两种类型而不是一种。

还记得ShopProductWriter类吗？它的write()方法只为一种类型而设计：ShopProduct。我们如何使之仍然可以正常工作呢？可以从write()方法定义中删除类的类型提示，但是稍后难以保证传递类型正确的对象给write()。我们可以增加自己的类型检查代码到方法体中：

```
class ShopProductWriter {
    public function write( $shopProduct ) {
        if ( ! ( $shopProduct instanceof CdProduct )  &&
             ! ( $shopProduct instanceof BookProduct ) ) {
            die( "wrong type supplied" );
        }
         $str  = "{$shopProduct->title}: " .
                $shopProduct->getProducer() .
                " ({$shopProduct->price})\n";
        print $str;
    }
}
```

注意示例中的instanceof操作符。如果左边操作数中的对象是右边操作数所表示的类型，则instanceof结果为true。

我们被迫再次增加了代码的复杂性。不仅要检测$shopProduct是否符合write()方法中的两种类型，还要确保每种类型和另一种类型一样，都支持相同的字段和方法。如果我们可以限制参数必须为特定的一种产品类型，就可以在方法参数中使用类型提示，那么代码会更加清楚简洁[①]，而且也因为此时可以确定ShopProduct类只支持某一种接口。

ShopProduct类中的CD和图书这两部分合并之后不能很好地工作，但是它们似乎也不能分开存在。我们希望能为每种类型提供一个单独的工具，把书和CD当做单独的类型，但同时希望提供公用的功能来避免重复，又允许每种类型处理不同的方法调用。这正是我们使用继承的原因。

3.5.2　使用继承

创建继承树的第一步是找到现有基类元素中不适合放在一起，或者不需要进行特殊处理的类方法。

我们知道getPlayLength()和getNumberOfPages()方法不适合放在一起。我们还需为getSummaryLine()方法创建不同的实现。下面我们以此为基础来设计两个派生类（子类）：

```
class ShopProduct {
    public $numPages;
```

① 我们不需要在方法内判断对象类型，我们确定传递进来的参数对象一定支持特定的类方法（例如参数为ShopProduct对象，则一定支持getProducer()方法。——译者注

```
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    function __construct(   $title, $firstName,
                            $mainName, $price,
                            $numPages=0, $playLength=0 ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
        $this->numPages          = $numPages;
        $this->playLength        = $playLength;
    }

    function getProducer() {
        return "{$this->producerFirstName}".
               " {$this->producerMainName}";
    }

    function getSummaryLine() {
        $base  = "$this->title ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}

class CdProduct extends ShopProduct {
    function getPlayLength() {
        return $this->playLength;
    }

    function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": playing time - {$this->playLength}";
        return $base;
    }
}

class BookProduct extends ShopProduct {
    function getNumberOfPages() {
        return $this->numPages;
    }

    function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": page count - {$this->numPages}";
        return $base;
    }
}
```

要创建一个子类，必须在类声明中使用extends关键字。在本例中，我们创建了两个新类——BookProduct和CdProduct，它们都继承自ShopProduct类。

由于派生类没有定义构造方法，所以在实例化时会自动调用父类的构造方法。子类默认继承了父类所有的public和protected方法（但是没有继承private方法或属性）。也就是说，我们可以在从CdProduct类实例化的对象中调用getProducer()方法，虽然getProducer()是在ShopProduct类中定义的。

```
$product2 =   new CdProduct(    "Exile on Coldharbour Lane",
                                "The", "Alabama 3",
                                10.99, null, 60.33 );
print "artist: {$product2->getProducer()}\n";
```

这样两个子类都继承了父类的公共部分。我们可以把BookProduct对象当做一种ShopProduct对象，传递BookProduct或CdProduct对象给ShopProductWriter类的write()方法来开展需要的工作。

注意CdProduct和BookProduct类都覆写了getSummaryLine()方法，提供了自己独特的实现。派生类可以扩展和修改父类的功能。

该方法超类的实现似乎有点多余，因为它的两个子类都重写了该方法。不过新的子类可能会用到它所提供的基本功能。该方法的存在为客户端代码提供了保证：所有的ShopProduct对象都将提供getSummaryLine()方法。稍后你将看到如何通过基类提供同样的保证，而不必提供任何实现。每个ShopProduct子类都继承其父类的属性。BookProduct和CdProduct都使用各自的getSummaryLine()方法访问$title属性。

可能一开始，继承是一个不太容易理解的概念。通过定义一个从其他类继承而来的类，我们确保一个类拥有其自有的功能和父类的功能。另一种可以帮助你理解继承的思路是“搜索”。当我们调用$product2->getProducer()时，在CdProduct类中并没有找到这样的方法，那么就查找ShopProduct中的默认实现。另一方面，当调用$product2->getSummaryLine()时，在CdProduct中找到getSummaryLine()并调用。

对属性的访问也是一样的。当我们在BookProduct类的getSummaryLine()方法中访问$title时，并没有在BookProduct类中找到这个属性，而是从父类ShopProduct中获得。两个子类中的$title属性是等价的，因为它属于超类。

看看ShopProduct的构造方法，就会发现我们仍然在基类中管理本应在子类中处理的数据。BookProduct类应该处理$numPages参数和属性，CdProduct类应该处理$playLength参数和属性。要完成这个工作，我们需要在每个子类中分别定义构造方法。

1. 构造方法和继承

在子类中定义构造方法时，需要传递参数给父类的构造方法，否则你得到的可能是一个构造不完整的对象。

要调用父类的方法，首先要找到一个引用类本身的途径：句柄（handle）。PHP为此提供了parent关键字。

要引用一个类而不是对象的方法，可以使用::而不是->。所以

```
parent::__construct()
```

意味着“调用父类的__construct()方法”。修改我们的例子，让每个类只处理自己的数据。


```
class ShopProduct {
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    function __construct(   $title, $firstName,
                            $mainName, $price ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
    }

    function getProducer() {
        return "{$this->producerFirstName}".
               " {$this->producerMainName}";
    }

      function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}

class CdProduct extends ShopProduct {
    public $playLength;

    function __construct(   $title, $firstName,
                            $mainName, $price, $playLength ) {
        parent::__construct(    $title, $firstName,
                                $mainName, $price );
        $this->playLength = $playLength;
    }

    function getPlayLength() {
        return $this->playLength;
    }

    function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        $base .= ": playing time - {$this->playLength}";
        return $base;
    }
}
```

```
class BookProduct extends ShopProduct {
    public $numPages;

    function __construct(   $title, $firstName,
                            $mainName, $price, $numPages ) {
        parent::__construct(    $title, $firstName,
                                $mainName, $price );
        $this->numPages = $numPages;
    }

    function getNumberOfPages() {
        return $this->numPages;
    }

    function getSummaryLine() {
        $base  = "$this->title ( $this->producerMainName, ";
        $base .= "$this->producerFirstName )";
        $base .= ": page count - $this->numPages";
        return $base;
    }
}
```

每个子类都会在设置自己的属性前调用父类的构造方法。基类现在仅知道自己的数据。子类一般是父类的特例。我们应该避免告诉父类任何关于子类的信息，这是一条经验规则。

注解 在PHP 5以前的版本中，构造方法使用类的名称，之后使用`__construct()`来命名。如果使用旧的语法，调用父类的构造方法将会绑定到特定的类：`parent::ShopProduct()`。如果改变了类的层次结构，这样做将会产生问题。许多bug的生成就是因为程序员改变了类的直接父类，却忘记更新构造方法。如果使用统一的调用方式（用`parent::__construct()`来调用），无论使用的父类是否改变，子类中都不需要修改代码。当然，仍然需要传递正确的参数给父类的构造方法。

---

2. 调用被覆写的方法

`parent`关键字可以在任何覆写父类方法的方法中使用。覆写一个父类的方法时，我们并不希望删除父类的功能，而是扩展它，通过在当前对象中调用父类的方法可达到这个目的。如果回头看两个类中`getSummaryLine()`方法的实现，将会发现它们重复了许多代码。事实上，我们应该利用`ShopProduct`类中已经存在的功能，而不应该重复开发。

```
// ShopProduct类
    function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }

// BookProduct类
    function getSummaryLine() {
```

```
        $base = parent::getSummaryLine();
        $base .= ": page count - {$this->numPages}";
        return $base;
    }
```

我们在基类ShopProduct中为getSummaryLine()方法完成核心功能。接着在子类中简单地调用父类方法，然后增加更多数据到摘要字符串，而不再重复实现相同的功能。

在掌握了“继承”的基础知识后，接下来将学习属性和方法的“可见性”。

3.5.3 public、private、protected：管理类的访问

到目前为止，我们已将所有的属性定义为public，不论是隐式的定义或显式的定义（在前面加public关键字）。如果在属性声明中使用了旧的var关键字，方法和属性的默认设置为public。

类中的元素可以被声明为public、private或protected。

- 在任何地方都可以访问public属性和方法。
- 只能在当前类中才能访问private方法或属性，即使在子类中也不能访问。
- 可以在当前类或子类中访问protected方法或属性，其他外部代码无权访问。

这对我们有什么用呢？可见性关键字允许我们只公开类中客户需要的部分，这给对象设置了一个清晰的接口。

通过阻止客户访问特定的属性，访问控制也有助于预防代码中的bug。例如，我们想要ShopProduct对象支持打折功能，可以增加$discount属性和setDiscount()方法。

```
// ShopProduct类
    public $discount = 0;
// ...
    function setDiscount( $num ) {
        $this->discount=$num;
    }
```

除了通过setDiscount()设置折扣量，还可以创建getPrice()方法来获得已经实施的折扣。

```
// ShopProduct类
    function getPrice() {
        return ($this->price - $this->discount);
    }
```

这里有一个问题。我们只想公开调整过的价格，但是客户也可以很容易地绕过getPrice()方法来访问$price属性：

```
print "The price is {$product1->price}\n";
```

这里将输出原始的价格，而不是希望展示的折扣价。我们可以通过将$price属性设置为私有（private）来禁止直接输出。这样做可以阻止直接访问，强制客户使用getPrice()方法。任何试图从ShopProduct类外访问$price属性的操作都将失败。对于类外面的世界来说，这个属性就像不存在一样。

设置属性为private有时是一种过于严格的方案，因为private属性不能被子类访问。假设我们的商业规则是只有书不能打折，这时可以覆写getPrice()方法，让它返回$price属性，并且不进行折扣计算。

```
// BookProduct类
    function getPrice() {
        return $this->price;
    }
```

由于在ShopProduct类中而不是在BookProduct类中声明属性$price为private，所以这里无法访问这个属性。解决办法是声明$price为protected，让子类也可以访问。要记住的是，protected属性或方法不可以在声明它的类层次之外被访问。protected属性和方法只能在被定义的类中或者该类的子类中被访问。

一般来说，我们倾向于严格控制可访问性。最好将类属性初始化为private或protected，然后在需要的时候再放松限制条件。类中的许多（如果不是大多数）方法都可以是public，但是如果拿不定主意的话，就限制一下。有些类方法只为类中其他方法提供本地功能，与类外部的代码没有任何联系，应该将其设置为private或protected。

1. 访问方法

当客户程序员需要使用类中保存的值时，通常比较好的做法是不要允许直接访问属性，而是提供方法来取得需要的值，这样的方法被称为访问方法，也可称为getter和setter。

之前已经看到由访问方法带来的一个好处。可以使用访问方法根据环境过滤属性，就像getPrice()方法那样。

也可以使用setter方法来强制属性类型。我们已经见过，类的类型提示可以用于约束方法参数，但是它不能直接控制属性类型。还记得ShopProductWriter类使用ShopProduct对象来输出清单数据吗？我们将改进它，使其一次可以输出许多ShopProduct对象：

```
class ShopProductWriter {
    public $products = array();

    public function addProduct( ShopProduct $shopProduct ) {
        $this->products[] = $shopProduct;
    }

    public function write() {
        $str =  "";
        foreach ( $this->products as $shopProduct ) {
            $str .= "{$shopProduct->title}: ";
            $str .= $shopProduct->getProducer();
            $str .= " ({$shopProduct->getPrice()})\n";
        }
        print $str;
    }
}
```

ShopProductWriter类现在更加实用，它可以保存许多ShopProduct对象，并一次性地为所有对象输出数据。我们必须期望客户程序员尊重我们的类的本意。尽管类中提供了addProduct()

方法，但还是不能阻止程序员直接操作$products属性。人们不仅能够增加错误的对象到$products数组属性中，甚至还能覆写整个数组并用基本数据类型的值来替换它。通过设置$products属性为private，可以预防这些错误操作：

```
class ShopProductWriter {
    private $products = array();
//...
```

3

现在外部代码就无法破坏$products属性了。所有的访问都必须通过addProduct()方法，我们在方法声明中使用的类的类型提示确保只有ShopProduct类型的对象可以被增加到数组属性中。

2. ShopProduct类

在本章最后，我们改进一下ShopProduct类，并且对其子类采取一定的访问控制：

```
class ShopProduct {
    private $title;
    private $producerMainName;
    private $producerFirstName;
    protected $price;
    private $discount = 0;

    public function __construct(   $title, $firstName,
                            $mainName, $price ) {
        $this->title             = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName  = $mainName;
        $this->price             = $price;
    }

    public function getProducerFirstName() {
        return $this->producerFirstName;
    }

    public function getProducerMainName() {
        return $this->producerMainName;
    }

    public function setDiscount( $num ) {
        $this->discount=$num;
    }

    public function getDiscount() {
        return $this->discount;
    }

    public function getTitle() {
        return $this->title;
    }

    public function getPrice() {
        return ($this->price - $this->discount);
```

```
    }

    public function getProducer() {
        return "{$this->producerFirstName}".
               " {$this->producerMainName}";
    }

    public function getSummaryLine() {
        $base  = "{$this->title} ( {$this->producerMainName}, ";
        $base .= "{$this->producerFirstName} )";
        return $base;
    }
}

class CdProduct extends ShopProduct {
    private $playLength = 0;

    public function __construct(   $title, $firstName,
                            $mainName, $price, $playLength ) {
        parent::__construct(    $title, $firstName,
                                $mainName, $price );
        $this->playLength = $playLength;
    }

    public function getPlayLength() {
        return $this->playLength;
    }

    public function getSummaryLine() {
        $base = parent::getSummaryLine();
        $base .= ": playing time - {$this->playLength}";
        return $base;
    }

}

class BookProduct extends ShopProduct {
    private $numPages = 0;

    public function __construct(   $title, $firstName,
                            $mainName, $price, $numPages ) {
        parent::__construct(    $title, $firstName,
                                $mainName, $price );
        $this->numPages = $numPages;
    }

    public function getNumberOfPages() {
        return $this->numPages;
    }

    public function getSummaryLine() {
        $base = parent::getSummaryLine();
        $base .= ": page count - {$this->numPages}";
        return $base;
```

```
    }

    public function getPrice() {
        return $this->price;
    }
}
```


这个版本的ShopProduct系列类实质上并没有增加新功能。所有的方法都被明确设置为public，所有的属性则为private或protected。我们增加了许多访问方法使类变得更加完善。

3.6 小结

本章介绍了大量基础知识，先实现了一个内容为空的类，最后实现了有完整继承特性的类。我们学习了一些设计相关的内容，特别是类型和继承，了解了PHP对可见性的新支持并研究了它的部分用途。在下一章中，我们将学习更多PHP面向对象特性。

第4章

高级特性

我们已经知道，借助类型提示和访问控制可以更好地控制类的接口。本章将继续深入探讨PHP对面向对象开发的支持。

本章包括以下内容。

- 静态方法和属性：通过类而不是对象来访问数据和功能。
- 抽象类和接口：设计和实现相分离。
- 错误处理：异常。
- Final类和方法：限制继承。
- 拦截器方法：自动委托。
- 析构方法：对象销毁前的清理工作。
- 克隆对象：创建对象的副本。
- 把对象解析成字符串：创建摘要型方法。
- 回调：用匿名函数为组件添加功能。

4.1 静态方法和属性

第3章的所有例子使用的都是对象。我们把类当做生成对象的模板，把对象作为活动组件，对象的方法可以被调用，对象的属性可以被访问。之前的例子也暗示了，面向对象编程中的实际操作都是通过类的实例（而不是类本身）完成的。毕竟，类仅仅是对象的模板。

事实并非如此简单。我们不仅可以通过对象访问方法和属性，还可以通过类来访问它们。这样的方法和属性是“静态的”（static），必须用static关键字来声明。

```
class StaticExample {
    static public $aNum = 0;
    static public function sayHello() {
        print "hello";
    }
}
```

注解 static关键字是在PHP 5中引入的，在PHP 4程序中不能使用。

静态方法是以类作为作用域的函数。静态方法不能访问这个类中的普通属性，因为那些属性属于一个对象，但可以访问静态属性。如果修改了一个静态属性，那么这个类的所有实例都能访问到这个新值。

因为是通过类而不是实例来访问静态元素，所以访问静态元素时不需要引用对象的变量，而是使用::（两个冒号）来连接类名和属性或类名和方法。

```
print StaticExample::$aNum;
StaticExample::sayHello();
```


在第3章中，我们已经熟悉了这个语法。我们曾使用::和parent来访问覆写的方法。现在和之前一样，只不过访问的是类而不是对象数据。一个子类可以使用parent关键字来访问父类，而不使用其类名。要从当前类（不是子类）中访问静态方法或属性，可以使用self关键字。self指向当前类，就像伪变量$this指向当前对象一样。因此，在StaticExample类的外部可以使用其类名访问属性$aNum：

```
StaticExample::$aNum;
```

而在StaticExample类内部，可以使用self关键字：

```
class StaticExample {
    static public $aNum = 0;
    static public function sayHello() {
        self::$aNum++;
        print "hello (".self::$aNum.")\n";
    }
}
```

注解 只有在使用parent关键字调用方法的时候，才能对一个非静态方法进行静态形式的调用[①]。除非是访问一个被覆写的方法，否则永远只能使用::访问被明确声明为static的方法或属性。在文档中，你将会经常看到使用static语法来引用方法或属性。这并不意味着其中的方法或属性必须是静态的，只不过说明它属于特定的类。例如，ShopProductWriter类的方法write()可以表示为ShopProductWriter::write()，虽然write()方法并不是静态的。[②]你将在本书中看到这种语法形式。

根据定义，我们不能在对象中调用静态方法，因此静态方法和属性又被称为类变量和属性，因此不能在静态方法中使用伪变量$this。

那么，我们为什么要使用静态方法或属性呢？因为静态元素有很多有用的特性。首先，它们在代码中的任何地方都可用（假设你可以访问该类）。也就是说，你不需要在对象间传递类的实例，也不需要将实例存放在全局变量中，就可以访问类中的方法。其次，类的每个实例都可以访问类中定义的静态属性，所以你可以利用静态属性来设置值，该值可以被类的所有对象使用。最

---

① 其他任何情况都需要先生成一个对象，然后使用->符号调用对象中的方法。——译者注

② 注意，这只是文档中的一种惯例写法，在代码中不能这样写，否则会出错。——译者注

后，不需要实例对象就能访问静态属性或方法，这样我们就不用为了获取一个简单的功能而实例化对象。

下面构建ShopProduct类的一个静态方法来自动实例化ShopProduct对象。使用SQLite定义表products，如下所示：

```
CREATE TABLE products (
                            id INTEGER PRIMARY KEY AUTOINCREMENT,
                            type TEXT,
                            firstname TEXT,
                            mainname TEXT,
                            title TEXT,
                            price float,
                            numpages int,
                            playlength int,
                            discount int )
```

下面构建getInstance()方法，其参数为数据库记录的ID和PDO对象。用它们获取数据库的一行记录，然后返回ShopProduct对象。我们可以把这些方法添加到上一章创建的ShopProduct类中。PDO代表PHP Data Object。PDO类为不同的数据库应用程序提供了统一的接口。

```
// ShopProduct类
    private $id = 0;
    // ...
    public function setID( $id ) {
        $this->id = $id;
    }
    // ...
    public static function getInstance( $id, PDO $pdo ) {
$stmt = $pdo->prepare("select * from products where id=?");

        $result = $stmt->execute( array( $id ) );

        $row = $stmt->fetch( );

        if ( empty( $row ) ) { return null; }

        if ( $row['type'] == "book" ) {
            $product = new BookProduct(
                                    $row['title'],
                                    $row['firstname'],
                                    $row['mainname'],
                                    $row['price'],
                                    $row['numpages'] );
        } else if ( $row['type'] == "cd" ) {
            $product = new CdProduct(
                                    $row['title'],
                                    $row['firstname'],
                                    $row['mainname'],
                                    $row['price'],
                                    $row['playlength'] );
        } else {
            $product = new ShopProduct(
```

```
                                $row['title'],
                                $row['firstname'],
                                $row['mainname'],
                                $row['price'] );
        }
        $product->setId(        $row['id'] );
        $product->setDiscount(  $row['discount'] );
        return $product;
    }
//...
```

4

如上所示，getInstance()方法根据类型标志聪明地判断需要实例化的精确范围，返回特定类型的ShopProduct对象。为了保持代码简洁，我们没有进行任何错误处理。在真实项目中，我们并不能完全相信PDO对象初始化后就一定能够与正确的数据库对话。事实上，我们可以把PDO封装在类中来确保这一行为。你可以在第13章中了解更多与面向对象程序设计和数据库相关的信息。

这个方法在类中会比在对象中更有用。我们可以轻松地将原始数据转换为一个对象，而不需要一开始就使用ShopProduct对象。这个方法并没有使用任何实例的属性或方法，所以没有理由不把它定义为static。只要有一个有效的PDO对象，我们就可以在程序的任何地方调用这个方法：

```
$dsn = "sqlite://home/bob/projects/products.db";
$pdo = new PDO( $dsn, null, null );
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$obj = ShopProduct::getInstance( 1, $pdo );
```

这样的方法就像"工厂"一样，可以接受原始数据（比如一列数据或配置信息），然后据此产生对象。"工厂"这一术语在代码设计中用于生成对象实例。我们会在后面介绍工厂模式的例子。

4.2 常量属性

有些属性不能改变，例如42是生命、宇宙及万事万物的终极答案。[①]错误和状态标志经常需要被硬编码[②]进类中。虽然它们是公共的、可静态访问的，但客户端代码不能改变它们。

PHP 5可以在类中定义常量属性。和全局常量一样，类常量一旦设置后就不能改变。常量属性用const关键字来声明。常量不像常规属性那样以美元符号开头。按照惯例，只能用大写字母来命名常量，如下所示：

```
class ShopProduct {
    const AVAILABLE      = 0;
    const OUT_OF_STOCK   = 1;
    // ...
```

常量属性只包含基本数据类型的值。不能将一个对象指派给常量。像静态属性一样，只能通过类而不能通过类的实例访问常量属性。引用常量时不需要用美元符号作为前导符，如下所示：

① 出自道格拉斯·亚当斯的著名科幻小说《银河系漫游指南》。——译者注

② 硬编码的值始终不变。——译者注

```
print ShopProduct::AVAILABLE;
```

给已经声明过的常量赋值会引起解析错误。

当需要在类的所有实例中都能访问某个属性，并且属性值无需改变时，应该使用常量。

4.3 抽象类

引入抽象类（abstract class）是PHP 5的一个主要变化。这个新特性正是PHP朝面向对象设计发展的另一个标志。

抽象类不能被直接实例化。抽象类中只定义（或部分实现）子类需要的方法。子类可以继承它并且通过实现其中的抽象方法，使抽象类具体化。

你可以用abstract关键字定义一个抽象类。我们将上一章创建的ShopProductWriter类重新定义为抽象类。

```
abstract class ShopProductWriter {
    protected $products = array();

    public function addProduct( ShopProduct $shopProduct ) {
        $this->products[]=$shopProduct;
    }
}
```

和普通的类一样，可以创建抽象类的方法和属性，但是实例化一个抽象类会产生下面的错误：

```
$writer = new ShopProductWriter();
// 输出:
// 严重错误:不能实例化抽象类
// shopproductwriter ...
```

大多数情况下，抽象类至少包含一个抽象方法。抽象方法用abstract关键字声明，其中不能有具体内容。你可以像声明普通类方法那样声明抽象方法，但要以分号而不是方法体结束。下面我们为ShopProductWriter类添加抽象方法write()：

```
abstract class ShopProductWriter {
    protected $products = array();

    public function addProduct( ShopProduct $shopProduct ) {
        $this->products[]=$shopProduct;
    }

    abstract public function write();
}
```

创建抽象方法后，要确保所有子类中都实现了该方法，但实现的细节可以先不确定。

如果像下面这样创建了一个继承自ShopProductWriter的类，但是不实现write()方法：

```
class ErroredWriter extends ShopProductWriter{}
```

那么程序将出现下面的错误：

```
PHP Fatal error:  Class ErroredWriter contains 1 abstract method and
must therefore be declared abstract or implement the remaining methods
 (ShopProductWriter::write) in...
```

所以，抽象类的每个子类都必须实现抽象类中的所有抽象方法，或者把它们自身也声明为抽象方法。扩展类不仅仅负责简单实现抽象类中的方法，还必须重新声明方法。新的实现方法的访问控制不能比抽象方法的访问控制更严格。新的实现方法的参数个数应该和抽象方法的参数个数一样，重新生成对应的类型提示。

下面我们来定义ShopProductWriter()的两个实现：

```
class XmlProductWriter extends ShopProductWriter{
    public function write() {
         $str = '<?xml version="1.0" encoding="UTF-8"?>'."\n";
         $str .= "<products>\n";
         foreach ( $this->products as $shopProduct ) {
            $str .= "\t<product title=\"{$shopProduct->getTitle()}\">\n";
            $str .= "\t\t<summary>\n";
            $str .= "\t\t{$shopProduct->getSummaryLine()}\n";
            $str .= "\t\t</summary>\n";
            $str .= "\t</product>\n";
        }
        $str .= "</products>\n";
        print $str;
    }
}

class TextProductWriter extends ShopProductWriter{
    public function write() {
        $str = "PRODUCTS:\n";
        foreach ( $this->products as $shopProduct ) {
            $str .= $shopProduct->getSummaryLine()."\n";
        }
        print $str;
    }
}
```

我们创建了两个类，每个类都有自己的write()方法实现。第一个输出XML，第二个输出文本。现在使用ShopProductWriter对象的方法不知道接收到的对象是这两个类中哪个类的实例，但是可以确定该对象一定实现了write()方法。值得注意的是，在将$products按照数组处理前，我们并不检查它的类型，因为这个属性在ShopProductWriter中已初始化为空数组。

在PHP 4中，我们可以用包含警告或die()语句的类方法来模拟抽象类。PHP 4中的这种做法强制一个派生类实现抽象方法，或者冒险调用它们。

```
class AbstractClass {
    function abstractFunction() {
        die( "AbstractClass::abstractFunction() is abstract\n" );
    }
}
```

这里存在一个问题，就是这个基类的抽象性只有在抽象方法被调用时（即代码运行时）才能

检测。在PHP 5中，抽象类在被解析时就被检测，所以更加安全。

4.4 接口

抽象类提供了具体实现的标准，而接口（interface）则是纯粹的模板。接口只能定义功能，而不包含实现的内容。接口可用关键字interface来声明。接口可以包含属性和方法声明，但是方法体为空。

定义一个接口：

```
interface Chargeable {
    public function getPrice();
}
```

如上所示，接口和类非常相似。任何实现接口的类都要实现接口中所定义的所有方法，否则该类必须声明为abstract。

一个类可以在声明中使用implements关键字来实现某个接口。这么做之后，实现接口的具体过程和扩展一个仅包含抽象方法的抽象类是一样的。下面我们让ShopProduct类实现Chargeable接口。

```
class ShopProduct implements Chargeable {
    // ...
    public function getPrice() {
        return ( $this->price - $this->discount );
    }
    // ...
```

ShopProduct类已经有一个getPrice()方法，那么实现Chargeable接口还有用吗？答案是肯定的，因为类型。实现接口的类接受了它继承的类及实现的接口的类型。

也就是说，CdProduct类同时属于

```
CdProduct
ShopProduct
Chargeable
```

这可以被客户端代码很方便地使用。你只要知道一个对象的类型，就知道它能做什么。所以方法

```
public function cdInfo( CdProduct $prod ) {
    // ...
}
```

现在知道$prod对象除了拥有ShopProduct类和Chargeable接口中定义的所有方法之外，还有getPlayLength()方法。

传递相同的对象作为参数，方法

```
public function addProduct( ShopProduct $prod ) {
     // ..
}
```

知道$prod支持ShopProduct中的所有方法，但如果没有进一步测试，就无法得知getPlay-Length()方法的存在。

再次传递相同的CdProduct对象作为参数，方法

```
public function addChargeableItem( Chargeable $item ) {
    //...
}
```

不知道关于ShopProduct或CdProduct类型的一切。这个方法只关注$item参数是否包含getPrice()方法。

由于任何类都可以实现接口（实际上，一个类可以实现多个接口），接口可以有效地将不相关的类型联结起来。我们可以定义一个全新的类来实现Chargeable接口：

```
class Shipping implements Chargeable {
    public function getPrice() {
        //...
    }
}
```

将Shipping对象传递给addChargeableItem()方法，就像将其传递给ShopProduct对象一样。

使用Chargeable对象的代码（即客户端代码）可以随时调用getPrice()方法，而对象中的任何其他方法都要和其类型关联起来，无论是通过对象自己的类、父类还是其他接口。这些都是客户端代码不需要关注的。

一个类可以同时继承一个父类和实现任意个接口。extends子句应该在implements子句之前：

```
class Consultancy extends TimedService implements Bookable, Chargeable {
    // ...
}
```

注意，Consultancy类实现了不止一个接口，此时可将它们放在implements关键字之后，并将其以逗号分隔。

PHP只支持继承自一个父类，因此extends关键字只能在一个类名之前。

4.5 延迟静态绑定：static 关键字

我们已经学习了抽象类和接口，接下来简单回顾一下静态方法。你看到了，静态方法可以用作工厂方法，工厂方法是生成包含类的实例的一种方法。如果你是一位像我一样懒惰的程序员，当看到类似下面这个例子中的重复代码时，你可能会很恼火：

```
abstract class DomainObject {
}


class User extends DomainObject {
    public static function create() {
```

```
        return new User();
    }
}

class Document extends DomainObject {
    public static function create() {
        return new Document();
    }
}
```

我创建了一个超类DomainObject。当然，在实际的项目中，该类包含的功能可被其扩展类所共用。接下来，我创建了两个子类User和Document，并在这两个类中包含了静态方法create()。

注解　*当构造函数创建对象时，我为什么使用静态工厂方法？第12章将介绍Identity Map模式。只有具有相同特征的对象没有被管理，Identity Map组件才会生成并管理这个新对象。如果目标对象已经存在，就返回该对象。像create()这样的工厂方法能为这类组件创建优秀的客户端。*

这段代码能够很好地工作，但大量的重复代码很烦人。我不想为每个DomainObject子类都创建与这段代码类似的标准代码。如果我把create()放在超类中会怎么样呢？

```
abstract class DomainObject {
    public static function create() {
        return new self();
    }
}

class User extends DomainObject {
}

class Document extends DomainObject {
}
Document::create();
```

哦，看起来很整洁。现在我把常见的代码放在一个位置，并使用self作为对该类的引用。但我对关键字self做了一个假设。实际上，self对该类所起的作用与$this对对象所起的作用并不完全相同。self指的不是调用上下文，它指的是解析上下文，因此，如果运行前面的例子，会得到：

```
PHP Fatal error:  Cannot instantiate abstract class DomainObject in ...
```

因此，self被解析为定义create()的DomainObject，而不是解析为调用self的Document类。PHP 5.3之前，在这方面都有严格的限制，产生过很多笨拙的解决方案。PHP 5.3中引入了延迟静态绑定的概念。该特性最明显的标志就是新关键字static。static类似于self，但它指的

是被调用的类而不是包含类。在本例中，它的意思是调用Document::create()将生成一个新的Document对象，而不是试图实例化一个DomainObject对象。

因此，现在在静态上下文中使用继承关系。

```
abstract class DomainObject {
    public static function create() {
        return new static();
    }
}

class User extends DomainObject {
}

class Document extends DomainObject {
}

print_r(Document::create());
```

```
Document Object
(
)
```

static关键字不仅仅可以用于实例化。和self和parent一样，static还可以作为静态方法调用的标识符，甚至是从非静态上下文中调用。假设我想为DomainObject引入组的概念。默认情况下，所有类都属于default类别，但我想能为继承层次结构的某些分支重写类别。

```
abstract class DomainObject {
    private $group;
    public function __construct() {
        $this->group = static::getGroup();
    }

    public static function create() {
        return new static();
    }

    static function getGroup() {
        return "default";
    }
}

class User extends DomainObject {
}

class Document extends DomainObject {
    static function getGroup() {
        return "document";
    }
}
```

```
class SpreadSheet extends Document {
}

print_r(User::create());
print_r(SpreadSheet::create());
```

在DomainObject类中定义了构造函数。该构造函数使用static关键字调用静态方法getGroup()。DomainObject提供了默认实现，但Document将其覆盖了。我还创建了一个新类SpreadSheet，该类扩展了Document类。下面是输出：

```
User Object
(
    [group:DomainObject:private] => default
)
SpreadSheet Object
(
    [group:DomainObject:private] => document
)
```

User类不需要实现太多功能。DomainObject构造函数调用了getGroup()类，并在本地进行查找。对于SpreadSheet，虽然搜索从被调用的类SpreadSheet本身开始，但它没有提供任何实现，因此调用类Document中的getGroup()方法。PHP 5.3以前未引入延迟静态绑定，self关键字只查找DomainObject类中的getGroup()，因此遇到self关键字的时候我无计可施。

4.6　错误处理

文件放错地方，数据库服务器未初始化，URL有变动，XML文件毁坏，权限设置得不对，超过了磁盘空间限制等，这些问题时常发生。要对付这些可能出现的错误，一个简单的类方法有时会充满了错误处理代码。

下面是一个简单的Conf类的例子，它用于存放、检索和设置XML配置文件中的数据：

```
class Conf {
    private $file;
    private $xml;
    private $lastmatch;

    function __construct( $file ) {
        $this->file = $file;
        $this->xml = simplexml_load_file($file);
    }

    function write() {
        file_put_contents( $this->file, $this->xml->asXML() );
    }

    function get( $str ) {
        $matches = $this->xml->xpath("/conf/item[@name=\"$str\"]");
        if ( count( $matches ) ) {
            $this->lastmatch = $matches[0];
```

```
            return (string)$matches[0];
        }
        return null;
    }

    function set( $key, $value ) {
        if ( ! is_null( $this->get( $key ) ) ) {
            $this->lastmatch[0]=$value;
            return;
        }
        $conf = $this->xml->conf;
        $this->xml->addChild('item', $value)->addAttribute( 'name', $key );
    }
}
```

Conf类使用SimpleXml扩展来访问键/值对。下面是它可以处理的格式：

```
<?xml version="1.0"?>
<conf>
    <item name="user">bob</item>
    <item name="pass">newpass</item>
    <item name="host">localhost</item>
</conf>
```

Conf类的构造器接受一个文件路径，然后传递给simplexml_load_file()。它将得到的SimpleXmlElement对象放在$xml属性中。get()方法使用XPath和给定的name属性来定位item元素，并返回值。set()改变已存在项的值或创建一个新项。最后，write()方法将新的配置数据保存到文件中。

和许多示例代码一样，Conf类非常简单。特别是，它没有任何处理配置信息不存在或不可写的策略。它看起来很乐观，假定XML文档的格式正确并且包含了需要的元素。

测试这些错误条件比较琐碎，但是我们仍然需要决定当它们发生时该如何应对。通常有两种选择。

首先，可以停止执行代码。这个办法很简单，但是过于激烈。这个粗陋的类会导致整个程序停止执行。虽然__construct()和write()这样的方法可以检测到错误，但是它们并不知道该如何处理错误。

其次，不在类中直接处理错误，而只返回某种错误标志。错误标志可以是布尔值（true或false）或整数值（比如0或-1）。一些类也可以设置错误字符串或标志，让客户端代码在失败之后可以获得更多信息。

许多PEAR包结合使用这两种方法，在发生错误时返回一个错误对象（PEAR_Error的实例），该对象标志着有错误发生，同时也包含错误的信息。虽然这种方法已经过时了，但是大量的PEAR类还没有升级，因为很多客户端代码还依赖于这些旧的错误处理方式。

这里有个问题，我们的返回值是不定的。PHP语言本身没有强制规定统一的返回值。在写作本书时，PHP还不支持返回对象的类型提示，所以有时本应返回一个对象或者基本类型的数据，但实际上返回的可能是一个错误标志。这时候，我们不得不依赖客户程序员来测试每次调用易于

出错方法时返回的数据类型。这样做是很危险的，没有谁可以被信赖！[①]

当返回一个错误值给调用代码时，不能确保客户端代码能比我们的方法更好地决定如何处理错误。在这种情况下，问题又回到原点。客户端代码不得不决定如何响应错误条件，甚至要重新设计一个处理错误的方法[②]。

异常

PHP 5引入了异常（exception），这是一种完全不同的处理错误的方式。如果你有使用Java或C++的经验，应该会非常熟悉异常的概念。异常能够解决本节之前遇到的所有问题。

异常是从PHP 5内置的Exception类（或其子类）实例化得到的特殊对象。Exception类型的对象用于存放和报告错误信息。

Exception类的构造方法接受两个可选的参数：消息字符串和错误代码。这个类提供了一些有用的方法来分析错误条件，如表4-1所示。

表4-1　Exception类的public方法

方　法	描　述
getMessage()	获得传递给构造方法的消息字符串
getCode()	获得传递给构造方法的错误代码
getFile()	获得产生异常的文件
getLine()	获得生成异常的行号
getPrevious	获得一个嵌套的异常对象
getTrace()	获得一个多维数组，这个数组追踪导致异常的方法调用，包含方法、类、文件和参数数据
getTraceAsString()	获得getTrace()返回数据的字符串版本
__toString()	在字符串中使用Exception对象时自动调用。返回一个描述异常细节的字符串

Exception类提供的错误通知和调试信息（特别是getTrace()和getTraceAsString()方法返回的信息）很有用。实际上，Exception类和之前讨论过的PEAR_Error类几乎相同，但Exception类存放了异常的更多相关信息。

1. 抛出异常

可以联合使用throw关键字和Exception对象来抛出异常。这会停止执行当前方法，并负责将错误返回给调用代码。下面修改一下__construct()方法，使其使用throw语句：

```
function __construct( $file ) {
    $this->file = $file;
    if ( ! file_exists( $file ) ) {
        throw new Exception( "file '$file' does not exist" );
```

① 通常我们希望封装好的类是完整和独立的，不需要从外部干预内部代码的执行，所以依赖程序员另外写代码来测试一个类中的方法是否出错，这是非常不合理的。——译者注

② 简单地说，我们需要把错误处理的责任集中放在类的内部，而不能依赖于调用该类的程序员和外部代码，因为通常使用该类的程序员并不知道怎么处理类内部的方法所引发的错误。本节主要强调了PHP 4中各种错误处理方法的局限性，读者看过后面的内容后再回过头来对比一下，会更易于理解。——译者注

```
    }
    $this->xml = simplexml_load_file($file);
}
```

write()方法也使用类似的结构：

```
function write() {
    if ( ! is_writeable( $this->file ) ) {
        throw new Exception("file '{$this->file}' is not writeable");
    }
    file_put_contents( $this->file, $this->xml->asXML() );
}
```


__construct()和write()方法现在可以在工作时不停地检查文件错误，而让其他更合适的代码来响应检测到的错误。

那么抛出异常时，客户端代码怎么知道如何处理异常呢？如果调用可能会抛出异常的方法，那么可以把调用语句放在try子句中。try子句由关键字try及其后的括号组成。try子句必须跟着至少一个catch子句才能处理错误，如下所示：

```
try {
    $conf = new Conf( dirname(__FILE__)."/conf01.xml" );
    print "user: ".$conf->get('user')."\n";
    print "host: ".$conf->get('host')."\n";
    $conf->set("pass", "newpass");
    $conf->write();
} catch ( Exception $e ) {
    die( $e->__toString() );
}
```

从表面上看，catch子句类似于方法声明。当抛出异常时，调用作用域中的catch子句会被调用，自动将Exception对象作为参数变量传入。

因为异常抛出时会停止执行类方法，所以此时控制权从try子句移交给了catch子句。

2. 异常的子类化

如果要创建用户自定义的异常类，可以从Exception类继承。你有两个理由这样做：首先，可以扩展异常类的功能；其次，子类定义了新的异常类型，可以进行自己特有的错误处理。

实际上，定义多个catch子句时，也只需一个try语句。调用哪一个catch子句取决于抛出异常的类型和参数中类的类型提示。下面我们定义Exception的一些简单子类：

```
class XmlException extends Exception {
    private $error;

    function __construct( LibXmlError $error ) {
        $shortfile = basename( $error->file );
        $msg = "[{$shortfile}, line {$error->line}, col {$error->column}] ➥
{$error->message}";
        $this->error = $error;
        parent::__construct( $msg, $error->code );
    }

    function getLibXmlError() {
```

```
        return $this->error;
    }
}

class FileException extends Exception { }
class ConfException extends Exception { }
```

当SimpleXml扫描到一个损坏的XML文件时，会生成LibXmlError对象。LibXmlError类和Exception类一样，有message和code属性。我们利用这个相似之处，在XmlException类中使用LibXmlError对象。FileException类和ConfException类只是简单地继承了Exception类，没有自己的内容。现在在代码中使用这些异常类，并修改__construct()和write():

```
// Conf类
    function __construct( $file ) {
        $this->file = $file;
        if ( ! file_exists( $file ) ) {
            throw new FileException( "file '$file' does not exist" );
        }
        $this->xml = simplexml_load_file($file, null, LIBXML_NOERROR );
        if ( ! is_object( $this->xml ) ) {
            throw new XmlException( libxml_get_last_error() );
        }
        print gettype( $this->xml );
        $matches = $this->xml->xpath("/conf");
        if ( ! count( $matches ) ) {
            throw new ConfException( "could not find root element: conf" );
        }
    }

    function write() {

        if ( ! is_writeable( $this->file ) ) {
            throw new FileException("file '{$this->file}' is not writeable");
        }
        file_put_contents( $this->file, $this->xml->asXML() );
    }
```

__construct()根据遇到的错误类型，抛出XmlException、FileException或ConfException。注意，传递了一个可选的参数LIBXML_NOERROR给simplexml_load_file()。这个参数用于抑制出错警告的直接输出，并在警告发生之后留给XmlException类来处理。遇到一个不规范的XML文件时，simplexml_load_file()不会返回对象，这样我就知道发生了错误，然后用libxml_get_last_error()访问错误。

如果$file属性指向一个不可写的文件，那么write()方法就会抛出FileException。

所以我们设定__construct()可以抛出3个异常中的任意一个。怎样利用这一点呢？下面是实例化Conf()对象的代码：

```
class Runner {
    static function init() {
        try {
            $conf = new Conf( dirname(__FILE__)."/conf01.xml" );
```

```
            print "user: ".$conf->get('user')."\n";
            print "host: ".$conf->get('host')."\n";
            $conf->set("pass", "newpass");
            $conf->write();
        } catch ( FileException $e ) {
            // 文件权限问题或者文件不存在
        } catch ( XmlException $e ) {
            // XML文件损坏
        } catch ( ConfException $e ) {
            // 错误的XML文件格式
        } catch ( Exception $e ) {
            // 后备捕捉器，正常情况下不应该被调用
        }
    }
}
```

我们为每种类型的异常提供了一个catch子句。catch子句的调用取决于抛出的异常类型。第一个匹配类型的子句将会被执行，所以记得要把通用的类型放到最后，特定的类型放到前面。例如，如果把Exception的catch子句放到XmlException和ConfException的子句之前，那么后面两个子句就不会被调用。这是因为这两个类都属于Exception类型，所以将匹配第一个子句，即Exception的catch子句。

当配置文件出错（例如，文件不存在或不可写）时，第一个catch子句（FileException）被调用；当XML文件解析出错（例如，一个元素的标签没有闭合）时，第二个子句（XmlException）被调用；如果有效的XML文件不包含需要的根元素conf，第三个子句（ConfException）将被调用。由于方法只生成以上3种异常，且都被明确处理，所以不会用到最后一个子句（Exception）。另外，有一个“后备”（backstop）子句通常是个好办法，以防开发时在代码中要增加新的异常。

利用这些细化的catch子句，可以针对不同的错误使用不同的恢复或失败机制。例如，你可以决定停止执行程序、记录错误并继续执行程序，或显式地再次抛出错误：

```
try {
    //...
} catch ( FileException $e ) {
    throw $e;
}
```

这里有一个技巧，就是抛出包装了当前异常的新异常。你可以在捕捉到的旧异常中增加一个声明，或增加具体环境信息，同时又保持旧异常中原来的数据。第15章会更多地讲述这个技术。

当异常没有被客户端代码捕获时，会发生什么呢？异常将会被再次抛出，客户的调用代码会优先捕获它。这样的过程会一再发生，直到异常被捕获或者不再抛出。如果异常到最后也没有被捕获，将会引发致命错误。下面演示了例子中没有捕获其中一个异常时将会发生的情况：

```
PHP Fatal error:  Uncaught exception 'FileException' with message
'file 'nonexistent/not_there.xml' does not exist' in ...
```

所以当抛出异常时，要强制要求客户端代码对它进行处理。这并不是推卸责任——当方法检测到错误却没有足够的信息来智能地处理异常时，应该抛出异常。例子中的write()方法知道何时写文件失败，为什么失败，但是不知道该如何处理。[①]类方法本来就应该这样，不应该承担过多的责任。如果让Conf类知道过多内容，它将会失去重点并且变得难以重用。

4.7 Final 类和方法

继承为类层次（class hierarchy）内部带来了巨大的灵活性。通过覆写类或方法，根据调用的是哪个类实例，调用同样的类方法可以得到完全不同的结果。但有时候，你可能需要类或方法保持不变。如果希望类或方法完成确定不变的功能，担心覆写它会破坏这个功能，那么需要使用final关键字。

final关键字可以终止类的继承。final类不能有子类，final方法不能被覆写。

下面定义一个final类：

```
final class Checkout {
    // ...
}
```

下面尝试生成Checkout类的子类：

```
class IllegalCheckout extends Checkout {
    // ...
}
```

这将会产生这样的错误：

```
PHP Fatal error: Class IllegalCheckout may not inherit from
final class (Checkout) in ...
```

如果只在Checkout中声明某个类方法为final，而不是将整个类声明为final，那么继承Checkout就不会出现致命错误。final关键字应该放在其他修饰词（例如protected或static）之前，如下所示：

```
class Checkout {
    final function totalize() {
        // 计算账单
    }
}
```

现在可以继承Checkout，但若覆写totalize()方法会引起致命错误：

```
class IllegalCheckout extends Checkout {
    final function totalize() {
        // 修改账单计算方法
    }
}
```

① 这些问题不是类方法本身应当考虑的，而应该是调用类方法的代码应该负责的工作。——译者注

```
// Fatal error: Cannot override final method
// checkout::totalize() in ...
```

高质量的面向对象代码往往强调定义明确的接口。在统一的接口背后，实现方式往往是多种多样的。不同的类或类的组合遵从公共统一的接口，但在不同的环境下可以表现出不同的行为。声明类或方法为final，将会减少这种灵活性。有时候我们确实需要这样做，稍后你会看到相关的例子。但是在声明final时，你应当仔细考虑一下。真的不需要覆写了吗？当然，过后你总可以改变想法，但如果你已经把代码发布到类库中供他人使用，那么修改起来就不方便了[①]。请慎重使用final。

4.8 使用拦截器

PHP提供了内置的拦截器（interceptor）方法，它可以“拦截”发送到未定义方法和属性的消息。它也被称为重载（overloading），但是自从这个术语在Java和C++中被赋予不同的含义之后，我认为它还是叫做拦截器比较好。

PHP 5支持3个内置的拦截器方法。它们类似于__construct()方法，当遇到合适的条件时就会被调用。表4-2描述了这些方法。

表4-2 拦截器方法

方法	描述
__get($property)	访问未定义的属性时被调用
__set($property, $value)	给未定义的属性赋值时被调用
__isset($property)	对未定义的属性调用isset()时被调用
__unset($property)	对未定义的属性调用unset()时被调用
__call($method, $arg_array)	调用未定义的方法时被调用

__get()和__set()方法用于处理类（或其父类）中未声明的属性。

当客户端代码试图访问未声明的属性时，__get()会被调用，并带一个包含要访问的属性名称的字符串参数。无论从__get()方法返回了什么，都会发送给客户端代码，就好像带有该值的目标属性存在一样。下面是一个简单的例子：

```
class Person {
    function __get( $property ) {
        $method = "get{$property}";
        if ( method_exists( $this, $method ) ) {
            return $this->$method();
        }
    }

    function getName() {
```

① 你可能需要在发布下一个版本时才能修正这个错误并提供给用户，因为用户不能随时得到最新的代码。
——译者注

```
        return "Bob";
    }

    function getAge() {
        return 44;
    }
}
```

访问未定义的属性时，__get()方法被调用。我们实现了__get()来获得属性名并构造新的字符串，把“get”放在属性名之前，然后将组合成的类方法字符串传给method_exists()函数（它接受对象和方法名作为参数），检查方法是否确实存在。如果方法存在，就调用它并把它的返回值传递给客户。所以，如果客户请求一个$name属性：

```
$p = new Person();
print $p->name;
```

getName()方法在后台被调用，输出：

```
Bob
```

如果方法不存在，则什么也不做。用户试图访问的属性被解析为NULL。

__isset()方法和__get()方法相似。当客户在一个未定义的属性上调用isset()时，__isset()被调用。下面看看如何扩展Person：

```
function __isset( $property ) {
    $method = "get{$property}";
    return ( method_exists( $this, $method ) );
}
```

现在，谨慎的用户在使用属性前可以先检查属性是否已经设置：

```
if ( isset( $p->name ) ) {
    print $p->name;
}
```

__set()方法在客户端代码试图要给未定义的属性赋值时被调用，它会接受两个参数：属性名和客户要设定的值，然后我们再决定如何使用这些参数。现在进一步修改Person类：

```
class Person {
    private $_name;
    private $_age;

    function __set( $property, $value ) {
        $method = "set{$property}";
        if ( method_exists( $this, $method ) ) {
            return $this->$method( $value );
        }
    }

    function setName( $name ) {
        $this->_name = $name;
```

```
        if ( ! is_null( $name ) ) {
            $this->_name = strtoupper($this->_name);
        }
    }

    function setAge( $age ) {
        $this->_age =  strtoupper($age);
    }
}
```


在这个例子中，我们使用了设置方法而不是获取方法。如果用户要给未定义的属性赋值，__set()方法就会被调用，其参数为属性名及要赋的值。我们检查了指定的设置方法是否存在，如果存在就调用。这样我们可以过滤掉已经赋值的属性。

注解 在PHP手册中，经常使用静态术语的表达方式（即使用::符号）来讨论类方法与属性，即使该方法和属性并非静态。当提及Person::$name属性时，要注意name属性不一定是静态属性，很可能需要通过对象来访问。

当创建Person对象并尝试设置一个名为Person::$name的属性时，因为这个类没有定义$name属性，所以__set()方法会被调用，其参数为字符串name和要设定的值。这一值如何使用则取决于__set()的实现。本例中，我们将set字符串与属性名称连接在一起，构造了一个类方法setName()。我们发现setName()方法已经存在，并且立即调用它。这样就可以把从外部得到的值存储到一个真实存在的属性（$_name）中。

```
$p = new Person();
$p->name = "bob";
// $_name 属性的值变成 'bob'
```

__unset()和__set()相对应。当unset()在一个未定义的属性上被调用时，__unset()会被调用，并以该属性的名称作为参数，然后你可以根据属性名进行必要的处理。下面的例子使用之前你见过的__set()中的技术，将该属性设置为null。

```
    function __unset( $property ) {
        $method = "set{$property}";
        if ( method_exists( $this, $method ) ) {
            $this->$method( null );
        }
    }
```

__call()方法可能是最有用的拦截器方法。当客户端代码要调用类中未定义的方法时，__call()会被调用。__call()接受两个参数，一个是方法的名称，另一个是传递给要调用方法的所有参数（数组）。__call()方法返回的任何值都会返回给客户，就好像调用一个真实存在的方法一样。

__call()方法对于实现委托也很有用。委托是指一个对象转发或者委托一个请求给另一个对象，被委托的一方替原先对象处理请求。这类似于继承，和在子类中调用父类的方法有点相似。

但在继承时，父类与子类的关系是固定的，而使用委托则可以在代码运行时改变使用的对象，这意味着委托比继承具有更大的灵活性。 我们用例子来解释一下。下面创建一个类将Person类的信息格式化并输出：

```
class PersonWriter {

    function writeName( Person $p ) {
        print $p->getName()."\n";
    }

    function writeAge( Person $p ) {
        print $p->getAge()."\n";
    }
}
```

当然，我们可以通过继承PersonWriter类以不同的方式输出Person类的信息。下面的代码结合使用__call()方法和PersonWriter对象来实现Person类：

```
class Person {
    private $writer;

    function __construct( PersonWriter $writer ) {
        $this->writer = $writer;
    }

    function __call( $methodname, $args ) {
        if ( method_exists( $this->writer, $methodname ) ) {
            return $this->writer->$methodname( $this );
        }
    }

    function getName()  { return "Bob"; }
    function getAge() { return 44; }
}
```

代码中Person类接受一个PersonWriter对象作为构造方法的参数，并将它存储在属性变量$writer中。在__call()方法中，我们使用参数$methodname，检查PersonWriter对象中是否存在同名的方法。如果相应的方法存在，我们就委托PersonWriter对象来处理对方法的调用，把当前类（Person）的实例作为参数传递给PersonWriter对象（使用$this伪变量)。因此，如果这样调用Person类：

```
$person = new Person( new PersonWriter() );
$person->writeName();
```

__call()方法会被调用，然后会在PersonWriter对象中查找writeName()方法，并调用之。这样，我们就可以不用手动在Person类中调用如下委托方法：

```
function writeName() {
    $this->writer->writeName( $this );
}
```

现在Person类神奇地增加了两个新方法。如果你要委托很多方法的处理，那么这样的自动委托可以为你节省很多时间，但代码也会变得有点不太清晰，不易理解。对于外部世界来说，你提供的是一个动态的接口，没有办法进行反射（reflection[①]），客户程序员无法一下子理清头绪。因为类与被委托类之间的交互可能比较模糊——用__call()来调用，而不是用显式的继承关系或者参数类型提示。拦截器方法是非常有用的，但在使用时要慎重考虑，而且最好附上文档，清楚地说明代码的细节。

本书后面还会详细介绍“委托”和“反射”。

4.9 析构方法

我们之前介绍过，在实例化对象时，__construct()方法会被自动调用。PHP 5还提供了一个对应的方法__destruct()。它只在对象被垃圾收集器收集前（即对象从内存中删除之前）自动调用。你可以利用这个方法进行最后必要的清理工作。

例如，有一个类需要把其自身的信息写入数据库。这时可以使用__destruct()方法在对象实例被删除时确保实例把自己保存到了数据库中：

```
class Person {
    private $name;
    private $age;
    private $id;

    function __construct( $name, $age ) {
        $this->name = $name;
        $this->age  = $age;
    }

    function setId( $id ) {
        $this->id = $id;
    }

    function __destruct() {
        if ( ! empty( $this->id ) ) {
            // 保存Person数据
            print "saving person\n";
        }
    }
}
```

这样无论Person对象何时从内存中删除，__destruct()方法都会被调用。当对一个对象调用unset()函数时，或者进程中不再引用某个对象时，对象就被销毁了。下面创建一个Person对象然后再销毁它，可以看到__destruct()方法所起的作用：

```
$person = new Person( "bob", 44 );
$person->setId( 343 );
```

① reflection，指的是程序对运行中的类自身进行检查，并能直接操作程序的内部属性。例如，获得类中各成员的名称并显示出来。——译者注

```
unset( $person );
// 输出:
// 保存person
```

这样的编程技巧很有趣，但在使用的时候要谨慎。__call()和__destruct()这一系列方法有时被称为魔法方法（magic method）。如果你读过魔幻小说，就会知道魔法并不总是一件好事情。魔法可能武断而出乎意料之外，它可能改变规则，并且导致隐性的代价。

例如，在__destruct()中，会把一些意外状况加到客户身上。假设Person类在__destruct()方法中执行了一次数据库写操作。现在有一个开发新手漫不经心地使用Person类，他根本没有注意到__destruct()方法，而是实例化了一组Person对象。通过传值给构造器，他把私下给CEO起的外号传递给了$name属性，并且把$age设置为150。他执行了几次测试脚本，并尝试各种有趣的姓名和年龄组合。

第二天早上，经理把他叫到会议室来，问他数据库中为什么会有这些无礼的Person数据。这是一个教训——不要过于信任魔法。

4.10 使用__clone()复制对象

在PHP 4中，复制对象只是简单地将一个变量赋值给另一个变量。

```
class CopyMe {}
$first = new CopyMe();
$second = $first;
// PHP 4: $second和$first是两个完全不同的对象
// PHP 5及以后的版本: $second和$first指向同一个对象
```

PHP 4的这种用法可能会引发很多bug，因为在变量赋值、调用方法、返回对象时都常常会无意中进行对象复制，而你可能并不知道。更糟的是，我们无法检查两个变量是否指向相同的对象。等值检测只会告诉你两者是否相等（==）或者两者是否相等且都是对象（===），但不会告诉你两者是否指向同一个对象。

在PHP中，对象的赋值和传递都是通过引用进行的。这意味着当我们之前的代码运行在PHP 5时，$first和$second这两个变量包含指向同一个对象的引用，而没有各自保留一份相同的副本。这正是我们处理对象时所希望的，但有时候我们也需要获得一个对象的副本，而不是引用。

PHP提供了clone关键字来达到这个目的。clone使用“值复制”方式（by-value copy）新生成一个对象。

```
class CopyMe {}
$first  = new CopyMe();
$second = clone $first;
// PHP 5及以后的版本: $second和$first现在是两个不同的对象
```

但这样复制对象还有问题。请回顾一下我们在上一节中实现的Person类。默认情况下，每一个Person对象都会有其标识符（即$id属性）。在实际开发中，$id属性可能会与数据库表中的某一条记录一一对应。如果允许复制$id属性，那么可能会有两个完全不同的对象指向数据库中的同一条记录，这显然不是客户程序员想要的结果。此时对一个对象所做的更新会影响另一个，

反之亦然。

幸运的是，在对象上调用clone时，我们可以控制复制什么。我们可以通过实现一个特殊的方法__clone()来达到这个目的（注意所有以两个下划线开头的方法都是PHP内置的方法）。当在一个对象上调用clone关键字时，其__clone()方法就会被自动调用。

实现__clone()方法时，要注意当前方法执行的环境。__clone()是在复制得到的对象上运行的，而不是在原始对象上运行的。我们给Person类添加上__clone()方法：


```
class Person {
    private $name;
    private $age;
    private $id;

    function __construct( $name, $age ) {
        $this->name = $name;
        $this->age  = $age;
    }

    function setId( $id ) {
        $this->id = $id;
    }

    function __clone() {

        $this->id   = 0;
    }
}
```

当在一个Person对象上调用clone时，产生了一个新的副本，并且新副本的__clone()方法会被调用。这意味着我们在__clone()中重新赋值会生效。本例中，我们确保一个新副本对象的$id属性被重设为0。

```
$person = new Person( "bob", 44 );
$person->setId( 343 );
$person2 = clone $person;
// $person2 :
//     name: bob
//     age: 44
//     id: 0.
```

这样的浅复制（shallow copy）可以保证所有基本数据类型的属性被完全复制。在复制对象属性时只复制引用，并不复制引用的对象，这可能不是你所期望的。比如，我们给Person类添加一个Account（账户）对象类型的属性。这个对象有一个$balance（余额）属性，可以将该属性也复制给对象新副本。但我们不希望原来的Person对象和复制得到的新对象具有对同一个账户的引用。

```
class Account {
    public $balance;
    function __construct( $balance ) {
        $this->balance = $balance;
    }
```

```
}

class Person {
    private $name;
    private $age;
    private $id;
    public $account;

    function __construct( $name, $age, Account $account ) {
        $this->name = $name;
        $this->age  = $age;
        $this->account = $account;
    }

    function setId( $id ) {
        $this->id = $id;
    }

    function __clone() {
        $this->id   = 0;
    }
}

$person = new Person( "bob", 44, new Account( 200 ) );
$person->setId( 343 );
$person2 = clone $person;

// 给$person充一些钱
$person->account->balance += 10;
// 结果$person2也得到了这笔钱，这不合理
print $person2->account->balance;
```

输出：

```
210
```

$person对象保存一个指向Account对象的引用，Account对象被标识为public，这样比较方便访问（实际使用中最好限制一下对属性的访问，在需要的时候再添加访问方法）。当创建新副本时，新对象中所保存的引用指向的是$person所引用的同一个Account对象。上例中我们给$person对象的Account属性加钱，发现$person2对象中的余额也增加了。

如果不希望对象属性在被复制之后被共享，那么可以显式地在__clone()方法中复制指向的对象：

```
function __clone() {
    $this->id   = 0;
    $this->account = clone $this->account;
}
```

4.11 定义对象的字符串值

PHP 5中引入的另一个受Java启发的功能是__toString()方法。在PHP 5.2之前，如果你打

印一个对象，PHP就会把对象解析成一个字符串来输出：

```
class StringThing {}
$st = new StringThing();
print $st;
```

```
Object id #1
```

从PHP 5.2起，这段代码会产生下面这样的错误：

```
PHP Catchable fatal error:  Object of class StringThing could not be
converted to string in ...
```

通过实现自己的__toString()方法，你可以控制输出字符串的格式。__toString()方法应当返回一个字符串值。当把对象传递给print或echo时，会自动调用这个方法，并用方法的返回值来替代默认的输出内容。下面在最简的Person类中添加__toString()方法：

```
class Person {
    function getName()  { return "Bob"; }
    function getAge() { return 44; }
    function __toString() {
        $desc  = $this->getName();
        $desc .= " (age ".$this->getAge().")";
        return $desc;
    }
}
```

现在把Person对象打印出来，该对象会被解析为：

```
$person = new Person();
print $person;
```

```
Bob (age 44)
```

对于日志和错误报告，__toString()方法非常有用。__toString()方法也可用于设计专门用来传递信息的类，比如Exception类可以把关于异常数据的总结信息写到__toString()方法中。

4.12　回调、匿名函数和闭包

虽然匿名函数不是严格意义上的面向对象的特性，但它非常有用，值得在此一提，因为可能会在使用回调的面向对象的应用中遇到它。此外，最近在这方面又开发出了一些有趣的新特性。

首先，来看两个类：

```
class Product {
    public $name;
    public $price;
```

```
    function __construct( $name, $price ) {
        $this->name = $name;
        $this->price = $price;
    }
}

class ProcessSale {
    private $callbacks;

    function registerCallback( $callback ) {
        if ( ! is_callable( $callback ) ) {
            throw new Exception( "callback not callable" );
        }
        $this->callbacks[] = $callback;
    }

    function sale( $product ) {
        print "{$product->name}: processing \n";
        foreach ( $this->callbacks as $callback ) {
            call_user_func( $callback, $product );
        }
    }
}
```

设计这段代码的目的是运行各种回调。代码中有两个类。Product只存储$name和$price属性。为简单起见，我将这两个属性设置为公有属性。请记住，在实际的应用中，你可能会将属性设置为私有或者受保护的并提供访问方法。ProcessSale由两个方法构成：registerCallback()接受一个不提示的标量，测试该标量并将其添加到回调数组中。实现测试功能的是内置的_callable()函数，该函数能确保传递进来的值能被call_user_func()或array_walk()等函数调用。

sale()方法接受一个Product对象，输出与该对象有关的一条信息，然后遍历$callback数组属性。该方法会把每个元素传递给调用该代码的call_user_func()，也就是将产品的引用传递给它。下面的所有例子都适用于这种框架。

回调为什么有用？利用回调，你可以在运行时将与组件的核心任务没有直接关系的功能插入到组件中。有了组件回调，你就赋予了其他人在你不知道的上下文中扩展你的代码的权利。

例如，假设ProcessSale类的一个用户想创建一条销售记录。如果该用户可以访问该类，那么他可能会直接在sale()方法中添加记录代码，但有时这种做法并不好。如果他不是ProcessSale类所在的包的维护者，那么他对该方法的修改会在下次更新包时被覆盖。即使他是该组件的维护者，向sale()方法中添加那么多附加的任务也是本末倒置，无法体现该方法的核心功能，这可能会导致该方法跨项目的可用性降低。下一节将再次介绍这些主题。

幸好，我创建了ProcessSale回调。下面创建的回调用来模拟记录过程：

```
$logger = create_function( '$product',
                           'print "    logging ({$product->name})\n";' );
```

```
$processor = new ProcessSale();
$processor->registerCallback( $logger );

$processor->sale( new Product( "shoes", 6 ) );
print "\n";
$processor->sale( new Product( "coffee", 6 ) );
```

我使用create_function()创建回调。你可以看到，它的参数是两个字符串。首先是一个参数列表，接着是函数体。结果通常被称为匿名函数，因为它没有以标准函数的方式命名，但它可被存储在一个变量中，作为参数传递给函数和方法。这就是我所做的：将函数存储在$logger变量中，然后将其传递给ProcessSale::registerCallback()。最后创建了两个产品并将其传递给sale()方法。你已经看到了所发生的事情，处理销售记录（实际上是输出与该产品有关的一条简单的消息），并且执行所有回调。实际的代码如下：

```
shoes: processing
    logging (shoes)

coffee: processing
    logging (coffee)
```

再来看一下create_function()这个例子。看看它有多难看。将要执行的代码放在一个字符串中是永远的痛。你需要转义变量和引号，并且如果回调达到了一定的大小，实际上它会变得十分难读。如果能找到一种优雅的方式来创建匿名函数，这个例子不就能变得整洁一些了吗？PHP 5.3及其后续版本提供了更好的方法来实现该功能。你可以简单地在一条语句中声明并分配函数。使用了新语法后的create_function()示例如下所示：

```
$logger2 = function( $product ) {
   print "    logging ({$product->name})\n";
};

$processor = new ProcessSale();
$processor->registerCallback( $logger2 );

$processor->sale( new Product( "shoes", 6 ) );
print "\n";
$processor->sale( new Product( "coffee", 6 ) );
```

唯一的区别就是匿名变量的创建。你看到了，它变得整洁了许多。我只是以内联方式使用function关键字，并且没有函数名。注意，因为这是一条内联语句，所以在代码块的末尾需要使用分号。当然，如果你想在旧版的PHP上运行代码，执行create_function()时会停顿一段时间。输出结果和前面的例子一样。

当然，回调并一定要是匿名的。你可以使用函数名（甚至是对象引用和方法）作为回调，如下例所示：

```
class Mailer {
    function doMail( $product ) {
        print "    mailing ({$product->name})\n";
```

```
    }
}

$processor = new ProcessSale();
$processor->registerCallback( array( new Mailer(), "doMail" ) );

$processor->sale( new Product( "shoes", 6 ) );
print "\n";
$processor->sale( new Product( "coffee", 6 ) );
```

我创建了Mailer类，该类只有一个方法doMail()，接受$product对象并输出与该对象有关的一条消息。调用registerCallback()时，我传递给它一个数组。数组的第一个元素是$mailer对象，第二个元素是字符串（该字符串与我想要调用的方法的名称匹配）。记住，register-Callback()会检查其参数的可调用性。is_callable()非常智能，能够测试这类数组。数组形式的有效回调应该以对象作为其第一个元素，以方法名作为其第二个元素。这里我通过了这个测试，下面是输出结果：

```
shoes: processing
    mailing (shoes)

coffee: processing
    mailing (coffee)
```

当然，你也可以让方法返回匿名函数，像下面这样：

```
class Totalizer {
    static function warnAmount() {
        return function( $product ) {
            if ( $product->price > 5 ) {
                print "    reached high price: {$product->price}\n";
            }
        };
    }
}

$processor = new ProcessSale();
$processor->registerCallback( Totalizer::warnAmount() );
...
```

除了使用warnAmount()作为匿名函数的工厂方法很方便之外，这里没有什么有趣的内容了。但除了生成匿名函数之外，利用该结构还可以做更多的事情，比如利用闭包。这些新风格的匿名函数可以引用在其父作用域中声明的变量。这个概念有时候很难理解。打个比方来说，就好像匿名函数还记得它被创建时所在的作用域。假设我想让Totalizer::warnAmount()做两件事：首先是让它接受一个随机的目标金额；其次是记录售出产品的总价格。当总价格超出目标金额时，让该函数执行一项操作（你可能猜到了，在本例中就是输出一条消息）。

利用use子句，就可以让匿名函数追踪来自其父作用域的变量：

```
class Totalizer {
```

```
    static function warnAmount( $amt ) {
        $count=0;
        return function( $product ) use ( $amt, &$count ) {
            $count += $product->price;
            print "   count: $count\n";
            if ( $count > $amt ) {
                print "   high price reached: {$count}\n";
            }
        };
    }
}

$processor = new ProcessSale();
$processor->registerCallback( Totalizer::warnAmount( 8) );

$processor->sale( new Product( "shoes", 6 ) );
print "\n";
$processor->sale( new Product( "coffee", 6 ) );
```


Totalizer::warnAmount()返回的匿名函数在其use子句中指定了两个变量：第一个变量是$amt，它是warnAmount()的实参；第二个闭包变量是$count。$count在warnAmount()函数体中声明，初始值为0。注意，在use子句中我在$count变量前面加了一个$。这意味着该变量可以用匿名函数中的引用而不是值来访问。在匿名函数的函数体中，我为$count增加了产品的价格（$product的值），然后测试新的总计值是否超过$amt。如果已经达到目标值，就输出一个通知。

下面是实际的代码：

```
shoes: processing
   count: 6

coffee: processing
   count: 12

   high price reached: 12
```

这段代码说明了回调跟踪了两次调用之间的$count。$count和$amt仍然和函数相关，因为它们出现在声明时所在的作用域中，并且在use子句中指定。

4.13 小结

本章探讨了很多PHP面向对象方面的高级特性。随着你深入阅读本书，一些特性会慢慢熟悉起来。特别是抽象类、异常和静态方法，之后我们会经常提到。

在下一章中，我们会搁置PHP的内置对象特性，讲述如何使用类和函数来帮助你更好地使用对象。

第5章 对象工具

正如前一章所介绍的，PHP通过类和方法等语言结构支持面向对象编程，同时也通过对象相关的函数和内置类为面向对象提供广泛支持。

在本章中，我们将学习一些用于组织、测试和操作对象及类的工具和技术。

本章包括以下内容。

- 包（package）：将代码按逻辑分类打包。
- 命名空间：从PHP 5.3开始，可以将代码元素封装在独立的单元中。
- 包含路径：为你的类库代码设置访问路径。
- 类函数和对象函数：测试对象、类、属性和方法的函数。
- 反射API（Reflection API）：一组强大的内置类，可以在代码运行时访问类信息。

5.1 PHP 和包

包是一组相关类的集合，这些类以某种方式组合在一起。包可以把系统的一部分和其他部分分隔开来。一些程序语言支持包并为它们提供不同的命名空间。PHP现在还没有包的概念，但是PHP 5.3支持命名空间。我们将在下一节讨论这一特性。

由于很长一段时间内还面临旧代码并与之兼容，我们将讲述旧的把类按照包来组织的方式。

5.1.1 PHP 包和命名空间

尽管PHP本身不支持包的概念，开发人员却一直在使用文件系统和命名模式将代码组织成类似于包的结构。首先，我们先介绍命名模式和一个新的相关特性——命名空间支持。

PHP 5.3之前，开发人员必须在全局上下文中命名文件。换句话说，如果你将一个类命名为`ShoppingBasket`，那么这个类在整个系统中立即可用。这会导致两个主要的问题。最严重的问题就是可能导致命名冲突。你可能认为“没有这种可能性”，因为你记得要给每个类取不同的名字，对吗？但问题是我们对库代码的依赖程度与日俱增，这当然是好事，因为这样做能提高代码的重用率，但如果你的项目像下面这样做的话：

```
// my.php

require_once "useful/Outputter1.php"
```

```
class Outputter {
    // 输出数据
}
```

并且包含的文件这样做的话：

```
// useful/Outputter1.php
class Outputter {
    //
}
```

你能猜到，对吧？会发生下面这样的事：

```
Fatal error: Cannot redeclare class Outputter in ../useful/Outputter1.php on
 line 3
```

当然，针对该问题有一种常见的解决方法：将包名放在类名前面，这样就能保证类名的唯一性。

```
// my.php

require_once "useful/Outputter2.php";
class my_Outputter {
    // 输出数据
}

// useful/Outputter2.php

class useful_Outputter {
    //
}
```

这里的问题就是，项目越来越复杂时，类名也会变得越来越长。这不是什么大问题，但却会影响代码的可读性，并且在工作中类名会越来越难记。此外，输入错误也会浪费掉总体的编码时间。

在未来的几年中，我们仍将被这个问题所困扰，因为我们大多数人都会在很长一段时间中以这样那样的形式维护遗留代码。鉴于这个原因，本章后面还是使用原来的方法处理包。

1. 命名空间来救场

命名空间是人们期待已久的一个特性。本书上一版提到了一个建议实现，用于在PHP 6的开发代码中加入这个特性。程序员们时不时就会在邮件列表中讨论一番这个特性的优缺点。

对于PHP 5.3来说，这些争论都是学术性的。命名空间是语言的一部分，并且它们始终存在。

那么，什么是命名空间？从本质上说，命名空间就是一个容器，你可以将类、函数和变量放在其中。在命名空间中，你可以无条件地访问这些项。在命名空间之外，必须导入或引用命名空间，才能访问它所包含的项。

搞不清楚吗？那么来看一个例子吧！下面我使用命名空间重写了上一个例子：

```
namespace my;
require_once "useful/Outputter3.php";
```

```
class Outputter {
    // 输出数据
}

// useful/Outputter3.php
namespace useful;

class Outputter {
    //
}
```

注意namespace关键字。你可能已经猜到了，它用于创建命名空间。如果你正在使用这个功能，那么命名空间声明必然是其文件中的第一条语句。我创建了两个命名空间：my和useful。通常你可能需要创建层次更深的命名空间。你可以从创建一个组织或项目标识符开始，然后通过包进一步限定。PHP支持声明嵌套的命名空间。为此，你只需使用反斜杠字符将每一层分开就可以了。

```
namespace com\getinstance\util;

class Debug {
    static function helloWorld() {
        print "hello from Debug\n";
    }
}
```

如果打算提供代码库，我可能会使用一个域：getinstance.com，然后使用该域名作为命名空间。这是Java开发人员通常用来命名包的技巧。他们会反转域名，这样就可以按从最通用到最具体的方式运行。标识了代码库以后，接下来我会定义包。在本例中，我使用util。

那么我怎样调用方法呢？实际上，这取决于从何处进行调用。如果从命名空间内部调用方法，你可以直接调用该方法：

```
Debug::helloWorld();
```

这称为非限定名。因为我已经位于com\getinstance\util命名空间中了，所以不必在类名前加任何种类的路径。如果打算从命名空间环境之外访问类，可以这样做：

```
com\getinstance\util\Debug::helloWorld();
```

下列代码的输出结果是什么？

```
namespace main;

com\getinstance\util\Debug::helloWorld();
```

这是一个很难回答的问题。实际上，下面是我的输出：

```
PHP Fatal error: Class 'main\com\getinstance\util\Debug' not found in .../listing5.04.
 php on line 12
```

因为我使用了相对命名空间。PHP会在main命名空间中寻找com\getinstance\util，但没有找到。就像创建绝对URL和文件路径时，以分隔符作为开始一样，你也可以使用这种方式创建绝对命名空间。这个示例版本修正了前面的错误：

```
namespace main;

\com\getinstance\util\Debug::helloWorld();
```

这个前导的反斜杠告诉PHP从根命名空间而不是从当前命名空间开始搜索。

但命名空间不应该在减少输入方面有所帮助吗？当然，Debug类声明很短，但这些调用和使用老旧的命名规范中的调用一样啰嗦。使用use关键字可以解决这一问题。利用该关键字，你可以为当前命名空间中的其他命名空间起别名。下面给出一个例子：


```
namespace main;
use com\getinstance\util;
util\Debug::helloWorld();
```

导入com\getinstance\util命名空间，并隐式地为其使用了别名util。注意，我没有使用前导反斜线字符作为开始。这里从全局命名空间而不是从当前命名空间搜索要使用的参数。如果不想引用命名空间，可以导入Debug类本身：

```
namespace main;
use com\getinstance\util\Debug;
util\Debug::helloWorld();
```

但是如果main命名空间中已经包含了Debug类，会发生什么？我认为你可以猜到。下面是代码和一些输出：

```
namespace main;
use com\getinstance\util\Debug;
class Debug {
    static function helloWorld() {
        print "hello from main\Debug";
    }
}

Debug::helloWorld();
```

```
PHP Fatal error: Cannot declare class main\Debug because the name is already in use
 in .../listing5.08.php on line 13
```

我似乎兜了个圈子，又遇到了类命名冲突的问题。幸好该问题有一个解决方案。我可以显式使用别名：

```
namespace main;

use com\getinstance\util\Debug as uDebug;

class Debug {
    static function helloWorld() {
```

```
        print "hello from main\Debug";
    }
}

uDebug::helloWorld();
```

在use关键字之后使用as子句，可以将别名Debug修改为uDebug。

如果在命名空间中编写代码，而你想访问的类保存在全局（非命名空间的）空间中，那么你可以在该名称前加反斜杠。下面是在全局空间中声明的方法：

```
// global.php: 无命名空间

class Lister {
    public static function helloWorld() {
        print "hello from global\n";
    }
}
```

下面是一些引用该类的命名空间代码：

```
namespace com\getinstance\util;
require_once 'global.php';
class Lister {
    public static function helloWorld() {
        print "hello from ".__NAMESPACE__."\n";
    }
}

Lister::helloWorld();  // 访问本地
\Lister::helloWorld(); // 访问全局空间
```

命名空间的代码声明了它自己的Lister类。非限定名访问本地版本。使用单个反斜杠限定的名称将访问全局空间中的类。

下面是上一个代码段的输出。

```
hello from com\getinstance\util
hello from global
```

这段输出值得一看，因为它说明了__NAMESPACE__常量的操作。这将输出当前的命名空间，在调试时很有用。

使用你已经见过的语法，可以在同一个文件中声明多个命名空间。你还可以使用命名空间关键字加大括号形式的语法。

```
namespace com\getinstance\util {
    class Debug {
        static function helloWorld() {
            print "hello from Debug\n";
        }
    }
}
```

```
namespace main {
    \com\getinstance\util\Debug::helloWorld();
}
```

如果必须将多个命名空间组合到一个文件中，那么推荐上面这种做法，但通常认为为每个文件定义一个命名空间是最好的做法。

大括号语法提供的一项功能就是可以在一个文件中切换到全局空间。前面我曾使用require_once从全局空间获取代码。实际上，我也可以使用刚刚介绍的这种命名空间语法，并在文件中执行所有操作。

```
namespace {
    class Lister {
        //...
    }
}

namespace com\getinstance\util {
    class Lister {
        //...
    }

    Lister::helloWorld();  // 访问本地
    \Lister::helloWorld(); // 访问全局空间
}
```

如果打开命名空间块的时候不指定名称，那就可以进入全局空间。

注解 不能在同一个文件中同时使用大括号命名空间语法和行命名空间语法。你必须选择其中的一种，并且在整个文件中坚持使用这种语法。

---

2. 使用文件系统模拟包

无论在哪种版本的PHP中，我们都可以使用提供了包结构的文件系统来组织类。比如，我们可以创建util和business两个目录并用require_once()语句包含类文件，如下所示：

```
require_once('business/Customer.php');
require_once('util/WebTools.php');
```

也可以使用include_once()达到相同的效果。include()和require()语句的不同在于它们如何处理错误。使用require()调用文件发生错误时，将会停止整个程序；调用include()时遇到相同的错误，则会生成警告并停止执行包含文件，跳出调用代码然后继续执行。require()和require_once()用于包含库文件时更加安全，而include()和include_once()则适用于加载模板这样的操作。

注解 require()和require_once()都是语句而不是函数，这意味着使用它们的时候可以省略括号。就我而言，我更喜欢使用括号。但如果使用括号，要小心一些喜欢卖弄理论的人来烦你，他们会尝试解释你的错误。

图5-1以文件资源管理器的方式展示了util和business包。

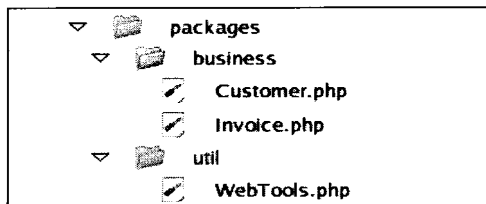


图5-1　使用文件系统来组织PHP包

注解　require_once()接受文件路径作为参数，然后把文件包含到当前脚本中。函数只能在文件没有被包含过的情况下才能包含它。这种只使用一次的方法在访问库代码时特别有用，因为它防止了类和方法的重复定义。在同一个文件被包含到脚本的不同部分时，如果使用了require()或include()之类的函数而不是require_once()，就有可能出现类或方法重复定义的错误。

使用require()和require_once()比使用include()和include_once()函数好。因为使用require()函数访问文件出错时，会报告致命错误并停止整个脚本的执行，而使用include()函数访问文件时，遇到相同的错误则只会引起包含文件执行的中断，然后在调用脚本中生成警告。前者行为更加激烈，因而更加安全。

和使用require()相比，require_once()需要额外的开销。如果想尽可能地减少系统执行时间，应该考虑使用require()。在很多情况下，效率和便利之间是平衡关系。

就PHP而言，这个结构没有什么特别的地方。你只是将库脚本放在不同的目录中。它确实有助于形成清晰的组织结构，并且可以和命名空间或者命名约定同时使用。

3. PEAR风格的命名方式

即使PHP 5.3可用的那一刻便更新到该版本，你可能也不会一直使用命名空间。通常，老板、客户和主机托管公司升级很慢，可能还没有升级到PHP 5.3版本，通常他们有充分的理由这么做。即使你的项目运行在最新版本的PHP上，仍可能会发现使用的还是旧版的代码。如果你有充足的时间为使用命名空间重新编写项目，那就太棒了，但大多数人没有这么幸运。

那么在没有命名空间的支持下，我们应该如何处理命名冲突呢？一个办法是使用PEAR包的命名规则。

注解　PEAR（PHP Extension and Application Repository，PHP扩展和应用程序库）是PHP官方维护的软件包集合，也是增强PHP功能的工具。PEAR核心包默认随着PHP绑定发布，其他非核心包可以使用命令行工具来添加。你可以在http://pear.php.net上找到各种PEAR包。在第15章中，我们会介绍PEAR相关的其他内容。

之前说过，PEAR使用文件系统来定义包。然后，每个类都根据包路径来命名，每个路径名

以下划线来分隔。

例如，PEAR有一个XML包，XML包中有一个RPC子包，RPC包中有一个Server.php文件。在Server.php中定义的类并不是直接叫做Server，否则迟早会和PEAR项目或用户代码中另一个Server类冲突。所以，这个类被命名为XML_RPC_Server。这个类名很冗长，不讨人喜欢，但是我们一看到这样的类名就知道它所在的环境。

4. 包含路径

组织组件时，你必须记住两件事。第一件事我已经介绍过了，那就是文件系统中文件和目录存放的位置。此外，你还要考虑组件之间互相访问的方式。现在我们来研究一下包含路径的问题。当我们包含一个文件时，可以使用相对路径（相对于当前工作目录），也可以使用绝对路径来引用该文件。

到目前为止，我们所看到的例子使用的都是相对路径，如：

```
require_once('business/User.php');
```

但这需要当前的工作目录中包含business目录。如果以后business目录不存在了，就需要修改代码。使用相对路径包含库时，通常会使用冗长的require_once()语句：

```
require_once('../../projectlib/business/User.php');
```

当然，也可以使用绝对路径：

```
require_once('/home/john/projectlib/business/User.php');
```

没有哪一种方法是完美的。使用绝对路径太过详细，并且把库文件固定在某个地方了。

使用绝对路径时，我们把库绑定到了特定的文件系统。无论何时在新服务器上安装写好的程序时，所有的require语句都需要为新的文件路径而改变。

使用相对路径，我们把脚本工作目录和类库目录的位置关系固定住了。这样如果不修改require()语句，就无法重新定位文件系统中的文件，而且不通过复制文件很难实现项目间共享。无论哪种情况，都失去了“包”在所有附加目录中的意义。它到底是business包，还是projectlib/business包呢？

要使程序代码不再直接依赖于类库代码程序中包含的类库，需要将调用类库的代码和类库代码进行解耦（decouple，使依赖不再过于紧密），即让

```
business/User.php
```

可以在系统的任何地方被引用。我们只需把包放到include_path指定引用的目录中即可。通常，可以在PHP的核心配置文件php.ini中设置include_path。它定义了一个目录列表，在类Unix系统中以冒号分隔，在Windows系统中则以分号分隔。

```
include_path = ".:/usr/local/lib/php-libraries"
```

如果使用Apache作为Web服务器，还可以在服务器应用程序的配置文件（通常是httpd.conf）或使用下面的语法在Apache配置文件（通常是.htaccess）中设置include_path：

```
php_value   include_path   value    .:/usr/local/lib/php-libraries
```

注解 .htaccess文件在一些主机托管公司提供的Web空间中特别有用，该文件提供了对服务器环境非常有限的访问权。

在使用文件系统函数（如fopen()或require()）时，如果没有找到相应的文件目录，就会在include_path指定的目录中自动查找——从include_path中指定的第一个目录开始找起（要使用本功能，需要指定fopen()函数的第三个参数为true）。直到找到目标文件时，停止搜索，由文件函数执行任务。

因此，把包目录放到包含目录以后，只需要在require()语句中指定包和文件。①

你可以修改php.ini文件，增加目录到include_path中来构建自己的类库。（注意，对于PHP服务器模块需要重启服务器使修改生效）。

如果没有修改php.ini文件的权限，可以在程序中使用set_include_path()函数来设置包含路径。set_include_path()函数接受包含路径（和在php.ini中出现的一样）并仅为当前进程改变include_path设置。php.ini文件可能已经为include_path定义了一个有效的值，因此我们可以不用覆盖这个值，直接使用get_include_path()函数来访问它并加入自己的目录。下面的代码将目录添加到当前的包含路径：

```
set_include_path( get_include_path().":/home/john/phplib/");
```

如果是在Windows平台下工作，要使用分号而不是冒号来分隔每个目录路径。

5.1.2 自动加载

在某些情况下，你可能希望一个文件定义一个类，一一对应，便于管理。这种方法会有额外的开销（包含文件会带来开销），但是这种组织方法非常有用，特别是系统需要在运行时使用新类的情况下（例如你将在第11章和第12章中看到的Command模式）。使用这种方法，每个类文件的名称都和它所包含的类名相关，因此可以在名为ShopProduct.php的文件中定义ShopProduct类。另外也可使用PEAR命名规则，把文件命名为ShopProduct.php，但是类可能要根据它的路径命名为business_ShopProduct。

PHP 5引入了__autoload()拦截器方法来自动包含类文件。__autoload()应该被写成单个参数的方法。当PHP引擎遇到试图实例化未知类的操作时，会调用__autoload()方法（如果已经定义），并将类名当做字符串参数传递给它。__autoload()的编写者应该自己定义一种策略来定位和包含缺失的类文件。

下面定义一个__autoload()方法：

```
function __autoload( $classname ) {
    include_once( "$classname.php" );
}

$product = new ShopProduct( 'The Darkening', 'Harry', 'Hunter', 12.99 );
```

① 例如，原来使用require_once('/home/john/projectlib/business/User.php');来包含类库文件，当将/home/john/projectlib/目录加入include_path后，只需要使用require_once('business/User.php');即可。——译者注

如果没有包含定义了ShopProduct类的文件，那么ShopProduct的实例化就会失败。在这里，PHP引擎知道我们定义了__autoload()函数并将ShopProduct字符串作为参数传递给它。这里只是尝试包含ShopProduct.php文件。当然，只有文件在当前工作目录或包含目录中，才会正常工作。在这里，我们没有更简单的处理包的方法了。如果使用PEAR命名规则，则可以这样实现：

```
function __autoload( $classname ) {
    $path = str_replace('_', DIRECTORY_SEPARATOR, $classname );
    require_once( "$path.php" );
}

$y = new business_ShopProduct();
```

如上所示，__autoload()函数将$classname中的下划线转换为目录分隔（DIRECTORY_SEPARATOR）字符（Unix系统中为/）。我们尝试包含类文件（business/shopProduct.php）。如果类文件存在，并且所包含的类命名正确，就可以正确地实例化对象。当然，这需要程序员遵循在类名中不使用下划线的命名习惯（除了在分隔包时）。

命名空间怎么样？它需要测试反斜线字符，如果该字符存在的话，就添加一个转换：

```
function __autoload( $classname ) {
    if ( preg_match( '/\\\\/', $classname ) ) {
        $path = str_replace('\\', DIRECTORY_SEPARATOR, $classname );
    } else {
        $path = str_replace('_', DIRECTORY_SEPARATOR, $classname );
    }
    require_once( "$path.php" );
}
```

同样，我又对类文件和目录以及它们与命名空间或PEAR风格的类名之间的关系做了一些假设。考虑到导入和使用别名的灵活性，你关心的可能是在命名空间中调用类的各种方式。毕竟我可以对business\ShopProduct使用任意的别名，例如Percy。传递给__autoload的值总被规范化成完全限定的名称，没有前导的反斜杠，这是好事。

__autoload()方法是一种根据类和文件的结构，管理类库文件包含的有效方法。

注解　__autoload是个强大的工具，但也存在一些限制。尤其是，在一个进程中你只能定义它一次。如果需要动态地修改自动加载函数，那么你应该看一下spl_autoload_register函数（http://www.php.net/spl_autoload_register），它提供了那项功能。

5.2　类函数和对象函数

PHP提供了一系列强大的函数来检测类和对象。这有什么用处呢？类不是都写在代码里吗？如果想了解类和对象，直接查看代码不就行了？

事实上，在PHP程序运行时你可能无法知道正在使用的类是哪个。例如，你要设计一个能够透明地被第三方类使用的系统。在这种情况下，你很可能需要实例化一个只给出类名的对象。PHP

允许使用字符串来动态地引用类：

```
// Task.php

namespace tasks;

class Task {
    function doSpeak() {
        print "hello\n";
    }
}


// TaskRunner.php

$classname = "Task";

require_once( "tasks/{$classname}.php" );
$classname = "tasks\\$classname";
$myObj = new $classname();
$myObj->doSpeak();
```

你可以从配置文件或通过比较Web请求和目录内容获得赋值给$classname的字符串，然后利用该字符串来加载类文件，实例化需要的对象。注意，在这个代码段中，我已经构造了一个命名空间限定条件（namespace qualification）

如果想让系统能够运行用户创建的插件，这个功能非常有用。在开发实际项目时，需要先检查类是否存在、它是否拥有将要使用的方法等，以避免可能存在的风险。

现在，PHP中的一些类函数已被PHP 5中功能更强大的反射API（本章稍后将会介绍）所取代，然而这些类函数的简单易用使它们在很多情况下成为首选。

5.2.1　查找类

class_exists()函数接受表示类的字符串，检查并返回布尔值。如果类存在，则返回true，否则返回false。

使用该函数，可以使之前的代码段更安全一些。

```
// TaskRunner.php
$classname = "Task";

$path = "tasks/{$classname}.php";
if ( ! file_exists( $path ) ) {
    throw new Exception( "No such file as {$path}" );
}

require_once( $path );
$qclassname = "tasks\\$classname";
if ( ! class_exists( $qclassname ) ) {
    throw new Exception( "No such class as $qclassname" );
}
```

```
$myObj = new $qclassname();
$myObj->doSpeak();
```

当然，我们无法确定该类的构造方法是否需要参数。在这个级别的安全上，应该求助于本章稍后介绍的反射API。不过，我们还是可以在使用类之前用class_exists()确定它是否存在。

注解 记住，在使用任何由外部提供的数据之前都要小心检测和处理。在文件路径的情况下，应该转换或删除点号和目录分隔符，防止粗心的用户改变目录和包含不需要的文件。

也可以用get_declared_classes()函数来获得脚本进程中定义的所有类的数组。

```
print_r( get_declared_classes() );
```

它将会列出用户定义的类和PHP内置的类。注意它只返回在函数调用时已声明的类。你可以继续运行require()或require_once()来增加脚本中类的数目。

5.2.2 了解对象或类

之前介绍过，我们可以使用类的类型提示限制方法的参数类型，但不能用它来确定对象的类型。在写作本书的时候，PHP不允许限制从方法或函数返回的对象的类型，显然这是考虑到了将来可能的扩展。

PHP中有很多用于检测对象类型的基本工具。首先，可以使用get_class()函数检查对象的类。它接受任何对象作为参数，并以字符串的形式返回类名。

```
$product = getProduct();
if ( get_class( $product ) == 'CdProduct' ) {
    print "\$product is a CdProduct object\n";
}
```

在这段代码中，我们从getProduct()函数获得了一些东西。要完全确定它是一个CdProduct对象，可以使用get_class()方法。

注解 第3章已经介绍过CdProduct和BookProduct类。

下面是getProduct()函数：

```
function getProduct() {
    return new CdProduct(    "Exile on Coldharbour Lane",
                                "The", "Alabama 3", 10.99, 60.33 );
}
```

getProduct()只是简单地实例化并返回CdProduct对象。在本节中，我们会更好地使用这个函数。

get_class()函数是一个比较特别的工具。我们经常想确认一个类的类型。我们可能想知道对象是否属于ShopProduct家族，但是并不关心这个类是BookProduct还是CdProduct。为此，PHP提供了instanceof操作符。

注解　PHP 4并不支持instanceof关键字，而是提供了is_a()函数，但是这个函数在PHP 5.0中不再使用了。从PHP 5.3开始，又重新启用了is_a()。

instanceof操作符有两个操作数，要检测的对象在关键字左边，类或接口名在右边。如果对象是给定类型的实例，则返回true。

```
$product = getProduct();
if ( $product instanceof ShopProduct  ) {
    print "\$product is a ShopProduct object\n";
}
```

5.2.3　了解类中的方法

可以使用get_class_methods()函数来得到一个类中所有方法的列表。该函数需要一个类名作为参数，返回包含类中所有方法名的数组。

```
print_r( get_class_methods( 'CdProduct' ) );
```

假设CdProduct类存在，可以看到类似这样的输出结果：

```
Array
(
    [0] => __construct
    [1] => getPlayLength
    [2] => getSummaryLine
    [3] => getProducerFirstName
    [4] => getProducerMainName
    [5] => setDiscount
    [6] => getDiscount
    [7] => getTitle
    [8] => getPrice
    [9] => getProducer
)
```

在这个例子中，我们将类名传递给get_class_methods()，并用print_r()函数打印出返回的数组。另外，也可以传递对象给get_class_methods()，将得到相同的结果。

除非你运行的是早期的PHP 5版本，否则只有声明为public的方法才会显示到这个结果列表中。

正如之前见到的，我们可以以字符串变量的形式给出方法名，然后通过对象动态地调用类方法，如下所示：

```
$product = getProduct();     //获得对象
$method = "getTitle";        //定义方法名
print $product->$method();   //调用方法
```

当然，这可能存在风险。如果类方法不存在，会发生什么呢？脚本会和预期的一样因发生错误而失败。不过还好我们已经学会了一种检测方法是否存在的办法：

```
if ( in_array( $method, get_class_methods( $product ) ) ) {
    print $product->$method(); //调用方法
}
```

在上面的代码中，在调用某个方法之前，先检测该方法是否存在于get_class_methods()返回的数组中。其实PHP为此提供了更专业的工具，你可以用函数is_callable()和method_exists()来检查。两个函数中is_callable()更高级些，它接受字符串变量形式的方法名作为第一个参数，如果类方法存在且可被调用，则返回true。如果要检测类中的方法可否被调用，可以给函数传递一个数组而不是类方法名作为参数。数组必须包含对象或类名，以将其作为它的第一个元素，要检查的方法名则作为第二个元素。如果该方法在类中存在，函数会返回true。


```
if ( is_callable( array( $product, $method) ) ) {
    print $product->$method(); //调用方法
}
```

is_callable()可以接受另一个参数：一个布尔值。如果将该参数设置为true，函数仅检查给定方法或函数名称的语法是否正确，而不检查其是否真正存在。

method_exists()函数的参数为一个对象（或者类名）和一个方法名，并且如果给定方法在对象的类中存在，则返回true。

```
if ( method_exists( $product, $method ) ) {
    print $product->$method(); //调用方法
}
```

警告 在PHP 5中，一个方法存在并不意味着它就可以被调用。对于private、protected和public方法，method_exists()都返回true。

5.2.4 了解类属性

就像查询类的方法一样，你也可以查询类的属性。get_class_vars()函数接受类名作为参数，返回关联数组。在返回的数组中，属性名作为键名，属性值作为键值。下面我们用它对CdProduct对象进行检测。为了说明这个问题，我们给类增加了一个public属性：CdProduct::$coverUrl。

```
print_r( get_class_vars( 'CdProduct' ) );
```

只显示public属性：

```
Array
(
    [coverUrl] =>
)
```

5.2.5 了解继承

类函数也允许我们绘制继承关系。例如，可以使用get_parent_class()来找到一个类的父

类。这个函数需要一个对象或类名作为参数，并且如果父类存在的话，就返回父类的名字。如果不存在这样的类，即我们检测的类没有父类，那么该函数就返回false。

```
print get_parent_class( 'CdProduct' );
```

就像你所期待的那样，这会得到父类ShopProduct。

也可以使用is_subclass_of()函数检测类是否是另一个类的派生类。它需要接受一个子类对象和父类的名字。如果第二个参数是第一个参数的父类的话，该函数就返回true。

```
$product = getProduct(); //获得对象
if ( is_subclass_of( $product, 'ShopProduct' ) ) {
    print "CdProduct is a subclass of ShopProduct\n";
}
```

is_subclass_of()只会告诉你类的继承关系，而不会告诉你类是否实现了一个接口。因此，如果需要检测一个类是否实现了某个接口，应当使用instanceof操作符。或者你可以使用一个函数，该函数是SPL（Standard PHP Library，标准PHP类库）的一部分。class_implements()使用一个类名或一个对象引用作为参数，并且返回一个由接口名构成的数组。

```
if ( in_array( 'someInterface', class_implements( $product )) ) {
    print "CdProduct is an interface of someInterface\n";
}
```

5.2.6 方法调用

我们已经遇到过使用字符串来动态调用方法的例子：

```
$product = getProduct();     //获得对象
$method = "getTitle";        //定义方法名
print $product->$method();   //调用方法
```

PHP还提供call_user_func()方法来达到相同的目的。call_user_func()可以调用方法或函数。要调用一个函数，需要将字符串作为它的第一个参数：

```
$returnVal = call_user_func("myFunction");
```

要调用类方法，则需要一个数组。数组的第一个元素是一个对象，第二个元素则是要调用的方法名。

```
$returnVal = call_user_func( array( $myObj, "methodName") );
```

你也可以传递任意参数给call_user_func()，作为目标方法或函数所需要的参数，如下所示：

```
$product = getProduct(); //获得对象
call_user_func( array( $product, 'setDiscount' ), 20 );
```

我们的动态调用等价于：

```
$product->setDiscount( 20 );
```

也可以直接用字符串代替方法名，如下所示：

```
$method = "setDiscount";
$product->$method(20);
```

call_user_func()方法很有用，但call_user_func_array()函数更好用。call_user_func_array()的使用方法（如选择目标方法或函数等）都和call_user_func()相同，但它把目标方法所需的任何参数当做数组来接受。

那么它为什么好用呢？因为有时你得到的参数是数组的形式。除非你预先知道要处理的参数数目，否则很难传递它们。在第4章中，我们看到拦截器方法可以创建delegator（委托器）类。下面是__call()方法的简单例子：

```
function __call( $method, $args ) {
    if ( method_exists( $this->thirdpartyShop, $method ) ) {
        return $this->thirdpartyShop->$method( );
    }
}
```

正如我们看到的，当未定义的方法被客户端代码调用时，会调用__call()方法。在这个例子中，在$thirdpartyShop属性中保存了一个对象。如果在存储的对象中找到的方法和$method参数匹配，则调用它。但要注意，这里我们假设目标方法不需要任何参数，这样可能会出问题。当编写__call()方法时，无法告诉调用代码在调用中的$args数组有多大。如果直接传递$args给委托方法（delegate method），将传递一个单独的数组参数，而不是它所期望的独立参数。call_user_func_array()完美地解决了这个问题：

```
function __call( $method, $args ) {
    if ( method_exists( $this->thirdpartyShop, $method ) ) {
        return call_user_func_array(
                    array( $this->thirdpartyShop,
                    $method ), $args );
    }
}
```

5.3 反射 API

PHP中的反射API就像Java中的java.lang.reflect包一样。它由一系列可以分析属性、方法和类的内置类组成。它在某些方面和已经学过的对象函数相似，比如get_class_vars()，但是更加灵活，而且可以提供更多信息。反射API也可与PHP最新的面向对象特性一起工作，如访问控制、接口和抽象类。旧的类函数则不太容易与这些新特性一起使用。

5.3.1 入门

反射API不仅仅被用于检查类。例如，ReflectionFunction类提供了关于给定函数的信息，ReflectionExtension类可以查看编译到PHP语言中的扩展。表5-1列出了API中的部分类。

表5-1 反射API的部分类

类	描 述
Reflection	为类的摘要信息提供静态函数export()
ReflectionClass	类信息和工具
ReflectionMethod	类方法信息和工具
ReflectionParameter	方法参数信息
ReflectionProperty	类属性信息
ReflectionFunction	函数信息和工具
ReflectionExtension	PHP扩展信息
ReflectionException	错误类

利用反射API中的这些类，可以在运行时访问对象、函数和脚本中的扩展的信息。

由于反射API非常强大，你应该经常使用反射API而少用类和对象函数。一会儿你就会发现它是不可缺少的类测试工具。例如，你也许想要生成类结构的图表或文档，或者想保存对象信息到数据库，检查对象的访问方法（getter和setter）来提取字段名。反射的另一用途是根据命名规则创建一个调用模板类中方法的框架。

5.3.2 开始行动

我们已经学过一些用于检查类属性的函数，这些函数很有用，但经常有一定的局限性。现在我们来看一个可以完成这项任务的工具——ReflectionClass。ReflectionClass提供揭示给定类所有信息的方法，无论这个类是用户定义的还是PHP自带的内置类。ReflectionClass的构造方法接受类名作为它的唯一参数：

```
$prod_class = new ReflectionClass( 'CdProduct' );
Reflection::export( $prod_class );
```

创建ReflectionClass对象后，就可以使用Reflection工具类输出CdProduct类的相关信息。Reflection有一个静态方法export()，用于格式化和输出Reflection对象管理的数据（即任何实现Reflector接口的类的实例）。下面是调用Reflection::export()所生成的输出摘要：

```
Class [ <user> class CdProduct extends ShopProduct ] {
  @@ fullshop.php 53-73
  - Constants [0] {
  }
  - Static properties [0] {
  }
  - Static methods [0] {
  }
  - Properties [2] {
    Property [ <default> private $playLength ]
    Property [ <default> protected $price ]
  }

  - Methods [10] {
    Method [ <user, overwrites ShopProduct, ctor> public method __construct ] {
```

```
          @@ fullshop.php 56 - 61

          - Parameters [5] {
            Parameter #0 [ <required> $title ]
            Parameter #1 [ <required> $firstName ]

            Parameter #2 [ <required> $mainName ]

            Parameter #3 [ <required> $price ]
            Parameter #4 [ <required> $playLength ]
          }
        }

        Method [ <user> public method getPlayLength ] {
          @@ fullshop.php 63 - 65
        }

        Method [ <user, overwrites ShopProduct, prototype ShopProduct> public method
     getSummaryLine ] {
          @@ fullshop.php 67 - 71
        }
      }
    }
```

可以看到，Reflection::export()可以提供类的相关信息。Reflection::export()提供了CdProduct几乎所有的信息，包括属性和方法的访问控制状态、每个方法需要的参数以及每个方法在脚本文档中的位置。将该函数与调试函数var_dump()相比较，var_dump()函数是汇总数据的通用工具，但使用var_dump()在提取摘要前必须实例化一个对象，而且它也无法提供像Reflection::export()提供的那么多的细节。

```
$cd = new CdProduct("cd1", "bob", "bobbleson", 4, 50 );
var_dump( $cd );
```

输出结果：

```
object(CdProduct)#1 (6) {
  ["playLength:private"]=>
  int(50)
  ["title:private"]=>
  string(3) "cd1"
  ["producerMainName:private"]=>
  string(9) "bobbleson"
  ["producerFirstName:private"]=>
  string(3) "bob"
  ["price:protected"]=>
  int(4)
  ["discount:private"]=>
  int(0)
}
```

var_dump()和它的姊妹函数print_r()是检测PHP代码中数据的利器，但对于类和函数，反射API提供了更高层次的功能。

5.3.3 检查类

Reflection::export()可以为调试提供大量有用的信息，我们还可以通过特定的方式使用API。下面直接使用Reflection类。

之前你已经知道如何实例化ReflectionClass对象：

```
$prod_class = new ReflectionClass( 'CdProduct' );
```

接着，让我们使用ReflectionClass对象来研究脚本中的CdProduct。这个类属于哪一种类型呢？可以创建实例吗？下面这个函数回答了这些问题：

```
function classData( ReflectionClass $class ) {
  $details = "";
  $name = $class->getName();
  if ( $class->isUserDefined() ) {
    $details .= "$name is user defined\n";
  }
  if ( $class->isInternal() ) {
    $details .= "$name is built-in\n";
  }
  if ( $class->isInterface() ) {
    $details .= "$name is interface\n";
  }
  if ( $class->isAbstract() ) {
    $details .= "$name is an abstract class\n";
  }
  if ( $class->isFinal() ) {
    $details .= "$name is a final class\n";
  }
  if ( $class->isInstantiable() ) {
    $details .= "$name can be instantiated\n";
  } else {
    $details .= "$name can not be instantiated\n";
  }
  return $details;
}

$prod_class = new ReflectionClass( 'CdProduct' );
print classData( $prod_class );
```

我们创建一个ReflectionClass对象，通过传递类名CdProduct给ReflectionClass的构造方法来把它指派给变量$prod_class，然后$prod_class被传递给函数classData()，以证明一些方法可以被用于查询类。

这些方法是自解释的，不过这里还是对每个代码片段做简短说明。

- ReflectionClass::getName()返回要检查的类名。
- 如果类已经在PHP代码中定义，ReflectionClass::isUserDefined()方法返回true;如果是内置类，则ReflectionClass::isInternal()返回true。
- 可以用ReflectionClass::isAbstract()来测试一个类是否是抽象的，用ReflectionClass::isInterface()来测试类是否是接口。

- 如果想得到类的实例，可以用ReflectionClass::isInstantiable()测试是否可行。

你甚至可以检查用户自定义类的相关源代码。ReflectionClass对象提供自定义类所在的文件名及文件中类的起始和终止行。

下面是利用ReflectionClass来获取源代码的简单示例：

```
class ReflectionUtil {
  static function getClassSource( ReflectionClass $class ) {
    $path = $class->getFileName();
    $lines = @file( $path );
    $from = $class->getStartLine();
    $to   = $class->getEndLine();
    $len  = $to-$from+1;
    return implode( array_slice( $lines, $from-1, $len ));
  }
}

print ReflectionUtil::getClassSource(
  new ReflectionClass( 'CdProduct' ) );
```

ReflectionUtil是一个简单的类，其中只有一个静态方法ReflectionUtil::getClassSource()。这个方法只接受ReflectionClass对象作为它的参数，并返回相应类的源代码。ReflectionClass::getFileName()提供到类文件的绝对路径，所以代码可以正确打开它。file()函数获得由文件中所有行组成的数组。ReflectionClass::getStartLine()提供类的起始行，Reflection Class::getEndLine()则查找类的终止行。因此，可以简单地用array_slice()提取出代码行。

为了让事情简单化，这段代码忽略了错误处理。在实际的程序中，应该要检查参数和结果代码。

5.3.4 检查方法

就像ReflectionClass可以用于检查类一样，ReflectionMethod对象可以用于检查类中的方法。

获得ReflectionMethod对象的方法有两种：从ReflectionClass::getMethods()获得ReflectionMethod对象的数组；如果需要使用特定的类方法，ReflectionClass::getMethod()可接受一个方法名作为参数并返回相应的ReflectionMethod对象。

下面我们使用ReflectionClass::getMethods()来获取对象的所有方法并用ReflectionMethod类来逐一检查这些方法的细节信息。

```
$prod_class = new ReflectionClass( 'CdProduct' );
$methods = $prod_class->getMethods();

foreach ( $methods as $method ) {
  print methodData( $method );
  print "\n----\n";
}

function methodData( ReflectionMethod $method ) {
  $details = "";
```

```
    $name = $method->getName();
    if ( $method->isUserDefined() ) {
      $details .= "$name is user defined\n";
    }
    if ( $method->isInternal() ) {
      $details .= "$name is built-in\n";
    }
    if ( $method->isAbstract() ) {
      $details .= "$name is abstract\n";
    }
    if ( $method->isPublic() ) {
      $details .= "$name is public\n";
    }
    if ( $method->isProtected() ) {
      $details .= "$name is protected\n";
    }
    if ( $method->isPrivate() ) {
      $details .= "$name is private\n";
    }
    if ( $method->isStatic() ) {
      $details .= "$name is static\n";
    }
    if ( $method->isFinal() ) {
      $details .= "$name is final\n";
    }
    if ( $method->isConstructor() ) {
      $details .= "$name is the constructor\n";
    }
    if ( $method->returnsReference() ) {
      $details .= "$name returns a reference (as opposed to a value)\n";
    }
    return $details;
  }
```

这段代码使用ReflectionClass::getMethods()得到ReflectionMethod对象数组，然后循环遍历数组，传递每个对象给methodData()。

methodData()中各个方法的名称说明了各方法的设计目的：这段代码检查类方法是否为用户定义的、内置的、抽象的、public、protected、static或final等。也可以检查方法是否是类的构造方法，是否返回引用。

注意，在PHP 5中，如果被检测的方法只返回对象（即使对象是通过引用赋值或传递的），那么ReflectionMethod::returnsReference()不会返回true。只有当被检测的方法已经被明确声明返回引用（在方法名前面有&符号）时，ReflectionMethod::returnsReference()才返回true。

如你所期望的，可以像之前使用ReflectionClass那样获得类方法的源代码：

```
class ReflectionUtil {
  static function getMethodSource( ReflectionMethod $method ) {
    $path = $method->getFileName();
    $lines = @file( $path );
    $from = $method->getStartLine();
```

```
    $to    = $method->getEndLine();
    $len   = $to-$from+1;
    return implode( array_slice( $lines, $from-1, $len ));
  }
}

$class = new ReflectionClass( 'CdProduct' );
$method = $class->getMethod( 'getSummaryLine' );
print ReflectionUtil::getMethodSource( $method );
```

ReflectionMethod提供了getFileName()、getStartLine()和getEndLine()方法，因此我们可以很容易地获取方法的源代码。

5.3.5 检查方法参数

在PHP 5中，声明类方法时可以限制参数中对象的类型，因此检查方法的参数变得非常必要。为此，反射API提供了ReflectionParameter类。要获得ReflectionParameter对象，需要Reflection Method对象的帮助。ReflectionMethod::getParameters()方法可返回ReflectionParameter对象数组。

ReflectionParameter可以告诉你参数的名称，变量是否可以按引用传递（即方法声明前有一个&符号），还可以告诉你参数类型提示和方法是否接受空值作为参数。

下面举例说明如何使用ReflectionParameter：

```
$prod_class = new ReflectionClass( 'CdProduct' );
$method = $prod_class->getMethod( "__construct" );
$params = $method->getParameters();

foreach ( $params as $param ) {
    print argData( $param )."\n";
}

function argData( ReflectionParameter $arg ) {
  $details = "";
  $declaringclass = $arg->getDeclaringClass();
  $name  = $arg->getName();
  $class = $arg->getClass();
  $position = $arg->getPosition();
  $details .= "\$$name has position $position\n";
  if ( ! empty( $class )  ) {
    $classname = $class->getName();
    $details .= "\$$name must be a $classname object\n";
  }

  if ( $arg->isPassedByReference() ) {
    $details .= "\$$name is passed by reference\n";
  }

  if ( $arg->isDefaultValueAvailable()  ) {
    $def = $arg->getDefaultValue();
    $details .= "\$$name has default: $def\n";
  }
```

```
    return $details;
  }
```

以上代码使用ReflectionClass::getMethod()方法得到一个ReflectionMethod对象，然后使用ReflectionMethod::getParameters()获得ReflectionParameter对象数组，接着把ReflectionPara meter对象作为参数传递给argData()函数，argData()函数通过该对象获取参数的相关信息。

在argData()函数中，首先用ReflectionParameter::getName()获得参数的变量名。如果参数有使用对象类型提示的话，ReflectionParameter::getClass()方法返回Reflection-Class对象。接着，用isPassedByReference()检查参数是否为引用。最后检查默认值的可用性，然后将该默认值添加到返回字符串中。

5.3.6 使用反射 API

我们已经学习了反射API的基础知识，下面详细介绍如何使用反射API。

假设我们要创建一个类来动态调用Module对象，即该类可以自由加载第三方插件并集成进已有的系统，而不需要把第三方的代码硬编码进原有的代码。要达到这个目的，可以在Module接口或抽象类中定义一个execute()方法，强制要求所有的子类都必须实现该方法。可以允许用户在外部XML配置文件中列出所有Module类。系统可以使用XML提供的信息来加载一定数目的Module对象，然后对每个Module对象调用execute()。

然而，如果每个Module需要不同的信息来完成任务，应该怎么做呢？在这种情况下，XML文件可以为每个Module提供属性键和值，Module的创建者可以为每个属性名提供setter方法。代码要确保根据某个属性名调用正确的setter方法。

下面是Module接口和几个实现类的基础：

```
class Person {
    public $name;
    function __construct( $name ) {
        $this->name = $name;
    }
}

interface Module {
  function execute();
}

class FtpModule implements Module {
  function setHost( $host ) {
    print "FtpModule::setHost(): $host\n";
  }

  function setUser( $user ) {
    print "FtpModule::setUser(): $user\n";
  }
```

```
    function execute() {
      // 执行一些操作
    }
  }

  class PersonModule implements Module {
    function setPerson( Person $person ) {
      print "PersonModule::setPerson(): {$person->name}\n";
    }

    function execute() {
      // 执行一些操作
    }
  }
```

这里，PersonModule和FtpModule都提供了execute()方法的空实现。每个类也实现了setter方法。这些方法除了报告自己被调用之外不做任何事情。我们的系统规定所有的setter方法必须带有单个参数：要么是一个字符串，要么是可以用字符串参数来实例化的对象。PersonModule::set Person()方法希望有一个Person对象作为参数，所以在例子中设计了Person类。

要使用PersonModule和FtpModule，下一步是创建ModuleRunner类。它使用索引为模块名的多维数组来展示XML文件提供的配置信息。下面是代码：

```
class ModuleRunner {
    private $configData
            = array(
                    "PersonModule" => array( 'person'=>'bob' ),
                    "FtpModule"    => array( 'host'
                                             =>'example.com',
                                            'user'   =>'anon' )
              );
    private $modules = array();
  // ...
}
```

ModuleRunner::$configData属性包括了对两个Module类的引用。对于每个模块元素，代码保存一个包含属性集的子数组。ModuleRunner的init()方法用于创建正确的Module对象，如下所示：

```
class ModuleRunner {
  // ...

     function init() {
        $interface = new ReflectionClass('Module');
        foreach ( $this->configData as $modulename => $params ) {
            $module_class = new ReflectionClass( $modulename  );
            if ( ! $module_class->isSubclassOf( $interface ) ) {
               throw new Exception( "unknown module type: $modulename" );
            }
            $module = $module_class->newInstance();
            foreach ( $module_class->getMethods() as $method ) {
                $this->handleMethod( $module, $method, $params );
```

```
                    // 我们将在下一个代码示例中介绍handleMethod()
                }
                array_push( $this->modules, $module );
            }
        }

    //...
  }

  $test = new ModuleRunner();
  $test->init();
```

init()方法循环遍历ModuleRunner::$configData数组，它尝试为对每个模块元素创建ReflectionClass对象。以不存在的类名调用ReflectionClass的构造方法时，会抛出异常，所以在真实的情况下，我们应该在这里包含更多的错误处理。我们使用ReflectionClass::isSubclassOf()方法来确保模块类属于Module类型。

在调用每个Module的execute()方法前，我们必须先创建Module对象的实例，而这正是Reflec tionClass::newInstance()方法的设计目的。newInstance()方法可接受任意数目的参数，这些参数将被传递到相应类的构造方法。如果一切正常，它返回类的实例（对于要正式发布的产品代码，应确保代码的安全：在实例化对象前，需要确认每个Module对象的构造方法是否需要参数）。

ReflectionClass::getMethods()返回一个包含所有可用ReflectionMethod对象的数组。对于数组中的每个元素，代码调用ModuleRunner::handleMethod()方法，它有3个参数：一个Module实例、ReflectionMethod对象以及一个和Module关联的属性数组。handleMethod()检验并调用Module对象的setter方法。

```
class ModuleRunner {
  // ...
  function handleMethod( Module $module, ReflectionMethod $method, $params ) {
    $name = $method->getName();
    $args = $method->getParameters();

    if ( count( $args ) != 1 ||
       substr( $name, 0, 3 ) != "set" ) {
      return false;
    }

    $property = strtolower( substr( $name, 3 ));
    if ( ! isset( $params[$property] ) ) {
      return false;
    }

    $arg_class = $args[0]->getClass();
    if ( empty( $arg_class ) ) {
      $method->invoke( $module, $params[$property] );
    } else {
      $method->invoke( $module,
          $arg_class->newInstance( $params[$property] ) );
```

```
            }
        }
    }
```

handleMethod()首先检查方法是否为有效的setter。在这段代码中，一个有效的setter必须命名为setXXXX()，并且有且只有一个参数。

如果经检验参数是有效的，则从方法名中提取属性名，即在方法名的开头移除set，并转换结果子串为小写字母。该字符串用于检测$params数组参数。数组包含用户提供的与Module对象相关的属性。如果$params数组不包含某个属性，那么代码将不再继续执行并返回false。

如果从模块方法得到的属性名与$params数组中的某个元素相匹配，则可以继续调用正确的setter方法。要那样做，代码必须检查setter方法第一个（且唯一）参数的数据类型。ReflectionParameter::getClass()方法可提供这个信息。如果方法返回空值，setter希望参数类型为基本数据类型；否则，它期望参数是一个对象。

要调用setter方法，我们需要使用一个新的反射API方法——ReflectionMethod::invoke()。它以一个对象和任意数目的方法作为参数。如果提供的对象与方法不匹配，则ReflectionMethod::invoke()会抛出异常。我们可以通过两种途径调用invoke()方法：如果setter方法不需要对象参数，可以用用户提供的属性字符串来调用ReflectionMethod::invoke()；如果方法需要对象作为参数，可以使用属性字符串来实例化正确类型的对象，然后传递给setter。

上例假设所需的对象能够通过传递单个字符串参数给构造方法来实例化。当然，最好在调用ReflectionClass::newInstance()前检查一下。

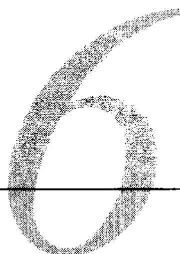
在ModuleRunner::init()方法运行时，ModuleRunner对象存储着许多Module对象，而所有Module对象都包含着数据。ModuleRunner类现在可以用一个类方法来循环遍历每个Module对象，并逐一调用各Module对象中的execute()方法。

5.4 小结

本章介绍了一些可以用于管理库和类的技术及工具。我们介绍了PHP新的命名空间特性，我们可以综合使用包含路径、命名空间、PEAR类命名惯例和文件系统来提供灵活的类组织。我们还学习了PHP的对象和类函数，然后是功能强大的反射API。最后，我们使用Reflection类构建了一个简单的示例，学习了如何使用反射API。

第 6 章

对象与设计



我们已经了解了PHP中面向对象的一些具体知识，本章中我们将跳出这些技术细节，回过头来考虑如何更好地使用PHP中的各种面向对象特性。本章将介绍一些对象和设计相关的主题。还将学习UML，它是一种用于描述面向对象系统的强大的、图形化的语言。

本章包括以下内容。

- 设计基础：设计是什么？面向对象设计和过程式编程有什么不同？
- 类的内容：如何决定一个类中应该包含什么？
- 封装（encapsulation）：在类的接口后面隐藏实现和数据。
- 多态（polymorphism）：使用一个共同的父类，允许在运行时透明地替换特定的子类。
- UML：使用图表来描述面向对象结构。

6.1 代码设计的定义

对“代码设计”的理解涉及系统的定义：确定系统的需求、作用域和目标。系统需要做什么？谁需要使用它？系统输出的内容是什么？系统可以满足一定的需求吗？从底层上看，设计是定义系统组成并组织各组件间关系的过程。本章从另一个角度考虑：类和对象的定义与配置。

那么什么是系统的参与者呢？面向对象的系统由一系列类组成。决定系统中这些类的角色是非常重要的，而类由方法组成，所以在定义类时，必须决定哪些方法应该放在一起。另外，类与类之间常常通过继承关系联系起来以便遵循公用的接口。因此，这些接口或类型应该是设计系统时首先要考虑的。

你还可以为类定义其他关系。你可以创建由其他类型的对象组成的类，也可以创建用于管理多个其他对象实例的类。你还可以创建只是简单地用到其他对象的类。类之间的潜在关系往往在类的结构中就已经确定（例如类方法参数中的类型提示指定了要使用的对象类型），但是对象之间的实际关系只在代码执行时才产生，这些对象间的关系增加了代码的灵活性。在本章中，我们将看到如何为这些关系建立模型。在本书的后续章节中，我们还会深入研究。

在设计过程中，必须决定某个操作何时属于某个类型，何时属于这个类型使用的另一个类。在每个地方你都面临选择，你的决定可能使代码更清晰、更优雅，也可能会让你陷入困境。

在本章中，我们将介绍一些影响你做决定的因素。

6.2 面向对象设计和过程式编程

面向对象设计和传统的过程式编程有什么不同呢？很多人认为最大的不同在于面向对象编程中包含对象。事实上，这种说法并不准确。在PHP中，你经常会发现过程式编程也使用对象，也可能遇到类中包含过程式代码的情况。类的出现并不能说明使用了面向对象设计。甚至对于Java这种强制把一切都包含在类中的语言，使用对象也不能说明使用了面向对象设计。

面向对象和过程式编程的一个核心区别是如何分配职责。过程式编程表现为一系列命令和方法的连续调用。控制代码根据不同的条件执行不同的职责。这种自顶向下的控制方式导致了重复和相互依赖的代码遍布于整个项目。面向对象编程则将职责从客户端代码中移到专门的对象中，尽量减少相互依赖。

为了说明以上几点，我们分别用面向对象和过程式代码的方式来分析一个简单的问题。假设我们要创建一个用于读写配置文件的工具。为了重点关注代码的结构，示例中将忽略具体的功能实现。

我们先按过程式方式来解决这个问题。首先，用下面的格式来读写文本：

```
key:value
```

只需要两个函数：

```
function readParams( $sourceFile ) {
    $prams = array();
    // 从$sourceFile中读取文本参数
    return $prams;
}

function writeParams( $params, $sourceFile ) {
    // 写入文本参数到$sourceFile
}
```

`readParams()`函数的参数为源文件的名称。该函数试图打开文件，读取每一行内容并查找键/值对，然后用键/值对构建一个关联数组。最后，该函数给控制代码返回数组。`writeParams()`以关联数组和指向源文件的路径作为参数，它循环遍历关联数组，将每对键/值写入文件。下面是使用这两个函数的客户端代码：

```
$file = "./param.txt";
$array['key1'] = "val1";
$array['key2'] = "val2";
$array['key3'] = "val3";
writeParams( $array, $file );  // 将数组写到文件
$output = readParams( $file ); // 从文件读取数组
print_r( $output );
```

这段代码较为紧凑并且易于维护。`writeParams()`被调用来创建Param.txt并向其写入如下的内容：

```
key1:val1
key2:val2

key3:val3
```

现在，我们被告知这个工具需要支持如下所示的XML格式：

```
<params>
    <param>
        <key>my key</key>
        <val>my val</val>
    </pram>
</params>
```

如果参数文件以.xml结尾，就应该以XML模式读取参数文件。虽然这并不难调节，但可能会使我们的代码更难维护。这时我们有两个选择：可以在控制代码中检查文件扩展名，或者在读写函数中检测。我们使用后面那种方法：

```
function readParams( $source ) {
    $params = array();
    if ( preg_match( "/\.xml$/i", $source )) {
        // 从$source中读取XML参数
    } else {
        // 从$source中读取文本参数
    }
    return $params;
}

function writeParams( $params, $source ) {
    if ( preg_match( "/\.xml$/i", $source )) {
       // 写入XML参数到$source
    } else {
        // 写入文本参数到$source
    }
}
```

注解 示例代码总是在寻求平衡。它需要足够清晰以抓住要点，所以通常省略了错误检查。也就是说，这里的代码主要是说明设计和代码重复的问题，并不是提供解析和写入文件数据的最佳途径。因此，代码中忽略了与设计不相关的内容。

如上所示，我们在两个函数中都要检测XML扩展名，这样的重复性代码会产生问题。如果我们还被要求支持其他格式的参数，就要始终保持readParams()和writeParams()函数的一致性。

下面我们用类来处理相同的问题。首先，创建一个抽象的基类来定义类型接口：

```
abstract class ParamHandler {
    protected $source;
    protected $params = array();

    function __construct( $source ) {
        $this->source = $source;
    }

    function addParam( $key, $val ) {
        $this->prams[$key] = $val;
```

```
    }

    function getAllParams() {
        return $this->params;
    }

    static function getInstance( $filename ) {
    if ( preg_match( "/\.xml$/i", $filename )) {
            return new XmlParamHandler( $filename );
        }
        return new TextParamHandler( $filename );
    }

    abstract function write();
    abstract function read();
}
```

我们定义addParam()方法来允许用户增加参数到protected属性$params，getAllParams()则用于访问该属性，获得$params的值。

我们还创建了静态的getInstance()方法来检测文件扩展名，并根据文件扩展名返回特定的子类。最重要的是，我们定义了两个抽象方法read()和write()，确保ParamHandler类的任何子类都支持这个接口。

注解 把一个用于生成子对象的静态方法放在父类中是很方便的，然而这样的设计也有不足之处。ParamHandler类型现在只能与条件语句中规定的类一起工作。如果需要处理其他格式的文件，怎么办呢？当然，如果你是ParamHandler的维护者，可以修改getInstance()方法。但是如果你只是这段代码的使用者，修改这个类就不是那么容易了（实际上，修改它并不难，但是每次重新安装这个包的时候都需要再次修改）。我们将会在第9章中讨论对象创建的问题。

现在，我们定义多个子类。为了保持示例简洁，再次忽略实现细节：

```
    function write() {
        // 写入XML文件
        // 使用$this->params
    }

    function read() {
        // 读取XML文件内容
        // 并赋值给$this->params
    }

}

class TextParamHandler extends ParamHandler {

    function write() {
        // 写入文本文件
```

```
            // 使用$this->params
        }

        function read() {
            // 读取文本文件内容
            // 并赋值给$this->params
        }
    }
```

这些类简单地提供了write()和read()方法的实现。每个类都将根据适当的文件格式进行读写。

客户端代码将完全自动地根据文件扩展名来写入数据到文本和XML格式的文件：

```
$test = ParamHandler::getInstance( "./params.xml" );
$test->addParam("key1", "val1" );
$test->addParam("key2", "val2" );
$test->addParam("key3", "val3" );
$test->write(); // 写入XML格式中
```

我们还可以从两种文件格式中读取：

```
$test = ParamHandler::getInstance( "./params.txt" );
$test->read(); // 从文本格式中读取
```

那么，我们可以从这两种解决方案中学习到什么呢？

6.2.1 职责

在过程式编程的例子中，控制代码的职责是判断文件格式，它判断了两次而不是一次。条件语句被绑定到函数中，但这仅是将判断的流程隐藏起来。对readParams()的调用和对writeParams()的调用必须发生在不同的地方，因此我们不得不在每个函数中重复检测文件扩展名（或执行其他检测操作）。

在面向对象代码中，我们在静态方法getInstance()中进行文件格式的选择，并且仅在getInstance()中检测文件扩展名一次，就可以决定使用哪一个合适的子类。客户端代码并不负责实现读写功能。它不需要知道自己属于哪个子类就可以使用给定的对象。它只需要知道自己正在使用ParamHandler对象，并且ParamHandler对象支持write()和read()方法。过程式代码忙于处理细节，而面向对象代码只需一个接口即可工作，并且不用考虑实现的细节。由于实现由对象负责，而不是由客户端代码负责，所以我们能够很方便地增加对新格式的支持。

6.2.2 内聚

内聚（cohesion）是一个模块内部各成分之间相关联程度的度量。理想情况下，你应该使各组件职责清晰、分工明确。如果代码间的关联范围太广，维护就会很困难——因为你需要在修改某部分代码的同时修改相关的代码。

前面的ParamHandler类将相关的处理过程集中起来。用于处理XML的类方法间可以共享数据，并且一个类方法中的改变可以很容易地反映到另一个方法中（比如改变XML元素名）。因此，

我们可以说ParamHandler类是高度内聚的。

另一方面，过程式的例子则把相关的过程分离开，导致处理XML的代码在多个函数中同时出现。

6.2.3 耦合

当系统各部分代码紧密绑在一起时，就会产生紧密耦合，这时在一个组件中的变化会迫使其他部件随之改变。紧密耦合并不是过程式代码特有的，但是过程式代码比较容易产生耦合问题。

我们可以在过程式示例中看到耦合的产生。在writeParams()和readParams()函数中，使用了相同的文件扩展名测试来决定如何处理数据。因此，我们要改写一个函数，就不得不同时改写另一个函数。例如，如果要增加一种新的文件格式，就要在两个函数中按相同的方式都加上相应的扩展名检查代码，这样两个函数才能保持一致。当我们要新增参数相关的函数时，问题只会更复杂[①]。

面向对象的示例中则将每个子类彼此分开，也将其与客户端代码分开。如果需要增加新的参数格式，只需简单地创建相应的子类，并在父类的静态方法getInstance()中增加一行文件检测代码即可。

6.2.4 正交

正交（orthogonality）指将职责相关的组件紧紧组合在一起，而与外部系统环境隔开，保持独立[②]。Andrew Hunt和David Thomas在*The Pragmatic Programmer*一书（Addison-Wesley, 1999）中有所介绍。

正交主张重用组件，期望不需要任何特殊配置就能把一个组件插入到新系统中。这样的组件有明确的与环境无关的输入和输出。正交代码使修改变得更简单，因为修改一个实现只会影响到被改动的组件本身。最后，正交代码更加安全。bug的影响只局限于它的作用域之中。内部高度相互依赖的代码发生错误时，很容易在系统中引起连锁反应。

如果只有一个类，松散耦合和高聚合是无从谈起的。毕竟，我们可以把整个过程式示例的全部代码塞到一个被误导的类里。因此，我们如何才能在代码中达到一个平衡呢？通常，首先考虑哪些类应该存在于系统之中。

6.3 选择类

定义类的界限往往会比我们想象的更加困难，当系统不断发展时尤其如此。

模拟真实世界是比较直接的办法。面向对象系统经常反映真实的事物——Person、Invoice和Shop类等，即定义一个类就是找到系统中的事物，然后通过方法给予它们动作。这个想法不错，但也有不足之处。如果把类看做名词——一系列动作的执行者，那么随着开发和需求的改变，你

① 例如，增加一个处理参数的函数processParams()，则该函数中也要添加判断文件扩展名的代码，这样代码维护就变得非常困难。——译者注

② 在理想的正交情况下，任何操作都没有副作用。每个组件发生的改变都只影响到其自身。——译者注

会发现它因为需要而做越来越多的事情。

我们回顾一下在第3章中创建的ShopProduct。系统为了给顾客提供产品，因此要定义一个ShopProduct类，但是这是唯一的办法吗？为了访问产品数据，我们需要提供诸如getTitle()和getPrice()等方法。当我们需要输出发票和提货单的摘要信息时，需要定义write()方法。当顾客要求得到不同格式的产品摘要信息时，除了创建write()方法之外，还要创建writeXML()和writeXHTML()方法。否则，需要增加条件代码到write()中并根据选项标志输出不同格式。

无论哪一种方式，ShopProduct类现在做了太多的事情。它忙着管理显示方案及产品数据。

我们应该如何定义类呢？最好的办法是让一个类只有一个主要的职责，并且任务要尽可能独立。你可以把类的职责用多个词来形容，最好不超过25个词，不要用到词“且”或者“或”。如果句子太长或者有复杂的子句，就应该考虑用你所描述的一部分任务来定义新类。

因此，ShopProduct类应该主要负责处理产品数据。如果我们显示不同格式的摘要信息，则有了一个新的职责：产品显示。正如第3章所述，我们实际上基于这两个不同的职责定义了两个类型：ShopProduct类型负责处理产品数据；ShopProductWriter类型负责显示产品信息。每个类只承担自己的任务。

注解　设计原则并非一成不变。比如，有时你会看到在另一个不相关的类中出现了保存对象数据的代码。这似乎违反了类应该有单一职责的原则，但事实上，这样做可以为类方法完全访问对象实例的属性提供最大的便利。使用本地方法进行对象持久化也可让我们不用创建与被保存的类相对应的持久化类，避免产生依赖。我们会在第12章中介绍更多对象持久化的策略。记住要避免对于设计原则的宗教性崇拜。设计原则并不能代替你来分析问题。要掌握设计原则背后的内涵，这比设计原则本身更为重要。

6.4　多态

多态或称“类切换”是面向对象系统的基本特性之一。在本书中，我们已经多次遇到过多态。

多态是指在一个公用接口后面维护多个实现。这听起来比较复杂，但实际上我们已经很熟悉它了。如果代码中存在大量条件语句，就说明需要使用多态。

在第3章中，我们第一次创建ShopProduct类时，只创建了一个类。该类除了管理一般的产品之外，还有管理图书和CD的功能。我们使用条件语句来提供相应的产品摘要信息：

```
function getSummaryLine() {
    $base  = "$this->title ( $this->producerMainName, ";
    $base .= "$this->producerFirstName )";
    if ( $this->type == 'book' ) {
        $base .= ": page count - $this->numPages";
    } else if ( $this->type == 'cd' ) {
        $base .= ": playing time - $this->playLength";
    }
    return $base;
}
```

这些语句暗示我们存在两个子类的原型：CdProduct和BookProduct。

同样，在上文面向过程的例子中我们也使用了条件语句，这些语句中暗含了我们最终要实现的面向对象结构的萌芽。我们在代码的两个部分重复使用相同的条件语句。

```
function readParams( $source ) {
    $params = array();
    if ( preg_match( "/\.xml$/i", $source )) {
        // 从$source读XML参数
    } else {
        // 从$source读文本参数
    }
    return $params;
}

function writeParams( $params, $source ) {
    if ( preg_match( "/\.xml$/i", $source )) {
        // 写XML参数到$source
    } else {
        // 写文本参数到$source
    }
}
```

每个子句都暗示着我们最终生成的一个子类：XmlParamHandler或TextParamHandler，它们继承了抽象基类ParamHandler的write()和read()方法。

```
// 返回XmlParamHandler或TextParamHandler
$test = ParamHandler::getInstance( $file );


$test->read();  // 可能是XmlParamHandler::read()或TextParamHandler::read()
$test->addParam("key1", "val1" );
$test->write(); // 可能是XmlParamHandler:: write()或TextParamHandler:: write()
```

要特别注意的是多态并没有消除条件语句。像ParamHandler::getInstance()这样的方法经常通过switch或if语句决定要返回的对象，但多态可以把条件代码集中到一个地方。

就像我们看到的那样，PHP强制接口由抽象类定义。这非常有用，因为我们可以确定子类将会实现抽象父类中定义的所有方法，包括类类型提示和方法的访问控制。客户端代码因此可以使用一个公共父类的任意子类而不需要改写代码（只要客户端代码仅依赖于父类中定义的功能）。这个规则唯一的缺憾是：无法强制规定类方法返回的数据类型。

注解 *在写作本书的时候，类方法返回类型提示有计划增加到未来版本的PHP中，但结果尚未可知。*

PHP 不能强制限定类方法返回的数据类型意味着不同子类的方法可能返回不同类型的对象或基本数据类型，这会损害类型的互换性[1]。你应该尝试使返回值保持一致。类中一些方法可能

① 客户端代码不能随意切换使用不同子类的对象。——译者注

利用PHP的松散类型特征根据环境返回不同的数据类型，而另外一些类方法则和客户端代码之间有一定的“约定”，约定返回某个特定的数据类型。如果订下约定的是一个抽象父类，则子类会延续该约定，返回同样类型的数据。如果确定返回一个特定类型的对象，当然也可以返回子类型的实例。虽然PHP编译器没有强制返回类型，你可以在项目中靠人为的约定来使多个方法保持一致。你可以在源代码中使用注释来标明方法的返回值类型。

6.5 封装

简单地说，封装就是对客户端代码隐藏数据和功能。封装也是面向对象的重要概念之一。

要实现封装，最简单的办法是将属性定义为private或protected。通过对客户端代码隐藏属性，我们创建了一个接口并防止在偶然情况下污染对象中的数据。

多态是另外一种封装。通过把不同的实现放在公共接口之后，我们对客户端代码隐藏了功能的实现。也就是说，任何在接口背后发生的改变对外界的系统来说都是可忽略的。我们可以增加新类或改变类中的代码，而不会产生错误。接口与其背后的工作机制是分开来的。这些机制越独立，改进或修正代码对系统的影响越小。

从某种程度上说，封装是面向对象编程的关键。我们的目标是使系统中的每一部分都尽可能独立。类和方法应当能够接收到足够的信息来执行它们承担的任务，而这些任务应该非常清晰，有明确的定义和范围。

在PHP 5中，private、protected和public关键字的引入使封装变得容易。尽管如此，在使用封装时还是要深思熟虑。但PHP 4并没有正式支持隐藏数据，你不得不使用文档和命名惯例来说明数据的隐私性。例如，以下划线开头的属性通常是私有属性：

```
var $_touchezpas;
```

当然，你要认真检查代码，因为属性的隐私并没有真正受到严格的限制。尽管如此，这样写代码几乎不会出错，因为代码的结构和风格看起来清晰漂亮，你很容易就可以确定哪些属性是私有的。

出于同样的原因，甚至在PHP 5中，我们也可以在需要进行类选择时打破常规，使用instanceof操作符来找出所用对象属于哪一个具体的子类。

```
function workWithProducts( ShopProduct $prod ) {

    if ( $prod instanceof cdproduct ) {

        // CD 产品特有的操作

    } else if ( $prod instanceof bookproduct ) {

        // 图书产品特有的操作

    }

}
```

这样做通常有很好的理由，但通常来说会带来一些不确定的因素。在代码中查询特定子类型时会产生一个依赖关系。如果我们利用多态来隐藏特定子类型的特性，ShopProduct的继承关系改变时就不会产生任何错误。但上面的代码会破坏这种情况。使用上面的代码时，如果我们改写了CdProduct和BookProduct类，就有可能在workWithProducts()方法中产生预想不到的错误。

这个例子说明了两件事。首先，封装可以帮助我们创建正交的[①]代码。其次，封装的范围不怎么重要，无论封装的规模是大是小，类和客户端代码都必须同时关注封装的实现。

6.6 忘记细节

如果你和我一样，可能会一遇到问题就开始努力寻找解决办法：设计函数来解决一个问题，或构造一个正则表达式，或者下载PEAR包，或者可能会从旧的项目中寻找解决类似问题的代码来使用。在设计阶段，让大脑空白一段时间会给你带来意想不到的好处。请清空脑海中这些和细节相关的念头。

你可以只考虑系统中的关键参与者：项目需要的对象类型和这些对象的接口。当然，你的经验会告诉你如何思考：一个打开文件的类需要一个路径，数据库代码需要管理表名和密码等。但最好不要被经验所左右，而要让代码中的结构和关系来引导你，你会发现在一个定义良好的接口之后加入实现代码是很容易的。接着你可以灵活地选择、改进或扩展一个可能需要的实现，而不会影响到外界的系统。

为了强调接口，我们按抽象基类而不是具体的子类来思考。例如，在之前的参数读取示例代码中，接口是设计中最重要的部分。我们需要一个可以读写键/值对的对象类型。对于该对象类型来说，读写键/值对就是最重要的职责，而不是持久存储或获取数据。我们围绕着抽象类ParamHandler来设计系统，并只为实际读写需要增加具体的处理方案。用这种方法，我们一开始就在系统中使用了多态和封装。这个结构也具有类切换的能力[②]。

我们一开始就已经知道将会有文本和XML两种类型的ParamHandler实现，这毫无疑问会影响我们对于接口的设计。这些具体的实现有可能会干扰到我们设计出抽象的接口，因为我们思考的东西增加了。设计接口的时候我们或多或少总得花一番心思。[③]

"四人组"在《设计模式》一书中用一句话总结了这个规则——"为接口而不是实现而编程"（Program to an interface, not an implementation）。这是一句值得摘录到你的笔记本中的话。

6.7 4个方向标

没有人能在设计的时候绝对正确。大部分人都要不停地修改代码，因为需求可能发生变化，或者我们加深了对问题的认识。

---

① 指非常独立的，依赖性很小的。——译者注

② 客户端代码在使用对象时可以很方便地切换到另一个子类的对象而不需要修改代码。——译者注

③ 设计软件的时候，克服复杂度是一个非常重要的原则。我们应尽量减少同一时间在脑中思考的对象，要能分清主次，把问题抽象出来。所谓的"心智把戏"，指的是权衡哪些东西是真正需要放到系统中的过程。*Code Complete*一书中有涉及相关话题。——译者注

修改代码时，很容易失去控制。在这里加一个方法，在那里加一个新类，我们的系统开始逐渐变得混乱。事实上，代码本身就可以指出自己的改进方向。代码中的某些关键点就像坏味代码一样，告诉我们代码可能需要修改或者至少需要检查一下设计是否需要重新设计。本节从中挑选出4个说明代码需要检查的“路标”[①]。

6.7.1　代码重复

重复（duplication）是代码中最常见的问题之一。如果你在写代码的时候，总是感觉似曾相识，很可能你的代码已经重复了。

请认真查看系统中代码重复的地方，很可能它们本应该放在一起。重复通常意味着紧密耦合。如果你改变了一个地方的代码，是否要同时修改另一个地方的代码呢？如果需要修改，那么这两处代码很可能本应该放在同一个地方。

6.7.2　类知道的太多

从一个方法传递参数到另一个方法时可能会产生问题。为什么不使用全局变量减少麻烦呢？使用全局变量可以让所有的方法都能获得数据。

全局变量有它们自己的作用，但使用时一定要慎重考虑。使用全局变量或者允许一个类知道它之外的领域的内容，你就可以把这个类绑定到外部环境中，让它很难重用，并无法保持独立。记住，我们要解开类及例程之间的耦合，尽量避免产生相互的依赖关系。我们要尽量把一个类限制在自己的环境中，本书后面的章节将会介绍一些相关的策略。

6.7.3　万能的类

你的类是否尝试一次完成很多工作？如果是的话，就要检查类的职责列表了。你会发现其中的一些功能可以提取出来，成为一个基类。

如果类的职责过多，那么在创建子类的时候会产生问题。你要用子类扩展什么样的功能？如果想要让子类负责更多的事情，该怎么办？这样可能会产生太多的子类或者过度依赖于条件语句。

6.7.4　条件语句

在项目中我们常有非常好的理由来使用`if`和`switch`语句，但有时这样的结构会让我们不得不使用多态。

如果你发现在一个类中频繁地进行特定条件的判断，特别是当你发现这些条件判断在多个方法中出现时，就说明这个类需要拆分成两个或者更多。检查一下条件代码的结构，看看是否应该将某些功能独立出来放在独立类中。拆分出来的几个类应该有一个共享的抽象基类，这时你需要知道如何传递正确的子类给客户端代码。本书将在第9章中介绍一些创建对象的模式。

6.8　UML

本书到目前为止，都是让代码来描述代码本身，例如书中用了很多简短的代码示例来说明继

① signpost，这里指代码中标志性的不良现象。出现这几个现象，就说明代码需要重构。——译者注

承和多态等概念。

因为PHP对于我们来说是通用的语言，所以这非常有用。但随着代码示例的增长以及复杂性的增加，仅使用代码来说明设计变得有些荒诞，很难从几行代码上看出整体情况。

UML是Unified Modeling Language（统一建模语言）的缩写，注意说这个词的时候要使用定冠词the。

这么正式的叫法（即the UML而不是UML）可能是由于UML的起源。根据Martin Fowler在*UML Distilled*（Addison Wesley, 1999）一书中的说法，UML成为一个标准之前，面向对象设计社区已经争论了很多年。

争论的结果是引入了一个强大的图形化的语法来描述面向对象系统。在本节中，我们只是蜻蜓点水地掠过，但在本书后面的章节中我们将不断使用UML。

类图可以清晰地描述结构和模式。通过类图我们可以清楚地看出结构和模式的意图，而如果只阅读代码片段和项目符号列表，则很难做到这一点。

6.8.1 类图

类图（class diagram）只是UML的一部分，但它们可能是最常用的。由于类图在描述面向对象关系时非常有用，所以本书中用到的UML一般都是类图。

1. 描述类

类是类图的主要部分。如图6-1所示，类用带有类名的方框来描述。

这个类被分为3部分，最先显示的是类的名称。下面两部分是可选的，用于显示类名之外的信息。在设计类图时，我们会发现图6-1这样的详细程度已经足够描述一些类。我们并非总要在类图中显示类的每个属性和方法，甚至不需要显示每个类。

通常用斜体的类名（如图6-2所示），或者增加`{abstract}`到类名下（如图6-3所示）来表示该类是抽象类。第一种方法比第二种方法常用，但是当你做笔记时第二种方法更有用。

图6-1 类

图6-2 抽象类

图6-3 用约束定义的抽象类

注解 `{abstract}`语法是约束的一个例子。“约束”在类图中用于描述特定元素将被使用。花括号中的文字并没有特殊的结构，它只是提供一个将应用于该元素的条件声明。

接口的定义方式和类相同，但接口必须使用一个衍型[①]（UML的一个扩展），如图6-4所示。

2. 属性

一般说来，属性（attribute）用于描述一个类的属性（property）。属性直接列在类名下面的

① 衍型是UML提供的一种扩展方式，UML是通用的语言，但是针对不同的应用领域和实际情况，用户可以通过自定义和扩展得到更有针对性的建模语言。——译者注

格子中，如图6-5所示。

图6-4　接口

图6-5　属性

下面举例看一下属性。属性前面的符号表示该属性可见性（visibility）的级别或者是访问控制。表6-1展示了可用的3种符号。

表6-1　可见性符号

符　号	可 见 性	说　明
+	public	所有代码都可访问
-	private	只有当前类可以访问
#	protected	只有当前类和它的子类可以访问

可见性符号之后是属性名。这里我们描述了`ShopProduct::$price`属性。冒号用于分隔属性名和它的类型及默认值（默认值为可选项，可以不提供）。

再次提醒，你只要提供必要的信息，不需要写上所有细节。

3. 操作

操作（operation）用于描述类方法，更准确地说，用于描述可以在类的实例上调用的操作。图6-6显示了`ShopProduct`类中的两个操作。

如上所示，操作和属性使用了相似的语法，可见性符号放在方法名之前。参数列表包含在括号之中。方法如果有返回类型的话，用冒号来描述。参数用逗号来分隔，并且遵守属性语法，参数名和它的数据类型间用冒号分隔。

如你所期望的那样，这个语法比较灵活。可以省略可见性标志和返回类型。我们通常只提供参数的类型，而参数名则不怎么重要。

4. 描述继承和实现

UML一般用“泛化”（generalization，也译为一般化）来描述继承关系。这个关系用从子类到父类的一条线来标识，线的顶端有一个空心闭合箭头。

图6-7显示了`ShopProduct`类和它的子类之间的关系。

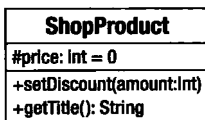


图6-6　操作

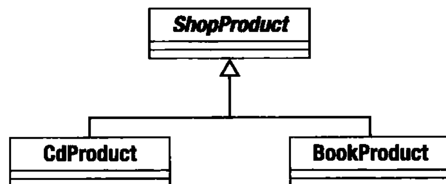


图6-7　描述继承

UML用“实现”来描述接口和实现接口的类之间的关系。因此如果`ShopProduct`类实现了

Chargeable接口，就可以把它加入到类图中，如图6-8所示。

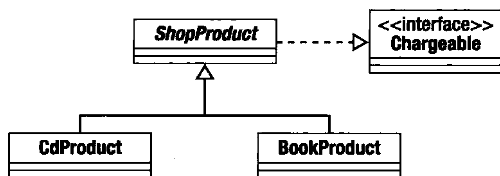


图6-8 描述接口实现

5. 关联

继承只是面向对象系统中诸多关系中的一种。当一个类的属性保存了对另一个类的一个实例（或多个实例）的引用时，就产生了关联。

在图6-9中，我们为两个类建立模型，并创建类之间的关联。

现在类之间的关系仍然有些模糊。我们只是指出Teacher对象拥有一个或多个对Pupil对象的引用，或者Pupil对象拥有一个或多个对Teacher对象的引用。这个关系很不确定。

事实上，我们可以用箭头来描述关联的方向。如果Teacher类拥有Pupil类的一个实例，但是Pupil类没有Teacher类的实例，那么我们可以让关联箭头从Teacher指向Pupil类。这种关联称为“单向关联”，如图6-10所示。

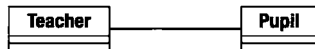


图6-9 类关联

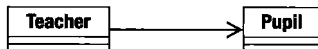


图6-10 单向关联

如果两个类间互相拥有对方的引用，可以用一个双向箭头来描述这种“双向关联”关系，如图6-11所示。

我们也可以在关联中指定一个类的实例被其他类引用的次数，这可以通过把次数或范围放在每个类旁边来说明。用星号（*）表示任意次数。在图6-12中，有一个Teacher对象和零个到多个Pupil对象。

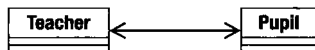


图6-11 双向关联

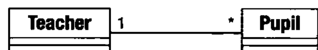


图6-12 关联中包含多个引用

在图6-13中，一个Teacher对象拥有5到10个对Pupil对象的引用。

6. 聚合和组合

与关联很相似，聚合（aggregation）和组合（composition）都描述了一个类长期持有其他类的一个或多个实例的情况。通过聚合和组合，被引用的对象实例成为引用对象的一部分。

在聚合的情况下，被包含对象是容器的一个核心部分，但是它们也可以同时被其他对象所包含。聚合关系用一条以空心菱形开头的线来说明。

在图6-14中，我们定义了两个类：SchoolClass和Pupil。SchoolClass类聚合了Pupil。

学生组成一个班级，但是相同的Pupil对象可以同时被不同的SchoolClass实例引用。如果我们要删除一个学校类，不需要同时删除它引用的这些学生类，因为学生还可以加入其他班级。

组合则是一个更强的关系。在组合中，被包含对象只能被它的容器所引用。当容器被删除时，它也应该被删除。组合关系可以用类似聚合关系的方式描述，但是菱形必须是实心的。图6-15说明了一个组合关系。

Person类持有对SocialSecurityData对象的引用，而该SocialSecurityData对象实例只

属于包含它的Person对象。

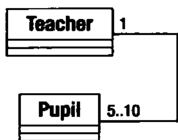


图6-13　关联中包含多个引用

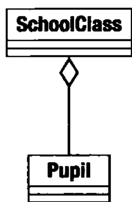


图6-14　聚合

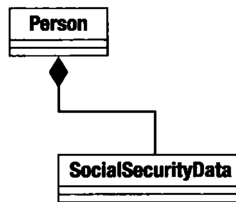


图6-15　组合

7. 描述使用

一个对象使用另一个对象的关系在UML中被描述为一个依赖关系。它是本节中最短暂的一种关系，因为它并非描述类之间的长久关系。

一个被使用的类可以作为类方法的参数传递或者作为方法调用的结果得到。

图6-16中的Report类使用了ShopProductWriter对象。这种使用关系由一条连接两个类的虚线和开放箭头表示。然而，Report类并没有把ShopProductWriter保存为类中的属性，而ShopProductWriter对象则将一组ShopProduct对象作为属性。

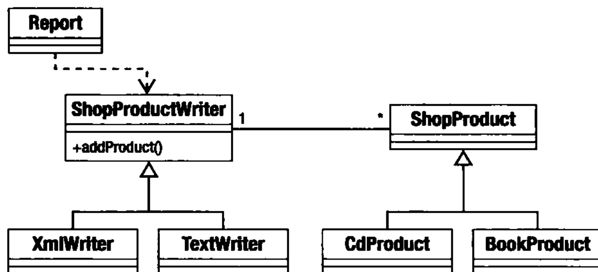


图6-16　依赖关系

8. 使用注解

类图可以捕捉到系统的结构，但类图并不能解释类处理任务的过程。图6-16展示了系统中的类。我们知道Report对象使用了ShopProductWriter，但是我们并不知道具体是如何使用的。在图6-17中，我们使用了注解来补充说明。

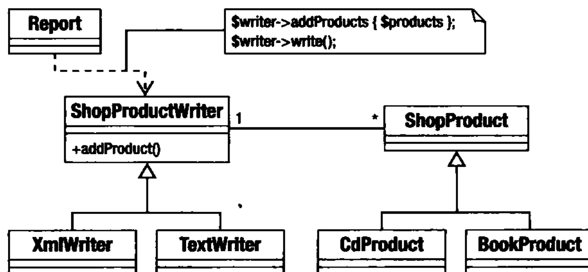


图6-17　使用注解来说明依赖关系

如图6-17所示，注解由一个折角的方框组成。它通常包含伪代码片段。

注解使类图变得易于理解。现在我们可以看到`Report`对象使用了`ShopProductWriter`来输出产品数据。但对象间的“使用”关系并非总是如此明显，甚至有时候，使用注解仍不能提供足够的信息来帮助理解。幸运的是，我们可以在系统中为对象的交互关系和类的结构建模。

6.8.2 时序图

时序图（sequence diagram）是基于对象而不是基于类的，它用于为系统中过程化的行为建模。

我们创建一个简单的时序图，为`Report`对象输出产品数据的过程建模。时序图从左到右地展现了系统中的参与者，如图6-18所示。

图6-18　时序图中的对象

在图6-18中，我们只使用类名来标记对象。如果在图中有同一个类的多个对象实例在独立工作，可以使用`label:class`（例如`product1:ShopProduct`）格式来包含对象名。

在图6-19中，我们从上到下展示了该过程中每个对象的生命周期（lifetime）。

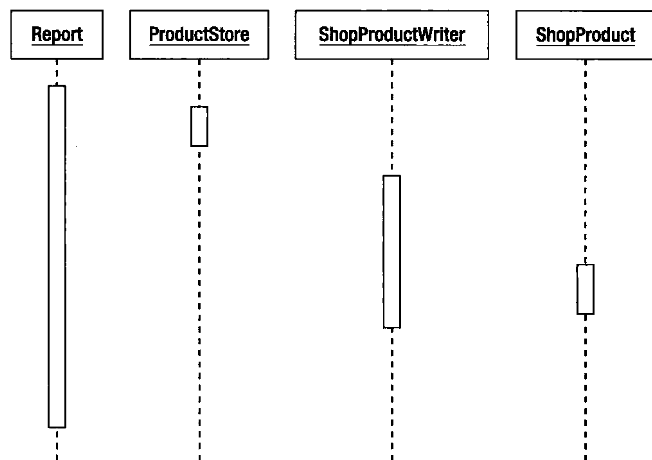


图6-19　时序图中对象的生命周期

垂直的虚线是生命线，展示了系统中对象的生命周期。生命线上的矩形说明了过程中的焦点，即某个对象的激活期。如果从上到下地看图6-19，就会发现系统中的执行过程是如何在对象间移动的。但如果不显示对象间传递的消息，这个图就很难被读懂，因此我们在图6-20中增加了消息传递的内容。

箭头表示消息从一个对象传递到另一个对象。返回值一般不写（你也可以用一条从被调用对象指向消息发送源对象的虚线来说明）。每个消息都用相关的方法调用来标记。你可以非常灵活地使用自己的标记方法，但UML在这里提供了一些语法，例如方括号说明一个条件，所以

```
[okToPrint]
write()
```

意味着只有在适当的条件下，write()调用才会运行。星号用于表示一个重复的操作。你还可以在方括号中进一步说明：

```
*[for each ShopProduct]
write()
```

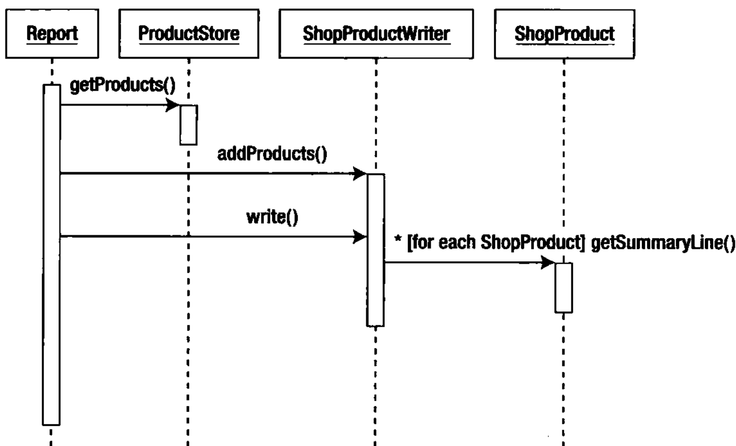


图6-20 完整的时序图

现在从上到下解释图6-20。首先，Report对象获得一个来自ProductStore对象的ShopProduct对象列表。Report对象传递这个ShopProduct列表给一个ShopProductWriter对象，而ShopProductWriter存放了对ShopProduct对象的引用（虽然我们只能从图中推断出这点）。ShopProductWriter对象为它引用的每个ShopProduct对象调用ShopProduct::getSummaryLine()，并添加执行结果到最终的输出结果中。

可以知道，时序图可以为过程建模，把动态的交互操作定格并清晰地展现出来。

注解 看图6-16和图6-20。在类图中，通过展示从ShopProductWriter和ShopProduct继承的类来说明多态。而现在当我们利用时序图为对象的交互建模时，这些细节变得透明。如果可能的话，我们想让对象用最通用的类型来工作，这样就可以隐藏实现的细节。

6.9 小结

在本章中，我们不仅学习了一些关键的面向对象设计理念，还学习了封装、松耦合和内聚等概念，它们是灵活的可重用的面向对象系统的重要概念。另外还学习了UML，它在本书后面的模式学习中将是必不可少的。

Part 3 第三部分

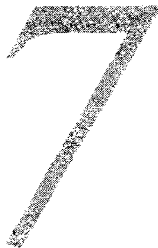
模　　式

本部分内容

- 第 7 章　什么是设计模式？为何使用它们
- 第 8 章　模式原则
- 第 9 章　生成对象
- 第 10 章　让面向对象编程更加灵活的模式
- 第 11 章　执行及描述任务
- 第 12 章　企业模式
- 第 13 章　数据库模式

第7章

什么是设计模式？为何使用它们



作为程序员，我们所遇到的大部分问题其实都已被其他程序员一再地处理了。设计模式意味着智慧。一个模式一旦成为通用模式，就能丰富我们的语言，使我们可轻松地分享设计思想及这些思想所带来的成果。设计模式提取了共同问题，定义了经过测试的解决方案并描述了可能的结果。许多书和文章都关注于编程语言、函数、类和方法的细节，而模式目录则是关注于你如何才能从这些语言基础（what）中继续深入，继而理解项目中出现的问题及找出相应的潜在解决方案（why和how）。

本章将介绍设计模式及它们流行的原因。

本章包括以下内容。

- 模式基础：什么是设计模式？
- 模式结构：每个设计模式的关键元素。
- 模式收益：为何模式值得你花时间学习？

7.1 什么是设计模式

> 在软件世界中，每个开发机构就像一个部落，而模式就是对部落的某种共同记忆的一种有形表现。
>
> ——Grady Booch，《J2EE核心模式》

> 模式便是特定环境下同类问题的一种解决方案。
>
> ——四人组（Gang of Four），《设计模式：可复用面向对象软件的基础》

正如上述引语所暗示的，设计模式便是分析过的问题和问题解决方案所阐释的优秀实践。

同样的问题总是不断出现，而我们作为Web程序员必须一次又一次地解决它们。比如如何处理一个请求？如何将请求数据转换成系统对应的指令？应该如何获得数据？又如何显示结果？随着时间流逝和经验积累，我们会或优雅或困难地回答这些问题，并总结出一些非正式的、可在

项目中重复使用的解决方案，而这些解决方案便是设计模式。

设计模式记录并规范化了这些问题及解决方案，使更广泛的开发社区可获得这些来之不易的经验。模式在本质上讲是（或者说应该是）自下而上而非自上而下的。它们来源于实践而不是空洞的理论。这并非指设计模式没有强有力的理论基础（你会在下一章中看到这些理论基础），但是模式是以真正的程序员使用的真实世界的技术为基础的的。著名的模式专家Martin Fowler便说他是在发现模式，而不是发明模式。正是因为这样的原因，许多模式都会给你造成似曾相识（a sense of déjà）的感觉。

一个模式目录并不是一本食谱。食谱中的配方可以直接遵循；具体的程序代码也可以只做很小改动便被复制和安置在一个项目中。你也不必理解使用的所有代码。设计模式只负责记录特定问题的**解决方法**，对于大范围的上下文环境，具体的实现细节可能差别甚大，而这些上下文环境包括所使用的编程语言、应用程序的性质、项目的大小及问题的细节等。

比方说，如果项目要求创建一个模板系统。根据所给的模板文件名，你必须解析它并建立一个对象树来代表所遇到的标签。

首先我们需要一个可以扫描文本的默认解析器，用于触发标记。当解析器找到一个匹配的标签时，它把找到的标签传递给另一个专门读内部标签的解析器。解析器继续解析模板数据，直到解析失败或者完成，或者找到另一个触发标记。如果找到了一个触发标记，也必须传递它给另一个专门的处理器——可能是一个参数解析器。这些部件共同组成一个递归下降解析器。

因此我们将需要这些解析器：`MainParser`（核心解析器）、`TagParser`（标签解析器）和`ArgumentParser`（参数解析器）。我们可以创建`ParserFactory`类来创建并返回这些对象。

当然，现实中并没有这么简单的事情。你总是会在事后才被告知必须在模板里面支持多种语法。现在，只能根据语法来创建一组相应的解析器：`OtherTagParser`和`OtherArgumentParser`等。

这时你的问题便是：需要根据不同的情况创建一组不同的对象，同时要这组对象在系统里相对于其他组件来说或多或少是透明的。这恰恰就是《设计模式》一书的摘要中为抽象工厂（Abstract Factory）模式所定义的问题：为一个相关或相依赖的对象家族提供统一的创建接口，并无需指定实体类。

这和我们的问题十分符合。正是问题的本质决定和形成了我们对该模式的使用。没有任何对解决方案的剪切和粘贴，而在第9章中，你可以看到对于抽象工厂模式的详细介绍。

恰当地为一个模式命名是很有意义的，因为模式命名象征着长期的专业过程中自然形成的公认的词汇表。这样的简写名称大大促进了软件设计的合作，并且它们的结论是经过量化及测试的。比如当你讨论系列解析器时，你可以简单地告诉同事系统会使用抽象工厂模式来创建每组相应的解析器。你的同事会明智地点头同意，无论立刻觉悟还是先记下稍后再查阅该模式的相关资料。也就是说，模式命名象征着一系列概念，这对于程序员间的交流非常有好处，这在本章后面会详细介绍。

最后，根据国际惯例，介绍模式时应当提及Christopher Alexander。他是一个建筑学家，极大地影响了初期的面向对象模式。他在*A Pattern Language*（牛津大学出版社，1977）一书中写道：

> 每个模式都描述着一种在我们的环境中一遍又一遍地出现的问题，并描述了对该问题的核心解决方案。以此方式你可以使用该方案上百万次，而从不需要重复做同样的事情。

这个始于问题及其广泛的环境继而进入解决方案的定义（被应用于建筑问题及解决方案）意义重大。这几年有一些对设计模式被滥用的指责，特别是被没经验的程序员所使用时。解决方案被应用于并不符合的问题及上下文中通常是滥用的信号。模式是类和对象的一种特殊组织形式，是以定义解决方案的应用条件并讨论其效果的形式来组织的。

本书中，我们将关注模式领域中特别具有影响力的一部分，即Erich Gamma、Richard Helm、Ralph Johnson和John Vlisside所著的《设计模式——可复用面向对象软件的基础》（Addison-Wesley, 1995）一书中所描述的模式。该书集中介绍了面向对象软件开发中的模式，并记录了一些在大多数现代面向对象项目中出现的传统模式。

《设计模式》一书非常重要，不仅因为它记录了关键模式，而且因为它也描述了形成和推动这些模式的设计原则。在下一章中我们便会看到其中的一些原则。

注解　《设计模式》及本书中所描述的模式是模式语言的真实实例，这些模式便是组织在一起的问题和解决方案的一个目录，以使模式可相互补充而形成一个相互关联的整体。当然也存在其他问题的模式语言，如视觉设计和项目管理（当然还有建筑）。但在此我所讨论的设计模式都是针对面向对象软件开发的问题和解决方案的。

7.2　设计模式概览

一个设计模式的核心由4部分组成：命名、问题、解决方案和效果。

7.2.1　命名

命名非常重要。命名丰富了程序员的语言，少许简短的文字便可表示相当复杂的问题和解决方案。命名必须兼顾简洁性和描述性。《设计模式》说道：找到一个好名字，成了我们在开发模式目录中最困难的部分之一。

Martin Fowler也表示了赞同："模式命名至关重要，因为模式的目的之一就是为开发者更有效的交流提供词汇"[《企业应用架构模式》（Addison-Wesley，2002）]。

在《企业应用架构模式》一书中，Martin Fowler改进了一个我曾在《J2EE核心模式》（Deepak Alur、Dan Malks和John Crupi著，Prentice Hall，2003）中了解到的数据库访问模式。他定义了两个新模式来描述旧模式的特例。显然他的逻辑是正确的：新模式之一对领域对象（domain object）进行了建模；另一个新模式对数据库表进行了建模。这区分了之前模式中含糊不清的地方。但最初我很不习惯他所采用的新术语，因为我使用旧的命名方式已经很长时间了，以至于旧的命名已经成为我的语言中的一部分。

7.2.2　问题

无论解决方案如何优雅（有些的确非常优雅），问题及问题发生的环境都是一个模式的基础。

找出问题比使用模式目录中的解决方案更难。这正是某些模式的解决方案被误用或过度使用的原因之一。

模式会小心谨慎地描述问题的空间[①]。问题会被置于环境中简明扼要地描述，通常还会带有典型的范例及一个或多个图表。问题会被分解为不同细节和各种表现。同时任何可以帮助发现问题的警告标志都会在模式中被描述。

7.2.3 解决方案

解决方案最初是和问题放在一起的，并常用UML类图和交互图更详细地进行描述。而模式通常也包含一个代码范例。

尽管代码也许是现成的，但解决方案从来不是简单的剪切及粘贴。模式描述了一个问题的一个解决方法，但在实现时可能会有上百种细微的差别。这就像农作物播种的操作，如果你简单地盲目遵循书本上的步骤，那么在收获季节很可能会挨饿。以模式为基础，但又能针对各种情况随机应变的方法会更实用。虽然问题的基本解决方案（使你的庄稼成长）总是相同的（播种、灌溉、收割），但是实际采用的步骤依赖于各种因素，如土壤类别、地理位置、土地的方位和当地的害虫等。

于是Martin Fowler把模式中的解决方案称为“半成品”。换句话说，编码人员必须理解概念并自己来完成具体的实现。

7.2.4 效果

在设计代码的时候，你所做的每一个决定都会带来不同的结果。当然我们总是希望能得到令人满意的针对问题的解决方案。解决方案一旦被部署，理想情况下它也许会非常适合与其他模式一同工作，但也要留心是否会带来风险。

7.3 《设计模式》格式

编写本书时，在我桌上有5份模式目录。看一下每个目录的模式，便可发现每一个都使用不一样的结构：其中一些比较正式；一些比较细致，有着许多的子分类；还有一些则比较松散。

这些目录中有一些定义良好的模式结构，其中包括由Christopher Alexander原创的格式（Alexandrian格式）和Portland模式库所钟爱的叙述性格式（Portland格式）。不过因为《设计模式》一书极具影响力，而且我们也会介绍该书所描述的很多模式，所以先研究一下该书所采用的模式结构。其主要组成部分如下所示。

- 意图：模式目的的简要概括。你应该一眼就能看出模式的要点。
- 动机：需要被解决的问题，通常根据一个典型的情况。叙述性的方式有助于使模式更容易被领会。
- 适用性：检验不同情况下你是否可以应用某模式。动机描述了一个典型问题，而适用性定义了特殊的情况，并衡量该解决方案在每种情况下的价值。
- 结构/交互：可能包含UML类图和交互图，用于描述解决方案中类和对象之间的关系。

① 即问题发生的条件和环境。——译者注

- 实现：着眼于解决方案的细节，介绍了应用解决方案时可能发生的问题，并提供了部署的技巧。
- 示例代码：我总是先跳到这一部分。我发现简单的代码范例是理解模式的捷径。范例通常都会被简化以突出解决方案的核心内容。示例代码可以用任何一种面向对象语言编写，当然本书中使用的是PHP。
- 已知应用：使用该模式（问题、上下文及解决方案）的真实系统。有些人会说一个模式要变得真实，它就必须出现在至少3个公开存在的真实系统中。这有时会被称为"三法则"（rule of three，意思是解决某个特定问题所采取的设计，一次出现是偶然现象，两次是巧合，3次出现才可称为一个模式）。
- 相关模式：一些模式意味着其他模式也要被采用。在使用某个设计模式时，可以创造出另一个模式适用的条件。这部分内容研究了可能存在的模式间的合作，也可能会讨论那些问题或解决方案中具有相似点的模式及任何需要预先使用的模式（当前模式可能以其他模式为基础）。

7.4　为什么使用设计模式

那么模式可以带来什么好处呢？如果模式是对问题的定义和对解决方案的描述，那么答案相当明显，即模式可以帮助解决共性的问题。当然，模式还能带来更多好处。

7.4.1　一个设计模式定义了一个问题

有多少次在项目到了某个阶段时你发现无法继续？这时你很可能必须以某种方式返回而不是继续尝试前进。

通过定义共性问题，模式能帮助你改进设计。有时找到解决方案的第一步便是认清你面对的问题。

7.4.2　一个设计模式定义了一个解决方案

在定义和识别当前存在的问题后，模式给你提供了解决方案及应用该方案所得结果的分析。尽管模式并不能代替你来作出决策，但至少能确认你使用的是一个可靠并经过测试的技术。

7.4.3　设计模式是语言无关的

模式以面向对象的方式来定义对象和解决方案。这意味着大多数模式可被直接应用于多种编程语言中。例如我第一次使用模式时阅读的是C++和Smalltalk的代码范例，却在Java环境中部署我的解决方案。另一些模式针对不同语言可能有不同的适用性或效果，但总是可行的。无论是哪种情况，模式总可以在不同语言中给你帮助。同时，一个建立于优秀的面向对象设计原则之上的应用能被相对简单地在不同语言中移植（尽管总有些问题需要处理）。

7.4.4　模式定义了一组词汇

通过给开发人员提供技术性的名称，模式使沟通更加丰富。想象一下，在一个设计会议上

我已经介绍过我的基于“抽象工厂”的解决方案，现在我需要向Bob描述我的管理系统数据的策略。

我：我正在考虑使用组合（Composite）模式。

Bob：我觉得你还需要再认真想想。

Bob并不同意我的观点。他从来没有同意过我的观点。但他知道我在说什么，因此知道我的想法为什么不好。让我们在没有设计模式词汇的情况下再重演一下刚才的场景。

我：我尝试使用一个相同类型的对象树。该类型的接口会提供添加该类型子对象的方法。通过这种方式，我们能在运行时建立起实现对象的复杂合并。

Bob：啊？

我们所提到的模式或模式中所描述的技术，常常可以互相补充、合作，比如组合模式适合于与访问者模式相互协作。

我：然后我们能使用访问者模式来汇总数据。

Bob：不明白你想说什么。

让我们先忽略Bob。我可不愿描述啰嗦的非模式版本。我们会在第10章中介绍组合模式，在第11章中介绍访问者模式。

即使没有模式语言，我们仍旧会使用这些技术。技术是超越它们的命名和组织的。如果模式不存在，技术仍会以它们自己的方式发展。但任何被广泛使用的工具最终都会获得一个名称。

7.4.5 模式是经过测试的

如果模式验证了高质量的实践，那么名称是否是模式目录仅有的真正原创的东西？从某种意义上说，好像是这样的。在面向对象的环境中，模式代表着最佳实践。对于一些经验非常丰富的程序员来说，这就像是再包装明显的最佳实践。对于我们而言，模式提供了发现问题和解决方案的途径，否则我们将不得不非常辛苦地去发掘。

模式使我们更容易得到好的设计方案。当模式目录变得越来越专业化，即使经验丰富的程序员也会在进入新领域的时候从中受益。例如，图形用户界面程序员可以快速理解企业级开发中的常见问题和解决方案。Web程序员能快速制定策略来避免PDA和智能手机项目的缺陷。

7.4.6 模式是为协作而设计的

模式生来就是“可生成的”（generative）和“可组成的”（composable）。这意味着你能应用一个模式，并因此创建适合应用另一个模式的条件。换句话说，在使用某个模式时你也许会发现另一扇门为你敞开了，你可以同时使用其他模式。

模式目录通常会关注于这类协作，而模式本身也会介绍潜在的模式组合。

7.4.7 设计模式促进良好设计

设计模式示范并应用了面向对象设计原则。因此设计模式能在一个具体环境中产生多个特定的解决方案，而你可以从一个新的角度来学习如何组合对象和类来达到目标。

7.5 PHP与设计模式

本章中只有很小一部分是专门针对PHP的，从某种程度上说它算是本章的一点特色。许多模式都可以应用于各种具有面向对象能力的语言，实现起来通常不会有什么问题。

当然，并不总是这样。一些企业模式只有在服务器请求对应一个应用进程的情况下才能很好地工作。PHP并不能以那样的方式工作。在PHP中，每一个请求都会开始一个新的脚本执行。这意味着要更谨慎地对待一些模式。例如，前端控制器（Front Controller）模式通常需要较多的初始化时间。如果初始化仅在应用启动时发生一次，那么还没问题，但如果每次请求都需要初始化，就会导致问题。这并不是说我们不能使用该模式，因为我在以前成功地使用过该模式。我们必须确保在讨论模式的时候考虑到PHP相关的问题，因为PHP正是本书研究所有模式时的具体环境。

在本节开头我提到过“具有对象能力的语言”。你根本不必定义任何类就可编写PHP代码（但如果使用PEAR，你将会使用到一些对象）。虽然本书几乎完全关注于用面向对象解决方案来解决编程问题，但并不是在鼓吹面向对象。模式和PHP能被有效地混合，而且它们也形成了本书的核心，另外它们也能与许多传统方式很好地共存。对此PEAR便是一个极好的证明。PEAR包优雅地使用了设计模式，它们在本质上趋于面向对象。但这个特性使它们在过程式项目中更加实用（而非不实用）。因为PEAR包是自我封装的，并将它们自身的复杂性都隐藏在十分干净的接口下，所以它们能被轻松地嵌入任何类型的项目中。

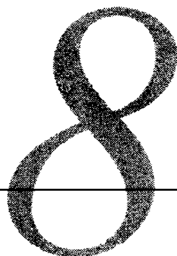
7.6 小结

本章介绍了设计模式及其结构（通过使用《设计模式》一书的格式），并给出了一些使用设计模式的理由。

要记住，设计模式并非像组件那样能被合并来构建系统的固定解决方案。它们是解决一般性问题的通用方法。这些解决方案体现了一些关键的设计原则，在下一章中我们将进一步研究这些设计原则。

第8章

模式原则



设计模式不只是简单地描述了问题的解决方案，而且很重视解决方案的可重用性和灵活性。为此，设计模式提出了一些关键的面向对象设计原则。本章会讲到一些设计原则，并在其他章节中详细介绍。

本章包括以下内容。

- 组合：如何通过聚合对象来获得比只使用继承更好的灵活性。
- 解耦：如何降低系统中元素间的依赖性。
- 接口的作用：模式和多态。
- 模式分类：本书将介绍的模式类别。

8.1 模式的启示

我最早是在Java语言中使用对象的。正如你可能预料到的那样，我花了不少时间才弄明白一些概念。但是一旦明白了，进展便会非常快，仿佛有一种豁然开朗的感觉。比如继承和封装的优雅曾让我激动不已，我意识到这是一种定义和构建系统的新途径。而多态，其实就是一个类型的实现在运行时进行了切换。

那时我桌上所有的书都关注语言特性和Java程序员应该学会使用的各种API。除了多态的简要定义之外，这些书几乎没有涉及设计策略。

单独的语言特性并不能形成面向对象设计。虽然我的项目满足了他们的功能需求，但完全没有用上继承、封装和多态这些设计方法。

因为我努力地为每一个可能性建造新类，所以我的继承层次体系逐渐变得更广、更深。而在这样的系统结构中，如果中间类对于环境没有足够的了解，如果没有将它们绑定到具体应用中，如果没有使它们只可用于当前局部环境中，那么在系统的层级中传递信息将会变得十分困难。

直到发现《设计模式》一书，我才意识到原来我没有完全理解什么是设计。虽然那时我自己也已经发现了一些核心模式，但是书中介绍的其他模式则提供了一种全新的思维方式。

我在设计中给了继承过多的特权，总是试图为我的类构建太多的功能。在面向对象系统里还有别的地方可以放置这些功能吗？

我在组合模式中找到了答案。通过以灵活的关系来组合对象，组件能在运行时被定义。《设计模式》将这提炼成了一个原则：组合优于继承（favor composition over inheritance）。在该模式中，运行时组合对象所达到的灵活性非常高，而这在单独的继承树中是不可能达到的。

8.2　组合与继承

继承是应对变化的环境及上下文设计的有效方式，然而它会限制灵活性，尤其当类承担多重责任的时候。

8.2.1　问题

如你所知，子类继承了父类的方法和属性（只要它们被声明为protected或public）。我们以此为基础来设计提供特殊功能的子类。

图8-1使用UML来描述一个简单的示例。

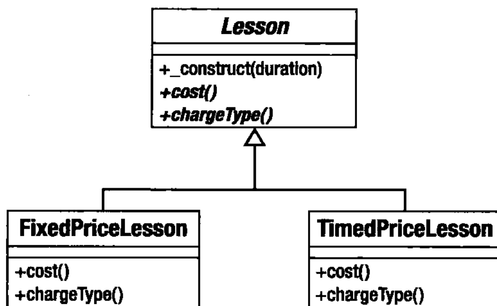


图8-1　一个父类和两个子类

图8-1中的Lesson抽象类是对大学课程的建模，它定义了抽象的cost()和chargeType()方法。图8-1中有两个实现类——FixedPriceLesson和TimedPriceLesson，它们为课程提供了不同的收费机制。

利用这种继承模式，我们可以在课程的实现之间切换。而客户端代码只知道它是在处理一个Lesson对象，因此费用的细节就会变得透明。

可是如果引入一组新的特殊性，又会怎样呢？比如我们需要处理演讲和研讨会。因为演讲和研讨会会以不同的方式注册登记和教授课程，所以它们会要求独立的类。因此在设计上现在会有两个分支。我们需要处理不同的定价策略并区分演讲和研讨会。

图8-2是一个比较费事的方案。

图8-2的体系明显是有缺陷的。在该体系中，我们不得不大量重复开发功能，否则无法使用继承树来管理价格机制。定价策略在Lecture和Seminar类的子类中被重复实现。

我们可能要考虑在父类Lesson中使用条件语句来移除那些不适宜的重复。我们是把定价逻辑从继承树中一并移除并迁移到父类中，但这与我们通常用多态替换条件的重构思想背道而驰。下面是一个修改过的Lesson类。

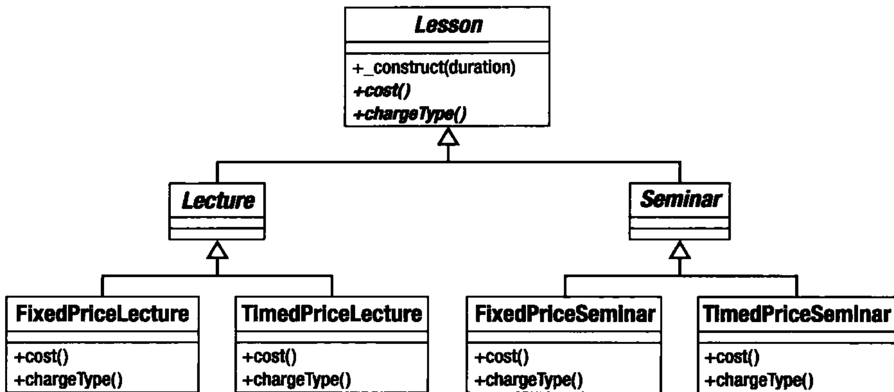


图8-2 拙劣的继承结构

```
abstract class Lesson {
    protected $duration;
    const     FIXED = 1;
    const     TIMED = 2;
    private   $costtype;

    function __construct( $duration, $costtype=1 ) {
        $this->duration = $duration;
        $this->costtype = $costtype;
    }

    function cost() {
        switch ( $this->costtype ) {
            CASE self::TIMED :
                return (5 * $this->duration);
                break;
            CASE self::FIXED :
                return 30;
                break;
            default:
                $this->costtype = self::FIXED;
                return 30;
        }
    }

    function chargeType() {
        switch ( $this->costtype ) {
            CASE self::TIMED :
                return "hourly rate";
                break;
            CASE self::FIXED :
                return "fixed rate";
                break;
            default:
                $this->costtype = self::FIXED;
                return "fixed rate";
```

```
        }
    }

    // Lesson的更多方法
}

class Lecture extends Lesson {
    // Lecture特定的实现
}

class Seminar extends Lesson {
    // Seminar特定的实现
}
```

下面演示如何使用这些类：

```
$lecture = new Lecture( 5, Lesson::FIXED );

print "{$lecture->cost()} ({$lecture->chargeType()})\n";


$seminar= new Seminar( 3, Lesson::TIMED );

print "{$seminar->cost()} ({$seminar->chargeType()})\n";
```

下面是相应的输出：

```
30 (fixed rate)
15 (hourly rate)
```

在图8-3中，我们能看到新的类图。

我们把类结构变得更加便于管理，不过这要付出一些代价。在父类的代码中使用条件语句是一种倒退。通常，我们会用多态来代替条件语句，而这里我们却使用了条件语句。如你所见，这里迫使我们在chargeType()和cost()方法中重复这些条件语句。

于是代码重复似乎是注定的。

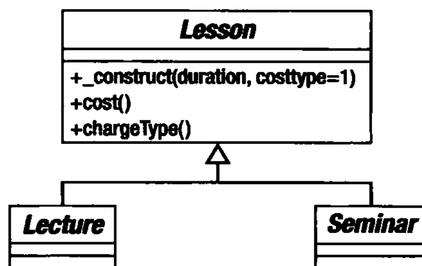


图8-3 改进后的继承体系：移除子类中的费用计算

8.2.2 使用组合

我们可以使用策略（Strategy）模式来解决这个问题。策略模式适用于将一组算法移入到一个独立的类型中。通过移走费用计算相关的代码，可以简化Lesson类型。这可以在图8-4中看到。

我们要创建一个名为CostStrategy的抽象类，它定义了抽象方法cost()和chargeType()。cost()方法需要一个Lesson实例，用于生成费用数据。我们提供了CostStrategy的两个实现。因为Lesson对象只需要CostStrategy类型的对象即可工作，不需要指定具体类型的子类（子类的类型），所以我们在任何时候都能通过创建CostStrategy的子类来添加新的费用算法，而这不

会要求对任何Lesson类进行修改。

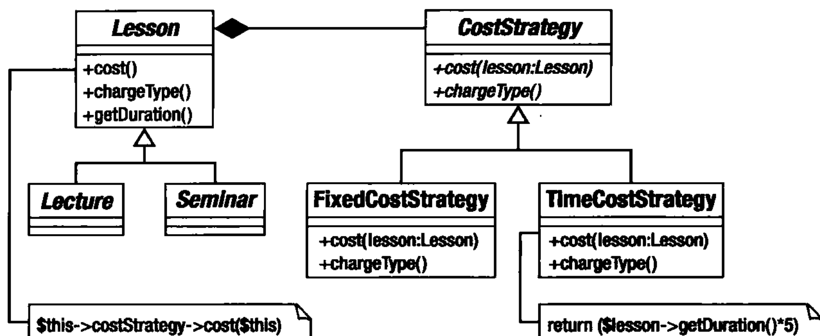


图8-4 将算法移入独立的类型中

图8-4描述的Lesson类的代码如下：

```
abstract class Lesson {
    private    $duration;
    private    $costStrategy;

    function __construct( $duration, CostStrategy $strategy ) {
        $this->duration = $duration;
        $this->costStrategy = $strategy;
    }

    function cost() {
        return $this->costStrategy->cost( $this );
    }

    function chargeType() {
        return $this->costStrategy->chargeType( );
    }

    function getDuration() {
        return $this->duration;
    }

    // Lesson类的更多方法
}

class Lecture extends Lesson {
    // Lecture特定的实现
}

class Seminar extends Lesson {
    // Seminar特定的实现
}
```

Lesson类需要一个作为属性的CostStrategy对象。Lesson::cost()方法只调用CostStrategy::cost()。同样，Lesson::chargeType()只调用CostStrategy::chargeType()。这种

显式调用另一个对象的方法来执行一个请求的方式便是所谓的“委托”。在我们的示例中，CostStrategy对象便是Lesson的委托方。Lesson类不再负责计费，而是把计费任务传给Cost-Strategy类。下面的代码执行了委托操作：

```
function cost() {
    return $this->costStrategy->cost( $this );
}
```

下面是CostStrategy类及其实现子类：

```
abstract class CostStrategy {
    abstract function cost( Lesson $lesson );
    abstract function chargeType();
}

class TimedCostStrategy extends CostStrategy {
    function cost( Lesson $lesson ) {
        return ( $lesson->getDuration() * 5 );
    }
    function chargeType() {
        return "hourly rate";
    }
}

class FixedCostStrategy extends CostStrategy {
    function cost( Lesson $lesson ) {
        return 30;
    }

    function chargeType() {
        return "fixed rate";
    }
}
```

通过传递不同的CostStrategy对象，我们可以在代码运行时改变Lesson对象计算费用的方式。这种方式有助于产生具有高度灵活性的代码。动态地组合及重组对象，远胜于将功能静态地建立在代码结构中。

```
$lessons[] = new Seminar( 4, new TimedCostStrategy() );
$lessons[] = new Lecture( 4, new FixedCostStrategy() );

foreach ( $lessons as $lesson ) {
    print "lesson charge {$lesson->cost()}. ";
    print "Charge type: {$lesson->chargeType()}\n";
}

lesson charge 20. Charge type: hourly rate
```

```
lesson charge 30. Charge type: fixed rate
```

如你所见，此结构的效果之一便是让我们关注类的职责。CostStrategy对象独立负责计算

费用，而Lesson对象则负责管理课程数据。

所以，组合使用对象比使用继承体系更灵活，因为组合可以以多种方式动态地处理任务，不过这可能导致代码的可读性下降。因为组合需要更多的对象类型，而这些类型的关系并不像在继承关系中那般有固定的可预见性，所以要理解系统中类和对象的关系会有些困难。

8.3 解耦

在第6章中，我们了解了创建独立组件的意义。如果类之间有非常强的依赖性，那么这样的系统就很难维护，因为系统里的一个改动会引起一连串的相关改动。

8.3.1 问题

重用性是面向对象设计的主要目标之一，而紧耦合（tight-coupling）便是它的敌人。当我们看到系统中一个组件的改变迫使系统其他许多地方也发生改变的时候，就可诊断为紧耦合了。为了能安全地做变动，我们总是期望创建能够独立存在的组件。在修改组件时，其独立程度会决定你的修改对系统中其他组件的影响程度，系统的其他组件甚至有可能会因此失败。

在图8-2中，我们看到过紧耦合的例子。因为费用计算逻辑在Lecture和Seminar类型中都存在，所以对TimedPriceLecture的一个改变将会迫使在TimedPriceSeminar中同样逻辑的相应变化。如果仅改动一个类而不改动其他类的代码，系统将无法正常工作，而且没有来自PHP引擎的任何警告。而我们的第一个解决方案（使用条件语句）在cost()和chargeType()方法之间生成了一个类似的依赖关系。

通过应用策略模式，我们将费用算法提取为CostStrategy类型，将算法放置在共同接口后，并且每个算法只需实现一次。

不过当系统中许多类都显式嵌入到一个平台或环境中时，其他类型的耦合仍时有发生。比如建立了一个基于MySQL数据库的系统。你可能会用一些诸如mysql_connect()和mysql_query()的函数来与数据库服务器交互。

如果现在你被要求在不支持MySQL的服务器上部署系统，比如要把整个项目都转换成使用SQLite，那么你可能被迫要改变整个代码，并且面临维护应用程序的两个并行版本的状况。

这里的问题不在于系统对外部平台的依赖。这样的依赖是无法避免的。我们确实需要使用与数据库交互的代码。但当这样的代码散布在整个项目中时，问题就来了。与数据库交互不是系统中大部分类的首要责任，因此最好的策略就是提取这样的代码并将其组合在公共接口后。这可以使类之间相互独立。同时，通过在一个地方集中你的“入口”代码，就能更轻松地切换到一个新的平台而不会影响到系统中更大的部分。这个把具体实现隐藏在一个干净的接口后面的过程，正是大家所知道的“封装”。

PEAR中的PEAR::MDB2包（沿袭自PEAR::DB）可以解决这个问题。该包支持对多个数据库的访问。最新的PDO扩展已将此模型移植到PHP语言中。

MDB2类提供了一个静态方法connect()，它接受一个DSN（Data Source Name，数据源名）字符串参数。根据这个字符串的构成，它返回MDB2_Driver_Common类的一个特定实现。因此对

于字符串"mysql://"，connect()方法返回一个MDB2_Driver_mysql对象，而对于一个以"sqlite://"开头的字符串，它将返回一个MDB2_Driver_sqlite对象。可以在图8-5中看到该类的结构。

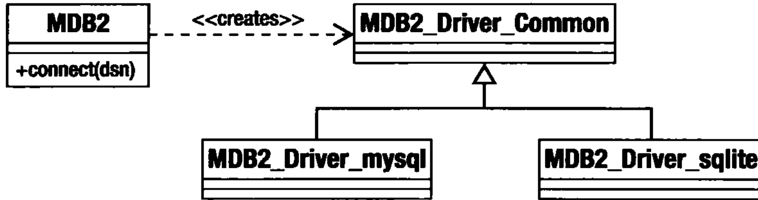


图8-5　PEAR::MDB2包将客户端代码从数据库对象分离

使用PEAR::MDB2包，可以从数据库平台的特殊性中将应用程序代码分离出来。你只要使用SQL语句，就可以在MySQL、SQLite、MSSQL和其他数据库上运行一个单一的系统而不需要改变一行代码（当然DSN除外，DSN是数据库环境必须被设定的一个点）。事实上，PEAR::MDB2包在一定程度上也能帮助你管理不同的SQL语法——这是使你放弃PDO的速度与便捷而选择使用PEAR::MDB2包的一个原因。

8.3.2　降低耦合

为了灵活处理数据库代码，我们应该将应用逻辑从数据库平台的特殊性中解耦出来。

在你自己的项目中，你会看到很多这种需要分离组件的情况。

例如，课程系统中应包含注册组件，从而向系统中添加新课程。添加了新课程后，应该通知管理员，这是注册程序的一部分。对于应该通过邮件发送通知还是通过文本消息发送通知，系统用户的意见不一致。实际上，他们太挑剔了，以至于你怀疑将来他们会想使用一种新的信息传达模式。此外，他们希望发生任何事情都会收到通知。所以，修改了通知模式的一处意味着要对多处做同样的修改。

如果已经硬编码了对Mailer类或Texter类的调用，那么系统就与特殊的通知模式紧密相关了。就像利用专门的数据库API时，系统就与某数据库平台紧密相关一样。

下面的这些代码对使用通知程序的系统隐藏了通知程序的实现细节：

```
class RegistrationMgr {

    function register( Lesson $lesson ) {

        // 处理该课程


        // 通知某人

        $notifier = Notifier::getNotifier();

        $notifier->inform( "new lesson: cost ({$lesson->cost()})" );
```

```
    }
}

abstract class Notifier {

    static function getNotifier() {
        // 根据配置或其他逻辑获得具体的类


        if ( rand(1,2) == 1 ) {
            return new MailNotifier();
        } else {
            return new TextNotifier();
        }
    }

    abstract function inform( $message );
}

class MailNotifier extends Notifier {
    function inform( $message ) {
        print "MAIL notification: {$message}\n";
    }
}

class TextNotifier extends Notifier {
    function inform( $message ) {
```

```
        print "TEXT notification: {$message}\n";
    }
}
```

我创建了RegistrationMgr作为Notifier类的示例客户端。Notifier是抽象类，但它却实现了一个静态方法getNotifier()，该方法获取具体的Notifier对象（TextNotifier或MailNotifier）。在实际的项目中，Notifier对象的选择是由一种灵活的机制（比如配置文件）所决定的。这里，我做了手脚，随机选择Notifier对象。TextNotifier和 MailNotifier只是输出传递给它们的包含一个标识符的消息，以显示调用了哪个对象。

注意，具体应该使用哪个Notifier对象取决于Notifier::getNotifier()方法。我可以从系统的上百个不同部分发送消息，但只需在该方法中对Notifier进行一项修改即可。

下面是调用RegistrationMgr的部分代码：

```
$lessons1 = new Seminar( 4, new TimedCostStrategy() );
$lessons2 = new Lecture( 4, new FixedCostStrategy() );
$mgr = new RegistrationMgr();
$mgr->register( $lessons1 );
$mgr->register( $lessons2 );
```

下面是运行后的输出结果：

```
TEXT notification: new lesson: cost (20)
MAIL notification: new lesson: cost (30)
```

图8-6显示了这些类。

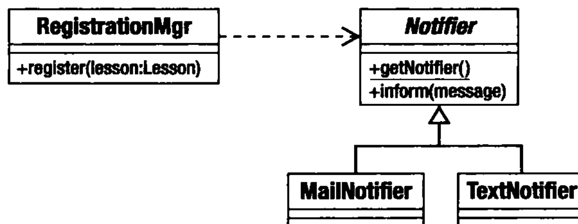


图8-6　Notifier类将客户端代码与Notifier实现分离

注意，图8-6中的结构和图8-5中由MDB2组件组成的结构非常相似。

8.4　针对接口编程，而不是针对实现编程

这个设计原则是贯穿本书的主题之一。在第6章中（以及本章最后一部分），我们了解到可以

把不同的实现隐藏在父类所定义的共同接口下。然后客户端代码需要一个父类的对象而不是一个子类的对象，从而使客户端代码可以不用关心它实际得到的是哪个具体实现。

我们在Lesson::cost()和Lesson::chargeType()中创建的并行条件语句，就是需要多态的常见标志。这样的条件语句使代码很难维护，因为条件表达式的改变必然要求与之对应的代码主体也随之改变，所以条件语句有时会被称作实现了一个“模拟继承”。

而通过把计费算法放置在一个实现CostStrategy的独立的类中，我们可以移除重复代码，也可以使在未来加入新的计费策略变得更加容易。

从客户端代码的角度看，类方法参数为抽象或通用类型通常都是不错的主意。如果参数对对象类型要求过于严格，就会限制代码在运行时的灵活性。

当然，如何使用参数类型提示来调整参数对象的“通用性”是需要仔细权衡的。选择过于通用，则会降低方法的安全性。而如果需要某个子类型的特有功能，那么方法接受另一个子类类型则可能会有风险。

尽管如此，若参数的类型匹配限制过于严格，那么将无法得到多态带来的好处。下面是修改过的Lesson类里的一段代码：

```
function __construct( $duration,
    FixedPriceStrategy $strategy ) {
    $this->duration = $duration;
    $this->costStrategy = $strategy;
}
```

这个示例中的设计有两个问题。第一，Lesson对象现在与一个特定的费用策略绑定，这使我们丧失了组合动态组件的能力。第二，对FixedPriceStrategy类的显式引用迫使我们必须维护这个特殊实现。

而通过要求一个公共的接口，你能将任何CostStrategy实现合并到一个Lesson对象：

```
function __construct( $duration, CostStrategy $strategy ) {
    $this->duration = $duration;
    $this->costStrategy = $strategy;
}
```

换句话说，我们把Lesson类从具体的费用计算中分离出来了。我们所做的就是提供接口并保证所提供的对象会实现接口。

当然，面向接口编程无法回答如何实例化对象的问题。当我们说Lesson对象能在运行时与任何CostStrategy接口绑定时，我们回避了这么一个问题：“但是CostStrategy对象从哪里来呢？”

当创建一个抽象父类时，常会碰到如何实例化它的子类的问题。你会选择实例化哪个子类来对应相应的条件呢？这个主题在《设计模式》模式目录中形成了一个独立的类别，我们会在下一章研究其中一些模式。

8.5 变化的概念

一旦作出设计决定，解释它就很容易。但是你如何决定从哪里开始设计呢？

《设计模式》建议你“把变化的概念封装起来”（encapsulate the concept that varies）。在我们的示例中，“变化的概念”便是费用算法。计费不仅仅是示例中两个可能的策略之一，而且它也是明显需要扩充的：如特殊供应、海外学生费用、推介性折扣等各种各样的可能性。

我们发现为这个变化直接创建子类是不合适的，于是使用了条件语句。通过把变化的元素放入同一个类中，我们强调了封装的适用性。

《设计模式》建议积极搜寻类中变化的元素，并评估它们是否适合用新类型来封装。根据一定条件，变化的元素（如计费算法）可被提取出来形成子类（如`TimedCostStrategy`和`FixedCostStrategy`），而这些元素共同拥有一个抽象父类（`CostStrategy`）。而这个新类型（`CostStrategy`）能被其他类使用。这么做有以下好处：

- 专注于职责；
- 通过组合提高灵活性；
- 使继承层级体系更紧凑和集中；
- 减少重复。

那么如何发现变化的元素呢？误用继承便是一个标志。误用的表现可能包括一次实现不同分支（如演讲/研讨会，固定/按时间的费用计算）的继承；也可能包括子类化某个算法，而该算法对于该对象类型的核心职责是偶然的。当然，适合封装“变化元素”的另一个标志便是出现了条件表达式。

8.6　父子关系

模式的一个问题便是不必要或不恰当地使用模式。这让模式在某些领域名声不佳。因为模式解决方案很棒，所以它会引诱你把模式应用在任何你认为合适的地方，无论它们是否真的适合用来达到目标。

极限编程（XP，eXtreme Programming）提供了几个可以使用的相关原则。第一个是“你还不需要它”（You aren't going to need it，常简称为YAGNI）。这通常被应用在应用程序的功能上，但是对于模式来说也有意义。

当使用PHP开发较大的项目时，我会把应用程序分离到各个层中，把应用程序逻辑从表现层和持久化层中分离开来。我通常联合使用各种核心模式和企业模式。

然而当为一个小型商务网站建立一个用户反馈表单时，我可能只在单一页面脚本中使用过程式代码。此时，不需要大量的灵活性，不需要以后基于最初版本进行大量扩展，也不需要使用那些在更庞大的系统中解决问题的模式，而是应用极限编程的第二个原则：“用最简单的方式来完成任务”（Do the simplest thing that works）。

当使用模式目录时，通过示例代码能巩固你脑中解决方案的结构和流程。然而在应用模式之前，要特别注意目录中“问题”或者“何时使用”那部分，并且熟读模式的效用。在某些情况下，错误的治疗会比疾病本身更糟。

8.7　模式

本书并非简单罗列几个模式。但在接下来的几章中，我会介绍一些关键的设计模式，提供用

PHP实现的示例并在PHP编程的语境下讨论它们。

被描述的模式将会从主要的模式目录［包括《设计模式》、《企业应用架构模式》（Martin Fowler著，Addison-Wesley，2003）和《J2EE核心模式》（Alue等著，Prentice Hall出版社，2001）］中提取出来，我将以《设计模式》一书的分类为起点，将模式分为以下几种。

8.7.1 用于生成对象的模式

这类模式关注对象的实例化。考虑到“面向接口编程”原则，这是一个重要的分类。如果在设计中使用抽象父类，那么我们必须考虑从具体子类实例化对象的策略。实例化得到的对象会在系统中被传递。

8.7.2 用于组织对象和类的模式

这类模式帮助我们组织对象的组成关系。更简单地说，就是这些模式教我们如何合并对象和类。

8.7.3 面向任务的模式

这类模式描述了如何让类和对象合作来达成特定目标。

8.7.4 企业模式

我们着眼于一些描述典型因特网编程问题和解决方案的模式。它们很大程度上来自于《企业应用架构模式》和《J2EE核心模式》这两本书，用于处理表现逻辑及应用逻辑。

8.7.5 数据库模式

数据库存取数据及对象-数据库映射的相关模式。

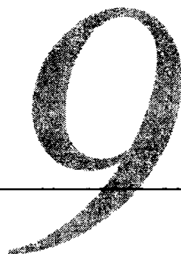
8.8 小结

在本章中，我们学习了一些设计模式背后的设计原则，讨论了在运行时使用组合来合并及再合并对象（这比单独使用继承的灵活性更高），介绍了解耦并提取组件来使它们更具适用性，回顾了接口（它将客户端代码与实现的具体细节相分离）的重要性。

在接下来的几章中，我们将详细地讨论一些设计模式。

第9章

生成对象



创建对象是一件棘手的事情。利用多态带来的灵活性（在运行时切换不同的具体实现），我们可以采用多种面向对象设计方案来处理优美且简洁的抽象类。为了达到这样的灵活性，我们必须仔细考虑生成对象的策略。这是本章要研究的主题。

本章包括以下内容。

- 单例（Singleton）模式：生成一个且只生成一个对象实例的特殊类。
- 工厂方法（Factory Method）模式：构建创建者类的继承层级。
- 抽象工厂（Abstract Factory）模式：功能相关产品的创建。
- 原型（Prototype）模式：使用克隆来生成对象。

9.1 生成对象的问题和解决方法

对象创建有时会成为面向对象设计的一个薄弱环节。在前一章中，我们看到“针对接口编程，而不是针对实现编程”的原则。就此而言，鼓励在类中使用抽象的超类。这使代码更具灵活性，可以让你在运行时使用从不同的具体子类中实例化的对象。但这样做也有副作用，那就是对象实例化被推迟。

类Employee的构造方法以姓名字符串为参数，实例化了一个特定对象：

```
abstract class Employee {
    protected $name;
    function __construct( $name ) {
        $this->name = $name;
    }
    abstract function fire();
}

class Minion extends Employee {
    function fire() {
        print "{$this->name}: I'll clear my desk\n";
    }
}

class NastyBoss {
```

```
    private $employees = array();

    function addEmployee( $employeeName ) {
        $this->employees[] = new Minion( $employeeName );
    }

    function projectFails() {
        if ( count( $this->employees ) > 0 ) {
            $emp = array_pop( $this->employees );
            $emp->fire();
        }
    }
}

$boss = new NastyBoss();
$boss->addEmployee( "harry" );
$boss->addEmployee( "bob" );
$boss->addEmployee( "mary" );
$boss->projectFails();

// 输出:
// Mary: 我马上打包走人
```

如你所看到的，我们定义了一个抽象基类Employee（雇员）以及一个受压迫员工的具体实现Minion。NastyBoss::addEmployee()方法通过接受的名字字符串来实例化新的Minion对象。一旦NastyBoss（苛刻的老板）对象遇到了麻烦（通过NastyBoss::projectFails()方法），它就会解雇一个Minion。

由于在NastyBoss类中直接实例化Minion对象，代码的灵活性受到了限制。如果NastyBoss对象可以使用Employee类的任何实例，那么代码在运行时就能应对更多特殊的Employee。在图9-1中，应该可以找到你熟悉的多态。

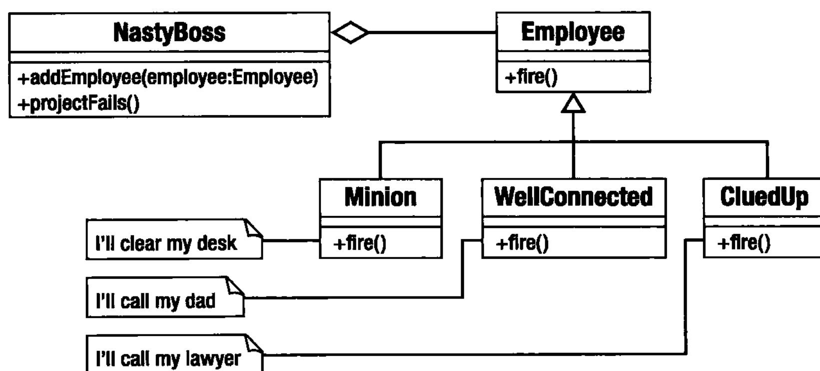


图9-1 抽象类型的使用启用了多态

如果NastyBoss类不实例化Minion对象，那么Minion对象从何而来？许多人通常在方法声明中限制参数类型来巧妙避开这个问题，然后除了在测试时实例化对象，在其他时候尽量避免提及。

```
class NastyBoss {
    private $employees = array();

    function addEmployee( Employee $employee ) {
        $this->employees[] = $employee;
    }

    function projectFails() {
        if ( count( $this->employees ) ) {
            $emp = array_pop( $this->employees );
            $emp->fire();
        }
    }
}

// 新Employee类
class CluedUp extends Employee {
    function fire() {
        print "{$this->name}: I'll call my lawyer\n";
    }
}

$boss = new NastyBoss();
$boss->addEmployee( new Minion( "harry" ) );
$boss->addEmployee( new CluedUp( "bob" ) );
$boss->addEmployee( new Minion( "mary" ) );
$boss->projectFails();
$boss->projectFails();
$boss->projectFails();
// 输出:
// mary: 我立马走人
// bob: 我会打电话给我的律师
// harry: 我立马走人
```

虽然这个版本的NastyBoss类能与Employee类型一起工作，而且也能从多态中获益，但我们仍旧没有定义创建对象的策略。实例化对象是一件麻烦事，但是我们又不得不去做。这一章讲的是使用具体类的类和对象。这样，不使用具体类的类和对象不一定非要实例化对象。

如果说这里存在一个原则的话，那便是“把对象实例化的工作委托出来”。之前的示例已然隐含了这个原则，即要求将一个Employee对象传递给NastyBoss::addEmployee()方法。然而我们也可以委托一个独立的类或方法来生成Employee对象，其效果是一样的。下面给Employee类添加一个实现了对象创建策略的静态方法。

```
abstract class Employee {
    protected $name;
    private static $types = array( 'minion', 'cluedup', 'wellconnected' );

    static function recruit( $name ) {
        $num = rand( 1, count( self::$types ) )-1;
        $class = self::$types[$num];
        return new $class( $name );
    }
```

```
    function __construct( $name ) {
        $this->name = $name;
    }
    abstract function fire();
}

// 新Employee类

class WellConnected extends Employee {
    function fire() {
        print "{$this->name}: I'll call my dad\n";
    }
}
```

正如你所看到的，这里通过一个姓名字符串来随机实例化具体的Employee子类。现在我们可以将实例化的细节委托给Employee类的recruit()方法。

```
$boss = new NastyBoss();
$boss->addEmployee( Employee::recruit( "harry" ) );
$boss->addEmployee( Employee::recruit( "bob" ) );
$boss->addEmployee( Employee::recruit( "mary" ) );
```


我们曾在第4章中见过类似的例子。我们在ShopProduct类中放置了一个静态方法getInstance()。getInstance()负责以数据库查询为基础生成正确的ShopProduct子类。因此ShopProduct类具有双重角色。它定义了ShopProduct类型，同时它也作为实体ShopProduct对象的工厂。

注解 本章中常常提到“工厂”（factory）这个术语。工厂就是负责生成对象的类或方法。

```
// ShopProduct类

    public static function getInstance( $id, PDO $dbh ) {
        $query = "select * from products where id = ?";
        $stmt = $dbh->prepare( $query );

        if ( ! $stmt->execute( array( $id ) ) ) {
            $error=$dbh->errorInfo();
            die( "failed: ".$error[1] );
        }

        $row = $stmt->fetch( );
        if ( empty( $row ) ) { return null; }

        if ( $row['type'] == "book" ) {
            // 实例化一个BookProduct对象
        } else if ( $row['type'] == "cd" ) {
            $product = new CdProduct();
            // 实例化一个CdProduct对象
        } else {
            // 实例化一个ShopProduct对象
```

```
        }
        $product->setId( $row['id'] );
        $product->setDiscount( $row['discount'] );
        return $product;
    }
```

getInstance()方法使用一系列if/else语句来决定实例化哪个子类。像这样的条件语句在工厂代码中十分常见。尽管我们常尝试从项目中消除大量的条件语句，但生成对象确实需要使用这些条件语句。一般来说这不是一个严重问题，因为我们将代码中并行的条件语句转移到getInstance()来，由getInstance()来决定对象生成。

在本章中，我们将研究一些用于生成对象的关键模式，它们来源于《设计模式》一书。

9.2　单例模式

全局变量是面向对象程序员遇到的引发bug的主要原因之一。这是因为全局变量将类捆绑于特定的环境，破坏了封装（参见第6章及第8章）。如果新的应用程序无法保证一开始就定义了相同的全局变量，那么一个依赖于全局变量的类就无法从一个应用程序中提取出来并应用到新应用程序中。

尽管这并不是我们想要的，但全局变量不受保护的本质的确是个很大的问题。一旦开始依赖全局变量，那么某个类库中声明的全局变量和其他地方声明的全局变量迟早会发生冲突。我们已经看到过PHP易受到类名冲突的影响，但全局变量的冲突更加糟糕——PHP并不会对全局变量冲突发出任何警告。你也许只是发现代码行为有些古怪。更糟的是，在开发环境中你也许根本发现不了任何问题。因此如果在类库中使用全局变量，用户可能在尝试将你的类库与其他类库一同部署时遇到冲突。

然而，全局变量仍是一个诱惑。因为有时我们为了使所有类都能访问某个对象，会不惜忍受全局访问的缺陷。

我提到过，命名空间在一定程度上避免了命名冲突。你至少可以将变量的作用域定义在包中，这意味着第三方的库与你的系统产生冲突的可能性大大降低了。即便如此，命名空间内部还是存在命名冲突。

9.2.1　问题

经过良好设计的系统一般通过方法调用来传递对象实例。每个类都会与背景环境保持独立，并通过清晰的通信方式来与系统中其他部分进行协作。有时你需要使用一些作为对象间沟通渠道的类，此时就不得不引入依赖关系。

假设有一个用于保存应用程序信息的Preferences类。我们可能会使用一个Preferences对象来保存诸如DSN（用于保存数据库的表及用户信息）字符串，URL根目录、文件路径等数据。这些信息在你每次部署程序时都可能会有所不同。该对象也可被用作一个“公告板”，它是可以被系统中其他无关对象设置和获取消息的中心。

但在对象中传递Preferences对象并不总是个好主意。你可以让原来并不使用Preferences

对象的类强制性地接受Preferences对象，以便这些类能传递Preferences对象给其他对象。但这样做产生了另一种形式的耦合。

我们还需要保证系统中的所有对象都使用同一个Preferences对象。我们不希望一些对象在一个Preferences对象上设值，而其他对象从另外一个完全不同的Preferences对象上读取数据。

让我们提炼出这个问题的几个关键点：

- Preferences对象应该可以被系统中的任何对象使用。
- Preferences对象不应该被储存在会被覆写的全局变量中。
- 系统中不应超过一个Preferences对象。也就是说，Y对象可设置Preferences对象的一个属性，而Z对象不需要通过其他对象（假设Y和Z都可以访问Preferences对象）就可以直接获得该属性的值。

9.2.2 实现

为了解决这个问题，我们可以从强制控制对象的实例化开始。下面创建了一个无法从其自身外部来创建实例的类。这听起来似乎有些难，其实只要简单地定义一个私有的构造方法即可：

```
class Preferences {
    private $props = array();

    private function __construct() { }

    public function setProperty( $key, $val ) {
        $this->props[$key] = $val;
    }

    public function getProperty( $key ) {
        return $this->props[$key];
    }
}
```

当然，目前Preferences类是完全不能用的。我们设置了一个不合常理的访问限制。由于构造方法被声明为private，客户端代码无法实例化对象。因此setProperty()和getProperty()方法目前也是多余的。

不过我们可以使用静态方法和静态属性来间接实例化对象。

```
class Preferences {
    private $props = array();
    private static $instance;

    private function __construct() { }

    public static function getInstance() {
        if ( empty( self::$instance ) ) {
            self::$instance = new Preferences();
        }
        return self::$instance;
```

```
    }

    public function setProperty( $key, $val ) {
        $this->props[$key] = $val;
    }

    public function getProperty( $key ) {
        return $this->props[$key];
    }
}
```

$instance属性设置为private及static，因此不能在类外部被访问。而getInstance()方法在类内部，因此可以访问$instance属性。因为getInstance()方法是public且static的，所以在脚本的任何地方都可被调用。

```
$pref = Preferences::getInstance();
$pref->setProperty( "name", "matt" );

unset( $pref ); // 移除引用

$pref2 = Preferences::getInstance();
print $pref2->getProperty( "name" ) ."\n"; // 该属性值并没有丢失
```

输出结果正是我们最早为Preferences对象写入的值，此时仍可访问。

```
matt
```

静态方法不能访问普通的对象属性，因为根据静态的定义，它只能被类而不是对象调用。但静态方法可以访问一个静态属性。所以当getInstance()被调用时，我们会检查Preferences::$instance属性。如果为空，那么创建一个Preferences对象实例并把它保存在$instance属性中，然后我们把实例返回给调用代码。因为静态方法getInstance()是Prefrences类的一部分，所以尽管构造方法是私有的，但是实例化Preferences对象完全没有问题。

图9-2展示了单例模式。

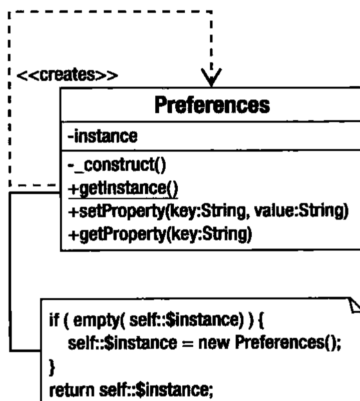


图9-2 单例模式示例

9.2.3 结果

那么使用单例对象与使用全局变量相比，又如何呢？首先是坏的一面。单例和全局变量都可能被误用。因为单例在系统任何地方都可以被访问，所以它们可能会导致很难调试的依赖关系。如果改变一个单例，那么所有使用该单例的类可能都会受到影响。在这里，依赖本身并不是问题。毕竟，我们在每次声明一个有特定类型参数的方法时，也就创建了依赖关系。问题是，单例对象的全局化的性质会使程序员绕过类接口定义的通信线路。当单例被使用时，依赖便会被隐藏在方法内部，而并不会出现在方法声明中。这使得系统中的依赖关系更加难以追踪，因此需要谨慎小心地部署单例类。

然而，我认为适度地使用单例模式可以改进系统的设计。在系统中传递那些不必要的对象令人厌烦，而单例可以让你从中解放出来。

在面向对象的开发环境中，单例模式是一种对于全局变量的改进。你无法用错误类型的数据覆写一个单例。这种保护在不支持命名空间的PHP版本里尤其重要。因为在PHP中命名冲突会在编译时被捕获，并使脚本停止运行。

9.3 工厂方法模式

面向对象设计强调"抽象类高于实现"。也就是说，我们要尽量一般化而不是特殊化。工厂方法模式解决了当代码关注于抽象类型时如何创建对象实例的问题。答案便是用特定的类来处理实例化。

9.3.1 问题

假设有一个关于个人事务管理的项目，其功能之一是管理Appointment（预约）对象。我们的业务团队和另一个公司建立了关系，目前需要用一个叫做BloggsCal的格式来和他们交流预约相关的数据。但是业务团队提醒我们将来可能要面对更多的数据格式。

在接口级别上，我们可以立即定义两个类。其一是需要一个数据编码器来把Appointment对象转换成一个专有的格式，将这个编码器命名为ApptEncoder类；另外需要一个管理员类来获得编码器，并使用编码器与第三方进行通信，我们将这个管理类命名为CommsManager类。使用模式术语来描述的话，CommsManager便是创建者（creator），而ApptEncoder是产品（product）。你可以在图9-3中看到这个结构。

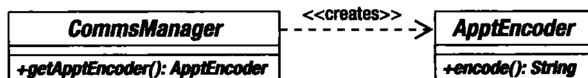


图9-3 抽象的创建者类和产品类

但是如何得到一个具体的ApptEncoder对象？

我们可以要求传递ApptEncoder给CommsManager，但这只是延缓了问题，而我们希望问题在此可以彻底解决。因此，先在CommsManager类中直接实例化BloggsApptEncoder对象。

```
abstract class ApptEncoder {
    abstract function encode();
}

class BloggsApptEncoder extends ApptEncoder {
    function encode() {
        return "Appointment data encoded in BloggsCal format\n";
    }
}

class MegaApptEncoder extends ApptEncoder {
    function encode() {
        return "Appointment data encoded in MegaCal format\n";
    }
}
class CommsManager {
    function getApptEncoder() {
        return new BloggsApptEncoder();
    }
}
```

CommsManager类负责生成BloggsApptEncoder对象。当团队合作关系发生改变，而我们被要求转换系统来使用一个新的格式MegaCal时，可以只添加一个条件语句到CommsManager::getApptEncoder()方法。毕竟，这是我们以前用过的策略。下面来建立一个新的可以同时处理BloggsCal和MegaCal格式的CommsManager的实现。

```
class CommsManager {
    const BLOGGS = 1;
    const MEGA = 2;
    private $mode = 1;

    function __construct( $mode ) {
        $this->mode = $mode;
    }

    function getApptEncoder() {
        switch ( $this->mode ) {
            case ( self::MEGA ):
                return new MegaApptEncoder();
            default:
                return new BloggsApptEncoder();
        }
    }
}

$comms = new CommsManager( CommsManager::MEGA );
$apptEncoder = $comms->getApptEncoder();
print $apptEncoder->encode();
```

在类中我们使用常量标志定义了脚本可能运行的两个模式：MEGA和BLOGGS。在getApptEncoder()方法中使用switch语句来检查$mode属性，并实例化适当的ApptEncoder。

这种方式仍有一点小小的缺陷。通常情况下，创建对象的确需要指定条件，但有时候条件语

句会被当做坏的“代码味道”的象征。如果重复的条件语句蔓延在代码中，我们不应该感到乐观。CommsManager类已经能够提供交流日历数据的功能。但是，假设我们所使用的协议要求提供页眉和页脚数据来描述每次预约，情况又将如何呢？下面扩展之前的例子来支持getHeaderText()方法。

```
class CommsManager {
    const BLOGGS = 1;
    const MEGA = 2;
    private $mode ;

    function __construct( $mode ) {
        $this->mode = $mode;
    }

    function getHeaderText() {
        switch ( $this->mode ) {
            case ( self::MEGA ):
                return "MegaCal header\n";
            default:
                return "BloggsCal header\n";
        }
    }
    function getApptEncoder() {
        switch ( $this->mode ) {
            case ( self::MEGA ):
                return new MegaApptEncoder();
            default:
                return new BloggsApptEncoder();
        }
    }
}
```

正如你看到的，支持页眉输出的需求迫使我们重复条件判断语句。如果我们加入新协议，这种做法就变得不适用了。如果还需要加入getFooterText()方法，工作量就会更大。

那么，总结一下我们的问题：

- 在代码运行时我们才知道要生成的对象类型（BloggsApptEncoder或者MegaApptEncoder）；
- 我们需要能够相对轻松地加入一些新的产品类型（如一种新业务处理方式SyncML）；
- 每一个产品类型都可定制特定的功能（如getHeaderText()和getFooterText()）。

另外，请注意我们正在使用条件语句，并且之前已介绍过这些条件语句可以被多态代替，而工厂方法模式恰好能让我们用继承和多态来封装具体产品的创建。换句话说，我们要为每种协议创建CommsManager的一个子类，而每一个子类都要实现getApptEncoder()方法。

9.3.2 实现

工厂方法模式把创建者类与要生产的产品类分离开来。创建者是一个工厂类，其中定义了用于生成产品对象的类方法。如果没有提供默认实现，那么就由创建者类的子类来执行实例化。一

般情况下，就是创建者类的每个子类实例化一个相应产品子类。

把CommsManager重新指定为抽象类。这样我们可以得到一个灵活的父类，并把所有特定协议相关的代码放到具体的子类中。你可以在图9-4中看到这个改变。

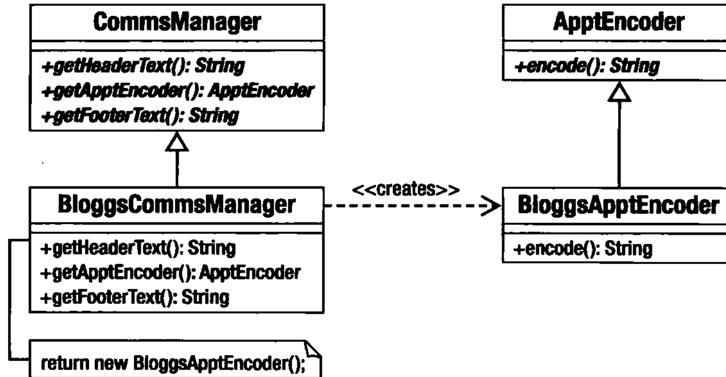


图9-4　具体的创建者类和产品类

下面是简化过的代码：

```
abstract class ApptEncoder {
    abstract function encode();
}

class BloggsApptEncoder extends ApptEncoder {
    function encode() {
        return "Appointment data encode in BloggsCal format\n";
    }
}

abstract class CommsManager {
    abstract function getHeaderText();
    abstract function getApptEncoder();
    abstract function getFooterText();
}

class BloggsCommsManager extends CommsManager {
    function getHeaderText() {
        return "BloggsCal header\n";
    }

    function getApptEncoder() {
        return new BloggsApptEncoder();
    }

    function getFooterText() {
        return "BloggsCal footer\n";
    }
}
```

BloggsCommsManager::getApptEncoder()方法返回一个BloggsApptEncoder对象。调用getApptEncoder()的客户端代码可以预期得到一个ApptEncoder类型的对象，并且不需要知道被给予的是什么具体产品。在一些编程语言中，方法的返回类型是强制规定的，因此调用诸如getApptEncoder()之类方法的客户端代码可以完全确定它将接收到的是ApptEncoder对象。而在PHP中，这只能靠人为约定。因此，在代码中注明返回类型或者通过命名约定来传达返回类型的信息相当重要。

注解 *在写作本书时，检查返回类型是在PHP未来版本中被提议实现的特性。*

现在当我们被要求实现MegaCal时，只需要给CommsManager抽象类写一个新实现。图9-5展示了MegaCal类。

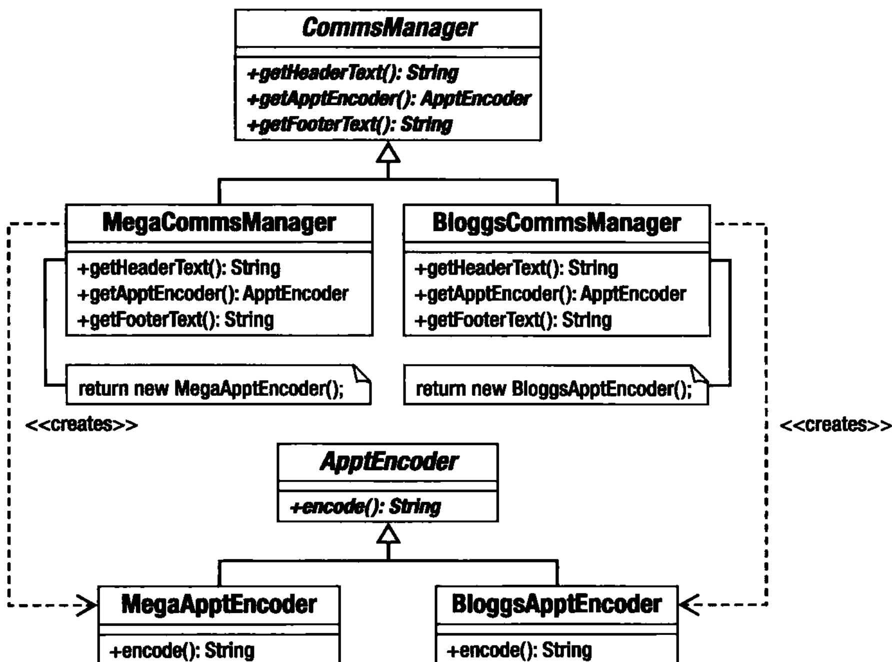


图9-5 扩展设计以支持新的协议

9.3.3 结果

请注意我们的创建者类与产品的层次结构是非常相似的。这是使用工厂方法模式的常见结果，这形成了一种特殊的代码重复，因而被一些人所不喜欢。另一个问题是该模式可能会导致不必要的子类化。如果你为创建者类创建子类的原因只是为了实现工厂方法模式，那么最好再考虑一下（这就是为什么在例子中引进页眉和页脚）。

在示例中我们只关注“预约”功能。如果想扩展一下功能，使其能够处理“待办事宜”及联

系人，那么将会遇到新问题。我们需要一个可以同时处理一组相关实现的架构。正如我们在下一节中将看到的，工厂方法模式经常和抽象工厂模式一起使用。

9.4 抽象工厂模式

在大型应用中，我们很可能需要工厂来产生一组相关的类。抽象工厂模式可解决这个问题。

9.4.1 问题

再来看一下之前实现的个人事务管理的示例。我们以BloggsCal和MegaCal这两种格式管理编码。我们可以通过加入更多编码格式来使此结构“横向”增长，但如何通过给不同类型的PIM对象加入编码器使它“纵向”增长呢？事实上，我们一直在应用该模式。

在图9-6中，可以看到我们使用的相似的类层次结构，分别是预约（`Appt`）、待办事宜（`Ttd`）和联系人（`Contact`）。

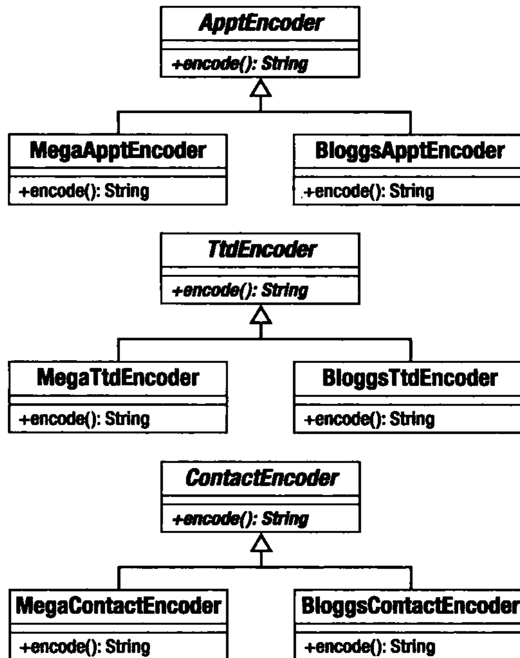


图9-6 产品家族

BloggsCal的类与其他格式（如MegaCal）互相没有关联（尽管它们实现了一个公共的接口），但它们的功能相似。所以如果系统目前正在使用`BloggsTtdEncoder`，那么它应该也能使用`BloggsContactEncoder`。

看一下如何做到这一点。我们仍从接口开始着手，就如在工厂方法模式中所做的那样（见图9-7）。

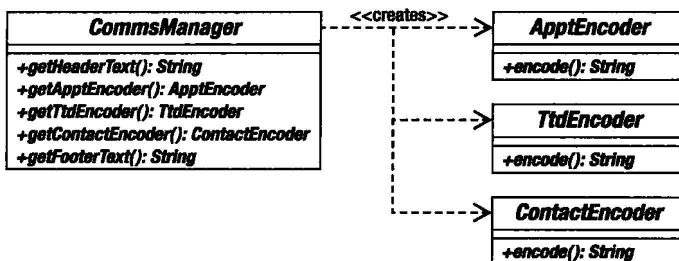


图9-7　抽象创建者及其抽象产品

9.4.2　实现

CommsManager抽象类定义了用于生成3个产品（ApptEncoder、TtdEncoder和Contact-Encoder）的接口。我们需要先实现一个具体的创建者，然后才能创建一个特定类型的具体产品。图9-8创建了BloggsCal格式的创建者。

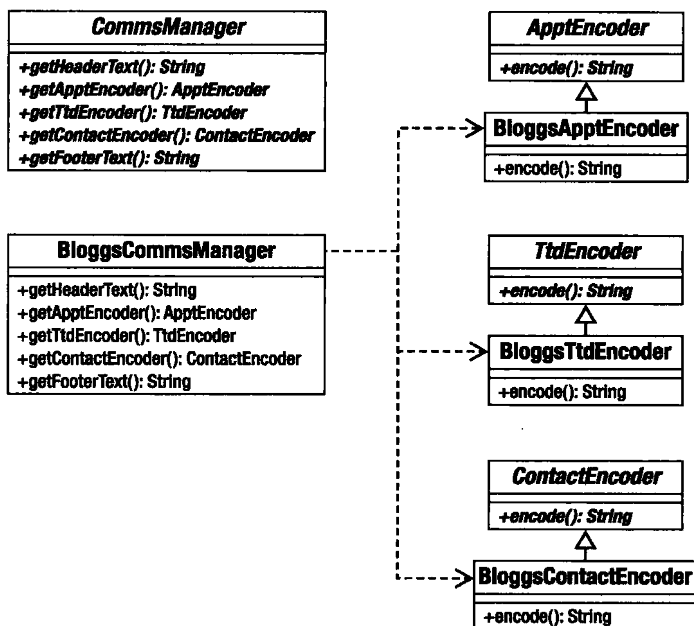


图9-8　增加具体创建者和一些具体产品

下面是CommsManager和BloggsCommsManager的代码：

```
abstract class CommsManager {
    abstract function getHeaderText();
    abstract function getApptEncoder();
    abstract function getTtdEncoder();
    abstract function getContactEncoder();
    abstract function getFooterText();
```

```
}

class BloggsCommsManager extends CommsManager {
    function getHeaderText() {
        return "BloggsCal header\n";
    }

    function getApptEncoder() {
        return new BloggsApptEncoder();
    }

    function getTtdEncoder() {
        return new BloggsTtdEncoder();
    }

    function getContactEncoder() {
        return new BloggsContactEncoder();
    }

    function getFooterText() {
        return "BloggsCal footer\n";
    }
}
```

请注意，在这个例子中使用了工厂方法模式。getContact()是CommsManager的抽象方法，并在BloggsCommsManager中被实现。设计模式间经常会这样协作：一个模式创建可以把它自己引入到另一个模式的上下文环境。在图9-9中，我们加入了对MegaCal格式的支持。

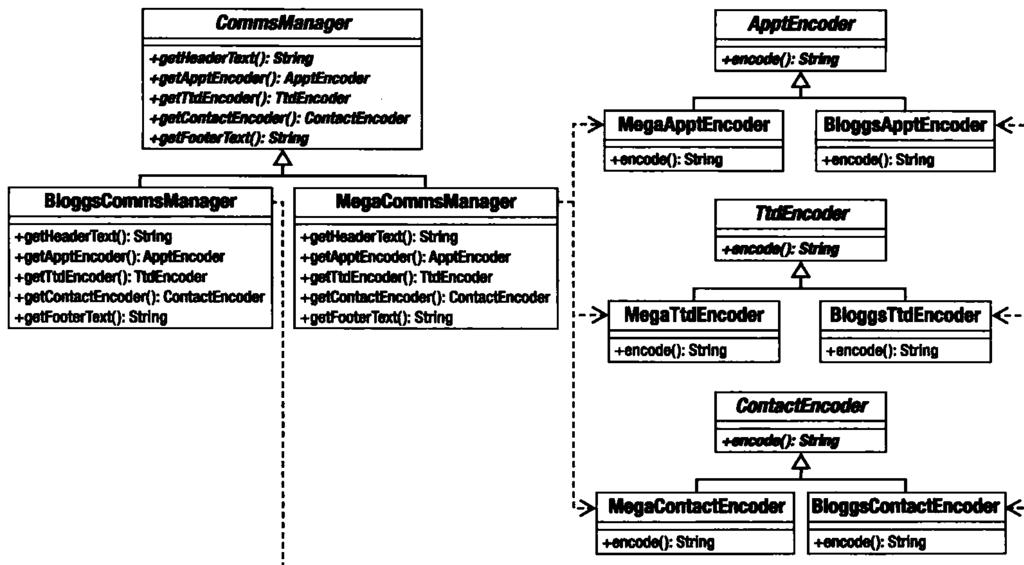


图9-9 添加具体创建者和一些具体产品

9.4.3 结果

那么这个模式带来了什么？

- 首先，将系统与实现的细节分离开来。我们可以在示例中添加或移除任意数目的编码格式而不会影响系统。
- 对系统中功能相关的元素强制进行组合。因此，通过使用BloggsCommsManager，可以确保只使用与BloggsCal相关的类。
- 添加新产品将会令人苦恼。因为不仅要创建新产品的具体实现，而且为了支持它，我们必须修改抽象创建者和它的每一个具体实现。

抽象工厂模式的许多实现都会使用工厂方法模式，这可能是因为大多数范例都是用Java或者C++写的。但是PHP并不会强制规定方法的返回类型，这给了我们一些可能利用的灵活性。

我们可以创建一个使用标志参数来决定返回什么对象的单一的make()方法，而不用给每个工厂方法创建独立的方法。

```
abstract class CommsManager {
    const APPT    = 1;
    const TTD     = 2;
    const CONTACT = 3;
    abstract function getHeaderText();
    abstract function make( $flag_int );
    abstract function getFooterText();
}

class BloggsCommsManager extends CommsManager {
    function getHeaderText() {
        return "BloggsCal header\n";
    }
    function make( $flag_int ) {
        switch ( $flag_int ) {
            case self::APPT:
                return new BloggsApptEncoder();
            case self::CONTACT:
                return new BloggsContactEncoder();
            case self::TTD:
                return new BloggsTtdEncoder();
        }
    }

    function getFooterText() {
        return "BloggsCal footer\n";
    }
}
```

可以看出，类的接口变得更加紧凑。但是这样做也有一定代价。在使用工厂方法时，我们定义了一个清晰的接口并强制所有具体的工厂对象遵循它。而使用单一的make()方法，我们必须在所有的具体创建者中支持所有的产品对象。同时，我们也引入了平行条件，因为每个具体创建者都必须实现相同的标志检测（flag test）。客户类[①]无法确定具体的创建者是否可以生成所有产品，因为make()方法的内部需要对每种情况都进行考虑并作出选择。

另一方面，我们可以创建更灵活的创建者。创建者基类可以提供make()方法来保证每个产品家族有一个默认实现。具体子类可以选择性地改变这一行为。子类可以实现自己的make()方法并调用，由此决定生成何种对象。这些创建者子类可以自行决定是否在执行自己的make()方

① 指调用创建者类的类。——译者注

法后调用父类的make()方法。

我们会在下一节中介绍抽象工厂模式的另一个变形。

9.4.4　原型模式

平行继承层次的出现是工厂方法模式带来的一个问题。这是一种让一些程序员不舒服的耦合。每次添加产品家族时，你就被迫去创建一个相关的具体创建者（比如编码器BloggsCal和BloggsCommsManager匹配）。在一个快速增长的系统里会包含越来越多的产品，而维护这种关系便会很快令人厌烦。

一个避免这种依赖的办法是使用PHP的clone关键词复制已存在的具体产品。然后，具体产品类本身便成为它们自己生成的基础。这便是原型模式。使用该模式我们可以用组合代替继承。这样的转变则促进了代码运行时的灵活性，并减少了必须创建的类。

9.4.5　问题

假设有一款"文明"（Civilization）风格的网络游戏，可在区块组成的格子中操作战斗单元（unit）。每个区块分别代表海洋、平原和森林。地形的类别约束了占有区块的单元的移动和格斗能力。我们可以有一个TerrainFactory对象来提供Sea、Forest和Plains对象，我们决定允许用户在完全不同的环境里选择，于是Sea可能是MarsSea和EarthSea的抽象父类。Forest（森林）和Plains（平原）对象也会有相似的实现。这里的分支便构成了抽象工厂模式。我们有截然不同的产品体系（Sea、Plains、Forest），而这些产品家族间有超越继承的紧密联系，如Earth（地球）和Mars（火星），它们都同时存在海洋、森林和平原地形。图9-10所示的类图展示了如何对这些产品应用抽象工厂和工厂方法模式。

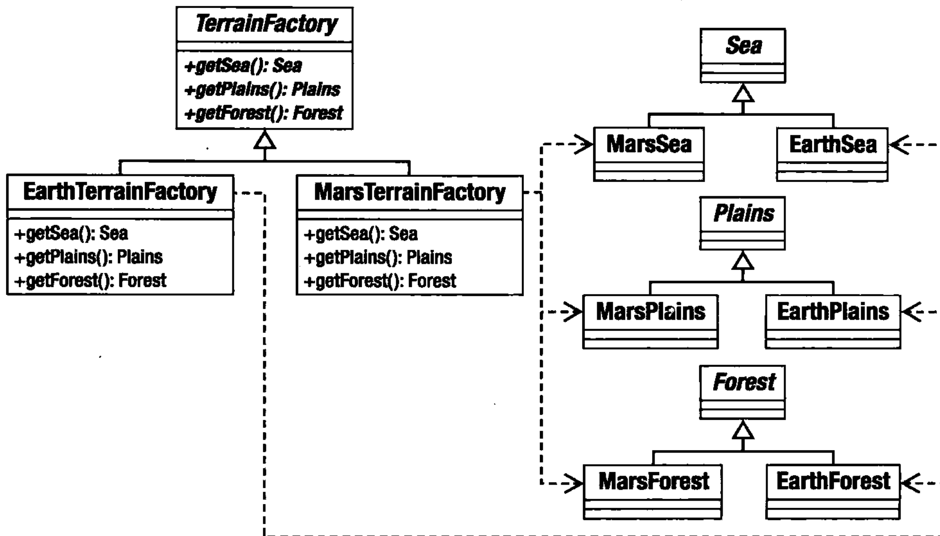


图9-10　通过抽象工厂方法处理地形

你可以看到，我们依赖继承来组合工厂生成的terrain（地形）家族产品。这的确是一个可行的解决方案，但这需要有一个大型的继承体系，并且相对来说不那么灵活。当你不想要平行的继承体系而需要最大化运行时的灵活性时，可以使用抽象工厂模式的强大变形——原型模式。

9.4.6 实现

当使用抽象工厂模式或工厂方法模式时，必须决定使用哪个具体的创建者，这很可能是通过检查配置的值来决定的。既然必须这样做，那为什么不简单地创建一个保存具体产品的工厂类，并在初始化时就加入这种做法呢？这样除了可以减少类的数量，还有其他好处。下面是在工厂中使用原型模式的简单代码：

```
class Sea {}
class EarthSea extends Sea {}
class MarsSea extends Sea {}

class Plains {}
class EarthPlains extends Plains {}
class MarsPlains extends Plains {}

class Forest {}
class EarthForest extends Forest {}
class MarsForest extends Forest {}

class TerrainFactory {
    private $sea;
    private $forest;
    private $plains;

    function __construct( Sea $sea, Plains $plains, Forest $forest ) {
        $this->sea = $sea;
        $this->plains = $plains;
        $this->forest = $forest;
    }

    function getSea( ) {
        return clone $this->sea;
    }

    function getPlains( ) {
        return clone $this->plains;
    }

    function getForest( ) {
        return clone $this->forest;
    }
}

$factory = new TerrainFactory( new EarthSea(),
    new EarthPlains(),
    new EarthForest() );
print_r( $factory->getSea() );
```

```
print_r( $factory->getPlains() );
print_r( $factory->getForest() );
```

可以看到，我们加载了一个带有产品对象实例的具体的TerrainFactory对象。当客户端代码调用getSea()时，返回在初始化时缓存的Sea对象的一个副本。我们不仅仅省掉了一些类，而且有了额外的灵活性。想要在一个有类似地球海洋和森林并有类似火星平原的星球上玩游戏吗？你没有必要写一个新的创建者类——只需简单改变添加到TerrainFactory的参数即可。

```
$factory = new TerrainFactory( new EarthSea(),
    new MarsPlains(),
    new EarthForest() );
```

因此原型模式使我们可利用组合所提供的灵活性，不过我们得到的可不止这个。因为在运行时保存和克隆对象，所以当生成新产品时，可以重新设定对象状态。设想Sea对象有$navigability属性（可导航性），该属性影响船在海洋区块的移动能量值，并可以被设置来调整游戏的难度等级。

```
class Sea {
    private $navigability = 0;
    function __construct( $navigability ) {
        $this->navigability = $navigability;
    }
}
```

现在当我们初始化TerrainFactory对象时，可以加入一个具有导航修改仪的Sea对象，而这对TerrainFactory处理的所有Sea对象都有效。

```
$factory = new TerrainFactory( new EarthSea( -1 ),
    new EarthPlains(),
    new EarthForest() );
```

如果你想生成的对象是由其他对象组合而成的，以上代码显然颇具灵活性。假设所有的Sea对象都能包含Resource对象（FishResource、OilResource等）。通过配置的值，可以给所有的Sea对象默认提供一个FishResouce对象。要记住的是如果产品对象引用了其他对象，那么你应该实现__clone()方法来保证你得到的是深复制（deep copy）。

注解 我们在第4章中介绍了对象克隆。clone关键字能给应用的对象生成一个浅复制（shallow copy）。也就是说，产品对象会具有和源对象一样的属性。如果源对象的任何属性是对象，那么这些对象属性不会被直接复制到产品中，相反产品会引用同样的对象属性。通过实现__clone()方法，可以改变克隆的默认行为，并定制对象复制的方式，而这在clone关键词被使用时会被自动调用。

```
class Contained { }

class Container {
    public $contained;
```

```
    function __construct() {
        $this->contained = new Contained();
    }

    function __clone() {
        // 确保被克隆的对象持有的是self::$contained的克隆而不是引用
        $this->contained = clone $this->contained;
    }
}
```

9.5 某些模式的骗术

我保证过这一章主要介绍如何创建对象，不会列举许多与面向对象无关的例子。然而本章的一些模式很狡猾地避过了对象生成相关的决策过程，看起来似乎与对象生成无关——如果并不是创建本身的话。

单例模式不在此列，因为单例模式中的对象创建逻辑是内建并明确的。抽象工厂模式产品家族的创建则分布到了不同的具体创建者中。那么我们如何选择具体的创建者呢？原型模式把相似的问题呈现在我们面前。这两个模式都用于创建对象，但它们延缓了应该创建哪个对象或者哪组对象的决定。

系统经常根据配置的值来决定选择哪个特定的具体创建者。而这些配置信息可以在数据库、配置文件或服务器文件中（如Apache服务器的配置文件通常是.htaccess），或者干脆以PHP变量或属性进行硬编码。因为PHP应用程序必须为每个请求重新配置，所以需要让脚本的初始化尽可能地简单。正因为此，我经常选择在PHP代码中通过硬编码来设定配置标志。这可以通过手写或者写一个自动生成类文件的脚本来完成。下面是一个包含日历协议类型的标记的类：

```
class Settings {
    static $COMMSTYPE = 'Bloggs';
}
```

有了这个标志（尽管并不优雅），就能创建一个类，它利用该标志决定用哪个CommsManager来处理请求。这里单例与抽象工厂模式的结合使用是相当常见的，代码如下：

```
require_once( 'Settings.php' );

class AppConfig {
    private static $instance;
    private $commsManager;

    private function __construct() {
        // 仅运行一次
        $this->init();
    }

    private function init() {
        switch ( Settings::$COMMSTYPE ) {
            case 'Mega':
                $this->commsManager = new MegaCommsManager();
```

```
                break;
            default:
                $this->commsManager = new BloggsCommsManager();
        }
    }

    public static function getInstance() {
        if ( empty( self::$instance ) ) {
            self::$instance = new self();
        }
        return self::$instance;
    }

    public function getCommsManager() {
        return $this->commsManager;
    }
}
```

AppConfig类是一个标准的单例。因此，我们可以在系统中任何地方得到AppConfig实例，并且确保得到的一直是同一个实例。init()方法会被类的构造方法调用，且因此在一个进程中只会运行一次。它检查Settings::$COMMSTYPE属性，并根据该属性的值来实例化一个具体的CommsManager对象。现在我们的代码可以得到并使用一个CommsManager对象，而不需要知道它的具体实现或者它所生成的具体类。

```
$commsMgr = AppConfig::getInstance()->getCommsManager();
$commsMgr->getApptEncoder()->encode();
```

9.6 小结

本章介绍了一些用于生成对象的窍门，讨论了支持全局访问实例的单例模式，以及应用多态原则生成对象的工厂方法模式。之后，我们结合使用工厂方法模式和抽象工厂模式来生成一系列创建者类，用于实例化一组相关对象。最后，研究了原型模式，学习了如何使用对象克隆使对象组合可用于生成对象。

第 10 章

让面向对象编程更加灵活的模式

第9章讨论了创建对象的策略，本章将研究构建类和对象的策略。我们将特别关注“组合可比继承提供更多的灵活性”（composition provides greater flexibility than inheritance）这一原则。本章中研究的模式依然来自于《设计模式》一书。

本章包括以下内容。

- 组合（Composite）模式：将一组对象组合为可像单个对象一样被使用的结构。
- 装饰（Decorator）模式：通过在运行时合并对象来扩展功能的一种灵活机制。
- 外观（Facade）模式：为复杂或多变的系统创建一个简单的接口。

10.1 构造可灵活创建对象的类

在第4章中，我曾说过初学者常常对对象和类感到困惑。这也不全对。事实上，即使不是初学者，有时也会大伤脑筋，比如我们会尝试让UML类图中描述的静态继承结构与这些类的动态对象的关系保持一致，有时这个目标并不容易实现。

你还记得“组合优于继承”这个模式原则吗？该原则体现了类和对象之间的组织弹性。为了使项目更具灵活性，我们需要将类按一定结构组织起来，以便它们的对象在代码运行时能被构建为有用的结构。

这是本章前两个模式的共同主题。继承在这两个模式中很重要，但是组合所提供的描述结构及扩展功能的机制也很重要。

10.2 组合模式

组合模式也许是将继承用于组合对象的最极端的例子。组合模式的设计思想简单而又非常优雅，并且非常实用。但要注意的是，因为这个模式很好用，你也许会经不起诱惑而过度使用它。

组合模式可以很好地聚合和管理许多相似的对象，因而对客户端代码来说，一个独立对象和一个对象集合是没有差别的。虽然该模式实际上非常简单，但还是经常令人感到迷惑。其中一个

原因就是模式中类的结构和对象的组织结构非常相似。组合模式中继承的层级结构是树型结构，由根部的基类开始，分支到各个不同的子类，在其继承树中可以轻松地生成枝叶，也可以很容易地遍历整个对象树中的对象。

如果你以前不熟悉这个模式，也许会感到有些困惑。让我们举例说明，单个物体有时可以被当成一个集合来看待。例如，用谷类和肉类（或者是你更加喜欢的大豆）可以制作一种食物——香肠。然后我们将香肠当做单个物体来处理。就像我们可以食用、烹调、购买或者出售肉类一样，我们也可以食用、烹调、购买或者出售由肉类组成的香肠。我们也许会用香肠和其他组合成分来制作馅饼，从而将一个组合物合成一个更大的组合物。在这个例子中，我们处理组件的方式和处理集合的方式是一样的。组合模式有助于我们为集合和组件之间的关系建立模型。

10.2.1　问题

管理一组对象是很复杂的，当对象中可能还包含着它们自己的对象时尤其如此。这类问题在开发中非常常见。比如发票中会包含描述产品或服务的条目，或者待办事宜列表会包含多个子任务。在CMS（Content Management System，内容管理系统）中，我们无法离开页面、文章、多媒体组件等。从外部来管理这些结构相当困难。

让我们回到之前虚构的一个场景，我们以一个叫做“文明”的游戏为基础设计一个系统。玩家可以在一个由大量区块所组成的地图上移动战斗单元。独立的单元可被组合起来一起移动、战斗和防守。我们先定义一些战斗单元的类型。

```
abstract class Unit {
    abstract function bombardStrength();
}

class Archer extends Unit {
    function bombardStrength() {
        return 4;
    }
}

class LaserCannonUnit extends Unit {
    function bombardStrength() {
        return 44;
    }
}
```

Unit类定义了一个抽象方法bombardStrength()，用于设置战斗单元对邻近区块的攻击强度，然后我们在Archer（射手）和LaserCannonUnit（激光炮）这两个类中都实现了bombard-Strength()方法。当然，这3个类也应该包含移动能力和防御能力等内容，但为了简单起见，我们暂时先省略这些功能。现在我们可以这样定义一个独立的类来组合战斗单元：

```
class Army {
    private $units = array();

    function addUnit( Unit $unit ) {
        array_push( $this->units, $unit );
```

```
    }

    function bombardStrength() {
        $ret = 0;
        foreach( $this->units as $unit ) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}
```

Army（军队）类有一个addUnit()方法，用于接受Unit对象。Unit对象被保存在数组属性$units中。同时我们在Army的bombardStrength()方法中通过一个简单的迭代遍历所聚合的Unit对象并调用bombardStrength()方法，计算出总的攻击强度。

如果问题一直如此简单，那么这样的模型还是非常令人满意的。但是当我们加入一些新的需求，会发生什么呢？比如军队应该可以与其他军队进行合并。同时，每个军队都会有它自己的ID，这样军队以后还能从整编中解散出来。比如今天大公爵（ArchDuke）和Soames将军的勇士们可能一起并肩作战，但是当国内发生叛乱时，公爵可能会在任何时候将它的军队调回。因此，我们不能仅支持将军队与军队合并，还要能够在必要的时候再把原来的军队抽离出来。

我们可以修改Army类，使之可以像添加Unit对象一样添加Army对象：

```
function addArmy( Army $army ) {
    array_push( $this->armies, $army );
}
```

同时，我们需要修改bombardStrength()方法，遍历所有的军队（army）及部队单位（unit）：

```
function bombardStrength() {
    $ret = 0;
    foreach( $this->units as $unit ) {
        $ret += $unit->bombardStrength();
    }

    foreach( $this->armies as $army ) {
        $ret += $army->bombardStrength();
    }

    return $ret;
}
```

此时，这个新的需求（即添加和抽取军队）还不算太复杂。但是记住，我们还需要对defensiveStrength()、movementRange()等方法做相似的修改。慢慢地，我们的游戏会变得越来越完善。现在客户提出了新要求：运兵船可以支持最多10个战斗单元以改进它们在某些地形上的活动范围。显然，运兵船与用于组合战斗单元的Army对象相似，但它也有一些自己的特性。虽然我们能够通过改进Army类来处理TroopCarrier对象，但是我们知道以后可能会有更多对部队进行组合的需求。显然，我们需要一个更灵活的模型。

让我们再回顾一下已经建立的这个模型。所有已创建的类都需要bombardStrength()方法。

实际上，客户端代码并不需要去区分对象是Army、Unit还是TroopCarrier类型。就功能上说，它们都是相同的：它们都要移动、攻击和防御。这些包含其他对象的对象需要提供添加和移除其他对象的方法。这些相似性会给我们带来一个必然的结论：因为容器对象与它们包含的对象共享同一个接口，所以它们应该共享同一个类型家族。

10.2.2　实现

组合模式定义了一个单根继承体系，使具有截然不同职责的集合可以并肩工作。在之前的例子中我们已看到以上两点。组合模式中的类必须支持一个共同的操作集，以将其作为它们的首要职责。对我们来说，bombardStrength()方法便支持这样的操作。类同时必须拥有添加和删除子对象的方法。

图10-1中的类图展示了应用于“文明”游戏问题的组合模式。

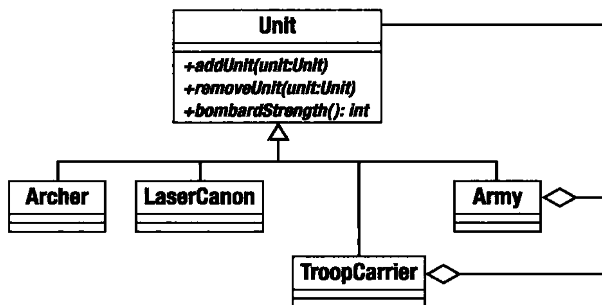


图10-1　组合模式

可以看到，模型中的所有部队单位都扩展自Unit类。然后，客户端代码可以肯定任何Unit对象都支持bombardStrength()方法，因此完全可以像处理Archer对象那样处理Army对象。

Army和TroopCarrier类都被设计为组合对象，用于包含Unit对象。Archer和LaserCannon类则是局部对象[①]，具有部队单位的操作功能，但不能包含其他Unit对象。这里有个问题，就是局部对象是否需要遵循与图10-1所示的组合对象同样的接口。图10-1中的TroopCarrier和Army实现了与其他作战单元的组合，而即使在局部类中，也实现了addUnit()方法。我们稍后将讨论这个问题。下面是重写过的Unit抽象类：

```
abstract class Unit {
    abstract function addUnit( Unit $unit );
    abstract function removeUnit( Unit $unit );
    abstract function bombardStrength();
}
```

可以看到，我们为所有的Unit对象设计了基本功能。现在看看一个组合对象是如何实现这些抽象方法的：

① 也称为树叶对象。称为树叶是因为组合模式为树型结构，组合对象为枝干，单独存在的对象为树叶。树叶对象应该是最小单位，其中不能包含其他对象。——译者注

```
class Army extends Unit {
    private $units = array();

        function addUnit( Unit $unit ) {
         if ( in_array( $unit, $this->units, true ) ) {
             return;
         }
         $this->units[] = $unit;
    }

    function removeUnit( Unit $unit ) {

        $this->units = array_udiff( $this->units, array( $unit ),

                        function( $a, $b ) { return ($a === $b)?0:1; } );

    }


    function bombardStrength() {
        $ret = 0;
        foreach( $this->units as $unit ) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}
```

在我们将Unit对象加入$units数组属性前，addUnit()方法会先检查是否已经添加过同一个Unit对象，而removeUnit()也会使用类似的检查在属性中移除指定的Unit对象。

注解 我在removeUnit()方法中使用了一个匿名回调函数。该函数会检查数组元素的$units属性，以检查数组元素的相等性。匿名函数是在PHP 5.3中引入的，如果你运行旧版的PHP，使用create_function()方法可以达到相似的效果：

```
$this->units = array_udiff( $this->units, array( $unit ),
                create_function( '$a,$b', 'return ($a === $b)?0:1;' ) );
```

然后Army对象可保存任何类型的Unit对象，包括Army对象本身或者如Archer或Laser-CannonUnit这样的局部对象。因为所有的部队单位都保证支持bombardStrength()方法，所以我们的Army::bombardStrength()方法只需遍历$units属性，调用每个Unit对象的bombard-Strength()方法，就可以计算出整支军队总的攻击强度。

组合模式的一个问题是如何实现add和remove方法。一般的组合模式会在抽象超类中添加add和remove方法。这可以确保模式中的所有类都共享同一个接口，但这同时也意味着局部类必须也实现这些方法：

```
class UnitException extends Exception {}

class Archer extends Unit {
```

10

```
    function addUnit( Unit $unit ) {
        throw new UnitException( get_class($this)." is a leaf" );
    }

    function removeUnit( Unit $unit ) {
        throw new UnitException( get_class($this)." is a leaf" );
    }

    function bombardStrength() {
        return 4;
    }
}
```

实际上，我们并不希望在Archer对象中添加Unit对象[①]，所以在addUnit或removeUnit方法被调用时会抛出异常。这要求修改所有局部类的add/remove方法，因此我们可以修改之前设计的代码，在Unit类的addUnit()/removeUnit()方法中抛出异常。

```
abstract class Unit {
    abstract function bombardStrength();

    function addUnit( Unit $unit ) {
        throw new UnitException( get_class($this)." is a leaf" );
    }

    function removeUnit( Unit $unit ) {
        throw new UnitException( get_class($this)." is a leaf" );
    }
}

class Archer extends Unit {
    function bombardStrength() {
        return 4;
    }
}
```

这样做可以去除局部类中的重复代码，但是同时组合类不再需要强制性地实现addUnit()和removeUnit()方法了，这可能会带来问题。

在下一节中，我们会深入讨论组合模式所带来的一些问题。现在先让我们回顾一下组合模式所带来的益处。

- 灵活：因为组合模式中的一切类都共享了同一个父类型，所以可以轻松地在设计中添加新的组合对象或者局部对象，而无需大范围地修改代码。
- 简单：使用组合结构的客户端代码只需设计简单的接口。客户端代码没有必要区分一个对象是组合对象还是局部对象（除了添加新组件时）。客户端代码调用Army::bombardStrength()方法时也许会产生一些幕后的委托调用，但是对于客户端代码而言，无论是过程还是效果，都和调用Archer::bombardStrength()方法是完全相同的。
- 隐式到达（implicit reach）：组合模式中的对象通过树型结构组织。每个组合对象中都保

① 因为按常理，我们无法往一个射手中添加一个射手。这些基本单元应该无法添加其他单元。——译者注

存着对子对象的引用。因此对树中某部分的一个小操作可能会产生很大的影响。比如，我们可能会将一个父Army对象的某个子Army对象删除，并将其添加到另外一个父Army对象上去。这个简单的动作看似只对子Army对象产生作用，但实际上却影响了子Army对象中所引用的Unit对象及其子对象的状态。

- 显式到达（explicit reach）：树型结构可轻松遍历。可以通过迭代树型结构来获取组合对象和局部对象的信息，或对组合对象和局部对象执行批量处理。在下一章中，我们在处理访问者模式时会看到一个与此相关的强大技术。

通常来说，从客户的角度，我们最能体会到使用模式来到的好处，因此接下来让我们创建一些Army对象：

```
// 创建一个Army对象
$main_army = new Army();

// 添加一些Unit对象
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );

// 创建一个新的Army对象
$sub_army = new Army();

// 添加一些Unit对象
$sub_army->addUnit( new Archer() );
$sub_army->addUnit( new Archer() );
$sub_army->addUnit( new Archer() );

// 把第二个Army对象添加到第一个Army对象中去
$main_army->addUnit( $sub_army );

// 所有的攻击强度计算都在幕后进行
print "attacking with strength: {$main_army->bombardStrength()}\n";
```

我们创建了一个新的Army对象，并为它添加了一些简单的Unit对象，然后我们创建了另一个Army对象，并将其添加到第一个Army对象中。当我们调用第一个Army对象的Unit::bombardStrength()方法时，就可以在幕后计算各个局部对象的攻击强度并得出总的攻击强度。组合结构的所有复杂性都被完全隐藏了。

10.2.3 效果

你也许会像我一样，在看到Archer类的代码时就想起了警报——为什么我们要把addUnit和removeUnit()这样的冗余方法加到并不需要它们的局部类中？答案在于Unit类型的透明性。

客户端代码在得到一个Unit对象时，知道其中一定包含addUnit()方法。组合模式的原则便是局部类和组合类具有同样的接口。不过这未必能真正帮助我们，因为我们仍然不知道调用Unit对象的addUnit()方法是否安全。

如果我们将add/remove方法降到下一级对象中，以便这些方法仅在组合类中使用，那么又

会产生另一个问题——使用Unit对象时，我们并不知道该对象是否默认支持addUnit()方法。但是，将冗余的类方法留在局部类中让人感到不舒服。这样做毫无意义而且给系统设计带来歧义，因为接口应该关注于自己独有的功能。

我们可以十分轻松地将组合类分解为CompositeUnit子类型。首先删除Unit的add/remove动作：

```
abstract class Unit {
    function getComposite() {
        return null;
    }

    abstract function bombardStrength();
}
```

注意一下新增的getComposite()方法，不过我们稍后再来看这个方法。现在我们需要一个新的拥有addUnit()和removeUnit()方法的抽象类。我们甚至可以给这两个方法加上默认实现：

```
abstract class CompositeUnit extends Unit {
    private $units = array();

    function getComposite() {
        return $this;
    }

    protected function units() {
        return $this->units;
    }

    function removeUnit( Unit $unit ) {
                $this->units = array_udiff( $this->units, array( $unit ),

                        function( $a, $b ) { return ($a === $b)?0:1; } );

    }

    function addUnit( Unit $unit ) {
        if ( in_array( $unit, $this->units, true ) ) {
            return;
        }
        $this->units[] = $unit;
    }
}
```

因为CompositeUnit类继承自Unit类，并没有实现抽象的bombardStrength()方法，所以CompositeUnit类也被声明为抽象的，虽然它本身并没有抽象方法。Army类（或任何其他组合类）现在可扩展CompositeUnit了，而我们示例中的类现在将以图10-2的形式来组织。

虽然我们删除了局部类中冗余、烦人的add/remove方法，但客户端代码在使用addUnit()方法前却仍必须检查对象是否为一个CompositeUnit类型的对象。

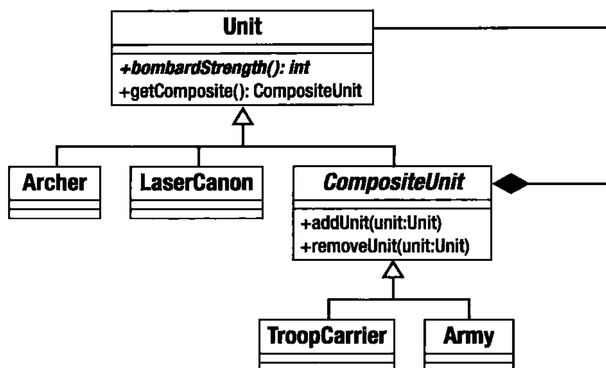


图10-2 从基类中移走add/remove方法

这就是添加getComposite()方法的原因。该方法默认返回一个null值，它仅在CompositeUnit类中才返回CompositeUnit本身。所以如果该方法返回一个对象，那么便可以调用它的addUnit()方法。下面是使用这种思路的客户端代码：

```
class UnitScript {
    static function joinExisting( Unit $newUnit,
                                  Unit $occupyingUnit ) {
        $comp;

        if ( ! is_null( $comp = $occupyingUnit->getComposite() ) ) {
            $comp->addUnit( $newUnit );
        } else {
            $comp = new Army();
            $comp->addUnit( $occupyingUnit );
            $comp->addUnit( $newUnit );
        }
        return $comp;
    }
}
```

joinExisting()方法有两个Unit对象参数。第一个参数是一个区块的新来者，第二个参数是之前的占据者。如果第二个Unit对象是一个CompositeUnit对象，那么第一个Unit对象被添加给它。但如果不是的话，方法将会创建一个新Army对象并将这两个对象添加到Army对象中。我们一开始并不知道$occupyingUnit参数是否包含CompositeUnit对象，但是通过调用getComposite()可以解决这个问题。若getComposite()返回对象，那么我们可以直接添加新的Unit对象。如果不是，那么我们创建一个新的Army对象，并将两个Unit对象都添加给它。

我们还可以进一步简化该模型，让Unit::getComposite()方法返回一个添加当前Unit的Army对象。或者也可以返回前面的模型（即不从结构上区分组合或局部对象），并用Unit::addUnit()方法来做同样的事情：创建一个Army对象，并将两个Unit对象都添加给它。虽然这样代码很简洁，但是这假设你已经预先知道了部队单元所要聚合的组合类型。当你设计如getComposite()和addUnit()这样的方法时，业务逻辑将决定你所能做的假设。

而这正是组合模式缺陷的症状之一。简化的前提是使所有的类都继承同一个基类。简化的好处有时会以降低对象类型安全为代价。模型变得越复杂，你就不得不手动进行越多的类型检查。比如有一个Cavalry（骑兵）对象，假设游戏规则规定不能将一匹马放到运兵船上，在组合模式中我们并没有自动化的方式来强制执行这个规则：

```
class TroopCarrier {

    function addUnit( Unit $unit ) {
        if ( $unit instanceof Cavalry ) {
            throw new UnitException("Can't get a horse on the vehicle");
        }
        super::addUnit( $unit );
    }

    function bombardStrength() {
        return 0;
    }
}
```

这里我们不得不使用instanceof操作符来检查传给addUnit()方法的对象类型。特殊对象越来越多，组合模式开始渐渐显得弊大于利。在大部分局部对象可互换的情况下，组合模式才最适用。

需要担心的另外一个问题是组合操作的成本。Army::bombardStrength()方法便是典型的例子，该方法会逐级调用对象树中下级分支的方法。如果一个对象树中有大量子Army对象，一个简单的调用可能会导致系统崩溃。虽然bombardStrength()方法自身并不一定会带来多大的系统开销，但是如果一些局部对象为了获得返回值而执行一系列复杂的计算，那么会发生什么呢？解决问题的一个办法是在父级对象中缓存计算结果，这样可使接下来的调用减少系统开销。尽管如此，你还是需要小心地保证缓存值不会失效。因此你需要设计一个当对树进行操作时可移除过期缓存的策略，而这也许会要求你给子对象加上对父对象的引用。

最后，在对象持久化上需要注意一些地方。虽然组合模式是一个优雅的模式，但它并不能将自身轻松地存储到关系数据库里。这是因为在默认的情况下，你是通过级联引用来访问整个结构的。因此，按正常的途径从数据库构造一个组合结构，你将不得不使用多个昂贵的查询。我们可以通过给对象树中的每个节点赋于ID值来避免这个问题，于是所有组件都可从数据库中一次性取出。但是获得这些对象后，我们仍旧需要重建父子之间的引用关系，以便它们再保存回数据库。虽然这不难，可并不优雅，甚至有些混乱。

虽然组合模式难以使数据保存在关系型数据库中，但其数据却非常适合持久化为XML。这是因为XML元素通常是由树型结构的子元素组合而成的。

10.2.4　组合模式小结

如果你想像对待单个对象一样对待组合对象，那么组合模式十分有用。可能是因为组合对象本质上和局部对象（比如Army和Archer）相似，也可能因为在环境中组合对象与局部对象（如发票中的条目）的特征相同。因为组合模式是树型结构，所以对整体的操作会影响到局部，而通

过组合，对客户端代码来说局部的数据又是透明的。组合模式使这些操作和查询对客户端代码透明。对象树可以方便地进行遍历（在下一章中我们将可以看到）。你也可以轻松地在组合结构中加入新的组件类型。

但另一方面，组合模式又依赖于其组成部分的简单性。随着我们引入复杂的规则，代码会变得越来越难以维护。组合模式不能很好地在关系数据库中保存数据，但却非常适合使用XML持久化。

10.3 装饰模式

组合模式帮助我们聚合组件，而装饰模式则使用类似结构来帮助我们改变具体组件的功能。该模式同样体现了组合的重要性，但组合是在代码运行时实现的。继承是共享父类特性的一种简单的办法，但可能会使你将需要改变的特性硬编码到继承体系中，而这常会降低系统灵活性。

10.3.1 问题

将所有功能建立在继承体系上会导致系统中的类“爆炸式”增多。更糟糕的是，当你尝试对继承树上不同的分支做相似的修改时，代码可能会产生重复。

让我们回到之前的游戏中。下面的代码定义了Tile（区域）类及其子类：

```
abstract class Tile {
    abstract function getWealthFactor();
}

class Plains extends Tile {
    private $wealthfactor = 2;
    function getWealthFactor() {
        return $this->wealthfactor;
    }
}
```

我们定义了一个Tile类。Tile表示部队单元所在的一个区域。每个Tile对象都有明确的特征。在本例中，我们定义了一个getWealthFactor()方法，用于计算当某个特定区域被一个玩家所占有后的收益。如你所见，Plains（平原）对象的财富系数是2。显然，Tile对象还管理其他数据。它们也许会有对图片的引用，用于在屏幕上绘图。这里我们还是先保持简单。

我们还需要修改Plains对象的行为，用于处理一些自然资源和人类滥用的效果。我们希望为地表钻石的分布和污染造成的破坏建模。有一个办法是从Plains对象派生：

```
class DiamondPlains extends Plains {
    function getWealthFactor() {
        return parent::getWealthFactor() + 2;
    }
}

class PollutedPlains extends Plains {
    function getWealthFactor() {
        return parent::getWealthFactor() - 4;
```

```
    }
}
```

现在，我们能非常轻松地获得一个被污染的区块的财富系数：

```
$tile = new PollutedPlains();
print $tile->getWealthFactor();
```

图10-3是本例的类图。

这样的结构显然不具灵活性。我们可以获得有钻石的Plains对象，也可以获得被污染的Plains对象。但是我们能同时获得既含有钻石又被污染的Plains对象吗？显然不可以，除非我们自己希望自找麻烦来创建一个PollutedDiamondPlains类。当我们引入既含有钻石又被污染的Forest类，情况会变得更糟。

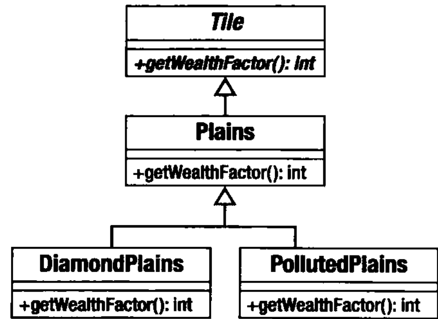


图10-3　通过继承树实现子类的变化（variation）

当然这是极端的例子，但已经证明了我们的观点——功能定义完全依赖于继承体系会导致类的数量过多，而且代码会产生重复。

关于这个问题，我们再举一个更真实的例子。真正的Web应用要在返回响应给用户之前执行一系列操作，例如验证用户及记录请求等。也许还需要将请求中的原始输入处理成某种数据结构。最后，还必须执行核心操作。此时我们遇到了同样的问题。

在派生的LogRequest类、StructureRequest类和AuthenticateRequest类中，我们可以对基类ProcessRequest类的功能进行扩展，并加上额外的处理。你可以在图10-4中看到该层级关系。

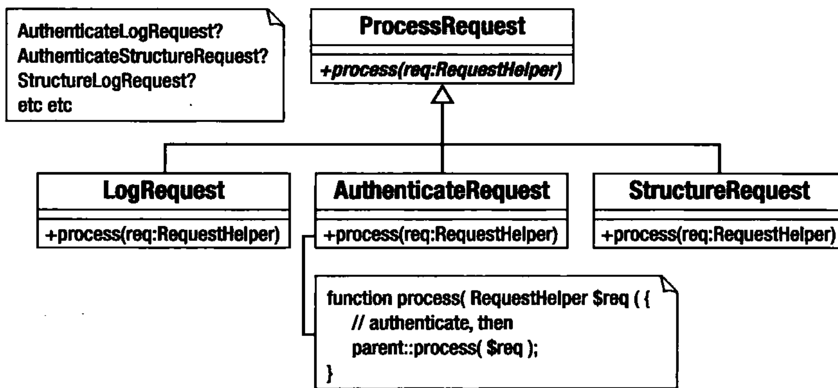


图10-4　不同的操作被硬编码

但是当需要执行记录和验证，而不需要准备数据时，该怎么办？创建一个LogAndAuthenticateProcessor类？显然，我们必须找出一个更灵活的方案。

10.3.2 实现

装饰模式使用组合和委托而不是只使用继承来解决功能变化的问题。实际上，Decorator类会持有另外一个类的实例。Decorator对象会实现与被调用对象的方法相对应的类方法。用这种办法可以在运行时创建一系列的Decorator对象。

让我们重写游戏示例来演示该模式：

```
abstract class Tile {
    abstract function getWealthFactor();
}

class Plains extends Tile {
    private $wealthfactor = 2;
    function getWealthFactor() {
        return $this->wealthfactor;
    }
}

abstract class TileDecorator extends Tile {
    protected $tile;
    function __construct( Tile $tile ) {
        $this->tile = $tile;
    }
}
```


我们和之前一样声明了Tile和Plains类，不同的是引入了一个新类TileDecorator。因为没有实现getWealthFactor()方法，所以必须被声明为抽象类。我们定义了以一个Tile对象为参数的构造方法，传入的对象会被保存在$tile属性中。该属性被声明为protected，以便子类可以访问它。下面我们重新定义Pollution和Diamond类：

```
class DiamondDecorator extends TileDecorator {
    function getWealthFactor() {
        return $this->tile->getWealthFactor()+2;
    }
}

class PollutionDecorator extends TileDecorator {
    function getWealthFactor() {
        return $this->tile->getWealthFactor()-4;
    }
}
```

这些类都扩展自TileDecorator类，这意味着它们拥有指向Tile对象的引用。当getWealth-Factor方法被调用时，这些类都会先调用所引用的Tile对象的getWealthFactor()方法，然后执行自己特有的操作。

通过像这样使用组合和委托，可以在运行时轻松地合并对象。因为模式中所有的对象都扩展自Tile，所以客户端代码并不需要知道内部是如何合并的。getWealthFactor()方法在任何Tile对象中都是可用的，无论该对象是一个装饰器对象还是真正的Tile对象。

```
$tile = new Plains();
print $tile->getWealthFactor(); // 2
```

Plains组件简单地返回2。

```
$tile = new DiamondDecorator( new Plains() );
print $tile->getWealthFactor(); // 4
```

DiamondDecorator有一个指向Plains对象的引用，在加上自己执行得到的2之前，它会先调用Plains对象的getWealthFactor()方法：

```
$tile = new PollutionDecorator(new DiamondDecorator( new Plains() ));
print $tile->getWealthFactor(); // 0
```

PollutionDecorator引用了DiamondDecorator对象，而DiamondDecorator对象又拥有对Plains对象的引用。

本例的类图如图10-5所示。

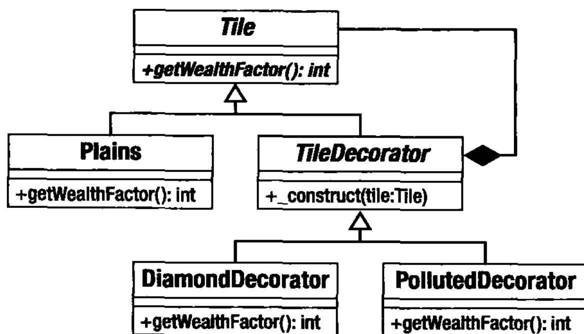


图10-5　装饰模式

这样的模型极具扩展性。我们可以非常轻松地添加新的装饰器或者新的组件。通过使用大量的装饰器，我们可以在运行时创建极为灵活的结构。本例中的组件类Plains的行为可以很方便地被改变，而不需要改动原来的类。通俗地说，这意味着不需要创建PollutedDiamondPlains对象，就可以构造一个拥有钻石并被污染的Plains对象。

装饰模式所建立的管道[①]对于创建过滤器非常有用。Java的IO包便广泛使用了装饰器类。客户端程序员可将核心组件与装饰对象合并，从而对核心方法（如read()）进行过滤、缓冲、压缩等操作。我们的Web请求示例也能被开发为一个具有可配置管道的模型。下面便是使用装饰模式的一个简单实现：

```
class RequestHelper{}

abstract class ProcessRequest {
    abstract function process( RequestHelper $req );
}
```

① 多个装饰器，尤如管道般串联起来。——译者注

```
class MainProcess extends ProcessRequest {
    function process( RequestHelper $req ) {
        print __CLASS__.": doing something useful with request\n";
    }
}

abstract class DecorateProcess extends ProcessRequest {
    protected $processrequest;
    function __construct( ProcessRequest $pr ) {
        $this->processrequest = $pr;
    }
}
```

和之前一样,我们定义了一个抽象基类(ProcessRequest)、一个具体的组件(MainProcess)和一个抽象装饰类(DecorateProcess)。MainProcess::process()方法仅仅报告方法被调用,并没有其他功能。DecorateProcess为它的子类保存了一个ProcessRequest对象。下面是一些简单的具体装饰类:

```
class LogRequest extends DecorateProcess {
    function process( RequestHelper $req ) {
        print __CLASS__.": logging request\n";
        $this->processrequest->process( $req );
    }
}

class AuthenticateRequest extends DecorateProcess {
    function process( RequestHelper $req ) {
        print __CLASS__.": authenticating request\n";
        $this->processrequest->process( $req );
    }
}

class StructureRequest extends DecorateProcess {
    function process( RequestHelper $req ) {
        print __CLASS__.": structuring request data\n";
        $this->processrequest->process( $req );
    }
}
```

装饰类的每个process()方法在调用引用的ProcessRequest对象的process()方法前输出一条信息。现在我们可以在运行时合并这些类的初始对象,创建过滤器来对每一个请求按不同的顺序执行不同的操作。下面的代码将所有具体类的对象组合为一个过滤器:

```
$process = new AuthenticateRequest( new StructureRequest(
                                    new LogRequest (
                                    new MainProcess()
                                    )));
$process->process( new RequestHelper() );
```

执行代码会得到下面的输出结果:

```
Authenticate
Request: authenticating request
StructureRequest: structuring request data
LogRequest: logging request
MainProcess: doing something useful with request
```

注解　事实上，这个示例也是企业模式中"拦截过滤器"的例子。拦截过滤器在《J2EE核心模式》一书中有所阐述。

10.3.3　效果

和组合模式一样，装饰模式有时也会令人困惑。一定要记住：组合和继承通常都是同时使用的。因此，LogRequest是继承自ProcessRequest的，但是却表现为对另外一个ProcessRequest对象的封装。

因为装饰对象作为子对象的包装，所以保持基类中的方法尽可能少是很重要的。如果一个基类具有大量特性，那么装饰对象不得不为它们包装的对象的所有public方法加上委托。你可以用一个抽象的装饰类来实现，不过这仍旧会带来耦合，并可能导致bug的出现。

有些程序员创建的装饰类与其中拥有的对象并不共享相同的对象类型。只要这些装饰类提供相同的接口，这个策略也是可行的。使用PHP内建的拦截方法来实现自动委托，可以带来很多好处（如通过实现__call()方法来捕捉装饰类中不存在的方法，并自动调用子对象对应的相同的方法）。但是，这样的话会丧失类型检查带来的安全性。就我们的例子来说，客户端代码要求传入的参数是一个Tile或ProcessRequest对象，并能确定该对象可以提供某些接口，无论传入的对象是否经过大量地装饰。

10.4　外观模式

可能你曾经有过在项目中集成第三方代码的经历。无论第三方代码是否面向对象，集成通常都是巨大而且复杂的工作，很容易让人畏缩。而对一个客户程序员来说，你的代码也可能是一个挑战，即使他可能仅需要访问很小一部分特性。外观模式可以为复杂系统创建一个简单、清晰的接口。

10.4.1　问题

在系统中总会逐渐形成大量仅在系统自身内部有用的代码。系统也应该像类那样，提供定义清晰的公共接口，并对外隐藏内部结构。但是，系统中哪部分应该设置为公开，哪部分设置为隐藏并不容易决定。

当使用子系统（如Web论坛或者图库）的代码时，你也许会发现自己过于深入地调用子系统的逻辑代码。如果子系统代码总是在不断变化，而你的代码却又在许多不同地方与子系统代码交互，那么随着子系统的发展，你也许会发现维护代码变得非常困难。

类似地，当你创建自己的系统时，将不同的部分分层是很好的做法。例如，你可能会把系统

分为应用逻辑层、数据库交互层和表现层等。你应该尽可能地使这些分层互相独立，以便项目中某一部分的修改尽量不影响到其他地方。如果某层中的代码与另一层的代码存在耦合的话，那么这个目标就很难达成了。

下面是一些故意让人混淆的不着边际的程序代码，其功能只是从文件中获取log信息并将它转换为对象数据：

```
function getProductFileLines( $file ) {
    return file( $file );
}

function getProductObjectFromId( $id, $productname ) {
    // 一些数据库查询
    return new Product( $id, $productname );
}

function getNameFromLine( $line ) {
    if ( preg_match( "/.*-(.*)\s\d+/", $line, $array ) ) {
        return str_replace( '_',' ', $array[1] );
    }
    return '';
}

function getIDFromLine( $line ) {
    if ( preg_match( "/^(\d{1,3})-/", $line, $array ) ) {
        return $array[1];
    }
    return -1;
}

class Product {
    public $id;
    public $name;
    function __construct( $id, $name ) {
        $this->id = $id;
        $this->name = $name;
    }
}
```

假设上面代码的内部其实是更复杂的，这样我们可以继续使用它们而非从头开始重写。我们的目的是将包含类似下面数据的文件转换为一个对象数组：

```
234-ladies_jumper 55
532-gents_hat 44
```

而我们必须调用所有这些方法（注意为了简单起见，我们并不计算出价格的最终数字）：

```
$lines = getProductFileLines( 'test.txt' );
$objects = array();
foreach ( $lines as $line ) {
    $id = getIDFromLine( $line );
    $name = getNameFromLine( $line );
    $objects[$id] = getProductObjectFromID( $id, $name  );
}
```

如果在项目中像这样直接调用这些方法，那么我们的代码会和子系统紧紧地耦合在一起。当子系统变化时，或者我们决定将其与子系统完全断开时，代码就会出问题。可见，我们需要在这些子系统和代码中引入一个入口。

10.4.2 实现

下面这个简单的类为上面的过程式代码提供了一个接口：

```
class ProductFacade {
    private $products = array();

    function __construct( $file ) {
        $this->file = $file;
        $this->compile();
    }

    private function compile() {
        $lines = getProductFileLines( $this->file );
        foreach ( $lines as $line ) {
            $id = getIDFromLine( $line );
            $name = getNameFromLine( $line );
            $this->products[$id] = getProductObjectFromID( $id, $name  );
        }
    }

    function getProducts() {
        return $this->products;
    }

    function getProduct( $id ) {
        return $this->products[$id];
    }
}
```

从客户端代码的角度来看，现在从一个log文件访问Product对象简单多了：

```
$facade = new ProductFacade( 'test.txt' );
$facade->getProduct( 234 );
```

10.4.3 效果

外观模式是一个十分简单的概念，它只是为一个分层或一个子系统创建一个单一的入口。这会带来许多好处。首先，有助于分离项目中的不同部分。其次，对于客户端开发者来说，访问代码变得简洁，非常方便。另外，由于只在一个地方调用子系统，减少了出错的可能性，并因此可以预估子系统修改带来的问题所在。Facade类还能使客户端代码避免不正确地使用子系统中复杂的内部方法，从而减少错误的发生。

尽管外观模式十分简单，但你很容易忘记它，特别是在你熟悉所使用的子系统时。这里存在着某种平衡。一方面，为复杂系统创建简单接口的好处是明显的；另一方面，你可能会过度抽象系统。总之，如果你想获得简洁的客户端访问代码，或者想把系统中的修改对客户端代码隐藏，

那么你也许应该实现外观模式。

10.5 小结

在本章中，我们研究了类和对象的几种组织方式，其中特别关注了使用组合以得到继承无法达到的灵活性。在组合模式和装饰模式中，使用继承是为了更好地组合对象，并为客户端代码提供统一的接口。

我们也可以看到在这些模式中委托是如何被有效使用的。最后，我们研究了简单却强大的外观模式。外观模式是众多模式中一个被很多人运用了多年却没有得到命名的模式。外观模式可以为一个分层或一个子系统提供一个简洁的入口。在PHP中，外观模式也可用于创建封装过程式代码块的对象封装器。

第11章 执行及描述任务

本章介绍的几个模式能帮助我们解决实际问题，例如解析一种迷你语言，或封装一个算法。

本章包括以下内容。

- 解释器（Interpreter）模式：构造一个可以用于创建脚本化应用的迷你语言解释器。
- 策略（Strategy）模式：在系统中定义算法并以它们自己的类型封装。
- 观察者（Observer）模式：创建依赖关系，当有系统事件发生时通知观察者对象。
- 访问者（Visitor）模式：在对象树的所有节点上应用操作。
- 命令（Command）模式：创建可被保存和传送的命令对象。

11.1 解释器模式

编程语言总是用其他编程语言编写的（至少一开始是这样）。例如，PHP就是用C编写的。虽然听起来可能有些怪，但出于同样的原因，我们也能用PHP定义和运行我们自己发明的编程语言。当然，我们创建的语言运行起来会比较慢，功能也很有限。尽管如此，迷你语言是非常有用的，这将在本章中介绍。

11.1.1 问题

当我们用PHP创建Web（或者命令行）接口时，我们会给用户访问功能的权限。在设计接口时，我们需要权衡易用性和功能。通常，你为用户提供的功能越多，你的接口就会越混乱。当然一个优秀的接口设计可以提供很大的帮助，但是如果90%的用户只使用你特性中相同的30%，堆积在功能上的成本便可能会弊大于利了，而此时你也许该考虑为了大多数用户简化系统。但是那些使用系统中高级特性的另外10%的高级用户，又怎么办呢？你可以通过其他方法来满足他们的要求。通过给这些用户提供一种领域语言（通常被称为DSL——Domain Specific Language，领域特定语言），你也许能有效地扩展你应用程序的功能。

当然，我们有一个编程语言马上可以使用，那就是PHP。下面就是允许用户为我们的系统写脚本的一个例子：

```
$form_input = $_REQUEST['form_input']
```

```
// 用户可能输入： "print file_get_contents('/etc/passwd')"
eval( $form_input )
```

但使用这种方式来使应用程序具有脚本化的能力显然是非常愚蠢的。原因非常明显，可归结为两点：安全性和复杂性。上面的代码已经很好地揭示了其中的安全问题。用户通过我们的脚本执行PHP代码，取得了对运行脚本的服务器的访问权限，而复杂性是另一个巨大的缺陷。无论你的代码多么清晰，普通用户不太容易扩展它，特别是无法通过浏览器窗口来扩展它。

但是通过设计一个迷你语言，能够解决这两个问题。你可以将语言设计得更灵活，降低用户损坏系统的可能性，并使功能更加集中。

假设有一个问答程序，制作者设计问题并为回答者提交的答案制定评分规则。问答评分必须没有人为干涉，即使有些答案是用户在文本框中输入的。

比如这个问题：

《设计模式》有几个作者？

正确答案为"four"或"4"。我们可以创建一个Web界面来允许命题人使用正则表达式来标记正确答案：

```
^4|four$
```

然而，大多数命题人并不了解正则表达式。为简单起见，我们可以实现一个对用户更加友好的机制来标记回应，如：

```
$input equals "4" or $input equals "four"
```

这里我们设计了一种支持变量、equals操作符和布尔逻辑（or和and）操作符的语言。程序员通常都喜欢给东西命名，因此我们可以把这个语言称为MarkLogic。因为我们以后可能还需要更多的语言特性，所以MarkLogic应该是易于扩展的。不过先让我们暂时把分析输入的问题放在一边，先实现一个能在运行时集合这些元素并得出一个答案的机制。和你期望的一样，解释器模式正好可以用在此处。

11.1.2 实现

我们的语言是由表达式组成的。如表11-1所示，即使像MarkLogic这样简单的语言，也需要由很多元素组成。

表11-1 MarkLogic语言的语法元素

描 述	EBNF名	类 名	范 例
变量	variable	VariableExpression	$input
字符串	<stringLiteral>	LiteralExpression	"four"
布尔：与	andExpr	BooleanAndExpression	-$input equals '4' and $other equals '6'
布尔：或	orExpr	BooleanOrExpression	-$input equals '4' or $other equals '6'
相等测试	equalsExpr	EqualsExpression	$input equals '4'

表11-1列出了EBNF名称。EBNF是一种可以用来描述一个语言语法的符号。EBNF是Extended

Backus-Naur Form（扩展巴科斯范式）的缩写。它由一系列语句（也称production，部件）组成，每一条语句又由一个名称及一个描述组成。描述中包含对其他部件和终端（terminal，即本身不包含对其他部件的引用的元素）的引用。下面使用EBNF来描述我们语言的语法：

```
expr     ::= operand (orExpr | andExpr )*
operand  ::= ( '(' expr ')' | <stringLiteral> | variable ) ( eqExpr )*
orExpr   ::= 'or' operand
andExpr  ::= 'and' operand
eqExpr   ::= 'equals' operand
variable ::= '$' <word>
```

有一些符号具有特殊意义（这在正则表达式中很常见）：比如*表示零或者多个，|表示“或者”。另外，我们可以用括号把元素括起来。所以在示例中，表达式（expr）由Operand和零个或多个orExpr或andExpr组成。Operand可以是一个带括号的表达式，也可以是一个带引号的字符串（为此我已经忽略了部件）或者一个后跟一个或多个eqExpr实例的变量。一旦你熟悉了一个部件对另一个部件的引用，EBNF将变得相当易懂。

在图11-1中，我们将语法中的各元素用类来描述。

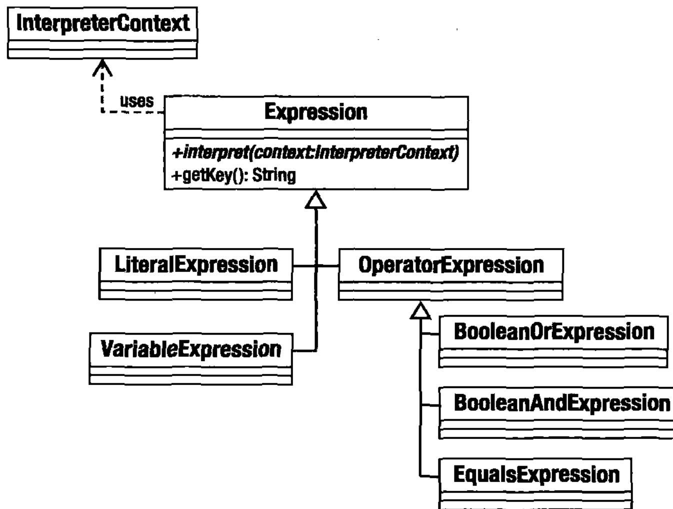


图11-1　MarkLogic语言的解释器类

如图11-1所示，BooleanAndExpression和它的兄弟类都继承自OperatorExpression（操作符表达式）。这是因为这些类都是对其他Expression对象执行操作的，而VariableExpression（变量表达式）和LiteralExpression（字符表达式）直接对值进行操作。

所有Expression对象都实现了抽象基类Expression的interpret()方法。interpret()方法将InterpreterContext对象用作共享的数据存储。每个Expression对象都可以在InterpreterContext对象中储存数据。InterpreterContext对象可以被传给其他Expression对象。因为Expression基类实现了一个返回唯一句柄的getKey()方法，所以可以从Interpreter-

Context对象中轻松获取数据。下面看一下Expression对象是如何工作的。

```
abstract class Expression {
    private static $keycount=0;
    private $key;
    abstract function interpret( InterpreterContext $context );

    function getKey() {
        if ( ! asset( $this->key ) ) {
            self::$keycount++;
            $this->key=self::$keycount;
        }
        return $this->key;
    }
}

class LiteralExpression extends Expression {
    private $value;

    function __construct( $value ) {
        $this->value = $value;
    }

    function interpret( InterpreterContext $context ) {
        $context->replace( $this, $this->value );
    }
}

class InterpreterContext {
    private $expressionstore = array();

    function replace( Expression $exp, $value ) {
        $this->expressionstore[$exp->getKey()] = $value;
    }

    function lookup( Expression $exp ) {
        return $this->expressionstore[$exp->getKey()];
    }
}

$context = new InterpreterContext();
$literal = new LiteralExpression( 'four');
$literal->interpret( $context );
print $context->lookup( $literal ) . "\n";
```

输出结果如下：

```
four
```

先介绍一下InterpreterContext类。可以看到，它其实是关联数组$expressionstore的前端，该数组用于保存数据。replace()方法以一个Expression对象（作为数组元素的键）及一个任意类型的值（作为数组元素的值）为参数，然后成对添加到$expressionstore数组中。

11

InterpriterContext类同时也提供了lookup()方法来获取数据。

Expression类定义了抽象的interpret()方法，并定义了具体方法getKey()，其中后者使用静态计数器的值来生成、存储和返回一个标识。

getKey()方法会被InterpreterContext::lookup()和InterpreterContext::replace()方法使用，用于索引数据。

LiteralExpression类定义了一个以值为参数的构造函数。LiteralExpression类的interpret()方法要求传入一个InterpreterContext对象作为参数。我们只调用InterpreterContext的replace()方法，其中会调用getKey()来定义键值和$value属性，在其他Expression类中也是如此。interpret()方法总是将结果记录到InterpreterContext对象中。

示例中同时还包括一些客户端代码，如实例化了一个InterpreterContext对象和一个LiteralExpression对象（参数为"four"）。我们把InterpreterContext对象传给LiteralExpression::interpret()，然后Interpret()方法在InterpreterContext对象中保存键/值对。我们通过调用InterpreterContext对象的lookup()来获得value值。

让我们再定义其他的终端类，如VariableExpression。VariableExpression稍微有点复杂：

```
class VariableExpression extends Expression {
    private $name;
    private $val;

    function __construct( $name, $val=null ) {
        $this->name = $name;
        $this->val = $val;
    }

    function interpret( InterpreterContext $context ) {
        if ( ! is_null( $this->val ) ) {
            $context->replace( $this, $this->val );
            $this->val = null;
        }
    }

    function setValue( $value ) {
        $this->val = $value;
    }

    function getKey() {
        return $this->name;
    }
}
$context = new InterpreterContext();
$myvar = new VariableExpression( 'input', 'four');
$myvar->interpret( $context );
print $context->lookup( $myvar ). "\n";
// 输出: four

$newvar = new VariableExpression( 'input' );
```

```
$newvar->interpret( $context );
print $context->lookup( $newvar ). "\n";
// 输出: four

$myvar->setValue("five");
$myvar->interpret( $context );
print $context->lookup( $myvar ). "\n";
// 输出: five
print $context->lookup( $newvar ) . "\n";
// 输出: five
```

VariableExpression类的构造函数有两个参数，分别是变量名与变量值，保存在相应的类属性变量中。我们提供了setValue()方法，使客户端代码可以在任何时候改变value值。

interpret()方法首先会检查$val属性是否是个非空值。如果$val属性非空，则把值设置到InterpreterContext中，然后我们把$val属性设为null，以防在同变量名的Variable-Expression的另一个实例改变InterpreterContext中的值后该实例的interpret()被再次调用。目前变量值只支持字符串类型。如果要扩展语言，就要考虑让它和其他Expression对象一起工作，那么它就要能包含检查和操作的结果。目前，VariableExpression还没有这些功能。注意我们覆盖了getKey()方法，这样与变量值对应的是变量名而不是静态ID。

为了获得结果，我们设计的语言中的Operator（操作符）表达式总是和另外两个Expression对象一起工作。因此，这些Operator表达式继承自一个共同的父类也是合理的。下面是OperatorExpression（操作符表达式）类的代码：


```
abstract class OperatorExpression extends Expression {
    protected $l_op;
    protected $r_op;

    function __construct( Expression $l_op, Expression $r_op ) {
        $this->l_op = $l_op;
        $this->r_op = $r_op;
    }

    function interpret( InterpreterContext $context ) {
        $this->l_op->interpret( $context );
        $this->r_op->interpret( $context );
        $result_l = $context->lookup( $this->l_op );
        $result_r = $context->lookup( $this->r_op  );
        $this->doInterpret( $context, $result_l, $result_r );
    }

    protected abstract function doInterpret( InterpreterContext $context,
                                             $result_l,
                                             $result_r );
}
```

OperatorExpression是个抽象类，它实现了interpret()方法，但同时也定义了一个抽象方法doInterpret()。

OperatorExpression的构造函数以两个Expression对象——$I_op和$r_op为参数，这两个

Expression对象会被保存在OperatorExpression的类属性里。

OperatorExpression的interpret()方法首先调用操作对象属性的interpret()方法。(如果你读了前面的一章内容，可能会注意到这里我们正在使用组合模式。)一旦运行了操作对象，interpret()则需要通过调用每个属性的InterpreterContext::lookup()方法来获得操作对象的返回值。然后，调用doInterpret()，由子类来决定怎么处理这些操作结果。

注解　doInterpret()是模板方法模式的一个实例。在该模式中，父类定义并调用了一个抽象方法，并留由子类来决定提供一个怎么样的实现。该模式能使具体类的开发更流畅，因为共享功能由父类处理，而子类只需关注于干净并有限的目标。

下面是EqualsExpression（相等表达式）类，用于检查两个Expression对象是否相等：

```
class EqualsExpression extends OperatorExpression {
    protected function doInterpret( InterpreterContext $context,
                                    $result_l, $result_r ) {
            $context->replace( $this, $result_l == $result_r );
    }
}
```

EqualsExpression仅实现了doInterpret()方法，该方法检查由interpret()方法传递来的两个操作对象的结果是否相等，并把结果保存到InterpreterContext对象中。

下面介绍一下BooleanOrExpression（布尔或表达式）类和BooleanAndExpression（布尔与表达式）类：

```
class BooleanOrExpression extends OperatorExpression {
    protected function doInterpret( InterpreterContext $context,
                                    $result_l, $result_r ) {
        $context->replace( $this, $result_l || $result_r );
    }
}

class BooleanAndExpression extends OperatorExpression {
    protected function doInterpret( InterpreterContext $context,
                                    $result_l, $result_r ) {
        $context->replace( $this, $result_l && $result_r );
    }
}
```

与检查是否相等不同，BooleanOrExpression类使用逻辑或操作，并通过Interpreter-Context::replace()方法保存结果。BooleanAndExpression类则使用了逻辑与操作。

现在我们的语言有足够的能力来处理之前的迷你语言片段：

```
$input equals "4" or $input equals "four"
```

我们用Expression系列类创建上面语句相对应的$statement：

```
$context = new InterpreterContext();
$input = new VariableExpression( 'input' );
```

```
$statement = new BooleanOrExpression(
    new EqualsExpression( $input, new LiteralExpression( 'four' ) ),
    new EqualsExpression( $input, new LiteralExpression( '4' ) )
);
```

我们先实例化一个没有赋值的'input'变量，然后创建一个BooleanOrExpression对象来比较两个EqualsExpression对象的结果。先将变量$input中的VariableExpression对象和含有字符串"four"的一个LiteralExpression对象比较，然后再将其和含有字符串"4"的一个LiteralExpression对象比较。

现在根据准备好的$statement变量，我们可以给$input变量赋值来运行代码：

```
foreach ( array( "four", "4", "52" ) as $val ) {
    $input->setValue( $val );
    print "$val:\n";
    $statement->interpret( $context );
    if ( $context->lookup( $statement ) ) {
        print "top marks\n\n";
    } else {
        print "dunce hat on\n\n";
    }
}
```

事实上，我们根据3个不同的值运行了代码3次。第一次运行时，临时变量$val的值为"four"，通过调用VariableExpression的setValue()方法赋值给输入对象VariableExpression，然后调用最顶端的Expression对象（即含有对语句中所有其他Expression对象引用的BooleanOrExpression对象）的interpret()方法。让我们看一下调用的内部过程：

- $statement调用$l_op属性（第一个EqualsExpression对象）的interpret()方法。
- 第一个EqualsExpression对象调用自身的$l_op属性（对输入对象VariableExpression的引用，其当前值为"four"）的interpret()方法。
- 输入对象VariableExpression通过调用InterpreterContext::replace()将它的当前值写入所提供的InterpreterContext对象。
- 第一个EqualsExpression对象调用自身$r_op属性（即值为"four"的一个LiteralExpression对象）的interpret()方法。
- LiteralExpression对象将自身的键值注册入InterpreterContext。
- 第一个EqualsExpression对象从InterpreterContext对象中取出$l_op("four")和$r_op ("four")的值。
- 第一个EqualsExpression对象比较这两个值是否相等，并将结果（即true）及其键（key）注册到InterpreterContext对象中。
- 对象树的顶端对象$statement（即BooleanOrExpression）调用自身$r_op属性的interpret()，然后与$l_op属性一样获得一个值（这里是false）。
- $statement对象从InterpreterContext对象中获得每个操作对象的值，并将值用||进行比较。在这里它用||比较true和false，所以结果是true。这个最终的结果也会被保存在InterpreterContext对象中。

11

上面所有步骤仅仅是循环中的第一次迭代过程。下面是最终的输出结果：

```
four:
top marks

4:
top marks

52:
dunce hat on
```

在进一步阅读前，你可能需要多看几遍这部分内容，否则对象和类的树型关系可能会让你困惑。Expression类在继承体系中的排列方式正如Expression对象在运行时被组合成树的方式。当你回头阅读代码时，一定要牢记这个特点。

图11-2展示了示例的完整类图。

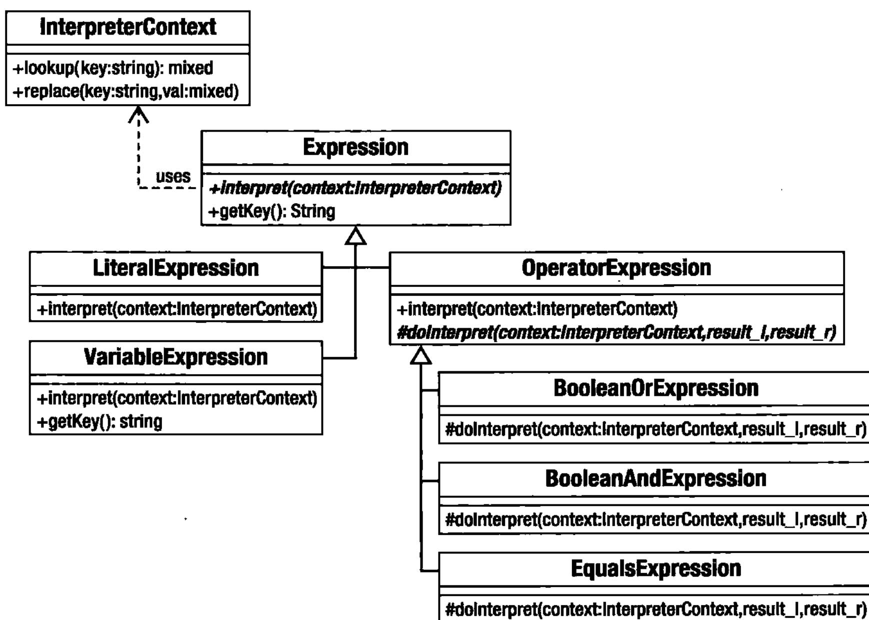


图11-2　解释器模式

11.1.3　解释器的问题

创建了解释器模式的核心类后，解释器很容易进行扩展。但是当语言变得复杂，需要创建的类的数量会很快增加。因此解释器模式最好应用于相对小的语言。如果你需要的是一个全能的编程语言，那么最好使用第三方工具。

因为解释器类经常执行非常相似的任务，所以最好仔细检察创建的类是否重复。

很多人第一次接触解释器模式时很兴奋，但很快发现这个模式其实并不涉及语言解析，因而感到失望。这意味着我们还不能给用户提供一个非常友好的语言。附录B提供了一些粗略的代码，

演示了如何解析迷你语言。

11.2 策略模式

有些程序员会赋予类太多的功能。他们这样做的目的可能是为了让类执行一些相关的操作。如果这些操作需要根据环境变化，那么就需要将类分解为子类。你在明白这个道理之前，就已经有过不得不这样做的经历。

11.2.1 问题

我们已经创建了一个标记语言，现在继续使用之前的例子。测验需要问题，因此我们创建了`Question`类，并为其添加了`mark()`方法。现在假设用户回答问题时可以使用多种不同的标记方式，应该怎么办呢？

比如支持简单的MarkLogic语言、直接匹配以及正则表达式这3种标记方式。你可能首先想到的是使用子类来实现这些差异，如图11-3所示。

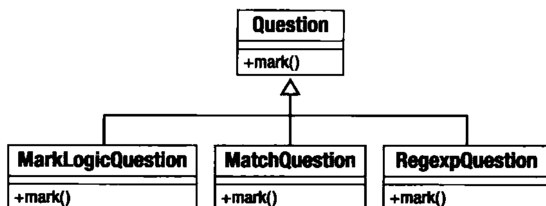


图11-3 根据不同的标记策略定义子类

如果只考虑这一方面的变化，这样做还能适应我们的需求。但是想象一下，如果我们被要求支持不同类型的问题——基于文本的问题和支持多媒体的问题，如图11-4所示，在一个继承树中创建多个子类的方案会产生一些问题。

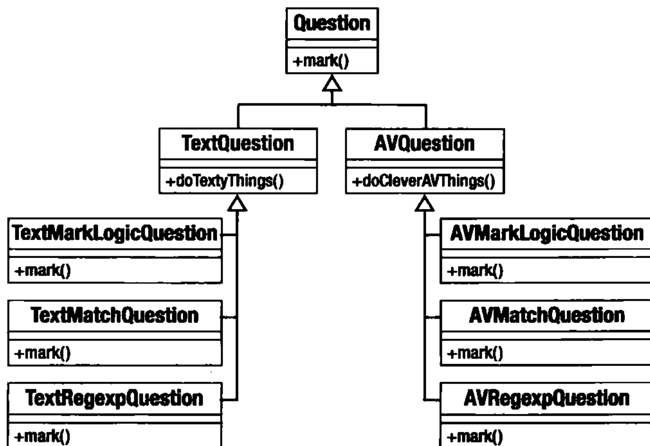


图11-4 根据两种强制分类定义子类

从图11-4中我们可以看到，不仅需要在体系中创建大量的类，而且会造成代码重复。因为我们的这个标记逻辑会在继承体系的每个分支中重复出现。

只要发现自己正不断地在继承树的各个分支中重复同一个算法（无论是通过子类还是通过重复条件语句），请将这些算法抽象成独立的类型。

11.2.2 实现

和所有优秀的模式一样，策略模式简单而强大。当类必须支持同一个接口的多种实现时（比如上例中的多种标记机制），最好的办法常常是提取出这些实现，并将它们放置在自己的类型中，而不是通过继承原有的类去支持这些实现。

因此，我们可以将示例中的标记方式放在一个Marker类型中。图11-5展示了这种新结构。

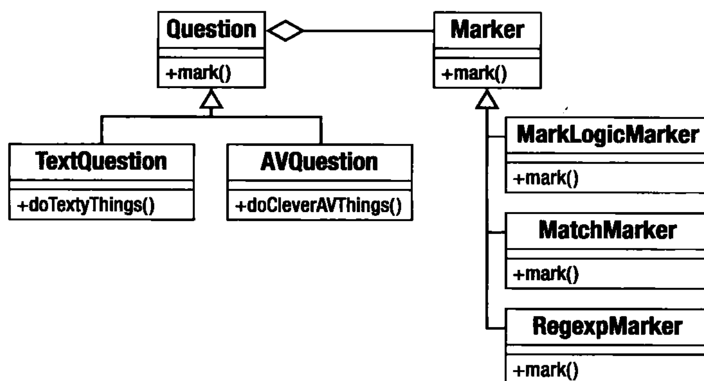


图11-5　提取算法到它们自己的类型

还记得《设计模式》中的“组合优于继承”原则吗？这便是一个很棒的示例。通过定义和封装标记算法，可以减少子类并增加灵活性。我们可以随时加入新的标记策略，而不需要改变Question类的代码。所有Question类所知道的仅是它们有一个可自由处理的Marker实例以及自身的接口保证支持mark()方法，而实现的细节便完全是其他类的问题了。

下面是Question类的代码：

```
abstract class Question {
    protected $prompt;
    protected $marker;

    function __construct( $prompt, Marker $marker ) {
        $this->marker=$marker;
        $this->prompt = $prompt;
    }

    function mark( $response ) {
        return $this->marker->mark( $response );
    }
}
```

```
class TextQuestion extends Question {
    // 处理文本问题特有的操作
}

class AVQuestion extends Question {
    //处理语音问题特有的操作
}
```

可以看到，我们把TextQuestion和AVQuestion的区别留给大家思考。Question基类提供了所有的实际功能，保存了一个$prompt属性和一个Marker对象。当Question::mark()使用终端用户的响应作为参数时，mark()方法只委托Marker对象来解决问题。

让我们再定义一些简单的Marker对象：

```
abstract class Marker {
    protected $test;

    function __construct( $test ) {
        $this->test = $test;
    }

    abstract function mark( $response );
}

class MarkLogicMarker extends Marker {
    private $engine;
    function __construct( $test ) {
        parent::__construct( $test );
        // $this->engine = new MarkParse( $test );
    }

    function mark( $response ) {
        // return $this->engine->evaluate( $response );
        // 模拟的返回值
        return true;
    }
}

class MatchMarker extends Marker {
    function mark( $response ) {
        return ( $this->test == $response );
    }
}

class RegexpMarker extends Marker {
    function mark( $response ) {
        return ( preg_match( $this->test, $response ) );
    }
}
```

Marker类本身没有什么特别之处。不过注意一下，附录B中的简单解析器使用了这里的MarkParse对象。我们目前不讨论该示例，因此我们只让MarkLogicMarker::mark()返回true。

11

在此我们更关注于结构设计，而不关注具体策略的实现细节。我们可以用RegexpMarker方便地替换掉MatchMarker，而不需要修改Question类的代码。

当然，你仍必须决定在具体的Marker对象中用什么标记方式。在现实项目中，有两个办法来解决这个问题。一个是在界面上提供单选按钮，让用户选择所要使用的标记策略。另一个是使用标记自身所使用的结构特征，如一个纯文本的匹配语句：

```
five
```

一个由冒号开头的MarkLogic语句：

```
:$input equals 'five'
```

和一个用/标识的正则表达式：

```
/f.ve/
```

下面是一些客户端代码的示例：

```
$markers = array(   new RegexpMarker( "/f.ve/" ),
                    new MatchMarker( "five" ),
                    new MarkLogicMarker( '$input equals "five"' )
        );

foreach ( $markers as $marker ) {
    print get_class( $marker )."\n";
    $question = new TextQuestion( "how many beans make five", $marker );
    foreach ( array( "five", "four" ) as $response ) {
        print "\tresponse: $response: ";
        if ( $question->mark( $response ) ) {
            print "well done\n";
        } else {
            print "never mind\n";
        }
    }
}
```

我们构造了3个策略对象，轮流使用它们协助构造TextQuestion对象。然后，TextQuestion对象对两个答案进行处理。

MarkLogicMarker类目前只是个占位符，它的mark()方法总是返回true，然而示例中被注释掉的代码确实能与附录B中的解析器示例或第三方解析器一起很好地工作。

下面是输出结果：

```
RegexpMarker
        response: five: well done
        response: four: never mind
MatchMarker
        response: five: well done
        response: four: never mind
MarkLogicMarker
        response: five: well done
        response: four: well done
```

记住目前MarkLogicMarker类还是一个伪类[1]。因为它一直返回true，所以无论用户怎么回答都是正确的。

在示例中，我们通过mark()方法把具体数据（$response变量）从客户端传给策略对象（Marker对象）。有时候你并不知道策略对象需要多少信息来调用其中的方法。这时你可以将客户端对象实例传递给策略对象，将需要获得什么数据的决定权委托给策略对象来完成，然后策略对象可以查询客户端来创建客户端代码所需要的数据。

11.3 观察者模式

正交性是我们之前讨论过的一个话题。程序员的目标应该是创建在改动和转移时对其他组件影响最小的组件。如果你对一个组件所做的改变会引起代码库其他地方的一连串改变，那么开发任务会很快变成一个产生bug和消除bug的恶性循环。

当然，正交性通常只是一个梦想。系统中的组件必然包含着对其他组件的引用，然而你可以使用不同的策略来使引用尽量减少。我们已经见过很多多态的示例，有时候客户端代码被规定使用某种特定类型的组件接口，但仍可以在运行时改变使用的具体组件。

在某些情况下，你可能希望进一步减弱各组件之间的联系。下面假设有一个负责处理用户登录的类：

```
class Login {
    const LOGIN_USER_UNKNOWN = 1;
    const LOGIN_WRONG_PASS = 2;
    const LOGIN_ACCESS = 3;
    private $status = array();

    function handleLogin( $user, $pass, $ip ) {
        switch ( rand(1,3) ) {
            case 1:
                $this->setStatus( self::LOGIN_ACCESS, $user, $ip );
                $ret = true;
                break;
            case 2:
                $this->setStatus( self::LOGIN_WRONG_PASS, $user, $ip );
                $ret = false;
                break;
            case 3:
                $this->setStatus( self::LOGIN_USER_UNKNOWN, $user, $ip );
                $ret = false;
                break;
        }
        return $ret;
    }

    private function setStatus( $status, $user, $ip ) {
        $this->status = array( $status, $user, $ip );
    }
```

① dummy，空有其形，没有实际功能。——译者注

```
    function getStatus() {
        return $this->status;
    }

}
```

当然在一个实际的例子中，handleLogin()方法会使用存储机制来验证用户数据。但在我们的示例中，这个类用rand()函数来模拟登录过程。调用handleLogin()有3个潜在的结果。状态标签会被设置为LOGIN_ACCESS、LOGIN_WRONG_PASS或LOGIN_USER_UNKNOWN。

因为Login类是保护业务团队财富的大门，所以在开发阶段和之后的日子它会一直受到关注。例如市场部门可能会要求你将用户登录的IP地址记入日志。你也可以调用Logger类：

```
function handleLogin( $user, $pass, $ip ) {
    switch ( rand(1,3) ) {
        case 1:
            $this->setStatus( self::LOGIN_ACCESS, $user, $ip );
            $ret = true;
            break;
        case 2:
            $this->setStatus( self::LOGIN_WRONG_PASS, $user, $ip );
            $ret = false;
            break;
        case 3:
            $this->setStatus( self::LOGIN_USER_UNKNOWN, $user, $ip );
            $ret = false;
            break;
    }
    Logger::logIP( $user, $ip, $this->getStatus() );
    return $ret;
}
```

考虑到安全问题，系统管理员可能会要求有用户登录失败时发一封邮件到管理员邮箱。你可以再一次回到登录方法并添加一个新的调用：

```
    if ( ! $ret ) {
        Notifier::mailWarning( $user, $ip,
                               $this->getStatus()  );
    }
```

另外，业务开发团队可能会要求特别关注某个ISP，并要求当特殊用户登录时设置一个cookie。实际开发中以上这样的要求可能有不少。

这些都是非常容易满足的要求，但如果直接在代码中加入功能来满足需求，会破坏我们的设计。Login类很快会紧紧嵌入到这个特殊的系统中。如果没有逐行仔细检查代码，然后移除特别针对旧系统的一切代码，就无法把它提取出来放置到其他产品中。当然这未必很难，但是我们会因此走上剪切粘贴代码的开发道路。既然系统里有两个相似但又不同的Login类，那么我们会发现如果修改任一个Login类的代码，另一个类也需要进行修改，最后我们可能不得不放弃对这两个类的同步，因为修改太麻烦了。

那么我们该怎么来拯救这个Login类呢？观察者模式用在这里最合适不过了。

实现

观察者模式的核心是把客户元素（观察者）从一个中心类（主体）中分离开来。当主体知道事件发生时，观察者需要被通知到。同时，我们并不希望将主体与观察者之间的关系进行硬编码。

为了达到这个目的，我们可以允许观察者在主体上进行注册。Login类有3个新方法——attach()、detach()和notify()，并强制它使用Observable接口：

```
interface Observable {
    function attach( Observer $observer );
    function detach( Observer $observer );
    function notify();
}

// ... Login类
class Login implements Observable {

    private $observers;
    //...

    function __construct() {

        $this->observers = array();

    }


    function attach( Observer $observer ) {
        $this->observers[] = $observer;
    }

    function detach( Observer $observer ) {
        $newobservers = array();
        foreach ( $this->observers as $obs ) {
            if (  ($obs !== $observer) ) {
                $newobservers[]=$obs;
            }
        }
        $this->observers = $newobservers;
    }

    function notify() {
        foreach ( $this->observers as $obs ) {
            $obs->update( $this );
      }
  }
//...
```

现在Login类管理着一系列观察者对象。这些观察者可以由第三方通过attach()方法添加进Login类，也可以通过detach()来移除。notify()方法用来告诉观察者一些相关事情发生了。notify()方法会遍历观察者列表，调用每个观察者的update()方法。

Login类在它的handleLogin()方法中调用notify()方法:

```
function handleLogin( $user, $pass, $ip ) {
        switch ( rand(1,3) ) {
            case 1:
                $this->setStatus( self::LOGIN_ACCESS, $user, $ip );
                $ret = true; break;
            case 2:
                $this->setStatus( self::LOGIN_WRONG_PASS, $user, $ip );
                $ret = false; break;
            case 3:
                $this->setStatus( self::LOGIN_USER_UNKNOWN, $user, $ip );
                $ret = false; break;
        }
        $this->notify();
        return $ret;
    }
```

然后让我们定义Observer接口:

```
interface Observer {
    function update( Observable $observable );
}
```

任何实现这个接口的对象都可以通过attach()方法加入Login类中。下面创建一个具体的实例:

```
class SecurityMonitor extends Observer {
    function update( Observable $observable ) {
        $status = $observable->getStatus();
        if ( $status[0] == Login::LOGIN_WRONG_PASS ) {
            //发送邮件给系统管理员
            print __CLASS__.":\tsending mail to sysadmin\n";
        }
    }
}
$login = new Login();
$login->attach( new SecurityMonitor() );
```

注意一下这个观察者对象是如何使用Observable实例来获取更多事件信息的。主体类决定了是否给观察者提供查询状态的方法。在上面的实例中，我们定义了一个叫做getStatus()的方法，该方法可供观察者调用来获取玩家的当前状态。

但我们添加的这个方法依然存在问题。要调用Login::getStatus()，SecurityMonitor类就必须了解更多信息①。虽然调用发生于一个Observable对象上，但无法保证该对象也是一个Login对象。这里我们有几种办法来解决这个问题。我们可以扩展Observable接口并在其中添加getStatus()方法，或者将其重命名为如ObservableLogin这样的名称，以此说明该对象是特定类型的Login对象。

还有一个办法：继续保持Observable接口的通用性，由Observer类负责保证它们的主体是

① 必须确定调用对象中存在getStatus()方法，否则调用会出错。——译者注

正确的类型。它们甚至能将自己添加到主体上。因为将会存在多个Observer类型的对象，而且我们计划执行一些它们共有的任务，所以先创建一个抽象超类：

```
abstract class LoginObserver implements Observer {
    private $login;
    function __construct( Login $login ) {
        $this->login = $login;
        $login->attach( $this );
    }

    function update( Observable $observable ) {
        if ( $observable === $this->login ) {
            $this->doUpdate( $observable );
        }
    }

    abstract function doUpdate( Login $login );
}
```

LoginObserver类的构造函数需要一个Login对象作为参数。LoginObserver保存对Login对象的引用，并且调用Login::attach()方法。当update()被调用时，LoginObserver会检查参数传入的Observable对象是否是正确的引用，然后LoginObserver会调用模板方法doUpdate()。现在我们可以创建一批LoginObserver对象，它们能够判断使用的是Login对象，而不是任意Observable对象：

```
class SecurityMonitor extends LoginObserver {
    function doUpdate( Login $login ) {
        $status = $login->getStatus();
        if ( $status[0] == Login::LOGIN_WRONG_PASS ) {
            // 发送邮件给系统管理员
            print __CLASS__.":\tsending mail to sysadmin\n";
        }
    }
}

class GeneralLogger  extends LoginObserver {
    function doUpdate( Login $login ) {
        $status = $login->getStatus();
        // 记录登录数据到日志
        print __CLASS__.":\tadd login data to log\n";
    }
}

class PartnershipTool extends LoginObserver {
    function doUpdate( Login $login ) {
        $status = $login->getStatus();
        // 检查IP地址
        // 如果匹配列表，则设置cookie
        print __CLASS__.":\tset cookie if IP matches a list\n";
    }
}
```

11

当实例化对象时，创建和添加LoginObserver类的任务就完成了：

```
$login = new Login();
new SecurityMonitor( $login );
new GeneralLogger( $login );
new PartnershipTool( $login );
```

因此我们在主体类和观察者之间创建了一个很灵活的关系，其类图如图11-6所示。

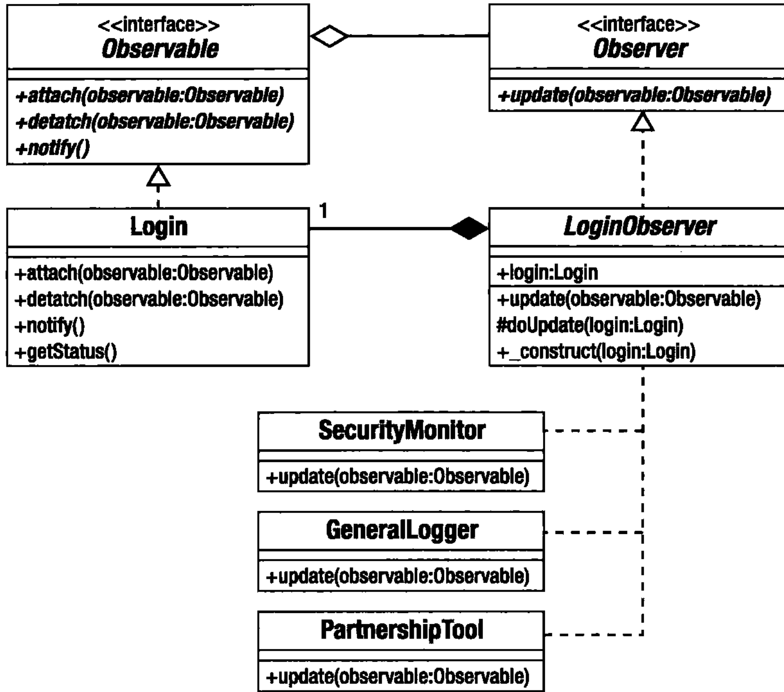


图11-6　观察者模式

PHP通过内置的SPL（Standard PHP Library，PHP标准类）扩展提供了对观察者模式的原生支持。SPL是一套可以帮助PHP程序员处理很多面向对象问题的工具，堪称是一把面向对象的瑞士军刀。其中的观察者（Observer）由3个元素组成：SplObserver、SplSubject和SplObjectStorage。SplObserver和SplSubject都是接口，与之前示例中的Observer和Observable接口完全相同。SplObjectStorage是一个工具类，用于更好地存储对象和删除对象。下面是改进过的示例代码：

```
class Login implements SplSubject {
    private $storage;
    //...

    function __construct() {
        $this->storage = new SplObjectStorage();
    }
```

```
    function attach( SplObserver $observer ) {
        $this->storage->attach( $observer );
    }

    function detach( SplObserver $observer ) {
        $this->storage->detach( $observer );
    }

    function notify() {
        foreach ( $this->storage as $obs ) {
            $obs->update( $this );
        }
    }
    //...
}

abstract class LoginObserver implements SplObserver {
    private $login;
    function __construct( Login $login ) {
        $this->login = $login;
        $login->attach( $this );
    }

    function update( SplSubject $subject ) {
        if ( $subject === $this->login ) {
            $this->doUpdate( $subject );
        }
    }

    abstract function doUpdate( Login $login );
}
```

使用SplObserver和SplSubject与使用我们自定义的Observer和Subject没有什么区别，当然我们不需要再声明那些接口，并且我们必须根据新的类型名称来修改类型提示，而SplObjectStorage的功能非常有用。在最初的示例中，Login::detach()方法对$observable数组应用了array_udiff（以及一个匿名函数），查找并移除参数对象。SplObjectStorage类可以在幕后为我们完成这些工作。它实现了attach()和detach()方法并且可被传给foreach语句，而且可以像一个数组一样被迭代。

注解 你可以在http://www.php.net/spl获得更多关于SPL的信息。特别值得一提的是，你在那里可以发现各种类型的迭代器。我们将在第13章中介绍PHP内置的迭代器。

解决这个问题的另一个办法是通过update()方法传递特定的状态信息，而不传递主体类的对象实例。作为一个快速解决方案，这是我在项目初始阶段常采用的。因此在示例中，update()需要的是一个状态标志、用户名以及IP地址（为了方便移植也可以将这几项数据放在一个数组中），而不是一个Login实例。这样我们不再需要为Login类编写一个状态方法。另一方面，如果传递主体类的对象实例到update()，因为主体类中存储了很多状态信息，所以传递其实例可使

得观察者更具灵活性。

你也可以严格限定类型，让Login类拒绝与其他任何类型一起工作，只接收观察者类的具体类型（如LoginObserver）。如果你决定这么干，那么也许需要考虑对传给attach()方法的对象进行某些运行时检查，否则你可能需要考虑同时改变Observable接口。

这里我们又一次在运行时使用“组合”的思想来建立一个灵活和可扩展的模型。Login类可以无条件地从上下文环境中提取出来并放入另一个项目中。在新项目中，它可以和一批新的观察者一起工作。

11.4　访问者模式

可以看到，许多模式遵循“组合比继承更灵活”的原则，在代码运行时建立一定的对象结构。无处不在的组合模式便是一个很好的示例。当使用对象集合时，你可能需要对结构上每一个单独的组件应用各种操作。这样的操作可以内建于组件本身，毕竟在组件内部调用其他组件是最方便的。

但这种方式也存在问题，因为你并不知道所有可能需要执行的操作。如果每增加一个操作，就在类中增加一个对于新操作的支持，类就会变得越来越臃肿。访问者模式可以解决这个问题。

11.4.1　问题

回想一下上一章介绍的组合模式的示例。我们在游戏中创建了可以由战斗单元组成的军队，它的整体和局部可以互换处理。我们也看到了操作可以在组件内部实现。一般情况下，局部对象自己会执行操作，而组合对象则会调用它们的子对象来执行操作。

```
class Army extends CompositeUnit {

    function bombardStrength() {
        $ret = 0;
        foreach( $this->units() as $unit ) {
            $ret += $unit->bombardStrength();
        }
        return $ret;
    }
}

class LaserCannonUnit extends Unit {
    function bombardStrength() {
        return 44;
    }
}
```

如果操作很容易整合到组合类中，那么这样做没有任何问题。但是，还有更多的周边任务会使接口并不是那么够用。

这里有个转储叶节点的文本信息的操作，该操作会被添加到抽象类Unit中。

```
// Unit
function textDump( $num=0 ) {
```

```
        $ret = "";
        $pad = 4*$num;
        $ret .= sprintf( "%{$pad}s", "" );
        $ret .= get_class($this).": ";
        $ret .= "bombard: ".$this->bombardStrength()."\n";
        return $ret;
    }
```

然后这个方法可以在CompositeUnit类里被覆盖：

```
    // CompositeUnit
    function textDump( $num=0 ) {
        $ret = parent::textDump( $num );
        foreach ( $this->units as $unit ) {
            $ret .= $unit->textDump( $num + 1 );
        }
        return $ret;
    }
```

我们可能还需要继续创建统计树中单元个数的方法、保存组件到数据库的方法和计算军队的食物消耗的方法。

为什么我们要在Composite的接口中加入这些方法呢？只有一个答案有说服力：添加这些不同的操作有利于在组合结构中较为轻松地访问相关节点。

虽然可以轻松遍历的确是组合模式的一大优势，但并非每个需要遍历对象树的操作都要在Composite接口中占据位置。

因此工作中的重点便是充分利用对象结构提供的轻松遍历的优势，但同时避免类过度膨胀。

11

11.4.2 实现

让我们从接口开始。首先在Unit抽象类中定义accept()方法：

```
    function accept( ArmyVisitor $visitor ) {
        $method = "visit".get_class( $this );
        $visitor->$method( $this );
    }

    protected function setDepth( $depth ) {
        $this->depth=$depth;
    }

    function getDepth() {
        return $this->depth;
    }
```

可以看到，accept()方法要求一个ArmyVisitor对象作为参数。PHP允许我们在ArmyVisitor上动态定义一个我们希望调用的方法。这让我们不必在类的继承体系中每一个叶节点上实现accept()。代码中接着添加了两个好用的getDepth()和setDepth()方法，这两个方法可在树中类获取和设置一个单元的深度。父单元通过CompositeUnit::addUnit()添加子单元时，setDepth()方法会被调用。

```
function addUnit( Unit $unit ) {
    foreach ( $this->units as $thisunit ) {
        if ( $unit === $thisunit ) {
            return;
        }
    }
    $unit->setDepth($this->depth+1);
    $this->units[] = $unit;
}
```

接下来，我们在抽象的组合类中定义另一个accept()方法：

```
function accept( ArmyVisitor $visitor ) {
    $method = "visit".get_class( $this );
    $visitor->$method( $this );
    foreach ( $this->units as $thisunit ) {
        $thisunit->accept( $visitor );
    }
}
```

这个accept()方法和Unit::accept()方法基本一样，但是多了一些内容。它根据当前类的名称构造了一个方法名称，然后通过传入的ArmyVisitor对象来调用对应的方法。因此如果当前类是Army，则该方法调用ArmyVisitor::visitArmy()；如果当前类是TroopCarrier，则调用ArmyVisitor::visitTroopCarrier()；依此类推。在此之后，accept()方法会遍历调用accept()方法的所有子对象。事实上，因为accept()覆盖了父类中的同名方法，所以这里我们可以去除重复代码：

```
function accept( ArmyVisitor $visitor ) {
    parent::accept( $visitor );
    foreach ( $this->units as $thisunit ) {
        $thisunit->accept( $visitor );
    }
}
```

用这种方式消除重复代码还是非常令人满意的，尽管在本例中我们只节省了一行代码。在这两个例子中，accept()方法都允许我们做两件事情：

- 为当前组件调用正确的访问者方法；
- 通过accept()方法将访问者对象传递给当前对象元素所有的子元素（假设当前组件是组合体）。

我们还需要定义ArmyVisitor接口。accept()方法应该会给你一些提示。访问者类也应该为继承体系中的每一个具体类定义一个accept()方法。这样我们就能为不同的对象提供不同的功能。下面的类中定义了一个默认的visit()方法，当类中没有为特定的Unit类提供特殊处理时，该方法会被自动调用。

```
abstract class ArmyVisitor  {
    abstract function visit( Unit $node );

    function visitArcher( Archer $node ) {
        $this->visit( $node );
```

```
    }

    function visitCavalry( Cavalry $node ) {
        $this->visit( $node );
    }

    function visitLaserCannonUnit( LaserCannonUnit $node ) {
        $this->visit( $node );
    }

    function visitTroopCarrierUnit( TroopCarrierUnit $node ) {
        $this->visit( $node );
    }

    function visitArmy( Army $node ) {
        $this->visit( $node );
    }
}
```

因此，现在剩余的工作只是给ArmyVisitor提供具体实现。下面是ArmyVisitor对象的一个实现，用于转储文本：

```
class TextDumpArmyVisitor extends ArmyVisitor {
    private $text="";

    function visit( Unit $node ) {
        $ret = "";
        $pad = 4*$node->getDepth();
        $ret .= sprintf( "%{$pad}s", "" );
        $ret .= get_class($node).": ";
        $ret .= "bombard: ".$node->bombardStrength()."\n";
        $this->text .= $ret;
    }

    function getText() {
        return $this->text;
    }
}
```

让我们来看看客户端调用代码，然后走一下整个流程：

```
$main_army = new Army();
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );
$main_army->addUnit( new Cavalry() );

$textdump = new TextDumpArmyVisitor();
$main_army->accept( $textdump  );
print $textdump->getText();
```

以上代码执行后的结果如下：

```
Army: bombard: 50
    Archer: bombard: 4
```

```
LaserCannonUnit: bombard: 44
Cavalry: bombard: 2
```

我们先创建一个Army对象。因为Army是组合体，所以我们用它的addUnit()方法加入了更多Unit对象。然后我们创建TextDumpArmyVisitor对象，并把它传给Army::accept()。accept()方法构造并调用TextDumpArmyVisitor::visitArmy()方法。在这里，我们没有给Army对象提供什么特殊的处理，因此调用通用的visit()方法，并将对Army对象的引用传给visit()。visit()会调用Army对象的方法（包括一个新的getDepth()方法，它用于获得该对象在对象体系中的深度）来产生摘要数据。这时完成对visitArmy()的调用，接着Army::accept()操作会轮流调用子对象的accept()，同时将访问者传递给该方法。这样，ArmyVisitor类会访问对象树中的每一个对象。

就这样，我们创建了一种新的机制。现在只需添加几个方法，新功能就可以方便地添加到组合类中，而不需要包含它们的接口，也不会产生大量重复的遍历代码。

为了体现游戏的公平性，军队需要缴纳税金。征税者访问军队，并向找到的每个单位征税。不同单位的税率不同。下面的代码可以展示利用访问者类中特定方法的优势：

```
class TaxCollectionVisitor extends ArmyVisitor {
    private $due=0;
    private $report="";

    function visit( Unit $node ) {
        $this->levy( $node, 1 );
    }

    function visitArcher( Archer $node ) {
        $this->levy( $node, 2 );
    }

    function visitCavalry( Cavalry $node ) {
        $this->levy( $node, 3 );
    }

    function visitTroopCarrierUnit( TroopCarrierUnit $node ) {
        $this->levy( $node, 5 );
    }

    private function levy( Unit $unit, $amount ) {
        $this->report .= "Tax levied for ".get_class( $unit );
        $this->report .= ": $amount\n";
        $this->due += $amount;
    }

    function getReport() {
        return $this->report;
    }

    function getTax() {
        return $this->due;
```

```
    }
}
```

在这个简单的示例中，我们没有直接将Unit对象传给各种visit方法，而是利用了这些方法的特殊特性来向不同类型的Unit对象征收不同的费用。

下面是客户端代码：

```
$main_army = new Army();
$main_army->addUnit( new Archer() );
$main_army->addUnit( new LaserCannonUnit() );
$main_army->addUnit( new Cavalry() );
$taxcollector = new TaxCollectionVisitor();
$main_army->accept( $taxcollector );
print "TOTAL: ";
print $taxcollector->getTax()."\n";
```

TaxCollectionVisitor像之前一样被传给Army对象的accept()方法。Army在调用其子对象的accept()之前，先将对自己的一个引用传给visitArmy()方法。组件并不知道它们的访问者所执行的操作。它们通过公共的接口简单协作，使每个组件都会正确地将自己传给对应自身类型的方法。

除了在ArmyVisitor类中定义的方法以外，TaxCollectionVisitor还提供了两个摘要方法：getReport()和getTax()。调用这两个方法会得到你期望的数据：

```
Tax levied for Army: 1
Tax levied for Archer: 2
Tax levied for LaserCannonUnit: 1
Tax levied for Cavalry: 3
TOTAL: 7
```

图11-7展示了示例中的各参与者。

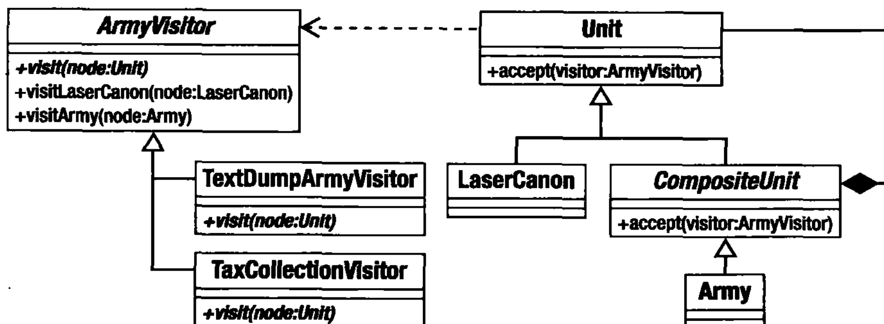


图11-7 访问者模式

11.4.3 访问者模式的问题

访问者模式也是一个简单实用的模式，然而使用这个模式时有些地方需要注意。

首先，虽然完美地符合组合模式，但事实上访问者可以用于任何对象集合。举例来说，你可

以把访问者用于每个对象都保存对兄弟节点引用的一组对象。

其次，外部化操作可能破坏封装。也就是说，你可能需要公开被访问对象的内部来让访问者能对它们做任何有用的操作。例如，在第一个访问者例子中，为了给TextDumpArmyVisitor对象提供信息，我们被迫给Unit接口提供了一个额外的方法。我们也在观察者模式中见过这种情况。

由于迭代（遍历）与访问者对象执行的操作是分离的，你必须在一定程度上放开控制力度。比如，要创建一个在子节点迭代前后都能工作的visit()方法不太容易。一个解决方法就是把迭代的职责转移到访问者对象上。但这样做的问题是你可能要在所有访问者中重复遍历代码。

通常我更倾向于在被访问类内部进行遍历，但外部遍历操作又具有不可替代的优势，你可以在不同的访问者中使用不同的方式来处理被访问类。

11.5　命令模式

近几年，我的每个项目几乎都用到了命令模式。命令模式最初来源于图形化用户界面设计，但现在广泛应用于企业应用设计，特别促进了控制器（请求和分发处理）和领域模型（应用逻辑）的分离。说得更简单一点，命令模式有助于系统更好地进行组织，并易于扩展。

11.5.1　问题

所有系统都必须决定如何响应用户请求。在PHP中，这个决策过程通常是由分散的各个PHP页面来处理。比如当用户访问一个PHP页面（如feedback.php）时，用户明确地告诉系统他所要求的功能和接口。但现在PHP开发者日益倾向于在设计系统时采用单一入口的方式（我们将在下一章中讨论）。无论是多个入口还是单个入口，接收者都必然将用户请求委托给一个更加关注于应用逻辑的层来进行处理。这个委托在用户请求不同页面时尤为重要。如果没有委托，代码重复将会不可避免地蔓延在整个项目中。

让我们想象一下，假设一个有很多任务要执行的项目，需要允许某些用户登录，某些用户可以提交反馈。我们可以分别创建login.php和feedback.php页面来处理这些任务，并实例化专门的类以完成任务。不过遗憾的是，系统中的用户界面很难被精确地一一对应到系统任务。比如我们可能要求每个页面都有登录和反馈的功能。如果页面必须处理很多不同的任务，就应该考虑将任务进行封装。封装之后，向系统增加新任务就会变得简单，并且可以将系统中的各部分分离开来。当然，这时我们可以使用命令模式。

11.5.2　实现

命令对象的接口极为简单，因为它只要求实现一个方法execute()。

在图11-8中，Command被定义为一个抽象类。同样简单地，它也可以被定义为接口。我喜欢将它定义为抽象类，因为有时基类也可以为它的衍生对象提供有用的公共功能。

图11-8　Command类

命令模式由3部分组成：实例化命令对象的客户端（client）、部署命令对象的调用者（invoker）和接受命令的接收者（receiver）。

通过客户端，接收者可以在命令对象的构造方法中被传递给命令对象，或者通过某种工厂对象被获得。相对而言，我更喜欢后一种办法，它可以保持构造方法参数清晰明了，而且所有的Command对象都可以用完全相同的方式实例化。

让我们创建一个具体的Command类：

```
abstract class Command {
    abstract function execute( CommandContext $context );
}

class LoginCommand extends Command {
    function execute( CommandContext $context ) {
        $manager = Registry::getAccessManager();
        $user = $context->get( 'username' );
        $pass = $context->get( 'pass' );
        $user_obj = $manager->login( $user, $pass );
        if ( is_null( $user_obj ) ) {
            $context->setError( $manager->getError() );
            return false;
        }
        $context->addParam( "user", $user_obj );
        return true;
    }
}
```

LoginCommand被设计为与AccessManager（访问管理器）对象一起工作。AccessManager是一个虚构出来的类，它的任务就是处理用户登录系统的具体细节。注意Command::execute()方法要求使用CommandContext对象作为参数（《J2EE核心模式》中将其描述为RequestHelper）。通过CommandContext机制，请求数据可被传递给Command对象，同时响应也可以被返回到视图层。以这种方式使用对象是很有好处的，因为我们可以不破坏接口就把不同的参数传递给命令对象。从本质上说，CommandContext只是将关联数组变量包装而成的对象，但我们仍会经常扩展它来执行额外的任务。下面是一个简单的CommandContext实现：

```
class CommandContext {
    private $params = array();
    private $error = "";

    function __construct() {
        $this->params = $_REQUEST;
    }

    function addParam( $key, $val ) {
        $this->params[$key]=$val;
    }

    function get( $key ) {
        return $this->params[$key];
    }

    function setError( $error ) {
        $this->error = $error;
```

```
    }

    function getError() {
        return $this->error;
    }
}
```

因此通过使用CommandContext对象，LoginCommand能够访问请求数据：提交的用户名和密码。我们使用了一个简单的类Registry，它带有用于生成通用对象的静态方法，可以返回LogicCommand所需要的AccessManager对象。如果AccessManager报告一个错误，则LoginCommand保存错误信息到CommandContext对象中以供表现层使用并返回false。如果一切正常，LoginCommand只返回true。注意Command对象不应该执行太多的逻辑。它们应该负责检查输入、处理错误、缓存对象和调用其他对象来执行一些必要的操作。如果你发现应用逻辑过多地出现在Command类中，通常需要考虑重构代码。这样的代码会导致代码重复，因为它们不可避免地会在不同的Command类中被复制粘贴。你至少需要考虑这些应用逻辑的功能应该属于哪部分代码。最好把这样的代码迁移到业务对象中或者放入一个外观层中。现在我们仍然缺少客户端代码（即用于创建命令对象的类）及调用者类（使用生成的命令的类）。在一个Web项目中，选择实例化哪个命令对象的最简单的办法是根据请求本身的参数来决定。下面是一个简化的客户端代码：

```
class CommandNotFoundException extends Exception {}

class CommandFactory {
    private static $dir = 'commands';

    static function getCommand( $action='Default' ) {
        if ( preg_match( '/\W/', $action ) ) {
            throw new Exception("illegal characters in action");
        }
        $class = UCFirst(strtolower($action))."Command";
        $file = self::$dir.DIRECTORY_SEPARATOR."{$class}.php";
        if ( ! file_exists( $file ) ) {
            throw new CommandNotFoundException( "could not find '$file'" );
        }
        require_once( $file );
        if ( ! class_exists( $class ) ) {
            throw new CommandNotFoundException( "no '$class' class located" );
        }
        $cmd = new $class();
        return $cmd;
    }
}
```

CommandFacotry类在commands目录里查找特定的类文件。文件名是通过CommandContext对象的$action参数来构造的，该参数是从请求中被传到系统中的。如果文件和类都存在，那么会返回命令对象给调用者。我们可以在这里添加更多的错误检查，比如保证找到的类是Command类的子类，保证构造方法没有参数等，但目前的版本对我们来说已经足够说明问题。这种方式的优点是你可以随时将新的Command类添加到commands目录下，然后系统便立即支持它了。

下面是一个简单的调用者：

```
class Controller {
    private $context;
    function __construct() {
        $this->context = new CommandContext();
    }

    function getContext() {
        return $this->context;
    }

    function process() {
        $cmd = CommandFactory::getCommand( $this->context->get('action') );
        if ( ! $cmd->execute( $this->context ) ) {
            // 处理失败
        } else {
            // 成功
            // 现在分发视图
        }
    }
}

$controller = new Controller();
// 伪造用户请求
$context = $controller->getContext();
$context->addParam('action', 'login' );
$context->addParam('username', 'bob' );
$context->addParam('pass', 'tiddles' );
$controller->process();
```

在调用Controller::process()之前，我们通过在控制器的构造函数中实例化的CommandContext对象上设置参数伪造了一个Web请求。process()方法将实例化命令对象的工作委托给CommandFactory对象，然后它在返回的命令对象上调用execute()方法。注意，控制器对命令内部是一无所知的。因为命令执行的细节与控制器是相互独立的，所以我们可以随时添加新的Command类而对当前的结构影响很小。

让我们再创建一个Command类：

```
class FeedbackCommand extends Command {

    function execute( CommandContext $context ) {
        $msgSystem = Registry::getMessageSystem();
        $email = $context->get( 'email' );
        $msg   = $context->get( 'msg' );
        $topic = $context->get( 'topic' );
        $result = $msgSystem->send( $email, $msg, $topic );
        if ( ! $result ) {
            $context->setError( $msgSystem->getError() );
            return false;
        }
        return true;
    }
}
```

11

注解　我们将在第12章中再次讨论命令模式，并实现一个更完整的Command工厂类。本节实现的框架实际上是另外一个模式——前端控制器模式的简化版本。

当这个类以FeedbackCommand.php的文件名来保存，并保存在正确的Commands目录下时，它就会被调用来响应Action为feedback的请求，而不需要对控制器或者CommandFactory做任何修改。

图11-9展示了命令模式的各个部分。

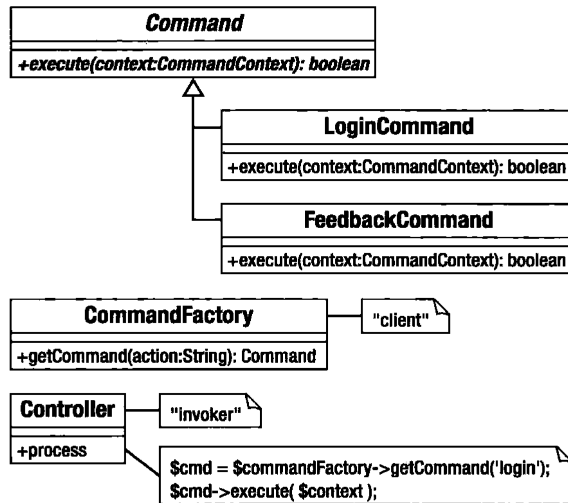


图11-9　命令模式的各个部分

11.6　小结

在本章中，我们结束了对《设计模式》一书中模式的研究。我们用解释器模式设计了一个迷你语言并创建了该语言的引擎；使用了策略模式（另一种使用组合的方式）来增加系统灵活性并减少对于重复子类的需要；使用了观察者模式解决了将系统事件通知到多个不同组件的问题；也重新回顾了组合模式的示例，并通过访问者模式学习了如何在对象树中访问每个节点元素，并对它们执行多种操作；最后看到了命令模式如何帮助我们建立一个具有扩展性的多层系统。

在下一章中，我们会介绍《设计模式》一书中未提及的一些模式，特别是面向企业应用的设计模式。

第12章

企业模式

PHP是一个为Web开发而设计的语言。在PHP 5之后，PHP开始大力支持对象，因此你可以享受到设计模式带来的好处，就像使用其他面向对象语言（特别是Java）那样。

本章将举一个简单的例子来说明设计模式的使用。请注意，选择使用某种模式时，并不一定也要使用与该模式配合良好的其他所有模式，而且示例中介绍的部署这些模式的方法也不是唯一可行的方法。示例主要用于帮助理解模式的核心思想，你可以从中提取自己需要的内容，并应用于实际开发当中。

本章介绍的内容很多，是全书最长和最复杂的章节，相信读者很难一次性读完。本章分为一个简要介绍和两个主要部分。你可以在读完其中的某部分时休息一下。

12.1节中介绍了一些独立的模式。尽管这些内容有时是相互关联的，但可以直接跳到任何一个你想了解的模式进行阅读和学习，在你有空的时候再阅读其他相关模式的内容。

本章包括以下内容。

- 架构概述：企业应用程序分层。
- 注册（Registry）模式：管理应用程序数据。
- 表现层：管理和响应用户请求，并把数据呈现给用户。
- 业务逻辑层：处理系统的真实任务——解决业务问题。

12.1 架构概述

因为涉及的内容比较广泛，所以首先概述一下模式，然后介绍如何构建分层的应用程序。

12.1.1 模式

下面是本章将要讨论的设计模式。你可以从头看到尾，也可以根据需要和兴趣选择性地阅读。注意命令模式没有在此单独介绍（在第11章介绍过），但会在前端控制器和应用控制器模式中提及。

- 注册表：该模式用于使数据对进程中所有的类都有效。通过谨慎的序列化操作，注册表对象可以用于存储跨会话甚至跨应用程序实例的数据。
- 前端控制器：在规模较大的系统中，该模式可用于尽可能灵活地管理各种不同的命令和视图。

- 应用控制器：创建一个类来管理视图逻辑和命令选择。
- 模板视图：创建模板来处理和显示用户界面，在显示标记中加入动态内容。尽量少使用原始代码[①]。
- 页面控制器：页面控制器满足和前端控制器相同的需求，但较为轻量级，灵活性也小一些。如果想快速得到结果而且系统也不太复杂的话，可以使用这种模式管理请求和处理页面逻辑。
- 事务脚本：如果想要快速完成某个任务，可以使用本模式。通过简单的规划，用"过程式"的代码来实现程序逻辑。但本模式的可伸缩性不佳。
- 领域模型：和事务脚本相反，使用本模式可以为业务参与者和过程构建基于对象的模型。

12.1.2　应用程序和层

本章大部分模式是用来使程序中不同的"层"（tier，也称为layer）独立工作的。就像类的使命是执行特定的任务，企业应用系统中的层也是如此，不过更为粗旷。图12-1展示了一个系统中分工明确的各个层。

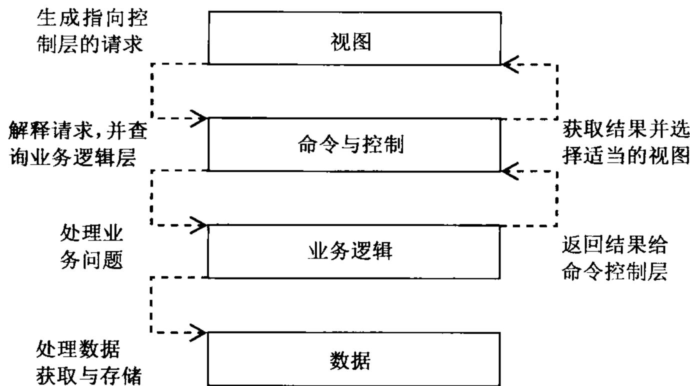


图12-1　一个典型企业系统中的各个层

图12-1所示的结构并不是固定不变的，其中一些层可以合并，而且层之间的交互策略可能根据系统的复杂度而不同。无论如何，图12-1展示的模型强调灵活性和重用，而许多企业应用就是根据"灵活"和"重用"的原则进行扩展的。

- 视图层包括系统用户实际看到和交互的界面。它负责显示用户请求的结果及传递新的请求给系统。
- 命令和控制层处理用户的请求。它委托业务逻辑层处理和满足请求，然后选择最适合的视图，将结果显示给用户。实际上，这个层和视图层常常合并为表现层。即使这样，显示的任务应当严格地与请求处理和业务逻辑调用分离开来。
- 业务逻辑层负责根据请求执行业务操作。它执行需要的计算并整理结果数据。

① 执行业务操作的代码。——译者注

- 数据层负责保存和获取系统中的持久信息。在某些系统中，命令和控制层使用数据层来获取它所需要的业务对象。但在其他系统中，数据层通常尽可能地被隐藏起来。

那么为什么要用这种方式划分系统呢？答案是解耦（decoupling）。通过分离业务逻辑层与视图层，当添加新的接口到系统时，系统内部只需要做很小的改动。

假设有一个管理事件列表的系统（读到本章末尾，你将会非常熟悉这个例子）。终端用户需要一个漂亮的HTML接口，而维护系统的管理员可能需要一个命令行接口来构建自动化系统，同时，你可能需要开发支持手机和其他手持设备访问的版本，甚至可能考虑使用REST式API或SOAP等协议。

如果你以前把系统的底层逻辑和HTML视图层混合在一起(尽管这种写法备受批评,但在PHP项目中依然很普遍)，上面所提的这些需求将会让你不得不重写代码。另一方面，如果已经创建了分层的系统，就可以直接使用新的显示方案而不用重新考虑业务逻辑和数据层。

同样，项目的持久性策略也可能改变。你应该能够在对系统的其他层影响最小的情况下更换存储模型。

将系统分层的另一个原因是测试。Web应用程序是很难测试的。任何一种自动测试在需要在一端解析HTML接口并在另一端使用在线数据库时都会很为难。也就是说，测试工作必须运行在完全部署的系统上，并且冒着破坏本应受保护的真实系统的风险。在分层系统中，任何需要与其他层直接打交道的类通常都扩展自抽象父类或者实现了同一个接口。这个父类型可以支持多态。在测试环境中，一个完整的层可以被一组虚拟的对象（通常称为stub或mock对象）所代替。例如，通过这种方法，我们可以使用虚拟的数据层来测试业务逻辑层。你可以在第18章读到更多关于测试的内容。

即使系统只有一个简单的接口，并且你觉得测试是多余的时候，分层仍是非常有用的。通过创建独立分工的层，可以构建一个易于扩展和调试的系统。将具有同样功能的代码放在同一个地方可以减少代码重复（而不是将系统和数据库调用或显示方案绑定在一起），因此添加功能到系统中会相对简单，因为这些改变是纵向而不是横向的。

在分层系统中，一个新功能可能需要一个新的接口组件、额外的请求处理、更多的业务逻辑和对存储机制的修改。这些修改是纵向的。在一个没有分层的系统中，如果要增加新的功能，则可能需要记住5个甚至更多和数据库表相关的页面。新的接口可能会在数十个地方被调用，因此需要为系统增加这部分的代码。这就是横向修改。

当然，实际上我们并不能完全避免这种横向依赖，特别当修改页面的导航部分的时候。尽管如此，一个分层的系统有助于最小化横向修改的需要。

注解 本章中有很多模式，它们的命名和定义都来自于Martin Fowler著的《企业应用架构模式》及Alur等人著的影响力很大的《J2EE核心模式》。为了保持一致性，当这两本书对于同一模式的命名不同时，我倾向于使用Fowler的命名约定。因为Fowler的工作不只用于Java，有更广泛的应用范围。Alur等人主要关注EJB（Enterprise Java Beans），这意味着他们的很多模式特别适合于分布式的系统结构，而这对于PHP世界并不常用。

如果你认为本章有用的话，我推荐你也深入学习这两本书。即使你不懂Java，作为一个面向对象的PHP程序员，你也可以从中找到合适的例子来学习。

本章所有例子都围绕一个虚拟的事件列表系统，系统的名称叫Woo，它是What's On Outside（外头发生了什么事）的缩写。

系统由场所（venue，如剧院、俱乐部和电影院）、空间（space，如屏幕1和楼上）和事件（event，如电影*The Long Good Friday*、*The Importance of Being Earnest*）组成。

本系统的操作包括创建场所、添加空间到场所和列出系统中的所有场所。

记住，本章的目标是阐述主要的企业设计模式而不是构建一个实际系统。由于设计模式之间常常相互依赖，本章中的大部分示例也常常会互相重叠，以充分利用本章其他地方介绍的基础知识。本章的代码主要是用来解释企业模式的概念，因此无法符合实际系统中的所有标准，甚至为了简洁起见，还忽略了错误检查。读者应该把这些例子当成学习设计模式的途径，不要直接当做框架或程序中的一部分。

12.2 企业架构之外的基础模式

本书中的大部分模式都可以在企业架构的层中找到自己的位置，但是有些模式非常基础，所以将它们放在了该结构之外。注册表（Registry）模式就是这样的一个例子。实际上，注册表是跳出层约束的有效途径之一。大多数模式都只能用在某个层，但注册表是一个例外，它可以不受层的约束。

12.2.1 注册表

注册表的作用是提供系统级别的对象访问功能。我们通常把“全局变量是不好的”当做信条。不过，凡事都有两面性，全局性的数据访问非常具有吸引力。事实确实如此，面向对象架构师经常要重新设置全局变量并使用新变量名。我们在第9章中介绍了单例模式。单例对象不能被覆盖，因此它不像普通全局变量那样缺陷明显（全局变量可能在任何时候任何地方被修改，带来很大不确定性，容易造成隐性的bug），但是同样会给系统带来耦合。

即使如此，单例模式依然非常有用，很多程序员（包括我）都无法弃之不用。

问题

就像我们看到的那样，很多企业系统都被分为几个层，每个层都只通过事先定义好的通道和相邻的层交流。对层的分离使程序变得灵活，替换或修改每个层时可以最小化对系统其他部分的影响。但当你需要在一个层中获得不相邻的另一个层所需要的信息时，该怎么办呢？

下面假设我们在ApplicationHelper类中读取配置信息：

```
// woo\controller\ApplicationHelper
function getOptions() {
    if ( ! file_exists( "data/woo_options.xml" ) ) {
        throw new woo_base_AppException(
            "Could not find options file" );
    }
    $options = simplexml_load_file( "data/woo_options.xml" );
    $dsn = (string)$options->dsn;
    // 接下来怎么做呢
    // ...
}
```

获取信息很简单，但是如何让数据层也能使用这些信息呢？如何使系统的各部分都能读取这些配置信息呢？

一个办法是在系统中把这些信息从一个对象传递到另一个对象：从一个负责处理请求的控制器对象传递到业务逻辑层的对象，再传递到负责和数据库对话的对象。

这个办法是可行的。实际上，我们可以传递ApplicationHelper对象，或者是一个特定的Context对象。无论哪种方式，都是通过系统的层之间的联系将上下文信息从一个对象传递给另一个需要的对象。

折中方案是必须修改所有对象的接口，判断上下文对象是否是它们需要的。很显然，有时候这种方案会破坏松散耦合。

注册表模式提供了第三种解决方案，但该方案也存在一些问题。

注册表类提供静态方法（或单例对象的的实例化方法）来让其他对象访问其中的数据（通常是对象）。整个系统中的每个对象都可以访问这些数据对象。

Registry一词来自Fowler的《企业应用架构模式》一书，但是和所有模式一样，注册表模式的实现无处不在。David Hunt和David Thomas（*The Pragmatic Programmer*）把注册表类比作警察局的事件公布栏。一个值班的警探把案情证据和介绍贴在公布栏上，另一个换班的警探可以取下使用。我也见到过有些地方把注册表模式称为“白板”（whiteboard）或“黑板”（blackboard）。

12.2.2 实现

图12-2显示了一个Registry对象，它的作用是存储和提供Request对象。

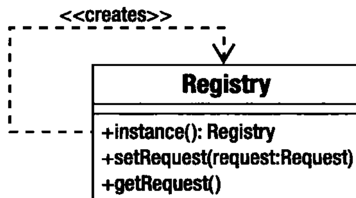


图12-2 一个简单的注册表

下面是注册表类的代码：

```
class Registry {
    private static $instance;
    private $request;

    private function __construct() { }

    static function instance() {
        if ( ! isset( self::$instance ) ) { self::$instance = new self(); }
        return self::$instance;
    }

    function getRequest() {
        return $this->request;
```

```
    }

    function setRequest( Request $request ) {
        $this->request = $request;
    }
}
// 用于测试的空类
class Request {}
```

然后你可以在系统的某个地方添加Request对象：

```
$reg = Registry::instance();
$reg->setRequest( new Request() );
```

然后在系统的另外一个地方获得它：

```
$reg = Registry::instance();
print_r( $reg->getRequest() );
```

可以看到，Registry是一个单例对象（本书在第9章中介绍过单例类）。代码通过instance()方法创建并返回Registry类的单个实例，然后用于设置和获得Request对象。尽管PHP无法强制限定返回类型，但setRequest()方法参数的类型提示保证了getRequest()返回的值是一个Request对象。

我们还可以改写注册表类，让它基于键来存储和获取数据：

```
class Registry {
    private static $instance;
    private $values = array();

    private function __construct() { }

    static function instance() {
        if ( ! isset( self::$instance ) ) { self::$instance = new self(); }
        return self::$instance;
    }

    function get( $key ) {
        if ( isset( $this->values[$key] ) ) {
            return $this->values[$key];
        }
        return null;
    }

    function set( $key, $value ) {
        $this->values[$key] = $value;
    }
}
```

这样做的好处是不需要为希望存储和访问的每个对象都创建类方法。坏处就是重新引入了全局变量。任意的字符串都可以作为存储对象的键，这意味着添加一个对象到系统时可以随意覆盖已有的键/值对。但这样的类似映射的结构在开发时很有用。如果我们清楚要存取什么数据，也可以转而使用显式命名的类方法。

你也可以在系统中使用注册表对象作为普通对象的工厂。注册表类不存储一个要提供的对象，而是先创建一个对象实例，然后存储对该对象的引用。它也可以做一些幕后的配置工作，例如从配置文件中得到数据或合并一组对象。

```
// 类Registry...
function treeBuilder() {
    if ( ! isset( $this->treeBuilder ) ) {
        $this->treeBuilder = new TreeBuilder( $this->conf()->get('treedir') );
    }
    return $this->treeBuilder;
}

function conf() {
    if ( ! isset( $this->conf ) ) {
        $this->conf = new Conf();
    }
    return $this->conf;
}
```

代码中的TreeBuilder和Conf都是虚构的类，它们的功能也是虚构的。需要TreeBuilder对象的客户类可以简单地调用Registry::treeBuilder()，而不会增加自己初始化的复杂性。复杂性主要来自于应用级别的数据，比如虚构的Conf对象，系统中绝大部分类都不应该与Conf类打交道。

注册表对象对于测试也很有用。静态方法instance()可提供模拟的Registry对象。下面看看如何修改instance()以达到这个目的：

```
static function testMode( $mode=true ) {
    self::$instance=null;
    self::$testmode=$mode;
}

static function instance() {
    if ( self::$testmode ) {
        return new MockRegistry();
    }
    if ( ! isset( self::$instance ) ) { self::$instance = new self(); }
    return self::$instance;
}
```

当系统测试到这一步时，你可以使用测试模式在模拟的注册表中切换。用这个办法可以提供stub对象（在测试中代替真实环境的模拟对象）或mock对象（和stub对象类似，但也可以分析对它们的调用并作出响应）。

```
Registry::testMode();
$mockreg = Registry::instance();
```

你可以参见第18章了解更多关于mock对象和stub对象的内容。

1. 注册表、作用域和PHP

作用域通常用于描述代码结构中对象或值的可见程度。变量的生命周期可以用时间来衡量。

变量作用域有3个级别。标准级别是指一个HTTP请求从开始到结束的周期。

PHP也内置了对会话变量的支持。在一次请求结束后，会话变量会被序列化并存储到文件系统或者数据库中，然后在下一个请求开始时取回。存放在cookie中的会话ID和通过查询字符串传递的会话ID被用于跟踪该会话的拥有者。因此，你可以认为某些变量拥有会话级别的作用域。利用这一点，可以在几次请求之间存放对象，保存用户访问的踪迹到数据库中。显然，要小心避免持有同一个对象的不同版本，因此当你把一个会话对象存到数据库时，需要考虑使用一定的锁定策略。

在其他语言中，特别是Java和Perl（运行在Apache模块ModPerl上），有一个“应用程序作用域”的概念。内存中的变量可以被程序中的所有对象实例访问。这和PHP很不同，但是在更大规模的应用中，为了访问配置变量，访问应用程序级别的数据是很有用的。你可以构建一个注册表类来模拟应用程序作用域，但需要注意一些地方。

图12-3演示了`Registry`类的一种结构，它以之前描述过的3个变量作用域级别为基础。

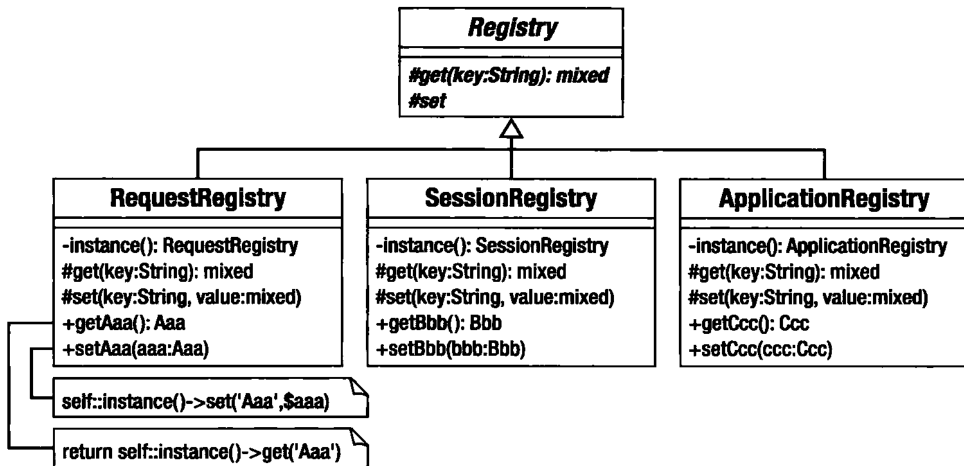


图12-3　为不同的作用域实现注册表类

基类`Registry`定义了两个`protected`方法：`get()`和`set()`。客户端代码并不能直接使用它们，因为我们想强制规定提供`get`和`set`操作的对象类型。基类也可以定义其他的`public`方法，比如`isEmpty()`、`isPopulated()`和`clear()`，但我把它当做练习留给你来完成。

注解　*在真实世界的系统中，可能你会想扩展这个结构，让它包括其他的继承层。你可以在各个子类中保留具体的`get()`和`set()`方法，但是在特定的领域类中包含定制为`public`的`getAaa()`和`setAaa()`方法。定制的对象会成为单例对象。通过这种方式，你可以重用核心的存储和获取操作，即在多个项目中重复使用同一个注册表类。*

下面是抽象类的代码：

```
namespace woo\base;

abstract class Registry {
    abstract protected function get( $key );
    abstract protected function set( $key, $val );
}
```

注解 注意，在这些例子中我使用了命名空间。因为我想在本章构建一个基础且完整的系统，使用包的层次结构并且利用命名空间为项目带来的简洁性和清晰性是有道理的。

请求级别的注册表类非常简单。和之前的例子不同，我们隐藏了Registry唯一的实例，并提供静态方法来设置和获取对象。除了这些，它的功能就是维护一个关联数组。

```
namespace woo\base;
// ...
class RequestRegistry extends Registry {
    private $values = array();
    private static $instance;

    private function __construct() {}
    static function instance() {
        if ( ! isset(self::$instance) ) { self::$instance = new self(); }
        return self::$instance;
    }

    protected function get( $key ) {
        if ( isset( $this->values[$key] ) ) {
            return $this->values[$key];
        }
        return null;
    }

    protected function set( $key, $val ) {
        $this->values[$key] = $val;
    }

    static function getRequest() {
        return self::instance()->get('request');
    }

    static function setRequest( \woo\controller\Request $request ) {
        return self::instance()->set('request', $request );
    }
}
```

会话级别的注册表只使用PHP内置的会话支持：

```
namespace woo\base;
// ...
class SessionRegistry extends Registry {
    private static $instance;
    private function __construct() {
```

12

```
        session_start();
    }

    static function instance() {
        if ( ! isset(self::$instance) ) { self::$instance = new self(); }
        return self::$instance;
    }

    protected function get( $key ) {
        if ( isset( $_SESSION[__CLASS__][$key] ) ) {
            return $_SESSION[__CLASS__][$key];
        }
        return null;
    }

    protected function set( $key, $val ) {
        $_SESSION[__CLASS__][$key] = $val;
    }

    function setComplex( Complex $complex ) {
        self::instance()->set('complex', $complex);
    }

    function getComplex( ) {
        return self::instance()->get('complex');
    }
}
```

可以看到，这个类使用了PHP中的预定义变量$_SESSION来设置和取回值。我们在构造器中使用session_start()方法开启会话。要一直使用会话，必须确保在使用这个类之前没有发送任何文字给用户。

可以想到，应用程序级别的注册表实现起来相当复杂。和本章中的所有代码示例一样，下面的代码只是用于解释概念，并不能直接应用于正式产品中。

```
namespace woo\base;
// ...
class ApplicationRegistry extends Registry {
    private static $instance;
    private $freezedir = "data";
    private $values = array();
    private $mtimes = array();

    private function __construct() { }

    static function instance() {
        if ( ! isset(self::$instance) ) { self::$instance = new self(); }
        return self::$instance;
    }

    protected function get( $key ) {
        $path = $this->freezedir . DIRECTORY_SEPARATOR . $key;
        if ( file_exists( $path ) ) {
```

```
            clearstatcache();
            $mtime=filemtime( $path );
            if ( ! isset($this->mtimes[$key] ) ) { $this->mtimes[$key]=0; }
            if ( $mtime > $this->mtimes[$key] ) {
                $data = file_get_contents( $path );
                $this->mtimes[$key]=$mtime;
                return ($this->values[$key]=unserialize( $data ));
            }
        }
        if ( isset( $this->values[$key] ) ) {
            return $this->values[$key];
        }
        return null;
    }
    protected function set( $key, $val ) {
        $this->values[$key] = $val;
        $path = $this->freezedir . DIRECTORY_SEPARATOR . $key;
        file_put_contents( $path, serialize( $val ) );
        $this->mtimes[$key]=time();
    }

    static function getDSN() {
        return self::instance()->get('dsn');
    }

    static function setDSN( $dsn ) {
        return self::instance()->set('dsn', $dsn);
    }
}
```

这个类使用序列化来存储和获取每个属性的值。get()方法检查某个值相应的文件是否存在。如果文件存在并且在上次读之后被修改过，该方法将数据反序列化并返回其内容。因为访问每个变量都要打开一次文件的做法效率不高，所以在这里你应使用另外一种办法——把所有的属性都保存到一个文件中。set()方法改变了$key引用的属性在类中和在文件中的值，它也更新了$mtimes属性的值。$mtimes是一个保存修改次数的数组，被用于检测保存文件是否被更新过。如果get()被调用，文件对应的$mtimes数组元素会被检查，以此判断文件在对象上次写入之后是否被修改了。

如果你在安装PHP时启用了shm扩展[①]，就可以使用该扩展中的函数来实现应用程序注册表。下面是一个简单的例子：

```
namespace woo\base;
// ...

class MemApplicationRegistry extends Registry {
    private static $instance;
```

① System V shared memory，*nix系统下的共享内存模块，注意在Windows平台下不可用。如果读者想实现一个高效又跨平台的应用程序注册表类，可以使用memcached等第三方工具，或者使用APC，将数据缓存在内存中供随时取用。——译者注

```
    private $values=array();
    private $id;
    const DSN=1;

    private function __construct() {
        $this->id = @shm_attach(55, 10000, 0600);
        if ( ! $this->id ) {
            throw new Exception("could not access shared memory");
        }
    }

    static function instance() {
        if ( ! isset(self::$instance) ) { self::$instance = new self(); }
        return self::$instance;
    }

    protected function get( $key ) {
        return shm_get_var( $this->id, $key );
    }

    protected function set( $key, $val ) {
        return shm_put_var( $this->id, $key, $val );
    }

    static function getDSN() {
        return self::instance()->get(self::DSN);
    }

    static function setDSN( $dsn ) {
        return self::instance()->set(self::DSN, $dsn);
    }

}
```

如果想要使用这段代码的变体，请务必先阅读下一节，因为这里有一些地方需要慎重考虑。

2. 效果

SessionRegistry和ApplicationRegistry都将数据序列化后保存到文件系统，因此有件事非常重要：从不同请求取回的对象是同一对象的不同副本，而不是对同一对象的引用。这对于SessionRegistry来说没什么关系，因为访问对象的是同一个用户。对于ApplicationRegistry来说，这却是严重的问题。如果混乱地保存数据，两个进程将会产生冲突。如下所示：

```
进程 1 访问一个对象
进程 2 访问一个对象
进程 1 改变一个对象
进程 2 改变一个对象
进程 1 保存一个对象
进程 2 保存一个对象
```

进程1所做的改变被进程2的保存覆盖了。如果你真的想为数据创建一个共享空间，需要为ApplicationRegistry实现一个锁定方案来防止这样的冲突，或者把ApplicationRegistry当做一个大型的只读资源，这是本章稍后将会在例子中采用的方案。我们先初始化数据，其后的交

互只能是只读的。代码只有在找不到存储文件的情况下才计算新的值并写入文件。因此，你可以通过删除存储文件来强制重新加载配置数据。你需要修改类代码来实现这样的只读功能。

要注意的另外一点是并不是所有的对象都适合序列化输出。特别是资源类型（resource类型）的数据（如数据库连接句柄），它们无法被序列化。你要自己想办法在序列化时处理句柄，并在反序列化时重新搜取句柄。

注解　*管理序列化的一个办法是实现魔术方法`__sleep()`和`__wakeup()`。当对象被序列化时，`__sleep()`被自动调用。你可以用`__sleep()`方法在保存对象前执行任何清理工作。它将会返回一个字符串数组，说明你想要保存的字段。当对象被反序列化时，`__wakeup()`方法会被调用。你可以使用`__wakeup()`重新获得存储对象时使用的文件或数据库句柄。*

虽然序列化是PHP中一个非常高效的功能，你仍然要小心所保存的内容。一个看上去很简单的对象中很可能包含了一个指向其他对象的引用，而该引用可能指向一个从数据库获得的巨大的对象集合。

`Registry`对象使数据全局有效。这意味着注册表的任何客户端代码都可以在自己的类中加上对于注册表类的依赖。但如果你过分依赖`Registry`对象来存放大量系统数据，将会导致严重问题。`Registry`对象最好不要存储过多数据，里面的数据集合也要经过良好定义。

12.3　表现层

当一个请求到达系统时，系统必须能够理解请求中的需求是什么，然后调用适当的业务逻辑进行处理，最后返回响应结果。对于简单的程序，整个过程可能完全放在视图之中，只有重量级的逻辑和持久化操作相关的代码才放在封装好的类库中。

注解　*一个视图是指视图层中一个单独的元素。它可能是一个PHP页面（或视图元素集合），负责显示数据和让用户生成新请求。在像Smarty这样的模板系统中，一个视图即指一个模板。*

随着系统的增长，这种默认方案不能满足处理请求、调用业务逻辑和派发视图的要求。

本节我们将研究表现层管理以上3个主要功能的策略。视图层与命令和控制层的边界通常很模糊，因此我们常把这两个层统称为“表现层”。

12.3.1　前端控制器

本模式和传统PHP应用程序的“多入口”方式相反。前端控制器模式用一个中心来处理所有到来的请求，最后调用视图来将结果呈现给用户。前端控制器模式是Java企业应用的核心模式之一。本模式在《J2EE核心模式》中有详细的讲解，它同时也是最有影响力的企业模式之一。在PHP中，这个模式并没有受到广泛喜爱，部分原因是初始化前端控制器所需要的开销会导致系统性能下降。

现在我写的大部分系统都开始向前端控制器模式转移。虽然我有时没有使用完整的前端控制

器模式，但是我发现在项目中使用前端控制器模式确实可以提供我需要的灵活性。

1. 问题

当请求可以发送到系统中多个地方时，我们很难避免代码重复。你可能需要验证用户、把术语翻译成多种语言或者只访问公用数据。当多个页面都要执行同一个操作时，我们可以从一个页面复制该操作相关的代码并粘贴到另一个页面。但是这样的话，当需要修改系统中某个部分时，其他部分也要随着改变，给代码维护带来困难。因此我们要尽量避免这种情况。当然，首先要做的是把公共操作集中到类库代码中。但即使这样，对库函数和方法的调用代码仍然会分布到系统中各个部分。

当系统控制器和视图混杂在一起时，管理视图的切换和选择是另一个难点。在一个复杂的系统中，随着输入和逻辑层中操作的成功执行，一个视图中的提交动作可能会产生任意数目的结果页面。从一个视图跳到另一个视图时，可能会产生混乱，特别当某个视图被用在多个地方的时候。

2. 实现

在核心部分，前端控制器模式定义了一个中心入口，每个请求都要从这个入口进入系统。前端控制器处理请求并选择要执行的操作。操作通常都定义在特定的Command对象中。Command对象是根据命令模式组织的。

图12-4展示了一个前端控制器的结构。

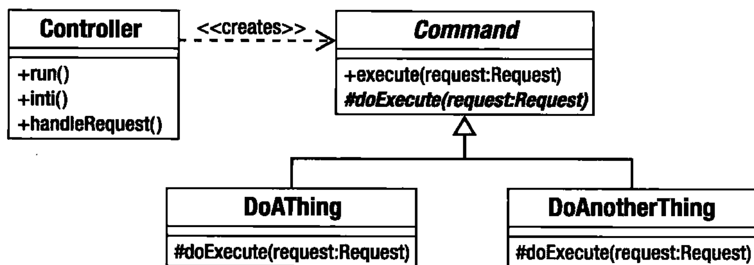


图12-4　控制器类和命令层次结构

实际开发时，你可能会部署一些助手类来协助控制器的处理过程，但现在我们还是先从控制器的核心部分开始研究。下面是一个简单的Controller类：

```
namespace woo\controller;

//...
class Controller {
    private $applicationHelper;
    private function __construct() {}


    static function run() {
        $instance = new Controller();
        $instance->init();
        $instance->handleRequest();
    }
```

```
    function init() {
        $applicationHelper
            = ApplicationHelper::instance();
        $applicationHelper->init();
    }

    function handleRequest() {
        $request = new \woo\controller\Request();
        $cmd_r = new \woo\command\CommandResolver();
        $cmd = $cmd_r->getCommand( $request );
        $cmd->execute( $request );
    }
}
```

这个Controller类非常简单，而且没有考虑错误处理。系统中的控制器负责分配任务给其他类。其他类完成了绝大部分实际工作。

run()只是一个便捷方法，用于调用init()和handleRequest()。run()是静态方法，而且本类的构造方法被声明为private，因此客户端代码只能通过run()方法来实例化控制器类，并执行相关操作。我们可以使用只包含两行代码的index.php文件来完成这个工作：

```
require( "woo/controller/Controller.php" );
\woo\controller\Controller::run();
```

init()和handleRequest()方法的不同体现了PHP的特性。在某些编程语言中，init()只在应用第一次启动时运行，而handleRequest()在用户的每个请求到来时运行。尽管init()在每次请求中都会被调用，但是这个类还是注意到了启动和请求处理间的区别[①]。

init()方法中获得ApplicationHelper（应用程序助手）类的一个对象实例。这个类的作用是管理应用程序的配置信息。控制器的init()方法调用ApplicationHelper中同名的init()方法，用于初始化应用程序要使用的数据。

handleRequest()方法通过CommandResolver来获取一个Command对象，然后调用Command对象的execute()方法来执行实际操作。

● *应用程序助手*

ApplicationHelper类并不是前端控制器的核心，但前端控制器通常都需要通过应用助手类来获取基本的配置数据，因此我们需要讨论一下获取配置数据的策略。下面是一个简单的ApplicationHelper：

```
namespace woo\controller;
//...
class ApplicationHelper {
    private static $instance;
    private $config = "/tmp/data/woo_options.xml";

    private function __construct() {}
```

① 因为PHP是解释型语言，每次请求中产生的数据在该请求结束时都将被销毁，无法供给下一次请求使用，所以下一次请求还要重新调用init()方法。——译者注

```
    static function instance() {
        if ( ! self::$instance ) {
            self::$instance = new self();
        }
        return self::$instance;
    }



    function init() {
        $dsn = \woo\base\ApplicationRegistry::getDSN( );
        if ( ! is_null( $dsn ) ) {
            return;
        }
        $this->getOptions();
     }

     private function getOptions() {
        $this->ensure( file_exists( $this->config  ),
                            "Could not find options file" );

        $options = SimpleXml_load_file( $this->config );
        print get_class( $options );
        $dsn = (string)$options->dsn;
        $this->ensure( $dsn, "No DSN found" );
        \woo\base\ApplicationRegistry::setDSN( $dsn );
        // 设置其他值
    }

    private function ensure( $expr, $message ) {
        if ( ! $expr ) {
            throw new \woo\base\AppException( $message );
        }
    }
}
```

这个类的作用是读取配置文件中的数据并使客户端代码可以访问这些数据。可以看到，这个类实现了单例模式。使用单例模式使它能够为系统中所有的类服务。另外，你也可以把这个类的代码当成一个标准类并确保它被传递给其他感兴趣的对象。本书在第9章及本章的前面部分已经讨论了使用单例模式需要注意的问题。

现在我们已经实现了ApplicationRegistry（应用注册表），我们还应重构代码，把ApplicationHelper改写为注册表，而不是两个任务重叠的单例对象。重构代码的建议前一节中已经提过（将ApplicationRegistry的核心功能从领域对象的存取中分离出来），留给读者当做练习。

因此init()方法只负责加载配置数据。实际上，它检查ApplicationRegistry，看数据是否已经被缓存。如果Registry对象中的值已经存在，init()就什么都不做。如果系统初始化要做大量工作，这样的缓存机制是很有用的。在将应用程序初始化和独立请求相分离的编程语言中，可以使用复杂的初始化操作。但在PHP中，你不得不尽量使用缓存来减少初始化操作。

缓存可以有效地保证复杂而且耗费时间的初始化过程只在第一次请求时发生，而之后所有的请求都能从中受益。

如果是第一次运行（或者缓存文件已被删除——这是一种简单而有效的强制重新读取配置信息的方法），getOptions()方法将被调用。

在现实世界中，我们需要做比示例代码更多的工作。示例中所做的工作只是获取一个DSN。首先，getOptions()方法检查配置文件是否存在（路径存放在$config属性中），然后从配置文件中加载XML数据并设置DSN。

注解 在这些例子中，ApplicationRegistry和ApplicationHelper都使用了硬编码的文件路径。在实际项目中，这些文件路径应该是可配置的而且可以从一个注册表对象或一个配置对象中获取。实际的路径可以在安装时用构建工具（如PEAR或Phing，参见第15章和第19章）来设置。

注意类中使用了一个技巧来抛出异常，避免了在代码中到处使用下面这样的条件语句和throw语句：

```
if ( ! file_exists( $this->config ) ) {
    throw new \woo\base\AppException(
        "Could not find options file" );
}
```

这个技巧就是ApplicationHelper类在ensure()方法中集合了检测表达式和throw语句。只用一行代码就能确定条件是否为真（如果不为真，则抛出异常）：

```
$this->ensure( file_exists( $this->config  ),
                              "Could not find options file" );
```

缓存对系统开发者和使用者都有好处。系统可以方便地维护一个易于使用的XML配置文件，同时使用缓存意味着系统能以很快的速度访问配置文件中的数据。当然，如果类的用户还是程序员，或者并不经常修改配置，你可以直接在助手类中（或者是用一个单独的文件）包含PHP数据结构，不用把配置数据单独放到XML文件中。虽然这种写法有风险，但是代码执行效率最高。

- 命令解析器

控制器需要通过某种策略来决定如何解释一个HTTP请求，然后调用正确的代码来满足这个请求。你可以很容易地在Controller类中包含这个策略，但是我更喜欢使用一个特定的类来完成这个任务。因为这样的代码易于重构和实现多态。

前端控制器通常通过运行一个Command对象（本书在第11章中介绍过命令模式）来调用应用程序逻辑。Command对象通常根据请求中的参数或URL的结构（例如，可以使用Apache配置来确定URL中的哪个字段用于选择命令）来决定选择哪个命令。在下面的例子中，我们将使用一个简单的参数cmd。

有多种方法可以用来根据给定的参数选择命令。你可以在一个配置文件或一个数据结构（逻辑方案）中测试该参数，或者直接查找文件系统（物理方案）中是否存在与参数相对应的类文件。

逻辑方案更灵活一些，但是工作量也更大些（包括设置和维护）。你可以在12.3.2节找到使用该方案的例子。

上一章介绍过一个使用物理方案的命令工厂的例子。下面对该例子做微小改动，使用反射（reflection）来增强安全性：

```
namespace woo\command;
//...

class CommandResolver {
    private static $base_cmd;
    private static $default_cmd;

    function __construct() {
        if ( ! self::$base_cmd ) {
            self::$base_cmd = new \ReflectionClass( "\woo\command\Command" );
            self::$default_cmd = new DefaultCommand();
        }
    }

    function getCommand( \woo\controller\Request $request ) {
        $cmd = $request->getProperty( 'cmd' );
        $sep = DIRECTORY_SEPARATOR;
        if ( ! $cmd ) {
            return self::$default_cmd;
        }
        $cmd=str_replace( array('.', $sep), "", $cmd );
        $filepath = "woo{$sep}command{$sep}{$cmd}.php";
        $classname = "woo\\command\\{$cmd}";
        if ( file_exists( $filepath ) ) {
            @require_once( "$filepath" );
            if ( class_exists( $classname) ) {
                $cmd_class = new ReflectionClass($classname);
                if ( $cmd_class->isSubClassOf( self::$base_cmd ) ) {
                    return $cmd_class->newInstance();
                } else {
                    $request->addFeedback( "command '$cmd' is not a Command" );
                }
            }
        }

        $request->addFeedback( "command '$cmd' not found" );
        return clone self::$default_cmd;
    }
}
```

这个简单的类用于查找请求中包含的cmd参数。假设参数被找到，并与命令目录中的类文件相匹配，而该文件也正好包含了cmd类，则该方法创建并返回相应类的实例。

如果其中任意条件未满足，getCommand()方法使用默认的Command对象。

你或许想知道，为什么实例化Command类时不需要提供参数：

```
if ( $cmd_class->isSubClassOf( self::$base_cmd ) ) {
```

```
        return $cmd_class->newInstance();
    }
```

这是因为Command类本身的代码：

```
Namespace woo\command;
//...

abstract class Command {

    final function __construct() { }

    function execute( \woo\controller\Request $request ) {
        $this->doExecute( $request );
    }

    abstract function doExecute( \woo\controller\Request $request );
}
```

通过声明构造方法为final，任何子类都不能覆盖这个构造方法。因此，所有Command类的构造方法都不需要参数。

注意绝对不可使用未经检查的用户输入数据。示例代码中包含了一个测试来确保提供的"cmd"字符串中没有路径元素，因此只有正确目录下的文件才可以被调用（而不是../../../tmp/DodgyCommand.php这样的错误路径）。你还可以只接受与配置文件中的值匹配的命令字符串，这样可使代码更加安全。

创建命令类的时候，要尽可能让其中不包含应用逻辑。一旦它们包含了具体的业务处理逻辑，你就会发现它们变成了混乱的事务脚本，代码重复也随之而来。命令是一种中转站：它们解释请求，调用领域逻辑来改变对象，然后把对象传递到表现层中。一旦命令类中开始负责做比这些更复杂的工作，就需要对代码进行重构。还好重构并不是太难。因为很容易发现命令类中是否做了太多的事情，而解决办法也很清楚——把具体的功能转移到外观层或者领域类中。

- **请求**

在PHP中，对系统的请求会被自动封装到一个全局有效的数组[①]中。但在上面的例子中，我们仍然使用类来封装请求。Request对象被传递给CommandResolver，再被传递给具体的Command。

为什么我们不让这些类直接查询$_REQUEST、$_POST或$_GET数组呢？我们当然可以这样做，但是把请求处理集中到一个地方，我们就有了更多选择。比如，你可以对所有请求使用过滤器。或者像下一个例子那样，可以从非HTTP请求中收集请求参数，允许应用程序在命令行或者测试脚本中运行。当然，在命令行的环境下，如果程序要使用会话，你需要自己设计一种会话存储机制。你可以使用注册表模式，根据程序的环境生成不同的Registry类。

Request对象也可以用于存储需要和视图层交换的数据。从这个角度理解，Request对象也可以提供响应请求的功能。

① 常称之为超全局变量，superglobal。如$_REQUEST、$_POST或$_GET等。——译者注

下面是一个简单的Request类：

```
namespace woo\controller;
//...

class Request {
    private $properties;
    private $feedback = array();

    function __construct() {
        $this->init();
        \woo\base\RequestRegistry::setRequest($this );
    }

    function init() {
        if ( isset( $_SERVER['REQUEST_METHOD'] ) ) {
            $this->properties = $_REQUEST;
            return;
        }

        foreach( $_SERVER['argv'] as $arg ) {
            if ( strpos( $arg, '=' ) ) {
                list( $key, $val )=explode( "=", $arg );
                $this->setProperty( $key, $val );
            }
        }
    }

    function getProperty( $key ) {
        if ( isset( $this->properties[$key] ) ) {
            return $this->properties[$key];
        }
    }

    function setProperty( $key, $val ) {
        $this->properties[$key] = $val;
    }

    function addFeedback( $msg ) {
        array_push( $this->feedback, $msg );
    }

    function getFeedback( ) {
        return $this->feedback;
    }

    function getFeedbackString( $separator="\n" ) {
        return implode( $separator, $this->feedback );
    }
}
```

如上所示，这个类的主要功能是设置和获取属性。init()方法负责填充私有的$properties数组。注意init()方法同时支持命令行参数和HTTP请求，这在测试和调试程序时是非常有用的。

有了一个Request对象后，就可以通过getProperty()方法得到HTTP请求中的参数。getProperty()方法接受一个字符串的键名为参数并返回相应的值（存放在$properties数组中）。你也可以通过setProperty()添加数据。

这个类也维护着一个$feedback数组。控制器类可以通过$feedback数组方便地传递消息给用户。

- 命令

我们已经介绍过Command基类，而且本书第11章也详细介绍过命令模式，因此在此我们不再深入讨论命令。让我们直接看一个具体的Command对象：

```
namespace woo\command;
//...

class DefaultCommand extends Command {
    function doExecute( \woo\controller\Request $request ) {
        $request->addFeedback( "Welcome to WOO" );
        include( "woo/view/main.php");
    }
}
```

如果请求没有指明调用哪一个特定的Command对象，本对象就会由CommandResolver提供。

你可能已经注意到，抽象基类已经实现了execute()方法，该方法会向下调用由子类实现的doExecute()方法。因此你可以修改基类代码，添加初始化和清理操作等功能，这些功能对于所有命令对象都有效。

execute()方法的参数是一个Request对象，因此execute()方法可以访问用户输入的数据，同时也可以调用Request对象中的setFeedback()方法。DefaultCommand通过setFeedback()方法设置了一个欢迎消息。

最后，命令对象通过调用include()方法来分配视图。将命令和视图的对应关系写到Command类中是个最简单的分配方案，它已经足够应付小型的系统。在12.3.2节中，你可以看到更多灵活的方案。

main.php文件包含了一些HTML代码和对Request对象的调用代码，用于获得反馈信息（本章后面将会详细介绍视图）。现在我们的系统已经比较完整，可以运行了。执行代码后，我们可以看到：

```
<html>
<head>
<title>Woo! it's Woo!</title>
</head>
<body>

<table>
<tr>
<td>
Welcome to WOO</td>
</tr>
</table>
```

```
</body>
</html>
```

如上所示，在默认命令中设置的反馈消息被输出。下面让我们回顾一下得到这个输出结果的整个过程。

● 概述

本节中这些类的细节可能会掩盖了前端控制器模式的简洁性。图12-5用一个序列图（sequence diagram）来展示一个请求的生命周期。

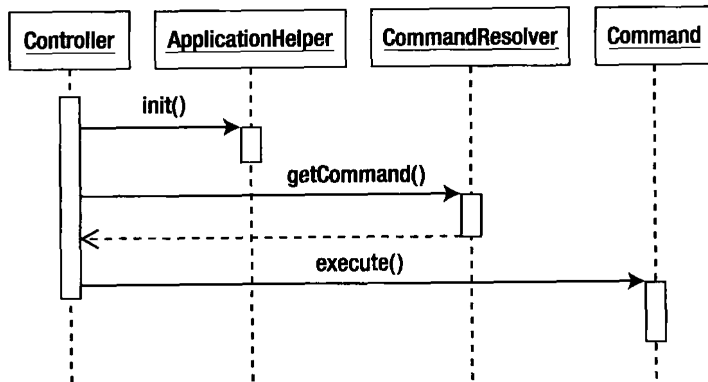


图12-5 前端控制器的执行流程

可以看到，前端控制器将初始化工作委托给ApplicationHelper对象（使用了缓存来缩短装载时间），然后Controller从CommandResolver对象获取一个Command对象。最后，它调用Command::execute()进入业务逻辑。

在以上示例采取的实现方案中，Command自己负责分配视图层。在下一节中，将会看到如何进一步改进这个方案。

3. 效果

是否使用前端控制器需要慎重考虑。在你看到它的好处之前，需要做大量的前期开发工作。如果你的项目需要很快完成，或者项目太小以致于前端控制器在系统中的比重太大，那么使用前端控制器就不太适合。

之前已经说过，一旦在项目中成功部署了前端控制器，就会发现可以很快地在其他项目中再次使用它。你可以把前端控制器的功能提取到共享库代码中，打造出一个可重用的开发框架。

前端控制器的另一个缺点是每次请求都要加载所有的配置信息。当然，我们在开发项目的时候，即使没有使用前端控制器也常常这样。但是前端控制器通常还需要额外的信息，比如命令和视图的分布结构等。

这些开销很容易通过缓存来降低。最有效的方法是把配置信息直接写到PHP代码中去。如果你是系统唯一的维护者，这样做是可以的，但是如果还有其他不懂技术的用户，最好把配置信息单独放到一个配置文件中。但你仍然可以读取配置文件，把内容转换成PHP数据并写入缓存文件

这样也可以提供PHP代码缓存。原生PHP缓存创建后，系统将会一直使用缓存中的数据，直到配置文件发生变化导致缓存需要重建。你可以把数据序列化后放入缓存，就像在ApplicationRegistry类中那样，这个办法效率不是很高但是非常方便。

使用前端控制器的好处是集中了系统的表现逻辑，这意味着你可以在一个地方（在某个类集合中）同时处理请求和选择视图。这样做能降低代码重复和bug发生的几率。

前端控制器也是易于扩展的。搭建好前端控制器的核心部分后，你可以很方便地增加新的Command类和视图。

在这个例子中，命令对象自己处理了视图分配的工作。如果使用一个对象来帮助前端控制器选择视图（或命令），那么可以更好地控制项目的导航（navigation，指跳转到系统的某个部分或某个页面的功能）部分。如果没有这样的对象，系统中的导航部分很难维护。下一节将介绍这样的对象。

12.3.2 应用控制器

如果系统的规模较小，我们可以让命令对象自己调用视图，但这并不是最佳的选择。最好是尽可能地将命令和视图分离开来。

应用控制器负责映射请求到命令，并映射命令到视图。这种分离意味着你可以更加容易地改变视图（即模板）而不用改动核心代码。同时，你也可以改变应用程序的流程而不需要修改核心部分的代码。通过将Command分离出来，我们可以更加容易地把同一个Command用在系统中的不同地方。

1. 问题

回到我们的例子，管理员需要能够增加一个场所（venue）到系统中，并将场所关联到一个空间（space）。因此系统需要支持AddVenue和AddSpace命令。我们可以直接使用请求参数（如cmd=AddVenue）将命令映射到一个类（AddVenue）。

一般来说，如果AddVenue命令成功执行，AddSpace命令将被调用。这种关系可以硬编码到类本身，即通过AddVenue调用AddSpace。然后，AddSpace加载一个视图，该视图中包含用于增加空间到场所的HTML表单。

以上两个命令至少和两个不同的视图关联——一个包含输入表单的核心视图和一个显示错误或欢迎信息的视图。根据上面讨论过的逻辑，我们可以直接在Command类中包含这些视图（使用条件语句来判断何时展示哪个视图）。

当系统的规模较小的时候，这样将视图硬编码到命令类中是可行的。但随着系统的发展，命令类不断增多，这种处理方式就有问题了。比如我们要让AddVenue使用特定的视图，并且想要修改一个命令调用另一个命令的逻辑（例如成功增加场所后想调用一个额外的视图）。如果每一个命令只使用一次，一个命令只与其他命令和一个视图有联系，那么你应该将命令间的关系及命令与视图间的关系硬编码到每个命令类中。否则，请继续阅读本章后面的内容。

应用控制器类可以替你接管这个工作，把Command类解放出来，让Command类集中精力完成自己的工作，包括处理输入、调用应用程序逻辑和处理结果等。

2. 实现

和其他模式一样，应用控制器模式的关键是接口。应用控制器是一个类（或一组类），前端控制器可以基于用户请求用它们来获取命令，并在命令执行后找到合适的视图来展示给用户。你可以在图12-6中看到这种关系的核心部分。

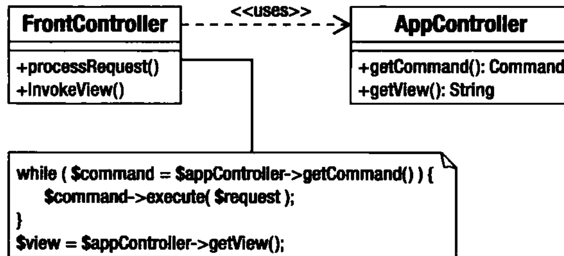


图12-6　应用控制器模式

本章中所有模式的目标都是尽可能地简化客户端代码可操作的应用入口，即前端控制器，虽然我们必须在接口背后部署一个实现。此处的做法并不是唯一方案。请记住本模式的关键在于模式中的参与者（包括应用控制器、命令和视图等）互相通信的方式，而不是实现的细节。

我们从使用应用控制器的代码（即客户端代码）开始研究。

- **前端控制器**

让我们看看FrontController如何使用AppController类（简化和忽略了错误处理）：

```
function handleRequest() {
    $request = new Request();
    $app_c = \woo\base\ApplicationRegistry::appController();

    while( $cmd = $app_c->getCommand( $request ) ) {
        $cmd->execute( $request );
    }
    $this->invokeView( $app_c->getView( $request ) );
}

function invokeView( $target ) {
    include( "woo/view/$target.php" );
    exit;
}
```

上面代码和本章前面介绍过的前端控制器的例子的不同之处在于Command对象是在一个循环中被获取并执行的。另外，本例也使用AppController来获得它要包含的视图名称。注意上面的代码使用了注册表对象来获取AppController对象。

那么我们该如何根据一个cmd参数来调用一系列的命令和最终的视图呢？

- **实现概述**

在不同的操作阶段，Command类可能需要加载不同的视图。AddVenue命令的默认视图可以是一个数据输入表单。如果用户添加错误类型的数据，页面可能会重新显示表单，或者显示一个错

误信息页面。如果一切顺利，场所将被添加到系统中，然后我们将会进入命令链的下一环节，如AddSpace命令。

Command对象通过设置状态标志（status flag）告诉系统它们当前的状态。下面定义的状态标志（在Command超类中被定义为属性）可以被所有的命令类识别：

```
private static $STATUS_STRINGS = array (
    'CMD_DEFAULT'=>0,
    'CMD_OK' => 1,
    'CMD_ERROR' => 2,
    'CMD_INSUFFICIENT_DATA' => 3
);
```

应用控制器通过Request对象找到并实例化正确的Command类。一旦Command对象开始运行，就会被设置为某种状态。Command和状态被用来与一个数据结构做比较，以决定接下来执行哪个命令，如果不需要再执行任何命令，则决定加载哪个视图。

● 配置文件

系统管理员可以通过设置配置选项来决定命令和视图工作的方式。下面是一个简化版本：

```
<control>
    <view>main</view>
    <view status="CMD_OK">main</view>
    <view status="CMD_ERROR">error</view>

    <command name="ListVenues">
        <view>listvenues</view>
    </command>

    <command name="QuickAddVenue">
        <classroot name="AddVenue" />
        <view>quickadd</view>
    </command>

    <command name="AddVenue">
        <view>addvenue</view>
        <status value="CMD_OK">
          <forward>AddSpace</forward>
        </status>
    </command>

    <command name="AddSpace">
        <view>addspace</view>
        <status value="CMD_OK">
            <forward>ListVenues</forward>
        </status>
    </command>
    ...
</control>
```

这段简化过的XML片段展示了从Command类中抽取命令流及它们与视图间关系的一种办法。配置选项都包含在一个control元素中。这个办法的原理是基于查找。最外面定义的是通用元素，

它们可以被command元素内部等效的元素覆盖。

第一个元素view定义了可用于所有命令的默认视图。如果没有指定调用特定视图，则此处定义的默认视图会被调用。同级的另一个view元素声明了status属性，该属性与Command类中的状态标志相对应。每种状态都用Command对象设置的一个标志来描述，这样可以表明任务当前执行的阶段。因为这些view元素比第一个view元素指定了更为具体的内容，所以其优先级更高。如果一个命令设置标志为CMD_OK，则相应的menu视图会被加载，除非还有一个更为具体的元素覆盖它。

设置了这些默认值之后，文档接下来显示command元素。默认情况下，这些元素直接映射到Command类（以及它们在文件系统中的类文件），正如之前的CommandResolver那样。因此，如果cmd参数被设置为AddVenue，那么配置文档中的对应元素就会被选中。字符串"AddVenue"被用于组合成一个指向AddVenue.php类文件的文件路径。

配置文件还支持别名。因此，如果cmd被设置为QuickAddVenue，那么下面的元素被使用：

```
<command name="QuickAddVenue">
    <classroot name="AddVenue" />
    <view>quickadd</view>
</command>
```

在这里，被命名为QuickAddVenue的command元素并没有直接映射到一个类文件。真正的映射由classroot元素定义。这样我们可以在不同的流和视图中引用AddVenue类。

command元素从外到内生效。内部的元素优先级较高。通过在一个command元素内设置一个view元素，我们把该view绑定到命令。

```
<command name="AddVenue">
    <view>addvenue</view>
    <status value="CMD_OK">
      <forward>AddSpace</forward>
    </status>
</command>
```

因此在这里，addvenue这个视图与AddVenue命令（在Request对象的cmd参数中设置）相关联。这意味着当AddVenue命令被调用时，addvenue.php视图总是会被加载（除非status条件被匹配，这时会调用其他命令或者显示其他视图）。如果AddVenue类设置状态标志为CMD_OK，那么本视图（默认视图）将被取代。

status元素可以只包含另一个视图来取代默认视图。但在上面的代码中，forward（转向）元素指定的不是视图，而是命令（AddSpace）。通过转向另一个命令，配置文件把处理视图的任务交给该命令来执行。

● *解析配置文件*

以上模型相当灵活，易于控制内容显示和命令的处理流程。但你肯定不想针对每次请求都解析配置文件。对此我们有一个解决方案。ApplicationHelper提供了将配置数据缓存起来的功能。

下面是ApplicationHelper的部分代码：

```
private function getOptions() {
    $this->ensure( file_exists( $this->config  ),
                        "Could not find options file" );
    $options = @SimpleXml_load_file( $this->config );

    // ……设置DSN……

    $map = new ControllerMap();

    foreach ( $options->control->view as $default_view ) {
        $stat_str = trim($default_view['status']);
        $status = \woo\command\Command::statuses( $stat_str );
        $map->addView( 'default', $status, (string)$default_view );
    }
    // ……省略了更多解析代码……
    \woo\base\ApplicationRegistry::setControllerMap( $map );
}
```

XML解析是一个很烦琐但没什么挑战性的工作（即使使用PHP 5中极为好用的SimpleXML包），因此在此没有提供更多细节。只有在配置数据未被缓存到ApplicationRegistry对象中时，getOptions()方法才会被调用。

● *存储配置数据*

上一节中我们需要的缓存对象是ControllerMap，它其实是将3个数组封装而成的。当然我们也可以直接使用数组，但使用ControllerMap更保险，可以确保每个数组都遵循一定的格式。下面是ControllerMap类的代码：

```
namespace woo\controller;
//...

class ControllerMap {
    private $viewMap = array();
    private $forwardMap = array();
    private $classrootMap = array();

    function addClassroot( $command, $classroot ) {
        $this->classrootMap[$command]=$classroot;
    }

    function getClassroot( $command ) {
        if ( isset( $this->classrootMap[$command] ) ) {
            return $this->classrootMap[$command];
        }
        return $command;
    }

    function addView( $command='default', $status=0, $view ) {
        $this->viewMap[$command][$status]=$view;
    }

    function getView( $command, $status ) {
```

```
        if ( isset( $this->viewMap[$command][$status] ) ) {
            return $this->viewMap[$command][$status];
        }
        return null;
    }

    function addForward( $command, $status=0, $newCommand ) {
        $this->forwardMap[$command][$status]=$newCommand;
    }

    function getForward( $command, $status ) {
        if ( isset( $this->forwardMap[$command][$status] ) ) {
            return $this->forwardMap[$command][$status];
        }
        return null;
    }
}
```

$classroot属性是一个将命令句柄（配置文件中命令元素的名称）映射到Command类名称的关联数组。类名称中去除了前缀，如woo_command_AddVenue类在此的名称为AddVenue。这样可以检测cmd参数是一个别名还是一个特定的类文件。在解析配置文件时，addClassroot()方法会被调用来为这个数组赋值。

$forwardMap和$viewMap数组都是二维数组，用于将命令和状态组合起来。

回顾一下以下片段：

```
<command name="AddVenue">
    <view>addvenue</view>
    <status value="CMD_OK">
      <forward>AddSpace</forward>
    </status>
</command>
```

解析时会添加正确的元素到$viewMap属性中：

```
$map->addView( 'AddVenue', 0, 'addvenue' );
```

同时也会为$forwardMap属性赋值：

```
$map->addForward( 'AddVenue', 1, 'AddSpace' );
```

应用控制器类会依次调用这些操作。假如AddVenue返回CMD_OK（其值为1，如果返回CMD_DEFAULT则为0）。应用控制器会在$forwardMap数组中查找相应的元素（先找特定的Command和状态标志，后找一般的Command和状态标志）。找到的第一个匹配的元素会被返回：

```
$viewMap['AddVenue'][1];  // AddVenue CMD_OK [MATCHED]
$viewMap['AddVenue'][0];  // AddVenue CMD_DEFAULT
$viewMap['default'][1];   // DefaultCommand CMD_OK
$viewMap['default'][0];   // DefaultCommand CMD_DEFAULT
```

同样，我们也查找数组中匹配的元素来得到一个视图。

下面是应用控制器的代码：

```
namespace woo\controller;
//..

class AppController {
    private static $base_cmd;
    private static $default_cmd;
    private $controllerMap;
    private $invoked = array();

    function __construct( ControllerMap $map ) {
        $this->controllerMap = $map;
        if ( ! self::$base_cmd ) {
            self::$base_cmd = new \ReflectionClass( "\\woo\\command\\Command" );
            self::$default_cmd = new \woo\command\DefaultCommand();
        }
    }

    function getView( Request $req ) {
        $view = $this->getResource( $req, "View" );
        return $view;
    }

    function getForward( Request $req ) {
        $forward = $this->getResource( $req, "Forward" );
        if ( $forward ) {
            $req->setProperty( 'cmd', $forward );
        }
        return $forward;
    }

    private function getResource( Request $req,
                                  $res ) {
        // 得到前一个命令及其执行状态
        $cmd_str = $req->getProperty( 'cmd' );
        $previous = $req->getLastCommand();
        $status = $previous->getStatus();
        if (! $status ) { $status = 0; }
        $acquire = "get$res";
        // 得到前一个命令的资源及其状态
        $resource = $this->controllerMap->$acquire( $cmd_str, $status );
        // 查找命令并且状态为0的资源
        if ( ! $resource ) {
            $resource = $this->controllerMap->$acquire( $cmd_str, 0 );
        }

        // 或者'default'命令和命令状态
        if ( ! $resource ) {
            $resource = $this->controllerMap->$acquire( 'default', $status );
        }

        // 其他情况获取'default'失败，状态为0
        if ( ! $resource ) {
            $resource = $this->controllerMap->$acquire( 'default', 0 );
        }
```

12

```
        return $resource;
    }

    function getCommand( Request $req ) {
        $previous = $req->getLastCommand();
        if ( ! $previous ) {
            // 这是本次请求调用的第一个命令
            $cmd = $req->getProperty('cmd');
            if ( ! $cmd ) {
                // 如果无法得到命令，则使用默认命令
                $req->setProperty('cmd', 'default' );
                return  self::$default_cmd;
            }
        } else {
            // 之前已经执行过一个命令
            $cmd = $this->getForward( $req );
            if ( ! $cmd ) { return null; }
        }

        //  在$cmd变量中保存着命令名称，并将其解析为Command对象
        $cmd_obj = $this->resolveCommand( $cmd );
        if ( ! $cmd_obj ) {
            throw new \woo\base\AppException( "couldn't resolve '$cmd'" );
        }

        $cmd_class = get_class( $cmd_obj );
        if ( isset( $this->invoked[$cmd_class] ) ) {
            throw new \woo\base\AppException( "circular forwarding" );
        }

        $this->invoked[$cmd_class]=1;
        // 返回Command对象
        return $cmd_obj;
    }

    function resolveCommand( $cmd ) {
        $classroot = $this->controllerMap->getClassroot( $cmd );
        $filepath = "woo/command/$classroot.php";
        $classname = "\\woo\\command\\{$classroot}";
        if ( file_exists( $filepath ) ) {
            require_once( "$filepath" );
            if ( class_exists( $classname) ) {
                $cmd_class = new ReflectionClass($classname);
                if ( $cmd_class->isSubClassOf( self::$base_cmd ) ) {
                    return $cmd_class->newInstance();
                }
            }
        }
        return null;
    }
}
```

getResource()方法执行查找操作，用于转向或选择视图（分别由getForward()和getView()

负责）。注意它会优先查找最具体的命令字符串和状态标志的组合，然后才搜索通用的组合。

getCommand()方法负责返回转向中需要使用的所有命令。工作过程是这样的：当请求第一次被接受到，就会生成一个cmd属性。本次请求之前没有任何Command执行过。Request对象储存着与这个过程相关的信息。如果cmd属性未被赋值，则类方法使用默认值，并返回默认的Command类。$cmd字符串变量被传递给resolveCommand()，该方法用于得到一个Command对象。

当在请求中第二次调用getCommand()方法时，Request对象将持有一个对之前执行过的Command对象的引用。这时getCommand()将根据该命令和状态标志判断是否需要转向（转向通过getForward()实现）。如果getForward()方法找到一个匹配的对象，会返回一个字符串。该字符串可解析成一个命令对象并供控制器使用。

getCommand()中需要注意的另一点是我们通过判断来避免循环转向。数组的索引为Command类的名称。在添加元素时，如果该元素已经存在，说明该命令之前已经被获取过。这会让我们陷入死循环，这可不是我们想要的，因此如果发生这种情况，就抛出一个异常。

应用控制器可能使用的用于获取视图和命令的策略可能差别很大，因为这些过程对于外部系统来说都是隐藏在幕后进行的。图12-7展示了前端控制器如何通过一个应用控制器来得到Command对象和视图。

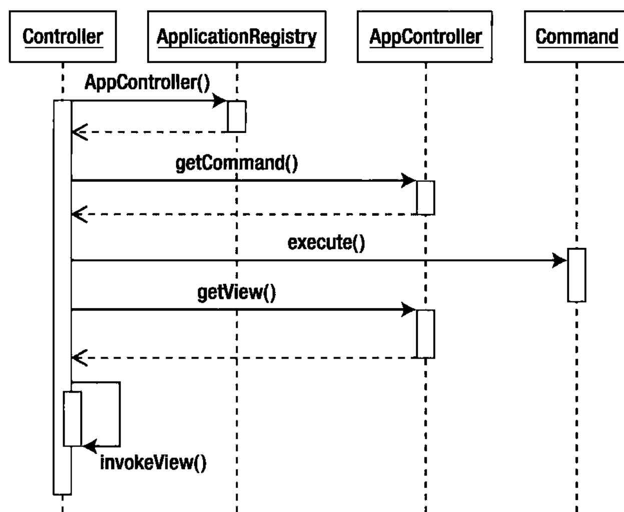


图12-7　通过应用控制器来获得命令和视图

- Command基类

你可能已经注意到了，AppController类需要使用Request对象中存储的之前执行过的命令。这个工作由Command基类完成：

```
namespace woo\command;
//...

abstract class Command {
```

```
    private static $STATUS_STRINGS = array (
        'CMD_DEFAULT'=>0,
        'CMD_OK' => 1,
        'CMD_ERROR' => 2,
        'CMD_INSUFFICIENT_DATA' => 3
    );
    private $status = 0;

    final function __construct() { }

    function execute( \woo\controller\Request  $request ) {
        $this->status = $this->doExecute( $request );
        $request->setCommand( $this );
    }

    function getStatus() {
        return $this->status;
    }

    static function statuses( $str='CMD_DEFAULT' ) {
        if ( empty( $str ) ) { $str = 'CMD_DEFAULT'; }
        // 将字符串转化为状态数
        return self::$STATUS_STRINGS[$str];
    }
    abstract function doExecute( \woo\controller\Request $request );

}
```

Command类定义了一个状态字符串数组（本例中经过了缩减）。statuses()方法用于将一个字符串（如"CMD_OK"）转换为相应的数字，getStatus()方法返回当前Command对象的状态标志。如果你想更严格一些，可以让statuses()方法在失败的时候抛出异常。实际上，如果元素未定义，statuses()将默认返回null。execute()方法使用抽象方法doExecute()返回的值来设置状态标志，并将它缓存到Request对象中。

● 一个具体的Command类

一个AddVenue命令类看起来是如此简单：

```
namespace woo\command;
//...

class AddVenue extends Command {

    function doExecute( \woo\controller\Request $request ) {
        $name = $request->getProperty("venue_name");
        if ( ! $name ) {
            $request->addFeedback( "no name provided" );
            return self::statuses('CMD_INSUFFICIENT_DATA');
        } else {
          $venue_obj = new \woo\domain\Venue( null, $name );
            $request->setObject( 'venue', $venue_obj );
            $request->addFeedback( "'$name' added ({$venue_obj->getId()})" );
```

```
            return self::statuses('CMD_OK');
        }

    }
}
```

在下一章中，你还会遇到这些代码，到时这些代码会具有实际的功能。下面是一个stub对象Venue，它将用于该命令。

```
namespace woo\domain;

class Venue {
    private $id;
    private $name;

    function __construct( $id, $name ) {
        $this->name = $name;
        $this->id = $id;
    }

    function getName() {
        return $this->name;
    }
    function getId() {
        return $this->id;
    }
}
```

现在你只要注意doExecute()方法返回一个状态标志，而抽象基类将状态标志保存在属性中。命令对象被调用并且设置了状态后，系统应该如何响应呢？这完全由配置文件决定。根据之前定义的XML文件，如果返回CMD_OK，转向机制会促使AddSpace类被实例化。通过这种方式，当请求包含cmd=AddVenue时，整个事件链会被触发。如果请求包含cmd=QuickAddVenue，则转向不会发生，而是直接显示quickaddvenue视图。

注意本例中并没有实现将一个Venue对象保存到数据库的功能。下一章将会介绍相关内容。

3. 效果

要实现一个完整的应用控制器模式示例是相当困难的，因为你需要做很多的工作，包括得到和应用元数据（metadata）来描述命令与请求、命令与命令以及命令与视图之间的关系。

正因如此，我只在项目需要时才这样做。当我在命令类中添加条件语句来加载不同的视图或者调用不同的命令时，如果觉得命令和显示的逻辑不太容易控制，才会想到使用应用控制器。

当然，应用控制器可以使用各种机制来创建命令与视图之间的关系，而不仅限于上面介绍的那种方法。即使你的请求字符串、命令名称和视图的关系是固定的，你仍然可以从构建应用控制器中受益。它会给程序带来更大的灵活性，便于你在将来重构代码时适应更复杂的需求。

12.3.3 页面控制器

尽管我很喜欢前端控制器，但并不是所有情况都适合使用前端控制器。前端控制器在前端做了很多工作，适合于较大的系统，而对于那些希望即时返回结果的简单项目来说不太适合。页面

控制器模式则是一个通用的策略，可能你已经在使用。下面我们研究一下页面控制器。

1. 问题

和前端控制器一样，页面控制器也需要处理请求、领域逻辑和表现之间的关系。这些工作对于任何一个企业应用项目来说都是必需的。区别在于对你的约束。

如果你有一个相对简单的项目，使用过于复杂的设计方案没什么意义，还可能会导致项目交付日期延迟，页面控制器正适合于这类项目。

假设我们的Woo项目中想用一个页面给用户展示所有场所的一个列表。即使数据读取部分的代码已经完成，如果前端控制器部分的代码未完成，我们还是无法轻松得到一个可用的结果。

在这个例子中，视图是一个场所列表，请求是获取场所列表。将来这个请求也不会用到其他视图。那么这里最简单的解决方案就是将这个视图与控制器联系起来，放到同一个页面里去。

2. 实现

尽管在现实中采用页面控制器的项目可能会导致各种复杂问题，但该模式本身是很简单的。使用页面控制器时，控制逻辑与一个或多个视图相关联。在最简单的情况下，这意味着控制逻辑被放在视图里面，尽管它可以被分离出来，特别当一个视图与其他视图相连时（这时你可能需要根据不同情况转向到不同页面）。

下面是最简单的页面控制器的代码：

```
<?php
require_once("woo/domain/Venue.php");
try {
    $venues = \woo\domain\Venue::findAll();
} catch ( Exception $e ) {
    include( 'error.php' );
    exit(0);
}

// 下面是默认的页面
?>
<html>
<head>
<title>Venues</title>
</head>
<body>

<h1>Venues</h1>

<?php foreach( $venues as $venue ) { ?>
    <?php print $venue->getName(); ?><br />
<?php } ?>

</body>
</html>
```

上面的代码分为两个部分。视图部分处理显示工作，而控制器部分负责管理请求并调用应用逻辑。尽管视图和控制器被放在同一个页面里（见图12-8），但它们的代码是分开的。

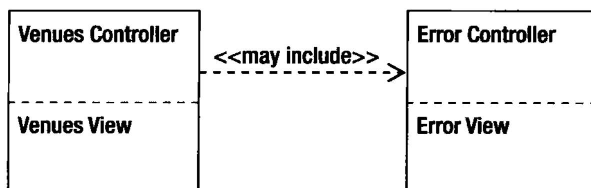


图12-8 页面控制器嵌入在视图中

代码几乎没完成什么实际工作，除了幕后执行的数据库操作（你可以在13.1节看到更多相关内容）。页面上部的PHP代码尝试得到一组Venue对象，然后把它们存储在$venues全局变量中。

如果发生错误，本页面会转向error.php页面（通过include()），然后调用exit()来停止执行本页面后面的内容。我倾向于使用include()的办法来转向，这个办法比通过HTTP来转向好，因为HTTP转向比较耗资源并且会破坏你在内存中已创建的程序环境（例如已经生成了某些数据并保存在内存中）。如果没有执行include()，页面下部的HTML就会被输出。

上面的代码很简单，但在现实中，任何大小和复杂度的项目应该都需要更多的功能支持。

前面的页面控制器代码没有将控制器从视图中很明显地分离出来。下面，我们实现一个页面控制器基类的雏形：

```
namespace woo\controller;
//...

abstract class PageController {
    private $request;
    function __construct() {
        $request = \woo\base\RequestRegistry::getRequest();
        if ( is_null( $request ) ) { $request = new Request(); }
        $this->request = $request;
    }

    abstract function process();

    function forward( $resource ) {
        include( $resource );
        exit( 0 );
    }

    function getRequest() {
        return $this->request;
    }
}
```

本类使用了我们之前见过的一些工具类，尤其是Request类和RequestRegistry类。Page-Controller类主要负责访问一个Request对象，并管理视图的加载。在实际项目中，需求可能会增长很快，我们可以相应地设计越来越多的子类来满足各种需求。

子类可以放在视图内部，并像上面的例子中那样默认显示，也可以从视图中分离出来。我认为分离出来可以使代码更为整洁，因此我采用了这种方法。下面的PageController类尝试添加

一个新场所到系统中：

```
namespace woo\controller;
//...

class AddVenueController extends PageController {
    function process() {
        try {
            $request = $this->getRequest();
            $name = $request->getProperty( 'venue_name' );
            if ( is_null( $request->getProperty('submitted') ) ) {
               $request->addFeedback("choose a name for the venue");
               $this->forward( 'add_venue.php' );
            } else if ( is_null( $name ) ) {
               $request->addFeedback("name is a required field");
               $this->forward( 'add_venue.php' );
            }

            // 创建对象便可将它添加到数据库
            $venue = new \woo\domain\Venue( null, $name );
            $this->forward( "ListVenues.php" );
        } catch ( Exception $e ) {
            $this->forward( 'error.php' );
        }
    }
}

$controller = new AddVenueController();
$controller->process();
```

AddVenueController类中只实现了process()方法。process()负责检查用户提交的数据。如果用户未提交表单，或者表单填写错误，则会加载默认视图（add_venue.php），提供反馈信息并显示表单。如果成功添加一个新用户，那么该方法会调用forward()方法，把用户传递到ListVenues页面控制器。

注意视图文件的格式。我通常习惯将视图文件名全部小写，而对于类文件则采用骆驼命名方式（每个单词首字母大写），以此区分视图文件和类文件。

下面是与AddVenueController类相关的视图文件：

```
<?php
require_once( "woo/base/RequestRegistry.php" );
$request = \woo\base\RequestRegistry::getRequest();
?>
<html>
<head>
<title>Add Venue</title>
</head>
<body>
<h1>Add Venue</h1>

<table>
<tr>
```

```
<td>
<?php
print $request->getFeedbackString("</td></tr><tr><td>");
?>
</td>
</tr>
</table>

<form action="AddVenue.php" method="get">
    <input type="hidden" name="submitted" value="yes"/>
    <input type="text" name="venue_name" />
</form>
</body>

</html>
```

可以看出，视图中的工作就是显示数据，并提供一个可以生成新请求的机制。请求生成后会发送给PageController，而不是返回给视图。记住，只有PageController类才负责处理请求。

你可以在图12-9中看到页面控制器模式更复杂的结构。

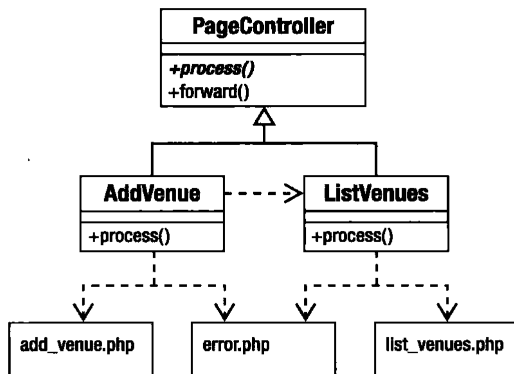


图12-9 页面控制器类体系及其视图加载

3. 效果

页面控制器的优点是非常简单，极易理解，只要有一点Web开发的经验就会使用。我们为venues.php发送一个请求，马上就可以得到我们想要的结果。甚至可能出什么错也是我们可以预料到的，例如“服务器错误”或者“页面不存在”。

如果你将页面控制器与视图分离开来，代码写起来可能会复杂一些，但这种“一对一”的关系仍然相当清晰。

可能会让人迷惑的是视图的加载。当页面控制器执行完某个操作时，会加载一个视图。有时候，你可能需要用同一段代码来加载另一个页面控制器。例如，当AddVenue成功地添加了一个场所，系统不再需要显示添加场所的表单，这时它会委托另一个页面控制器ListVenues来加载另一个视图。你必须清楚何时使用另一个视图，何时使用另一个页面控制器。页面控制器要负责

保证它的视图中存在必需的数据。

尽管页面控制器类可能也会将任务委托给Command对象来完成，但这样做的好处和采用前端控制器时不同。前端控制器类需要判断一个请求的目的，而页面控制器类则已经知道请求的目的。在一个命令对象中执行较少的请求检查和逻辑层调用是很容易的，在页面控制器类中也是如此。这样的好处是你不再需要一个选择Command对象的机制。

代码重复是个问题，但如果使用一个通用的页面控制器基类，就可以尽量减少代码重复，同时也可以减少加载配置数据的时间，因为你可以避免加载当前操作并不需要的数据。当然，在使用前端控制器时也可以做到，但在使用前端控制器时，很难发现哪些数据是当前操作需要的，哪些是不需要的。

当视图调用很复杂时页面控制器最大的缺陷便会显现出来，特别当同一个视图被多次并以不同方式使用时（例如添加或编辑场景）。你会被条件判断语句和状态检查语句弄糊涂，这样代码的可读性就大大降低了。

但一个项目早期使用页面控制器，将来转而使用前端控制器也是可以的，特别在使用PageController基类的时候更是如此。

根据我的经验，如果一个项目我可以花少于一周的时间完成，而且将来该项目也不会变得更加复杂的话，我会选择页面控制器。如果我在开发一个大型项目，它将会不断增长，而且有很复杂的视图逻辑，那么我将使用前端控制器。

12.3.4　模板视图和视图助手

模板视图是PHP天生具有的功能，你可以将表现标记（HTML）和系统代码（PHP代码）结合起来。之前介绍过，这是福亦是祸。因为你能够很方便地将模板和代码结合起来，提高开发的效率，但同时也会给将来的后续开发和长期维护带来很糟的后果。

使用PHP时，在视图里编程是应当尽量避免的。因为如果视图中也混入了业务代码，就会很难维护。

为了解决这个问题，视图助手（View Helper）模式提供了一个助手类，这个类可以针对某个视图设计，也可以由多个视图所共享，总之可以辅助完成更多的任务。

1. 问题

现在已经很少有人在页面里直接嵌入SQL查询和其他业务逻辑了，但这种情况仍没有完全消失。本书前几章已经介绍过这样做的坏处，因此在此只简单说说。

Web页面包含过多的代码会使网页设计师感到为难，因为显示部件被放到了各种循环和条件语句中。

在表现层中加入业务逻辑还会增大代码间的耦合性。如果你想添加一个新视图，就不得不复制粘贴大量的业务逻辑代码到新视图中去。

由于某些操作被多个视图所调用，如果在模板中嵌入应用代码，那么你不得不将某段代码从一个页面复制到另一个页面。这样做的后果就是bug越来越多，代码维护越来越难。

为了防止这种情况发生，你应该把应用处理逻辑放在视图外的一个地方，只允许视图执行“显

示数据”的功能。通常的做法是先得到数据，再将数据传递给视图。当一个视图需要访问系统数据时，你可以提供一个视图助手对象来帮助视图达到目的。

2. 实现

如果你已经搭建好了外部框架，编写一个视图层应该不是很难。当然，很多设计和信息架构的问题仍然存在，但那恐怕要再写一本书才能讨论清楚。

模板视图是Fowler取的名字，它是绝大多数企业级程序员常用的一个模式。在一些编程语言中，实现模板视图的办法是虚构一个模板系统，将特定的标签（tag）解析成系统中的值。在PHP中正是如此。我们可以使用一个模板引擎，例如Smarty。但我个人倾向于直接使用PHP语言本身已经有的功能，只是在使用的时候要谨慎一些。

视图要工作，必须能够获得数据。我喜欢定义一个视图助手来访问Request对象及其他我们需要的对象。

下面是一个简单的视图助手类：

```
namespace woo\view;
class VH {
    static function getRequest() {
        return \woo\base\RequestRegistry::getRequest();
    }
}
```

目前这个类唯一的功能就是访问一个Request对象。但你可以对其进行扩展来获得更多功能。如果你发现自己在视图中写了一段好几行的代码，那么很可能这些代码可以放到视图助手中。在较大的项目中，你可以将视图助手设计为继承层级结构，并实现多个视图助手对象，从而为系统的不同部分提供不同的功能。

下面是一个使用了视图助手和Request对象的简单视图：

```
<?php
require_once("woo/view/ViewHelper.php");
$request = \woo\view\VH::getRequest();  // Controller caches this
$venue = $request->getObject('venue'); // Command caches this
?>

<html>
<head>
<title>Add a Space for venue <?php echo $venue->getName() ?></title>
</head>
<body>
<h1>Add a Space for Venue '<?php print $venue->getName() ?>'</h1>
<table>
<tr>
<td>
<?php print $request->getFeedbackString("</td></tr><tr><td>"); ?>
</td>
</tr>
</table>

<form method="post">
```

```
    <input type="text"
     value="<?php echo $request->getProperty( 'space_name' ) ?>" name="space_name"/>
    <input type="hidden" name="venue_id" value="<?php echo $venue->getId() ?>" />
    <input type="submit" value="submit" />
</form>

</body>
</html>
```

视图文件（add_space.php）通过视图助手得到一个Request对象，并使用该对象中的方法来得到动态数据，然后将数据传递给视图页面。具体地说，getFeedback()方法返回命令对象设置的反馈信息，而getObject()方法可以得到为视图层缓存的任意对象，getProperty()方法用于访问HTTP请求中的任意参数。如果你只运行这个视图，那么Venue和Request对象不可用。回去检查Controller类，看看设置Request对象的位置，再检查一下AddVenue命令类，看看基于Request存储的Venue对象。

你可以将视图助手作为代理，调用Request对象的各种有用的方法，这样视图本身甚至可以不需要持有对Request对象的引用。

显然，这个例子并没有将业务代码从视图中分离出来，但它很严格地限定了业务代码的量和类型。页面中只有简单的print语句和少量的方法调用。页面设计师应该可以很容易地修改这样的模板。

有点麻烦的是if语句和循环。这两个功能不容易委托给视图助手来执行，因为它们通常和特定格式的输出相绑定。我倾向于在模板视图中直接使用简单的条件语句和循环语句（在创建有几行数据的HTML表格时很常用），但尽可能让这些语句保持简单。如果可能，我会把测试语句等委托给视图助手来实现。

3. 效果

如何传递数据给视图层是一个需要考虑的问题，因为视图并没有固定的接口来保证它所处的环境。我希望每个视图和系统之间存在着一个“协议”。视图可以告诉系统：“如果我被调用，我就拥有访问对象A、对象B以及对象C的权利。”由系统来保证做到这一点[①]。

令人惊讶的是，这个办法确实很有效，尽管你可以让视图更为严格（添加某些视图特定的助手类）。如果你这样做了，在助手类中添加访问方法就很安全[②]。我们不再需要通过糟糕的Request::getObject()方法，那只是一个对关联数组的封装。

尽管现在我们可以安全地在视图中访问变量，但是为视图和视图助手类同时维护一个平行的系统是很麻烦的。我倾向于为视图层动态注册对象，通过Request对象、SessionRegistry对象或者RequestRegistry对象。

模板通常是被动的，其中的值来自于上一次请求的结果，但有时候视图可能需要一些额外的请求。视图助手很适合于提供这样的功能，让视图可以轻松获得数据。但即使是视图助手，也应该尽量不做这样的工作，而要通过使用外观模式把这些工作委托给命令对象或者领域层（domain

① 保证调用视图前已经传递必需的数据给视图。——译者注

② 因为可以确定视图中该属性已经定义并有值，可以访问。——译者注

layer）来完成。

注解 本书在第10章中介绍过外观模式。读者可以看看《J2EE核心模式》中外观模式的一个应用实例：会话外观模式（用于细粒度的网络事务）。Fowler还描述了一种称为“服务层”（Service Layer）的模式，用于提供简单接口来访问一个层内部的复杂功能。

12.4 业务逻辑层

如果说控制层精心安排了外部世界与系统内部的通信，那么逻辑层的工作则是处理应用程序的业务（business）部分。业务逻辑层应当远离那些外部的“噪音”，例如查询字符串分析、构建HTML表格和组装返回信息等。“业务逻辑”是整个应用程序的根本目的所在，系统的其他部分都是为这部分服务的。

在一个典型的面向对象应用中，业务逻辑层通常由很多模型类组成，这些类负责为系统要解决的问题建立模型。在后面我们将会看到，如何构建模型是一个灵活的设计决策，它也需要经过慎重的前期规划。

现在开始研究，尽快让系统运转起来。

12.4.1 事务脚本

事务脚本（Transaction Script）模式（见《企业应用架构模式》）描述了不同系统的实现方式。该模式简单易懂而高效，尽管随着系统增长它的优势会下降。事务脚本自己处理请求，而不是委托给特定的对象来完成。它是一种典型的快速工具（quick fix），也是一种很难进行分类的模式，因为它将本章中其他层的几个元素都结合在一起。我把它当做业务逻辑层的一部分，因为事务脚本的目的在于完成系统的业务目的。

1. 问题

发送到系统的每个请求都需要进行处理。我们已经看到，很多系统都有一些访问和过滤请求数据的层，但这些层需要调用一些类来满足要求。我们将会看到，有时候可能需要将这些类拆开，才能实现系统的功能和责任，大概要用到外观接口。这种方式需要你在设计时非常小心。在有些项目中（特别是小型的、工期较紧的项目），在设计上花费过多时间是让人无法接受的。在这种情况下，你可以把业务逻辑写成一系列过程式操作。每个操作处理一个特定的请求。

我们的问题就是如何提供一个快速而有效的机制来满足系统目标，而不需要投入大量时间和精力在复杂的设计上。

事务脚本模式的好处在于你可以很快就得到想要的结果。每个脚本都能很好地处理输入的数据并操作数据库来保证得到想要的结果。除了在同一个类中组织相关的方法外，我们还要确保事务脚本类在它们自己的层中（也就是说尽可能独立于命令层、控制层和视图层）。为此，我们需要做一点前期的规划工作。

业务逻辑层中的类通常与表现层明确地分离开来，但常常会嵌入到数据层中。这是因为获取和保存数据是系统工作的关键。本章后面将介绍如何将业务逻辑对象与数据层分离开。事务脚本

类，则通常了解数据库的所有信息（但它们可以通过入口类来处理实际查询的细节）。

2. 实现

现在回到之前的事件列表示例。在那个示例中，系统有3个关系数据库表：venue、space和event。一个场所（venue）可能有多个空间（space）（例如，一个影院可以有多个舞台，一个舞厅可以有多个房间，等等）。每个空间都是多个事件发生的地方。下面是数据表的结构：

```
CREATE TABLE 'venue' (
  'id' int(11) NOT NULL auto_increment,
  'name' text,
  PRIMARY KEY  ('id')
)
CREATE TABLE 'space' (
  'id' int(11) NOT NULL auto_increment,
  'venue' int(11) default NULL,
  'name' text,
  PRIMARY KEY  ('id')
)
CREATE TABLE 'event' (
  'id' int(11) NOT NULL auto_increment,
  'space' int(11) default NULL,
  'start' mediumtext,
  'duration' int(11) default NULL,
  'name' text,
  PRIMARY KEY  ('id')
)
```

显然，我们的系统需要添加场所和事件的功能。每个功能都可以称为一个事务。我们可以为每个功能设计一个类（并根据命令模式来组织，正如第11章介绍的）。但在这个例子中，我们将把所有方法放到一个类中，但作为继承体系的一部分。其结构如图12-10所示。

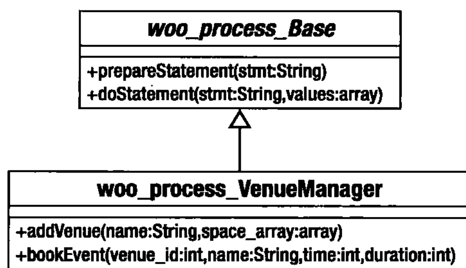


图12-10 事务脚本类及其基类

那么为什么这个例子需要一个抽象基类呢？因为对于任何规模的项目，我们都需要添加更多具体的子类到继承体系中。由于这些子类大多要和数据库打交道，把一些核心的数据库访问功能放到基类里就是一个很好的选择。

事实上，本模式有一个专有的名称（Fowler称之为层超类型[1]），但大多数程序员平时使用时

① Layer Supertype，指某一类型充当某个层中所有类型的超类型。——译者注

并没有想这么多。如果一个层中的多个类有很多共同点，那么将它们放到同一个类型中，把常用功能放在基类中，是很有意义的。在本章后面，你可以体会到这一点。

在本示例中，基类得到一个PDO对象，然后存储在一个静态属性中。它也提供了用于缓存数据库语句和查询数据库的各种类方法。

```
namespace woo\process;
//...

abstract class Base {
    static $DB;
    static $stmts = array();

    function __construct() {
        $dsn = \woo\base\ApplicationRegistry::getDSN( );
        if ( is_null( $dsn ) ) {
            throw new \woo\base\AppException( "No DSN" );
        }

        self::$DB = new \PDO( $dsn );
        self::$DB->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
    }

    function prepareStatement( $stmt_s ) {
        if ( isset( self::$stmts[$stmt_s] ) ) {
            return self::$stmts[$stmt_s];
        }
        $stmt_handle = self::$DB->prepare($stmt_s);
        self::$stmts[$stmt_s]=$stmt_handle;
        return $stmt_handle;
    }

    protected function doStatement( $stmt_s, $values_a ) {
        $sth = $this->prepareStatement( $stmt_s );
        $sth->closeCursor();
        $db_result = $sth->execute( $values_a );
        return $sth;
    }
}
```

我们使用ApplicationRegistry来得到一个数据源（DSN）字符串，然后传递给PDO对象的构造方法。

prepareStatement()方法调用PDO类的prepare()方法，返回一个SQL语句的句柄。该句柄最终将被传递给execute()方法。在prepareStatement()方法中，我们把数据资源（即SQL句柄）缓存在一个静态数组$stmts中。我们使用SQL语句本身作为数组元素的索引。

prepareStatement()方法可以被子类直接调用，但更多情况下是通过doStatement()方法来调用的。doStatement()方法的参数是一个SQL语句和一个混合数组（字符串和整数）。该数组应当包含在SQL语句执行时传递给数据库的值。doStatement()方法在调用prepareStatement()时使用SQL语句，然后调用PDOStatment::execute()方法来获得语句资源。如果

有错误发生，我们就抛出异常。可以看到，这些操作对于事务脚本来说都是隐藏于幕后的。事务脚本只需构造出SQL语句并继续关注业务逻辑即可。

下面是VenueManager类的开始部分，用于设置我们的SQL语句：

```
namespace woo\process;
//...

class VenueManager extends Base {
    static $add_venue =  "INSERT INTO venue
                          ( name )
                          values( ? )";
    static $add_space  = "INSERT INTO space
                          ( name, venue )
                          values( ?, ? )";
    static $check_slot = "SELECT id, name
                          FROM event
                          WHERE space = ?
                          AND (start+duration) > ?
                          AND start < ?";
    static $add_event =  "INSERT INTO event
                          ( name, space, start, duration )
                          values( ?, ?, ?, ? )";
    //...
```

这里没什么新内容，都是些事务脚本将要使用的SQL语句。它们被构造成能够被PDO的prepare()方法接受的格式。问号代表占位符，值将被传递给execute()方法。

下面让我们看看设计用来满足某个特定业务需求的第一个类方法：

```
function addVenue( $name, $space_array ) {
    $ret = array();
    $ret['venue'] = array( $name );
    $this->doStatement( self::$add_venue, $ret['venue']);
    $v_id = self::$DB->lastInsertId();
    $ret['spaces'] = array();
    foreach ( $space_array as $space_name ) {
        $values = array( $space_name, $v_id );
        $this->doStatement( self::$add_space, $values);
        $s_id = self::$DB->lastInsertId();
        array_unshift( $values, $s_id );
        $ret['spaces'][] = $values;
    }
    return $ret;
}
```

可以看到，addVenue()方法的参数是一个场所名称和一个包含多个空间名称的数组。该方法使用这两个参数来为venue表和space表赋值。同时，它也创建了一个包含相关信息的数据结构，每一行数据都有一个新生成的ID值。

addVenue()方法中省去了很多数据库操作，交由基类来实现。我们把场所名称传递给doStatement()方法。如果有错误发生，将会抛出一个异常。我们在此并未捕捉任何异常，因此doStatement()（通过扩展）或prepareStatement()方法抛出的任何异常也会被addVenue()

抛出。这正是我们想要的结果，但我们需要在代码文档中作出说明，说明本方法抛出异常的情况。

创建了场所后，我们遍历$space_array数组，对应每一个数组元素在space数据表中添加一行记录。注意我们为每一行新增的记录添加了一个外键，其值为场所的ID，这样就可以把space的记录和venue关联起来。

第二个事务脚本同样很简单：

```
function bookEvent( $space_id, $name, $time, $duration ) {
    $values = array( $space_id, $time, ($time+$duration) );
    $stmt = $this->doStatement( self::$check_slot, $values, false ) ;
    if ( $result = $stmt->fetch() ) {
        throw new \woo\base\AppException( "double booked! try again" );
    }
    $this->doStatement( self::$add_event,
        array( $name, $space_id, $time, $duration ) );
}
```

这段事务脚本的目的是添加一个事件到events表中，和一个空间相关联。注意，我们使用了$check_slot中包含的SQL语句来确保要添加的事件不与同一个空间的另一个事件冲突。

3. 效果

事务脚本模式是快速获得结果的有效途径，同时也是程序员们使用了多年，但没有想到要为之命名的模式之一。通过我们添加到基类中的几个助手方法，你可以关注应用逻辑而不需要花太多时间在数据库存取上。

有时候事务脚本并不是那么受人欢迎。原本我认为自己在写一个较为复杂而且要大量使用面向对象的应用程序，但随着项目截止日期的临近，我发现自己将越来越多的业务逻辑和控制器放在一起，而那些逻辑我原本是想作为领域模型（见下一节）之上的一个层的。尽管这样做使得代码不太优雅，但我不得不承认项目并不必要因此而重新设计。

在大部分情况下，你会在开发小型项目时使用事务脚本，特别当你确定它不会成长为一个大型项目的情况下。这种方式使项目不太容易扩展，因为事务脚本总是不可避免地互相渗入，从而导致代码重复。你可以通过一些办法来重构代码，但很难完全消除这些问题。

在我们的例子中，我们直接在事务脚本类中嵌入数据库操作代码。你可以知道，我们其实应当把数据库操作和项目的应用逻辑分离开来。我们可以将数据库操作从事务脚本类中抽取出来，然后创建一个入口类来让事务脚本与数据库进行交互。

12.4.2 领域模型

领域模型是原始的逻辑引擎，是本章中其他模式想要创建、照顾和保护的。它是对项目中各种个体的抽象表达。它也是一个平台，你的业务问题可以在其上展示出来，而不需要纠缠于各种技术细节，如数据库和网页等。

如果这么说有点天花乱坠的感觉，那么让我们把它拉近现实吧。领域模型象征着真实世界里项目中的各个参与者。“万物皆对象”的原则在此体现得淋漓尽致。在其他地方对象总是带着种种具体的责任，而在领域模型中，它们常常描述为一组属性及附加的代理。它们是做某些事的某些东西。

1. 问题

如果使用过事务脚本模式，可能会发现当不同的事务脚本要执行相同的任务时，重复会变成一个问题。有时候我们可以通过重构代码来解决，但慢慢地复制粘贴可能成了开发中难以避免的一部分。

你可以使用领域模型来抽象和具体化系统中的参与者和操作过程。例如，你可以创建`Space`类和`Event`类，而不是通过脚本来添加space数据到数据库中然后关联到事件。将事件和空间相连，只需要调用`Space::bookEvent()`即可，非常简单。类似地，检查时间是否冲突可以使用`Event::intersects()`，等等。

显然，对于Woo这样的简单例子来说，事务脚本更实用，而且足以满足要求。但当领域逻辑变得复杂后，领域模型就变得很有吸引力了。复杂的逻辑可以很容易地被处理，而且当你为应用领域建模之后，不再需要那么多条件语句了。

2. 实现

从设计角度来说，领域模型是很简单的。领域模型的复杂性主要来自于尝试使模型纯粹(pure)，即将模型从应用中其他层分离出来。

将领域模型的参与者从表现层分离出来并不难，只要确保这些参与者保持独立即可，但将这些参与者与数据层分离开则不太容易。在理想状态下，领域模型应当只包含它要表达和解决的问题，但在现实中领域模型很难完全除去数据库操作。

将领域模型类直接映射到关系数据库的表是很常见的做法，这会使开发变得简单。图12-11是一个类图，描述了Woo系统中的参与者。

图12-11中的3个对象与3个数据表相对应（数据表在事务脚本示例中定义过）。这种直接的映射使系统变得易于管理，但在现实中并不总是可行的，特别当数据库在应用程序之前就已经存在的情况下。有时对象与表的关联本身就可能导致问题。如果一不小心，很可能就会在模型中放入过多数据库操作，而不是那些你想解决的问题。

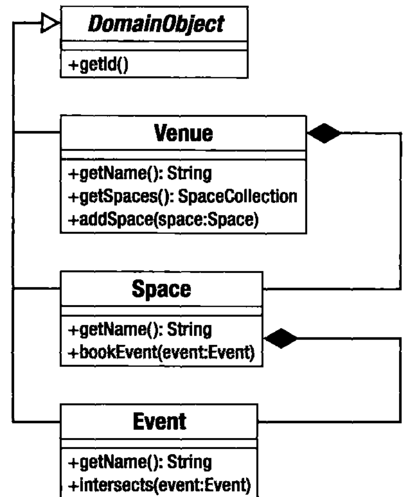


图12-11　领域模型的简单示例

领域模型常常映射到数据库结构上，但这并不意味着模型类应该了解数据库相关的信息。通过将模型与数据库分离，整个层会更易于测试，更不会受到数据库结构改变的影响，甚至不会受到存储机制的影响。领域模型只关注每个类本身要完成的核心工作和承担的责任。

下面是一个简化过的`Venue`对象及其父类：

```
namespace woo\domain;


abstract class DomainObject {
    private $id;
```

```
    function __construct( $id=null ) {
        $this->id = $id;
    }

    function getId( ) {
        return $this->id;
    }

    static function getCollection( $type ) {
        return array(); // dummy
    }

    function collection() {
        return self::getCollection( get_class( $this ) );
    }
}


class Venue extends DomainObject {
    private $name;
    private $spaces;

    function __construct( $id=null, $name=null ) {
        $this->name = $name;
        $this->spaces = self::getCollection("\\woo\\domain\\Space");
        parent::__construct( $id );
    }

    function setSpaces( SpaceCollection $spaces ) {
        $this->spaces = $spaces;
    }

    function getSpaces() {
        return $this->spaces;
    }

    function addSpace( Space $space ) {
        $this->spaces->add( $space );
        $space->setVenue( $this );
    }

    function setName( $name_s ) {
        $this->name = $name_s;
        $this->markDirty();
    }

    function getName( ) {
        return $this->name;
    }
}
```

这个类和不需要持久化的类之间有些不同。我们使用了SpaceCollection类型的对象（而不是数组）来存储Venue可能包含的Space对象。（不论你是否使用数据库，类型安全的数组都是不

错的选择）。因为本类使用了一个特别的集合对象而不是Space对象数组，构造方法需要在起始处实例化一个空对象。这通过调用基类的一个静态方法来实现：

```
$this->spaces = self::getCollection("\\woo\\domain\\Space");
```

注解　本章和下一章都将讨论Venue和Space对象的修正。这是两个简单的域对象，共用功能核心。如果你正在编码，那么应该能够将我介绍的概念应用到其中的一个类。Space类可能不能维护Space对象的集合，但它能以同样的方式管理Event对象。

下一章将会详细介绍这里使用的集合对象。现在，我们只要知道超类返回一个空数组就可以了。

构造方法的参数是$id，它被传递给基类并保存起来。$id参数表示数据表中某条记录的唯一ID。注意我们也调用了基类中的markDirty()方法［该方法将在工作单元（Unit of Work）模式中介绍］。

3. 效果

领域模型设计得简单还是复杂取决于业务逻辑的复杂度。使用领域模型的好处是：当你设计模型时可以专注于系统要解决的问题，而其他问题（如持久化和表现等）可以由其他层来解决——至少理论上是这样的。

在实际项目中，我相信绝大多数程序员在设计领域模型时还是把一半的注意力都放在数据库上。因为你一定不希望设计出一个系统结构后，你（甚至你的同事）不得不编写复杂的数据存取代码。

在设计和规划阶段，将领域模型和数据层分离会导致一定的代价。你也可能会将数据库代码直接放入模型中（尽管你可能会使用一个数据入口来处理实际的SQL）。对于相对简单的模型，特别当类与数据表一一对应时，这种方法是完全可行的，可以减少因协调对象和数据库而创建外部系统导致的时间耗费。

12.5　小结

本章包含了大量基础内容（虽然也省略了很多内容），希望读者没有被本章的大量代码吓倒。设计模式应该被应用在合适的地方，并在必要的时候组合使用。当你觉得项目需要时，可以使用本章介绍过的模式，但不要认为在开始项目前一定要打造出一个完整的框架。本章介绍的内容基本上足以构成一个框架的基础。而且，也从架构方面为你选择部署已有框架提供了深入的建议。

但要打造一个完整的框架，只使用本章的内容还不足够。本章仅仅提到了持久化的部分内容，如集合和映射，但并没有深入介绍。在下一章中，我们将介绍数据库相关的设计模式，以将对象从数据存储的细节中解放出来。

第 13 章

数据库模式

不论程序是否复杂，绝大多数的Web应用程序都需要在一定程度上进行数据持久化操作[①]。例如商店需要记录产品和顾客的相关信息，游戏需要获得玩家和游戏的状态，社交网站要追踪你的238位朋友以及你对20世纪八九十年代的流行乐队的喜爱程度等。无论是什么类型的应用程序，都会有在后台存储数据的时候。在本章中，我们将讨论相关的一些模式。

本章包括如下内容。

- 数据层接口：定义存储层和其他部分之间的接口。
- 对象监听：跟踪对象，避免重复，自动保存和插入数据。
- 灵活查询：允许客户端程序员在构建查询时不考虑底层的数据库。
- 生成结果对象的列表：创建可迭代的数据集合。
- 管理数据库组件：让我们再次使用抽象工厂模式。

13.1 数据层

当和客户交流时，主要围绕着表现层，例如字体、颜色以及易用性通常都是讨论的主要内容。但当和开发人员讨论问题时，最经常讨论的通常是数据库。数据库本身不是问题，我们可以信任数据库本身的能力，除非运气太差。问题在于如何将数据库表中的行和列转换成项目中的数据结构。本章的示例代码可以帮助我们深入研究这个主题。

本章介绍的内容并非都是独立存在于数据层中。我们常常组合使用多个模式来解决持久性问题。Clifton Nock、Martin Fowler和Alur等人都曾经描述过这些模式。

13.2 数据映射器

如果你认为我在12.4.2节没有深入介绍如何在数据库中存储和获取Venue对象的问题，那么这里你可以得到答案，那就是Alur等人在《J2EE核心模式》一书中（在该书中作为一种数据访问对象）以及Martin Fowler在《企业应用架构模式》一书中介绍过的数据映射器（Data Mapper）模式。实际上，数据访问对象的说法并不准确，因为它还生成数据传输对象（data transfer object），

① 将数据以某种方式长期存储起来。——译者注

但这些说法已经相当接近。

可以想到，数据映射器是一个负责将数据库数据映射到对象的类。

13.2.1　问题

对象间的组织关系和关系数据库中的表是不同的。就像你知道的那样，数据库表可以看成是由行和列组成的格子。表中的一行可以通过外键和另一个表（甚至同一个表）中的一行关联，而对象的组织关系更为复杂：一个对象可能包含其他对象；不同的数据结构可能通过不同的方式组织相同的对象，例如在代码运行时组合或重新组合对象形成新的关系。关系数据库是一个管理大量表格式数据的优秀解决方案，但是类和对象通常封装更小的集中式信息块。

类和关系数据库之间的这种分歧通常被称为“对象关系阻抗不匹配”或简单称为“阻抗不匹配”。

那么该如何转换两者的数据呢？可以设计一个独立的类（或一组类）来负责这个工作，从而有效地在领域模型中隐藏数据库操作并管理数据转换中不可以避免的冲突。

13.2.2　实现

即使小心地编写代码，也有可能只用一个mapper类就为多个对象服务，但在使用领域模型时，通常习惯为每一个领域类实现一个映射类。

图13-1展示了3个具体的mapper类和一个抽象父类。

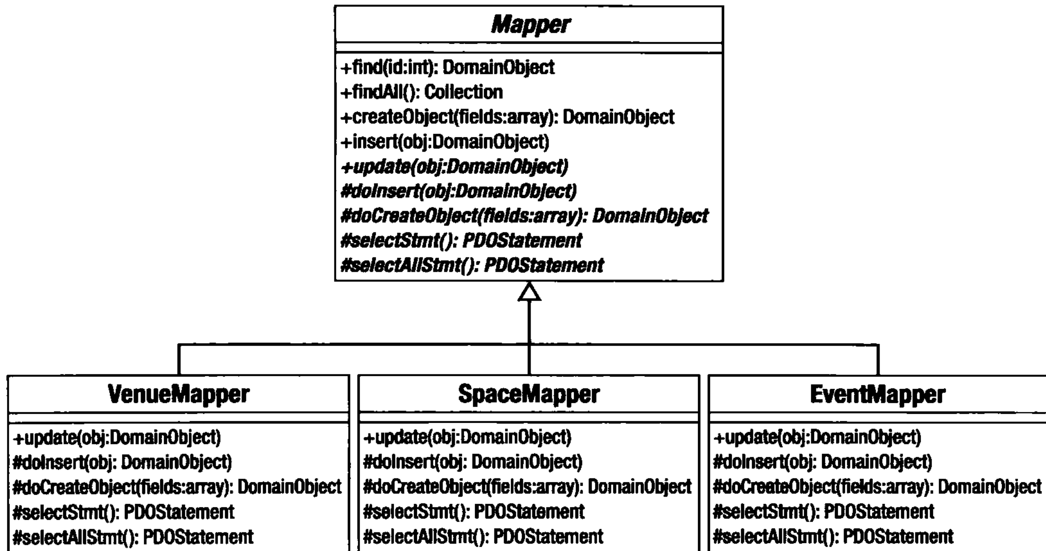


图13-1　映射器类

实际上，由于Space对象属于Venue对象，SpaceMapper类也可以并入VenueMapper类。但为了方便，在此将它们区分开来。

可以看到，这些类共同的操作是保存和加载数据。基类实现公共的功能，而处理特殊对象的操作则分派给子类来实现，例如创建对象和构造数据库查询等。

基类通常在一个操作之前或之后执行内务处理（housekeeping），这就是为什么模块方法（Template Method）模式被用于显式地分派（比如，从具体的方法insert()调用抽象方法doInsert()）。子类决定了实现基类中的哪些方法，本章稍后将会看到这一点。

下面是简化过的Mapper基类：

```
namespace woo\mapper;
//...

abstract class Mapper {
    protected static $PDO;
    function __construct() {

        if ( ! isset(self::$PDO) ) {
            $dsn = \woo\base\ApplicationRegistry::getDSN( );
            if ( is_null( $dsn ) ) {
                throw new \woo\base\AppException( "No DSN" );
            }
            self::$PDO = new \PDO( $dsn );
            self::$PDO->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
        }
    }

    function find( $id ) {
        $this->selectStmt()->execute( array( $id ) );
        $array = $this->selectStmt()->fetch( );
        $this->selectStmt()->closeCursor( );
        if ( ! is_array( $array ) ) { return null; }
        if ( ! isset( $array['id'] ) ) { return null; }
        $object = $this->createObject( $array );
        return $object;
    }

    function createObject( $array ) {
        $obj = $this->doCreateObject( $array );
        return $obj;
    }

    function insert( \woo\domain\DomainObject $obj ) {
        $this->doInsert( $obj );
    }

    abstract function update( \woo\domain\DomainObject $object );
    protected abstract function doCreateObject( array $array );
    protected abstract function doInsert( \woo\domain\DomainObject $object );
    protected abstract function selectStmt();
}
```

构造方法使用ApplicationRegistry来获得DSN，以便供PDO扩展使用。你可以使用请求作用域的注册表对象或独立的单例对象来获得DSN。从控制层传递数据到映射器并非总是明智

之举。另一种创建映射器的方法是由Registry类来处理。映射器不是在类内部实例化PDO对象，而是将PDO对象作为构造方法的参数来传入类中。

```
namespace woo\mapper;
//...
abstract class Mapper {
    protected $PDO;
    function __construct( \PDO $pdo ) {
        $this->pdo = $pdo;
    }
}
```

客户端代码可以通过\woo\base\Request类中的Registry::getVenueMapper()方法从Registry对象中获得一个新的VenueMapper。getVenueMapper()方法将会实例化一个Mapper，并生成该PDO对象。对于以后的请求，该方法会返回缓存着的Mapper。这里的代价是Registry必须获得系统的更多信息，而Mapper不需要和全局配置数据打交道。

insert()方法除了把真正的操作分配给doInsert()之外，什么都不做。我们把insert()方法声明为抽象方法，由子类中的方法来负责具体工作。

find()方法负责调用预编译过的SQL语句（由具体子类提供）并获得行数据，然后调用createObject()方法将数据从数组转换为对象。当然，将数组转换为对象可以有多种方式，因此这些细节由抽象方法doCreateObject()来处理。createObject()也只是一个方法定义，将真正要执行的操作委托给具体子类来完成。稍后我们将增加内务处理，这样使用模板方法模式才有意义。

根据不同的条件，子类中也将定制各种方法来查找数据（比如，我们想要查找属于Venue对象的Space对象）。

我们可以从子类的角度来看看这个过程：

```
namespace woo\mapper;
//...

class VenueMapper extends Mapper {
    function __construct() {
        parent::__construct();
        $this->selectStmt = self::$PDO->prepare(
                            "SELECT * FROM venue WHERE id=?");
        $this->updateStmt = self::$PDO->prepare(
                            "update venue set name=?, id=? where id=?");
        $this->insertStmt = self::$PDO->prepare(
                            "insert into venue ( name )
                             values( ? )");
    }

    function getCollection( array $raw ) {
        return new SpaceCollection( $raw, $this );
    }
    protected function doCreateObject( array $array ) {
        $obj = new \woo\domain\Venue( $array['id'] );
```

```
        $obj->setname( $array['name'] );
        return $obj;
    }

    protected function doInsert( \woo\domain\DomainObject $object ) {
        print "inserting\n";
        debug_print_backtrace();
        $values = array( $object->getName() );
        $this->insertStmt->execute( $values );
        $id = self::$PDO->lastInsertId();
        $object->setId( $id );
    }

    function update( \woo\domain\DomainObject $object ) {
        print "updating\n";
        $values = array( $object->getName(), $object->getId(), $object->getId() );
        $this->updateStmt->execute( $values );
    }

    function selectStmt() {
        return $this->selectStmt;
    }
}
```

这个类也经过了精简，但保留了主要功能。构造方法预编译了一些SQL语句，以供将来使用。这个工作也可以放到静态方法中去，以便被所有VenueMapper对象实例共享；或者像之前说的那样，可以将一个Mapper对象存放在Registry中，从而节省重复实例化的开销。这些代码重构的工作留给读者来完成。

Mapper类实现了find()方法，它调用selectStmt()来获得预编译过的SELECT语句。如果一切顺利，Mapper将调用VenueMapper::doCreateObject()，这里，我们使用关联数组来生成Venue对象。

13

从客户端代码的角度来看，这个过程很简单：

```
$mapper = new \woo\mapper\VenueMapper();
$venue = $mapper->find( 12 );
print_r( $venue );
```

print_r()方法可以快速判断find()是否执行成功。在我的系统中（venue表中有一行记录的ID为12），这个代码片段的输出结果如下所示：

```
woo\domain\Venue Object
(
    [name:woo\domain\Venue:private] => The Eyeball Inn
    [spaces:woo\domain\Venue:private] =>
    [id:woo\domain\DomainObject:private] => 12
)
```

doInsert()和update()方法的处理过程与find()相反，它们都以DomainObject为参数，从中取出记录数据，然后调用PDOStatement::execute()。注意，doInsert()方法在传入的

DomainObject对象上设置了一个ID。由于在PHP中对象是通过引用来传递的，设置ID后客户端代码可以看到这个改变。

另一点要注意的是，doInsert()和update()并不是真正的类型安全。它们接受任何DomainObject对象而不会报错或者发出警告。你可以使用instanceof来检测对象，如果传递的是错误的对象，就抛出异常。这样可以预防出现bug。

再次从客户端代码的角度来看插入和更新操作：

```
$venue = new \woo\domain\Venue();
$venue->setName( "The Likey Lounge-yy" );
// 插入对象到数据库
$mapper->insert( $venue );
// 从数据库中读出刚才插入的对象，验证插入操作是否生效
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
// 修改对象
$venue->setName( "The Bibble Beer Likey Lounge-yy" );
// 调用update来更新记录
$mapper->update( $venue );
// 再次读出对象数据，验证更新操作是否生效
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
```

处理多行记录

find()方法非常简单，因为它只需要返回一个对象。如果需要从数据库中获得大量的数据，该怎么做呢？第一种办法是返回一个对象数组。这样做是可行的，但也存在问题。

如果返回一个对象数组，那么数组中的每个对象都需要先被实例化，如果你的数组中有1 000个对象，那么实例化这么多对象会导致不必要的资源浪费。另一种办法是返回数组，然后让调用代码来实例化对象。这也是可行的，但是它偏离了Mapper类的核心目的。

其实还有一种方法来解决这个问题——使用PHP内置的Iterator接口。

实现Iterator接口的类必须定义一些用于查询数据集的方法。这样，你的类就能够像数组一样通过foreach循环遍历。有些人认为PHP的数组支持如此强大，Iterator接口实现是没有必要的。我将会告诉你至少有3个理由让我们使用PHP内置的Iterator接口。

表13-1显示了Iterator接口要求实现的方法。

表13-1　Iterator接口定义的方法

名　称	描　述
rewind()	指向列表开头
current()	返回当前指针处的元素
key()	返回当前的键（比如，指针的值）
next()	返回当前指向的元素并且将指针向前移动一步
valid()	确定当前指针处有一个元素

为了实现一个Iterator，需要实现它的方法并跟踪当前元素在数据集中的位置。对于客户端代码来说，如何获取数据，如何对数据进行排序或者筛选都是隐藏于幕后的。

下面的类实现了Iterator接口，其中封装了一个数组，同时将一个Mapper对象作为构造方法的参数，你很快就会明白为什么要这样做。

```
namespace woo\mapper;
//...

abstract class Collection implements \Iterator {
    protected $mapper;
    protected $total = 0;
    protected $raw = array();

    private $result;
    private $pointer = 0;
    private $objects = array();

    function __construct( array $raw=null, Mapper $mapper=null ) {
        if ( ! is_null( $raw ) && ! is_null( $mapper ) ) {
            $this->raw = $raw;
            $this->total = count( $raw );
        }
        $this->mapper = $mapper;
    }

    function add( \woo\domain\DomainObject $object ) {
        $class = $this->targetClass();
        if ( ! ($object instanceof $class ) ) {
            throw new Exception("This is a {$class} collection");
        }
        $this->notifyAccess();
        $this->objects[$this->total] = $object;
        $this->total++;
    }

    abstract function targetClass();

    protected function notifyAccess() {
        // 暂时留空
    }
    private function getRow( $num ) {
        $this->notifyAccess();
        if ( $num >= $this->total || $num < 0 ) {
            return null;
        }
        if ( isset( $this->objects[$num]) ) {
            return $this->objects[$num];
        }

        if ( isset( $this->raw[$num] ) ) {
            $this->objects[$num]=$this->mapper->createObject( $this->raw[$num] );
            return $this->objects[$num];
        }
    }

    public function rewind() {
       $this->pointer = 0;
    }

   public function current() {
```

13

```
            return $this->getRow( $this->pointer );
        }

        public function key() {
            return $this->pointer;
        }

        public function next() {
             $row = $this->getRow( $this->pointer );
             if ( $row ) { $this->pointer++; }
             return $row;
        }

        public function valid() {
            return ( ! is_null( $this->current() ) );
        }
    }
```

调用构造方法的时候可以不使用参数，也可以使用两个参数（最终被转换为一组对象的原始数据和一个映射器的引用）。

假设客户端代码提供了$raw参数（客户端程序可能是一个Mapper对象），那么$raw参数就会和$mapper参数一起被存放到一个类属性中。如果提供了$raw参数，那么同时也需要Mapper的一个实例，因为需要使用Mapper对象把每行记录转换为一个对象。

如果没有传递参数给构造方法，那么类中的数据为空，但我们可以通过add()方法添加数据到集合中。

类有两个数组：$objects和$raw。如果客户需要一个特定的元素，getRow()方法先在$objects对象中查找元素是否已经被实例化。如果是的话，则返回它。否则，方法在$raw中查找原始数据。只有Mapper对象存在的时候，才会显示$raw数据，因此相应的行数据可以传递给我们之前遇到的Mapper:: createObject()方法。它返回一个缓存在$objects数组中具有相应索引的DomainObject对象。最新创建的DomainObject对象被返回给用户。

类的其他部分很简单，就是处理$pointer属性，以及调用getRow()。另外还有一个notify-Access()方法。本章稍后会介绍延迟加载（Lazy Load）模式，在该模式中notifyAccess()方法很重要。

你可能已经注意到Collection类是一个抽象类。我们需要为每个领域类提供一个特定的实现：

```
namespace woo\mapper;
//...


class VenueCollection
        extends Collection
        implements \woo\domain\VenueCollection {

    function targetClass( ) {
        return "\woo\domain\Venue";
```

```
    }
}
```

VenueCollection类继承自Collection类，并实现了targetClass()方法。它和父类中的add()方法一起检查对象类型，确保只有Venue对象才可以添加到数据集合中。如果你想更安全的话，也可以在构造方法中进行额外的类型检查。

很明显，这个类只能和VenueMapper一起工作。但在实际情况中，这是一个类型安全（type-safe）的集合，特别是从领域模型的角度考虑。

当然，Event和Space对象也有相应的类。

注意，VenueCollection实现了一个接口：woo\domain\VenueCollection。这是独立接口（Separated Interface）技术的一部分，我稍后会介绍这种技术。实际上，利用VenueCollection，domain包可以定义相应的Collection类，这个类与mapper包中的Collection类不同。Domain对象指的是woo\domain\VenueCollection对象而不是woo\mapper\VenueCollection对象。这样，过一段时间之后，mapper实现就可能会被删除，用domain包中的完全不同的类代替，且不会有太多修改。

```
namespace woo\domain;


interface VenueCollection extends \Iterator {
    function add( DomainObject $venue );
}

interface SpaceCollection extends \Iterator {
    function add( DomainObject $space );
}

interface EventCollection extends \Iterator {
    function add( DomainObject $event );
}
```

图13-2展示了一些集合类（Collection）。

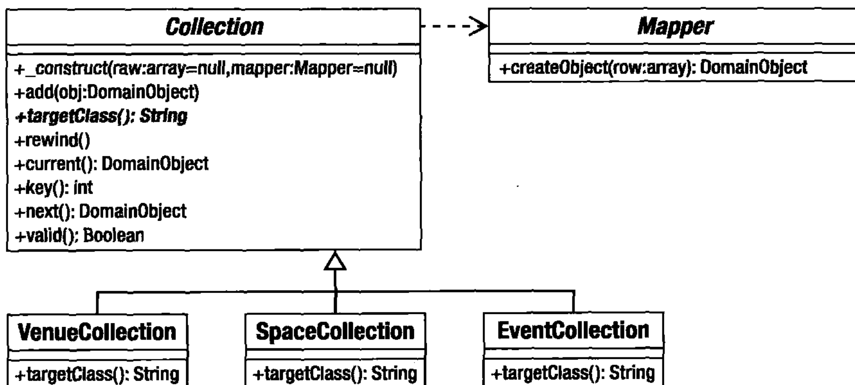


图13-2 用集合类管理多行记录

因为领域模型需要实例化Collection对象，而且有时候（特别是测试的时候）我们可能需要使用不同的集合子类，所以在领域层上提供了一个工厂类来生成Collection对象。下面获得一个空的VenueCollection对象：

```
$collection = \woo\domain\HelperFactory::getCollection("woo\\domain\\Venue");
$collection->add( new \woo\domain\Venue( null, "Loud and Thumping" ) );
$collection->add( new \woo\domain\Venue( null, "Eeezy" ) );
$collection->add( new \woo\domain\Venue( null, "Duck and Badger" ) );

foreach( $collection as $venue ) {
    print $venue->getName()."\n";
}
```

以上这个集合对象没有多少功能，但读者可以尝试自行添加一些功能，例如添加element-At()、deleteAt()、count()等方法。

父类DomainObject可以提供便捷方法来得到集合对象。

```
// 命名空间woo\domain;
// ...

// DomainObject

    static function getCollection( $type ) {
        return HelperFactory::getCollection( $type );
    }

    function collection() {
        return self::getCollection( get_class( $this ) );
    }
```

这个类有两种获得Collection对象的途径：静态方法和实例化方法。无论是静态方法还是实例化方法，都只是使用类名调用HelperFactory::getCollection()。在第12章领域模型的例子中，我们使用过静态方法getCollection()。图13-3展示了HelperFactory类。注意它既可以用于获取集合，也可以用于获取映射器。

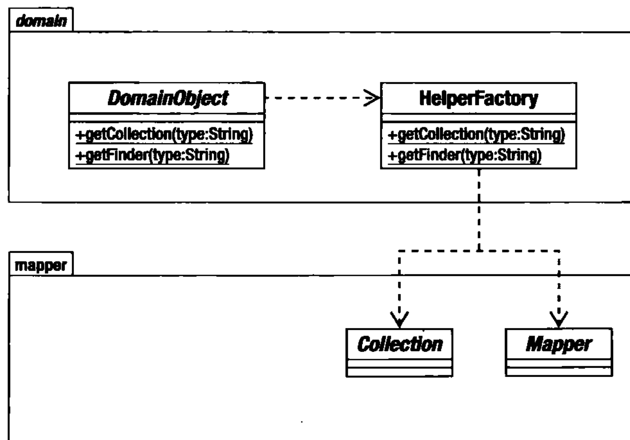


图13-3　使用工厂对象作为获取持久化工具的媒介

按照图13-3所示的结构，你需要在domain包中为Mapper和Collection创建接口。这样，领域对象可以与Mapper包（除了HelperFactory本身）完全隔离开来。Fowler把这个基础模式称为“分离接口”（Separated Interface）。这个模式很有用，因为有的用户有时候需要放弃整个映射器，转而使用另一个映射器。如果实现了分离接口模式，就可以使用getFinder()方法来返回一个Finder接口，而Mapper对象会实现这个接口。然而在大部分示例中，我们把这个改进工作留到以后来完成。在示例中，我们还是用getFinder()返回干净和简单的Mapper对象。

鉴于此，可以对Venue类进行扩展，用于管理Space对象的持久化。Venue类可以提供方法来添加Space对象到SpaceCollection或设置一个全新的SpaceCollection。

```
// Venue
// 命名空间woo\domain;
// ...

    function setSpaces( SpaceCollection $spaces ) {
        $this->spaces = $spaces;
    }

    function getSpaces() {
        if ( ! isset( $this->spaces ) ) {
            $this->spaces = self::getCollection("woo\\domain\\Space");
        }
        return $this->spaces;
    }

    function addSpace( wSpace $space ) {
        $this->getSpaces()->add( $space );
        $space->setVenue( $this );
    }
```

13

setSpaces()被VenueMapper类用于构建Venue对象。该方法建立在集合中所有Space对象都指向当前Venue的基础上。我们可以很容易地在该方法中增加检查，但现在先省去检查代码，保持代码简洁。注意调用getSpaces()方法时，只实例化了$spaces属性。稍后，我们将解释如何通过延迟实例化（lazy instantiation）来减少数据库请求。

VenueMapper需要为它所创建的每个Venue对象建立一个SpaceCollection。

```
// VenueMapper
// 命名空间woo\mapper;
// ...

    protected function doCreateObject( array $array ) {
        $obj = new w\woo\domain\Venue( $array['id'] );
        $obj->setname( $array['name'] );
        $space_mapper = new SpaceMapper();
        $space_collection = $space_mapper->findByVenue( $array['id'] );
        $obj->setSpaces( $space_collection );
        return $obj;
    }
```

VenueMapper::doCreateObject()方法得到一个SpaceMapper并从中获得一个Space-

Collection。可以看到，SpaceMapper类实现了findByVenue()方法。我们可以通过该方法查询数据库，得到包含多个对象的结果集。为了简洁，省略了之前woo\mapper\Mapper代码中的Mapper::findAll()方法。下面重新贴出来：

```
// Mapper
// namespace woo\mapper;
// ...

  function findAll( ) {
        $this->selectAllStmt()->execute( array() );
        return $this->getCollection(
            $this->selectAllStmt()->fetchAll( PDO::FETCH_ASSOC ) );
   }
```

findAll()方法调用了子方法selectAllStmt()，该子方法和selectStmt()一样，需要有一个预编译的SQL语句对象，才能获得表中的所有记录。下面是由SpaceMapper类创建的PDOStatement对象：

```
// SpaceMapper::__construct()
        $this->selectAllStmt = self::$PDO->prepare(
                            "SELECT * FROM space");
//...
        $this->findByVenueStmt = self::$PDO->prepare(
                            "SELECT * FROM space where venue=?");
```

上面的代码包含了另外一个SQL语句——$findByVenueStmt，它用于查找某个Venue中的所有Space对象。

findAll()方法调用了另一个新方法——getCollection()，并把所找到的数据传递给它。下面是SpaceMapper::getCollection()方法的代码：

```
function getCollection( array $raw ) {
    return new SpaceCollection( $raw, $this );
}
```

在一个完整的Mapper类中，应该将getCollection()和selectAllStmt()声明为抽象方法，这样所有的映射器都能够返回一个包含持久领域对象的集合。但为了获得属于某个Venue的所有Space对象，我们需要对返回的集合进行一定的限制。我们已经看到用于获取数据的预编译SQL语句，下面是生成结果数据集合的SpaceMapper::findByVenue()方法的代码：

```
function findByVenue( $vid ) {
    $this->findByVenueStmt->execute( array( $vid ) );
    return new SpaceCollection(
        $this->findByVenueStmt->fetchAll(), $this );
}
```

除了使用的SQL语句之外，findByVenue()方法与findAll()方法基本相同。用findByVenue()方法得到结果集后，Venue对象通过Venue::setSpaces()方法来设置结果集。

通过这种方式，Venue对象得到了数据库中的最新数据，连同它们的Space对象一起放在一个类型安全的集合中。在未被请求之前，集合中的对象不会被实例化。

图13-4展示了客户端类获取SpaceCollection的过程，以及SpaceCollection类如何与SpaceMapper::createObject()交互以将原始数据转换为返回给客户端的对象。

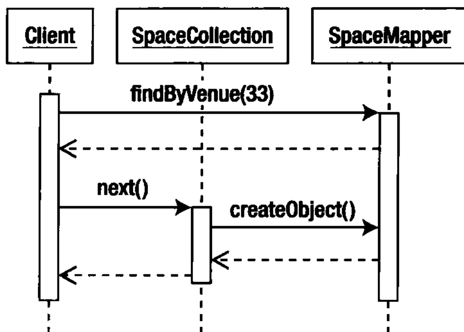


图13-4 获取SpaceCollection并用它获得一个Space对象

13.2.3 效果

使用映射器添加Space对象到Venue的缺点是不得不两次访问数据库。在大多情况下，这是值得付出的代价。需要注意的另一点是，Venue::doCreateObject()获取SpaceCollection的工作也可以移到Venue::getSpaces()中进行，这样第二次的数据库连接只在需要时才发生。getspaces()方法如下所示：

```
// Venue
// 命名空间woo\domain;
// ...

    function getSpaces() {
        if ( ! isset( $this->spaces ) ) {
            $finder = self::getFinder( 'woo\\domain\\Space' );
            $this->spaces = $finder->findByVenue( $this->getId() );
        }
        return $this->spaces;
    }
```

当性能问题比较显著时，可以不使用SpaceMapper，直接使用一条SQL联结一次性取回所有的数据。

当然，这会导致代码的可移植性降低，性能优化总是要付出一定代价的。

最后，Mapper类的“粒度”并非一成不变。如果对象类型被存在另外一个对象中，你可能只需要为容器对象准备一个Mapper。

数据映射器模式最强大的地方在于消除了领域层和数据库操作之间的耦合。Mapper对象在幕后运作，可以应用于各种对象关系映射。

而数据映射器模式最主要的缺点在于需要创建大量具体的映射器类。但是很多样板代码可以通过程序自动生成。利用反射是一个很好的办法。通过反射，你可以为映射器类生成干净和通用的类方法。你可以查询领域对象，找出其setter和getter方法（可能根据参数命名规则），然后自动

13

生成一个映射器基类。本章使用的映射器类正是通过这种方法生成的。

使用映射器时还要注意不要一次加载过多对象，尽管Iterator可以帮助我们。因为Collection对象只持有原始数据，第二次请求（Space对象）只有当访问特定的Venue并把数组转换为对象时才产生。这种“延迟加载”机制还可以加强，本章后面会介绍。

要小心波浪式加载（ripple loading），即当创建映射器时，如果该映射器使用了另一个映射器来获得一个对象属性，那么可能会导致一个性能上的“雪崩”。第二个映射器可能使用了更多的映射器来获得它想要的对象。如果你不小心，可能表面上看起来非常简单的一个查找操作，会导致数十个类似的操作发生。

同时也要注意数据库查询的效率，对SQL语句进行预编译和优化。如果同一条SQL语句可以应用于多种数据库那当然很好，但我们更希望SQL语句可以让查询效率达到最高。尽管通过条件语句（或者策略类）来选择不同的数据库查询语句会使代码看起来繁杂难看，但不要忘记这些优化对于客户端代码来说都是被隐藏的。

13.3　标识映射

你是否还记得PHP 4中令人苦恼的“对象按值传递”的问题？你原以为两个变量指向同一个对象，结果其实不是，而是分别指向两个很相似的对象。现在，这个问题又出现了。

13.3.1　问题

让我们看看调用数据映射器的代码：

```
$venue = new \woo\domain\Venue();
$venue->setName( "The Likey Lounge" );
$mapper->insert( $venue );
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
$venue->setName( "The Bibble Beer Likey Lounge" );
$mapper->update( $venue );
$venue = $mapper->find( $venue->getId() );
print_r( $venue );
```

这段代码说明我们添加到数据库的对象也可以通过一个映射器来得到，并且这两个对象是完全等同的。所谓“完全等同”，就是说在任何方面都一样，除了它们是不同的对象外。代码中将新的Venue对象赋值给了旧的Venue对象，把旧的对象覆盖了。遗憾的是，有时候你无法这样做。在同一个请求中，可能多次引用同一个对象。如果你修改了该对象的某个版本并保存到数据库中，如何确保该对象的另一个版本（可能已经保存到了Collection对象中）不会被你的修改所覆盖？

在一个系统中，重复的对象不只会带来风险，还可能导致系统性能降低。在一个进程中，一些常用的对象可能会被调用三四次，我们没必要每次都把对象重新保存到数据库中。

好在解决这个问题相当容易。

13.3.2　实现

一个标识映射只是一个对象，它的任务是跟踪系统中所有对象，并帮助系统避免将一个对象

看成两个对象。

实际上，标识映射本身并不主动阻止系统将一个对象看成两个，它只负责管理所有对象的信息。下面是一个简单的标识映射类：

```
namespace woo\domain;
//...

class ObjectWatcher {
    private $all = array();
    private static $instance;

    private function __construct() { }

    static function instance() {
        if ( ! self::$instance ) {
            self::$instance = new ObjectWatcher();
        }
        return self::$instance;
    }

    function globalKey( DomainObject $obj ) {
        $key = get_class( $obj )."."..$obj->getId();
        return $key;
    }

    static function add( DomainObject $obj ) {
        $inst = self::instance();
        $inst->all[$inst->globalKey( $obj )] = $obj;
    }

    static function exists( $classname, $id ) {
        $inst = self::instance();
        $key = "$classname.$id";
        if ( isset( $inst->all[$key] ) ) {
            return $inst->all[$key];
        }
        return null;
    }
}
```

图13-5展示了一个标识映射对象如何与其他类合作。

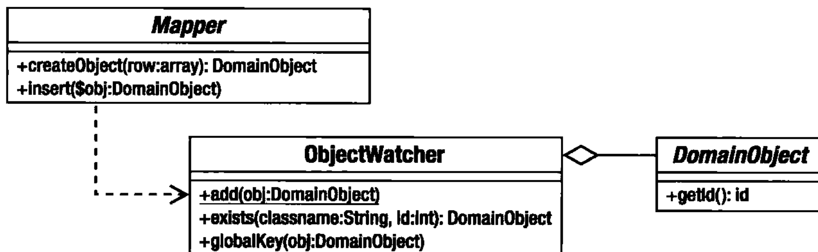


图13-5 标识映射

使用标识映射的技巧在于如何标识对象，即如何通过一种方式给对象贴上“标签”。有很多办法可以达到这个目的。但使用数据表中的键（key，系统中的所有对象都已经用过）来标识对象不是个好办法，因为在一个数据表中的ID可能和另一个数据表中的ID重复。

你也可以用数据库保存一个全局的键表。每当你创建一个对象时，就遍历该数据表，并将一个全局的键与新建的对象联系起来。这样做的性能耗费不大，而且实现起来也比较简单。

可以看到，我采用了另一种更简单的办法。我将对象的类名与数据表的ID联结起来。由于不可能存在两个ID都为4的woo\domain\Event对象，这种办法是很安全的。

globalKey()方法用来处理相关的细节。该标识映射类使用add()方法来添加新的对象。每个对象在数组属性$all中都被贴上标签。

exists()方法的参数是类名和$id（而不是对象）。我们不再需要实例化某个对象来判断它是否已经存在。该方法从这一数据中创建一个键，并检查它是否为$all属性中某个元素的索引。如果找到某个对象，则返回对该对象的引用。

我只用了一个对象作为标识映射——ObjectWatcher。该Mapper类可以生成对象，所以可以在此加入检查代码。

```
// Mapper
namespace woo\mapper;
// ...

 private function getFromMap( $id ) {
        return \woo\domain\ObjectWatcher::exists
                ( $this->targetClass(), $id );
    }

    private function addToMap( \woo\domain\DomainObject $obj ) {
        return \woo\domain\ObjectWatcher::add( $obj );
    }

    function find( $id ) {
        $old = $this->getFromMap( $id );
        if ( $old ) { return $old; }
        // 数据库操作
        return $object;
    }

    function createObject( $array ) {
        $old = $this->getFromMap( $array['id']);
        if ( $old ) { return $old; }
        // 创建对象
        $this->addToMap( $obj );
        return $obj;
    }

    function insert( \woo\domain\DomainObject $obj ) {
        // 用新id更新insert. $obj
        $this->addToMap( $obj );
    }
```

该类中有两个便捷方法：addToMap()和getFromMap()。这样我们不用去记静态调用Object-Watcher的完整语法。更重要的是，这些方法会向下调用子类（如VenueMapper等）来得到当前需要实例化的对象的类名。

获得类名可以通过调用targetClass()来实现。该方法是一个抽象方法，所有具体的映射器子类都会实现该方法。targetClass()返回映射器类生成的对象的类名。下面是SpaceMapper类实现的targetClass()方法：

```
protected function targetClass() {
    return "woo\\domain\\Space";
}
```

find()和createObject()都检查一个对象是否存在（通过传递表ID给getFromMap()）。如果对象已经存在，就会被返回给客户端代码，并结束当前方法的执行。但如果对象不存在，就会执行对象实例化。在createObject()中，新生成的对象会被传递给addToMap()，以避免将来的冲突。

那么我们为什么要执行这个过程两次呢（在find()和createObject()方法中都调用getFromMap()）？原因在于集合。当调用createObject()方法生成对象时，我们需要确认封装在一个集合对象中的记录不是陈旧的，而且要确保返回最新的对象给用户。

13.3.3 效果

如果在创建对象或添加对象到数据库时使用标识映射，系统中对象出现重复的可能性就变成了0。

当然，这只能对当前进程有效。不同进程间不可避免地会在同一时间访问同一个对象的不同版本。有时候你要考虑到并发访问可能会引起冲突导致数据损坏。如果问题很严重，可能需要采用一定的“锁定”策略。你也可以考虑将对象保存到共享内存中或者使用一个外部对象缓存系统，如Memcached。你可以在http://danga.com/memcached/了解Memcached的更多信息，也可以在http://www.php.net/memcache了解如何结合使用PHP和Memcached。

13.4 工作单元

在什么时候保存对象呢？在未使用“工作单元”（Unit of Work，见Martin Fowler的《企业应用架构模式》一书，由David Rice详细描写）模式之前，我总是从表现层发出保存命令。这实际上是一个“昂贵的”设计决定。

工作单元可使你只保存那些需要保存的对象。

13.4.1 问题

曾经有一天，我把要使用的SQL语句用echo输出到浏览器时发现了一个问题：在同一次请求中重复保存了同一条数据。我的系统是用组合的命令来搭建的，结构良好，但这意味着一条命令可能被多次调用，而且每次调用完都会执行清理工作。

我不仅保存了同一个对象两次，而且还保存了一些原本没必要保存的对象。

这个问题和标识映射中的问题有点类似。标识映射的问题是在处理过程开始时加载了不必要的对象，而我的问题则是在处理过程结束时。就像这两个问题是互补的一样，它们的解决方案也是互补的。

13.4.2　实现

为了判断哪些数据库操作是必需的，需要跟踪与对象相关的各种事件。跟踪操作最好放在被跟踪的对象中。

你也需要持有一份对象列表，找出每个数据库操作（插入、更新和删除）需要的对象。下面我们只以插入和更新操作为例。我们把对象列表放在哪里比较好呢？之前我们已经有了一个ObjectWatcher对象，现在对之进行改进：

```
// ObjectWatcher
// ...
    private $all = array();
    private $dirty = array();
    private $new = array();
    private $delete = array(); // 本例中未使用
    private static $instance;
// ...
    static function addDelete( DomainObject $obj ) {

        $self = self::instance();

        $self->delete[$self->globalKey( $obj )] = $obj;

    }

    static function addDirty( DomainObject $obj ) {
        $inst = self::instance();
        if ( ! in_array( $obj, $inst->new, true ) ) {
            $inst->dirty[$inst->globalKey( $obj )] = $obj;
        }
    }

    static function addNew( DomainObject $obj ) {
        $inst = self::instance();
        // 我们还没有ID
        $inst->new[] = $obj;
    }

    static function addClean( DomainObject $obj ) {
        $self = self::instance();
        unset( $self->delete[$self->globalKey( $obj )] );
        unset( $self->dirty[$self->globalKey( $obj )] );
        $self->new = array_filter( $self->new,
                function( $a ) use ( $obj ) { return !( $a === $obj ); }
                );
    }
```

```
    function performOperations() {
        foreach ( $this->dirty as $key=>$obj ) {
            $obj->finder()->update( $obj );
        }
        foreach ( $this->new as $key=>$obj ) {
            $obj->finder()->insert( $obj );
        }
        $this->dirty = array();
        $this->new = array();
    }
```

ObjectWatcher类仍然是一个标识映射，但增加了跟踪系统中所有对象的功能（通过$all属性）。上面的代码直接在类中添加了一些功能。

你可以在图13-6中了解到ObjectWatcher类中工作单元功能是如何工作的。

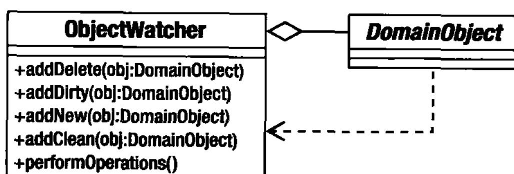


图13-6 工作单元

通常当对象从数据库中取出，然后被修改了，我们就说这个对象“脏”了。“脏”对象被保存在$dirty数组属性中（通过addDirty()方法），直到更新数据库。客户端代码可以自行决定脏对象是否触发数据库更新。把脏对象标识为“干净”的（通过addClean()方法），这样数据库就不会被更新。当然，新创建的对象会被添加到$new数组中（通过addNew()方法），该数组中的对象将会被插入到数据库中。目前我们没有实现删除记录的功能，但原理已经很清楚了。

addDirty()方法和addNew()方法都要添加一个对象到它们各自的数组属性中。addClean()方法则从$dirty数组中删除了一个对象，表示该对象不需要被更新。

当要处理这些数组中的所有对象时，performOperations()方法会被调用（可能是在控制器类中被调用，也可能是在控制器的助手类中被调用）。performOperations()方法会遍历$dirty和$new数组，更新或添加对象。

ObjectWatcher类提供了一个更新和添加对象到数据库的机制，但代码中目前并没有实现添加对象到ObjectWatcher中的功能。

由于是在这些对象上进行操作，由这些对象来执行通知是很适合的。下面是一些工具方法，我们可以将它们添加到DomainObject类，注意其中的构造方法。

```
// DomainObject
namespace woo\domain;
//...

 abstract class DomainObject {
    private $id = -1;
```

```
    function __construct( $id=null ) {
        if ( is_null( $id ) ) {
            $this->markNew();
        } else {
            $this->id = $id;
        }
    }

    function markNew() {
        ObjectWatcher::addNew( $this );
    }

    function markDeleted() {
        ObjectWatcher::addDelete( $this );
    }

    function markDirty() {
        ObjectWatcher::addDirty( $this );
    }

    function markClean() {
        ObjectWatcher::addClean( $this );
    }

    function setId( $id ) {
        $this->id = $id;
    }

    function getId( ) {
        return $this->id;
    }

    function finder() {
        return self::getFinder( get_class( $this ) );
    }

    static function getFinder( $type ) {
        return HelperFactory::getFinder( $type );
    }
    //...
```

在看工作单元的代码之前，要注意这里的Domain类存在着finder()方法和getFinder()方法。它们的工作方式与collection()及getCollection()一样，查询一个简单的工厂类——HelperFactory来获得需要的映射器对象。它们之间的关系如图13-3所示。

可以看到，如果没有传递$id参数给构造方法，构造方法会将当前对象标记为“新”对象（通过调用markNew()）。这是一种“魔术”手段，在使用的时候要谨慎。这段代码直接选定了一个新对象来插入到数据库中，而不需要对象创建器的介入。假设你的团队来了一个新成员，他写了一段临时的代码来测试领域类的行为。他看到这里没有任何持久化操作的代码，就会认为测试应该是安全的，事实是这样吗？假设这些测试对象（可能还带有临时的名称）被保存到数据库中，就会出问题。“魔术”是很好的，但更重要的是代码的清晰易懂。最好要求客户端代码传递特定

的标志给构造方法，为等待插入的对象排队。

我们还需要往Mapper类中添加一些代码：

```
// Mapper
    function createObject( $array ) {
        $old = $this->getFromMap( $array['id']);
        if ( $old ) { return $old; }
        $obj = $this->doCreateObject( $array );
        $this->addToMap( $obj );
        $obj->markClean();
        return $obj;
    }
```

创建对象的工作包括通过在构造方法中调用ObjectWatcher::addNew()将其标记为“新”，因此我们必须调用markClean()，否则从数据库中取出的每个对象在请求结束时都会被保存，这不是我们想要的。

我们要在领域模型类中调用markDirty()。记住，脏对象是从数据库读出后被修改过的对象。这是本模式中带有“鱼腥味”的一个方面。显然，确保把被各个类方法修改过的对象标记为脏对象是很重要的，但由于这个工作是手动完成的，这意味着人为的错误是不可避免的。

下面是Space对象中调用markDirty()的一些方法：

```
namespace woo\domain;

//...

class Space extends DomainObject {

//...

    function setName( $name_s ) {
        $this->name = $name_s;
        $this->markDirty();
    }

    function setVenue( Venue $venue ) {
        $this->venue = $venue;
        $this->markDirty();
    }
```

下面的代码用于添加一个新的Venue和Space到数据库（摘自Command类）：

```
    $venue = new \woo\domain\Venue( null, "The Green Trees" );
    $venue->addSpace(
        new \woo\domain\Space( null, 'The Space Upstairs' ) );
    $venue->addSpace(
        new \woo\domain\Space( null, 'The Bar Stage' ) );

    // 可能在控制器中调用，也可能在控制器助手类中调用
    \woo\domain\ObjectWatcher::instance()->performOperations();
```

我已经为ObjectWatcher添加了一些调试代码，因此你可以看到在请求结束时发生了什么：

```
inserting The Green Trees
inserting The Space Upstairs
inserting The Bar Stage
```

因为高层次的控制器对象通常会调用performOperations()，所以通常你只需要创建或修改一个对象，然后工作单元类（ObjectWatcher）只会在请求结束时执行一次。

13.4.3　效果

这个模式非常实用，但需要注意一些地方。你必须保证所有被修改的对象都被标记为“脏”，否则会导致一些难以发现和解决的bug。

你可能想了解一下测试已修改对象的其他办法。反射就是一个好的选择，但在使用时要注意程序的效率，毕竟使用模式的目的是提升效率，而不是降低效率。

13.4.4　延迟加载

延迟加载是绝大部分Web程序员自学的核心模式之一，因为它是一个用于避免过多数据库查询的极为重要的机制，是我们都需要的。

13.4.5　问题

在本章的示例中，我们已经在Venue、Space和Event对象间建立起了一定的关系。当创建一个Venue对象时，会传递SpaceCollection对象给Venue对象。如果罗列在Venue对象中的每个Space对象，Venue对象会自动发出数据库请求来获得与每个Space关联的Event。这些Event对象被保存在EventCollection对象中。如果不想查看任何事件，就没有必要去连接数据库。如果要查询多个Venue，而每个Venue都有两三个Space，每个Space又有数十上百个事件，那么这将是一个代价非常高的过程。

显然，有时候我们需要避免使用这样的自动加载功能。

下面是SpaceMapper中获得Event数据的代码：

```
protected function doCreateObject( array $array ) {
    $obj = new \woo\domain\Space( $array['id'] );
    $obj->setname( $array['name'] );
    $ven_mapper = new VenueMapper();
    $venue = $ven_mapper->find( $array['venue'] );
    $obj->setVenue( $venue );
    $event_mapper = new EventMapper();
    $event_collection = $event_mapper->findBySpaceId( $array['id'] );
    $obj->setEvents( $event_collection );
    return $obj;
}
```

doCreateObject()方法先获得与Space相关的Venue对象，这不耗什么资源，因为Venue对象通常已经保存在ObjectWatcher中。然后doCreateObject()方法调用EventMapper::findBySpaceId()方法，而这里正是可能导致系统问题的地方。

13.4.6　实现

你可能知道，“延迟加载”就是要在客户端代码真正需要的时候才去获得数据。

达到这个目的最简单的办法就是在包含对象中详细说明延迟。我们在Space对象中可能会这么写：

```
// Space
function getEvents() {
    if ( is_null($this->events) ) {
        $this->events = self::getFinder('woo\\domain\\Event')
            ->findBySpaceId( $this->getId() );
    }
    return $this->events;
}
```

这个方法检查$event属性是否已经被赋值。如果没有，则获取一个查找器（finder，这里指一个Mapper）并用其$id属性来得到与当前Space相关的EventCollection。显然，为了让这个方法避免执行不必要的数据库查询，我们也可能需要修改SpaceMapper的代码，让它不再自动预先加载EventCollection对象（在之前的例子中会自动加载）。

这个办法是可行的，但有点麻烦。有没有更好的办法呢？

首先让我们回到Iterator（用于生成Collection对象），接口下隐藏了我们的秘密（当客户端代码访问Iterator时，原始数据还未被用来实例化一个领域对象）。也许我们可以隐藏更多内容。

我们的办法是创建EventCollection对象，延迟数据库访问直到请求要求它访问数据库。这意味着一个客户端对象（例如Space）实例化时，不再需要知道它持有着一个空的Collection。一旦客户端代码需要，它就会持有一个正常的EventCollection对象。

下面是DeferredEventCollection对象：

```
namespace woo\mapper;
//...

class DeferredEventCollection extends EventCollection {
    private $stmt;
    private $valueArray;
    private $run=false;

    function __construct( Mapper $mapper, \PDOStatement $stmt_handle,
                          array $valueArray ) {
        parent::__construct( null, $mapper );
        $this->stmt = $stmt_handle;
        $this->valueArray = $valueArray;
    }

    function notifyAccess() {
        if ( ! $this->run ) {
            $this->stmt->execute( $this->valueArray );
            $this->raw = $this->stmt->fetchAll();
            $this->total = count( $this->raw );
        }
        $this->run=true;
    }
}
```

可以看到，这个类是EventCollection类的子类，其构造方法参数是EventMapper对象和PDOStatement对象，以及一个与预编译语句相匹配的数组。在第一次实例化时，这个类只是保存它的属性并静静等待，并没有查询数据库。

你可能记得Collection基类定义了一个空方法notifyAccess()（见13.2节）。从外部世界调用Collection类的方法时，notifyAccess()会被调用。

DeferredEventCollection覆盖了这个类方法。现在如果有人想访问Collection，则该类知道是时候终止伪装并获取一些真正的数据。获取数据是通过调用PDOStatement::execute()完成的。和PDOStatement::fetch()方法一起，这可以获得适合传递给Mapper::createObject()的数组。

下面是EventMapper类中实例化DeferredEventCollection对象的类方法：

```
// EventMapper
namespace woo\mapper;
// ...
function findBySpaceId( $s_id ) {
        return new DeferredEventCollection(
                    $this,
                    $this->selectBySpaceStmt, array( $s_id ) );
    }
```

13.4.7 效果

无论你是否在领域类中显式地添加延迟加载代码，延迟加载都是一个好习惯。

除了类型安全之外，使用集合对象而非数组的好处是你可以使用延迟加载。

13.5 领域对象工厂

数据映射器模式很灵活，但也有些缺陷。映射器类中要包含很多的功能，如组装SQL语句、将数组转换为对象或将对象转换为数组、添加数据到数据库。这么多功能会使一个映射器类很强大，但也会在一定程度上降低它的灵活性。当映射器需要处理多种查询或者其他类需要共享映射器类中的功能时，映射器的缺陷更为突出。下面我们将分解数据映射器，把它分成多个更为具体的模式。这些细粒度的模式可以组合起来，构成一个完整的数据映射器。这些模式在Clifton Nock著的*Data Access Patterns*（Addison Wesley, 2003）一书中有详细介绍，在此使用该书中对这些模式的命名。

让我们从一个核心的功能开始：创建领域对象。

13.5.1 问题

我们知道数据映射器是一个天然的隔离层。映射器可以在内部使用createObject()来创建领域对象，但集合（Collection）对象也需要它创建领域对象。这就要求我们在创建Collection对象时，将一个映射器的引用传递给集合。如果允许回调，这样做没有问题（正如我们在访问者和观察者模式中所做的那样），但如果能把创建对象的功能从映射器中移出来，会显得更为灵活。

这样创建对象的功能可以被Mapper和Collection类共享。

领域对象工厂（Domain Object Factory）在*Data Access Patterns*一书中有详细介绍。

13.5.2 实现

假设有多个映射器类，每一个都有它们自己的领域对象。只要将createObject()方法从映射器类中移出来放到一个独立的类中（并按平行的对象层级排行），就构成了领域对象工厂模式，如图13-7所示。

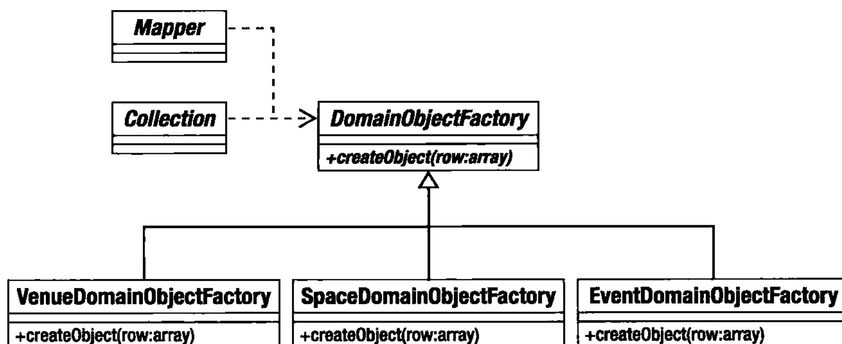


图13-7 领域对象工厂类

领域对象工厂类只有一个核心的功能[①]，因此基类的代码很简单：

```
namespace woo\mapper;
// ...

abstract class DomainObjectFactory {
    abstract function createObject( array $array );
}
```

下面是具体的实现：

```
namespace woo\mapper;
// ...
class VenueObjectFactory extends DomainObjectFactory {
    function createObject( array $array ) {
        $obj = new \woo\domain\Venue( $array['id'] );
        $obj->setname( $array['name'] );
        return $obj;
    }
}
```

当然，你也可能需要将对象缓存起来以避免重复实例化或不必要的数据库交互，就像我在Mapper类中所做的那样。你可以把映射器中的addToMap()和getFromMap()方法移到这里，或者在ObjectWatcher与createObject()间建立观察者的关系。这些细节留待读者自行处理。只要

① 创建领域对象。——译者注

记住，你要想办法避免系统中领域对象的复制。

13.5.3 效果

领域对象工厂消除了数据库原始数据与对象字段数据之间的耦合。你可以在createObject()方法中执行任意的调整。这个过程对于客户端程序员来说是透明的，只要最后能够提供数据即可。

这个功能从映射器类中移出后，就可以被其他组件使用。例如，下面是一个改写过的Collection实现：

```
namespace woo\mapper;
// ...

abstract class Collection {
    protected $dofact;
    protected $total = 0;
    protected $raw = array();

    // ...

    function __construct( array $raw=null, ➥
\woo\mapper\DomainObjectFactory $dofact=null ) {
        if ( ! is_null( $raw ) && ! is_null( $dofact ) ) {
            $this->raw = $raw;
            $this->total = count( $raw );
        }
        $this->dofact = $dofact;
    }
// ...
```

DomainObjectFactory可以即时生成对象：

```
    if ( isset( $this->raw[$num] ) ) {
        $this->objects[$num]=$this->dofact->createObject( $this->raw[$num] );
        return $this->objects[$num];
    }
```

由于领域对象工厂消除了自身与数据库间的耦合，它们可以更高效地进行测试。例如，我们可以创建一个模拟的DomainObjectFactory来测试Collection代码。这样比模拟整个映射器对象要简单得多（你可以在第18章中了解更多mock和stub对象相关的内容）。

把一个组件分解成几个部分会导致类的数量增加，同时也可能会使代码不易被理解。即使每个组件及其之间的关系都很符合逻辑，定义也很清晰，我发现有时候要从数十个名称相似的类中找出一个还是挺难的。

好事多磨。数据映射器还有另一个问题。Mapper::getCollection()方法很方便，但存在同样的问题——其他类可能需要得到一个Collection对象，但不想访问数据库。我们有两个抽象的组件：Collection和DomainObjectFactory。根据使用的领域对象，我们可能需要各种具体的子类，例如VenueCollection和VenueDomainObjectFactory，或者SpaceCollection和SpaceDomainObjectFactory。这个问题显然需要使用抽象工厂模式来解决。图13-8显示了

PersistenceFactory类。我们将使用抽象工厂模式来组织多个不同的组件。

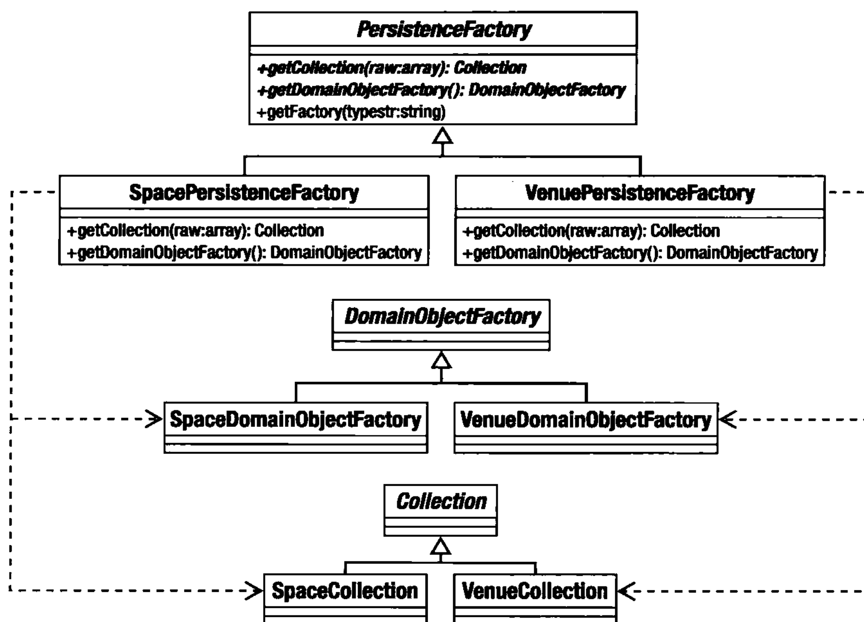


图13-8 使用抽象工厂模式来组织相关的组件

13.6 标识对象

在查找领域对象时，上面介绍的映射器实现方案有点儿不太灵活。查找一个对象没有问题，查找所有相关的领域对象也很简单。但如果要查找符合特定条件的对象，你需要添加一个类方法来构造数据库查询语句（例如EventMapper::findBySpaceId()）。

标识对象［Identity Object，Alur等人也称之为数据传输对象（Data Transfer Object）］封装了数据库查询条件，从而解除了系统与数据库语法之间的耦合。

13.6.1 问题

通常我们很难预先知道需要查询数据库中的哪些内容。领域对象越复杂，数据库查询时所使用的限定条件就可能越多。你可以根据需要往映射器类中添加多个类方法，这可以在一定程度上解决这个问题。但这个方案不怎么灵活，而且会导致代码重复，因为你在一个（或多个）映射器类中使用了很多相似但又不同的数据库查询。

标识对象封装了数据库查询中的条件，这样不同的条件组合可以在运行时构成不同的数据库查询。假设我们有一个名为Person的领域对象，客户代码可能通过调用标识对象方法来查找一个年龄在30与40之间、身高低于6英尺[①]的男人。这个Person类应该设计得尽量灵活以便条件可

① 1英尺=0.3048米。——编者注

以自由组合（因为你可能对人的身高不感兴趣了，或者想要除去年龄条件限制）。标识对象可以在一定程度上限制客户端代码的查询条件。如果你没有为将来的要求写好代码，等以后有需求时就可能需要修改查询语句。但通过不同的条件组合，我们可以让数据库查询有一定的灵活性。下面让我们看看标识对象是如何实现的。

13.6.2 实现

一般情况下，标识对象里包含着一系列类方法，你可以调用它们来构建各种查询条件。设置标识对象的状态后，你可以将它传递到一个负责构建SQL语句的类方法中。

图13-9展示了多个典型的IdentityObject类。

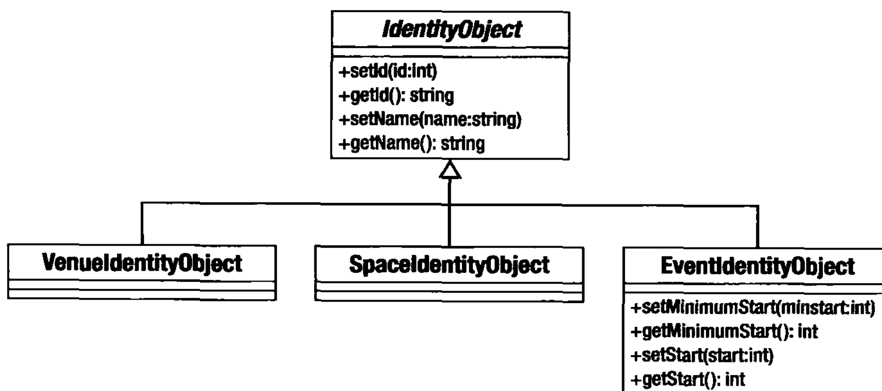


图13-9 通过标识对象来管理查询条件

你可以使用基类来管理常用的操作，并保证多个标识对象都属于同一个类型。下面的代码是一个实现方案（比图13-9中的类要简单一些）：

```
namespace woo\mapper;
//...

class IdentityObject {
    private $name = null;
    function setName( $name ) {
        $this->name=$name;
    }

    function getName() {
        return $this->name;
    }
}

class EventIdentityObject
    extends IdentityObject {
    private $start = null;
    private $minstart = null;
```

```
    function setMinimumStart( $minstart ) {
        $this->minstart = $minstart;
    }

    function getMinimumStart() {
        return $this->minstart;
    }

    function setStart( $start ) {
        $this->start = $start;
    }

    function getStart() {
        return $this->start;
    }
}
```

这非常简单。类的作用就是将数据保存起来，然后在请求的时候取出。下面的代码使用SpaceIdentityObject来构建WHERE子句：

```
$idobj = new EventIdentityObject();
$idobj->setMinimumStart( time() );
$idobj->setName( "A Fine Show" );
$comps = array();
$name = $idobj->getName();
if ( ! is_null( $name ) ) {
    $comps[] = "name = '{$name}'";
}
$minstart = $idobj->getMinimumStart();
if ( ! is_null( $minstart ) ) {
    $comps[] = "start > {$minstart}";
}

$start = $idobj->getStart();
if ( ! is_null( $start ) ) {
    $comps[] = "start = '{$start}'";
}

$clause = " WHERE " . implode( " and ", $comps );
```

这个模型可以很好地工作，但还不够符合"延迟"的精神。对于一个较大的领域对象来说，编写大量的getter和setter方法是很烦的。在这个模型中，你需要编写代码将WHERE语句中的每种条件都输出。即使在这样简单的示例中，我也懒得写getter和setter方法（里面没有setMaximumstart()方法），由此可以想象在真实世界中我多么乐于创建标识对象。

幸好有多种办法来得到数据并生成SQL。例如，我曾经在基类中给字段名关联数组赋值。它们通过比较给自己添加索引：大于、等于、小于或等于。子类提供了便捷方法来添加数据到底层的数据结构中，然后SQL构建器可以遍历数组结构来动态地创建查询。我相信实现这样一个系统

13

有很多途径，这里先看一个改进过的版本。

我将使用“流畅接口”，也就是说类的setter方法会返回类的对象实例，这样用户就可以像写句子一样连接对象。这很对我这种懒人的胃口，但我仍希望对于客户端程序来说能有一个灵活的途径来定义条件。

下面从创建woo\mapper\Field类开始，该类用来保存对比数据（用于与各字段的值对比），并最终生成一个WHERE子句：

```
namespace woo\mapper;

class Field {
    protected $name=null;
    protected $operator=null;
    protected $comps=array();
    protected $incomplete=false;

    // 设置字段名，例如age
    function __construct( $name ) {
        $this->name = $name;
    }

    // 添加操作符和值（例如 > 40）到$comps属性中
    function addTest( $operator, $value ) {
        $this->comps[] = array( 'name' => $this->name,
            'operator' => $operator, 'value' => $value );
    }

    // comps是一个数组，因此我们有多种方法来检查字段
    function getComps() { return $this->comps; }

    // 如果$comps为空，则我们有比较数据并且本字段不能用于数据库查询
    function isIncomplete() { return empty( $this->comps); }
}
```

这个简单的woo\mapper\Field类接受并保存字段名。通过addTest()方法，woo\mapper\Field类创建了operator和value元素数组。这样，我们可以对单个字段进行多个对比测试。下面是新的IdentityObject类：

```
namespace woo\mapper;

class IdentityObject {
    protected $currentfield=null;
    protected $fields = array();
    private $and=null;
    private $enforce=array();

    // 标识对象实例化时可以不带参数，也可以以字段名为参数
    function __construct( $field=null, array $enforce=null ) {
        if ( ! is_null( $enforce ) ) {
```

```
        $this->enforce = $enforce;
    }
    if ( ! is_null( $field ) ) {
        $this->field( $field );
    }
}

// 需要的字段名称
function getObjectFields() {
    return $this->enforce;
}

// 使用一个新字段
// 如果当前字段不完整，将抛出一个错误
// 例如，age而不是 age > 40
// 本方法返回当前对象的引用
// 所以我们可以使用流畅语法
function field( $fieldname ) {
    if ( ! $this->isVoid() && $this->currentfield->isIncomplete() ) {
        throw new \Exception("Incomplete field");
    }
    $this->enforceField( $fieldname );
    if ( isset( $this->fields[$fieldname] ) ) {
        $this->currentfield=$this->fields[$fieldname];
    } else {
        $this->currentfield = new Field( $fieldname );
        $this->fields[$fieldname]=$this->currentfield;
    }
    return $this;
}

// 标识对象是否已经设置了字段
function isVoid() {
    return empty( $this->fields );
}

// 传入的字段名称是否合法
function enforceField( $fieldname ) {
    if ( ! in_array( $fieldname, $this->enforce ) &&
         ! empty( $this->enforce ) ) {
        $forcelist = implode( ', ', $this->enforce );
        throw new \Exception("{$fieldname} not a legal field ($forcelist)");
    }
}

// 给当前字段添加一个相等操作符
// 例如 'age' 变成age=40
// 本方法返回当前对象的引用（通过operator()）
function eq( $value ) {
    return $this->operator( "=", $value );
```

```
    }

    // 小于
    function lt( $value ) {
        return $this->operator( "<", $value );
    }

    // 大于
    function gt( $value ) {
        return $this->operator( ">", $value );
    }


    // 操作符方法是否得到当前字段并添加了操作符和测试值
    private function operator( $symbol, $value ) {
        if ( $this->isVoid() ) {
            throw new \Exception("no object field defined");
        }
        $this->currentfield->addTest( $symbol, $value );
        return $this;
    }

    // 以关联数组形式返回目前创建的所有对比
    function getComps() {
        $ret = array();
        foreach ( $this->fields as $key => $field ) {
            $ret = array_merge( $ret, $field->getComps() );
        }
        return $ret;
    }
}
```

我们可以这样使用上面的类：

```
$idobj->field("name")->eq("The Good Show")
      ->field("start")->gt( time() )
                      ->lt( time()+(24*60*60) );
```

我们先创建一个标识对象，然后调用add()创建一个Field对象并将其赋值给$currentfield属性。注意add()方法返回了对identity对象的引用。这样我们可以在调用add()方法之后接着调用其他方法。对比方法eq()、gt()等都会调用operator()。operator()会检查当前是否有一个可用的Field对象，如果有的话，它会把操作符的符号和值传递给Field对象。同样，eq()方法也会返回对当前对象的引用，这样我们可以继续添加更多的测试或者再次调用add()来操作一个新的字段。

注意客户端代码使用的方式就像在写一个句子：field“name”equals“The Good Show”and field “start” is greater than the current time, but less than a day away（字段"name"的值等于"The Good Show"并且字段"start"的值大于当前时间，但在一天之内）。

当然，通过这种方式写代码比较简洁，但代码的安全性也下降了。这正是$enforce数组的

设计目的。子类可以使用一定的限制条件调用基类的构造方法：

```
namespace woo\mapper;

class EventIdentityObject extends IdentityObject {
    function __construct( $field=null ) {
        parent::__construct( $field,
            array('name', 'id','start','duration',  'space' ) );
    }
}
```

EventIdentityObject类现在强制限定几个字段。现在如果尝试随便访问一个字段，就会出现下面的错误：

```
PHP Fatal error:  Uncaught exception 'Exception' with message 'banana not a
legal field (name, id, start, duration, space)'...
```

13.6.3 效果

标识对象允许客户端程序员定义各种SQL查询条件，而不需要直接使用数据库查询。同时，使用标识对象也让你不用针对特定的数据库查询构建各种查询方法。

标识对象的部分目的在于对用户隐藏数据库细节。这非常重要，因此如果你创建了一个自动化的解决方案，就像例子中使用的的流畅接口那样，你使用的标签应该明确地指向你的领域对象，而不是列名。当领域对象与列名不同时，你需要提供一个使用别名的机制。

当使用特定的实体对象时，对于每个领域对象来说，使用抽象工厂是很有用的（例如前一节中介绍的PersistenceFactory），这可以为它们提供其他领域对象相关的对象。

既然我们可以表述搜索条件（search criteria），就可以利用搜索条件本身来构建查询。

13.7 选择工厂和更新工厂模式

我曾提到映射器类的职责。如果使用了合适的模式，映射器类就不需要创建对象或集合。使用标识对象来处理搜索条件后，映射器类不再需要实现多种find()方法。我们的下一个目标是将创建数据库查询语句的责任从映射器中剔除出去。

13.7.1 问题

任何系统要和数据库打交道，就要使用一定的数据库查询语句，但系统本身是由领域对象和业务规则而不是数据库组成的。本章介绍的大多数模式可以为树状的领域结构和表格式的数据库之间搭建一座桥梁。在将领域数据转换为数据库可以理解的格式时，我们需要进行解耦。

13.7.2 实现

当然，在数据映射器模式中，我们就已经见过这个功能了。但在本节中，我们可以见识到标识对象模式带来的好处。它可以更动态地生成查询语句，因为各种查询条件组合的可能性非常多。

图13-10展示了简单的选择工厂（SelectionFactory）和更新工厂（UpdateFactory）。

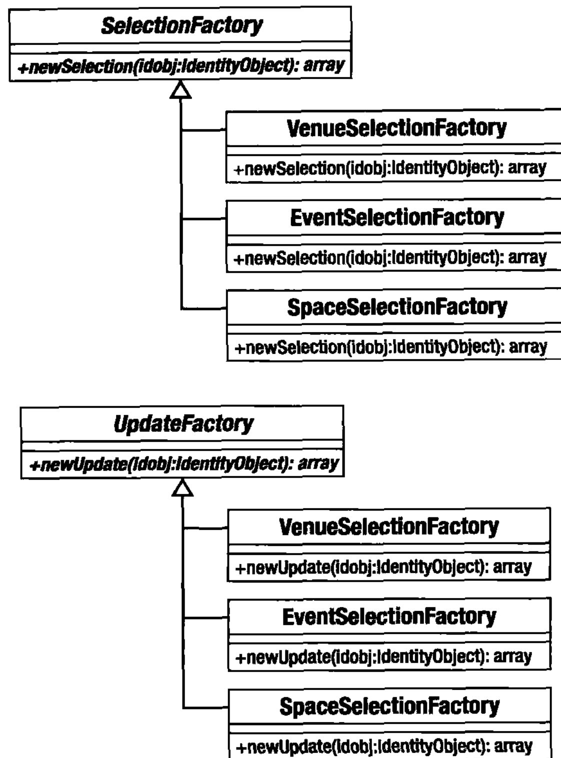


图13-10　选择工厂和更新工厂

选择工厂和更新工厂通常是与系统中的领域对象平行组织的，这可能以标识对象为媒介。鉴于此，它们也可能作为我们的`PersistenceFactory`：抽象工厂作为领域对象执行持久化操作的场所。下面是更新工厂的基类：

```
namespace woo\mapper;

abstract class UpdateFactory {

    abstract function newUpdate( \woo\domain\DomainObject $obj );

    protected function buildStatement( $table, array $fields, array $conditions=null ) {
        $terms = array();
        if ( ! is_null( $conditions ) ) {
            $query  = "UPDATE {$table} SET ";
            $query .= implode ( " = ?,", array_keys( $fields ) )." = ?";
            $terms = array_values( $fields );
            $cond = array();
            $query .= " WHERE ";
            foreach ( $conditions as $key=>$val ) {
                $cond[]="$key = ?";
                $terms[]=$val;
```

```
            }
            $query .= implode( " AND ", $cond );
        } else {
            $query  = "INSERT INTO {$table} (";
            $query .= implode( ",", array_keys($fields) );
            $query .= ") VALUES (";
            foreach ( $fields as $name => $value ) {
                $terms[]=$value;
                $qs[]='?';
            }
            $query .= implode( ",", $qs );
            $query .= ")";
        }
        return array( $query, $terms );
    }
}
```

从接口的角度看，以上这个类所做的唯一工作是定义newUpdate()方法。该方法会返回一个查询字符串数组以及要使用的值。buildStatement()方法负责执行创建update语句的通用操作，在子类中可以具体实现以满足不同标识对象的需要。buildStatement()的参数是表名、包含字段名及其值的数组以及查询条件关联数组。该方法将这3个参数组合起来以创建出一个Update语句。下面是一个具体的UpdateFactory类：

```
namespace woo\mapper;
//...

class VenueUpdateFactory extends UpdateFactory {

    function newUpdate( \woo\domain\DomainObject $obj ) {
        // 未移除类型检查
        $id = $obj->getId();
        $cond = null;
        $values['name'] = $obj->getName();
        if ( $id > -1 ) {
            $cond['id'] = $id;
        }
        return $this->buildStatement( "venue", $values, $cond );
    }
}
```

在上面的代码中，我们直接使用DomainObject。如果在系统中执行update时需要操作多个对象，可以使用标识对象来定义要使用哪些对象。这些对象可以组成一个$cond数组（在上面的代码中，$cond数组只包含id数据）。

newUpdate()提取生成查询所需的数据。在这个过程中，对象数据被转换成数据库信息。

注意newUpdate()的参数可以是任何DomainObject，所以所有的UpdateFactory都可以共享一个接口。还可以在代码中加入一些更严格的类型检查，以阻止错误类型的对象被传入。

SelectionFactory类的结构与UpdateFactory类很相似。下面是SelectionFactory的基类：

```
namespace woo\mapper;
```

13

```
//...

abstract class SelectionFactory {
    abstract function newSelection( IdentityObject $obj );

    function buildWhere( IdentityObject $obj ) {
        if ( $obj->isVoid() ) {
            return array( "", array() );
        }
        $compstrings = array();
        $values = array();
        foreach ( $obj->getComps() as $comp ) {
            $compstrings[] = "{$comp['name']} {$comp['operator']} ?";
            $values[] = $comp['value'];
        }
        $where = "WHERE " . implode( " AND ", $compstrings );
        return array( $where, $values );
    }
}
```

同样，这个类也是一个抽象类，定义了公共接口。newSelection()的参数是IdentityObject。工具方法buildWhere()的参数也是IdentityObject。buildWhere()方法通过IdentityObject::getComps()来获取所需的信息，用于创建一个WHERE子句并构建一系列值。不论是创建WHERE子句还是构建值，该方法都返回一个包含两个元素的数组。

下面是一个具体的SelectionFactory类：

```
namespace woo\mapper;

//...

class VenueSelectionFactory extends SelectionFactory {

    function newSelection( IdentityObject $obj ) {
        $fields = implode( ',', $obj->getObjectFields() );
        $core = "SELECT $fields FROM venue";
        list( $where, $values ) = $this->buildWhere( $obj );
        return array( $core." ".$where, $values );
    }
}
```

这个类创建了核心的SQL语句，然后调用buildWhere()添加条件子句。实际上，两个SelectionFactory之间的区别就在于它们创建的SQL语句中数据表的名称不同。如果没有特定的需要，也可以把这两个类合并成一个SelectionFactory。SelectionFactory类会查询PersistenceFactory指定的数据表。

13.7.3　效果

如果使用通用的标识对象，那么使用参数化的SelectionFactory类就更为简单。如果你需要硬编码的标识对象，即包含一系列getter和setter方法的标识对象，你恐怕不得不为每个领域对象实现一个SelectionFactory类。

结合使用标识对象和查询语句工厂的最大好处之一是可以生成各种各样的查询语句，但这样做会导致缓存问题。这些方法即时生成查询，重复劳动的次数不计其数。你可能很值得尝试一下，设计一种比较不同标识对象的机制，这样你可以从缓存中方便地获得需要的字符串。同时你也可以考虑从更高的层次使用数据库语句池[①]。

组合使用这些模式的另一个问题就是，这些模式是灵活的，但并不足够灵活。我的意思是它们被设计为适用于各种限制条件，而没有设计任何例外情况。映射器类的创建和维护有点麻烦，但在其干净的API后面，却是最适合进行性能调节和数据处理的地方。这些看起来优雅的模式都致力于它们自己的职责，并强调通过组合来使用，这样就很难走捷径实现一些不优雅但强大的功能。

幸好，我们还有较高层次的接口——还有一个控制器层来处理这些。

13.8 数据映射器中剩下些什么

现在，我们已经从数据映射器中去除了创建对象、查询和数据集合的功能，不再需要管理任何数据库查询条件。那么现在数据映射器中还剩下些什么功能呢？之前我们创建了很多对象，并让它们进行交互。我们需要在这些对象之上添加一个对象，用于缓存和处理数据库连接（尽管数据库连接的工作也可以被委托给其他的对象）。Clifton Nock把这些数据层控制器称为领域对象组装器。

下面是一个例子：

```
namespace woo\mapper;

//...

class DomainObjectAssembler {
    protected static $PDO;

    function __construct( PersistenceFactory $factory ) {
        $this->factory = $factory;
        if ( ! isset(self::$PDO) ) {
            $dsn = \woo\base\ApplicationRegistry::getDSN( );
            if ( is_null( $dsn ) ) {
                throw new \woo\base\AppException( "No DSN" );
            }
            self::$PDO = new \PDO( $dsn );
            self::$PDO->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
        }
    }

    function getStatement( $str ) {
        if ( ! isset( $this->statements[$str] ) ) {
            $this->statements[$str] = self::$PDO->prepare( $str );
        }
        return $this->statements[$str];
```

① database statement pooling，把整条语句缓存起来，而不是只缓存语句中的字段值。——译者注

```
    }

    function findOne( IdentityObject $idobj ) {
        $collection = $this->find( $idobj );
        return $collection->next();
    }

    function find( IdentityObject $idobj ) {
        $selfact = $this->factory->getSelectionFactory(  );
        list ( $selection, $values ) = $selfact->newSelection( $idobj );
        $stmt = $this->getStatement( $selection );
        $stmt->execute( $values );
        $raw = $stmt->fetchAll();
        return $this->factory->getCollection( $raw );
    }

    function insert( \woo\domain\DomainObject $obj ) {
        $upfact = $this->factory->getUpdateFactory(  );
        list( $update, $values ) = $upfact->newUpdate( $obj );
        $stmt = $this->getStatement( $update );
        $stmt->execute( $values );
        if ( $obj->getId() < 0 ) {
            $obj->setId( self::$PDO->lastInsertId()  );
        }
        $obj->markClean();
    }
}
```

可以看到，这个类不是一个抽象类。它没有将自己分成多个特定的子类，而是使用PersistenceFactory来确保自己为当前领域对象得到正确的组件。

图13-11展示了我们创建的较高层次的参与者（即我们从映射器中抽出的部分）。

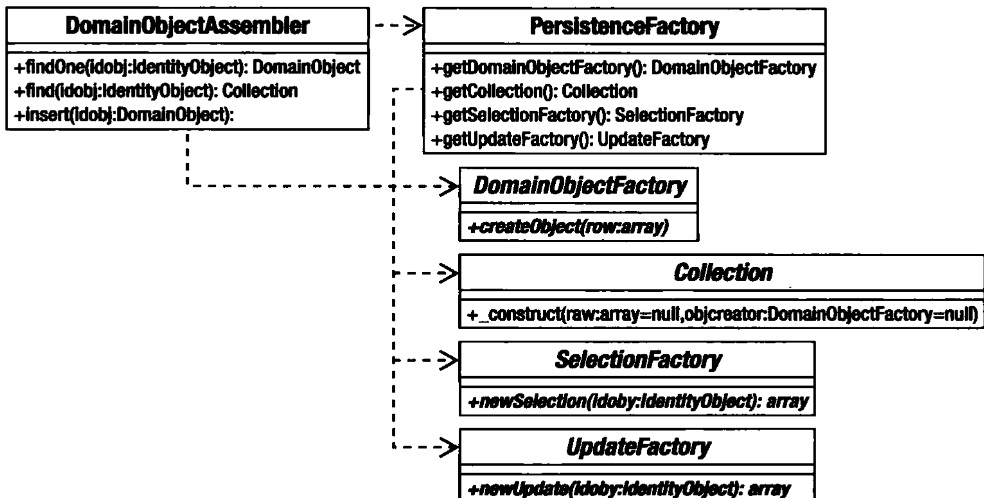


图13-11　本章使用的一些持久化类

PersistenceFactory类除了负责连接数据库和执行查询外，还负责管理Selection-Factory对象和UpdateFactory对象。当执行SELECT查询时，它也使用Collection类，或者直接使用DomainObjectFactory来生成返回值。

从客户端代码的角度看，获取DomainObjectFactory是很容易的，只要得到正确的Persis-tenceFactory对象即可：

```
$factory = \woo\mapper\PersistenceFactory::getFactory("woo\\domain\\Venue" );
$finder = new \woo\mapper\DomainObjectAssembler( $factory );
```

但如果在PersistenceFactory中直接添加getFinder()，就可以更容易达到目的。上面的代码可以改写成：

```
$finder = \woo\mapper\PersistenceFactory::getFinder( 'woo\\domain\\Venue' );
```

其他内容由读者自行实现。

客户端程序员可以继续得到一组Venue对象：

```
$idobj = $factory->getIdentityObject()->field('name')
        ->eq('The Eyeball Inn');
$collection = $finder->find( $idobj );


foreach( $collection as $venue ) {
    print $venue->getName()."\n";
}
```

13.9 小结

像平常一样，选择使用哪些模式总是由你面临的问题决定。我在实际开发中偏好于使用数据映射器和标识对象。我喜欢灵活的自动解决方案，但也需要确定自己能够分解整个系统并在必要的时候手动完成一些工作，如维护一个干净的接口和分离的数据库层。例如，我可能需要优化一条SQL查询语句，或使用联结从多个表中得到数据。即使你使用的是一个基于复杂模式的第三方开发框架，也会发现ORM（Object-Relational Mapping，对象关系映射）并不总能够满足你的要求。优秀的框架或系统的一个特点是：在不破坏原有完整性的前提下，可以轻松地加入新功能。我喜欢优雅、漂亮的解决方案，但我也是一个现实主义者！

同样，本章介绍了很多内容，主要介绍了以下模式。

- 数据映射器（Data Mapper）：创建用于将领域模型对象映射到关系型数据库的特定类。
- 标识映射（Identity Map）：跟踪系统中的所有对象，以避免重复实例化或不必要的数据库操作。
- 工作单元（Unit of Work）：自动保存对象到数据库，确保只将修改过的对象和新创建的对象插入数据库。
- 延迟加载（Lazy Load）：延迟创建对象或数据库查询，直到确实需要。

- 领域对象工厂（Domain Object Factory）：封装创建对象的功能。
- 标识对象（Identity Object）：允许客户端程序员自行组装数据库查询条件，而与底层数据库无关。
- 查询语句工厂（Query Factory）：包括选择工厂和更新工厂（Selection and Update）封装创建SQL查询的逻辑。
- 领域对象组装器（Domain Object Assembler）：创建一个在较高层次管理数据存取的控制器。

在下一章中，我们可以从大量的代码中解脱出来，介绍更多可以促使项目成功的实践经验。

Part 4 第四部分

实　践

本部分内容

- 第 14 章　良好和糟糕的实践
- 第 15 章　PEAR 和 Pyrus
- 第 16 章　用 phpDocumentor 生成文档
- 第 17 章　使用 Subversion 进行版本控制
- 第 18 章　使用 PHPUnit 进行测试
- 第 19 章　用 Phing 实现项目的自动构建

第14章 良好和糟糕的实践

到目前为止，我们已经讨论过编码的方方面面，其中特别关注了构建灵活、可重用的工具和应用的设计。但编写代码并不是开发的全部。你很可能已经通过图书和课程牢固地掌握了一门编程语言，但在运行和部署项目时仍然会遇到问题。

在本章中，我们将超越代码，介绍一些有助于成功开发项目的工具和技术。本章包括以下内容。

- 第三方包：如何获得它们及何时使用它们。
- 构建：创建和部署包。
- 版本控制：在开发过程中保持一致。
- 文档：编写易于理解、使用和扩展的代码。
- 单元测试：自动检测和预防bug的工具。
- 持续集成：利用这个实践和工具集来自动化项目构建和测试，并且在出现问题时得到警告。

14.1 超越代码

当我第一次从一个人独立开发转向团队开发时，我惊奇地发现，团队中其他成员似乎知道很多东西。在开发中我们总要面对这样的争论：最好的文本编辑器是哪个？整个团队应该统一使用一个IDE吗？我们要强制规定一个编码标准吗？如何测试我们的代码？开发的时候要写文档吗？有时候这些问题比编码本身还重要。通过日常工作的积累，我的同事们看起来已经对这些问题的答案都了然于胸。

我所读过的PHP、Perl和Java图书主要是介绍如何编写代码的。我已经说过，大多数编程图书很少把注意力从函数和语法上转移到代码设计上。如果连设计都不在讨论范围之内，那么其他更广泛的话题（比如版本控制和测试）就更不在讨论之中了。这并不是批评其他书——如果一本书涵盖了一门语言的主要特性，那么它恐怕没有足够的篇幅来介绍其他内容。

但在学习写代码时，我发现自己忽略了项目日常开发的很多技巧。我发现其中有些细节决定了项目开发的成败。在本章和接下来的几章中，我们将超越代码详细讨论一些决定项目成败的工具和技术。

14.2 借一个轮子

当我们在项目开发时面对一个有挑战性的、特定的需求时（例如需要解析某种特定格式，或者需要某种特定的通信协议与远程服务器交互），创建一个专用的组件是个好主意。这也是提高编程水平的一个好方法。在创建包时，你能够深入认识到问题所在，还能把今后可以广泛使用的新技术储备起来。这样也能增长自己的开发经验和技巧。在系统内部实现所有功能，用户就可以不用下载第三方软件包。不使用第三方软件包偶尔也可以避免出现一些版权许可上的问题。如果你自己设计了一个组件，其工作情况良好并严格按照预期运行，这是多么美妙的事啊，一定让你很有成就感！

当然，这样做也有缺点。很多软件包需要投入数千小时的人工来开发，而你可能没有这么多时间。你可能只开发可以满足自己项目特别需要的功能来解决这一问题，而第三方工具则可以满足项目的其他需要。于是就有了这么一个问题：如果存在一个可以自由使用的工具，为什么还要浪费自己的精力去重新打造一遍呢？你有充足的时间和资源去开发、测试和调试软件包吗？使用别人的软件包是否是更好的选择？

我最反对“重新发明轮子”。找出问题并设计解决方案，这是我们作为程序员每天工作的核心内容。设计一个良好的架构，比写一些“胶水”式的代码来把三四个组件“粘”在一起更有意义。每当我想从头开发某些功能时，我就会提醒自己想想以前完成的项目。从头开发虽然没有让我的项目失败，但却很花费时间，让我投入过多。我往往会忙于处理组件间的关系和类图，沉浸于那些组件的细节之中，而离项目的宏观规划越来越远。

现在当我规划项目时，如果某个功能是系统特有的，我才会尝试自己开发。例如，你的程序可能要生成或读取RSS源、验证E-mail地址、自动发送广告、验证用户，或者从一个标准格式的配置文件中读取内容，而这些功能均可由外部软件包提供。

确定了系统需求之后，你应该首先访问PEAR的官方站点（http://pear.php.net）。PEAR（PHP Extension and Application Repository，PHP扩展与应用库）是由PHP官方维护并经过严格质量控制的软件库。它也是一种用于无缝安装和管理软件包间依赖关系的机制。我会在下一章中详细介绍PEAR，包括如何使用PEAR的功能来定制软件包。为了简要说明PEAR库的强大功能，下面列出了几个可以使用PEAR包实现的功能：

- 使用Cache_Lite缓存输出；
- 使用Benchmark测试代码的执行速度；
- 使用MDB2抽象数据库访问细节；
- 使用File_HtAccess操作Apache的.htaccess文件；
- 使用XML_RSS来解析或加密RSS新闻源；
- 使用Mail_Mime来发送邮件附件；
- 使用Config解析配置文件格式；
- 使用Auth进行身份验证。

PEAR网站把软件包按主题分类。你会发现软件包非常齐全，可以满足很多项目需求。如果

PEAR上没有你想要的软件包，也可以通过搜索引擎来查找。不管如何，你应该花点时间看看是否有现成的软件包可以使用，避免“重复发明轮子”。

确定了需求并且找到了一个可以使用的软件包，这并非说明你就可以使用这个软件包了。利用软件包可以使你避免进行不必要的开发，但有时却会增加系统开销，不能带来实际好处。例如客户可能需要程序能够发送电子邮件，这并不意味着你就应该直接使用PEAR的Mail包。PHP本身就提供了一个极好用的`mail()`函数，因此你首先应当看一下`mail()`函数是否能够满足要求。当你发现需要根据RFC822标准来验证所有电子邮件地址或者需要发送图片作为邮件附件时，才应该考虑使用PEAR的Mail包。

许多程序员（包括我自己），常常花费过多时间从头开发功能，这样有时候会影响项目的完成。过于强调原创性或许是很多重复代码涌现的原因。

高效率的程序员只把原创代码看做构建项目的一个工具。这些程序员常常会利用手边的资源，并将它们有效地部署到项目中。如果有一个现成的软件包可以使用，那就是一个好选择。让我们借用一下Perl程序员们的名言：优秀的程序员都是懒惰的（good coders are lazy）。

14.3　合作愉快

萨特的名言“他人即地狱”（Hell is other people[①]）在现实项目中屡屡被证明是正确的。客户与开发者之间缺乏沟通，常会导致产品功能缺陷或者开发优先级冲突等。但这样的冲突同样会产生于有沟通和合作良好的团队成员之间，特别是当他们要共享代码的时候。

一旦一个项目中的开发者不止一个人时，版本控制就成了问题。一个程序员可能会在适合的目录下工作，并在开发过程中不时地把工作目录复制一份保存起来。但新的程序员加入开发时，这样的策略很快就被破坏。如果新程序员也在同一个开发目录下工作，那么两个程序员就可能会覆盖对方的代码，除非他们非常小心，并总是各自修改不同的文件。

或者，两个程序员可以各自工作于不同版本的代码。这样可以互不干扰，直到两个版本要合并。但除非程序员们完全在不同的文件上工作，否则合并代码将会令人非常头疼。

这就是我们要使用Subversion或类似工具的原因。使用Subversion，可以从代码基库中导出自己的版本代码并在其上工作，直到完成工作。然后你可以更新代码，得到你的同事所做的任何改变。Subversion会自动把这些改变合并到你的文件中，并在出现它无法处理的冲突时提醒你。一旦你测试这个混合文件无误，就可以把文件提交到中心版本库（central Subversion repository）中，供别的程序员使用。

Subversion还有其他好处。它记录下了一个项目的所有变更记录，你可以回滚或者读取到项目任意一个阶段的状态。你也可以创建分支，这样可以维护一个公开的发布版本，同时又有一个用于内部开发的版本。

一旦在项目开发时使用了Subversion，你就一定会喜欢上它。刚开始使用Subversion的时候，同时在项目的多个分支上工作可能在概念上会有所不适应，但你很快就会发现使用Subversion的

① 出自萨特的哲理剧《禁闭》。——译者注

好处。版本控制实在是太好用了。本书将在第17章中介绍Subversion。

14.4 为你的代码插上双翼

你是否感觉到有些程序很难安装和部署？对于在特定环境上开发的项目来说尤其如此。这些项目已经和环境绑定在一起，包括密码、目录、数据库和辅助程序调用等都被写进了代码。部署这样的项目很麻烦，开发团队不得不检查源代码，修改其中的配置来使项目符合新的运行环境。

把配置信息集中到一个配置文件或者类中，可以缓解上面的问题。这样配置可以在一个地方进行修改。但即使这样，安装仍然很麻烦。程序安装的难易程度会影响到项目的流行程度，同时也会影响到开发者在开发期间频繁部署程序和部署多个程序的难度。

如同那些重复性的工作和较费时间的工作，安装过程应该能够自动完成。一个程序的安装器应该能够决定安装位置、检查和改变权限，创建数据库并初始化变量等。事实上，一个安装器应该能够完成将程序从源代码库读出来并部署到发布环境的整个工作。

当然，自动安装并不意味着不需要向代码中添加环境信息，但它可以使安装变得容易，只需回答几个问题或者提供一两行命令行代码即可。

对于开发者来说，安装器也很有好处。一旦安装器从某个分发目录开始安装，运行一次后，它就可以缓存配置信息，使后面的安装更加容易。当你再次从该目录开始安装时，你将不需要提供数据库名和安装目录等这些配置信息（它们已经被第一次安装记录下来了），于是安装就变得简单。对于需要通过版本控制频繁更新本地开发环境代码的开发者来说，这样的功能非常重要。版本控制使获得最新的代码变得简单，但在部署代码的时候有时会遇到一些困难。

对于开发者来说，有多种创建工具。例如PEAR就是一种安装软件的解决方案。大多数情况下，你使用PEAR从其官方站点上下载代码。但是，你也可以创建自己的PEAR包，并提供给你的用户下载和安装。PEAR安装器非常适合于自我封装的、功能集中的包。它对于包里文件的角色和安装位置有较为严格的要求，并且主要负责在安装过程中把文件A安装到目录B这样的操作。本书将在第15章中详细介绍PEAR。

如果需要在程序安装时拥有更高的灵活性，你也许需要一个更灵活并且可以扩展的安装器。在第19章中，我们将学习一个叫做Phing的工具。这个开源工具是Ant（Java的创建工具）的PHP移植版本。Phing是用PHP写的，也用于PHP项目，但它的结构和用法都和Ant相似，并且使用相同XML格式的创建文件。

PEAR在项目配置较少的时候非常好用，而Phing则更适合于需要更高灵活性的项目。可能一开始你会觉得麻烦，但以后就会体会到Phing的好处。你不仅可以通过Phing来自动复制文件和转换XSLT等，你也可以方便地编写自己定制的任务以扩展它。Phing是用PHP 5编写的，充分利用了PHP 5的面向对象特性，其功能侧重模块化、易于扩展。

14.5 文档

我的代码写得很清晰优雅，不需要文档。哪怕只是看一眼，就能理解其作用。我想也许你的代码也会是这样，但对于其他人来说则并非如此。

其实从某种程度上说，优秀的代码可以为其自身提供文档功能。通过定义一个清晰的接口，或为每个类和方法定义好职责，并为它们起个描述性的名称，别人就可以从代码中了解很多内容。但你还可以进一步改善代码，使之更为清晰，易于理解：代码的易读性比编程的技巧更为重要，除非技巧能带来很大的或急需的性能上的提升。

属性、变量和参数的命名对于增强代码的可读性非常重要。请尽可能选择描述性的名字。我通常会在变量命名中增加变量数据类型的信息——特别在给参数变量命名时。

```
public function setName( $name_str, $age_int ) {
    //...
}
```

但是，无论你的代码是多么清晰易读，光有代码还是不够的。我们之前介绍过，面向对象设计经常需要组合多个类来构成继承、聚合等关系。当你在这样的结构中看到单个类时，如果没有一些明显的提示的话，很难想象出整个结构是什么样的。

同时，每个程序员都知道编写文档是很痛苦的事。你会倾向于在开发时忽略文档，因为代码总是不停地变化，而你的项目总在不断尝试改进代码。于是当项目达到某个相对稳定的阶段，你会突然发现这时为代码添加文档已经是个繁重的工作。之前谁知道会创建这么多类和方法呢？现在项目开发的截止日期已经逼近，因此我们不得不减少花在写文档上的时间，而把时间花在保证质量上。

这样的态度可以理解，但是目光过于短浅。如果一年后你要再次查看代码，结果会如何呢？下面是http://www.bash.org收集的一个程序员的聊天记录（通过IRC聊天服务发布的）：

<@Logan>：我很偶然地花了一分钟看我写过的代码。

<@Logan>：我在想“这家伙到底想干什么”。

没有文档，你注定要重演以上的故事：浪费时间去猜测你以前写代码时的意图和理由。这种情况非常糟糕，而且当你要把代码交给同事时，事态会更严重，代价也更大。缺乏文档的代码会浪费你的工作时间，因为你的同事不得不花时间来调试你的代码，或者理解其中杂乱相关的类。

很显然我们应该在编写代码的同时编写文档，但写文档的过程是否可以更为合理和方便？正如你心中所想，答案是肯定的。这个解决方案是也从Java软件中借鉴而得到的。phpDocumentor（http://www.phpdoc.org/）是JavaDoc（Java编写的文档工具，随Java SDK一起发布）的一个PHP实现版。从一个程序员的角度看，使用phpDocumentor是很简单的。你只需添加特定格式的代码注释到所有类、大多数方法和一些属性前即可。phpDocumentor会自动解析这些注释并生成带有超链接的网页文档。即使你没有提供这些注释，phpDocumentor也会阅读代码并为找到的类作出总结，或把多个类链接在一起。这是phpDocumentor的一大好处。你可以通过点单击链接从一个类跳到另一个类，并且可以很方便地观察继承关系。

本书将在第16章中介绍phpDocumentor。

14.6　测试

当你创建类时，可能非常确定它可以很好地工作。你会在开发时把它放到合适的位置，也会

在整个系统中测试它是否集成成功，新功能是否可用并且运行良好。

但你能够保证这个类一直运行良好吗？这个问题似乎有点奇怪。毕竟你已经检查过这个类，它怎么会突然就不能工作了呢？是的，如果你不再往系统中增加代码，它会一直正常运行。但是，项目是不断演变的，组件周围的环境可能发生变化，而且组件本身也可能发生变化。

让我们一一来考虑这些问题。首先，改变组件周围的环境是否会导致出错？假设某个组件很独立，与周围环境之间耦合很小，但其中的类会使用到一些对象。这些对象会返回特定的值，执行操作和接受数据等。如果这些行为中的任意一个改变了，类的操作可能出错——有些明显的错误比较容易捕捉，你可以得知出错文件的名称和出错的代码所在的行号。但更多隐蔽的错误并非发生于编程语言的引擎级别上，它并不直接报错，但会让你的组件感到困惑，不知该如何处理[①]。如果一个类依赖于另一个类的数据，那么当后者的数据改变时，可能会导致使用数据的类作出错误的决定。这时代码并未改变，但程序出错了。

编写完一个类后，你也可能需要改动类中的代码，通常只是很小的改动。因为改动很小且显而易见，所以你可能会觉得没有必要仔细检查改动的后果。后来你可能已经忘了这些改动的地方，除非你在修改代码时作了记录（例如可能在类文件的末尾用注释写上）。但是这些微小的改变很有可能引起严重的后果。如果你在适合的地方部署测试用具（test harness），这些问题可能会被你捕捉到。

测试用具是一系列自动化测试的集合，它可以应用于你的整个系统，也可以应用于单个类。经过良好部署后，测试用具可以帮助你阻止bug的产生及重复产生。在开发时，在一个地方的代码改变可能会引起一系列错误，测试用具可以帮助定位这些错误的位置并消灭它们。这意味着你可以在修改代码时更为自信，不用担心破坏系统的功能。在改进系统功能时，看到一系列未通过的测试是很有成就感的。因为这些原本可能传播到系统中的错误，现在被你捕捉到了。

14.7 持续集成

你曾经做过使一切都井井有条的计划吗？首先是分配：可能是代码或学校项目。分配工作非常庞杂可怕，而且很容易出错。但是你拿出一张纸，并把大任务分成多个可管理的小任务，就没有那么难了。至于读什么书、写什么组件都由你决定。或许你会用不同的颜色来标识任务。单独来说，任何一个任务都没那么可怕，事实也证明的确如此。只要按计划进行，就可以按时完成项目。只要你每天都做一点，就不会有什么问题。放松点！

有时，计划能起到护身符的作用。它就像一面盾牌，使你免遭质疑，使你摆脱对可能颜面尽失的恐惧感。几周之后，你将意识到计划并没有那么神奇——实际的工作还得由你来做。当然到那时为止，在计划的“宽慰”之下，你一直在得过且过、顺其自然。除了重做计划别无他法。这次你就不会认为计划有那么大魔力了。

测试和构建也是如此。你必须运行测试。你必须构建项目，并定期在新环境中构建，否则“魔力”就会消失。

① 例如没有语法上的错误，不报错，但可能逻辑不对。——译者注

如果编写测试对你来说很痛苦的话，那么运行测试可能非常烦琐，会令人头疼，尤其是当测试已变得非常复杂，测试失败中断计划时。当然，如果你经常运行测试，失败的可能性就会小一些，并且实际出现的失败会为你实现头脑中的新代码提供难得的思路。

沙盒使用起来很不错，毕竟所需的所有工具都在那儿：小的脚本程序（能使你的工作变得轻松）、开发工具和有用的库。但问题在于，你的项目也会变得过于依赖沙盒。项目开始依赖未提交的代码，或者是你已经忽略的生成文件的依赖关系。这就意味着，除了你工作的位置，项目可能已经被破坏了。

唯一的解决方案就是构建、构建再构建，并且每次都在新环境中进行。

当然，这条建议非常好，但说起来容易做起来难。程序员往往都喜欢编写代码，他们希望会议和家务活越少越好。于是，CI（Continuous Integration，持续集成）出现了。CI既是一种实践也是一套能尽可能简化该实践的工具。理想情况下，构建和测试应该完全自动进行，至少可以通过单个命令或单击操作来启动。可以跟踪所有问题，而且你能在问题变得过于严重之前得到通知。第20章会详细介绍CI。

14.8 小结

开发者的目标总是开发一个能够使用的系统。编写高质量的代码是实现这个目标的基础，但还不够。

在本章中，我们介绍了PEAR（它也是下一章的主题），讨论了团队协作的两个辅助手段——文档和版本控制，了解了版本控制需要使用自动化的创建工具，于是介绍了Ant（一个Java构建工具）的PHP实现Phing，最后讨论了软件测试和CI（自动化构建和测试的工具）。

第 15 章

PEAR和Pyrus

程序员们总是希望生成可复用的代码，而这正是面向对象开发的主要目标之一。我们希望把有用的功能从特定的开发背景下抽象出来，将其变成一个可以多次重复使用的工具。换个角度讲，如果程序员喜欢重用，他们就会讨厌复制。通过创建可以重用的代码库，可以避免在多个项目中实现类似的解决方案。

但即使我们尽量避免重复自己的代码，还是存在一个更大的问题。想想看，你创建的每个工具曾经有多少程序员都已经实现过？这是大规模的劳动力浪费。难道程序员不能团结合作，集中精力把某个工具做得更好，而不是各自作战创造出具有同样功能的数百个版本？这正是PEAR（PHP Extension and Application Repository，PHP扩展与应用程序库）诞生的原因。

PEAR是PHP代码类库（通常以包为单位）的集合，这些代码都经过了严格的质量控制。它也是一个客户端-服务器端机制，用于部署、安装软件包和管理软件包间的依赖关系。

本章包括以下内容。

- PEAR基础：这个奇怪的水果是什么。
- 安装PEAR包：只需一个简单的命令。
- 使用Pyrus，它是PEAR的“胞弟”。
- 在项目中添加PEAR包：一个示例及一些关于错误处理的注解。
- package.xml：剖析生成文件。
- 创建自己的频道：为用户提供透明的依赖管理和包下载功能。

15.1 什么是 PEAR

PEAR的核心部分是许多“包”的集合。PEAR可以按功能分成不同的类别，例如网络、邮件和XML。PEAR库是集中管理的，因此如果使用官方发布的PEAR包，通常该软件包的代码质量是值得信任的。

你可以在http://pear.php.net上找到所有可用的包。在为项目添加某个功能之前，应该先到PEAR站点上查看一下是否有现成的软件包，这是一个很好的习惯。

PEAR是随PHP绑定发布的（至少本书写作的时候是这样），也就是说PHP自带了PEAR的核心类库（除非你在编译PHP时使用了`-without-pear`参数）。PEAR软件包会被安装到可配置的路

径下（在Linux或Unix系统下通常是/usr/local/lib/php）。通常，可以通过pear命令来检查安装路径：

```
$ pear config-get php_dir
/usr/local/lib/php
```

PEAR的核心包（PEAR基础类，PEAR Foundation Classes）提供了整个类库的骨架——包括错误处理和对命令行参数的处理等核心功能。

注解　如果使用Unix发布版安装PHP，可以首先进行最低限度的安装。例如，要想在Fedora 12上安装PHP和PEAR，需要使用如下命令：

```
sudo yum install php
sudo yum install php-pear
```

如果希望使用你的发布版的包管理工具来管理PHP，可以参考该发布版的文档。上面我们已经看到了PEAR命令行工具的使用，pear命令可用来与PEAR的各个方面交互，因此它是PEAR重要的一部分。pear命令支持很多子命令。例如，我们用过config-get这个子命令，它用来显示特定配置设置的值。可以通过config-show子命令来查看所有配置及其值：

```
$ pear config-show
```

```
$ pear config-show
Configuration (channel pear.php.net):
=====================================
Auto-discover new Channels     auto_discover    <not set>
Default Channel                default_channel  pear.php.net
HTTP Proxy Server Address      http_proxy       <not set>
PEAR server [DEPRECATED]       master_server    pear.php.net
Default Channel Mirror         preferred_mirror pear.php.net
Remote Configuration File      remote_config    <not set>
PEAR executables directory     bin_dir          /usr/bin
...
```

15.2　了解 Pyrus

在进一步学习之前，我将介绍PEAR世界中的一个新成员，Pyrus。Pyrus是下一代PEAR应用。这个名字来自乔木和灌木属，这一属包括梨树。顺便说一下，拉丁语的Pirus表示“梨树”。你可以在http://pear2.php.net上下载Pyrus。你要下载的是phar包（PHP Archive）。如果你熟悉Java，可以把phar包看成PHP中的jar文件。phar实际上是压缩成zip文件的代码包，用这种方式可以很灵活地传递库代码。下载之后，你可像下面这样简单地运行Pyrus：

```
php pyrus.phar
```

```
Pyrus version 2.0.0a1 SHA-1: 27EB8EB427EA50C05691185B41BBA0F0666058D0
Pyrus: No user configuration file detected
It appears you have not used Pyrus before, welcome! Initialize install?
...
```

可以看到，第一次运行时Pyrus会提供配置选项。你需要告诉它在哪保存配置文件，在哪保存包。你还可以决定开发过程中使用PEAR还是Pyrus。它们都将安装PEAR包，但它们的安装位置不能完全兼容。

在撰写本书时，Pyrus使用的是PHP 5.3.1，这个PHP版本是很新的。这意味着你可能会不时遭遇使用问题（例如，当前版本可能无法卸载包）。在一段时间里，你可能还是要使用PHP 5.2。如果确实如此，就需要使用PEAR。与PEAR配合使用的第三方工具可能还不适用于Pyrus，但Pyrus是趋势。从长远角度来看，移植来得更实惠。

因为有充分的理由相信两种方式都是很好的选择，所以本章将介绍与PEAR和Pyrus都相关的内容。

先给你报告一个最新消息：Pyrus不支持`config-get`命令，但是它支持`config-show`。

`$ php pyrus.phar config-show`

```
Pyrus version 2.0.0a1 SHA-1: 27EB8EB427EA50C05691185B41BBA0F0666058D0
Using PEAR installation found at /usr/share/pear2
System paths:
  php_dir => /usr/share/pear2/php
  ext_dir => /usr/lib/php/modules
...
```

注解 虽然使用标准的PHP构建，Pyrus能开盒即用，但我再次强调，Linux发布版安装的PHP可能并没有提供你所需的一切。Pyrus需要phar、simplexml、libxml2、spl和pcre扩展。为了运行Pyrus，Fedora 12需要再次执行如下命令：

```
yum install php-xml
```

你最喜欢的发布版也可能有它自己的安装问题。记住，你需要PHP 5.3.1及更新的版本。

虽然PEAR和Pyrus都支持很多子命令，但`Install`是最常用的。`Install`用于安装PEAR包。

15.3 安装 PEAR 包

选定包之后，只需输入一行命令就可以下载并安装这个包。下面的命令用于安装名为Log的PEAR包（Log是一个用于记录错误日志的代码包）：

```
$ pear install pear/Log
```

注解 在大多数情况下，需要`root`权限来安装、升级和删除PEAR的包，因为它需要在home空间之外执行写入操作。如果没有这些空间的写权限，也没有关系。15.5节将介绍如何改变默认的安装路径，以便把包安装到自己可写的空间。

安装是非常简单的。PEAR安装程序和PHP是绑定发布的，它会自动定位Log代码包的位置并下载和安装它。下面是执行命令后的输出结果：

```
Did not download optional dependencies: pear/DB, pear/MDB2,
      use --alldeps to download automatically
pear/Log can optionally use package "pear/DB" (version >= 1.3)
pear/Log can optionally use package "pear/MDB2" (version >= 2.0.0RC1)
downloading Log-1.12.0.tgz ...
Starting to download Log-1.12.0.tgz (38,479 bytes)
.........done: 44,555 bytes
install ok: channel://pear.php.net/Log-1.9.11
```

可以看到，Log包还有一些可选的依赖包，可以不用理睬它们，除非你需要那些包的特有功能。注意最后一行，PEAR告诉我们它从PEAR频道pear.php.net获取Log包。实际上，我们安装pear/Log而非Log表达的也是同样的意思。我们会在后面介绍PEAR频道。

Pyrus也可以安装Log包：

php ./pyrus.phar install pear/Log

```
Pyrus version 2.0.0a1 SHA-1: 27EB8EB427EA50C05691185B41BBA0F0666058D0
Using PEAR installation found at /usr/share/pear2

Connected...

Installed pear.php.net/Log-1.12.0
Optional dependencies that will not be installed, use --optionaldeps:
pear.php.net/DB depended on by pear.php.net/Log
pear.php.net/MDB2 depended on by pear.php.net/Log
pear.php.net/Mail depended on by pear.php.net/Log
```

从这里往后，我们将只提及这两个系统的差异，不再分别举例说明各自的用法了。

如果你想安装的代码包强制性地依赖某些代码包，那么安装就会中止并报告一条警告信息：

```
pear/dialekt requires package "pear/Fandango" (version >= 10.5.0)
No valid packages found
```

你可以先安装所依赖的代码包，然后重新安装，也可以在使用`pear install`命令时带上`-o`参数。

```
pear install -o dialekt
```

`-o`参数确保PEAR安装程序会自动安装任何必要的代码包。有些PEAR包会有一些可选的依赖包，这些软件包在使用`-o`参数时将不会被安装。使用参数`-a`代替`-o`即可自动安装所有可选的软件包。

注意　默认情况下，Pyrus将尝试安装依赖。它支持-o标志，使用这个标志也会导致安装可选的依赖。

虽然PEAR被设计成通过网络在线访问代码库，但很多程序员也会私下创建与PEAR兼容的代码包，因为这样更方便安装。只要给出压缩包（tar或gzip的包）的地址，用PEAR来安装它就和安装官方的包一样简单：

```
$ pear install -o http://www.example.com/dialekt-1.2.1.tgz
```

```
downloading dialekt-1.2.1.tgz ...
Starting to download dialekt-1.2.1.tgz (1,783 bytes)
....done: 1,783 bytes
install ok: channel://pear.php.net/dialekt-1.2.1
```

也可以先下载一个包，然后通过命令行在本地安装它。下面我们使用Unix命令wget来获取dialekt代码包，然后通过命令行在本地安装它：

```
$ wget -nv http://127.0.1.2:8080/dialekt-1.2.1.tgz
20:21:40 URL:http://127.0.1.2:8080/dialekt-1.2.1.tgz [1783/1783]
    -> "dialekt-1.2.1.tgz.1" [1]
$ pear install dialekt-1.2.1.tgz
install ok: channel://pear.example.com/Dialekt-1.2.1
```

还可以通过引用XML文件（通常命名为package.xml）来安装PEAR包，该XML文件提供了要安装的文件的信息。

```
$ pear install package.xml
install ok:  channel://pear.example.com/Dialekt-1.2.1
```

PEAR 频道

PEAR在1.4版本中引入了频道（channel）这个概念。这个强大的功能允许你把整个代码库放在pear.php.net这个官方站点之外，以便更新和管理依赖关系。也就是说，你的程序可以同时使用多个PEAR代码库中的包，然后PEAR可以帮用户取得依赖的包。在频道出现以前，自行开发PEAR包的开发人员不得不告诉用户安装所依赖的包或者把依赖的包绑定到发布的程序中。

Sebastian Bergmann的PHPUnit包就是一个真实的例子。为了安装这个包，首先需要让PEAR知道到哪个频道去找它：

```
$ pear channel-discover pear.phpunit.de
Adding Channel "pear.phpunit.de" succeeded
Discovery of channel "pear.phpunit.de" succeeded
```

使用discover建立了和频道的对话后，就可以用phpunit/作为前缀引用其中的一个包。实际上，phpunit是pear.phpunit.de的别名。运行channel-info，就可以获得这个别名：

```
$ pear channel-info pear.phpunit.de
Channel pear.phpunit.de Information:
====================================
Name and Server         pear.phpunit.de
Alias                   phpunit
Summary                 PHPUnit channel server
...
```

注解 Pyrus不支持这个channel-info命令。

然后，就可这样来安装PHPUnit了：

```
$ pear install -a phpunit/PHPUnit
```

```
Unknown remote channel: pear.symfony-project.com
phpunit/PHPUnit can optionally use package "channel://pear.symfony-project.com/YAML"
(version >= 1.0.2)
phpunit/PHPUnit can optionally use PHP extension "pdo_mysql"
phpunit/PHPUnit can optionally use PHP extension "soap"
phpunit/PHPUnit can optionally use PHP extension "xdebug" (version >= 2.0.5)
downloading PHPUnit-3.4.11.tgz ...
Starting to download PHPUnit-3.4.11.tgz (254,439 bytes)
.....................................................done: 254,439 bytes
downloading Image_GraphViz-1.2.1.tgz ...
Starting to download Image_GraphViz-1.2.1.tgz (4,872 bytes)
...done: 4,872 bytes

install ok: channel://pear.phpunit.de/PHPUnit-3.4.11
install ok: channel://pear.php.net/Image_GraphViz-1.2.1
```

注意使用了-a参数，它告诉PEAR下载所有依赖的包。在这个例子中，Image_GraphViz包来自pear.php.net频道。你可能也注意到了该命令没能安装包YAML。这是因为我没有在pear.symfony-project.com频道上运行channel-discover。Pyrus将这视做致命的错误，所以在设置了-o（可选依赖）标志并运行channel-discover之前，你要确保已发现了所有相关频道。

15.4 使用 PEAR 包

安装PEAR包之后，就可以立刻在项目中使用它。PEAR目录应该已在包含路径（include path）之中，安装PEAR包便可以保证这一点。下面我们安装PEAR_Config及相关的包：

```
$ pear install -a Config
downloading Config-1.10.11.tgz ...
Starting to download Config-1.10.11.tgz (27,718 bytes)
.........done: 27,718 bytes
downloading XML_Parser-1.2.8.tgz ...
Starting to download XML_Parser-1.2.8.tgz (13,476 bytes)
...done: 13,476 bytes
install ok: channel://pear.php.net/Config-1.10.11
install ok: channel://pear.php.net/XML_Parser-1.2.8
```

下面的代码显示如何包含这个包：

```
require_once("Config.php");

class MyConfig {
    private $rootObj;

    function __construct( $filename=null, $type='xml' ) {
        $this->type=$type;
        $conf = new Config();
        if ( ! is_null( $filename ) ) {
            $this->rootObj = $conf->parseConfig($filename, $type);
        } else {
            $this->rootObj = new Config_Container( 'section', 'config' );
            $conf->setroot($this->rootObj);
```

```
        }
    }

    function set( $secname, $key, $val ) {
        $section=$this->getOrCreate( $this->rootObj, $secname );
        $directive=$this->getOrCreate( $section, $key, $val );
        $directive->setContent( $val );
    }

    private function getOrCreate( Config_Container $cont, $name, $value=null ) {
        $itemtype=is_null( $value )?'section':'directive';
        if ( $child = $cont->searchPath( array($name) ) ) {
            return $child;
        }
        return $cont->createItem( $itemtype, $name, null );
    }

    function __toString() {
        return $this->rootObj->toString( $this->type );
    }
}
```

代码在一开始包含了Config.php。大多数PEAR包都是这样调用的，它们会提供一个单一的顶级访问入口[①]。包内有的require语句均由该包自行处理。

本例使用了Config包提供的类：Config和Config_Container。通过Config包，我们可以访问和创建不同格式的配置文件。MyConfig类使用Config包来操作配置数据。下面是一个用法示例：

```
$myconf = new MyConfig();
$myconf->set("directories", "prefs", "/tmp/myapp/prefs" );
$myconf->set("directories", "scratch", "/tmp/" );
$myconf->set("general", "version", "1.0" );
echo $myconf;
```

这默认会生成XML格式的输出：

```
<config>
  <directories>
    <prefs>/tmp/myapp/prefs</prefs>
    <scratch>/tmp/</scratch>
  </directories>
  <general>
    <version>1.0</version>
  </general>
</config>
```

和大多数示例代码一样，上面这个类不是完整的——它还需要额外的错误检查和将配置数据写入文件的方法。但它已经非常实用了，这要感谢PEAR包。通过传递不同类型的字符串给Config，我们可以用不同的格式（比如PHP应用本身使用的INI格式）来显示之前的输出。当然，Config包的相关细节超出了本章范围，但是该包属于官方的PEAR包，你可以在网站http://pear.php.net/

① 例如Config包使用Config.php作为包访问入口，HTTP_Client包使用Client.php作为包访问入口。——译者注

上找到其API的介绍。不论什么情况下，相信你总是期望能够以最容易的方式将PEAR包的功能集成到自己的代码中。PEAR包通常会提供较好的简洁明了API文档。

注解 PEAR包的缺点在于它对旧版PHP的支持使得它很难符合较新版本的要求。和很多PEAR包一样，Config现在依赖于已弃用的语言功能，而考虑到向后兼容性，不能轻易丢掉那些功能。要想关闭与此相关的警告，你可以在php.ini文件中像下面这样设置错误报告指令：

```
error_reporting = E_ALL & ~E_DEPRECATED
```

PEAR 的错误处理

很多情况下，官方的PEAR包都使用标准的PEAR错误处理类`PEAR_Error`。如果操作出错，通常返回`PEAR_Error`对象来代替预期的返回值。这个动作应该记录下来，可以使用静态方法`PEAR::isError()`来检查返回值。

```
$this->rootObj = @$conf->parseConfig($filename, $type);
if ( PEAR::isError( $this->rootObj ) ) {
    print "message:    ". $this->rootObj->getMessage()   ."\n";
    print "code:       ". $this->rootObj->getCode()      ."\n\n";
    print "Backtrace:\n";

    foreach ( $this->rootObj->getBacktrace() as $caller ) {
        print $caller['class'].$caller['type'];
        print $caller['function']."() ";
        print "line ".$caller['line']."\n";
    }
    die;
}
```

下面，我们检测`Config::parseConfig()`的返回值。

```
PEAR::isError( $this->rootObj )
```

功能上等同于

```
$this->rootObj instanceof PEAR_Error
```

因此在条件语句的判断中，我们知道`$this->rootObj`是`PEAR_Error`对象而不是`Config_Container`对象。

如果我们确定得到的是一个`PEAR_Error`对象，可以获得该错误的详细信息。有3个最常用的对象方法：`getMessage()`返回该错误的描述信息；`getCode()`返回和错误类型相对应的整数（通常类库的作者会将该数值定义成常量，并提供相应的文档说明）；最后，`getBacktrace()`返回一个数组，这个数组包括导致该错误发生的类和方法。这3个方法使我们能够跟踪代码执行的过程并准确定位错误发生的起始位置。可以看到，`getBacktrace()`返回的是一个数组，它描述了每一个导致错误的方法或函数，如表15-1所示。

表15-1 由PEAR_Error::getBacktrace()提供的字段

字　段	描　述
file	PHP文件的完整路径
args	传递给方法或函数的参数
class	类名（如果错误发生在类中）
function	函数或类方法的名称
type	如果错误发生在类中，调用该方法的类型（::或->）
line	行号

在PHP 5出现以前，PEAR_Error会“弄脏”方法的返回值。随着PHP 4退出历史舞台，PEAR_Error将逐渐被淘汰。

虽然仍有许多包继续使用PEAR_Error，并且还可能会继续使用它一段时间，但是更多的PEAR包开始使用PEAR_Exception。如果你用过XML_Feed_Parser包，那么就将捕获异常而不是检测返回类型：

```
$source="notthere";
try {
    $myfeed = new XML_Feed_Parser( $source );

} catch ( XML_Feed_Parser_Exception $e ) {
    print "message: ".       $e->getMessage()       ."\n";
    print "code: ".          $e->getCode()          ."\n";
    print "error class: ".   $e->getErrorClass()    ."\n";
    print "error method: ".$e->getErrorMethod()     ."\n";
    print "trace: ".         $e->getTraceAsString()."\n";
    print "error data: ";
    print_r(                 $e->getErrorData() );
}
```

一般来说，PEAR包会对PEAR_Exception进行扩展，这样能够增加任何需要的功能，更主要的是为了能够使用catch子句来区别具体的Exception类型。当然，PEAR_Exception本身继承自Exception，因此可以使用第4章介绍过的Exception类的那些标准类方法。除此之外，新增的方法也很有用。比如getErrorClass()和getErrorMethod()告诉我们产生错误的类和方法。getErrorData()可能在关联数组中包含了额外的错误信息，虽然这是留给扩展类来实现的。在抛出异常前，PEAR_Exception对象可以用另外一个Exception对象或者Exception对象数组来初始化。通过这种方式，PEAR包可以将Exception对象封装起来。通过调用PEAR::getCause()，可以得到封装过的异常。返回值可能是以下3种情况之一：一个封装过的Exception对象；如果超过一个Exception对象，则返回一个数组；如果什么都没有，则返回null。

PEAR_Exception也使用观察者（Observer）模式，允许注册回调函数或方法。当异常抛出时，该函数或方法会被自动调用。首先，创建一些错误条件：

```
class MyPearException extends PEAR_Exception {
}
```

```
class MyFeedThing {
    function acquire( $source ) {
        try {
            $myfeed = @new XML_Feed_Parser( $source );
            return $myfeed;
        } catch ( XML_Feed_Parser_Exception $e ) {
            throw new MyPearException( "feed acquisition failed", $e );
        }
    }
}
```

我扩展了PEAR_Exception，并创建了一个封装XML_Feed_Parser的简单类。如果XML_Feed_Parser构造方法抛出异常，就捕获它并将其传给MyPearException的构造函数，然后再重新抛出。利用这个小技巧，可以把错误封装成我自己定制的错误并进行处理，同时该错误仍然绑定到导致出错的那行代码。

下面是客户端类和两行调用它的代码：

```
class MyFeedClient {
    function __construct() {
        PEAR_Exception::addObserver( array( $this, "notifyError") );
    }

    function process() {
        try {
            $feedt = new MyFeedThing();
            $parser = $feedt->acquire('wrong.xml');
        } catch ( Exception $e ) {
            print "an error occurred. See log for details\n";
        }
    }

    function notifyError( PEAR_Exception $e ) {
        print get_class( $e ).":";
        print $e->getMessage()."\n";
        $cause = $e->getCause();
        if ( is_object( $cause ) ) {
            print "[cause] ".get_class( $cause ).":";
            print $cause->getMessage()."\n";
        } else if ( is_array( $cause ) ) {
            foreach( $cause as $sub_e ) {
                print "[cause] ".get_class( $sub_e ).":";
                print $sub_e->getMessage()."\n";
            }
        }
        print "----------------------\n";
    }
}

$client = new MyFeedClient();
$client->process();
```

示例中的语句都设计成注定要出错。首先注意构造方法。PEAR_Exception::addObserver()

是一个以回调方法为参数的静态方法，其参数可以是函数名或包含对象引用和方法名的数组。每次抛出PEAR_Exception时，都会调用这个回调方法或函数。这个小技巧允许我们设计MyFeedClient来记录所有的异常。

process()方法将一个不存在的文件传递给MyFeedThing::acquire()，acquire()把文件传递给XML_Feed_Parser的构造方法，显然这一定会触发错误。我们捕获不可避免的异常然后输出简单的消息。notifyError()是在MyFeedClient构造方法中引用的回调方法，注意它期望得到一个PEAR_Exception对象。在本例中，我们只是查询该对象并输出错误信息。但在实际项目中，可能会把错误信息数据发送到日志。注意对PEAR_Exception::getCause()的调用。它可能会返回一个数组或单独的Exception对象，所以我们对两种情况都进行了处理。如果运行代码，就会得到下面的输出结果：

```
XML_Feed_Parser_Exception:Invalid input: this is not valid XML
----------------------
MyPearException:feed acquisition failed
[cause] XML_Feed_Parser_Exception:Invalid input: this is not valid XML
----------------------
an error occurred. See log for details
```

在这个例子中，两个异常的抛出（第一个由XML_Feed_Parser抛出，第二个由MyFeedThing抛出）都会调用日志方法。XML_Feed_Parser_Exception对象在日志输出中生成了第二种形式，因为它的抛出会生成一个MyPearException对象，而该对象的错误处理与notifyError()的错误处理不同。

15.5 创建自己的PEAR包

PEAR包通常都有较好的文档，并且易于使用。但PEAR包是如何创建的，如何创建自己的PEAR包呢？本节我们来分析一下PEAR包的组成。

15.5.1 package.xml

package.xml文件是任何PEAR包的核心，它提供了某个包的相关信息，决定了该包如何被安装、安装在哪里，并定义了该包所依赖的类库。无论是通过URL、本地文件系统，还是通过tar/gzip压缩文件方式来安装PEAR包，PEAR安装程序都需要package.xml文件来获得指示。

无论包设计和规划得有多好，如果缺少了构建文件，安装就会失败。下面尝试安装一个缺少package.xml的包：

```
$ pear install baddialekt.tgz
could not extract the package.xml file from "baddialekt.tgz"
Cannot initialize 'baddialekt.tgz', invalid or missing package file
Package "baddialekt.tgz" is not valid
install failed
```

PEAR安装程序首先要解压缩文件到一个临时目录，然后查找package.xml并失败。既然package.xml是如此重要，那么package.xml的内容是什么呢？

15.5.2　package.xml 的组成

package.xml文件必须以XML声明开头，然后是一个根元素package，其中包含所有的元素：

```
<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.11" version="2.0"
 xmlns="http://pear.php.net/dtd/package-2.0"
 xmlns:tasks="http://pear.php.net/dtd/tasks-1.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://pear.php.net/dtd/tasks-1.0
 http://pear.php.net/dtd/tasks-1.0.xsd
 http://pear.php.net/dtd/package-2.0
 http://pear.php.net/dtd/package-2.0.xsd">

<!-- additional elements here -->

</package>
```

上面的例子会导致安装出错。PEAR安装程序有几个必需的元素。首先，我们要提供概述信息。

```
 <name>Dialekt</name>
 <channel>pear.example.com</channel>
 <summary>A package for translating text and web pages into silly tones
of voice</summary>
 <description>Be the envy of your friends with this hilarious dialect
translator. Easy to extend and altogether delightful.
 </description>

<!-- additional elements here -->
```

我们新增的这几个元素的用途都可以顾名思义。name元素定义该包的名称，summary元素包含一行对该包的描述文字，description元素则提供更详细的说明。除了channel元素外，所有这些元素通常情况下都是必需的。如果不想把包增加到频道中，可以使用uri元素来代替channel元素，其中包含一个指向包文件的URI。

```
<uri>http://www.example.com/projects/Dialekt-1.2.1</uri>
```

文件名不能包含扩展名，即使包文件本身可能以.tgz扩展结尾。

接下来应该提供该包的开发团队的信息。至少要包括lead元素：

```
<lead>
  <name>Matt Zandstra</name>
  <user>mattz</user>
  <email>matt@example.com</email>
  <active>yes</active>
</lead>
```

在此之后，可以用类似的办法定义其他项目参与者。除了lead，还可以使用developer、contributor或helper元素。这些元素的命名按照PEAR社区默认的使用惯例，但是它们也可以用于非PEAR的项目。user元素表示PEAR的贡献者用户名。大部分团队都使用类似的处理方法允许用户使用这样的用户名登录到Subversion或开发服务器中。

将文件放进项目之前，还需要提供一些细节信息：

```
<date>2010-02-13</date>
<time>18:01:44</time>
<version>
 <release>1.2.1</release>
 <api>1.2.1</api>
</version>
<stability>
 <release>beta</release>
 <api>beta</api>
</stability>
<license uri="http://www.php.net/license">PHP License</license>
<notes>initial work
</notes>
```

虽然这些元素的用途你应该都能从其命名中猜到，但是有两个功能特别值得一提。version元素内部的release元素是指当前软件包的版本号。该release元素被用于PEAR中软件包相互依赖性的计算。如果安装系统时需要Dialekt 1.0.0，而安装用户的系统上只有0.2.1，那么PEAR将会暂停安装或尝试获取较高的版本，这取决于PEAR运行的模式。另外，api元素使你可以追踪代码中可能影响兼容性的接口的变动。

stability元素与version类似，也用于分隔release和api。可用的值有snapshot、devel、alpha、beta或stable。你应该选择一个最合适的词来描述你的项目。

如果根据指定的许可证条款（比如GNU的GPL许可证）发布自己的包，需要在license元素中添加相关信息。

和summary及description不同，notes元素允许在添加的内容中使用换行符。

15.5.3 contents 元素

现在，可以指定包中的文件与目录了。contents元素定义了要包含到压缩包（有时称为tarball，因为它是用tar或Gzip工具压缩生成的）的文件。你可以用dir和file元素来描述压缩包文件的结构。

下面是一个简化的例子：

```
<contents>
  <dir name="/">
   <dir name="data">
    <file name="alig.txt" role="data" />
    <file name="dalek.txt" role="data" />
   </dir> <!-- /data -->
   <dir name="Dialekt">
    <file name="AliG.php" role="php" />
    <file name="Dalek.php" role="php" />
   </dir>
 </contents>
```

每个文件在PEAR包中都有一个角色。每个角色（role）都和默认的位置（可配置）相关联。表15-2描述了常见的角色。

表15-2　一些常见的PEAR文件角色

角　色	描　述	PEAR配置名称	示例位置
php	PHP文件	php_dir	/usr/local/lib/php
test	单元测试文件	test_dir	/usr/local/lib/php/test/<package>
script	命令行脚本	bin_dir	/usr/local/bin
data	资源文件	data_dir	/usr/local/lib/php/data/<package>
doc	文档文件	doc_dir	/usr/local/lib/php/doc/<package>

在安装时，角色为doc、data和test的文件并不会直接安装到相应的目录下，而是在test_dir和data_dir目录下再建一个与该包同名的目录，然后把文件复制进去。[①]

在PEAR项目中，每个部分都有一个角色，每个角色都有其特定的安装位置。如果没有足够的权限来操作默认的路径，可以用pear命令行工具来自定义安装位置：

```
$ pear config-set php_dir ~/php/lib/

$ pear config-set data_dir ~/php/lib/data/
$ pear config-set bin_dir ~/php/bin/
$ pear config-set doc_dir ~/php/lib/doc/

$ pear config-set test_dir ~/php/lib/test/
```

注解　Pyrus使用set（而非config-set）来执行同样的设置。

现在PEAR将使用你自己的目录而不是表15-2中所定义的默认目录。记住如果你改变了默认的安装路径，就要把lib目录加到包含路径中去：可以加在php.ini文件中，也可以加在.htaccess文件中或者在代码中使用ini_set()函数。另外，要确定bin目录在shell的环境变量中，这样命令行才能找到相应的命令。

我们的例子围绕一个虚构的包Dialekt展开。下面是这个包的目录和文件结构：

```
./package.xml
./data
    ./data/dalek.txt
    ./data/alig.txt
./script
    ./script/dialekt.sh
    ./script/dialekt.bat
./cli-dialekt.php
./Dialekt.php
./Dialekt
    ./Dialekt/AliG.php
    ./Dialekt/Dalek.php
```

可以看到，我们已经在数据结构中添加了一些标准的PEAR角色，因此其中包括了data和script

---

① 举个例子，我们安装HTML_QuickForm，其doc角色对应目录为/usr/local/lib/php/doc/HTML_QuickForm，则实际放置文档文件的目录为该软件包安装目录的子目录，如/usr/local/lib/php/doc/HTML_QuickForm/docs。data和test角色同理。——译者注

目录。在最高一层目录下，有两个PHP文件——cli-dialekt.php和Dialekt.php，它们会被安装到PEAR根目录下（默认为/usr/local/php/lib）。Dialekt.php是调用客户代码的入口文件。用户要能使用以下代码来包含Dialekt：

```
require_once("Dialekt.php");
```

其余的PHP文件（Dalek.php 和AliG.php）存放在Dialekt目录下，而Dialekt目录会被添加到PEAR根目录下（它们用于实现具体的功能，用以呈现网页和文本文件）。Dialekt.php将会包含这两个文件。为了让PEAR可以在命令行下调用Dialekt，我们实现了一个shell脚本，它将被复制到PEAR的script目录下。另外，Dialekt会使用存储在文本文件中的配置信息，这些文件将被安装到PEAR的data目录下。

下面是完整的contents元素：

```
<contents>
  <dir name="/">
   <dir name="data">
    <file name="alig.txt" role="data" />
    <file name="dalek.txt" role="data" />
   </dir> <!-- /data -->
   <dir name="Dialekt">
    <file name="AliG.php" role="php" />
    <file name="Dalek.php" role="php" />
   </dir> <!-- /Dialekt -->
   <dir name="script">
    <file name="dialekt.bat" role="script">
     <tasks:replace from="@php_dir@" to="php_dir" type="pear-config" />
     <tasks:replace from="@bin_dir@" to="bin_dir" type="pear-config" />
     <tasks:replace from="@php_bin@" to="php_bin" type="pear-config" />
    </file>
    <file name="dialekt.sh" role="script">
     <tasks:replace from="@php_dir@" to="php_dir" type="pear-config" />
     <tasks:replace from="@bin_dir@" to="bin_dir" type="pear-config" />
     <tasks:replace from="@php_bin@" to="php_bin" type="pear-config" />
    </file>
   </dir> <!-- /script -->
   <file name="cli-dialekt.php" role="php" />
   <file name="Dialekt.php" role="php">
    <tasks:replace from="@bin_dir@" to="bin_dir" type="pear-config" />
   </file>
  </dir> <!-- / -->
 </contents>
```

在上述代码中，我们增加了一个新元素tasks:replace，它可以让from属性指定的内容被to属性指定的内容（更确切地说是指pear-config命令指定的to属性相应选项的值）所替换。例如，在安装之前我们的Dialekt.php文件看起来是这样的：

```
<?php
/*
 * Use this from PHP scripts, for a CLI implementation use
 * @bin_dir@/dialekt
 */
class Dialekt {
```

```
    const DIALEKT_ALIG=1;
    const DIALEKT_DALEK=2;
//...
}
```

安装之后，代码的注释部分就会变成这样：

```
/*
 * Use this from PHP scripts, for a CLI implementation use
 * /home/mattz/php/bin/dialekt
 */
```

15.5.4 依赖

尽管PEAR的软件包通常都是独立的，但有时它们也会用到其他软件包，这就产生了依赖（dependency）。如果用户的系统上未安装依赖的软件包，则使用它的软件包不会像预期的那样运行。

dependencies标签是package.xml文件必备的元素。在dependencies标签中，你至少要指定PHP和PEAR安装器（PEAR installer）的版本。

```
<dependencies>
  <required>
   <php>
    <min>5.3.0</min>
   </php>
   <pearinstaller>
    <min>1.4.1</min>
   </pearinstaller>
   <!-- other dependencies here if required -->
  </required>
 </dependencies>
```

php和pearinstall都可以包含min、max和exclude元素。exlude定义了不能和包兼容的软件版本，可以在这里提供任意多个版本。pearinstaller元素也能包含recommended元素，可以在其中为包设置首选的安装器。

如果required元素中的这些依赖软件包未安装，PEAR默认会拒绝安装这个包。一个包可以依赖于其他包、PHP扩展（比如zlib或GD）或者特定的PHP版本。比如这里，我们强调Dialekt可以使用Fandango包的10.5.0或更高版本（注意，我把以下代码放在了required元素中）：

```
<package>
 <name>Fandango</name>
 <channel>pear.example.com</channel>
 <min>10.5.0</min>
</package>
```

注意channel元素。如果在安装PEAR包时使用了-a参数（指定获取所有依赖包），则PEAR会到channel元素指定的频道去查找需要的包。你必须指定channel元素或uri元素。uri元素应指向一个包文件：

```
<package>
 <name>Fandango</name>
 <uri>http://www.example.com/packages/fandango-10.5.0.tgz</uri>
</package>
```

package元素可接受与pearinstaller元素相同的依赖说明符，另外还可接受conflicts元素，用于定义与本软件包冲突的版本。

除了package外，也可以指定extension、os或arch。表15-3总结了这些依赖元素。

表15-3　package.xml依赖的类型

元　素	描　述
php	PHP程序
package	PEAR 包
extension	PHP扩展（能够编译到PHP中，比如zlib或GD）
arch	操作系统和处理器结构
os	操作系统

到目前为止，我们指定的都是强制性的依赖软件包。实际上，在requires之后可以指定optional元素，它与requires接受相同的依赖元素。当PEAR安装包时发现某个可选的依赖包不存在，会发出警告，但安装仍将继续。你可以在optional元素中增加依赖包，这样在没有某些需要的包或扩展时，你的包也能运行。

如果用户运行pear install命令时使用了-o标志：

```
pear install -o package.xml
```

那么PEAR就会下载和安装required元素指定的所有必需的依赖包（注意，给Pyrus传递-o将安装可选包）。如果运行命令时使用了-a标志，也会自动下载相关包，但将下载所有依赖包（包括必需的和可选的）。

15.5.5　使用 phprelease 进行灵活的自定义安装

不仅可以用contents元素定义PEAR包中的文件，还可以使用phprelease来调整实际安装到用户系统中的文件。下面是包中的两个phprelease元素：

```
<phprelease>
<installconditions>
 <os>
  <name>unix</name>
 </os>
</installconditions>
<filelist>
 <install as="dialekt" name="script/dialekt.sh" />
 <install as="dalek" name="data/dalek.txt" />
 <install as="alig" name="data/alig.txt" />
 <ignore name="script/dialekt.bat" />
</filelist>
</phprelease>
```

15

```
<phprelease>
 <installconditions>
  <os>
   <name>windows</name>
  </os>
 </installconditions>
 <filelist>
  <install as="dialekt" name="script/dialekt.bat" />
  <install as="dalek" name="data/dalek.txt" />
  <install as="alig" name="data/alig.txt" />
  <ignore name="script/dialekt.sh" />
 </filelist>
</phprelease>
```

installconditions元素用于决定phprelease元素是否执行。它可以接受os、extension、arch和php等元素。这些元素和同名的依赖包一样工作。你既可以提供符合installconditions的phprelease元素，也可以提供一个默认的版本来执行。如果安装环境与installconditions不匹配，就会自动执行默认版本。

让我们关注unix版本的phprelease。install元素指定dialekt.sh文件在安装时应重命名为dialekt。

我们指定了要安装的数据文件应去掉.txt扩展名。我们不需要指定dialekt子目录——它会自动用data角色的目录包含文件。注意install元素的as元素也去除了contents元素中指定的前导目录部分，这意味着它们将被安装于<data_dir>/dialekt/dalek和<data_dir>/dialekt/alig。

还要注意在Unix模式下不需要安装dialekt.bat脚本文件。ignore元素被用于忽略该文件。准备好这一切后，PEAR包就可以在本地安装了。

15.5.6 准备发布包

我们已经创建了自己的包并生成了package.xml文件，现在也该归档并压缩我们的产品了。

一个简单的PEAR命令就可以完成这件事情。必须在项目的根目录下运行以下子命令：

```
$ pear package package.xml
Analyzing Dialekt/AliG.php
Analyzing Dialekt/Dalek.php
Analyzing cli-dialekt.php
Analyzing Dialekt.php
Package Dialekt-1.2.1.tgz done
```

执行以上命令会生成一个可用于发布的、打包并压缩过的文件（里面包含所有必需的文件以及package.xml文件）。你可以把软件包发布给用户，让他们直接下载。如果你的包依赖于其他软件包，可以使用package元素的uri子元素代替频道。但如果需要很多互相依赖的包，也许你需要看看下一节介绍的内容。

15.5.7 创建自己的 PEAR 频道

为什么要创建自己的频道呢？除了看起来比较酷以外，主要的好处是可以利用PEAR自动化的依赖关系管理以及方便的安装和升级功能。对于用户来说非常方便，只需要一个完整的URL

路径就可以完成安装。如果你设计了一个类库系统，其中既包括工具型的包，也包括完整的应用，那么如何管理这个类库将变得复杂。而对于用户来说，管理项目中多个互相依赖的包也是一件头痛的事情，特别是当软件包还在不断改进的情况下。

在这一节中，我将重点介绍针对Pyrus创建和托管频道的机制。有两个原因。首先，Pyrus是趋势，在接下来的几年中，Pyrus可能会得到普及。Pyrus是从头开始编写的，无需考虑向后兼容性，这样就会使代码更加整齐，结构更加清晰。其次，在Pyrus官方网站（http://pear2.php.net）上可以下载到所需的包。对于频道管理来说，虽然从2006年开始就有了一种相当好的面向PEAR的解决方案，但还只能从外部网站（http://greg.chiaraquartet.net）获得，PEAR站点上还是没有相关资源。不管原因是什么，它都不能给予软件将继续得到支持的信心。

当然，由于Pyrus还是新生事物，使用过程中难免会出现一些小问题。我将对此进行介绍，并给出解决方案。

即使仍使用PEAR构建包，那么也完全可以使用Pyrus频道管理来提供PEAR包。在开始之前，先看看有哪些系统要求。

- 服务器的root权限。
- 对于Web服务器（如Apache）的管理权限。同时，Web服务器要支持二级域名（如pear.your-server.com）。

如果你对服务器没有这类权限，也不要担心，可以使用第三方提供程序，比如Google Code（http://code.google.com）来托管频道。不管你决定怎样托管它，首先需要定义频道并向其中添加一些包。

1. 使用PEAR2_SimpleChannelServer定义频道

PEAR2_SimpleChannelServer实际上是一种误称。它只是一个工具，用于定义通道和组织包文件，以为服务做准备，而不是服务器或服务器组件。

在写作本书时，Pyrus网站声称应该使用Pyrus本身来安装PEAR2_SimpleChannelServer。实际上，当前这样做会导致一个错误。但是，你可以从http://pear2.php.net/get/PEAR2_SimpleChannel-Server-0.1.0.phar获得这个phar包。

> **注解** 你可能会发现PEAR2_SimpleChannelServer的安装将得到改进。可以登录http://pear2.php.net/PEAR2_SimpleChannelServer查看进度。

有了phar文件以后，可以将它放在某个位置，并将其重命名为方便使用的名字（我选择使用/usr/share/pearscs.phar），然后就可以运行它来安装基本的频道环境了。

```
php /usr/share/pearscs.phar create pear.appulsus.com "Appulsus PHP repository" appulsus
```

```
Created pear.appulsus.com
      | ./channel.xml
      | ./rest/
      | ./get/
```

创建子命令需要频道名，通常是一个主区域或子区域（稍后将介绍）、摘要或可选的别名。如果没有给出别名，系统建议从名称实参中选择一个。我选择的是appulsus。接下来创建一个名为channel.xml的文件，该文件会定义频道。该命令还会创建两个空的目录get/和rest/。我想稍后通过Web访问这两个目录，所以我在Web目录中运行这个命令。

频道创建完了，可以添加一些分类了。

```
php /usr/share/pearscs.phar add-category productivity "things to help you work"
php /usr/share/pearscs.phar add-category fun "the fun never stops"
```

add-category子命令接受两个参数：类别的名称和描述。它只会修正channel.xml文件。

向系统中添加包之前，我必须确保新频道能被识别，否则当构建包的时候PEAR或Pyrus会发出警告信息。为此，我需要另一个Pyrus包PEAR2_SimpleChannelFrontend。

2. 使用PEAR2_SimpleChannelFrontend管理PEAR频道

再说一遍，Pyrus网站上关于使用包的最佳方式的说法并不一致。在本书写作时，推荐的安装方式：

```
php pyrus.phar install PEAR2_SimpleChannelFrontend
```

会导致一个错误。读到此处的时候，你可以从包页面http://pear2.php.net/PEAR2_SimpleChannelServer上得到更多有用的指导。现在，你可以通过http://pear2.php.net/get/PEAR2_SimpleChannel-Frontend-0.1.0.phar获得phar文件，也可以通过http://pear2.php.net/get/查看归档文件的更多较新版本。

现在已经有了PEAR2_SimpleChannelFrontend phar文件，要想能通过Web获得它，可以将该文件重命名为index.php，并将它放在保存channel.xml文件并且通过Web可访问的目录中。该位置应该与定义频道的区域和子区域相匹配，所以我在前面选择pear.appulsus.com作为频道名。我应该配置Apache 2，以便pear.appulsus.com会解析成index.php所在的目录。下面是从httpd.conf文件（Apache Web服务器的配置文件）中提取的内容：

```
<VirtualHost *:80>
ServerAdmin webmaster@appulsus.com
DocumentRoot /var/www/pear
ServerName pear.appulsus.com
ErrorLog logs/pear.appulsus.com-error_log
TransferLog logs/pear.appulsus.com-access_log
</VirtualHost>
```

这只是确保对http://pear.appulsus.com的请求被路由到了DocumentRoot目录（/var/www/pear），该目录中存放着最近重命名的index.php文件。SimpleChannelFrontend包还要求将一些模块重写（mod_rewrite）指令应用到频道目录：

```
<Directory "/var/www/pear">
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteBase /
RewriteCond %{REQUEST_FILENAME} !-f
```

```
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . index.php [L]
</IfModule>
</Directory>
```

现在已经有了足够的内容来运行浏览器测试。图15-1显示了PEAR2_SimpleChannelFrontend-0.1.0.phar生成的默认页面。

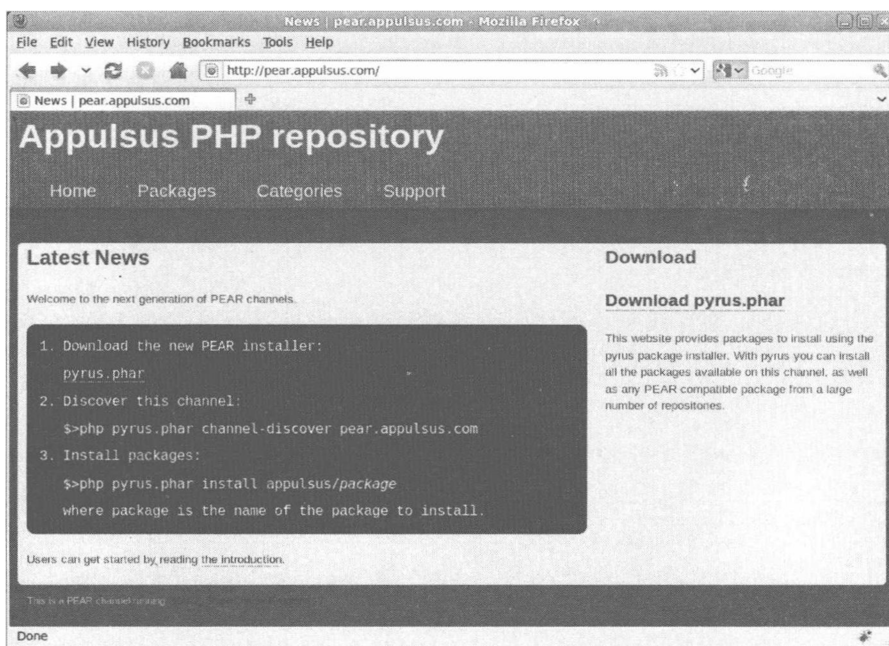


图15-1 频道前端的默认页面

这意味着我已经有了自己的频道——可以使用远程命令行来证实这一点：

```
pear channel-discover pear.appulsus.com
```

```
Adding Channel "pear.appulsus.com" succeeded
Discovery of channel "pear.appulsus.com" succeeded
```

注意，我正在客户端使用PEAR。我想证明这些Pyrus工具可以为运行传统PEAR安装的用户提供服务。到现在为止进展顺利！

3. 管理包

既然pear.appulsus.com可以被识别为频道，那么我也可以修改Dialekt的package.xml了：

```
<name>Dialekt</name>
<channel>pear.appulsus.com</channel>
<summary>A package for translating text and web pages into silly tones of voice</summary>
```

重新生成PEAR包：

```
Analyzing Dialekt/AliG.php
Analyzing Dialekt/Dalek.php
Analyzing cli-dialekt.php
Analyzing Dialekt.php
Package Dialekt-1.2.1.tgz done
```

和以前一样，生成了Dialekt-1.2.1.tgz包，但这次是为pear.appulsus.com频道准备的。现在我可以将这个包移动或上传到我的频道目录。在运行命令进行发布之前，首先需要检查php.ini文件中的设置。顺便说一下，如果你不知道该文件在哪，可以通过命令行运行：

```
php --ini
```

PHP会告诉你具体的位置。找到了该位置以后，打开php.ini文件，找到下面这一行：

```
phar.readonly = Off
```

如果你没有在指定位置找到这行代码或者找到的代码有所不同，你就必须自己添加或修改。如果没有这个设置，发布就可能失败。现在终于准备好发布了。切换到自己的频道目录并运行：

```
php /usr/share/pearscs.phar release Dialekt-1.2.1.tgz mattz
```

```
Release successfully saved
```

再强调一次，我调用了pearscs.phar文件。记住，它是PEAR_SimpleChannelServer包。它需要包文件的路径和维护者的名字，仅此而已。现在我有了包含一个包的频道，我还可以将它和一个类别联系起来：

```
php /usr/share/pearscs.phar categorize Dialekt fun
```

```
Added  Dialekt to fun
```

图15-2显示了如何通过浏览器确认新包。

当然，空谈不如实践，最终还得看安装过程。因此，你可以通过远程系统运行如下命令：

```
pear install appulsus/Dialekt
```

```
downloading Dialekt-1.2.1.tgz ...
Starting to download Dialekt-1.2.1.tgz (1,913 bytes)
....done: 1,913 bytes
install ok: channel://pear.appulsus.com/Dialekt-1.2.1
```

安装成功，剩下要做的就是让别人知道这个频道了！

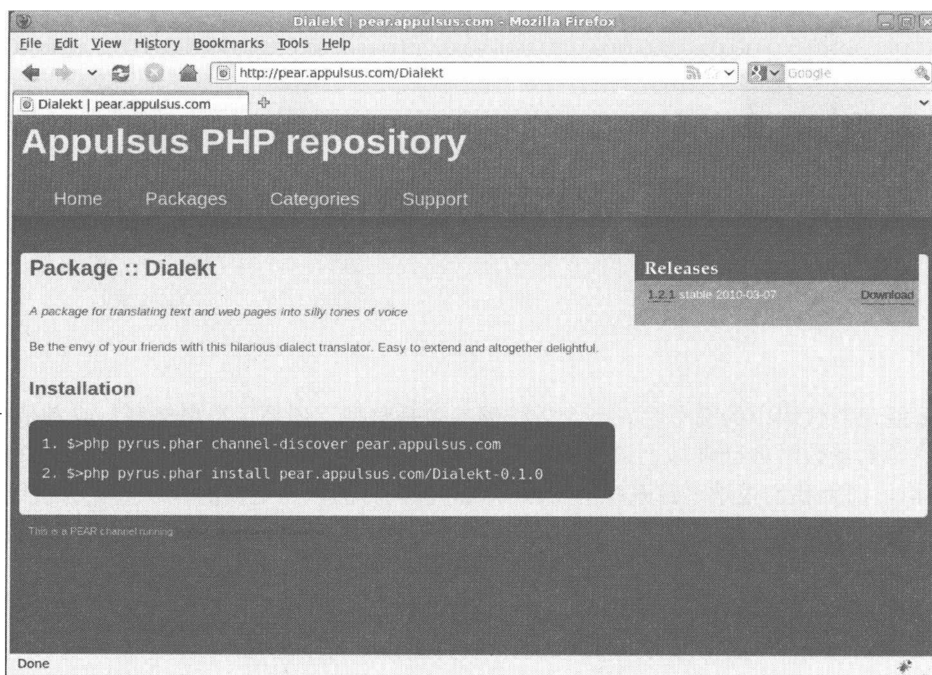


图15-2 频道包页面

15.6 小结

PEAR的内容非常多，限于篇幅我只能简单进行介绍。但是从本章中你应该可以学会如何利用强大的PEAR包加速项目开发。通过package.xml和PEAR安装器（和Pyrus——未来的替代品），你也可以让其他用户访问你的代码。通过创建频道，你可以自动化依赖下载，而且允许第三方包使用你的包，而不需要把你的包打包在一起，也不需要管理复杂的依赖关系。

PEAR最适合于开发良好封装的、自包含的软件包。对于大型应用来说，我们还需要其他解决方案。在本书后面，我们将介绍Phing，它是一种构建完整应用程序的强大工具。

第16章

用phpDocumentor生成文档

你能记清代码中的细节吗？一个方法的参数是整型，还是字符串类型？或者是布尔型？看到参数时你可以马上识别出其类型吗？是否已经整理过代码？编码总是一件麻烦而复杂的事情，而且我们很难追踪系统的工作方式和编码目的。当更多的程序员参与项目开发时，情况可能变得更糟。无论你是否需要在有潜在危险或有特别功能的代码块上做标记，文档都可以提供帮助。对于一个大型代码库来说，文档存在与否往往决定着项目的成败。

本章包括以下内容。

- phpDocumentor程序：安装phpDocumentor并在命令行下运行。
- 文档语法：DocBlock注释和文档标签。
- 为代码编写文档：使用DocBlock注释来提供类、属性和方法的相关信息。
- 创建文档中的链接：链接到网站和文档的其他元素。

16.1 为什么要使用文档

喜欢和厌恶文档的程序员都很多。当你迫于项目最后期限的压力以及经理或客户盯着你的任务时，首先被放弃的通常是文档。这时候我们只想着早点得到能用的代码。当然我们要尽可能编写优雅的代码（尽管这可能也是一种牺牲），但是对于一个正在飞速发展的项目来说，文档就像在浪费时间一样。毕竟你在几天之内可能就要对类进行多次改动。当然，每个人都赞同应该有一份良好的文档，只是没有人愿意降低生产效率来编写文档。

假设有一个很大的项目，其代码库非常庞大，它由聪明的人们编写的优秀代码组成。团队成员已经为这个项目（或相关子项目）工作了5年以上，他们彼此非常熟悉，也完全理解代码。当然，项目的文档也比较少。在每个人的头脑里都有一幅项目的大局图和一个非正式的编码规范来说明所有地方该如何处理。后来队伍扩大了，来了两个新程序员。原来的成员向新人很好地介绍了这个复杂系统的架构，接着就让他们开始工作。这时候代码不写文档的问题就暴露出来了：如果有文档，新人可能一两个星期就能熟悉代码并开始工作；如果没有文档，可能需要几个月才能读懂代码。例如对于一个没有文档的类，新的程序员不得不跟踪每个方法的参数，追踪每个被引用的全局变量，检查继承体系中所有的方法。如果你像我一样曾经是这样一个新成员，不久之后你就会喜欢上文档。

缺少文档要付出很大代价：当新团队成员加入到项目时或者同事被调到他们不熟悉的领域时，需要花时间来熟悉代码；当编程人员深入调试项目组件时，可能会不停地陷入项目中的陷阱，如调用被标记为私有的代码，以错误的类型传递参数变量，不必要地再次创建已经存在的函数等。

因为你不会直接感受到缺少文档的痛苦，所以很难养成写文档的习惯。但如果把文档当做代码一样看待，那么写文档并不困难。如果把文档当做代码本身的一部分增加到源代码中，写文档的过程就会变得很容易，然后你可以运行工具将注释提取出来放到美观的网页中去。本章要介绍的正是这样一个工具。

phpDocumentor由一个名为JavaDoc的Java工具移植而来。这两个工具都能从源代码中提取特定的注释，利用从程序员的注释和源代码中得到的代码结构来构建复杂的API文档。

16.2 安装

安装phpDocumentor最简单的办法是使用PEAR命令。

```
pear upgrade PhpDocumentor
```

注解 *在类似Unix的系统上安装或升级PEAR包，通常需要使用root权限来运行pear命令。*

该命令将访问http://pear.php.net并在系统上自动安装或升级phpDocumentor。

你也可以从SourceForge.net的http://sourceforge.net/projects/phpdocu/files/上下载phpDocumentor包。这里的包是zip和tar格式的。文件系统有了这个包之后，如果编译的PHP支持zlib，就可以直接使用PEAR来安装。

```
pear install PhpDocumentor-1.4.3.tgz
```

另外，你也可以解压缩存档文件，然后直接在解压缩得到的目录中使用phpDocumentor。命令行接口由phpdoc文件处理。你需要把库目录phpDocumentor放到包含路径中。

```
tar -xvzf PhpDocumentor-1.4.3.tgz
cd PhpDocumentor-1.4.3
chmod 755 phpdoc
./phpdoc -h
```

这里把包解压缩到分发目录中，执行phpdoc脚本时使用了-h标志，它可以产生一个使用消息。

如果在安装时遇到了问题，可以查看phpDocumentor包中的INSTALL文件，该文件提供了详细的说明和疑难解答。

16

16.3 生成文档

在写代码之前就生成文档似乎有点怪，但实际上phpDocumentor可以解析源代码的代码结构，因此它甚至可以在开发之前就收集项目的相关信息。

假设我们将为项目megaquiz生成文档。该项目由两个目录组成——command和quiztools，这两个目录中都放置着类文件。目录名即项目中的包名。phpDocumentor可以作为命令行工具运行，

也可以通过Web GUI（Web图形用户界面）来访问。我们将重点讨论命令行方式，因为它更易于将文档的更新操作嵌入到编译工具或shell脚本中。调用phpDocumentor的命令是phpdoc，运行该命令时还需提供一系列参数以生成文档。下面是一个例子：

```
phpdoc -d megaquiz/ \
       -t docs/megaquiz/ \
       -ti 'Mega Quiz' \
       -dn 'megaquiz'
```

-d参数指定为哪个目录下的代码生成文档，-t参数指定将生成的文档置于哪个目录，-ti用来设置项目标题，-dn用来定义默认包名。

如果对未生成文档的项目运行此命令，将得到大量详细信息。在图16-1中，你可以看到输出的菜单页。

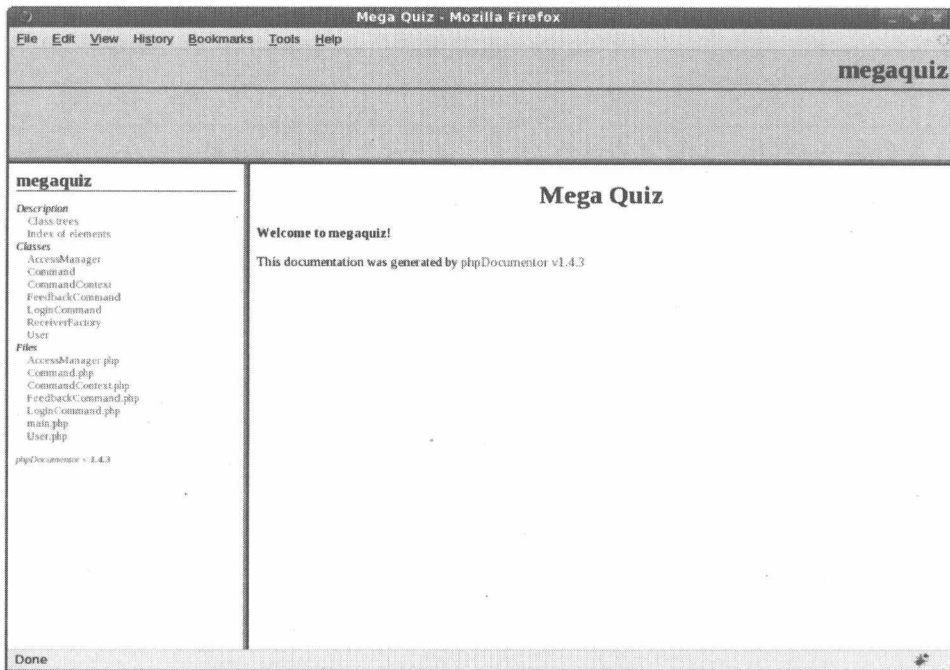


图16-1　phpDocumentor的基本的输出菜单

如图16-1所示，项目中所有的类和文件都列在左侧的框架中。项目名和包名都被组合在文档中。所有的类名都是超链接。在图16-2中，你可以看到第11章中创建的Command类的部分文档。

phpDocumentor非常智能，它能够识别出Command是一个抽象类，并且是FeedbackCommand和LoginCommand的父类。值得注意的是，它还说明了execute()方法需要的参数名和类型。

这样的描述已经足够展示一个大项目的基本轮廓，因此比起没有任何文档来说，它已经是一个很大的改进。我们还可以向源代码中增加注释，以便得到更详细的信息。

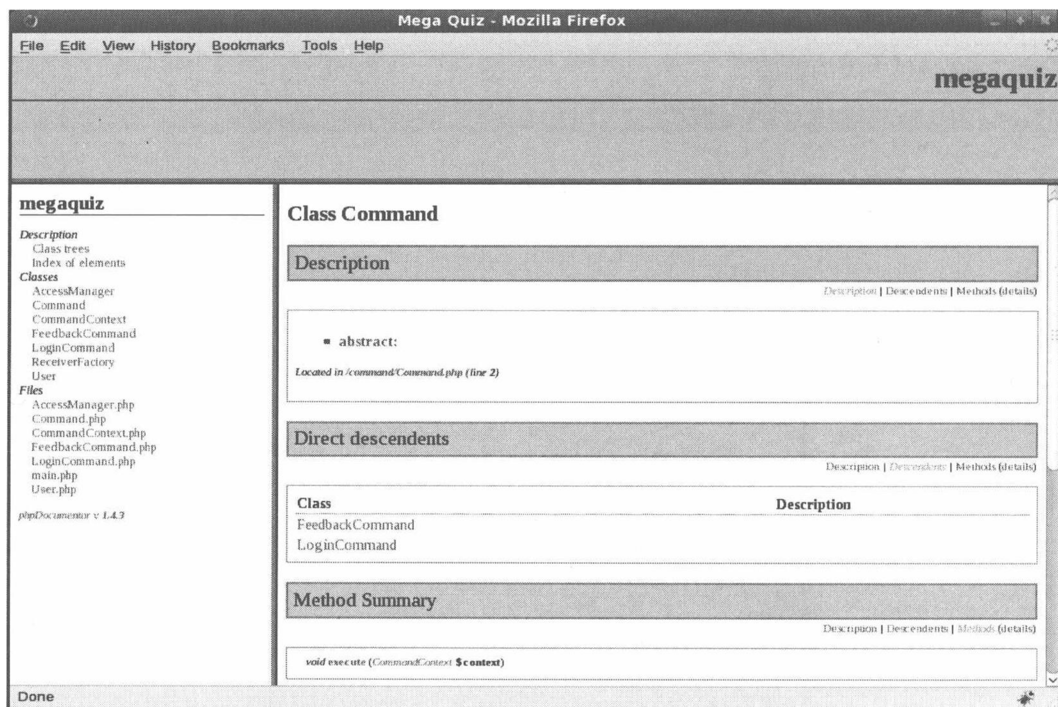


图16-2 Command类的默认文档

16.4 DocBlock 注释

DocBlock（文档区）注释有特定的格式，以便被文档程序识别。它们采用标准的多行注释格式，即每行注释之前有单独附加的星号：

```
/**
 * 我的DocBlock注释
 */
```

phpDocumentor要求DocBlock中包含特定的内容。该注释块包含普通的文字说明（用于说明文件、类、方法或属性），也包含被称为标签的特殊关键字。标签可由符号@定义，并且有可能和参数关联。下面的DocBlock被放置在类的开头，用来告诉phpDocumentor该类属于哪个包：

```
/**
 * @package command
 */
```

如果在项目中所有类前都加上package注释（当然，要用适当的包名），phpDocumentor将整理我们的类。你可以从图16-3中看到包含软件包的phpDocumentor输出。

在图16-3中，可以看到包名显示在导航栏中（见图16-3的右上角）。除了默认的megaquiz

包，还有command和quiztools包。因为我们当前正在查看command包中的类，所以左侧导航栏只显示command包中类的链接。

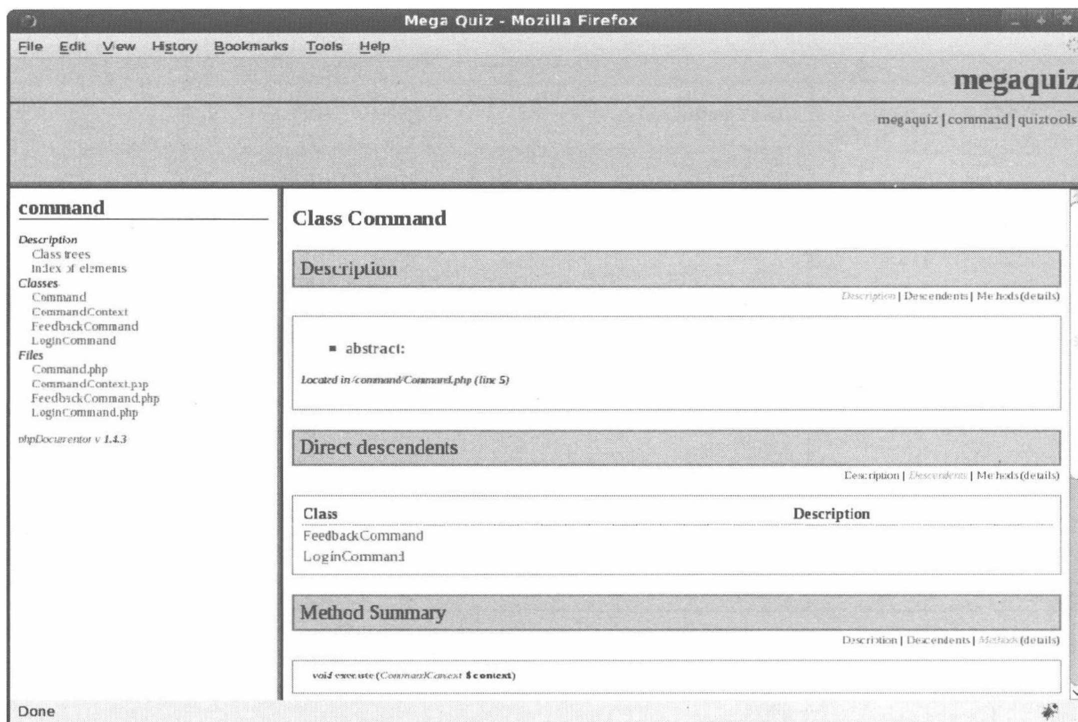


图16-3　识别@package标签后输出的文档

通常来说，文档中的包与目录结构是一一对应的，因此command包与command目录相对应。然而这并不是必需的。例如，第三方开发者可能希望创建一个Command类来作为command包的一部分，但是该类放在她自己指定的目录中，而不是放在command目录中。因此@package标签让你负责指定类和包的关系，但同时也可以让你不使用文件系统即可猜测包名。

16.5　类的文档

下面增加更多的标签和文字，以便用于类或文件级别的DocBlock。我们将标识一个类，解释它的用法并增加作者和版权信息。

下面是Command类的全部代码：

```
/**
 * 定义命令的核心功能
 * Command类通过execute()方法在系统中执行特定任务
 * the execute() method
 *
```

```
 * @package command
 * @author  Clarrie Grundie
 * @copyright 2004 Ambridge Technologies Ltd
 */
abstract class Command {
    abstract function execute( CommandContext $context );
}
```

DocBlock注释有了明显的改进。第一个句子是一个单行的摘要，它在输出文档中被重点强调，并且被提取到“概述”中。后面几行的文字包含了更详细的描述，在此你可以向后来的程序员提供详细的用法。可以看到，这一部分可以包含指向项目中其他元素的链接、描述性文字和代码片段。我们还可以包含`@author`和`@copyright`标签，分别用于指定作者和版权。你可以在图16-4中看到扩展类注释后的效果。

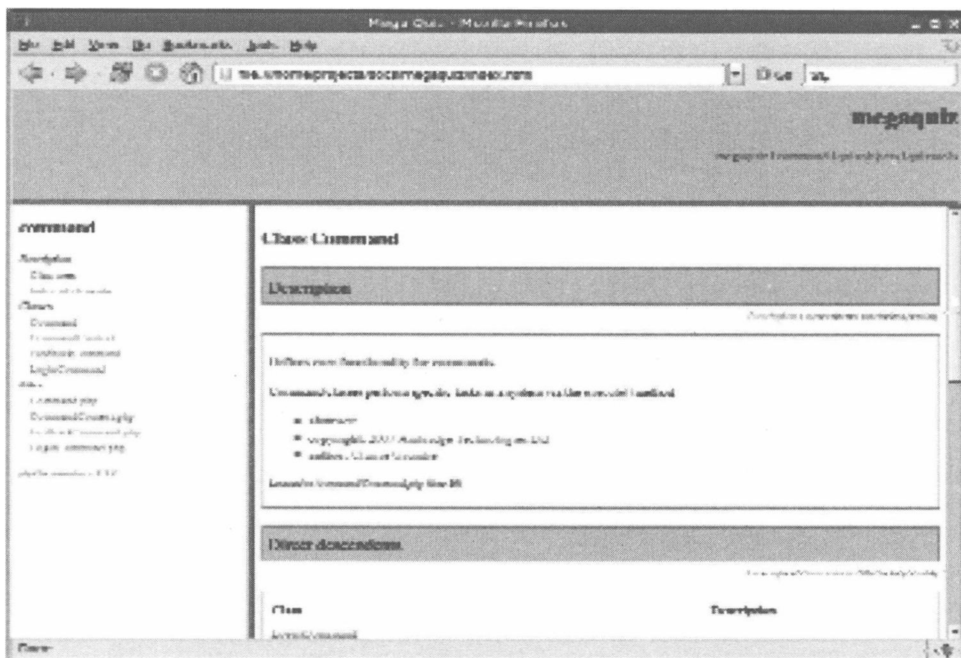


图16-4　类在文档输出中的情况

注意我们并不需要告诉phpDocumentor Command是抽象类。实际上，phpDocmentor并不需要我们的帮助就可以自动完成对类的检查。但是要注意DocBlock是上下文相关的。phpDocumentor知道我们生成了图中显示的那个类的文档，因为DocBlock位于类的声明之前。

注解　*在写作本书时，phpDocumentor并不支持命名空间。但有相关记录显示，项目的维护程序Greg Beaver可以提供此功能（<http://lists.bluga.net/pipermail/ phpdocumentor-devel/2008-September/000066.html>）。*

16.6　文件的文档

虽然我倾向于把每个文件都看成是一个类，但是在一些项目中需要提供文件级别的文档。

首先，phpDocumentor要求提供文件注释。如果没有提供对当前文件的DocBlock，跳出的警告将会打乱程序的出错报告，尤其是在大项目中。文件的注释应当是文档中第一个DocBlock。它必须包含@package标签，后面不能直接跟代码。也就是说，如果增加文件级别的DocBlock，那么应该在第一个类声明前添加一个类级别的注释。

许多开源项目要求每个文件都包含许可公告或指向许可的链接。因此，为了不在每个类中都包含基本的许可信息，可以使用页面级的DocBlock注释。为此，我们可以使用@license标签。@license之后是一个指向许可文档的URL和描述性文字：

```
/**
 * @license http://www.example.com/lic.html Borsetshire Open License
 * @package command
 */
```

license标签中的URL在phpDocumentor输出中将变成可单击的超链接。

16.7　属性的文档

PHP中的属性类型是混合的，即一个属性能潜在地包含任何类型的值。在某些情况下，我们可能需要这种灵活性，但是大多数时候，属性应该是一个特定的数据类型。phpDocumentor允许使用@var标签来为属性生成文档。

下面为CommandContext类中的部分属性添加文档：

```
class CommandContext {
/**
 * 应用程序的名称
 * 用于不同的客户端程序来生成不同的出错信息（或其他用途）
 * @var string
 */
    public $applicationName;

/**
 * 封装好的键/值对
 * 本类本质上是对本数组的封装
 * @var array
 */
    private $params = array();

/**
 * 出错信息
 * @var string
 */
    private $error = "";
// ...
```

如你所见，我们为每个属性各提供了一个摘要性的说明，其中前两个属性的信息更为详细。

我们使用@var标签来定义每个属性的类型。如果使用像平常那样的phpdoc命令行参数生成这一部分的输出，将只能看到$applicationName公有属性的文档。这是因为私有方法及属性默认情况下不会出现在文档中。

你可以自己决定是否对私有元素生成文档，而这取决于文档是给谁看的。如果是为客户程序员生成文档，那么可能应该隐藏类的内部细节。如果项目正处于开发之中，那么团队成员可能需要更详细的文档。你可以通过使用-pp（--parseprivate）命令行参数使phpDocumentor包含私有元素：

```
phpdoc -d /home/projects/megaquiz/ \
       -t /home/projects/docs/megaquiz/ \
       -ti 'Mega Quiz' \
       -dn 'megaquiz' \
       -pp on
```

注意必须明确设置-pp的值为on，只使用-pp是不够的。从图16-5中可以看到各属性的文档。

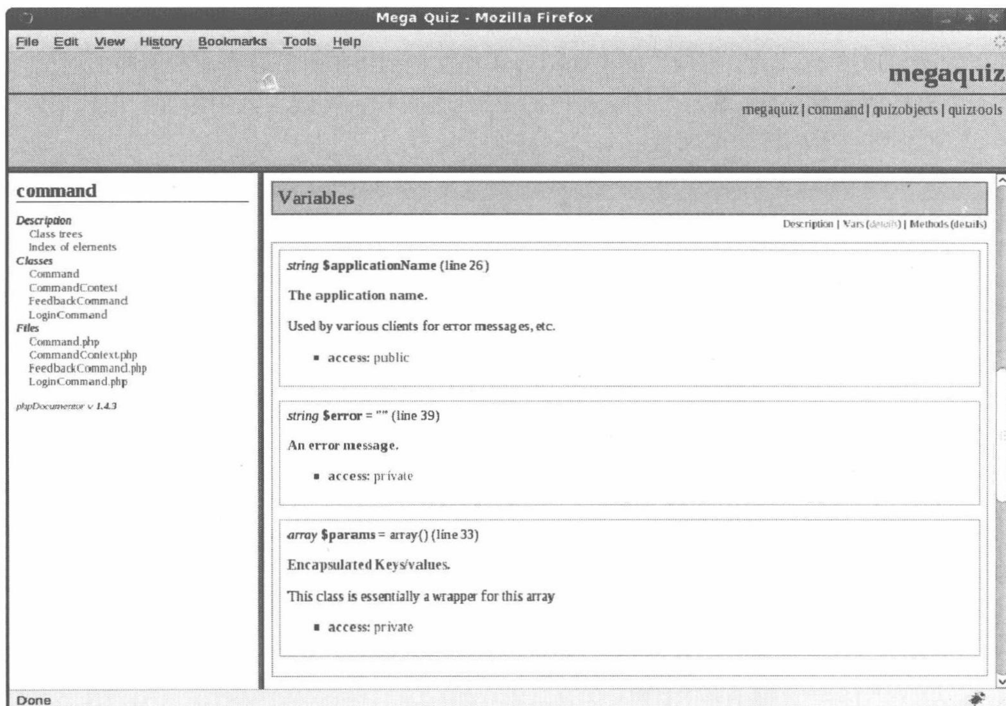


图16-5 属性的文档

16.8 方法的文档

方法的文档和类的文档是项目文档的核心部分。至少，读者需要理解类方法的参数、执行的操作和它的返回值。

和类级别的DocBlock注释一样，方法的文档应该由两部分文字组成：一行摘要和一个可选的描述。可以用@param标签提供方法每个参数的信息。每个@param标签应该以新行开始，后跟参数名称、类型和简要描述。

因为PHP没有限制返回类型，所以为方法的返回值添加文档非常重要。你可以使用@return标签来达到这个目的。@return应该以新行开始，后面跟着返回值的类型和简要描述。完整的注释如下：

```
/**
 * 执行当前类所封装的核心操作
 * Command类封装了一个单一的操作。这些类
 * 可以很方便地添加到项目中，也可以很方便地删除
 * 还可以在初始化及execute()调用后被轻松保存
 * @param  $context CommandContext Shared contextual data
 * @return bool执行失败后返回false，成功时返回true
 */
   abstract function execute( CommandContext $context );
```

添加比代码内容还多的文档看起来可能会有点奇怪。但是抽象类的文档特别重要，因为它为程序员理解如何扩展类提供了方向。如果你担心添加文档后PHP引擎必须浪费解释的时间，那么可以在安装程序时删除注释，这是一件很简单的事情，你可以向构建工具中添加一段代码来完成。在图16-6中，你可以看到输出的文档。

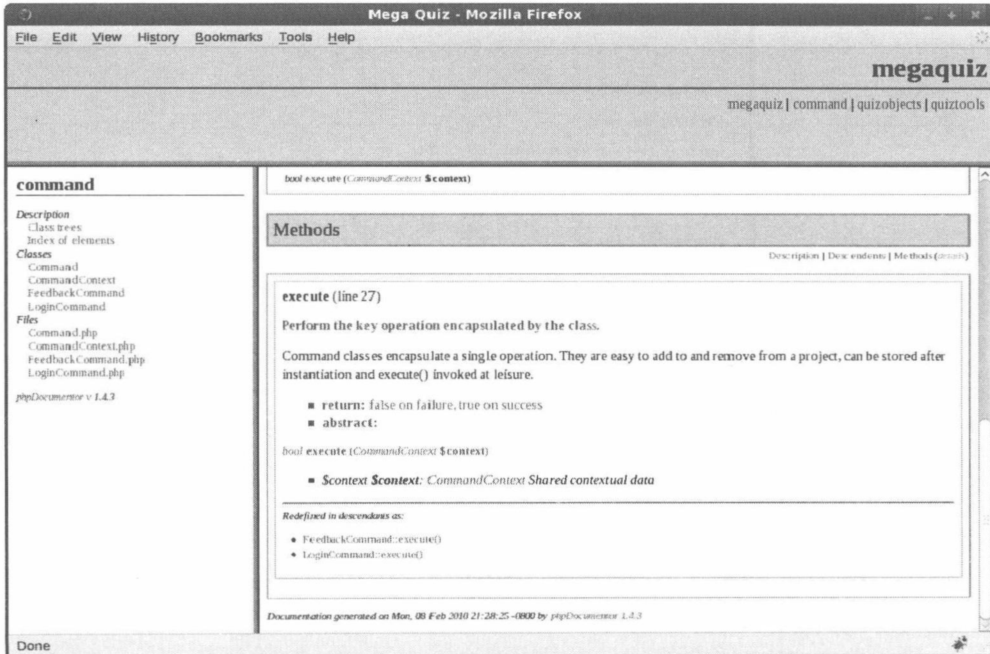


图16-6　方法的文档

16.9 在文档中创建链接

phpDocumentor生成的文档带有超链接。但是有时你会想要生成自己的超链接，用于指向文档的其他元素或外部的站点。在本节中，我们将看到用于完成此任务的标签，并接触一个新的语法：行内标签（inline tag）。

当你写一个DocBlock注释时，可能希望能够关联到相关的类、属性或方法。要实现这一特性，可以使用@see标签。@see需要一个对元素的引用，而且格式必须是这样的：

```
class
class::method()
```

或者

```
class::$property
```

因此在下面的DocBlock注释中，我们为CommandContext对象生成文档，并强调它通常在Command::execute()方法中使用：

```
/**
 * 封装数据以供在命令对象间传递
 * Command对象需要根据环境得到不同的数据
 * CommandContext对象会作为参数传递给Command::execute()
 * 其中包含键/值对形式存在的数据
 * Command类会自动得到超全局变量$_REQUEST中的数据
 *
 * @package command
 * @author  Clarrie Grundie
 * @copyright 2004 Ambridge Technologies Ltd
 * @see Command::execute()
 */

class CommandContext {
// ...
```

如图16-7所示，@see标签被解析为超链接。单击该链接，将会跳到Command类的execute()方法。

注意，我们也可以在DocBlock描述文本中嵌入对Command::execute()的引用。我们还可以用@link标签将其转换为链接。你可以像使用@see一样在一行的开头加入@link，也可以在行内使用它。为了从周围的环境中识别出行内标签，必须用大括号把行内标签括起来。因此，要使指向Command::execute()的链接可用，应使用下面的语法：

```
// ...
 * Commands对象需要根据环境得到不同的数据
 * CommandContext被传递给{@link Command::execute()}
 * 方法，其中包含着键/值对形式存在的数据
 * Command类会自动得到超全局变量$_REQUEST中的数据
//...
```

16

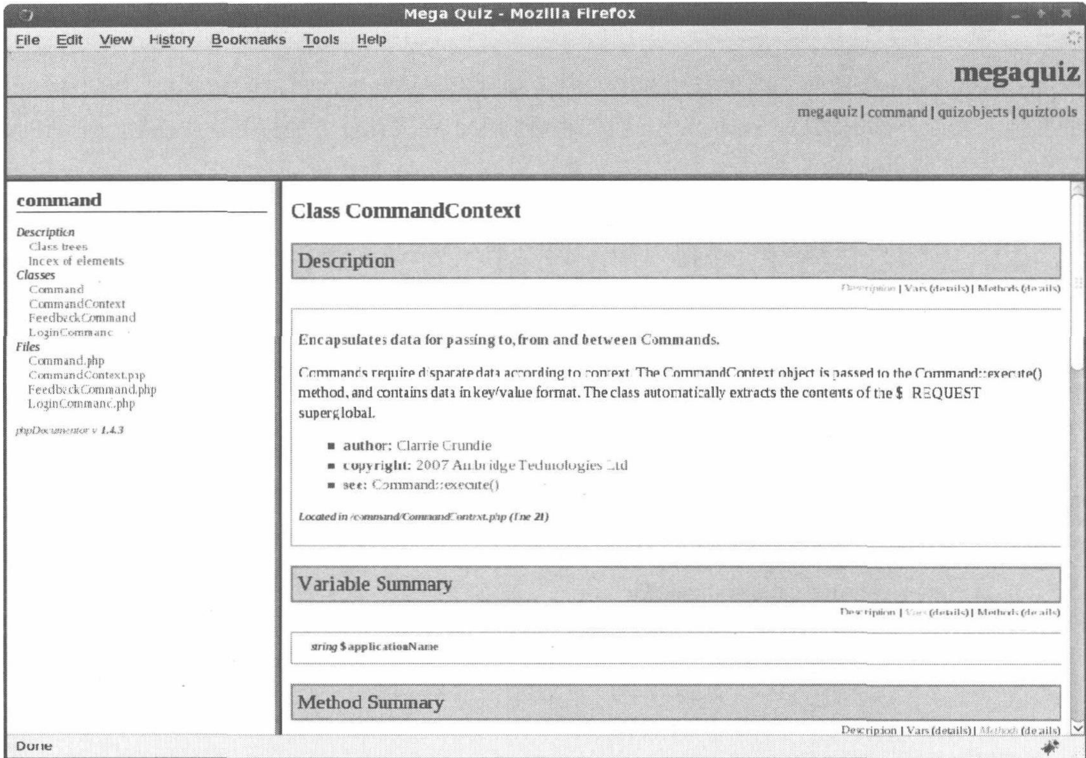


图16-7　用@see标签创建链接

代码中`@link`标签只包含对元素（`Command::execute()`）的引用，这个字符串将变成可单击的。如果在此增加一些文字描述，则文字描述也将会变成可单击的。

`@link`也可被用于指向URL。只要将元素引用替换为URL即可：

```
@link http://www.example.com More info
```

URL再次成为超链接指向的地址，其后的描述则变成可以单击的文字。

有时你可能想要生成一个反向链接。`Command`使用`CommandContext`对象，因此我们可能想创建一个从`Command::execute()`到`CommandContext`类的链接，同时也想创建一个从`Command-Context`类到`Command::execute()`的链接。当然，你完全可以使用两个`@link`或两个`@see`标签来达到目的。但如果使用`@uses`标签，则只需一个`@uses`标签就可以处理双向链接：

```
/**
 * 执行类中封装的核心操作
 * ...
 * @param  $context {@link CommandContext} Shared contextual data
 * @return bool执行失败时返回false，成功时返回true
 * @link http://www.example.com More info
 * @uses CommandContext
```

```
 */
    abstract function execute( CommandContext $context );
```

通过增加@uses标签，可以在Command::execute()文档中创建一个链接：Uses:CommandContext。而在CommandContext类文档中，将出现一个新链接：Used by:Command::execute()。

如图16-8所示，你可以看到最新的输出结果。注意我们没有在行内使用@link，因此输出的URL是单行的列表项。

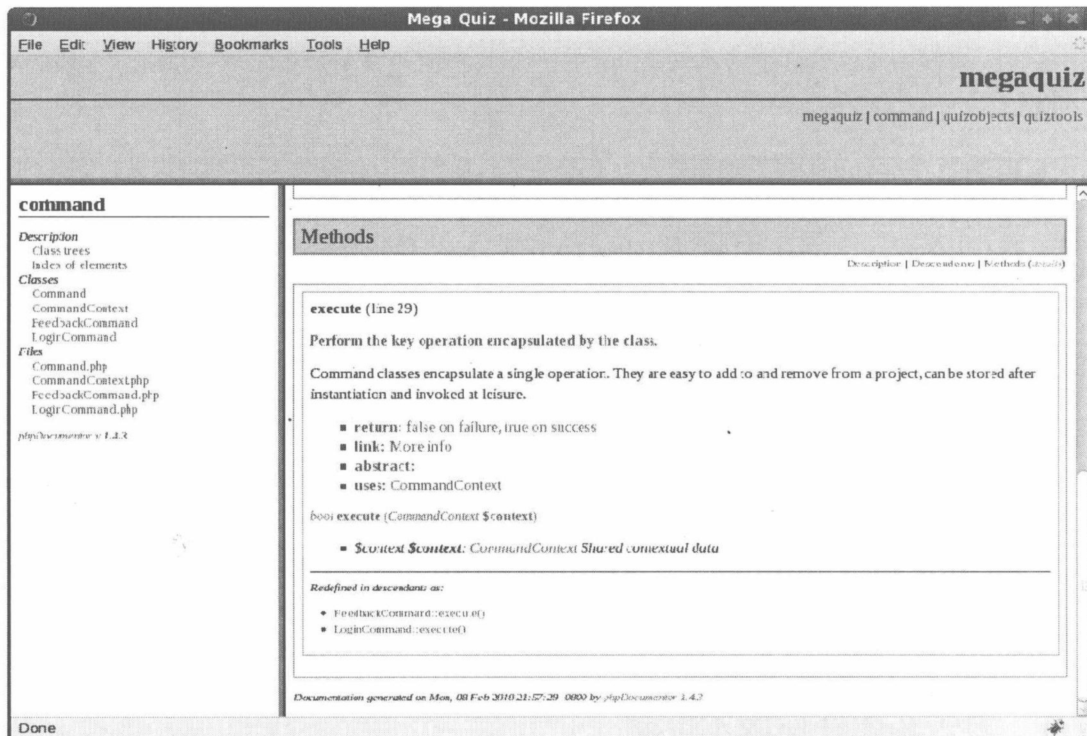


图16-8 包含@link和@uses标签的文档

16.10 小结

本章介绍了phpDocumentor的核心特性，探讨了DocBlock注释语法以及可以与它一起使用的标签，讨论了如何为类、属性和方法添加文档，并且提供了足够的标签来转换文档，这对于改善团队协作意义重大（特别是结合使用构建工具和版本控制工具时）。除了本章介绍的内容之外，phpDocumentor这个奇妙的工具还有更多功能，你可以通过phpDocumentor的官方站点http://www.phpdoc.org来了解。

第 17 章

使用Subversion进行版本控制

所有灾难都有它们爆发的临界点，在秩序被打破的那一刹那，一切都将失去控制。你的项目是否有类似的经历？你能够发现那关键的一刻吗？原因很多：可能那一刻你只是进行一次“小变动”，结果发现周围的一切忽然崩溃了（更糟糕的是，你可能不知道如何恢复原状）；可能因为项目组里的3个人同时在编辑某组类，并且“好心地”在别人的代码上添加了新内容；也可能是因为给某个bug打过的两次补丁代码又莫名其妙地消失了。我们需要一种用于管理协作开发的工具，它允许你捕捉项目快照[①]，在必要的时候将代码恢复到上一版本，而且可以合并开发的多个分支的代码。在本章中，我们将介绍Subversion，它可以完成以上这些工作，甚至更多。

本章包括如下内容。

- 基本配置：安装Subversion的一些技巧。
- 导入：开始一个新项目。
- 提交修改：将工作成果保存到代码库。
- 更新：将别人的工作成果和自己的工作成果合并。
- 分支：平行维护项目的几份代码。

17.1 为什么要使用版本控制

如果以前没有用过版本控制，相信版本控制将会改变你的生活（如果你是一名程序员）。我们常常非常辛苦地让项目达到一个稳定状态，接着又要开始新一轮的忙乱。当你在开发的时候，很难恢复到之前的稳定版本。当然，你也可以在项目稳定时通过复制整个开发目录来保存一份快照代码。假设你的同事在同一个代码库中工作。他可能和你一样保存了一份代码稳定的副本。不同的是，他的副本是他的代码的快照，而不是你的。当然，他也有一个凌乱的开发目录。因此，你的项目可能有4个版本需要协调。如果一个项目有4个程序员和一个Web UI 设计师，你是不是会苦恼得想躺下来休息？

Subversion专门负责解决这类问题。使用Subversion，所有的开发人员都可以从中央代码库检

① snapshot，即取得项目在某个时刻的代码。——译者注

查他们自己的副本。无论他们的代码何时达到稳定，他们都可以更新手头的副本，把自己最近的工作成果和公共代码中的改变合并起来。在他们解决了可能存在的冲突之后，就可以把新的稳定版本上传到中央代码库中。现在，在你的项目中只有一份正式的源代码。每个开发者都可以将自己的代码合并到中央代码库中，这意味着你不再需要手动协调多条开发线。更妙的是，你可以根据日期或标记来检查代码库的代码。因此当你的代码达到稳定时，可以一边继续编写代码，一边向客户演示你的代码。例如，你可以使用任意的标签来标记已经稳定的代码版本。当客户忽然来到办公室查看进展时，你可以使用该标签来获取正确的代码。

Subversion还有更多功能。你可以同时管理多个开发版本。假设有一个成熟的项目，你已经发布了第一个版本，正在开发第2版，那么1.*n*版本可以被抛弃吗？当然不是，用户还在不停地发现bug并且请求修正。你可能还要过几个月才发布第2版，那么应该在哪一个版本中修改代码并测试呢？Subversion可以让你维护不同的代码库分支。因此，你可以在当前产品代码上为1.*n*版本的代码创建一个用于修正bug的分支。很关键的一点是，这个分支可以和版本2的主干代码合并在一起，因此新发布的版本（第2版）也将可以从1.*n*版的改进中受益。

注解 Subversion并不是唯一可用的版本控制系统。你也可以使用Git(http://git-scm.com/)或Mercurial(http://mercurial.selenic.com/)。Git和Mercurial是时新且逐渐流行的版本控制系统，都使用分散模型。

下面让我们在实践中学习Subversion的这些功能。

17.2 获得 Subversion

如果使用类似Unix的操作系统（比如Linux或FreeBSD），那么系统可能已经安装好Subversion客户端。

在命令行输入：

```
$ svn help
```

你可以看到一些使用信息，告诉你如何使用Subversion。如果你还没有安装Subversion，请参考发布文档。一般来说，你可以通过简单的安装机制（比如Yum或Apt）很方便地安装Subversion。

注解 本章所有的命令行输入都使用粗体来显示。美元符号（$）表示命令行提示符。

如果你得到一条错误信息，那么需要自行下载和安装Subversion。在这里可以获得Subversion源代码和二进制包：http://Subversion.apache.org/。

注解 如果你使用GUI而非命令行，那么应该看一下RapidSVN，它是Subversion的跨平台前端，可以在http://rapidsvn.tigris.org/上找到它。如果你是Windows用户，还应该评估一下TortoiseSVN（http://tortoisesvn.tigris.org/）。

17

17.3　配置Subversion代码库

无论你是在本地安装Subversion服务器，还是在远程服务器上安装并开放给多个客户端来访问，你都需要建立一个代码库。每个Subversion客户端都需要知道该代码库的位置。在本节中，我们将学习建立Subversion的必要步骤，无论是在单独的机器上还是通过因特网。首先假设你已经拥有Linux机器的root账号。为创建和管理一个代码库，我们需使用svnadmin工具。

创建代码库

可以用一条简单的Subversion命令来创建Subversion代码库：create。它将创建一个良好配置的Subversion代码库目录。

下面我们在/var/local/svn目录下创建了一个代码库。一般来说，只有root用户才可以创建和修改/var/local下的目录，因此我们要以root身份运行下面的命令：

```
$ svnadmin create --fs-type fsfs /var/local/svn
```

命令执行之后没有输出任何提示信息，但是你可以发现它在/var/local目录中创建了一个名为svn的目录。在这里fs-type标志并不是必需的，因为是fsfs是默认设置。这条指令要求Subversion使用文件存储版本信息。可选的bdb指定Berkeley DB管理这一数据。

我们假设这台Linux机器中有多个用户，他们都需要提交和更新代码库。我们需要确保他们都有对/var/local/svn目录的写权限。这可以通过增加这些用户到一个共同用户组来实现，使该用户组拥有/var/local/svn目录的写权限。

你可以用下面的命令新建一个用户组（组名为svnusers）：

```
$ groupadd svnusers
```

你必须以root的身份运行groupadd命令，执行完之后系统中就有了这个新的用户组。

首先，把用户bob添加到svnusers组中。你可以通过文件/etc/group来管理用户组。在/etc/group中，可以看到这样一行命令：

```
$ svnusers:x:504:
```

我们可以用下面的命令把bob增加到组中：

```
$ usermod -aG svnusers bob
```

现在，如果你再次查看/etc/group，会发现bob已经和svnusers组关联起来。

```
$ svnusers:x:504:bob,
```

接下来，我们要让svnusers组中的任意用户都可以写var/local/svn。通过把var/local/svn目录的用户组改为svnusers，就可以达到我们的目的：

```
$ chgrp -R svnusers /var/local/svn/
$ chmod -R g+rws /var/local/svn/
```

代码的最后一行可以使以后在svn目录下创建的目录都属于svnusers用户组。

为了访问Subversion库中的项目，必须使用URL来指定它的位置。现在我准备省点事，假设我已经创建了一个名为megaquiz的项目。要介绍的另一个重要内容是名为list（简写为ls）的命令，该命令列出库中某个位置的所有文件：

```
$ svn ls file:///var/local/svn/megaquiz
```

Subversion URL与你输入到浏览器地址栏中的内容非常相似。Subversion URL一开始是一个模式（你想要进行连接的连接类型），可能是一个服务器名，后跟一个路径（其中包括库的位置，后接任意数量的项目目录）。因为上面的代码段在其模式中指定了文件系统，所以无需提供服务器。

假设数据库正在运行sshd，并且已正确配置了防火墙，那么也可以使用ssh通过远程机器访问同一个库：

```
$ svn ls svn+ssh://localhost/var/local/svn/megaquiz
```

Subversion还支持若干其他通信方式。根据库服务器的配置方式，也可以使用WebDav协议（http或https模式）或svn网络协议（svn模式）。现在我已经在服务器上建立了Subversion库，因此就使用ssh（更确切地说，是建立在ssh上的svn协议，也就是svn+ssh的复合模式），除了接下来将讨论的一些问题，ssh还算是一种易于使用、安全的通信机制。配置Subversion以便与SSH结合使用是非常简单的。如果你的服务器配置为可以接受ssh连接，那么就可以像上面那样访问Subversion库。但是还有几件令人头疼的事。例如，如果事先没有在Subversion机器上为用户分配shell账号，那么用户就很难拥有对Subversion库的完整访问权限。如果想允许信不过的用户访问代码库，可以设置chroot jail，chroot jail对用户账号有极其严格的限制。这些有关系统管理的内容与本章主题已经相差太远了！更简单的解决方案是，禁用任何你不希望其拥有命令行访问权限的用户的登录访问。你可以在创建用户账号时执行这项操作。查看adduser命令的帮助页面，了解详细信息。

不断地为每个Subversion命令输入密码或者加密短语（pass phrase），这也是令用户很烦的一件事。我已经在Subversion机器上设置了用户bob，所以他可以使用svn+ssh模式远程访问代码库。那么怎么才能让他能更容易验证自己呢？SSH配置的细节超出了本书的讲述范围。

注解 Michael Stahnke撰写的*Pro OpenSSH*（Apress，2005）全面地介绍了SSH。

简言之，Bob应该使用ssh-keygen程序在其客户机上生成一个公钥。他会被提示创建加密短语。他应该在.ssh/id_rsa.pub（其中.ssh在客户机主目录中）中找到该公钥，并复制这个公钥，将其附加到Subversion服务器上名为.ssh/authorized_keys（其中.ssh位于主目录中）的文件。

17.4 开始项目

要在一个项目上使用Subversion，必须将该项目加入代码库中。这可以通过导入该项目的目录和任何已有的内容来实现。

17

导入之前，要先仔细查看自己的文件和目录，删除临时内容。查找临时内容通常是很麻烦的事。要查找的临时内容包括自动生成的文件（比如phpDocumentor输出的文件）、构建目录和安装日志等。

注解　通过编辑Subversion在主目录中创建的.subversion/config中生成的配置文件，可以指定在导入、提交和更新过程中要忽略的文件和模式。查找选项global_ignores，该选项可能需要被去掉注释。该选项提供了文件名通配符示例，你可以修改这些示例来排除由构建过程、编辑器和IDE创建的各种锁文件和临时目录。

在项目明确之后，你应该考虑一下如何组织版本。使用Subversion可以管理项目的多个版本，可以轻松分支项目以创建新版本，然后再将修改合并回其出处。虽然你可以随心所欲地组织版本，但很多开发人员都遵循一些约定。最具代表性的是，开发过程中要坚持一条开发主线作为所有分支的来源和最终目标。这一分支称为主干（trunk）。实际上，由于Subversion的灵活性，该分支只是一个名为trunk的目录。你还需要一个目录来保存分支。如果你遵循这些约定，可以将该目录命名为branches。最后，你可能需要一个位置来保存快照分支。快照分支在本质上与其他分支没有什么不同，但它们的目的是提供项目开发过程中某一特定时刻的快照，而不是平行开发的一个站点。快照分支应该保存到tags目录中。

导入后，你可以在代码库中任意移动目录，但由于我在导入时知道我想要什么，最好还是先设置目录结构。如果我的目录结构如下所示：

```
megaquiz/
    quiztools/
    commands
    quizobjects/
```

我可能会在导入前增加另一层目录，因此最终的目录结构如下所示：

```
megaquiz/
    branches/
    tags/
    trunk/
        quiztools/
        commands
        quizobjects/
```

可以看到，branches和tags并没有什么神奇之处，它们只是普通的目录。一切就绪以后，就可以导入项目了：

```
$ svn import megaquiz svn+ssh://localhost/var/local/svn/megaquiz
```

接下来对Subversion命令的用法进行讲解。Subversion是一个非常大的包，由很多子命令和开关语句构成。import接受指向服务器上新目录的URL参数。目录名实际是项目名。可以看到，import子命令也接受你想要导入的目录的路径作为参数。如果你没有指定该路径，那么Subversion会导入当前的工作目录。

17.5 更新和提交

本章虚构了一个名叫Bob的团队成员。Bob和我们一起开发MegaQuiz项目，他是一名很好的开发人员，但有一个非常常见又令人讨厌的毛病：他总是插手其他人的代码。

Bob聪明且好奇心强，容易为新的开发方式而激动，而且热衷于优化新代码。这样的结果就是，在项目开发的每个地方都能看到Bob的身影：Bob添加了文档；Bob实现了一个我们闲聊时的提议。我们都对Bob感到很头疼，却又不得不面对即成事实，必须将自己的代码和Bob所做的工作合并在一起。

下面的文件是quizobjects/User.php，除了光秃秃的骨架之外，里面什么都没有：

```
<?php
class User {}
?>
```

我们决定添加一些文档。就像上一章所说的那样，应该增加文件和类的注释。首先增加文件注释到文件中：

```
<?php
/**
 * @license   http://www.example.com Borsetshire Open License
 * @package   quizobjects
 */

class User {}
?>
```

与此同时，像以前那样，Bob在自己的沙盒中创建了类注释：

```
<?php

/**
 * @package   quizobjects
 */
class User {}
?>
```

因此现在我们有了两个不同版本的User.php。此时，Subversion代码库只包含了最近导入的MegaQuiz版本。我们决定将我们的修改添加到Subversion代码库中。这只需要一条命令，但建议用两条命令：

```
$ svn update quizobjects/User.php
```

```
At revision 1.
```

Update子命令通知Subversion将代码库中保存的修改合并到本地文档中。在我们提交自己的工作之前，最好先查看一下其他人的修改是否和我们自己的修改有冲突，并在我们自己的沙盒中解决冲突。这个命令的输出表明没有第三方更改要应用。

17

运行update会将文件库版本的任何修改都应用到本地副本。如果省略了文件路径，将对当前位置下的所有文件执行该操作。

你可能希望在本地合并修改之前知道对哪些文件进行了修改。为此可以使用status子命令：

```
$ svn status --show-updates
```

运行该命令会得到更新能够在本地影响到的文件的列表。

不管选择哪个子命令，现在我可以继续进行并确认修改。

```
$ svn commit quizobjects/User.php  -m 'added doc level comment'
```

```
Sending        quizobjects/User.php
Transmitting file data .
```

我使用commit子命令将新数据提交到Subversion库中。注意，我使用-m开关在命令行中添加了一条消息，而不是通过编辑器添加的。

现在轮到Bob更新并提交了：

```
$ svn update quizobjects/User.php
```

```
Conflict discovered in 'quizobjects/User.php'.
Select: (p) postpone, (df) diff-full, (e) edit,
        (mc) mine-conflict, (tc) theirs-conflict,
        (s) show all options:
```

只要修改不会重叠，Subversion就会将两个源的数据合并到同一个文件中。Subversion不知道怎么处理对相同代码行的修改。优先处理哪个修改呢？是让代码库覆盖Bob的修改还是相反？两个修改可以并存吗？应该先接受哪一个？Subversion没法选择，它只能报告这个冲突并让Bob解决这个问题。遇到冲突时，系统会给Bob列出很多令人费解的选项。Subversion其实提供了对这些选项的解释。如果Bob按了s键：

```
        (s) show all options: s
  (e)  edit             - change merged file in an editor
  (df) diff-full        - show all changes made to merged file
  (r)  resolved         - accept merged version of file
  (dc) display-conflict - show all conflicts (ignoring merged version)
  (mc) mine-conflict    - accept my version for all conflicts (same)
  (tc) theirs-conflict  - accept their version for all conflicts (same)
  (mf) mine-full        - accept my version of entire file (even non-conflicts)
  (tf) theirs-full      - accept their version of entire file (same)
  (p)  postpone         - mark the conflict to be resolved later
  (l)  launch           - launch external tool to resolve conflict
  (s)  show all         - show this list
```

遇到冲突时，你首先想做的可能是弄清楚发生了什么。dc选项会给出答案，该选项会显示所涉及文件的冲突部分。选择dc:<?php时，Bob会看到如下内容：

```
<<<<<<< MINE (select with 'mc') (2,4)
/**
 * @package  quizobjects
 */
||||||| ORIGINAL (2,0)
=======
/**
 * @license   http://www.example.com Borsetshire Open License
 * @package  quizobjects
 */

>>>>>>> THEIRS (select with 'tc') (2,5)
class User {}
?>
```

Subversion包括Bob的注释和导致冲突的修改，以及可以告诉Bob哪部分冲突源自哪里的元数据。冲突信息由一行等号分开。Bob的输入由一行小于号后接MINE来表示。来自代码库的数据由一行大于号后接THEIRS表示。

既然Bob已经看到了这个冲突，他可以执行相关操作来处理这个冲突。从前面的输出可以看到，Bob可以接受库版本：rc只显示冲突，并就地保留他的无争议的修改；rf可以使用服务器上的文档覆盖他的整个文档。他可以覆盖库版本：mc会强制使用只有冲突的版本，mf会用他自己的版本覆盖整个库版本。他可以选择推迟操作，这会使文档被本地标记为冲突，直到对该文件运行svn resolve。但是，他最有可能选择e选项，并手动解决冲突。在本例中，他删除了元数据，并把内容按顺序排好。

```
<?php
/**
 * @license   http://www.example.com Borsetshire Open License
 * @package  quizobjects
 */

/**
 * @package  quizobjects
 */
class User {}
?>
```

保存修改并关闭了编辑窗口之后，Bob仍需确认他编辑的内容，即选择r，以便最终能解决这个冲突。即使在这种情况下，这些修改还是没有提交。Bob必须显式地提交这个修改的文件，以便将修改保存到代码库中。这里有一个重要的原则。update是从代码库到本地版本自上而下地工作。只因为检测到冲突而要修改这个流程是不可能的。

```
$ svn commit -m 'added class comment' quizobjects/User.php
```

```
bob@localhost's password:
Sending        quizobjects/User.php
Transmitting file data .
Committed revision 3.
```

到现在为止，Bob和我都只更新并提交了同一个文件。在忽略文件参数的情况下，可以将这些命令应用到项目中的每个文件和目录。下面从项目的根目录运行update：

```
$ svn update
```

```
U    quizobjects/User.php
Updated to revision 3.
```

Subversion会访问项目中的每个目录，找到User.php文档并更新它的内容。于是，Bob的修改就会合并到我的文档中。

我们也可以通过相同的方式提交代码。在这个例子中，我们在两个文件command/Command.php和 quiztools/AccessManager.php中做了一些小修改：

```
$ svn commit -m'documentation amendments'
```

```
Sending        command/Command.php
Sending        quiztools/AccessManager.php
Transmitting file data ..
Committed revision 4.
```

Subversion再次遍历当前工作目录下的所有目录。它在遇到一个修改过的文件前什么也不做。遇到一个修改过的文件后，对比代码库，找出修改的内容。

17.6　增加和删除文件及目录

项目包含的内容会随着开发过程不断变化。版本控制软件必须考虑到这一点，允许用户添加新文件以及删除无用的文件。

17.6.1　添加文件

使用add命令可以添加新文件到Subversion。下面我们添加Question.php文件到项目中：

```
$ touch quizobjects/Question.php

$ svn add quizobjects/Question.php
```

```
A         quizobjects/Question.php
```

在实际开发中，可能要先增加一些内容到Question.php中。在这里我们使用touch命令来创建一个空文件。根据add命令的输出结果，我们知道添加一个文件后，必须调用commit命令来完成添加。

```
$ svn commit -m'initial checkin'
```

```
Adding         quizobjects/Question.php
Transmitting file data .
Committed revision 5.
```

Question.php现在被加入到代码库中了。

17.6.2 删除文件

如果想要删除一些文件，可以使用命令remove。

```
$ svn remove quizobjects/Question.php
```

```
D         quizobjects/Question.php
```

你同样需要使用commit命令来完成这个工作。

```
$ svn commit -m'removed Question'
```

```
Deleting       quizobjects/Question.php
Committed revision 6.
```

17.6.3 添加目录

我们也可以使用add和remove命令来添加和删除目录。假设Bob想添加一个新目录：

```
$ mkdir resources
$ touch resources/blah.gif
$ svn add resources/
```

```
A         resources
A         resources/blah.gif
```

这里需要注意一点，即resources的内容是被怎样自动添加到代码库的。

17.6.4 删除目录

就像你预期的那样，remove命令可以用来删除目录。假设我们想要删除resources目录：

```
$ svn remove resources/
```

```
D         resources/blah.gif
D         resources
```

注意，这个子命令是递归执行的。为了应用修改，还需要提交修改。

17.7 标记和导出项目

经过一段时间的开发，项目终于达到了一个稳定的状态，现在你可能想对项目进行发布或部署。Subversion有两种方法可以帮助你。第一，可以生成一个不包含Subversion元数据的项目版本。第二，可以标记项目开发的这个时刻，以便于以后可以返回这个状态。

17.7.1 标记项目

其他版本控制系统在命令级别内置了标签的概念，但在Subversion中，标签实际上就是一个

副本。它没有任何特殊的地方。是作为快照、还是作为参考副本，都是用户约定的问题。还记得我第一次导入项目时创建的目录吗？也就是trunk目录，它是工作主目录。此外，还有branch和tags目录。要创建标签，只需让Subversion把当前项目复制到tags目录即可。

那么如何实现呢？我检查了trunk目录，所以现在没有可用的其他本地目录。实际上，我可以命令Subversion在库中创建这个副本。

```
$ svn copy svn+ssh://localhost/var/local/svn/megaquiz/trunk  \
   svn+ssh://localhost/var/local/svn/megaquiz/tags/megaquiz-release1.0.0 \
   -m'release branch'
```

```
Committed revision 9.
```

因为此处我只处理URL，严格来讲这是一项服务器操作，所以以这种方式运行copy命令之前，你要确保已提交了所有想包括的内容。你也可以在工作副本中进行复制，并提供文件路径而不是URL。这个操作的代价可不小，需要在本地维护标签和分支。注意，在执行copy操作时也同时给标签指定了名字。我已将trunk复制到了tags/megaquiz-release1.0.0，可以使用list命令再次确认：

```
$ svn list svn+ssh://localhost/var/local/svn/megaquiz/tags/
```

```
megaquiz-release1.0.0/
```

现在，使用checkout命令可以随时获取该快照。然而，通常不希望标签成为并行开发（相关内容参考本章介绍分支的那一节）的基础，但你可能想导出添加了标签的副本，为打包做准备。

17.7.2　导出项目

可以看到，从代码库中检出代码时，项目中包含管理目录（一般目录名是svn）。取决于你的配置方式和打包工具，这些目录对于你的项目来说可能是无用的，应该在发布项目时删除。Subversion提供了export命令来导出项目代码的干净版本。

```
$ svn export svn+ssh://localhost/var/local/svn/megaquiz/tags/megaquiz-release1.0.0 \
   megaquiz1.0.0
```

```
A    megaquiz1.0.0
A    megaquiz1.0.0/quizobjects
A    megaquiz1.0.0/quizobjects/User.php
A    megaquiz1.0.0/quiztools
A    megaquiz1.0.0/quiztools/AccessManager.php
A    megaquiz1.0.0/main.php
A    megaquiz1.0.0/command
A    megaquiz1.0.0/command/Command.php
A    megaquiz1.0.0/command/CommandContext.php
A    megaquiz1.0.0/command/FeedbackCommand.php
A    megaquiz1.0.0/command/LoginCommand.php
```

导出的修订9。export的第一个参数指定了源，在本例中是我在上一节创建的标签。第二个

参数指定了在需要时Subversion将创建的目标目录。

17.8 创建项目分支

现在我们的项目已经发布了，似乎不再需要对它进行改进了，是吗？毕竟，它编写得那么优雅，基本上没有bug，而且系统很有针对性，用户不再需要新功能。

但是，回到现实中来，我们还得继续写代码。这至少有两个原因：bug报告会源源不断地到来；版本1.2.0要添加的新功能清单也在不断膨胀。我们如何解决这些问题呢？我们需要修正bug，同时还要继续进行基本功能的开发。我们可以把修正bug当做开发的一部分并且在下一个版本稳定的时候发布，但是这样的话用户需要等待很长的时间才能看到修正的成果，这显然是不可接受的。另一方面，我们也可以更为频繁地发布新版本。但这样可能存在破坏原本稳定功能的危险。显然，我们的开发需要有两条支线。

Subversion允许我们在一个项目中同时保持多个平行的分支。我们可以在主干（常称为trunk或head）中像以前那样继续工作。下面我们将增加一段新的试验性的代码。我们使用一个特定的文件command/FeedbackCommand.php作为例子：

```
class FeedbackCommand extends Command {
    function execute( CommandContext $context ) {
        // 这里是新开发的存在风险的代码
        $msgSystem = ReceiverFactory::getMessageSystem();
        $email = $context->get( 'email' );
        $msg = $context->get( 'pass' );
        $topic = $context->get( 'topic' );
        $result = $msgSystem->dispatch( $email, $msg, $topic );
        if ( ! $user ) {
            $this->context->setError( $msgSystem->getError() );
            return false;
        }
        $context->addParam( "user", $user );
        return true;
    }
}
```

这里我们增加了一个注释来模拟代码的增加。这时候，用户开始报告他们不能在系统中使用反馈（feedback）功能。我们在这个文件中找到bug产生的位置：

```
//...
$result = $msgSystem->dispatch( $email, $msg, $topic );
if ( ! $user ) {
    $this->context->setError( $msgSystem->getError() );
//...
```

实际上，我们要检查的应该是`$result`而不是`$user`。当然我们可以在这里修正bug，但是用户要到我们的代码稳定之后才能看到这个修改。作为替代方案，我们可以创建项目的一个分支。

实际上，我会在创建发布标签的同时做这件事。

```
$ svn copy svn+ssh://localhost/var/local/svn/megaquiz/trunk  \
```

17

```
svn+ssh://localhost/var/local/svn/megaquiz/branches/megaquiz-branch1.0.0
-m'release branch'
```

```
Committed revision 10.
```

使用副本和前面的标签示例有什么区别？对Subversion而言，没什么区别。我只是复制到了branches目录，而不是tags目录。要说有区别，那就是我的目的不同。我想提交这个副本，而不是将它作为快照使用。

要使用新分支，必须先签出。我要在发布的时候修正代码。我退出了开发项目目录（这样当前的工作目录就不包含Subversion管理目录了），然后签出项目。

```
$ cd ..
$ svn checkout svn+ssh://localhost/var/local/svn/megaquiz/branches/megaquiz-branch1.0.0
 \
megaquiz-branch1.0.0
```

```
A     megaquiz-branch1.0.0/quizobjects
A     megaquiz-branch1.0.0/quizobjects/User.php
A     megaquiz-branch1.0.0/quiztools
A     megaquiz-branch1.0.0/quiztools/AccessManager.php
A     megaquiz-branch1.0.0/main.php
A     megaquiz-branch1.0.0/command
A     megaquiz-branch1.0.0/command/Command.php
A     megaquiz-branch1.0.0/command/CommandContext.php
A     megaquiz-branch1.0.0/command/FeedbackCommand.php
A     megaquiz-branch1.0.0/command/LoginCommand.php
Checked out revision 10.
```

在签出该分支之前，我退出了megaquiz-trunk目录。现在在同一层级有两个目录：megaquiz-trunk包含主干代码，我必须在此提交新功能，虽然这样做有风险。实际上，我现在就要这么做：

```
$ cd megaquiz-trunk/
$ svn commit -m'added new risky dev on trunk'
```

```
Sending        command/FeedbackCommand.php
Transmitting file data .
Committed revision 11.
```

另一方面，megaquiz-branch1.0.0是我的bug修正分支。此处我只提交缺陷修正。下面是我的修正：

```
class FeedbackCommand extends Command {

    function execute( CommandContext $context ) {
        $msgSystem = ReceiverFactory::getMessageSystem();
        $email = $context->get( 'email' );
        $msg = $context->get( 'pass' );
        $topic = $context->get( 'topic' );
```

```
        $result = $msgSystem->dispatch( $email, $msg, $topic );
        if ( ! $result ) {
            $this->context->setError( $msgSystem->getError() );
            return false;
        }
        $context->addParam( "user", $user );
        return true;
    }
}
```

我已经将

```
if ( ! $user ) {
```

修改成

```
if ( ! $result ) {
```

现在提交

```
$ cd ../megaquiz-branch1.0.0/
$ svn commit -m'fixed bug'
```

```
Sending        command/FeedbackCommand.php
Transmitting file data .
```

提交了修订12。我已经在meagquiz-branch1.0.0目录中编辑了该内容。通过编辑和提交，我的修改最终应用到了分支而不是主干，我可以将其作为点版本标记、导出并发布。

工作到此并未结束。现在已经修复了bug，我们需要把这个修改合并到主干上。

注解 合并还是不合并呢？有时候需要考虑一下。比如，在某些情况下，bug修复可能只是临时的工作，主干代码重构时bug可能会被自然消灭；或者因为规则改变，bug修复无效。这里需要一个权衡。但是，我工作过的大多数团队都倾向于合并分支上的修改到主干。新增的功能一般要合并到主干中，并尽快发布给用户，遵循“尽可能早尽可能频繁地发布”（release early and often）的原则。

为此，Subversion提供了合并命令。将工作目录修改为本地目录，并调用merge命令，将源URL传递给它。

```
$ svn merge svn+ssh://localhost/var/local/svn/megaquiz/branches/megaquiz-branch1.0.0
```

```
--- Merging r12 into '.':
U    command/FeedbackCommand.php
```

现在，查看主干中FeedbackCommand的版本时，我们会发现所有的修改都被合并了。

```
function execute( CommandContext $context ) {
    // 这里是新开发的存在风险的代码
```

```
        $msgSystem = ReceiverFactory::getMessageSystem();
        $email = $context->get( 'email' );
        $msg = $context->get( 'pass' );
        $topic = $context->get( 'topic' );
        $result = $msgSystem->dispatch( $email, $msg, $topic );
        if ( ! $result ) {
            $this->context->setError( $msgSystem->getError() );
            return false;
        }
        $context->addParam( "user", $user );
        return true;
    }
```

execute()方法包含了主干上开发的新内容以及我们修复bug的成果。

分支是Subversion中较为高级的一个特性，它常涉及一些有难度的问题。对于大型或开发周期很长的项目来说，分支是一个非常重要的技术。

17.9 小结

Subversion包含了很多命令，其中每个命令都有多个选项和功能。限于篇幅，本章仅提供简要的介绍。尽管如此，即使你只使用本章介绍的功能，也能得到很多好处。无论在防止数据丢失方面还是在改进协同开发方面，这些功能对你的实际工作都很有帮助。

在本章中，我们学习了Subversion的基础知识。在导入项目之前，我们先学会如何配置代码库。我们学会了检出、提交和更新代码以及标记和导出某个代码版本。在本章末尾，我们了解了分支及其在维护项目并行开发和bug修复中的作用。

本章曾提及一个问题。我们要求开发人员检出他们自己的项目版本。不过，项目一般都不会就地运行，而他们为了对代码进行测试，需要在本地机器上部署代码。有时他们只需要简单地复制一些目录，但是更多的时候部署会涉及整个项目配置的问题。在第18章中，我们将讨论自动化这一过程的技术。

第18章

使用PHPUnit进行测试

为了能够持续稳定地运行，系统中的每个组件都必须与其他相关组件协同工作。因此从某种意义上说，开发会破坏系统的运行。当改进类和包时，必须牢记需要同时修改任何使用它们的代码。某些改动甚至会引起连锁效应：会影响到距离被修改代码相当远的组件。高度警惕和对系统中依赖关系的深入认知也许能帮助你解决这个问题。但即使我们保持警惕，对依赖也有深入认识，随着系统复杂性的快速增长，我们还是很难预见到每一个有害的影响，因为系统总是由许多开发者一起协作开发的。为了解决这个问题，我们需要对每个组件进行定期的测试。测试是一个重复而且复杂的任务，但我们可以实现测试的自动化。

在PHP程序员可采用的测试方案中，PHPUnit也许是最常用的，也是功能最全的单元测试工具。在本章中，你将会学习以下关于PHPUnit的内容。

- 安装：使用PEAR安装PHPUnit。
- 编写测试：创建测试用例（test case）并使用断言（assertion）方法。
- 处理异常：确认失败的策略。
- 运行多个测试：将测试组合为套件。
- 构造断言逻辑：使用约束（constraint）。
- 伪造组件：模拟（mock）和桩（stub）。
- 测试Web应用：可能使用额外的工具。

18.1 功能测试与单元测试

测试在任何项目中都是一个基本要素。即使你的测试流程并不规范，也一定使用了一系列确认系统是否正常工作的操作。这种不规范的测试流程很快就会变得乏味，并且会逐渐让人形成一种撞大运的心态。

有一种测试方法是从一个项目的接口开始，为用户可能使用系统的各种方式建模。而这也是手工测试时通常会使用的方式，虽然也有众多的框架可以自动化该过程。这些功能测试有时又被称为验收测试（acceptance test），因为一个成功执行的动作列表可以作为一个项目阶段完成的依据。使用该方法时，通常会把系统看做一个黑盒——测试只针对产品功能，并不关注项目内部结构和处理过程。

功能测试从外部着手，而单元测试（本章的主题）则从内部着手。单元测试更加关注于类，并将测试方法组合到测试用例中。每个测试用例通过严格的检测来处理一个类，检查每个方法是否如预期般成功执行或失败。单元测试的目标是尽可能地在隔离周边环境的情况下测试每个组件。只有隔离了周围环境的影响，才能发现被测试的组件与周边组件间的耦合是否真正被解开。

测试可以作为项目构建过程的一部分，直接从命令行或者甚至通过一个网页来执行。本章主要采用命令行方式进行测试。

单元测试是保证系统设计质量的好办法。测试揭示了类和方法的职责。一些程序员甚至提倡测试先行（Test-First）的开发方式。他们认为应该在写一个类之前先写好测试。测试设定了一个类的目的，保证了一个干净的接口和简短并集中的方法。我个人曾未渴望达到这样纯粹的程度——也许是因为这不符合我的编码风格。然而，我每前进一步都会努力写测试。测试给我提供了重构代码时的安全保障。因为我知道在系统的其他地方可以捕获非预期的错误，所以可以放心地删除或替换掉项目中的某个代码包。

18.2　手工测试

上一节提及测试是每个项目的基本要素。我也说过测试在每个项目中都是不可避免的。我们都使用测试，但也常常将测试抛之脑后。

现在，让我们创建一些需要进行测试的类。下面是一个用于保存和获取用户信息的类。为了便于演示，它使用了数组而非我们通常期望使用的User对象：

```
class UserStore {
    private $users = array();

    function addUser( $name, $mail, $pass ) {
        if ( isset( $this->users[$mail] ) ) {
            throw new Exception(
                "User {$mail} already in the system");
        }

        if ( strlen( $pass ) < 5 ) {
            throw new Exception(
                "Password must have 5 or more letters");
        }

        $this->users[$mail] = array( 'pass' => $pass,
                                     'mail' => $mail,
                                     'name' => $name );
        return true;
    }

    function notifyPasswordFailure( $mail ) {
        if ( isset( $this->users[$mail] ) ) {
            $this->users[$mail]['failed']=time();
        }
    }

    function getUser( $mail ) {
        return ( $this->users[$mail] );
```

```
        }
    }
```

UserStore类通过addUser()方法接收用户数据，并通过getUser()方法来获取用户数据。用户的E-mail地址被作为用户数据的key值。如果你和我一样，那么你会在开发的时候同时写一些实例实现来检查代码是否如你预期那般运行，如下面的代码所示：

```
$store=new UserStore();
$store->addUser(  "bob williams",
                  "bob@example.com",
                  "12345" );
$user = $store->getUser(  "bob@example.com" );
print_r( $user );
```

这段代码我可能会加在类文件的尾部。当然这样的测试都是手工执行的，我要盯紧结果来确认UserStore::getUser()所返回的数据是否与我开始时添加的信息一致。不过这仍称得上是一种测试。

下面的客户类使用UserStore对象，用于检查用户是否提供了正确的验证信息：

```
class Validator {
    private $store;

    public function __construct( UserStore $store ) {
        $this->store = $store;
    }

    public function validateUser( $mail, $pass ) {
        if ( ! is_array($user = $this->store->getUser( $mail )) ) {
            return false;
        }
        if ( $user['pass'] == $pass ) {
            return true;
        }
        $this->store->notifyPasswordFailure( $mail );
        return false;
    }
}
```

该类需要用到一个保存在$store属性中的UserStore对象。validateUser()方法中用到了UserStore对象，用于验证E-mail对应的用户是否存在，以及密码是否与给定的参数相符。如果这两个条件都成立，则方法返回true。再一次，我边写边测：

```
$store = new UserStore();
$store->addUser(  "bob williams", "bob@example.com", "12345" );
$validator = new Validator( $store );
if ( $validator->validateUser( "bob@example.com", "12345" ) ) {
    print "pass, friend!\n";
}
```

我先实例化一个UserStore对象并添加测试数据，接着传递给随后实例化的Validator对象，然后我便能确认用户名和密码。

一旦对最终所得的测试结果满意，我便会删除或注释掉这些合理的测试代码。这是一种巨大的资源浪费，因为这些测试完全可以构成对系统进行详尽测试的基础。PHPUnit便是一个可以对

系统进行全面测试的工具。

18.3　引入 PHPUnit

PHPUnit是测试工具xUnit家族的一员。xUnit家族的祖先是SUnit，它是由Kent Beck创造的一个用于测试Smalltalk语言所创建系统的框架。但是如果没有Java版的实现jUnit及新兴的如XP（eXtreme Programming，极限编程）和Scrum这样的敏捷方法论，xUnit框架可能并不会成为如此流行的工具。

PHPUnit的当前版本由Sebastian Bergmann创建。Sebastian Bergmann在2007年初将PHPUnit2（他也是作者）改名为PHPUnit，并将其从pear.php.net转移到了pear.phpunit.de。因此，当你安装的时候必须先告诉pear去哪里搜索PHPUnit框架：

```
$ pear channel-discover pear.phpunit.de
$ pear channel-discover pear.symfony-project.com
$ pear install phpunit
```

注解　我将用粗体来显示命令行命令，以便于区分命令和命令的输出结果。

注意，我添加了另一个频道：pear.symfony-project.com。这对满足托管在那儿的PHPUnit的依赖是必要的。

18.3.1　创建测试用例

安装好PHPUnit以后，就可以为UserStore类编写测试了。对每个目标组件的测试都应该编写在一个继承自PHPUnit包中PHPUnit_Framework_TestCase的类中。下面演示如何创建一个最简单的测试用例类：

```
require_once 'PHPUnit/Framework/TestCase.php';

class UserStoreTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
    }

    public function  tearDown() {
    }

    //...
}
```

我将测试用例类命名为UserStoreTest。在测试用例中使用被测试类的名字不是必需的，但是大多数开发者都会这么做。这样的命名习惯可以使我们在组件和测试的数量越来越多的情况下，也能快速找到某一个测试用具[①]。我们通常也会将测试用例放在与被测试类所在目录相对应

① 在软件测试中，测试用具是指一个包含了软件和测试数据的集合，用以测试一个程序单元，使之在不同的条件下运行，并监控它的行为和输出。测试用具应该包含测试环境的搭建和清理，选择运行单个测试或者所有测试的方法，分析输出结果的手段以及错误的标准报告等。——译者注

的目录中。使用这样的目录结构，你甚至可以从命令行打开一个测试而无需查看它是否存在。测试用例类中的每一个测试都独立运行。setUp()方法会在每个测试方法前被自动调用，从而允许我们为测试设立一个稳定的预备环境。tearDown()方法会在每个测试方法运行后被调用。如果你的测试改变了系统的周边环境，可以使用tearDown()来恢复状态。由setUp()和tearDown()方法管理的公共平台就是所谓的测试设备[①]。

为了测试UserStore类，首先需要它的一个实例。我们可以在setUp()方法中实例化UserStore对象，并将该对象实例赋值给一个属性。下面加入一个测试方法：

```
require_once('UserStore.php');
require_once('PHPUnit/Framework/TestCase.php');


class UserStoreTest extends PHPUnit_Framework_TestCase {
    private $store;

    public function setUp() {
        $this->store = new UserStore();
    }

    public function tearDown() {
    }

    public function  testGetUser() {
        $this->store->addUser(  "bob williams", "a@b.com", "12345" );
        $user = $this->store->getUser(  "a@b.com" );
        $this->assertEquals( $user['mail'], "a@b.com" );
        $this->assertEquals( $user['name'], "bob williams" );
        $this->assertEquals( $user['pass'], "12345" );
    }
}
```

测试方法必须以“test”开头，并且没有参数。这是因为测试用例类是通过反射来操作的。

注解　第5章详细介绍了反射。

PHPUnit在运行测试时会检查测试类的每一个方法，但只会调用那些以“test”开头的方法。

上面的代码可以测试获取用户信息的操作是否正常。我们不需要在每个测试方法中实例化UserStore，因为我们已经在setUp()方法中处理过对象实例化。因为setUp()方法会被每个测试方法调用，所以可以确保$store属性在每次被调用时都拥有一个新实例化的对象。

在testGetUser()方法中，我们先通过UserStore::addUser()方法提供虚构的数据，然后获取数据并测试数据中的每一个元素。

18.3.2　断言方法

编程中的术语“断言”是指用来判断系统中某个假设是否成立的语句或方法。使用断言的常

① fixture，指测试所需对象的集合。——译者注

见做法是定义一个什么是什么的预期，如$cheese是"blue"或$pie是"apple"。如果你的预期模糊不清，那么就会产生某种警告。断言是一种非常好的改进系统安全性的方法，因此有一些编程语言默认支持断言，同时允许你在产品环境中关闭断言（如Java）。PHPUnit通过一系列静态方法来支持断言。

在之前的例子中，我使用了一个继承得到的静态方法：assertEquals()。它会比较它的两个参数来检查它们是否相等。如果它们不匹配，那么测试方法会被记录为失败的测试。通过子类化PHPUnit_Framework_TestCase，便能访问一系列断言方法。表18-1列举了这些方法。

表18-1　PHPUnit_Framework_TestCase的断言方法

方　法	描　述
assertEquals($val1, $val2, $delta, $message)	当$val1与$val2不相等时失败（$delta表示允许的误差幅度）
assertFalse($expression, $message)	检查$expression表达式，当$expression的结果不是false时失败
assertTrue($expression, $message)	检查$expression表达式，当$expression的结果不是true时失败
assertNotNull($val, $message)	如果$val是null时失败
assertNull($val, $message)	如果$val不是null时失败
assertSame($val1, $val2, $message)	如果$val1和$val2不是对同一个对象的引用或者它们是不同类型或不同值的变量，则失败
assertNotSame($val1, $val2, $message)	如果$val1和$val2是同对一个对象的引用或者是同样的类型和值的变量，则失败
assertRegExp($regexp, $val, $message)	如果$val与正则表达式$regexp不匹配，则失败
assertType($typestring, $val, $message)	如果$val的类型与$type不符，则失败
assertAttributeSame($val, $attribute, $classname, $message)	如果$val与$classname::$attribute的值或类型不相同，则失败
fail()	使测试方法直接返回失败

18.3.3 测试异常

开发者通常希望代码可用并运行良好。这种心理经常会贯彻到测试中，特别是在测试自己写的代码时。这时存在一个诱惑，就是只测试方法是否正常工作，而很容易忘掉对方法失败进行测试的重要性。比如一个方法的错误检查是否完善？当它应该抛出异常时会不会抛出异常？它是否能抛出正确的异常？在问题发生前只执行了一半的操作，在错误发生后是否会进行清理？ 作为一个测试者，你的任务就是检查所有这些情况。幸好PHPUnit可以帮助你。

下面的测试用于检查操作失败时UserStore类的行为：

```
//...
public function  testAddUser_ShortPass() {
    try {
        $this->store->addUser(  "bob williams", "bob@example.com", "ff" );
    } catch ( Exception $e ) { return; }
    $this->fail("Short password exception expected");
}
//...
```

如果回头看UserStore::addUser()方法，你会看到当用户密码长度小于5个字符时会抛出异常。上面的测试尝试确认这点。我们在try子句中添加了一个密码不合法的用户。如果预期的异常被抛出，那么一切正常并且我们直接返回。该方法中的最后一行代码按理不应该被执行，因此在那里我们调用了fail()方法。如果addUser()方法并未抛出我们所预期的异常，那么catch语句就不会被执行，而fail()方法就会被调用了。

另一个可用于测试异常是否被正确抛出的类是PHPUnit_Extensions_ExceptionTestCase，它是PHPUnit的扩展类。该类提供了一个setExpectedException()断言方法，可以传入你期望抛出的异常类型（Exception或者Exception的子类）。如果测试方法执行结束时没有抛出正确的异常，那么测试便会失败。

下面重新编写先前的测试：

```
require_once('PHPUnit/Framework/TestCase.php');
require_once('UserStore.php');

class UserStoreTest extends PHPUnit_Framework_TestCase {
    private $store;

    public function setUp() {
        $this->store = new UserStore();
    }

    public function testAddUser_ShortPass() {
        $this->setExpectedException('Exception');
        $this->store->addUser(  "bob williams", "bob@example.com", "ff" );
    }
}
```

18.3.4 运行测试套件

如果测试UserStore类，那么也应该测试Validator类。下面是ValidateTest类的一个简化版本，它用于测试Validator::validateUser()方法：

```
require_once('UserStore.php');
require_once('Validator.php');
require_once('PHPUnit/Framework/TestCase.php');


class ValidatorTest extends PHPUnit_Framework_TestCase {
    private $validator;

    public function setUp() {
        $store = new UserStore();
        $store->addUser(  "bob williams", "bob@example.com", "12345" );
        $this->validator = new Validator( $store );
    }

    public function tearDown() {
    }

    public function testValidate_CorrectPass() {
```

```
        $this->assertTrue(
            $this->validator->validateUser( "bob@example.com", "12345" ),
            "Expecting successful validation"
            );
    }
}
```

现在我们有了多个测试用例，那么如何一起运行这些测试用例呢？最好的方法是将测试类放到一个称为test的目录中，然后指定这个目录，PHPUnit便会在此执行所有测试。

```
$ phpunit test/
```

```
PHPUnit 3.4.11 by Sebastian Bergmann.
.....

Time: 1 second, Memory: 3.75Mb

OK (5 tests, 10 assertions)
```

对于大型项目，在子目录中进一步组织测试可能会更好，最好子目录结构与包的结构相同，然后你就可以在需要时指定每个包了。

18.3.5　约束

在大多数情况下，你会在测试时使用PHPUnit中现成的断言。而事实上，你只要通过AssertTrue()方法就可完成很多测试工作。但在PHPUnit 3.0以后，PHPUnit_Framework_TestCase包含了一套返回PHPUnit_Framework_Constraint对象的工厂方法。你可以合并这些对象并将其传给PHPUnit_Framework_TestCase::AssertThat()来构造你自己的断言。

我们马上来写一个示例。UserStore对象应该不允许添加重复的E-mail地址。下面便是确认此功能的测试：

```
class UserStoreTest extends PHPUnit_Framework_TestCase {
    //....

    public function  testAddUser_duplicate() {
        try {
            $ret = $this->store->addUser(  "bob williams", "a@b.com", "123456" );
            $ret = $this->store->addUser(  "bob stevens", "a@b.com", "123456" );
            self::fail( "Exception should have been thrown" );
        } catch ( Exception $e ) {
            $const = $this->logicalAnd(
                        $this->logicalNot( $this->contains("bob stevens")),
                        $this->isType('array')
                    );
            self::AssertThat( $this->store->getUser( "a@b.com"), $const );
        }
    }
```

该测试先给UserStore对象添加一个用户，接着给另一个用户添加相同的E-mail地址。测试因此确认了第二次调用addUser()时会有异常抛出。在catch子句中，我们用便捷方法构建了一

个约束对象。这会返回对应的PHPUnit_Framework_Constraint的实例。让我们分解上例中的组合约束对象：

```
$this->contains("bob stevens")
```

这里返回了一个PHPUnit_Framework_Constraint_TraversableContains对象。当传给AssertThat方法时，如果测试得到的数组没有包含值为"bob stevens"的元素，那么该对象会生成一个错误。但是例中通过将该约束传给另外一个PHPUnit_Framework_Constraint_Not对象来否定这种说法。代码中再次使用了TestCase类中的便捷方法（该方法实际上定义于父类Assert中）。

```
$this->logicalNot( $this->contains("bob stevens"))
```

现在，如果测试的值（必须可被遍历，traversable）包含匹配"bob stevens"字符串的元素，则AssertThat断言会失败。这样我们就能建立非常复杂的逻辑结构。此时，该约束可以总结为："如果测试的值是数组且不包含"bob stevens"字符串，则测试不会失败。"通过这种方式，我们可以建立更多约束，然后通过将约束和测试值同时传给AssertThat()方法来执行约束。

当然，我们可以使用标准的断言方法来达成目的，但是约束有几个优点。首先，它们能使代码逻辑清晰。其次，（也是更重要的）约束可以重复使用。你可以建立一个复杂的约束库并在不同的测试中使用它们。你甚至可以合并复杂的约束：

```
$const = $this->logicalAnd(
    $a_complex_constraint,
    $another_complex_constraint );
```

表18-2列出了TestCase类中的一些约束方法。

表18-2 常用的约束方法

TestCase类的方法	约束失败，除非以下条件成立
greaterThan($num)	测试值大于$num
contains($val)	测试值（可遍历）包含匹配$val的元素
identicalTo($val)	测试值是对同一个对象$val的引用。$val不是对象时，则指相同的类型和值
greaterThanOrEqual($num)	测试值大于或等于$num
lessThan($num)	测试值小于$num
lessThanOrEqual($num)	测试值小于或等于$num
equalTo($value, $delta=0, $depth=10)	测试值等于$val。$delta定义了数字比较时的误差幅度，而$depth决定了对数组或者对象递归的深度
stringContains($str, $casesensitive=true)	测试值包含字符串$str。默认情况下区分大小写
matchesRegularExpression($pattern)	测试值可匹配正则表达式$pattern
logicalAnd(PHPUnit_Framework_Constraint $const, [, $const..])	通过所有的约束
logicalOr(PHPUnit_Framework_Constraint $const, [, $const..])	至少通过一个约束
logicalNot(PHPUnit_Framework_Constraint $const)	没有通过约束

18.3.6 模拟与桩

单元测试的目标就是尽可能在不受外部环境影响的条件下测试系统中的组件。但是只有极少数组件以这样的真空形式存在。甚至与外部耦合非常小的类也需要访问其他对象，例如以其他对象作为类方法参数。许多类也直接与数据库或文件系统一起工作。

我们已经看到过处理这个问题的一种办法。我们可以利用`setUp()`和`tearDown()`方法来管理测试装置。测试装置就是一组供测试使用的公用资源，可能包含数据库连接、配置对象、文件系统的暂存区域等。

另一种方式就是伪造测试类的环境。该方式会创建假装在进行真实工作的对象。例如，你可能会传一个伪造的数据库映射给测试对象的构造函数。因为伪造对象和真实的映射类（继承自一个共同的抽象基类或甚至重写该类本身）是相同类型，所以测试目标对此一无所知。你可以预先准备好具有正确数据的虚拟对象（fake object）。这类为单元测试提供沙箱的对象就是所谓的桩[1]。它们非常有用，因为它们使你能仅关注于要测试的类本身而无需同时注意整个系统。

但虚拟对象的功能还不止于此。因为你所测试的对象可能会以某种方式调用虚拟对象，所以你可以准备好虚拟对象以便它按照预期那样确保调用可以进行。以这种类似间谍的方式使用虚拟对象被称为“行为确认”。对于对象行为的预期正是mock对象与stub对象不同的地方[2]。

创建mock对象的办法有两种。首先，你可以自己创建相应的类来确保返回特定的值，以供方法调用。这是一个简单的过程，但需要花费一些时间。

PHPUnit提供了另一个更为易用和动态的解决方案。它会为你在运行时自动生成mock对象。PHPUnit会检查你想要模拟的对象并创建一个重写该对象方法的子类。一旦有了mock对象的实例，你就可以调用其中的方法来准备数据或设定测试成功的条件。

下面举个例子。`UserStore`类包含了一个`notifyPasswordFailure()`方法，该方法会设定特定用户的密码字段。当尝试设置密码失败时，该方法就会被`Validator`调用。下面的例子创建了`UserStore`类的mock对象，以便它能为`Validator`对象提供数据及确认`notifyPassword-Failure()`方法如预期地被调用：

```
class ValidatorTest extends PHPUnit_Framework_TestCase {
    //...

    public function testValidate_FalsePass() {
        $store = $this->getMock("UserStore");
        $this->validator = new Validator( $store );

        $store->expects($this->once() )
              ->method('notifyPasswordFailure')
              ->with( $this->equalTo('bob@example.com') );

        $store->expects( $this->any() )
```

① stub，可理解为占位用的假对象。——译者注

② 简单地说，stub只是占位，而mock对象还关注于对行为的预期，这是两者的区别。如果读者想了解更多内容，建议阅读Martin Fowler的*Mocks Aren't Stubs*一文。——译者注

```
                ->method("getUser")
                ->will( $this->returnValue(array("name"=>"bob@example.com",
                  "pass"=>"right")));

        $this->validator->validateUser("bob@example.com", "wrong");

    }
}
```

mock对象使用了类似于语言的流畅接口[①]。使用这样的语言结构，调用从左到右进行，每次调用都返回对一个对象的引用，而该对象的引用可被后面的方法（仍会返回对象）继续调用。这样编写代码很简单，但调试的时候会有点麻烦。

上面的范例调用了PHPUnit_Framework_TestCase的getMock()方法，并将要模拟的类名"UserStore"传给它。该方法会动态生成一个类，并为该类实例化一个对象。mock对象被保存在$store中并被传给Validator。这样做不会引起错误，因为该对象来源于UserStore的子类。这样我们欺骗了Validator，使之接受了一个间谍。

由PHPUnit生成的mock对象有一个expects()方法。该方法要求一个匹配对象（它实际上是PHPUnit_Framework_MockObject_Matcher_Invocation类型。你不需要知道其确切类型，就可以在TestCase中使用便捷方法来生成匹配对象）作为参数。匹配对象定义了期望的基数，即一个方法应该被调用的次数。

表18-3列出了TestCase类中可用的匹配方法。

表18-3 一些匹配方法

TestCase类的方法	匹配失败，除非以下条件成立
any()	零次或者任意次调用相应方法（对于用于返回值但不测试调用的stub对象非常有用）
never()	没有调用相应方法
atLeastOnce()	一次或者更多次调用相应方法
once()	仅调用一次相应方法
exactly($num)	对相应方法调用了$num次
at($num)	对相应方法在第$num次的调用（对mock对象的每一次调用都会被记录和索引）

设置匹配要求后，还需要指定它应用于哪个方法。例如expects()方法返回的对象（如果必须知道的话，其类型为PHPUnit_Framework_MockObject_Builder_InvocationMocker）有一个method()方法。你可以只用方法名来调用这个方法。这样足矣完成真正的模拟：

```
$store = $this->getMock("UserStore");
$store->expects( $this->once() )
            ->method('notifyPasswordFailure');
```

我们还需要确定传给notifyPasswordFailure()方法的参数。InvocationMocker::method()会返回它所调用的对象的一个实例。InvocationMocker对象有一个以变量列表为参数

① 指的是PHP 5中可以使用多个->符号连续调用对象方法。这里$store、expects、method、with连起来就像一句话，所以称为“类似于语言”。——译者注

18

的with()方法。该方法也可以以约束对象为参数，因此你也能对参数范围进行检测。有了with()方法，我们就能完成整个语句并确保传递正确的参数给notifyPasswordFailure()。

```
$store->expects($this->once() )
      ->method('notifyPasswordFailure')
      ->with( $this->equalTo('bob@example.com') );
```

现在你知道这种调用方式被称为“流畅接口”的原因了。它读起来就像一个句子：“$store对象期望调用一次notifyPasswordFailure()方法，该方法以bob@example.com为参数。”

注意示例中将一个约束对象传递给了with()方法。实际上，这有点多余——因为任何无修饰的参数都会在内部被强制转换成约束，所以也可以像这样写：

```
$store->expects($this->once() )
      ->method('notifyPasswordFailure')
      ->with( 'bob@example.com' );
```

有时候，你可能只想将PHPUnit的mock对象作为stub使用，即那些能够返回值以使测试运行的对象。在这种情况下，你可以在method()方法中调用InvocationMocker::will()。will()方法要求传入之前相关方法所要预备好的返回值（当方法被重复调用时会是多个值）。你可以通过调用TestCase::returnValue()或TestCase::onConsecutiveCalls()来得到返回值，然后传递给will()方法。下面让我们看看代码，这样比较容易理解。下面是前例中的代码片段，用来让UserStore返回值：

```
    $store->expects( $this->any() )
  ->method("getUser")
  ->will( $this->returnValue(
       array(  "name"=>"bob williams",
               "mail"=>"bob@example.com",
               "pass"=>"right")));
```

准备好UserStore的mock对象并预期对getUser()会有任意次的调用，接着我们要关心的是提供数据而非对方法调用进行测试。因此，代码调用will()，并将调用TestCase::returnValue()所返回的结果数据作为参数传递给will()方法。（调用TestCase::returnValue()会返回一个PHPUnit_Framework_MockObject_Stub_Return对象，通常你只需记住获得它的便捷方法即可。）

你也可以调用TestCase::onConsecutiveCalls()并把返回结果传递给will()。TestCase::onConsecutiveCalls()方法接受任意个数的参数，其中每一个参数都由mock方法被重复调用时返回。

18.3.7　失败是成功之母

大多数人都认为测试是一个好东西，但通常只有在它几次挽救你于危难之后，你才会渐渐真的爱上它。让我们来模拟一个系统某部分的修改导致了意外结果的情形。

假设Userstore类已经运行了一段时间，在一次代码审核中，我们想改进该类来生成User对象，而非生成关联数组。下面是新版本的代码：

```
class UserStore {
```

```
    private $users = array();

    function addUser( $name, $mail, $pass ) {

        if ( isset( $this->users[$mail] ) ) {
            throw new Exception(
                "User {$mail} already in the system");
        }

        $this->users[$mail] = new User( $name, $mail, $pass );
        return true;
    }

    function notifyPasswordFailure( $mail ) {
        if ( isset( $this->users[$mail] ) ) {
            $this->users[$mail]->failed(time());
        }
    }

    function getUser( $mail ) {
        if ( isset( $this->users[$mail] ) ) {
            return ( $this->users[$mail] );
        }
        return null;
    }
}
```

下面是简单的User类：

```
class User {
    private $name;
    private $mail;
    private $pass;
    private $failed;

    function __construct( $name, $mail, $pass ) {

        if ( strlen( $pass ) < 5 ) {
            throw new Exception(
                "Password must have 5 or more letters");
        }

        $this->name       = $name;
        $this->mail       = $mail;
        $this->pass       = $pass;
    }

    function getName() {
        return $this->name;
    }

    function getMail() {
        return $this->mail;
    }

    function getPass() {
```

```
        return $this->pass;
    }

    function failed( $time ) {
        $this->failed = $time;
    }
}
```

当然，我们要修改UserStoreTest类来应对这些修改。下面是原本设计为使用数组的代码：

```
public function  testGetUser() {
    $this->store->addUser(  "bob williams", "a@b.com", "12345" );
    $user = $this->store->getUser(  "a@b.com" );
    $this->assertEquals( $user['mail'], "a@b.com" );
    //...
```

它会被改为使用对象的代码：

```
public function  testGetUser() {
    $this->store->addUser(  "bob williams", "a@b.com", "12345" );
    $user = $this->store->getUser(  "a@b.com" );
    $this->assertEquals( $user->getMail(), "a@b.com" );
    // ...
```

但是当我们重新运行测试套件时，会得到一个警告（这说明我们的工作还没有完成）：

```
$ php AppTests.php
```

```
PHPUnit 3.0.6 by Sebastian Bergmann.

...FF

Time: 00:00

There were 2 failures:

1) testValidate_CorrectPass(ValidatorTest)
Expecting successful validation
Failed asserting that <boolean:false> is identical to <boolean:true>.
/project/wibble/ValidatorTest.php:22

2) testValidate_FalsePass(ValidatorTest)
Expectation failed for method name is equal to <string:notifyPasswordFailure> ➥
when invoked 1 time(s).
Expected invocation count is wrong.

FAILURES!
Tests: 5, Failures: 2.
```

从以上结果可以看到ValidatorTest有问题。再检查一下Validator::validateUser()方法：

```
public function validateUser( $mail, $pass ) {

    if ( ! is_array($user = $this->store->getUser( $mail )) ) {
        return false;
    }
```

```
        if ( $user['pass'] == $pass ) {
            return true;
        }
        $this->store->notifyPasswordFailure( $mail );
        return false;
    }
```

其中调用了getUser()方法。虽然getUser()现在返回一个对象而非一个数组，但方法并没有产生警告。getUser()原本在成功时返回所请求的用户数组或在失败时返回null，因此我们可以使用is_array()函数检查返回值是否是数组，以此来验证用户。但现在getUser()返回一个对象，从而validateUser()方法将始终返回false。如果没有测试框架，那么Validator将毫无疑问地拒绝所有的用户，而且不报错或警告。

现在，想象一下在没有适当测试框架的情况下，你在星期五的晚上做了这个非常小的修改。也许你会收到一条令人抓狂的、将你从酒吧、躺椅或者餐馆中拉出来的短信："你干了什么？我们所有的客户都被关在了外面！"

最狡诈的bug不会导致PHP解释器报告错误。它们躲藏在完全合法的代码下，并毫无声息地破坏系统逻辑。很多bug不会表现在你正在编写的代码中，这些bug也许是由你修改的地方引起的，但其影响在几天甚至几星期后才出现在另一处。一个测试框架至少能帮助你捕获其中一部分，从而提前预防问题的出现，而不是事后寻找问题所在。

请边编写代码边测试，并且经常运行测试。如果有人报告了一个bug，首先添加测试来确认bug是否存在，再修复bug使测试通过——bug有在同一地方重复出现的怪癖。编写测试来证明bug已修复，然后给这个修复提供对应于后续问题的保障，这便是所谓的*回归测试*(regression testing)。顺带说一下，如果你将回归测试放置在一个单独的目录中，那么记得用描述性的名字来给文件命名。我的团队曾在一个项目中用Bugzilla的bug号来命名回归测试，最终我们有了一个包含400个测试文件的目录，而每个文件都有类似test_973892.php这样的文件名，结果寻找一个测试成了很麻烦的事情。

18.4 编写 Web 测试

你的Web系统应该设计成这样：通过命令行或API调用可以轻松地调用。第12章介绍了一些这方面的技巧。尤其是，如果创建Request类来封装HTTP请求，你可以通过请求参数填充实例，就像根据命令行或方法参数列表来填充实例一样简单。该系统随后会在不清楚上下文的状态下运行。

如果你的Web系统很难在不同的上下文中运行，就说明设计可能有问题。例如，如果你将很多文件路径硬编码到了组件中，就可能导致紧密耦合。你应该考虑将造成组件与上下文耦合的元素移动到封装对象中，然后通过中心代码库获得这些封装对象。第12章中介绍的注册表模式可以帮助你实现这一过程。

一旦Web系统可以直接通过方法调用运行，你会发现在不使用其他工具的情况下，高层次的Web测试相对会更容易编写。

但是，即使是经过深思熟虑之后开发出来的项目，恐怕也需要经过重构才能具备测试条件。

以我的编程经验来看，这通常会改进设计。我打算改进第12章和第13章的WOO示例的一个方面来进行单元测试，以此说明这一过程。

18.4.1　为测试重构 Web 应用

从测试人员的角度来看，WOO示例已达到了测试的理想状态。因为系统使用单个Front Controller，所以有一个简单的API接口。下面是简单的Runner.php类：

```
require_once( "woo/controller/Controller.php");
\woo\controller\Controller::run();
```

此时把它添加到单元测试中并不难，是吗？但命令行参数谁来处理？从某种意义上说，Request类就是做这件事的：

```
// \woo\controller\Request
    function init() {
        if ( isset( $_SERVER['REQUEST_METHOD'] ) ) {
            $this->properties = $_REQUEST;
            return;
        }

        foreach( $_SERVER['argv'] as $arg ) {
            if ( strpos( $arg, '=' ) ) {
                list( $key, $val )=explode( "=", $arg );
                $this->setProperty( $key, $val );
            }
        }
    }
```

init()方法会检测它是否运行在服务器环境中，并相应地填充$properties数组（直接填充或通过setProperty()填充）。对于命令行调用来说，这很奏效。这意味着我可以运行下面这样的内容：

```
$ php runner.php cmd=AddVenue venue_name=bob
```

并得到下列结果：

```
<html>
<head>
<title>Add a Space for venue bob</title>
</head>
<body>
<h1>Add a Space for Venue 'bob'</h1>
<table>
<tr>
<td>
'bob' added (5)</td></tr><tr><td>please add name for the space</td>
</tr>
</table>
[add space]
<form method="post">
```

```
    <input type="text" value="" name="space_name"/>
    <input type="hidden" name="cmd" value="AddSpace" />
    <input type="hidden" name="venue_id" value="5" />
    <input type="submit" value="submit" />
</form>
</body>
</html>
```

虽然对于命令行来说，这种方法很奏效，但通过方法调用传递参数，还是有点麻烦。一种解决方法是：在调用控制器的`run()`方法之前，手动设置`$argv`数组。但这种方法不够优雅，我不太喜欢。直接使用魔法数组（magic array）明显就是错误的，而且两端又都要涉及字符串操作，就是错上加错。仔细观察一下这个控制器类，可以发现改进设计和可测试性的机会。下面是`handleRequest()`方法的部分代码：

```
// \woo\controller\Controller
    function handleRequest() {
        $request = new Request();
        $app_c = \woo\base\ApplicationRegistry::appController();
        while( $cmd = $app_c->getCommand( $request ) ) {
            $cmd->execute( $request );
        }
        \woo\domain\ObjectWatcher::instance()->performOperations();
        $this->invokeView( $app_c->getView( $request ) );
    }
```

这个方法会被静态的`run()`方法调用。我首先注意到的就是，这个方法明显有问题：`Request`对象直接被实例化。这意味着我无法像预期那样加入插桩代码。是该使用线程了。`Request`方法中发生了什么？下面是构造函数：

```
// \woo\controller\Request
    function __construct() {
        $this->init();
        \woo\base\RequestRegistry::setRequest($this );
    }
```

情况好像更糟了。为了让其他组件可以访问它，`Request`类将其自身指向了`RequestRegistry`。再三思考之后，我发现有两样东西我不喜欢。首先，代码暗示在使用`Registry`访问`Request`对象之前，必须执行一次直接调用。其次，出现了一些不必要的耦合。`Request`类并不一定需要知道`RequestRegistry`。

那么我怎样改进设计，同时使系统更容易进行测试？我希望将实例化推迟到`RequestRegistry`中进行。这样的话，我就可以稍后扩展`RequestRegistry::instance()`的实现，从而返回使用伪组件填充的`MockRequestRegistry`。我喜欢欺骗系统。所以首先删除了`Request`对象中的`setRequest()`那一行。现在我将`Request`的实例化推迟到`RequestRegistry`对象中进行。

```
namespace woo/controller;

//...
```

```
class RequestRegistry extends Registry {
    private $request;

// ...

    static function getRequest() {
        $that = self::instance();
        if ( ! isset( $that->request ) ) {
            $that->request = new \woo\controller\Request();
        }
        return $that->request;
    }
}
```

最后，我必须在Controller中替换这个直接实例化：

```
// \woo\controller\Controller
    function handleRequest() {

        $request = \woo\base\RequestRegistry::getRequest();
        $app_c = \woo\base\ApplicationRegistry::appController();
        while( $cmd = $app_c->getCommand( $request ) ) {
            $cmd->execute( $request );
        }
        \woo\domain\ObjectWatcher::instance()->performOperations();
        $this->invokeView( $app_c->getView( $request ) );
    }
```

完成这些重构以后，我的系统更容易进行测试了，同时我的设计也必然得到了改进。现在开始编写测试。

18.4.2　简单的 Web 测试

下面这个测试用例会对WOO系统执行一个非常基础的测试：

```
class AddVenueTest extends PHPUnit_Framework_TestCase {

    function testAddVenueVanilla() {
        $this->runCommand("AddVenue", array("venue_name"=>"bob") );
    }

    function runCommand( $command=null, array $args=null ) {
        $request = \woo\base\RequestRegistry::getRequest();
        if ( ! is_null( $args ) ) {
            foreach( $args as $key=>$val ) {
                $request->setProperty( $key, $val );
            }
        }
        if ( ! is_null( $command ) ) {
            $request->setProperty( 'cmd', $command );
        }
        woo\controller\Controller::run();
    }
}
```

实际上，这段代码并没有进行足够的测试，不足以证明系统可以被调用。真正的工作是在runCommand()方法中完成的。这里没有什么特别聪明的内容。我通过RequestRegistry得到Request对象，并使用方法调用中提供的键和值填充该对象。因为Controller会用相同的源填充Request对象，它将使用我设置的值。

实际运行测试，事实证明一切正常，输出结果如我所料。问题在于输出结果是由视图给出的，因此很难测试。缓冲输出就能很好地解决这个问题：

```
class AddVenueTest extends PHPUnit_Framework_TestCase {
    function testAddVenueVanilla() {
        $output = $this->runCommand("AddVenue", array("venue_name"=>"bob") );
        self::AssertRegexp( "/added/", $output );
    }

    function runCommand( $command=null, array $args=null ) {
        ob_start();
        $request = \woo\base\RequestRegistry::getRequest();
        if ( ! is_null( $args ) ) {
            foreach( $args as $key=>$val ) {
                $request->setProperty( $key, $val );
            }
        }
        if ( ! is_null( $command ) ) {
            $request->setProperty( 'cmd', $command );
        }
        woo\controller\Controller::run();
        $ret = ob_get_contents();
        ob_end_clean();
        return $ret;
    }
}
```

将系统的输出放在缓冲区中，我就可以使用runCommand()方法将其返回了。出于演示目的，我在返回值中增加了一条简单的断言。

下面是在命令行中看到的结果：

```
$ phpunit test/AddVenueTest.php
```

```
PHPUnit 3.4.11 by Sebastian Bergmann.
.
Time: 0 seconds, Memory: 4.00Mb
OK (1 test, 1 assertion)
```


如果你打算以这种方式对系统运行大量测试，那么应该创建一个Web UI超类来持有runCommand()。

有些问题就不说了，而你可能在自己的测试中遇到这些问题。你要确保系统使用的是可配置的存储位置。因为你不希望测试和开发使用相同的位置存储数据。这是另一个改进设计的机会。查找硬编码的文件路径和DSN值，将它们转移到Registry中设置，然后确保测试在沙盒中进行，

但要在测试用例的setUp()方法中设置这些值。查看MockRequestRegistry中的交换操作，通过它可以换入stub、mock，以及其他各种虚构的组件。

与此类似的很多方法都很适合测试Web应用的输入和输出，但有一些很明显的限制。该方法不能捕获浏览器的历史记录。如果Web应用使用JavaScript、Ajax和其他的客户端脚本，那么测试由你的系统生成的文本并不能判断用户是否能看到正常的界面。

很幸运，有一种解决方案。

18.4.3 Selenium

Selenium（http://seleniumhq.org/）由一组定义Web测试的命令（有时称为selenese）组成。它还提供了一些工具，用于编写和运行浏览器测试，这些工具还可以将测试绑定到现有的测试平台上。其中的一个平台就是PHPUnit，太棒了！

为了简单介绍Selenium，我将使用Selenium IDE编写一个简单的WOO测试，然后输出测试结果，并将它作为PHPUnit测试用例来运行。

1. 获取Selenium

可以通过http://seleniumhq.org/download/下载Selenium组件。对于本例来说，要下载SeleniumIDE和Selenium RC。

如果你使用Firefox浏览器（为运行IDE，需运行该浏览器），你会发现Selenium IDE在下载过程中直接安装（在你确认了一两个提示以后），并出现在工具菜单中。

Selenium RC需要多动几下手。下载了这个包以后，找到名为selenium-remote-control-1.0.3.zip的文件（当然你的版本号可能不同）。解压缩该文件，查找jar（Java ARchive）文件，该文件可能类似于selenium-server-1.0.3/selenium-server.jar。将该文件复制到某个中心位置。此时你需要将Java安装到系统中，才能继续执行。如果系统上已经安装了Java，那么可以启动Selenium 服务器。

下面我将服务器复制到主目录下的lib目录中，然后启动服务器：

```
$ cp selenium-server-1.0.3/selenium-server.jar ~/lib/
$ java -jar ~/lib/selenium-server.jar
```

```
13:03:28.713 INFO - Java: Sun Microsystems Inc. 14.0-b16
13:03:28.745 INFO - OS: Linux 2.6.31.5-127.fc12.i686 i386
13:03:28.787 INFO - v2.0 [a2], with Core v2.0 [a2]
13:03:29.273 INFO - RemoteWebDriver instances should connect to:
http://192.168.1.65:4444/wd/hub
13:03:29.276 INFO - Version Jetty/5.1.x
13:03:29.284 INFO - Started HttpContext[/selenium-server/driver,/selenium-server/driver]
13:03:29.286 INFO - Started HttpContext[/selenium-server,/selenium-server]
13:03:29.286 INFO - Started HttpContext[/,/]
13:03:29.383 INFO - Started org.openqa.jetty.jetty.servlet.ServletHandler@b0ce8f
13:03:29.383 INFO - Started HttpContext[/wd,/wd]
13:03:29.404 INFO - Started SocketListener on 0.0.0.0:4444
13:03:29.405 INFO - Started org.openqa.jetty.jetty.Server@192b996
```

现在可以继续执行了。

2. 创建测试

Selenese是Selenium语言，简单且功能强大。你完全可以使用文本编辑器以传统方式编写测试，但是到现在为止，Selenium IDE是最容易的编写测试的方法。可以从工具窗口中启动它。

打开控制面板以后，应该向Base URL字段添加一个地址。对这个地址的要求是，以它为基础的相对链接应对要进行测试的系统可用。IDE控制面板的右下角的按钮上会出现一个红点。而且按钮已经被按下，这意味着工具已处于记录模式。

图18-1显示IDE此时此刻应有的状态。

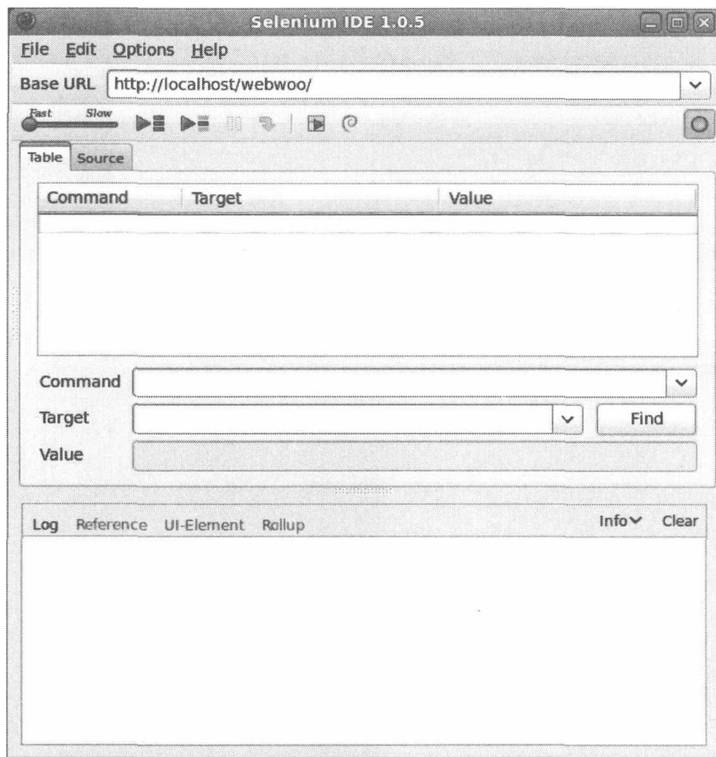


图18-1 Selenium IDE 控制面板

这里，我使用的是基准URL http://localhost/webwoo/。这是已安装的WOO应用实例的地址。我打算从http://localhost/webwoo/?cmd=AddVenue开始测试，所以我在浏览器的地址栏中输入了该URL。转到该地址后，我想先进行一项健全性测试。AddVenue页中包含字符串“no name provided.”，我想通过测试对此进行验证，所以我在浏览器中右击该文本。系统给出一个选项，可以选择Selenium命令`verifyWebText`，如图18-2所示。

同时，Selenium记录了我对该页面的两次访问，以及我对验证文本的请求，如图18-3所示。

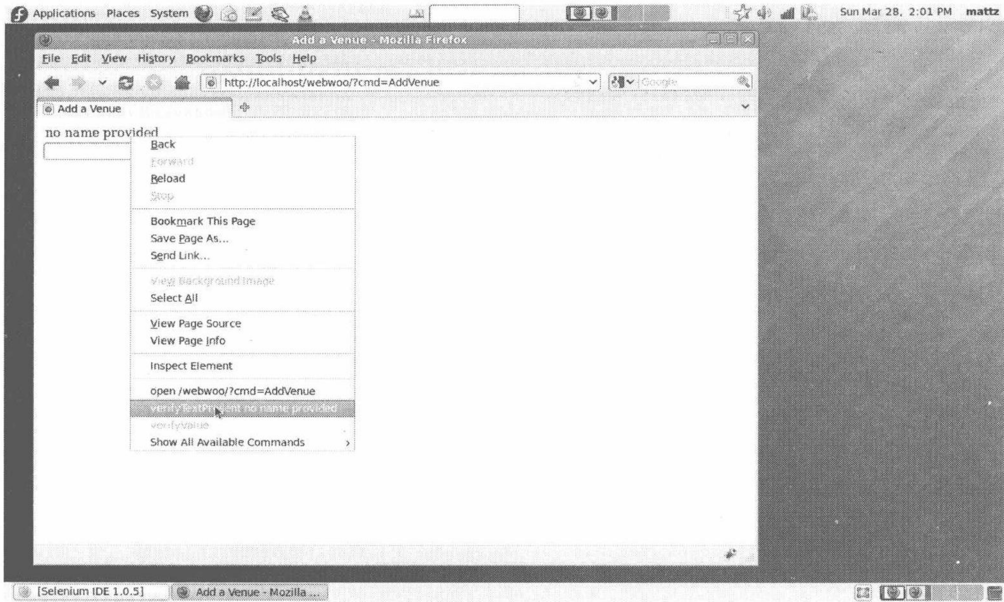


图18-2　验证Web页面上的文本

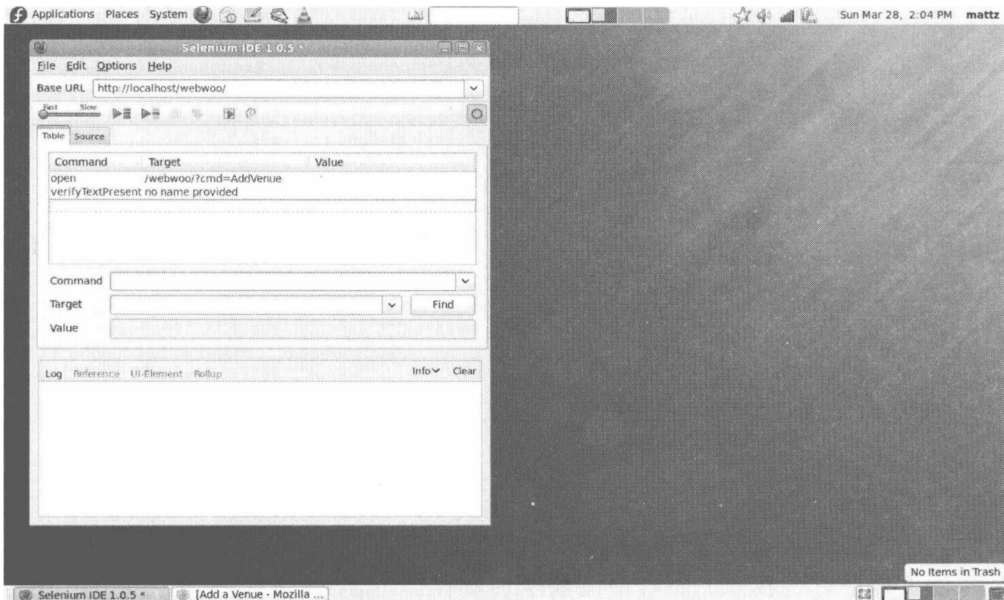


图18-3　Selenium IDE生成测试

注意，每个命令都被分成了3部分：命令、目标和值。划分出来的这几部分又分别称为操作、访问器和断言。实际上，命令稍后会命令测试引擎去某处（访问器）做一些事情（操作），然后

确认结果（断言）。.

现在，我可以返回Woo Web界面，添加一个地点，确认一些文本；添加一个场所，然后再次确认。最后，得到一个可运行的测试用例。我可以单击IDE控制面板中绿色的“播放”按钮，在IDE中运行它。失败的测试命令会用红色标记，通过的测试命令用绿色标记。

现在可以通过File（文件）菜单来保存测试用例，以后再次运行它。你也可以将测试作为PHPUnit类导出。为此，从Options（选项）菜单选择Format（格式），并选择PHPUnit，如图18-4所示。

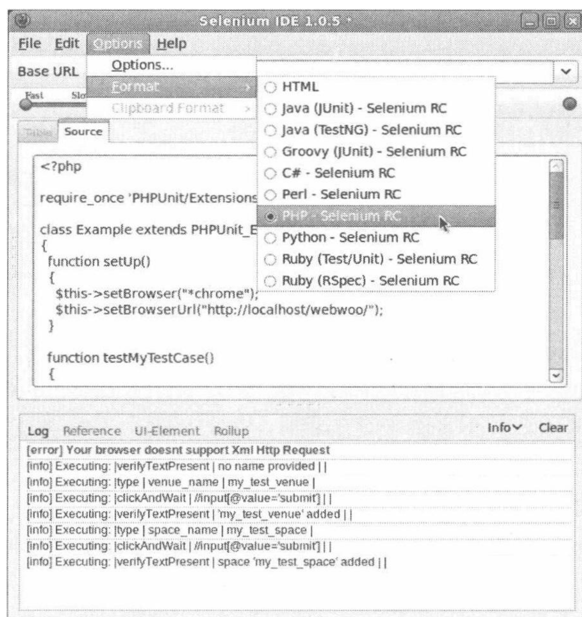


图18-4 修改格式

注意该面板底部的记录面板。你可以看到成功运行测试后生成的报告。因为我已经设置了正确的格式，现在只需保存文件就可以了。你可能已将猜到了，从File（文件）菜单中选择Save as（另存为）。下面是已保存的文件的内容：

```
class Example extends PHPUnit_Extensions_SeleniumTestCase
{
  function setUp()
  {
    $this->setBrowser("*firefox");
    $this->setBrowserUrl("http://localhost/webwoo/");
  }

  function testMyTestCase()
  {
    $this->open("/webwoo/?cmd=AddVenue");
```

```
        try {
            $this->assertTrue($this->isTextPresent("no name provided"));
        } catch (PHPUnit_Framework_AssertionFailedError $e) {
            array_push($this->verificationErrors, $e->toString());
        }
        $this->type("venue_name", "my_test_venue");
        $this->click("//input[@value='submit']");
        $this->waitForPageToLoad("30000");
        try {
            $this->assertTrue($this->isTextPresent("'my_test_venue' added"));
        } catch (PHPUnit_Framework_AssertionFailedError $e) {
            array_push($this->verificationErrors, $e->toString());
        }
        $this->type("space_name", "my_test_space");
        $this->click("//input[@value='submit']");
        $this->waitForPageToLoad("30000");
        try {
            $this->assertTrue($this->isTextPresent("space 'my_test_space' added"));
        } catch (PHPUnit_Framework_AssertionFailedError $e) {
            array_push($this->verificationErrors, $e->toString());
        }
    }
}
```

我已将默认浏览器chrome改成了firefox，除此之外，没有对测试进行任何修改。记住，我不久之前已经启动了Selenium服务器。该服务器必须运行，否则使用Selenium的PHPUnit测试会失败。正是这一服务器启动了浏览器（在本例中是Firefox，也可以使用其他被支持运行测试的大部分现代浏览器）。

保存测试并运行服务器以后，我就可以执行测试用例了：

```
$ phpunit seleniumtest.php
```

```
PHPUnit 3.4.11 by Sebastian Bergmann.
.
Time: 11 seconds, Memory: 4.00Mb
OK (1 test, 3 assertions)
```

如果运行测试，你不但可以看到这个输出，还可以看到服务器调用的浏览器窗口突然弹出，并以极快的速度执行这些操作。这类繁重的工作过去都要手动完成，现在可以自动完成了。

当然，这里只介绍了Selenium的很少一部分内容，但希望能起到抛砖引玉的作用。如果你想了解更多内容，http://seleniumhq.org/docs/index.html 提供了完整的Selenium手册。另外，你还应该看一下PHPUnit网站上的Selenium文档，网址为http://www.phpunit.de/manual/current/en/selenium.html。

18.5　警告

我们有时会过分夸大自动化测试所带来的好处。我在项目中使用PHPUnit进行单元测试，同

时也进行功能测试。也就是说，我在系统级别进行测试，也在类级别进行测试。由此我看到了真实且可见的好处，但我同时认为测试也有一定代价。

测试会增加很多开发成本。例如当加强项目的安全性时，你需要多花时间，这可能会影响项目发布。编写测试要花费时间，而运行测试也要花费时间。在一个系统上，我们可能有多套运行于多个数据库和多个版本控制系统的功能测试。如果添加一些与上下文环境相关的变量，那么我们会面对运行测试套件的真实障碍。当然，测试不运行就毫无意义。这个问题的一个解决方法就是充分自动化测试，这样就可以由类似cron的计划性程序来触发测试。另一个方法则是维护整套测试的一个子集，该子集可被开发者轻松运行，就像提交代码一样。这些子集都应该和运行时间长且缓慢的完整测试放在一起。

另一个要考虑的问题是许多测试用具脆弱的天性。测试也许会让你有信心修改代码，但是当测试覆盖范围随着系统复杂度增长而增长时，一下破坏多个测试变得越来越容易。当然，这常常就是你所要的效果。你也许想要了解预期的行为何时不会发生或者非预期的行为何时发生。

但是，一个测试用具时常会因为一个细微的相关改变而无法工作，如一个反馈字符串的格式的改变。每一个无法正常运行的测试都是紧急事件，但是如果不得不修改30个测试用例来解决一个架构上或者输出的小变动，那会让人觉得很沮丧。单元测试通常不易出现这类问题，因为基本上每个组件的单元测试都是独立的。

你必须权衡保持测试与一个不断发展的系统步调一致的成本。不过基本上，我认为这是值得的。

你也能做一些事情来降低测试用具的脆弱性。将某些可能发生的变化写进测试中便是一个不错的主意。例如，我倾向于使用正则表达式来测试输出而不是直接测试是否相等。这样当在输出字符串中移除一个换行符时不至于使测试失败。当然，测试太容易通过也是一种危险，因此你需要慎重决定。

另一个问题就是你要在什么范围内用mock和stub来虚拟被测试组件之外的系统。有人认为要尽可能地孤立组件并模拟它所需的一切。这在一些项目中可行。但在有些项目中，我发现维护大量的mock会花费过多的时间。不仅因为你要让测试与系统保持一致，而且必须使mock对象始终保持最新。假设改变一个方法的返回类型。如果你未能更新相应stub对象的方法来返回新的类型，那么测试就会出现错误。在一个复杂的模拟系统中，可能会有bug悄悄“爬入”mock对象的真实威胁。调试测试是很困难的工作，特别当系统本身并没有故障的时候。

我倾向于随机应变。默认情况下我都会使用mock和stub，但是如果测试成本开始提高，我会毫不犹豫地迁移到真正的组件上。尽管这也许会减少一些对于测试对象的关注，但是也会带来好处，那就是发生在组件环境的错误至少是系统中真正存在的问题。当然你也可以同时使用真实和虚拟的组件。例如我总是在测试模式中使用一个内存数据库。如果一直使用PDO，这会特别容易。下面是一个简单的类，它使用PDO与数据库交互：

```
class DBFace {
    private $pdo;
    function __construct( $dsn, $user=null, $pass=null ) {
        $this->pdo = new PDO( $dsn, $user, $pass );
```

```
        $this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }

    function query( $query ) {
        $stmt = $this->pdo->query( $query );
        return $stmt;
    }
}
```

如果DBFace被传递到我们系统中并被映射类所使用，那么用它来访问内存模式的SQLite是很简单的事：

```
public function setUp() {
    $face = new DBFace("sqlite::memory:");
    $face->query("create table user ( id INTEGER PRIMARY KEY, name TEXT )");
    $face->query("insert into user (name) values('bob')");
    $face->query("insert into user (name) values('harry')");
    $this->mapper = new ToolMapper( $face );
}
```

你也许已经知道，我并非一个测试的空想家。我通常结合使用真实和虚假的对象来进行“欺骗”，并且由于准备数据是重复性的，我通常将测试装置集中到被Martin Fowler称为Object Mother的类中。这些类是为测试生成预备对象的简单工厂。很多人并不喜欢这种共享装置。

上面已经指出了测试让你不得不面对的一些问题，但我必须重申以下几点来说明测试是值得的。测试可以：

- 帮助预防bug（开发和重构时）；
- 帮助发现bug（扩大测试覆盖范围时）；
- 鼓励你关注于系统设计；
- 使你不再害怕修改代码会导致比它们所解决的问题更多的问题，从而改进代码设计；
- 给你迁移代码的信心。

在为之编写过测试的每一个项目中，我总会在某些情况下因测试而受益。

18.6　小结

本章重温了作为开发者所会编写却也会经常放弃的各种测试，并且介绍了PHPUnit。PHPUnit使你能在开发过程中编写出不被随意丢弃的测试[1]，并且可以将这些测试保存起来并因此受益！我们演示了如何实现一个测试用例，并且介绍了可用的断言方法。我们讲解了约束和mock对象的用法。此外，我们展示了测试重构如何改进你的设计，以及测试Web应用的一些技巧（使用了PHPUnit和Selenium）。最后我冒着可能使某些人不满的风险，警告读者测试需要付出一定的代价，并讨论了如何权衡这些代价。

---

① 传统的测试方式则在测试通过后将测试代码删除。正如本章开头介绍，这是一种浪费。——译者注

第19章 用Phing实现项目的自动构建

如果说版本控制是硬币的一面，那么自动构建就是硬币的另一面。版本控制允许多个开发者合作开发一个项目，而当多个开发者各自独立部署项目时，自动构建就是最重要的技术。某个开发人员开发的程序目录可能位于/usr/local/apache/htdocs，而另一个开发人员的程序可能位于/home/bibble/public_html。不同开发人员可能使用不同的数据库密码、库目录或邮件机制。如果灵活配置代码库，也许可以容忍这些差别，但改变配置并手动复制目录是很麻烦的事——特别当你需要一天（或者一小时）内就重新部署好几次代码的时候。

你已经见过PEAR处理安装的例子。你当然也想通过PEAR包向最终用户交付项目，因为利用该机制最容易实现安装（用户可能已经在系统上安装了PEAR，而且PEAR支持网络安装）。PEAR能完美地处理安装的最后几个阶段，但在创建包之前，还需要使许多工作自动完成。例如，从版本控制库中提取文件。你应该将测试和编译文件运行到构建目录中。最后，自动化PEAR包的创建过程。本章将介绍Phing，phing用于处理这类操作。

本章包括以下内容。

- 获取并安装Phing：谁构建我们的构建器？
- 属性：设定和获取数据。
- 类型：描述项目的复杂部分。
- 目标：根据功能把构建过程分解成可调用的、相互依赖的多个部分。
- 任务：要完成的事。

19.1 什么是 Phing

Phing是一个用于创建[①]项目的PHP工具，它与非常流行而且很强大的Java工具Ant非常相似。Ant之所以如此命名，是因为它与蚂蚁很像，很小但可以完成非常庞大的工作量。Phing和Ant都使用XML文件（通常是build.xml）来决定如何安装项目或者完成其他工作。

PHP确实需要一个安装程序的解决方案。在过去，我们有很多选择。首先，我们可以使用make，它是Unix下的一个build工具，常用于C和Perl项目，然而make的语法非常严格，而且使用make需

① build，指项目安装和部署。——译者注

要了解很多与shell相关的知识，甚至包括脚本编写——这对很多PHP程序员来说相当困难，因为他们很少接触Unix或Linux命令行编程。另外，在常见的创建操作方面（如转换文件名和内容），make提供的内置工具也很少。它只是简单地组合shell命令。使用make来创建项目，我们的程序很难跨平台安装。因为不同系统安装的make版本未必相同，甚至可能没有安装make。即使安装了make，也未必安装了make所必需的各种命令和相关文件（配置文件）。

Phing与make的关系可以从Phing的全名看出：Phing是PHing Is Not Gnu make的缩写。这种递归的命名方式是程序员间常见的小幽默（例如GNU是Gnu is Not Unix的缩写）。

Phing是一个纯PHP的程序，它通过解析用户创建的XML文件来决定如何进行操作。常见的操作包括从一个分发目录复制文件到几个目标目录。当然，Phing还有其他很多功能，它可以生成文档、运行测试、调用命令、执行任意的PHP代码、创建PEAR包、替换文件中的关键词、删除注释以及生成压缩包等。如果你还需要更多功能，也可以自己扩展Phing的功能。

Phing本身是用PHP语言编写的程序，所以你只要安装有PHP引擎就可以运行Phing。因为Phing主要是用来安装PHP应用的应用程序，所以最好使用命令行形式的PHP，这样比较安全稳定。

我们已经知道PEAR包的安装极其简单，而且PEAR支持自动安装机制。因为PEAR和PHP是绑定发布的，那我们为什么不使用PEAR机制来安装项目呢？事实上，PEAR简化了项目安装，且能很好地支持依赖（这样可确保软件包之间的兼容）。

开发中很多麻烦的任务都必须实现自动化，包括包创建。我们可以用Phing来开发项目，然后使用PEAR把项目打包成一个PEAR包来发布。这个方法正是Phing自己发布新版本的方法。

19.2　获取和安装 Phing

如果安装工具本身不容易安装，显然会是件很可笑的事。如果你已经安装了PHP 5或者更高版本（如果还没有的话，这本书显然不适合你），Phing的安装极其简单。

你只需要输入两行命令即可安装Phing：

```
$ pear channel-discover pear.phing.info
$ pear install phing/phing
```

以上命令将把Phing当做一个PEAR包来安装。在Unix或者Linux系统上，需要有PEAR目录的写权限，即要使用root账号来运行以上命令。

如果在安装时遇到问题，也可以直接访问http://phing.info/trac/wiki/Users/Download，该下载页面上有足够的安装说明。

19.3　编写 build 文档

现在你已经准备好了！我们先测试一下：

```
$ phing -v
Phing version 2.4.0
```

Phing命令的-v参数用于指示脚本返回Phing的当前版本信息。在你阅读本书时，版本号可能已经变了，但返回的信息应该是相似的。

现在我们不带任何参数运行phing命令：

```
$ phing
Buildfile: build.xml does not exist!
```

可以看到，Phing提示缺少用于指示操作的文件。默认情况下，Phing会查找名为build.xml的文件。我们创建一个最简单的文件build.xml，让出错信息消失：

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz" default="main">
    <target name="main"/>
</project>
```

以上是build文件中必须包含的内容，如果把以上内容保存为build.xml并且再次运行phing命令，就会得到更多信息：

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml


megaquiz > main:

BUILD FINISHED

Total time: 0.1107 seconds
```

实际上我们没完成什么真正的工作，因为这只是我们的起点。让我们再来看看build文件。因为我们正在处理XML，所以有一个XML声明。你可能知道，XML的注释格式如下所示：

```
<!--本行为XML注释，任何内容都将被忽略-->
```

build文件的第二行将被忽略。在build文件中，你可以写入任意行注释。随着文件越来越多，你会发现使用注释的好处。如果没有合适的注释，很难读懂大的build文件。

任何build文件的第一部分都是project元素，该元素可以包含最多4个属性。其中，name和default属性是必需的。name属性确定了项目名，default属性定义了未通过命令行指定目标时默认执行的目标。可选的description属性可提供摘要信息。我们还可以使用basedir属性指定安装目录。如果没有指定basedir的话，就使用当前的工作目录。这几个属性的相关信息如表19-1所示。

表19-1 project元素的属性

属　性	是否必需	描　述	默 认 值
name	是	项目名称	无
description	否	项目摘要	无
default	是	默认运行的目标	无
basedir	否	执行build的文件系统环境	当前目录（.）

定义了project元素之后，必须至少创建一个目标，然后使用default属性指定该目标为默认执行的目标。

19.3.1　目标

从某种意义上说，目标和函数很相似。目标是达到特定目的的操作的集合：例如复制一个目录到另外一个地方或者生成文档。

在前面的例子中，我们定义了一个最简单的目标：

```
<target name="main"/>
```

可以看出，目标必须至少包含name属性。我们已经在project元素中使用过main这个目标。因为默认目标为main，所以不带命令行参数地运行Phing时，就会调用该目标。我们的输出结果可以证明这一点：

```
megaquiz > main:
```

目标可以依赖于其他目标。通过建立目标间的依赖关系，Phing就可以得知运行各个目标的顺序。我们增加一个依赖关系到build文件中：

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main"
>
    <target name="runfirst" />
    <target name="runsecond" depends="runfirst"/>
    <target name="main" depends="runsecond"/>
</project>
```

可以看到，我们为target元素引入了一个新属性depends。depends告诉Phing，它指定的目标必须在当前目标之前执行，因此在一个目标复制指定文件到目录之后，另一个目标才在那个目录中转换所有文件。我们在代码中增加了两个新的目标——runsecond和runfirst，而main依赖于runsecond，runsecond依赖于runfirst。我们来看看使用这个build文件运行Phing时会发生什么：

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > runfirst:

megaquiz > runsecond:

megaquiz > main:

BUILD FINISHED

Total time: 0.3029 seconds
```

由以上结果可知，依赖关系生效了。Phing遇到main目标时，先查看它的依赖关系，并退回到runsecond。Phing接着发现runsecond依赖于runfirst，因此调用runfirst。在满足了对于runfirst的依赖之后，Phing才调用runsecond。最后，main被调用。depends属性可以一次指定多个目标，其中只要用逗号隔开即可。

设置了多个目标后，就可以在命令行中执行命令以覆盖project元素的default属性：

```
$ phing runsecond
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > runfirst:

megaquiz > runsecond:

BUILD FINISHED

Total time: 0.2671 seconds
```

通过传递目标名称作为命令参数，就忽略了默认目标。与命令行参数相匹配的target将被调用（该target依赖的target也会被调用）。这在调用特殊任务时是很有用的，比如清除一个build目录或者运行安装完成后的脚本。

target元素也支持可选的description（描述）属性，可以用它来简要描述某个目标的目的：

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main"
         description="A quiz engine">
    <target name="runfirst"
            description="The first target" />
    <target name="runsecond"
            depends="runfirst"
            description="The second target" />
    <target name="main"
            depends="runsecond"
            description="The main target" />
</project>
```

增加description属性到目标中并不会影响安装过程。但如果运行Phing时使用了-projecthelp参数，description属性的内容用于总结整个项目：

```
$ phing -projecthelp
Buildfile: /home/bob/working/megaquiz/build.xml
A quiz engine
Default target:
-------------------------------------------------------------------------------
 main       The main target

Main targets:
-------------------------------------------------------------------------------
```

```
main       The main target
runfirst   The first target
runsecond  The second target
```

注意示例中也增加了description属性到project元素中。

19.3.2 属性

Phing允许使用property元素设置值。

属性和脚本中的全局变量很相似。因此，通常在build文件的开头声明属性，这样开发人员才能方便地在build文件的其他地方使用属性的值。下面我们创建一个和数据库信息有关的build文件：

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main"
>

    <property name="dbname"  value="megaquiz" />
    <property name="dbpass"  value="default" />
    <property name="dbhost" value="localhost" />

    <target name="main">
        <echo>database: ${dbname}</echo>
        <echo>pass:     ${dbpass}</echo>
        <echo>host:     ${dbhost}</echo>
    </target>
</project>
```

上面我们引入一个新元素：property。property必须要有name和value属性。注意我们往main目标中增加了任务。echo是一个示例任务。在下一节中我们将更完整地完成这个任务，而现在只要知道echo会完成我们想要的任务就够了——它输出自己的内容。注意我们在这里引用属性值的语法：先写一个美元符号，然后把属性名放在一对花括号中，这样Phing就知道用属性值替换属性名称。

```
${propertyname}
```

这个build文件的目的是定义3个属性并输出它们的值。让我们看看运行结果：

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

     [echo] database: megaquiz
     [echo] pass:     default
     [echo] host:     localhost

BUILD FINISHED
```

```
Total time: 0.4402 seconds
```

掌握了属性的用法之后，我们来进一步探索目标的用法。target元素接受两个附加的属性：if和unless。这两个属性的值为property的名称。当我们使用if和property的名称时，target只能在给定的property已经被赋值时执行；如果property并未定义，则target会默默退出。下面我们把dbpass属性注释掉，并让main任务使用if属性来请求使用dbpass属性：

```
<property name="dbname"  value="megaquiz" />
<!--<property name="dbpass"  value="default" />-->
<property name="dbhost" value="localhost" />

<target name="main" if="dbpass">
    <echo>database: ${dbname}</echo>
    <echo>pass:     ${dbpass}</echo>
    <echo>host:     ${dbhost}</echo>
</target>
```

再次运行phing：

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

BUILD FINISHED

Total time: 0.2628 seconds
```

可以看到，这里没有产生任何错误，但是main任务并没有执行。我们为什么要这样做呢？有另一种办法在项目中设置属性——通过命令行指定属性。用-D参数及之后的属性赋值来告诉Phing你将传递给它一个属性。参数格式应该如下所示：

```
-Dname=value
```

在上例中，如果想要通过命令行使dbname属性有效，可以这样：

```
$ phing -Ddbpass=userset
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

     [echo] database: megaquiz
     [echo] pass:     userset
     [echo] host:     localhost

BUILD FINISHED

Total time: 0.4611 seconds
```

dbpass属性被赋值，则main目标的if属性得到满足，目标main被允许执行。

就像期望的那样，unless属性的作用和if属性的作用相反。如果unless指定的属性已经被

19

定义，则目标将不会运行。如果想从命令行禁止某一个特定目标执行，unless属性就会派上用场。因此可以向main目标添加如下内容：

```
<target name="main" unless="suppressmain">
```

只有suppressmain未定义时，main才会执行：

```
$ phing -Dsuppressmain=yes
```

上面我们已经完整介绍了target元素，下面看一下该元素支持的属性，如表19-2所示。

表19-2　target元素的属性

属　性	是否必需	描　述
name	是	目标名
depends	否	当前目标依赖的目标
if	否	只有给定属性出现时才会执行目标
unless	否	只有给定属性不出现时才会执行目标
description	否	对目标意图的简短摘要

在命令行中设置的属性会覆盖build文件中该属性的定义，而在build文件中，属性值也可以被覆盖：默认情况下，如果属性被声明了两次，则该属性的值以第一次声明为准。但你可以在第二个property元素中设置override属性，这样该属性的值以新设置的属性值为准。如下所示：

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main"
>

    <property name="dbpass"  value="default" />

    <target name="main">
        <property name="dbpass" override="yes" value="specific" />
        <echo>pass:     ${dbpass}</echo>
    </target>

</project>
```

我们设置了dbpass属性，并给它一个初始值"default"。在main目标中，我们再次设置了这个属性，并为该属性增加了值为"yes"的override属性，并提供一个新值。这个新值反映在输出中：

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:
```

```
     [echo] pass:     specific

BUILD FINISHED

Total time: 0.3802 seconds
```

如果在第二个property元素中未设置override属性，则dbpass的值为"default"。值得注意的是目标与函数不同，它没有局部作用域的概念。如果在一个任务中覆盖了一个属性，那么在整个build文件的其他任务中该属性都会被覆盖。当然，你也可以在覆盖之前把一个属性值存放在临时属性中，然后在完成操作后再恢复属性的值。

到现在为止，我们一直通过自定义的方式来处理属性。Phing还可提供内置属性。使用内置属性的方式与使用自定义属性的方式相同，如下例所示：

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
        default="main"
>

    <target name="main">
        <echo>name:     ${phing.project.name}</echo>
        <echo>base:     ${project.basedir}</echo>
        <echo>home:     ${user.home}</echo>
        <echo>pass:     ${env.DBPASS}</echo>
    </target>

</project>
```

上面我们引用了一些内置的Phing属性。phing.project.name的值为project元素中name属性定义的项目名称；project.basedir的值为项目根目录；user.home为执行Phing的用户的home目录（在提供默认安装位置时很有用）。

最后，以env作为前缀的属性是操作系统的环境变量。因此通过指定${env.DBPASS}，我们可以得到名称为DBPASS的环境变量。使用这个build文件运行Phing：

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

     [echo] name:     megaquiz
     [echo] base:     /home/bob/working/megaquiz
     [echo] home:     /home/bob
     [echo] pass:     ${env.DBPASS}

BUILD FINISHED

Total time: 0.1120 seconds
```

注意，最后一个属性${env.DBPASS}未被转换。当Phing没有找到相关属性时，默认显示该属性名。如果我们设置了DBPASS环境变量，那么再次运行时可以在输出中看到变量的值：

```
$ export DBPASS=wooshpoppow
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:
     ...
     [echo] pass:     whooshpoppow

BUILD FINISHED

Total time: 0.2852 seconds
```

现在我们已经知道3种设置属性的办法：property元素、命令行参数和环境变量。

你可以用目标来确保某个属性已被赋值。假设项目需要一个dbpass属性，我们希望用户通过命令行（该方式比其他的属性赋值方法优先级高）设置dbpass。如果用户没有在命令行提供dbpass属性的值，可以尝试使用环境变量。如果在环境变量中也没有找到相应的定义，就使用默认值作为该属性的值：

```
<?xml version="1.0"?>
<!-- build xml -->

<project name="megaquiz"
         default="main"
>

    <target name="setenvpass" if="env.DBPASS" unless="dbpass">
        <property name="dbpass" override="yes" value="${env.DBPASS}" />
    </target>

    <target name="setpass" unless="dbpass" depends="setenvpass">
        <property name="dbpass" override="yes" value="default" />
    </target>

    <target name="main" depends="setpass">
        <echo>pass:     ${dbpass}</echo>
    </target>

</project>
```

像平常一样，默认的main目标首先被调用。Phing发现它依赖于目标setpass，所以Phing退回到setpass目标，接着发现setpass依赖于setenvpass目标，于是Phing再退到setenvpass目标。只有当dbpass属性未定义并且env.DBPASS已经设置时，setenvpass目标才能执行。只有当这些条件都满足时，才可以用property元素来设置dbpass属性。此时dbpass通过命令行参数或者环境变量定义。如果两者均未定义dbpass，则dbpass属性此时仍然为未设置状态。但只有当dbpass未设置时，Phing才会执行setpass目标。此时setpass目标把属性设置为默认的字符串

"default"。

19.3.3 类型

在学习过属性之后，现在我们来看看数据。实际上，Phing支持一系列叫做类型的特定元素，其中包含着项目安装过程中所需的各种有用信息。

1. FileSet

假设你需要在build文件中指定一个目录。当然你可以使用属性来指定这个目录，但是如果开发人员使用了不同的操作系统（它们支持不同的目录分隔符），那么这时就会出现问题。解决办法是使用FileSet数据类型。FileSet是平台无关的，因此如果描述目录路径时使用了斜杠，那么在Windows机器下运行时，它们会自动把斜杠转换为反斜杠。可以像下面这样定义一个最简单的fileset元素：

```
<fileset dir="src/lib" />
```

可以看到，我们使用dir属性来指定目录。你也可以增加id属性，用于以后引用该fileset：

```
<fileset dir="src/lib" id="srclib">
```

在指定要包含或不包含的文件的类型时，FileSet数据类型特别有用。当安装一系列文件时，你可能不想安装某些符合特定模式的文件，那么可以在excludes属性中设置这样的条件：

```
<fileset dir="src/lib" id="srclib"
        excludes="**/*_test.php **/*Test.php" />
```

注意excludes属性中使用的语法。两个星号**表示src/lib的任何目录或子目录。单个星号*表示0到任意个字符。因此在dir属性中定义的目录的所有子目录中，以_test.php或Test.php结尾的文件都被排除。excludes属性可以同时支持多个模式，并且模式间用空格分隔。

includes属性和excludes属性的语法相同。也许我们的src/lib目录包含了多个非PHP文件，对开发人员来说它们是有用的，但是在安装中并不需要。我们可以用excludes来排除这些文件，但仅用includes来定义我们想要包含的文件会更简单。在下面的例子中，如果文件没有以.php结尾就不会被安装：

```
<fileset dir="src/lib" id="srclib"
        excludes="**/*_test.php **/*Test.php"
        includes="**/*.php" />
```

当建立includes和excludes规则后，fileset元素会变得很长。幸好你可以把exclude规则拿出来放到一个exclude子元素中。对于include规则也是如此。现在我们重写FileSet：

```
<fileset dir="src/lib" id="srclib">
    <exclude name="**/*_test.php" />
    <exclude name="**/*Test.php" />
    <include name="**/*.php" />
</fileset>
```

fileset元素的属性如表19-3所示。

表19-3　fileset元素的属性

属　性	是否必需	描　述
id	否	引用元素的唯一句柄
dir	否	fileset目录
excludes	否	排除模式的列表
includes	否	包含模式的列表
refid	否	当前fileset指向的fileset的ID

2. PatternSet

在fileset元素（或其他元素）中构建文件命名模式时，会导致重复使用exclude和include元素。在上一节的例子中，我们定义了测试文件和普通代码文件的模式。随着时间的推移，我们可能要加入越来越多的模式（例如也许我们想增加.conf和.inc文件作为代码文件）。如果其他fileset元素也使用了这些文件模式，就不得不去修改所有fileset元素中的文件命名模式。

把模式集中起来放到patternset元素中可以解决这个问题。patternset元素组合了include和exclude元素，因此可以在其他元素中引用它们。下面，我们从fileset中提取include和exclude元素，并把它们放到patternset元素中：

```
<patternset id="inc_code">
    <include name="**/*.php" />
    <include name="**/*.inc" />
    <include name="**/*.conf" />
</patternset>

<patternset id="exc_test">
    <exclude name="**/*_test.php" />
    <exclude name="**/*Test.php" />
</patternset>
```

上面我们创建了两个patternset元素，把id属性分别设置为inc_code和exc_test。inc_code的include元素用于包含代码文件，exc_test的exclude元素则用于排除测试文件。现在，我们可以在fileset中使用patternset元素：

```
<fileset dir="src/lib" id="srclib">
    <patternset refid="inc_code" />
    <patternset refid="exc_test" />
</fileset>
```

为了引用一个已经存在的patternset，必须使用另外一个patternset元素。第二个元素必须设置一个refid属性。refid属性说明了在当前环境中你想使用的patternset元素的id。我们可以用这种方式重复使用同一个patternset元素：

```
<fileset dir="src/views" id="srcviews">
    <patternset refid="inc_code" />
</fileset>
```

如果修改了名为inc_code的patternset，则所有改动都会影响到使用它的元素。通过使用FileSet，可以把exclude规则放到excludes属性或一系列exclude子元素中。include规则也是

如此。

patternset元素的属性可参考表19-4。

表19-4 patternset元素的属性

属　性	是否必需	描　述
id	否	引用元素的唯一句柄
excludes	否	排除文件的规则
includes	否	包含文件的规则
refid	否	当前patternset指向的patternset的ID

3. FilterChain

以上介绍了选择文件的机制。FilterChain（过滤链）则是一个灵活转换文本文件内容的机制。

与所有的类型一样，只定义一个filterchain元素并不起作用。filterchain元素和它的子元素都必须包含一个task（任务）元素。task元素用于告诉Phing执行一系列动作。后面我们将介绍task元素。

使用一个filterchain元素可以把任意数目的过滤器组合在一起。文件过滤器就像一个管道，第一个过滤器过滤完文件，把结果传递给第二个过滤器，依次类推。把多个过滤器合并到一个filterchain元素中，可以灵活地转换文件内容。

让我们创建一个filterchain，它可以删除传递给它的任何文本中的PHP注释：

```
<filterchain>
    <stripphpcomments />
</filterchain>
```

StripPhpComments任务用于删除PHP注释。如果在源代码中编写了详细的API文档，会使开发变得容易，但是也会使项目变得臃肿。因为所有的工作都发生在源目录中，所以应当在安装程序时剔除注释。

注解 如果使用一个自动安装工具（build tool），必须要确保没有人修改过已安装的代码。安装程序将会覆盖任何修改过的文件，导致所有修改都失效。我曾经见过这样的情形。

让我们预先使用下一节才介绍的内容，把filterchain元素放到一个任务内：

```
<target name="main">
    <copy todir="build/lib">
        <fileset refid="srclib"/>
        <filterchain>
            <stripphpcomments />
        </filterchain>
    </copy>
</target>
```

Copy是最常用的任务之一，用于从一个地方复制文件到另一个地方。可以看到，我们在todir属性中定义了目标目录。文件源则由我们在前一节中创建的fileset元素定义。之后是filter-

chain元素，任何由Copy任务复制的文件都会执行这个转换。

Phing支持多种过滤器，其功能包括剔除换行符（StripLineBreaks）和用空格替换制表符（TabTo Spaces），甚至还有用XSLT转换源文件的XsltFilter。不过最常用的过滤器可能是ReplaceTokens，它允许我们将源代码中特定变量的值替换为build文件中定义的属性的值。属性也可能从环境变量中得到或由命令行得到。这个功能在定制安装时非常有用。为了方便地浏览项目中的变量，可以把标记都集中到一个配置文件中。

ReplaceTokens可以接受两个属性——begintoken和endtoken，可以用它们来指定变量的边界符。如果不提供这两个属性，Phing就会默认变量的边界符为@。为了识别和替换标记，必须增加token元素到replacetokens元素。下面在例子中增加replacetokens元素：

```
<copy todir="build/lib">
        <fileset refid="srclib"/>
        <filterchain>
            <stripphpcomments />
            <replacetokens>
                <token key="dbname" value="${dbname}" />
                <token key="dbhost" value="${dbhost}" />
                <token key="dbpass" value="${dbpass}" />
            </replacetokens>
        </filterchain>
    </copy>
```

可以看到，token元素需要指定key和value属性。下面看一下在项目中运行这一任务的效果。原始文件还在源目录src/lib/Config.php中：

```
/**
 * 临时而粗糙的Config类
**/
class Config {
    public $dbname ="@dbname@";
    public $dbpass ="@dbpass@";
    public $dbhost ="@dbhost@";
}
```

main目标中包含了Copy任务，运行之后会得到如下输出：

```
$ phing

Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > main:

     [copy] Copying 8 files to /home/bob/working/megaquiz/build/lib
[filter:ReplaceTokens] Replaced "@dbname@" with "megaquiz"
[filter:ReplaceTokens] Replaced "@dbpass@" with "default"
[filter:ReplaceTokens] Replaced "@dbhost@" with "localhost"

BUILD FINISHED

Total time: 0.1413 seconds
```

我们没有改变原始文件，但是通过Copy任务在build/lib/Config.php复制了原始文件：

```
class Config {
    public $dbname ="megaquiz";
    public $dbpass ="default";
    public $dbhost ="localhost";
}
```

我们不仅删除了注释，还将文件中变量的值替换为build文件中属性的值。

19.3.4 任务

任务是build文件中的元素。如果没有任务，很多事情都无法完成，这就是我们在上一节中提前使用task元素的原因。在此我们再详细介绍一下task元素。

1. Echo

Echo任务的作用在最基本的“Hello World”例子中已经得到了完美体现。在实际项目中，我们可以用它来告诉用户你将要做什么或者你已经做了什么，也可以通过显示属性的值来检查安装过程是否正常。可以看到，echo元素开始标签到结束标签间的文本都会被输出到浏览器：

```
<echo>The pass is '${dbpass}', shhh!</echo>
```

另外，你也可以把想输出的内容放到一个msg属性中：

```
<echo msg="The pass is '${dbpass}', shhh!" />
```

这样做的效果和把内容放在<echo></echo>标签内部是一样的：

```
[echo] The pass is 'default', shhh!
```

2. Copy

Copy（复制）是安装过程中最主要的工作。一般来说，你可以创建一个目标，负责从源目录中复制文件并把这些文件集中到临时安装目录中，这就是构建过程。然后设置另外一个目标，负责复制这些经过组合（并且已转换的内容）的文件到它们的输出位置，这就是安装过程。把构建过程和安装过程分开来并不是必需的，但分开这两个过程便于你在修改了源代码后检查结果，而且你可以修改临时目录中的某些文件并安装程序到另一个地方，而不用再次从源代码目录复制一遍所有文件（这非常耗时）。

最简单的Copy任务只需指定一个源文件和一个目标目录（或目标文件）：

```
<copy file="src/lib/Config.php" todir="build/conf" />
```

如上所示，我们用file属性来指定源文件。你可能已经很熟悉todir属性，它用于指定目标目录。如果目标目录不存在，Phing就会为你自动创建。

如果需要指定目标文件，而不是包含目录，可以使用tofile属性来代替todir：

```
<copy file="src/lib/Config.php" tofile="build/conf/myConfig.php" />
```

同样，如果需要的话可以创建build/conf目录，但是这次Config.php被重命名为myConfig.php。

可以看到，如果想一次复制多个文件，需要增加一个fileset元素到Copy任务中：

```
<copy todir="build/lib">
    <fileset refid="srclib"/>
</copy>
```

源文件由fileset元素srclib定义，因此我们只要在copy中设置todir属性即可。

Phing会很智能地检查在目标文件被创建之后源文件是否被修改过。如果没有任何改变，Phing不会执行复制。也就是说，你可以构建很多次，但只有修改过的文件才会被重新安装。只要其他东西不改变，这样做很好。如果有些文件被根据配置的replacetokens元素对内容进行转换，你可能想要确保文件在每次调用Copy任务时都进行内容转换。你可以通过设置overwrite属性来达到目的：

```
<copy todir="build/lib" overwrite="yes">
    <fileset refid="srclib"/>
    <filterchain>
        <stripphpcomments />
        <replacetokens>
            <token key="dbpass" value="${dbpass}" />
        </replacetokens>
    </filterchain>
</copy>
```

现在无论什么时候运行Copy任务，符合fileset元素的文件都会被替换，无论源文件是否有更新。

copy元素的相关信息请参见表19-5。

表19-5　copy元素的属性

属　性	是否必需	描　述	默认值
todir	是（如果没有使用tofile）	复制文件到该目录	无
tofile	是（如果没有使用todir）	复制文件为该文件	无
file	否	源文件	无
tstamp	否	是否根据文件的时间戳判断文件更新	false
includeemptydirs	否	是否复制空目录	false
overwrite	否	如果目标文件存在的话，是否覆盖	no

3. Input

要输出内容给用户，可以使用echo元素。要收集用户输入的信息，可以使用获取命令行变量和环境变量的方法。但是，这些方法都无法实现交互。

注解　*允许用户在安装程序时设置变量是为了在切换构建环境时更加灵活。对于数据库密码来说，另一个好处是不用将这些敏感数据放到build文件中。当然，安装完成之后，密码还是会被存入一个源文件中，因此开发者可以视情况决定如何保障系统安全。*

input元素允许我们向用户输出一个提示消息。Phing会等待用户输入信息并把该信息赋值给一个属性：

```
<target name="setpass" unless="dbpass">
    <input message="You don't seem to have set a db password"
           propertyName="dbpass"
           defaultValue="default"
           promptChar=" >" />
</target>

<target name="main" depends="setpass">
    <echo>pass:     ${dbpass}</echo>
</target>
```

我们再次拥有一个默认目标：main。它依赖于另外一个目标setpass，setpass负责保证dbpass属性已经定义。最后，我们使用了target元素的unless属性，它可以确保target在dbpass属性被设置的时候不运行。

setpass目标只有一个input任务元素。input元素可以有message属性，该属性可以包含对用户的提示信息。propertyName属性是必需的，它指定了用户输入的内容赋值给哪个变量。如果用户在看到提示的时候没有输入内容就直接按了Enter键，该变量的值将被设置为defaultValue属性的值。最后，你可以使用promptChar属性定制提示符——它可以给用户输入数据提供一个可视化的提示。现在使用前面的目标运行Phing:

```
$ phing
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > setpass:

You don't seem to have set a db password [default] > mypass

megaquiz > main:

     [echo] pass:     mypass

BUILD FINISHED

Total time: 6.0322 seconds
```

input元素的相关信息请参见表19-6。

表19-6 input元素的属性

属 性	是否必需	描 述
propertyName	是	将被赋值的属性
Message	否	提示信息
defaultValue	否	如果用户未输入，指派给属性的值
validArgs	否	可接受输入值列表，之间由逗号分隔。如果用户输入的值不在这个列表中，Phing就会重新显示提示信息
promptChar	否	用户应该提供输入的命令行提示符

4. Delete

安装通常就是创建、复制和转换文件，但有时也要用到删除。比如我们想要执行全新安装时，

就需要删除无关的文件。前面介绍过，只有在上次安装之后修改过的源文件才会被从源目录复制到目标目录。通过删除build目录，就可以确保每次都重新复制文件。

删除目录的代码如下：

```
<target name="clean">
    <delete dir="build" />
</target>
```

当使用clean（目标名）运行phing时，就会调用delete任务元素。下面是Phing的输出结果：

```
$ phing clean
Buildfile: /home/bob/working/megaquiz/build.xml

megaquiz > clean:
   [delete] Deleting directory /home/bob/working/megaquiz/build

BUILD FINISHED
```

delete元素也可以带一个file属性，该属性用于指向某个特定的文件。另外，你也可以添加一个fileset子元素到delete元素中来删除多个文件。

19.4　小结

正式的项目开发很少在一个地方同时进行。开发中的代码库需要与安装版本的代码库分开，这样正在进行开发的代码才不会污染正式产品的代码。版本控制允许开发人员获取一个项目的代码并在自己的地方使用，而这要求他们能够轻松地配置自己的项目环境。最后也是最重要的，用户（可能是你自己，可能一年后你已经忘记了代码的细节）应该在看完Read Me文件之后就知道如何安装项目。

在本章中，我们介绍了Phing的基础知识。Phing是一个神奇的工具，它把Apache Ant的大量功能引入到了PHP中。我们只了解了Phing的基本功能，但一旦你使用了本章介绍的目标、任务、类型和属性，就会发现使用其他高级特性也很容易，例如创建文件压缩包、自动生成PEAR包和从构建文件中直接执行PHP代码等。

如果Phing仍不能满足构建要求，那也没有关系，因为Phing是可以扩展的（正如Ant那样）——你可以自己扩展Phing，创建自己特有的任务。即使你不想扩展Phing，也应该花点时间研究一下Phing的源代码。Phing完全使用面向对象的PHP编写，其代码充满了各种设计示例。

Part 5 第五部分

结论

本部分内容

- 第 20 章　持续集成
- 第 21 章　对象、模式与实践

第20章

持续集成

前几章已经介绍了很多工具，设计这些工具的目的是用来支持管理有序的项目。单元测试、文档、构建和版本控制都出奇地有用。但工具，特别是测试工具，可能有些麻烦。

即使只需花几分钟即可运行测试，你也常常会过于关注编写代码而忽略测试。不仅如此，你的客户和同事一直都在等待新的功能，继续编写代码的诱惑一直都在，但bug刚产生时会更容易修正。因为更容易弄清楚是哪个修改导致了这个问题，因而就能更快地解决问题。

本章将介绍CI（Continuous Integration，持续集成），它是将测试和构建进行自动化的一种实践，因而本章还会将近几章介绍的工具和技术结合起来。

本章包括如下内容。

- 定义CI。
- 为CI准备项目。
- CI服务器CruiseControl。
- 使用phpUnderControl为PHP专门定制CruiseControl。
- 定制CruiseControl。

20.1 什么是持续集成

在过去，当你完成了所有前期工作之后，才会进行集成。集成过程中你会发现仍有大量的工作要做。集成就是将项目的所有部分打包，以便进行发布和部署。集成过程一点也不好玩，实际上非常难。

集成还与QA联系密切。如果产品不符合发布要求，那么就不能发布。这就需要进行测试，进行大量的测试。如果在集成阶段之前没怎么进行过测试，可能会遇到很多难以想象的情况。

第18章提到过，早测试、勤测试才是最好的做法。我们也从第15章和第19章了解到，在设计项目的一开始就应该考虑到将来的部署。大多数人都知道这是最理想的做法，但有多少次真正这么做了？

如果你进行的是面向测试的开发（我更喜欢这样称呼测试先行的开发，因为它能更好地反映出我见过的大多数优秀项目的真实情况），那么编写测试并没有你想象得那么难。毕竟你是一边

编写代码一边编写测试。每次开发组件的时候，你都会写一段代码（可能是在类文件的底部），用来实例化对象，调用这些对象的方法。如果你将这些一次性的、零散的代码（写这些代码的目的是为了在开发期间考察组件）收集起来，这样就得到了一个测试用例。将这些代码放在一个类中，并将其添加到你的测试套件中即可。

很奇怪，通常人们不会运行测试。随着时间的推移，测试要花更长时间运行。与已知问题相关的失败会慢慢出现，这样就很难诊断新问题。此外，你也会怀疑是其他人提交的代码导致测试失败，当你解决其他人引入的问题时，就没有时间继续做自己的工作了。因此，最好运行几个与你的工作相关的测试，而不是针对整个套件进行测试。

运行测试失败，并修正测试暴露出来的问题，会把问题搞得越来越复杂。消灭bug过程中的最耗费时间和精力的地方通常在于诊断而非解决。通常，寻找测试失败的原因可能要花几个小时，而解决问题只需要几分钟。不过，如果在提交后的几分钟内或几小时内测试失败，你可能更容易知道应该去哪儿寻找问题。

构建软件也会遇到类似的问题。如果不经常安装项目，你可能会发现在开发机（development box）上一切都运行得很好，但安装时却会运行失败，显示出一条很难理解的错误消息。距离最初构建的时间越久，可能就越难找出失败的原因。

通常很多失败原因都很简单：未声明对系统中库的依赖或是忘记签入（check in）某些类文件。如果你就在旁边，这些问题都很容易解决，但如果你恰巧不在办公室的时候构建失败，那该怎么办呢？不管是哪个倒霉的团队成员接手构建和发布项目的工作，他都不知道你的设置，不可能轻易找到这些没有签入的文件。

项目中涉及的人员越多，集成问题也就越大。你可能很欣赏并且尊重所有的团队成员，但他们不运行测试的可能性比你大得多。他们会在星期五的下午4点，就在你即将宣布项目可以发布的那一刻，提交一周的开发工作。

持续集成会将构建和测试的过程自动化，因而减少了此类问题。

CI是实践和工具的集合。作为一项实践，CI要求频繁提交项目代码（至少每天提交一次）。对于每一次提交，都必须运行测试并构建每一个包。你已经见过CI所需的一些工具，特别是PHPUnit和PEAR。但仅有工具是不够的，还需要高级一些的系统来调节并自动化这个过程。

如果没有这个高级系统（CI服务器），CI实践也会因我们避免麻烦的天性而走样。毕竟，相对来说我们更喜欢编写代码。

有这样一个系统有3个明显的好处。首先，该系统会经常构建和测试项目。这也是CI的终极目标和优势。而且这个自动的过程又引入了新的维度。测试和构建与开发并不在一条线上。CI发生在幕后，不需要你停下手头的工作就能运行测试。和测试一样，CI也鼓励好的设计。为了能使CI实现远程自动安装，你从一开始就应该考虑让安装更容易。

我遇到过这种情况不知道有多少次了：项目的安装过程好像很神秘，只有很少几个开发人员知道。“你的意思是你没有设置URL重写？”一个老手有些蔑视地问。“说实话，你知道Wiki里面有重写规则，只要复制粘贴到Apache配置文件中就可以了。”

开发时记得使用CI，这意味着系统能更容易测试和安装。这也可能意味着要提前多做一些工

作，但这些工作可以使接下来的过程变得容易很多。

因此，我们一开始得先把一些代价太大的基础工作停下来。实际上，在接下来的大多数小节中，你会发现你已经见过这些准备步骤了。

准备好 CI 项目

当然，首先我需要一个能持续集成的项目。现在，我不愿意重新编写测试，所以我要找一些包含测试的代码。最显而易见的候选项目就是我在第18章中介绍PHPUnit时创建的项目I。我打算将其命名为userthing，因为它与一件事有关，其中包含一个`User`对象。

首先，图20-1是我的项目目录。

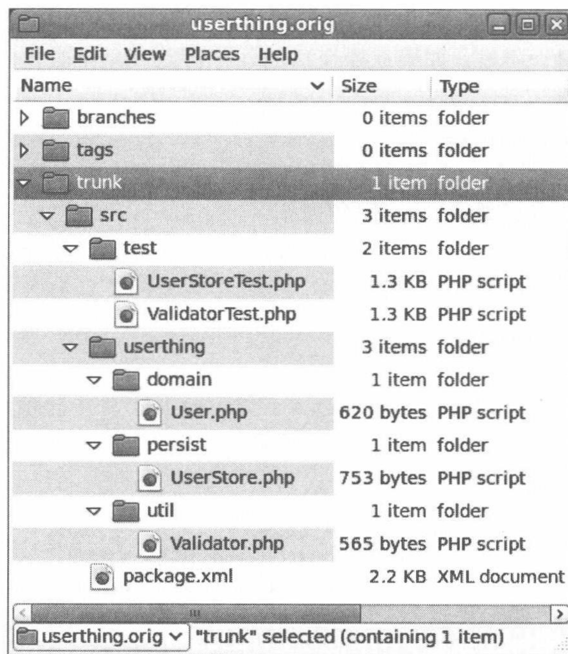


图20-1　示例项目的一部分，用来说明CI

可以看到，我稍稍整理了一下这个结构，向其中添加了一些包目录。在代码中，我通过使用命名空间实现了对包结构的支持。

有了一个项目后，我应该将其添加到版本控制系统中。

1. CI和版本控制

版本控制是CI的核心。CI系统需要获得最新的项目版本，而无需人为干预（至少在设置完了之后）。

你可能已经注意到了，我将`userthing`的代码移入了名为trunk的目录中。这是因为我打算将项目导入到Subversion中，并且branches、tags和trunk目录都是该系统中的约定。

下面是导入指令：

```
$ svn import userthing.orig/ file:///var/local/svn/userthing -m'first import'
```

下面是签出指令：

```
$ svn checkout file:///var/local/svn/userthing/trunk userthing
```

第17章已经详细介绍了Subversion。

既然已经有了项目的本地版本，那么我打算将工作目录改为src/，以试用我的CI实现将用到的各种工具：

```
$ cd userthing/src
```

2. 单元测试

单元测试是持续集成的关键。成功构建一个包含故障代码的项目并不是什么好事。我在第18章中介绍PHPUnit时介绍过单元测试。也许你还没有看到那一章，最好先安装这个重要的工具：

```
$ pear channel-discover pear.phpunit.de
$ pear install phpunit
```

在第18章中，我还为userthing项目的一个版本编写了测试，这一章会用到这个项目的代码。下面再次运行单元测试，以确保重新组织并没有破坏新内容。

```
$ phpunit test
```

```
PHPUnit 3.4.11 by Sebastian Bergmann.
.....
Time: 0 seconds, Memory: 4.50Mb
OK (5 tests, 6 assertions)
```

这里，我引用文件系统来调用测试。我将test目录作为参数传递给PHPUnit，它会自动搜索测试文件。但是，phpUnderControl更倾向于引用单个类来运行测试，稍后你将看到这个CI工具。要满足这个要求，可以添加一个测试套件类。下面是UserTests.php：

```
require_once 'PHPUnit/Framework.php';
require_once 'test/UserStoreTest.php';
require_once 'test/ValidatorTest.php';


class UserTests {

    public static function suite() {
        $suite = new PHPUnit_Framework_TestSuite();
        $suite->addTestSuite('UserStoreTest');
        $suite->addTestSuite('ValidatorTest');

        return $suite;
    }
}
```

注解 在本例中，我将测试类保存在全局命名空间中。其中测试和要测试的组件之间的关系非常密切或是一对一的关系。但是，将每个测试类放在相同的目标命名空间或平行的目录结构中，看上去会整洁一些。这样，你看一眼就能通过测试的命名空间及其文件的位置，判断出测试及其测试对象之间的关系。

利用PHPUnit_Framework_TestSuite类可以将单个测试用例收集到测试套件中。在命令行中调用它：

```
$ phpunit test/UserTests
```

```
PHPUnit 3.4.11 by Sebastian Bergmann.
.....
Time: 1 second, Memory: 4.50Mb
OK (5 tests, 6 assertions)
```

3. 文档

透明性是CI的原则之一。因此，当你在持续集成环境中查看构建时，能够检查文档是否是最新的，其中是否包括最新的类和方法是很重要的。第16章已经介绍了phpDocumentor，因此我们已经像下面这样运行过安装：

```
pear upgrade PhpDocumentor
```

我还是运行这个工具，确认一下：

```
$ mkdir docs
$ phpdoc --directory userthing --target docs/
```

这会生成一些空文档。在CI服务器上发布该文档后，我一定会补上这些文档。

4. 代码覆盖率

如果没有对你编写的代码应用测试，那么依赖于测试并没有什么好处。PHPUnit具有报告代码覆盖率的功能。下面是从PHPUnit的使用信息中提取出来的内容：

```
  --coverage-html <dir>    Generate code coverage report in HTML format.
  --coverage-clover <file> Write code coverage data in Clover XML format.
  --coverage-source <dir>  Write code coverage / source data in XML format.
```

要使用这个功能，必须安装Xdebug扩展，要了解Xdebug的更多信息，可查看http://pecl.php.net/package/Xdebug（安装信息可在http://xdebug.org/docs/install上找到）。也可以使用Linux发布版的包管理系统直接安装。这个包管理系统可以在Fedora 12上运行，例如：

```
$ yum install php-pecl-xdebug
```

下面运行PHPUnit，同时启用报告代码覆盖率的功能：

```
$ mkdir /tmp/coverage
$ phpunit --coverage-html /tmp/coverage test
```

```
PHPUnit 3.4.11 by Sebastian Bergmann.
.....
Time: 0 seconds, Memory: 5.25Mb
OK (5 tests, 6 assertions)
Generating code coverage report, this may take a moment.
```

现在可在浏览器中看到报告，如图20-2所示。

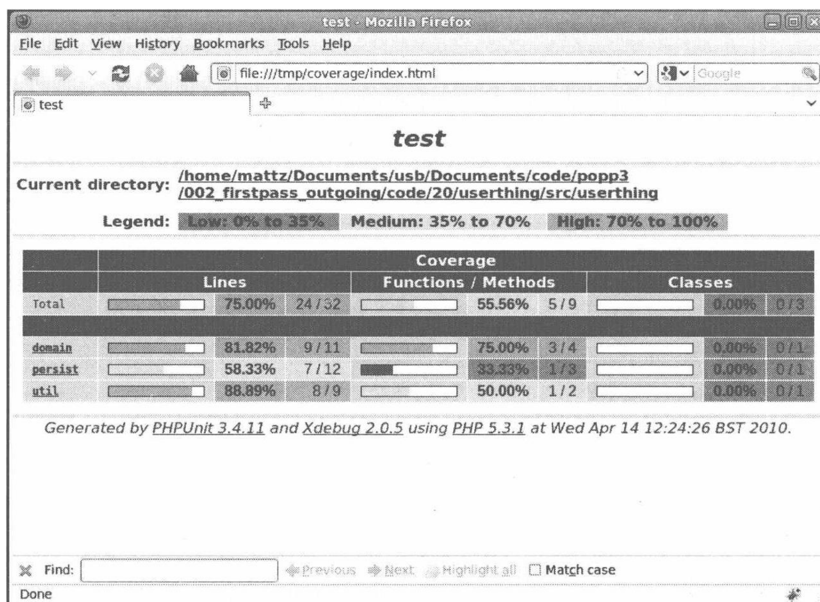


图20-2 代码覆盖率报告

实现完全覆盖和充分地测试一个系统并不是一回事，这一点很重要。另一方面，了解测试中的空白是好事。从图20-2中可以看到，我还有一些工作要做。

5. 编码标准

把大括号放在哪儿最合适，是否要使用Tab键或空格进行缩进，如何命名私有变量，我可以为这些问题争论上一整天。如果能用工具来实现我自己的样式，岂不是很好？使用PHP_CodeSniffer就可以实现。CodeSniffer可以将一套编码标准应用到项目上并生成报告，告诉你存在哪些问题。

听起来这似乎很令人讨厌。实际上也可能如此。但这种工具也有一些合理且颇具技巧的用法，接下来我会介绍这些用法，但首先要试用一下这个工具。先进行安装：

```
$ sudo pear install PHP_CodeSniffer
```

现在我打算将Zend编码标准应用到我的代码中：

```
$ phpcs --standard=Zend userthing/persist/UserStore.php
```

```
FILE: ...userthing/src/userthing/persist/UserStore.php
--------------------------------------------------------------------------------
FOUND 10 ERROR(S) AND 0 WARNING(S) AFFECTING 8 LINE(S)
--------------------------------------------------------------------------------
  6 | ERROR | Opening brace of a class must be on the line after the definition
  7 | ERROR | Private member variable "users" must contain a leading underscore
  9 | ERROR | Opening brace should be on a new line
 13 | ERROR | Multi-line function call not indented correctly; expected 12
    |       | spaces but found 16
...
```

很明显，我必须调整代码样式，才能将代码提交给Zend。

但是，一个团队也可以定义自己的编码指南。实际上，从一开始就遵循公共标准可能比选择使用哪套规则更加重要。如果代码库是一致的，它就更易读，因此更容易使用。例如，命名规范可帮助阐明变量或属性的作用。

编码规范能降低安全风险，降低代码中出现bug的几率。

但这也是危险区域。很多编码样式决策都太过主观，而且人们会不同程度地坚持自己的做事方式。利用CodeSniffer可以定义自己的规则，因此建议你让整个团队理解并接受这套规则，这样就不会有人觉得他们的编程过程变成噩梦了。

自动化工具的另一个好处就是客观。如果你的团队一定要制定一套编码规范，那么使用严格的脚本来修改你的样式，当然比让一位严肃的团队成员来做更好。

6. PHP代码浏览器

你可能比较钟情于令人激动的IDE，或者像我一样，更喜欢用vi做编辑。不管使用哪种方式，当查看的报告告诉你编码方式多么糟糕，更重要的是当你试着理解失败的测试，能够立即终止代码是非常不错的功能。这也正是PHP_CodeBrowser包的功能。

这是很前沿的代码，要安装的话，你就要告诉PEAR你做好了接受alpha版的准备：

```
$ sudo pear config-set preferred_state alpha
```

```
config-set succeeded
```

然后就可以安装了。

```
$ pear install --alldeps phpunit/PHP_CodeBrowser
```

```
downloading PHP_CodeBrowser-0.1.2.tgz ...
Starting to download PHP_CodeBrowser-0.1.2.tgz (76,125 bytes)
.................done: 76,125 bytes
install ok: channel://pear.phpunit.de/PHP_CodeBrowser-0.1.2
```

如果一切进展顺利，你就可以使用命令行工具phpcb了。我打算在源代码中使用它。phpcb希望对PHPUnit生成的日志文件有访问权限，所以首先要运行测试。

```
$ mkdir log
$ phpunit --log-junit log/log.xml test/
```

现在就可以运行phpcb了：

```
$ mkdir output
$ phpcb --log log --output output/ --source userthing/
```

这会将文件写入output目录。图20-3显示了输出结果，打开生成的index.html文件就可以在浏览器中查看输出结果了。

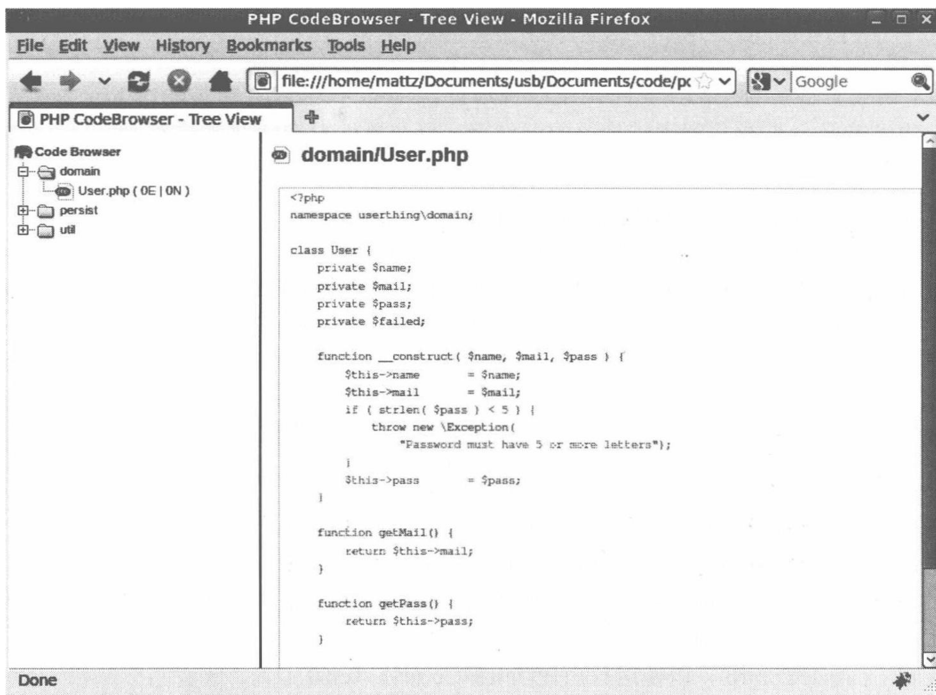


图20-3 PHP代码浏览器

7. 构建

当能够就地访问代码时，你也应该检查，看看是否能构建和部署包。为此，我在包中包含了package.xml文件。下面测试构建和安装过程：

```
$ pear package
```

```
Analyzing userthing/domain/User.php
Analyzing userthing/util/Validator.php
Analyzing userthing/persist/UserStore.php
Warning: in UserStore.php: class "UserStore" not prefixed with package name "userthing"
Warning: in Validator.php: class "Validator" not prefixed with package name "userthing"
Warning: in User.php: class "User" not prefixed with package name "userthing"
```

```
Warning: Channel validator warning: field "date" - Release Date "2010-03-07" is not today
Package userthing-1.2.1.tgz done
Tag the released code with `pear svntag package.xml'
(or set the SVN tag userthing-1.2.1 by hand)
```

有些警告有点过时了，因为我的类使用命名空间而不是包名作为前缀的命名规范。不过，构建成功了，现在开始部署。

```
$ pear install --force userthing-1.2.1.tgz
```

```
install ok: channel://pear.appulsus.com/userthing-1.2.1
```

因此，现在有很多有用的工具可用来监视项目。当然，我很快就会对运行这些工具失去兴趣。实际上，我可能又会重新回到集成阶段的旧观点：只在接近发布的时候才拿出这些工具，到那时它们作为早期警告系统的功效就失去意义了。我需要CI服务器来为我运行这些工具。

20.2　CruiseControl 和 phpUnderControl

CruiseControl是一个用Java编写的持续集成服务器，由ThoughtWorks（Martin Fowler就职于该公司）于2001年发布。经过完全重写的2.0版已于2002年末发布。

根据配置文件（config.xml）中的指令，CruiseControl会为它所管理的项目启动一个构建循环（build loop）。对于每个项目，这个过程都需要几步来完成，Ant生成文件（记住，Ant是最初的Java工具，Phing就是在它的基础上建立的）中定义了所需的具体步骤。运行构建之后，CruiseControl会再次根据配置文件调用工具来生成报告。

构建的结果可以在Web应用中查看，该应用是CruiseControl的“代言人”。

可以配置CruiseControl来运行我们想要的任何工具并生成报告。毕竟，设计CruiseControl的目的就是将任意数量的测试和构建系统组合起来。这需要大量工作。我确信你希望能有这样一种现成的工具，它已经将你所见过的一些PHP工具合并到一起了。phpUnderControl就提供了这项功能。它定制了CruiseControl，以便运行PHPUnit和CodeSniffer等工具，并将它们的报告合并到Web界面中。

使用phpUnderControl之前，必须安装CruiseControl。

注解　为什么使用CruiseControl？因为CruiseControl由ThoughtWorks开发，得到了很好的构建并且具备优良的“血统”。它是免费且开源的。针对支持使用PHP进行集成的工具，其开发活动正如火如荼地进行着。实际上，其中的很多工具都可从phpunit.de网站下载，对于持续集成的支持和互操作性来说，这是好兆头。该网站中还有很多CI服务器的解决方案。如果你正在寻找本机PHP实现，一定要看一下Xinc（http://code.google.com/p/xinc/）。

20.2.1　安装 CruiseControl

CruiseControl是一个Java系统，因此需要安装Java。具体的安装过程因系统而异。在Fedora

发布版上安装时，可能需要像下面这样做：

```
yum install java-1.6.0-openjdk-devel
```

在Debian系统中，应该这样做：

```
sudo apt-get install sun-java6-jdk
```

另外，可以直接从www.java.com下载Java。

确认安装了Java之后（如果没有安装的话，Java网站会告诉你），需要下载CruiseControl。通过http://cruisecontrol.sourceforge.net/download.html可下载最新版本。最终你会得到一个名称类似于cruisecontrol-bin-2.8.3.zip的文件。现在你可以将目录移动到中心位置，然后启动CruiseControl脚本。

```
$ unzip cruisecontrol-bin-2.8.3.zip
$ mv cruisecontrol-bin-2.8.3 /usr/local/cruisecontrol

$ cd /usr/local/cruisecontrol/
$ export JAVA_HOME=/usr/lib/jvm/java-1.6.0-openjdk/
$ ./cruisecontrol.sh
```

注意export这一行。和很多Java应用一样，CruiseControl也需要知道Java可执行文件位于何处。你可以看到在我的系统中该文件的位置，在你的系统中，该文件的位置可能有所不同。可以运行

```
ls -al `which java
```

或者

```
locate javac | grep bin
```

以找到JAVA_HOME目录。java和javac（java编译器）二进制文件通常位于bin目录中。在JAVA_HOME目录中应该包含其父目录，而不是bin目录本身。

注解 顺利通过概念证明以后，你要确保启动集成服务器时，CruiseControl能自动启动。Felix De Vliegher所写的博客文章中介绍了CruiseControl的启动脚本，网址为http://felix.phpbelgium.be/blog/2009/02/07/setting-up-phpundercontrol/。

如果一切进展顺利，你会看到文本一滚而过，但这就是全部内容了。当你从这种失落的感觉中恢复过来以后，打开浏览器，在地址栏中输入http://localhost:8080/dashboard，看看你是否准备好继续执行了。浏览器中显示的内容如图20-4所示。

注解 我正在开发机上本地运行CruiseControl，所以我的URL都指向localhost。当然，你也可以为CI服务器使用不同的主机。

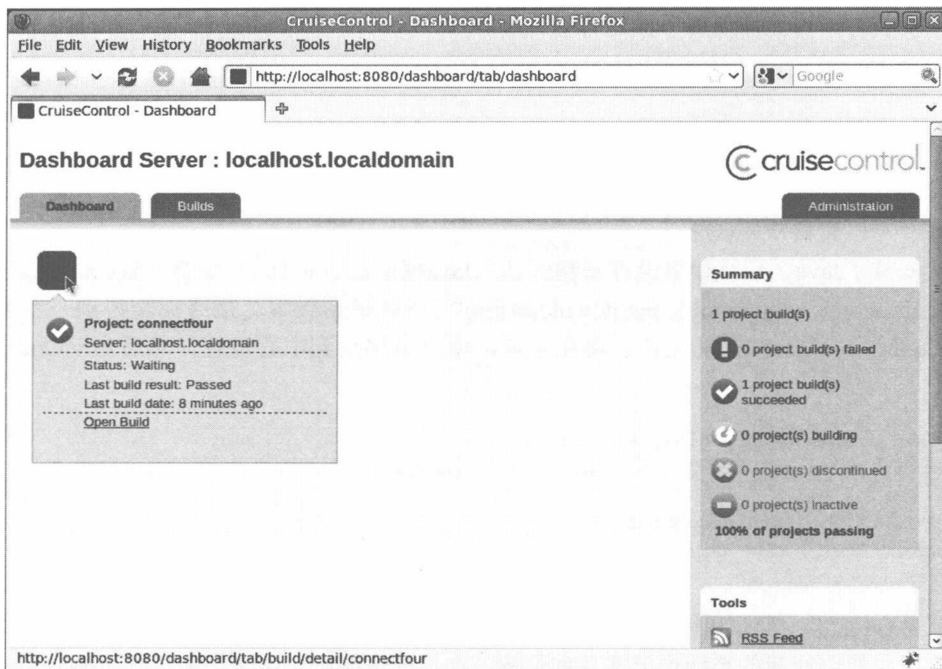


图20-4　CruiseControl的Dashboard界面

20.2.2　安装 phpUnderControl

像CruiseControl一样，phpUnderControl的作用是封送（marshal）其他工具。因此要确保已经具备了一些前提条件。我已经安装了本章的一些工具，下面再安装一个：

```
$ pear channel-discover components.ez.no
$ pear install -a ezc/Graph
```

ezcGraph包用于生成有用的状态信息。现在该包已经准备就绪了，可以安装phpUnderControl了：

```
$ pear config-set preferred_state beta
$ pear channel-discover pear.phpunit.de
$ pear install --alldeps phpunit/phpUnderControl
```

可以看到，在撰写本书时phpUnderControl还只有beta版。安装完以后，应该可以访问命令行工具phpuc。你可以使用usage标志来检查该工具：

```
$ phpuc --usage
```

```
Usage: phpuc.php <command> <options> <arguments>
For single command help type:
    phpuc.php <command> --help
```

```
Available commands:
  * clean          Removes old build artifacts and logs for a specified project.
  * delete         Deletes a CruiseControl project with all logs and artifacts.
  * example        Creates a small CruiseControl example.
  * graph          Generates the metric graphs with ezcGraph
  * install        Installs the CruiseControl patches.
  * merge-phpunit  Merges a set of PHPUnit logs into a single new file.
  * project        Creates a new CruiseControl project.
```

因此，我已经在系统中安装了该包。现在需要改进CruiseControl环境，以支持phpUnderControl。你看到了，phpuc又给出了一个安装步骤：`install`命令。

注解 这种分为两部分的安装机制很有用。PEAR擅长就地获取库代码、可运行的脚本和所支持的数据文件。就复杂的安装而言，例如Web应用和数据库驱动系统的安装，将可配置的安装命令作为应用的一部分提供通常是个好主意。当然，Phing可能是进行辅助安装时的不错选择。大多数用户可能都没有Phing，所以将安装逻辑构建到应用命令中更好一些。

```
$ phpuc install /usr/local/cruisecontrol/
```

现在重新启动CruiseControl：

```
$ cd /usr/local/cruisecontrol/
$ kill `cat cc.pid`
$ ./cruisecontrol.sh
```

CruiseControl在cc.pid文件中存储进程id。我使用它杀死当前进程，然后运行cruisecontrol.sh脚本，以重新启动。现在可以访问http://localhost:8080/cruisecontrol/，以确保重新注册了CruiseControl。新界面如图20-5所示。

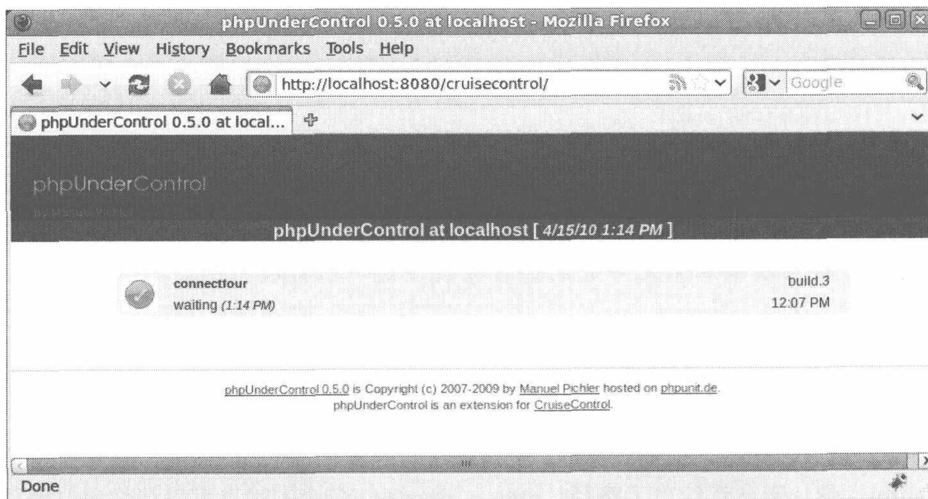


图20-5 phpUnderControl

既然phpUnderControl已经准备就绪，我需要让CruiseControl获得并构建userthing项目。

20.2.3 安装项目

我是一个程序员，我喜欢在文本编辑器中编写代码。但是像很多人一样，我讨厌写配置文件，幸好phpuc提供了一个工具，可用来生成项目的目录和配置。

注解 记住，我在/usr/local/cruisecontrol中安装了CruiseControl。本节讨论的所有文件和目录都与这个位置有关。

如果打算将userthing手动添加到CruiseControl，我会先编辑一个名为config.xml的配置文件，可以在CruiseControl的顶层目录中找到该文件。在该文件中，我告诉CruiseControl它应该识别出该项目并告诉它一些关键位置、构建计划。我还设置了一些发布程序，帮助生成报告。

然后我将在projects目录中创建名为userthing的工作目录。userthing项目目录中最重要的文件可能就是build.xml。这个文件在第19章出现过。Phing基于Ant，Ant是CruiseControl用来构建项目的工具，以运行任意的评估工具。Ant和Phing生成文件的语法是完全相同的。

CruiseControl有点复杂，不好掌握。http://cruisecontrol.sourceforge.net/main/ configxml.html上有学习config.xml的很好的参考信息，编写本书时，其中记录了123个XML元素。当你深入研究CI时，可能会用到它。这些文档非常有用，让人一看就知道短时间内构建和测试是不可能的。

phpuc结合project命令可以解决这个问题。下面的代码创建了所需的文件和目录，并修改了配置文件。

```
phpuc project --source-dir src  \
--version-control svn \
--version-control-url file:///var/local/svn/userthing/trunk \
--test-dir test \
--test-case UserTests \
--test-file UserTests.php \
--coding-guideline Zend \
--project-name userthing \
/usr/local/cruisecontrol/
```

注解 如果在与版本控制库不在一起的远程系统上运行CruiseControl，我还需设置user和password选项。

其中大部分代码的含义一看便知。CruiseControl需要访问userthing的源代码，因此要通过phpuc告知CruiseControl我正在使用Subversion，并为它提供该代码的URL，然后我还会告诉它去哪里寻找测试，以及我想应用的编码标准。要设置这些内容，phpUnderControl就要知道去哪里寻找cruisecontrol目录。因此我给出了路径。phpuc project命令的输出可以让你更好地了解到，要传递这个信息需要做哪些工作。

```
Performing project task.
   1. Creating project directory: projects/userthing
   2. Creating source directory:  projects/userthing/source
   3. Creating build directory:   projects/userthing/build
   4. Creating log directory:     projects/userthing/build/logs
   5. Creating build file:        projects/userthing/build.xml
   6. Creating backup of file:    config.xml.orig
   7. Searching ant directory
   8. Modifying project file:     config.xml
Performing checkout task.
   1. Checking out project.
   2. Preparing config.xml file.
   3. Preparing build.xml checkout target.
Performing PhpDocumentor task.
   1. Creating apidoc dir:  project/userthing/build/api
   2. Modifying build file: project/userthing/build.xml
   3. Modifying config file:        config.xml
Performing PHP_CodeSniffer task.
   1. Modifying build file: project/userthing/build.xml
Performing PHPUnit task.
   1. Creating coverage dir: project/userthing/build/coverage
   2. Modifying build file:  project/userthing/build.xml
   3. Modifying config file: config.xml
Performing PHP_CodeBrowser task.
   1. Creating browser dir: project/userthing/build/php-code-browser
   2. Modifying config file: config.xml
Performing ezcGraph task.
   1. Modifying config file: config.xml
```

phpuc可以帮助你了解这段代码的作用。可以看到，它为每一个任务修改了config.xml或build.xml文件（或两个文件都修改了），这些任务是我希望CruiseControl运行的任务。

你读到这里的时候，以这种方式运行phpuc可能已经能顺利通过了，但在写作本书时，还有一些问题需要先解决。毕竟，phpUnderControl此时还只有beta版。

首先，phpuc将一个元素写入config.xml文件中，这会使CruiseControl 2.8.3出现错误。如果你在config.xml中找到下面这行：

```
<currentbuildstatuspublisher file="logs/${project.name}/buildstatus.txt"/>
```

将这行注释掉或者删除，就可以避免致命的错误。

其次，phpuc会将一个对它自身的调用写入主配置文件config.xml中。下面是相关的部分：

```
<project name="userthing" buildafterfailed="false">
  <!-- ... -->

  <publishers>
    <!-- ... -->

    <execute command="/usr/bin/phpuc graph logs/${project.name}
      artifacts/${project.name}"/>
```

```
    </publishers>

  </project>
```

利用CruiseControl，你可以添加发布程序，从而为用户提供自定义的报告。遗憾的是，在写作本书时，使用phpuc graph命令会出现一个bug阻止该命令的运行。解决办法是从PEAR库中删除一个文件：

```
$ rm /usr/share/pear/phpUnderControl/Graph/Input/ClassComplexityInput.php
```

其中/usr/share/pear是我的PEAR目录。使用如下命令可以找到你自己的PEAR目录：

```
$ pear config-get php_dir
```

因为现在这个问题可能已得到解决，所以可以跳过这一步，但如果没有生成项目度量报告，记得要执行这一步。

最后，我要进行一处修改，这处修改与我的设置相关，而非与phpuc命令存在的问题相关。我喜欢将源代码放在项目的子目录（src/）中。这样我就可以在顶层目录中添加管理脚本（housekeeping script）、文档和其他杂项。phpuc会要求我在命令行参数中指定测试目录，但它会以源代码根目录为参考点创建调用。因此，如果我告诉phpuc测试目录位于src/test/处，我的测试套件位于UserTests.php中，那么它将创建一个对src/test/UserTest.php的调用。

因为我的测试被设计为从src/目录运行，所以会失败。我所需的语句都使用相对于src/的路径作为起点，而不是使用其父目录作为起点。在向你展示如何修改这一情况的同时，我们也来看一下build.xml文件。

注解 回想一下CruiseControl环境：config.xml位于顶层目录，并负责处理应用范畴的配置。项目的具体构建目标位于Ant文件中，该文件位于projects/userthing/build.xml中。当我运行phpuc project命令时，phpUnderControl为我创建了userthing目录和生成文件。

下面是phpunit任务和一些上下文信息：

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

<project name="userthing" default="build" basedir=".">

  <target name="build" depends="checkout,php-documentor,php-codesniffer,phpunit"/>
  <!-- ... -->

  <target name="phpunit">
    <exec executable="phpunit" dir="${basedir}/source" failonerror="on">
      <arg line=" --log-junit ${basedir}/build/logs/phpunit.xml
                 --coverage-clover ${basedir}/build/logs/phpunit.coverage.xml
                 --coverage-html ${basedir}/build/coverage UserTests test/UserTests.php"/>
    </exec>
  </target>
```

```
</project>
```

你看到了，这与Phing文档一样，被分成多个target元素，通过它们的depends特性相互关联。就目前的情况来看，phpunit任务将会失败。它正调用test/UserTest.php，但是是通过${basedir}/source的上下文进行调用的。为此，我们仅需要修改exec元素，这样它就能通过${basedir}/source/src来运行。

```
<exec executable="phpunit" dir="${basedir}/source/src" failonerror="on">
```

现在准备运行项目。

1. 运行phpUnderControl / CruiseControl

首先，需要重启CruiseControl：

```
$ kill `cat cc.pid`
$ ./cruisecontrol.sh
```

现在，访问http://localhost:8080/cruisecontrol，可以查看初始构建的结果。控制面板中应该会显示已添加了userthing项目，并显示构建结果。单击项目名称，打开“概览”界面，如图20-6所示。

图20-6　概览界面

可以看到，虽然CodeSniffer会警告格式问题，但构建过程进展顺利。单击CodeSniffer选项卡，查看全部的警告信息，或者单击Code Browser（代码浏览器）选项卡，在代码环境中查看这些警告信息。代码浏览器界面如图20-7所示，其中打开的是与上下文环境相关的errors/notices选项卡。

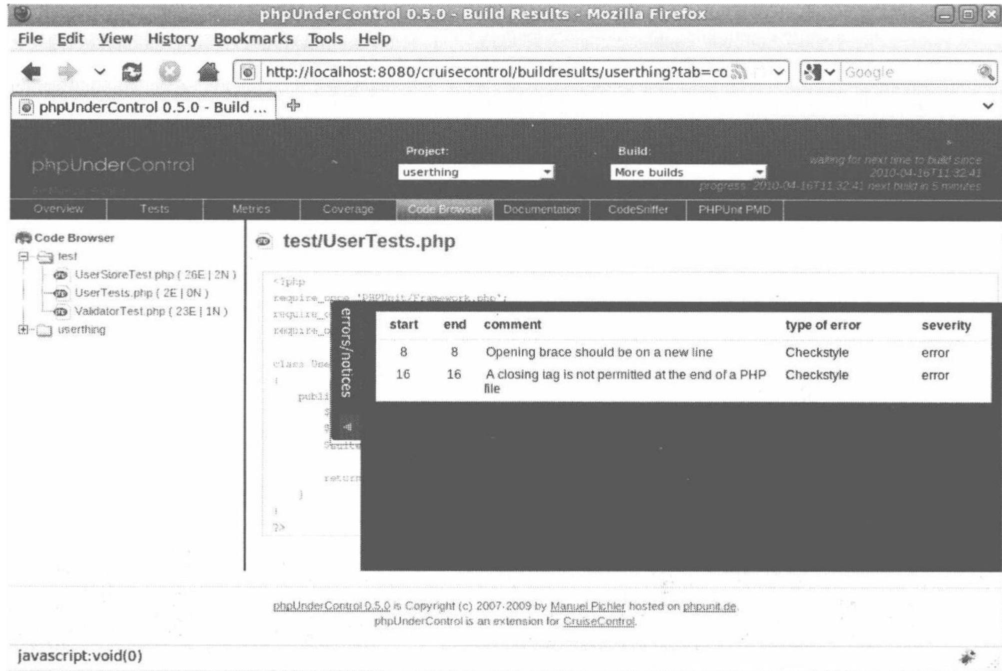


图20-7　代码浏览器显示“错误”的具体信息

我先前测试过的大多数工具都可以在这些选项卡中找到。可以查看代码覆盖率，检查测试结果，并浏览文档。CruiseControl会定期更新项目并运行构建（默认为5分钟）。单击Metrics（度量）选项卡，查看概览信息，如图20-8所示。

2. 测试失败

现在为止，一切进展顺利，虽然userthing短期之内不会被加入到Zend代码库，但当测试失败后，测试还继续进行着，所以我最好是破坏其中的部分内容，看看CruiseControl是否会报告这个问题。

下面是命名空间`userthing\util`中`Validate`类的一部分：

```
public function validateUser( $mail, $pass ) {
    // 使之总是失败
    return false;

    $user = $this->store->getUser( $mail );
    if ( is_null( $user ) ) {
```

```
        return null;
    }

    if ( $user->getPass() == $pass ) {
        return true;
    }

    $this->store->notifyPasswordFailure( $mail );
    return false;
}
```

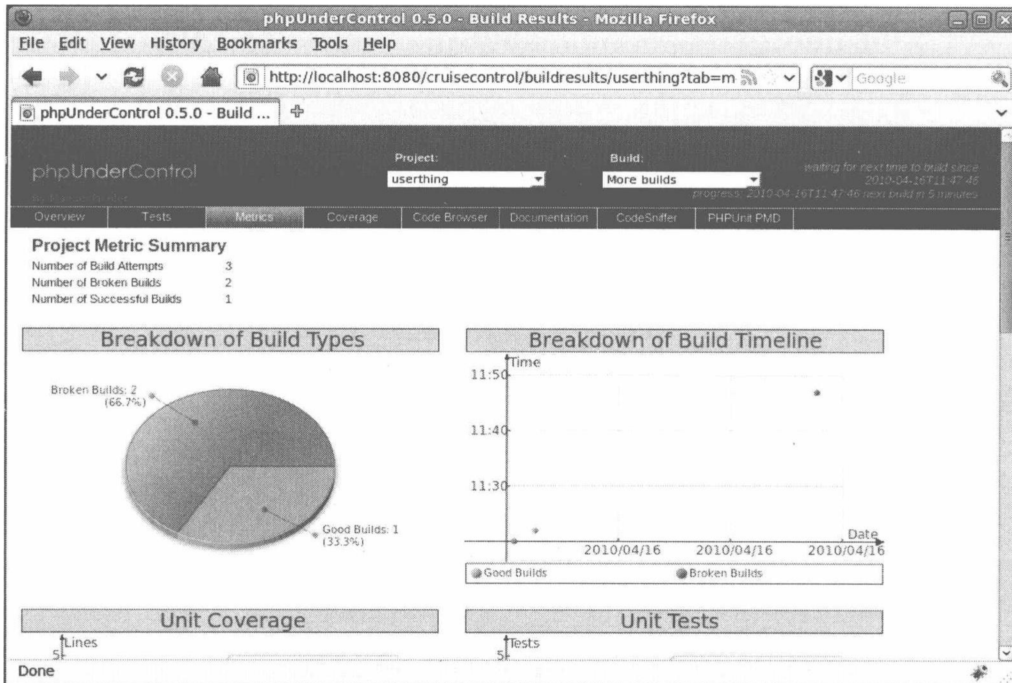


图20-8 Metrics选项卡的界面

看一下我是如何破坏这个方法的。现在validateUser()会一直返回false。下面这个测试会因这种情况而失败。这些代码位于test/ValidatorTest.php中：

```
public function testValidate_CorrectPass() {
    $this->assertTrue(
        $this->validator->validateUser( "bob@example.com", "12345" ),
        "Expecting successful validation"
        );
}
```

修改完以后，要做的就是提交和等待。果然，没过多久，状态页上就用警告常用的橘红色高亮显示了userthing。单击项目名称以后，选择Test（测试）选项卡，就能看到错误报告了，如图

20-9所示。

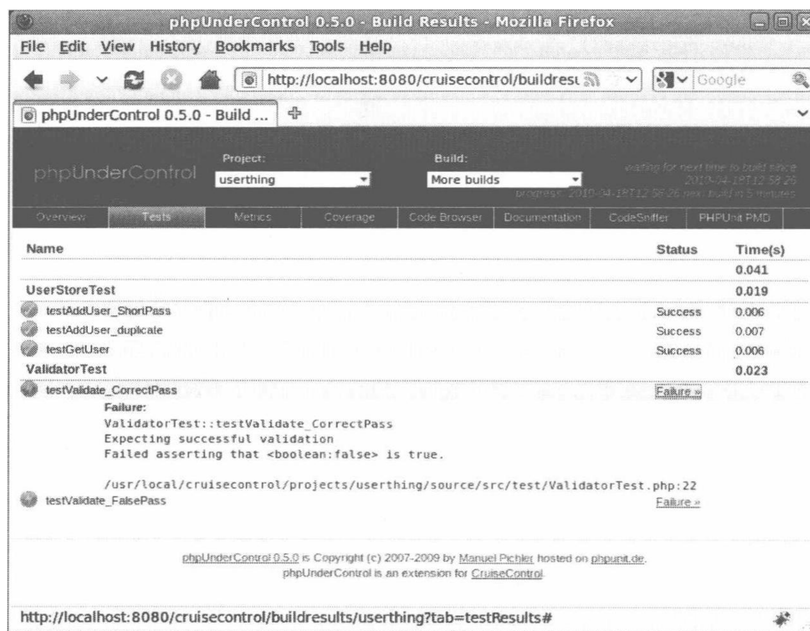


图20-9 测试失败

3. 失败通知

只要经常访问CruiseControl Web界面，那么在该界面上显示错误报告就是有用的，但也存在着一定的危险，系统无声无息地运行，我们就会忘记访问该界面。可以让CruiseControl来提醒我们。为此，可以使用内置的发布程序：email，将它添加到项目的config.xml文件的publisher元素中：

```
<email mailhost="smtp.somemail.com"
    mailport="465"
    username="some.user@somemail.com"
    password="somepass"
    usessl="true"
    returnaddress="ci_guy@getinstance.com"
    buildresultsurl="http://localhost:8080/cruisecontrol/buildresults/${project.name}"
    returnname="CruiseControl">

        <always address="builds@userthing-team.com" />
        <failure address="panic@userthing-team.com" reportWhenFixed="true" />

</email>
```

email元素包含了连接到邮件服务器所需的全部信息。我假设需要SSL连接，所以不能省略mailport和usessl元素。always元素定义了一个地址，不管构建成功还是失败，都要向该地址

发送消息。failure元素定义了一个地址，向这个地址发送失败通知。此外，将reportWhenFixed设置为true，当构建再次成功时，失败收件人（failure recipient）也将收到明确的消息。

失败消息非常短，包括主题行，主题行会给出构建状态和指向完整报告的URL。

注解 如果想查看更详细的消息，应该看一下htmlemail发布程序，它与email发布程序共用一套特性和子元素，同时也提供了其他选项，帮助定义内联消息的格式。要了解htmlemail发布程序的更多信息，可访问http://cruisecontrol.sourceforge.net/main/configxml.html#htmlemail。

---

4. 添加自定义构建目标

到现在为止，我一直使用phpUnderControl支持的工具集。毕竟它给我们带来了很多方便。但是，你应该知道，可以非常轻松地将你自己的工具添加到CruiseControl中。

到现在为止，最大的遗漏就是创建和安装包时进行的测试。这很重要，因为CI就是用来进行构建和测试的。创建PHPUnit测试用例是一种方法，它会尝试构建并安装包。

为了说明CruiseControl的一些功能，我建议从产品生成文件内部执行构建和安装。我已经向你展示了build目标，它会通过其depends特性调用其他目标。下面我添加一个新的依赖关系：

```
<target name="build"
    depends="checkout,php-documentor,php-codesniffer,phpunit,install-package"/>
```

install-package目标尚不存在。现在添加内容。

```
<target name="make-package">
  <exec executable="pear" dir="${basedir}/source/src"
      failonerror="on"
      output="${basedir}/build/builderror/index.txt">
    <arg line=" package" />
 </exec>
</target>

<target name="install-package" depends="make-package">
</target>
```

实际上，目前install-package是空的。因为它依赖于另一个新的目标make-package，所以必须先运行make-package。我使用名为exec的Ant任务调用pear package命令。这会在${basedir}/source/src目录中查找名为package.xml的文件。我怎么知道这个文件在那个目录中？这要得益于这个文件中的checkout目标，它会调用Subversion并更新${basedir}/source中的userthing项目。

我使用output特性将错误消息发送到build/builderror目录下的文件中。其他任务使用build目录，该目录已经存在于合适的位置处了，但builderror是新的，所以需要通过命令行创建它。

重新启动CruiseControl，没有什么不同。再说一次，只有在某些内容发生变化时，我才能看到好处。因此，接下来我要创建一个错误。下面我将在package.xml文件中创建一个错误：

```
<summary>A sample package that demo's aspects of CI</summary>
```

```
<wrong />
<description>Consisting of tests, code coverage, Subversion, etc
</description>
```

提交以后，执行到wrong元素时，pear package命令不会继续执行，将拒绝构建包。因为exec元素中设置了一个failonerror特性，所以构建会失败，稍后CruiseControl会提醒开发团队。

CruiseControl Web界面中显示了失败的构建，参见图20-10。

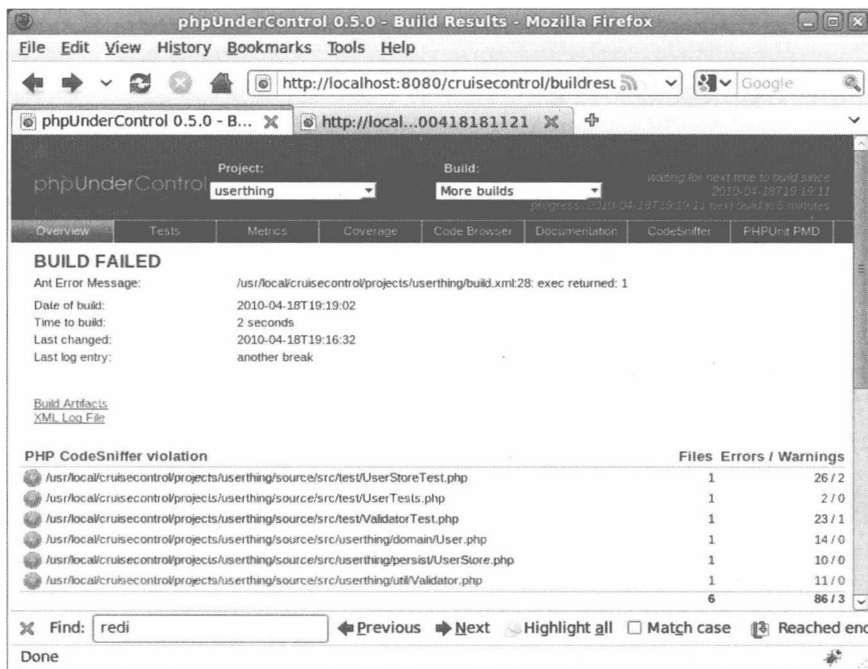


图20-10 构建失败

可以看到，已经发生了错误，但错误的具体原因还不清楚。错误消息只显示构建失败。实际上，如果单击XML Log File链接，最终能在一堆XML中找到这个错误。有几种方法可以使这个错误更容易被发现。一种方法是修改由Ant生成的失败消息。另一种方法是使用发布程序创建一个artifact目录。

artifact是可以合并到CruiseControl界面中的随机输出。到目前为止，本章介绍的所有报告实际上都是从artifact目录中获得的。CruiseControl提供了artifactspublisher元素，该元素位于config.xml文件中项目的publishers元素下。你已经见过email元素了，它是一个同级元素。artifactspublisher只是将构建过程中生成的输出或由后续处理生成的输出移动到artifact目录中。因此，这里将artifactspublisher元素添加到config.xml目录中。

```
<artifactspublisher dir="projects/${project.name}/build/builderror"
```

```
        dest="artifacts/${project.name}"
        subdirectory="builderror"/>
```

这段代码会将builderror目录从projects/userthing/build复制到artifacts/userthing/<datestamp>/builderror。当CruiseControl再次遇到构建错误时，我可以单击Build Artifacts链接（如图20-10所示），然后单击artifact目录链接，转到如图20-11所示的页面。

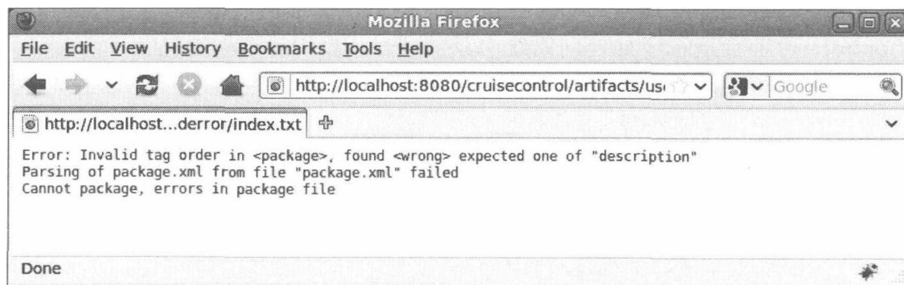


图20-11　简单的artifact报告

我已经为make-package目标设置了一个基本的报告，可以继续对install-package执行相同的操作。最好同时实现它。

```
<target name="install-package" depends="make-package">
  <fileset id="package.ref"
       dir="${basedir}/source/src/"
       includes="userthing*.tgz" />
  <pathconvert property="package.file" refid="package.ref" />

  <exec executable="pear" dir="${basedir}/source/src"
      failonerror="on"
      output="${basedir}/build/builderror/index.txt">
    <arg value="install" />
    <arg value="--force" />
    <arg value="--installroot=${basedir}/build/install" />
    <arg file="${package.file}" />
  </exec>

  <delete>
  <fileset refid="package.ref" />
  </delete>
</target>
```

同样，这个目标主要由对exec任务的调用组成。首先，我创建一个与userthing包文件（userthing-1.2.1.tgz）匹配的fileset，并使用新任务pathconvert将其转换成扩展路径。现在我可以将一些内容传递给由exec调用的命令，该命令就是pear install。因为output特性是在exec元素中设置的，所以可以像以前一样访问失败信息。最后，删除包文件。清理包文件很重要，因为我假设该目录中只有一个文件与userthing*.tgz匹配。如果让旧的包文件仍留在src目录中，下次增加版本号的时候可能会出现意想不到的结果。

20.3 小结

本章涉及了前面几章中介绍过的工具，并通过CruiseControl将它们集中在了一块。我介绍过，借助于phpUnderControl，持续集成服务器可以与PHP项目实现完美契合。我为CI准备了一个小项目，为其应用了一系列工具，包括PHPUnit（为的是进行测试并检测代码覆盖率）、PHP_CodeSniffer、PHP_CodeBrowser、phpDocumentor和Subversion，然后设置了CruiseControl和phpUnderControl，并向你展示了如何向系统中添加项目。我们对该系统进行了考察，最后，介绍了如何扩展CruiseControl，以便让它在出现bug时用电子邮件通知你。此外，还对构建和安装进行了测试。

第21章

对象、模式与实践

从对象基础到设计模式原则，再到一些实用工具和技术，本书的所有内容都致力于实现一个目标：成功的PHP项目。

本章将会重述本书的主题和要点。

- PHP与对象：PHP如何不断增强对面向对象编程的支持及如何使用这些特性。
- 对象与设计：总结一些面向对象设计原则。
- 模式：是什么让它们如此之酷。
- 模式原则：概括许多设计模式背后的面向对象原则。
- 实用工具：重温之前提到的工具，并介绍一些之前没有涉及的工具。

21.1 对象

第2章介绍过，很长一段时间内对象只是PHP的一个候补功能。PHP 3对对象的支持非常弱，当时对象只是在其华丽外表下的关联数组而已。对于对象热爱者来说，虽然在PHP 4中对象机制已经发生了根本的改进，但仍存在很大的问题。例如对象的赋值和传递原本应该默认通过引用进行，但PHP 4中并非如此。

PHP 5的诞生最终树立了对象的核心地位。当然在PHP中你仍然可以不用声明类，但是毫无疑问PHP语言本身已经为面向对象设计进行了优化。

在第3章、第4章和第5章中，我们详细介绍了PHP对面向对象的支持。PHP 5引入的新特性有：反射、异常、私有和保护的方法和属性、`__toString()`方法、`static`修饰符、抽象类和抽象方法、`final`方法和属性、接口、迭代器、拦截器方法、类型提示、`const`修饰符、按引用传递、`__clone()`方法和`__construct()`构造方法、迟静态绑定、命名空间等。这个列表并不完整，但其长度已经足以揭示面向对象编程和PHP在未来的关联程度。

我还期待一些未列入PHP发展计划的特性，比如对于基本数据类型的类型提示。我也希望PHP能对方法返回值类型进行提示——在方法声明中规定方法返回值的类型。我希望Zend引擎对当前类方法和重写得到的类方法都强制进行类型检查（这个特性可能会在PHP未来版本中出现）。

也许我有点吹毛求疵。其实Zend Engine 2和PHP 5已经确立了面向对象设计在PHP中的核心地位，从而对新的开发者开放了使用PHP语言的大门，同时也给PHP程序员提供了新的开发方式。

在第6章中，我们介绍了对象能够给项目设计带来的好处。因为对象和设计是本书的一个中心主题，所以我们在此详细总结一下它们。

21.1.1　选择

没有人规定你只能用类和对象开发。无论是过程式还是面向对象的客户端代码，都可以访问经过良好设计的、由面向对象代码提供的清晰接口。即使你对编写对象丝毫不感兴趣（如果你仍然在阅读本书，那不太可能），你也可能会发现自己正在使用对象，例如你可能使用PEAR包。

21.1.2　封装和委托

对象只专注于它们自己的业务，“关着门”来执行分配给它们的任务。它们提供一个可以传递请求和结果的接口。任何不需要被暴露的数据和实现的具体细节都隐藏在这个外表之下。

这样的处理方式给予了面向对象项目和过程化项目不同的形态。在一个面向对象的项目中，控制器的内容非常少，只由少数必需的对象及从这些对象获得和传递数据的调用组成。

而一个过程式项目更倾向于成为一个“干涉主义者”，其控制逻辑涉及的范围更广，如引用变量、检查返回值、根据环境执行不同的操作等。

21.1.3　解耦

解耦就是去除组件之间的相互依赖关系，以便对一个组件的改变不会牵涉其他组件。设计良好的对象是自闭的。也就是说，它们不需要引用它们自身以外的对象来恢复之前的某个调用中已获得的细节。

通过维持内部状态，对象减少了对全局变量的需求，而全局变量正是导致紧耦合的重要原因之一。在使用全局变量时，你会把系统一个部分绑定到另一部分上。如果组件（无论是函数、类或者是代码块）引用了一个全局变量，那么会存在另外一个组件意外地使用与第一个组件相同的变量名并覆盖该变量值的风险，而第三个组件也可能依赖于第一个组件所设置的这个变量值。改变第一个组件，可能会导致第三个组件无法工作。面向对象设计的目的就是减少类似的相互依赖，让每个组件尽可能自给自足。

代码重复是导致紧耦合的另一个原因。如果在项目的不同地方重复使用一个算法，就会产生紧耦合。当你打算改变该算法时，会发生什么呢？你必须牢记该算法出现的每一个地方并作出相应的修改。如果忘记了其中某个地方，系统就可能出错。

引起代码重复的一个原因便是平行条件。如果项目需要在某特殊环境（比如运行在Linux之上）上执行某种操作，而在另一种环境（如运行在Windows上）进行不同的操作，那么你会在系统的不同部分经常发现同样的`if/else`语句。如果我们要添加一个新的环境（如Mac OS系统）及相关操作，那么必须保证所有的条件语句都随之更新。

面向对象编程为处理这类问题提供了一种技术——可以用多态来代替条件语句。多态，也叫类切换（class switching），即根据环境透明地使用不同子类。因为每个子类和父类支持同样的接口，所以客户端代码既不知道也不在意使用的是哪一个具体实现。

条件代码并没有从面对对象系统里去除，它只不过是被最小化和集中化了。条件代码必须用

于决定哪一个具体的子类去服务客户端代码。这个类型检查通常只发生一次，并且只在一个地方发生，因此减少了耦合产生的几率。

21.1.4 复用性

封装促进了解耦，也促进了复用①。如果组件能够自给自足并仅通过公共接口与外部系统交互，那么就可以从一个系统移植到另一个系统，而不需要改动其中的代码。

事实上，这种情况过于理想化，在现实中很少见。即使优秀的正交代码也是针对特定项目的。比如当创建一组用于管理特定网站内容的类时，你可能需要在项目规划阶段研究针对客户的具体特性。这些特殊性很可能将会成为项目的基础。

复用的另一个小技巧：把那些可能会在多个项目中使用的类集中起来。也就是说，不要复制一个易于复用的类到一个新项目中。这会导致宏观上的紧耦合，因为你很可能会在一个项目中改变了该类中的代码，却忘了在另一个项目中这么做。更好的做法是将这些可以公用的类集中起来，放到一个可被多个项目共享的代码仓库中。

21.1.5 美学

这不是用来说服那些尚未信服的人的。对我而言，面向对象代码确实能带来某种美学上的愉悦感。我们把杂乱的实现隐藏于干净的接口后，使对象对于它的客户端代码而言变成一个外表简单的东西。

我爱多态的整洁优雅，这样API允许你操纵众多不同的对象，可以随时改变使用的对象，并且整个过程很透明——这使对象可以被巧妙地叠加起来，就像孩子们搭积木一样。

当然，也有人争论说反之才对。面向对象的代码会导致令人痛苦的类名，这些类名必须与方法名结合使用并进而导致更加吃力的调用。这在PEAR中尤为如此，因为PEAR为弥补PHP不支持命名空间的缺陷而在类名中包含包名。现在命名空间已经成为PHP语言的一部分。但是为了使代码能够向下兼容，要去除旧的命名惯例还得等些日子。

值得一提的是，一个漂亮的解决方案并不总是最佳的或者最有效的。有时候你也许会想使用完整的面向对象的解决方案，但实际上一个简单的脚本或者一些系统命令就可以完成这个工作。

另一个合理的批评就是面向对象代码可能会导致类和对象的混乱，从而使代码变得难以阅读。不可否认这种情况时有发生，但提供包含使用范例的详细文档可以改善这种状况。

21.2 模式

最近有一个Java程序员在申请与我有些关系的某个公司的职位。在求职信中，他为只使用过几年模式而不好意思，就好像设计模式是新发现似的。而事实上，很可能这个经验丰富的程序员使用模式的时间比他自己所认为的时间长得多。

模式描述了公共问题及经过测试的解决方案。模式对真实世界的最佳实践进行了命名、整理和组织。它们不是发明创造出的组件或者教条主义的条款。如果模式没有记下在模式孵化时就存

① reuse，也称重用，即一段代码或程序可以在不同项目或同一项目的不同地方重复使用。

在的共同实践，那么这个模式将是无效的。

记住，模式语言的概念源于建筑学领域。在模式作为一个描述空间及功能问题的解决方案被提出之前，人们就已经建造了几千年的庭院和拱门。

之前就说过，设计模式常会挑起某种伴随宗教或政治争论的情绪。信徒们的眼中带着福音传道的光芒，手持《设计模式》徘徊在走廊上。他们和未入会的人搭讪，背出模式名就好像背出一篇信仰的文章，所以一些批评家将设计模式看做炒作也不足为奇。

在Perl和PHP等语言中，是否使用模式也因为它与面向对象编程的紧密联系而备受争议。在这两种语言中，你可以自由选择使用过程式编程或面向对象编程。在PHP中，使用对象只是一个设计决策而不是一个已确定的事实（如Java，你只能选择面向对象模式），是否使用对象及设计模式只是个人偏好，而不是因为模式引发更多的模式，对象会产生更多的对象。

21.2.1　模式给我们带来了什么

第7章介绍了设计模式。现在让我们回顾一下模式带来的好处。

1. 经过实践检验

首先，模式是已经证明可行的特定问题的解决方法。把模式等同于菜谱是危险的：菜谱可以盲从，而模式是天生的“半成品”（Martin Fowler），需要更加周全地处理。但模式与菜谱有相同之处：它们都经过尝试，并且在被记录之前经过了测试。

2. 模式促进其他模式的使用

模式经常可以和其他模式结合使用。使用模式解决问题会不可避免地产生衍生物，从而形成引入补充模式的条件。当选择相关模式时，要小心地确认真实的需求和问题，不要为了让代码看起来优雅而叠加无用代码。模式有时会诱发为架构而编程的愚行。

3. 公共词汇表

模式促进了公共词汇的发展，这些公共词汇是用于描述问题和解决方法的。命名非常重要——它代表着描述。有了通用的命名，我们可以从一个单词中快速获知大量内容。当然，命名对那些还没有共享词汇表的模式来说，带来了晦涩的含义，这就是为什么有时模式会如此令人愤怒。

4. 模式改善设计

下一节将讨论，合理使用模式会促进优秀的设计。当然，要注意避免一些典型的误用。毕竟模式不是仙丹，无法包治百病。

21.2.2　模式和设计原则

设计模式归根结底是为了促进设计。用好设计模式可以帮助你建立松耦合和灵活的代码。但是也有批评者指出，模式可能被新接触模式的程序员滥用。因为模式的实现形成了那些完美优雅的结构，所以它能诱使我们忘记“好的设计的目的在于解决问题”（good design always lies in fitness for purpose）。记住，模式的存在是为了解决问题。

当我刚开始学着使用设计模式时，代码中到处都是我创建的抽象工厂。因为我需要创建对象，所以我想当然地使用抽象工厂。

但事实上，我那个时候懒得动脑筋，给自己制造了很多不必要的工作。我需要创建的一组对象确实是相互关联的，但它们之间并不能互相替代。在你有一组可互相替换的对象并需要根据环境生成对象时，传统的抽象工厂模式才是一个适用的模式。要使用抽象工厂，你需要给每个类型的对象创建工厂类，并创建一个类作为工厂类。仅仅描述这个过程就令人疲倦了。

如果只创建一个基础的工厂类，那么代码便可以更干净。如果要生成一组平行对象，那么将基础工厂类重构为抽象工厂即可。

使用模式并不能保证一定会产生优秀的设计，因此在开发时请记住这样一个原则（它有两种表述方式）：保持简单和笨拙（Keep it simple, stupid，常称为KISS原则）和做最简单的事情来完成任务（Do the simplest thing that works）。极限编程（XP）程序员也给了我们另外一个相关的原则：你不需要它（You aren't going to need it，常称为YAGNI原则），即只有在真正需要时才去实现。

记住这些忠告之后，我们可以继续保持对模式的热爱。第9章提到过，模式倾向于具体化那些可以应用于所有代码的普适原则。

1. 组合优于继承

继承关系很强大。我们使用继承来支持运行时的类切换（多态），而这是本书许多模式和技术的核心所在。但在设计中过分依赖继承，就可能导致有局限性的结构，或产生大量令人痛苦的重复代码。

2. 避免紧耦合

本章已经谈论过这个话题，但是为了追求完整性，仍值得在这里再提一下。你永远也逃脱不了改变一个组件就要改变项目其他部分的现实，但你可以通过一些方法来最小化这种可能性，比如避免重复（代表性的例子就是平行条件），避免滥用全局变量（或单例对象）。当可用抽象类来促进多态性时，应该避免使用具体的子类。最后一点引出了另一个原则。

3. 面向接口编程，而不是面向实现编程

用定义清晰的公共接口来设计软件组件可使各组件职责分明。如果在一个抽象父类中定义接口，并且客户类只要使用该抽象类型即可工作，那么你就解开了客户端代码与具体子类的耦合。

要牢记YAGNI原则。如果只需要一个类型的实现，那么就不要急着创建抽象父类。你只需在单独的具体类中定义一个清晰的接口即可。一旦你发现该类在同一时间做了不止一件事情，你就可以把该具体类设计成为两个子类的抽象父类。因为客户端代码只要求父类的类型即可工作，所以即使它对类中的改变一无所知，仍可正常工作。

具体类中出现条件语句是需要把具体实现分割的经典信号。这时你要创建一个抽象父类，然后在不同的子类中实现不同的功能。

4. 封装变化的概念

如果你发现自己创建了很多子类，可能你应该认真想想自己实现各个子类的动机。当实现子类的目的与父类类型的主要功能无关时，更应该好好思考一下是否真的需要创建子类。

比如，假设有一个`UpdatableThing`类型，你发现自己还创建了`FtpUpdatableThing`、`Http-UpdatableThing`和`FileSystemUpdatableThing`等多个子类型，但`UpdatableThing`类型的职责是为了成为一个“*可以更新的东西*”——存储和获取的机制只是附加功能。这里`Ftp`、`Http`和`File-`

System是可变的，并且它们应该属于它们自己的类型——我们可以将其命名为UpdateMechanism。针对不同的实现，UpdateMechanism会有不同的子类。这样，你可以在不影响UpdatableThing类型的情况下添加你所需要的更新机制。

再次注意一下，在这里我们用动态的运行时结构替换了之前的静态编译期结构，这把我们带回了我们的首要原则："组合优于继承。"

21.3　实践

本节（及在第14章中）所提到的问题常被书和程序员所忽略。在我自己的编程生涯中，我发现这些工具和技巧对于项目是否成功非常重要，至少与设计对于项目的作用一样重要。不过可以肯定的是，文档和自动构建这样的工具不会像组合模式那样让人惊叹。

注解　让我们想想组合模式的优点：一个简单的继承树，它的对象可以在运行时组合起来形成其他树形结构，但数量级上要更灵活和复杂。通过共享单一的接口，多重对象可以对外部有统一的呈现。简单与复杂之间、单一与多样之间的相互作用会让你脉搏加速——那不仅仅是软件设计，那简直是一首诗。

尽管与设计模式相比，文档、构建、测试和版本控制都过于平淡，但它们的重要性一点都不亚于设计模式。在现实世界，如果开发者们不能方便地贡献或者理解源代码，那么奇妙的设计是不可能生存下来的。没有自动测试，系统变得很难维护和扩展；没有构建工具，没人愿意部署你的作品。因为PHP的用户基础在不断扩大，所以保证项目质量过关并易于部署是我们作为开发者的职责。

一个项目存在于两种模式中。它是代码和功能的组合，同时也是一组文件和目录、一个合作的基础、一个源代码和系统目标的集合以及转换的主题。从这个角度理解，项目不仅是代码，还包含很多代码之外的东西。构建、测试、文档和版本控制的机制需要得到和代码一样的关注。

21.3.1　测试

虽然测试是项目的外部工具之一，但是它与代码本身结合相当密切。完全的解耦是不可能的（可以说是种奢望），而测试框架是监控代码改变所引起的其他关联反应的有效方法。改变一个方法的返回类型可能会影响到其他地方的客户端代码，从而导致作出改变之后的几个星期或者几个月出现bug。一个测试框架至少给了你一半的机会来捕获这类错误（测试工作完成得越好，发现并解决bug的几率越大）。

测试也是改进面向对象设计的一种工具。测试帮助程序员关注类的接口，并且仔细思考每一个方法的职责和行为。本书在第18章中曾介绍过可用于测试的PHPUnit2包。

21.3.2　文档

你的代码也许并不如你自认为的那样清晰。当一个陌生人第一次访问你的代码库时，可能会望而却步。即使作为代码作者的你，最终也会忘了代码是如何组合在一起的。本书第16章介绍了phpDocumentor，它允许你为代码添加文档，并且能够自动生成带有超链接的文档。

phpDocumentor的输出允许用户通过单击超链接从一个类切换到另一个类，所以它在面向对

象的项目环境中特别有用。因为类总是包含在各自的文件中，所以如果你想读另一个类时，要新打开一个文件，这样很麻烦。有了带超链接的文档，就方便多了。

21.3.3 版本控制

团队协作并不容易。让我们来直面这个问题吧，因为人与人的沟通并不容易。程序员之间的协作甚至更为困难。一旦你整理出了团队中的角色和任务，还要处理的就是源代码本身的冲突。在第17章中已经介绍过，Subversion（类似工具还有CVS和Git等）让你可以把多个程序员的工作合并到一个单独的代码仓库里。当代码产生冲突时，Subversion就会标识出代码中产生冲突的位置以帮助你解决问题。

即使你是一个独立工作的程序员，版本控制也是必需的。正因为Subversion支持分支，你才能维护一个软件的发布版本，并同时开发它的下一个版本，并将稳定发布版本的bug修复合并到开发分支中。

Subversion也提供了项目代码的任何一个提交记录，就是说你可以通过日期或标志回滚到任意时刻的版本。相信我，总有一天版本控制会挽救你的项目。

21.3.4 自动构建

没有自动构建的版本控制的作用是有限的。不管复杂性如何，任何一个项目都是要花时间去部署的。不同的文件需要被移到系统的不同地方，配置文件需要被转换来包含与当前平台和数据库特性相关的值，数据库表需要被建立或者转换。本书曾提及过两个用于安装应用程序的工具。首先，PEAR（见第15章）是独立包和小型应用的理想工具。第二个构建工具是Phing（见第19章）。Phing是一个强大而灵活的工具，可用于自动安装那些庞大而复杂的项目。

自动构建把部署从一个琐碎杂事变成一件只有一两行命令的事情。你可以不费吹灰之力地通过构建工具调用测试框架或输出文档。每一个新版本发布的时候，用户发现他们不再需要花一个下午的时间来复制文件和改变配置时，他们激动的惊叫声会让你难以忘怀。

21.3.5 持续集成

能够测试和构建项目还不够，你还要不断地这样做。随着项目变得越来越复杂，管理的分支越来越多，不断地测试和构建也变得越来越重要。你应该构建和测试稳定的分支，通过这些分支，可以创建较小的bug修正版、一两个实验开发分支，以及开发主干。如果你希望手动完成所有这些工作，即使有构建和测试工具帮忙，也不可能再抽出时间考虑编写代码的事了。当然，所有程序员都讨厌这个结果，构建和测试就照例能省就省了。

第20章介绍了持续集成，它既是一种实践也是一个工具集，用来尽可能地自动化构建和测试过程。

21.3.6 我们还遗漏了什么

由于篇幅限制，有一些工具本书没有介绍，但它们对任何一个项目来说都是非常有用的。

在这些未被提及的工具中，最重要的也许是Bugzilla，这个名字可能会让你联想到两件事情：首先，它是一个跟踪bug的工具；其次，它是Mozilla项目的一部分。

与Subversion一样，Bugzilla是一个可以提高生产力的工具。一旦用过，就再也不想放弃。你可以从http://www.bugzilla.org下载Bugzilla。

Bugzilla被设计来允许用户报告项目的问题，但是就我的经验而言，它经常被用来描述特定的需求，并分配任务给不同的团队成员。

在任何时候，你都可以得到一个待解决bug的快照。你可以根据产品名称、bug所有者、版本号以及优先权来缩小搜索范围。每个bug都有自己的页面，在该页面里你可以讨论任何相关问题。讨论内容和bug状态的变化都可以通过邮件发送给开发团队的每个成员，因此你可以轻松地跟踪项目进展，不用重复单击Bugzilla上的bug链接。

相信我，你一生中都会需要Bugzilla。

每个正式项目都至少需要一个邮件列表，从而让用户及时了解到项目的变化和使用问题，而开发者也可以讨论架构和资源分配。我最爱的邮件列表软件是Mailman（http://www.gnu.org/software/mailman/），它是免费的，相对比较容易安装，并且具有很高的可配置性。如果你不想安装自己的邮件列表软件，有很多站点可以供你免费运行邮件列表或者新闻组。

尽管代码中的行内文档十分重要，项目仍旧需要创建一堆书面材料，其中包括使用说明、未来需求预测、客户评估、会议记录以及项目公告等。在一个项目的生命周期里，这些材料都是不固定的。我们通常需要一个机制来允许人们协作编写这些书面文档。

wiki（夏威夷语，“非常快”）是用于创建协作的带有超链接文档网页的完美工具。你只需单击一个按钮就可以创建或编辑页面，并且为匹配相应页面名称的文字自动生成超链接。wiki看起来如此简单、必要且具有明显的意义，它是一种你肯定想到过却没有真正使用的工具。这里有许多可供挑选的wiki系统。我曾使用过一个叫做FosWiki的wiki系统，它相当不错，可以在http://foswiki.org/上下载到。FosWiki是用Perl写的。当然也有用PHP写的wiki，著名的有PhpWiki（可在http://phpwiki.sourceforge.net上下载到）和DokuWiki（可以在http://wiki.splitbrain.org/wiki: dokuwiki上找到）。

21.4 小结

本章总结了本书，重温了构成本书的几个核心主题。本章在此没有讨论任何单独的模式或者对象功能等具体问题，而是合理总结本书所涉及的内容。

由于众口难调，而且篇幅有限，本书没能介绍每个人都喜欢的内容，但我希望本书会给读者带来一些启发：PHP正在飞速成长，它现在是世界上最流行的编程语言之一。我希望PHP仍旧是爱好者们所钟爱的语言，而新的PHP程序员会欣然地发现他们通过一点点代码就可以得到很多收获。此外，越来越多的专业团队正使用PHP创建庞大的系统。这些项目比那些只是为了实现功能而开发（just-do-it）的项目更具价值。通过扩展，PHP提供了对众多应用和类库的接口，因此PHP一直是一种充当多面手的编程语言。另一方面，PHP对面向对象的支持也促进了程序员对一些工具的使用。一旦开始以面向对象的思维来思考，你就可以得到其他程序员来之不易的经验。你能理解和学习由PHP以外的语言（如Smalltalk、C++、C#或者Java）开发的模式语言。使用精细设计和最佳实践来满足未来挑战是我们开发人员的责任。我们希望未来是可以重用的。

Part 6

第六部分

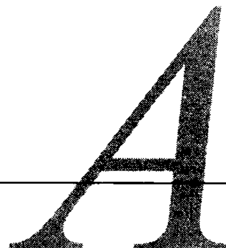
附　　录

本部分内容

- 附录 A　参考文献
- 附录 B　简单的解析器

附录A

参考文献



A.1 图书

Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford, UK: Oxford University Press, 1977.

Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Englewood Cliffs, NJ: Prentice Hall PTR, 2001.

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 1999.

Fogel, Karl, and Moshe Bar., *Open Source Development with CVS, Third Edition*. Scottsdale, AZ: Paraglyph Press, 2003.

Fowler, Martin, and Kendall Scott. *UML Distilled, Second Edition: A Brief Guide to the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1999.

Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

Fowler, Martin. *Patterns of Enterprise Application Architecture*. Reading, MA: Addison-Wesley, 2003.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

Hunt, Andrew, and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Reading, MA: Addison-Wesley, 2000.

Kerievsky, Joshua. *Refactoring to Patterns*. Reading, MA: Addison-Wesley, 2004.

Metsker, Steven John. *Building Parsers with Java*. Reading, MA: Addison-Wesley, 2001.

Nock, Clifton. *Data Access Patterns:Database Interactions in Object-Oriented Applications*. Reading, MA: Addison-Wesley, 2004.

Shalloway, Alan, and James R Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Reading, MA: Addison Wesley, 2002.

Stelting, Stephen, and Olav Maasen. *Applied Java Patterns*. Palo Alto, CA: Sun Microsystems Press, 2002.

A.2 文章

Beaver, Greg. "Setting Up Your Own PEAR Channel with Chiara_Pear_Server—The Official Way." http://greg.chiaraquartet.net/archives/123-Setting-up-your-own-PEAR-channel-the-official-way.html

Beck, Kent, and Erich Gamma. "Test Infected: Programmers Love Writing Tests." http://junit.sourceforge.net/doc/testinfected/testing.htm

Collins-Sussman, Ben, Brian W. Fitzpatrick, C. Michael Pilato. "Version Control with Subversion" http://svnbook.red-bean.com/

Lerdorf, Rasmus. "PHP/FI Brief History." http://www.php.net/manual/phpfi2.php#history

Suraski, Zeev. "The Object-Oriented Evolution of PHP." http://www.devx.com/webdev/Article/10007/0/ page/1

A.3 网站

Bugzilla: http://www.bugzilla.org

CruiseControl: http://cruisecontrol.sourceforge.net/

CVS: http://www.cvshome.org/

CvsGui: http://www.wincvs.org/

CVSNT: http://www.cvsnt.org/wiki

DokuWiki: http://wiki.splitbrain.org/wiki:dokuwiki

Foswiki: http://foswiki.org/

Eclipse: http://www.eclipse.org/

Java: http://www.java.com

GNU: http://www.gnu.org/

Git: http://git-scm.com/

Google Code: http://code.google.com

Mailman: http://www.gnu.org/software/mailman/

Martin Fowler: http://www.martinfowler.com/

Memcached: http://danga.com/memcached/

Phing: http://phing.info/trac/

PHPUnit: http://www.phpunit.de

PhpWiki: http://phpwiki.sourceforge.net

PEAR: http://pear.php.net

PECL: http://pecl.php.net/

Phing: http://phing.info/

PHP: http://www.php.net

PhpWiki: http://phpwiki.sourceforge.net

PHPDocumentor: http://www.phpdoc.org/

Portland Pattern Repository's Wiki (Ward Cunningham): http://www.c2.com/cgi/wiki

Pyrus: http://pear2.php.net

RapidSVN: http://rapidsvn.tigris.org/

QDB: http://www.bash.org

Selenium: http://seleniumhq.org/

SPL: http://www.php.net/spl

Subversion: http://subversion.apache.org/

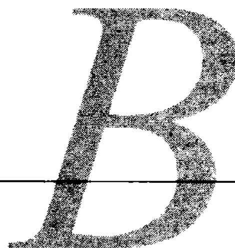
Ximbiot—CVS Wiki: http://ximbiot.com/cvs/wiki/

Xdebug: http://xdebug.org/

Zend: http://www.zend.com

附录 B

简单的解析器



第11章介绍的解释器模式并没有涉及语言解析。不包含解析器（parser）的解释器显然是不完整的，因为用户需要自己编写PHP代码来调用解释器。第三方解析器通过配置可以有效地与解释器模式进行结合，而在实际的项目中这也是最好的选择。本附录将介绍一个非常简单的面向对象解析器，它可以和第11章构建的MarkLogic语言的解释器一起工作。要注意，这些例子只是为了清楚地描述概念，而并不适合应用于真实世界中。

注解 本附录将介绍的解析器代码及其主要结构基于Steven Metsker所著的*Building Parsers with Java*（Addison-Wesley, 2001）。也许我所写的版本过于简单，但应该可以说明问题。Steven允许本书使用他的创意，非常感谢。

B.1 扫描器

要解析一条语句，必须先将它拆分成一些单词和字符（常称为令牌[1]）。下面的类使用一些正则表达式来定义令牌。这段程序还提供了方便的结果栈，这将在后面被用到。下面是Scanner类的代码：

```
namespace gi\parse;

class Scanner {

    // token类型
    const WORD          = 1;
    const QUOTE         = 2;
    const APOS          = 3;
    const WHITESPACE    = 6;
    const EOL           = 8;
    const CHAR          = 9;
    const EOF           = 0;
    const SOF           = -1;
```

① token，即特定的字符。——译者注

```
protected $line_no = 1;
protected $char_no = 0;
protected $token = null;
protected $token_type = -1;
// Reader用于访问原始字符，Context则用于保存结果数据
function __construct( Reader $r, Context $context ) {
    $this->r = $r;
    $this->context = $context;
}

function getContext() {
    return $this->context;
}

// 读取所有的空白字符
function eatWhiteSpace( ) {
    $ret = 0;
    if ( $this->token_type != self::WHITESPACE &&
         $this->token_type != self::EOL ) {
        return $ret;
    }
    while ( $this->nextToken() == self::WHITESPACE ||
            $this->token_type == self::EOL ) {
        $ret++;
    }
    return $ret;
}

// 获得当前令牌类型所对应的字符串或者由$int参数对应的类型
function getTypeString( $int=-1 ) {
    if ( $int<0 ) { $int=$this->tokenType(); }
    if ( $int<0 ) { return null; }
    $resolve = array(
        self::WORD =>       'WORD',
        self::QUOTE =>      'QUOTE',
        self::APOS =>       'APOS',
        self::WHITESPACE => 'WHITESPACE',
        self::EOL =>        'EOL',
        self::CHAR =>       'CHAR',
        self::EOF =>        'EOF' );
    return $resolve[$int];
}

// 当前令牌类型（由整型表示）
function tokenType() {
    return $this->token_type;
}

// 获得当前令牌的内容
function token() {
    return $this->token;
}

// 如果当前令牌是单词，则返回true
```

```
function isWord( ) {
    return ( $this->token_type == self::WORD );
}

// 如果当前令牌是引号，则返回true
function isQuote( ) {
    return ( $this->token_type == self::APOS ||
             $this->token_type == self::QUOTE );
}

// 源语句中当前行数
function line_no() {
    return $this->line_no;
}

// 源语句中当前字符位置
function char_no() {
    return $this->char_no;
}

// 复制当前对象
function __clone() {
    $this->r = clone($this->r);
}

// 在源语句中前进到下一个令牌。设置当前令牌并跟踪当前行数及字符数
function nextToken() {
    $this->token = null;
    $type;
    while ( ! is_bool($char=$this->getChar())   ) {
        if ( $this->isEolChar( $char ) ) {
            $this->token = $this->manageEolChars( $char );
            $this->line_no++;
            $this->char_no = 0;
            $type = self::EOL;
            return ( $this->token_type = self::EOL );

        } else if ( $this->isWordChar( $char ) ) {
            $this->token = $this->eatWordChars( $char );
            $type = self::WORD;

        } else if ( $this->isSpaceChar( $char ) ) {
            $this->token = $char;
            $type = self::WHITESPACE;

        } else if ( $char == "'" ) {
            $this->token = $char;
            $type = self::APOS;

        } else if ( $char == '"' ) {
            $this->token = $char;
            $type = self::QUOTE;

        } else {
```

```
                $type = self::CHAR;
                $this->token = $char;
            }

            $this->char_no += strlen( $this->token() );
            return ( $this->token_type = $type );
        }
        return ( $this->token_type = self::EOF );
    }

    // 返回包含下一个令牌的类型和内容的数组
    function peekToken() {
        $state = $this->getState();
        $type = $this->nextToken();
        $token = $this->token();
        $this->setState( $state );
        return array( $type, $token );
    }

    // 获得ScannerState对象，它用来保存解析器的当前位置及当前令牌的数据
    function getState() {
        $state = new ScannerState();
        $state->line_no       = $this->line_no;
        $state->char_no       = $this->char_no;
        $state->token         = $this->token;
        $state->token_type    = $this->token_type;
        $state->r             = clone($this->r);
        $state->context       = clone($this->context);
        return $state;
    }

    // 用ScannerState对象来恢复扫描器的状态
    function setState( ScannerState $state ) {
        $this->line_no      = $state->line_no;
        $this->char_no      = $state->char_no;
        $this->token        = $state->token;
        $this->token_type   = $state->token_type;
        $this->r            = $state->r;
        $this->context      = $state->context;
    }

    // 获得源语句中的下一个字符
    private function getChar() {
        return $this->r->getChar();
    }

    // 获得非单词字符之前的所有字符
    private function eatWordChars( $char ) {
        $val = $char;
        while ( $this->isWordChar( $char=$this->getChar() )) {
            $val .= $char;
        }
        if ( $char ) {
            $this->pushBackChar( );
```

```
        }
        return $val;
    }

    // 获得非空白字符之前的所有字符
    private function eatSpaceChars( $char ) {
        $val = $char;
        while ( $this->isSpaceChar( $char=$this->getChar() )) {
            $val .= $char;
        }
        $this->pushBackChar( );
        return $val;
    }

    // 在源语句中移动到前一个字符
    private function pushBackChar( ) {
        $this->r->pushBackChar();
    }

    // 判断参数是否为单词字符
    private function isWordChar( $char ) {
        return preg_match( "/[A-Za-z0-9_\-]/", $char );
    }

    // 判断参数是否为空白字符
    private function isSpaceChar( $char ) {
        return preg_match( "/\t| /", $char );
    }

    // 判断参数是否为换行符
    private function isEolChar( $char ) {
        return preg_match( "/\n|\r/", $char );
    }

    // 处理\n、\r或\r\n
    private function manageEolChars( $char ) {
        if ( $char == "\r" ) {
            $next_char=$this->getChar();
            if ( $next_char == "\n" ) {
                return "{$char}{$next_char}";
            } else {
                $this->pushBackChar();
            }
        }
        return $char;
    }
    function getPos() {
        return $this->r->getPos();
    }

}

class ScannerState {
    public $line_no;
```

```
    public $char_no;
    public $token;
    public $token_type;
    public $r;
}
```

我们在类的起始处定义了一些常量，以便与各种类型的令牌一一对应，包括字符、单词、空格以及引号等。我们用对应每个令牌的方法来检测类型：isWordChar()、isSpaceChar()等。类中最核心的方法就是nextToken()，它将尝试匹配给定字符串中的下一个令牌。Scanner存储着一个Context对象。Parser对象使用Context对象来获取解析完目标文本后的结果。

也请注意另一个类：ScannerState。Scanner被设计为便于Parser对象存储状态，获取状态，并且当Parser执行出错时还能恢复状态。getState()方法填充并返回ScannerState对象，而在需要时setState()则会使用ScannerState对象重置状态。

下面是Context类：

```
namespace gi\parse;
//...

class Context {
    public $resultstack = array();

    function pushResult( $mixed ) {
        array_push( $this->resultstack, $mixed );
    }

    function popResult( ) {
        return array_pop( $this->resultstack );
    }

    function resultCount() {
        return count( $this->resultstack );
    }

    function peekResult( ) {
        if ( empty( $this->resultstack ) ) {
            throw new Exception( "empty resultstack" );
        }
        return $this->resultstack[count( $this->resultstack ) -1 ];
    }
}
```

可以看到，Context其实只是一个简单的栈，一个供解析器使用的信息交流点。它的作用与解释器模式中的Context类相似，但并不是同一个类。

注意Scanner自身并不直接操作文件或者字符串，而是用Reader对象来代替自己执行操作。这样做能使我们轻松地切换数据的不同来源。下面是Reader接口及一个具体实现String-Reader：

```
namespace gi\parse;
```

```
abstract class Reader {

    abstract function getChar();
    abstract function getPos();
    abstract function pushBackChar();
}

class StringReader extends Reader {
    private $in;
    private $pos;

    function __construct( $in ) {
        $this->in = $in;
        $this->pos = 0;
    }

    function getChar() {
        if ( $this->pos >= strlen( $this->in ) ) {
            return false;
        }
        $char = substr( $this->in, $this->pos, 1 );
        $this->pos++;
        return $char;
    }

    function getPos() {
        return $this->pos;
    }

    function pushBackChar() {
        $this->pos--;
    }

    function string() {
        return $this->in;
    }
}
```

以上代码从一个字符串中一次读取一个字符。当然，我们也能轻松地实现一个从文件读取字符的版本。

理解Scanner的最好途径就是使用它。下面的代码将我们的测试语句拆分成令牌：

```
$context = new \gi\parse\Context();
$user_in = "\$input equals '4' or \$input equals 'four'";
$reader = new \gi\parse\StringReader( $user_in );
$scanner = new \gi\parse\Scanner( $reader, $context );

while ( $scanner->nextToken() != \gi\parse\Scanner::EOF ) {
    print $scanner->token();
    print "\t{$scanner->char_no()}";
    print "\t{$scanner->getTypeString()}\n";
}
```

我们先初始化一个Scanner对象，然后通过重复调用nextToken()方法来迭代给定字符串中的令牌。token()方法返回的是被匹配到的字符。char_no()则告诉我们匹配的字符在字符串中的位置。getTypeString()方法返回的是代表当前令牌的一个常量字符串。该程序的输出结果如下所示：

```
$       1       CHAR
input   6       WORD
        7       WHITESPACE
equals  13      WORD
        14      WHITESPACE
'       15      APOS
4       16      WORD
'       17      APOS
        18      WHITESPACE
or      20      WORD
        21      WHITESPACE
$       22      CHAR
input   27      WORD
        28      WHITESPACE
equals  34      WORD
        35      WHITESPACE
'       36      APOS
four    40      WORD
'       41      APOS
```

当然，我们可以匹配出更小的令牌，但显然上述结果已经足够满足我们的要求了。拆分字符串相对比较简单，但接下来如何通过代码创建语法呢？

B.2　解析器

创建语法的一个办法是创建Parser对象树。下面是我们将使用的抽象类Parser的代码：

```
namespace gi\parse;
abstract class Parser {

    const GIP_RESPECTSPACE = 1;
    protected $respectSpace = false;
    protected static $debug = false;
    protected   $discard = false;
    protected $name;
    private static $count=0;

    function __construct( $name=null, $options=null ) {
        if ( is_null( $name ) ) {
            self::$count++;
            $this->name = get_class( $this )." (".self::$count.")";
        } else {
            $this->name = $name;
        }
        if ( is_array( $options ) ) {
            if ( isset( $options[self::GIP_RESPECTSPACE] ) ) {
```

```
                $this->respectSpace=true;
            }
        }
    }

    protected function next( Scanner $scanner ) {
        $scanner->nextToken();
        if ( ! $this->respectSpace ) {
            $scanner->eatWhiteSpace();
        }
    }

    function spaceSignificant( $bool ) {
        $this->respectSpace = $bool;
    }

    static function setDebug( $bool ) {
        self::$debug = $bool;
    }

    function setHandler( Handler $handler ) {
        $this->handler = $handler;
    }

    final function scan( Scanner $scanner ) {
        if ( $scanner->tokenType() == Scanner::SOF ) {
            $scanner->nextToken();
        }
        $ret = $this->doScan( $scanner );
        if ( $ret && ! $this->discard && $this->term() ) {
            $this->push( $scanner );
        }
        if ( $ret ) {
            $this->invokeHandler( $scanner );
        }

        if ( $this->term() && $ret ) {
            $this->next( $scanner );
        }
        $this->report("::scan returning $ret");
        return $ret;
    }

    function discard() {
        $this->discard = true;
    }

    abstract function trigger( Scanner $scanner );

    function term() {
        return true;
    }

// private/protected
```

```
    protected function invokeHandler(
            Scanner $scanner ) {
        if ( ! empty( $this->handler ) ) {
            $this->report( "calling handler: ".get_class( $this->handler ) );
            $this->handler->handleMatch( $this, $scanner );
        }
    }

    protected function report( $msg ) {
        if ( self::$debug ) {
            print "<{$this->name}> ".get_class( $this ).": $msg\n";
        }
    }

    protected function push( Scanner $scanner ) {
        $context = $scanner->getContext();
        $context->pushResult( $scanner->token() );
    }

    abstract protected function doScan( Scanner $scan );
}
```

我们先看一下scan()方法，它是大部分业务逻辑所在的地方，其参数是一个Scanner对象。Parse首先做的就是调用抽象的doScan()方法，然后把工作委托给具体的子类。doScan()返回true或者false，稍后我们将给出具体子类的示例。

如果doScan()报告成功，并且其他一些条件都满足，那么解析结果将被压入context对象的结果栈中。Scanner对象中保存的Context被Parser对象用于访问结果。实际上，成功解析的结果压入栈发生在Parser::push()方法中。

```
protected function push( Scanner $scanner ) {
    $context = $scanner->getContext();
    $context->pushResult( $scanner->token() );
}
```

除了解析失败之外，有两个条件能预防错误结果被推入Scanner对象的栈中。首先，客户端代码可以调用discard()方法来使Parser放弃一个成功的匹配。这会将属性$discard设为true。其次，只有终端解析器（即不由其他解析器组成）应该把它们的结果压入栈中。组合解析器（它是CollectionParser的实例，也称为集合解析器）则会让它们的子解析器来替它压入结果。我们使用term()方法来检测一个解析器是否是终端解析器。在集合解析器中，term()方法会被覆盖来返回false。

如果某个具体的解析器成功地匹配，则我们将调用另一个方法invokeHandler()。一个Scanner对象会被传递给该方法。如果Handler（即实现了Handler接口的对象）对象被添加到这个Parser中（使用setHandler()方法），那么它的handleMatch()方法会被调用。我们使用Handler来使语法解析成功后做些真正有用的工作，下面很快就会介绍相关内容。

再回过头来看一下scan()方法，在返回doScan()的结果之前，我们调用了Scanner对象的

nextToke()和eatWhiteSpace()方法来移动它的位置。

除了doScan()方法之外，再注意一下抽象方法trigger()。trigger()被用于判断一个解析器是否应该去尝试匹配。如果trigger()返回false，则解析将不执行。我们来看一个具体的终端解析器——CharacterParse，其设计目的是匹配某个特定的字符：

```
namespace gi\parse;

class CharacterParse extends Parser {
    private $char;

    function __construct( $char, $name=null, $options=null ) {
        parent::__construct( $name, $options );
        $this->char = $char;
    }

    function trigger( Scanner $scanner ) {
        return ( $scanner->token() == $this->char );
    }

    protected function doScan( Scanner $scanner ) {
        return ( $this->trigger( $scanner ) );
    }
}
```

构造方法的参数是要匹配的字符，还有一个可选参数是调试用的解析器的名称。trigger()方法只检查Scanner当前指向的字符令牌是否跟构造函数传入的字符一致。因为不需要进一步扫描，所以doScan()方法只调用了trigger()方法。

可以看到，终端匹配（terminal matching）相当简单。下面再看一下集合解析器。首先定义一个通用的父类，然后继续创建一个具体的例子。

```
namespace gi/parse;

// 本抽象类拥有子解析器
abstract class CollectionParse extends Parser {
    protected $parsers = array();

    function add( Parser $p ) {
        if ( is_null( $p ) ) {
            throw new Exception( "argument is null" );
        }
        $this->parsers[]= $p;
        return $p;
    }

    function term() {
        return false;
    }
}

class SequenceParse extends CollectionParse {
```

```
    function trigger( Scanner $scanner ) {
        if ( empty( $this->parsers ) ) {
            return false;
        }
        return $this->parsers[0]->trigger( $scanner );
    }

    protected function doScan( Scanner $scanner ) {
        $start_state = $scanner->getState();
        foreach( $this->parsers as $parser ) {
            if ( ! ( $parser->trigger( $scanner ) &&
                    $scan=$parser->scan( $scanner )) ) {
                $scanner->setState( $start_state );
                return false;
            }
        }
        return true;
    }
}
```

CollectionParse抽象类只实现了一个add()方法来添加解析器，并覆盖Item()方法使其返回false[①]。

SequenceParse::trigger()只检测它包含的第一个子Parser，然后调用其trigger()方法。被调用的Parser会先调用CollectionParse::trigger()来判断是否有必要调用CollectionParse::scan()。一旦CollectionParse::scan()被调用，doScan()也将被调用，且所有子Parser的trigger()和scan()方法都将被依次执行。如果CollectionParse::doScan()中的某个结果是失败的，则doScan()报告执行失败并退出。

解析的主要问题在于测试。SequenceParse对象的每个总解析器中，会包含完整的解析器树。这些解析器会根据一个或多个令牌启动Scanner，结果将通过Context对象注册。如果Parser列表中的最后一项返回false，那么SequenceParse怎么处理其他项注册到Context中的结果呢？要么所有测试都成功，否则队列没有意义，因此我们没有选择，只能让Context和Scanner回溯。为此，需要在doScan()的开始保存状态。并在返回false或失效之前调用setState()。当然，如果返回的是true，就用不着回溯了。

为求完整，下面列出了Parser系列类中的其他类：

```
namespace gi\parse;

// 此解析器在一个或多个子解析器匹配时匹配
class RepetitionParse extends CollectionParse {
    private $min;
    private $max;

    function __construct( $min=0, $max=0, $name=null, $options=null ) {
        parent::__construct( $name, $options );
        if ( $max < $min && $max > 0 ) {
```

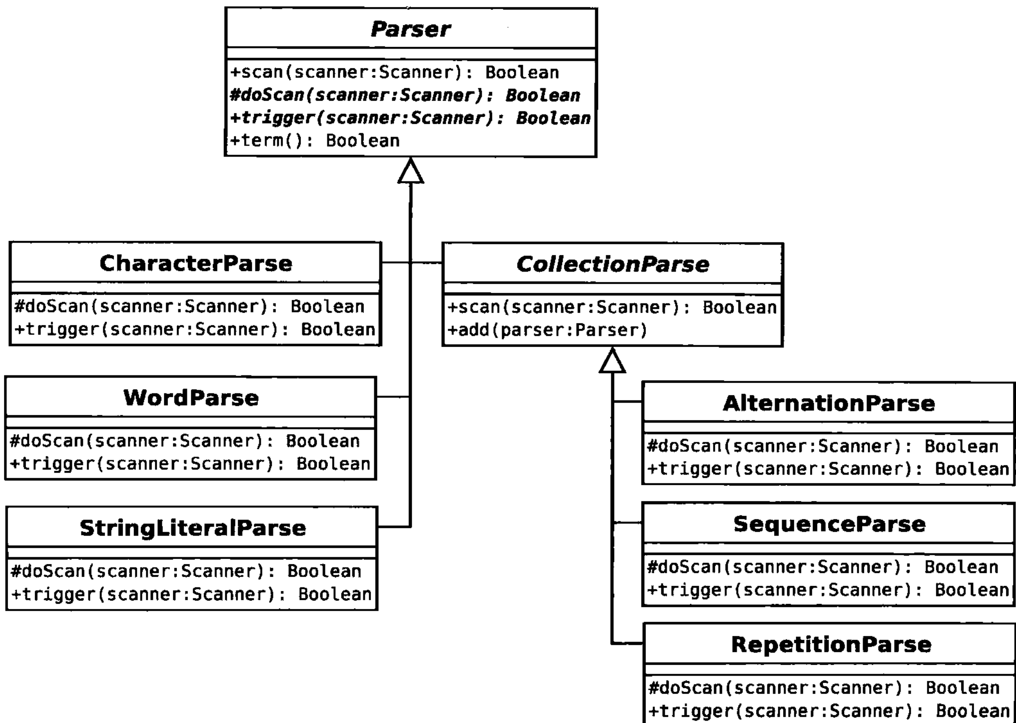
① 表明自己不是一个终端解析器。——译者注

```
            throw new Exception(
                "maximum ( $max ) larger than minimum ( $min )");
        }
        $this->min = $min;
        $this->max = $max;
    }

    function trigger( Scanner $scanner ) {
        return true;
    }

    protected function doScan( Scanner $scanner ) {
        $start_state = $scanner->getState();
        if ( empty( $this->parsers ) ) {
            return true;
        }
        $parser = $this->parsers[0];
        $count = 0;

        while ( true ) {
            if ( $this->max > 0 && $count >= $this->max ) {
                return true;
            }

            if ( ! $parser->trigger( $scanner ) ) {
                if ( $this->min == 0 || $count >= $this->min ) {
                    return true;
                } else {
                    $scanner->setState( $start_state );
                    return false;
                }
            }
            if ( ! $parser->scan( $scanner ) ) {
                if ( $this->min == 0 || $count >= $this->min ) {
                    return true;
                } else {
                    $scanner->setState( $start_state );
                    return false;
                }
            }
            $count++;
        }
    return true;
    }
}

// 此解析器在两个子解析器之一匹配时匹配
class AlternationParse extends CollectionParse {

    function trigger( Scanner $scanner ) {
        foreach ( $this->parsers as $parser ) {
            if ( $parser->trigger( $scanner ) ) {
                return true;
            }
```

```
        }
        return false;
    }

    protected function doScan( Scanner $scanner ) {
        $type = $scanner->tokenType();
        foreach ( $this->parsers as $parser ) {
            $start_state = $scanner->getState();
            if ( $type == $parser->trigger( $scanner ) &&
                 $parser->scan( $scanner ) ) {
                 return true;
            }
        }
        $scanner->setState( $start_state );
        return false;
    }
}

// 此终端解析器匹配一个字符串常量
class StringLiteralParse extends Parser {

    function trigger( Scanner $scanner ) {
        return ( $scanner->tokenType() == Scanner::APOS ||
                 $scanner->tokenType() == Scanner::QUOTE );
    }

    protected function push( Scanner $scanner ) {
        return;
    }

    protected function doScan( Scanner $scanner ) {
        $quotechar = $scanner->tokenType();
        $ret = false;
        $string = "";
        while ( $token = $scanner->nextToken() ) {
            if ( $token == $quotechar ) {
                $ret = true;
                break;
            }
            $string .= $scanner->token();
        }

        if ( $string && ! $this->discard ) {
            $scanner->getContext()->pushResult( $string );
        }

        return $ret;
    }
}

// 此终端解析器匹配一个词
class WordParse extends Parser {

    function __construct( $word=null, $name=null, $options=null ) {
```

```
        parent::__construct( $name, $options );
        $this->word = $word;
    }

    function trigger( Scanner $scanner ) {
        if ( $scanner->tokenType() != Scanner::WORD ) {
            return false;
        }
        if ( is_null( $this->word ) ) {
            return true;
        }
        return ( $this->word == $scanner->token() );
    }

    protected function doScan( Scanner $scanner ) {
        $ret = ( $this->trigger( $scanner ) );
        return $ret;
    }
}
```

通过组合终端和非终端的Parser对象，可以打造一个合理完善的解析器。示例程序中所用到的所有Parser类如图B-1所示。



图B-1 Parser系列类

使用组合模式的用意是让客户在创建语法时符合EBNF规范。表B-1列出了类和对应的EBNF片段。

表B-1　组合解析器和EBNF

类	EBNF范例	描　述
AlternationParse	orExpr|andExpr	一个或者另一个
SequenceParse	'and' operand	一个列表（按指定顺序）
RepetitionParse	(eqExpr)*	0个或者多个

接下来写一些客户端代码来实现我们的迷你语言。下面的EBNF片段在第11章中提到过：

```
expr     ::= operand (orExpr | andExpr )*
operand  ::= ( '(' expr ')' | <stringLiteral> | variable ) ( eqExpr )*
orExpr   ::= 'or' operand
andExpr  ::= 'and' operand
eqExpr   ::= 'equals' operand
variable ::= '$' <word>
```

下面这个简单的类基于上面的EBNF片段来创建语法：

```
class MarkParse {
    private $expression;
    private $operand;
    private $interpreter;
    private $context;

    function __construct( $statement ) {
        $this->compile( $statement );
    }

    function evaluate( $input ) {
        $icontext = new InterpreterContext();
        $prefab = new VariableExpression('input', $input );
        // 添加输入变量至Context
        $prefaB->interpret( $icontext );

        $this->interpreter->interpret( $icontext );
        $result = $icontext->lookup( $this->interpreter );
        return $result;
    }

    function compile( $statement_str ) {
        // 建立解析树
        $context = new \gi\parse\Context();
        $scanner = new \gi\parse\Scanner(
            new \gi\parse\StringReader($statement_str), $context );
        $statement = $this->expression();
        $scanresult = $statement->scan( $scanner );

        if ( ! $scanresult || $scanner->tokenType() != \gi\parse\Scanner::EOF ) {
            $msg  = "";
```

```
            $msg .= " line: {$scanner->line_no()} ";
            $msg .= " char: {$scanner->char_no()}";
            $msg .= " token: {$scanner->token()}\n";
            throw new Exception( $msg );
        }

        $this->interpreter = $scanner->getContext()->popResult();
    }

    function expression() {
        if ( ! isset( $this->expression ) ) {
            $this->expression = new \gi\parse\SequenceParse();
            $this->expression->add( $this->operand() );
            $bools = new \gi\parse\RepetitionParse( );
            $whichbool = new \gi\parse\AlternationParse();
            $whichbool->add( $this->orExpr() );
            $whichbool->add( $this->andExpr() );
            $bools->add( $whichbool );
            $this->expression->add( $bools );
        }
        return $this->expression;
    }

    function orExpr() {
        $or = new \gi\parse\SequenceParse( );
        $or->add( new \gi\parse\WordParse('or') )->discard();
        $or->add( $this->operand() );
        $or->setHandler( new BooleanOrHandler() );
        return $or;
    }

    function andExpr() {
        $and = new \gi\parse\SequenceParse();
        $and->add( new \gi\parse\WordParse('and') )->discard();
        $and->add( $this->operand() );
        $and->setHandler( new BooleanAndHandler() );
        return $and;
    }

    function operand() {
        if ( ! isset( $this->operand ) ) {
            $this->operand = new \gi\parse\SequenceParse( );
            $comp = new \gi\parse\AlternationParse( );
            $exp = new \gi\parse\SequenceParse( );
            $exp->add( new \gi\parse\CharacterParse( '(' ))->discard();
            $exp->add( $this->expression() );
            $exp->add( new \gi\parse\CharacterParse( ')' ))->discard();
            $comp->add( $exp );
            $comp->add( new \gi\parse\StringLiteralParse() )
                ->setHandler( new StringLiteralHandler() );
            $comp->add( $this->variable() );
            $this->operand->add( $comp );
            $this->operand->add( new \gi\parse\RepetitionParse( ) )
                ->add($this->eqExpr());
```

```
            }
            return $this->operand;
        }

        function eqExpr() {
            $equals = new \gi\parse\SequenceParse();
            $equals->add( new \gi\parse\WordParse('equals') )->discard();
            $equals->add( $this->operand() );
            $equals->setHandler( new EqualsHandler() );
            return $equals;
        }

        function variable() {
            $variable = new \gi\parse\SequenceParse();
            $variable->add( new \gi\parse\CharacterParse( '$' ))->discard();
            $variable->add( new \gi\parse\WordParse());
            $variable->setHandler( new VariableHandler() );
            return $variable;
        }
    }
```

这个类看起来比较复杂，但其实它所做的只是创建我们已经定义过的语法。大部分方法的名称与EBNF部件名称（即每行EBNF的开头，如eqExpr和andExpr）相似。看一下expression()方法，可以看到我们创建的规则与先前在EBNF中定义的规则相同：

```
    // expr     ::= operand (orExpr | andExpr )*
        function expression() {
            if ( ! isset( $this->expression ) ) {
                $this->expression = new \gi\parse\SequenceParse();
                $this->expression->add( $this->operand() );
                $bools = new \gi\parse\RepetitionParse( );
                $whichbool = new \gi\parse\AlternationParse();
                $whichbool->add( $this->orExpr() );
                $whichbool->add( $this->andExpr() );
                $bools->add( $whichbool );
                $this->expression->add( $bools );
            }
            return $this->expression;
        }
```

在代码和EBNF符号中，我们定义了由相关操作符构成的序列，紧跟着orExpr或andExpr的零个或多个实例。注意，我们将该方法返回的Parser保存至一个属性变量中，这是为了避免从中调用expression()时出现死循环。

除创建语法的方法外，类中只有两个方法：compile()和evaluate()。compile()可以通过构造函数直接或者自动被调用，该方法要求传入一个字符串句子并使用它创建一个Scanner对象。compile()方法会调用返回组成语法的Parser对象树的expression()方法，然后compile()会将Scanner对象传给Parser::scan()方法。如果原生代码没有解析，compile()将抛出一个异常。反之，将获得Scanner对象的结果栈中的编译结果，即Expression对象。结果将被存放在$interpreter属性中。

evaluate()方法为Expression树创建了一个值变量。它预定义了一个名为input的Variable Expression对象，然后将其注册给Context对象，再将Context对象传给主Expression对象。$input变量类似于PHP中的$_REQUEST变量，可以在MarkLogic里直接使用。

注解　查看第11章，可以获得更多关于VariableExpression类的信息，该类是解释器模式示例的一部分。

evaluate()方法通过调用Expression::interpret()方法来生成最终的结果。记住，我们需要从Context对象中取得解释器的结果集。

到目前为止，我们知道了如何解析文本以及如何创建语法，也通过第11章知道了如何使用解释器模式去组合Expression对象并处理查询。但还需要知道的是，如何结合这两个处理过程，怎样在解释器中获得一个解析树。解决办法是在与Parser对象相关的Handler对象中使用Parser::setHandler()方法。下面看一下管理变量的方法。我们在variable()方法中将VariableHandler与Parser关联：

```
$variable->setHandler( new VariableHandler() );
```

下面是Handler接口：

```
namespace gi\parse;

interface Handler {
    function handleMatch(   Parser $parser,
                            Scanner $scanner );
}
```

下面是VariableHandler：

```
class VariableHandler implements \gi\parse\Handler {
    function handleMatch( \gi\parse\Parser $parser, \gi\parse\Scanner $scanner ) {
        $varname = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult( new VariableExpression( $varname ) );
    }
}
```

如果VariableHandler所关联的解析器与一个扫描操作相匹配，那么handleMatch()会被调用。根据定义，栈中最后一项便是变量的名称。我们删除这个变量，并用具有正确名称的VariableExpression对象来取代它。用类似的方法，我们可以创建EqualsExpression和LiteralExpression等对象。

下面是剩下的处理程序代码：

```
class StringLiteralHandler implements \gi\parse\Handler {
    function handleMatch( \gi\parse\Parser $parser, \gi\parse\Scanner $scanner ) {
        $value = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult( new LiteralExpression( $value ) );
    }
}
```

```
class EqualsHandler implements \gi\parse\Handler {
    function handleMatch( \gi\parse\Parser $parser, \gi\parse\Scanner $scanner ) {
        $comp1 = $scanner->getContext()->popResult();
        $comp2 = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
            new EqualsExpression( $comp1, $comp2 ) );
    }
}

class BooleanOrHandler implements \gi\parse\Handler {
    function handleMatch( \gi\parse\Parser $parser, \gi\parse\Scanner $scanner ) {
        $comp1 = $scanner->getContext()->popResult();
        $comp2 = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
            new BooleanOrExpression( $comp1, $comp2 ) );
    }
}

class BooleanAndHandler implements \gi\parse\Handler {
    function handleMatch( \gi\parse\Parser $parser, \gi\parse\Scanner $scanner ) {
        $comp1 = $scanner->getContext()->popResult();
        $comp2 = $scanner->getContext()->popResult();
        $scanner->getContext()->pushResult(
            new BooleanAndExpression( $comp1, $comp2 ) );
    }
}
```

记住，我们仍需要使用第11章中的解释器示例，可以像下面这样来使用MarkParse类：

```
$input      = 'five';
$statement = "( \$input equals 'five')";

$engine = new MarkParse( $statement );
$result = $engine->evaluate( $input );
print "input: $input evaluating: $statement\n";
if ( $result ) {
    print "true!\n";
} else {
    print "false!\n";
}
```

执行代码会产生如下输出：

```
input: five evaluating: ( $input equals 'five')
true!
```

[General Information]
书名=深入PHP　面向对象、模式与实践
作者=（美）赞德斯彻著
页数=450
SS号=12802239
出版日期=2011.07