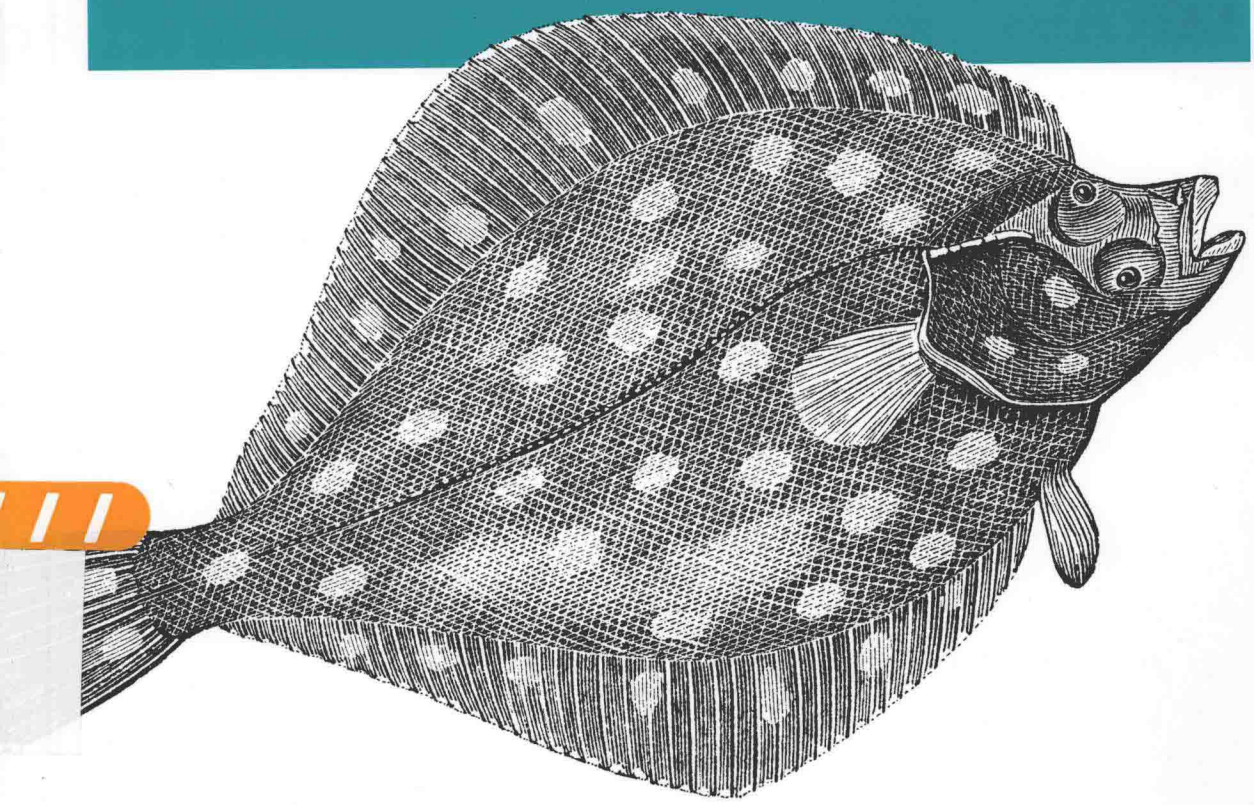
Learning PHP Design Patterns

Learning

PHP设计模式



O'REILLY®

中国电力出版社

William Sanders 著

苏金国 王宇飞 等译

Learning PHP设计模式

通过学习如何在代码中使用设计模式，可以更高效地构建服务器端应用，在这个过程中，你的PHP编程水平也将逐步提高。本书利用大量浅显易懂的例子告诉你如何应用多种面向对象模式，并展示了这些模式在一些成熟的实际项目中的具体应用。

需要学习这些可重用的模式如何帮助你解决复杂的问题，如何组织面向对象代码，以及只改变一些小部分来完成整个大项目的修改。利用你手上的这本《Learning PHP设计模式》，将了解如何采用一种更精巧的编程风格，这将大大减少开发时间。

- 学习设计模式概念，包括如何选择模式来处理特定的问题。
- 对面向对象编程概念有一个概要了解，如组合、封装、多态和继承。
- 应用创建型设计模式动态地创建页面（采用一种工厂方法而不是直接实例化）。
- 使用结构型设计模式对原有的对象或结构做出修改，而无需改变原来的代码。
- 使用行为型模式帮助对象协同工作来完成任务。
- 使用代理和职责链等行为型模式与MySQL交互。
- 探索使用PHP内置设计模式接口的方法。

William Sanders博士，哈特福德大学多媒体Web设计和开发方向教授。多年来一直积极地参与PHP设计模式方面的工作。作为《ActionScript 3.0 Design Patterns》（O'Reilly）一书的合著者，他还出版过50余本计算机以及与计算机相关的图书。

O'REILLY®
oreilly.com.cn

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-5123-5272-8
9 787512 352728 >

定价：58.00元

Learning PHP设计模式

William Sanders 著
苏金国 王宇飞 等译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

Learning PHP设计模式/（美）桑德（Sanders, W.）著；苏金国等译. －北京：中国电力出版社，2014.2

书名原文：Learning PHP Design Patterns

ISBN 978-7-5123-5272-8

I. ①L… II. ①桑… ②苏… III. ①PHP语言－程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2013）第285791号

北京市版权局著作权合同登记

图字：01-2013-6930号

封面设计/ Karen Montgomery，张健

出版发行/ 中国电力出版社（http://www.cepp.sgcc.com.cn）

地　　址/ 北京市东城区北京站西街19号（邮政编码100005）

经　　销/ 全国新华书店

印　　刷/ 北京丰源印刷厂

开　　本/ 787毫米×980毫米　16开本　21印张　385千字

版　　次/ 2014年2月第一版　2014年2月第一次印刷

印　　数/ 0001－3000册

定　　价/ 58.00元（册）

Learning PHP设计模式

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

谨以此书纪念我的父亲William B. Sanders（1917－2012）。

目录

第5部分 MySQL和PHP设计模式

前言

随着PHP日益成为很多程序员首选的服务器端程序，将专业技术和编程结构相结合已经势在必行。设计模式（Design patterns）这一概念借用自Christopher Alexander的《The Timeless Way of Building》（牛津大学出版社）一书，是指对给定上下文（环境）中某个经常出现的问题得出的一种一般性可重用的解决方案。在日常的开发工作中，PHP程序员总会在某种软件开发环境中遇到“经常出现的问题”，PHP设计模式就是针对这些“经常出现”的PHP编程问题提出的一组解决方案。简单地说，PHP设计模式是用来处理专业软件开发现实问题的工具。

它们并不是具体的库或模板，而是可以用来解决问题的更为一般性的结构。我总喜欢把设计模式想成是建立循环结构。需要处理某种迭代时就会使用循环。没错，当然也可以用其他方法来处理迭代，不过循环确实是一个非常灵活的工具，可以大大节省开发过程的时间。（你是愿意编写10000次相同的代码行还是更愿意使用循环呢？相比之下，使用循环实在是简洁得多）！

另外，如果说循环是迭代的一种“罐装解决方案”，可以认为设计模式也是一种“罐装解决方案”。在PHP中可以采用多种不同方式使用循环结构，如`for`语句、`while`语句以及其他类似的结构。同样，设计模式也可以采用多种不同的方式实现，这取决于所要解决的问题的本质。

不过，之所以采用设计模式，最重要的原因在于，这些模式可以为复杂的问题提供解决方案。随着开发的程序变得越来越庞大，毋庸置疑，它们也会变得越来越复杂。在面向对象编程（object-oriented programming，OOP）环境中，这种复杂性会有所降低，因为你处理的是封装的模块，而在顺序或过程式编程中，做出任何改变都有可能导致程序像一摞纸牌一样坍塌。设计模式不仅为一般性编程问题提供了解决方案，而且通过提供对

象之间的松耦合还支持对复杂的大程序做出修改。所以如果要进行修改，并不需要从头开始重新编程，即使是很庞大、很复杂的程序，也完全可以只增加必要的改动，而其他一切均保持不变。

另外，设计模式的宗旨就是重用。毕竟，程序员一直在重用同样的算法。那么为什么不按照同样的思路重用更大的结构呢？一方面，框架和模板可以支持重用，不过它们往往过于特定。这就引入了另一种做法：可以重用结合设计模式的PHP程序，特别是在庞大的复杂程序中。由于结合设计模式的程序可以很容易地做出修改，因此对相同类型的特定问题重用这些程序也很容易。减少开发时间和资源不仅可以节省成本，还可以更好地为你的客户服务。客户将得到能充分满足其功能需求的良构程序，另外开发人员也可以基于不太可能崩溃的坚实基础很容易地完成修改（毕竟，顾客的需求总是在不断改变！）。

本书面向对象

在某种程度上，所有优秀的程序员都会意识到需要跳出顺序和过程式编程的束缚，而转向下一种更合理的编程方式，这就是面向对象编程。要转向OOP，需要从观念上有所转变：并不是把编程看做是一系列语句，而应当看做是对象之间的一种交互和通信。设计模式的基础便是OOP，也就是说，OOP原则将转换为可重用代码模式。这些正是很多专业程序员使用的工具。由于编程设计模式的开发需要学术领域和商业领域的精诚合作，这些概念并不仅限于针对某一个问题，不过同时也确实可以用来处理实际问题。《PHP设计模式学习指南》面向的是专业的程序员，他们希望进一步节省其开发和再开发时间，并为客户提供高质量的代码。

特别需要指出，这本书对那些对编程充满热情的人也非常适用。他们可能认为整晚编程都不算什么，因为他们认为这很有意思，上床睡觉只是为了醒来后可以开始另一个程序。如果编程总是全新的，每天都会有新的发现或bug，开发人员就必须全力拼搏，需要用复杂而新颖的方式充分运用大脑，这种体验有点像悟禅。如果你有这种体验，应该懂得我的意思。这没有办法说清楚，也不好解释。（我自己甚至都无法解释清楚，我也不知道为什么会对编程中遇到的挑战和激励如此着迷。）

设计模式对理解力提出了更高的挑战，这本书并不适合那些对PHP和编程都很陌生的人。如果你刚开始学习PHP，可以看一看Robin Nixon的《Learning PHP, MySQL, JavaScript, and CSS, 2nd Edition》（O'Reilly），然后再来考虑PHP设计模式。另外，这本书（或所有关于设计模式的高深的书）无法承诺你能很快、很容易地掌握设计模式。这种学习是一个漫长的旅程，最明智的建议就是学着享受这个旅程。这需要花许多时间和功夫。

本书假设

这本书假设你已经知道如何用PHP编程，而且希望你的编程水平能更上几个台阶。实际上，这里假设你已经是一个很好的PHP程序员，而且用过MySQL，知道如何开发HTML页面和使用CSS。这里还假设你知道学习PHP设计模式绝不可能一蹴而就。设计模式的学习就像是一个渐近完成的蜕变。

本书内容

这本书分为5大部分。

第1部分是对OOP的一个复习/介绍：

第1章介绍面向对象编程（object-oriented programming，OOP），以及如何利用模块化更容易地处理复杂的编程问题。

第2章讨论OOP中的一些基本概念，如抽象、封装、继承和多态，以及实现这些概念的PHP结构。

第3章继续讨论设计模式中的基本概念、设计模式类别以及如何选择特定的模式来处理特定的问题。

第4章介绍统一建模语言（Unified Modeling Language，UML），并解释这本书将如何应用UML。

第2部分介绍创建型设计模式：

第5章介绍工厂方法（Factory Method）模式，其目的是创建对象，这种模式属于类设计模式。这一章提供的例子包括动态创建可以显示图片、正文体和标题正文的页面。

第6章介绍如何使用原型（Prototype）模式，其目的也是创建对象，这种模式属于对象设计模式。如果创建一个对象作为原型，然后通过克隆来创建更多实例以节省开销，这种情况就可以使用原型模式。

第3部分解释结构型设计模式：

第7章介绍如何使用类和对象适配器（Adapter）模式。这一章给出的例子展示了如何对现有的结构进行修改，从而允许开发人员增加新的功能。

第8章解释如何利用装饰器（Decorator）模式改变原有的对象而不会对更大的程序带来破坏。你会看到如何用约会网站上的不同首选项装饰男性和女性约会对象。

第4部分介绍行为型设计模式：

第9章介绍如何使用模板方法（Template Method）模式——这也是最容易创建和使

用的设计模式之一。另外，你会看到如何在设计模式编程中应用著名的好莱坞原则。这一章最后还有一个特别之处：我们将结合两个不同的模式解决一个问题。

第10章介绍了状态（State）设计模式，并指出如何使用状态图映射状态过程和变化。

第5部分介绍了结合MySQL时使用的另外4个行为型设计模式：

第11章提供了通用连接类和代理（Proxy）设计模式，可以用来为MySQL数据库中存储的用户名和口令增加安全性。

第12章解释了策略（Strategy）设计模式与状态（State）模式有哪些重要差别（尽管它们有相同的类图）。通过一个调查表例子，这一章说明了策略模式如何应用于不同的MySQL请求。

第13章给出了很多例子来说明如何使用职责链（Chain of Responsibility）模式，这些例子包括一个咨询台，以及（结合工厂方法模式）对一个日期定时器自动响应来显示图像和文本。

第14章才开始研究如何使用PHP内置设计模式接口。观察者（Observer）设计模式可以使用标准PHP库提供的接口。这一章还给出了另一个例子，将结合使用观察者设计模式和内置接口基于PHP和MySQL建立一个简单的内容管理系统（CMS）。

本书约定

本书使用以下印刷约定：

斜体（*italic*）

表示新术语、URLs、email地址、文件名和文件扩展名。

定宽字体（`Constant width`）

程序代码清单会使用定宽字体，另外在正文段落中也会用定宽字体指示程序元素，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

定宽粗体（**`Constant width bold`**）

显示需要由用户逐字键入的命令或其他文本。

定宽斜体（*`Constant width italic`*）

用定宽斜体显示的文本将替换为用户提供的值或由上下文确定的值。

注意：表示提示、建议或一般注意事项。

警告：这个图标表示警示或告诫。

使用代码示例

这本书将成为你工作的助手。一般说来，如果书中提供了代码示例，你可以在你的程序和文档中使用这些代码，除非复制使用了本书的大部分代码，否则不需要联系我们申请获得许可。例如，如果只是编写一个程序，其中用到了本书的几个代码段，这是不需要许可的。销售或发行O'Reilly图书的示例光盘则需要得到许可。如果引用本书的文字以及利用书中的示例代码回答一个问题，这不需要专门获得许可。但是如果在你的产品文档大量使用本书中的示例代码，这是需要获得许可的。

我们希望大家使用代码时能注明引用出处，但并不强求。引用通常包括书名、作者、出版商和ISBN。例如："《Learning PHP Design Patterns》，by William Sanders（O'Reilly）。版权所有 2013 William B. Sanders, 978-1-449-34491-7"。

如果你认为对代码示例的使用超出了合理的使用范畴或上述许可范围，请随时联系我们：*permissions@oreilly.com*。

Safari®图书在线

Safari图书在线（*www.safaribooksonline.com*）是一个应需而变的数字图书馆，通过图书和视频方式提供世界顶尖作者在技术和商业领域积累的专家经验。技术专家、软件开发人员、Web设计人员和企业以及有创意的专业人员都使用Safari图书在线作为其主要资源来完成研究、解决问题、深入学习和资质培训。

Safari图书在线为机构、政府部门和个人提供了多种产品组合和定价程序。订阅者可以在一个可以快捷搜索的数据库中访问多家出版社提供的成千上万种图书、培训视频和正式出版前手稿，如O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology以及其他数十家出版公司。关于Safari图书在线的更多信息，请访问我们的在线网站。

联系我们

请将关于本书的意见和问题通过以下地址提供给出版商：

美国：

O'Reilly Media，Inc.
1005 Gravenstein Highway North
Sebastopol，CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

针对这本书，我们还建有一个网页，列出了有关勘误、示例和其他信息。可以通过以下地址访问这个页面：

http://oreil.ly/php_design_patterns

如果对这本书有什么意见，或者要询问技术上的问题，请将电子邮件发至：

bookquestions@oreilly.com

要想了解O'Reilly 图书、课程、会议和新闻的更多信息，请访问我们的网站：

http://www.oreilly.com
http://www.oreilly.com.cn

查找我们：Facebook: *http://facebook.com/oreilly*

关注我们：Twitter: *http://twitter.com/oreillymedia*

查看我们：YouTube: *http://www.youtube.com/oreillymedia*

致谢

我要感谢所有以不同方式帮助过我的人。哈特福德大学与我一同完成多媒体Web设计和开发项目的同事们一直在帮助我，回答我的诸多问题。John Gray教授是我们的系主任，感谢他一如既往地鼓励和帮助我。Brian Dorn博士与我办公室相邻，经常热情地为我答疑解惑，耐心细致而且见识卓越。

能够在Northeast PHP会议开幕式上遇到Boston PHP小组的Michael Bourque真是我的幸运，感谢他对这个项目的鼓励。非常期待与Michael和Boston PHP小组更多的合作，更深入地研究高级PHP编程。

O'Reilly Media为我提供了3位能力超群的技术审校，Robin Nixon是《Learning PHP, MySQL & JavaScript, 2nd Edition》（O'Reilly）的作者，在PHP方面提出了很多修正、

建议和很有见地的想法，使代码在很多不同方面得以改进。Aaron Saray是《Professional PHP Design Patterns》（Wrox）的作者，总是详细而无私地给出他的建议。他是一位卓越的编辑，甚至最微小的瑕疵都逃不过他的眼睛。Aaron与我采用的设计模式实现方法完全不同，不过这样可以为对设计模式感兴趣的PHP开发人员提供一个更宽的视角。最后一定要提到Dmitry Sheiko，他也是一位热情的技术审校，有自己的博客，可以从中找到他实现的PHP设计模式。

O'Reilly Media的高级编辑Rachel Roumeliotis把各个部门很好地联系在一起，推动整个项目顺利进行。Maria Gulick也是一位能干的O'Reilly编辑，负责项目修订过程中遇到的种种问题。排版编辑Jasmine Kwityn总能找到和修正我根本没有注意到的细节问题。整个过程由Waterside Productions的Margot Maley Hutchison发起，永远感谢她的支持。

我的妻子Delia比别人更能理解我，因为她最近刚刚出版了她自己的一本书，很清楚这个过程的艰辛。我们忠实的瑞士山地犬WillDe对这个写作过程倒是不太留意，只要有食物，它就会乖乖地跟着你走。

第1部分

轻松掌握设计模式基础

所有妥协的前提是彼此让步，但是基本原则决不能让步。
对原则问题的妥协只能算屈服，因为只有给予，没有索取。
——圣雄甘地

人们总是先谈最基本、最高级的内容，
然后才会谈到细节的变化。
——小奥利弗·温德尔·霍姆斯

这个世界上，让人遗憾的是，
丢掉好习惯比摆脱坏习惯容易得多。
——威廉·萨默塞特·毛姆

编程习惯

从最初正式开始编程开始，多年来我养成了一些习惯，从顺序编程慢慢转向过程式编程，而且这些年经常结合使用这两种编程方式。从某些方面来讲，这是因为我总在探索不同的语言。我最早接触的语言是在大学里使用的Fortran II，然后用过Basic、FORTH、PostScript，后来转向汇编语言和机器语言。我更感兴趣的是学习不同的语言，但并不特别善于编程。接下来伴随着互联网时代，我们迎来了Java、JavaScript、PHP、C#和ActionScript 3.0，当然还有其他语言。其中大多数语言都（部分）基于C++中的某些结构。这些语言各有不同，不过我仍保留着原来的老习惯。

很偶然的机会，经人介绍我接触到Jonathan Kaye博士提出的状态机（state machines）。他向我展示了如何从不同的状态而不是从控制流的角度考虑。依照状态机，我发现了状态设计模式，后来又有幸读到Erich Gamma、Richard Helm、Ralph Johnson和John

Vlissides的大作《Design Patterns: Elements of Reusable Object-Oriented Software》（Addison-Wesley出版）。《Design Patterns》中的例子都是用SmallTalk或C++编写的。因为我不懂SmallTalk，对C++也知之甚少，所以我不得不把重心放在那些概念性的内容上。有趣的是，这倒是一件好事，因为我没有固守用某种特定语言实现的例子。至于实现PHP设计模式，这并不是从SmallTalk转换成PHP的问题，而是要将面向对象编程（object-oriented programming，OOP）和设计模式概念直接应用于PHP。

慢慢地，但很明确，我的编程习惯开始发生改变，偶尔增加一个小小的OOP设计，以及不时地增加一个设计模式。如今，除了OOP我已经不再考虑用其他方式编程。根据心理学家的说法，一个习惯的养成平均需要66天，不过对于我来说，这种改变花费的时间更长，过程也更缓慢。我总处于非常忙碌的开发阶段，一方面要完成顾客的项目，另一方面希望使用OOP和设计模式，需要在这二者之间做出选择时，时间压力总是胜出。不过，越来越多的OOP已逐步潜入我的习惯性编程实践中，尽管我还没有意识到，但实际上我的顾客得到的已经是基于OOP和设计模式开发的更为健壮可靠的应用。第1部分共包括4章，将帮助你了解OOP的有关概念：

- PHP与面向对象编程
- OOP基本概念
- 基本设计模式概念
- 结合使用设计模式和UML

注重实质，淡化形式

我认识的大多数优秀的程序员都有自己的某种风格，会形成专业的编程习惯。总的来说，看到好的OOP程序时，你会发现从命名变量到对代码增加注释等各个方面都能体现某种风格。变量名很清晰，代码中的注释能明确地描述代码，使其他程序员能清楚地知道如何将这些代码加入他们自己的模块。不过这本书中，我们将尽量缩减代码中的注释，因为对代码的解释将由这本书的正文来完成。另外，我发现，加入太多注释可能会影响对代码结构的清楚认识。所以，对我们来说，可以看到并感受到对象作为完整的实体，代码不会因为大段的注释而被“大卸八块”。（但是如果不是作为某本编程书的示例程序，我还是非常同意需要增加充分的注释。）

出于某种原因，PHP很受一些不良设计模式例子的困扰。这里所说的“不良”设计模式例子并不是指那些玩具示例，而是指编写设计模式时不够全面，总是“缺东少西”。这就类似于写一个需要终止条件的循环结构。例如，策略模式需要一个Context（上下文），就像循环结构需要一个终止条件一样。

为了尽可能准确，我使用了《Design Patterns: Elements of Reusable Object-Oriented Software》中讨论的设计模式，另外还用到了那本书中使用的统一建模语言（UML）。尽管在《Design Patterns》出版之后UML又有了新版本（UML2），不过为了便于学习PHP设计模式，另外也为了方便理解本书中没有讨论的模式，如果你想了解《Design Patterns》中提供的其他模式，学习使用原来的UML版本会有帮助。

第1章

PHP与面向对象编程

一旦一个新概念的时代到来，没有什么能阻挡它的脚步。
——维克多·雨果

不要祈望工作适合你的能力，而要祈望自己的能力去适应工作。
——菲利普斯·布鲁克斯

强大的能力来自于心中暗藏的幻想，相信自己有与生俱来的控制力量。
——安德鲁·卡内基

无知是罪恶，知识则是我们飞向天堂的翅膀。
——威廉·莎士比亚

1.1 中级和高级编程初探

开始学习阅读时，我们读到的故事都很短小，词汇量不大，而且单词也很简单。对付这些简单的小故事只需要简单的小工具。不过，随着阅读的深入，逐步接触到莎士比亚的作品时，我们就需要一组更复杂、更庞大、更高深的工具了。如果一个幼儿园老师给小朋友们讲《哈姆雷特》，孩子们很可能听不明白，不过如果随着年龄的增长，为他们提供一些渐进的阅读工具，等到他们上高中时，就能很好地阅读、理解并且深深爱上《哈姆雷特》了。这本书就是面向那些准备读“哈姆雷特”版PHP的开发人员。

要想从这本书有所收获，首先需要对PHP有一定的了解和使用经验。这一系列的另外一些图书可以提供很好的参考，如果你没有PHP经验，可以先从David Sklar的《Learning PHP 5》和Robin Nixon的《Learning PHP, MySQL, and JavaScript, 2nd Edition》入门，这两本书都由O'Reilly出版。当然，你可能已经通过很多其他书、课程或在线教程学习了PHP。关键是首先你要知道如何用PHP编程。另外，我们考虑使用PHP 5，而不会讨论

之前的版本，如PHP 4的最后版本（PHP 4.4.9）。这是因为，实现面向对象编程所需的所有内容在PHP 5之前还尚未实现。

1.2 为什么采用面向对象编程

尽管OOP出现至今已经40多年了，但仅仅在最近15年左右它才变得越来越重要。在很大程度上这是由于Java的影响，因为Java包含了内建的OOP结构。后来又出现了很多与Internet相关的更新的语言，如JavaScript、ActionScript 3.0和PHP 5，它们也在风格或结构中融合了OOP。1998年，Colgate大学的两位教授Alexander Nakhimovsky和Tom Myers撰写了《JavaScript Objects》(Wrox)，展示JavaScript可以结合OOP。所以OOP并不是一个新鲜事物，即使对于那些主要使用Internet语言编程的人来说，这也不是一个新概念，我们甚至可以说，在设计为向计算机提供指令的大多数语言中，这是一种“经过实践得到证明”的编程方法。

一定要花些时间来了解OOP，这非常重要，因为了解OOP是理解设计模式的前提。所以，尽管你可能对PHP 5有着丰富的编程经验，但如果你没有OOP经验，还是需要多花些时间来学习第1部分。

1.2.1 解决问题更容易

设计计算机程序就是为了解决人类的问题。这里有一个称为“动态编程”的过程，这是一种将大问题分解为小问题的技术。其策略是先解决各个较小的问题，然后把所有结果汇总在一起形成一个更大的解决方案。例如，假设你计划前往廷巴克图（Timbuktu）[译注1]旅行，（听上去这可能不是一个复杂的问题，不过你可以自己试试看，看能不能在一个在线旅游网站上找到从你所在的城市到达廷巴克图的航班。）下面来分解这个问题：

1. 廷巴克图（也叫做Tombouctou或Timbuctu）是否存在？（是/否）：答案 = 是。
2. 廷巴克图有机场吗？（是/否）：答案 = 是，机场标识= TOM。
3. 有没有前往TOM的航班？（是/否）答案 = 可能有。巴马科（Bamako）和莫普提（Mopti）都有去廷巴克图的航班，不过伊斯兰反对派从2012年7月1日起控制了廷巴克图，在得到进一步通知之前，航班均被取消。
4. 反对派现在是否还控制着廷巴克图？（是/否）：如果答案 = 是，没有航班；如果答案 = 否，可能有航班。

译注1：马里中部的一个城市，靠近尼日尔河，位于巴马科东北部。始建于11世纪，在14世纪成为主要贸易中心（以金和盐贸易为主），1593年被摩洛哥人洗劫，从此不再有昔日的辉煌。

5. 如果有航班，廷巴克图的旅游业或商业能保证安全吗？（是/否）：答案 = 否。

6. 从我的国家有到马里（廷巴克图所在国家）的签证吗？（是/否）：答案 = 是。

7. 需要预防接种疫苗吗？（是/否）：答案 = 是。

可以看到，前往和离开廷巴克图是一件很复杂的事情，不过上面只是一些简单的问题，只需要回答“是”或者“否”。这个列表中还可以包括更多问题，不过每个问题的答案都可以只有两种选择。如果回答“可能”，说明还需要询问更多问题来得到一个明确的是/否答案。

1.2.2 模块化

把一个问题分解为小的子问题，这个过程就是模块化（modularization）过程。从你的城市前往廷巴克图很复杂，但是这种复杂性可以通过模块化转化为一组“是/否”步骤，与此类似，任何其他复杂的问题也可以采用这种方式模块化。图1-1用图解的方式描述了这个过程。

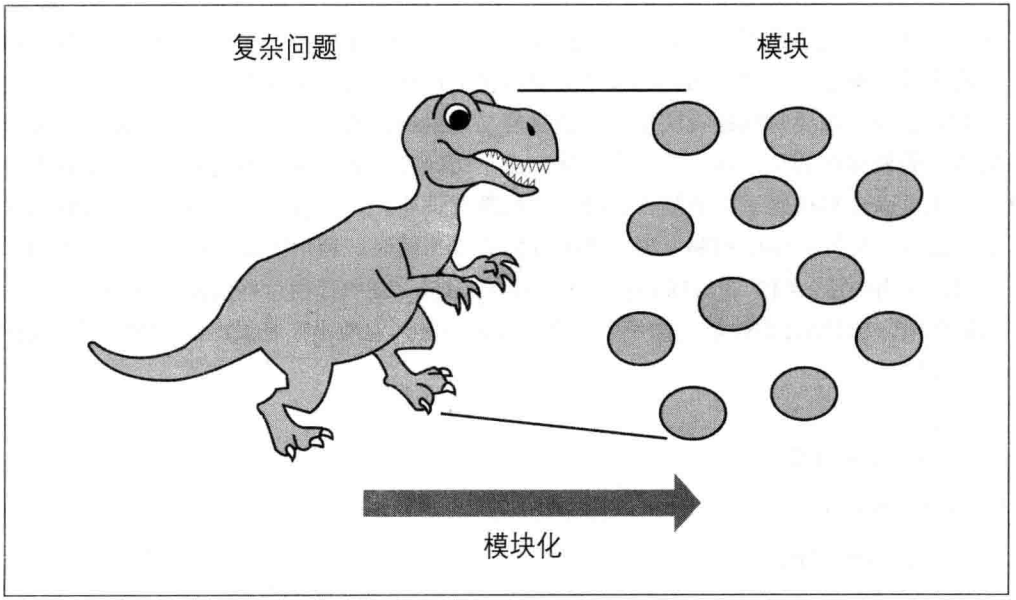


图1-1：即使是最复杂的问题也可以分解为模块

你可能认为模块化看起来并不太难。没错。问题越复杂，模块化就越有意义。所以，OOP编程的出发点绝不是要更复杂，而是要把复杂简单化。即使是最困难的编程问题也可以采用这种“分而治之”的策略加以解决。

1.3 类与对象

一旦对一个问题模块化，又该如何处理这些模块呢？可以看到，分解一个复杂问题会把它变换成多个简单的子问题，不过你需要有一种方法来组织这些模块，让它们相互协作共同处理所要解决的大问题。一种做法是，把一个模块看做是相关函数的一个集合。在编程中，这些模块称为类（class）。类本身由两大部分组成，分别称为属性和方法。属性（properties）是不同类型的数据对象，如数字、字符串、null和布尔类型。通常数据作为抽象数据类型存储，如存储为变量、常量和数组；方法（Methods）是处理这些数据的函数。

1.3.1 单一职责原则

可以把类看作是有共同特征的对象的集合。特征的“共同性”并不是指这些对象是相同的，而是指它们都处理模块（即类）的共同问题。要记住，模块的目的是解决比较复杂的问题的某一方面，这样我们就得到了面向对象编程的首要原则之一：单一职责原则，这表示一个类应当只有一个职责。

并不是说一个类不能有多个职责，不过要记住，我们之所以把一个复杂的问题分解为简单的模块，就是为了把它转化为多个容易解决的问题。通过对类限制单一职责，不仅能提醒自己为什么要对问题模块化，还能更容易地组织模块。下面来看一个实现单一职责的类。假设你在为某个客户建一个网站，由于访问者可能从各种不同的设备浏览这个网站（从桌面计算机到平板电脑再到智能手机都有可能），所以你希望确定究竟是什么类型的设备，另外具体使用哪个浏览器浏览你的Web页面。利用PHP，可以很容易地写一个类，使用内建数组$_SERVER和相关元素HTTP_USER_AGENT提供这些信息。下面给出的代码清单中，TellAll类就是这样一个有单一职责的类，它能提供查看PHP页面的用户代理的有关信息：

```
<?php
//保存为TellAll.php

class TellAll
{
    private $userAgent;

    public function __construct()
      {
        $this->userAgent=$_SERVER['HTTP_USER_AGENT'];
        echo $this->userAgent;
      }
}

$tellAll=new TellAll();

?>
```

如果在一个iMac上通过Safari浏览器加载这个类，会显示以下信息：

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_4) AppleWebKit/ 534.57.2 (KHTML,
like Gecko) Version/5.1.7 Safari/534.57.2
```

而在一个iPad上使用Opera Mini浏览器进行测试时，结果如下：

```
Opera/9.80 (iPad; Opera Mini/7.0.2/28.2051;U;en) Presto/2.8.119 Version/11.10
```

这个类表示一个更复杂操作中的一个模块（这个类只是这个操作的一部分），它符合OOP中“高质量”类的标准，满足单一职责，其职责就是查找关于用户代理的信息。

1.3.2 PHP中的构造函数

PHP类有一个特有的特性：使用`__construct()`语句作为构造函数（constructor function）。大多数计算机语言都用类名作为构造函数名；不过，使用`__construct()`语句有一个好处，可以消除对函数用途的疑问，也就是说，可以清楚地看出它是一个构造函数。

类实例化时，类中的构造函数会自动启动。对于`TellAll`类，不论你是否希望，结果都会立即输出到屏幕上。如果只是用于演示，这是可以的，不过作为一个模块，其他模块可能只是想使用关于设备和/或浏览器的信息，而不是将它显示出来。所以，你会看到，并不是所有类都包含一个构造函数。

1.4 客户类作为请求者

在`TellAll`类中，我在最后加入了一个小触发器来启动这个类。除了`Client`类以外，并不推荐其他类自启动。根据我们使用PHP的经验，大多数情况下都会使用一个`form`标记从HTML启动PHP程序，如下所示：

```
<form action="dataMonster.php" method="post">
```

所以，你应该对这种从外部源启动PHP文件的做法很熟悉了。同样，包含类的PHP文件也应当由其他模块（类）使用，而不是自启动。

随着我们对设计模式的研究逐步深入，你可能会看到经常出现一个名为`Client`的类。这个`Client`在大型项目中承担着不同的角色，不过主要作用是从构成设计模式的类发出请求。这里显示了`Client`类以及`TellAll`类修改后的版本（MobileSniffer）。`Client`使用的这个新类在很多方面与`TellAll`有所不同，对于整个项目来说更有用，还可以在其他项目中重用。尽管`MobileSniffer`类提供同样的用户代理信息，不过利用其属性和方法，这个类可以采用更有用的方式提供这些信息。通过UML（Unified Modeling Language，

统一建模语言）图，可以看到Client会实例化MobileSniffer（虚线）。图1-2显示了这两个类的简单类图。

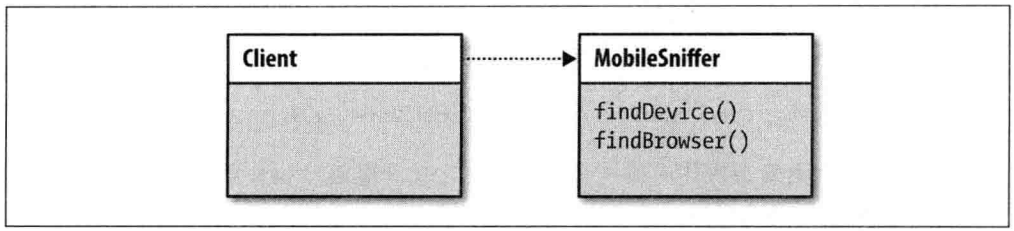


图1-2：Client类实例化MobileSniffer类，而且可以使用它的属性和方法

如果Client类自动实例化，Client使用MobileSniffer信息的选择就会减少。请看下面的代码清单来了解如何创建这个类：

```
<?php
//用户代理作为对象属性
class MobileSniffer
{
    private $userAgent;
    private $device;
    private $browser;
    private $deviceLength;
    private $browserLength;

    public function __construct()
      {
        $this->userAgent=$_SERVER['HTTP_USER_AGENT'];
        $this->userAgent=strtolower($this->userAgent);

        $this->device=array('iphone','ipad','android','silk','blackberry',
          'touch');
        $this->browser= array('firefox','chrome','opera','msie','safari',
        'blackberry','trident');
        $this->deviceLength=count($this->device);
        $this->browserLength=count($this->browser);
      }
    public function findDevice()
    {
        for($uaSniff=0;$uaSniff < $this->deviceLength;$uaSniff ++)
        {
            if(strstr($this->userAgent,$this->device[$uaSniff]))
            {
                return $this->device[$uaSniff];
            }
        }
    }

    public function findBrowser()
    {
        for($uaSniff=0;$uaSniff < $this->browserLength;$uaSniff ++)
```

```
        {
            if(strstr($this->userAgent,$this->browser[$uaSniff]))
            {
                return $this->browser[$uaSniff];
            }
        }
    }
}

?>
```

php.ini文件中嵌入错误报告

我在一所大学工作，在这个环境中，系统管理员大多是学生，知识和能力水平各有不同，往往需要继续磨炼来提高技能。通常，他们都会忘记适当地设置*php.ini*文件来报告错误。因此，我养成了一个习惯，总会在我的代码最前面加上以下代码行：

```
ini_set("display_errors","1");
ERROR_REPORTING(E_ALL);
```

对于某些人来说，增加这些代码行有些烦人，不过我会把它们包含在Client类中，以此作为一个提醒，指示开发应用时错误报告对于反馈重要情况是多么重要。学习OOP和设计模式也在很大程度上依赖于这些反馈。

使用MobileSniffer，Client实例化这个类，其使用方法如以下代码清单所示：

```
<?php
ini_set("display_errors","1");
ERROR_REPORTING(E_ALL);
include_once('MobileSniffer.php');
class Client
{
    private $mobSniff;
    public function __construct()
    {
        $this->mobSniff=new MobileSniffer();
        echo "Device = " . $this->mobSniff->findDevice() . "<br/>";
        echo "Browser = " . $this->mobSniff->findBrowser() . "<br/>";
    }
}

$trigger=new Client();
?>
```

通过使用Client类，提供了一种方法从而可以更有效地使用MobileSniffer类。MobileSniffer没有必要自行启动，通过使用一个return语句，任何调用MobileSniffer的类都能得到这个数据。Client可以采用它希望的任何方式使用MobileSniffer返回的数据。在这里，Client会以指定的格式将这个数据输出到屏幕上，如图1-3所示。

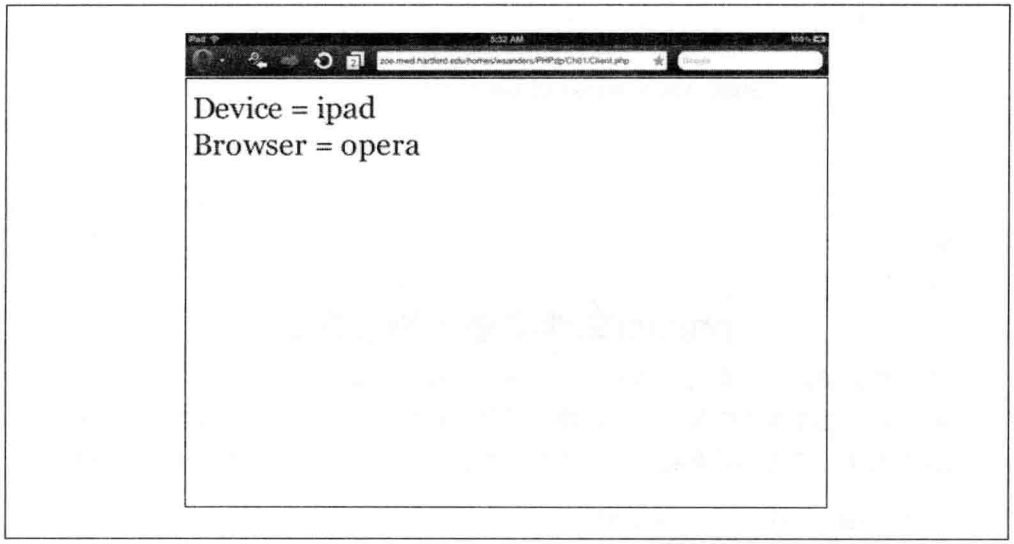


图1-3：客户使用MobileSniffer的数据并发送到屏幕

可以在MobileSniffer类中直接指定数据格式，不过，这样一来，灵活性和可用性都会降低。由于让Client以最通用的方式使用这个数据，这就允许它对数据做任意处理。例如，可以不是格式化数据来提供屏幕输出，完全可以利用这个数据调用一个CSS文件，针对特定的设备和/或浏览器来指定数据的格式。如果在MobileSniffer类中预先指定数据格式，用它标识CSS文件时，就要首先去除之前不需要的格式。要记住，设计模式最重要的特性之一就是对象的重用。

完全掌握移动：难上加难

写这本书时，移动设备的数量和种类还在不断增加，你编写的任何PHP代码都肯定会漏掉一些设备和/或浏览器。甚至仅仅掌握设备还不够，因为有些设备（如iPad和iPad Mini）会有不同的屏幕分辨率，另外还有不同的屏幕大小。可以这么说，如果你计划创建可以在不同设备上浏览的Web页面，系统中就应当有一个模块能够相应更新，而不会破坏你的程序。所以，不论检测和响应多种设备的最新技术是什么，都要做好改变的准备。可以计划从一开始就加入新设备，如Microsoft's Surface，或者可以准备一个模块，它能结合到现有的应用中，而且这个模块中的变化不会破坏整个系统。

现在你可能会想，“我可以写一个更好的算法来筛选设备和浏览器”。这当然可以，实际上，可能你也必须这么做，因为随着新设备和浏览器的引入，对于一个需要用到设备/浏览器信息的程序，必须加入这些新设备和浏览器的信息。不过，如果保持这两个方法

（findDevice()和findBrowser()）的结构不变，你将可以完成所希望的任何修改和改进，而不会导致整个程序受到破坏。你必须考虑到有一个更大、更复杂的程序，而且要考虑到将会做出修改。如果你有过修改大型程序的经历，应该知道一个改变有可能影响到整个程序，并带来灾难性的破坏，而调试也会成为一场噩梦。OOP和设计模式的主要作用之一就是能够改变单个模块而不会破坏整个程序。

1.5 速度如何

几乎每一个程序员都希望程序能以最优的速度运行，为此，我们总在寻求最优的算法。不过，现在需要改变关注点，重视另外一种速度——创建和更新一个程序所花费的时间。如果一个程序周期中某个操作要运行1亿次，这个操作的速度即使只有小小的改变也会对整个程序带来重大影响，不过如果这个操作只使用一次，压缩这个操作的时间几毫秒只是浪费时间。同样，由于增加了几行代码而必须修改整个程序，也是对时间的浪费。

1.5.1 开发和修改速度

假设你有一个合同，要为某个顾客更新和维护一个程序。你们已经协商好将要完成的更新，你希望既能满足客户的要求，同时用最短的时间完成这些更新。例如，假设你的顾客每周会降价销售不同的商品，所以每周都需要做文本和图像更新。一种解决方案是，使用time()函数建立一个每周更新，然后你要做的只是在数据库中增加最新的图像URL和文本。实际上，如果提前备有文本和图像，你甚至可以放心地去度假，让PHP替你完成工作。这样的维护确实很轻松，你还可以同时满足多个顾客的需要。

你有没有考虑过建立一个每次做出修改时都必须重写程序的维护系统？可能不会。这样做不仅很慢，而且成本很高。所以，如果修改速度很重要，你的程序就需要同时考虑操作速度和开发速度。算法处理的是操作速度，而设计模式解决的是开发速度。

1.5.2 团队速度

与团队合作时还存在另外一个速度问题。处理比较大、比较复杂的程序时，团队需要了解并协商一个共同的计划和目标，以便高效地创建和维护大型程序。OOP与设计模式有很多共同的东西，其中就包括它们能提供一种公共的语言，可以加快团队工作。对于理解OOP和设计模式的人来说，“工厂”、“状态机”和“观察者”都是指同一个东西。

最重要的是，设计模式提供了一种编程方法，允许团队中的各个程序员分头工作，最后将工作汇集在一起。就像一个生产汽车的装配线，每个小组装配汽车的一个不同部件。为此，他们需要一种开发模式，而且要理解不同部件之间的关系。通过这种方式，每个

人在完成自己的工作的同时，都能了解其他人的工作并与之协同配合。他们不用知道另一个工人的工作细节，只需要知道他们在为同一个计划努力。

1.6 顺序和过程式编程有什么问题

“东西没损坏的话，那就别去修理它了”，很多人都有这种想法。如果一个方案可行，你可能会想当然地接受这个方案。不过，这种态度绝对是取得进展和提升的障碍。比如说，如果要从一个地方到另一个地方，可以步行过去。不过，要是距离很远，比如从一个国家的最南端到最北端，坐飞机前往会好得多。OOP和设计模式相对于顺序和过程式编程就相当于坐飞机相对于步行。

1.6.1 顺序编程

大多数程序员开始编程时都是逐条地写语句，建立一系列代码行来执行一个程序。例如，下面就是一个很好的PHP顺序程序：

```
<?php

$firstNumber=20;
$secondNumber=40;
$total= $firstNumber + $secondNumber;
echo $total;

?>
```

这里的变量是抽象数据类型，利用算术运算加法（add）运算符（+），将两个变量的值结合为第三个值。echo语句将结合这两个值得到的总和输出到屏幕上。

将两个数相加对于PHP来说是一个很简单的问题，只要是处理简单的问题，就可以使用简单的解决方案。

1.6.2 过程编程

随着程序员开始编写越来越长的程序来完成更复杂的任务，这些顺序代码就会开始相互纠缠，变成所谓的“意大利面式”代码。GOTO语句允许顺序程序员在程序中跳转来完成一个过程，所以这样很容易陷入混乱。

过程编程引入了函数。function（函数）就是一个小对象，可以利用一条语句调用某个操作来完成一个序列。例如，下面是以上顺序程序的过程式版本：

```
<?php

function addEmUp($first,$second)
{
```

```
    $total=$first + $second;
    echo $total;
}
addEmUp(20,40);

?>
```

函数或过程（procedure）允许程序员将代码序列分组为模块，以便在程序中重用。更进一步，通过提供参数，程序员可以为一个函数输入不同的实参，这样就能利用不同的具体值使用这个函数。

类似于OOP，过程编程也使用了模块化和重用。不过，过程编程没有提供类（利用类，编程任务可以打包到对象）。类对象（类实例）可以处理自己的数据结构，这是函数无法单独做到的。因此，如果采用过程编程，完成大型任务往往需要很长的序列。另外，采用过程编程时，团队合作也比较困难，因为不同的团队成员无法像采用OOP那样轻松地处理独立但相互关联的类。

1.6.3 即时回报还是长期回报

就在不久前，我刚刚在博客里发了一篇文章，题为“No Time for OOP and Design Patterns”（OOP和设计模式时不我待）（*http://bit.ly/108IFSF*）。之所以发表这篇文章，是因为很多开发人员都找出各种理由说他们没有时间在他们的工作中结合OOP和设计模式，尽管他们确实有心为之。就是针对这种现象我才有感而发。他们可能解释说某个项目的最后期限已经迫在眉睫，为了按时交工，他们只好对一个现在能用的程序做些修补，这个程序可能使用顺序和过程编程。如果刚好有一个类能满足某个特定目标，他们可能也会包含一两个类，不过仅此而已。

在学习OOP和PHP设计模式时，你需要记住两点，这是Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides最早在《Design Patterns: Elements of Reusable Object-Oriented Software》中指出的：

- 设计面向对象软件很困难。
- 设计可重用面向对象软件更困难。

当然不能把这些说法作为放弃学习OOP和设计模式的理由，而应当由此看出OOP和设计模式为什么这么有意义。知识会增加技能的价值。得到知识越困难，说明它越有价值。

不要期望能轻松、快速地掌握OOP和设计模式，要在你的PHP编程中逐步渗入，一次结合一点，总有一天你会发现它的价值所在。过一段时间后，你会拥有更多技能，有更深入的理解，你可能会遇到某个项目，发现可以在其中重用之前另外一个项目的大部分程序结构。在最近的一个项目中，我决定使用“策略”（Strategy）设计模式。这个项目包

含一个表，其中有105个字段，另外顾客希望得到一组功能。通过采用一种“策略”设计，每个策略都是一个类，分别提供一个算法来处理一个相当基本的PHP问题（比如，对MySQL数据库中的数据完成筛选、更新和删除）。建立这种设计需要花一些时间，不过，一旦建立，就能很容易地对程序做出修改（顾客的需求总在不断改变！）。一段时间之后，顾客又要求我基于一个MySQL数据库结合使用前端和后端PHP完成一个类似的项目。这一次我不再一切都从头开始，而是直接采用“策略”模式，改变算法，几乎瞬时就完成了项目。同样达到了目的，且做法更巧妙，与闷声不响长时间埋头苦战相比，我的顾客得到的软件反而好得多。

有些时候，我们必须放弃原来的老习惯，提升我们的能力。如今，很多程序员都在努力更新自己的技能，以便与时俱进，能够适应移动设备。如果做不到这一点，他们就会失去很多机会——最终他们原来拥有的技能将会过时。我们知道每过一段时间就必须更新我们的技能，利用最新PHP版本的种种好处，结合它提供的诸多新技术，沿着这个行业的发展方向前进。OOP和设计模式所蕴含的概念恰能应对所有这些变化，会让我们成为更棒的程序员，为顾客提供更好的软件。凡事总有第一步。现在花点时间，将来开发项目时就能少花时间，不至于不知所措。另外，你会逐步成长为一个更优秀的程序员，这本身就是学习OOP和设计模式的一个很有说服力的理由。

总之，学习OOP和设计模式可以帮助你把事情做得更好，并享受其中的乐趣。

第2章

OOP基本概念

真理形成的两大要素——事实和抽象。
——雷·德·古尔蒙

不要把我们宝贵的遗产视同儿戏，
请珍视这个有序而自由的伟大国度，
因为，如果我们蹒跚跌倒，
如今的自由和文明将走向毁灭。
——亨利·卡伯特·洛奇

每个人都会得到遗传给他的一切，他继承丰厚的遗产。
我说的并不只是继承的财富，
我是指那些中层和上层人物认为理所当然的东西，
所谓的裙带关系，也就是关系网。
——托妮·莫里森

平常清醒的意识，也就是我们所说的理性思维，
只是一种特殊的意识，在其屏障之外，
很有可能还潜伏着完全不同的其他意识。
——威廉·詹姆斯

2.1 抽象

如果你刚刚接触OOP，别指望马上就能全部弄明白。如果你越来越多地使用OOP，随着这些知识的逐步积累，你将会经常有种恍然大悟的感觉。PHP确实提供了很多重要的OOP特性，而且在PHP语言中采用了自己的方式实现这些特性。如果你熟悉其他OOP语

言，会发现PHP在很多方面与它们完全不同，比如PHP允许常量作为接口的一部分。理解抽象非常重要，不论对于OOP还是对于设计模式来说，抽象都是重要的基石，用得越多，抽象的意义就越突出。

抽象概念对于面向对象编程和设计模式极其重要，所以如果只是泛泛而谈是远远不够的。计算领域中对抽象的一般理解与自然语言中我们每天使用的抽象概念有所不同。例如，“狗”（dog）是指具有类似狗的特征的一种现象。如果有人说“看那只狗”，他们就是在使用狗的抽象（换句话说，就是“狗”这个词）来指示具有狗特性的一种动物的某个具体实例。不过，通过使用“狗”（dog）和“猫”（cat），我们可以做出区分，指示很多不同的狗和猫。但是如果我们用“狗”这个词来指示一只猫，会有人纠正我们：“那不是一只狗，那是一只猫。”所以尽管抽象是一般概念，但是它们有足够的特定性，使我们能够区分特定的实例。

在《Object-Oriented Design with Applications, 3rd Edition》（Addison-Wesley出版）中，Grady Booch（OOP和设计模式的先驱）对抽象（abstraction）给出了一个明确的定义，很好地做了总结：

> 抽象（abstraction）指示一个对象的基本特征，使它与所有其他对象区分开，从而从查看者的角度提供了清晰定义的概念边界。

抽象很重要，因为它允许程序员对对象进行分组和分类。在某种程度上，所有类都是对数据的一组操作的抽象。关于抽象要记住：

> 抽象是用来处理复杂性的主要工具。一个问题越复杂，就越需要抽象来解决。

考虑下面这个有些矛盾的说法：抽象是处理复杂性的具体方法。我们会对现实中的相似性分组（对具体的相似性抽象），从而用更方便管理的方式加以处理。所以，我们不会这么说“我的忠实的、毛茸茸的、摇着尾巴的、爱舔来舔去的、名叫SyntaxError的湿鼻子朋友”，而会说“我的狗”。

2.1.1 抽象类

除了常规的类，PHP还提供了抽象类（abstract classes）。在OOP和设计模式中，抽象类可以为项目提供一种组织机制。抽象类不能实例化，只能由具体类（也就是可以实例化的类）继承抽象类的接口以及它的所有具体属性。

在学习后面的内容之前，需要对“interface”（接口或界面）这个词再做一些说明。我们熟悉的接口或界面包括用户界面、硬件接口和其他涉及计算机硬件和软件的各种关联。还有一种接口用来描述一个对象的“大纲”。首先，来考虑一个很简单的类，它只包含一个方法：

```
<?php
 class OneTrick
 {
     private $storeHere;
     public function trick($whatever)
     {
         $this->storeHere=$whatever;
         return $this->storeHere;
     }
 }

$doIt=new OneTrick();
$dataNow=$doIt->trick("This is perfect.");
echo $dataNow;

?>
```

接口的核心部分由类中操作（函数）定义的所有签名组成。签名（signature）包括一个操作的操作名和参数。实际上，签名还包括返回数据类型，不过由于PHP中数据类型的特殊性，我们将在“类型提示：类似数据类型”一节中再进一步讨论签名的这第3个要素。图2-1 显示了trick()方法的签名。

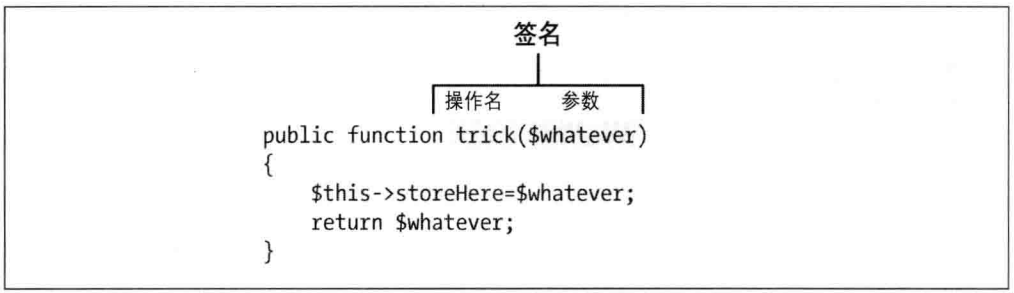


图2-1：PHP中一个操作的签名

得到一个对象的所有签名后，这就是它的接口。例如，OneTrick类有一个操作，它的签名由名字trick()和一个参数$whatever组成。

2.1.2 抽象属性和方法

要把trick()函数写为一个抽象函数，可以只包含接口，而不包括其他任何内容：

```
abstract public function trick($whatever);
```

如果一个类至少有一个抽象方法，它必然是一个抽象类。不过，抽象类也可以有具体方法。例如，下面的类就是一个抽象类，其中包含一个抽象方法：

```
<?php
 abstract class OneTrickAbstract
```

```
{
    public $storeHere;
    abstract public function trick($whatever);
}
?>
```

除了这个抽象方法，可以看到还有一个变量`$storeHere`。作为类中一部分的变量称为属性（property）。有些上下文中，会把属性和方法都简单称为属性，不过，大多数情况下，属性都是指变量和常量（抽象数据），而方法是指函数（对数据的操作）。

PHP没有抽象属性（abstract properties）之说。可以声明一个属性但不指定值，把它当做一个抽象属性，但与抽象方法不同，并不强制使用这些“抽象属性”。

如果在一个抽象类中声明了一个抽象方法，那么继承这个父类的各个子类中都必须实现这个方法。可以这样来考虑，把抽象类中的方法看作是一个合约，强制所有子类继承，所以它们必须遵循同样的标准。对现在来说，先把抽象类看做是一些容器，其中可以填充任何内容，而且可以放在任何交通工具上，如舰船、卡车和火车等。这些容器（如抽象类）将作为一个更大的结构的一部分。

如果一个类继承了某个类，则称为子类（child class），抽象类（或对另一个类有继承关系的任何其他类）则称为父类（parent class）。下面显示了如何实现抽象类`OneTrickAbstract`：

```
<?php
include_once('OneTrickAbstract.php');

class OneTrickConcrete extends OneTrickAbstract
{
    public function trick($whatever)
    {
        $this->storeHere="An abstract property";
        return $whatever . $this->storeHere;
    }
}

$worker=new OneTrickConcrete();
echo $worker->trick("From an abstract origin...");
?>
```

只要包含方法的签名，而且保证提供正确的可见性，就可以采用你希望的任何方式来实现抽象方法。在这里，签名包括方法名“trick”和一个参数`$whatever`，可见性是“公共”（public）。下面都是`trick()`方法的合法实现：

```
public function trick($whatever)
{
    $echo $whatever;
}
```

或：

```
public function trick($whatever)
{
    $half=$whatever/2;
    return $half;
}
```

或：

```
public function trick($whatever)
{
    $this->storehere=25;
    $quarter=$whatever * 100;
    return ($quarter / $this->storehere);
}
```

对于一个抽象方法，只要保证有正确的签名和可见性，就能修改它的具体实现，你可能想知道这有什么意义。在本章后面的“继承”一节中，你会更清楚地了解继承接口为什么很重要。

尽管抽象类通常都包括一些抽象方法，不过除了必须实现的抽象方法外，还可以根据需要增加具体方法和属性。另外，抽象类也可以只包括具体方法。

由于不允许多重继承，PHP 5.4提供了一个称为Trait的结构，这是一种代码重用机制。一个类可以继承另一个类，同时使用一个Trait，这就相当于多重继承。本书的例子中没有使用Trait，这里提到它是考虑到可能会有一些模式例子采用另外一种支持多重继承的语言实现，而你可能希望用PHP来实现这些模式例子，那时就可以使用Trait特性。

2.1.3 接口

OOP和设计模式的另一个重要组成是接口（interface）。与大多数抽象类一样，接口也有抽象方法。不过，不能像在抽象类中那样在接口中包含具体方法或变量。（作为抽象性的例外，接口中可以包含具体常量，不过这是PHP接口的独有特性。）关于接口，需要强调的一点是它们是OOP和设计模式中重要的结构要素。

要创建一个接口，需要使用`interface`语句而不是`class`。一般约定接口总以字母I或i开头；本书中，所有接口名都使用一个大写的I开头，后面是描述接口的一个大写字母。然后是一系列抽象方法，不过并不使用abstract语句。下面是一个简单的接口，其中包括3个方法：

```
<?php
interface IMethodHolder
{
    public function getInfo($info);
```

```
    public function sendInfo($info);
    public function calculate($first,$second);
}
?>
```

要实现一个接口，需要使用implements语句而不是实现抽象类时所用的extend。需要说明，以下代码清单使用了include_once()函数，使实现IMethodHolder接口的类可以使用该接口：

```
<?php
include_once('IMethodHolder.php');

class ImplementAlpha implements IMethodHolder
{
    public function getInfo($info)
    {
        echo "This is NEWS! " . $info . "<br/>";
    }
    public function sendInfo($info)
    {
        return $info;
    }
    public function calculate($first,$second)
    {
     $calulated = $first * $second;
     return $calulated;
    }
    public function useMethods()
    {
        $this->getInfo("The sky is falling...");
        echo $this->sendInfo("Vote for Senator Snort!") . "<br/>";
        echo "You make $" . $this->calculate(20,15) . " in your part-time
         job<br/>";
    }
}

$worker=new ImplementAlpha();
$worker->useMethods();
?>
```

测试这个程序时，应该可以看到以下输出：

```
This is NEWS! The sky is falling...
Vote for Senator Snort!
You make $300 in your part-time job
```

需要说明，除了实现接口中的3个方法外，ImplementAlpha类还包括一个方法useMethods()。只要保证实现了接口中的所有方法，就可以根据需要增加更多其他的方法和属性。

2.1.4 接口和常量

尽管不能在接口中包含变量，但是可以包含常量。要使用常量，需要用到“作用域解析操作符”（scope resolution operator），即双冒号操作符（::）。利用双冒号操作符，允许在类中以及接口实现中存取常量。

下面的代码示例给出了使用接口常量的一般格式：

```
$someVariable= InterfaceName::SOME_CONSTANT;
```

例如，以下接口包含了一个MySQL连接中使用的常量：

```
<?php
interface IConnectInfo
{
    const HOST ="localhost";
    const UNAME ="phpWorker";
    const DBNAME = "dpPatt";
    const PW ="easyWay";
    function testConnection();
}
?>
```

实现这个接口与实现所有其他PHP接口是类似的。可以在实现中使用作用域解析操作符来传递常量值：

```
<?php
include_once('IConnectInfoMethod.php');

class ConSQL implements IConnectInfo
{
    //使用作用域解析操作符传递值
    private $server=IConnectInfo::HOST;
    private $currentDB= IConnectInfo::DBNAME;
    private $user= IConnectInfo::UNAME;
    private $pass= IConnectInfo::PW;

    public function testConnection()
    {
      $hookup=new mysqli($this->server, $this->user, $this->pass,
        $this->currentDB);

        if (mysqli_connect_error())
        {
            die('bad mojo');
        }

        print "You're hooked up Ace! <br />" . $hookup->host_info;

        $hookup->close();
    }
}
$useConstant = new ConSQL();
```

```
$useConstant->testConnection();
?>
```

这里只有一个方法testConnection()，不过如果你愿意，接口中也可以只包含常量而不包括任何方法。通过使用接口名（IConnectInfo）、作用域解析操作符和常量名，可以将这些常量值传递到类属性（示例中的私有变量）。

抽象类和接口：两种“接口”类型

OOP和设计模式中有一些比较容易混淆的概念，例如，抽象类和接口就不太容易区分。要记住，这二者都有接口，也就是方法的一般规则。由类中各方法定义的所有签名的集合就是这个类（或对象）的接口。所以，你可能会经常看到一个模式的接口可能是一个抽象类，也可能是一个接口。实际上它们都表示所有签名的集合。

Gamma等人指出，“对象的接口描述了可以发送到这个对象的所有请求的集合”。这说明，如果Client类知道一个对象的接口，它就会知道能够请求什么以及如何请求。尽管抽象类和接口都有自己的接口，不过抽象类还可以包含具体方法（操作）和属性。

要理解设计模式，在很大程度上这要取决于你对“接口”一般用法的理解：它会作为一个对象（一个类）的签名集合。

2.1.5 类型提示：类似数据类型

OOP和设计模式中抽象有很多重要的结构要素，其中之一就是指定数据类型为接口而不是一个具体实现。这说明，对数据的引用要通过父类完成，这通常是一个接口或抽象类。（在这个上下文中，“接口”一词可以用来表示接口或抽象类。）

提供类型提示的基本格式如下所示：

```
function doWork(TypeHint $someVar)...
```

类型提示必须是类或接口的名字。在设计模式中，更倾向于使用抽象类或接口，因为它不会绑定一个具体实现的类型，而只是限制了结构。下面的例子显示了一个接口，另外提供了这个接口的两个实现，还给出了一个类，它在类型提示中使用了一个接口来建立宽松但明确的绑定。

接口

```
//IProduct.php
<?php
interface IProduct
```

```
{
    function apples();
    function oranges();
}

?>
```

FruitStore实现

```
//FruitStore.php
<?php
include_once('IProduct.php');

class FruitStore implements IProduct
{
    public function apples()
    {
        return "FruitStore sez--We have apples. <br/>";
    }

    public function oranges()
    {
        return "FruitStore sez--We have no citrus fruit.<br/>";
    }
}
?>
```

CitrusStore实现

```
//CitrusStore.php
<?php
include_once('IProduct.php');

class CitrusStore implements IProduct
{
    public function apples()
    {
        return "CitrusStore sez--We do not sell apples. <br/>";
    }

    public function oranges()
    {
        return "CitrusStore sez--We have citrus fruit.<br/>";
    }
}

?>
```

有类型提示的对象

```
//UseProducts.php
<?php
include_once('FruitStore.php');
include_once('CitrusStore.php');

class UseProducts
{
    public function __construct()
```

```
    {
        $appleSauce=new FruitStore();
        $orangeJuice=new CitrusStore();
        $this->doInterface($appleSauce);
        $this->doInterface($orangeJuice);
    }

    //IProduct 在 doInterface()中是类型提示

    function doInterface(IProduct $product)
    {
        echo $product->apples();
        echo $product->oranges();
    }
}

$worker=new UseProducts();
?>
```

测试UseProducts类时，会显示以下输出：

```
FruitStore sez--We have apples.
FruitStore sez--We have no citrus fruit.
CitrusStore sez--We do not sell apples.
CitrusStore sez--We have citrus fruit.
```

你在屏幕上看到的是IProduct接口的不同实现。需要指出最重要的一点：在doInterface()方法中，类型提示（type hint）IProduct能够识别实现IProduct接口的两个类。换句话说，并不是把它们分别识别为一个FruitStore实例和另外一个CitrusStore实例，而会识别它们共同的接口IProduct。

从实际的开发来讲，强制数据类型可以确保倘若给定方法中使用了代码提示，那么其中使用的对象（类）必然有给定的接口。另外，如果把一个接口（可以是一个抽象类或接口）作为代码提示，绑定会更宽松；它会绑定到接口而不是绑定到一个特定的实现。随着程序变得越来越大，只要遵循接口，就可以做任何改变而不会对程序造成破坏。不仅如此，所做的修改也不会与具体实现纠缠不清。

不能使用标量类型（如string或int）作为代码提示，不过可以使用数组、接口（如前例）和类作为代码提示。所以尽管没有另外一些语言那么灵活，但PHP中可以通过类型提示实现类型，这在OOP和设计模式编程中起着重要作用。

2.2 封装

读到关于封装（encapsulation）的内容时，通常都会遇到另外一个词：信息隐藏（information hiding）。这个词并不是不准确，不过一旦你理解了封装，就能更好地理解信息隐藏；解释封装时先从信息隐藏概念谈起并不一定有帮助。实际上，最好先了解什么是划分（compartments）。Grady Booch给出了以下描述：

> 封装就是划分一个抽象的诸多元素的过程，这些元素构成该抽象的结构和行为；封装的作用就是将抽象的契约接口与其实现分离。

一旦把一个复杂的大问题模块化为多个可解决的子问题，就可以利用封装来得到这些较小的抽象，并对它们完成划分（compartmentalizing）。

2.2.1 日常生活中的封装

在日常生活中总会不时地遇到封装。例如，你开车兜风，汽车由很多对象构成，你并不清楚其中大多数对象是如何工作的。点火装置发动引擎，虽然你可能并不了解电池启动发动机、内燃机的细节或者汽车里的电力系统，但你只需要把钥匙插入点火装置，拧动钥匙就能发动汽车。那些细节的复杂性对你是隐藏的，你只会采用某种有限的方式来了解它们。你处理的是一个封装系统，并不需要知道它具体是如何工作的，只需要知道如何访问控制机制，你可以把它想成是汽车的一个用户界面（User Interface，UI）。

你驾驶着汽车沿路行驶时，摇下车窗，假设有一辆车慢慢靠近你，一个人从那辆车的车窗伸出手来，想要抓住你的方向盘来驾驶你的车。可以把这认为是破坏封装（breaking encapsulation）。你不希望这种情况发生，所以你会关上车窗，这样汽车外面的任何人都无法进入到你封闭的车里，不会影响到你开车。

在编程中，正是因为封装，对象才成为一个对象。对象有一些特性（功能），要想访问对象的这些功能，应当由程序的结构来控制，就像你的汽车的结构会控制你对汽车部件的访问一样。类通过限制访问其方法和属性来实现封装。除了通过指定的路径外，你不希望外在的影响控制类的属性，不希望有人非法使用这些属性或者改变属性的状态，就好像你不希望有人抓住你的方向盘一样。所以在介绍封装时如果谈到信息隐藏（information hiding），这是指一个模块的细节可能是隐藏的，只能通过适当的访问渠道来使用这个模块，而不能利用这个模块的细节。

2.2.2 通过可见性保护封装

在PHP中，可见性（visibility）是指对类属性的存取（或访问）。（其他语言可能使用访问（access）一词，另外用存取方法（accessors）表示访问类型。）与其他OOP语言类似，PHP使用3种类型的可见性：private（私有）、protected（保护）和public（公共）。可以采用这些可见性来封装和访问程序。

Private

要封装一个程序元素，最容易的方法就是设置它为私有（private）。这说明，这个属性只能在同一类中访问；它只对同一个类中的元素可见。可以考虑下面这个类：

```
<?php
class PrivateVis
{
    private $money;

    public function __construct()
    {
        $this->money=200;
        $this->secret();
    }

    private function secret()
    {
        echo $this->money;
    }
}
$worker=new PrivateVis();
?>
```

实例化这个类时，构造函数会自动启动类。由于构造函数是PrivateVis类的一部分，所以它能访问所有私有属性和方法。要引用同一个类中的属性或方法，PHP要求对象使用$this语句。另外，私有方法secret()可以访问私有属性（变量）$money，因为它们都在同一个类中。这个类外部的对象只能看到secret()方法的输出结果，而无法改变这个类中方法或属性的状态。

如果通过公共__construct函数（构造函数）实例化类，另一个对象可以访问私有属性（变量或常量）和方法。

Protected

如果属性的可见性为私有，只允许同一个类中的元素访问这个属性，与之不同，如果属性的可见性为保护（protected），则允许这个类以及子类都可以访问该属性。抽象方法和具体方法的可见性都可以设置为保护（protected）。下面的例子显示了如何使用具有保护可见性的抽象类和具体实现：

```
<?php
//ProtectVis.php
abstract class ProtectVis
{
    abstract protected function countMoney();
    protected $wage;

    protected function setHourly($hourly)
    {
        $money=$hourly;
        return $money;
    }
}
?>
```

子类扩展一个包含保护方法的抽象类时，对于抽象方法，在使用之前必须先实现这个抽象方法。对于有保护可见性的具体方法，子类可以直接使用，而无需另行实现：

```
<?php
//ConcreteProtect.php
include_once('ProtectVis.php');

class ConcreteProtect extends ProtectVis
{
    function __construct()
    {
        $this->countMoney();
    }
    protected function countMoney()
    {
        $this->wage="Your hourly wage is $";
        echo $this->wage . $this->setHourly(36);
    }
}
$worker=new ConcreteProtect();
?>
```

注意抽象方法countMoney()使用了继承来的setHourly()方法。继承的所有属性和方法都需要$this语句。第一次访问保护属性时，要使用公共构造函数。

尽管封装性不及私有可见性，但保护可见性能够对程序中更大的结构完成封装。这个更大的结构由父类（不论是抽象类还是具体类）和它的子类构成。

Public

要访问封装的对象，必须提供公共（public）可见性。要想作为一个真正有用的类，其中至少要有一些方法是可见的（即使只是构造函数）。当然，所有构造函数都是公共的。所以，如果你的程序包括一个构造函数，这个类之外的对象就能通过实例化这个类来访问它。例如，下面的代码显示了一个公共方法如何使用类中的私有方法和属性：

```
<?php
//PublicVis.php
class PublicVis
{
    private $password;
    private function openSesame($someData)
    {
        $this->password=$someData;
        if($this->password=="secret")
        {
            echo "You're in!<br/>";
        }
        else
        {
            echo "Release the hounds!<br/>";
        }
```

```
    }
    public function unlock($safe)
    {
        $this->openSesame($safe);
    }

}
$worker=new PublicVis();
$worker->unlock("secret");
$worker->unlock("duh");
?>
```

通常，包含公共方法或属性是为了提供一个途径，从而可以与一个没有构造函数自动启动的对象保持交流。在PublicVis中，$password属性和openSesame()方法都是私有的。不过，unlock()方法是公共的，因为它是PublicVis类的一部分，所以它能访问该类的私有属性和方法。

2.2.3 获取方法和设置方法

为了保持封装，同时提供可访问性，OOP设计建议使用获取方法（getters）和设置方法（setters），也分别称为存取方法（accessors）和修改方法（mutators）。不建议直接访问一个类，通过直接赋值来得到或修改属性值，这些工作完全可以由获取方法/设置方法来完成。一般地，使用获取方法和设置方法必须适度；滥用获取方法和设置方法会破坏封装。下面的例子显示了一个PHP类使用获取方法和设置方法的一种用法：

```
<?php
//GetSet.php
class GetSet
{
    private $dataWarehouse;

    function __construct()
    {
        $this->setter(200);
        $got= $this->getter();
        echo $got;
    }

    private function getter()
    {
        return $this->dataWarehouse;
    }

    private function setter($setValue)
    {
        $this->dataWarehouse=$setValue;
    }
}
$worker=new GetSet();

?>
```

获取方法/设置方法是私有的，所以这个访问是封装的。另外，在这个实现中，设置方法值放在类中，所以它相当于一个相当大的数据容器。

对于面向对象系统中的数据处理，Allen Holub在《Holub on Patterns》（Apress出版）中给出以下建议：

> 不要直接请求完成一个工作所需的信息，而应当请求拥有这个信息的对象为你完成工作。

在GetSet类的例子中，通过实例化类：

```
$worker=new GetSet();
```

就很好地做到了这一点。它没有暴露实现细节。不过单独来看，GetSet类似乎没有太大用处，因为要想指定一个值，唯一的途径就是在类中硬编码设置。

从某种程度上讲，设计模式的目的是建立对象之间的通信链路。很多所谓的OOP并没有正确使用获取方法和设置方法，允许公开访问获取方法和设置方法只会破坏封装。下面的解释引用自David Chelimsky的“Single Responsibility Applied to Methods（对方法应用单一职责）”（*http://tinyurl.com/9256gey*），对面向对象编程和过程式编程做了比较，这段描述有助于澄清OOP中通信的作用：

> 在过程式编程中，过程（process）完全写在一处，即按顺序编写一系列指令。而在面向对象（OO）编程中，过程表示为对象之间的一系列消息。一个对象向另一个对象发送一个消息，就会完成过程的一部分，然后再向另一个对象发送一个新的消息，这又会完成过程的另一部分，如此继续。要修改一个过程，只需要重新组织消息序列，而不是改变一个过程（procedure）。

保持封装同时保持对象（类）之间通信的过程正是设计模式的一个工作。要找出一种方法来建立通信，同时不破坏封装，这可能很困难，所以设计模式就相当于一种“秘诀”，可以指出如何使用可通信的类建立一个程序。

2.3 继承

就其本质来讲，继承（inheritance）是一个简单的概念。一个类如果扩展了另一个类，就会拥有这个类的所有属性和方法。这就允许开发人员创建新类来扩展其他功能。例如，如果你有一个类FurryPets，可以让Dogs和Cats扩展这个类，这样一来，这两个类都会继承FurryPets的属性，另外可以进一步扩展从而区分Dogs和Cats。下面显示了一个简单的PHP例子：

FurryPets.php

```
<?php
//FurryPets.php
class FurryPets
{
    protected $sound;
    protected function fourlegs()
    {
        return "walk on all fours";
    }
    protected function makesSound($petNoise)
    {
        $this->sound=$petNoise;
        return $this->sound;
    }
}
?>
```

Dogs.php

```
<?php
//Dogs.php
include_once('FurryPets.php');
class Dogs extends FurryPets
{
    function __construct()
    {
        echo "Dogs " . $this->fourlegs() . "<br/>";
        echo $this->makesSound("Woof, woof") . "<br/>";
        echo $this->guardsHouse() . "<br/>";
    }

    private function guardsHouse()
    {
        return "Grrrrr" . "<br/>";
    }
}
?>
```

Cats.php

```
<?php
//Cats.php
include_once('FurryPets.php');
class Cats extends FurryPets
{
    function __construct()
    {
        echo "Cats " . $this->fourlegs() . "<br/>";
        echo $this->makesSound("Meow, purrr") . "<br/>";
        echo $this->ownsHouse() . "<br/>";
    }

    private function ownsHouse()
    {
        return "I'll just walk on this keyboard." . "<br/>";
```

```
    }
}

?>
```

Client.php

```
<?php
//Client.php
include_once('Dogs.php');
include_once('Cats.php');
class Client
{
    function __construct()
    {
        $dogs=new Dogs();
        $cats=new Cats();
    }
}
$worker=new Client();
?>
```

Client类发出请求，dogs和cats对象的构造函数生成以下输出：

```
Dogs walk on all fours
Woof, woof
Grrrrr

Cats walk on all fours
Meow, purrr
I'll just walk on this keyboard.
```

对于这两个对象，fourlegs()方法会生成完全相同的输出；makesSound()的输出则取决于为参数提供的实参；最后一点，这两个类都有自己其特定的方法——guardsHouse()和ownsHouse()。

继承有助于为程序中包含的不同类建立一种结构。不过，为了保证类之间的松绑定，通常会继承抽象类，而且是浅继承，只有一层子类。如果程序通过深层次继承绑定到具体类，即使对父类做简单的修改，也会对子类带来严重的破坏。

2.4 多态

基本说来，多态（polymorphism）就是指多种形态，不过只有在OOP上下文中这么讲才有意义，否则这种解释没有多大帮助。多态的真正价值在于，可以调用有相同接口的对象来完成不同的工作。在一个大型复杂结构中（一个很大的程序），增加和修改可能会对程序带来巨大影响，除非有一个公共的接口（父类或接口）。例如，下面的程序中有两个类，它们有一个共同的接口。接口的实现有很大不同：

```
<?php
//Poly.php
```

```
interface ISpeed
{
    function fast();
    function cruise();
    function slow();
}

class Jet implements ISpeed
{
    function slow()
    {
        return 120;
    }

    function cruise()
    {
        return 1200;
    }

    function fast()
    {
        return 1500;
    }
}

class Car implements ISpeed
{
    function slow()
    {
        $carSlow=15;
        return $carSlow;
    }

    function cruise()
    {
        $carCruise=65;
        return $carCruise;
    }

    function fast()
    {
        $carZoom=110;
        return $carZoom;
    }
}

$f22=new Jet();
$jetSlow=$f22->slow();
$jetCruise=$f22->cruise();
$jetFast=$f22->fast();
echo "<br/>My jet can take off at $jetSlow mph and cruises at $jetCruise mph.
However, I can crank it up to $jetFast mph if I'm in a hurry.<br/>";

$ford=new Car();
$fordSlow=$ford->slow();
$fordCruise=$ford->cruise();
$fordFast=$ford->fast();
```

```
echo "<br/>My car pokes along at $fordSlow mph in a school zone and cruises at
$fordCruise mph on the highway. However, I can crank it up to $fordFast mph
if I'm in a hurry.<br/>";
?>
```

Jet和Car类对ISpeed接口的实现有很大不同。不过，基于这样一个公共接口，在一个给定的程序结构中做出修改或增补时，可以放心地请求或使用接口方法，而不必担心程序会崩溃。

2.4.1 一个名字，多个实现

要了解多态，最简单的办法就是查看接口的不同实现，如前面的*Poly.php*例子。Jet类中实现的各个方法只返回一个基本类型值，而Car类首先将一个基本类型值传入一个变量，再返回这个变量的值。二者都使用了相同的接口。

对于PHP类和接口的多态使用，有一点要注意， PHP函数的签名中并不包括返回类型。例如，C#中的方法签名会同时包含返回数据的类型以及是否希望有返回值。例如，请看下面这个C#代码段，这里声明了一个接口：

```
//C#接口
interface IFace
{
    string stringMethod();
    int numMethod(int intProp);
    void noReturnMethod();
}
```

C#语言不需要为方法加入关键字function，签名只包括返回数据类型、方法名和参数（如果有的话）。所以第一个方法（stringMethod()）希望返回一个字符串，而且这个方法的所有实现都必须包括return关键字。类似地，第二个方法（numMethod()）希望返回一个整数，而且必须有一个整数参数。同样地，这个方法的所有实现都需要有一个return语句。不过，第三个方法（noReturnMethod()）返回类型为void，说明这个方法的实现中不必有return语句。实际上，对于一个返回数据类型为void的方法，如果具体实现中包含有return，会导致一个错误并失败。

在PHP中，方法签名中不包括任何返回信息，如果实现很混乱，可能返回不同的数据类型，而且其中一些有返回值而另外一些没有返回值，这就会很麻烦。多态的价值在于，即使形态可能不同，但是总能依靠接口指导你的实现。接口中的信息越多，如指示了希望哪种类型的行为（有没有返回数据）和哪种数据类型，就能越容易地让多种不同的形态很好地协作。在*Poly.php*例子中，所有方法都包括一个整数和一个return语句。所以尽管PHP没有强类型的签名，不过完全可以在你的计划和实现中加以强制。可以在接口中使用注释语句来做到。例如：

```
//PHP接口
interface IFace
{
//@return字符串
function stringMethod();

//@return整数并使用整数属性
    function numMethod($intProp);

    //不使用return
    function noReturnMethod();
}
```

如果接口（包括抽象类）中有一个扩展的虚签名（使用注释），就能提供和利用强类型语言的优点。

2.4.2 设计模式中的内建多态性

在本书及其他地方看到不同的设计模式时，应该注意到，这些设计模式中已经充分内建有多态性。例如，“策略”设计模式包含一个Strategy接口，由不同的算法提供具体的实现。Strategy声明一个公共的接口，不同的算法由这个公共接口派生具体实现。例如，在一个PHP“策略”设计中，可以为数据库提供一个接口，对应不同的数据库动作（如更新、删除或插入）分别有不同的实现。对于一个利用MySQL表完成数据处理的程序，通过维护一个公共接口，可以为具体的更新、选择或搜索动作增加新算法，而不用担心破坏这个程序。

2.5 慢慢来

这一章集中介绍了这本书中将要出现的最重要的OOP概念。不过，读到后面各章了解到各个设计模式时，你会发现这些概念还会反复出现，可能是设计模式的某个方面，或者出现在一个设计模式的PHP实现中。所以要记住，随着对设计模式的继续深入，现在还不明白的内容会越来越清楚。第3章将介绍基于OOP概念和思想的一些关键的设计模式概念。

使用即时贴

我的所有OOP和设计模式书上，都有不同颜色的即时贴。即时贴也叫做“索引标签”、“页面标记”或者其他名字的标签，不过不论你们那里的文具店怎么称呼它，建议你一定要买一些来标记可能需要反复阅读的一些重点内容。如果你使用平板电脑（如iPad或Kindle）来阅读，可以使用电子书签。不同的阅读器（和软件）页面标记的功能各有不同。

要适度地使用记号笔。我的一些书上所有内容都加了记号以示强调，不过这与什么都不强调是一样的。另外，在我的平板电脑上，可以利用选择来突出显示一些文档，不过同样地，使用突出显示也要注意，不要过分滥用。

第3章

基本设计模式概念

大型组织往往很松散。不，更确切地讲，几乎可以认为它根本没有组织。
——吉尔伯特·K·切斯特顿

如果你唯一的工具是一把锤子，你很可能会把一切问题都看成钉子。
——亚伯拉罕·马斯洛

因为事物有它自己的方式，它们不会墨守成规。
——贝尔托·布莱希特

3.1 MVC实现编程松耦合和重新聚焦

追溯到20世纪70年代，那时用户界面（UI）还很原始，程序员决定将图形用户界面（graphical user interface，GUI）中需要的关键元素分开。分别为各个部分指定特定的任务，而且每个部分都与其他部分通信。这些部分分组为两大类：域对象（domain objects）和表现对象（presentation objects）。域对象用于建模，表示用户所感知的真实世界，表现对象则是屏幕上看到的内容。这些部分划分为一个模型（model）、一个视图（view）和一个控制器（controller），这就引入了模型－视图－控制器（Model-View-Control，MVC）设计模式。

MVC中的域元素就是模型。模型负责数据部分，也称为企业或应用逻辑。以一个自动调温器为例，自动调温器有一个值表示温度，这可以是具体的环境温度，也可以是关闭或打开（*Off*或*On*）加热器所需的温度。因此自动调温器中可以有一组数据，分别设置到一组变量中：

```
$a = 当前温度
$b = 关闭加热器的温度
$c = 打开加热器的温度
```

在某个给定时刻，具体的值可能是：

```
$a=65;
$b=67;
$c=64;
```

这些值由控制器设置以及一个读取环境温度的温度计生成。但这些值是如何生成的以及在哪里生成并不重要，这部分工作由模型负责。

MVC的表现部分有两个元素：视图和控制器。在前面所述的自动调温器中，视图就是一个窗口，为访问者显示温度和设置。模型提供环境温度，控制器提供打开和关闭加热器所需的温度，并发送到视图来显示。控制器就是调整打开/关闭加热器温度值的设备（不过，视图会显示控制器）。图3-1显示了一个自动调温器中的MVC组成。

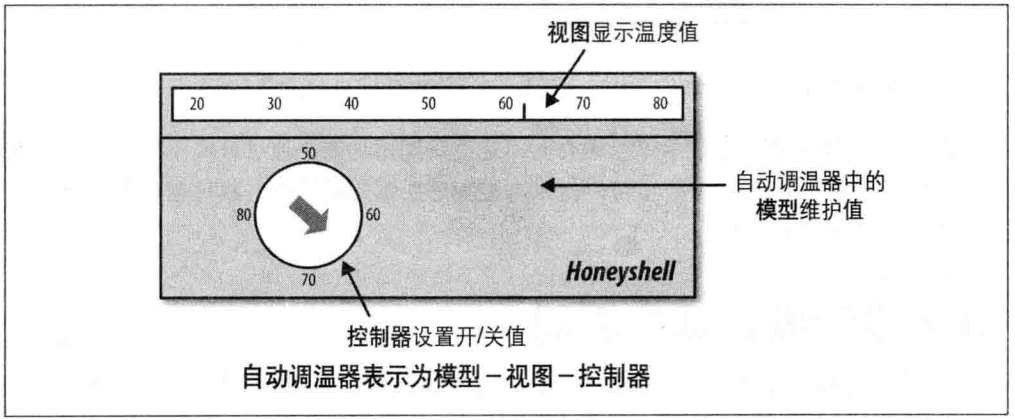


图3-1：自动调温器中的MVC组成

墙上挂着的自动调温器与计算机程序中表示的虚拟调温器是不同的，在计算机程序中，视图要与控制器通信来确定如何处理用户输入。记住，在一个计算机显示中，用户界面是视图的一部分，用户可以看到它并做出改变；然后视图可以将用户做出的这些改变传递给控制器，控制器再把这些信息发送给模型。图3-2以类图方式显示了这个对象。

MVC有很多不同的形式，也有很多不同的实现。由于MVC的简单性和可用性，这种模式得到了广泛使用，其中也不乏滥用。不过，关键是MVC代表着一种结构，可以提供各个部分之间的松耦合。例如，假设你想把视图从一个模拟视图变成数字视图，这很容易做到，因为模型和控制器都是自包含的实体，它们实际上并不关心视图显示什么。类似地，控制器可以把温度设置从华氏度改为摄氏度，只要为模型发送了信息，它并不关心使用哪一种温度格式。

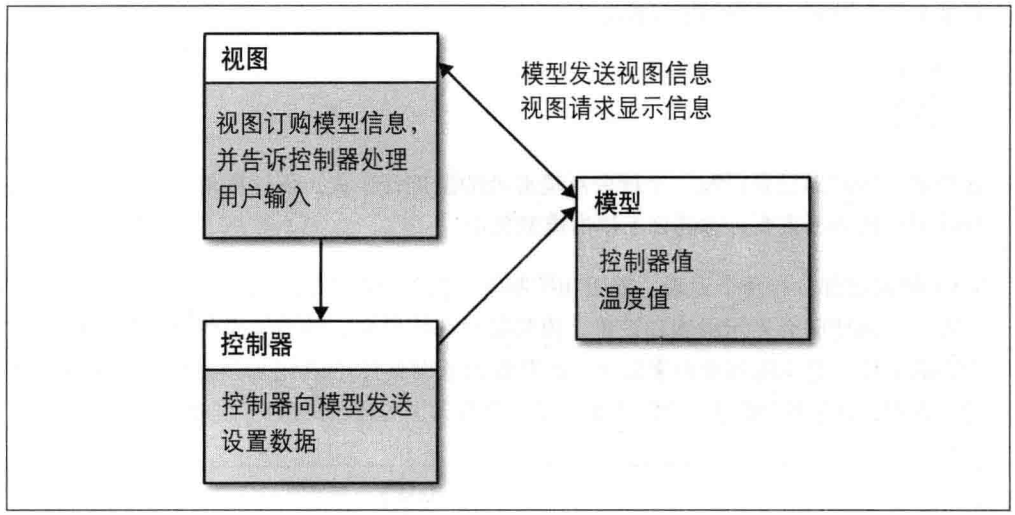


图3-2：MVC类图

MVC的重要性还在于它展示了松耦合而不是直接调用功能。通过分离不同的元素（或参与者）来完成一个任务，MVC可以提供大型程序所需的灵活性。程序越庞大，就越需要MVC提供的模块灵活性。

3.2 设计模式基本原则

MVC是设计模式开发中很重要的一点，不过这还只是一个起点。对于编程中MVC的使用、误用和滥用，已经有大量相关的文档做过介绍。如果过度使用，就像是只用一个工具（比如说一个锤子）来盖整个房子，要用它完成所有工作，包括锯木材、测量和打钻等任务。

在里程碑性的《Design Patterns: Elements of Reusable Object-Oriented Software》一书中（由Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides撰写，他们通常被称作“四人帮”，或简称为GoF），就是从MVC开始讨论设计模式中的各种工具。他们将MVC的特点描述为：通过在视图和模型之间建立一个订购/通知协议，实现视图和模型的解耦合。不过，GoF继续指出，尽管很多基本设计模式与MVC有关，但还有大量常用模式（可以用来帮助计算机程序员完成诸多工作）与MVC完全无关。所以，我们并不是讨论以某种方式与MVC关联的设计模式，而更需要分析与MVC无关的设计模式。我们要研究一般意义上的设计模式原则，再分别讨论各个设计模式。

为什么叫“四人帮”？

我并不喜欢讲行话或者那些小范围的流行语，把《Design Patterns: Elements of Reusable Object-Oriented Software》的作者Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides简称为“四人帮”或GoF可能有些过于简单。不过，就像4位作者完成的这本书一样，列出整个书名太长，而简称为《Design Patterns》（设计模式）又太短，所以GoF这个名字倒也巧妙。含义很明确，即使你恰好知道这个说法的渊源，也要知道这里的“四人帮”与其他人完全没有关联，这只是一种方便的叫法，OOP和设计模式程序员都知道这是什么，仅此而已。

3.2.1 第一个设计模式原则

第一个原则对于PHP来说有些困难；按接口而不是按实现来编程。以最简单的方式来讲，按接口而不是按实现编程是指，要将变量设置为一个抽象类或接口数据类型的实例，而不是一个具体实现的实例。如果按接口编程，可以将设计与实现解耦合，这样有很多好处，例如，可以很容易地将一个复杂的数据库实现替换为一个简单得多的模拟实现，以便于测试。有些语言的变量声明中包含数据类型，在这些语言中，只需要指定接口作为数据类型，而不是变量实例化的具体类。例如，在一个强类型语言中，可以有下面的声明：

*Interface **IAlpha***

接口名

*Class **AlphaA***

实现**IAlpha**；AlphaA的数据类型为**IAlpha**

*Variable **useAlpha***

声明类型为**IAlpha**（例如，IAlpha useAlpha）

useAlpha

实例化新的**AlphaA()**

但在PHP中，不能声明一个变量的数据类型为抽象父类（抽象类或接口），因为如果不实例化一个类实例，就不能声明数据类型。另外，不能实例化一个抽象类的对象。通过使用以下格式实例化一个具体对象，变量可以“得到”一个数据类型：

```
$useAlpha = new AlphaA();
```

名为`$useAlpha`的变量是类`AlphaA`的一个实例，并认为这是它的数据类型。所以`$useAlpha`的数据类型就是`AlphaA`，这是一个具体类的具体实现。不过，尽管这个实例

的数据类型是一个具体类，实际上，这个具体类的父类也同时可以作为它的数据类型。很有必要再重复强调一下这个概念：

> 一个对象实例的数据类型不仅是它实例化的对象类型，该对象的父类也将作为它的数据类型。

如果没有强类型机制，怎么做到呢？在PHP中，可以利用代码提示保证按接口编程。

3.2.2 代码提示中使用接口数据类型

第2章“类型提示：类似数据类型”一节中给出了一个例子，展示了如何对接口IProduct使用代码提示。有两个不同的类FruitStore和CitrusStore，它们都实现了相同的接口。如果一个方法要求有一个代码提示参数IProduct，通过使用代码提示，这两个类都可以作为这个方法的实参。这个代码提示指定实参必须是IProduct数据类型。仔细检查这个代码段，可以了解如何按接口编写操作。图3-3显示了按接口编写PHP操作的第一步：将对象实例化为接口实现的实例（完整的程序参见“类型提示：类似数据类型”一节）。

IProduct 具体实现的实例

```
public function __construct()
{
    $appleSauce=new FruitStore();
    $orangeJuice=new CitrusStore();
    $this->doInterface($appleSauce);
    $this->doInterface($orangeJuice);
}
```

图3-3：实例化IProduct的具体实现

图3-3所示的实例化与按接口编程的原则正好背道而驰：变量被实例化为具体实现的实例，而不是其共同接口的实例。由于PHP是弱数据类型语言，还有一点你看不出来，实际上接口也会作为具体实现的数据类型。

使用类型提示时，程序员必须提供满足指定类型提示的对象。如果这个类型提示是一个接口，程序就会正常工作，就好像你指定了接口类型一样。图3-4显示了这个过程的细节。

通过查看图3-3和图3-4，可以看到这里最关键的方法是doInterface()，它包含一个类型提示。两个具体实现的具体实例（FruitStore或CitrusStore）都可以作为doInterface()方法的实参，其输出都是可预测的。只要接口的任何其他实现与IProduct接口一致（包

括返回数据类型），无论程序变得多复杂都没有关系。只要保证接口，你可以任意做出修改和增补，它们不会破坏程序的其他部分。

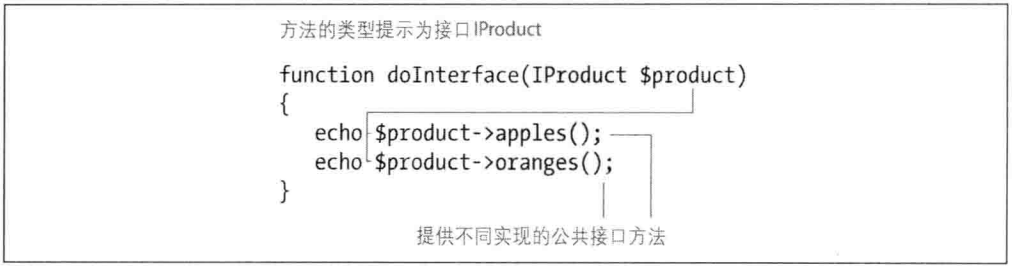


图3-4：方法类型提示为接口IProduct

3.2.3 抽象类及其接口

为了便于解释按接口而不是按实现编程的原则，这里的例子中使用的接口是用关键字`interface`创建的。不过，在继续学习后面的内容之前，要了解接口这个概念是指方法及其签名，而不是关键字`interface`。每个类都有一个接口，由其方法签名构成。由于大多数设计模式很少由一个具体类扩展，所以你要了解扩展一个抽象类就类似于实现一个接口。

在下面的例子中，两个简单的实现扩展了一个简单的抽象类。接下来给出一个使用了类型提示的Client类，由此展示可以使用抽象类实现按接口编程。代码中的注释有助于强制返回期望的返回值。

首先，抽象类`IAbstract`有一个保护属性（`$valueNow`），另外有两个保护的抽象方法（`giveCost`和`giveCity`），还有一个公共函数`displayShow`（这不是一个抽象方法）：

```
<?php
abstract class IAbstract
{
    //对所有实现都可用的属性
    protected $valueNow;

    /*所有实现都必须包含以下两个方法: */
    //必须返回十进制值
    abstract protected function giveCost();
    //必须返回字符串值
    abstract protected function giveCity();

    //这个具体函数对
    //所有类实现都可用
    //而不覆盖内容

    public function displayShow()
    {
```

```
        $stringCost =$this->giveCost();
        $stringCost = (string)$stringCost;
        $allTogether=("Cost: $" . $stringCost . " for " . $this->giveCity());
        return $allTogether;
    }
}
?>
```

接下来给出这个抽象类的两个不同扩展，它们分别提供了抽象方法的不同实现：

```
//NorthRegion.php
<?php
include_once('IAbstract.php');

class NorthRegion extends IAbstract
{
    //必须返回十进制值
    protected function giveCost()
    {
        return 210.54;
    }
    //必须返回字符串值
    protected function giveCity()
    {
        return "Moose Breath";
    }
}
?>

//WestRegion.php
<?php
include_once('IAbstract.php');

class WestRegion extends IAbstract
{
    //必须返回十进制值
    protected function giveCost()
    {
        $solarSavings=2;
        $this->valueNow=210.54/$solarSavings;
        return $this->valueNow;
    }
    //必须返回字符串值
    protected function giveCity()
    {
        return "Rattlesnake Gulch";
    }
}
?>
```

利用一个抽象类的两个不同实现，可以看到，这里所说的“按接口编程”实际上是指类的接口，而不是使用关键字*interface*定义的接口结构。最后，Client类建立了一个包含代码提示的方法，指定这个抽象类作为接口：

```
<?php
include_once('NorthRegion.php');
include_once('WestRegion.php');

class Client
{
    public function __construct()
    {
        $north=new NorthRegion();
        $west= new WestRegion();
        $this->showInterface($north);
        $this->showInterface($west);
    }

    private function showInterface(IAbstract $region)
    {
        echo $region->displayShow() . "<br/>";
    }
}
$worker=new Client();
?>
```

输出如下：

```
Cost: $210.54 for Moose Breath
Cost: $105.27 for Rattlesnake Gulch
```

对于不同的地区，结果值是不同的，因为两个具体类NorthRegion和WestRegion分别采用了不同方式来实现这个抽象方法。如果使用了一个不正确的数据类型（例如，一个字符串），你会看到以下错误消息：

```
Catchable fatal error: Argument 1 passed to Client::showInterface()
must be an instance of IAbstract, string given, called in /Library/
```

所以，就其自身而言，类型提示可以帮助你尽可能遵守第一个设计模式原则，即按接口而不是按实现来编程。

如果想了解这种编程风格的好处，可以增加IAbstract抽象类的SouthRegion和EastRegion实现。记住为giveCost()方法使用一个小数值，对giveCity()则要使用一个字符串值。保持二者的其他接口一致，把它们增加到Client类中。可以看到，只要保持接口一致，完成增补和修改很容易。

3.2.4 第二个设计模式原则

有些OOP程序员认为对象重用就等同于继承。一个类可以有大量属性和方法，扩展这个类就可以重用所有那些对象元素，而不用重新编写代码。可以扩展类，再增加必要的新属性和方法，就一切OK了。不过最后对于紧密绑定的对象，一味扩展可能会带来问

题。这个问题属于过度继承，这也是第二个原则的前提：应当优先选择对象组合而不是类继承。

那么对象组合与类继承有什么区别呢？这个说法并不是要完全消除继承。实际上，这表示开发程序时如果有机会使用组合，就应当优先使用组合而不是继承。这样一来，子类就不会因为继承到大量不用的属性和方法而变得过度膨胀。

3.2.5 使用客户的基本组合

要了解使用继承和组合之间的区别，可以通过一个简单的例子来说明，这里将使用一个父类和一个子类展示继承，另外使用两个单独的类展示组合。在查看代码之前，图3-5显示了使用继承和组合的差别。

使用继承时，客户可以完成一个实例化来实现数学和文本功能。利用组合，客户要用两个不同的实例来访问这两个类的功能。设计模式中的组合通常是指模式中一个参与者内部的组合。

首先来看使用继承的代码。第一个类（父类）是一个简单的类，包含两个方法来完成加法和除法计算：

```
<?php
//DoMath.php
class DoMath
{
    private $sum;
    private $quotient;

    public function simpleAdd($first,$second)
    {
        $this->sum=($first + $second);
        return $this->sum;
    }

    public function simpleDivide($dividend, $divisor)
    {
        $this->quotient=($dividend/$divisor);
        return $this->quotient;
    }
}
?>
```

第二个类用于增加文本功能。一个方法是将数字转换为字符串，另一个方法是建立一个格式化输出。通过扩展，这个类继承了DoMath的所有功能：

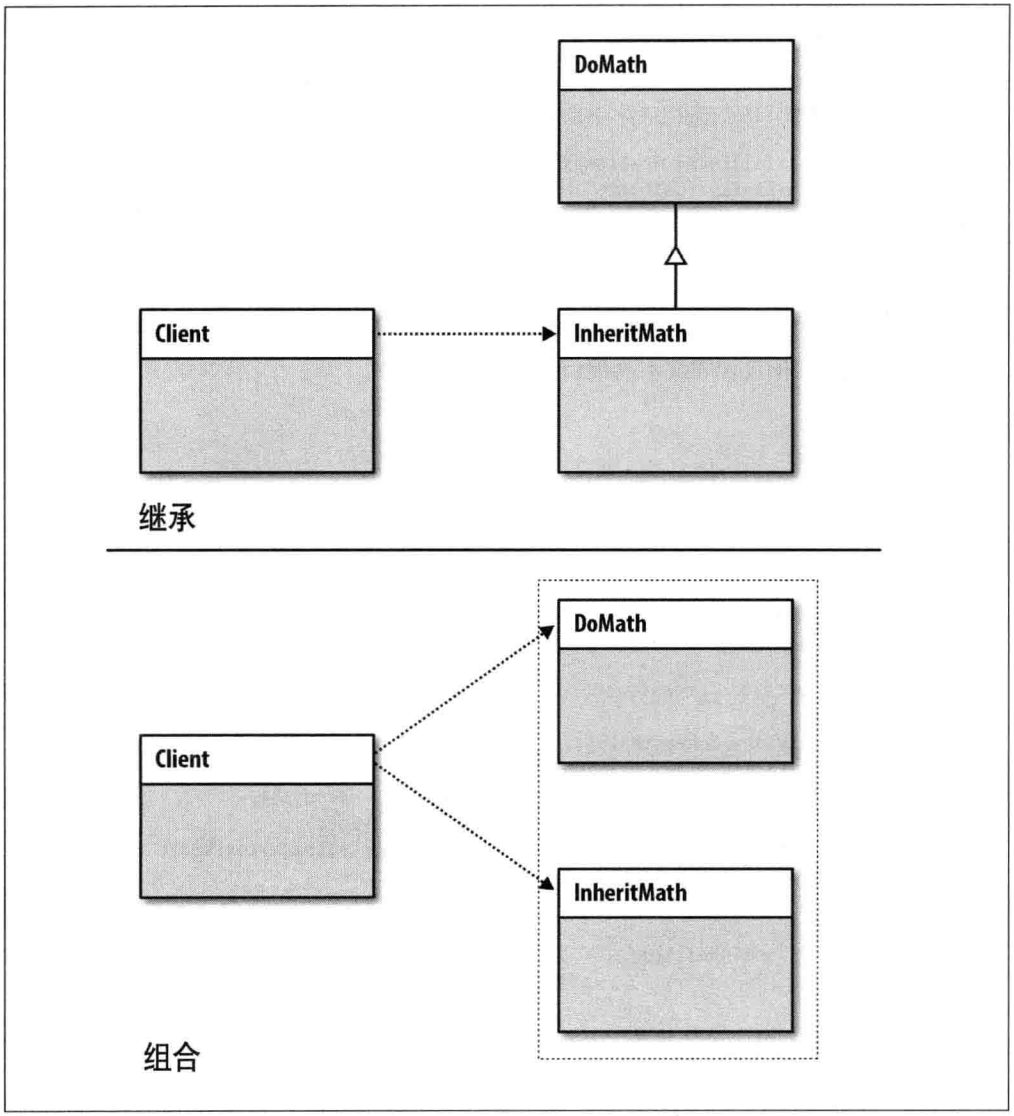


图3-5：继承和组合

```
<?php
//InheritMath.php
include_once('DoMath.php');
class InheritMath extends DoMath
{
    private $textOut;
    private $fullFace;

    public function numToText($num)
    {
```

```
        $this->textOut=(string)$num;
        return $this->textOut;
    }

    public function addFace($face, $msg)
    {
        $this->fullFace= "<strong>" . $face . "</strong>: " . $msg;
        return $this->fullFace;
    }
}
?>
```

Client类实例化InheritMath类，不仅能够使用由DoMath类继承的所有功能，还能使用InheritMath类自身包含的文本功能：

```
<?php
//ClientInherit
include_once('InheritMath.php');
class ClientInherit
{
    private $added;
    private $divided;
    private $textNum;
    private $output;

    public function __construct()
    {
        $family=new InheritMath();
        $this->added=$family->simpleAdd(40,60);
        $this->divided=$family->simpleDivide($this->added,25);
        $this->textNum=$family->numToText($this->divided);
        $this->output=$family->addFace("Your results",$this->textNum);
        echo $this->output;
    }
}
$worker=new ClientInherit();
?>
```

输出是一个格式化的计算结果值：

```
Your results: 4
```

这个输出使用了4个不同方法，其中两个是从父类继承的方法。

再来看组合，Client类使用两个不同的类，分别包含两个方法。DoMath类等同于继承例子中的父类，所以首先来分析DoText类：

```
<?php
//DoText.php
class DoText
{
    private $textOut;
    private $fullFace;
```

```
    public function numToText($num)
    {
        $this->textOut=(string)$num;
        return $this->textOut;
    }

    public function addFace($face, $msg)
    {
        $this->fullFace= "<strong>" . $face . "</strong>: " . $msg;
        return $this->fullFace;
    }
}
?>
```

DoText类看起来与InheritMath类很相似，实际上也确实如此。不过，它没有继承DoMath类。

在这个例子中，通过组合，客户使用了这两个不同的类，结果是一样的。不过，客户必须实例化两个对象而不是一个。除此之外，组合例子中使用的客户与继承例子中使用的客户非常相似：

```
<?php
//ClientCompose.php
include_once('DoMath.php');
include_once('DoText.php');

class ClientCompose
{
    private $added;
    private $divided;
    private $textNum;
    private $output;

    public function __construct()
    {
        $useMath=new DoMath();
        $useText=new DoText();
        $this->added=$useMath->simpleAdd(40,60);
        $this->divided=$useMath->simpleDivide($this->added,25);
        $this->textNum=$useText->numToText($this->divided);
        $this->output=$useText->addFace("Your results",$this->textNum);
        echo $this->output;
    }
}
$worker=new ClientCompose();
?>
```

结果完全相同，不过Client类必须包含多个类。看起来好像继承更胜一筹，不过在较大的程序中，组合可以避免维护多个继承层次上的各个子类，而且还可以避免可能导致的错误。例如，父类的一个改变会逐级向下传递到子类实现，这可能会影响子类使用的某个算法。

3.2.6 委托：IS-A和HAS-A的差别

在设计模式领域中，你会看到有些类的构造中使用了其他类。一个类将一个任务传递给另一个类时，这就是委托（delegation）。正是这一点使组合拥有了强大的能力。

使用继承时，每一个子类是另一个类或多个类的一部分（*IS-A*关系）；而采用组合，对象可以使用一个不同的类或一组类完成一系列任务（*USE-A*关系），这并不是说不能使用继承。实际上，大多数设计模式同时包含有继承和组合。不过，要避免使用继承形成一长串子类、孙子类、曾孙子类等，设计模式方法建议使用浅继承，另外尽量使用多个类的功能。这种方法有助于避免紧密绑定，另外倘若一个具体类有子类，修改这个类设计可能导致程序崩溃，而浅继承可以避免这种情况。

3.3 设计模式作为备忘录

要确定何时使用委托以及如何使用，应当包含多少继承，以及如何确保OOP编程中的重用时，可以把设计模式看作是一个备忘录。可以迅速查看一般设计，这些设计往往使用类图展示需要在哪里采用继承和组合。通过使用统一建模语言（Unified Modeling Language，UML），可以逐步了解如何查看一个类图，并很快地找出其中不同的部分［称为参与者（participants）］。第4章会介绍结合使用UML和PHP设计模式的有关细节。

设计模式的组织

这本书将沿用“四人帮”在《设计模式》一书中的设计模式组织。总的说来，设计模式是按作用和范围来组织的。设计模式的作用可以分为3大类：

- 创建型
- 结构型
- 行为型

这种分类也反映了一般所认为的模式所要完成的目标。按范围划分可以分为两大类：

- 类
- 对象

这一节将简要介绍这些类别划分，并解释这样划分对于选择和理解设计模式的好处。

创建型模式

顾名思义，创建型模式就是用来创建对象的模式。更确切地讲，这些模式是对实例化过

程的抽象。如果程序越来越依赖组合，就会减少对硬编码实例化的依赖，而更多地依赖于一组灵活的行为，这些行为可以组织到一个更为复杂的集合中。创建型模式提供了一些方法来封装系统使用的具体类的有关知识，还可以隐藏实例创建和组合的相关信息。

结构型模式

这些模式所关心的是组合结构应当保证结构化。结构型类模式（structural class patterns）采用继承来组合接口或实现。结构型对象模式（structural object patterns）则描述了组合对象来建立新功能的方法。了解结构型模式对于理解和使用相互关联的类（作为设计模式中的参与者）很有帮助。

行为型模式

到目前为止，绝大多数模式都是行为型对象。这些模式的核心是算法和对象之间职责的分配。Gamma等人指出，这些设计模式描述的不只是对象或类的模式，它们还描述了类和对象之间的通信模式。

类模式

在两类范围中，第一类范围是类（class）。这些类模式的重点在于类及其子类之间的关系。在GoF的《设计模式》一书介绍的24个设计模式中，类范围中只包括4种模式。这一点并不奇怪，因为类模式中的关系是通过继承建立的，而且GoF更多地强调组合而不是继承。类模式是静态的，因此在编译时已经固定。

对象模式

尽管大多数设计模式都属于对象范围（object scope），不过与类范围中的那些模式一样，很多模式也会使用继承。对象设计模式与类模式的区别在于，对象模式强调的是可以在运行时改变的对象，因此这些模式更具动态性。

3.4 选择设计模式

学习设计模式时，很重要的一部分是要学习如何选择最合适的模式。要记住，设计模式并不是模板，它们只是一些一般策略，可以用来处理面向对象编程中经常出现的一般问题。这本书会分别介绍按作用划分的3大类别以及按范围划分的两大类别中的一个设计模式。另外，还有3章（第12章到第14章）专门讨论可以利用设计模式结合使用PHP和MySQL的常用方法。最后这3章中讨论的3个模式都属于对象模式，按作用来讲是行为型模式，这也是GoF介绍最多的一类模式。

3.4.1 是什么导致了重新设计

选择设计模式时，首先要问的一个问题是："是什么导致了重新设计？"例如，假设你建立了一个在线咨询台，用户发出请求，然后数据库做出响应。不过，可以想见，咨询请求和咨询响应的类型都会改变。如果你的程序依赖于某些特定的操作，一旦有改变就可能带来问题。所以，不要建立硬编码的操作来满足请求，职责链（Chain of Responsibility）设计模式可以提供一种更好的方法，它允许沿着一个链传递请求，这样就可以有多个对象都有机会处理这个请求。

这本书会指出不同的设计模式可以解决哪些类型的问题。这些章节和模式将提供一个上下文，不仅帮助你了解设计模式的一般原则，还可以了解特定模式处理的具体问题。

3.4.2 什么会变化

选择设计模式时还要考虑一个问题：设计中什么会变化。不是查看重新设计的原因，这种方法是要查看你希望哪些方面可以改变而无需重新设计。可以看到，现在的重点将转换为封装那些变化的概念。表3-1给出了这本书将要介绍的9个设计模式，按模式的作用、范围和可能变化的方面做了划分。

表3-1：设计模式作用、范围和变化

作用	范围	模式名	可能变化的方面
创建型	类	工厂方法	实例化对象的子类
	对象	原型	实例化对象的类
结构型	类	适配器*	对象的接口
	对象	适配器* 装饰器	对象职责而不派生子类
行为型	类	模板方法	算法中的步骤
	对象	状态	对象状态
	对象	策略	算法
	对象	职责链	可以满足请求的对象
	对象	观察者	依赖于其他对象的对象数；当前可以有多少个依赖对象

注意： *适配器模式有两种配置：一个类适配器模式和一个对象适配器模式。将在第7章分别介绍。

每个模式都有一个一般用法。变化的部分必须在具体的上下文中理解，后面介绍各个模式时，就能更清楚地了解这些变化。

解决特殊问题：设计模式是标准答案吗？

因为设计模式是编程中反复出现的常见问题的解决方案，有些人会误以为它们就是一些毫无变化的“标准答案”，要严格按它要求的去编程。这就像是在说，PHP中的循环结构是一种编程约束。类似于设计模式，引入循环就是为了处理反复出现的编程问题；对于循环来说，所处理的就是重复问题。用一个循环处理100次迭代肯定要强过编写100行顺序代码。类似地，与每次做出改变时都重写整个程序相比，使用松耦合的设计模式肯定更胜一筹。所以，你会发现循环结构有很多不同的用法和实现，类似地，你也会发现很多不同的设计模式实现。

3.4.3 设计模式与框架有什么区别

与框架相比，设计模式是体系结构中更小的元素，也更为抽象。另外，设计模式没有框架那么特定。因此，设计模式更可重用，也比框架更灵活。

框架的优点与模板有些类似：它们更有指示性，可以更清楚地指示所解决问题的结构。为了提供这种易用性，它们不得不放弃了体系结构的灵活性。如果使用框架，构建应用会快得多，但是所构建的应用会受到框架本身的约束。框架可以包含面向对象结构，通常框架是分层的，每一层处理更大设计中的一个方面。框架的一些特性在设计模式中也有体现，不过，设计模式没有框架那么特定和具体，也没有那么庞大。

第4章

结合使用设计模式和UML

与狂热想象的美妙梦境相比，现实显得一文不名；
如此一来，现实终遭厌弃。
——爱米尔·杜尔凯姆

历史上以及当前的社会结构中，社会形态的最初阶段都是如此：
——个紧紧闭合的小圆圈，
独立于周边那些千奇百怪甚至在某种程度上敌对的圆圈。
——格奥尔格·齐美尔

人们只有将个人的生活与社会的历史这两者放在一起认识，
才能真正理解它们。
——C·怀特·米尔斯

4.1 为什么是统一建模语言 (UML)

在《设计模式》（Addison-Wesley出版）一书中，Gamma、Helm、Johnson和Vlissides采用了一种UML，它与20世纪90年代的UML标准稍有不同，与当前的UML 2.0标准有也略有差异。不过，这些差别很小，我们将采用GoF版本的UML，这样读者们就能将这本书中使用PHP设计模式与“四人帮”原来介绍的设计模式进行比较。所以不论人们认为哪个版本的UML最适用于编程，你会发现这本书中的UML与Gamma等人原先采用的UML始终保持一致。

如果你还不熟悉UML，可能需要耐心一些。使用设计模式时，只有在实现设计和具体使用PHP的情况下才会更清楚地了解这些设计模式。由于PHP使用的是最弱的数据类型机制，直接从一个UML实现设计相当困难，除非你能对数据类型做出调整，而且对模式本身非常了解。

4.2 类图

前面各章显示了一些简单的类图，这一节将介绍类图的更多细节，并介绍如何结合设计模式使用类图。一般而言，设计模式类图显示了参与者（类和接口）之间的关系和通信。例如，图4-1显示了一个典型的设计模式类图。在这个特定的类图中，可以看到5个参与者：

- **客户（Client，隐含）**
- 创建者（Creator）
- **具体创建者（ConcreteCreator）**
- 产品（Product）
- **具体产品（ConcreteProduct）**

各个参与者分别是一个类或接口（抽象类或接口）。接口名用斜体，具体类用roman加粗字体。注意`Client`显示在一个浅灰色的方框中。这表示它在这个模式中是隐含的，对于工厂方法（Factory Method）模式，`Client`类并不是这个模式中的一部分。

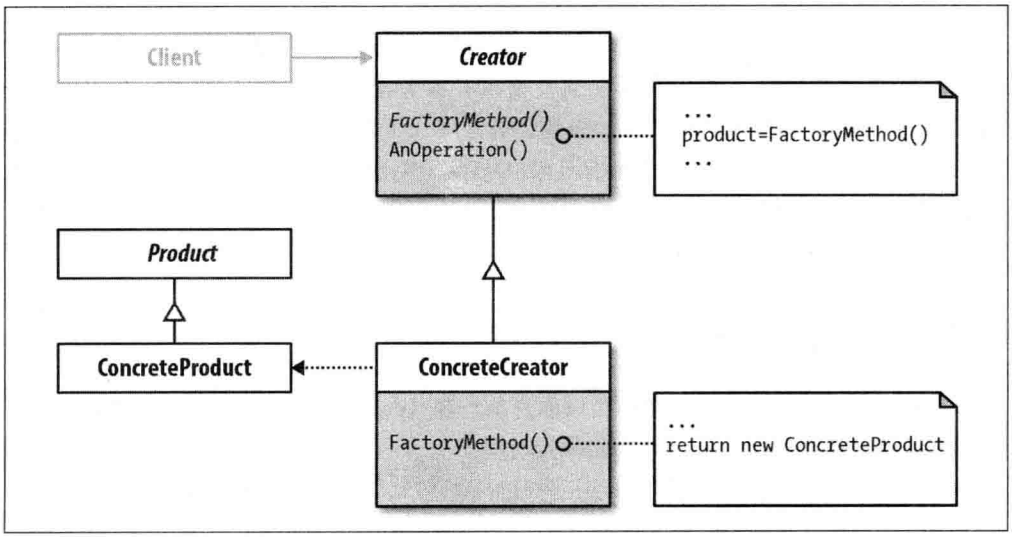


图4-1：类图（工厂方法）

`Client`类通常会作为设计模式中的一部分；有时它是隐含的，有时则根本不在类图中出现。客户从主程序发出请求，箭头有助于显示`Client`与程序主要部分之间的各种不同关系。通常，程序员会用“main”来表示客户，不过在设计模式中，“main”这个词有些误导。程序的主要部分是各个相互关联的参与者，而不是客户。

图4-2显示了同一个模式，只是去掉了伪代码注释，以便更清楚地强调模式中的基本元素。

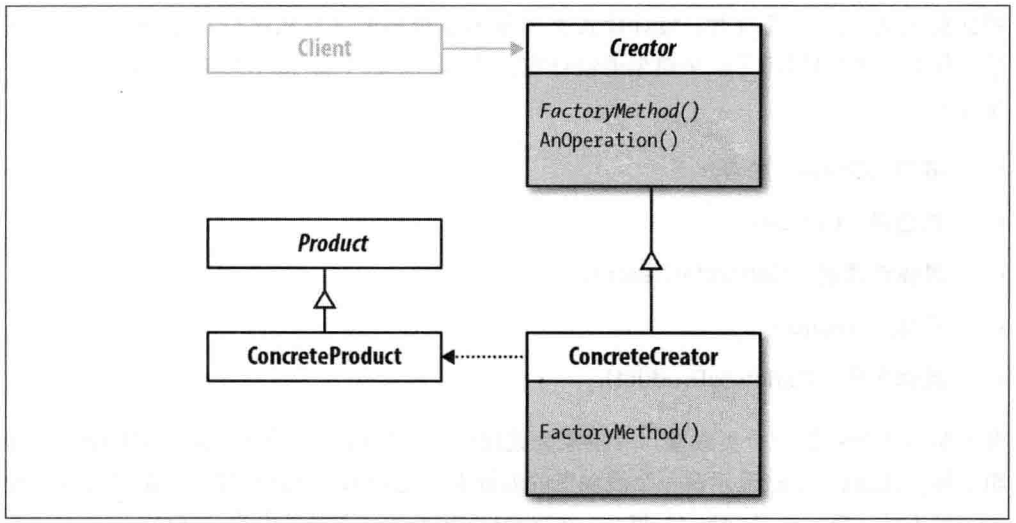


图4-2：类图（工厂方法模式，不过去除了注释）

工厂方法（Factory Method）模式通过一个工厂（Creator）实例化对象，从而将实例化过程与请求者分离。客户希望得到（请求）某种类型的产品，不过并不是直接从ConcreteProduct实例化这个产品，而是要求客户（如类图中所示）通过Creator做出请求。如果没有类图，就不能清楚地指示客户从哪里做出请求，另外所有参与者的关系也不是很明了。类图提供了一个设计模式视图，从中很容易看出UML描述的关系。这些关系都有具体的特质，下面几节分别介绍模式参与者和连接所使用的各种符号。

4.3 参与者符号

“四人帮”将构成设计模式的类和接口称为参与者（participants）。参与者名用粗体显示。接口（接口和抽象类）显示为斜体。有些参与者名下面有一条线，并附带一组关键方法。抽象方法使用斜体，具体方法用常规字体显示。图4-3更详细地显示了Creator接口。

在接口的具体实现中，抽象方法也是具体的。图4-4显示了Creator接口的实现。

注意这个接口中有一个名为AnOperation()的方法，具体类中并没有包含这个方法。这是因为，这个方法已经实现，可以由ConcreteCreator直接继承。不过，具体类中必须实现FactoryMethod()，因为这是父类中的一个抽象方法。

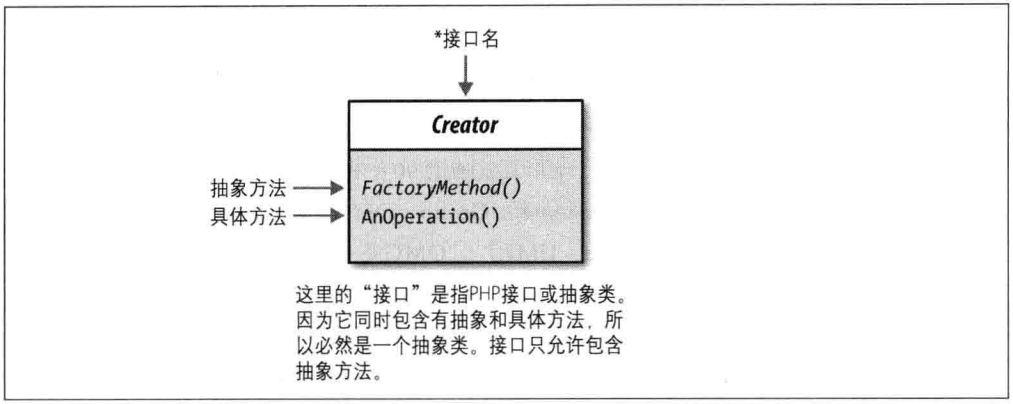


图4-3：接口参与者中的元素

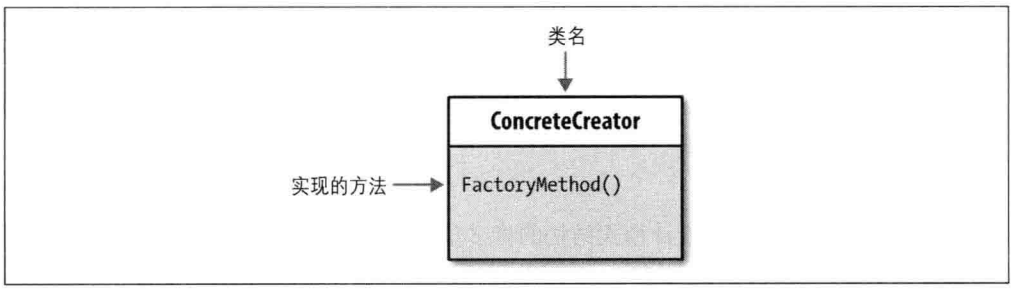


图4-4：由抽象类或接口实现的具体类和方法

关于参与者，最后需要说明的是伪代码注释（pseudocode annotations）。图4-5更清楚地显示了从工厂方法模式去除的注释。

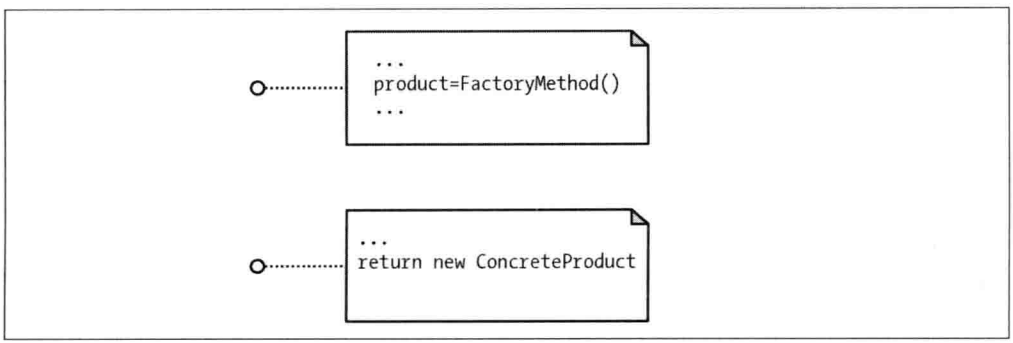


图4-5：伪代码注释

伪代码注释提供了有关设计模式结构的进一步信息。设计注释的目的是为了让开发人员更好地了解参与者代码将要做什么。

什么是最好的UML？

关于OOP和设计模式，一般来讲，最好的资源就是源代码。前面已经指出，本书中的UML与“四人帮”使用的UML一致，不过，最早的UML出现在Booch、Jacobson和Rumbaugh的作品中。他们是20世纪90年代中期最早的UML开发人员，并向对象管理组织（Object Management Group，OMG）提交了他们的统一建模语言（Unified Modeling Language，UML），OMG是计算机行业的一个标准组织。UML在1997年作为行业标准得到采纳，后来随着模型的改变和完善陆续有所修订。要了解UML，有一个很好的资源可以参考，这就是《Object-Oriented Analysis and Design with Applications, 3rd Edition》（Addison-Wesley出版）一书中的第5章。这一章长达100页，详细介绍了OMG UML的一个最新版本。Booch、Jacobson和Rumbaugh编写的2005版《Unified Modeling Language User Guide, 2nd Edition》用大约500页提供了UML的诸多细节。所以，如果你想深入了解UML，这些资源一定不会让你失望。

4.4 关系说明

伪代码注释可以提供有关设计模式结构的更多信息。设计注释的目的是为了让开发人员更好地了解参与者代码将要做什么（见图4-6）。

在UML的不同解释中，这些关联及其含义可能有所变化，这个UML图有些简化，描述了《设计模式》一书中第1章介绍的各种关系，如果需要，可以参考那一章来了解更多细节。如果想使用这本书中未涵盖的其他PHP设计模式，就要熟悉GoF使用的UML。

4.4.1 相识关系

设计模式中最基本、可能也是最常见的关系就是相识关系（acquaintance）。相识关系是指一个参与者包含另一个参与者的引用。这种关系用一个简单箭头指示，如`Client`与`ISubject`之间以及`Proxy`和`RealSubject`之间的简单箭头，如图4-7中代理（Proxy）设计模式所示。

下面给出这样一个关联关系的具体例子，请看`Proxy`类的代码清单，可以看到PHP中的关联是怎样的：

```
<?php
class Proxy extends ISubject
{
    private $realSubject;
```

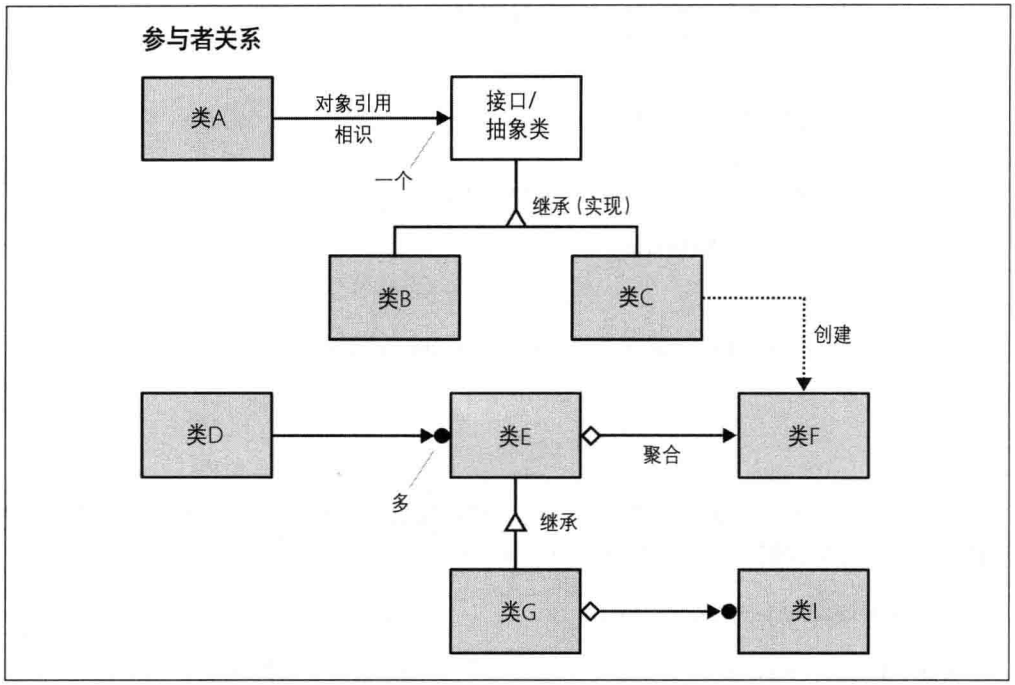


图4-6：类关系

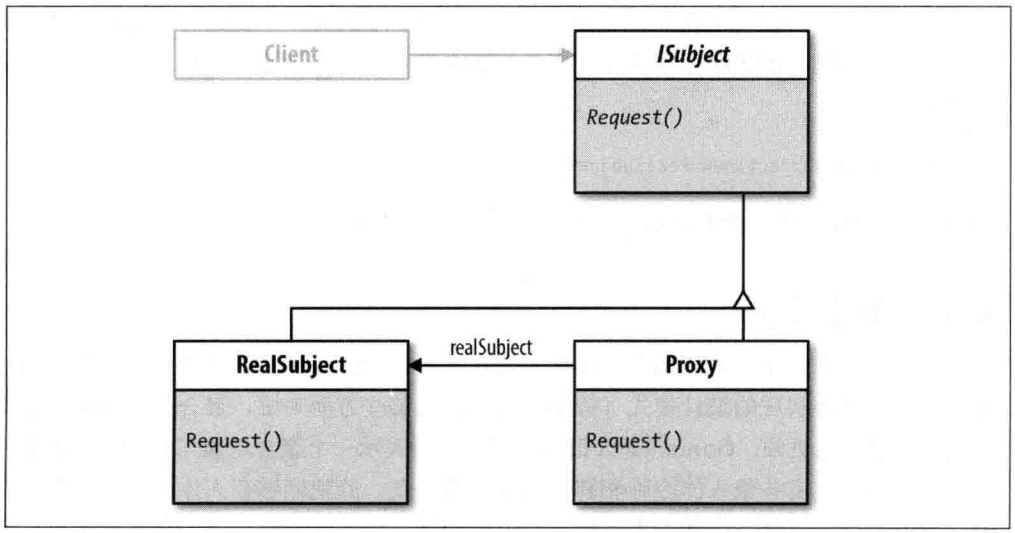


图4-7：Client与ISubject以及Proxy与RealSubject之间的关联关系

```
protected function request()
{
```

```
            $this->realSubject=new RealSubject();
            $this->realSubject->request();
        }

        public function login($un,$pw)
        {
            //计算口令等
            if($un==="Professional" && $pw==="acp")
            {
                $this->request();
            }
            else
            {
                print "Cry 'Havoc,' and let slip the dogs of war!";
            }
        }
    }
    ?>
```

一般说来，一个类包含另一个类的引用时，只需要有一个声明。在强类型语言（如C#）中，声明要包含数据类型，“关联”引用如下所示：

```
private RealSubject realSubject;
```

在前面的Proxy代码清单中可以看到，还有一个同名的私有变量的声明：

```
private $realSubject;
```

问题在于，变量可以实例化为任何数据类型。所以，要包含一个引用，必须扩展该变量的一个具体实例，或者必须在声明该变量的类中实例化。在这里，我们完成了这个变量的实例化：

```
$this->realSubject=new RealSubject();
```

所以，现在Proxy类包含RealSubject类的一个引用，也就与它建立了一个相识关系。

4.4.2 聚合关系

“四人帮”指出，单个代码集都无法展示聚合关系，所以要记住，这里使用的例子只是显示了使用聚合实现的设计模式（策略模式）。从某些方面来讲，聚合关系与相识关系类似，不过关系更强。Gamma等人指出，聚合关系表示一个聚合对象与它的所有者有相同的生命期。这有些像人的心脏和肺。只要心脏供血，肺就能够在人体内输送氧化的血液。它们都是可以独立运转的生理器官，但是如果一个停止工作，另一个也会停止，无法继续正常工作。

策略（Strategy）设计模式中，Context类和Strategy接口之间就有一种聚合关系，如图4-8所示。

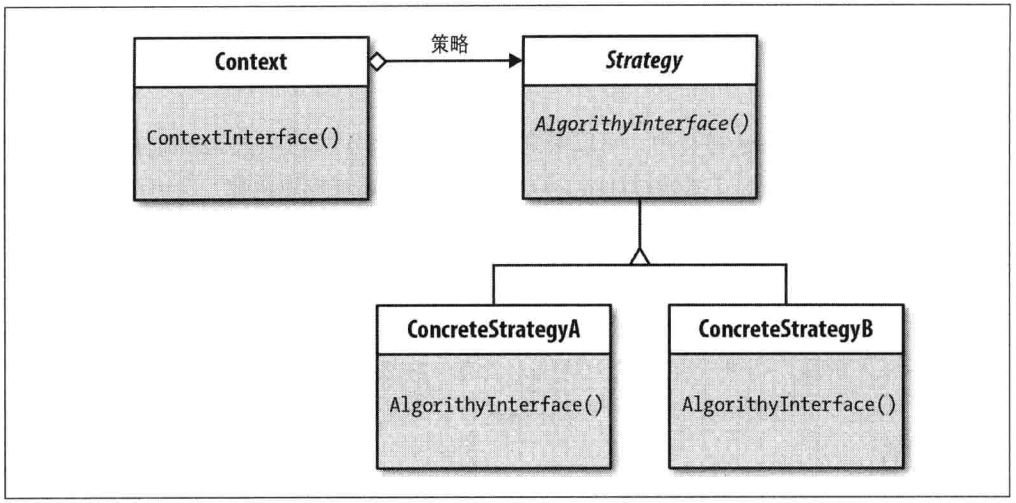


图4-8：策略模式类图，ConText类与Strategy接口之间有聚合关系

下面来看ConText类的PHP代码：

```
<?php

class Context
{
    private $strategy;

    public function __construct(IStrategy $strategy)
    {
        $this->strategy = $strategy;
    }

    public function algorithm($elements)
    {
        $this->strategy->algorithm($elements);
    }
}

?>
```

代码提示引用指向IStrategy接口。（接口名IStrategy取自这个设计模式中的参与者Strategy）这样一来，可以看到ConText类通过代码提示包含了Strategy接口的一个引用，而不必在Context参与者（类）中实例化一个IStrategy实现。实例化ConText类的一个实例时，Client类必须提供一个具体的策略实现。例如，发出请求的客户要包含一个ConcreteStrategyA：

```
<?php
interface IStrategy
{
```

```
    public function algorithm($elements);
}

?>
```

在这个简单的IStrategy接口中，可以看到，它只包含一个方法algorithm()。不过，看起来ConText类已经实现了algorithm()方法。实际上，通过构造函数，ConText类已经配置有IStrategy的一个具体实现。这个关系的重要特点是要指出两个对象如何构成一个聚合，另外它们要有相同的生命期。

在所有关系中，聚合关系最难解释，也最难理解，因为它有很多变种。现在我们并不打算进一步深入解释，否则可能会过于简化，会有一些表述不到位的地方，后面我们还会在这本书中逐步学习使用聚合的一些设计模式，在这个过程中，将进一步解释聚合关系的特定用法。

4.4.3 继承和实现关系

“四人帮”对继承和实现使用了相同的记法：三角形。图4-9分别显示了一个有子类的抽象类和一个有子类的接口。

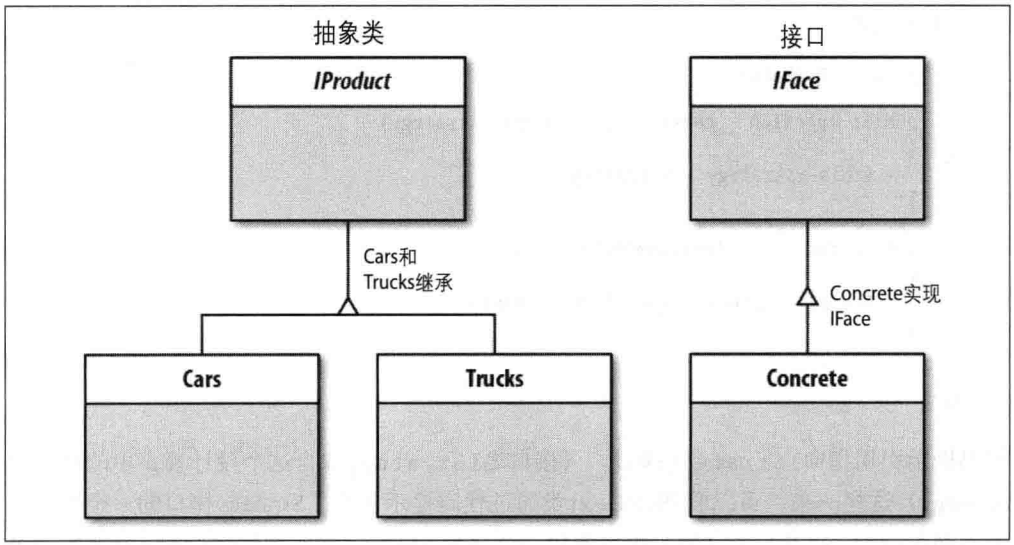


图4-9：继承和实现都使用同样的三角形记法

单个继承或实现使用一个三角形（线上三角形），而指示多重继承或实现时，会把三角形水平置于具体类的公共连接线上。

你可能想知道GoF为什么选择对继承和实现都使用三角形符号。实际上，这是因为几乎

所有使用类继承的设计模式都是从抽象类继承。抽象类意味着需要实现抽象方法，所以不论采用接口还是抽象类，实现都会作为这个关系的一部分。在《设计模式》一书中，GoF就曾多次交替使用抽象类和接口（把它们都简称为接口），因为它们都包含子类使用的接口。类似地，在很多设计模式中，究竟使用抽象类还是接口并不重要。在设计模式的一个实现中，程序员可能使用了抽象类，而在同一个设计模式的另一个实现中，可能会使用接口。

术语的多重含义

在读“四人帮”的书时，人们遇到的困难之一就是书中某些术语经常交替使用。刚开始你可能还会努力区分，但过不了多久，你可能就想给作者写封信，告诉他们把术语统一一下，不过就像是编程中的多态一样，看起来写作中也有“多态”。下面给出一些例子，并指出如何处理这些术语：

接口

GoF将抽象类和接口都称为接口（interfaces）。他们（以及我）所指的是抽象类或接口中未实现的方法和属性的集合。接口最重要的特性是能够在模式中提供松绑定。

实现

与接口（interface）的使用类似，你会发现“实现”（implements）一词也会用在不同的上下文中。从某种程度上讲，抽象类和接口都必须实现。抽象类中的所有抽象方法需要由某个子类实现，所以尽管子类只是继承一个抽象类的接口，但它至少还必须实现这个抽象类中的一些抽象方法。所以，对于一个实现某个抽象类或接口的具体类，涉及抽象类时，这个具体类的一个一般引用会同时意味着继承和实现。只是“实现”比“继承和实现”说起来要更容易一些。另外，这种说法也算给出一个回旋空间，表示子类必须确定一个具体实现。

操作

在这本书中，包括类图中，你会经常遇到“操作”（operation）这个词。一般来讲，它指示完成某个工作的一段程序。操作可能与函数（function）或方法（method）是同义词，也可能指示类之间发生的多个不同事件。例如，操作可能表示一段代码，可以从设计模式中的一个具体类实例化一个实例。你可能还会遇到这样一种操作：包装ClassB来实例化ClassC的一个实例。在这种情况下，操作（operation）可能只是函数的一部分，或者只是构造函数中的一条语句。

灵活思考

在这本书和“四人帮”的书中，你要用一种不同的方式考虑编程。与设计算法相比，思路要更开阔，特别是要考虑对象之间的关系。这可能涉及算法方面的一些考虑，不过基本说来，考虑到问题所在上下文中的相互关系，将问题放在一个更大的上下文中来考虑，这样会更为合理。好的算法固然很重要，不过算法要由方法或对象体现。重要的是对象之间的相互影响，以及如何使用和重用这些对象。如果使用正确，面向对象组件就能处理新的算法。只要对象与其他对象之间建立了关系，算法改变并不会影响整个程序。正是出于这个原因，你需要考虑关系（relations），而不仅仅是处理当前问题的一系列代码。

4.4.4 创建关系

在一个设计模式中，一个对象创建另一个对象的实例时，采用的记法是一条虚线，并有一个箭头指向实例化的对象。在使用工厂的模式中（如工厂方法设计模式（见第5章）或抽象工厂设计模式），你会看到这种关系，不过另外一些模式中也可以看到这种关系。图4-10显示了工厂方法模式的一个类图，这里`ConcreteCreator`使用一个`factoryMethod()`方法来创建`ConProductA`和`ConProductB`的实例。`Client`包含`Creator`和`Product`接口的引用，所以它能指定具体的产品请求。

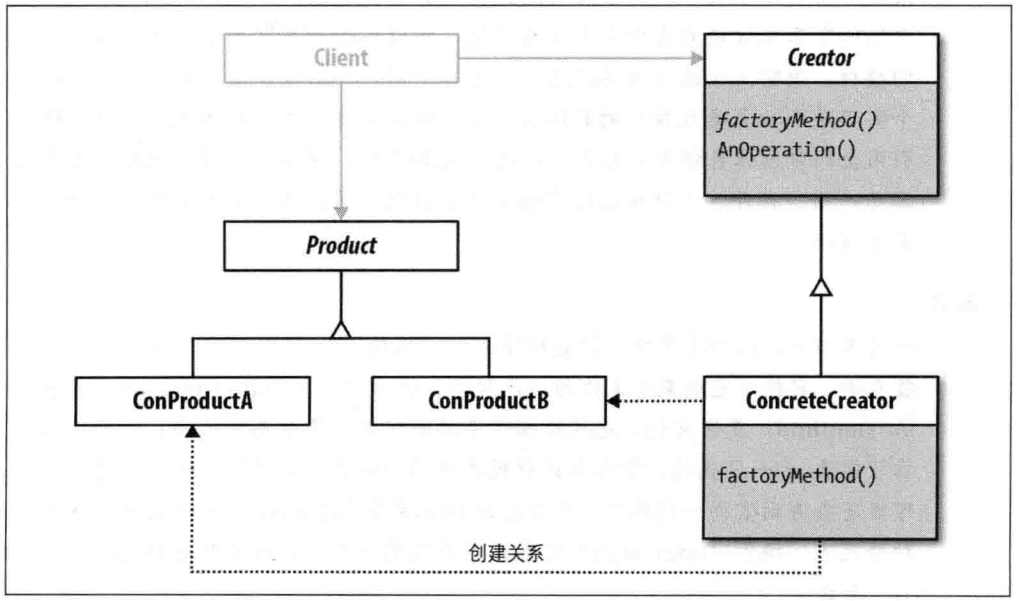


图4-10：具体创建者和具体产品之间的创建关系

使用PHP讨论类之间的相识关系时，有些情况下，可能有必要实例化一个类来包含另一个类或接口的引用。这种实例化的目的是要创建一个数据类型来包含引用，而不是为了使用这个实例化的对象。在PHP中如果需要这样做，请不要使用虚线。这种关系是相识关系，而非创建关系，因为尽管创建了一个实例，但是它只是用来建立这种相识关系。

4.4.5 多重关系

有时你会看到一个类图中相识或聚合关系的箭头末端有一个圆球，如图4-11所示。

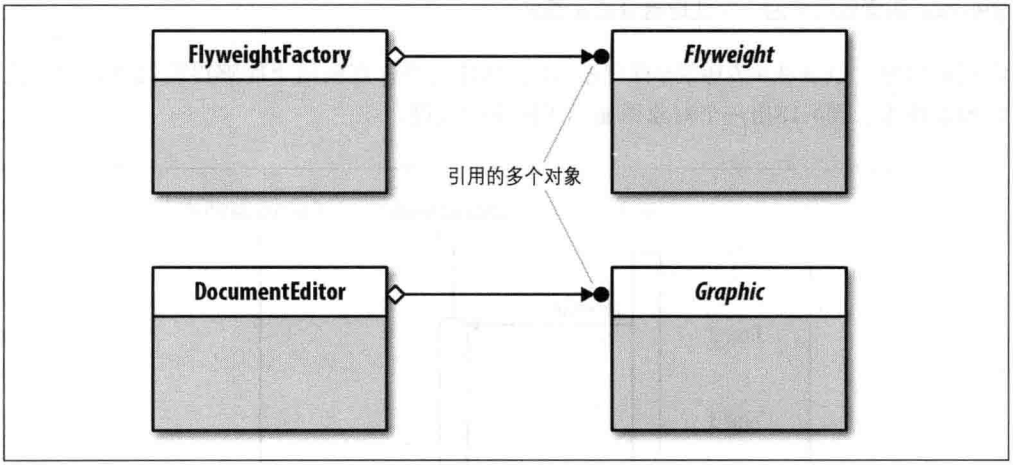


图4-11：黑色圆球指示引用多个对象

黑色圆球表示引用或聚合多个对象。可以使用享元（Flyweight）模式引用共享对象或不同对象的多个实例。

4.5 对象图

类图会显示类，与之不同，对象图只显示实例，箭头则指示所引用的对象。对象图采用的命名约定是使用类实例名，前面加一个小写字母a。所以，类`Client`的实例就是`aClient`。对象图模块下一半中的文本是目标对象或方法名。

在职责链（Chain of Responsibility）设计模式中可以看到一个使用对象图的很好的例子。如果可能有多个对象能够处理一个请求，如一个咨询台应用，就可以使用这种模式。客户发出一个请求，然后由一个处理器接口传递这个请求，直到找到一个具体的处理器来处理这个请求。图4-12展示了这个处理器链。

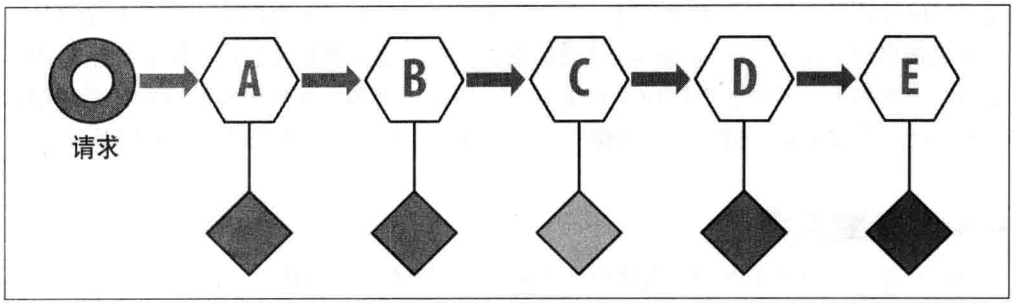


图4-12：职责链模式沿一个处理器链传递请求

在图4-12中，请求从A传递到B再传递到C，依此类推，直到请求能够得到处理，并终止处理器搜索。图4-13用一个对象图显示了同样的过程。

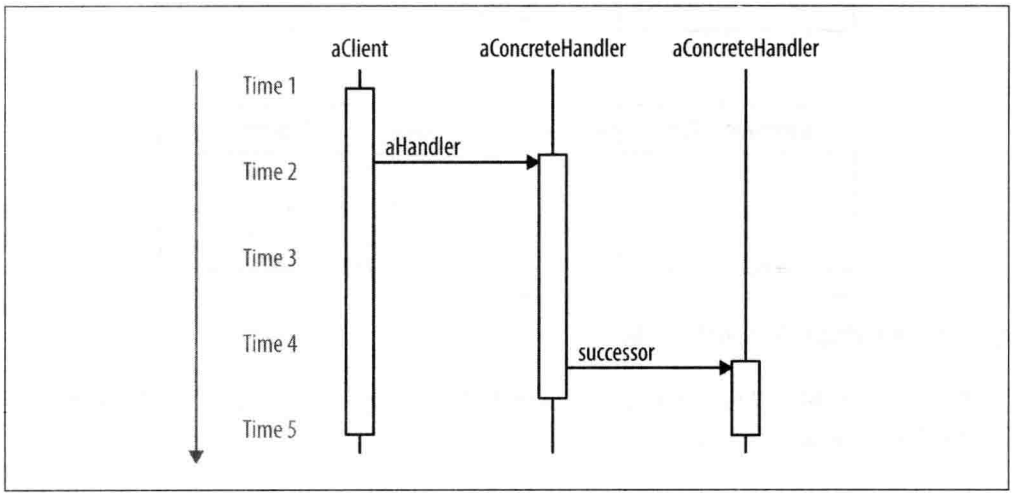


图4-13：职责链模式的对象图

对象图提供了另一种表示方法，可以查看一个设计模式特定实现以及通过这个模式生成的对象之间的关系。对象图可以清楚地表明对象关系。

4.6 交互图

GoF使用的最后一种图是交互图，可以跟踪执行请求的顺序。这些交互图在对象命名方面类似于对象图，不过其时间轴是垂直的，即从上到下。带箭头的实线指向请求的方向，带箭头的虚线指示实例化。图4-14同样显示了图4-13中的职责链序列，不过这里采用了交互图的形式（这里增加了灰色时间方向箭头和时间标签，但在具体交互图中它们并不出现）。

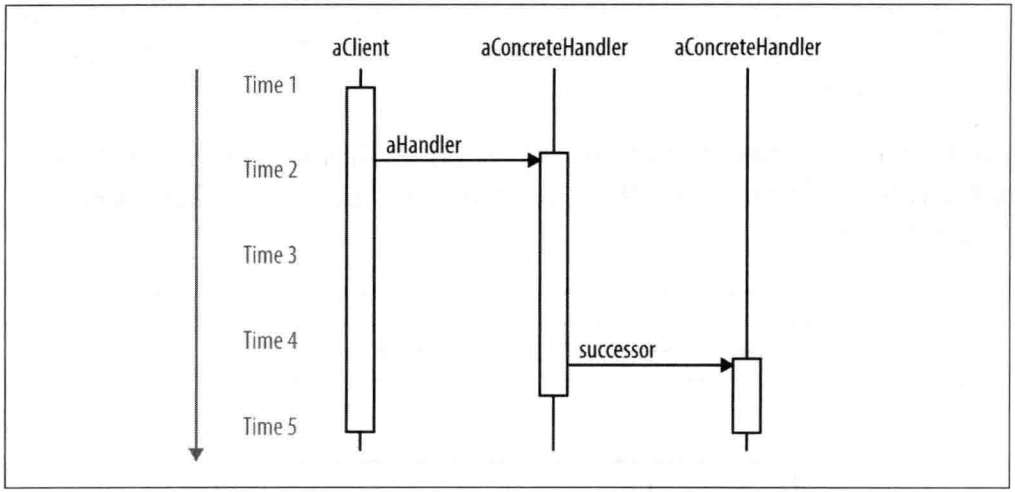


图4-14：职责链模式的交互图

垂直矩形指示该时刻对象是活动的。如图4-14中所示，客户对象全程都是活动的，请求从客户发出时，第一个具体处理器活动，此时第二个具体处理器并不是活动的，直到请求从第一个具体处理器传出时它才会变为活动。

4.7 面向对象编程中图和记法的作用

在分析第一个设计模式并用PHP实现之前，我想先稍停片刻，回顾一下我们要用这些图和记法做什么。也许你不记得了，所以这里再重申一句：这些图和记法可以用来解释、分析和构建设计模式。它们对我们有很大帮助，可以充分加以利用。不过，如果它们没有用，就应该果断丢掉，去创建新的工具。

多年来，我所知道的唯一的编程图就是传统的流程图。流程图看起来很好，但我很少在程序规划中使用，也很少用它来指导程序的编写。我了解它们的用处。不过，现在查看流程图时，我发现它们对于面向对象编程来说可能并不是得力的工具。流程图有利于指导顺序编程。与之相反，为OOP开发的UML看起来则完全不同。它们将程序分解为模块，称为类，而UML图的记法可以显示关系和交互。

4.8 UML工具

如果你有心，可能会找到很多帮助画UML图的工具，不过我建议你不要这么做。在《Unified Modeling Language User Guide, 2nd Edition》中，Grady Booch讨论自动化UML工具时指出：

> 尽管这些工具提供了自动化的标记支持，但它们还可能助纣为虐，帮助不好的设计人员更快地创建一些可怕的设计，如果没有这些工具，这些“次品”设计可能不会这么快出炉。

如果把UML当做一种辅助思考的工具，在代码设计方面你可能会更有把握，而且能更好地理解设计模式。例如，图4-15显示了我在PHP项目中使用的图（策略设计模式），而且得到了很好的效果。

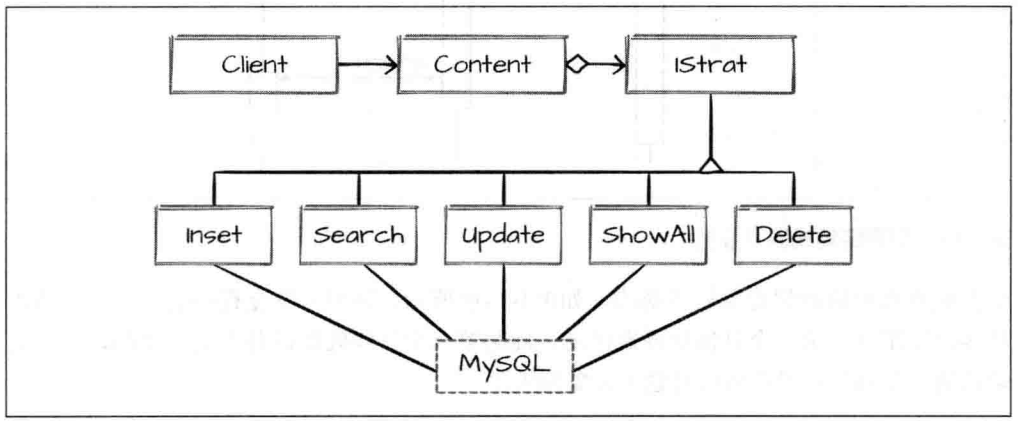


图4-15：在纸上画出设计草图

项目开始之后，随着更多细节、标注、注释和部分代码的加入，我会画更多草图。在勾画图案和细节时，我必须仔细思考。对于更大规模的项目和编程团队，有些UML绘图工具在简洁方面可能比较有优势，对于程序员之间的交流会很方便。不过，写这本书时，我的主要UML工具仍然是纸和笔（我最喜欢在卡通日历已经翻过不要的那些日历页上写写画画）。

4.9 其他UML

对于UML，现在不应当考虑“标准”，而应考虑如何有效地加以利用。Booch建议采用一种图表分类法，划分为结构图和行为图，交互图就是行为图的一种主要子类型。在这些图中，我发现状态图（statecharts）对于考虑使用状态机的设计尤其有用，特别是状态（State）设计模式。状态图可以帮助开发人员重点关注状态、状态改变、触发器和状态变迁。所以，尽管GoF使用的UML集中没有包括状态图，但如果要考虑的问题涉及状态机（如状态（State）设计模式的相关问题），使用状态图会很方便。（关于状态图的例子，参见第10章“状态设计模式”。）

开始学习设计模式时，除了熟悉说明和注释之外，还要知道它们是什么意思。例如，要了解聚合的含义，这比知道它由一个末尾有菱形、前端有箭头的线表示更重要。另外，如果开始使用设计模式UML，你可以更好地解决已经提供了设计模式解决方案的问题。使用一支笔和一张纸，就能更好地连接你的思维过程，这比你使用那些自动化工具更好，你还有机会更好地了解设计模式。

第2部分

创建型设计模式

想象是创造的开始。
你会想象你想要的东西，
你会渴望你想象的东西，
最终你会创造你渴望得到的东西。
——乔治·萧伯纳

如果没有文明以及它所蕴含的相对自由，
即使是一个完美的社会，也不过是一个丛林。
正是因为这个原因，
所有真正的创造都是为未来献上的一份礼物。
——阿尔贝·加缪

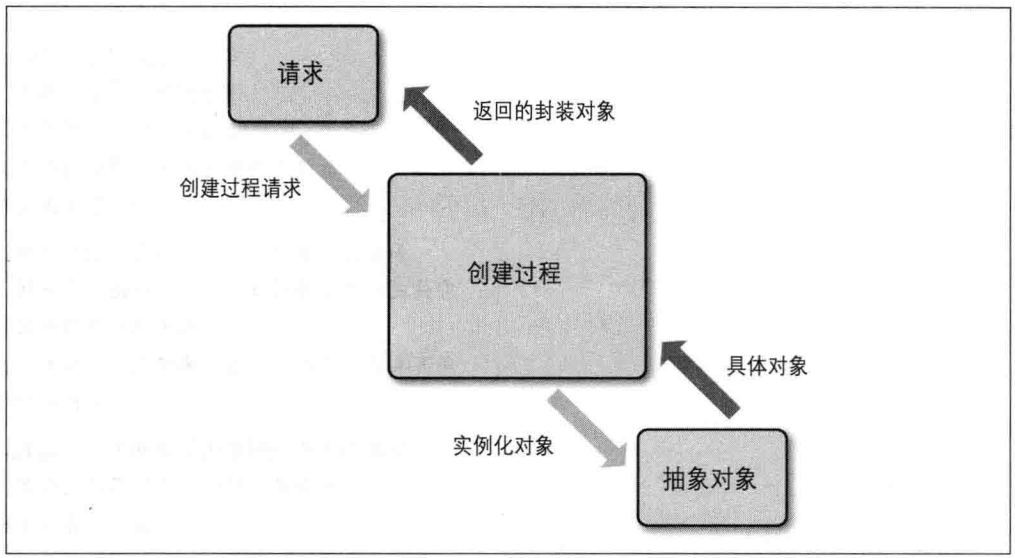
如果真主开始创建这个世界之前问过我，
我会建议创建一个更简单的世界。
——“智者”阿方索十世

创建型设计模式强调的是实例化过程。设计这些模式的目的是隐藏实例的创建过程，并封装对象使用的知识。Gamma、Helm、Johnson和Vlissides所列的5个创建型设计模式包括：

- 抽象工厂（Abstract Factory）
- 生成器（Builder）
- 工厂方法（Factory Method）
- 原型（Prototype）
- 单例（Singleton）

在这5个设计模式中，我们选择工厂方法（Factory Method）和原型（Prototype）模式作为将用PHP实现的创建型设计的例子。工厂方法模式是这5个设计模式中唯一的一种类设计模式，尽管相当简单，却是一种非常有效的模式。原型模式属于对象类模式，可以使用PHP `__clone()`方法实现。首先基于原型实例化（创建）一个对象，然后由这个实例化对象进一步克隆其他对象。你会发现这种模式很容易使用，相当方便。

使用创建型模式时，最有意思的是，程序和系统越来越依赖于对象组合而不是依赖于类继承时，创建型模式就会很重要——确切地说是至关重要。程序变成由对象构成的系统，而对象又由其他对象组合而成，所以任何单个对象的创建都不应该依赖于创建者。换句话说，对象不应与创建对象的过程紧密绑定。这样一来，就不会由于请求对象的特定特性而影响对象组合。第2部分介绍的设计模式会告诉你如何以最优的方式实现创建过程。图II-1概要描述了创建型模式如何工作。



图II-1：创建型模式结构

第5章

工厂方法设计模式

妇女运动发源于工厂里的工人，这是打破旧模式的一个绝好的运用。
——艾玛·博尼诺

设计就是将形式和内容汇集在一起的方法。
如同艺术，设计也有多种定义；而不是单一的一个定义。
设计可以是艺术。设计可以是美学。
设计是如此简单，正因如此，它又是如此复杂。
——保罗·兰德

创建你自己的方法。不要过分依赖于我给出的方法。
要确定最适合你的做法！
不过拜托要尽量打破常规。
——康斯坦丁·斯坦尼斯拉夫斯基

5.1 什么是工厂方法模式

作为一种创建型设计模式，工厂方法（Factory Method）模式就是要创建“某种东西”。对于工厂方法模式，要创建的“东西”是一个产品，这个产品与创建它的类之间不存在绑定。实际上，为了保持这种松耦合，客户会通过一个工厂发出请求。再由工厂创建所请求的产品。也可以换种方式考虑，利用工厂方法模式，请求者只发出请求，而不具体创建产品。图5-1显示了工厂方法模式的类图。

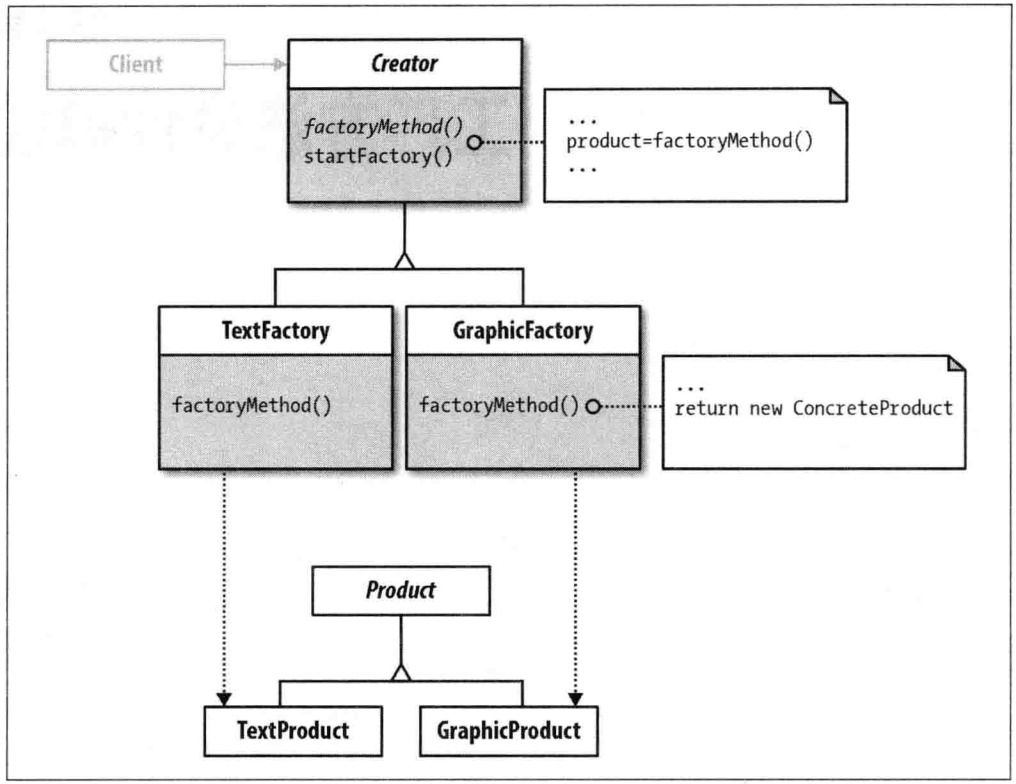


图5-1：实现的工厂方法类图

Client类是隐含的。从图5-1可以看到，Client包含Creator（工厂接口）的一个引用，用于请求一个图像或文本产品。不过，Client并不会实例化所请求的产品。将由具体的工厂实例化所请求的产品。可以想象一下，假设你想为一个好莱坞晚会订购有黑色和橙色糖霜的纸托巧克力蛋糕，你会给面包师（创建者）打电话，请他为你制作这些纸托蛋糕（产品）。你自己并不参与所请求对象的创建，但会得到你请求的纸托蛋糕。

5.2 何时使用工厂方法

从某个角度讲，如何选择设计模式取决于你希望能够改变什么。表3-1指出，如果实例化对象的子类可能变化，就要使用工厂方法模式。在这一章的例子中，Project接口的子类会变化；它们是不同的区域。你会看到，所开发的项目就是一些对象，由文本（文字）和图片（地图）组成。开始时，我们假设开发人员并不知道将会有多少个区域。换句话说，对象的数目和类型是未知的。一个类无法预计它要创建的对象数目，所以你不希望类与它要创建的类紧密绑定。

如果一个类要创建的对象数目固定，而且是已知的，那么构建这个类时，就可以采用一种可预测的方式创建指定数目的对象。例如，如果你在开发一个世界地图应用，由不同的对象表示7大洲，可以肯定这些对象不会改变。另一方面，如果你要为不同种类的昆虫创建一个网站，肯定会不断发现新的昆虫，或者有些昆虫会有改变，还有可能某些品种会在很短的时间内灭绝。要处理这种变化，程序必须有充分的灵活性。这种项目就可以考虑使用工厂方法设计模式。

5.3 最简单的例子

下面开始介绍工厂方法设计模式，第一个例子只返回文本。这是一个涉及地图和文本文字的项目，开发人员知道必须为这个项目创建不同的文本和图像元素。但他并不知道究竟需要创建多少个图像－文本对，甚至不确定客户希望增加什么。客户只是告诉他要有一个地图图像，还要增加相应的描述性文本。所以，首先他创建了一个很小的工厂方法设计，在屏幕上输出文本，分别显示“图像”信息和“文本”信息。如果一切正常，下一步可以修改这个项目以适应任意数目的文本和图像，这应该不难。

5.3.1 工厂的工作

第一步是建立工厂：Creator接口。在这个实现中，使用了一个抽象类作为Creator接口。仔细查看类图，可以看到，有一个代码标注指示了一个具体方法`startFactory()`。由于这个接口中使用了一个具体方法，由此我们知道，这个接口肯定是一个抽象类而不是接口。接口中只能包含抽象方法，所以这必然是一个抽象类。另外，这个设计需要一个抽象方法`factoryMethod()`。在一个抽象类中，所有这些方法都要明确指示为抽象方法；否则它们会作为具体方法。Creator接口是模式中的第一个参与者，*Creator.php*显示了这个参与者的代码：

```
<?php
//Creator.php
abstract class Creator
{
    protected abstract function factoryMethod();

    public function startFactory()
    {
        $mfg= $this->factoryMethod();
        return $mfg;
    }
}
?>
```

需要注意，伪代码注释指示`startFactory()`方法需要返回一个产品（product）。在实现

中，startFactory()希望由factoryMethod()返回一个产品对象。所以，factoryMethod()的具体实现要构建并返回由一个按Product接口实现的产品对象。

有两个具体工厂类扩展了Creator，并实现了factoryMethod()方法。factoryMethod()实现通过一个Product方法（getProperties()）返回一个文本或图像产品。TextFactory和GraphicFactory实现中加入了这些内容：

```
<?php
//TextFactory.php
include_once('Creator.php');
include_once('TextProduct.php');
class TextFactory extends Creator
{
    protected function factoryMethod()
    {
        $product=new TextProduct();
        return($product->getProperties());
    }
}
?>

<?php
//GraphicFactory.php
include_once('Creator.php');
include_once('GraphicProduct.php');
class GraphicFactory extends Creator
{
    protected function factoryMethod()
    {
        $product=new GraphicProduct();
        return($product->getProperties());
    }
}
?>
```

这两个工厂实现是类似的，只不过一个创建TextProduct实例，而另一个创建GraphicProduct实例。

Product

工厂方法设计模式中的第二个接口是Product。由于这是第一个实现，也是最简单的实现，所有文本和图像属性都只实现一个方法getProperties()：

```
<?php
//Product.php
interface Product
{
    public function getProperties();
}
?>
```

建立方法而无属性，利用这个实现，我们可以明确想要用getProperties()方法做什么。在PHP中，给定方法签名（方法名和可见性），我们可以让这个抽象方法做任何事情（比如，它可以有一个返回值），只要方法名和可见性与签名一致，就不会有问题。

可以看到，在工厂方法的这个实现中，getProperties()方法引入了多态（polymorphism），将用这个方法返回“文本”或“图像”。我们知道，只要有正确的签名，它就能提供我们想要的结果。同一个方法getProperties()有多个（poly）不同的形态（morphs），这就是多态。在这种情况下，其中一种形式返回文本，而另一种返回图像：

```
<?php
//TextProduct.php
include_once('Product.php');
class TextProduct implements Product
{
    private $mfgProduct;

    public function getProperties()
    {
        $this->mfgProduct="This is text.";
        return $this->mfgProduct;
    }
}
?>
```

你可能在想，“这好办，只是返回一个字符串变量而已”。目前确实如此。不过，你可以在实现中放入你想要的任何东西，工厂方法设计将会创建这个对象，并把它返回给Client来使用。所以，你看到输出“This is text”或“This is a graphic”时，可以想象成你可能希望创建和使用的任何对象。下一个实现就会利用消息“text graphic”返回一个抽象图像：

```
<?php
//GraphicProduct.php
include_once('Product.php');
class GraphicProduct implements Product
{
    private $mfgProduct;

    public function getProperties()
    {
        $this->mfgProduct="This is a graphic.<3";
        return $this->mfgProduct;
    }
}
?>
```

这两个工厂和产品实现分别覆盖了抽象方法，来创建两个不同的工厂和产品，它们都符合所实现的接口。

5.3.2 客户

这个模式最后一个参与者是隐含的：即客户。我们并不希望Client类直接做出产品请求。实际上，我们希望能够通过Creator接口做出请求。这样一来，如果以后我们增加了产品或工厂，客户可以做同样的请求来得到更多类型的产品，而不会破坏这个应用：

```
<?php
//Client.php
include_once('GraphicFactory.php');
include_once('TextFactory.php');
class Client
{
    private $someGraphicObject;
    private $someTextObject;

    public function __construct()
    {
        $this->someGraphicObject=new GraphicFactory();
        echo $this->someGraphicObject->startFactory() . "<br />";
        $this->someTextObject=new TextFactory();
        echo $this->someTextObject->startFactory() . "<br />";
    }
}

$worker=new Client();
?>
```

如果一切正常，输出将是：

```
This is a graphic.<3
This is text.
```

注意Client对象并没有向产品直接做出请求，而是通过工厂来请求。重要的是，客户并不实现产品特性，而留给产品实现来体现。

5.4 适应类的修改

设计模式的真正价值并不是提高操作的速度，而是加快开发的速度。在简单的应用中，比如目前这个工厂方法模式的例子，可能很难看出这一点。不过，随着我们逐步做出修改，就会越来越明显地看出设计模式的价值。

5.4.1 增加图像元素

第一步是修改产品，把一个图像加载到一个HTML文档中。就其本身而言，向web页面增加图像非常简单，但是随着PHP程序变得越来越复杂，难度也会增大。下面的代码显示了修改后的GraphicProduct类：

```
<?php
//GraphicProduct.php
include_once('Product.php');
class GraphicProduct implements Product
{
    private $mfgProduct;

    public function getProperties()
    {
        $this->mfgProduct="<!doctype html><html><head><meta charset='UTF-8' />";
        $this->mfgProduct.="<title>Map Factory</title>";
        $this->mfgProduct.="</head><body>";
        $this->mfgProduct.="<img src='Mali.png' width='500' height='500' />";
        $this->mfgProduct.="</body></html>";
        return $this->mfgProduct;
    }
}
?>
```

同样地，可以看到多态的作用。同一个getProperties()方法现在有一个完全不同的实现。这是不是意味着客户必须改变请求？不，请求一个图像时，Client类只是去掉了文本请求，如下所示：

```
<?php
//Client.php
include_once('GraphicFactory.php');
class Client
{
    private $someGraphicObject;
    private $someTextObject;

    public function __construct()
    {
        $this->someGraphicObject=new GraphicFactory();
        echo $this->someGraphicObject->startFactory() . "<br />";
    }
}

$worker=new Client();
?>
```

这与第一个客户做出的请求完全相同。不过，由于GraphicProduct对象改变了，所以输出也会改变，如图5-2所示。

在图5-2中可以看到这个图像包含有文本，不过这个文本是图像的一部分，而不是放置在文档中的HTML文本。

5.4.2 调整产品

在PHP中得到产品并不太困难，不过随着网站不断发展，变得越来越复杂，能够简单地做出改变显得越来越重要。下一步是处理文本和图像，使它们能一同放在一个文档中。

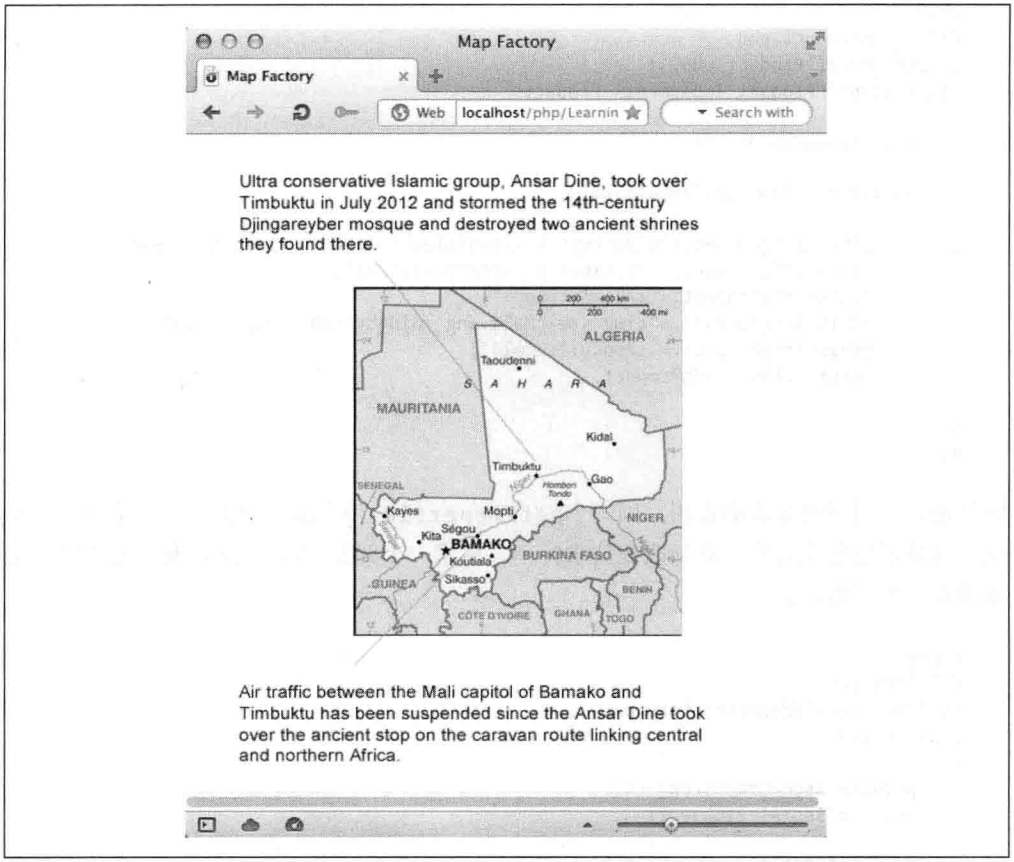


图5-2：提供嵌入文本的图像对象

如图5-2所示，文本已经集成到图像中，所以这里的修改展示了可以在页面中加载图像，而不必改变来自`GraphicFactory`类的客户请求。如果要调整多个产品，也可以这么做吗？

参考CIA的*World Factbook*提供的资料，我们集成了一个地图和文字说明，如图5-3所示。

引入新产品时，网站会不断壮大，变得越来越复杂，工厂方法模式有助于简化这些日益复杂的网站做出的请求。要创建如图5-3所示的网站，只需要改变文本和图像产品。应用中的所有其他参与者都保持不变，因为请求只依赖于接口，而不依赖于具体的产品。

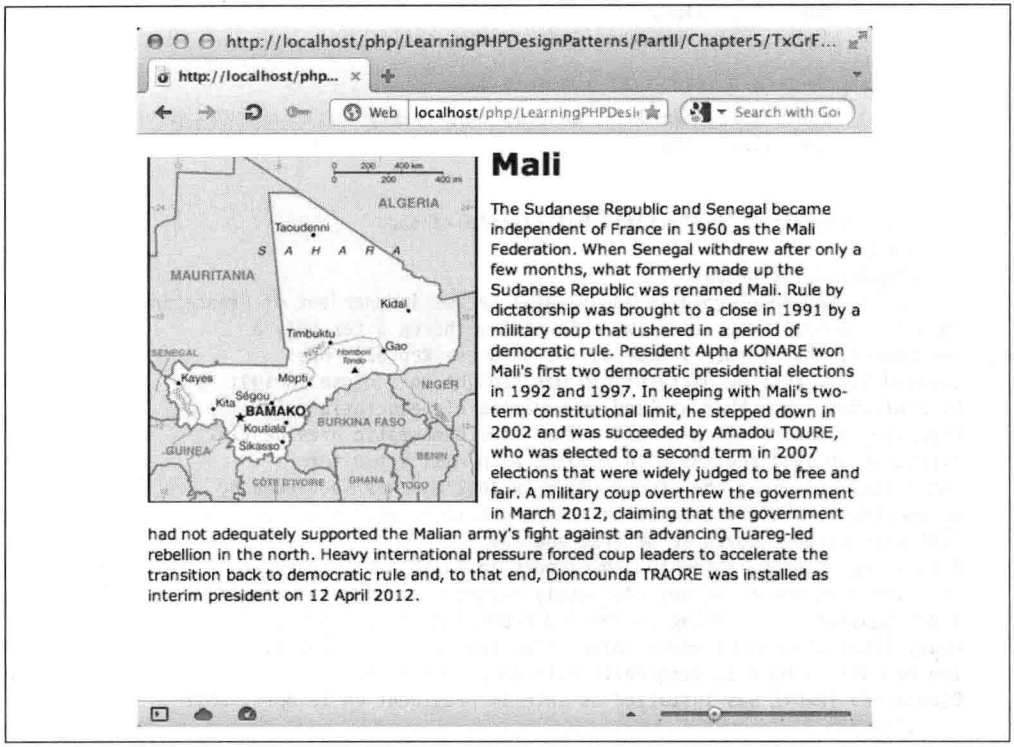


图5-3：调整的文本和图像对象

5.4.3 修改文本产品

修改文本产品相当简单。返回的产品将包含某种格式和一个标题，另外还会向请求客户返回同样的变量，并在屏幕上显示。下面的代码清单显示了TextProduct类中的改变：

```
<?php
//TextProduct.php
include_once('Product.php');
class TextProduct implements Product
{
    private $mfgProduct;

    public function getProperties()
    {
        //开始heredoc格式化
        $this->mfgProduct =<<<MALI
        <!doctype html>
        <html><head>
        <style type="text/css">
        header {
            color: #900;
            font-weight: bold;
```

```
            font-size: 24px;
            font-family: Verdana, Geneva, sans-serif;
        }
        p {
            font-family: Verdana, Geneva, sans-serif;
            font-size: 12px;
        }
    </style>
    <meta charset="UTF-8"><title>Mali</title></head>
    <body>
    <header>Mali</header>
    <p>The Sudanese Republic and Senegal became independent of France in
1960 as the Mali Federation. When Senegal withdrew after only a
few months, what formerly made up the Sudanese Republic was
renamed Mali. Rule by dictatorship was brought to a close in 1991
by a military coup that ushered in a period of democratic rule.
President Alpha KONARE won Mali's first two democratic presidential
elections in 1992 and 1997. In keeping with Mali's two-term
constitutional limit, he stepped down in 2002 and was succeeded by
Amadou TOURE, who was elected to a second term in 2007 elections
that were widely judged to be free and fair.
A military coup overthrew the government in March 2012, claiming
that the government had not adequately supported the Malian army's
fight against an advancing Tuareg-led rebellion in the north.
Heavy international pressure forced coup leaders to accelerate
the transition back to democratic rule and, to that end,
Dioncounda TRAORE was installed as interim president on 12 April 2012
    </p>
    </body></html>
MALI;
        return $this->mfgProduct;
    }
}
?>
```

文本对象的改变看起来很简单，不过Product的getProperties()方法仍保持相同的接口，请求工厂返回一个属性对象。采用Heredoc格式，开发人员可以正常编写HTML代码，而不必把每一行文字都用引号引起来，另外heredoc变量中还可以接受PHP变量和常量（在本章后面的“辅助类”一节中，你会看到一个“辅助”类如何负责完成文本的格式化）。

5.4.4 修改图像产品

查看图像对象类时，可以看到，其方法和接口与只返回无HTML格式的文本时完全相同：

```
<?php
//GraphicProduct.php
include_once('Product.php');
class GraphicProduct implements Product
```

```
{
    private $mfgProduct;

    public function getProperties()
    {
        $this->mfgProduct="<img style='padding: 10px 10px 10px 0px';
        src='Mali.png' align='left' width='256' height='274'>";
        return $this->mfgProduct;
    }
}
?>
```

与文本产品和多态魔法类似，getProperties()方法就像蟑螂一样有强大的复原能力。工厂对象没有任何改变， Client仍发出同样的请求，只是由请求“文本”改为请求“图像”产品。

5.4.5 增加新产品和参数化请求

到目前为止，你已经看到，改变图像和文本并不会扰乱工厂方法设计，不过如果开始增加更多的地图和文字说明，会怎么样呢？有没有必要每次增加一个新的区域就增加一个新的具体工厂类？这意味着需要为每个新区域增加一个新工厂和产品；下面来看一种参数化工厂方法设计。图5-4显示了这样一个工厂方法实现。

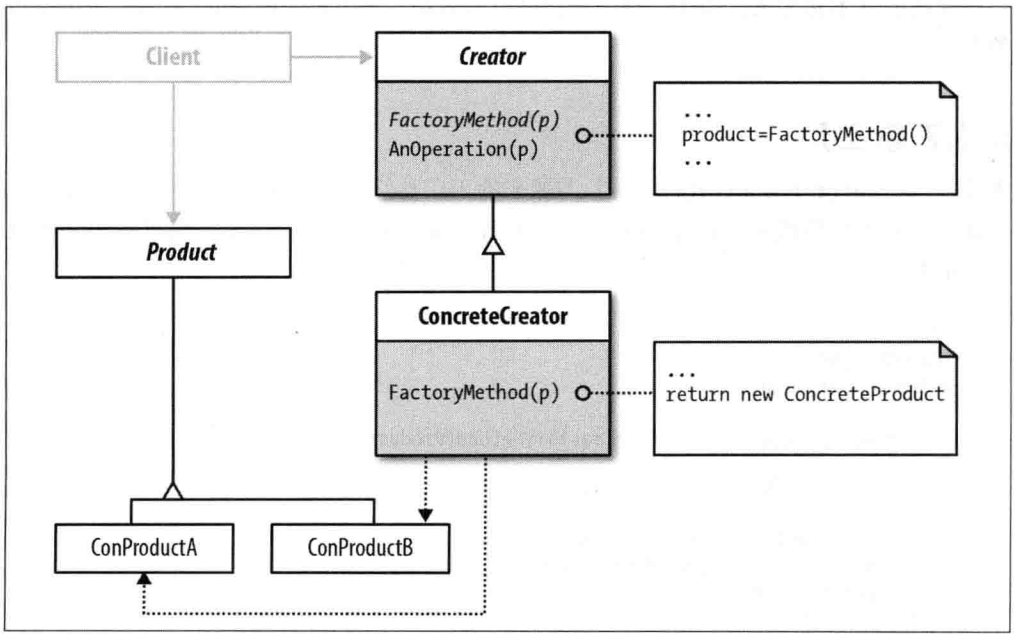


图5-4：一个具体创建者负责生产多个产品

图5-4中的类图与原类图有一些不同。这两个类图都准确地描述了工厂方法设计模式，它们都完成同样的目标，不过它们的实现有所不同。

如图5-1所示，参数化工厂方法设计模式与一般的工厂方法设计模式的主要区别之一是客户包含工厂和产品的引用。在参数化请求中，Client类必须指定产品，而不只是产品工厂。factoryMethod()操作中的参数是由客户传入的一个产品；所以客户必须指出它想要的具体产品。不过，这个请求仍然通过Creator接口发出。所以，尽管客户包含一个产品引用，但通过Creator，客户仍与产品分离。

5.4.6 一个工厂多个产品

对于大多数请求，参数化工厂方法更为简单，因为客户只需要处理一个具体工厂。工厂方法操作有一个参数，指示需要创建的产品。而在原来的设计中，每个产品都有自己的工厂，不需要另外传递参数；产品实现依赖于各个产品的特定工厂。

要从参数化工厂方法设计模式实现多个产品，只需使用Product接口实现多个具体产品。另外，由于产品要同时包含文本和图像，所以在这个例子中，并不是分别有这两个单独的产品，可以建立一个类，将文本和图像作为一个统一的实体来处理，这并不违反单一职责原则，即每个类应当只有一个职责。对于这个类来说，这个单一职责就是显示描述一个区域的文本和图像。由于这是一个很简单的应用，所以每个Product类的职责也相当简单。

5.4.7 新工厂

新工厂（Creator和CountryCreator）与原来的工厂类似，不过它们还包含一个参数和代码提示。根据代码提示，只要按接口（Product）编程就可以继续开发，而不需要Product接口的具体实现：

```
<?php
//Creator.php
abstract class Creator
{
    protected abstract function factoryMethod(Product $product);

    public function doFactory($productNow)
    {
        $countryProduct=$productNow;
        $mfg= $this->factoryMethod($countryProduct);
        return $mfg;
    }
}
?>
```

在这个新Creator抽象类中可以看到，factoryMethod()和startFactory()操作都需要一个参数。另外，由于代码提示指定了一个Product对象，而不是Product的一个特定实现，所以可以接受Product的任何具体实例。

具体的创建者类CountryCreator实现了factoryMethod()，并提供了代码提示要求的参数。当然，这个类继承了将由Client使用的startFactory()方法：

```
<?php
//CountryFactory.php
include_once('Creator.php');
include_once('Product.php');
class CountryFactory extends Creator
{
    private $country;

    protected function factoryMethod(Product $product)
    {
        $this->country=$product;
        return($this->country->getProperties());
    }
}
?>
```

这个具体创建者类包含一个私有变量$country，其中包含客户请求的特定产品。它再使用Product的方法getProperties()将产品返回给客户。

5.4.8 新产品

具体产品中的变化并不会改变原来的Product接口，这个接口与原来完全相同：

```
<?php
//Product.php
interface Product
{
    public function getProperties();
}
?>
```

这意味着具体产品也必须有相同的接口，可以看到，也确实如此。不过，新的产品实现中同时包含图像和文本。文本嵌在类本身（可能来自于一个文本文件或一个数据库），另外还嵌在图像中，用一个简单的<img src...>标记显示。下面的类给出了一个例子，这里的文本和地图图像选自CIA的*World Factbook*：

```
<?php
//TextProduct.php
include_once('FormatHelper.php');
include_once('Product.php');
```

```
class KyrgyzstanProduct implements Product
{
    private $mfgProduct;
    private $formatHelper;

    public function getProperties()
    {
        $this->formatHelper=new FormatHelper();
        $this->mfgProduct=$this->formatHelper->addTop();
        $this->mfgProduct.=<<<KYRGYZSTAN
        <img src='Countries/Kyrgyzstan.png' class='pixRight' width='600'
           height='304'>
        <header>Kyrgyzstan</header>
        <p>A Central Asian country of incredible natural beauty and proud
           nomadic traditions, most of Kyrgyzstan was formally annexed to
           Russia in 1876. The Kyrgyz staged a major revolt against the
           Tsarist Empire in 1916 in which almost one-sixth of the Kyrgyz
           population was killed. Kyrgyzstan became a Soviet republic in 1936
           and achieved independence in 1991 when the USSR dissolved. Nationwide
           demonstrations in the spring of 2005 resulted in the ouster of
           President Askar AKAEV, who had run the country since 1990.
           Subsequent presidential elections in July 2005 were won overwhelmingly
           by former prime minister Kurmanbek BAKIEV. Over the next few years,
           the new president manipulated the parliament to accrue new powers
           for himself. In July 2009, after months of harassment against
           his opponents and media critics, BAKIEV won re-election in a
           presidential campaign that the international community deemed
           flawed. In April 2010, nationwide protests led to the resignation
           and expulsion of BAKIEV. His successor, Roza OTUNBAEVA, served as
           transitional president until Almazbek ATAMBAEV was inaugurated in
           December 2011. Continuing concerns include: the trajectory of
           democratization, endemic corruption, poor interethnic relations,
           and terrorism.
        </p>
KYRGYZSTAN;
        $this->mfgProduct .=$this->formatHelper->closeUp();
        return $this->mfgProduct;
    }
}
?>
```

原来的工厂方法设计中，区分了图像和文本的工厂，不过这两个设计的输出并没有变化。图5-5显示了你可能看到的结果。

你可能已经注意到，图像显示在右边而不是左边，另外与Mali的地图相比要更大一些，不过除此以外，几乎没有不同。这里新增的内容是增加了一个名为FormatHelper的类实例。这是一个“辅助”类，需要在设计模式和特定实现的上下文中才能更好地解释这个类，不过，首先再来Client类，因为它也有变化——现在它也需要一个参数。

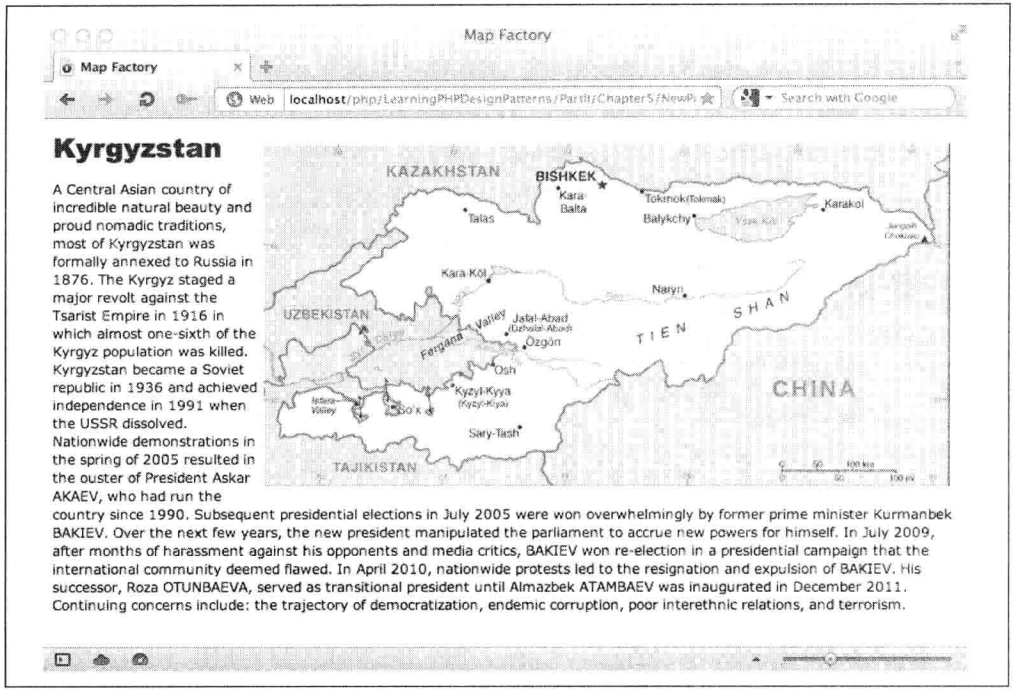


图5-5：参数化工厂允许多个特定产品

5.4.9 有参数的客户

从本章前面的例子可以看到，客户只是通过特定产品工厂的工厂接口做出一个请求。修改后，Client类现在必须包含一个参数：

```
<?php
//Client.php
include_once('CountryFactory.php');
include_once('KyrgyzstanProduct.php');
class Client
{
    private $countryFactory;

    public function __construct()
    {
        $this->countryFactory=new CountryFactory();
        echo $this->countryFactory->doFactory(new KyrgyzstanProduct());
    }
}

$worker=new Client();
?>
```

5.4.10 辅助类

有些任务最好由一个单独的对象来处理，而不应结合到某个参与者中，设计模式中的辅助类正是用来完成这样一些任务。可以认为辅助类就类似于一个外部CSS文件。可以为每一个类增加同样的CSS，不过如果把CSS放在一个单独的文件中，并通过加入一个<link>标记来调用样式表实现CSS的重用，这样会高效得多。类似地，如果需要重用一组HTML格式标记，可以把它们打包到另一个对象中以便重用。下面显示了这个应用使用的辅助类：

```
<?php
class FormatHelper
{
private $topper;
private $bottom;

    public function addTop()
    {
        $this->topper="<!doctype html><html><head>
        <link rel='stylesheet' type='text/css' href='products.css'/>
        <meta charset='UTF-8'>
        <title>Map Factory</title>
        </head>
        <body>";
        return $this->topper;
    }

    public function closeUp()
    {
        $this->bottom="</body></html>";
        return $this->bottom;
    }
}
?>
```

这里不仅提供了一个HTML包装器，还调用了CSS文件*products.css*。为方便起见，辅助类包含两个公共方法（addTop()和closeUp()），并将需要增加到HTML页面顶部和底部的HTML代码分别放在这两个方法中。采用这种方式，可以将实例化的具体产品放在正确的HTML格式化标记之间。

CSS样式表还为开发人员和设计人员提供了一些选择。这里给出的两个CSS类（pixLeft和pixRight）分别允许对图像左对齐和右对齐：

```
@charset "UTF-8";
/* CSS Document */
img
{
    padding: 10px 10px 10px 0px;
}
```

```
.pixRight
{
    float:right; margin: 0px 0px 5px 5px;
}

.pixLeft
{
    float:left; margin: 0px 5px 5px 0px;
}

header
{
    color:#900;
    font-size:24px;
    font-family:"Arial Black", Gadget, sans-serif;
}

body
{
    font-family:Verdana, Geneva, sans-serif;
    font-size:12px;
}
```

甚至可以把CSS文件想成是一个“辅助”类。类似地，你可能希望在更大规模的设计中增加一些JavaScript或jQuery脚本作为外部辅助类。

5.4.11 文件图

我还发现一种不算官方的图也很有帮助，即文件图（file diagram）。基本说来，文件图包括一个设计模式中用到的文件和文件夹，另外还包括一些指示关系标注的链接。这与类图很类似，不过文件图中提供的是用到的具体文件。图5-6显示了本章中使用的最后一个工厂方法模式的文件图。

在图中可以看到，尽管这些辅助类和资源不是设计模式的一部分，但是在产品会用到。虚线框指示它们与模式本身是分离的，大箭头表示具体产品会用到它们。

5.4.12 产品改变：接口不变

使用设计模式的一大好处就是可以很容易地对类做出改变，而不会破坏更大的程序。之所以能够更容易地做出改变，其秘密在于保持接口不变，而只改变内容。

可以将文本从具体产品中取出，这个修改能够进一步简化问题。将文字说明放在文本文件中，然后再加载到变量中，这样一来，不仅可以更容易地改变文本内容，具体类也会更简洁。

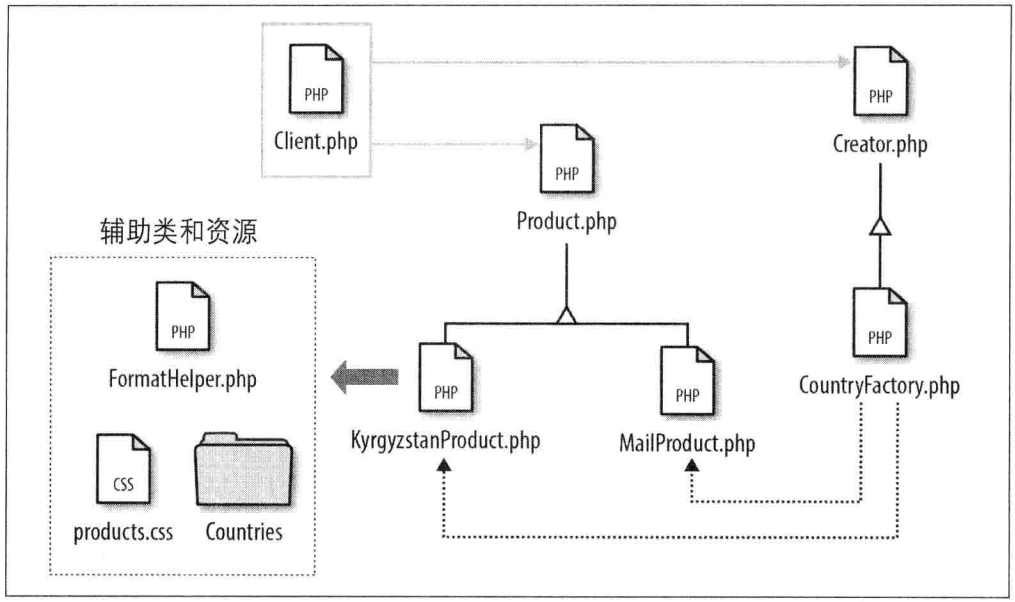


图5-6：增加的辅助类和资源

在这个新的具体产品中，由一个小例程将文本增加到一个私有变量$countryNow中。所以具体产品中不再包含一大堆乱七八糟的文本，这里使用了5行代码将文字说明放在一个变量中。下面显示了一个新产品（Moldova），并给出了处理文本的一种新方法：

```
<?php
//MoldovaProduct.php
include_once('FormatHelper.php');
include_once('Product.php');

class MoldovaProduct implements Product
{
    private $mfgProduct;
    private $formatHelper;
    private $countryNow;

    public function getProperties()
    {
        //从外部文本文件加载文本说明
        $this->countryNow = file_get_contents("CountryWriteups/Moldova.txt");

        $this->formatHelper=new FormatHelper();
        $this->mfgProduct=$this->formatHelper->addTop();
        $this->mfgProduct.="<img src='Countries/Moldova.png' class='pixRight'
          width='208' height='450'>";
        $this->mfgProduct .="<header>Moldova</header>";
        $this->mfgProduct .="<p>$this->countryNow</p>";
        $this->mfgProduct .=$this->formatHelper->closeUp();
        return $this->mfgProduct;
```

```
        }
    }
?>
```

可以看到，那些乱七八糟的文本已经没有了。不过，getProperties()接口仍然不变，而且保持不变至关重要。只要这个接口不变，在工厂方法设计模式中做任何改变或增补都不会导致破坏。即使增加一个额外的外部资源，也不会有影响。图5-7显示了新的结果和原来的结果，可以看到，结果是类似的，只是国家不同。

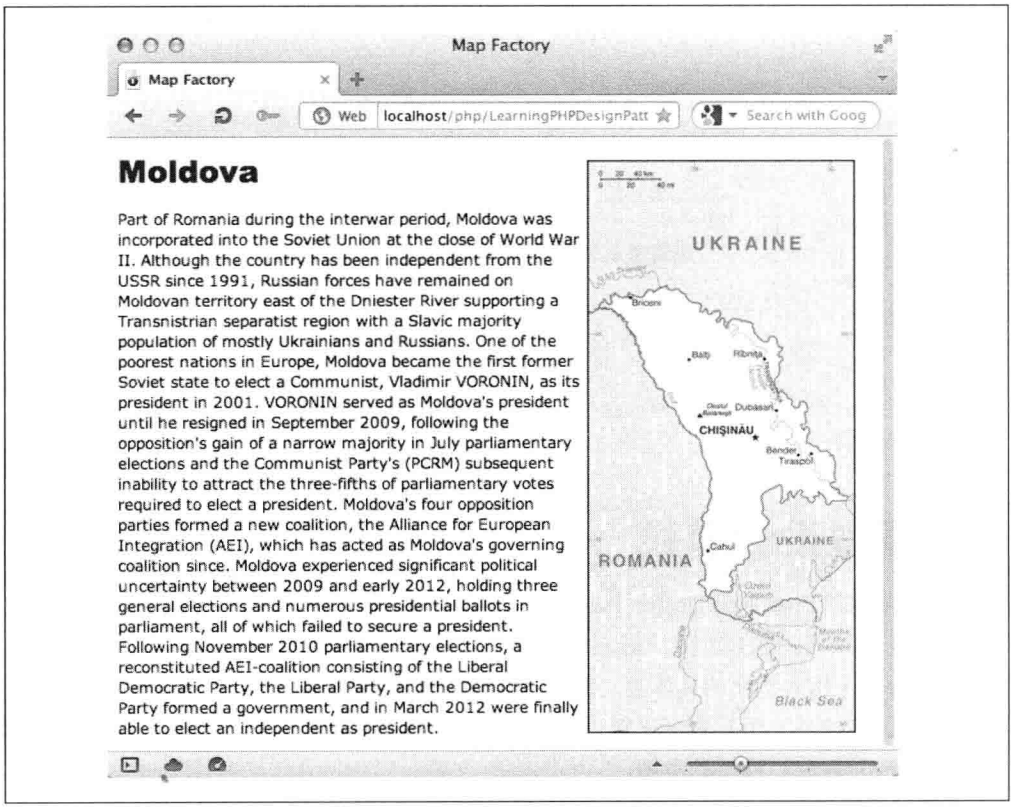


图5-7：利用外部资源创建

随着产品越来越复杂，可能不只是简单地将文本和图像放在一个HTML文档中，接口将越发重要。幸运的是，与试图让任意数目的类和对象都保持不变相比，保持接口不变要容易得多。正是因为这个原因，使用工厂方法模式可以简化复杂的创建过程，关键就在于它会维持一个公共接口。

第6章

原型设计模式

所谓原创不过是深思熟虑的模仿。最具原创力的作家往往互相抄袭。

——伏尔泰

为了与他人相似，我们舍弃了真我的四分之三。

——亚瑟·叔本华

就像你的行为原则会被定为全世界的准则那样行事。

——伊曼努尔·康德

6.1 原型设计模式

原型设计模式（Prototype Design Pattern）很有意思，因为它使用了一种克隆（cloning）技术来复制实例化的对象。新对象是通过复制原型实例创建的。在这里，实例（instance）是指实例化的具体类。原型设计模式的目的是通过使用克隆以减少实例化对象的开销。与其从一个类实例化新对象，完全可以使用一个已有实例的克隆。图6-1显示了原型（Prototype）模型的类图。

注意`Client`类是原型设计模式中不可缺少的一部分。客户通过Prototype接口创建一个具体原型的实例，Prototype接口中包含一个克隆方法。幸运的是，PHP有一个内置的`clone()`方法，可以在设计模式中使用。你会看到，这种设计的基本原理相当简单。

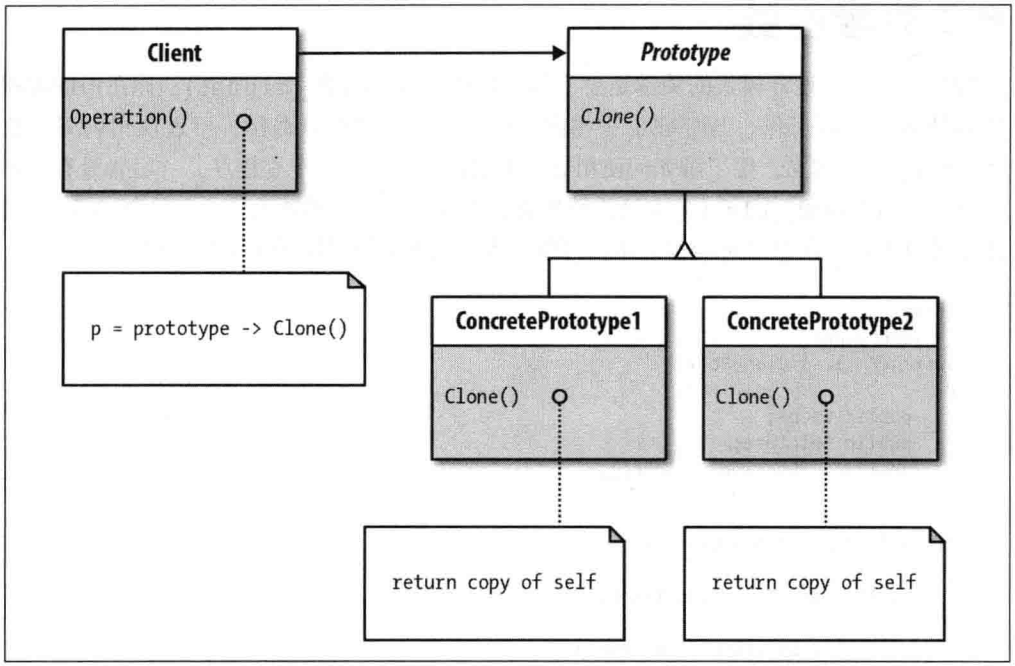


图6-1：原型类图

6.2 何时使用原型模式

如果一个项目要求你创建某个原型对象的多个实例，就可以使用原型模式。例如，在关于进化发展的研究中，科学家通常会使用果蝇作为研究对象。果蝇能很快繁殖，而且产生变异（基本进化变异事件）的概率也更大。例如，一般研究可能会使用1500万只果蝇，由于雌性几乎一出生就会产卵（1小时内），与其他生物相比，找到和记录变异的可能性会大得多，比如大象的孕育期就长达22个月。记录变异时，可以将雄性和雌性果蝇原型作为基础，变异则是某个雄性或雌性实例的克隆。因此，只需完成两个实例化（一个雄性和一个雌性），然后就可以根据需要克隆多个变异，而不需要由具体类另外创建实例。

原型模式还可以用来创建一种组织结构，可以根据实际的组织来创建和填充其中的位置（数目有限）。例如，可以使用组合创建一个绘图对象，然后克隆为不同版本的绘图对象，这就使用了原型模式。最后再来看一个使用原型模式的例子，在游戏开发中，可以通过克隆一个原型士兵，来增加军队的人数和兵种。

6.3 克隆函数

PHP中使用原型设计模式的关键是要了解如何使用内置函数__clone()。你在PHP编程中可能做过其他工作，与那些工作相比，这个要求可能有点奇怪，不过使用内置函数__clone()相当容易。要了解如何使用这个方法，来看下面这个小程序。（尽管没有必要扩展一个包含抽象__clone()方法的抽象类，不过这些例子都显示了一个包含__clone()方法的抽象类，所以这可能很有用，可以在本书后面的原型例子中得到证明。）

```
<?php
//CloneMe.php
abstract class CloneMe
{
    public $name;
    public $picture;
    abstract function __clone();
}

class Person extends CloneMe
{
    public function __construct()
    {
        $this->picture="cloneMan.png";
        $this->name ="Original";
    }

    public function display()
    {
        echo "<img src='$this->picture'>";
        echo "<br />$this->name <p />";
    }

    function __clone() {}
}
$worker=new Person();
$worker->display();

$slacker = clone $worker;
$slacker->name="Cloned";
$slacker->display();
?>
```

具体类Person扩展了CloneMe抽象类。在这个例子中，$worker实例化一个Person实例。所以现在$worker是一个Person对象。接下来，第二个实例变量$slacker克隆Person实例$worker。它可以像$worker一样访问相同的属性，而且能够像Person类的直接实例一样改变这些属性。图6-2显示了这个例子的结果。

__clone()方法不能直接访问。实际上，如果类定义中包含一个__clone()方法（采用以下格式）：

```
$anotherInstance = clone $someInstance;
```

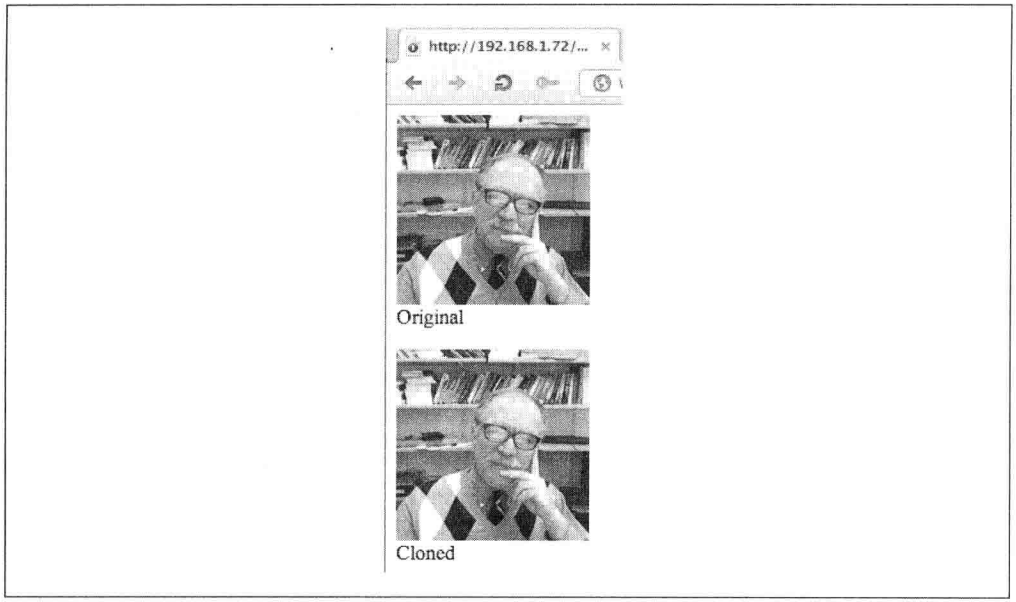


图6-2：浏览器中显示的原对象和克隆对象

对于所克隆的实例，clone关键字会为该实例的类实例化另一个实例（副本）。PHP文档指出：

> 使用clone关键字可以创建一个对象副本（如果可能，会调用对象的__clone()方法）。但不能直接调用对象的__clone()方法。

6.3.1 克隆不会启动构造函数

关于克隆过程，有一点要注意：克隆不会启动构造函数中的动作。克隆可以使用构造函数生成的默认赋值，但是如果构造函数生成一个动作，如一旦实例化就打印一条消息，克隆并不会显示这个消息（“Hello, clone!”）。下面的例子显示了实例化时构造函数会发出一个消息，而是在克隆操作时却不会产生这个消息：

```
<?php
class HelloClone
{
    private $builtInConstructor;
    public function __construct()
    {
        echo "Hello, clone!<br />";
        $this->builtInConstructor="Constructor creation<br />";
    }

    public function doWork()
    {
```

```
            echo $this->builtInConstructor;
            echo "I'm doing the work!<p />";
        }
    }
    //启动构造函数
    $original=new HelloClone();
    $original->doWork();

    //克隆不启动构造函数
    $cloneIt = clone $original;
    $cloneIt->doWork();

    ?>
```

实例化时会显示消息“Hello, clone!”，但是克隆操作没有显示任何消息。结果见以下输出：

```
Hello, clone!
Constructor creation
I'm doing the work!
Constructor creation
I'm doing the work!
```

对于原型设计模式（以及其他使用克隆的方法）来说，这意味着不能依赖于构造函数提供重要的输出或返回结果。不过，这可能不是一件坏事。实际上，这是一个很好的编程实践。

6.3.2 构造函数不要做具体工作

为Google制订编码原则的Miško Hevery在讨论单元测试时指出，构造函数不要做具体的工作（*http://bit.ly/1a9MWr*）。这个说法是在单元测试上下文（程序的测试部分）中提出的，不过它也适用于设计模式。Hevery的主要观点是，如果一个类实例化时要完成大量初始化，结果往往不灵活，而且这是过度耦合的设计。同样地，如果一个构造函数输出某些结果，客户除了能看到构造函数发送的结果外，没有任何其他选择，毕竟客户可能并不想要这些结果，或者至少并不想在给定时间得到这些结果。

要正确理解Hevery的观点，并不是说构造函数不能根据需要为属性赋值。也就是说，客户的构造函数可能与模式中其他参与者完全不同，因为它要向参与者做出请求。

对于这个概念，即构造函数不应完成具体的工作，一种做法是忽略模式类中的构造函数，除非你有充分的理由包含这些构造函数；另外一种做法是，允许在需要时调用操作，让客户负责实例化和克隆的有关事务。所以，尽管我们可能发现使用`__clone()`函数时存在限制，但这些限制可能更有助于完成更好的OOP程序。

6.4 最简单的原型例子

第一个例子将考虑一个使用果蝇的实验。研究的目标是建立一个原型果蝇，然后一旦出现变异，就构建这个变异果蝇。对于5000万只果蝇，你会得到大量变异，不过你只对果蝇的眼睛颜色、每秒翅膀扇动次数和复眼单元数（因为果蝇的眼睛包含上百个感光单位，每个感光单位都有自己的水晶体和一组光线接收单元）感兴趣，它们可能根据性别和繁殖情况有变化。其他变异也有可能，不过这个研究只考虑这3个变量。

研究果蝇

之所以选择原型模式完成果蝇研究，原因在于原型提供了一个起点，可以根据这个起点来度量变异。具体类建立了果蝇标准值的一个基准，可以用与这个基准的偏差来度量变异。由一个抽象类提供基准变量（为了便于这个例子使用，所有变量都是公共的）。

抽象类接口和具体实现

原型（`IPrototype`）的两个具体类实现分别表示不同性别的果蝇，包括性别变量（gender）和不同性别的行为（交配和产卵）。抽象类还包含一个基于`__clone()`方法的抽象方法。

```
<?php
//IPrototype.php
abstract class IPrototype
{

    public $eyeColor;
    public $wingBeat;
    public $unitEyes;
    abstract function —clone();
}
?>
```

`IPrototype`的这两个实现的区别体现在性别上，性别用常量标识，一个是`MALE`，另一个是`FEMALE`。雄果蝇有一个`$mated`布尔变量，雄果蝇交配之后，这个布尔变量会设置为`true`，雌果蝇有一个`$fecundity`变量，其中包含一个数字值，表示这只雌性果蝇的繁殖能力（产卵个数）：

```
<?php
//MaleProto.php
include_once('IPrototype.php');
class MaleProto extends IPrototype
{
    const gender="MALE";
    public $mated;
```

```
    public function __construct()
    {
        $this->eyeColor="red";
        $this->wingBeat="220";
        $this->unitEyes="760 ";
    }
    function __clone(){}
}
?>
```

重要的是，IPrototype的这两个具体实现都实现了一个__clone()方法，尽管这个实现仅仅是为语句增加了两个大括号。__clone()方法是PHP的内置方法，这个方法的封装代码完全能够完成这个设计模式所需的工作，所以“实现”只不过是增加一个函数签名：

```
<?php
//FemaleProto.php
include_once('IPrototype.php');
class FemaleProto extends IPrototype
{
    const gender="FEMALE";
    public $fecundity;

    public function __construct()
    {
        $this->eyeColor="red";
        $this->wingBeat="220";
        $this->unitEyes="760 ";
    }
    function __clone(){}
}
?>
```

这两个实现都使用直接量（具体的数字、字符串或布尔值）来赋值。所以，可以用与这些值的偏差来度量变异。

客户

尽管设计模式中这种情况并不少见，不过在原型设计模式中，Client类确实是一个不可缺少的部分。原因在于，尽管要将子类具体实现作为实例的模板，但使用相同的模板克隆实例的具体工作是由Client类完成的。

这个原型的两个具体实现都非常简单，共享变量eyeColor、wingBeat和unitEyes都使用了直接赋值；它们甚至没有获取方法/设置方法。对现在来说，这样是可以的，因为我们的重点是查看类实例的克隆实现如何使用这些属性；首先从具体类实例化$fly1和$fly2，$c1Fly、$c2Fly和$updatedCloneFly则分别是这两个类实例的克隆。

```
<?php
//Client.php
function __autoload($class_name)
```

```
{
    include $class_name . '.php';
}
class Client
{
    //用于直接实例化
    private $fly1;
    private $fly2;

    //用于克隆
    private $c1Fly;
    private $c2Fly;
    private $updatedCloneFly;

    public function __construct()
    {
        //实例化
        $this->fly1=new MaleProto();
        $this->fly2=new FemaleProto();

        //克隆
        $this->c1Fly = clone $this->fly1;

        $this->c2Fly = clone $this->fly2;
        $this->updatedCloneFly = clone $this->fly2;

        //更新克隆
        $this->c1Fly->mated="true";
        $this->c2Fly->fecundity="186";
        $this->updatedCloneFly->eyeColor="purple";
        $this->updatedCloneFly->wingBeat="220";
        $this->updatedCloneFly->unitEyes="750";
        $this->updatedCloneFly->fecundity="92";

        //通过类型提示方法发送
        $this->showFly($this->c1Fly);
        $this->showFly($this->c2Fly);
        $this->showFly($this->updatedCloneFly);
    }

    private function showFly(IPrototype $fly)
    {
        echo "Eye color: " . $fly->eyeColor . "<br/>";
        echo "Wing Beats/second: " . $fly->wingBeat . "<br/>";
        echo "Eye units: " . $fly->unitEyes . "<br/>";
        $genderNow=$fly::gender;
        echo "Gender: " . $genderNow . "<br/>";
        if($genderNow=="FEMALE")
        {
            echo "Number of eggs: " . $fly->fecundity . "<p/>";
        }
        else
        {
            echo "Mated: " . $fly->mated . "<p/>";
        }
    }
```

```
}
$worker=new Client();
?>
```

对于多个类引用，使用PHP __autoload()方法要比include_once方法更容易。通过使用__autoload()方法，不论Client类引用多少个参与者或辅助类，所有类都会自动包含。对于学习设计模式来说，使用__autoload()有一点不及include_once方法：include_once方法会显示所有正在使用的类。在本书后面，会采用__autoload()或include_once方法来包含外部文件中的类，具体选择哪一个技术，这取决于引用的类名对于理解程序有多大用处。（文件名就基于类名。）

下面的输出显示了雄果蝇和雌果蝇具体类（“模板”类）的两个克隆，相对于具体类实例，这两个克隆没有任何改变，第3个有紫色眼睛，而且复眼单元数与模板类不同，由此可以表明这是一个“变异”的克隆。产卵个数在标准偏差范围内，这不算是一种变异。

```
Eye color: red
Wing Beats/second: 220
Eye units: 760
Gender: MALE
Mated: true
Eye color: red
Wing Beats/second: 220
Eye units: 760
Gender: FEMALE
Number of eggs: 186

Eye color: purple
Wing Beats/second: 220
Eye units: 750
Gender: FEMALE
Number of eggs: 92
```

原型模式要依赖客户通过一个克隆过程使用具体原型。在这个设计中，客户是完成克隆的参与者，由于克隆是原型设计中的关键要素，所以客户是一个基本参与者，而不仅仅是一个请求类。

6.5 为原型模式增加OOP

以上给出了一个最简单的例子，强调了参与者与结果之间的关系。为了尽量减少代码，以便更清楚地“看出”结构，这些类并没有实现我们希望的OOP应用中的那种封装。换句话说，模式实现中的参与者都相当“骨感”，这样才有利于看到类之间的关系。例如，在原型模式的这个最简单的实现中，客户能直接改变原型的属性值，也就是说，可以通过简单的赋值改变属性。例如，客户使用以下代码改变变异的眼睛颜色：

```
$this->updatedCloneFly->eyeColor="purple";
```

抽象或具体原型类中都没有提供获取方法/设置方法，也没有使用其他结构来更好地封装这些属性。

6.5.1 现代企业组织

在创建型设计模式方面，现代企业组织就非常适合采用原型实现。现代企业组织往往是复杂而庞大的层次结构，像面向对象编程一样，要为很多共同的特征建模。下面这个原型例子也相当简单，重点在于理解类关系，不过这个例子得到了更好的封装，更符合OOP的要求。图6-3中的类图显示了组织原型的参与者和总体结构。

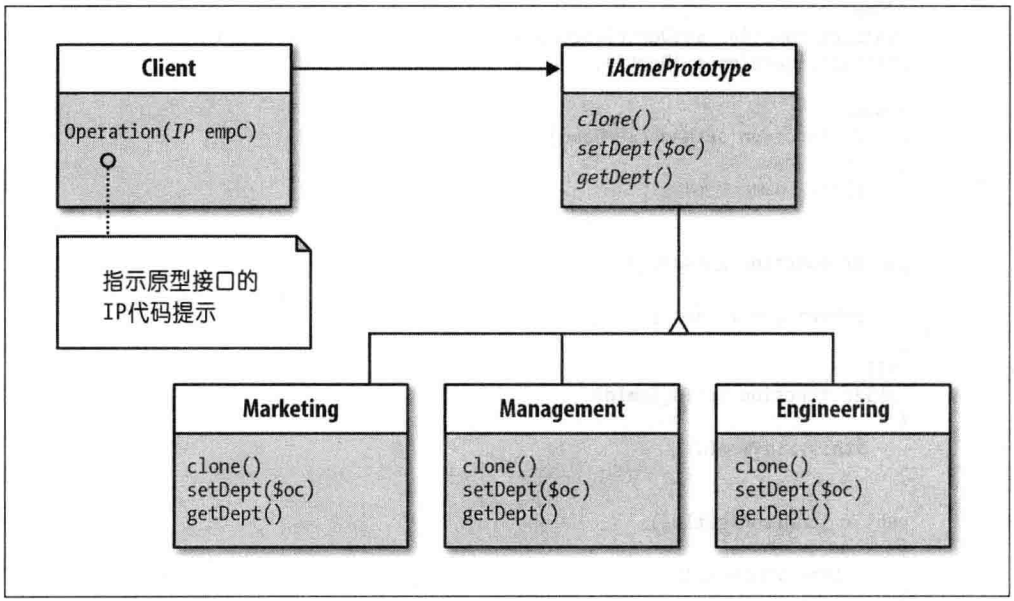


图6-3：组织原型类图

这里描述的软件工程公司是一个典型的现代组织。工程部（Engineering Department）负责创建产品，管理部（Management）处理资源的协调和组织，市场部（Marketing）负责产品的销售、推广和整体营销。

6.5.2 接口中的封装

在这个原型实现中，首先为程序的接口（一个抽象类）增加OOP。与所有原型接口一样，这个接口包含一个抽象克隆操作。另外它还包含一些抽象和具体的获取方法和设置方法。其中有一个抽象获取方法/设置方法对，但要由3个具体原型实现为这个抽象获取

方法/设置方法对提供具体实现。其他获取方法和设置方法分别应用于员工名、ID码和照片等属性。注意所有这些属性都是保护属性（protected），所以尽管具体的获取方法和设置方法有公共可见性，但由于操作中使用的属性具有保护可见性，这就提供了某种程度的封装：

```
<?php
//IAcmePrototype.php
abstract class IAcmePrototype
{
    protected $name;
    protected $id;
    protected $employeePic;
    protected $dept;

    //Dept
    abstract function setDept($orgCode);
    abstract function getDept();

    //Name
    public function setName($emName)
    {
        $this->name=$emName;
    }

    public function getName()
    {
        return $this->name;
    }
    //ID
    public function setId($emId)
    {
        $this->id=$emId;
    }

    public function getId()
    {
        return $this->id;
    }

    //Employee Picture

    public function setPic($ePic)
    {
        $this->employeePic=$ePic;
    }

    public function getPic()
    {
        return $this->employeePic;
    }

abstract function __clone();
}

?>
```

利用这些获取方法和设置方法，所有属性的值都通过继承的保护变量来设置。采用这种设计，扩展类及其实例可以得到更好的封装。

6.5.3 接口实现

3个IAcmePrototype子类都必须实现“dept”(部门)抽象方法以及__clone()方法。类似地，每个具体原型类还包含一个常量UNIT，它提供一个赋值，可以由实例（不论是直接实现还是通过克隆创建）作为标识。首先来看如何构建Marketing类：

```
<?php
//Marketing.php
include_once('IAcmePrototype.php');
class Marketing extends IAcmePrototype
{
    const UNIT="Marketing";
    private $sales="sales";
    private $promotion="promotion";
    private $strategic="strategic planning";

    public function setDept($orgCode)
    {
        switch($orgCode)
        {
            case 101:
            $this->dept=$this->sales;
            break;

            case 102:
            $this->dept=$this->promotion;
            break;

            case 103:
            $this->dept=$this->strategic;
            break;

            default:
            $this->dept="Unrecognized Marketing ";
        }
    }

    public function getDept()
    {
        return $this->dept;
    }

    function __clone(){}
}
?>
```

setDept()方法的实现使用了一个参数。并不是直接输入市场部的部门名，这个方法希望得到一个数字码。如果希望得到从某个类派生的一个对象，可以使用一个类型提示（指定对象/接口类型）来构建这个方法，不过类型提示不允许是标量类型，如int。通

过使用一个switch/case语句，并将参数作为比较变量，这个类限制了3种可接受的情况，或者默认为“Unrecognized Marketing”（未识别的市场部门）。一旦匹配，操作将使用一个有指定赋值的私有变量。同样地，这有助于封装类和设置方法。获取方法（getDept()）则使用同样的私有变量。

另外两个原型实现也类似，不过需要注意，它们在私有变量中分别存储了不同的部门。类似地，这两个原型实现中常量UNIT分别有不同的值：

```
<?php
//Management.php
include_once('IAcmePrototype.php');
class Management extends IAcmePrototype
{
    const UNIT="Management";
    private $research="research";
    private $plan="planning";
    private $operations="operations";

    public function setDept($orgCode)
    {
        switch($orgCode)
        {
            case 201:
            $this->dept=$this->research;
            break;

            case 202:
            $this->dept=$this->plan;
            break;

            case 203:
            $this->dept=$this->operations;
            break;

            default:
            $this->dept="Unrecognized Management";
        }
    }

    public function getDept()
    {
        return $this->dept;
    }
    function __clone(){}
}
?>
```

在这3个具体原型实现中，switch/case语句中期望的值都有所不同。类似地，私有属性的名字和值也有所不同：

```
<?php
//Engineering.php
```

```
include_once('IAcmePrototype.php');
class Engineering extends IAcmePrototype
{
    const UNIT="Engineering";
    private $development="programming";
    private $design="digital artwork";
    private $sysAd="system administration";

    public function setDept($orgCode)
    {
        switch($orgCode)
        {
            case 301:
            $this->dept=$this->development;
            break;

            case 302:
            $this->dept=$this->design;
            break;

            case 303:
            $this->dept=$this->sysAd;
            break;

            default:
            $this->dept="Unrecognized Engineering";
        }
    }

    public function getDept()
    {
        return $this->dept;
    }
    function __clone(){}

}
?>
```

以上这3个具体原型实现分别有其特定用途，不过它们都符合接口，可以创建各个原型实现的一个实例，然后根据需要克隆多个实例。这个工作最后由Client类完成。

6.5.4 组织客户

客户的基本设置非常简单。我们的计划是：分别创建各个具体原型的一个实例，然后按以下列表来克隆各个实例：

- 市场部门实例
  - 市场部门克隆
  - 市场部门克隆
- 管理部门实例

- 管理部门克隆

- 工程部门实例
  - 工程部门克隆
  - 工程部门克隆

将来只使用这些克隆对象。使用获取方法和设置方法将各个特定情况的信息赋给这些克隆对象。以下客户代码显示了这个实现：

```
<?php
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}

class Client
{
    private $market;
    private $manage;
    private $engineer;

    public function __construct()
    {
        $this->makeConProto();

        $Tess=clone $this->market;
        $this->setEmployee($Tess,"Tess Smith",101,"ts101-1234","tess.png");
        $this->showEmployee($Tess);

        $Jacob=clone $this->market;
        $this->setEmployee($Jacob,"Jacob Jones",102,"jj101-2234","jacob.png");
        $this->showEmployee($Jacob);

        $Ricky=clone $this->manage;
        $this->setEmployee($Ricky,"Ricky Rodriguez",203,"rr203-5634","ricky.png");
        $this->showEmployee($Ricky);

        $Olivia=clone $this->engineer;
        $this->setEmployee($Olivia,"Olivia Perez",302,"op301-1278","olivia.png");
        $this->showEmployee($Olivia);

        $John=clone $this->engineer;
        $this->setEmployee($John,"John Jackson",301,"jj302-1454","john.png");
        $this->showEmployee($John);
    }

    private function makeConProto()
    {
        $this->market=new Marketing();
        $this->manage=new Management();
        $this->engineer=new Engineering();
    }
```

```
    private function showEmployee(IAcmePrototype $employeeNow)
    {
        $px=$employeeNow->getPic();
        echo "<img src=$px width='150' height='150'><br/>";
        echo $employeeNow->getName() . "<br/>";
        echo $employeeNow->getDept() . ": " . $employeeNow::UNIT . "<br/>";
        echo $employeeNow->getID() . "<p/>";
    }

    private function setEmployee(IAcmePrototype $employeeNow,$nm,$dp,$id,$px)
    {
        $employeeNow->setName($nm);
        $employeeNow->setDept($dp);
        $employeeNow->setID($id);
        $employeeNow->setPic("pix/$px");
    }
}
$worker = new Client();
?>
```

客户的构造函数类包含3个私有属性，用来分别实例化这3个具体原型类。makeConProto()方法生成必要的实例。

接下来，使用克隆技术创建一个“员工”实例。然后，这个克隆实例向一个设置方法（setEmployee()）发送特定的实例信息，这个设置方法使用IAcmePrototype接口类型提示。不过，需要说明，它只对第一个参数使用类型提示，其他参数都没有类型提示，并不要求它们派生自IAcmePrototype接口。克隆“员工”可以使用IAcmePrototype抽象类的所有设置方法以及具体原型类实现的setDept()方法。

要使用各个员工的数据，Client类可以使用继承的获取方法。图6-4分别显示了5个员工克隆对象的输出。

可以根据需要增加更多的克隆，而且只需要对具体原型类完成一次实例化。使用原型模式时，并不是建立具体类的多个实例，而只需要一个类实例化和多个克隆。

6.5.5 完成修改，增加特性

要记住，最重要（可能也是最基本）的是，设计模式允许开发人员修改和增补程序，而不必一切都从头开始。例如，假设公司总裁决定公司增加一个新的部门，比如研究部门（Research），这会很难吗？一点也不难。Research类可以扩展IAcmePrototype抽象类，然后实现抽象获取方法和设置方法来反映这个研究部门的组织。需要说明，Client类中获取方法和设置方法使用的代码提示指示一个接口，而不是一个抽象类的具体实现。所以，只要增加的单元正确地实现了这个接口，就能顺利地增加到应用中，而不会带来波动，也不需要对程序中的其他参与者进行重构。

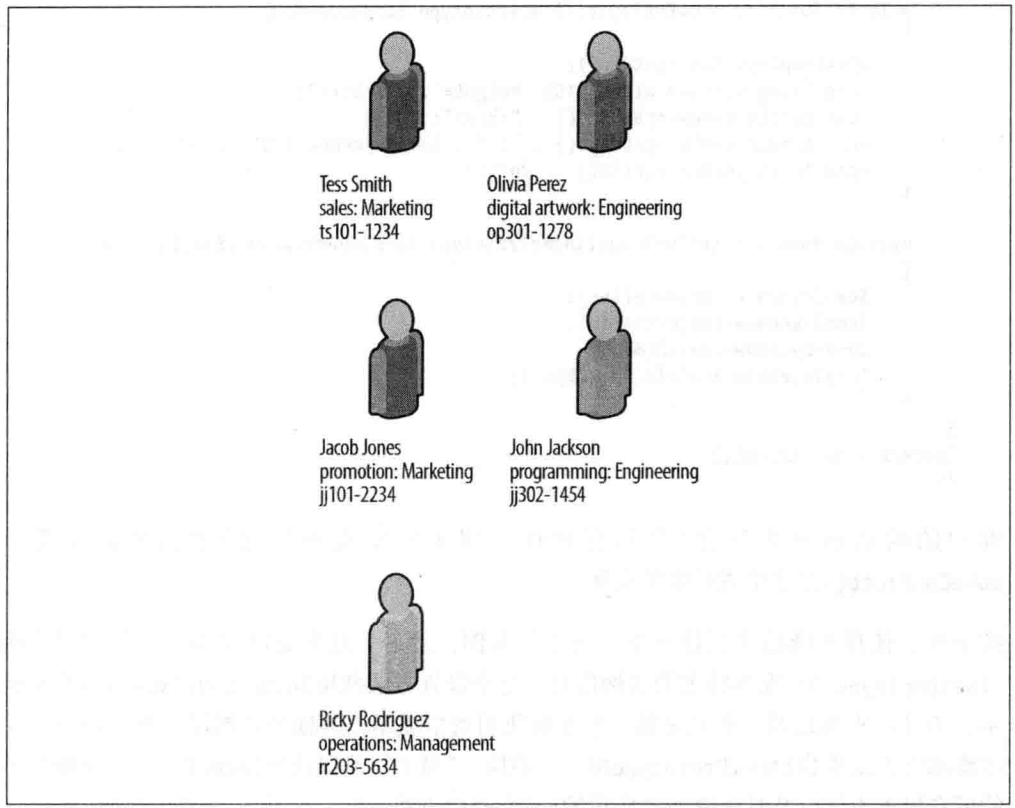


图6-4：组织中的克隆对象

不仅可以增加更多的具体类，还可以很容易地对各个类进行修改，而不会造成破坏。例如，假设这个组织的市场部决定，除了现有的部门外，他们还需要一个特殊的在线市场部。这样一来，`switch/case`操作需要一个新的分支（case），还要有一个新的私有属性（变量）来描述新增的这个部门。这个改变将封装在单独的类中，而不会影响其他参与者。由于这种改变不会带来破坏，所以应用的规模越庞大，这一点就越重要。可以看到，原型设计模式不仅支持一致性，还支持灵活的改变。

6.5.6 动态对象实例化

查看`Client`类时，你可能会想到：一个真正的组织肯定会有更多的员工，把它们统统都硬编码到客户中，这看起来并不是处理这个问题的明智方法。太对了，正是因为这个原因，我们需要考虑如何由数据库、XML文件或者在别处存储的数据动态地创建克隆，而不是依赖客户中的一行代码来创建。

使用数据库中的一条记录时，可以把这个数据传递到PHP程序进行处理。不过，数据并

不是作为具体类的实例来存储，而是作为某种数值或字符串数据。例如，下面的数据取自Client类：

```
$Tess=clone $this->mar;
$this->setEmployee($Tess,"Tess Smith",101,"ts101-1234","tess.png");
$this->showEmployee($Tess);
```

正常情况下，setEmployee()方法参数中的数据来自一个数据库，并发送给客户。第一个参数应当是一个实现了IAcmePrototype接口的对象。所以，现在的问题是，如何根据来自一个数据库的数据动态地创建（克隆）对象？

对象变量

在动态创建方面，PHP看来是最善解人意的语言之一。要在PHP中用一个变量实例化一个类，这个过程相当简单。下面的代码就是利用变量中的值来创建类的一个实例：

```
//Class name = MyClass
$myVar = "MyClass";
$myObj =new $myVar;
```

就这么简单。变量$myObj已经实例化一个MyClass实例。

这个代码等价于以下代码：

```
$myObj = new MyClass();
```

使用这些技术，可以由数据库、数组或者程序可以访问的其他地方的数据动态地创建和克隆对象。

在下面的例子中，假设数据并非来自数组，而是来自于一个数据库。这个原则也同样适用。还需要说明，这里使用了一个接口而不是抽象类：

```
<?php
interface IPrototype
{
    const PROTO="IPrototype";
    function __clone();
}
class DynamicObjectNaming implements IPrototype
{
    const CONCRETE=" [Concrete] DynamicObjectNaming";

    public function __construct()
    {
        echo "This was dynamically created.<br/>";
    }

    public function doWork()
    {
```

```
        echo "<br/>This is the assigned task.<br/>";
    }

    function __clone() {}
}

$employeeData = array('DynamicObjectNaming','Tess','mar', 'John',
  'eng', 'Olivia','man' );
$don=$employeeData[0];
$employeeData[6]=new $don;
echo $employeeData[6]::CONCRETE;
$employeeData[6]->doWork();

$employeeName=$employeeData[5];
$employeeName = clone $employeeData[6];
echo $employeeName->doWork();
echo "This is a clone of " . $employeeName::CONCRETE . "<br/>";
echo "Child of: " . $employeeName::PROTO;
?>
```

运行程序时，会看到以下输出：

```
This was dynamically created.
[Concrete] DynamicObjectNaming
This is the assigned task.

This is the assigned task.
This is a clone of [Concrete]
DynamicObjectNaming
Child of: IPrototype
```

可以注意到，在最开始的实例化中，构造函数会打印以下结果：“This was dynamically created。”（这是动态创建的对象）这是因为实例化启动了构造函数中的操作。但在克隆过程中，克隆对象并没有启动构造函数。不过，克隆可以使用由类实例化生成的所有赋值，这些值会传递到克隆对象。

6.6 PHP世界中的原型

由于PHP是一个服务器端语言，也是与MySQL数据库交互的一个重要工具，所以原型设计模式尤其适用。并不是为数据库中的每一个元素分别创建对象，PHP可以使用原型模式创建具体类的一个实例，然后利用数据库中的数据克隆其余的实例（记录）。

了解克隆过程之后，与直接实例化类对象的过程相比，你可能会问：“这有什么区别？”换句话说：为什么克隆比直接实例化类对象需要的资源少？它们的区别并不能直接看到。一个对象通过克隆创建实例时，它不会启动构造函数。在DynamicObjectNaming类应用中，你已经看到这样一个例子，直接实例化会启动构造函数，而克隆没有启动构造函数。克隆能得到原始类的所有属性，甚至还包含父接口的属

性，另外还继承了传递到实例化对象的所有值。构造函数生成的所有值以及存储在对象属性中的值都会成为克隆对象的一部分。所以没有必要返回构造函数。如果发现你的克隆对象确实需要访问构造函数生成的值但又无法访问，这说明需要对类进行重构，使实例能拥有它们需要的一切信息，而且可以把这些数据传递给克隆对象。

总的说来，原型模式在很多不同类型的PHP项目中都很适用，如果解决一个问题需要用到创建型模式，都可以使用原型模式。这一章给出了几个不同的例子，不过既然你已经知道原型模式是什么，而且很清楚如何使用这种模式，不妨把目光放长远些，努力寻找机会来利用这种模式。这样一来，你将节省大量开发时间，而且可以改善你的设计，从而能很好地应对将来可能出现的改变。

第3部分

结构型设计模式

我们观察到的材料体和材料力
无非是空间结构中的形状和变化。
——埃尔温·薛定谔

生存单元需要能量不仅是为了完成它的所有功能，
同时也是为了维护它的自身结构。
——阿尔伯特·圣捷尔吉

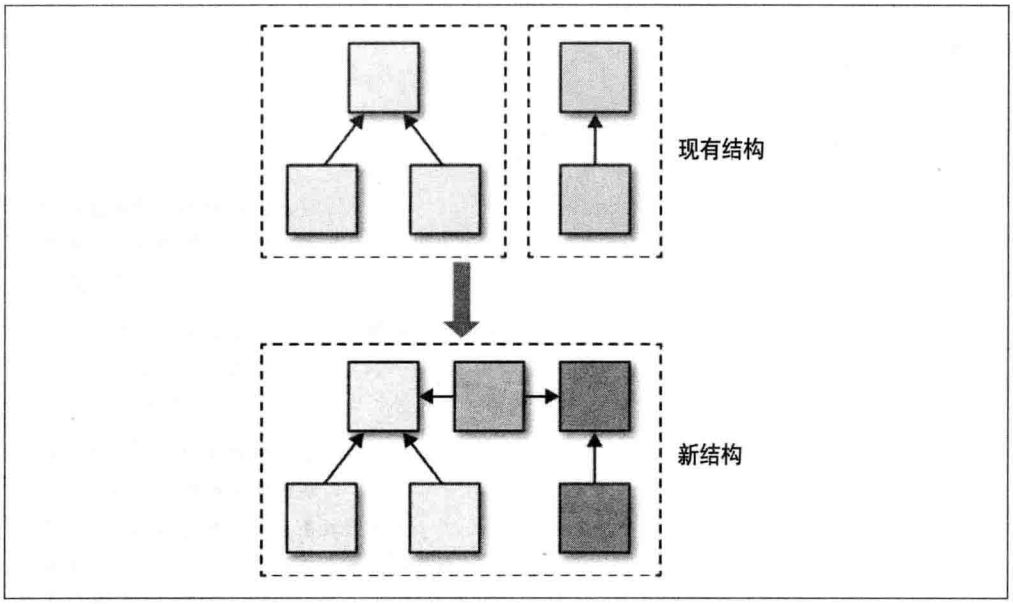
对于材料世界中最为重要的概念，
从太阳系的结构直到人类起源，
大多数教条主义宗教通常都有很多谬误，
而且在这方面总展示出超乎寻常的“天分”。
——乔治·G·辛普森

结构型设计模式研究的是如何组合对象和类来构成更大的结构。在类结构型设计中，要通过多个接口来创建新结构。一个类可能继承多个父类来创建一个新的结构。更常见的是对象结构结合不同的对象来形成新的结构。Gamma，Helm，Johnson和Vlissides将下面7种模式称为结构型模式：

- 适配器模式（Adapter）（类和对象）
- 桥接模式（Bridge）
- 组合模式（Composite）
- 装饰器模式（Decorator）
- 外观模式（Pacade）

- 享元模式（Flyweight）
- 代理模式（Proxy）

对于这7种模式，第3部分将详细讨论两个版本的适配器模式（类适配器和对象适配器）和装饰器模式。如果需要通过适配（使用多重继承或组合）来结合两个不兼容的系统，适配器就非常重要。但有一个棘手的问题需要处理：PHP不支持多重继承，不过可以看到，PHP提供了一种变通方法来实现类适配器模式。另外，组合不仅适用于对象适配器模式，也适用于装饰器模式。图III-1提供了结构型设计模式的一个可视化表示。



图III-1：结构型模式强调由现有结构创建新结构

结构型设计模式的重点是创建新结构而不破坏原有的结构。在这个基础上，结构型模式可以保持并提升松耦合标准以实现重用和灵活改变。

第7章

适配器模式

能生存下来的物种,不是最强壮的,也不是最聪明的,而是能因应环境变化的物种。

——查尔斯·达尔文

智力的评测要看其变革能力。

——阿尔伯特·爱因斯坦

科学的本源就在于对一些确定的经验领域调整认知。

——恩斯特·马赫

7.1 什么是适配器模式

这一章算是“买一送一”：我们会同时介绍对象适配器和类适配器。这一章的内容很多，不过其中最有意思的是查看使用继承和使用组合的差别。类适配器设计模式使用继承，如图7-1中的类图所示。

从这个类图可以看到，这个模式实现中一个类有双重继承。要知道，双重继承在PHP 5中是不允许的，不过实现双重继承有一些替代方法，可以结合继承和实现来正确地实现这种模式。

你已经知道，设计模式有一个重要格言：组合优先于继承。再来看第二个适配器模式（图7-2），Adapter参与者使用组合来包含Adaptee的一个引用，这里没有使用继承。

一般来说，组合要优先于继承，因为参与者之间的绑定更宽松，在重用、结构和修改等方面有很多优点，这与使用继承不同，继承具体类或者所继承的类中包含已实现的方法时，存在一种紧密绑定，使用组合就没有这种紧密绑定的缺点。

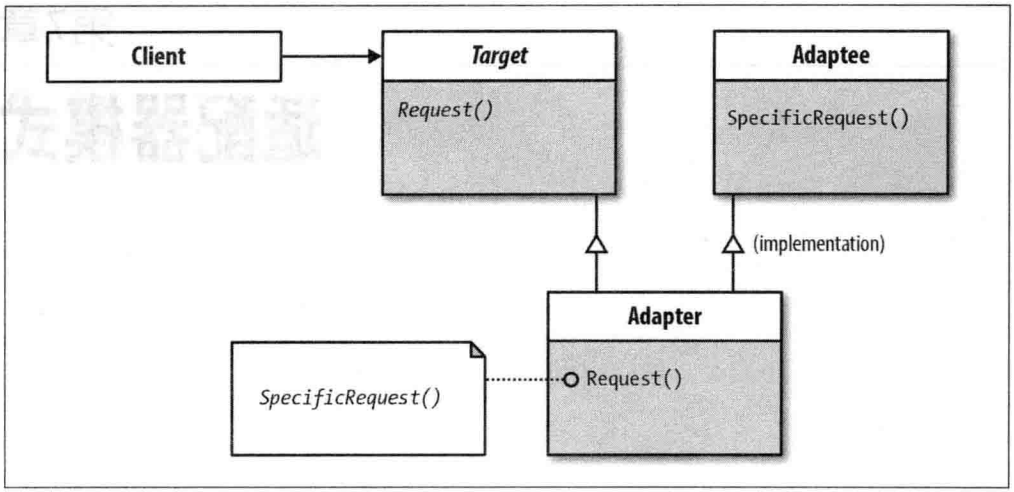


图7-1：使用继承的适配器类图

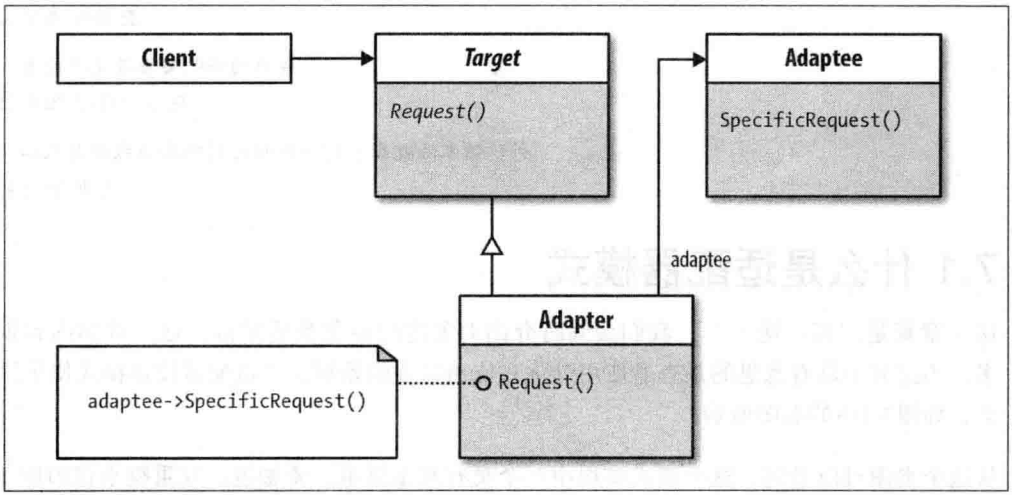


图7-2：使用组合的适配器类图

7.2 何时使用适配器模式

适配器很容易理解；我们一直都在使用适配器。大多数家庭都有手机转接器，用来为移动电话充电，这就是一种适配器。如果只有USB连接头，就无法将移动电话插到标准插座上。实际上，必须使用一个适配器，一端接USB插头，另一端接插座。图7-3显示了为移动电话或电脑充电的一个典型的适配器。

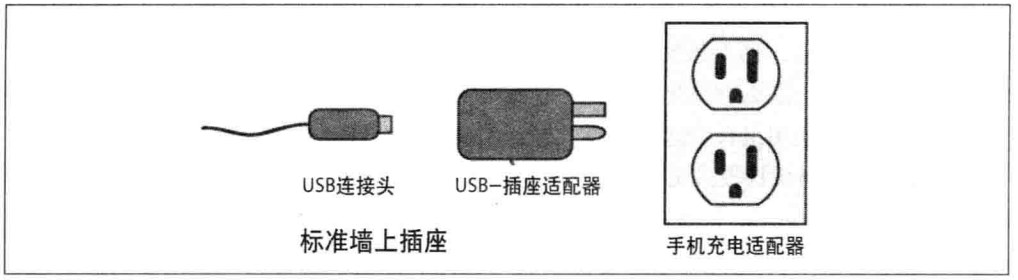


图7-3：适配器在电气领域很常见

如果有人问：“什么时候使用适配器模式？”从某种程度上讲，这个问题与“什么时候使用某种适配器”是一样的，会得到类似的答案。例如，如果有人问如何为移动电话充电，你可能会告诉他：需要用到一个适配器，使USB连接头和墙上插座相互兼容。

分解这个问题可以得到：

- USB连接头——不变
- 标准墙上插座——不变

当然，你可以拿出你的电气工具，改装USB连接头，或者重新安装墙上插座，不过这样会带来很多额外的工作，而且可能会把连接头或插座弄坏（甚至两个都弄坏！）。或者，你也可以找一个适配器。最可取的做法就是找一个适配器。软件开发也是如此。

假设你和你的同事已经开发了一个PHP程序，可以很容易地为客户创建定制的桌面web设计。这个系统可以处理所有工作，包括网站的外观以及一个MySQL数据库的处理。对于笔记本电脑和台式机的屏幕来说，很适合采用一种多栏设计以及相应的用户体验设计（User experience design，UX）。实际上，这样得到的系统非常类似于用PHP创建的一个桌面CMS。

某一天，一些客户要求你为这个网站增加一个移动版本。设计人员告诉你移动平台更适合采用水平设计，所以你必须重新设计外观。相应地，整个用户体验设计也必须修改。设计人员开发出一个移动模块后，你发现这个模块与你原先的桌面模块并不兼容。图7-4描述了这个问题。

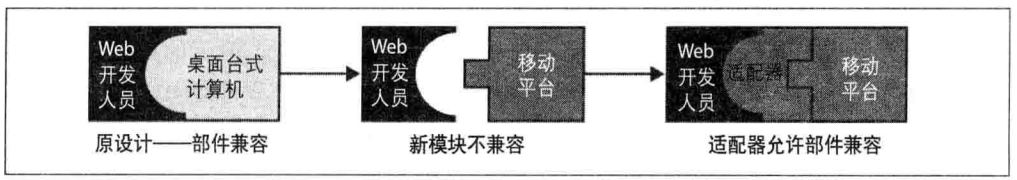


图7-4：与移动电话适配器一样，适配器设计模式也遵循同样的原则

我们不想改变web开发模块（这个系统原本工作得很好，其中包含链接web页面和数据库的所有类和操作），也不想改变新开发的移动模块（毕竟，这是我们为这个移动网站辛辛苦苦创建的），那么你该怎么做呢？答案很简单，就像为手机充电一样，可以使用一个适配器。在“使用组合的适配器模式”一节中，你会找到这样一个例子，从中可以了解如何使用适配器设计模式更新系统来包含一个新模块。

适配器模式还有很多其他的应用。很多专业的web开发人员可能会使用开发商提供的一些特殊的加速器、UI或其他增强模块，为了与现有软件一同使用，他们通常需要某种适配器。类似地，如果两个不同的开发小组开发了不兼容的模块，也可以使用一个适配器，与要求某个小组重新开发模块相比，使用适配器通常更节省时间。（当然，最好建议这些开发小组将来能更好地沟通，不要再开发不兼容的模块！）

7.3 使用继承的适配器模式

类适配器设计模式很简单，不过与对象适配器模式相比，类适配器模式的灵活性稍弱。类适配器模式更简单的原因在于，适配器（Adapter）会从被适配者（Adaptee）继承功能，所以适配器模式中需要重新编写的代码比较少。当然，由于给定了一个将由适配器（Adapter）继承的具体被适配者（Adaptee），这种绑定很紧密，所以使用类适配器模式创建应用时，必须非常清楚将在哪里发生适配。

由于类适配器包含双重继承，如图7-1中所示，下面的PHP例子首先用了一点技巧。实际上，PHP并不支持双重继承（别担心，Java也不支持双重继承）。你会发现很多设计精巧的例子都展示了PHP可以模拟双重继承，不过最重要的是，PHP本身可以很好地处理接口实现和类继承的组合。下面的语句展示了一个正确的结构，这里不仅继承了一个类，同时还实现了一个接口：

```
class ChildClass extends ParentClass implements ISomeInterface
```

所以，实现类适配器模式时，参与者必须包括一个PHP接口。

最简单的类适配器例子：货币兑换

我们想找一个既有用又简短的例子，所以想到了货币兑换/转换器。假设有一个企业网站在同时销售软件服务和软件产品，目前，所有交易都在美国进行，所以完全可以用美元来完成所有计算。现在开发人员希望能有一个转换器能处理美元与欧元的兑换，而不要改变原来按美元计算交易金额的类。通过增加一个适配器，现在程序既可以用美元计算也可以用欧元计算。

首先，假设有一个很好的类DollarCalc，它能累加所购买服务和产品的价格，然后返回总金额：

```
<?php
//DollarCalc.php
class DollarCalc
{
    private $dollar;
    private $product;
    private $service;
    public $rate=1;

    public function requestCalc($productNow,$serviceNow)
    {
        $this->product=$productNow;
        $this->service=$serviceNow;
        $this->dollar=$this->product + $this->service;
        return $this->requestTotal();
    }

    public function requestTotal()
    {
        $this->dollar*=$this->rate;
        return $this->dollar;
    }
}
?>
```

查看这个类，可以看到其中有一个属性$rate，requestTotal()方法使用$rate来计算一次交易的金额。在这个版本中，这个值设置为1，另外将两个参数变量的值乘以1，实际上总金额无需再乘以兑换率。不过，如果要为客户提供折扣或者要增加额外服务或产品的附加费，$rate变量会很方便。这个类并不是类适配器模式的一部分，不过这是一个起点。

加入欧元

客户宣布她的公司决定向欧洲拓展，希望进入卢森堡市场小试牛刀。你很快了解到卢森堡属于欧元区，所以需要开发一个应用，能够用欧元完成同样的计算。你希望这个欧元计算能像DollarCalc一样，所要做的就是改变变量名。

```
<?php
//EuroCalc.php
class EuroCalc
{
    private $euro;
    private $product;
    private $service;
    public $rate=1;

    public function requestCalc($productNow,$serviceNow)
    {
```

```
            $this->product=$productNow;
            $this->service=$serviceNow;
            $this->euro=$this->product + $this->service;
            return $this->requestTotal();
        }

        public function requestTotal()
        {
            $this->euro*=$this->rate;
            return $this->euro;
        }
    }
    ?>
```

接下来，再把应用的其余部分插入到EuroCalc类中，你已经做好了准备。不过，你知道客户的所有数据都是按美元计算的。换句话说，如果不重新开发整个程序，就无法在系统中“插入”这个欧元计算。但你不想这么做。为了加入EuroCalc，你需要一个适配器。

创建一个欧元适配器

我们先停一下，理一理思路。图7-5显示了一个类图，这里使用了这个类适配器模式实现中的类名。

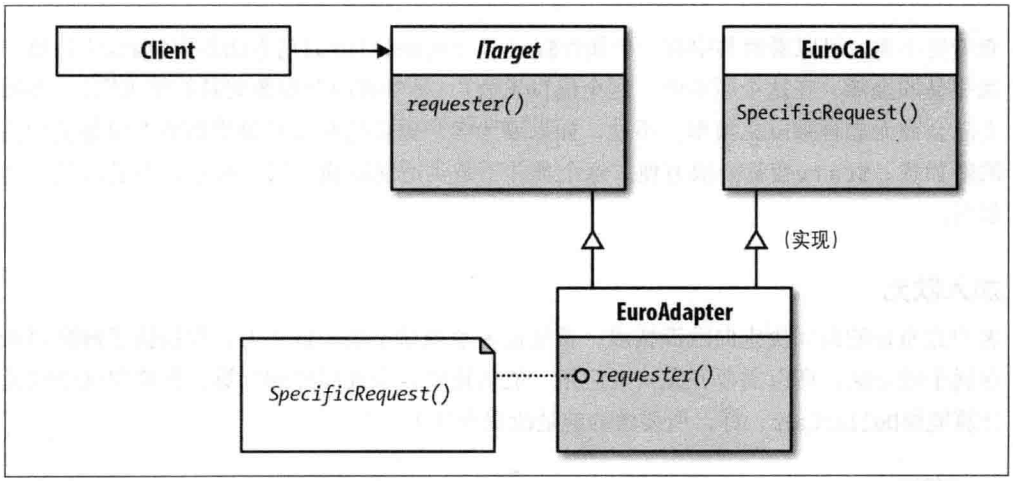


图7-5：使用继承实现的适配器类图

就像找一个适配器来适应欧洲的插座一样，可以创建一个适配器，使你的系统能够使用欧元。幸运的是，类适配器模式正是为这种情况设计的。首先需要创建一个接口。在这个类图中，这个接口名为ITarget。它只有一个方法requester()。requester()是一个抽象方法，要由接口的具体实现来实现这个方法：

```
<?php
//ITarget.php
//Target
interface ITarget
{
    function requester();
}
?>
```

现在开发人员可以实现requester()方法，请求欧元而不是美元。

在使用继承的适配器设计模式中，适配器（Adapter）参与者既实现ITarget接口，还实现了具体类EuroCalc。创建EuroAdapter不需要做太多工作。大部分工作已经在EuroCalc类中完成，这是一个具体类。现在要做的就是实现requester()方法，使它能把美元值转换为欧元值：

```
<?php
//EuroAdapter.php
//Adapter
include_once('EuroCalc.php');
include_once('ITarget.php');
class EuroAdapter extends EuroCalc implements ITarget
{
    public function __construct()
    {
        $this->requester();
    }

    function requester()
    {
        $this->rate=.8111;
        return $this->rate;
    }
}
?>
```

类适配器模式中，一个具体类会继承另一个具体类，有这种结构的设计模式很少见。大多数设计模式中，几乎都是继承一个抽象类，并由具体类根据需要实现其抽象方法和属性。换句话说，一般谈到继承时，都是具体类继承抽象类。

由于既实现了一个接口又扩展了一个类，所以EuroAdapter类同时拥有该接口和具体类的接口。通过使用requester()方法，EuroAdapter类可以设置rate值（兑换率），从而能使用被适配者的功能，而无需做任何改变。

下面来看具体是如何工作的，Client类从EuroAdapter和DollarCalc类发出请求。可以看到，原来的DollarCalc仍能很好地工作，不过它没有ITarget接口。EuroAdapter类同时实现了接口和具体类，通过使用类型提示，可以保证与其接口一致：

```
<?php
//Client.php
//Client
include_once('EuroAdapter.php');
include_once('DollarCalc.php');

class Client
{
    private $requestNow;
    private $dollarRequest;

    public function __construct()
    {
        $this->requestNow=new EuroAdapter();
        $this->dollarRequest=new DollarCalc();
        $euro="&#8364;";
        echo "Euros: $euro" . $this->makeAdapterRequest($this->requestNow) .
          "<br/>";
        echo "Dollars: $" . $this->makeDollarRequest($this->dollarRequest);

    }

    private function makeAdapterRequest(ITarget $req)
    {
        return $req->requestCalc(40,50);
    }

    private function makeDollarRequest(DollarCalc $req)
    {
        return $req->requestCalc(40,50);
    }
}

$worker=new Client();
?>
```

由输出可以看到，现在美元或欧元都可以处理，这要归功于适配器模式：

```
Euros: €72.999
Dollars: $90
```

这个计算很简单，不过对于更为复杂的计算，继承要提供建立类适配器的Target接口的必要接口和具体实现。

7.4 使用组合的适配器模式

对象适配器模式使用组合而不是继承，不过它也会完成同样的目标。通过比较这两个版本的适配器模式，可以看出它们各自的优缺点。采用类适配器模式时，适配器可以继承它需要的大多数功能，只是通过接口稍做调整。在对象适配器模式中，适配器（Adapter）参与者使用被适配者（Adaptee），并实现Target接口。在类适配器模式中，适配器（Adapter）则是一个被适配者（Adaptee），并实现Target接口。

7.4.1 从桌面环境转向移动环境

PHP程序员经常会遇到这样一个问题：需要适应移动环境而做出调整。不久之前，你可能只需要考虑提供一个网站来适应多种不同的桌面环境。大多数桌面环境都使用一个布局，再由设计人员让它更美观。对于移动设备，设计人员和开发人员不仅需要重新考虑桌面和移动环境中页面显示的设计元素，还要考虑如何从一个环境切换到另一个环境。

这个例子使用了对象适配器模式，首先给出一个简单的PHP类，它提供一个简单的页面，其中包含文本和图片。对于文本，这个例子使用了一个占位文本（“lorem ipsum”[译注1]）作为填充内容（存储在一个文本文件中）。CSS也很简单，使用一个很基本的两栏设计，文本图片关系为左边显示文本，右边显示图片。CSS存储为一个单独的CSS文件。

桌面

要了解组合适配器，首先来看原来的类（它将需要一个适配器）。这个类使用了一个简单但很宽松的接口：

```
<?php
interface IFormat
{
    public function formatCSS();
    public function formatGraphics();
    public function horizontalLayout();
}
?>
```

它支持多个CSS和图片选择，不过其中一个方法指示一种水平布局，我们知道，这种布局并不适用于小的移动设备。这个接口的实现相当简单，下面给出这个接口实现（Desktop类）：

```
<?php
include_once("IFormat.php");
class Desktop implements IFormat
{
    private $head="<!doctype html><html><head>";
    private $headClose="<meta charset='UTF-8'>
    <title>Desktop</title></head><body>";

    private $cap="</body></html>";
    private $sampleText;
    public function formatCSS()
    {
        echo $this->head;
        echo "<link rel='stylesheet' href='desktop.css'>";
```

译注1：lorem ipsum是一篇常用于排版设计的拉丁文文章。

```
        echo $this->headClose;
        echo "<h1>Hello, Everyone!</h1>";
    }
    public function formatGraphics()
    {
        echo "<img class='pixRight' src='pix/fallRiver720.png' width='720'
            height='480' alt='river'>";
    }

    public function horizontalLayout()
    {
        $textFile = "text/lorem.txt";
        $openText = fopen($textFile, 'r');
        $textInfo = fread($openText, filesize($textFile));
        fclose($openText);
        $this->sampleText=$textInfo;
        echo "<div>" . $this->sampleText . "</div>";
        echo "<p/><div>" . $this->sampleText . "</div>";
    }

    public function closeHTML()
    {
        echo $this->cap;
    }
}
?>
```

这个类中的大多数代码都是在完成HTML页面的格式化，以便浏览。文本、图像以及页眉都是最简单的例子。CSS提供了一个样式表，用来设置调色板、体文本、页眉、“第二列”（以便将图像放在文本右边）以及图像周围的边距（也可以使用更为精细的桌面CSS设计，你可能想把它换成你自己的设计）：

```
@charset "UTF-8";
//desktop.css
/* CSS Document */
/*DDDCC5,958976,611427,1D2326,6A6A61*/
@media only screen and (min-device-width : 800px) {
}
body
{
    font-family:Verdana, Geneva, sans-serif;
    color:#1D2326;
    font-size:12px;
    background-color:#DDDCC5;
}

h1
{
    font-family:"Arial Black", Gadget, sans-serif;
    font-size:24px;
    color:#611427;
}
```

```
.pixRight
{
    float:right; margin: 0px 0px 5px 5px;
}

image
{
    padding: 10px 10px 10px 0px;
}
```

这个CSS保存在一个名为*desktop.css*的文件中，通过PHP生成的HTML `<link>`标记来调用。

图7-6显示了PHP输出。

图7-6：简单的两栏桌面设计

在图7-6中可以看到，这个布局对于小的移动设备来说太宽了。所以我们的目标是仍采用同样的内容，但调整为一种移动设计。

调整为移动设计。首先来看图7-7，这里显示了创建移动网站时要使用的具体类名和操作。我增加了`Mobile`类原来的接口（`IMobileFormat`），不过，由于使用了`MobileAdapter`，它要实现`IFormat`接口。

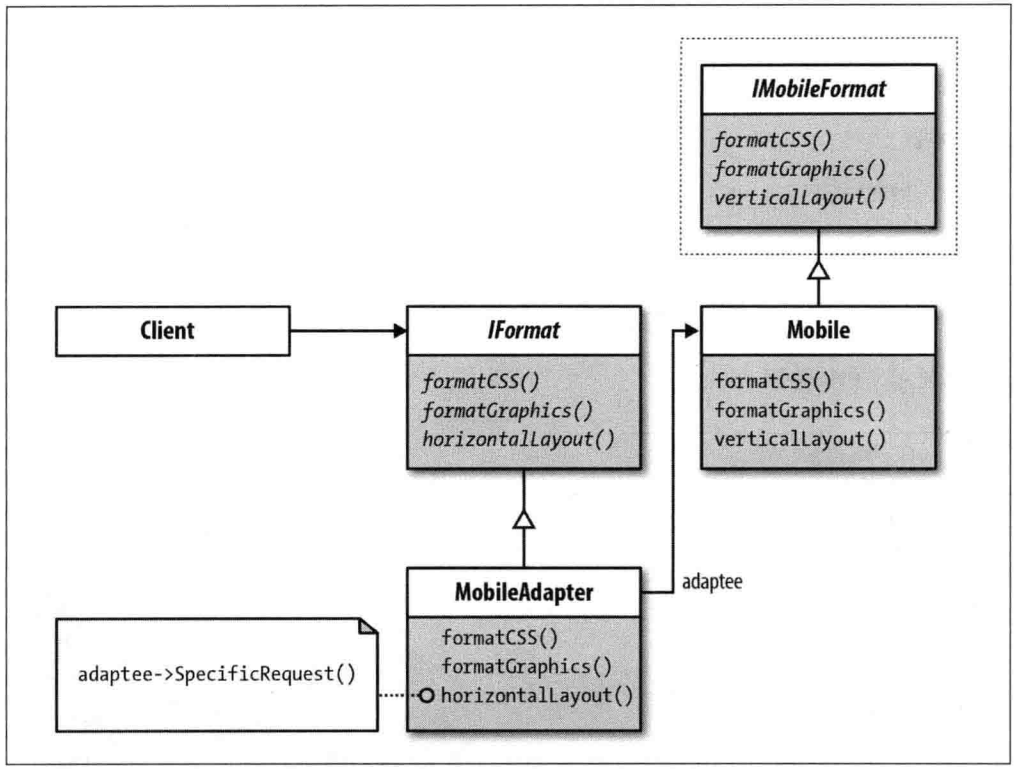


图7-7：对象适配器类图中的MobileAdapter

查看IMobileFormat和IFormat接口时，可以看到它们是不兼容的：一个包含方法horizontalLayout()，而另一个包含verticalLayout()方法。不过，通过使用MobileAdapter（适配器），它继承了Desktop类原来使用的IFormat（Target），现在Mobile（Adaptee）类可以以兼容的方式与IFormat接口的其他实现交互。

适配器模式也称为包装器模式，这一点并不奇怪。适配器（MobileAdapter）参与者“包装了”被适配者（Mobile）参与者，使被适配者可以使用同样的接口。要看包装过程是如何工作的，首先来看IMobileFormat接口和Mobile类，Mobile类与Desktop类几乎完全相同，只不过它有一个不同的接口：

```
<?php
interface IMobileFormat
{
    public function formatCSS();
    public function formatGraphics();
    public function verticalLayout();
}
?>
```

要记住，这个过程的目的是让两个不兼容的接口合作。它们的差别很小，不过你要知道，从桌面设计转换到移动设计时，最主要的区别是：桌面设计可以采用水平的多栏布局，而移动设计要使用垂直布局。下面的Mobile类实现了IMobileFormat（它看起来与Desktop类几乎完全相同）：

```
<?php
include_once('IMobileFormat.php');
class Mobile implements IMobileFormat
{
    private $head="<!doctype html><html><head>";
    private $headClose="<meta charset='UTF-8'>
    <title>Mobile</title></head><body>";

    private $cap="</body></html>";
    private $sampleText;
    public function formatCSS()
    {
        echo $this->head;
        echo "<link rel='stylesheet' href='mobile.css'>";
        echo $this->headClose;
        echo "<h1>Hello, Everyone!</h1>";
    }
    public function formatGraphics()
    {
        echo "<img src='pix/fallRiver960.png' width=device-width
           height=device-height alt='river'>";
    }

    public function verticalLayout()
    {
        $textFile = "text/lorem.txt";
        $openText = fopen($textFile, 'r');
        $textInfo = fread($openText, filesize($textFile));
        fclose($openText);
        $this->sampleText=$textInfo;
        echo "<p/><div>" . $this->sampleText . "</div>";
        echo "<p/><div>" . $this->sampleText . "</div>";
    }

    public function closeHTML()
    {
        echo $this->cap;
    }
}
?>
```

前面已经提到，它与Desktop类非常相似，不过图片的设置不同，而且提供了一个不同的图片。它还调用了另一个不同的CSS文件。这个CSS文件包含一个@media语句，提供了不同的分辨率选择：

```
@charset "UTF-8";
/* CSS Document */
/*DDDCC5,958976,611427,1D2326,6A6A61*/
```

```
@media only screen and (min-device-width : 640px) and (max-device-width : 960px)
{
    img { max-width: 100%; }
}

body
{
    font-family:Verdana, Geneva, sans-serif;
    color:#1D2326;
    font-size:24px;
    background-color:#DDDCC5;
}

h1
{
    font-family:"Arial Black", Gadget, sans-serif;
    font-size:48px;
    color:#611427;
}

image
{
    padding: 5px 5px 5px 0px;
}
```

尽管这个CSS看起来很重要，但在这里它只用于对移动设计的页面外观完成格式化。设计人员还可以做更多工作。重要的是，不论是桌面设计还是移动设计，负责页面外观的不同的类（Desktop类和Mobile类）要协同工作。下面来看关键的适配器（Adapter）参与者，它将Desktop类和Mobile类结合在一起：

```
<?php
include_once("IFormat.php");
include_once("Mobile.php");

class MobileAdapter implements IFormat
{
    private $mobile;

    public function __construct(IMobileFormat $mobileNow)
    {
        $this->mobile = $mobileNow;
    }

    public function formatCSS()
    {
        $this->mobile->formatCSS();
    }

    public function formatGraphics()
    {
        $this->mobile->formatGraphics();
    }

    public function horizontalLayout()
    {
```

```
            $this->mobile->verticalLayout();
        }
    }
    ?>
```

可以看到，MobileAdapter实例化时要提供一个Mobile对象实例。还要注意，类型提示中使用了IMobileFormat，来确保参数是一个Mobile对象。有意思的是，Adapter参与者通过实现horizontalLayout()方法来包含Mobile对象的verticalLayout()方法。实际上，所有MobileAdapter方法都包装了一个Mobile方法。碰巧的是，适配器参与者中的一个方法并不在适配器接口中（verticalLayout()）。它们可能完全不同，适配器只是把它们包装在适配器接口（IFormat）的某一个方法中。

Client类作为参与者

最后一步是使用MobileAdapter类启动应用。要记住，Client类是这个设计模式中不可缺少的一部分，尽管它只是做出请求，但其请求方式要与适配器模式的目标和设计相一致：

```
<?php
include_once('Mobile.php');
include_once('MobileAdapter.php');
class Client
{
    private $mobile;
    private $mobileAdapter;

    public function __construct()
    {
        $this->mobile = new Mobile();
        $this->mobileAdapter = new MobileAdapter($this->mobile);
        $this->mobileAdapter->formatCSS();
        $this->mobileAdapter->formatGraphics();
        $this->mobileAdapter->horizontalLayout();
        $this->mobile->closeHTML();
    }
}

$worker=new Client();

?>
```

适配器模式中的Client类必须包装Adaptee（Mobile）的一个实例，以便集成到Adapter本身。实例化Adapter时，Client使用Adaptee作为参数来完成Adapter的实例化。所以客户必须首先创建一个Adaptee对象（new Mobile()），然后创建一个Adapter实例（(new MobileAdapter($this->mobile)）。

Client类的大多数请求都通过MobileAdapter实例发出。不过，在这个代码的最后，它使

用了Mobile类的实例。由于应用不需要特殊的closeHTML()方法实现，Client直接调用了Mobile实例的这个方法。

图7-8显示了为一个移动设备配置相同内容时得到的结果。在这个例子中，所使用的移动设备是一个iPhone。

图7-8：为移动设备调整为单栏布局

关于如何设计和配置移动web应用，有很多相关的书和资料，这些设计都可以采用适配器模式。关键在于，适配器模式能够以一个桌面设计为起点，选择并使用适合移动设备的不同设计，而不会破坏原来的桌面设计实现。

桌面设计和移动设计：移动先行

尽管这个例子是从桌面设计开始，然后使用一个适配器模式来创建移动设计，不过移动开发人员和设计人员建议：更好的计划是从一个移动设计作为起点，再创建一个适配器来创建桌面设计。这样做的原因在于，对于移动环境，需要从最基本的内容开始，以后要为更大的浏览环境开发和设计时再增加必要的补充内容。（见Lyza Danger Gardner和Jason Grigsby所著的《Head First Mobile Web》[O'Reilly]）在这本书中，第14章介绍了如何使用PHP设计模式构建一个内容管理系统（content management system，CMS），它能够为不同的移动和桌面环境发布相同的信息。PHP所具有的一个强大特性就是能够区分不同的设备，并动态地提供适当的信息。

7.4.2 适配器和变化

PHP程序员要时刻面对变化。不同版本的PHP会有变化，可能增加新的功能，另外还可能取消一些功能。而且，随着PHP的大大小小的变化，MySQL也在改变。例如，`mysql`扩展包升级为`mysqli`，PHP开发人员需要相应调整，要改为使用`mysqli`中的新API。这里适合采用适配器模式吗？可能不适合。适配器可能适用，也可能不适用，这取决于你的程序如何配置。当然可以重写所有连接和交互代码，不过这可不是适配器模式的本意。这就像是重新安装USB连接头，想把它插进标准的墙上插座一样。不过，如果所有原来的`mysql`代码都在模块中，你可以修改这个模块（类），换入一个有相同接口的新模块，只是要使用`mysqli`而不是`mysql`。我不认为交换等同于适配器，不过道理是一样的。在适配器模式中，原来的代码没有任何改变。有变化的只是适配器。

如果需要结合使用两个不兼容的接口，这种情况下，适配器模式最适用。适配器可以完成接口的“联姻”。可以把适配器看作是一个婚姻顾问；通过创建一个公共接口来克服双方的差异。利用这种设计模式，可以促成二者的合作，而避免完全重写某一部分。

第8章

装饰器设计模式

在大多数男人和几乎所有女人看来，
魅力就是一种装饰。
——爱德华·摩根·福斯特

这不只是一般的糟糕，
真是夸张的糟糕，
简直糟糕得掉渣。
——多罗茜·帕克

为门庭增添光彩的是来做客的朋友。
——拉尔夫·沃尔多·爱默生

她很清楚装饰是很重要的建筑秘诀，
却从来没有俯就去建造那些装饰物。
——安东尼·特罗洛普

8.1 什么是装饰器模式

作为一种结构型模式，装饰器（Decorator）模式就是对一个已有的结构增加“装饰”。对于适配器模式，为现有结构增加的是一个适配器类，用来处理不兼容的接口。装饰器模式会向现有对象增加对象。装饰器也称为包装器（类似于适配器），Decorator参与者用具体组件包装Component参与者。图8-1显示了这个类图，不过在详细介绍这个类图之前，需要先考虑几个细节问题。

首先，有些设计模式包含一个抽象类，而且该抽象类还继承了另一个抽象类，这种设计模式为数不多，而装饰器模式就是其中之一。从图8-1可以看到，通过一个环状聚合将Decorator连接到Component参与者。（“四人帮”给出的类图中使用的是直线——参见

Freemans的《Head First Design Patterns》的第91页，与直线相比，这里的环线看起来更像“包装”。）可以称为Decorator装饰器“包装”了Component。

这个设计既要创建要装饰的组件，还要创建装饰器。可以把Component参与者想成是一个要装修的空房间，要配上家具和地毯——家具和地毯就是具体装饰器。还要注意，所有参与者通过Component共享一个公共的接口。

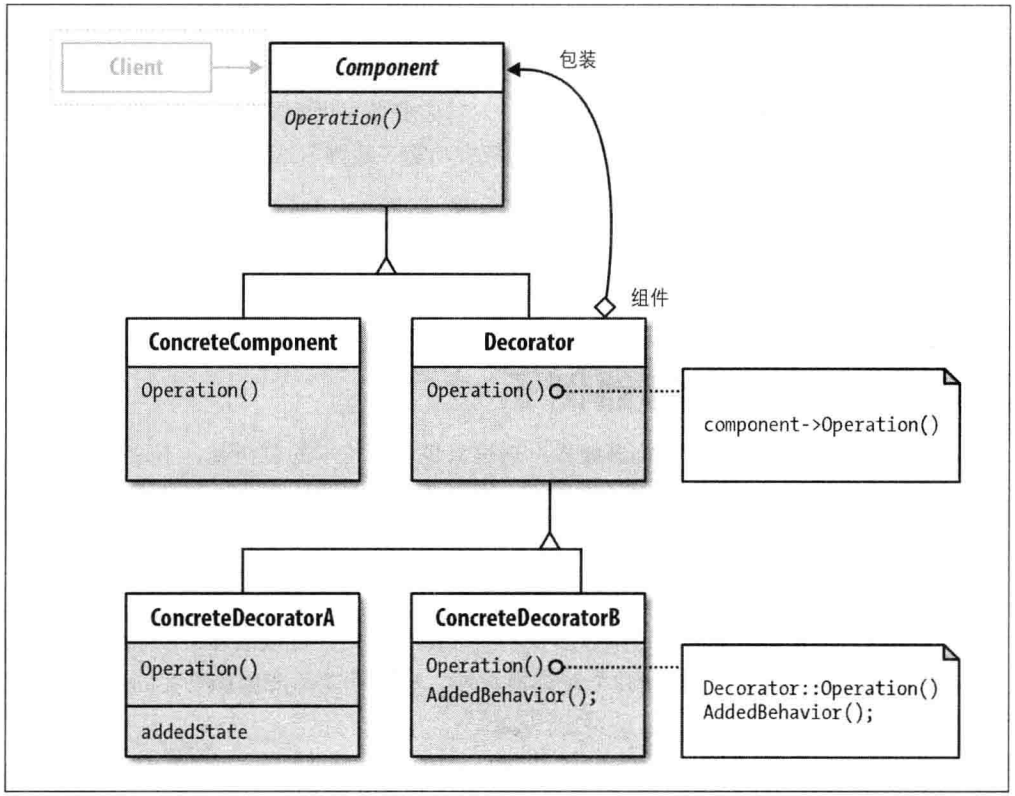


图8-1：装饰器类图 [隐含Client]

这个图中包含有Client，不过它实际上并不是这个模式的一部分——甚至连隐含包含都不算是。基于这种松耦合，可以直接请求任何具体组件或装饰器。不过，使用装饰器模式时，Client包含有Component接口的一个引用。

8.2 何时使用装饰器模式

“四人帮”指出了使用装饰器的一般原则。基本说来，如果想为现有对象增加新功能而不想影响其他对象，就可以使用装饰器模式。如果你好不容易为客户创建了一个网站

格式，主要组件的工作都很完美，客户请求新功能时，你肯定不希望推翻重来，再重新创建网站。例如，假设你已经构建了客户原先请求的所有组件，之后客户又有了新的需求，希望在网站中包含视频功能。你不用重写原先的组件，只需要“装饰”现有组件，为它们增加视频功能。这样一来，既可以保持原来的功能，还可以增加新功能。

有些项目可能有时需要装饰，而有时不希望装饰，这些项目体现了装饰器设计模式的另一个重要特性。假设你的基本网站开发模型可以满足大多数客户的要求。不过，有些客户还希望有一些特定的功能来满足他们的特殊需求。并不是所有人都希望或需要这些额外的功能。作为开发人员，你希望你创建的网站能满足客户的业务目标，所以需要提供“本地化”（customerization）特性，即针对特定业务提供的特性。利用装饰器模式，不仅能提供核心功能，还可以用客户要求的特有功能“装饰”这些核心功能。

由于Web和互联网日新月异（其范围更是飞速扩张），装饰器模式可能是你的开发工具包中的一个宝贵财富。使用装饰器模式就相当于你可以吃到蛋糕，或者至少可以装饰一些“糖霜”。

8.3 最简单的装饰器例子

第一个例子先考虑最简单的装饰器模式，这里只包含最基本的参与者，不过完全可以清楚地展示一个PHP应用。这个例子描述了一个web开发企业，企业计划建立一个基本网站，并提供一些增强功能。不过，web开发人员知道，尽管这个基本计划适用于大多数客户，但客户以后很可能还希望进一步提升。

从这个最简单的装饰器例子还可以看出，利用装饰器模式，可以很容易地增加多个具体装饰器。除了这个例子中提供的3个装饰器之外，看看你能不能增加更多的装饰器。实际上这很容易。

另外由于你能选择要增加的装饰器，所以企业不仅能控制功能，还可以控制项目的成本。装饰器模式很容易使用，不仅能选择装饰中增加什么，还能选择哪些不用增加。

8.3.1 Component接口

Component参与者是一个接口，在这里，它是一个抽象类`IComponent`。这个抽象类只有一个属性`$site`，另外有两个抽象方法`getSite()`和`getPrice`。Component参与者为具体组件和Decorator参与者抽象类建立了接口：

```
<?php
//IComponent.php
//Component接口
abstract class IComponent
```

```
{
    protected $site;
    abstract public function getSite();
    abstract public function getPrice();
}
?>
```

在PHP中，抽象方法只有一个很简短的签名，所以尽管getSite()和getPrice()会有期望的返回结果，但抽象方法声明中没有提供更多信息来指示应当生成何种类型的返回结果。

8.3.2 Decorator接口

这个例子中的装饰器接口可能会让你感到惊讶。这是一个抽象类，而且它扩展了另一个抽象类！这个类的作用就是维护组件接口（IComponent）的一个引用，这是通过扩展IComponent完成的：

```
<?php
//Decorator.php
//装饰器参与者
abstract class Decorator extends IComponent
{
    //继承 getSite()和getPrice()
    //这仍是一个抽象类
    //这里不需要实现任何一个抽象方法
    //任务是维护Component的引用
    //public function getSite() { }
    //public function getPrice() { }
}
?>
```

与所有其他抽象类一样，你也可以实现方法和增加属性。不过，在这个最简单的例子中，Decorator类的主要作用就是维护组件接口的一个链接（引用）。

讨论具体组件参与者之前，下面来看这个装饰器模式实现的类图，如图8-2所示。

所有这些参与者都有相同的接口。在所有装饰器模式实现中，你会发现，具体组件和装饰器都有相同的接口。它们的实现可能不同，另外除了基本接口的属性和方法外，组件和装饰器可能还会有额外的属性和方法。不过，从第一个例子我们可以得出，所有具体组件和装饰器都有共同的接口。

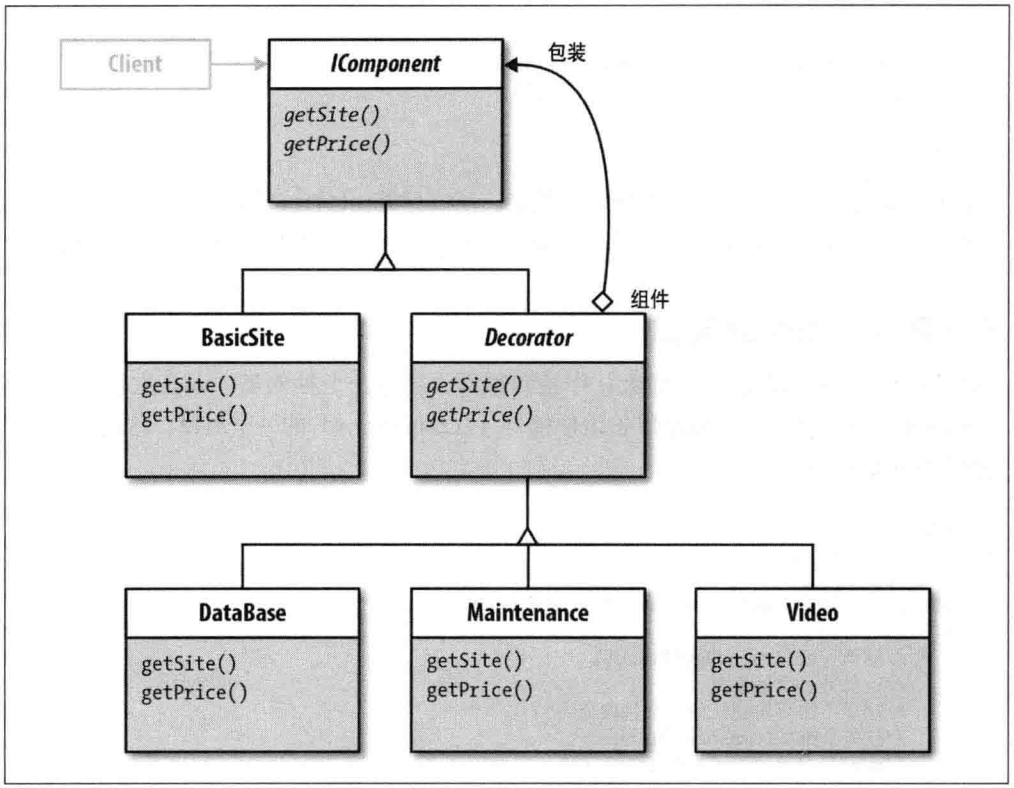


图8-2：最简单的装饰器模式实现的类图

8.3.3 具体组件

这个例子中只有一个具体组件，它生成一个网站名，另外生成一个基本网站报价（$1200）：

```
<?php
//BasicSite.php
//具体组件
class BasicSite extends IComponent
{
    public function __construct()
    {
        $this->site="Basic Site";
    }

    public function getSite()
    {
        return $this->site;
    }
```

```
    public function getPrice()
    {
        return 1200;
    }
}
?>
```

两个抽象方法都使用直接量赋值来实现（“Basic Site”和1200），不过灵活性并不体现在如何改变设置的值。实际上，要通过增加装饰器值来改变“Basic Site”值。下一小节会介绍这是如何做到的。

8.3.4 具体装饰器

这个例子中的具体装饰器与具体组件有相同的接口。实际上，它们是从Decorator抽象类（而不是IComponent类）继承了这个接口。不过，要记住，Decorator所做的就是继承IComponent接口。

Maintenance

```
<?php
//Maintenance.php
//具体装饰器
class Maintenance extends Decorator
{
    public function __construct(IComponent $siteNow)
    {
        $this->site = $siteNow;
    }
    public function getSite()
    {
        $fmat="<br/>   Maintenance ";
        return $this->site->getSite() . $fmat;
    }
    public function getPrice()
    {
        return 950 + $this->site->getPrice();
    }
}
?>
```

查看这个具体构造函数，你会发现，它看起来与装饰器几乎完全相同。不过，每个具体装饰器在它包装的具体组件价格上还会增加它自己的一个价格。另外两个具体装饰器与Maintenance装饰器也类似。

Video

```
<?php
//Video.php
//具体装饰器
```

```
class Video extends IComponent
{
    public function __construct(IComponent $siteNow)
    {
        $this->site = $siteNow;
    }
    public function getSite()
    {
        $fmat="<br/>   Video ";
        return $this->site->getSite() . $fmat;
    }
    public function getPrice()
    {
        return 350 + $this->site->getPrice();
    }
}
?>
```

Database

```
<?php
//DataBase.php
//具体装饰器
class DataBase extends Decorator
{
    public function __construct(IComponent $siteNow)
    {
        $this->site = $siteNow;
    }
    public function getSite()
    {
        $fmat="<br/>   MySQL Database ";
        return $this->site->getSite() . $fmat;
    }
    public function getPrice()
    {
        return 800 + $this->site->getPrice();
    }
}
?>
```

测试这个应用时，可以看到，在基本价格之上还会增加各个装饰器的价格。另外还指定了装饰器名的格式，增加了两个空格，使之缩进。

装饰器实现中最重要的元素之一就是构造函数，要为构造函数提供一个组件类型。由于这里只有一个具体组件，所有装饰器的实例化都会使用这个组件。使用多个组件时，装饰器可以包装应用中的一部分或全部组件，也可以不包装任何组件。

8.3.5 客户

Client类并不是这个设计模式的一部分，甚至连隐含包含都不算是，尽管如此，正确使

用Client类至关重要。每个装饰器必须在实例化时“包装”组件，这在上一节已经看到。不过，首先必须创建一个要包装的BasicSite类实例：

```
<?php
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    private $basicSite;

    public function __construct()
    {
        $this->basicSite=new BasicSite();
        $this->basicSite=$this->wrapComponent($this->basicSite);

        $siteShow=$this->basicSite->getSite();
        $format="<br/>  <strong>Total= $";
        $price=$this->basicSite->getPrice();

        echo $siteShow . $format . $price . "</strong><p/>";
    }

    private function wrapComponent(IComponent $component)
    {
        $component=new Maintenance($component);
        $component=new Video($component);
        $component=new DataBase($component);
        return $component;
    }
}
$worker=new Client()
?>
```

一旦Client实例化BasicSite，会有一个wrapComponent()方法检查这个实例，以确保参数有正确的数据类型（IComponent），然后分别实例化3个装饰器。（在装饰器设计模式的典型用法中，只会包含所选择的装饰器；可以注释掉两个装饰器，看看不同的组合会对总价有什么影响。）

wrapComponent()方法返回组件时（$this->basicSite），组件已经改变，现在已经包含了增加的装饰。由输出可以看到，具体组件实例现在由3个装饰器包装，不过，要记住，这里给出的是最简单的输出：

```
Basic Site
   Maintenance
   Video
   MySQL Database
   Total= $3300
```

也可以把wrapComponent()方法中的语句写在构造函数中，这样不用wrapComponent()方法Client也可以包装装饰器。不过，我们还是增加了wrapComponent()方法，这样就能把注意力更多地放在模式中的包装过程上。

8.4 关于包装器

适配器和装饰器模式都有另外一个名字“包装器”（wrapper）。实际上，在一些定义中，就把包装器描述为适配器。如果你还不太明确包装器的定义，可以先停一下，来看看如何编写一般的包装器，另外如何编写PHP包装器。

8.4.1 包装器包装基本类型

要了解包装器，最好的办法就是查看如何包装基本类型。下面的代码显示了如何将一个整数包装在一个对象中，以及如何获取这个整数：

```
<?php
class PrimitiveWrap
{
    private $wrapeMe;
    public function __construct($prim)
    {
        $this->wrapeMe=$prim;
    }

    public function showWrap()
    {
        return $this->wrapeMe;
    }
}
$myPrim=521;
$wrappedUp=new PrimitiveWrap($myPrim);
echo $wrappedUp->showWrap();
?>
```

如果需要将一个基本类型包装在对象中，如PrimitiveWrap，有些语言允许用内置的包装器来包装基本类型，相比之下，PHP中则要困难一些。一些语言（如Java）对于每一个基本类型都有相应的包装器类，所以在这些语言中，无需再构建新的包装器类。

8.4.2 PHP中的内置包装器

尽管PHP没有为所有基本类型都提供包装器，不过它确实有自己的包装器。例如，在第5章中就用到过一个包装器的例子，即file_get_contents()包装器。它将一个指定资源（如一个文件名，或者一个文件名的URL）绑定到一个流。下面的例子使用Ogden Nash写的一首小诗来说明这一点。首先，将这首小诗保存为一个文本文件（*celery.txt*）：

```
<strong>Celery</strong><p/>
Celery, raw<br/>
Develops the jaw,<br/>
But celery, stewed,<br/>
Is more quietly chewed. <p/>
--Ogden Nash<br/>
```

file_get_contents()包装器打开*celery.txt*，以便PHP程序输出。下面显示了这个包装器如何工作：

```
<?php
class TextFileLoader
{
    private $textNow;
    public function __construct()
    {
        $this->textNow = file_get_contents ("celery.txt");
        echo $this->textNow;
    }
}
$worker=new TextFileLoader();
?>
```

这个代码生成以下输出：

```
Celery

Celery, raw
Develops the jaw,
But celery, stewed,
Is more quietly chewed.

--Ogden Nash
```

文件名（*celery.txt*）“包装”在内置的file_get_contents()包装器中。

8.4.3 设计模式包装器

在第7章中，你已经看到对象适配器模式中的适配器（Adapter）参与者如何“包装”被适配者（Adaptee）。采用这种做法，就能创建一个与Adaptee兼容的接口。同样，从以上最简单的装饰器模式例子中可以看到，装饰器（Decorator）参与者可以“包装”一个组件对象，这样就能为这个已有的组件增加职责，而无须对它做任何修改。下面的代码展示了Client如何将组件对象（$component）包装在装饰器（Maintenance）中：

```
$component=new Maintenance($component);
```

类似于“接口”，在计算机编程中用到“包装器”时，不同的上下文中会有不同的用法和含义。一般来讲，在设计模式中使用“包装器”是为了处理接口的不兼容，或者希望为组件增加功能，包装器就表示用来减少不兼容性的策略。

8.5 包装多个组件的装饰器

尽管以上最简单的装饰器模式例子中只使用了一个具体组件，不过使用多个组件的情况也很常见。例如，图8-3显示了一个包含多个具体组件的类图。

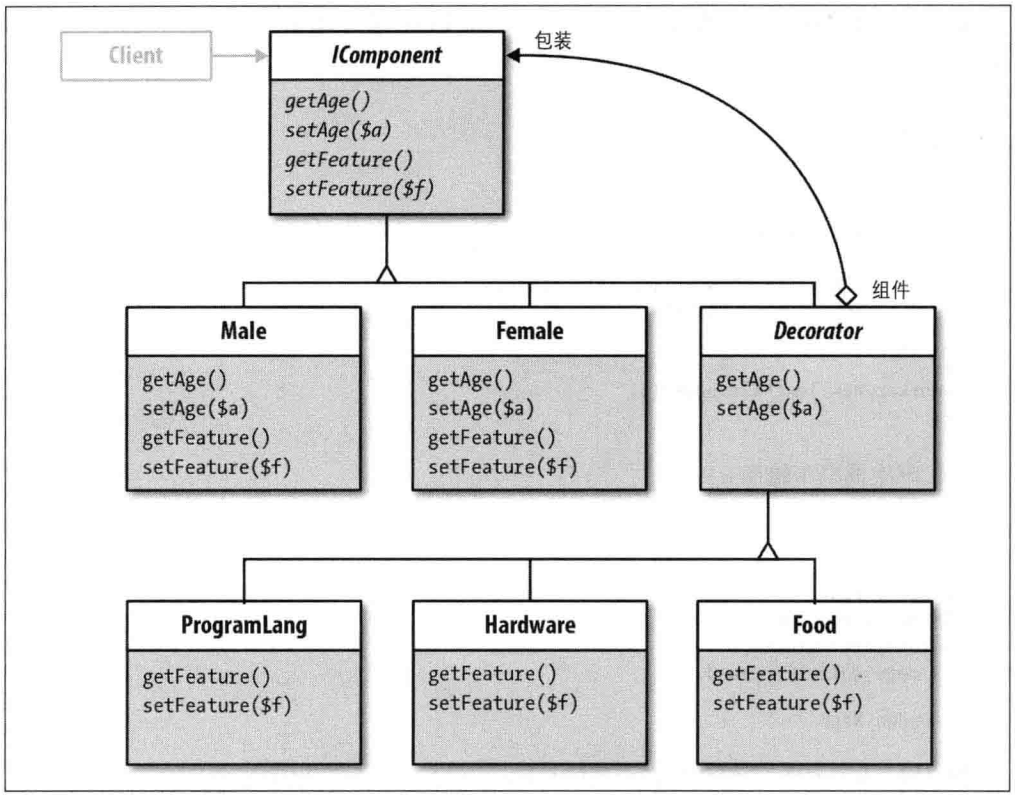


图8-3：装饰器模式中使用多个具体组件

Male和Female类表示IComponent抽象类的具体实现。装饰器采用数组方式增加属性。在这一节后面，我们会分别详细介绍这些属性。

8.5.1 多个具体组件

对于实现或装饰来说，包含多个具体组件并不成问题。只要这些具体组件与传入装饰器的组件有相同的接口，就可以在具体装饰器中包装任何组件，就像包含一个组件的程序或系统一样。

8.5.2 包含多个状态和值的具体装饰器

在图8-3中，继承IComponent抽象类接口的装饰器实现了一些方法。有几个问题需要说明。首先，装饰器（Decorator）实现了设置装饰状态时所有没有用到的方法。实际上，装饰器可以识别出组件中某些元素在装饰中不会改变（如组件的年龄和名字）。如果组件对应的方法只是在Decorator中实现，就不必在具体装饰器中重新实现这些方法。其次，在图8-1中可以看到，装饰器类图指示它至少要实现Component的一些接口。方法Operation()没有用斜体显示，这表示这个方法已经实例化。getAge()和setAge($a)方法已经实现，但是没有使用。如果它们未在Decorator中实现，就必须在各个具体装饰器中实现，因为继承的所有抽象方法都必须在子类中实现。

8.5.3 开发人员约会服务

为了说明如何实现一个包装多个组件的装饰器，下面给出一个例子，它为软件开发人员建立了一个约会服务。这里有两个组件，分别是Male和Female，可以分别为这两个组件装饰不同的约会关注点。可以用相同或不同的具体装饰采用任意组合来装饰这些组件。

基本说来，每个组件都有一个名字和指定的年龄。我们假设年龄不是装饰（不过在“实际生活”中，通常确实会把年龄作为“装饰”）。具体装饰会有不同的状态。各个装饰可以增加到具体组件，不仅如此，与第一个例子中一样，每个具体装饰可以有不同的状态。

组件接口

组件接口包括3个属性和5个方法。$date属性用来标识这是一个“约会”，而不是普通的日/月/年形式的日期对象。$ageGroup是该组件所属的组，$feature是由某个具体装饰提供的特性：

```
<?php
//IComponent.php
//Component 接口
abstract class IComponent
{
    protected $date;
    protected $ageGroup;
    protected $feature;

    abstract public function setAge($ageNow);
    abstract public function getAge();
    abstract public function getFeature();
    abstract public function setFeature($fea);
}
?>
```

所有方法都是抽象的，另外所有属性都是保护属性。

具体组件

两个具体组件[译注1]实现为Male和Female。它们分别有一个构造函数，将$date值（ID）设置为“Male”或“Female”。另外获取方法/设置方法设置了年龄和其他可能增加的装饰：

```
<?php
//Male.php
//Male 具体组件
class Male extends IComponent
{
    public function __construct()
    {
        $this->date="Male";
        $this->setFeature("<br/>Dude programmer features: ");
    }
    public function getAge()
    {
        return $this->ageGroup;
    }
    public function setAge($ageNow)
    {
        $this->ageGroup=$ageNow;
    }
    public function getFeature()
    {
        return $this->feature;
    }
    public function setFeature($fea)
    {
        $this->feature=$fea;
    }
}
?>
```

基于两个具体组件设置的初始特性，基本描述了男性（male）和女性（female）的特点：

```
<?php
//Female.php
//Female具体组件
class Female extends IComponent
{
    public function __construct()
    {
        $this->date="Female";
        $this->setFeature("<br />Grrrl programmer features: ");
    }
    public function getAge()
```

译注1：原文此处为“方法”，有误。

```
    {
        return $this->ageGroup;
    }
    public function setAge($ageNow)
    {
        $this->ageGroup=$ageNow;
    }
    public function getFeature()
    {
        return $this->feature;
    }
    public function setFeature($fea)
    {
        $this->feature=$fea;
    }
}
?>
```

可以使用$setFeature()方法为这两个组件增加其他特性作为装饰。可以把$setFeature()看作是一个组件增强器，而不只是一个组件设置方法。

包含组件方法的装饰器

装饰器接口扩展了组件接口。不过，如前所述，它还实现了年龄属性的获取方法和设置方法：

```
<?php
//Decorator.php
//装饰器参与者
abstract class Decorator extends IComponent
{
    public function setAge($ageNow)
    {
        $this->ageGroup=$this->ageGroup;
    }
    public function getAge()
    {
        return $this->ageGroup;
    }
}
?>
```

如果想增加一个else语句来捕获错误，这很容易做到。不过，目前采用了一种“安静失败”的做法，即不会传递对象。

具体装饰器

这个例子中的具体装饰器与前面最简单例子中的具体装饰器有很大不同。并不是只有一个状态，这些具体装饰器都包含数组，其中有多个属性值。除了使用一个包含4种不同语言的具体装饰器作为具体装饰，也可以有4个不同的具体装饰，分别表示一种不同的

计算机语言。不过，将相同的选择结合到一个数组中可以达到同样的目标，还可以保持松耦合：

```
<?php
//ProgramLang.php
//具体装饰器
class ProgramLang extends Decorator
{
    private $languageNow;

    public function __construct(IComponent $dateNow)
    {
        $this->date = $dateNow;
    }

    private $language=array("php"=>"PHP",
                        "cs"=>"C#",
                        "js"=>"JavaScript",
                        "as3"=>"ActionScript 3.0");

    public function setFeature($lan)
    {
        $this->languageNow=$this->language[$lan];
    }

    public function getFeature()
    {
        $output=$this->date->getFeature();
        $fmat="<br/>  ";
        $output .="$fmat Preferred programming language: ";
        $output .= $this->languageNow;
        return $output;
    }
}
?>
```

所有具体装饰器都有相同的格式。首先使用getFeature()为一个输出变量赋值，设置为所选择的元素，然后在这个输出变量后面加上格式化标记和一般的具体组件信息，共同作为输出：

```
<?php
//Hardware.php
//具体装饰器
class Hardware extends Decorator
{
    private $hardwareNow;

    public function __construct(IComponent $dateNow)
    {
        $this->date = $dateNow;
    }

    private $box=array("mac"=>"Macintosh",
                    "dell"=>"Dell",
```

```
                    "hp"=>"Hewlett-Packard",
                    "lin"=>"Linux");

    public function setFeature($hdw)
    {
        $this->hardwareNow=$this->box[$hdw];
    }

    public function getFeature()
    {
        $output=$this->date->getFeature();
        $fmat="<br/>  ";
        $output .="$fmat Current Hardware: ";
        $output .= $this->hardwareNow;
        return $output;
    }
}
?>
```

通过使用关联数组（即字符串元素作为键的数组），选择装饰属性时会更为容易一些。对于每一个特性，所有具体装饰器都有类似的数组：

```
<?php
//Food.php
//具体装饰器
class Food extends Decorator
{
    private $chowNow;

    public function __construct(IComponent $dateNow)
    {
        $this->date = $dateNow;
    }

    private $snacks=array("piz"=>"Pizza",
                    "burg"=>"Burgers",
                    "nach"=>"Nachos",
                    "veg"=>"Veggies");

    public function setFeature($yum)
    {
        $this->chowNow=$this->snacks[$yum];
    }

    public function getFeature()
    {
        $output=$this->date->getFeature();
        $fmat="<br/>  ";
        $output .="$fmat Favorite food: ";
        $output .= $this->chowNow . "<br/>";
        return $output;
    }
}
?>
```

客户应当能很容易地访问具体装饰，所以所有获取方法和设置方法都有public可见性。不过，不同属性（数组）中设置的值为private，这样可以增加组件装饰过程的封装性。

Client

最后，Client类请求一个组件和具体装饰。首先通过实例化选择一个具体组件，然后Client设置感兴趣的年龄组。Client将4个年龄组分别设置为字符串“Age Group N”，其中“N”是1～4之间的一个字符串值：

```
<?php
//Client.php
/*Age groups:
    18-29: Group 1
    30-39: Group 2
    40-49: Group 3
    50+ : Group 4
*/
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    //$hotDate是组件实例
    private $hotDate;

    public function __construct()
    {
        $this->hotDate=new Female();
        $this->hotDate->setAge("Age Group 4");
        echo $this->hotDate->getAge();
        $this->hotDate=$this->wrapComponent($this->hotDate);
        echo $this->hotDate->getFeature();
    }

    private function wrapComponent(IComponent $component)
    {
        $component=new ProgramLang($component);
        $component->setFeature("php");
        $component=new Hardware($component);
        $component->setFeature("lin");
        $component=new Food($component);
        $component->setFeature("veg");

        return $component;
    }
}
$worker=new Client()
?>
```

实例化这两个具体组件之一之后，Client将它们包装在某个具体装饰器中来完成装饰。

整个过程都在wrapComponent()方法中完成。创建各个实例时，setFeature()方法利用参数（由一个关联数组键构成）来完成组件的装饰。客户会生成以下输出：

```
Female
Age Group 4
Grrrl programmer features:
    Preferred programming language: PHP
    Current Hardware: Linux
    Favorite food: Veggies
```

可以看到，装饰器就是具体装饰器类中关联数组的元素。

8.6 HTML用户界面

到目前为止，这本书还没有介绍HTML用户界面（UI）。在第5部分中，所有应用都包括一个HTML用户界面。之所以目前还没有增加HTML用户界面，原因在于我们现在重点强调的是不同设计模式的基本原理。PHP并未嵌在一个HTML包装器中，所生成的HTML也只用于一般的格式化。

下面要用HTML UI来显示如何通过Client类（对象）连接到一个PHP设计模式。使用“开发人员约会服务”一节中的例子，可以看到，HTML与PHP之间的通信与其他项目中任何其他HTML→PHP程序中的通信是一样的。在PHP中，唯一有改变的是允许从HTML表单向Client类传递值。模式中的所有其他参与者都保持不变。

首先，来看我们完成的HTML UI，如图8-4所示。可以看到，页面使用单选钮来提供所有数据输入。

数据输入很简单，只需要提供这个页面的标准HTML/CSS文件。首先，页面使用了以下CSS文件：

```
@charset "UTF-8";
/* CSS Document */
/* devedate.css */
header
{
    font-family:Cracked;
    font-size:72px;
    color:#F00;
    background-color:#000;
}
h2
{
    font-family:"Courier New", Courier, monospace;
    font-size:24px;
}
```

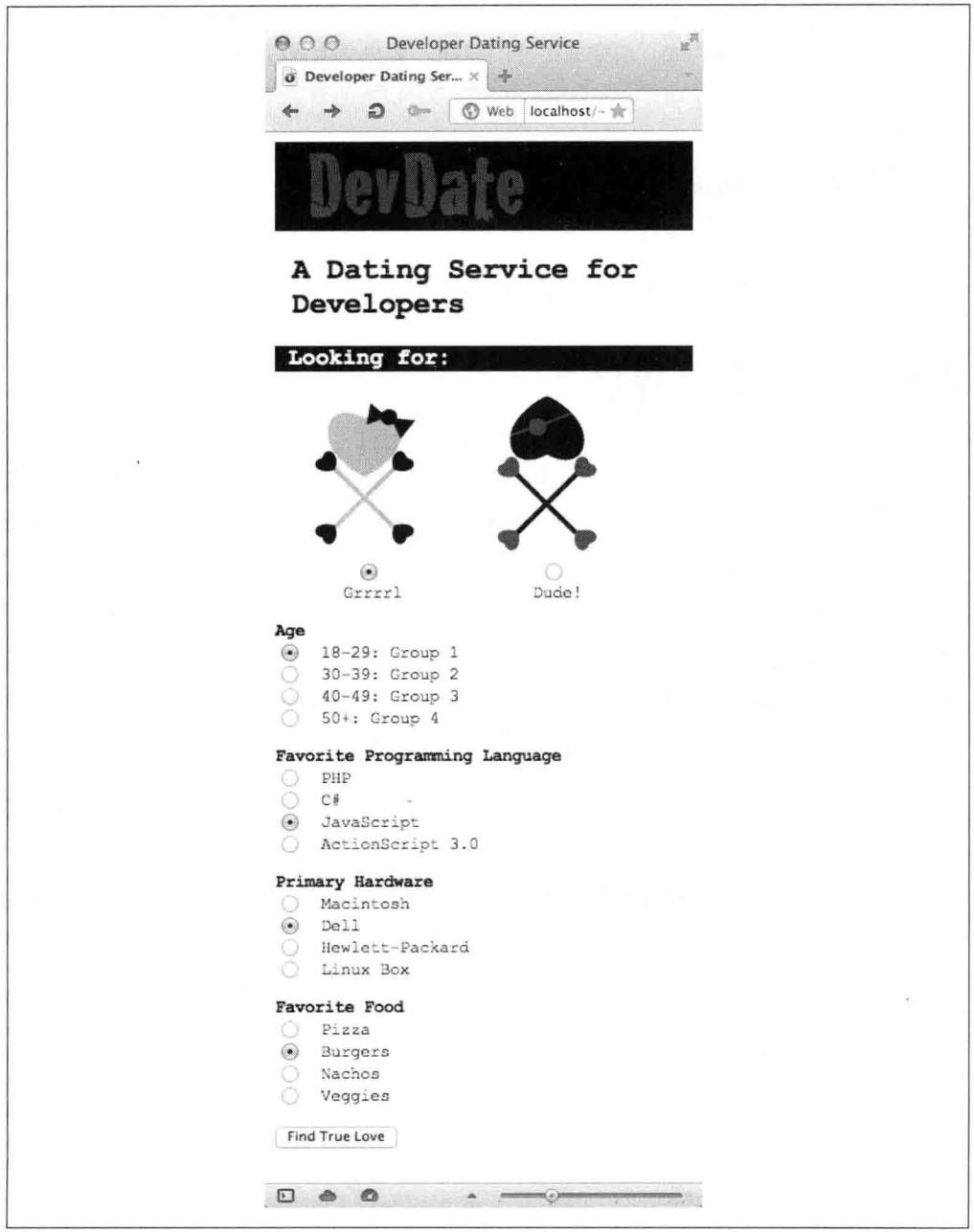


图8-4：装饰器的数据输入

```
h3
{
    font-family:"Courier New", Courier, monospace;
```

```
    font-size:18px;
    color:#fff;
    background-color:#000;
}

body
{
    font-family:"Courier New", Courier, monospace;
    font-size:14px;
}
#sex
{
    display:table;
}
#fem
{
    display:table-cell;
    width:150px;
    text-align:center;
}
aside
{
    display:table-cell;
    width:150px;
    text-align:center;
}
```

这个页面是针对移动设备的垂直布局建立的，不过也可以很好地用在平板电脑或桌面系统上。接下来给出页面的HTML，如以下代码所示。它调用了一个页面（*ClientH.php*）。类和文件名中的*H*表示它将接收HTML数据（可能需要为HTML或PHP脚本增加验证代码；如果某一组单选钮均未选择，就会抛出错误！）：

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<link rel="stylesheet" type="text/css" href="css/devdate.css">
<title>Developer Dating Service</title>
</head>

<body>
<article>
  <header>  DevDate</header>
  <h2> A Dating Service for <br/>
       Developers</h2>
  <h3> Looking for:</h3>
  <form action="ClientH.php" method="post">
    <div id="sex">
      <div id="fem"> <img src="pix/grrrl.png" width="95" height="128"
        alt="grrrl"><br/>
        <input type="radio" name="gender" value="Female">
        <br/>
         Grrrl </div>
```

```
      <aside> <img src="pix/dude.png" width="99" height="133" alt="dude"><br/>
        <input type="radio" name="gender" value="Male">
        <br/>
         Dude!
        <p/>
      </aside>
    </div>
    <strong>Age<br/>
    </strong>
    <input type="radio" name="age" value="Age Group 1">
     18-29: Group 1 <br/>
    <input type="radio" name="age" value="Age Group 2">
     30-39: Group 2<br/>
    <input type="radio" name="age" value="Age Group 3">
     40-49: Group 3<br/>
    <input type="radio" name="age" value="Age Group 4">
     50+: Group 4
    <p/>
    <strong>Favorite Programming Language<br/>
    </strong>
    <input type="radio" name="progLang" value="php">
     PHP<br/>
    <input type="radio" name="progLang" value="cs">
     C#<br/>
    <input type="radio" name="progLang" value="js">
     JavaScript<br/>
    <input type="radio" name="progLang" value="as3">
     ActionScript 3.0
    <p/>
    <strong>Primary Hardware<br/>
    </strong>
    <input type="radio" name="hardware" value="mac">
     Macintosh<br/>
    <input type="radio" name="hardware" value="dell">
     Dell<br/>
    <input type="radio" name="hardware" value="hp">
     Hewlett-Packard<br/>
    <input type="radio" name="hardware" value="lin">
     Linux Box
    <p/>
    <strong>Favorite Food<br/>
    </strong>
    <input type="radio" name="food" value="piz">
     Pizza<br/>
    <input type="radio" name="food" value="burg">
     Burgers<br/>
    <input type="radio" name="food" value="nach">
     Nachos<br/>
    <input type="radio" name="food" value="veg">
     Veggies
    <p/>
    <input type="submit" name="search" value="Find True Love">
  </form>
</article>
```

```
</body>
</html>
```

基本上，这个页面会把数据传递到一个PHP页面，就像通过PHP从一个HTML页面向一个MySQL数据库发送数据一样。所以，尽管数据发送到一个装饰器设计模式中的Client类，在传递数据方面，与以往向PHP文件传递其他数据并没有不同。

8.6.1 Client类传递HTML数据

前面已经指出，从HTML向一个PHP Client类发送数据与向一个数据库表发送数据是类似的。查看ClientH类，可以看到它类似于Client类，只不过使用了*DeveloperDating.html*文件的数据：

```
<?php
//ClientH.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class ClientH
{
    //$hotDate是组件实例
    private $hotDate;
    private $progLange;
    private $hardware;
    private $food;
    public function __construct()
    {
        $gender=$_POST["gender"];
        $age=$_POST["age"];
        $this->progLang=$_POST["progLang"];
        $this->hardware=$_POST["hardware"];
        $this->food=$_POST["food"];

        $this->hotDate=new $gender();
        $this->hotDate->setAge($age);
        echo $this->hotDate->getAge();
        $this->hotDate=$this->wrapComponent($this->hotDate);
        echo $this->hotDate->getFeature();
    }

    private function wrapComponent(IComponent $component)
    {
        $component=new ProgramLang($component);
        $component->setFeature($this->progLang);
        $component=new Hardware($component);
        $component->setFeature($this->hardware);
        $component=new Food($component);
        $component->setFeature($this->food);

        return $component;
    }
```

```
}
$worker=new ClientH()
?>
```

通过使用$_POST关联数组，可以把单选钮变量和值传递到PHP变量。性别（gender）和年龄（age）变量通过构造函数传递，而组件变量（$hotDate）和3个装饰器变量（$proLang、$hardware和$food）都声明为私有变量，取代在原来Client类中使用相应直接量。

8.6.2 从变量名到对象实例

要实例化一个对象实例，一般做法是为类实例指定一个命名变量。在这本书中，约定所有类名首字母大写，变量类实例名都使用小写，如下：

```
$someInstance= new SomeClass();
```

只要知道类名，这种格式会很合适。不过，如果需要通过一个从HTML页面传递的变量来声明多个组件，类名必须来自这个变量的值。利用PHP，这个过程极其容易，尤其是与使用某个eval函数的方法相比（使用eval函数也可以达到这个目的，不过烦琐得多）。

在第6章已经看到，通过一个变量命名和实例化一个类时，一般过程就是将类名赋至一个变量，然后用这个变量实例化这个类。例如，假设有一个类名为Nature；可以通过一个变量动态地实例化一个类实例：

```
$quack = "Nature";
$myNature= new $quack();
```

这样一来，$myNature是Nature类的一个实例。在类ClientH中，可以看到$hotDate变量必须实例化Female或Male组件类。用户要选择男性或女性，需要在两个单选钮中做出选择（两个单选钮都命名为*gender*）。两个值分别为类名Female和Male。所以，这些值传递到PHP变量$gender时，就会实例化相应的具体组件类。下面这行代码完成实例化：

```
$this->hotDate=new $gender();
```

需要说明，这个表达式包含new关键字，另外变量$gender后面有开始和结束小括号。

8.6.3 增加装饰

你可能希望最好包含一个查询来了解用户喜欢哪一类电影，以此衡量约会双方的契合度。例如，动作片、爱情片、科幻片和音乐片都可能是用户喜欢的电影类型。可以考察

一下你对装饰器设计模式的理解程度，自己动手增加一个具体装饰器类，由用户喜欢的电影类型来装饰所选择的具体组件。

这很简单。实际上，可以复制粘贴现有的某个具体组件，把它重命名为`Films`，然后改变类别选项，使用`Client`类中的直接量值测试。如果一切正常，再更新HTML UI，加入电影选择，并修改ClientH类向程序传递这个请求。

程序越庞大，OOP编程和结构型设计模式就越有用。这是因为，第3部分中讨论的所有模式都可以用来改变现有结构的功能，而不必重写原来的结构。

第4部分

行为型设计模式

伟大的发现和进步无疑都离不开很多人的合作。
也许人们会称赞我开辟了全新的道路。
不过，当我看到后来取得的发展时，
我觉得受赞扬的应当是别人而不是我。
——亚历山大·格拉汉姆·贝尔

一个人的行为如果不再有新的变化，他的行为也将不再明智。
——托马斯·卡莱尔

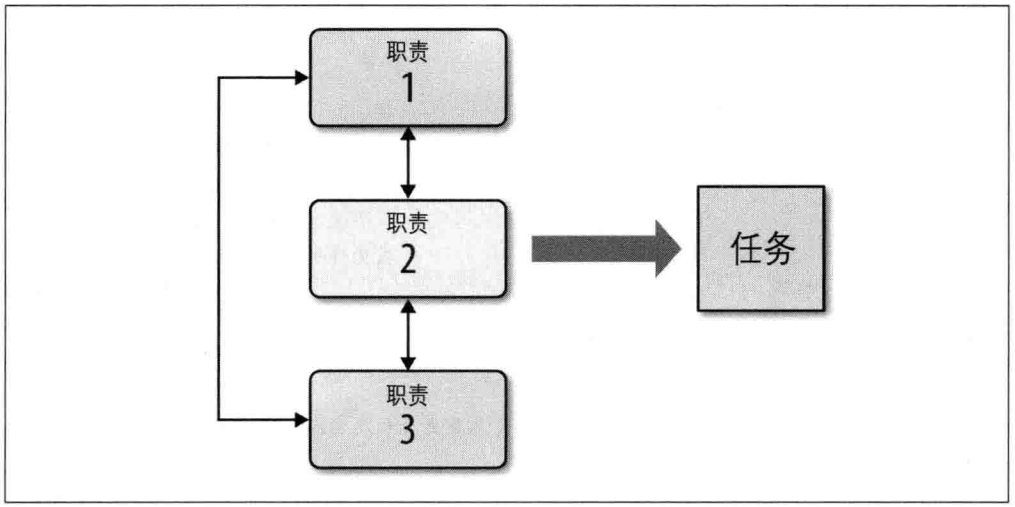
社会总是更大程度受人们交流所利用的媒介的影响，
而不是交流的内容。
——马歇尔·麦克卢汉

“四人帮”所提供的设计模式中，行为型设计模式占了绝大部分。这一部分将介绍一个类设计模式（模板方法模式）和一个对象设计模式（状态模式），不过第5部分还会结合PHP/MySQL例子介绍更多的行为型模式。概括来讲，Gamma、Helm、Johnson和Vlissides指出了11个行为型模式：

- 职责链模式（Chain of Responsibility）
- 命令模式（Command）
- 解释器模式（Interpreter）（类设计模式）
- 迭代器模式（Iterator）
- 中介者模式（Mediator）
- 备忘录模式（Memento）

- 观察者模式（Observer）
- 状态模式（State）
- 策略模式（Strategy）
- 模板方法模式（Template Method）（类设计模式）
- 访问者模式（Visitor）

理解行为型设计模式的关键是通信。对于这些模式，重点不再是构成一个设计模式的对象和类，而应转变为对象和类之间的通信。实际上，最好考虑对象如何相互合作来完成任务，从这个角度来理解行为型设计模式。



图IV-1：行为型模式强调模式参与者之间的通信

要重点考虑构成模式的元素之间的交互，这一点非常重要，有些模式的类图看起来是一样的，如状态模式和策略模式。不过，由于参与者通信的方式不同，另外它们处理职责的方式也不同，所以这些模式实际上有很大不同。

第9章

模板方法模式

学习重载和模板的规则并不难，
甚至会因此忽视它们正是构建精巧而高效的类型安全容器的关键。
——本贾尼·斯特劳斯特卢普

所有伟大传奇都成为人类行为的样板。
神话在我看来其实是曾经存在的故事。
——约翰·保曼

我也会考虑其他方面，
不过要尽早知道我看起来是什么样子。
这就像是一个模板；我会把人套进这个模板。
如此我才会感觉心安。
——朱迪·丹奇女爵士

9.1 什么是模板方法模式

首先，要区别两个概念，模板方法（Template Method）设计模式中使用了一个类方法`templateMethod()`，要知道这二者是不同的。`templateMethod()`是抽象类中的一个具体方法。这个方法的作用是对抽象方法序列排序，具体实现留给具体类来完成。关键在于，模板方法模式定义了操作中算法的“骨架”，而由具体类来实现。

关于模板方法模式，值得高兴的是它相当简短，实现很容易。只需要一个抽象类和一个具体类，如图9-1所示。

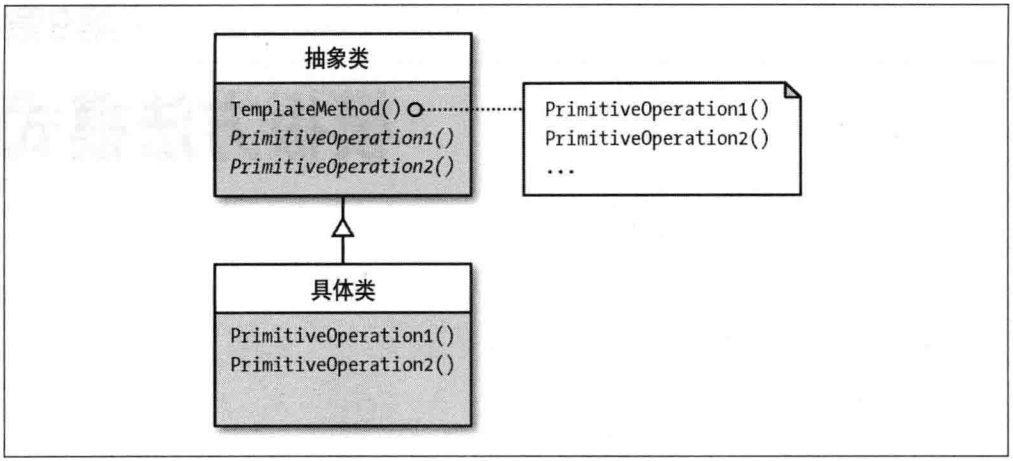


图9-1：模板方法类图

可以把抽象类中的模板方法操作看做是“基本操作的组织者”。后面提到模板方法设计模式时，我会将首字母大写（Template Method），谈到模板方法操作时，则会用小写（template method operation）。就像考虑婚礼司仪，她要按某种顺序安排仪式中的一系列事件（操作），包括宴会、切蛋糕、第一支舞、发表感言以及仪式中可能的所有事件。婚礼司仪会遵循一个模板，按一种给定的顺序合理地组织事件，不过她会让新娘的家人具体决定希望如何执行这些事件。例如，宴会可能是一个正式宴会，也可以是一个户外的烧烤晚会，舞蹈可能选择华尔兹也可能是街舞（hip-hop），乐队可以是整个乐团，也可以只请一位手风琴师。事件发生的顺序都是一样的，不过它们的实现有所不同。

9.2 何时使用模板方法

如果已经明确算法中的一些步骤，不过这些步骤可以采用多种不同的方法实现，就可以使用模板方法模式。“四人帮”指出，如果算法中的步骤不变，可以把这些步骤留给子类具体实现。在这种情况下，可以使用这种设计模式来组织抽象类中的基本操作（函数/方法）。然后由子类实现应用所需的这些操作。

还有一种用法稍微复杂一些，可能需要把子类共同的行为放在一个类中，以避免代码重复。“四人帮”援引了Opdyke和Johnson的“重构为一般性”（refactoring to generalize），以此描述将重复的代码组织到模板方法模式中的过程。如果使用多个类来解决同一个大型问题，可能很快就会出现重复代码。

最后一点，还可以使用模板方法模式控制子类扩展。这里涉及一个“钩子”操作，将在

“模板方法设计模式中的钩子”一节中讨论。可以利用“钩子”控制扩展：只在钩子操作所在的某些位置允许扩展。

9.3 最简单的例子：对图像和图题使用模板方法模式

在PHP编程中，可能经常会遇到一个问题：要建立带图题的图像。这个算法相当简单，无非是显示图像，然后在图像下面显示文本。

由于模板方法设计中只涉及两个参与者，所以这是最容易理解的模式之一，同时也非常有用。抽象类建立templateMethod()（具体的方法），并由具体类实现这个方法。

9.3.1 抽象类

抽象类是这里的关键，因为它同时包含具体和抽象方法。模板方法（这里是指方法本身）往往是具体方法，其操作是抽象的。

要加载一个图片，只需要一个操作使用HTML包装器调用图像URL。要得到文本作为图题并把它放在图像下面，这个操作也同样很简单。

两个抽象方法分别是addPix($pix)和addCaption($cap)。这两个操作都包含一个参数，分别表示图像的URL信息和图题字符串。templateMethod()函数中设置了这两个属性，属性可见性为保护（protected），以提供封装：

```
<?
//AbstractClass.php
abstract class AbstractClass
{
    protected $pix;
    protected $cap;

    public function templateMethod($pixNow,$capNow)
    {
        $this->pix=$pixNow;
        $this->cap=$capNow;
        $this->addPix($this->pix);
        $this->addCaption($this->cap);
    }
    abstract protected function addPix($pix);
    abstract protected function addCaption($cap);
}
?>
```

在模板方法函数中增加这两个参数，以便接收实参，再将接收到的实参进一步传递到保护属性。

9.3.2 具体类

要使用模板方法的具体类扩展这个抽象类，并实现基本操作addPix()和addCaption()。算法要求提供相应的代码，可以用来显示图像和适当的图题：

```
<?
//ConcreteClass.php
include_once('AbstractClass.php');
class ConcreteClass extends AbstractClass
{
    protected function addPix($pix)
    {
        $this->pix=$pix;
        $this->pix = "pix/" . $this->pix;
        $formatter = "<img src=$this->pix><br/>";
        echo $formatter;
    }
    protected function addCaption($cap)
    {
        $this->cap=$cap;
        echo "<em>Caption:</em>" . $this->cap . "<br/>";
    }
}
?>
```

这里并没有返回一个对象，而是由这两个操作使用echo语句将结果发送到屏幕。这样做是为了尽可能简化这个例子。现在客户可以使用addPix()和addCaption()方法了。

9.4 客户

“四人帮”讨论这种设计模式时并没有提到客户，不过类似于客户作为其他设计模式中的一部分，在这里客户就相当于UI连接的基础。所以，可能需要模板方法模式的任何上下文中，都可以考虑这样一个简单的Client类：

```
<?
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    function __construct()
    {
        $caption="Modigliani painted elongated faces.";
        $mo=new ConcreteClass();
        $mo->templateMethod("modig.png",$caption);
    }
}
```

```
$worker=new Client();
?>
```

注意，$mo变量实例化了ConcreteClass，但是它调用templateMethod()，这是从父类继承的一个具体操作。父类（AbstractClass）通过templateMethod()调用子类的操作。图9-2显示了相应的输出。

图9-2：模板方法建立的图片－图题序列

实际上，客户只需要提供图像URL和图题文本串。

9.5 好莱坞原则

“反向控制结构”概念另外也称为“好莱坞原则”（the Hollywood Principle）。这个原则是指父类调用子类的操作，而子类不调用父类的操作。（就像试镜之后，导演告诉年轻的演员，“如果你拿到这个角色，我们会通知你。不要打电话来询问；我们会打电话给你的”。正因如此，这种反向控制结构被称为好莱坞原则。）

与好莱坞原则关联最紧密的模式就是模板方法模式，因为它在父类中实现。除了templateMethod()，父类中的其他操作（方法）都是抽象和保护方法。所以，尽管客户实例化一个具体类，但它调用了父类中实现的方法。图9-3可以帮助你更清楚地理解这种反向控制结构。

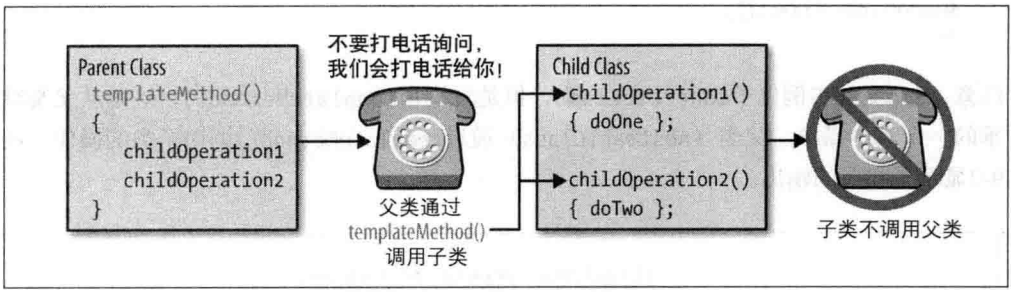


图9-3：好莱坞原则

不过，我们真正想问的是：为什么好莱坞原则对于OOP和设计模式很重要？从某些方面来讲，这个问题和答案的关键在于与顺序和过程编程思维划定界限。在过程编程中，关键问题是控制流（flow of control），在OOP中，关键问题则是对象职责（object responsibility）。由于过程编程的重点在于控制流，一些解释会使用“控制反向”（inversion of control）来解释好莱坞原则。控制反向从过程编程的角度来讲很有意义，不过在OOP中，大多数控制流都会通过对象职责和协作抽取出来。也就是说，并不是考虑控制流，而应考虑哪些对象将处理某些职责，另外对象如何协作来完成任务。

要在OOP上下文中准确地理解好莱坞原则，最简单的方法就是从框架以及框架中可能的改变来考虑 [这里我使用“框架”（framework）这个词来表示程序中的小结构，而不是第3章中与设计模式相区别的那些更大的结构。参见John Vlissides在1996年2月“C++ Report”中发表的“Pattern Hatching”]。John Vlissides指出，模板方法定义了框架，子类可以扩展或重新实现算法的可变部分，不过它们不能改变模板方法的控制流。从子类发出“调用”是重新实现父类的方法，这是好莱坞原则不允许的一种“调用”。只有父类可以做出“调用”来建立或改变框架（操作的执行顺序）。

要理解好莱坞原则，更好的办法可能是按幼儿园里老师和学生之间的关系来考虑，也就是幼儿园原则（Kindergarten Principle）。老师建立一些项目，让孩子们按某种顺序完成，比如数数、认时间，背单词。老师设定好顺序，但是孩子们具体如何完成或者如何实现则由孩子们自己决定。不过，孩子们不能改变老师定好的这个顺序。换句话说，孩子在数数练习中间不能说“我现在想背单词”。这个结构是有序的：

1. 数数
2. 认时间
3. 背单词

老师说，“抱歉，Elmo，我们现在要数数。过一会再背单词吧”。对孩子们来说，这个顺序是不可变的。所以只能由老师来掌控，没有其他可能。在这种情况下，控制反向是

不合适的：由父类建立顺序，子类按自己特有的方式完成这些操作。直观看来这并不是“反向”。所以，如果讨论好莱坞原则时谈到控制反向，要记住这个“反向”只在过程编程的控制流上下文中才有意义。

9.6 结合其他设计模式使用模板方法模式

设计模式并不是孤立存在的，它们存在于编码对象的大环境中，其中一些对象可能是其他设计模式。为了说明这一点，下面结合工厂方法模式来实现第一个例子，我们可以得到相同的结果，即显示一个有图题的图片。由于这个例子只使用了一个图像和图题，看起来有点像高射炮打蚊子——大材小用了，不过，关键是要知道这两个设计模式如何合作。如果你理解了模式之间的交互，就能结合使用更多的模式。模板方法模式将建立算法的顺序，以给定的顺序来显示图像和图题，工厂方法模式则具体创建对象。图9-4给出了相应的文件图，两个不同模式分别用灰边框区分。

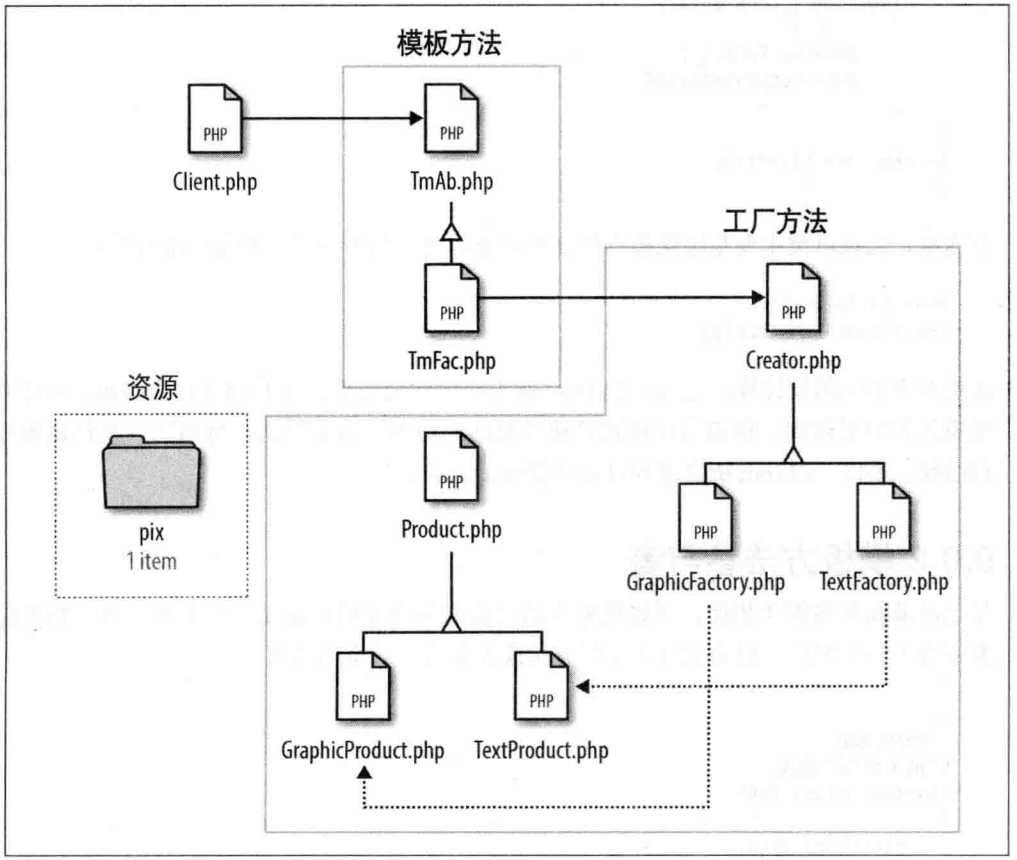


图9-4：有关联关系的模板方法和工厂方法文件图

结合使用这两个模式是因为它们分别承担不同的角色。模板方法模式建立算法的顺序（首先是图像，然后是图题），工厂方法模式则创建图像和图题。看到这个，你可能认为客户不得不做出请求；不过，情况正好相反。

9.6.1 客户工作负担减轻

客户的请求会通过模板方法模式做出。模板方法模式负责组织操作，从而能按适当的顺序自动地传递请求：

```
<?
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    function __construct()
    {
        $mo=new TmFac();
        $mo->templateMethod();
    }
}
$worker=new Client();
?>
```

在这里，Client类主要是错误检查和加载必要的类。只有两行代码与请求有关：

```
$mo=new TmFac();
$mo->templateMethod();
```

连接两个不同的设计模式之后，Client的工作居然会减少，这看起来有些讽刺，不过利用模式很容易做到。模板方法模式将建立算法的顺序，这些算法再调用工厂来创建图像和图题。所以，Client所请求的只是将接收请求的方法。

9.6.2 模板方法参与者

与之前最简单的例子相比，创建模板方法模式的两个类稍有改变。在某些方面，TmFac类相当于一个客户，因为它会向工厂方法模式中的工厂发出请求：

```
<?
//TmAb.php
//抽象模板方法类
abstract class TmAb
{
    protected $pix;
    protected $cap;
```

```
    public function templateMethod()
    {
        $this->addPix();
        $this->addCaption();
    }

    protected abstract function addPix();
    protected abstract function addCaption();

}
?>

<?
//TmFac.php
//具体模板方法
//调用工厂方法
class TmFac extends TmAb
{
    protected function addPix()
    {
        $this->pix=new GraphicFactory();
        echo $this->pix->doFactory();
    }
        protected function addCaption()
    {
        $this->cap=new TextFactory();
        echo $this->cap->doFactory();
    }
}
?>
```

templateMethod()操作指定TmFac类中方法的顺序，调用templateMethod()后会按指定的顺序来调用工厂。

9.7 工厂方法参与者

Client类与工厂方法模式之间没有任何通信。最初的请求由模板方法操作传递，不过它们没有留下任何“足迹”（暂且这么说）。工厂方法参与者所响应的是来自TmFac对象的请求：

```
<?php
//Creator.php
abstract class Creator
{
    protected abstract function factoryMethod();

    public function doFactory()
    {
        $mfg= $this->factoryMethod();
        return $mfg;
    }
```

```
}
?>

<?php
//GraphicFactory.php
class GraphicFactory extends Creator
{
    protected function factoryMethod()
    {
        $product=new GraphicProduct();
        return($product->getProperties());
    }
}
?>

<?php
//TextFactory.php

class TextFactory extends Creator
{
    protected function factoryMethod()
    {
        $product=new TextProduct();
        return($product->getProperties());
    }
}
?>

<?php
//Product.php
interface Product
{
    public function getProperties();
}
?>

<?php
//GraphicProduct.php

class GraphicProduct implements Product
{
    private $mfgProduct;

    public function getProperties()
    {
        $this->mfgProduct="<img src='pix/modig.png'>";
        return $this->mfgProduct;
    }
}
?>

<?php
//TextProduct.php
class TextProduct implements Product
{
    private $mfgProduct;
```

```
    public function getProperties()
    {
        $this->mfgProduct ="<div style='color:#cc0000; font-size:12px;
                            font-family:Verdana, Geneva, sans-serif'>
                            <strong><em>Caption:</em></strong> Modigliani
                            painted elongated faces.</div>";
        return $this->mfgProduct;
    }
}
?>
```

图9-5与图9-2几乎完全相同，只是图题的样式不同。从templateMethod()向工厂做出请求后，由工厂实现的产品会生成图题的样式。

图9-5：按照 templateMethod()中的步骤生成的图像和图题

图9-5的重点是，由于设计模式力求所有参与者（类和相应的对象）松耦合但相互连接，所以程序很灵活，完全可以通过所有类和接口来返回图像和图题。对于一个简单的图像和图题，可能注意不到这个好处。不过，对于更复杂的算法，如果有更多的步骤，模板方法模式可以提供一个可重用的顺序来组织职责集。产品以及产品中的变化都将通过同样的接口来处理。

9.8 模板方法设计模式中的钩子

有时模板方法函数中可能有一个你不想要的步骤，某些特定情况下你可能不希望执行这个步骤。例如，假设有一个模板方法，可以累加一个订单的总费用，另外加上税和运费，然后显示交易总金额。不过，顾客告诉你，如果买家购买需要送货的商品超过200美元，可以免运费。这里就可以用到模板方法钩子。

在模板方法设计模式中，利用钩子可以将一个方法作为模板方法的一部分，不过不一定会用到这个方法。换句话说，它是方法的一部分，不过它包含一个钩子，可以处理例外情况。子类可以为算法增加一个可选元素，这样一来，尽管仍按模板方法建立的顺序执行，但有可能并不完成模板方法期望的动作。对于上面这种运费可选的情况，钩子就是解决这个问题最理想的工具。

你可能认为，这与好莱坞原则有冲突（子类没有遵循父类设置的顺序），这样想也没错。好莱坞原则要求只有父类能够改变框架。钩子很特殊，因为尽管它实现了模板方法中的方法，但实现的方法有一个“后门”，也就是说，它会处理例外情况。

要了解钩子是如何工作的，来看一个简单的例子。一家公司销售GPS设备和地图，为顾客沿赞比西河徒步旅行提供帮助。它还提供租船业务。公司老板决定，如果买家购买的需要送货的商品超过200美元，可以免收运费。不过，这不包括租船的费用。所以，他需要这样一个程序，可以累加送货商品的费用，确定是否增加运费，然后加上租船的费用，最后显示总金额。图9-6显示了这个程序的用户界面。

UI HTML代码使用了一个表单，以便数据输入，表单中使用了单选钮和复选框。单选钮和复选框都有一个值，分别表示产品或服务的价格。下面的HTML代码清单中包括一个HTML数组，用来提供地图选择：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<link rel="stylesheet" type="text/css" href="zambezi.css">
<title>Zambezi Trading Post</title>
<body>
<img src="zambezi.png" width="800" height="183" alt="zam">
<h1>Adventure Outfitters</h1>
<form action="Client.php" method="post">
  <h2> Equipment</h2>
  <h3>Navigation</h3>
  <p><strong>GPS</strong></p>
  <input type="radio" name="gps" value="98">
   Curiosity $98 
  <input type="radio" name="gps" value="112">
   Cabot $112 
```

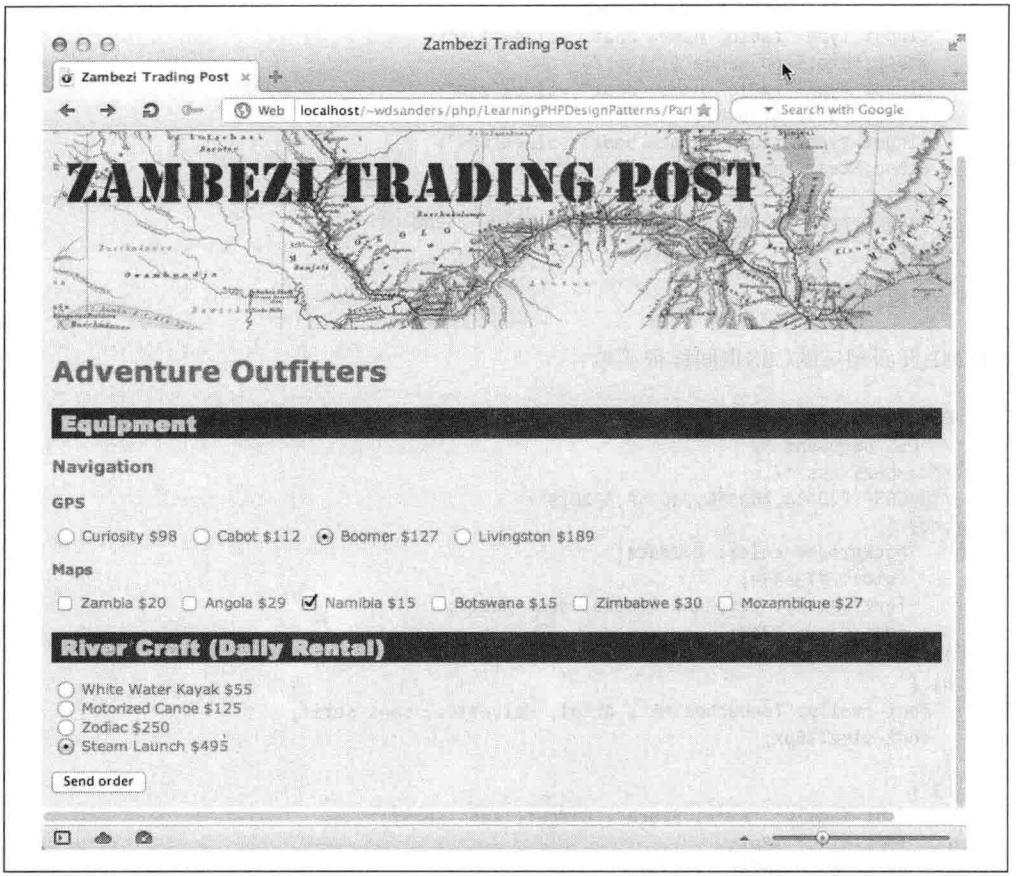


图9-6：结合模板方法使用钩子实现的PHP UI

```
<input type="radio" name="gps" value="127">
 Boomer $127 
<input type="radio" name="gps" value="189">
 Livingston $189 
<p><strong>Maps</strong></p>
<input type="checkbox" name="map[]" value="20">
&nbspZambia $20 
<input type="checkbox" name="map[]" value="29">
&nbspAngola $29 
<input type="checkbox" name="map[]" value="15">
&nbspNamibia $15 
<input type="checkbox" name="map[]" value="15">
&nbspBotswana $15 
<input type="checkbox" name="map[]" value="30">
&nbspZimbabwe $30 
<input type="checkbox" name="map[]" value="27">
&nbspMozambique $27<br/>
<h2> River Craft (Daily Rental)</h2>
<input type="radio" name="boat" value="55">
```

```
   White Water Kayak $55<br/>
  <input type="radio" name="boat" value="125">
   Motorized Canoe $125<br/>
  <input type="radio" name="boat" value="250">
   Zodiac $250<br/>
  <input type="radio" name="boat" value="495">
   Steam Launch $495
  <p/>
  <input type="submit" name="sender" value="Send order">
</form>
</body>
</html>
```

HTML页面相应的CSS也同样很简单：

```
@charset "UTF-8";
/* CSS Document */
/*zambezi.css */
/*D9C68F,F2DAC4,A69586,73635A,592D23*/
body {
    background-color: #f2dac4;
    color: #73635a;
    font-family: Verdana, Geneva, sans-serif;
    font-size: 12px;
}
h1 {
  font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
  font-size:36px;
}
h2 {
    font-family: "Arial Black", Gadget, sans-serif;
    font-size: 18px;
    background-color: #592d23;
    color: #d9c68f;
}
```

一旦用户单击“Send order”（发送订单）按钮，PHP Client类就会启动。不过，先来看这里采用的模板方法模式。

9.8.1 建立钩子

在模板方法接口中建立钩子方法看起来很有意思，尽管子类可以改变钩子的行为，但仍然要遵循模板方法中定义的顺序：

```
<?
//IHook.php
abstract class IHook
{
    protected $purchased;
    protected $hookSpecial;
    protected $shippingHook;
    protected $fullCost;
```

```
    public function templateMethod($total,$special)
    {
        $this->purchased=$total;
        $this->hookSpecial=$special;
        $this->addTax();
        $this->addShippingHook();
        $this->displayCost();
    }

    protected abstract function addTax();
    protected abstract function addShippingHook();
    protected abstract function displayCost();
}
?>
```

这里有3个抽象方法——addTax()、addShippingHook()和displayCost()，抽象类IHook实现的templateMethod()中确定了它们的顺序。在这里，钩子方法放在中间，实际上模板方法指定的顺序中，钩子可以放在任意位置。模板方法需要两个参数：一个是总花费，另外还需要一个变量用来确定顾客是否免收运费。这些值必须由Client类提供，Client类则从HTML文档接收原始数据。

9.8.2 实现钩子

一旦抽象类中建立了这些抽象方法，并指定了它们执行的顺序，子类将实现所有这3个方法：

```
<?
//ZambeziCalc.php
class ZambeziCalc extends IHook
{
    protected function addTax()
    {
        $this->fullCost = $this->purchased + ($this->purchased * .07);
    }
    protected function addShippingHook()
    {
        if(! $this->hookSpecial)
        {
            $this->fullCost += 12.95;
        }
    }
    protected function displayCost()
    {
        echo "Your full cost is $this->fullCost";
    }
}
?>
```

addTax()和displayCost()方法都是标准方法，只有一个实现。不过，addShippingHook()的实现有所不同，其中有一个条件来确定是否增加运费。这就是钩子。

9.8.3 客户以及捕获钩子

由于可以使用二进制字符，开发人员通常会忽视布尔变量的重要性。不过，实际上处理布尔变量很简洁，速度也很快，我发现它们对于捕获钩子尤其适合。在所有钩子操作中，必须有人警告控制流：将有不同的情况发生，而不是正常地执行模板方法中指定的那些默认操作。

用比较操作符设置布尔值

不必使用条件语句来建立一个布尔状态，使用比较操作符会更容易，也更简洁。客户中包含以下代码行，可以为前面的模板方法例子（Zambezi Trading Post.）中的钩子设置布尔值。

```
$this->special = ($this->buyTotal >= 200);
```

这会为布尔变量`$this->special`赋一个状态*true*或*false*。Zambezi Trading Post为货运商品超过200美元的所有订单提供了免收运费的特殊优惠。

接下来看`Client`类，可以看到模板方法设计模式中的请求如何访问钩子。

Client类

`Client`类根据从HTML表单接收的数据做出一个请求。它必须将租船的费用与购买设备的费用区分开，只根据设备购买费用来计算特殊折扣。将special设置为true或false后，再累加总金额，还要加上税费。由于变量`$special`是一个布尔变量，不会把它增加到总金额。实际上，`$special`会作为一个参数传递到`templateMethod()`方法：

```
<?
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    private $buyTotal;
    private $gpsNow;
    private $mapNow;
    private $boatNow;
    private $special;
    private $zamCalc;

    function __construct()
    {
        $this->setHTML();
        $this->setCost();
        $this->zamCalc=new ZambeziCalc();
```

```
        $this->zamCalc->templateMethod($this->buyTotal,$this->special);
    }

    private function setHTML()
    {
        $this->gpsNow=$_POST['gps'];
        $this->mapNow=$_POST['map'];
        $this->boatNow=$_POST['boat'];
    }

    private function setCost()
    {
        $this->buyTotal=$this->gpsNow;
        foreach($this->mapNow as $value)
        {
            $this->buyTotal+= $value;
        }
        //Boolean
        $this->special = ($this->buyTotal >= 200);
        $this->buyTotal += $this->boatNow;
    }
}
$worker=new Client();
?>
```

首先，利用从单选钮传入`$gpsNow`变量的值来确定设备的总费用；然后通过一个循环将所有传入复选框数组（`$mapNow`）的值累加起来，如上一节所述，此时已经建立了布尔变量`$special`；最后，加上通过一个单选钮传入的`$boatNow`值。设置`$special`之后，程序将`$boatNow`值与`$buyTotal`值相加。现在可以把这两个变量中的值发送到主程序，主程序将在`$templateMethod()`参数中使用这两个值。

这个例子很简单，用户的数据输入也很容易，最重要的是，只需要重写`Client`，就可以很容易地重用这个设计。换句话说，这个设计是可改变而且可重用的。这个例子中使用直接量来表示运费和税费，不过，它们同样也可以修改为不同的值取代直接量。完全可以用客户传入的值，甚至可以使用可计算的费用，而不是直接量。

9.9 短小精悍的模板方法模式

学习设计模式很不容易，并不适合那些心存畏惧的人，不过模板方法模式不仅很简单，还有很多经验可以借鉴。首先，要解释好莱坞原则，采用模板方法模式是最适合的。这个原则中的“调用”概念实际上是指“遵循某个顺序”。父类（抽象类）建立操作，并设置它们的顺序，而子类具体实现这些操作。“不要打电话来询问，我们会打电话给你的”可以改写为：“我们会建立面试、试镜和才艺测试；你要按我们建立的顺序完成这些测试，但是可以采用你喜欢的任何方式。只是不要改变这个顺序！”

模板方法模式还有一个重要的方面，要记住这是一个设计模式，可以与其他设计模式结合使用。你已经看到，它能与工厂方法模式很好地合作，同样地，这个模式还能与抽象工厂模式协作（而且这样更有帮助）。模板方法模式很简短，所以可以作为其他模式的一部分，也可以作为一个辅助类在很多其他模式中使用。

关于模板方法模式，最好的一点是它很容易学习，而且有重要的经验可以借鉴。不错，模板方法模式有很多应用，这也很重要，不过概念和思想才是设计模式中最重要的部分。作为一个很容易掌握和学习的模式，尽管它很短小，但确实很有意义。

第10章

状态设计模式

如果年轻人能够意识到他们很快就会固守习惯，
他们就会在可塑阶段更加留意自身的行为。
——威廉·詹姆斯

每一个行动中，我们必须把目光放到我们过去、现在和将来的行动之外，
还要超越这些行为影响到的其他人，而看到所有这一切之间的关系。
这样一来，我们就会非常谨慎。
——布莱士·帕斯卡

所有最后的决定都是在一种不会持续的心态下做出的。
——马塞尔·普鲁斯特

10.1 什么是状态模式

状态（State）设计模式是GoF提出的最吸引人的模式之一，也是一种最有用的模式。游戏通常就采用状态模式，因为游戏中的对象往往会非常频繁地改变状态。状态模式的作用就是允许对象在状态改变时改变其行为。还有很多其他模拟应用（不一定是游戏）也依赖于状态模式。你会发现状态模式有很多应用（这一章将会谈到）。图10-1用类图方式显示了这种基本设计模式。

如图所示，这里没有Client类；不过，GoF指出Context是客户的主要接口。查看这个模式时，可以认为Client类通过ConText类做出请求。要理解并且有效地使用状态设计模式，需要了解很多内容，所以对目前来说，先来单看一下这个类图，具体的细节会在后面逐步展开。

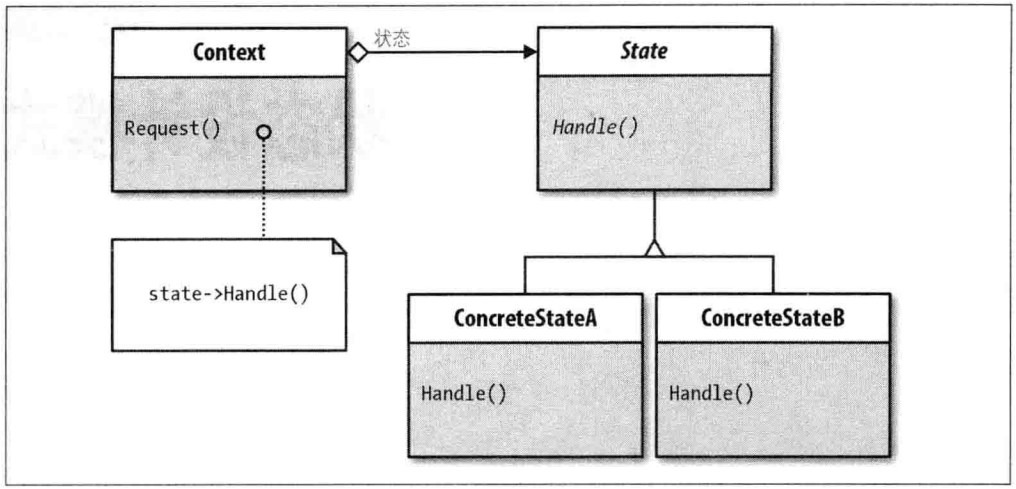


图10-1：状态设计模式类图

10.2 何时使用状态模式

如前所述，游戏和模拟器开发人员通常使用状态模式来处理不同的状态。PHP中几乎每一个应用都会有一些状态改变，一个对象依赖于它的状态时，它肯定会频繁地改变，在这种情况下，状态模式就有绝对的优势。

对象中频繁的状态改变非常依赖于条件语句。就其自身来说，条件语句本身没有什么问题（如`switch`语句或那些带`else`子句的语句）。不过，如果选项太多，以至于程序开始出现混乱，或者增加或改变选项需要花费太多时间，甚至成为一种负担，这就出现了问题。

先来看模拟器或游戏的一个简单例子。假设有一个3 × 3的矩阵，也就是9个状态，不同的单元格（共有9个单元格）会有不同的选择。考虑图10-2中的矩阵。

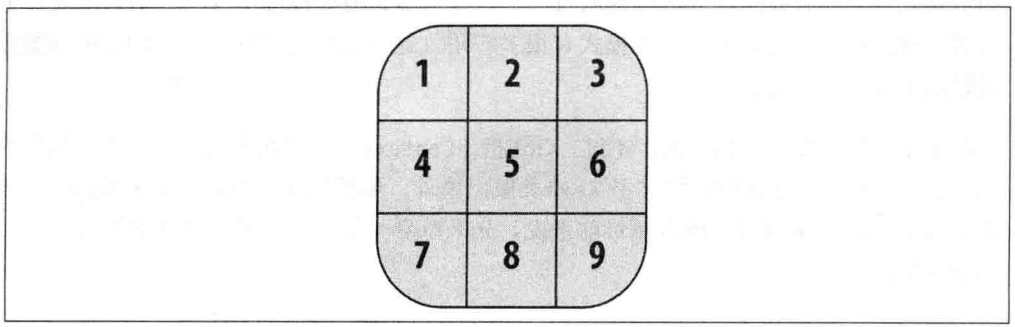


图10-2：3×3矩阵

假设你创建了一个程序，可以上下移动和左右移动，但是不能沿对角线方向移动。如果在单元格5（见图10-2）上，对象可以向任何方向移动（上下和左右都可以），不过除了这个单元格以外，采用传统方式编程时，对于其他单元格都需要某个条件语句。考虑单元格4的移动，可能会有类似下面的代码：

```
if($this->moveUp())
{
    $this->currentCell=$cell_1;
}
elseif ($this->moveDown())
{
    $this->currentCell=$cell_7;
}
elseif ($this->moveRight())
{
    $this->currentCell=$cell_5;
}
elseif ($this->moveLeft())
{
    $this->currentCell=$errorMove;
}
```

在此基础上，程序必须跟踪对象在哪些单元格“中”。可以看到，条件语句的个数（或switch语句中的case个数）可能会大幅增长，使程序变得纠缠不清。

对于状态设计模式，每个状态都有自己的具体类，它们实现一个公共接口。我们不打算查看对象的控制流，而是从另一个角度来考虑，即对象的状态。下一节分析状态机，通过介绍，你将更好地理解“以状态为中心”的思路。

10.3 状态机

状态机是一个模型，其重点包括不同的状态、一个状态到另一个状态的变迁，以及导致状态改变的触发器。要研究状态机，最好的起点是状态图（statechart），我们将用状态图来分析，而不是一个计算机流程图或类图。图10-3为一个简单的状态图，其中一个灯泡的状态从关（off）变为开（on）。

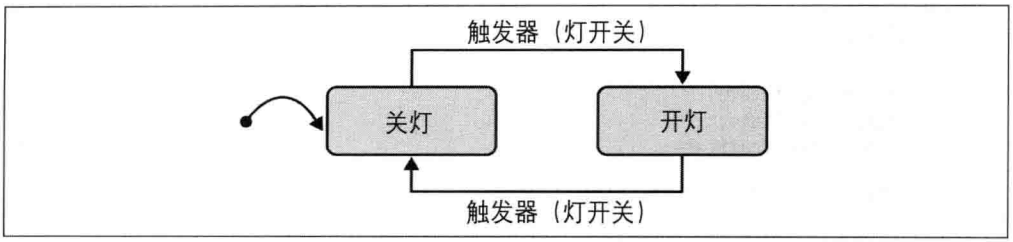


图10-3：显示灯泡状态的状态图

与类图相似，状态图强调的是控制流以外的东西。从图10-3可以看到状态模型的本质：

- 状态（关灯和开灯）
- 变迁（从关灯到开灯，以及从开灯到关灯）
- 触发器（灯开关）

对于打开和关上的灯，其状态、变迁和触发器都非常简单。变迁是即时的，触发器就是灯开关。不过，触发器请求一个改变之后，有些变迁可能需要渐进完成，或者要采用更复杂的方式完成。例如，天气触发树叶颜色的改变，这个变迁就是渐进完成的。不过，可以使用这个状态模型来理解从夏天状态变换到秋天状态，另外只要是通过一个触发器激活从一个状态到另一个状态的状态变迁，这种情况都可以使用这个状态模型来理解。

10.4 开灯关灯：最简单的状态设计模式

尽管状态机的概念相当简单，但要知道，即使是最基本的状态设计模式也可能很有难度。考虑到这一点，这个最简单的PHP模式例子将重点考虑创建模式的步骤。有一个好消息，如果你能完成一个简单的状态设计模式，那么更大规模的模式也能轻松应对。所以我们慢慢来，先来看这个模式中的各个参与者。

10.4.1 情境为王

所有状态模式都需要一个参与者来跟踪对象所处的状态。例如，对于前面的灯泡状态例子，就很有必要记住图10-2中的矩形。如果当前状态是单元格4，系统需要知道可以通过哪些变迁进入其他状态。这正是ConText类要完成的任务。要知道第一个例子只处理两个状态，即灯的状态（关或开）。Context要知道当前状态是什么。使用图10-3中的状态图作为指导，可以看到初始状态为关（由一个带箭头的黑色小球指示）。

作为一个参考点，首先来看ConText类。下面几小节将分析这个上下文，了解它的各个部分以及各部分的角色：

```
<?php
//Context.php
class Context
{
    private $offState;
    private $onState;
    private $currentState;

    public function __construct()
    {
        $this->offState=new OffState($this);
        $this->onState=new OnState($this);
```

```
        //开始状态为Off
        $this->currentState=$this->offState;
    }
    //调用状态方法触发器
    public function turnOnLight()
    {
        $this->currentState->turnLightOn();
    }

    public function turnOffLight()
    {
        $this->currentState->turnLightOff();
    }
    //设置当前状态
    public function setState(IState $state)
    {
        $this->currentState=$state;
    }

    //获得状态
    public function getOnState()
    {
        return $this->onState;
    }
        public function getOffState()
    {
        return $this->offState;
    }
}
?>
```

ConText类建立了3个属性，可见性均为私有（private）：

```
$offState
$onState
$currentState
```

前两个分别是两个状态的实例，第三个属性用来跟踪给定时间系统所处的状态。

ConText类中的状态实例

在构造函数中，Context实例化IState实现的两个实例——一个对应关（off）状态，另一个对应开（on）状态：

```
$this->offState=new OffState($this);
$this->onState=new OnState($this);
```

这个实例化过程用到了一种递归，称为自引用（self-referral）。构造函数参数中的实参写为$this，这是Context类自身的一个引用。状态类希望接收一个ConText类实例作为参数，为了在ConText类中实例化一个状态实例，它必须使用$this作为实参。

由于必然有某个状态作为启动时的当前状态，将$currentState属性赋为$offState值。（可以这样考虑，刚走进一个房间，当时灯是关着的。）这是OffState类的一个实例。查看图10-3中的状态图，可以看到开始状态是off状态，所以代码只是遵循了状态图中的结构。

调用状态方法：上下文触发器方法

Context类中的一些方法要调用状态类中的方法。可以把这些方法想成是触发器（triggers）。调用这些触发器，就会启动从当前状态到另外一个不同状态的变迁。举例来说，下面的方法就是一个触发器：

```
public function turnOnLight()
{
    $this->currentState->turnLightOn();
}
```

注意，Context方法的名字与状态方法稍有不同：这里是turnOnLight而不是turnLightOn。这些差别只是为了将Context类中的触发器方法与状态实例中的方法相区别。

设置当前状态

Context类最重要的作用是跟踪当前状态，从而为系统提供一个正确的上下文或窗口。再回到图10-2中的矩阵，矩阵中的每一步移动都取决于当前单元格。对于单元格9，可以移到单元格8或单元格6（不允许沿对角线方向移动）。不过，系统必须知道它的当前状态是单元格9，这样才能知道有哪些选择。

要设置一个当前状态，必须以某种方式向Context类发送信息，指定当前状态。这是通过某个状态类完成的。一旦触发一个状态，这个状态就会向Context发送一个消息，指示“我是当前状态”：

```
public function setState(IState $state)
{
    $this->currentState=$state;
}
```

setState()方法需要一个状态对象作为实参（由Istate类型提示指示）。触发器方法触发时，它会调用一个状态和相关的方法。这个状态的方法必须向Context发送一个消息，指出现在该状态是当前状态。所以最近触发的状态会调用setState()方法，使它成为当前状态。

状态获取方法

最后，Context类需要有办法得到当前状态发送的消息。这个消息通过获取方法传递。对于每一个状态，Context都要有相应的获取方法。由于这个例子有两个状态类，所以只需要两个获取方法。OffState的获取方法如下：

```
public function getOffState()
{
    return $this->offState;
}
```

OnState的获取方法如下：

```
public function getOnState()
{
    return $this->onState;
}
```

在实现的状态类中可以看到，这些方法将在设置当前状态时使用。

Context类小结

Context实例化所有状态的实例，并设置默认状态。Context包含有一些方法，可以通过调用具体状态中的相应方法来触发不同的状态。设置方法会跟踪当前状态。为了帮助跟踪当前状态，对于每个状态Context还有一个获取方法，会在状态有改变调用。

10.4.2 状态

如果了解了具体状态类如何实现IState接口，就更能看出Context类的意义。这个接口只包含两个要实现的状态方法：

```
<?php
//IState.php
interface IState
{
    public function turnLightOn();
    public function turnLightOff();
}
?>
```

在Context类中，这两个方法都称为状态改变触发器。不过，重要的细节都在以下这两个状态类的实现中：OnState和OffState。

OnState

```
<?php
//OnState.php
class OnState implements IState
```

```
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function turnLightOn()
    {
        echo "Light is already on-> take no action<br/>";
    }
    public function turnLightOff()
    {
        echo "Lights off!<br/>";
        $this->context->setState($this->context->getOffState());
    }
}
?>
```

OffState

```
<?php
//OffState.php
class OffState implements IState
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function turnLightOn()
    {
        echo "Lights on! Now I can see!<br/>";
        $this->context->setState($this->context->getOnState());

    }
    public function turnLightOff()
    {
        echo "Light is already off-> take no action<br/>";
    }
}
?>
```

OnState和OffState类是Istate的简单实现，会有文本消息指示这些状态。状态类在构造函数中包含了Context类的一个引用。应该记得，Context会实例化状态实例，并为状态构造函数类提供一个自引用。

默认状态是OffState。它必须实现IState方法turnLightOn和turnLightOff。Context调用turnLightOn方法，这会显示“Lights on! Now I can see”（开灯了，现在我能看见了）。然后它向Context方法getOnState发送一个消息，将OnState作为当前状态。不过，如果是调用OffState中的turnLightOff，就只有一个消息指示灯已经关了，不会有

任何动作。它不会重置Context中的当前状态，因为这已经是当前状态。基本说来，如果请求一个状态启动它自身，就什么也不会做。类似地，如果请求触发一个无法触发的状态，同样什么也不会做。

再来看图10-2中9个单元格的矩阵。从单元格3可以合法地移动到单元格2或单元格6。不过，在单元格3状态中，它不能上移或右移。所以，如果它接收到一个指令，要求启动一个它无法达到的状态，对于这种不可达到的状态，通常程序员会提供一个null条件。

10.4.3 客户通过上下文做出请求

Client的所有请求都通过Context做出。Client与任何状态类之间都没有直接连接，包括IState接口。下面的Client显示了触发两个状态类中所有方法的请求：

```
<?php
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    private $context;

    public function __construct()
    {
        $this->context=new Context();
        $this->context->turnOnLight();
        $this->context->turnOnLight();
        $this->context->turnOffLight();
        $this->context->turnOffLight();
    }
}
$worker=new Client();
?>
```

实例化一个Context实例之后，初始请求是打开灯，因为灯的默认状态是“关”（off）状态。第二个请求是一样的，不过它只生成一个消息，指示系统目前所处的状态正是请求的这个状态，什么也不会发生。以下输出显示了这些请求的结果：

```
Lights on! Now I can see!
Light is already on-> take no action
Lights off!
Light is already off-> take no action
```

*OffState*请求的工作类似于*OnState*请求。如果变迁是从一个状态到另一个状态（off → on），就会启动这个改变。不过，如果再次请求当前状态（off → off），只会生成一个消息，指示什么也不会发生。

10.5 增加状态

对于所有设计模式来说，很重要的一个方面是：利用这些设计模式可以很容易地做出修改。与其他模式一样，状态模式同样也很易于更新和改变。对于前面最简单的例子（只包含基本的*on/off*状态），为了查看增加状态有什么影响，下面这个例子会扩展*on/off*状态，变成一个3路灯泡。在这个新应用中，状态将加倍，变成4个状态：

- 关（Off）
- 开（On）
- 更亮（Brighter）
- 最亮（Brightest）

图10-4 显示了这个更新后的4态状态图。

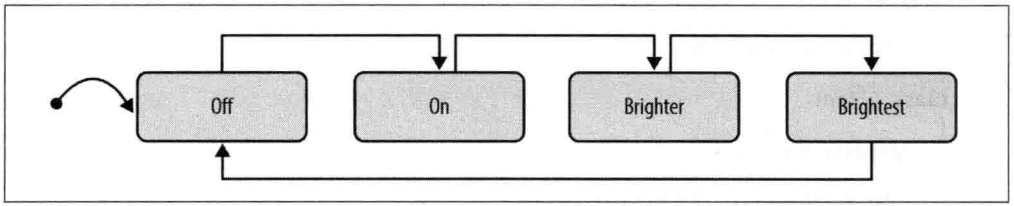


图10-4：三路灯泡的状态图

查看这4个状态，序列有所改变。“关”（off）状态只能变到“开”（on）状态，*on*状态不能变到*off*状态。实际上，现在规则有变化，*on*状态只能变到“更亮”（brighter）状态和“最亮”（brightest）状态。只有*brightest*状态可以变到“关”（off）状态。

10.5.1 改变接口

要改变的第一个参与者是接口`IState`。这个接口中必须指定相应的方法，可以用来迁移到`brighter`和`brightest`状态：

```
<?php
//IState.php
interface IState
{
    public function turnLightOff();
    public function turnLightOn();
    public function turnBrighter();
    public function turnBrightest();
}
?>
```

现在所有状态类都必须包括这4个方法，它们都需要结合到`Context`类中。

10.5.2 改变状态

状态设计模式中有改变时，这些新增的改变会对模式整体的其他各方面带来影响。不过，增加改变相当简单，因为状态图显示了变迁，而且在这种情况下，每个状态只有一个特定的变迁。我们不再使用文本，现在每个状态都有该状态的一个图形表示。如果变迁合法，就会显示一个特定的图像（如果变迁不合法，会显示*nada.png*图像）。

OffState

```
<?php
//OffState.php
class OffState implements IState
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function turnLightOn()
    {
        echo "<img src='lights/on.png'>";
        $this->context->setState($this->context->getOnState());
    }
    public function turnBrighter()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnBrightest()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnLightOff()
    {
        echo "<img src='lights/nada.png'>";
    }
}
?>
```

OnState

```
<?php
//OnState.php
class OnState implements IState
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function turnLightOn()
```

```
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnBrighter()
    {
        echo "<img src='lights/brighter.png'>";
        $this->context->setState($this->context->getBrighterState());
    }
        public function turnBrightest()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnLightOff()
    {
        echo "<img src='lights/nada.png'>";
    }
}
?>
```

BrighterState

```
<?php
//BrighterState.php
class BrighterState implements IState
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function turnLightOn()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnBrighter()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnBrightest()
    {
        echo "<img src='lights/brightest.png'>";
        $this->context->setState($this->context->getBrightestState());
    }
    public function turnLightOff()
    {
        echo "<img src='lights/nada.png'>";
    }
}
?>
```

BrightestState

```
<?php
//BrightestState.php
```

```
class BrightestState implements IState
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function turnLightOn()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnBrighter()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnBrightest()
    {
        echo "<img src='lights/nada.png'>";
    }
    public function turnLightOff()
    {
        echo "<img src='lights/off.png'>";
        $this->context->setState($this->context->getOffState());
    }
}
?>
```

需要说明，各个状态的方法中有且仅有一个方法会建立负图像。不过，三路灯具正是采用这种方式使用三路灯泡。打开灯之后，它必须经过另外两个状态（*brighter*和*brightest*），才会最后关掉。

10.5.3 更新Context类

最后一步是更新Context类，增加新的触发器，并加入新状态。另外，Context还需要为每个新状态增加状态实例和获取方法：

```
<?php
//Context.php
class Context
{
    private $offState;
    private $onState;
    private $brighterState;
    private $brightestState;

    private $currentState;

    public function __construct()
    {
        $this->offState=new OffState($this);
        $this->onState=new OnState($this);
```

```
        $this->brighterState=new BrighterState($this);
        $this->brightestState=new BrightestState($this);

        //开始状态为Off
        $this->currentState=$this->offState;
    }
    //调用状态方法
    public function turnOnLight()
    {
        $this->currentState->turnLightOn();
    }
    public function turnOffLight()
    {
        $this->currentState->turnLightOff();
    }
    public function turnBrighterLight()
    {
        $this->currentState->turnBrighter();
    }
    public function turnBrightestLight()
    {
        $this->currentState->turnBrightest();
    }
    //设置当前状态
    public function setState(IState $state)
    {
        $this->currentState=$state;
    }

    //获得状态
    public function getOnState()
    {
        return $this->onState;
    }
    public function getOffState()
    {
        return $this->offState;
    }
    public function getBrighterState()
    {
        return $this->brighterState;
    }
    public function getBrightestState()
    {
        return $this->brightestState;
    }
}
?>
```

增加的代码与之前状态的方法以及实例化是一样的。尽管Context类增加了代码，不过与原来的代码很类似，只是有更多内容。

10.5.4 更新客户

在最初的例子中，Client可以请求两个状态*on*或*off*。增加两个状态后，Client有了更多选择，不过默认状态仍然是*off*，第一个请求必然是*on*状态。一旦建立*on*状态，接下来可以请求*brighter*状态，然后是*brightest*状态，之后才能再次请求*off*：

```
<?php
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    private $context;

    public function __construct()
    {
        $this->context=new Context();
        $this->context->turnOnLight();
        $this->context->turnBrighterLight();
        $this->context->turnBrightestLight();
        $this->context->turnOffLight();
        $this->context->turnBrightestLight();
    }
}
$worker=new Client();
?>
```

修改后的Client中，按正确的序列发出请求，首先变迁到*on*状态。不过，在经过各个不同状态回到*off*状态之后，Client请求返回*brightest*状态。此时，会出现“错误指示灯”。图10-5显示了请求序列得到的不同图像。

前面4个灯泡从左到右分别显示了on、brighter、brightest和off状态。不过，第5个灯泡指示请求出现错误。在典型的状态模式实现中，不会出现错误消息（或图像）。这个请求将被简单忽略。

10.6 导航工具：更多选择和单元格

图10-2显示了一个包含9个单元格的3 × 3 矩阵。如果将各个单元格当做地图上的一个方格位置来完成导航，这需要一组移动规则。可以考虑4个简单的移动或变迁：

- 上（Up）
- 下（Down）
- 左（Left）
- 右（Right）

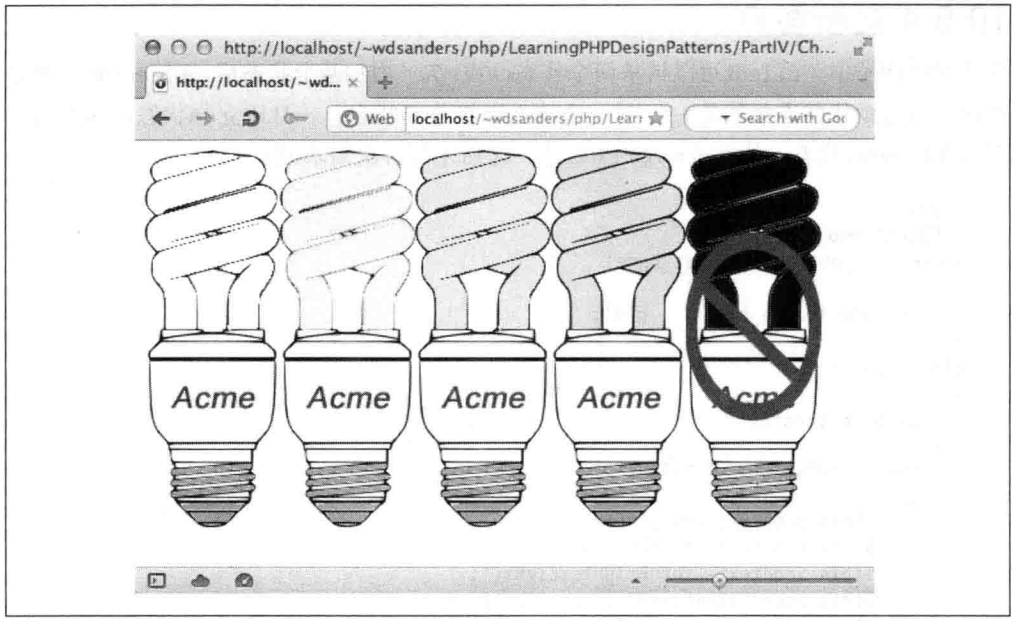


图10-5：更新的状态模式序列（含错误显示）

对于这些移动，规则要求单元格之间不能沿对角线方向移动。另外，规则还指出，从一个单元格移动到下一个单元格时，一次只能移动一个单元格。

在“何时使用状态模式”一节中提供了一系列条件语句，展示了在矩阵中完成导航的一种方法。不过，即使仅仅是这样一个3 × 3的简单矩阵，这种解决方案也会有大堆的条件和/或else子句，因此很快就会变得复杂而混乱。这个例子将展示如何把导航处理为一系列不同的状态。

10.6.1 建立一个矩阵状态图

要使用状态设计模式建立一个导航系统，与之前改变灯泡状态时一样，首先第一步是建立一个状态图。图10-6表示将要使用的矩阵，另外提供了改变状态（从一个单元格移动到下一个单元格）的规则。

除了有更多的变迁箭头，之前灯开关的*on/off*状态逻辑在这里也同样适用。举例来说，来看单元格2以及相应的改变状态的规则。它有以下移动选择：

- 上（不能上移）
- 下（下移至单元格5）

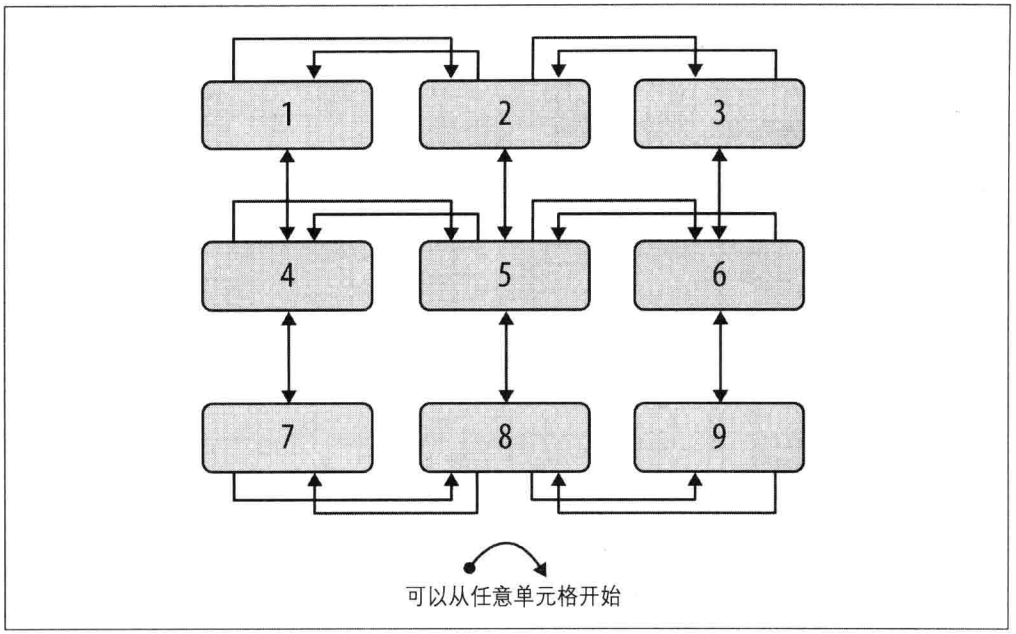


图10-6：矩阵导航状态图

- 左（左移至单元格1）
- 右（右移至单元格3）

查看任意一个单元格时，选择很清楚，现在考虑如何编写移动（或状态改变）会更有意义。与之前的*on*/*off*例子不同（其起点状态是*off*），这里的起点可以是任何单元格，如状态图中有开始箭头的标签所示。

10.6.2 建立接口

这个例子中的状态接口是`IMatrix`，反映了不同的移动而不是状态名。这里没有提供状态的名字，而是分别提供了一个变迁或移动名：

```
<?php
//IMatrix.php
//状态接口
interface IMatrix
{
    public function goUp();
    public function goDown();
    public function goLeft();
    public function goRight();
}
?>
```

尽管这个状态设计模式实现有9个状态，分别对应9个单元格，但一个状态最多只需要4个变迁。

10.6.3 上下文

对于状态中的4个变迁或移动方法，上下文必须提供相应方法来调用这些变迁方法，另外还要完成各个状态的实例化。这与只有两个状态的例子是类似的，只是有更多的方法和实例化：

```
<?php
//Context.php
class Context
{
    private $cell1;
    private $cell2;
    private $cell3;
    private $cell4;
    private $cell5;
    private $cell6;
    private $cell7;
    private $cell8;
    private $cell9;
    private $currentState;

    public function __construct()
    {
        $this->cell1=new Cell1State($this);
        $this->cell2=new Cell2State($this);
        $this->cell3=new Cell3State($this);
        $this->cell4=new Cell4State($this);
        $this->cell5=new Cell5State($this);
        $this->cell6=new Cell6State($this);
        $this->cell7=new Cell7State($this);
        $this->cell8=new Cell8State($this);
        $this->cell9=new Cell9State($this);

        //开始状态由开发人员来定
        $this->currentState=$this->cell5;
    }
    //调用状态方法
    public function doUp()
    {
        $this->currentState->goUp();
    }
    public function doDown()
    {
        $this->currentState->goDown();
    }
    public function doLeft()
    {
        $this->currentState->goLeft();
    }
```

```
    public function doRight()
    {
        $this->currentState->goRight();
    }

    //设置当前状态
    public function setState(IMatrix $state)
    {
        $this->currentState=$state;
    }
    //获得状态
    public function getCell1State()
    {
        return $this->cell1;
    }
    public function getCell2State()
    {
        return $this->cell2;
    }
    public function getCell3State()
    {
        return $this->cell3;
    }
    public function getCell4State()
    {
        return $this->cell4;
    }
    public function getCell5State()
    {
        return $this->cell5;
    }
    public function getCell6State()
    {
        return $this->cell6;
    }
    public function getCell7State()
    {
        return $this->cell7;
    }
    public function getCell8State()
    {
        return $this->cell8;
    }
    public function getCell9State()
    {
        return $this->cell9;
    }
}
?>
```

Context中的大部分代码都是用于为9个状态类分别创建获取方法。

10.6.4 状态

这9个状态表示3 × 3 矩阵中的不同单元格。为了唯一地显示各个单元格，会分别显示一个图像作为标签（有不同的数字和颜色）。这样一来，能够更清楚地看出穿过矩阵的路线。

这9个状态类都包含实现`IMatrix`接口所需的4个方法。在前面的状态例子中，如果一种选择不合法（如三路灯泡中从*on*状态直接变到*brightest*状态），并不是显示一个不合法的调用，这里只提供了注释行，指示代码中有一个不合法的移动。

Cell1State

```
<?php
//Cell1State.php
class Cell1State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        //不合法的移动
    }
    public function goRight()
    {
        echo "<img src='cells/two.png'>";
        $this->context->setState($this->context->getCell2State());
    }
    public function goUp()
    {
        //不合法的移动

    }
    public function goDown()
    {
        echo "<img src='cells/four.png'>";
        $this->context->setState($this->context->getCell4State());
    }
}
?>
```

Cell2State

```
<?php
//Cell2State.php
class Cell2State implements IMatrix
{
    private $context;
```

```
    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        echo "<img src='cells/one.png'>";
        $this->context->setState($this->context->getCell1State());
    }
        public function goRight()
    {
        echo "<img src='cells/three.png'>";
        $this->context->setState($this->context->getCell3State());
    }
    public function goUp()
    {
        //不合法的移动
    }
    public function goDown()
    {
        echo "<img src='cells/five.png'>";
        $this->context->setState($this->context->getCell5State());
    }
}
?>
```

Cell3State

```
<?php
//Cell31State.php
class Cell3State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        echo "<img src='cells/two.png'>";
        $this->context->setState($this->context->getCell2State());
    }
    public function goRight()
    {
        //不合法的移动
    }
    public function goUp()
    {
        //不合法的移动

    }
    public function goDown()
    {
        echo "<img src='cells/six.png'>";
```

```
            $this->context->setState($this->context->getCell6State());
        }
    }
    ?>
```

Cell4State

```
<?php
//Cell4State.php
class Cell4State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        //不合法的移动
    }
    public function goRight()
    {
        echo "<img src='cells/five.png'>";
        $this->context->setState($this->context->getCell5State());
    }
    public function goUp()
    {
        echo "<img src='cells/one.png'>";
        $this->context->setState($this->context->getCell1State());
    }

    public function goDown()
    {
        echo "<img src='cells/seven.png'>";
        $this->context->setState($this->context->getCell7State());
    }
}
?>
```

Cell5State

```
<?php
//Cell5State.php
class Cell5State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        echo "<img src='cells/four.png'>";
```

```
        $this->context->setState($this->context->getCell4State());
    }
    public function goRight()
    {
        echo "<img src='cells/six.png'>";
        $this->context->setState($this->context->getCell6State());
    }
    public function goUp()
    {
        echo "<img src='cells/two.png'>";
        $this->context->setState($this->context->getCell2State());
    }

    public function goDown()
    {
        echo "<img src='cells/eight.png'>";
        $this->context->setState($this->context->getCell8State());

    }
}
?>
```

Cell6State

```
<?php
//Cell6State.php
class Cell6State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        echo "<img src='cells/five.png'>";
        $this->context->setState($this->context->getCell5State());
    }
    public function goRight()
    {
        //不合法的移动
    }

    public function goUp()
    {
        echo "<img src='cells/three.png'>";
        $this->context->setState($this->context->getCell3State());

    }
    public function goDown()
    {
        echo "<img src='cells/nine.png'>";
        $this->context->setState($this->context->getCell9State());
    }
```

```
}
?>
```

Cell7State

```
<?php
//Cell71State.php
class Cell7State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        //不合法的移动
    }
    public function goRight()
    {
        echo "<img src='cells/eight.png'>";
        $this->context->setState($this->context->getCell8State());
    }

    public function goUp()
    {
        echo "<img src='cells/four.png'>";
        $this->context->setState($this->context->getCell4State());
    }
    public function goDown()
    {
        //不合法的移动
    }
}
?>
```

Cell8State

```
<?php
//Cell81State.php
class Cell8State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        echo "<img src='cells/seven.png'>";
        $this->context->setState($this->context->getCell7State());
    }
    public function goRight()
```

```
    {
        echo "<img src='cells/nine.png'>";
        $this->context->setState($this->context->getCell9State());
    }

    public function goUp()
    {
        echo "<img src='cells/five.png'>";
        $this->context->setState($this->context->getCell5State());
    }
    public function goDown()
    {
        //不合法的移动
    }
}
?>
```

Cell9State

```
<?php
//Cell9State.php
class Cell9State implements IMatrix
{
    private $context;

    public function __construct(Context $contextNow)
    {
        $this->context=$contextNow;
    }
    public function goLeft()
    {
        echo "<img src='cells/eight.png'>";
        $this->context->setState($this->context->getCell8State());
    }
    public function goRight()
    {
        //不合法的移动
    }
    public function goUp()
    {
        echo "<img src='cells/six.png'>";
        $this->context->setState($this->context->getCell6State());
    }
    public function goDown()
    {
        //不合法的移动
    }
}
?>
```

要想有效地使用状态设计模式，真正的难点在于要想象现实或模拟世界是怎样的。例如，如果你把矩阵中的各个单元格想象成一幢房子里的不同的房间，对于各个房间，就有不同的选择。要进入某些房间，可能首先要通过走廊或者先要进入另外一个房间。

10.6.5 客户选择一条路径

为了以可视化方式显示从一个状态到另一个状态的移动，图10-7显示了Client选择的一个路径。查看Client代码，把状态的改变看作是在矩阵中移动。

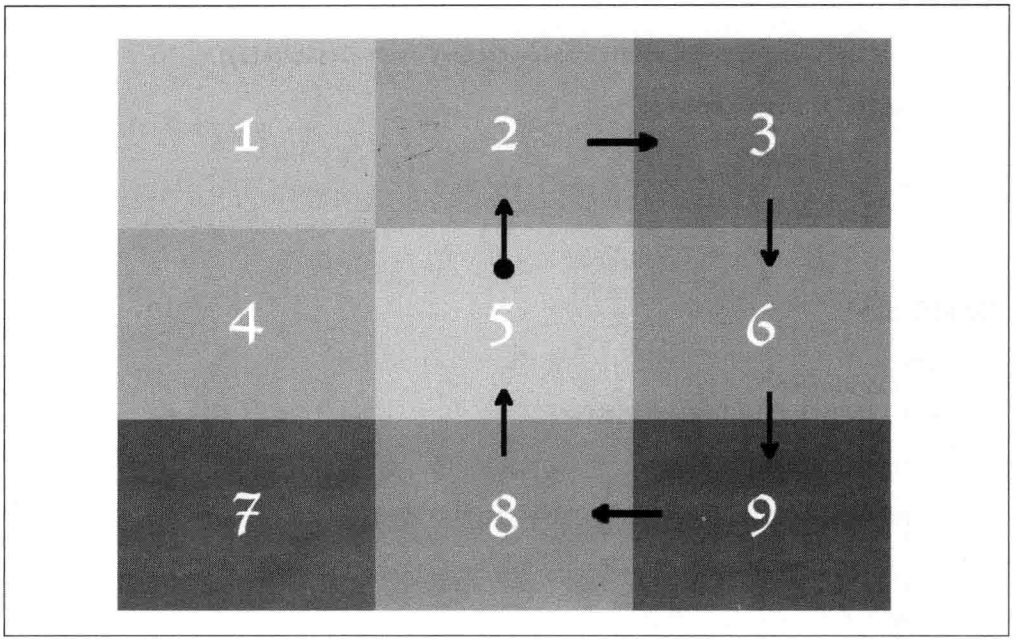


图10-7：经过不同状态得到的路径

从单元格5开始，路径包括以下移动：

1. 上移
2. 右移
3. 下移
4. 下移
5. 左移
6. 上移

Client按照这些移动（状态变迁）来创建指示的路径。将单元格5作为ConText类中的默认状态，Client会完成同样的移动：

```
<?php
//Client.php
function __autoload($class_name)
```

```
{
    include $class_name . '.php';
}
class Client
{
    private $context;

    public function __construct()
    {
        $this->context=new Context();
        $this->context->doUp();
        $this->context->doRight();
        $this->context->doDown();
        echo "<br/>";
        $this->context->doDown();
        $this->context->doLeft();
        $this->context->doUp();

    }
}
$worker=new Client();
?>
```

经过3个单元格之后，Client会给出一个换行（
），以防单元格超出浏览器屏幕边界。由图10-8可以看到程序如何显示这个路径。

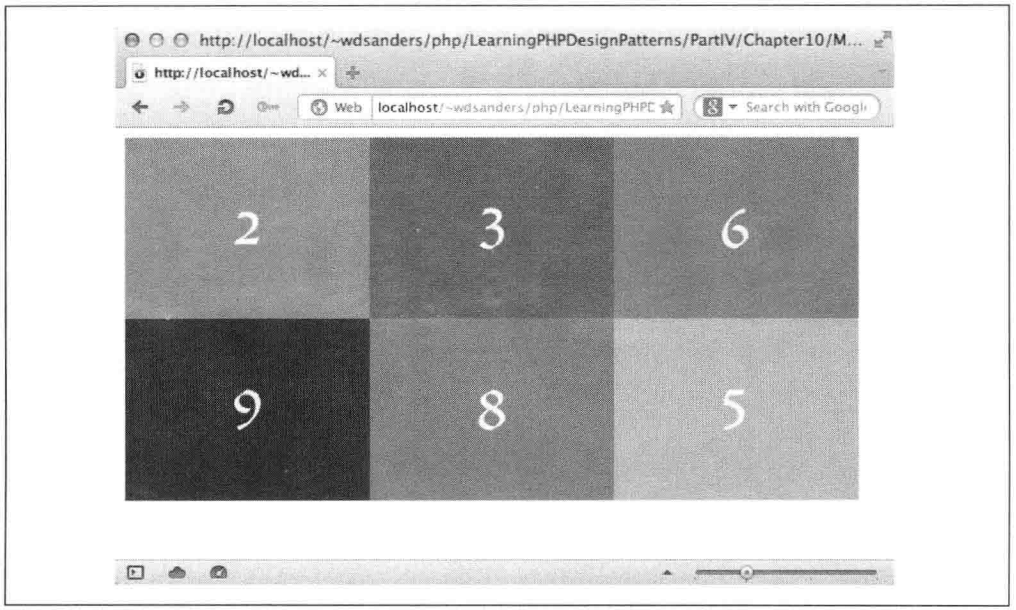


图10-8：客户“经过”的6个状态

将图10-8与图10-7比较，可以更好地看出（从左到右，从上到下）程序如何沿着矩阵中选定的路径进行显示。

10.7 状态模式与PHP

很多人把状态设计模式看做是实现模拟器和游戏的主要方法。总的说来，这确实是状态模式的目标，不过除此以外，状态模型（状态引擎）和状态设计模式在PHP中也有很多应用。用PHP完成更大的项目时，包括Facebook和WordPress，会有更多的新增特性和当前状态需求。Grady Booch（*http://ibm.co/yCL6se*）指出，Facebook一方面要保证网站更新，同时在以指数速度扩张时（即使有很多程序员也不能满足需要）仍要保持正常工作，要做到这一点难度越来越大。对于这种不断有改变和增长的情况，就可以采用可扩展的状态模式来管理。

PHP开发人员如何创建包含多个状态的程序，将决定状态模式的使用范围。所以不仅状态机在游戏和模拟世界中有很多应用，实际上状态模型还有更多适用的领域。只要PHP程序的用户会用到一组有限的状态，开发人员就可以使用状态设计模式。

第5部分

MySQL和PHP设计模式

我们必须兼顾蛇蝎般的强硬和鸽子般的温柔，
即坚定的信念和柔软的内心。
——马丁·路德·金

我的所有兴趣，包括推测和实际，
可以归结为以下3个问题：
1. 我能知道什么？
2. 我该怎么做？
3. 我希望怎么样？
——伊曼努尔·康德

坏人联手时，好人也必须协作；否则他们会一个接一个地倒下，
在这被人蔑视的斗争中，这只是一种无谓的牺牲。
——埃德蒙·伯克

MySQL在PHP设计模式中的角色

一些高级的MySQL程序员会结合关系数据库使用设计模式，第5部分中的这4章就将结合MySQL使用PHP设计模式。所以在接下来的4章中，你不需要放弃原来使用的数据库代码，也就是说，原先结合MySQL和PHP时使用的JOIN语句或其他关系数据库代码还能继续使用。

不过，结合PHP使用MySQL的情况相当普遍，如果没有一部分专门介绍如何结合MySQL使用OOP结构和设计模式，这绝对是一个严重的疏漏。另外，作为最后一部分，这里提供了一种增加更多PHP设计模式的方法。这些模式包括：

- 代理（Proxy）
- 策略（Strategy）
- 职责链（Chain of Responsibility）
- 观察者（Observer）

这些模式并非只能在结合PHP和MySQL的应用中使用，它们完全可以单独用于PHP。另外，通过后面给出的一个职责链例子，还可以看到一个应用中可以结合使用多个设计模式。

你还会看到使用`mysqli`处理连接的一个组合接口和类。这个“二人组”将在第5部分中一直使用，只要替换为你自己的连接信息（主机、用户名、密码和数据库），就可以利用这个组合接口和类建立你想要的任何MySQL连接。

尽管第5部分并没有真正的MySQL设计模式，不过你会发现一些可重用的结构，可以在其他结合了MySQL的PHP项目中使用。有经验的程序员都知道，即使只对PHP或MySQL代码做很小的改变，对于同时包含PHP和MySQL的程序来说，这也会带来可怕的影响。不过，你会发现良构的设计模式会妥善地处理各种变化，即使是在MySQL环境中也能从容应对。

第11章

通用类负责连接，代理模式保证安全

是艺术创造了生活，引发了兴趣，强调了重要性……，
我很清楚，艺术的力量和它的美妙是无可替代的。
——亨利·詹姆斯

每次想要写“very”时都把它换成“damn”；
编辑会把它删掉，这样一来，你的其他文字就能免遭打击。
——马克·吐温

科学就是以事实代替表象，用证明取代印象。
——约翰·拉斯金

11.1 一个简单的MySQL接口和类

PHP允许接口存储常量，在第2章中你已经看到一个这样的例子，实现接口的类都可以使用接口中存储的这些常量。连接到一个MySQL数据库时需要提供一组常量，这些常量就可以存储在接口中。使用PHP `mysqli`扩展包的例程提供了一些变形来建立连接，不过所有这些变形都需要主机、用户名、口令和数据库信息。一旦在类中建立了这个例程，只要程序需要一个MySQL连接，就可以重用这个例程。图11-1所示为这种设计的类图。

连接信息独立于客户，因为具体信息存储在接口的常量中。`UniversalConnect`类使用作用域解析操作符来访问存储在接口常量中的数据。

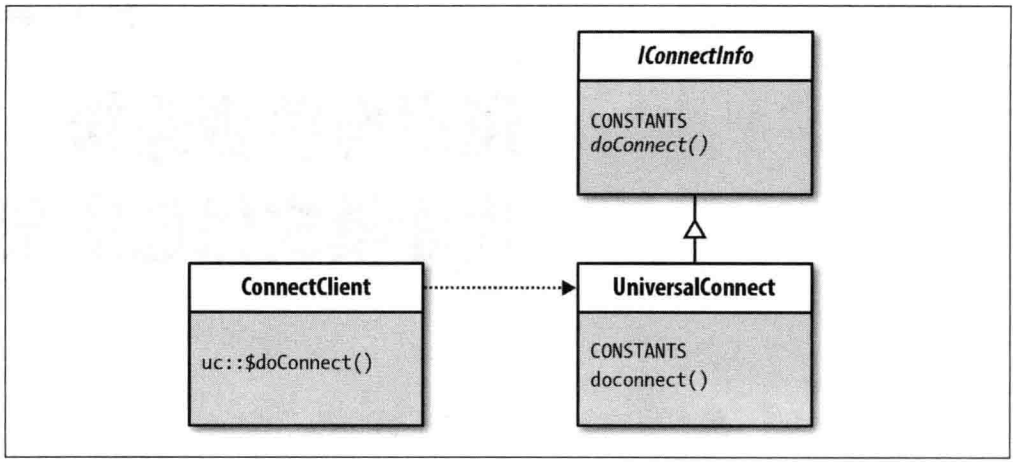


图11-1：MySQL通用连接类图

11.1.1 重要的接口

在设计模式中，接口通常允许开发人员创建松绑定的对象和类，不过这里有些不同，接口将用来存储数据——MySQL连接数据。所以，这个接口并不是一个用来组织操作的抽象结构，它包含一些具体的数据，实现这个接口的类都可以使用这些数据。

```
<?php
//文件名: IConnectInfo.php
interface IConnectInfo
{
    const HOST ="localhost";
    const UNAME ="phpWorker";
    const PW ="easyWay";
    const DBNAME = "dpPatt";

    public function doConnect();
}

?>
```

这样很容易隔离出这个接口进行修改，来指定你希望连接的任何主机。通过使用`mysqli`，可以为它提供这4个元素，建立与指定数据库的连接。通过替换接口，指定为类中定义的一组值，这样能减少开发所需的时间，因为通过这个接口可以隔离出程序中一个简单但必要的部分，任何程序都可以实现这个接口。

11.1.2 通用MySQL连接类和静态变量

所有`IConnectInfo`接口实现都使用作用域解析操作符来访问连接值。为了从接口向一个

实现类传递值并使用这个值，可以为变量赋一个私有静态状态，这有两个好处，一方面可以体会静态变量处理迅速的速度优势，另外还可以得到私有可见性提供的封装性：

```
<?php
include_once('IConnectInfo.php');

class UniversalConnect implements IConnectInfo
{
    private static $server=IConnectInfo::HOST;
    private static $currentDB= IConnectInfo::DBNAME;
    private static $user= IConnectInfo::UNAME;
    private static $pass= IConnectInfo::PW;
    private static $hookup;

    public function doConnect()
    {
        self::$hookup=mysqli_connect(self::$server, self::$user, self::$pass,
                    self::$currentDB);
        if(self::$hookup)
        {
            //调试时删除下一行前面的两个斜线
            //echo "Successful connection to MySQL:";
        }
        elseif (mysqli_connect_error(self::$hookup))
        {
            echo('Here is why it failed: ' . mysqli_connect_error());
        }
        return self::$hookup;
    }
}
?>
```

除了为连接变量指定静态状态值，UniversalConnect类中的mysqli变量（$hookup）也是静态的。这样一来，Client类做出请求时，就可以在一个更大的静态上下文中使用这个变量。

全局变量一定不好吗？

我总会尽量避免使用全局变量。全局变量很可能会破坏封装——它们总是横冲直撞，在程序中带来各种各样的麻烦。要在较大的程序中一路追查各种小问题，这会尤其麻烦。在某些方面，静态变量与全局变量有一些共同的特性，因此除非有很充分的理由，否则我不会使用全局变量。不过由于不断变化的编程环境的速度问题，最近我有了很多很好的理由来使用全局变量，但具体使用时我还是很谨慎。另外，如果需要反复实例化一个类，静态变量将有助于减少类的实例化。

有些人可能已经注意到，这本书并没有提供单例（Singleton）设计模式的例子。对此一个重要的原因是：如果正确地实现单例设计模式，它就相当于全局变量。

如今已经开发了很多实现有问题的单例设计模式，它们往往是无害的，不过还有一些实现正确的单例模式，这些则可能导致全局变量类型问题。有一位开发人员Miško Hevery把单例称为“病态的说谎者”（见*http://bit.ly/cI6mll*）。甚至《设计模式》一书的首席作者Erich Gamma也不赞成把单例作为一种设计模式（见*http://bit.ly/3Lg5IN*）。之所以避免使用单例，主要原因就在于它们相当于全局变量。

所以尽管我不认为这个例子中使用静态变量会导致问题，不过如果你很关心全局变量可能带来的影响，可以从所有代码中删除`static`关键字，另外有`private`可见性的变量都使用`$this->`格式。

使用静态变量最大的好处可能是可以轻松地使用这些静态变量的值，而不必每次程序中某个部分需要创建一个新连接时都实例化`UniversalConnect`类。例如，每次通过一个在线HTML表单为一个数据库表输入新数据时，都需要一个新连接。这说明，每个数据输入都会带来一个新的实例化，而使用静态变量就可以避免不断重新实例化连接类。

11.1.3 简单客户

在这里，客户是需要一个MySQL连接的程序。在一个典型应用中，客户可以是任何需要MySQL连接的具体类。除了一个`include_once()`操作来访问`UniversalConnect`类外，连接例程只有一行代码：

```
<?php
include_once('UniversalConnect.php');

class ConnectClient
{
    private $hookup;

    public function __construct()
    {
        //一行完成整个连接操作
        $this->hookup=UniversalConnect::doConnect();
    }
}

$worker=new ConnectClient();
?>
```

通过一个类和接口来建立一个简单的连接操作，这样可以大大减少开发时间。修改很容易，因为所有信息都存储在常量中。要改变主机、用户、口令或数据库名，只需要修改接口中相应的常量值。`UniversalConnect`类或使用`UniversalConnect`类的程序则不用做任何修改。

11.2 保护代理完成登录

代理（Proxy）模式是一种结构型设计模式。“四人帮”提出了4种代理模式：

远程代理（Remote proxy）

代理对象在一个地址空间，而实际对象在另一个地址空间中，此时代理就是远程的。除了使用远程代理作为防火墙，远程代理还可以用于在线游戏，在这种情况下，不同地方可能同时需要相同的代理对象。

虚拟代理（Virtual proxy）

虚拟代理可以缓存一个真实主题的有关信息，从而能延迟对这个真实主题的访问。有时在真实对象处理登录数据之前，高安全性登录会使用一个虚拟代理来完成登录。

保护代理（Protection proxy）

保护代理只有在验证过请求之后，才会把请求发送到真实主题。这个真实主题就是请求的目标，如访问数据库信息。根据用户的登录信息，很多保护代理会提供不同级别的访问；并不是建立一个真实主题，真实主题可能是多个，而且是受限的。

智能引用（A smart reference）

程序可以使用GoF所称的“裸指针”，不过，如果“裸指针”不能满足程序的需要，代理就相当于一个智能引用（或智能指针），可以在引用对象时完成额外的动作。例如，可能首先由作为智能引用的代理参与者加载一个数据库的数据。

代理模式有两个主要的参与者：一个代理主题（proxy subject）和一个真实主题（real subject）。客户通过*Subject*接口向Proxy提交请求，不过只有当请求首先通过Proxy之后才有可能访问`RealSubject`。图11-2显示了代理设计模式的类图。

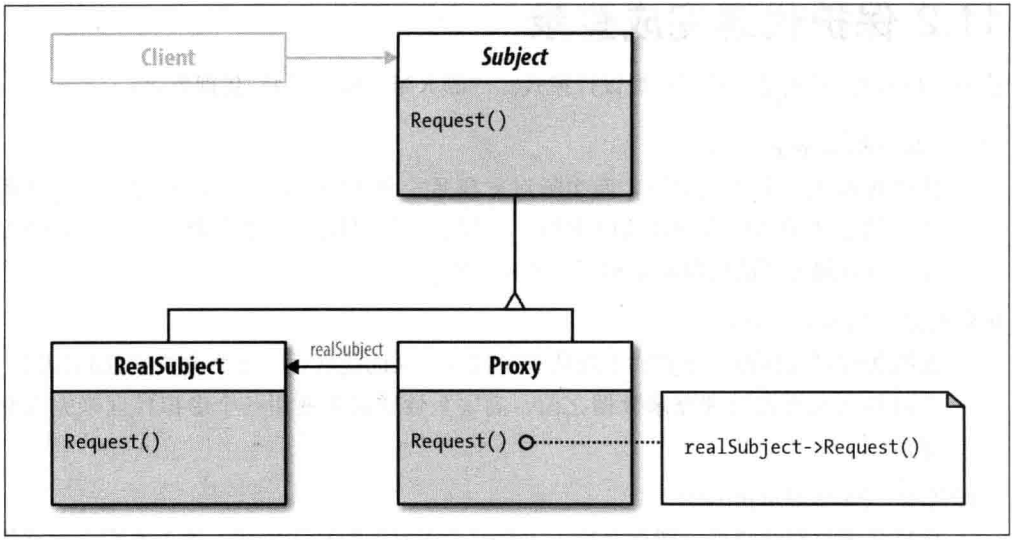


图11-2：代理模式类图

Proxy和RealSubject参与者都实现了Subject接口，不过Proxy的角色是一个“守门人”或封装模块。如果请求经由Proxy确认，Proxy再调用RealSubject为用户提供请求的内容。

11.2.1 建立登录注册

在这个程序中，要创建一个代理来完成需要注册的登录，首先建立一个表，其中只包含用户名和口令字段，除此以外没有其他信息。这是一个很简单的例子，可以进一步扩展，不过现在的重点是代理设计模式。下面的PHP脚本会创建存储用户信息的数据库表：

```
<?php
include_once('UniversalConnect.php');
class CreateTable
{
    private $tableMaster;
    private $hookup;

    public function __construct()
    {
        $this->tableMaster="proxyLog";
        $this->hookup=UniversalConnect::doConnect();

        $drop = "DROP TABLE IF EXISTS $this->tableMaster";

        if($this->hookup->query($drop) === true)
        {
            printf("Old table %s has been dropped.<br/>",$this->tableMaster);
        }
```

```
        $sql = "CREATE TABLE $this->tableMaster (uname NVARCHAR(15),
                pw NVARCHAR(120)";

        if($this->hookup->query($sql) === true)
        {
            echo "Table $this->tableMaster has been created successfully.<br/>";
        }
        $this->hookup->close();
    }
}
$worker=new CreateTable();
?>
```

这是一个很标准的类，用来创建存储用户名和口令的表。由口令的字符数NVARCHAR(120)可以看出，掩码口令最多为120个字符。要输入数据，下面的PHP会接收从一个HTML表单发送的数据：

```
<?php
include_once('UniversalConnect.php');
class HashRegister
{
    public function __construct()
    {
        $this->tableMaster="proxyLog";
        $this->hookup=UniversalConnect::doConnect();
        $username=$this->hookup->real_escape_string(trim($_POST['uname']));
        $pwNow=$this->hookup->real_escape_string(trim($_POST['pw']));

        $sql = "INSERT INTO $this->tableMaster (uname,pw) VALUES ('$username',
                md5('$pwNow'))";

    if($this->hookup->query($sql))
    {
        echo "Registration completed:";
    }

    elseif ( ($result = $this->hookup->query($sql))===false )
    {
        printf("Invalid query: %s <br/> Whole query: %s <br/>",
                $this->hookup->error, $sql);
          exit();
        }
        $this->hookup->close();
    }
}
$worker=new HashRegister();
?>
```

MD5函数将掩码作为一个32字符的十六进制数返回。md5()函数是一个比较原始但也比较简单的散列函数，这里使用这个散列函数只是为了说明一点：建立将在代理登录中使用的注册口令时可能会完成加密。

可以用下面的HTML5脚本完成注册——增加一个用户名和口令：

```
<!DOCTYPE HTML>
<html>
<head>
<link rel="stylesheet" href="proxy.css">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Username/Password Registration</title>
</head>
<body>
<header>Registration</header>
<section>
  <article id="entry">
    <form action="HashRegister.php" method="post">
      Username: (15 Characters Max-No spaces)<br/>
      <input type="text" name="uname" maxlength="15">
      <br/>
      Password: (10 Characters Max-No spaces)<br/>
      <input type="password" name="pw" maxlength="10">
      <br/>
      <input type="submit" value="Register">
    </form>
  </article>
</section>
</body>
</html>
```

最后给出有些文件会用到的CSS文件，这个CSS文件也很简单：

```
@charset "UTF-8";
/* CSS Document */
/*EFECCA,046380,002F2F */

body {
    margin-left:20px;
    font-family:Verdana, Geneva, sans-serif;
    background-color:#EFECCA;
}
header {
    font-family: "Arial Black", Gadget, sans-serif;
    font-size:24px;
    color:#002F2F;
}
#entry {
    font-size:11;
    color:#046380;
}
.subhead
{
    font-size:16px;
    font-family:Verdana, Geneva, sans-serif;
}
```

图11-3显示了用户在注册过程中看到的界面。（HTML文件都是分开的，这是为了更清楚地理解使用代理设计模式登录网站时不同元素在建立和执行登录时的作用。也可以把登录注册和登录本身都放在一个HTML5页面中，其中包含多个表单，一个表单用于注册，另一个用于登录。）

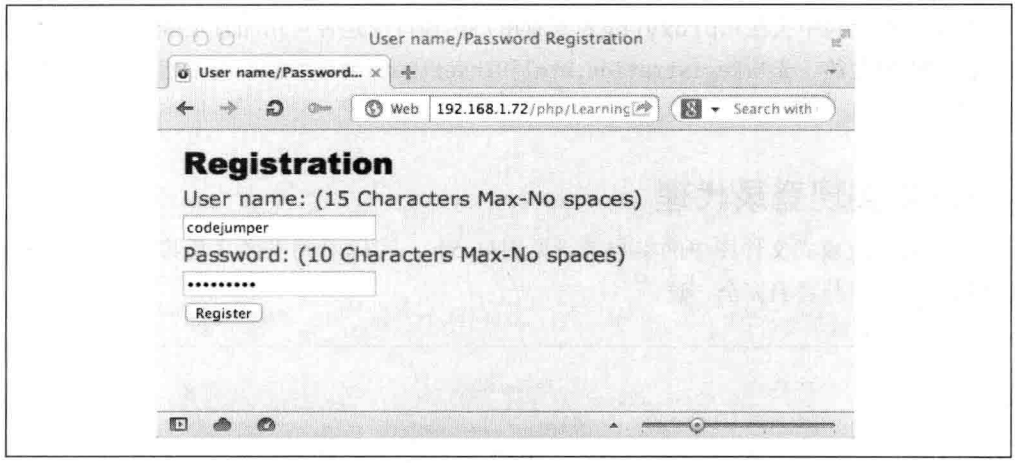


图11-3：注册屏幕

一旦用户注册，会有一个“确认”消息显示一个注册完成，如图11-4所示。

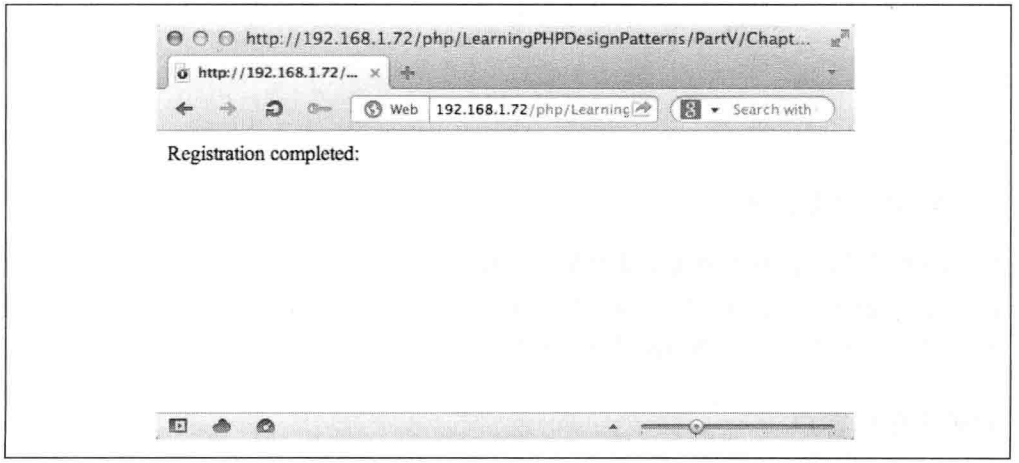


图11-4：注册确认

这个确认消息很简短，也可以嵌入到HTML代码中，并提供一个选项，提示用户现在就完成登录，不过，正如前面提到的，代码中所有部分都是模块化的，从而保证逻辑更为清晰。

两个PHP文件都使用UniversalConnect类来链接MySQL数据库，在这两个文件中，只使用一行代码就可以完成数据库的连接，而无需编写大堆的代码。

```
$this->hookup=UniversalConnect::doConnect();
```

为了测试这个代理模式实现，下面用不同口令“注册”多个用户。下一节将构建一个代理登录应用，其中会使用proxyLog表查看用户名和口令是否可用而且正确。所以在学习后面的内容之前，先用Registration.html和InsertData类在数据库表中增加一些用户名和口令。

11.2.2 实现登录代理

通过查看代理模式文件图中的参与者（见图11-5），可以看到类名（及其文件名）与设计模式中的参与者名完全一致。

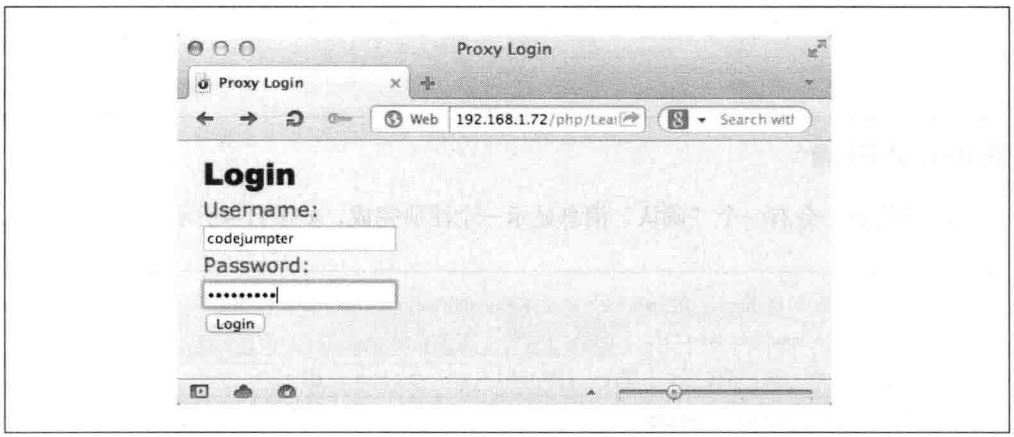


图11-5：登录代理文件图

Proxy类调用MySQL数据库来确保用户名和口令匹配，如果确实匹配，请求将传递到RealSubject类。所以尽管IconnectInfo接口和UniversalConnect类不属于这个设计模式，它们对于“守门人”角色却是很重要的组件。

登录表单和客户

用户在HTML文件的表单中放入登录信息。一旦用户输入数据，并点击提交按钮，信息会发送到Client：

```
<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

```
<title>Proxy Login</title>
<link rel="stylesheet" href="proxy.css">
</head>
<body>
<header>Login</header>
<section>
  <article id="entry">
    <form action="Client.php" method="post">
      Username: <br/>
      <input type="text" name="uname" >
      <br/>
      Password: <br/>
      <input type="password" name="pw" maxlength="10">
      <br/>
      <input type="submit" value="Login">
    </form>
</article>
</section>
</body>
</html>
```

最初的开发设计是由HTML表单将数据直接发送到Proxy类，不过经过重新考虑，完全可以利用Client来封装口令和用户名。Client还可以对HTML表单的数据进行修剪和过滤。Client使用一个私有方法（包含类型提示），将HTML表单的数据传递到Proxy类中的登录方法：

```
<?php
include_once("Proxy.php");
class Client
{
    private $proxy;
    private $un;
    private $pw;

    public function __construct()
    {
        $this->tableMaster="proxyLog";
        $this->hookup=UniversalConnect::doConnect();
        $this->un=$this->hookup->real_escape_string(trim($_POST['uname']));
        $this->pw=$this->hookup->real_escape_string(trim($_POST['pw']));
        $this->getIface($this->proxy=new Proxy());
    }

    private function getIface(ISubject $proxy)
    {
            $proxy->login($this->un,$this->pw);
    }
}
$worker=new Client();
?>
```

由于用户名和口令解装到一个私有变量中，并由一个私有方法（getIface）发送到Proxy

类，所以可以得到有效的封装，然后才会在Proxy类中完成比较以保证用户名和口令合法。

代理的工作

通过Client将HTML表单发送的数据传递到Proxy类时，必须将这个数据与MySQL表中存储的数据进行比较。由于Proxy和RealSubject类有一个共同的接口，它们必须分别实现ISubject接口：

```
<?php
interface ISubject
{
    function request();
}
?>
```

可以采用很多不同的方式来实现方法request。对于Proxy，如果用户名和口令得到验证，可以通过它将原始请求发送到RealSubject类。换句话说，Proxy要确定是否调用RealSubject的方法request。

设计代理时，可以通知用户口令和用户名是非法的，除此以外不做其他处理。不过也可以让用户返回登录页面或注册页面：

```
<?php
include_once("ISubject.php");
include_once('RealSubject.php');
include_once('UniversalConnect.php');
class Proxy implements ISubject
{
    private $tableMaster;
    private $hookup;
    private $logGood;
    private $realSubject;

    public function login($uNow,$pNow)
    {
        //由客户过滤；对口令掩码
        $uname=$uNow;
        $pw=md5($pNow);
        $this->logGood=false;
        //选择表和连接
        $this->tableMaster="proxyLog";
        $this->hookup=UniversalConnect::doConnect();

        //创建MySQL语句
        $sql = "SELECT pw FROM $this->tableMaster WHERE uname='$uname'";

        if($result=$this->hookup->query($sql))
        {
            $row=$result->fetch_array(MYSQLI_ASSOC);
            if($row['pw']==$pw)
```

```
            {
             $this->logGood=true;
            }
        $result->close();
        }
        elseif ( ($result = $this->hookup->query($sql))===false )
        {
            printf("Failed: %s <br/>", $this->hookup->error);
            exit();
        }
        $this->hookup->close();

        if($this->logGood)
        {
            $this->request();
        }
        else
        {
            echo "Username and/or Password not on record.";
        }
    }
    public function request()
    {
        $this->realSubject=new RealSubject();
        $this->realSubject->request();
    }
}

?>
```

大多数工作都是由login方法完成的。它接收口令和用户名，用md5()函数对口令计算散列，打开数据库，并查询数据表。如果发现一个匹配的用户名和口令对，则设置$logGood变量为true。如果$logGood为true，则调用request方法；否则只给出一个消息。

Proxy的request方法会调用RealSubject的request方法。在代理模式中，真实主题（请求的目标）的代理是Proxy类，所以它的任务是掩护真实主题，而不是重复真实主题的request方法。

真实主题

代理设计模式的关键是代理参与者。不过，用户想要的实际上是真实主题里的内容，所以对用户来说，只有真实主题是最重要的。可以告诉你一个好消息，真实主题几乎可以是任何东西。在这个例子中，真实主题是一个页面，其中包含一些OOP和设计模式技巧：

```
<?php
include_once('ISubject.php');
```

```
class RealSubject implements ISubject
{
    public function request()
    {
    $practice=<<<REQUEST
    <!DOCTYPE html>
    <html>
        <head>
        <meta charset="UTF-8">
        <link rel='stylesheet' type='text/css' href='proxy.css' />
        </head>
        <body>
        <header>PHP Tip Sheet:<br>
          <span class='subhead'>For OOP Developers</span></header>
        <ol>
               <li>Program to the interface and not the implementation.</li>
               <li>Encapsulate your objects.</li>
               <li>Favor composition over class inheritance.</li>
               <li>A class should only have a single responsibility.</li>
             </ol>
        </body>
    </html>
REQUEST;
echo $practice;
    }
}
?>
```

其中只有一个方法，这是实现ISubject接口时必须实现的一个方法——request。这个request方法只生成一些HTML，会显示一个有序列表。它也可以是任何内容，在这个模式中它有特定的位置，相当于火车最后一节的守车，没有反向影响。

从用户的角度来看，所有类和代码都是透明的。图11-6显示了用户在HTML5页面上登录时看到的屏幕。

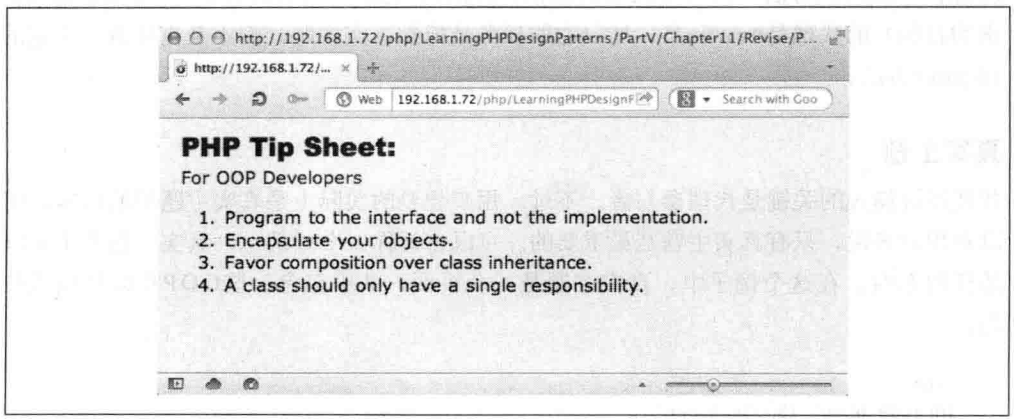


图11-6：HTML5登录模块

登录模块把请求发送到Client类，Client类从HTML5表单获得登录数据，并进一步通过Proxy login方法将请求转发给Proxy。当然，用户不会看到这些。用户看到的只是RealSubject 输出，如图11-7所示。

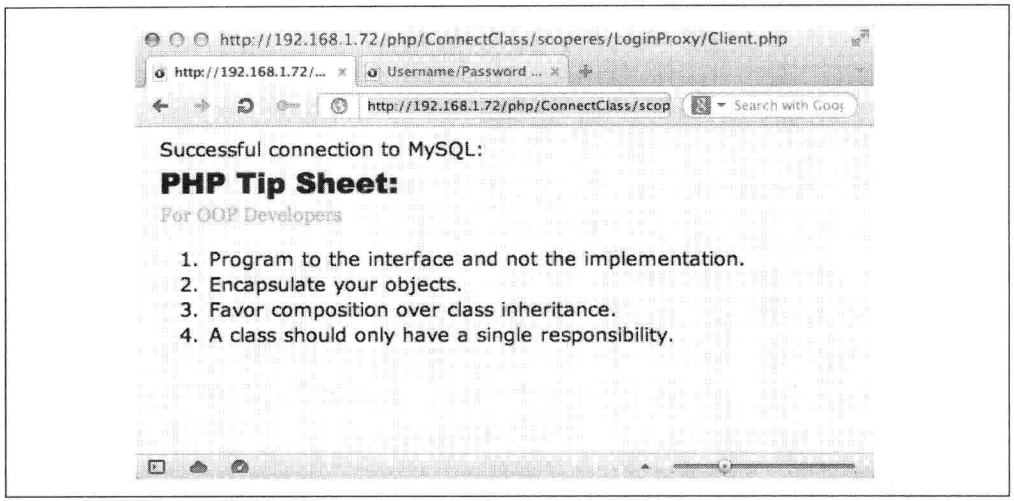


图11-7：RealSubject的输出

当然，RealSubject类的内容就是用户原本想看到的内容，用户根本不知道初始请求必须成功地通过Client和Proxy类。

如果输入了不正确的用户名和/或口令，或者用户根本没有注册，而导致请求无法完成，必须发送一些替代消息。在这里，可能会有一个类似图11-8的简单消息。通知用户无法找到他输入的数据。

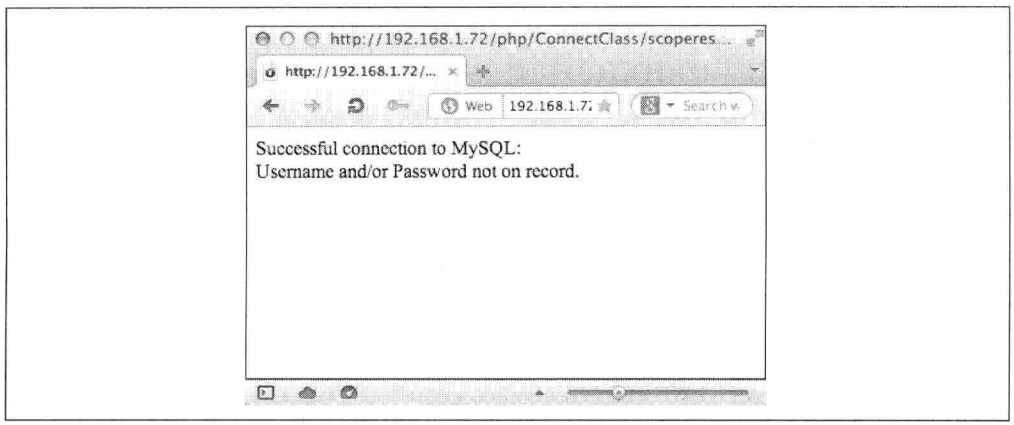


图11-8：Proxy发送的消息指示请求未找到

一般地，为用户提供消息时，会建议他注册，或者在登录时更仔细地输入信息，也可能会链接到另外一个页面，允许用户在这个页面中选择注册或者再次登录。不过，为清楚起见，这个例子力求最简。一旦熟悉了如何使用代理设计模式实现一个保护代理，完全可以稍做修改来建立无缝的注册和登录体验。毕竟，网站就是为了让用户更容易访问，代理的作用只是确保有合法权限的用户才能使用网站。

11.3 代理和真实世界安全

就其本身而言（特别是如本章例子所示），对于真实世界的网站，代理模式肯定是不够安全的。如果涉及金融交易，代理模式更无法保证其安全性。不过，如果一个代理模块把请求转发到一个高安全性模块，这仍保持了代理设计模式。实际上，可以把这个模式中的代理参与者看作是一个场所，在用户访问到真实主题之前可以在这里做一些真正保证高安全性的操作。在这个例子中，`md5()`函数显示了可以在这里生成一个口令散列码，不过MD5并不是一个高安全性的加密方法。可以为你的网站增加更多安全性。例如，在完成信用卡金融业务处理时，代理会把请求发送到一个信用卡处理公司的模块，负责完成金融业务，如果处理返回的结果指示交易成功，则会继续将请求传给真实主题，由真实主题本身为用户提供可用的产品或服务。

不过，绝对不要认为代理设计模式（不论是使用PHP还是任何其他编程语言）是一个高安全性的设计。实际上，它只是一个提供安全性的通用模型，安全性的细节要由代理中或代理之外的另一个模块处理。图11-9描述了真实世界高安全性环境中代理的角色。

对用户来说，所有请求都直接发送给真实主题。代理模块可以是一个简单的口令检查，也可以调用一个高安全性模块来处理敏感信息。

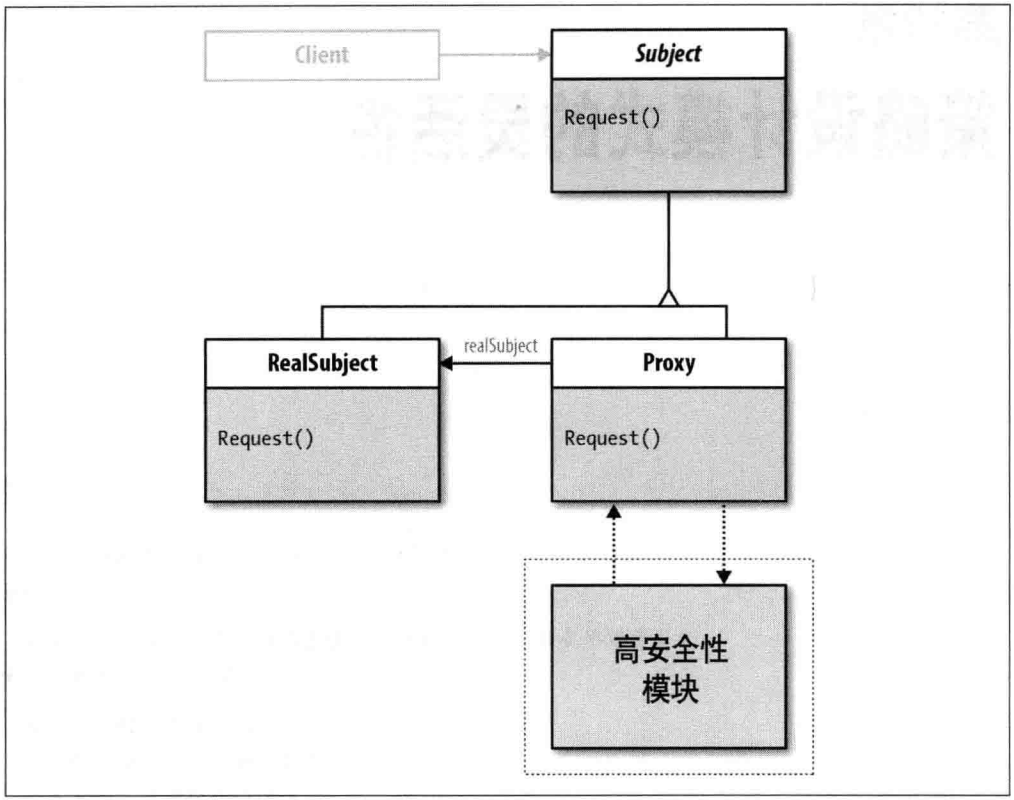


图11-9：为代理模式增加一个高安全性模块

第12章

策略设计模式的灵活性

是故胜兵先胜而后求战，败兵先战而后求胜。

——孙子

为战争做准备时，我经常发现计划毫无用处，不过规划仍不可或缺。

——德怀特·戴维·艾森豪威尔

我所做的就是改进他们的策略，
这可以启迪生命中一个重要的教训，
如果你发现有人比你更成功，
尤其是你们同在一个行业，要知道他们
肯定做到了你没有做到的事情。

——马尔克姆·X

12.1 封装算法

结合MySQL使用PHP时，要完成的任务之一是需要编写算法来处理对MySQL应用做出的不同类型的请求。一般的请求包括创建表，或者输入、选择、修改和删除数据。对于这些不同的请求，算法可能很简单，也可能很复杂，这取决于请求以及表的复杂性。

设计模式的主要原则之一是封装变化的内容。对于发送到PHP类的不同类型的请求，分别有不同的算法来处理这些MySQL请求，变化的显然是算法。这些变化可能很小，也可能是显著的变化，不过通过使用策略（Strategy）设计模式，我们可以大大简化这个过程。

一般来讲，使用设计模式时，首先要考虑“什么会导致重新设计”？然后要避免那些导致重新设计的因素。不过，有没有一种方法既能做出改变又无需重新设计呢？通过封

装那些变化的内容，程序员应首先确定程序中哪些会变化，然后封装这些特性。这样一来，一个设计需要改变时，可以改变封装的元素，而不会影响系统的其余部分。由于不同的MySQL任务需要不同的算法，可以封装这些算法（任务），所以策略设计模式非常适用。

12.1.1 区分策略和状态设计模式

首先来看图12-1，它是策略设计模式的类图。

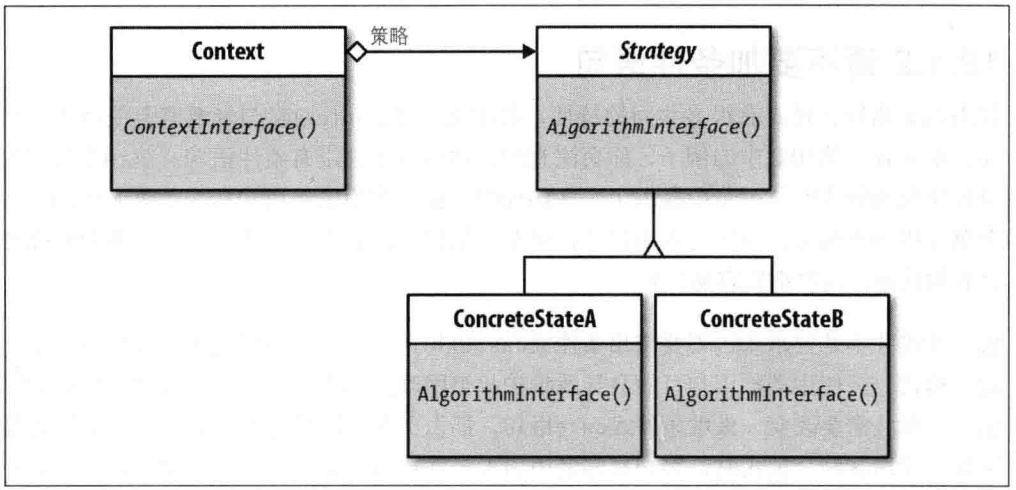


图12-1：策略设计模式类图

再回到第10章，看看图10-1。其中模式参与者的组织与图12-1中非常相似。在两个设计中，Context参与者都与一个接口有一个聚合关系。在状态模式中，接口为State，在策略模式中则是Strategy接口。除此以外，它们看起来是一样的。

要了解二者的差别，需要了解两个Context参与者与Strategy和State接口及其具体实现的关系有什么不同。表12-1对这些差别做了一个总结。

表12-1：状态和策略设计模式中上下文和变化的差别

模式	什么会变化？	上下文
状态	状态	维护子类当前状态的一个实例（定义了当前状态）。
策略	算法	配置为具体策略对象，这是一个封装的算法。

在第10章可以看到，Context包含一个变量，保存了当前的具体状态。这个具体状态提供了一些方法，可以从Context变量中记录的当前状态变迁到另一个状态。

不过，策略模式中的Context参与者并没有记录当前使用的策略。它没有理由这么做，这是因为，与不断改变的状态不同，一般来讲，改变的算法并不依赖于当前正在使用的算法。显然，有些情况下，执行一个算法之前可能首先要使用另一个算法，如试图访问一个表中的数据之前，需要先在表中插入数据。不过，这并不妨碍使用算法尝试从一个空表获取数据。但在状态模式中很容易出现这样一种情况：即一个状态只能进入某些状态，而不能转移到另外一些状态。在第10章中的三路灯泡例子中，如果灯泡处于第二个*on*状态，就不能进入第一个*on*状态，也不能进入*off*状态。它只能转移到第3个*on*状态。（见第10章中的图10-5）大多数算法都不是这样。

12.1.2 请不要加条件语句

状态和策略设计模式有很多共同的特性，其中之一是Context参与者要避免使用条件语句。如果查看第10章中的例子，你会注意到，这些例子都没有条件语句。第10章的“何时使用状态模式？”一节中提供了一些伪代码，指出使用条件语句从一个单元格到下一个单元格难度很大。（不过不用担心，要在一组已有的状态中增加新状态，或者要改变已有的状态，这些都很容易！）

设计模式并不是要求永远不要使用条件或case语句，不过在一些模式中（如状态和策略设计模式），使用条件语句可能会导致维护相当困难。如果要改变一个策略（封装的算法），而且需要改变一大堆条件或case语句，那么引入错误的可能性会更大。另外需要说明，使用这两个模式时，客户参与者中引入条件或case语句是可以接受的，因为客户所要做的就是做出请求。另外，在封装的算法中（具体策略），完成某个任务可能需要一个条件或case语句。类似地，在使用`mysqli`实现数据输出和错误检查时，往往都会有条件语句。利用策略，不再需要条件语句来选择所需的行为。不同的任务由不同的具体策略来处理，因为客户通过上下文请求具体策略，它必须知道有哪些可用的策略。这并不是说客户选择过程中不能使用条件语句，而是说条件语句不是上下文的一部分。

12.1.3 算法族

GoF的《设计模式》中，有些元素的介绍不太详细，其中就包括“算法族”（family of algorithms）的概念。开发人员需要定义一个算法族，但GoF并没有指定设计模式上下文中“族”（family）一词的确切含义。不过，在《Head First Design Patterns》一书中，Eric和Elizabeth Freeman提出了一个很简单但很有用的概念——行为集（a set of behaviors）。任何依赖于一组特定行为的项目都可以转换为一种策略设计模式，将那些行为封装为策略。也就是说，需要某种算法才能实现这些行为。通过将它们封装为具体的策略，就能使用、重用和完成修改。

这一章中的“族”由使用MySQL表通常所需的行为组成。一般需要输入、修改、获取和

删除数据。这些涉及数据操作的行为就构成了一个“族”，每个族成员可以转换为一个策略。实现这些策略需要不同的算法，还要结合使用MySQL命令和PHP mysqli类。把这些操作放在不同的具体类中（所有具体类实现一个公共接口），就成为了策略设计模式的一部分。

12.2 最简单的策略模式

为了查看使用MySQL连接的一般模式，第一个例子中并没有使用表，而是建立了一个一般模式，以便以后加入各个策略中的细节。由于HTML表单不能把选择的参数传递给PHP类或文件，这个例子会使用多个短小的PHP触发器脚本。这些触发器脚本调用客户的不同方法，客户再通过上下文调用所请求的具体策略。

图12-2显示了这个实现的文件图。对于每个策略，HTML文件中分别包含相应的表单，表单数据通过一个PHP触发器脚本传递到Client中的方法。Client再通过Context进一步将请求传递到一个具体策略。连接辅助类包括一个接口和类，可以用来连接一个MySQL数据库。

有些开发人员在处理MySQL数据库方面有着丰富的经验，对他们来说，图12-2看起来可能有些“过度设计”。不过，设计模式的目的是为了便于修改和重用对象。每个具体策略会得到封装，因此只要保持所实现的接口不变，对具体策略的任何修改都不会破坏系统。即使从这个最小策略的角度来看，也很容易看出具体策略中的行为是开放的，允许各种不同的实现。

12.2.1 客户和触发器脚本

Client类通过Context做出请求，来创建一个具体策略。这里利用了一组方法来完成对不同策略的请求。请求的关键是下面这两行代码：

```
$context=new Context(new ConcreteStrategy());
$context->algorithm();
```

每个Client方法提供了要实现的具体策略的名，algorithm()是具体策略中实现的一个Context方法。这个过程可以展示多态（polymorphism）是如何工作的。Client通过一个Context实例发出请求，来请求具体方法的算法，所以所有算法请求看起来都很像，都是$context->algorithm()。不过，Client要实例化Context，并提供一个具体策略作为参数。通过实现具体策略的algorithm()方法，这个参数允许Context使用所请求的具体策略。这样一来，通过采用策略模式，算法就可以独立变化，而不影响使用该算法的客户。在下面的例子中，没有使用多个不同的Client类，而是利用多个不同的触发器脚本来使用相同的客户：

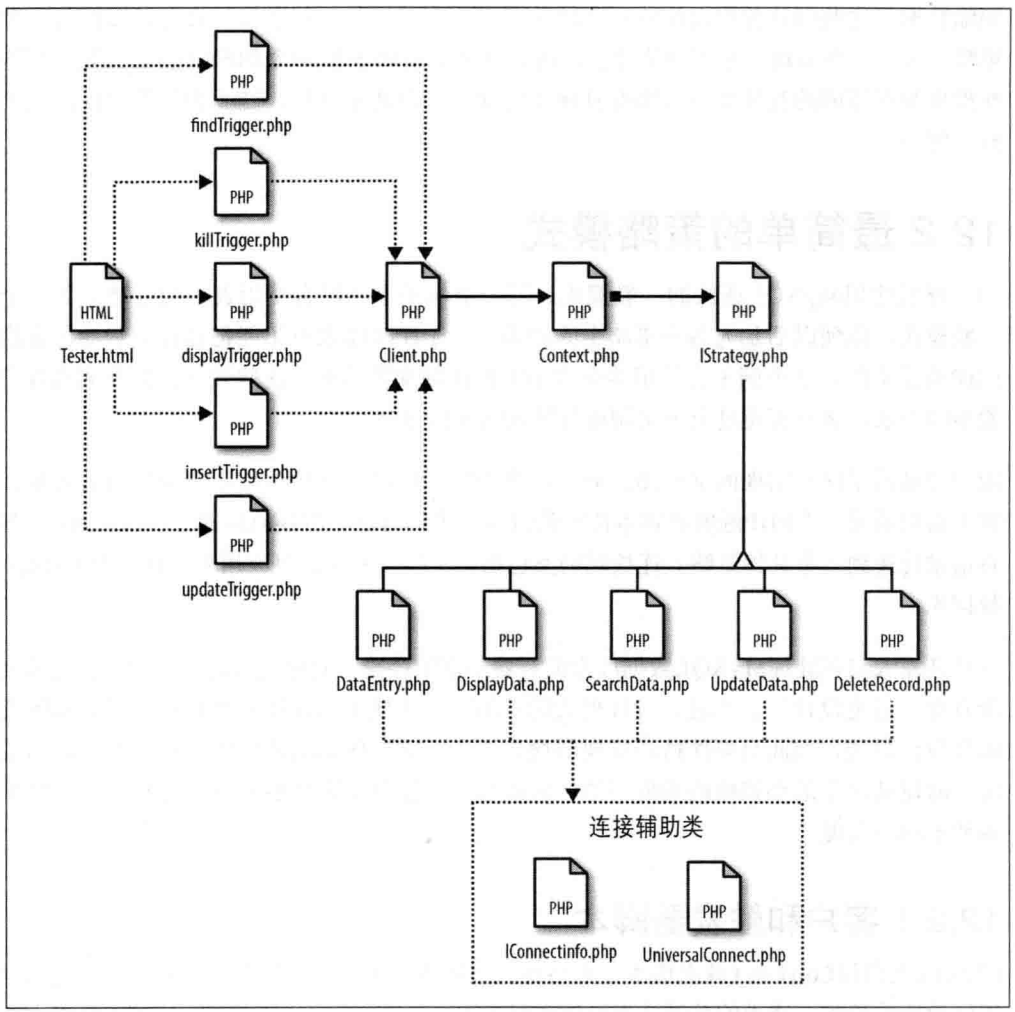


图12-2：策略模式文件图

```
<?php
class Client
{
    public function insertData()
    {
        $context=new Context(new DataEntry());
        $context->algorithm();
    }

    public function findData()
    {
        $context=new Context(new SearchData());
        $context->algorithm();
    }
```

```
    public function showAll()
    {
        $context=new Context(new DisplayData());
        $context->algorithm();
    }

    public function changeData()
    {
        $context=new Context(new UpdateData());
        $context->algorithm();
    }

    public function killer()
    {
        $context=new Context(new DeleteRecord());
        $context->algorithm();
    }
}
?>
```

为了触发不同具体策略（封装的算法）的方法，HTML会调用以下的一个PHP触发器脚本：

```
<?php
//insertTrigger.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
$trigger=new Client();
$trigger->insertData();
?>
-------------
<?php
//displayTrigger.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
$trigger=new Client();
$trigger->showAll();
?>
-------------
<?php
//findTrigger.php
    function __autoload($class_name)
{
    include $class_name . '.php';
}
$trigger=new Client();
$trigger->findData();
?>
-------------
<?php
```

```
//updateTrigger.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
    $trigger=new Client();
    $trigger->changeData();
?>
-------------
<?php
//killTrigger.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
$trigger=new Client();
$trigger->killer();
?>
```

HTML文档中的表单分别调用各个不同的PHP触发器。触发器脚本发出的请求传递到客户，客户再针对各个请求使用相应的一个方法。在策略设计模式中，客户通常会创建一个具体对象并传递到上下文。不过，最初的请求是由HTML文档发出：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Test</title>
</head>
<body>
Insert<br />
<form name="insert" action="insertTrigger.php" method="post">
  <input type="text" name="data">
  <br />
  <input type="submit" value="Insert">
</form>
<br />
Find Data
<form name="find" action="findTrigger.php" method="post">
  <input type="text" name="data">
  <br />
  <input type="submit" value="Find">
</form>
<br />
Display All Data
<form name="display" action="displayTrigger.php" method="post">
  <input type="submit" value="Show all data">
</form>
<br />
Update Data
<form name="change" action="updateTrigger.php" method="post">
  <input type="text" name="data">
  <br />
  <input type="submit" value="Change data in record">
```

```
</form>
<br />
Delete Record
<form name="killer" action="killTrigger.php" method="post">
  <input type="text" name="data">
  <br />
  <input type="submit" value="Delete Record">
</form>
</body>
</html>
```

在这个设计中，比较困难的一点是要保护$_POST数据，为此可以使用mysqli->real_escape_string()方法抽取由HTML文档发送的数据值。可以在提交的所有数据中包含一个额外的变量，指示客户将要使用一个具体策略请求方法，这样就能做出选择而无需使用大量触发器脚本。可以在客户中建立MySQL连接，取出数据、关闭连接，然后把请求继续传递到具体策略，在这里打开第二个连接，通过适当的策略处理这个请求。不过，这个例子的重点是设计模式，而不是解决所有安全问题。所以这里分别提供不同的触发器脚本来完成各个请求。

12.2.2 Context类和Strategy接口

在状态模式设计中，Context类相当于一个“跟踪者”（track keeper），它会跟踪当前的状态。在策略模式设计中，Context则有完全不同的功能，用于将请求与具体策略分离，使策略和请求可以独立地工作。这体现了请求与后果之间的另外一种松绑定。与此同时，它还有利于从Client发出请求。

Context不是一个接口（既不是抽象类也不是接口），不过它与Strategy接口有聚合关系。“四人帮”指定了以下特征：

- 用一个具体策略对象来配置（参见“客户与触发器脚本”一节中Client类如何实例化Context）。
- 维护Strategy对象的一个引用。
- 可以定义一个接口，允许Strategy访问其数据。

在下面的代码清单中，可以看到ConText类的上述特征：

```
<?php
class Context
{
    private $strategy;

    public function __construct(IStrategy $strategy)
    {
        $this->strategy = $strategy;
    }
```

```
    public function algorithm()
    {
        $this->strategy->algorithm();
    }
}
?>
```

对于以上的3个特性，第一，构造函数希望有一个IStrategy实现作为参数。第二，通过一个封装的属性$strategy（可见性为私有）来维护Strategy对象的一个引用。$strategy属性从构造函数参数接收Strategy对象实例，这将成为一个具体策略实例。第三，algorithm()方法实现了IStrategy的algorithm()方法，实现为通过Client选择的具体策略。由于Context和IStrategy构成一个聚合关系，所以Context具有抽象类或接口的某些特性。实际上，最好通过聚合来理解Context。查看策略接口IStrategy时，可以看到要实现的方法是algorithm()：

```
<?php
interface IStrategy
{
    public function algorithm();
}
?>
```

各个具体策略可以采用所需的方式实现这个方法。

12.2.3 具体策略

构成具体策略的封装算法族提供了所有可能的策略。对于这个最简单的例子，关键是要了解策略设计模式中不同的参与者如何协同工作。在这一节中，你会看到一个完全实现的例子。

5个具体策略包括以下类：

- DataEntry
- DisplayData
- SearchData
- UpdateData
- DeleteData

这些具体策略分别表示结合使用PHP和MySQL的典型算法。

DataEntry

第一个策略表示向一个表中输入数据：

```
<?php
class DataEntry implements IStrategy
//DataEntry.php
{
    public function algorithm()
    {
        $hookup=UniversalConnect::doConnect();
        $test = $hookup->real_escape_string($_POST['data']);
        echo "This data has been entered: " . $test . "<br/>";
    }
}
?>
```

DisplayData

在这个例子中并没有使用$_POST['data']，因为这个算法只显示字符串“Here's all the data!!”，它作为一个字符串直接量赋给变量$test：

```
<?php
//DisplayData.php
class DisplayData implements IStrategy
{
    public function algorithm()
    {
        $hookup=UniversalConnect::doConnect();
        $test = "Here's all the data!!";
        echo $test . "<br/>";
    }
}
?>
```

SearchData

搜索项在$_POST['data']中，将传递到$test变量：

```
<?php
//SearchData.php
class SearchData implements IStrategy
{
    public function algorithm()
    {
        $hookup=UniversalConnect::doConnect();
        $test = $hookup->real_escape_string($_POST['data']);
        echo "Here's what you were looking for " . $test . "<br/>";
}
    }
?>
```

UpdateData

“新”数据在$_POST['data']中，并传递到$test变量。在实际的实现中，还可能包含字段名：

```
<?php
//UpdateData.php
class UpdateData implements IStrategy
{
    public function algorithm()
    {
        $hookup=UniversalConnect::doConnect();
        $test = $hookup->real_escape_string($_POST['data']);
        echo "Your new data is now " . $test . "<br/>";
    }
}
?>
```

DeleteRecord

最后，会有一个唯一标识符传入$_POST['data']并存储在$test中，通过使用这个唯一标识符，可以从表中删除一个记录：

```
<?php
//DeleteRecord.php
class DeleteRecord implements IStrategy
{
    public function algorithm()
    {
        $hookup=UniversalConnect::doConnect();
        $test = $hookup->real_escape_string($_POST['data']);
        echo "The record " . $test . "has been deleted.<br/>";
    }
}
?>
```

连接接口和类

所有具体策略都实现相同的连接对象（与其他章中一样）。下面的接口包含了实际程序使用的名：

```
<?php
//Filename: IConnectInfo.php
interface IConnectInfo
{
    const HOST ="localhost";
    const UNAME ="alpha";
    const PW ="beta";
     const DBNAME = "gamma";

    public function doConnect();
}
?>
```

下面的连接类实现了IConnectInfo接口：

```
<?php
include_once('IConnectInfo.php');

class UniversalConnect implements IConnectInfo
{
    private static $server=IConnectInfo::HOST;
    private static $currentDB= IConnectInfo::DBNAME;
    private static $user= IConnectInfo::UNAME;
    private static $pass= IConnectInfo::PW;
    private static $hookup;

    public function doConnect()
    {
        self::$hookup=mysqli_connect(self::$server, self::$user, self::$pass,
                    self::$currentDB);
        if(self::$hookup)
        {
            echo "Successful connection to MySQL:<br/>";
        }
        elseif (mysqli_connect_error(self::$hookup))
        {
            echo('Here is why it failed: ' . mysqli_connect_error());
        }
        return self::$hookup;
    }
}
?>
```

通过使用一个连接类和单独的接口，可以更容易地重用和修改。这里只会修改接口中的常量值。

12.3 增加数据安全性和参数化算法来扩展策略模式

在上一节给出的最简单的例子中，可以看到结合使用MySQL数据库时PHP策略设计模式的所有基本元素。为了构建一个更健壮的例子，下面这个例子会为不同策略增加功能。这里还增加了一个辅助类，来处理数据从HTML客户到MySQL数据库的安全移动。这说明，客户可以使用通过`mysqli->real_escape_string($_POST['data'])`函数传递的数据做出安全的请求。`Client`类可以自己处理安全性，不过这样一来，除了做出请求外，会为`Client`类增加额外的责任。

12.3.1 数据安全性辅助类

通过使用`mysqli->real_escape_string($_POST['data'])`函数，在HTML表单和PHP类之间安全地传递数据，需要有一个MySQL连接，不过一旦打开连接并安全地传递了数据，可以再关闭这个连接，释放连接占用的资源。

考虑到会有不同的具体策略，这个辅助类对于保证各个具体策略的数据安全性分别提供了不同的方法。一个方法是向Client传回一个数组，其中包含该请求所需的数据。图12-3通过图示表示了辅助类与Strategy的关系。

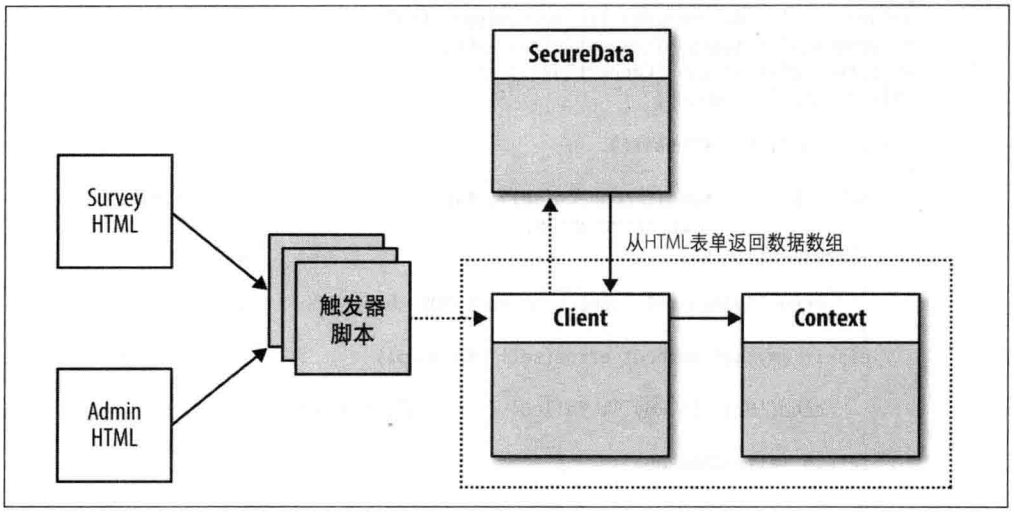


图12-3：增加一个辅助类处理数据安全性

图12-3中没有显示策略模式中Context类后面的部分，不过它仍然遵循图12-1所示的标准类图。另外，前面使用了MySQL辅助类处理数据库请求，SecureData类同样使用这个辅助类来创建MySQL连接。

对于每一个依赖于HTML表单数据的具体策略，SecureData类中分别提供了相应的方法。DisplayAll具体策略请求显示所有数据，所以它不需要由HTML表单传递的特殊数据：

```
<?php
//辅助类
//SecureData.php
class SecureData
{
    private $changeField;
    private $company;
    private $devdes;
    private $device;
    private $disappear;
    private $field;
    private $hookup;
    private $lang;
    private $newData;
    private $oldData;
    private $plat;
```

```
private $style;
private $term;
//$dataPack 将是一个数组
private $dataPack;

public function enterData()
{
    $this->hookup=UniversalConnect::doConnect();
    $this->company=$this->hookup->real_escape_string($_POST['company']);
    $this->devdes=$this->hookup->real_escape_string($_POST['devdes']);
    $this->lang= $this->hookup->real_escape_string($_POST['lang']);
    $this->plat= $this->hookup->real_escape_string($_POST['plat']);
    $this->style=$this->hookup->real_escape_string($_POST['style']);
    $this->device=$this->hookup->real_escape_string($_POST['device']);
    $this->dataPack=array(
        $this->company,
        $this->devdes,
        $this->lang,
        $this->plat,
        $this->style,
        $this->device
);

    $this->hookup->close();
}

public function conductSearch()
{
    $this->hookup=UniversalConnect::doConnect();
    $this->field=$this->hookup->real_escape_string($_POST['field']);
    $this->term=$this->hookup->real_escape_string($_POST['term']);
    $this->dataPack=array(
        $this->field,
        $this->term
        );
    $this->hookup->close();
}

public function makeChange()
{
    $this->hookup=UniversalConnect::doConnect();
    $this->changeField=$this->hookup->real_escape_string($_POST['update']);
    $this->oldData=$this->hookup->real_escape_string($_POST['old']);
    $this->newData=$this->hookup->real_escape_string($_POST['new']);
    $this->dataPack=array(
        $this->changeField,
        $this->oldData,
        $this->newData
        );
    $this->hookup->close();
}

public function removeRecord()
{
    $this->hookup=UniversalConnect::doConnect();
    $this->disappear=$this->hookup->real_escape_string($_POST['delete']);
```

```
            $this->dataPack=array($this->disappear);
            $this->hookup->close();
        }

        //将安全数据作为数组返回给请求客户
        public function setEntry()
        {
            return $this->dataPack;
        }
    }
    ?>
```

除了setEntry()之外，所有方法都生成一个名为dataPack的数组。setEntry()方法会返回这个dataPack数组的当前内容。取决于具体的请求，SecureData类生成将置于数组中的值，这会传回到Client，并通过algorithm()方法作为请求的一部分发送到一个具体策略。

12.3.2 为算法方法增加参数

第二个要增加的特性是修改Strategy算法方法。我们将增加一个数组作为函数的一个参数，这样可以增加灵活性，处理更多的内容。每个算法函数调用都包含一个数组，其中包含从HTML表单传递的数据：

```
<?php
interface IStrategy
{
    const TABLENOW ="survey";
    public function algorithm(Array $dataPack);
}
?>
```

同样，还要为接口增加一个常量TABLENOW。由于这个实现中各个具体策略都使用相同的表，而且PHP能够通过接口传递常量，因此可以建立一个松耦合而且可重用的代码。显然，如果不同的具体策略要使用不同的表，就必须在各个具体策略中指定表引用。参数中的类型提示指示要将数组用作为一个实参。

12.3.3 调查表

下面的脚本用于创建一个调查表（数据库表）。策略设计模式中可以使用更大或更小的表，为此只需要调整具体策略中所用数组的大小：

```
<?php
include_once('UniversalConnect.php');
class CreateTable
{
    private $tableMaster;
    private $hookup;
```

```
    public function __construct()
    {
        $this->tableMaster="survey";
        $this->hookup=UniversalConnect::doConnect();

        $drop = "DROP TABLE IF EXISTS $this->tableMaster";

        if($this->hookup->query($drop) === true)
        {
            printf("Old table %s has been dropped.<br/>",$this->tableMaster);
        }

        $sql = "CREATE TABLE $this->tableMaster (
            id SERIAL,
            company     NVARCHAR(40),
            devdes      NVARCHAR(10),
            lang        NVARCHAR(15),
            plat        NVARCHAR(15),
            style       NVARCHAR(20),
            device      NVARCHAR(10),
            PRIMARY KEY (id))";

        if($this->hookup->query($sql) === true)
        {
            printf("Table $this->tableMaster has been created successfully.
             <br/>");
        }
        $this->hookup->close();
    }
}
$worker=new CreateTable();
?>
```

这个特定的类用于创建数据库表，仅在开发和调试阶段使用。一旦建立了所要的表，而且希望安装在不同的系统上，可以删除下面这段代码：

```
$drop = "DROP TABLE IF EXISTS $this->tableMaster";

if($this->hookup->query($drop) === true)
{
    printf("Old table %s has been dropped.<br/>",$this->tableMaster);
}
```

并将以下代码：

```
$sql = "CREATE TABLE $this->tableMaster (
```

修改为：

```
$sql = "CREATE TABLE IF NOT EXISTS $this->tableMaster (
```

这样一来，如果原来的表中已经存储有数据，就不会被CreateTable类删除。

本章中所有其他的MySQL连接都会使用同样的UniversalConnect类。

12.3.4 数据输入模块

利用SecureData辅助类和修改后的IStrategy接口（可以为algorithm()方法包含一个参数），对于不同的HTML表单请求，Client可以根据相应的方法更容易地做出请求。在继续学习下面的内容之前，先来看看HTML表单中发出的请求。这里使用了两个表单：一个允许用户输入调查数据，另一个用于查看存储在MySQL表中的数据。两个表单都非常简单，是很通用的HTML表单。两个表单使用同一个CSS文件：

```
@charset "UTF-8";
/*survey.css */
/* CSS Document */
/*B2B2B2,B2A1A1,666666,8FB299*/
body
{
    background-color:#B2B2B2;
    color:#666666;
    font-family:Verdana, Geneva, sans-serif
}

h2
{
    font-family:"Arial Black", Gadget, sans-serif;
    color:#666666
}

h3
{
    font-family:"Trebuchet MS", Arial, Helvetica, sans-serif;
    background-color:#8FB299;
    color:#666666
}

th
{
    text-align:left;
    background-color:#8FB299;
    color:#666666
}
```

CSS只是用来稍做区分。

首先，调查表单只有一个简单的文本输入，另外可以利用单选钮做一些选择：

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="survey.css">
<meta charset="UTF-8">
<title>Programmer Profile Survey</title>
</head>
<body>
<h2>Programmer Survey</h2>
```

```
<form name="survey" action="insertTrigger.php" method="post">
  <input type="text" name="company">
   Company Name<br />
  <h3> Primary Role</h3>
  <input type="radio" name="devdes" value="developer">
   Developer<br />
  <input type="radio" name="devdes" value="designer">
   Designer<br />
  <h3> Primary Programming Language</h3>
  <input type="radio" name="lang" value="PHP">
   PHP<br />
  <input type="radio" name="lang" value="C#/ASP.NET">
   C# ASP.NET<br />
  <input type="radio" name="lang" value="PERL">
   PERL<br />
  <input type="radio" name="lang" value="JavaScript">
   JavaScript<br />
  <input type="radio" name="lang" value="ActionScript 3.0">
   ActionScript 3.0<br />
  <h3> Primary Development/Design Platform</h3>
  <input type="radio" name="plat" value="WinPC">
   Windows PC<br />
  <input type="radio" name="plat" value="Mac">
   Apple Macintosh<br />
  <input type="radio" name="plat" value="Linux">
   Linux<br />
  <h3> Primary Programming Style</h3>
  <input type="radio" name="style" value="sequential">
   Sequential<br />
  <input type="radio" name="style" value="procedural">
   Procedural<br />
  <input type="radio" name="style" value="OOP">
   Object Oriented Programming<br />
  <input type="radio" name="style" value="design patterns">
   Design Patterns
  <p />
  <h3> Primary Platform Development/Design</h3>
  <input type="radio" name="device" value="Desktop">
   Desktop<br />
  <input type="radio" name="device" value="Tablet">
   Tablet<br />
  <input type="radio" name="device" value="Smartphone">
   Smartphone
  <p />
  <input type="submit" value="Create Profile" name="sender">
</form>
</body>
</html>
```

图12-4显示了在一个平板设备上打开的调查表单。

第二个HTML文档提供了一个管理工具，可以用来检查数据库中的表。同样的，这个HTML表单也很简单，通过策略设计模式将数据从其来源（数据库表）放入HTML表单中：

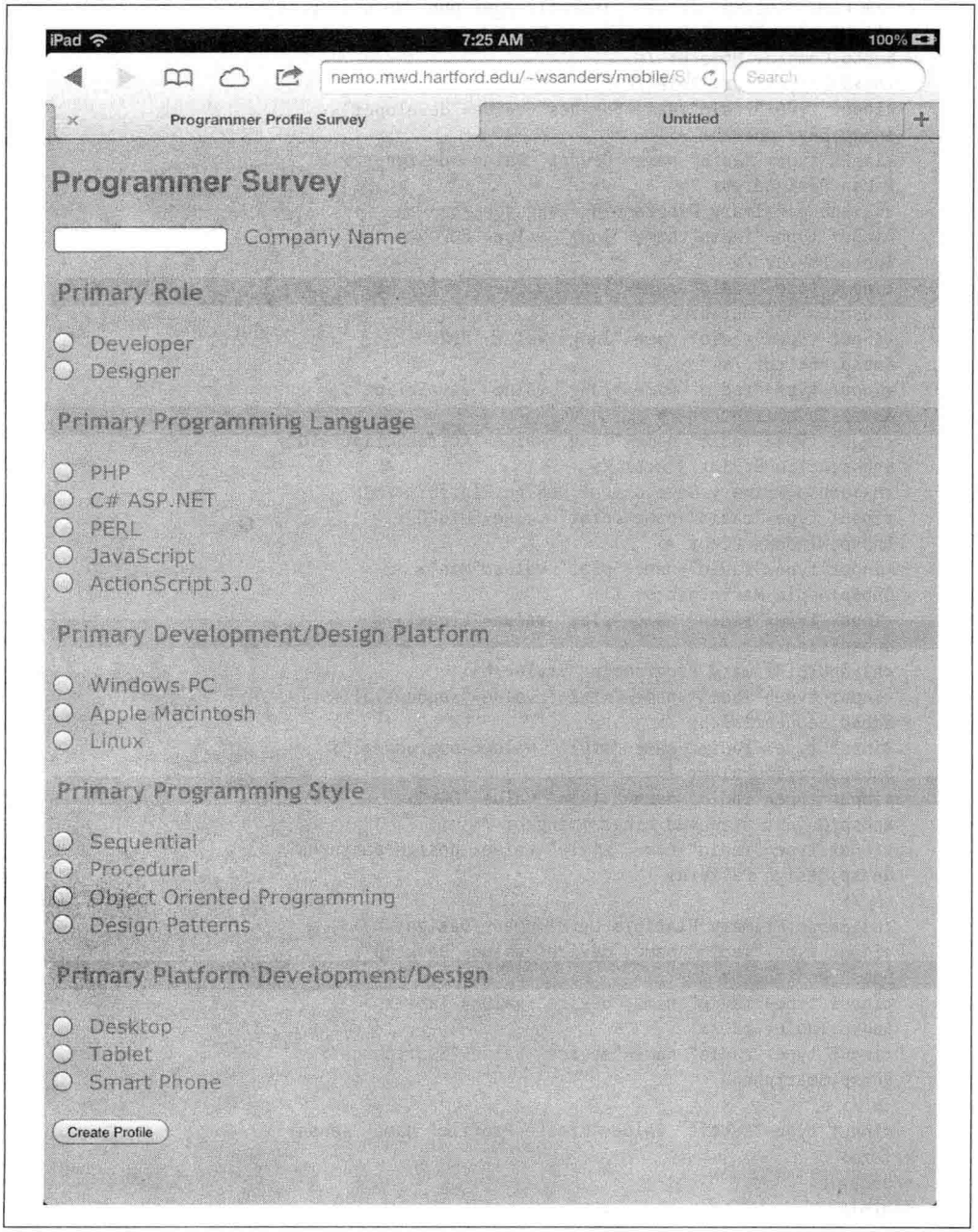


图12-4：利用策略模式处理的调查表用户界面

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" href="survey.css">
```

```
<meta charset="UTF-8">
<title>Administrative Module</title>
</head>

<body>
<h2>Administrative Module</h2>
<h3> Display all data</h3>
<form name="allData" action="displayTrigger.php" method="post">
  <input type="submit" value="Display Data" name="display">
</form>
<form name="search" action="findTrigger.php" method="post">
  <h3> Search Field</h3>
  <input type="radio" name="field" value="id">
   ID<br />
  <input type="radio" name="field" value="company">
   Company<br />
  <input type="radio" name="field" value="devdes">
   Designer/Developer<br />
  <input type="radio" name="field" value="lang">
   Computer Language<br />
  <input type="radio" name="field" value="plat">
   Development Platform<br />
  <input type="radio" name="field" value="style">
   Programming Style<br />
  <input type="radio" name="field" value="device">
   Target Device
  <p />
  <input type="text" name="term">
   Term to find
  <p />
  <input type="submit" value="Search" name="searcher">
</form>
<form name="search" action="changeTrigger.php" method="post">
  <h3> Change Field</h3>
  <input type="radio" name="update" value="id">
   ID <br />
  <input type="radio" name="update" value="company">
   Company<br />
  <input type="radio" name="update" value="devdes">
   Designer/Developer<br />
  <input type="radio" name="update" value="lang">
   Computer Language<br />
  <input type="radio" name="update" value="plat">
   Development Platform<br />
  <input type="radio" name="update" value="style">
   Programming Style<br />
  <input type="radio" name="update" value="device">
   Target Device
  <p />
  <input type="text" name="old">
   Old Value
  <p />
  <input type="text" name="new">
   New Value
  <p />
  <input type="submit" value="Change Value" name="changer">
```

```
</form>
<h3> Delete Record</h3>
<form name="killer" action="killTrigger.php" method="post">
  <input type="text" name="delete" size=3>
   Number of Record to Delete
  <p />
  <input type="submit" value="Permanently Delete Record" name="doa">
</form>
</body>
</html>
```

图12-5给出一个平板设备上显示的管理模块用户界面。

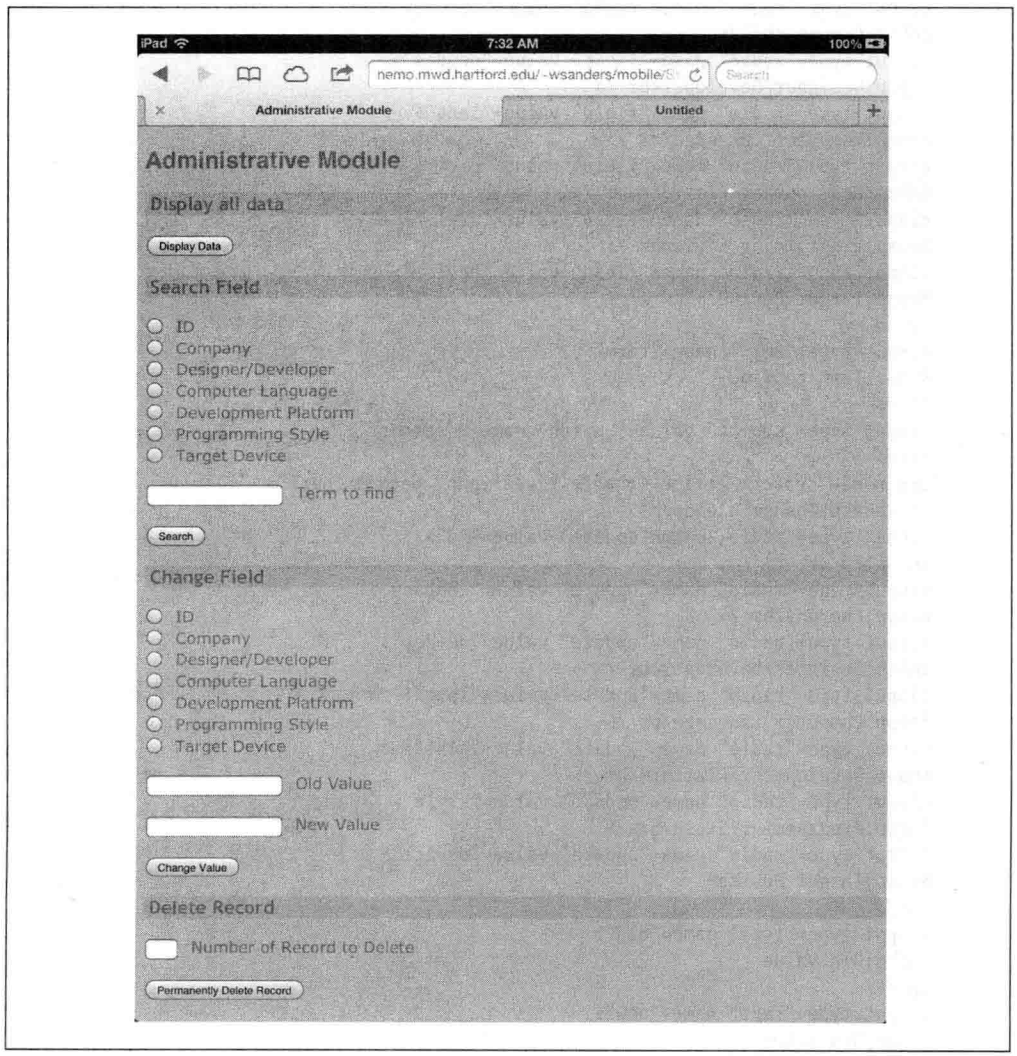


图12-5：平板设备上显示的管理模块用户界面

这两个用户界面（UI）都使用了移动布局，只有一列，可以调整这个布局以便在支持互联网功能的移动手机上查看。

两个HTML文档中的按钮分别表示不同的表单。每个表单调用一个触发器文件，再实例化Client类，并调用所需的方法来完成所请求的任务。

12.3.5 客户请求帮助

Client没有构造函数，不过有多个方法，可以做不同的请求。这些方法与最简单的例子类似，不过在前面讨论的SecureData辅助类（见“数据安全性辅助类”一节）的帮助下，这些方法可以做更多工作。

首先请回顾SecureData类的工作，然后再来分析Client：

```
<?php
//Client.php
class Client
{
    public function insertData()
    {
        $secure=new SecureData();
        $context=new Context(new DataEntry());
        $secure->enterData();
        $context->algorithm($secure->setEntry());
    }

    public function findData()
    {
        $secure=new SecureData();
        $context=new Context(new SearchData());
        $secure->conductSearch();
        $context->algorithm($secure->setEntry());
    }

    public function showAll()
    {
        $dummy=array(0);
        $context=new Context(new DisplayAll());
        $context->algorithm($dummy);
    }

    public function changeData()
    {
        $secure=new SecureData();
        $context=new Context(new UpdateData());
        $secure->makeChange();
        $context->algorithm($secure->setEntry());
    }

    public function killer()
    {
```

```
            $secure=new SecureData();
            $context=new Context(new DeleteRecord());
            $secure->removeRecord();
            $context->algorithm($secure->setEntry());
        }
    }
    ?>
```

除了showAll()方法外，Client中的所有方法都会首先实例化SecureData类、然后使用具体方法作为参数来创建一个上下文对象。接下来，SecureData对象调用具体策略的相应方法创建所需的数组。最后Client方法调用Context->algorithm()，并使用SecureData类返回$secure->setEntry()数组作为参数。数组的内容取决于HTML表单发送的用户输入以及所请求的策略类型。

“四人帮”指出，不论是否使用，所有具体策略类都有一个共同的接口。因此，所有具体策略类都必须实现Strategy接口的方法（IStrategy中的algorithm()方法）。不过，并不是所有具体策略都需要这个算法，当然实现方式也不完全相同。

可以看到，从某种程度上讲，Client类中的**showAll()**方法就不需要这个算法。这个方法没有使用SecureData类返回的一个数组，而是创建了一个dummy数组，并把它用作为Context->algorithm()的参数。这是为了满足IStrategy接口的需求，接口要求这个方法必须包含一个数组作为参数。

12.3.6 Context类重要的小改变

与这一章第一部分中最简单的例子相比，Context类几乎没有改动，只是为algorithm()方法增加了一个参数，这是更新后的IStrategy接口提出的要求。由于Context类和IStrategy之间有一种聚合关系，Context类必须包含IStrategy的一个引用。类似于前面最简单的例子，同样要用一个具体策略对象创建Context。这一部分未做任何改变。不过，它还包含一个方法，将实例化具体策略的algorithm()方法实现：

```
<?php
class Context
{
    private $strategy;
    private $dataPack;

    public function __construct(IStrategy $strategy)
    {
        $this->strategy = $strategy;
    }

    public function algorithm(Array $dataPack)
    {
        $this->dataPack=$dataPack;
        $this->strategy->algorithm($this->dataPack);
```

```
    }
}
?>
```

注意： Context类方法的名字也是“algorithm”，它要求有一个数组参数。两个方法都命名为algorithm，其目的是为了强调上下文和策略参与者之间的聚合关系。如果觉得有些混淆，可以把这个方法重命名为contextAlgorithm，使它与IStrategy的algorithm()方法有所区别。

Context还有另外一个属性$dataPack，反映了通过Context algorithm()方法传递的数组的名字。然后再传递到具体策略的algorithm()方法。

12.3.7 具体策略

通过一个数组向具体策略传递数据的根本目的是允许不同的策略对不同的请求做出响应。这样可以为设计提供灵活性，因为利用数组可以传递大量数据。在下面的具体策略类中可以看到，每个策略类都实现了IStrategy algorithm()方法，其中使用了通过方法数组参数传递的数据。

最简单的策略例子中使用了一个UniversalConnect类，这里的所有具体类同样使用了这个类。表名作为一个常量（TABLENOW）存储在IStrategy接口中。

DataEntry

在所有具体策略中，DataEntry类使用的数组最大。这是因为它必须插入HTML调查表提交的所有数据：

```
<?php
class DataEntry implements IStrategy
//DataEntry.php
{
    private $tableMaster;
    private $dataPack;
    private $hookup;
    private $sql;

    public function algorithm(Array $dataPack)
    {
        $this->dataPack=$dataPack;
        $comval=$this->dataPack[0];
        $devdesval=$this->dataPack[1];
        $langval=$this->dataPack[2];
        $platval=$this->dataPack[3];
        $styleval=$this->dataPack[4];
        $deviceval=$this->dataPack[5];

        $this->tableMaster=IStrategy::TABLENOW;
        $this->hookup=UniversalConnect::doConnect();
```

```
        $this->sql = "INSERT INTO $this->tableMaster
        (
             company,
             devdes,
             lang,
             plat,
             style,
             device
        )
        VALUES
        (
            '$comval',
            '$devdesval',
            '$langval',
            '$platval',
            '$styleval',
            '$deviceval'
        )";

        if($this->hookup->query($this->sql))
        {
            printf("Successful data entry for table: $this->tableMaster <br/>");
        }
        elseif ( ($result = $this->hookup->query($this->sql))===false )
        {
            printf("Invalid query: %s <br/> Whole query: %s <br/>",
                                $this->hookup->error, $this->sql);
            exit();
        }

        $this->hookup->close();
    }
}
?>
```

这里只是标准MySQL语句中使用了条件语句。作为这个类的核心，通用算法没有使用任何条件语句。

DisplayAll

将为这个类传递一个dummy数组。可以看到，它只是使用一个通用算法从数据库表向屏幕发送数据：

```
<?php
//DisplayAll.php
class DisplayAll implements IStrategy
{
    private $tableMaster;
    private $hookup;

    public function algorithm(Array $dataPack)
    {
        $this->tableMaster=IStrategy::TABLENOW;
        $this->hookup=UniversalConnect::doConnect();
```

```
        //创建查询语句
        $sql ="SELECT * FROM $this->tableMaster";
        //MySQL命令中的条件语句
        if ($result = $this->hookup->query($sql))
        {
            printf("Select returned %d rows.<p />", $result->num_rows);

            echo "<link rel='stylesheet' href='survey.css'>";
            echo "<table>";

            while ($finfo = mysqli_fetch_field($result))
            {
               echo "<th> {$finfo->name}</th>";
            }
            echo "</tr>\n";

            while($row=mysqli_fetch_row($result))
            {
               echo "<tr>";
               foreach($row as $cell)
               {
                   echo "<td>$cell</td>";
               }
               echo "</tr>";
               }
            echo"</table>";
            $result->close();
        }
    $this->hookup->close();
    }
}
?>
```

表格可以更好地显示数据，不过它可不是最优的实现。实际上，由于我们的重点是实现设计模式，所以这个格式要尽可能简单。图12-6显示了这个输出。

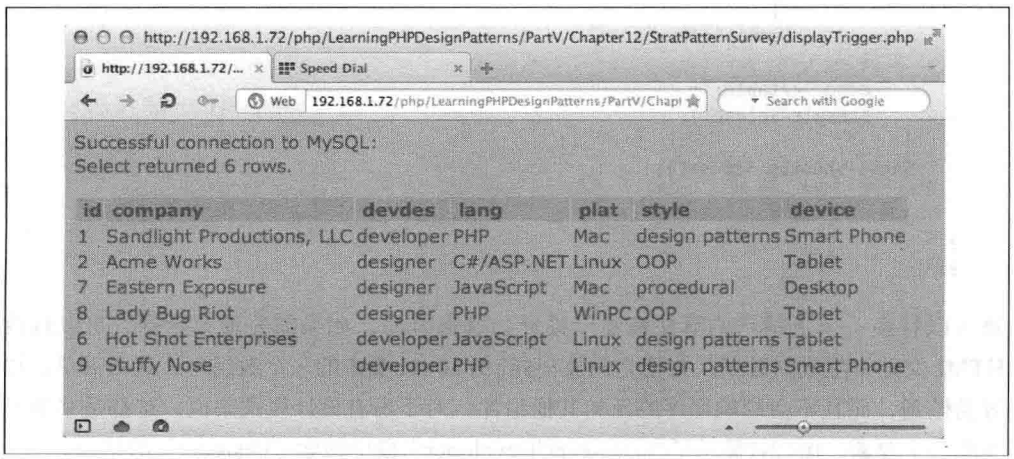


图12-6：DisplayAll具体策略的数据输出

SearchData

搜索算法将从一个指定字段选择一个指定值。字段名和搜索值通过数组作为一个参数传入algorithm()方法。如果匹配，则会显示匹配的记录，如果没有找到匹配，就不显示任何结果：

```
<?php
//SearchData.php
class SearchData implements IStrategy
{
    private $tableMaster;
    private $dataPack;
    private $hookup;
    private $sql;

    public function algorithm(Array $dataPack)
    {
        $this->tableMaster=IStrategy::TABLENOW;
        $this->hookup=UniversalConnect::doConnect();
        $this->dataPack=$dataPack;
        $field=$this->dataPack[0];
        $term=$this->dataPack[1];
        $this->sql = "SELECT * FROM $this->tableMaster WHERE $field='$term'";
        //MySQL查询中的条件语句用于数据输出
        if ($result = $this->hookup->query($this->sql))
        {
            echo "<link rel='stylesheet' href='survey.css'>";
            echo "<table>";
            while($row=mysqli_fetch_row($result))
            {
                echo "<br />";
                echo "<tr>";
                foreach($row as $cell)
                {
                    echo "<td>$cell</td>";
                }
                echo "</tr>";
        }
        echo "</table>";
        $result->close();
    }
    $this->hookup->close();

    }
}
?>
```

也可以替换一个更精巧的算法和显示设计，这很容易。如果需要更多数据，可以修改HTML文档中生成的数据，并把它们放在SecureData类中的一个数组中，这很简单，很容易做到，而且不会影响程序的任何其他组件，对于所有设计模式来说，这都是必不可少的一个要素。图12-7显示了在Designer/Developer字段中搜索“designer”的结果。

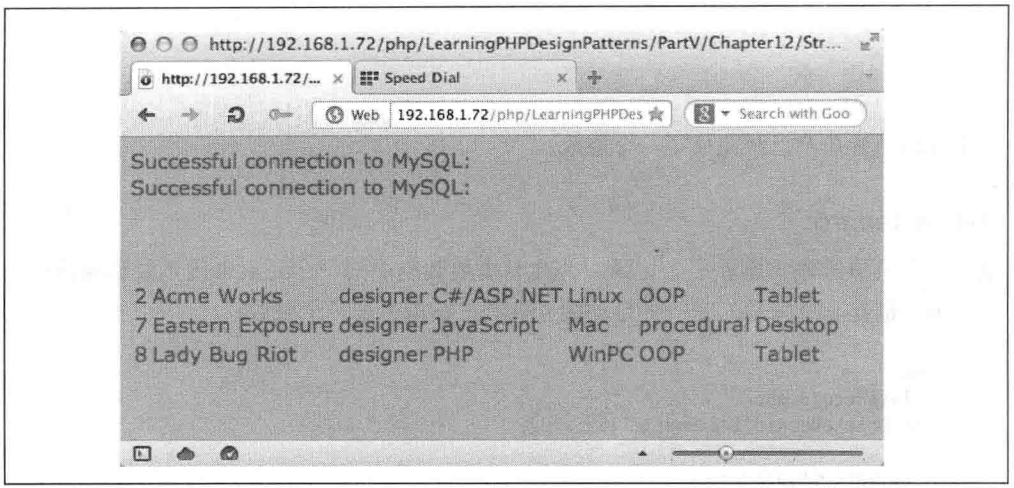


图12-7：输出显示“designer”的所有匹配结果

UpdateData

要改变当前字段中的值，这个实现只需要三个要素：字段名、原来的值和新值。这个算法很灵活，与所有其他具体策略一样，可以修改算法来反映具体的需求：

```
<?php
//UpdateData.php
class UpdateData implements IStrategy
{
    private $tableMaster;
    private $dataPack;
    private $hookup;
    private $sql;

    public function algorithm(Array $dataPack)
    {
        $this->tableMaster=IStrategy::TABLENOW;
        $this->hookup=UniversalConnect::doConnect();
        $this->dataPack=$dataPack;
        $changeField=$this->dataPack[0];
        $oldData=$this->dataPack[1];
        $newData=$this->dataPack[2];
        $this->sql = "UPDATE $this->tableMaster SET $changeField='$newData'
                      WHERE $changeField='$oldData'";
        //MySQL查询中的条件语句用于错误检查
        if ($result = $this->hookup->query($this->sql))
        {
             echo "$changeField changed from $oldData to: $newData";
        }
        else
        {
             echo "Change failed: " . $hookup->error;
```

```
        }
    }
}
?>
```

这个输出通知用户已经完成了一个修改。

DeleteRecord

最后一个具体策略将删除一条记录，为此只需要数组中的一个元素来传递要删除的记录号。由于创建表时采用的是一种自动编号机制，标识号是一个整数：

```
<?php
//DeleteRecord.php
class DeleteRecord implements IStrategy
{
    private $tableMaster;
    private $dataPack;
    private $hookup;
    private $sql;

    public function algorithm(Array $dataPack)
    {
        $this->tableMaster=IStrategy::TABLENOW;
        $this->hookup=UniversalConnect::doConnect();
        $this->dataPack=$dataPack;
        $destroy=$this->dataPack[0];
        $destroy= intval($destroy);

        $this->sql = "DELETE FROM $this->tableMaster WHERE id='$destroy'";
        //MySQL查询中的条件语句用于错误检查
        if ($result = $this->hookup->query($this->sql))
        {
            echo "Record #$destroy removed from table: $this->tableMaster";
        }
        else
        {
            echo "Removal failed: " . $hookup->error;
        }
    }
}
?>
```

这个类和相应的具体策略很简单，如果想增加一个更健壮的算法，也很容易做到，而不会破坏程序的其余部分。

12.4 灵活的策略模式

策略模式很灵活，改变算法时可以只改变一个实现，不仅如此，模式本身还可以有多个实现。一方面，这一章展示了一个最简单的策略设计模式，它可以调用不同的算法，这

些算法独立于具体策略之外的数据；另一方面，第二个例子使用了参数，可以向具体策略传递安全的数据。

“四人帮”指出，一种做法是由Context通过参数向Strategy操作传递数据。这正是第二个例子采用的做法。这种方法可以得到提交给策略的数据，同时保证Context与Strategy解耦合，但也可能向Strategy传递它不需要的数据。对于这个问题，可以使用一个数组来解决，利用这个数组（这也包括一个空数组），为Strategy传递数据时会有更大的灵活性。

特定的策略模式实现依赖于特定算法的需求以及它具体需要些什么。一些策略模式实现会存储其上下文的一个引用，因此没有必要传递数据。不过，这样一来，Context和Strategy会更紧密地耦合。

还有一个问题需要考虑：即策略模式所生成的对象（具体策略）个数。在这一章的例子中可以看到，它们都构建了大量的对象（类）来处理一个简单MySQL调查（由PHP驱动）的不同请求。还有可能构建更多对象。不过，相对于重用性以及改变模式所带来的好处，这可能不算太大的问题。构建设计模式是为了提高管理一个应用的速度，而不是为了提高执行代码的速度。如果采用了良构的策略模式，开发人员可以很容易地优化和重新优化封装的算法，而不会搞得一团乱麻。所以速度表现在重用和修改时间上，而且额外对象的开销很小。

第13章

职责链设计模式

大多数人并不真正想要自由，因为自由也会带来责任，
而大多数人都畏惧责任。
——西格蒙德·弗洛伊德

我们越来越睿智，并不是因为对过去的回忆，
而是因为对未来担负的责任。
——萧伯纳

若是自己承受太大的责任，
或者把所有责任都揽在自己身上，
这会毁了你。
——弗兰兹·卡夫卡

责任到此,不能再推。
——哈里·S·杜鲁门

13.1 推卸责任

职责链设计模式将请求的发送者与接收者分开，这样可以避免请求者与接收者的耦合。另外，这个模式允许将请求沿着一个链传递到多个不同的对象，使这些对象都有机会处理请求。发送者并不需要知道哪一个对象处理这个请求，而对象也不需要知道是谁发送了这个请求。在这二者之间不存在耦合。

职责链设计模式的好处是，任何对象都可以向处理请求的对象发送请求，而且处理请求的对象可以改变，从而可以结合更多或不同的对象来处理请求。所以请求者和请求处理器都可以改变，而不必担心对更大的系统带来破坏。图13-1显示了这个职责链设计模式的类图。

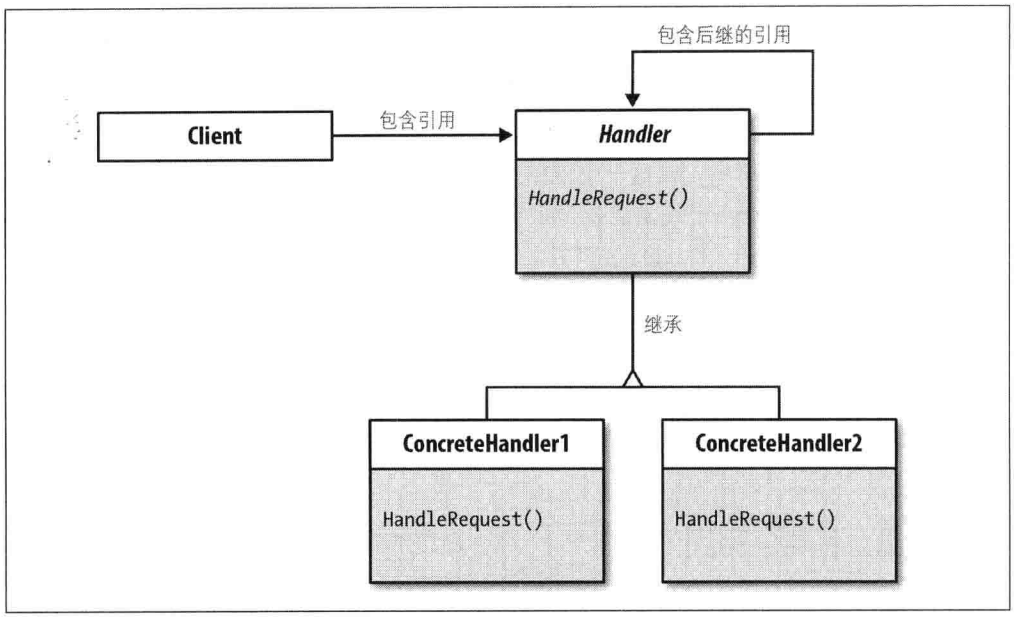


图13-1：职责链设计模式类图

注意，Client是这个程序中不可缺少的一部分。实际上，在这个结构中，它要完成很重要的工作，因为Client不仅要启动请求，可能还需要启动具体处理器中实现的后继。为此，一种方法是在Handler接口中建立后继方法，并在具体处理器中实现。当Client发出一个请求时，它可以同时建立这个链，并确定后继的顺序。

看到职责链模式时，有些人可能会认为“这只不过是一个大的switch语句而已”。从某种程度上看似乎如此，不过实际上并不是这样。它确实会检查一个请求，确定它是否与某个case条件匹配。不过，switch语句是固定的，而职责链会由各个具体处理器定义其后继。基于这种组织，可以把任意多个响应存储在一个MySQL数据库表中，这样一来，多个不同的“咨询台”可以使用相同的表以及相同的PHP职责链。由于每个具体处理器都包含一个方法来指定它自己的后继，Client通过具体处理器对象指定后继时也就定义了顺序。

另外，由于Client要启动请求链，它可以在开发人员指定的任何位置开始。假设具体请求处理器4、10、15和30可以作为你希望的新咨询台，就可以指定处理器4作为链中的第一个处理器，然后指定处理器10作为它的后继，再指定15为10的后继，而30作为15的后继。所以，职责链具有switch语句所没有的灵活性。

13.2 MySQL咨询台中的职责链

职责链的第一个实现是一个咨询台，用户可以从一系列帮助主题中选择一个要询问的主题。不论是作为PHP设计模式，还是使用MySQL作为一个文本数据存储系统，这都是一个简单的实现。基本说来，在这个实现中，用户将从一组单选钮中选择一个查询，发出请求，然后通过一个响应链搜索。找到正确的响应时，对象从一个MySQL表获取响应，并在屏幕上显示。

13.2.1 构建和加载响应表

首先要构建一个表，在其中存储文本响应。这一章中的MySQL连接同样使用第11章中的连接类和接口。为便于查看，这里再次给出这些类和接口。首先来看接口：

```
<?php
//文件名：IConnectInfo.php
interface IConnectInfo
{
    const HOST ="localhost";
    const UNAME ="phpWorker";
    const PW ="easyWay";
    const DBNAME = "dpPatt";

    public function doConnect();
}

?>
```

当然，你要使用你自己的连接信息。现在来看实现这个接口的连接类：

```
<?php
include_once('IConnectInfo.php');

class UniversalConnect implements IConnectInfo
{
    private static $server=IConnectInfo::HOST;
    private static $currentDB= IConnectInfo::DBNAME;
    private static $user= IConnectInfo::UNAME;
    private static $pass= IConnectInfo::PW;
    private static $hookup;

    public function doConnect()
    {
        self::$hookup=mysqli_connect(self::$server, self::$user, self::$pass,
                    self::$currentDB);
        if(self::$hookup)
        {
            //你可能想删除这个消息
            echo "Successful connection to MySQL:<br/>";
        }
        elseif (mysqli_connect_error(self::$hookup))
```

```
        {
            echo('Here is why it failed: ' . mysqli_connect_error());
        }
        return self::$hookup;
    }
}
?>
```

这一章需要的每一个MySQL连接都会使用这个连接接口和类。

响应表只需要一个ID、一个指示链对象的字段，以及一个文本字段（提供咨询台中的“帮助”文本）：

```
<?php
include_once('../UniversalConnect.php');
class CreateTable
{
    private $tableMaster;
    private $hookup;

    public function __construct()
    {
        $this->tableMaster="helpdesk";
        $this->hookup=UniversalConnect::doConnect();

        $drop = "DROP TABLE IF EXISTS $this->tableMaster";

        if($this->hookup->query($drop) === true)
        {
            printf("Old table %s has been dropped.<br/>",$this->tableMaster);
        }

        $sql = "CREATE TABLE $this->tableMaster (id INT NOT NULL AUTO_INCREMENT,
                chain VARCHAR(3), response TEXT, PRIMARY KEY (id))";

            if($this->hookup->query($sql) === true)
        {
            printf("Table $this->tableMaster has been created successfully.
                    <br/>");
        }
        $this->hookup->close();
    }
}

$worker=new CreateTable();
?>
```

这里增加了自动id字段，因为将来有可能对表进行编辑，这算是一种预备措施，不过如果你愿意也可以将其忽略。另外，这里首先删除一个表，然后再创建一个表，这种做法在开发中很常用。这样可以加速这个过程，并确保在继续处理之前先将原来的表删除。完成表的调试之后，可以使用MySQL命令来创建表（去掉之前首先删除表的那些代码）：

```
CREATE TABLE IF NOT EXISTS
```

（当然，你也可以使用MySQL管理工具来创建表！）

要让这个表真正发挥作用，需要在表中填入对应查询的响应，所以需要一个数据输入模块，包括一个PHP类和一个HTML表单。另外还必须更新咨询素材，所以除了数据输入外，还需要一个数据更新模块。先来看HTML：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<link href="help.css" rel="stylesheet" type="text/css" >
<title>Help Desk Data Entry and Update</title>
</head>
<body>
<h2>Help Desk Data Entry</h2>
<form action="InsertData.php" method="post" name="dataentry">
  Enter Query Chain Value (q1...qx)<br/>
  <input type="text" name="chain" size="3">
  <p/>
  Enter Help Information<br/>
  <textarea name="response" rows="8" cols="24"></textarea>
  <p/>
  <input type="submit" value="Enter data into table" name="send">
</form>
<h2>Update Information</h2>
<form action="UpdateData.php" method="post" name="dataupdate">
  Enter Query Chain Value (q1...qx)<br/>
  <input type="text" name="chain" size="3">
  <p/>
  Enter Help Information<br/>
  <textarea name="response" rows="8" cols="24"></textarea>
  <p/>
  <input type="submit" value="Update table" name="update">
</form>
</body>
</html>
```

InsertData和UpdateData类获得HTML表单发送的数据，将数据存储在helpdesk表中。

InsertData.php

```
<?php
include_once('UniversalConnect.php');
class InsertData
{
    private $tableMaster;
    private $hookup;

    public function __construct()
    {
```

```
        //指定表和链接
        $this->tableMaster="helpdesk";
        $this->hookup=UniversalConnect::doConnect();
        //从HTML表单
        $chain=$this->hookup->real_escape_string($_POST['chain']);
        $response=$this->hookup->real_escape_string($_POST['response']);

        //创建MySQL语句
        $sql = "INSERT INTO $this->tableMaster (chain,response) VALUES
               ('$chain','$response')";

        if($this->hookup->query($sql))
        {
            printf("Chain query: %s <br/>Response %s <br/> have been inserted
                into %s.",$chain,$response,$this->tableMaster);
        }
        //%s is a string from parameter
        elseif ( ($result = $this->hookup->query($sql))===false )
        {
              printf("Invalid query: %s <br/> Whole query: %s <br/>",
                  $this->hookup->error, $sql);
        exit();
        }
        $this->hookup->close();
    }
}
$worker=new InsertData();
?>
```

UpdateData.php

```
<?php
include_once('UniversalConnect.php');
class UpdateData
{
    private $tableMaster;
    private $hookup;

    public function __construct()
    {

        //制定表和链接
        $this->tableMaster="helpdesk";
        $this->hookup=UniversalConnect::doConnect();
        //从HTML表单
        $chain=$this->hookup->real_escape_string($_POST['chain']);
        $response=$this->hookup->real_escape_string($_POST['response']);

        //创建MySQL语句
        $sql = "UPDATE $this->tableMaster SET response='$response' WHERE chain=
               '$chain'";

        if($this->hookup->query($sql))
        {
            printf("Chain query: %s <br/>Response %s <br/> have been changed and
                set into %s.",$chain,$response,$this->tableMaster);
```

```
        }
        //%s is a string from parameter
        elseif ( ($result = $this->hookup->query($sql))===false )
        {
              printf("Invalid query: %s <br/> Whole query: %s <br/>",
                 $this->hookup->error, $sql);
          exit();
        }
        $this->hookup->close();
    }
}
$worker=new UpdateData();
?>
```

对于这个职责链咨询台，要在数据库表中输入或更新数据，这需要一个简单的表单。图13-2显示了这个数据输入/更新管理工具的外观。

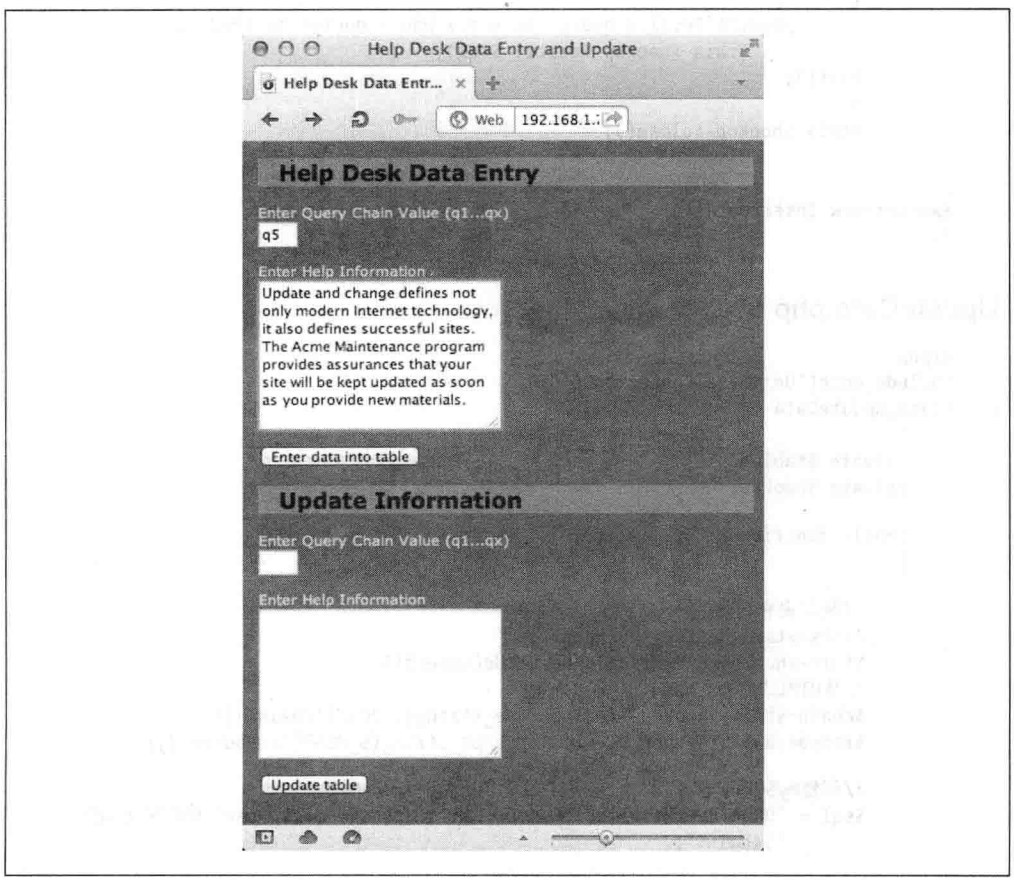


图13-2：输入响应数据

只有应用管理员才能输入帮助数据，用户无权访问这个数据输入表单。图13-2中的查询

链值（Query Chain Value）是用户搜寻的请求或帮助的一个标识符。这个特定的程序是这样组织的：请求用一个值来标识查询（q1到q（查询数））。在这个例子中，“咨询台”只有5个查询，所以是q1到q5。不过，完全可以有更多的查询。另外，一旦完成这个程序，如果想增加更多查询，可以使用咨询台数据输入模块（Help Desk Data Entry Module）来增加。接下来只需增加更多具体处理器，构成职责链模式。利用更新模块，咨询台就有了一个微型的内容管理系统（content management system，CMS）。

13.2.2 咨询台职责链

一旦输入请求的响应数据，可以使用职责链来提供一个程序，不仅获取信息，还可以对链序列排序。图13-3显示了这个咨询台应用的类图。

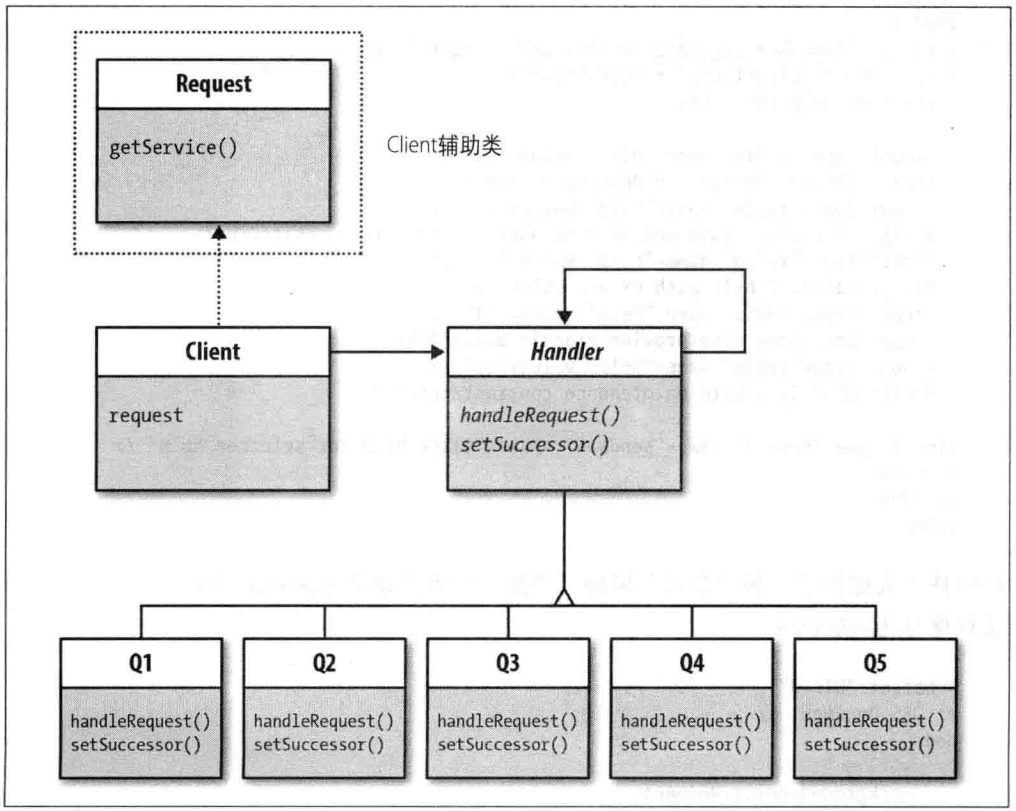


图13-3：咨询台类图

在职责链的这个实现中，Client使用一个辅助类（Request）首先发起一个请求，将这个请求发送到第一个具体处理器。Client会以所需的方式设置这个链序列。这里有5个具

体处理器类（从Q1到Q5），调用其中第一个具体处理器之后，这个具体处理器负责其余的工作，可能会处理查询，也可能将请求继续传递到下一个处理器类。

HTML数据输入、客户和请求参与者

数据输入表单是一个HTML文档，包含5个单选钮，分别对应不同的“帮助”请求。每个单选钮用一个值标识（表示相应的问题），从q1到q5：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<link href="help.css" rel="stylesheet" type="text/css" >
<title>Help Desk</title>
</head>
<body>
<img src="logo/acmelogo.png" width="143" height="143"/>
<form action="Client.php" method="post">
  <h2>Acme Help Desk</h2>
  <p/>
  <input type="radio" name="help" value="q1" />
    Website Design and Development<br/>
  <input type="radio" name="help" value="q2" />
    What about just adding a database on existing website?<br/>
  <input type="radio" name="help" value="q3" />
    Can Acme help with UX and UI?<br/>
  <input type="radio" name="help" value="q4" />
    Does Acme also provide graphic design?<br/>
  <input type="radio" name="help" value="q5" />
    What is a site maintenance contract?<br/>
  <p/>
<input type="submit" name="sendNow" value="Click here for selected help" />
</form>
</body>
</html>
```

CSS样式表提供了一种“公司”风格（当然，你可以建立你喜欢的任何样式）。将这个文件保存为*help.css*：

```
@charset "UTF-8";
/* CSS Document */
body
{
    background-color:#6A6A61;
    color:#DDDCC5;
    font-family:Verdana, Arial, Helvetica, sans-serif;
    font-size:11px;
    margin-left:12px;
}

h2
{
```

```
        background-color:#958976;
        color:#1D2326;
        font-family:Tahoma, Geneva, sans-serif
        font-size:18px;
        margin-left:0px;
        text-indent:1em;
    }
```

这里的想法是提供一种便捷的方法，允许用户选择，另外允许开发人员创建一个请求（将进入一个职责链流）。图13-4显示了这个用户界面，用户可以在这里选择要咨询的问题。

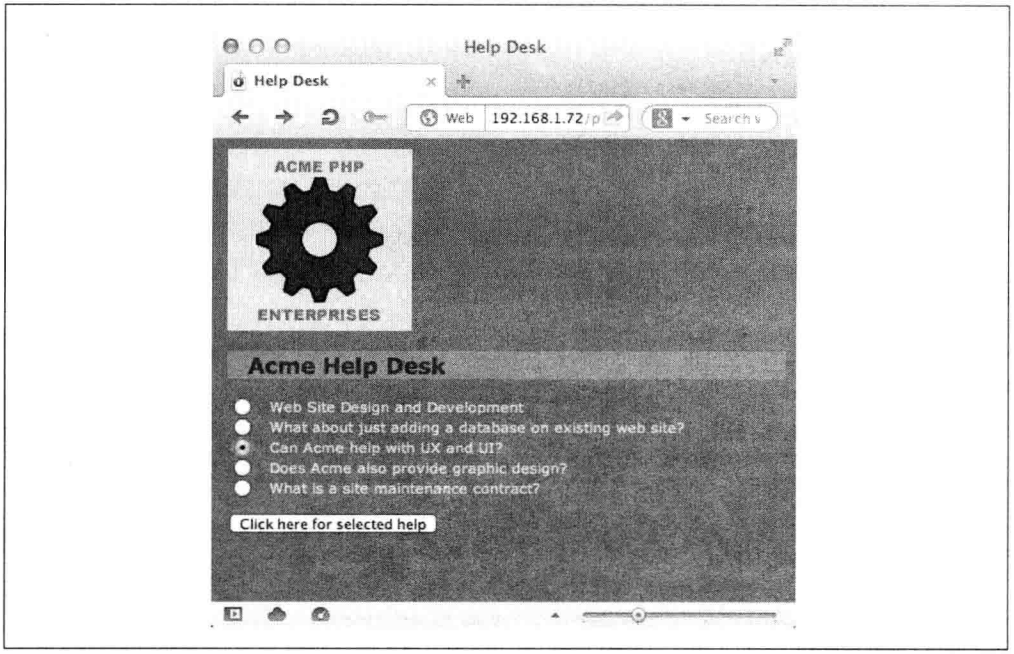


图13-4：数据输入用户界面

Client类处理从HTML表单发出的请求。它从一个$_POST变量接受请求，将请求保存在一个变量$queryNow中。然后Client实例化链中各个具体响应（处理器）的实例。实例化这些对象变量后，Client再使用setSuccessor()方法为各个具体处理器建立后继。所以，现在就会以序列顺序“加载”这个链：

```
<?php
function __autoload($class_name)
{
    include $class_name . '.php';
}
```

```
class Client
{
    private $queryNow;

    public function __construct()
    {
        if (isset($_POST['sendNow']))
        {
            $this->queryNow = $_POST['help'];
        }
        $q1 = new Q1();
        $q2 = new Q2();
        $q3 = new Q3();
        $q4 = new Q4();
        $q5 = new Q5();

        //设置后继
            $q1->setSuccessor($q2);
            $q2->setSuccessor($q3);
            $q3->setSuccessor($q4);
            $q4->setSuccessor($q5);

            //生成和处理加载请求
            $loadup = new Request($this->queryNow);
        //设置链首
             $q1->handleRequest ($loadup);
    }
}
$makeRequest = new Client();
?>
```

一旦实例化具体处理器，并分别指定了一个后继（但链上最后一个具体处理器除外，它没有后继），Client就可以做出请求了。为了帮助做出请求，Client使用了一个辅助类Request：

```
<?php
class Request
{
    private $value;

    public function __construct($service)
    {
        $this->value=$service;
    }

    public function getService()
    {
        return $this->value;
    }
}
?>
```

Request类提供了一个对象，可以沿着链传递，还包含一个方法来获取请求。

处理器接口和具体处理器

职责链的接口是一个抽象类。在这个实现中，它包括两个抽象方法，另外还包含多个属性（将由具体处理器使用）：

```
<?php
    abstract class Handler
    {
        protected $successor;
        protected $hookup;
        protected $tableMaster;
        protected $sql;
        protected $handle;
        abstract public function handleRequest($request);
        abstract public function setSuccessor($nextService);
    }
?>
```

这些具体处理器分别提供了方法来处理请求和设置后继。这个例子中，处理请求方法非常简单。handleRequest()方法传递请求（将请求作为参数）。（由Client首先发起请求，启动这个链——就像是点燃导火索。）如果$handle变量与Client通过Request辅助类传递的$request匹配，就会由这个具体处理器来处理这个查询。否则，它会将请求继续传递给链中的后继：

```
<?php
class Q1 extends Handler
{
    public function setSuccessor($nextService)
    {
        $this->successor=$nextService;
    }

    public function handleRequest ($request)
    {
        $this->handle="q1";
        if ($request->getService() == $this->handle)
        {
                $this->tableMaster="helpdesk";
            $this->hookup=UniversalConnect::doConnect();
            $this->sql = "SELECT response FROM $this->tableMaster WHERE chain=
                          '$this->handle'";

            if($result=$this->hookup->query($this->sql))
            {
             $row = $result->fetch_assoc();
             echo $row["response"];
            }
            $this->hookup->close();
        }
        else if ($this->successor != NULL)
        {
            $this->successor->handleRequest ($request);
```

```
        }
    }
}
?>
```

按照这个职责链的设计，用户只可有5个帮助请求，所以链尾的处理器后继为NULL——也就是说，Client没有为它定义后继。这样可以确保所有请求都有相应的处理器，不会有请求无法得到处理。

13.3 自动职责链和工厂方法

下面这个例子显示了职责链模式如何与工厂方法模式结合使用。这里不再由一个UI（用户界面）做出请求，职责链使用了一个日期函数作为“请求”。由此可以说明职责链的灵活性。接下来，程序中的具体处理器调用一个工厂方法应用，加载所请求的文本和图像。图13-5显示了这个应用中两个设计模式之间的关系。

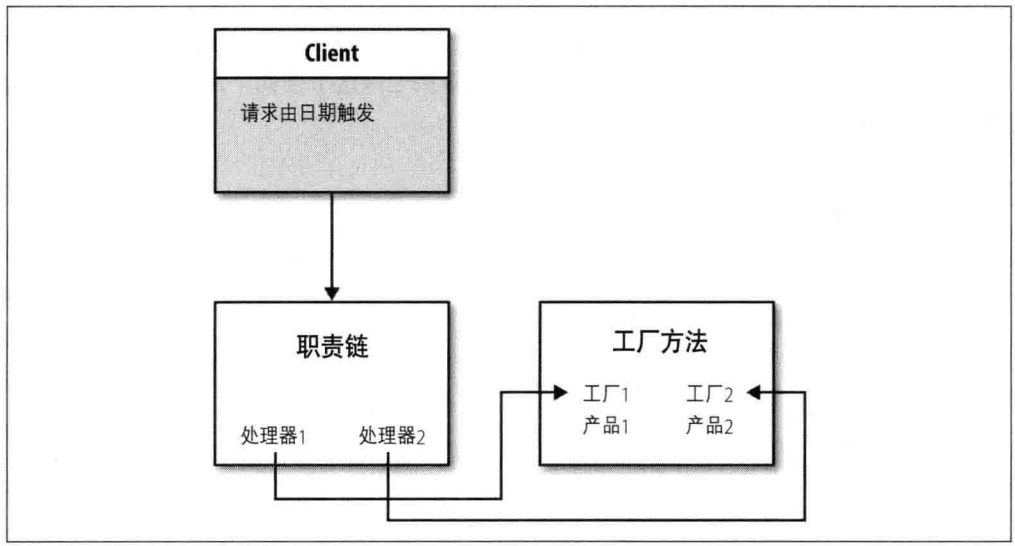


图13-5：职责链通过工厂方法模式处理请求

实际中，很多开发人员都可能根据应用的需要组合使用多个设计模式。工厂方法模式就经常与其他模式结合使用，以建立产品请求。

这个应用是针对学生每周要完成的一项加分作业设计的，这些学生们在研究全球饥荒情况。首先有一个地图，其中包括城市、河流、海洋和其他标志性地点，另外还要为页面增加一个照片和一个文本说明。学生们要找出正确的国家，明确饥荒情况、文明情况和性别差异，最后返回答案来得到学分。

13.3.1 职责链和日期驱动请求

链接到这个应用的地址后，会通过一个*index.php*文件自动启动Client。Client再把当前日期放在一个变量$queryNow中，这个变量将传递到一个Request辅助类。沿着职责链，查询会查找第一个合适的具体处理器作为工厂方法的客户，由它创建和显示图像和文字素材。

日期函数为什么改变：时间巴别塔

如果你用PHP创建过需要在世界范围使用的区分日期的应用，可能会注意到，可能要应用一些奇怪的时区规则（也可能不应用）。使用PHP 5.4时，日期对象要求你在*php.net/manual/en/timezones.php*中查找时区码，把它作为一个字符串直接量放在一个变量中来设置默认时区。例如，Client中使用下面的代码设置请求变量：

```
$tz= 'America/New_York';
date_default_timezone_set($tz);
$this->queryNow=getdate();
```

这里设置了美国东海岸的时区。有些国家，比如印度，时间与UTC相差半小时。中国西部的喀什与东部的北京相差一个时区，二者距离约3400 km（2100英里），按这个距离，在美国很可能跨4个时区。要把时区从“Europe/Minsk”设置为“Australia/Tasmania”，需要查看时区标识符（如果你住在印第安纳州，可能会查找8个时间标识符—“America/Indiana/Knox”——以及“America/Indiana/Indianapolis”和“America/Indianapolis”）。该学学地理了。

首先，要建立一个自动启动，下面的“触发器”文件会启动Client，并保存为*index.php*：

```
<?php
//index.php
//触发器文件
//可以用Clien类
//封装这个代码
//来替换这个文件
function __autoload($class_name)
{
    include $class_name . '.php';
}
$worker=new Client();
?>
```

除了由一个日期创建请求外，Client还负责建立后继列表：

```
<?php
//index.php 作为触发器
```

```
class Client
{
    private $queryNow;
    private $dateNow;

    public function __construct()
    {
        //得到所选时区的日期
        //参见php.net/manual/en/timezones.php
        $tz= 'America/New_York';
        date_default_timezone_set($tz);
        $this->queryNow=getdate();

            $d1 = new D1();
            $d2 = new D2();
            $d3 = new D3();
            $d4 = new D4();
            $d5 = new D5();
            $d6 = new D6();
            $d7 = new D7();
            $d8 = new D8();
            $d9 = new D9();
            $d10 = new D10();
            $d11 = new D11();
            $d12 = new D12();
            $d13 = new D13();
            $d14 = new D14();
            $d15 = new D15();

            $d1->setSuccessor($d2);
            $d2->setSuccessor($d3);
            $d3->setSuccessor($d4);
            $d4->setSuccessor($d5);
            $d5->setSuccessor($d6);
            $d6->setSuccessor($d7);
            $d7->setSuccessor($d8);
            $d8->setSuccessor($d9);
            $d9->setSuccessor($d10);
            $d10->setSuccessor($d11);
            $d11->setSuccessor($d12);
            $d12->setSuccessor($d13);
            $d13->setSuccessor($d14);
            $d14->setSuccessor($d15);
            //生成和处理加载请求
            $loadup = new Request($this->queryNow);
            $d1->handleRequest ($loadup);
    }
}
?>
```

这里相继建立了类D1到D15，它们是对应不同日期的具体处理器。如果第一个处理器（D1）与请求日期不匹配，则会继续查找下一个，直到找到正确的日期。这里的Request类与第一个职责链例子中相同，用作为Client辅助类：

```
<?php
class Request
{
    private $value;

    public function __construct($service)
    {
        $this->value=$service;
    }

    public function getService()
    {
        return $this->value;
    }
}
?>
```

处理器参与者同样是一个抽象类，作为一个接口。其中包含同样的方法，不过针对日期请求还增加了一些属性：

```
<?php
    abstract class Handler
    {
        protected $hungerFactory;
        protected $successor;
        protected $monthNow;
        protected $dayNow;
        protected $handleNow;
        abstract public function handleRequest($request);
        abstract public function setSuccessor($nextService);
    }
?>
```

接下来，实现请求处理的具体处理器包含有日期范围，用来确定这个具体处理器是否适合完成当前任务：

```
<?php
class D1 extends Handler
{
    public function setSuccessor($nextService)
    {
        $this->successor=$nextService;
    }

    public function handleRequest ($request)
    {
        $dateCheck= $request->getService();
        $this->monthNow=intval($dateCheck['mon']);
        $this->dayNow=intval($dateCheck['mday']);

        //$this->handleNow是一个布尔值
        //基于一个带日期范围的布尔表达式得到
        $this->handleNow=($this->monthNow == 9 && $this->dayNow >=15) &&
            ($this->monthNow == 9 && $this->dayNow <=22);
```

```
        if ($this->handleNow)
        {
                $this->hungerFactory=new HungerFactory();
            echo $this->hungerFactory->feedFactory(new C1());
        }
        else if ($this->successor != NULL)
        {
                $this->successor->handleRequest ($request);
        }
    }
}
?>
```

通过使用一个布尔变量（$handleNow）和布尔表达式，处理器首先查询$handleNow，如果为true，则指向一个工厂方法，由它加载必要的素材。从这个角度来看，职责链具体处理器（D1到D15）是一个客户，将从工厂方法做出一个请求。图13-6显示了这两个模式如何协作。

链接点的代码就包含在具体处理器对象（D1到D15）中。请求代码与所有其他客户请求并没有不同：

```
$this->hungerFactory=new HungerFactory();
echo $this->hungerFactory->feedFactory(new C1());
```

下一节将分析职责链具体处理器调用的工厂方法设计模式的作用。

13.3.2 工厂方法完成任务

工厂方法模式的工作与第5章中的一个例子类似（见图5-6），只是稍有区别。请求由具体处理器发出，而不是从“Client”类发出，不过这一点不会带来任何差异。如图13-6所示，handleRequest()方法相当于一个客户。

Creator和HungerFactory

标识客户来源之后，第一步构造Creator抽象类，这要作为工厂方法设计的接口：

```
<?php
//Creator.php
abstract class Creator
{
    protected $countryProduct;
    protected abstract function factoryMethod(Product $product);

    public function feedFactory(Product $productNow)
    {
        $this->countryProduct=$productNow;
        $mfg= $this->factoryMethod($this->countryProduct);
        return $mfg;
    }
```

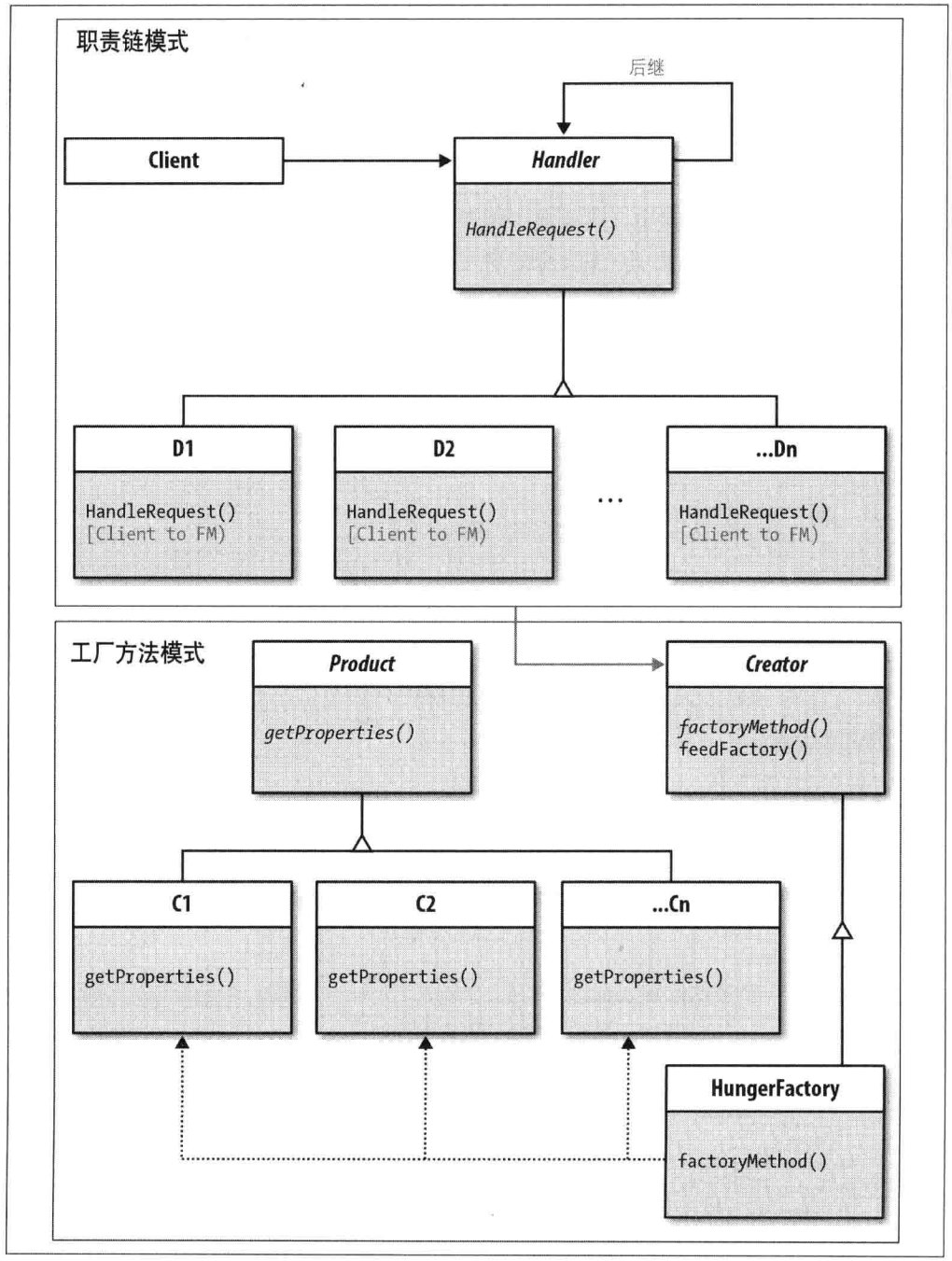


图13-6：职责链模式与工厂方法模式之间的关联

```
}
?>
```

factoryMethod()函数和feedFactory()方法使用了类型提示，以确保参数包含Product接口。不过，由于feedFactory()函数不是抽象方法，Creator的所有具体实现都会自动包含这个方法。

正如第5章中看到的，一些工厂方法模式实现会为不同的产品使用不同的工厂（具体创建者）。不过，如果产品类似，具体创建者可以处理任意多个不同的具体产品：

```
<?php
//HungerFactory.php
class HungerFactory extends Creator
{
    private $country;

    protected function factoryMethod(Product $product)
    {
        $this->country=$product;
        return($this->country->getProperties());
    }
}
?>
```

factoryMethod()实现包含一个参数，以便接收具体产品实例。在具体处理器中（即工厂的客户），这个参数在职责链中确定，并发送到工厂方法。各具体产品将满足通过职责链发送的原始请求。

产品和不同国家

最终结果显示了一个web页面，其中包含一个照片、一个地图和一段简短的文字说明。学生们使用这个应用研究全球饥荒情况，必须找出国家名、婴儿死亡率和不同文化程度的性别差异。

Product接口有一个方法getProperties()，HungerFactory使用这个方法得到所请求的产品。具体的产品类（C1到C15）显示了所需的各个部分：

```
<?php
class C1 implements Product
{
    private $mfgProduct;
    private $formatHelper;
    private $countryNow;

    public function getProperties()
    {
        //从外部文本文件加载文体说明
        $this->countryNow = file_get_contents("../hunger/c01/clue.txt");
```

```
        $this->formatHelper=new FormatHelper();
        $this->mfgProduct=$this->formatHelper->addTop();
        //加载图像
        $this->mfgProduct.="<img src='../hunger/c01/map.gif' width='300'
                            height='322'>";
        $this->mfgProduct .="<img class='pixLeft' src='../hunger/c01/pic.jpg'
                             width='200' height='400'>";
        $this->mfgProduct .="<p>$this->countryNow</p>";
        $this->mfgProduct .=$this->formatHelper->closeUp();
        return $this->mfgProduct;
    }
}
?>
```

集成产品之后，将返回给客户，在这里客户就是职责链中的具体处理器。图13-7显示了最后得到的产品。

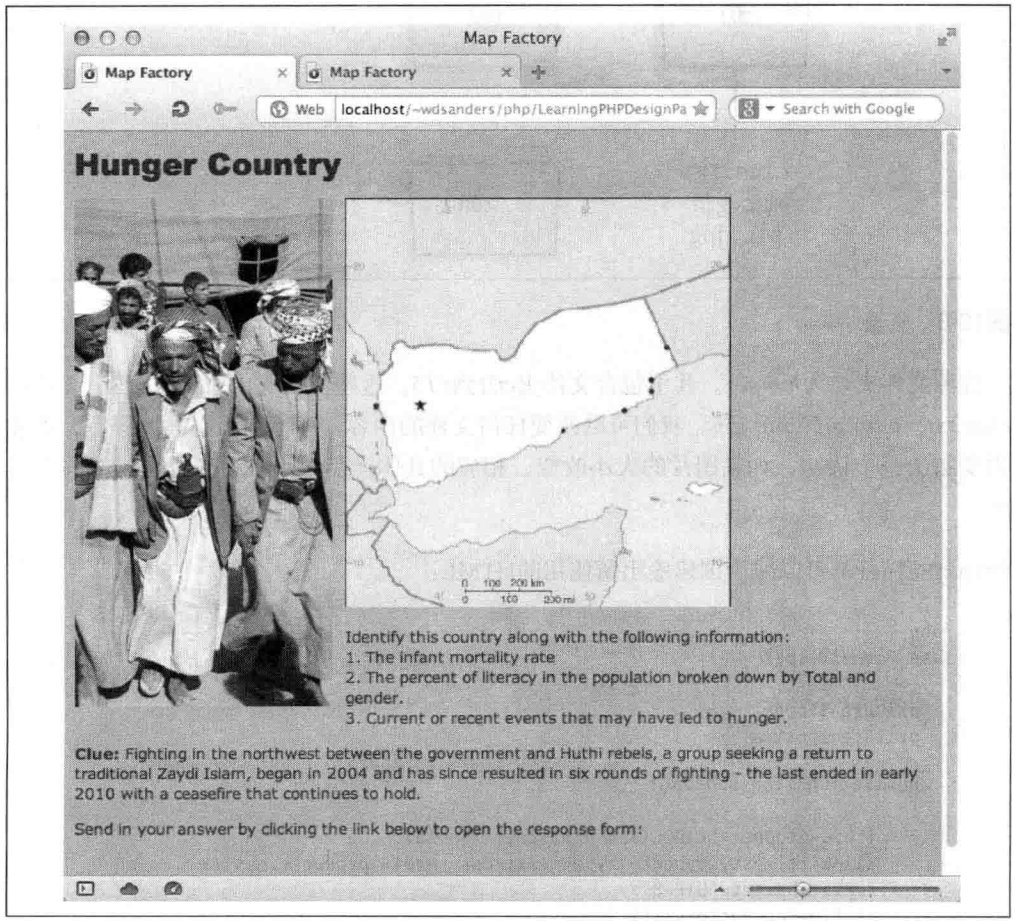


图13-7：具体产品显示

图片和文字部分放在单独的文件夹中。

辅助类、资源和样式

具体产品对象使用了外部样式表，另外有单独的文件夹提供文本和图片文件，还有一个辅助类来完成格式化。图13-8显示了整个应用的总布局。

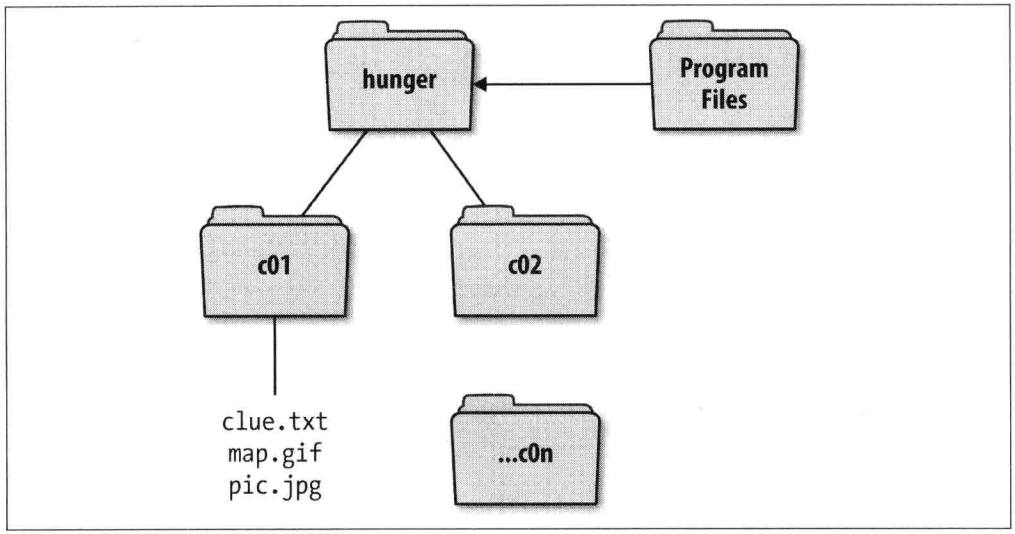


图13-8：资源

主资源文件夹名为*hunger*。其中包含文件夹*c01*到*c15*。这些文件夹中分别包含3个文件：*clue.txt*、*map.gif*和*pic.jpg*。我们可以改变任何文件的内容，而不用担心破坏程序。不过需要指出一个诀窍，如果图片的大小改变，相应的具体产品大小也必须改变，以与之匹配。

FormatHelper类可以减少创建输出所使用的HTML：

```
<?php
class FormatHelper
{
    private $topper;
    private $bottom;

    public function addTop()
    {
        $this->topper="<!DOCTYPE html><html><head>
        <link rel='stylesheet' type='text/css' href='products.css'/>
        <meta charset='UTF-8'>
        <title>Map Factory</title>
        </head>
        <body>
```

```
        <header>Hunger Country </header>";
    }

    public function closeUp()
    {
        $this->bottom="<br/><a href='Answer.html' target='_blank'>Click for
                        Credit</a>";
        $this->bottom.="</body></html>";
    }
}
?>
```

最后，需要利用外部样式表（products.css）把最终产品集成在一起，以便清楚地表达信息：

```
@charset "UTF-8";
/*C9DBF2,3B3E40,D9C6B0,8C7A70,BFACA4 */
/* CSS Document */
img
{
    padding: 10px 5px 0px 0px;
}

.pixRight
{
    float:right; margin: 0px 0px 5px 5px;
}

.pixLeft
{
    float:left; margin: 0px 5px 5px 0px;
}

header
{
    color:#900;
    font-size:24px;
    font-family:"Arial Black", Gadget, sans-serif;
}

body
{
    font-family:Verdana, Geneva, sans-serif;
    font-size:12px;
    background-color:#BFACA4;
    color:#3B3E40;
}

a
{
    text-decoration:none;
    color:#C9DBF2;
    font-weight:bold;
}
```

```
h1
{
    font-family:"Arial Black", Gadget, sans-serif;
    font-size:18px;
    background-color:#C9DBF2;
    color:#3B3E40;
}
```

对于所有这些部分，合理的组织至关重要，从长期来看，为这个应用增加新素材以及做出修改都极其容易。

13.4 易于更新

设计模式最重要的特性是：基于设计模式，开发人员可以顺利地做出改变以及增加新素材，而不会让整个应用像一摞纸牌一样坍塌。一般来讲，开发人员很可能为了达到一些短期目标而选择走捷径，以节省编程时间，但是从长远来看，这种做法有一个严重的后果：最后即使只做简单的修改也必须重构整个程序

上一节介绍的饥荒应用就是一个很好的例子，对于这样一个应用，重用和改变非常重要。由于这个应用是区分日期的，所以每个学期都需要改变，不过这很容易，因为所有日期查询都在D系列处理器（D1到D15）类中完成，这些处理器都作为职责链模式的一部分。同样，具体产品很容易修改，因为只要图片大小保持不变，所要修改的就只有新图像。名字是不变的（例如，*clue.txt*、*map.gif*），而且这样一来，开发人员不必操心程序在给定的文件夹中查找什么。如果增加了新素材，对应新增加的每个素材会增加一个新类，另外对于新增的资源还会增加一个新的文件夹。

出于某种原因，开发人员可能决定职责链采用一种不同的顺序。这很容易，只需要改变原客户的后继，其他所有内容都不用改变。不论如何计划，使用设计模式完成修改都会比尽可能缩减代码要容易得多。有些做法尽管短期奏效，但长期来看可能并不合适。

第14章

利用观察者模式构建多设备CMS

如果火星上有观察者，
他们可能会很惊讶我们居然能活这么久。
——诺姆·乔姆斯基

在梦境中……我们会让希望的事情发生，
因为对于梦境中所体验的经历，
我们不仅是观察者，同时也是创造者。
——毕尔·维拉亚特·汗

知识就是作为观察者所获得的结论，
经过科学培训的观察者会为我们提供所有能感知的现实。
——克里斯托弗·拉什

14.1 内置观察者接口

PHP 5.1.0以及更高版本有很多优秀的特性，其中之一就是提供了一组可以用于观察者设计模式的接口。这一章中，我们会从头开始介绍观察者模式，先不考虑PHP提供的内置特性。不过下面首先概要介绍SplObserver接口以及SplSubject和SplObjectStorage接口，利用这些接口，构建PHP观察者模式简直易如反掌。“SPL”是标准PHP库（Standard PHP Library）的简写，这个库中包括一组解决标准问题的接口和类。

不过，在继续介绍后面的内容之前，首先需要对观察者模式是什么以及它有什么作用有所认识。幸运的是，观察者模式的类图很详细，模型－视图－控制器（Model-View-Controller，MVC）模式中推崇的很多特性都可以在观察者模式中找到（你甚至可能会

认为这个模式是MVC的一种替代模式）。观察者模式的核心在于Subject和Observer接口。Subject包含一个给定的状态，观察者“订阅”这个主题，将主题的当前状态通知观察者。可以认为它是一个博客，有很多订阅者，会定期地为订阅或定期阅读博客的各类用户更新一组信息。每次博客改变时（其状态改变），所有订阅者“都会得到通知”。图14-1显示了观察者模式的类图。

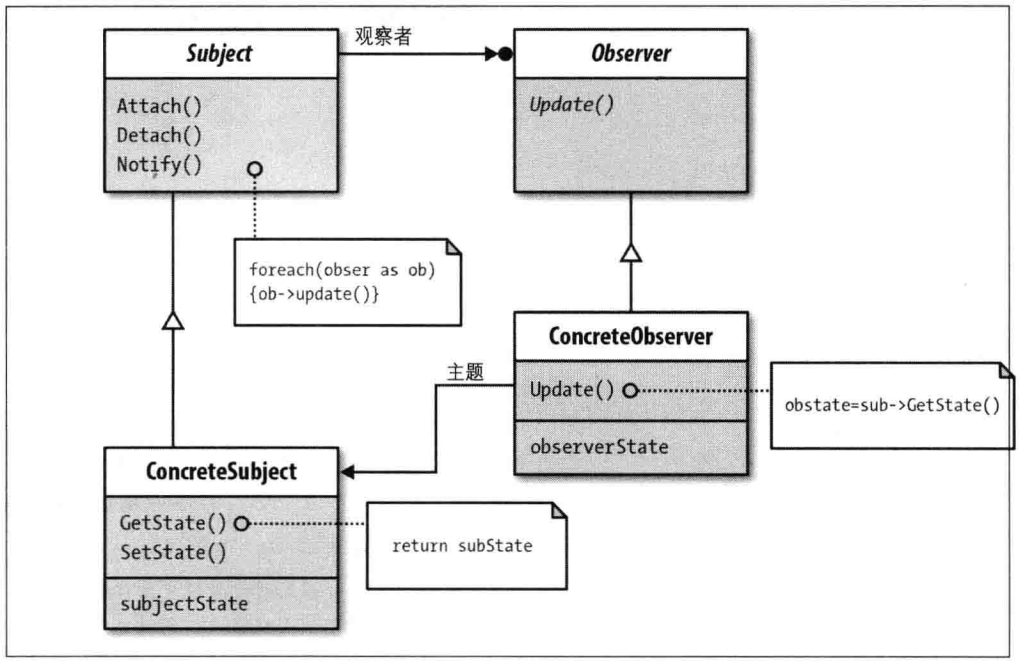


图14-1：观察者模式类图

这个模式最有意思（也可能是最复杂）的特性之一是Subject的方法。“*Subject*”用斜体表示这是一个接口（这里就是一个抽象类），抽象方法也显示为斜体。不过，从图14-1中可以看到，所有方法都没有用斜体。可以清楚地看到Subject生成哪些方法，Notify()方法甚至还有伪代码来帮助解释。你会看到有很多不同的观察者模式实现，甚至PHP还内置有自己的观察者模式实现。

14.2 何时使用观察者模式

设计观察者模式是为了让一个对象跟踪某个状态，知道状态何时改变，一旦状态改变，所有订阅对象都能得到通知。如果需要保证一个状态的一致性，但是这个给定状态可能有多个不同的视图，这种情况下观察者模式就很适用，而且很有帮助。利用观察者模式，可以维护一致性，同时记录创建一个给定状态的对象的个数。

观察者模式很直观。何必让多个对象创建或跟踪一个给定的状态呢？如果由一个对象完成这个工作，然后通知其他可能用到这个状态的对象，这样会合理得多。

对于PHP开发人员和设计人员来说，通常会遇到这样一个问题：要以文本、数字和图片形式获取一组数据，然后针对不同设备指定格式，为不同设备提供适当的Web表现。从智能手机到不同大小的平板电脑等移动设备，以及类似桌面计算机和笔记本电脑等非移动设备，都需要不同的设计配置。

不过，它们都需要同样的数据，图14-2描述了这种关系。

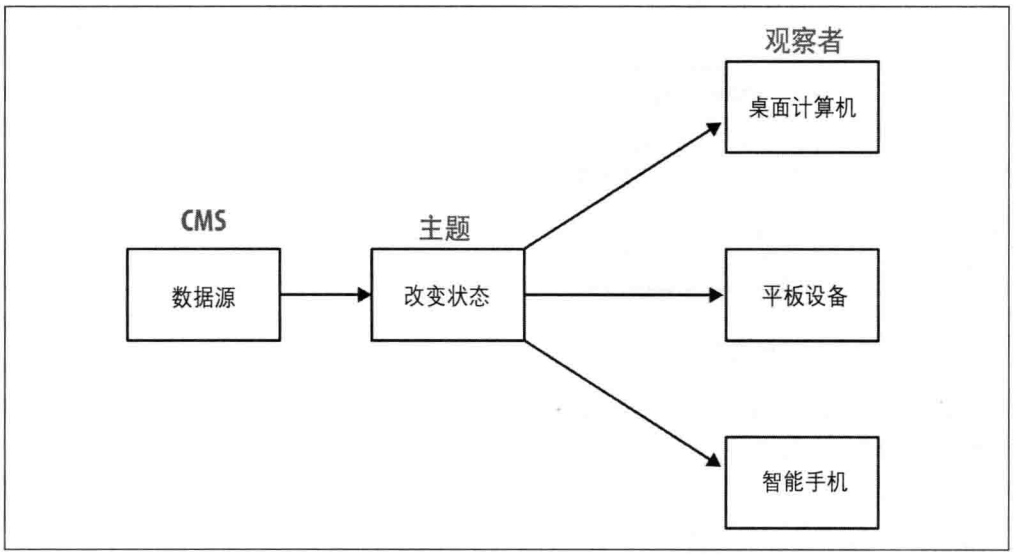


图14-2：CMS使用观察者模式

应用于一个内容管理系统（content management system，CMS）时，观察者设计模式可以确保所有配置都提供相同的内容。每个观察者可以使用相同的数据，开发人员和设计人员所要考虑的就是为不同的设备生成内容，得到Subject参与者的数据信息。

14.3 使用SPL实现观察者模式

可用于观察者设计模式的3个SPL接口/类如下：

- `SplSubject`
- `SplObserver`
- `SplObjectStorage`

下面简要地介绍各个接口/类，了解它们与观察者设计模式的关系。图14-3给出了一个简要的概念图。

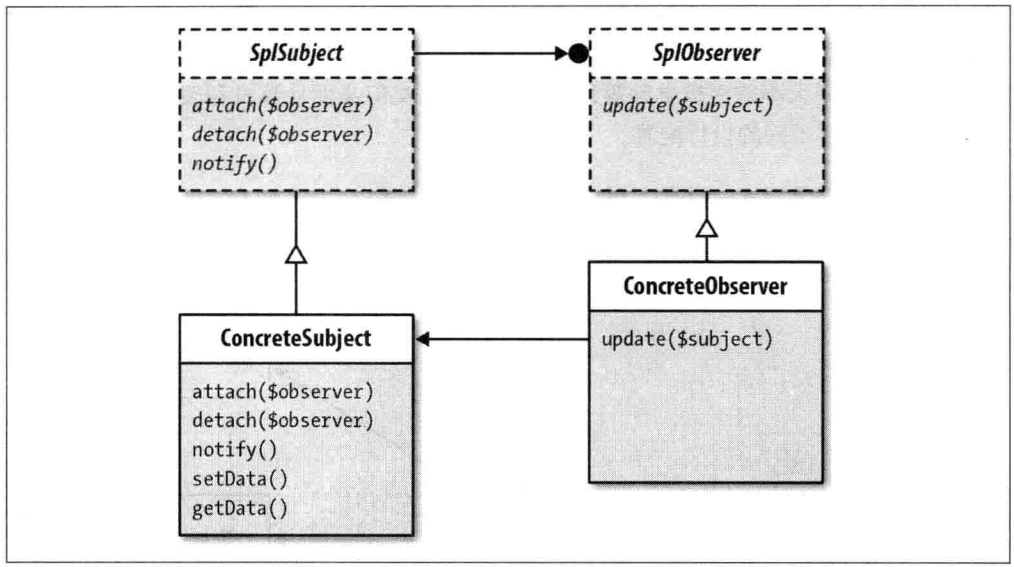


图14-3：SPL观察者模式类图

比较图14-3中的SPL观察者类图与图14-1中的GoF版本，所有相同的部分都保留，不过SPL类图中有更多抽象。

14.3.1 SplSubject

SplSubject接口有3个抽象方法，这与原来观察者类图中的方法类似。不过，SplSubject是一个接口，而原来Subject参与者通常是一个抽象类，因为它的方法不是抽象的。

《PHP Manual》给出了SplSubject的方法，如下所示：

```
abstract public void attach ( SplObserver $observer )
abstract public void detach ( SplObserver $observer )
abstract public void notify ( void )
```

从void可以看出，这些方法不返回任何结果。标准PHP接口中还要包含一个function关键字。如果自行创建这个接口，可能实现如下：

```
public function attach(SplObserver $observer);
public function detach(SplObserver $observer);
public function notify();
```

实现SplSubject接口时无需创建一个Subject接口或抽象类。重要的是，这个接口指定attach()和 detach()方法参数中$observer的数据类型必须是一个SplObserver对象。

14.3.2 SplObserver

SplObserver接口有一个方法update()，如下所示：

```
abstract public void update ( SplSubject $subject )
```

采用标准PHP接口格式，可以实现为：

```
public function update(SplSubject $subject);
```

update()方法对于观察者模式至关重要，因为它会得到Subject状态的最新变化，并交给观察者实例。不过，原来的GoF版本并未包含一个Subject数据类型的参数，在这一章后面的PHP观察者实现中，你还会看到未包含参数的实现以及这样做的限制。

14.3.3 SplObjectStorage

SplObjectStorage类与观察者设计模式没有内在的关系，不过通过它可以很方便地将观察者实例与一个主题实例相关联以及解除关联。尽管一般将SplObjectStorage类描述为从对象到数据或对象集的一个映射，但我更喜欢把它想成是一个数组（带有内置的attach()和detach()方法）。这个类提供了一种简单的方法，可以使观察者与发出状态改变通知的主题对象关联以及解除关联。

14.3.4 SPL具体主题

主题需要与观察者关联和解除关联，并通知订阅者（关联的观察者）发生了改变。私有变量$observers封装了这个属性。在这个实现中，$observers属性实例化为一个SplObjectStorage对象。可以把它想成是一个数组，然后关联各个$observer实例（SplObserver对象），存储单元是存储各个$observer实例的$observers。图14-4显示了这种关系。

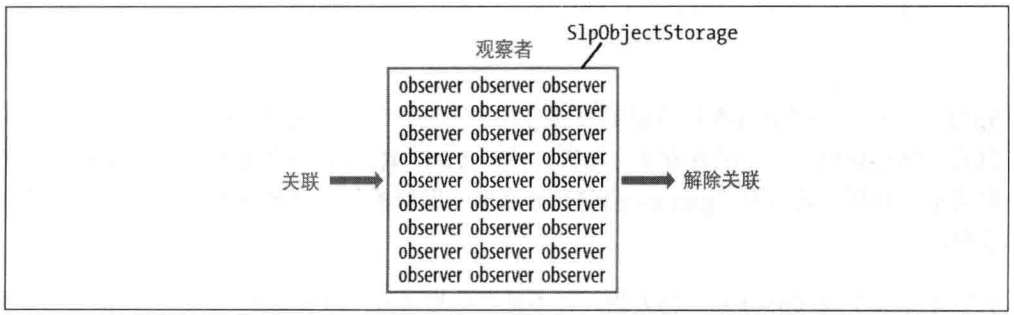


图14-4：SplObjectStorage中的具体观察者实例Observers

SplObjectStorage类最重要的方法是其内置attach()和detach()方法，利用这两个方法，可以很容易地指定哪些观察者实例将“订阅”和“解除订阅”更新通知：

```
<?
class ConcreteSubject implements SplSubject
{
    private $observers;
    private $data;

    public function setObservers()
    {
        $this->observers=new SplObjectStorage();
    }

    public function attach(SplObserver $observer)
    {
        $this->observers->attach($observer);
    }

    public function detach(SplObserver $observer)
    {
        $this->observers->detach($observer);
    }

    public function notify()
    {
        foreach ($this->observers as $observer) {

            $observer->update($this);
        }
    }

    public function setData($dataNow)
    {
        $this->data=$dataNow;
    }

    public function getData()
    {
        return $this->data;
    }
}
?>
```

SplSubject接口不包括获取方法和设置方法，不过这是观察者设计模式中的一部分，所以需要增加获取方法和设置方法。设置方法setData()包含一个参数，这是要增加的任何类型的数据。获取方法getData()存储当前主题状态，由具体观察者用来更新观察者数据。

另外还增加了setObservers()方法。并不是在构造函数中设置SplObjectStorage()实例（这需要ConcreteSubject类的一个新实例），也没有在setData()方法中设置观察者实

例，这里实现了一个单独的setObservers()方法，可以提供更松的耦合，并允许有多组观察者。

14.3.5 SPL具体观察者

具体观察者既简单又复杂。在这个例子中，它实现了更新函数来更新关联的观察者实例。update()方法的实现中包含一个代码提示，要求使用SplSubject接口作为方法参数。这就要求开发人员在所有update()调用中都要包含一个SplSubject实例：

```
<?
class ConcreteObserver implements SplObserver
{
    public function update(SplSubject $subject)
    {
        echo $subject->getData() . "<br />";
    }
}
?>
```

由于具体观察者只需要一个update()方法，所以不难看到，具体观察者可以由更健壮的类甚至其他设计模式的某些部分构成。在这一章后面会使用观察者模式开发一个CMS，你将看到更有用的观察者。

14.3.6 SPL客户

“SPL”Client类只是一个标准客户。这个客户按照SPL接口向具体主题和观察者发出多个请求，不过自己并没有实现SPL类或接口。

在这个例子中，Client创建了一个主题实例和3个具体观察者实例。然后使用setData()方法设置一个新状态，并将这3个观察者关联到这个主题。最后，它调用具体主题实例的notify()方法将当前状态发送给订阅的观察者：

```
<?php
//Client
function __autoload($class_name)
{
    include $class_name . '.php';
}
//
class Client
{
    public function __construct()
    {
        echo "<p>Create three new concrete observers, a new concrete subject:
              </p>";
        $ob1 = new ConcreteObserver();
        $ob2 = new ConcreteObserver();
```

```
        $ob3 = new ConcreteObserver();

        $subject = new ConcreteSubject();
        $subject->setObservers();
        $subject->setData("Here's your data!");
        $subject->attach($ob1);
        $subject->attach($ob2);
        $subject->attach($ob3);

        $subject->notify();

        echo "<p>Detach observer ob3. Now only ob1 and ob2 are notified:</p>";
        $subject->detach($ob3);
        $subject->notify();

        echo "<p>Reset data and reattach ob3 and detach ob2--only ob 1 and 3 are
              notified:</p>";
        $subject->setData("More data that only ob1 and ob3 need.");
        $subject->attach($ob3);
        $subject->detach($ob2);
        $subject->notify();
    }
}
$worker=new Client();
?>
```

Client的输出如下：

```
Create three new concrete observers, a new concrete subject:
Here's your data!
Here's your data!
Here's your data!

Detach observer ob3. Now only ob1 and ob2 are notified:

Here's your data!
Here's your data!

Reset data and reattach ob3 and detach ob2--only ob 1 and 3 are notified:

More data that only ob1 and ob3 need.
More data that only ob1 and ob3 need.
```

有一点需要注意，要把ConcreteSubject->attach()和ConcreteSubject->detach()方法与SplObjectStorage->attach()和SplObjectStorage->detach()方法区别开。ConcreteSubject类把SplObjectStorage的attach()和detach()方法包装在自己的attach()和detach()方法中。下面的伪代码显示了一个类版本，可以看到ConcreteSubject如何创建关联/解除关联（attach/detach）方法：

```
public function attach(SplObserver $observer)
    {
        SplObjectStorage->attach($observer);
    }
```

```
    public function detach(SplObserver $observer)
    {
        SplObjectStorage->detach($observer);
    }
```

如果把具体主题类的attach/detach方法改为其他名字，这可能同样会让人糊涂，所以只需要了解内置SPL attach/detach方法可以用来创建ConcreteSubject attach/detach方法。

14.4 自由的PHP和观察者模式

前面讨论了专门为设计模式设计的PHP内置SPL接口，现在转而考虑使用非SPL参与者，你可以从另一个角度了解观察者模式（更接近图14-1所示的原始结构）。二者的主要区别包括：

- Subject参与者是一个抽象类而不是一个接口。
- notify()方法在Subject中实现而不是在具体主题中实现。
- 具体主题使用一个数组对象而不是一个SplObjectStorage对象来存储观察者实例。

观察者参与者与SPL观察者基本上是一样的，只是Subject是一个抽象类（而不是接口），它包含一个具体notify()方法。这里只有一些很微小的差别，如具体观察者中包含一个$currentState属性。

14.4.1 抽象Subject类和ConcreteSubject实现

SPL实现中，Subject参与者是一个接口，而且所有方法都是抽象的，与SPL实现不同。下面这个实现更接近GoF原来提出的结构，使用了所实现的attach/detach方法。attach/detach方法包含一个Observer接口参数，这在结构上与SPL例子中使用SplObserver参数是一样的：

```
<?php
//Subject.php
abstract class Subject
{
    protected $stateNow;
    protected $observers=array();

    public function attachObser(Observer $obser)
    {
        array_push($this->observers,$obser);
    }

    public function detachObser(Observer $obser)
    {
        $position=0;
        foreach($this->observers as $viewer)
```

```
        {
            ++$position;
            if($viewer==$obser)
            {
                array_splice($this->observers,($position),1);
            }
        }
    }

    protected function notify()
    {
        foreach($this->observers as $viewer)
        {
            $viewer->update($this);
        }
    }
}
?>
```

notify()方法也是具体方法，这是继承得到的方法，将由Subject子类使用。因此，具体主题不必实现attachObser()、detachObser()或notify()，不过可以使用这些方法，而不需要在ConcreteSubject中增加任何代码：

```
<?php
//ConcreteSubject.php
class ConcreteSubject extends Subject
{
    public function setState($stateSet)
    {
        $this->stateNow=$stateSet;
        $this->notify();
    }

    public function getState()
    {
        return $this->stateNow;
    }
}
?>
```

在detachObser()方法中可以看到，必须迭代处理$observers数组来通知多个观察者。当然，即使没有使用SPL主题和观察者，也可以使用SplObjectStorage类。这样一来，可以使用一个SplObjectStorage对象（而不是一个数组）保存关联的观察者。另外，还可以使用SplObjectStorage类中内置的attach/detach方法。

14.4.2 观察者和多个具体观察者

到目前为止，一个观察者接口只对应一个具体观察者。在这个实现中，观察者模式则使用了多个观察者类型。最基本地，可以有多个具体观察者分别表示在一台桌面计算机、

一台与互联网连接的平板电脑和一部智能手机上提供的显示。主题生成一个图像URL，给定相同的主题状态，不同的观察者使用这个数据来生成图片。

Observer接口中只包含一个方法update()。它与SplObserver几乎完全一样：

```
<?php
//Observer.php
interface Observer
{
    function update(Subject $subject);
}
?>
```

抽象方法update()等待子类为它提供一个特定的实现。以下具体观察者的实现中差别很小，但这些差别对于开发来说很重要（不论是现在还是将来）。

ConcreteObserverDT（桌面计算机实现）

```
<?php
//ConcreteObserverDT.php
class ConcreteObserverDT implements Observer
{
    private $currentState;

    public function update(Subject $subject)
    {
        $this->currentState=$subject->getState();
        echo "<img src='desktop/$this->currentState'><br />";
    }
}
?>
```

ConcreteObserverTablet（平板电脑实现）

```
<?php
//ConcreteObserverTablet.php
class ConcreteObserverTablet implements Observer
{
    private $currentState;

    public function update(Subject $subject)
    {
        $this->currentState=$subject->getState();
        echo "<img src='tablet/$this->currentState'><br />";
    }
}
?>
```

ConcreteObserverPhone（智能手机实现）

```
<?php
//ConcreteObserverPhone.php
```

```
class ConcreteObserverPhone implements Observer
{
    private $currentState;

    public function update(Subject $subject)
    {
        $this->currentState=$subject->getState();
        echo "<img src='phone/$this->currentState'><br />";
    }
}
?>
```

可以看到，区别在于将在哪个目录中找到图像。例如，平板电脑实现使用了下面这行代码：

```
echo "<img src='tablet/$this->currentState'><br />";
```

另外两个实现则使用了按当前特定设备命名的目录，因为不论所订阅的主题生成什么消息作为当前状态，对于所有订阅者来说都是一样的。在这个特定的实现中，ConcreteSubject的当前状态是一个图像URL，图像的包装器是使用<img>标记的一个HTML行。

14.4.3 客户

在这里，**Client**很可能来自以下三种类型的设备：桌面计算机、平板电脑或手机。为了便于说明，下面客户会调用这3个实现来展示不同大小的显示：

```
<?php
//Client.php
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    public function __construct()
    {
        $sub=new ConcreteSubject();

        $ob1=new ConcreteObserverPhone();
        $ob2=new ConcreteObserverTablet();
        $ob3=new ConcreteObserverDT();

        $sub->attachObser($ob1);
        $sub->attachObser($ob2);
        $sub->attachObser($ob3);
        $sub->setState("decoCar.png");
    }
}
$worker=new Client();
?>
```

实例集包括一个具体主题和3个不同具体观察者的3个实例。这3个观察者实例都使用相同的主题源提供数据。通过使用具体主题的setState()方法，Client设置当前状态，所有观察者都将使用这个状态。图14-5给出了平板电脑设备中显示的结果。

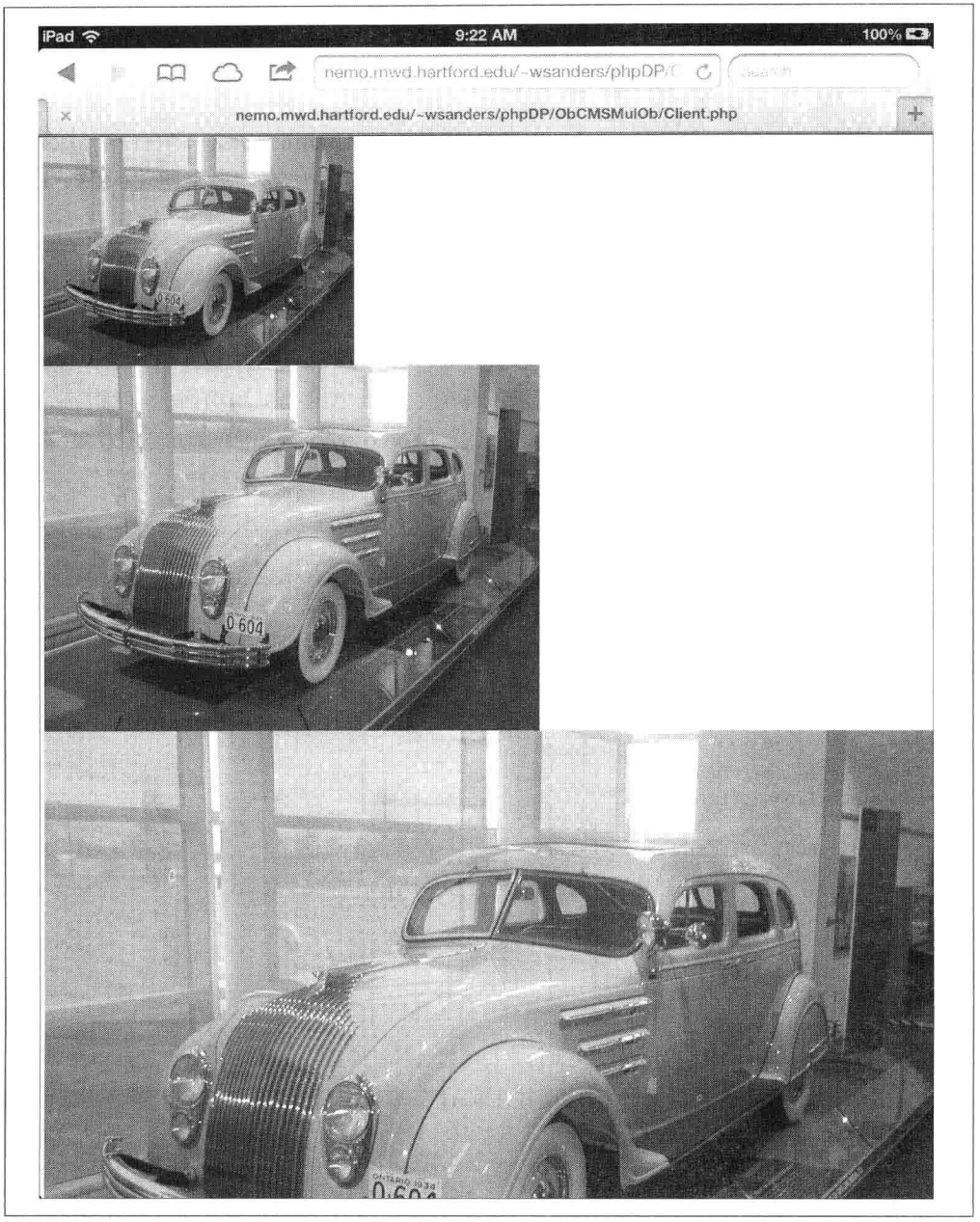


图14-5：给定同一个具体主题状态，不同的具体观察者生成不同大小的图像

随着移动和桌面设备的等级分得越来越细，一个好的网站可能需要更多组位图图像，以便用户得到最佳的体验。同样，可能需要更多CSS文件来优化不同设备上的文本显示。不过，由于有一个数据源而有多个订阅者，多个订阅者可以订阅同一个主题，所以跟踪当前状态会容易得多。

14.5 建立一个简单CMS

我们要构建的“CMS”将使用一个MySQL数据库来存储所用的数据。将在一个MySQL文本字段中存储设计模式的一个小结，最后，在第三个字段中存储一个图片URL。这个表还包含一个自动生成的字段，存储记录的唯一编号（整数）。

这里使用观察者模式从这个表得到信息，并把信息发送到所选择的观察者。数组中存储的数据集会分发给不同的观察者，他们可能通过桌面计算机、平板电脑或智能手机查看结果。这里提供了一个HTML选择表单（通过PHP生成为web页面）允许用户选择模式。手机浏览使用jQuery Mobile来优化浏览者的体验。平板电脑浏览和桌面计算机浏览都采用两栏布局，手机浏览使用单栏。

14.5.1 CMS工具

首先创建一个MySQL表，另外创建一个模块为Web页面输入和更新数据。先用本书一直使用的连接工具接口和类创建表。如下：

CMS表

```
<?php
//文件名: IConnectInfo.php
interface IConnectInfo
{
    const HOST ="localhost";
    const UNAME ="uname";
    const PW ="password";
    const DBNAME = "dataBase";

    public function doConnect();
}

?>
```

之前会提醒用户已经成功建立连接，现在去掉这个输出，因为这个连接类还会用于程序中的其他MySQL连接：

```
<?php
include_once('IConnectInfo.php');
class UniversalConnect implements IConnectInfo
```

```
{
    private static $server=IConnectInfo::HOST;
    private static $currentDB= IConnectInfo::DBNAME;
    private static $user= IConnectInfo::UNAME;
    private static $pass= IConnectInfo::PW;
    private static $hookup;

    public function doConnect()
    {
        self::$hookup=mysqli_connect(self::$server, self::$user, self::$pass,
                    self::$currentDB);
        if(self::$hookup)
        {
            //注释，用于调试
        }
        elseif (mysqli_connect_error(self::$hookup))
        {
            echo('Here is why it failed: ' . mysqli_connect_error());
        }
        return self::$hookup;
    }
}
?
```

下面的代码用来创建表：

```
<?php
include_once('UniversalConnect.php');
class CreateTable
{
    private $tableMaster;
    private $hookup;

    public function __construct()
    {
        $this->tableMaster="cms";
        $this->hookup=UniversalConnect::doConnect();

        $drop = "DROP TABLE IF EXISTS $this->tableMaster";

        if($this->hookup->query($drop) === true)
        {
            printf("Old table %s has been dropped.<br/>",$this->tableMaster);
        }

        $sql = "CREATE TABLE $this->tableMaster (
            id          SERIAL,
            dpHeader    NVARCHAR(50),
            textBlock   TEXT,
            imageURL    NVARCHAR(60),
                        PRIMARY KEY (id))";
        if($this->hookup->query($sql) === true)
        {
            printf("Table $this->tableMaster has been created successfully.
                   <br/>");
        }
```

```
            $this->hookup->close();
        }
    }
    $worker=new CreateTable();
    ?>
```

一旦建立了表，下面需要在这个表中增加数据（将在设计模式Web页面中使用）。

CMS数据输入和更新

CMS表存储Web页面所需的数据。用户界面（*Admin.html*）是一个简单的HTML文档，它调用PHP类来输入和更新文本说明：

```
<!doctype html>
<!-- Admin.html -->
<html>
<head>
<meta charset="UTF-8">
<link rel="stylesheet" type="text/css" href="desktop.css">
<title>CMS Admin Module</title>
</head>
<body>
<h1>CMS Administrative Module</h1>
<h2>Data Entry</h2>
<form action="DataEntry.php" method="post">
  <input type="text" name="dpHeader">
   Design pattern name<br />
  Write up for design pattern:<br />
  <textarea name="textBlock" cols="48" rows="16"></textarea>
  <p />
  <input type="text" name="imageURL">
   Graphic URL
  <p />
  <input type="submit" name="entry" value="Enter Page Data">
</form>
<h2>Data Update</h2>
<form action="DataUpdate.php" method="post">
  <input type="text" name="dpUpdate">
   Design pattern name to update<br />
  New write-up for design pattern:<br />
  <textarea name="newData" cols="48" rows="16"></textarea>
  <p />
  <input type="submit" name="update" value="Update Page Data">
</form>
</body>
</html>
```

这个数据输入模块向3个字段输入适当的数据（对应各个页面的页眉、体文本和图像）。

```
<?php
//DataEntry.php
include_once('UniversalConnect.php');
```

```
class DataEntry
{
    private $tableMaster;
    private $hookup;
    private $sql;

    public function __construct()
    {
        $this->tableMaster="cms";
        $this->hookup=UniversalConnect::doConnect();

        if ( $_POST['dpHeader'] )
        $dpHeader=$this->hookup->real_escape_string($_POST['dpHeader']);
        if ( $_POST['textBlock'] )
        $textBlock=$this->hookup->real_escape_string($_POST['textBlock']);
        if ( $_POST['imageURL'] )
        $imageURL=$this->hookup->real_escape_string($_POST['imageURL']);

        $this->sql = "INSERT INTO $this->tableMaster
            (dpHeader,textBlock,imageURL) VALUES
            ('$dpHeader','$textBlock','$imageURL')";

        if($this->hookup->query($this->sql))
        {
            printf("Successful data entry for table: $this->tableMaster <br/>");
        }
        elseif ( ($result = $this->hookup->query($this->sql))===false )
        {
              printf("Invalid query: %s <br/> Whole query: %s <br/>",
                      $this->hookup->error, $this->sql);
              exit();
        }
        $this->hookup->close();
    }
}
$worker=new DataEntry();
?>
```

这个CMS的第二个管理工具用来更新设计模式的内容说明。用户查看很长的文字说明时都会遇到一个问题：空间太小。即使采用单列布局，看起来文字说明也需要调整，这个工具允许Web管理员完成所需的各种调整。

```
<?php
//DataUpdate.php
include_once('UniversalConnect.php');
class DataUpdate
{
    private $tableMaster;
    private $hookup;
    private $sql;

    public function __construct()
    {
        $this->tableMaster="cms";
        $this->hookup=UniversalConnect::doConnect();
```

```
        if ( isset($_POST['dpUpdate'] ))
        $dpHeader=$this->hookup->real_escape_string($_POST['dpUpdate']);
        if ( $_POST['newData'] )
        $newData=$this->hookup->real_escape_string($_POST['newData']);

        $changeField="textBlock";

        $this->sql = "UPDATE $this->tableMaster SET $changeField='$newData'
                      WHERE dpHeader='$dpHeader'";

        if ($result = $this->hookup->query($this->sql))
        {
            echo "$changeField changed to:<br /> $newData";
        }
        else
        {
            echo "Change failed: " . $hookup->error;
        }
    }
}
$worker=new DataUpdate();
?>
```

最后，对于使用桌面计算机浏览页面的人，将CSS设置为两栏，一栏用于显示图片，另一栏显示文字说明，这对于单栏管理UI也同样适用：

```
//CSS
@charset "UTF-8";
/* desktop.css */
/* CSS Document */
/* 595241,B8AE9C,FFFFFF,ACCFCC,8A0917 */
body
{
    font-family:Verdana, Geneva, sans-serif;
    background-color:#ffffff;
    color:#595241;
    padding-right:10px;
}

h1
{
    font-family:"Arial Black", Gadget, sans-serif;
    text-align:center;
    color:#8A0917;
    background-color:#ACCFCC;
}

img
{
    padding-right:10px;
    float:left;
}
```

CSS文件*desktop.css*也可以用于来自非移动设备的请求，这样一来，将来引用时可以使用同一个文件。

14.5.2 多个设备观察者

通过这一章前面的例子，你对观察者模式的基本结构应该已经很熟悉了。这个项目的CMS部分要为Web页面存储和更新数据，所以这一部分需要获取数据，并把数据放在为不同浏览设备（智能手机、平板电脑或桌面计算机）配置的页面上。为了全面地认识整个项目和观察者设计模式的作用，图14-6给出了一个文件图，其中包括所有对象和相关的部分。

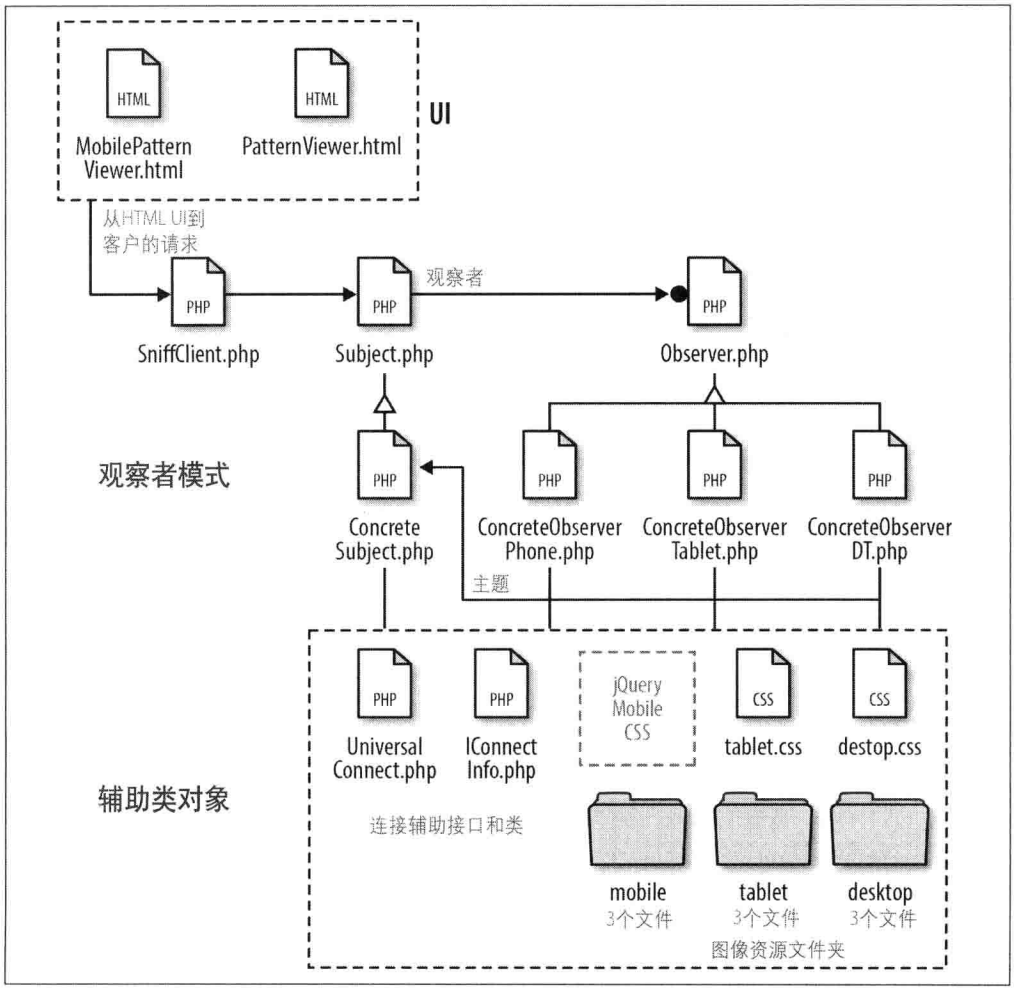


图14-6：观察者项目文件图

这个项目分为3组：HTML UI、PHP观察者模式以及辅助类对象。这一节接下来会解释这些部分如何协作。下面给出对象的通信序列：

1. 用户通过HTML UI请求Web页面。
2. 请求转发到SniffClient，由它确定浏览设备。
3. 根据浏览设备创建具体观察者实例。
4. 具体主题实例关联由设备确定的观察者。
5. 利用用户通过ConcreteSubject类选择的页面设置MySQL数据的当前状态。
6. 订阅（关联）观察者从具体主题接收数据。
7. 具体观察者从具体主题得到状态信息，并在屏幕上显示页面。

当然，所有请求对用户都是透明的。

两个HTML UI文档

为了避免请求一个页面（一种设计模式）时出现键入错误，要使用单选钮来选择设计模式：

```
<!doctype html>
<html>
<head>
<meta charset="UTF-8">
<link rel="stylesheet" type="text/css" href="desktop.css">
<title>Design Pattern Summary Viewer</title>
</head>
<body>
<h1>Select Pattern</h1>
<h2>Available Patterns</h2>
<form action="SniffClient.php" method="post">
  <input type="radio" name="dp" id="tm" value="Template Method" />
  <label for="tm">Template Method</label>
  <br/>
  <input type="radio" name="dp" id="bld" value="Builder" />
  <label for="bld">Builder</label>
  <br/>
  <input type="radio" name="dp" id="fm" value="Factory Method" />
  <label for="fm">Factory Method</label>
  <p/>
  <input type="submit" value="View Pattern">
</form>
</body>
</html>
```

图14-7显示了这个UI。

对于图14-7所示的通用UI，要在一个移动设备上读取并做出选择会很困难，所以我们为移动设备开发了另一个UI。幸运的是，整个移动设备格式都可以从jQuery Mobile（*http://jquerymobile.com*）得到，为手机设备开发的第二个UI中采用了这种格式：

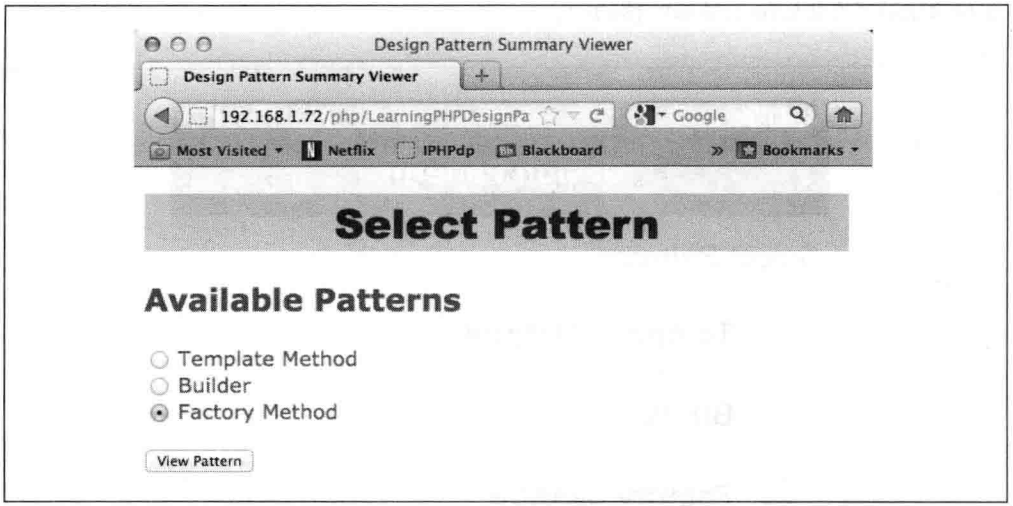


图14-7：请求Web页面的HTML UI

```
<!DOCTYPE html>
<html>
<head>
<title>Mobile Viewer</title>
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="stylesheet" href="http://code.jquery.com/mobile/1.2.0
                            /jquery.mobile-1.2.0.min.css" />
<script src="http://code.jquery.com/jquery-1.8.2.min.js"></script>
<script src="http://code.jquery.com/mobile/1.2.0/jquery.mobile-1.2.0.min.js">
            </script>
</head>
<body>
<div data-role="page">
  <div data-role="header">
    <h1>Design Patterns</h1>
  </div>
  <form action="SniffClient.php" method="post">
    <fieldset data-role="controlgroup">
      <legend> Select Pattern:</legend>
      <input type="radio" name="dp" id="tm" value="Template Method" />
      <label for="tm">Template Method</label>
      <input type="radio" name="dp" id="bld" value="Builder" />
      <label for="bld">Builder</label>
      <input type="radio" name="dp" id="fm" value="Factory Method" />
      <label for="fm">Factory Method</label>
    </fieldset>
    <input type="submit" data-theme="e" value="View Pattern">
  </form>
  <div data-role="footer"> PHP Patterns</div>
</div>
</body>
</html>
```

图14-8显示了在iPhone上显示的移动UI。

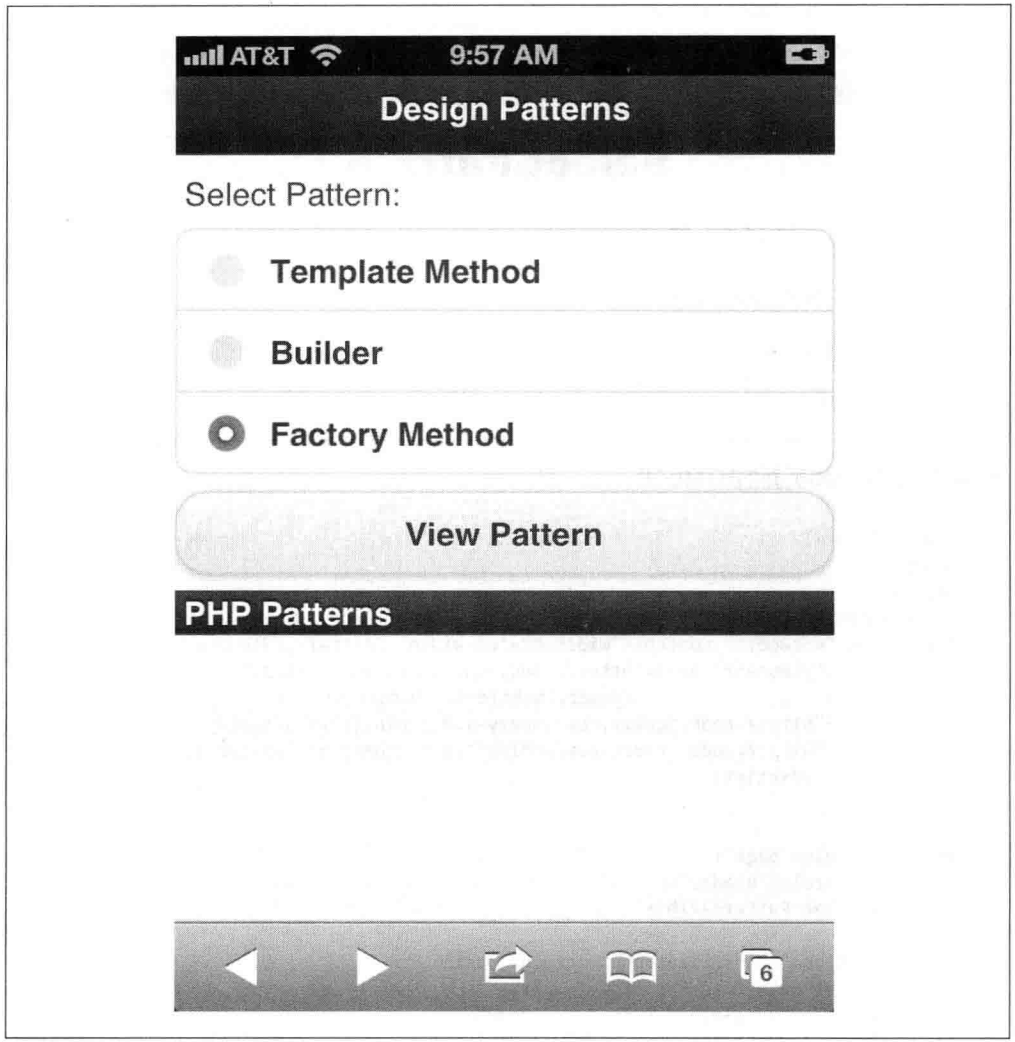


图14-8：移动UI

这两个UI的工作是一样的。用户选择一个要查看的模式，然后轻点一个按钮生成页面。遗憾的是，这个UI不像浏览页面那样会自动选择。

监听者客户

由于会根据用户的设备优化请求，所以客户的第一个任务就是确定在使用什么设备。SniffClient类从一个移动设备子集中选择设备（默认值为桌面设备）：

```
<?php
//用户代理作为对象属性
function __autoload($class_name)
{
    include $class_name . '.php';
}
class SniffClient
{
    private $userAgent;
    private $mobile=false;
    private $deviceObserver;
    private $dpNow;
    private $sub;

    public function __construct()
    {
        if (isset($_POST['dp'] ))
            $this->dpNow=$_POST['dp'];
        $this->sub=new ConcreteSubject();
        $this->userAgent=$_SERVER['HTTP_USER_AGENT'];
        if(stripos($this->userAgent,'iphone'))
        {
            $this->mobile=true;
            $this->deviceObserver=new ConcreteObserverPhone();
        }
        if(stripos($this->userAgent,'android'))
        {
            $this->mobile=true;
            $this->deviceObserver=new ConcreteObserverPhone();
        }
        if(stripos($this->userAgent,'blackberry'))
        {
            $this->mobile=true;
            $this->deviceObserver=new ConcreteObserverPhone();
        }
        if(stripos($this->userAgent,'ipad'))
        {
            $this->mobile=true;
            $this->deviceObserver=new ConcreteObserverTablet();
        }
        if(stripos($this->userAgent,'trident'))
        {
            $this->mobile=true;
            $this->deviceObserver=new ConcreteObserverTablet();
        }
        if(stripos($this->userAgent,'kindle fire'))
        {
            $this->mobile=true;
            $this->deviceObserver=new ConcreteObserverTablet();
        }
        if(stripos($this->userAgent,'silk'))
        {
            $this->mobile=true;
            $this->deviceObserver=new ConcreteObserverTablet();
        }
```

```
        if(!$this->mobile)
        {
            $this->deviceObserver=new ConcreteObserverDT();
        }
            $this->sub->attachObser($this->deviceObserver);
            $this->sub->setState($this->dpNow);
        }
}

$worker=new SniffClient();

?>
```

这个客户很简单，可以转换为一个职责链模式，根据需要增加新处理器。不过，为了展示如何使用观察者模式实现一个简单的CMS，并根据不同设备提供相应的功能，这个客户强调了观察者模式的这些重要功能。

Subject类

这个观察者实现的接口是一个抽象类，不过提供了具体方法来关联和解除关联观察者。类似地，这里实现了notify()方法，将由ConcreteSubject类使用：

```
<?php
//Subject.php
abstract class Subject
{
    protected $observers=array();

    public function attachObser(Observer $obser)
    {
        array_push($this->observers,$obser);
    }

    public function detachObser(Observer $obser)
    {
        $position=0;
        foreach($this->observers as $viewer)
        {
            ++$position;
            if($viewer==$obser)
            {
              array_splice($this->observers,($position),1);
            }
        }
    }

    protected function notify()
    {
        foreach($this->observers as $viewer)
        {
            $viewer->update($this);
        }
    }
```

```
}
?>
```

这个应用中的ConcreteSubject类连接到cms表，得到所需的数据以满足用户的请求。attach/detach方法与前面例子中使用的attach/detach方法是一样的，不过setState()方法与前例中的setState()方法有很大不同：

```
<?php
//ConcreteSubject.php
class ConcreteSubject extends Subject
{
    private $hookup;
    private $tableMaster;
    private $designPattern;
    private $stateSet=array();

    public function setState($dpNow)
    {
        $this->designPattern=strtolower($dpNow);
        $this->tableMaster="cms";
        $this->hookup=UniversalConnect::doConnect();

        //创建查询语句
        $sql = "SELECT * FROM $this->tableMaster WHERE dpheader=
                '$this->designPattern'";
        //从MySQL表将适当的数据增加到$stateSet数组
        if ($result = $this->hookup->query($sql))
        {
            while($row=$result->fetch_assoc())
            {
                    $this->stateSet[0]=$row["dpHeader"];
                    $this->stateSet[1]=$row["textBlock"];
                    $this->stateSet[2]=$row["imageURL"];
            }
            $result->close();
        }
    $this->hookup->close();
    //update()方法是notify()的一部分
    //这个方法在Subject中实现为具体方法
    $this->notify();
    }

    public function getState()
    {
        return $this->stateSet;
    }
}
?>
```

所需的全部数据都放在一个数组中（$stateSet），通过getState()方法使关联的观察者可以使用这些数据。

多个具体观察者

与前面的例子相比，Observer接口未做任何改变：

```
<?php
//Observer.php
interface Observer
{
    function update(Subject $subject);
}
?>
```

update()方法需要一个Subject实例作为参数，不过，除此以外，没有声明其他方法。

Observer接口的实现则完全不同。每个具体观察者使用相同的主题数据，不过它们创建的页面完全不同。

移动手机观察者。为移动手机生成页面时，最大的问题是文本和图像的大小。通过使用CSS和JavaScript，可以创建可动态调整大小的文本和按钮，从而建立一个可用的手机Web页面。不过，如果使用jQuery Mobile，很多工作都已经由jQuery Mobile完成。所以ConcreteObserverPhone类使用预建的jQuery Mobile来建立适用于手机的格式，如下所示：

```
<?php
//ConcreteObserverPhone.php
class ConcreteObserverPhone implements Observer
{
    private $currentState=array();
    private $dpHeader;
    private $bodytext;
    private $imageURL;

    public function update(Subject $subject)
    {
        $this->currentState=$subject->getState();
        $this->dpHeader=$this->currentState[0];
        $this->bodytext=$this->currentState[1];
        $this->imageURL=$this->currentState[2];
        $this->doMobile();
    }

    private function doMobile()
    {
$showPage = <<<MOBILE
    <html>
        <head>
            <title>Mobile Page</title>
            <meta name="viewport" content="width=device-width, initial-scale=1">
            <link rel="stylesheet" href="http://code.jquery.com/mobile/1.2.0/
                                        jquery.mobile-1.2.0.min.css" />
            <script src="http://code.jquery.com/jquery-1.8.2.min.js"></script>
            <script src="http://code.jquery.com/mobile/1.2.0/
                        jquery.mobile-1.2.0.min.js"></script>
```

```
        </head>
    <body>

        <div data-role="page">
        <div data-role="header">
             <h1>$this->dpHeader</h1>
        </div>

        <div data-role="content">
        <p>$this->bodytext</p>
        <img src="mobile/$this->imageURL" alt="image urls" width="100%">
        </div>
        </div>

    </body>
    </html>
MOBILE;
    echo $showPage;
    }
}
?>
```

图14-9给出了iPhone上显示的页面，这里采用了专门为移动手机建立的格式。可以看到，文本足够大，很容易阅读而不必调整大小。

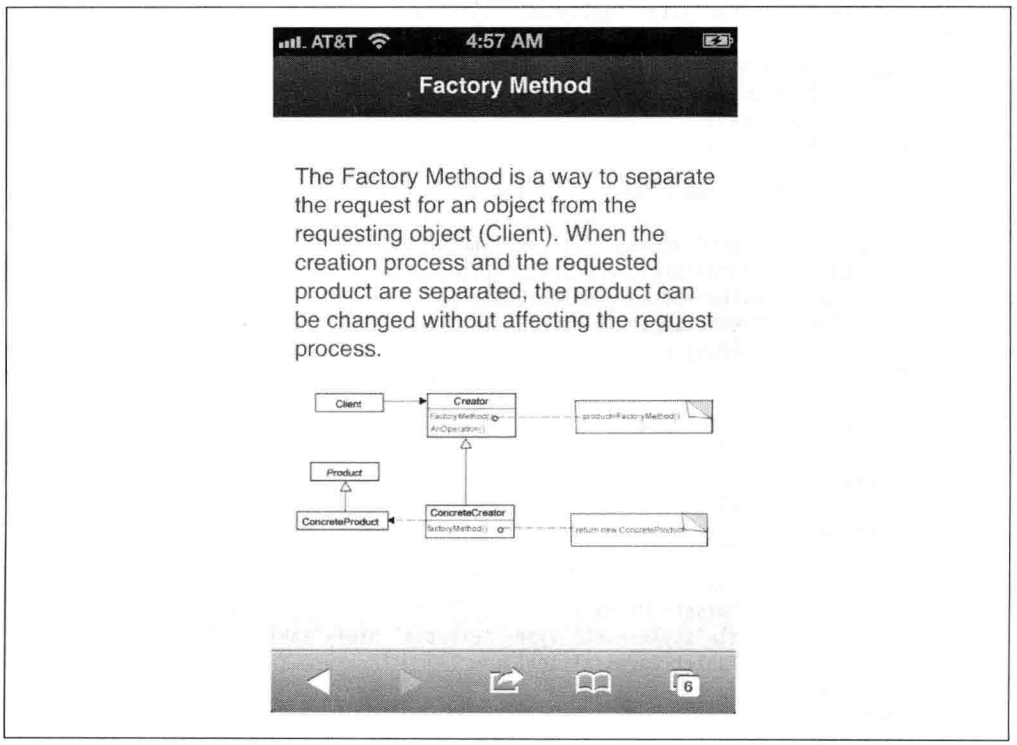


图14-9：通过PHP使用jQuery Mobile格式化的页面

PHP文件中包含一个很有用的工具来指定HTML页面的格式，它使用了heredoc（here document）格式化。基本说来，heredoc容器中的代码接收所有格式，包括双引号和单引号以及PHP变量。它有以下格式：

```
$stringVar = <<<CATCHER
<html>
  <body>
    <h1>Header</h1>
    Text that has dangerous "quotes" and outrageous ideas, even $variables.
  </body>
</html>
CATCHER;
echo $stringVar;
```

heredoc <<<操作符只在容器最前面使用。$stringVar赋值为CATCHER容器中的直接量，以便动态生成包含大量文本或代码的代码。

平板电脑观察者。观察者接口的ConcreteObserverTablet实现与ConcreteObserverPhone类类似，只不过它没有使用jQuery Mobile。它有自己的CSS，采用了两栏布局：

```
<?php
//ConcreteObserverTablet.php
class ConcreteObserverTablet implements Observer
{
    private $currentState=array();
    private $dpHeader;
    private $bodytext;
    private $imageURL;

    public function update(Subject $subject)
    {
        $this->currentState=$subject->getState();
        $this->dpHeader=$this->currentState[0];
        $this->bodytext=$this->currentState[1];
        $this->imageURL=$this->currentState[2];
        $this->doTablet();
    }

    private function doTablet()
    {
//Heredoc语法
$showPage = <<<TABLET
    <!DOCTYPE html>
    <html>
        <head>
            <meta charset="UTF-8">
            <link rel="stylesheet" type="text/css" href="tablet.css">
            <title>Tablet Page</title>
        </head>
    <body>
    <article>
        <header>
```

```
            <h1>$this->dpHeader</h1>
        </header>

        <section>
            <img src="tablet/$this->imageURL" alt="image urls" >
            <p>$this->bodytext</p>

        </section>
        </article>
    </body>
    </html>
//Heredoc文本末尾不能有边距
TABLET;
    echo $showPage;
    }
}
?>
```

图14-10显示了iPad中查看的一个页面。

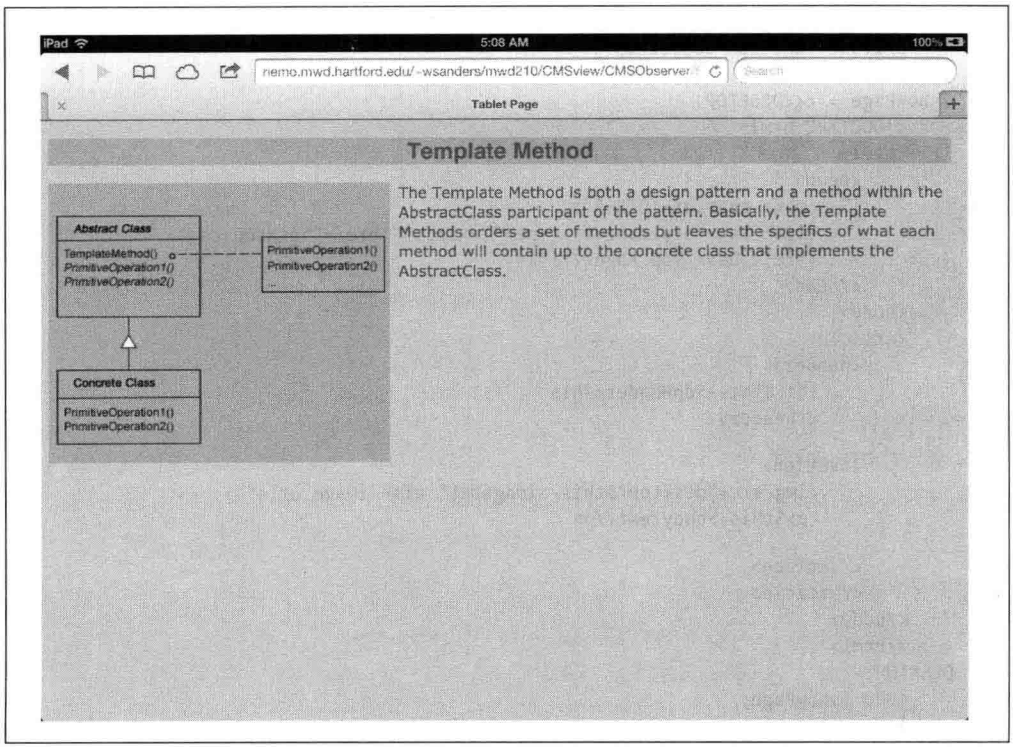


图14-10：主题数据的平板电脑（水平）视图

这两个视图的数据相同，不过使用了不同的CSS源。

桌面计算机视图。原来只能在桌面或笔记本计算机上查看Web页面，不过如今远非如此。下面的ConcreteObserverDT代码与前面两个观察者类似：

```
<?php
//ConcreteObserverDT.php
class ConcreteObserverDT implements Observer
{
    private $currentState=array();
    private $dpHeader;
    private $bodytext;
    private $imageURL;

    public function update(Subject $subject)
    {
        $this->currentState=$subject->getState();
        $this->dpHeader=$this->currentState[0];
        $this->bodytext=$this->currentState[1];
        $this->imageURL=$this->currentState[2];
        $this->doDesktop();
    }

    private function doDesktop()
    {
$showPage = <<<DESKTOP
    <!DOCTYPE html>
    <html>
        <head>
            <meta charset="UTF-8">
            <link rel="stylesheet" type="text/css" href="desktop.css">
            <title>Desk Top Page</title>
        </head>
    <body>
    <article>
        <header>
            <h1>$this->dpHeader</h1>
            </header>

        <section>
            <img src="desktop/$this->imageURL" alt="image urls" >
            <p>$this->bodytext</p>

        </section>
        </article>
    </body>
    </html>
DESKTOP;
    echo $showPage;
    }
}
?>
```

桌面视图与平板电脑视图非常相似，如图14-11所示。

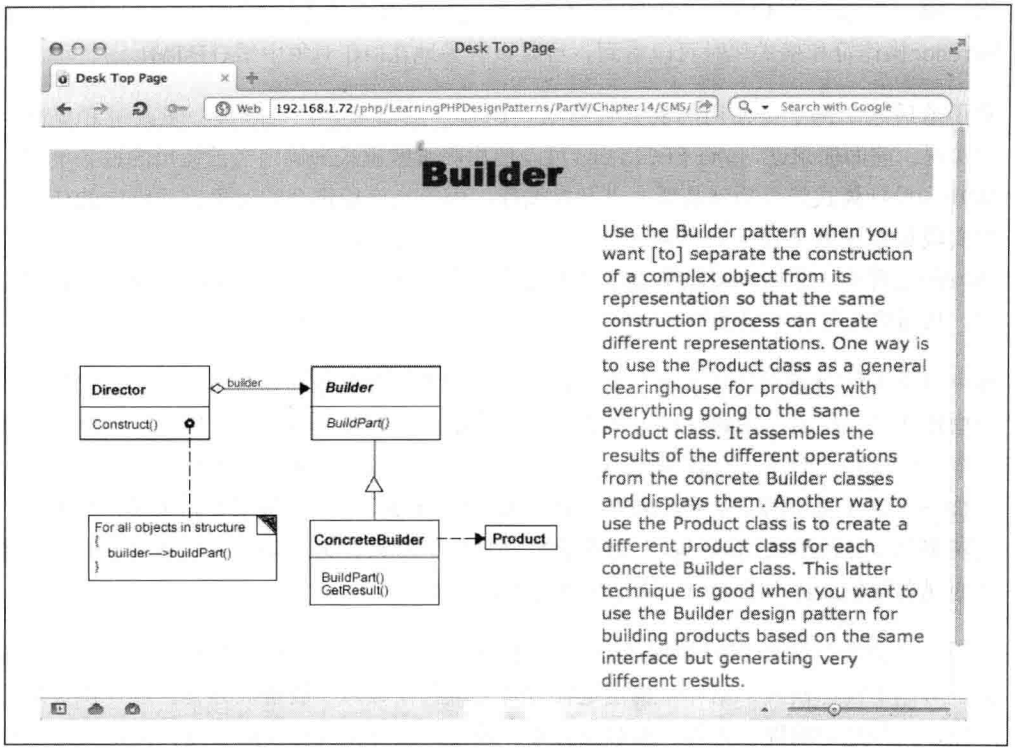


图14-11：桌面视图为图像留有足够的边距

对应3种不同的设备，图像也有所不同。对于桌面计算机设备，图片配置在500像素×500像素框架中，在平板电脑设备上，图片都有一个深褐色背景，手机上显示的图片则设置为适合单栏显示（减少栏间需要的额外空间）。不过，它们都会显示`Subject`提供的同样一组数据。观察者调用同名的不同图像，这些图像存储在名为*mobile*、*tablet*和*desktop*的文件夹（目录）中。

14.6 用OOP方式思考

PHP中观察者设计模式有很多特殊特性，其中一个特性就是专门为观察者模式设计提供了内置标准PHP库（Standard PHP Library，SPL）接口——`SplSubject`和`SplObserver`。总地说来，由这些库接口可以看出PHP社区对设计模式和OOP的明确支持。不过，开发观察者模式时并不要求必须采用所提供的这些库接口；Subject参与者可以是一个抽象类而不是接口（如CMS例子中所示）。

不过，与PHP 5中的很多内置特性类似，有些特性没有用到，或者会在OOP框架之外使

用。同一个文件中HTML与PHP之间的大量交互通常不属于OOP设计，不过前面使用heredoc操作符和格式化时可以看到，可以很容易地在PHP对象中结合HTML。

对于最后这个例子中使用的观察者模式，还有一点很重要，可以进一步改进和提高这个实现。前面提到过，SniffClient可以使用职责链模式来确定当前使用的设备类型。OOP和设计模式的目的就是要改进开发过程。由于允许松耦合的对象完成特定的任务，并实现彼此之间的通信，从这个角度讲，开发过程会快得多，而且更容易重用。学习OOP的过程中，要明确可以把设计模式作为指南，来帮助学习如何组织对象完成相互通信，从而解决各种不同类型的问题。

还可以对这个CMS例子做另一个改进，通过CMS增加新的设计模式时，可以在选择菜单中增加按钮。按照现在的设计，增加新的设计模式很容易，包括一个标题、一个图片和一个文字说明。对应一个新模式，并不会在菜单中生成一个新的单选钮供用户选择。每增加一个新模式，都必须重写菜单。如何为现有的CMS增加模块来更新菜单？另外如何更新菜单文档链接的内容页面？想想看要创建哪些对象，另外如何与现有对象通信或者改变现有对象。基于松耦合，这个任务并不太费劲。

对于一个需要反复操作的编程任务，中级和高级程序员会毫不犹豫地使用一个循环结构。类似于设计模式，循环结构也是针对常见编程任务的一般解决方案。设计模式也是一样，同样是对编程任务的一般解决方案。没错，设计模式更高级一些，而且相对于严格的顺序结构，我们对设计模式的结构也更为熟悉，如循环和函数。一旦你习惯了OOP和设计模式，可能会奇怪自己这么久以来没有OOP和设计模式是如何编程的，就像对于需要反复操作的任务，如果不使用循环来简化编程，简直让人无法想象。

作者介绍

William B. Sanders博士是哈特福德大学多媒体Web设计和开发方向的教授，主要讲授有关PHP、MySQL、C#、SQL、HTML5、CSS和ActionScript 3.0等内容的多门课程，另外还讲授其他一些Internet语言课程。他合著有《ActionScript 3.0 Design Patterns》（O'Reilly, 2007）一书，多年来一直积极地参与PHP设计模式方面的工作。他出版过45本计算机以及与计算机相关的书，使用过多种不同编程软件，从Basic到汇编语言到Flash Media Server都有涉猎。另外他还担任不同计算机软件公司的顾问和beta版测试人员，包括Macromedia和Adobe。他还是一位Apple iOS开发人员。

封面介绍

本书封面上的动物是一条阿拉斯加鲽鱼。阿拉斯加鲽鱼是一种海鱼，主要生活在太平洋东部阿拉斯加湾到北部楚克奇海和西部日本海地区。从海面以下600米到大陆架底部都能看到这种鲽鱼的身影。这种鲽鱼属于一种比目鱼，这说明它的眼睛只长在一侧，另一侧面向海底，是看不到的。这些特殊的比目鱼最长可以长到24英寸（2英尺），在长眼睛的一侧可能有5到7个小骨锥。

这种鲽鱼寿命最长为30年，这要归功于捕鱼业对这种鱼没有太大兴趣。阿拉斯加鲽鱼之所以数量众多，这也是捕鱼业更侧重其他底栖鱼（如黄鳍金枪鱼）的附带结果。