



Professional PHP Design Patterns

# PHP设计模式



(美) Aaron Saray  
梁志敏 蔡建

著  
译



清华大学出版社

# PHP 设计模式

(美) Aaron Saray 著

梁志敏 蔡建 译

清华大学出版社

北京



Aaron Saray

Professional PHP Design Patterns

EISBN: 978-0-470-49670-1

Copyright © 2009 by Wiley Publishing, Inc.

All Rights Reserved. This translation published under license.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字: 01-2009-7477

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

#### 图书在版编目(CIP)数据

PHP 设计模式 / (美) 萨莱 (Saray,A.) 著；梁志敏，蔡建 译.—北京：清华大学出版社，2010.6

书名原文：Professional PHP Design Patterns

ISBN 978-7-302-22843-1

I. P… II. ①萨… ②梁… ③蔡… III. PHP 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2010)第 095093 号

责任编辑：王军于平

装帧设计：孔祥丰

责任校对：胡雁翎

责任印制：李红英

出版发行：清华大学出版社

<http://www.tup.com.cn>

地 址：北京清华大学学研大厦 A 座

邮 编：100084

社 总 机：010-62770175

邮 购：010-62786544

投稿与读者服务：010-62776969,c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015,zhiliang@tup.tsinghua.edu.cn

印 刷 者：清华大学印刷厂

装 订 者：三河市新茂装订有限公司

经 销：全国新华书店

开 本：185×230 印 张：17 字 数：350 千字

版 次：2010 年 6 月第 1 版 印 次：2010 年 6 月第 1 次印刷

印 数：1~4000

定 价：36.00 元

---

产品编号：034809-01



# 作者简介

在 Aaron Saray 8 岁的时候，他接触到一台没有永久存储器的二手 Commodore 64 家用计算机，这使他开始着迷于计算机科学，并且了解了许多不同的语言和计算机。在 2001 年，Aaron 最终决定选择使用 PHP 语言。从那时开始，他坚持学习其他多种 Web 语言(如 HTML、CSS 和 JavaScript)，同时不断充实自己的 PHP 专业知识。在其从业过程中，Aaron 曾经为 Internet Service Provider(ISP)设计和维护过各种 Web 站点工具，为一家大型牙科保险公司的客户创建过基于 Web 的账户管理工具，还为基于 Internet 连接的 Point of Sales 系统开发过后台管理 Web 站点。在成为 Zend 认证工程师后，Aaron 开始应用 Web 开放源代码软件来创办运营自己的 Web 开发公司。在 <http://www.aaronsaray.com> 站点上，他一直都在发布开放源代码软件，并且不断更新以 PHP 内容为主的博客。



# 前 言

PHP 是目前的主流编程技术。我们可以看到大量 PHP 网站以及大量有关 PHP 的工作机会，还可以看到许多大型公司都使用这种开放源代码语言来支持其业务。这种最初毫不起眼的开放源代码语言如今已广泛植根于整个业界之内。PHP 如今已得到了人们的广泛重视，诸如 IBM 和 Microsoft 这样的公司都已支持这种企业级语言。PHP 友好地融合了许多新的观念和思想，其中最值得关注的是通过更健壮的、更灵活的和更经济的部署来开发 PHP 应用程序。与此同时，许多资深的编程人员也在 PHP 中引入了若干重要的概念，本书侧重的就是其中一个主要的概念：设计模式。

## 0.1 本书的适用范围

在确定本书所适用的读者对象时，作者非常难以取舍。是为刚接触 PHP 及其功能和概念的初级编码员而编写，还是为具有多年工作经验的编程人员而编写？是应当为了解 PHP 面向对象功能的专业读者而编写吗？本书是否需要继续包含对 PHP4 的支持？最后一个问题是容易回答：“当然，不再支持 PHP 4。”然而，考虑到 PHP 4 仍然被广泛部署，开发人员仍然在使用 PHP 4 创建新的功能，因此并不能轻易地给出这个答案。为了帮助更广泛的读者使用 PHP 实现设计模式，而不是仅仅作为 PHP 语言参考材料，本书采用了下列指导原则来确定适用的读者。

本书读者对象：

- 必须完全了解 PHP 语言，或者至少收藏过 <http://php.net> 网址。因为某些示例可能使用编程新手之前从未遇到过的函数。
- 必须大致掌握 PHP 中所使用的面向对象编程(Object Oriented Programming, OOP)技术。中级 OOP 编程人员会发现第 2 章中对 PHP 高级 OO 功能的探讨是非常有价值的。
- 必须使用 PHP 5 或更高版本，从而具有可用的面向对象编程功能的完整集合，并且能够执行示例和案例分析代码。
- 应当熟悉统一建模语言(Unified Modeling Language, UML)。

简单来说，对于在构建交互式应用程序方面具有一定经验(至少曾经建立过一个博客)的编程人员来说，本书中使用的示例和概念是极为有价值的。如果以前只使用 PHP 完成过简单的工作(如构建主题模板或联系表单)，那么读者会发现阅读与模式相关的章节是较为困难的。

## 0.2 本书的结构

本书分为 3 大部分：引言章节、参考章节以及案例分析章节。这几个部分具有不同的侧重点。

### 0.2.1 引言章节

第 1 章不仅对设计模式进行了简要的介绍，而且说明了在 PHP 中使用这些设计模式的要求。全世界才华横溢的 PHP 编程人员总是渴望学习新的知识。本章的目的在于：将 PHP 编程人员的视野范围从只基于 PHP 的概念扩展至体系结构更健全的设计模式领域。

第 2 章侧重于介绍一些工具，这些工具在 PHP 中能够用于构建各种设计模式概念的基础。通过回顾 PHP 的中级和高级 OOP 功能、标准 PHP 库以及现有的开放源代码 PHP 架构，本章将帮助读者更深入地理解 PHP 与设计模式。

### 0.2.2 参考章节

参考章节是本书的中间章节，也是设计模式最基本的部分。这些章节可以分为 4 个主要部分：名称、问题与解决方案、UML 图以及一个简单的面向对象的代码示例。上述内容基本上覆盖了设计模式的主要功能部分，同时不至于过分冗长。

### 0.2.3 案例分析

本书的最后一个部分是一个深入的案例分析，包括项目和计划的详细说明、对可用模式的分析以及逐步应用这些模式的方式。

#### 1. 功能分析

通常，当您获得一系列规范说明时，它们并不是最终的版本。在最初查看这些说明时，您应当对具体的体系结构有大致的了解。此时，您会希望了解项目的需求，以便确定它是一个仅仅应用一次的实例，还是一个可扩展的项目。需要在将来实现哪些功能？假如您不是这方面的专家，那么可能需要根据从业务分析中了解到的具体问题来寻求答案。

在研究案例时，您会收到客户提供的规范说明。本书将全面讲述查看规范说明、提出问题和理清思路的整个过程。最后，我们将提供一个已更新的规范说明文档。

## 2. 模式分析

在开发任何项目时，必须首先进入分析阶段。我遇到过太多这样的例子：在项目开始阶段，许多编程人员要么漫不经心，要么得意忘形，从而导致最终完成的项目往往不尽如人意。此时，应该重新认真查看规范说明以确定自己的工作计划。

## 3. 逐步生成代码

案例分析中的这个环节有些偏离本书设定的目标。该部分包含基于 UML 图的详细代码示例。本书将从模式层次分步骤阐明构建应用程序各部分的整个思考过程。然而，我们的重点并非针对语言具体功能的分析。因此，中级编程人员有时可能需要参考 PHP 手册。

生成全部代码后，您应当回顾自己的应用程序以及编码过程中的所有选择，从而确保没有比当前模式更适合的模式。设计模式并不意味着必须坚持严格的规则，但是必须针对具体的应用程序建立构造块和框架。为了创建体系结构更健全的代码库，我们完全可以变换项目中的设计模式。

## 0.3 使用本书的要求

因为本书的一个优点是更注重概念(而非应用)，所以使用本书的要求非常简单，如下所示：

- Windows 或 Linux 操作系统
- PHP 5.2 或更高版本
- MySQL 5.0 或更高版本

如果不满足上述要求，大部分样本代码仍然能够正常运行。然而，运行本书的最后一个案例分析时，必须满足这些要求。

## 0.4 源代码

学习本书中的示例时，读者既可以手动输入所有的代码，也可以使用本书附带的源代码文件。本书使用的所有源代码都可以从 <http://www.wrox.com> 和 <http://www.tupwk.com.cn> 上下载。登录到该站点，使用 Search 工具或使用书名列表就可以找到本书。随后，单击本书细目页面上的 Download Code 链接就可以获得所有源代码。

因为许多图书的书名都很相似，所以通过 ISBN 找到本书是最简单的查找方式。本书的 ISBN 是 978-0-470-49670-1。

下载代码之后，只需用自己喜欢的解压缩软件对它进行解压缩即可。此外，读者也可以进入 <http://www.wrox.com/dynamic/books/download.aspx> 上的 Wrox 代码下载主页，查看本书和其他 Wrox 图书的所有可用代码。

## 0.5 勘误表

请给 [wkservice@vip.163.com](mailto:wkservice@vip.163.com) 发电子邮件，我们就会检查您的反馈信息，如果是正确的，我们将在本书的后续版本中采用。

尽管我们竭尽全力来保证文章或代码中不出现错误，但错误总是难免的。如果在本书中发现了错误(例如拼写错误或代码错误)，请及时告知我们，我们将不胜感激。勘误表可以让其他读者避免被误导，当然，还有助于提供更高质量的信息。

要在网站上找到本书的勘误表，读者可以登录 <http://www.wrox.com> 站点，通过 Search 工具或书名列表查找到本书，然后在本书的细目页面上单击 Book Errata 链接。在这个页面上，读者可以查看到已提交的和 Wrox 编辑已发布的所有勘误项。完整的图书列表还应当包括每本书的勘误表，网址是 <http://www.wrox.com/misc-pages/booklist.shtml>。

如果在 Book Errata 页面没有找到您所发现的错误，那么请进入 <http://www.wrox.com/contact/techsupport.shtml> 页面并填写完相应的表格，从而向我们发送您的意见。我们会阅读您发送的信息，如果是正确的，那么我们将在本书的后续版本中加以采用。

## p2p.wrox.com

为了便于作者和读者之间的讨论，读者可以在 <http://p2p.wrox.com> 站点上加入 P2P 论坛。该论坛是一个基于 Web 的系统，用于发布与 Wrox 图书相关的信息和相关技术，并且支持与其他读者和技术用户的交流。这个论坛提供了订阅功能，一旦论坛上出现新贴，就会向您发送您所选择的主题。Wrox 作者、编辑、其他业界专家以及读者都会在这个论坛上进行讨论。

在 <http://p2p.wrox.com> 站点上有许多不同的论坛，这不仅能够帮助读者阅读本书，而且读者在开发自己的应用程序时也可以从这个论坛中获益。如果想加入这个论坛，那么需要执行下面的步骤：

- (1) 进入 <http://p2p.wrox.com> 站点，单击 Register 链接。
- (2) 阅读使用条款，单击 Agree 按钮。
- (3) 提供加入论坛所需的必要信息与愿意提供的可选信息，单击 Submit 按钮。
- (4) 随后用户会接收到一封电子邮件，其中的信息描述了如何验证账户以及完成加入过程。

**注意：**

不加入 P2P 也可以阅读论坛上的信息，但只有加入论坛后，用户才能够发布自己的信息。

加入论坛后，用户就可以发布新的信息，回复其他用户的贴子。用户可以随时在 Web 上阅读信息。如果希望某个论坛给自己发送新信息，那么可以在论坛列表中单击该论坛对应的 **Subscribe to this Forum** 图标。

要想了解如何使用 Wrox P2P 的更多信息，请确认阅读 P2P FAQ，从而了解论坛软件的工作原理以及许多针对 P2P 和 Wrox 图书的常见问题解答。要阅读 FAQ，可以单击任意 P2P 页面上的 FAQ 链接。



# 目 录

## 第 I 部分 初识设计模式与 PHP

<b>第 1 章 理解设计模式</b>	3
1.1 什么是设计模式	3
1.1.1 一个普通的示例	3
1.1.2 设计模式能够解决相同的问题	4
1.1.3 设计模式无所不在	5
1.1.4 设计模式的公共部分	6
1.2 设计模式未涵盖的内容	7
1.2.1 设计模式并非即插即用	7
1.2.2 设计模式是可维护的，但并非总是最有效的	8
1.2.3 设计模式是重构的必经之路，但不是最终目标	8
1.3 设计模式的相关论证	8
1.4 在 PHP 中使用设计模式的原因	9
1.5 本章小结	9
<b>第 2 章 使用现有的工具</b>	11
2.1 已有架构中的模式	11
2.1.1 PEAR 中的设计模式	11
2.1.2 Zend Framework 中的设计模式	13
2.1.3 Doctrine 中的设计模式	15
2.2 PHP 标准库	15
2.2.1 SPL Observer 与 SPL Subject	15

2.2.2 SPL 迭代器	16
2.3 使用具有模式的 Eclipse PDT	17
2.4 本章小结	22

## 第 II 部分 参 考 内 容

<b>第 3 章 适配器模式</b>	25
3.1 问题与解决方案	25
3.2 UML	26
3.3 代码示例	27
<b>第 4 章 建造者模式</b>	31
4.1 问题与解决方案	31
4.2 UML	33
4.3 代码示例	33
<b>第 5 章 数据访问对象模式</b>	37
5.1 问题与解决方案	37
5.2 UML	38
5.3 代码示例	39
<b>第 6 章 装饰器模式</b>	43
6.1 问题与解决方案	43
6.2 UML	45
6.3 代码示例	45
<b>第 7 章 委托模式</b>	49
7.1 问题与解决方案	49
7.2 UML	50

7.3 代码示例.....	51	14.2 UML .....	97
<b>第 8 章 外观模式 .....</b>	<b>55</b>	14.3 代码示例 .....	97
8.1 问题与解决方案.....	55	<b>第 15 章 代理模式 .....</b>	<b>101</b>
8.2 UML .....	56	15.1 问题与解决方案 .....	101
8.3 代码示例 .....	57	15.2 UML .....	103
<b>第 9 章 工厂模式 .....</b>	<b>61</b>	15.3 代码示例 .....	103
9.1 问题与解决方案.....	61	<b>第 16 章 单元素模式 .....</b>	<b>107</b>
9.2 UML .....	62	16.1 问题与解决方案 .....	107
9.3 代码示例 .....	63	16.2 UML .....	108
<b>第 10 章 解释器模式 .....</b>	<b>67</b>	16.3 代码示例 .....	109
10.1 问题与解决方案.....	67	<b>第 17 章 策略模式 .....</b>	<b>113</b>
10.2 UML .....	68	17.1 问题与解决方案 .....	113
10.3 代码示例 .....	69	17.2 UML .....	115
<b>第 11 章 迭代器模式 .....</b>	<b>73</b>	17.3 代码示例 .....	115
11.1 问题与解决方案.....	73	<b>第 18 章 模板模式 .....</b>	<b>121</b>
11.2 UML .....	75	18.1 问题与解决方案 .....	121
11.3 代码示例 .....	75	18.2 UML .....	123
<b>第 12 章 中介者模式 .....</b>	<b>81</b>	18.3 代码示例 .....	123
12.1 问题与解决方案.....	81	<b>第 19 章 访问者模式 .....</b>	<b>127</b>
12.2 UML .....	82	19.1 问题与解决方案 .....	127
12.3 代码示例 .....	83	19.2 UML .....	129
<b>第 13 章 观察者模式 .....</b>	<b>89</b>	19.3 代码示例 .....	129
13.1 问题与解决方案.....	89	<b>第 III 部分 PHP 设计案例分析</b>	
13.2 UML .....	91		
13.3 代码示例 .....	92	<b>第 20 章 需求分析 .....</b>	<b>135</b>
<b>第 14 章 原型模式 .....</b>	<b>95</b>	20.1 初始需求 .....	136
14.1 问题与解决方案.....	95	20.1.1 执行纲要 .....	136
		20.1.2 作用域 .....	136

## 目 录

20.1.3 假设/限制.....	137	22.2.1 应用程序核心的编程.....	167
20.1.4 详细需求.....	137	22.2.2 用户交互与管理 的编程.....	183
<b>20.2 初始需求分析.....</b>	<b>138</b>	22.2.3 联系方式管理的编程.....	209
20.2.1 大小/用户规模.....	139	<b>22.3 本章小结.....</b>	<b>241</b>
20.2.2 联系方式信息的类型.....	139	<b>第 23 章 使用更多设计模式进行 改进.....</b>	<b>243</b>
20.2.3 应用程序访问.....	140	23.1 处理联系方式的导入.....	243
20.2.4 联系方式同步.....	141	23.1.1 Outlook 联系方式 适配器.....	244
20.2.5 用户证书.....	142	23.1.2 构建联系方式数组.....	247
<b>20.3 最新需求文档.....</b>	<b>142</b>	23.2 去除视图中的逻辑.....	250
20.3.1 执行纲要.....	142	23.3 尽力实现设计模式.....	253
20.3.2 假设/限制.....	143	23.3.1 设计模式和错误检查.....	253
20.3.3 详细需求.....	143	23.3.2 设计模式和联系方式 管理.....	254
<b>20.4 对最新需求的讨论.....</b>	<b>144</b>	23.3.3 设计模式和视图类型.....	254
<b>20.5 本章小结.....</b>	<b>145</b>	23.3.4 设计模式和删除对象.....	254
<b>第 21 章 选择设计模式与计划.....</b>	<b>147</b>	23.3.5 分享您的设计模式完善 工作.....	255
21.1 设计核心.....	147	<b>23.4 本章小结.....</b>	<b>255</b>
<b>21.2 设计用户交互.....</b>	<b>149</b>		
21.2.1 身份验证和授权.....	149		
21.2.2 创建、编辑和删除 用户.....	154		
21.2.3 提供对所有用户的管理 访问.....	155		
<b>21.3 设计联系方式管理.....</b>	<b>157</b>		
21.3.1 处理联系方式与信息.....	157		
21.3.2 联系方式信息关系.....	158		
21.3.3 导入联系方式.....	159		
21.3.4 查看联系方式.....	160		
<b>21.4 本章小结.....</b>	<b>163</b>		
<b>第 22 章 应用程序编程.....</b>	<b>165</b>		
22.1 信息准备.....	165		
22.2 应用程序编程.....	166		

# 第 I 部分 初识设计 模式与 PHP

第 1 章 理解设计模式

第 2 章 使用现有的工具





# 第 1 章

## 理解设计模式

通常，如果拿到一本书并阅读自己不熟悉的主题 5 页以上，就会使读者觉得不安。超过 5 页就可能让读者放弃阅读、情绪烦躁，甚至高声抱怨太困难了！虽然本章的篇幅不止 5 页，但是请读者一定不要放弃阅读。在某种程度上，术语“设计模式(Design Pattern)”只是一个新奇的名称，它并非特别复杂。本章的亮点在于采用了读者可能已知的和经常使用的知识，并且将其提炼为更简明的定义。让我们开始了解到底什么是设计模式。

### 1.1 什么是设计模式

下面给出的 Steve 的故事有助于描述现实生活环境中的设计模式，我希望读者从未听过这个故事。

#### 1.1.1 一个普通的示例

Steve 在一家大型保险公司工作，他最近的工作是开发一个通过 Web 界面将客户信息显示给电话客服中心代表的软件。Steve 设计了一个复杂的系统，这个系统允许电话客服中心代表搜索某个客户、输入电话记录、更新客户保险金额信息以及处理付款。系统的安装非常顺利，几乎没有出现在产品环境中安装新软件时经常遇到的麻烦。Steve 非常高兴，倍感轻松，接下来似乎应当坐下来惬意地品尝一下咖啡了。

很快，保险公司最新的投资规模达到了以前的 3 倍。不仅 Steve 被叫回为电话客服中心软件提供新的可扩展性和增强处理，而且公司急切希望在其站点上为新招揽的客户提供某些新的功能。Steve 所在的部门为此也增加了两名新开发人员：Andy 和 Jason。

公司副总裁提出的要求是：在客户完成成功和安全的登录后，公司的站点应当允许客户自己进行相关支付操作。此外，系统应当显示客户呼入电话客服中心的次数。最后，系统应当显示客户账户对电话客服中心软件进行的所有改动的审计日志。

Steve 知道：简单地更新电话客服中心软件就能够提供审计日志，随后复制并改进代码就可以使用支付处理功能。但是，新加入的编程人员需要尽快投入到新系统的开发工作中。Steve 的老板希望两人接手 Steve 最熟悉的项目部分。考虑到 Steve 是经验最丰富的顶尖 PHP 编程人员，老板需要 Steve 尽快完成公司站点的其他部分，然后再投入到新加入编程人员对电话客服中心软件的审计功能的修改工作中。最后，Steve 还要负责为用户登录界面的新支付处理部分提供挂钩程序。

Steve 先前编写的代码不算太糟糕，但是 Jason 似乎要花费很长的时间才能理解这些代码并完成将支付处理部分移植入公司站点的工作。因此，Jason 决定采用自己的方法来编写代码，以便更快地完成任务。Jason 向 Steve 报告了自己的想法，随即按照这个思路着手开展工作。Andy 同样也忙忙碌碌，因为他刚获得计算机科学专业的硕士学位，所以没有太多经验应对时而混乱的代码。

经过加班加点的工作，这个团队终于成功地完成了对新代码的更改工作。Andy 认为整个开发还可以采用更好的体系结构。Steve 觉得：其他编程人员如果只是复制和粘贴他原先设计的代码，那么开发进度会更快；Jason 和 Andy 只需要进行一些改进，问题就会迎刃而解。Jason 则迫切想解开自己的困惑：为什么某些功能在代码中的不同部分所实现的方式不同。

随着公司 Web 站点的访问者越来越多，系统的性能开始变得越来越差。因此，Steve 的老板建议开发团队花时间对代码进行优化。

Jason 发现他为支付操作编写的方法与 Steve 编写的方法几乎完全相同。因此，Jason 将这些方法组合入一个类中。Steve 则开始查看他为电话客服中心站点编写的身份验证代码与他为公司站点的用户登录编写的类之间的共性。Andy 意识到他们创建的所有 PHP 页面的顶部都具有一组相同的函数调用。因此，他创建了一个引导类型类，以便将所有这些函数调用放在同一个位置，进而减少了代码的重复。

作为局外人，我们可以客观地看到很多问题。Steve 的代码可以按照自己的方式从通用性中受益。Andy 在软件设计方面所受的正规教育使他有时会对 PHP 完成任务的能力以及体系结构提出质疑。Jason 无法轻易地理解 Steve 设计的支付系统，因此他选择自己从头开始创建，从而导致代码的重复。最后，通过对软件的分析，这个开发团队开始在看上去混乱的代码库中发现了各种模式。自此，开发团队开始转向处理设计模式。

### 1.1.2 设计模式能够解决相同的问题

在上面的示例中，Steve 团队的错误实际上与设计模式概念的第一个重要部分有关。在软件开发中，该团队并未明确地创建模式。模式往往是通过实际环境中的实践和应用发现的。将支付应用系统和引导类型调用合并入特定类就属于在编程过程中标识模式的示例。

有人曾经说过，所有的音乐片段都能找到出处。如今，创作新的音乐只是将特定的音

符按照不同的节奏和速度进行重新排列。除了某些实现重大突破的特例之外，通常的软件开发也具有类似的情况。相同的问题会不断地重复出现，并且需要通用的解决方法。因此，对于这些相同的问题来说，设计模式是一种可重用的解决方案。

任何提及设计模式的书籍都离不开著名的“四人组(Gang of Four)”: Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，这 4 个人是最早介绍设计模式的书籍的作者。在自己的领域从事长期工作之后，他们开始注意到不同开发项目中出现的设计的特定模式。这 4 个人将这些想法汇集到一起，从而形成了最早的设计模式概念。通过将特定模式标识为用于未来开发的模板，他们能够将设计模式表达为易于理解的参照基准，以便人们能更好地理解大型和复杂的编程概念。

设计模式涵盖了从接口设计到体系结构的许多方面，甚至包括市场营销和度量。本书主要关注使用面向对象编程方法构造开发语言。

软件设计中的问题由下列 3 部分组成：

- **what:** 要考虑的相关业务和功能需求。
- **how:** 如何使用特定设计满足上述需求。
- **work:** 实际的实现，或者说如何投入应用和实践。

设计模式正好对应该过程的 how 部分，因此本书阐述了如何解决这些问题以及成功解决问题所需的部分实现工作。可以将 PHP 描绘为解决问题所采用的载体。一旦知道了软件需要完成的功能，就能够设计如何完成这些功能，从而通过更少的重构更容易地实现所有功能。

我们已经提到，本书的重点并不是对 PHP 语言的掌握以及对其复杂性的理解，而是在于描述通用的、经过时间检验的一系列方法及其与 PHP 的联系。

从前面的示例中可以看出，模式当然是从软件开发阶段开始的。不过，如果拥有涉及现有模式的完整资料，那么就能够更快地计划体系结构和做出更好的选择。此外，来自不同软件领域的编程人员能够识别该模式，并且只需将其改写为特定的语言类型。应用程序应采用一组清晰的模式，这有助于团队新成员理解项目，从而避免浪费太多时间。

### 1.1.3 设计模式无所不在

我们已经介绍过，Steve 的团队能够理解其软件中的基本模式以及创建可重用项。现在我们也能够进行自己的软件开发。有多少次使用自己的用户类创建相同的用户登录和身份验证系统？是否在希望的位置使用了 db() 函数？这些都是如何使用模式的示例。

在偏爱的 PEAR 或其他架构库中，能够发现更详细的和更接近于基础模式的示例。例如，使用 PEAR DB 是应用设计模式(特别是工厂方法)的示例。Zend Framework 也会使用各种不同的模式(如单元素模式和适配器模式)。

### 1.1.4 设计模式的公共部分

为了描述设计模式，“四人组”首先提出了一个文档标准，并且在著作中将该文档标准应用于所介绍的每种模式。“四人组”之后 PHP 相关书籍的所有作者都复制了这种严格的格式，并且不断地普及该文档形式。因为我觉得这方面的相关介绍非常多，并且只是结构问题，所以本书不会再做过多描述。这里只简单介绍每个模式的文档的 4 个主要部分：名称、问题与解决方案、统一建模语言(Unified Modeling Language, UML)图以及代码示例。

#### 1. 名称

在设计模式中，名称实际上比您最初想象的更重要。对于向项目和其他模式解释特定模式的行为和关系来说，适当的描述性命名约定大有帮助。

在本章的示例中，我们注意到 Jason 向 Steve 提及自己准备重写支付处理系统的一部分代码。作为高级编程人员，Steve 也许不会认可 Jason 采用的方法，但是无疑会建议在重写代码体系机构中使用某些模式。因此，整个团队都必须了解支付系统的基础概念，而 Jason 要负责具体的实现。

#### 2. 问题与解决方案

本章前面曾经介绍过，设计模式是为了使用通用的解决方案来解决相同的问题而出现的。这部分说明覆盖了项目中的主要问题或各种问题，并且表明为何特定的设计模式是其中一个较好的解决方案。

读者也许注意到，本书并没有使用“最佳解决方案”的说法，其原因在于没有人能够给出这样的结论。即使您发现所选择的设计模式最适合于特定的问题，但是为了更理想地将其应用于具体的项目，您仍然需要准备进行一定的改进工作。

#### 3. UML 图

UML 图说明了设计模式的通用结构。在某些情况下，为了说明设计模式的其他实现或者以更易于理解的分段形式阐明复杂的概念，可能需要生成多个 UML 图。

##### 什么是 UML？

统一建模语言(Unified Modeling Language, UML)图应当是编程工具库中的主要组成部分。UML 是对操作、对象和用例进行图形编程的标准方式。在使用 PHP 构建复杂的软件时，UML 图有助于清楚地表达您的设计。要最快地了解 UML 的最新信息，读者可以访问 Wikipedia 站点上的 [http://wikipedia.org/wiki/Unified\\_Modeling\\_Language](http://wikipedia.org/wiki/Unified_Modeling_Language) 页面。

读者可能注意到，我们能够宽松地根据通用模式图给出用于为具体项目生成 UML 图的构建块。当然，与示例相比，方法名、类名和属性是多样化的，并且更加复杂。

#### 4. 代码示例

具有实际经验的 PHP 编程人员非常喜欢设计模式的最后一个主要部分：代码示例。这些相对简单的示例用 PHP 代码表达设计模式的概念。本书介绍基于 PHP 的设计模式，其优点在于不必了解采用其他语言编写的模式示例。侧重于企业级设计模式的其他书籍都是采用由 Java 或 C 语言编写的示例，对于使用单一语言的编程人员来说，示例的效果稍有影响。

本书会不断地重申：代码示例只是例子，它们不能即插即用。代码示例可能不包含错误日志记录或处理、审计或完全安全的编程技术。这并非表明我不欣赏高质量的、安全的编程(之前与我同一团队的编程人员可以证明我是一个注重细节的人)，而是代码示例能够很好地说明我想解释的主要概念。

## 1.2 设计模式未涵盖的内容

通过介绍设计模式并未涵盖的内容来进一步阐明设计模式也十分重要。到目前为止，读者可能已经注意到本书为设计模式给出的定义涉及了相当大的范围。

### 1.2.1 设计模式并非即插即用

如果期望快速地了解设计模式的内容，并且在自己的下一个项目中复制和粘贴本书中的全部示例，那么您会大失所望。对于具体的编程项目来说，设计模式并非简单的“即插即用”式的解决方案。

设计模式并非解决问题的实际实现，甚至不是算法。例如，可以进行这样一个设计：建造的所有房屋都具有朝南的窗户，从而能够采集更多的光线和热量。事实上，并没有进行实际的建筑测量和确定窗户的位置，而只是将设计交给开发商(相当于示例中的编程人员)去具体实现。

另一个说明设计模式的类似方式是将其比喻为一个音阶上的音符。读者可能知道小音阶上的所有音符，但是按次序并以相同节拍演奏这些音符并不能弹出动听的曲子。您不可能在音阶书籍上随便找一个音阶，然后用吉他进行弹奏，同时不可能期望所有人都认为您是一位出色的作曲家。这令人非常烦恼，而且不能解决具体的音乐问题，即通过艺术手段表达特殊的情感。因此，本书将设计模式比喻为音阶。通过不同音阶能够构建一首曲子的很多小段，我们只需要适当应用、调整这些小段就能够创作出动听的曲子。

### 1.2.2 设计模式是可维护的，但并非总是最有效的

在应用程序中，设计模式并不总是提供最佳的效率和速度。设计模式的目的是帮助我们以便于重复和重用的方式设计解决方案。这意味着：设计模式可能不是专门适合于特定的环境，但是却具有更好的代码可维护性和可理解性。

### 1.2.3 设计模式是重构的必经之路，但不是最终目标

我的一位主管最近刚阅读过 Joshua Kerievsky 的一本著作，并且向我谈到最近得到的知识。他说为了使用所有设计模式，我们需要重构代码库。为此，我们讨论了重构的真正意义(特别是针对我们所处的具体环境)。

虽然很尊敬 Kerievsky，而且并非不同意他的观点，但是我确实觉得：在将各种设计模式结合在一起进行重构时，设计模式之间还是应当保持较大的差异。重构方法不仅会创建更为有效的代码库，而且会改善代码的可维护性和清晰性。设计模式是重构方法的重要途径，但不应当是最终目标。虽然我赞成使用一组非常详细的设计模式体系结构规范来启动一个项目，但是并不希望为了应用模式而套用某种模式。如果现实生活中存在的第一个摇滚乐队考虑到其他音乐人在混音中加入钢琴的习惯做法并且照搬一切，那么我们就无法欣赏到经典的、大众深深喜爱的、用吉他弹奏的摇滚音乐。

## 1.3 设计模式的相关论证

为了不使读者由于语言特性的不同而混淆核心概念，过去的大多数设计模式示例都非常简单和偏于理论化。过去学习过设计模式或者面向对象编程的读者会非常熟悉常见的正方形、圆形和椭圆形对象示例。

有关设计模式的书籍在其示例中大量使用简单的对象(如正方形和人)进行讨论。完美主义者认为应当尽可能详细地阐述设计模式的概念和实际应用，并且给出最简单的示例，从而不必分心于设计模式的真正实现。在数学课上，由于额外信息而讨厌应用题的人就属于这种完美主义者。以我的经验，未经过正规培训的 PHP 编程人员情愿看到更详尽的、采用代码形式展现的概念示例。在工作之初，这些编程人员可能通过复制和粘贴代码学到很多知识。

本书介绍的设计模式包含了说明模式的小规模和中等规模 PHP 代码示例。这种两阶段方式将模式的实际概念解释(适用于需要了解特定结构的人)与基于示例的模式论证(适用于更喜欢亲自动手的初学者)组合在一起。

本书的参考章节会使完美主义者感到满意，最后的案例研究部分则会满足偏爱代码示例的读者。要想了解参考章节的更多信息，读者可以阅读第II部分查看具体内容。

## 1.4 在 PHP 中使用设计模式的原因

PHP 支持可供企业使用的引擎，这便于初学者的学习。通过在已有的 HTML 文档中插入若干行代码，我们就可以进入 PHP 的世界。简单地将文件扩展名.html 修改为.php，添加一些代码片段，将文件部署至 PHP 服务器，您就成为了一名真正的 PHP 编程人员了。在 Zend 认证工程师(Zend Certified Engineer, ZCE)认证出现之前，PHP 编程人员不必具有高超的技能。即使成为了 ZCE，PHP 编程人员仍然可能缺乏开发可供企业使用的、体系健全的应用软件所需的某些基本知识。

如果觉得本章开始部分介绍的示例还不够令人鼓舞，那么我们可以告诉您：许多商用级播放器都是使用 PHP 开发的。因为 PHP 较为简陋，所以不像主要的企业级编程语言那样引人注目。然而，Zend 的努力工作以及大型 Internet 公司(例如 Yahoo! 和 Amazon)对 PHP 的广泛采用说明了 PHP 是可供企业使用的。随着企业级软件需求的引入，企业级方法也相应地不断增加。

目前，PHP 已经能支持基于本书所介绍各种概念的大量构建块。在使用 PHP3 或 PHP/FI 的时期，应用这些类型的设计模式可能非常困难或者无法实现。不要曲解我的意思，设计模式是以语言形式存在的，对于某些人来说，本书及本书中的示例可能毫无用处。

## 1.5 本章小结

本章使用了一个普通的编程示例讨论了在日常编程工作中流行的设计模式。通过扩展对模式的理解，读者可以将这个示例与实际的设计模式关联在一起。通过查看设计模式所涉及的领域以及未涵盖的内容，我们为设计模式提供了一个更为简明的定义。最后，本章指出 PHP 支持构建基本设计模式，并且提到 PHP 在某些大型企业中的地位，从而说明了在 PHP 中使用设计模式的原因。

既然已经理解了设计模式，下面就开始介绍对 PHP 使用大有帮助的、目前可用的工具。



# 第 2 章

## 使用现有的工具

第 1 章详细阐述了什么是设计模式，本章将介绍目前的工具集并且查看已使用这些工具的场合。

第 1 章简单地提及了常见的 PHP 架构和库(例如 PEAR 和 Zend Framework)，本章将深入介绍这些架构和库，从而使读者更详细地了解广泛使用的各种设计模式。此外，读者还将接触到一个以前就可能知道的架构。

PHP 5 引入了一个新的、称为 SPL 的类与函数标准集。在进行简单介绍后，本章会详细讨论为参考章节创建设计模式代码示例时有用的功能。

最后，本章还将介绍 Eclipse PDT IDE 中有助于创建和复制设计模式的功能。

### 2.1 已有架构中的模式

最普通的 PHP 架构能够成功，原因在于以下几点：细致的体系结构、可维护性与可扩展性。这也正是在最初体系结构中恰当使用设计模式的全部目标。本节将针对每种架构给出数个模式使用示例，其目的在于证明这些模式在日常编程中越来越普遍，从而使读者不会过分畏惧学习参考章节的内容。如果读者对这些架构中某种设计模式的示例感兴趣，那么可以翻阅相应的参考章节以进一步了解相关的详细信息。

设计模式只是用于构造程序的模板，重申这一观点非常重要。并非每个基于设计模式的体系结构都是某种模式的目标，并且也不会严格遵循该模式的手册规范。读者将会频繁看到：类是使用多个设计模式创建的，或者需将基本模式进行较大调整才能使其适用于特定的环境。

#### 2.1.1 PEAR 中的设计模式

PEAR 是一个较早的 PHP 扩展库。PEAR 是 PHP Extension and Application Repository

(PHP 扩展和应用库)的简写。PEAR Web 站点上 PEAR 快速摘要的表述几乎可以说是设计模式的优秀示例，尤其是这样的措词：“支持可重用的 PHP 组件”。虽然最新版本的 PHP 不再将 PEAR 构建入核心分布，但是我确信您在过去的 PHP 编程工作中接触过 PEAR。PEAR 所起的作用似乎稍落后于其他架构。不过，PEAR 仍然具有某些优秀的功能和以设计模式为中心的体系结构示例。

为了仔细讨论 PEAR 中的设计模式，本部分将回顾各种 PEAR 库，并且指出这些库在体系结构计划期间所实现的设计模式。事实上，数百个 PEAR 类都能够被用于说明基于设计模式的设计。

### 1. PEAR Mail

PEAR Mail 类是为使用各种后端系统发送邮件而创建的一个接口，这些后端系统包括 PHP mail function、sendmail 和 SMTP。

在这个类的 1.2.0b1 版本中，Mail.php 文件的第 49 行左右包含工厂类的一个实例。第 9 章定义了工厂设计模式。读者会在参考章节中看到这一特定模式。令人兴奋的是：如果使用过这个类，那么您在自己的应用程序中就使用过工厂模式！

PEAR Mail 类还包括一个用于测试电子邮件功能的模拟对象。使用一个函数添加“前发送”和“后发送”处理程序，并且在执行发送方法时调用这些处理程序，这样的方式非常接近于观察者设计模式。第 13 章定义了观察者设计模式。需要记住的是，对于体系结构设计来说，模式只是基础蓝本。就像在模拟对象中一样，为了满足具体的需求，您可能需要在实现期间修改模式。

### 2. PEAR MDB2

PEAR MDB2 是一个数据库抽象层，它能够用于 PHP 支持的所有关系数据库管理系统 (relational database management system, RDBMS)。MDB2 的文档骄傲地宣称一个大型的、面向对象的 API 非常适用于饱含设计模式的体系结构。

在回顾 2.5.0b2 版本时，我发现 MDB2 在 mdb2.php 文件的第 377 行左右使用了工厂模式。在创建这个库时，PEAR 编程人员看起来极大地依赖于工厂模式。通过在 mdb2.php 文件的第 484 行左右使用单元素模式，MDB2 还能够防止自身被复制。第 16 章定义了单元素设计模式。

最后，在 iterator.php 文件的第 54 行左右可以发现迭代器模式的一个优秀示例。第 11 章定义了迭代器设计模式。这个示例是迄今为止最详细的设计模式实现，该实现还扩展了标准 PHP 的功能性，本章后面部分将对此加以讨论。

### 3. PEAR DB DataObject

DB DataObject 是在 PEAR 中明显地应用设计模式的一个示例。第 5 章定义了数据访问对象设计模式。在这里，我不再指出 DB DataObjec 类实现该模式的具体位置，而是建议读者查看这个类的整个结构。自始至终，这都是一个以稳定的但并非过分冗长的方式实现设计模式的优秀示例。第 1 章曾经阐明过，基于设计模式的代码几乎没有“即插即用”的。DB DataObjec 类也一样，为了使这个类可用，就要求通过添加编程人员自己的代码来扩展基本的 DataObjec 类。

### 4. PEAR Log

PEAR Log 包是一个抽象的日志架构，它具有用于文件、数据库、系统日志、电子邮件、Firebug 和控制台的挂钩。1.11.3 版本使用了标准的工厂模式(Log.php 文件的第 151 行)和单元素模式(Log.php 文件的第 213 行)。PEAR Log 包还利用观察者模式(Log.php 文件的第 769 行)来处理某些可扩展性。

#### 2.1.2 Zend Framework 中的设计模式

Zend 公司成立于 PHP 发布之后，因 PHP 引擎而被大众所熟知。这个持续成长壮大的公司不仅为 PHP 提供了更专业的服务，而且还提供了其他软件产品，如 Zend Optimizer 和 Zend Studio。正因为所有这些经验，Zend 才能够将最新的 Web 2.0 功能、设计模式与旧产品完美地聚集在一个架构内。

Zend Framework 是一个简单的、面向对象的架构和库，它采用最优方法构建而成。读者肯定能看到某些设计模式被构建在这个架构的体系结构内。在回顾 Zend Framework 时，我打算直接讲述单独的设计模式，并且说明它们所属的该架构的具体模块。在本书中，我们将以 1.6 版本的 Zend Framework 为例。

#### 1. Zend Framework 中的单元素模式

单元素模式是 Zend Framework 中最常见的模式之一。简单地说，单元素模式是一种允许基本对象只具有一个实例的设计模式。

在 Auth.php 文件的第 68 行左右，Auth 模块使用 getInstance( )方法创建单元素实例。Feed 模块使用 setHttpClient( )和 getHttpClient( )这两个函数存储具体连接的单个实例。Layout 模块使用的一个单元素模式只能包含一个 Layout 的一个实例。最后，在 Registry.php 文件的第 49 行左右，Registry 类也使用了自己的 getInstance( )方法。某些作者将 Registry 列为独有的设计模式，但是我不敢苟同，我认为它只是一种语言扩展或组织模式。

## 2. Zend Framework 中的工厂模式

Zend Framework 中最常见的另一种模式是工厂模式。大致说来，工厂模式会以类似的方式创建与不同实例化对象的接口。

Zend Framework 的 Cache 模块具有一个名为 `factory()` 的方法，该方法负责实现这种功能。通过对查看该方法的外部用法与内部包含生成新类的复杂逻辑，这个特定的方法 (`Cache.php` 文件的第 82 行) 很好地说明了工厂模式。Db 模块也使用了一个名为 `factory()` 的方法来创建 Db 连接的实例。数据库抽象层是需要工厂模式的极佳实例，Zend Framework 和 PEAR 都证明了这个问题。Uri 模块是更传统的工厂模式示例，该模块使用更多的控制结构来判断正确实例化和返回的类。`switch` 语句的位置在 `Uri.php` 文件的第 107 行左右。

## 3. Zend Framework 中的迭代器模式

迭代器模式也是在 Zend Framework 中实现的一种重要模式。迭代器模式只是一种使对象可遍历的对象设计。在 PHP 中，我们可以将这种模式想象为将某个对象的属性加入一个数组。

Config 模块被用于存储配置选项，这些选项无疑需要被有计划地访问。某些选项只读的必要性增强了 PHP 不是只使用一个普通数组而是创建一个 `Iterator` 对象的需求。在 PHP 中，通过在创建 Config 类时实现 `Iterator` 接口就可以满足上述需求。Form 模块是迭代器模式的另一个实现，在该模块内，能够进行单步调试 HTML 表单的各种元素。所以分页是在内部对不同的结果进行迭代，所以 Paginator 模块是最容易被选用于迭代的模块之一。

## 4. Zend Framework 中的适配器模式

在 Zend Framework 中，适配器设计模式为架构宣称的功能提供了易于实现的手段。第 3 章定义了适配器设计模式。适配器模式会创建一个接口，该接口会将某个对象的方法改变为另一个对象所期望的事物。

Db 系列文件的一部分在 Zend/Db/Adapter 文件夹内，其中包含为调用数据库(如 Db2、MySQL 和 Oracle)内通用函数而创建一个简单结构的各种类。借助于适配器模式，编程人员不必考虑构建函数的确切 DSN 和 PHP 函数调用就能够调用类似 `connect()` 的方法。InfoCard 模块具有名为 `setAdapter()` 和 `getAdapter()` 的两个函数(分别位于 `InfoCard.php` 文件的第 127 行和第 139 行)，这样允许每个对公共函数的调用在不考虑其内部实际逻辑的情况下都能够完成相同的功能。除了使用迭代器模式之外，Paginator 模块也使用了适配器模式，从而为需要进行单步调试的所有项提供一个统一的接口。该模块的 `factory()` 方法清楚地说明了这个问题。看起来 Paginator 模块中遍布着多种设计模式。

## 5. Zend Framework 中的观察者模式

任何架构都充分利用了插入式类型体系结构，Zend Framework 也不例外。设计这种体系结构的最佳方式是采用观察者模式。观察者模式只是一种在源对象发生某种情况时向其他对象请求潜在动作的方式。

Filter 模块使用 addFilter( )方法(Filter.php 文件的第 51 行)提供为筛选流添加动作的方法。Filter.php 文件的第 65 行运行主函数 filter( )，从而将动作值传递给先前添加的每个对象。Log 模块也利用了观察者模式。Log( )方法将刚发生的所有动作或事件发送给等待对其进行处理的各种对象。运行 Validate 模块的 isValid( )方法时，我们也会应用使用观察者模式创建的功能性。通过位于 Validate.php 文件第 70 行的 addValidator( )函数，我们就可以添加要处理事件的对象。

### 2.1.3 Doctrine 中的设计模式

Doctrine 用于 PHP 的对象关系映射器(Object Relational Mapper, ORM)。这个面向对象的库为数据库抽象层提供了一个接口。Doctrine 也表现为数据访问对象设计模式的一个实现。“表现为一个实现”是保守的说法，实际上整个库集中于数据访问对象模式。

Doctrine 的主要功能性围绕于创建与数据库表相对应的对象的能力。Doctrine 并不通过使用 Select、Join 和 Union 语句创建 SQL 来直接访问数据，而是使用反映预定义关系的属性来创建对象。随后，不用考虑实际的数据库引擎和语言细节，使用简单的对象公共方法就能够执行复杂的数据操作。

## 2.2 PHP 标准库

Marcus Boerger 开发的 PHP 标准库(Standard PHP Library, SPL)是一个有价值的、各种接口和类的集合，并且作为扩展捆绑于 PHP。在 PHP 5 中，SPL 添加为核心发布的一个部分。手册声明这个接口和类的集合“能够解决标准问题”。初看起来，这句话重复了前面为设计模式给出的诠释。实际上，PHP 标准库的丰富特性将有助于用户在 PHP 中创建设计模式。接下来，让我们通过回顾 SPL 的一些主要元素来查看将来在基于模式的体系结构中可以使用的接口和类。

### 2.2.1 SPL Observer 与 SPL Subject

SplObserver 和 SplSubject 接口是观察者设计模式的一个实现。SplSubject 包含 3 个方法声明：attach( )、detach( )和 notify( )。attach( )和 detach( )被用于将对象与通知链相连接

并接收一个 `SplObserver` 对象。只要出现通知链应当知道的任何动作，`notify()` 函数就会被调用。`SplObserver` 接口包含一个 `update()` 方法，该方法只接收一个参数：一个 `SplSubject` 对象。只要这个 `SplSubject` 对象的值发生变化，就应当调用 `update()` 方法。

如果读者不熟悉观察者设计模式，那么可能会觉得非常茫然和困惑。不过，在学习设计模式的过程中，最需要记住的是存在上述方法。读者可能会发现：如果学会使用其中某些方法，那么也许就能够在下一个编程项目中开始应用设计模式。

## 2.2.2 SPL 迭代器

最初遇到 SPL 迭代器时，我并没有将其与我所知道的与设计模式相关的知识关联在一起。即使 SPL 迭代器所起的作用与您对迭代器设计模式期望的作用相同，甚至名称也是相同的，我仍然会完全忽略这种有意而为的体系结构。因为易于使用，所以我们最好在讲述具体的迭代器设计模式之前先回顾一下 SPL 迭代器。回顾这些接口之后，当我们进入第 II 部分的参考章节时，脑海中自然就会浮现出对迭代器的理解。

从根本上说，迭代器是一种对象，它能够为遍历其他一些数据结构或对象提供接口。编程人员并不知道迭代器实际的内在结构。例如，假设进入某个网上音乐零售店并收到同时采用 XML 和 CSV 格式的存货清单，那么您可以创建一个能够通读所有数据源的对象。这个对象的公共方法应当允许前向或后向遍历音乐集合，并且不必考虑接收清单中的数据结构类型。

因为本书第 I 部分的意图是提供能够创建自己的设计模式的构建代码块，所以本部分只讲述 SPL 中更为常见或重要的、使用迭代器的接口和类。要查看 SPL 的完整功能列表，读者可以访问 <http://php.net/spl>。

### 1. Iterator、RecursiveIterator 和 SeekableIterator 接口

SPL 中的 `Iterator` 接口允许大致按照访问数组的方式访问一个对象（但是不要与 `ArrayAccess` 接口相混淆）。`Iterator` 接口的成员函数是 `current()`、`key()`、`next()`、`rewind()` 和 `valid()`。如果熟悉在 PHP 内构建的数组功能性，那么您立刻就会明白如何使用 `Iterator` 接口。

`SeekableIterator` 接口在 `Iterator` 的各种方法中添加了另外一个函数：`seek()`。这个方法允许对象在其存储系统内移动至某个绝对位置。读者可以认为该方法几乎等同于通过在括号内定义索引值来访问数组。

`RecursiveIterator` 接口定义了一种易于遍历分层数据的方式，它包含 `Iterator` 的所有方法与其他两个方法：`hasChildren()` 和 `getChildren()`。简单地说，如果在当前访问层之下还存在其他层，那么 `hasChildren()` 方法会通知用户。`getChildren()` 方法返回一个对象集合，这个集合中的对象包含与父层相同的属性（可选择其他子层）。乍一看，这似乎混淆了抽象

性。不过，我们只需要将其与访问多维数组的编程化方式进行对比就可一目了然。

## 2. DirectoryIterator、LimitIterator 和 SimpleXMLIterator 类

除了 SPL 提供的一些有用接口之外，某些类的集合也能够提供有价值的可靠性。扩展这些类既可以节省时间，也可以巩固具体编程工作中基于模式的设计。

- LimitIterator 类的功能类似于 MySQL 中的 LIMIT 语句。为其提供已有的、具有偏移量和计数的 Iterator 接口时，LimitIterator 会被限制遍历预定义边界内的对象。在创建负责只对集合前 100 项起作用的对象时(即使集合非常大)，您会发现这个类非常有用。应用编程得到的 LimitIterator 结构可以查看具体的边界。
- DirectoryIterator 类提供了一组强有力的方法，从而往复循环文件系统上的某个目录。管理一组文件的通用 Iterator 方法集(如 next() 和 rewind())都只是冰山一角。这个类还集成了诸如 isDot() 和 isLink() 之类的有用函数。此外，被返回的每个文件都是 SplFileInfo 类的一个实例。虽然这里没有介绍 SplFileInfo 类，但是读者可以在 <http://php.net/splfileinfo> 上查找到返回文件系统信息的 25 个方法的集合的完整文档说明。设计模式如何能够提供非常强大的、可扩展的和体系结构健全的对象的基础？DirectoryIterator 就是其中一个优秀示例。
- SimpleXMLIterator 是一个直接使用 PHP 的 SimpleXML 扩展的 Iterator 类。正如根据其名称所预测的那样，除了其他接口之外，SimpleXMLIterator 也实现了 Iterator 接口，并且至少提供了常见的 next()、rewind()、valid() 和 key() 功能。迭代器模式的这种创新性实现是可重用对象的示例，同时也是我们使用模式的主要目标之一。

## 2.3 使用具有模式的 Eclipse PDT

Eclipse PDT 软件包是普通 Eclipse IDE 的一个以 PHP 为中心的发布。除了增加的功能(如 PHP 透视、JavaScript 和 CSS 语法突出显示)之外，Eclipse PDT 是我们的编程工具集中最重要的工具之一。

本章接下来的部分将基于 Windows 中的 Eclipse PDT 2.0。不过，读者也可以将这些步骤用于标准 Eclipse 安装，因为核心 Eclipse 发布具有公共的功能集。如果使用了不同的 IDE，并且从未尝试过 PDT，那么我建议您现在就按照下面的教程进行试验。读者可以从 <http://zend.com/pdt> 上获得最新的 PDT 软件包。如果继续使用不同的 IDE，那么您可能需要修改这些步骤，以便适用于具体的 IDE。您甚至可能希望在自己的技术博客中详细写下这些步骤。

## Eclipse PDT 中的代码段

代码段函数是 Eclipse PDT 中一个有用的功能，这个函数允许用户创建代码模板并将其插入当前文件。代码段的范围从简单的代码注释模板一直到复杂的可重用接口的函数定义集。

例如，我们准备使用 SPL Iterator 接口。在基于模块构建设计模式的库时，您可能会发现需要创建多个代码段。

### 1. 创建 Iterator 接口代码

在实现 Iterator 接口时，具体对象需要包含以下 5 个方法：current()、key()、valid()、next() 和 rewind()。您必须创建这些存根函数。如果您是一位非常优秀的编程人员，那么可能还会在代码中添加注释文档(即使非常分散稀疏)。下面给出了准备使用的代码库样本。

```
/**
 * Get the current element
 * @return mixed
 */
public function current()
{
}

/** 
 * Gets the current key
 * @return mixed
 */
public function key()
{
}

/**
 * Checks if current element exists
 * @return boolean
 */
public function valid()
{
}

/**
 * Moves pointer forward to next element
 */
public function next()
```

```

{
}

/**
 * Moves pointer to first element
 */
public function rewind()
{
}

```

既然得到了要使用的基本代码，接下来就应当研究 Eclipse PDT 的代码段功能性。

## 2. 创建代码段

打开 Eclipse 后，定位当前透视图中的 Snippets 标签。如果没有找到该标签，那么可能需要以下激活操作：

- (1) 选择 Windows | Show View | Other 命令。
- (2) 在 Show View 对话框中(见图 2-1)，展开 General 文件夹并单击 Snippets 选项。

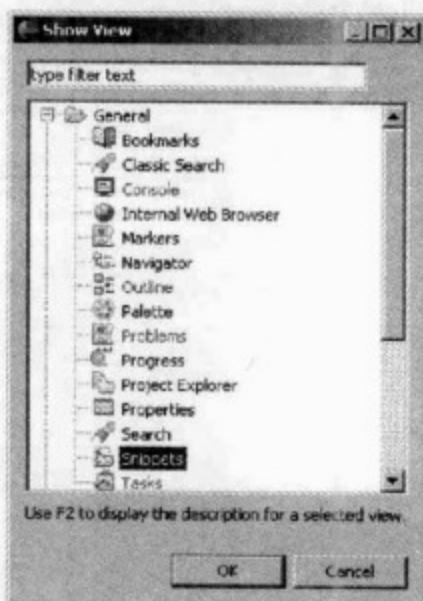


图 2-1

- (3) 单击 OK 按钮。此时在透视图上应当能够看到 Snippets 标签。

因为这是您的第一个代码段，所以接下来必须首先创建一个类别。要创建新的类别，需要执行下面的步骤：

- (1) 右击 Snippet 区域内部，然后选择 Customize 命令。此时会出现 Customize Palette 对话框。
- (2) 单击 Customize Palette 对话框左上部的 New 按钮。

## 第1部分 初识设计模式与 PHP

(3) 单击 New Category 选项(见图 2-2), 此时会出现如图 2-3 所示的 Create Category 对话框。

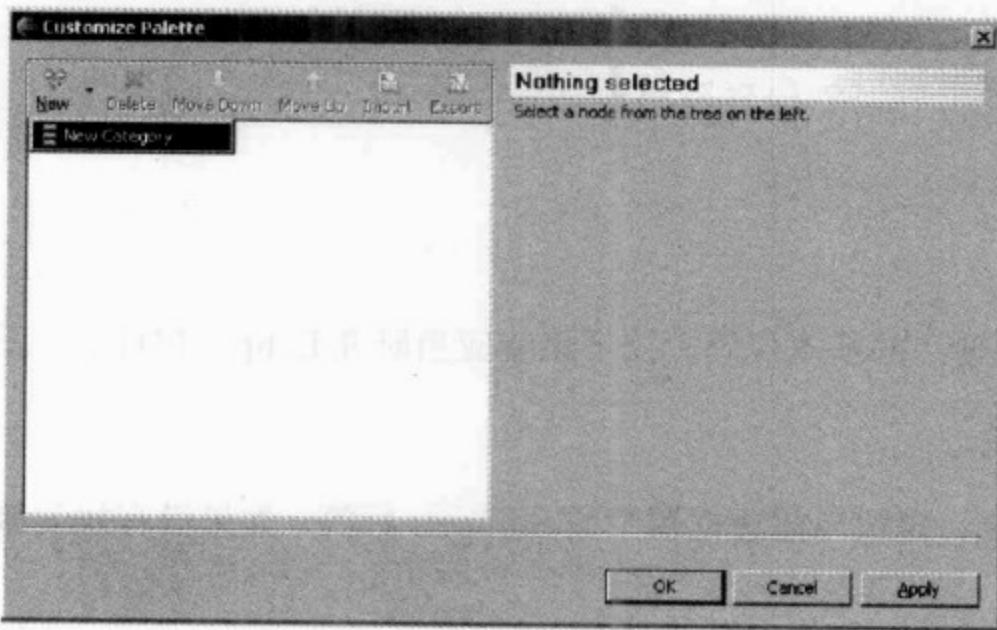


图 2-2

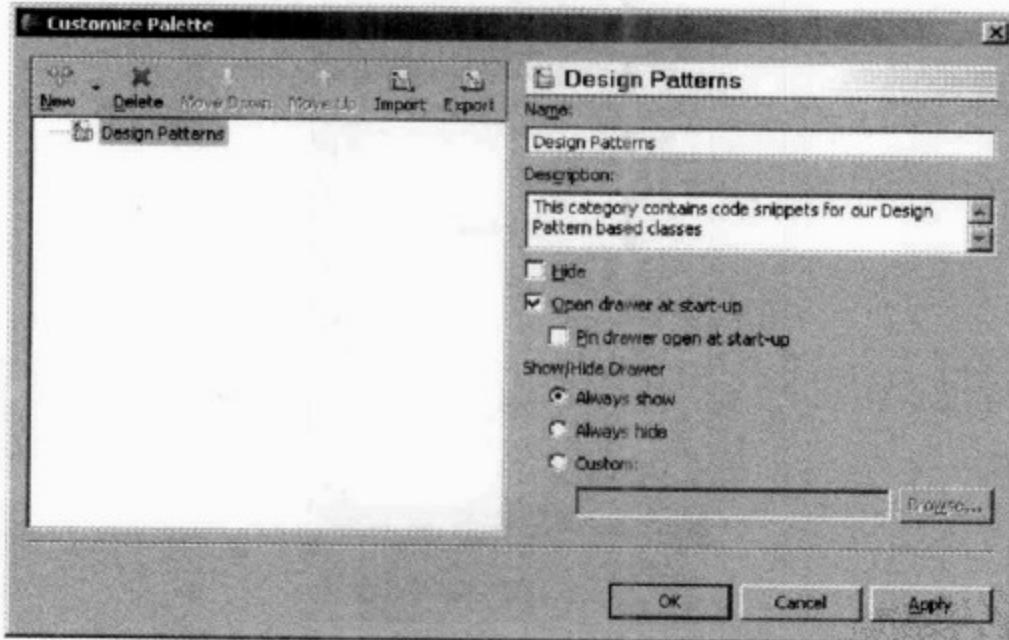


图 2-3

(4) 输入 Category 名称, 如果有必要, 还可以填写相应的说明。

(5) 单击 Apply 按钮。

要创建新模板, 需要执行下列步骤:

- (1) 同样单击 Customize Palette 对话框中的 New 按钮。
- (2) 单击 New Item 选项。
- (3) 输入名称和相应的说明(见图 2-4)。

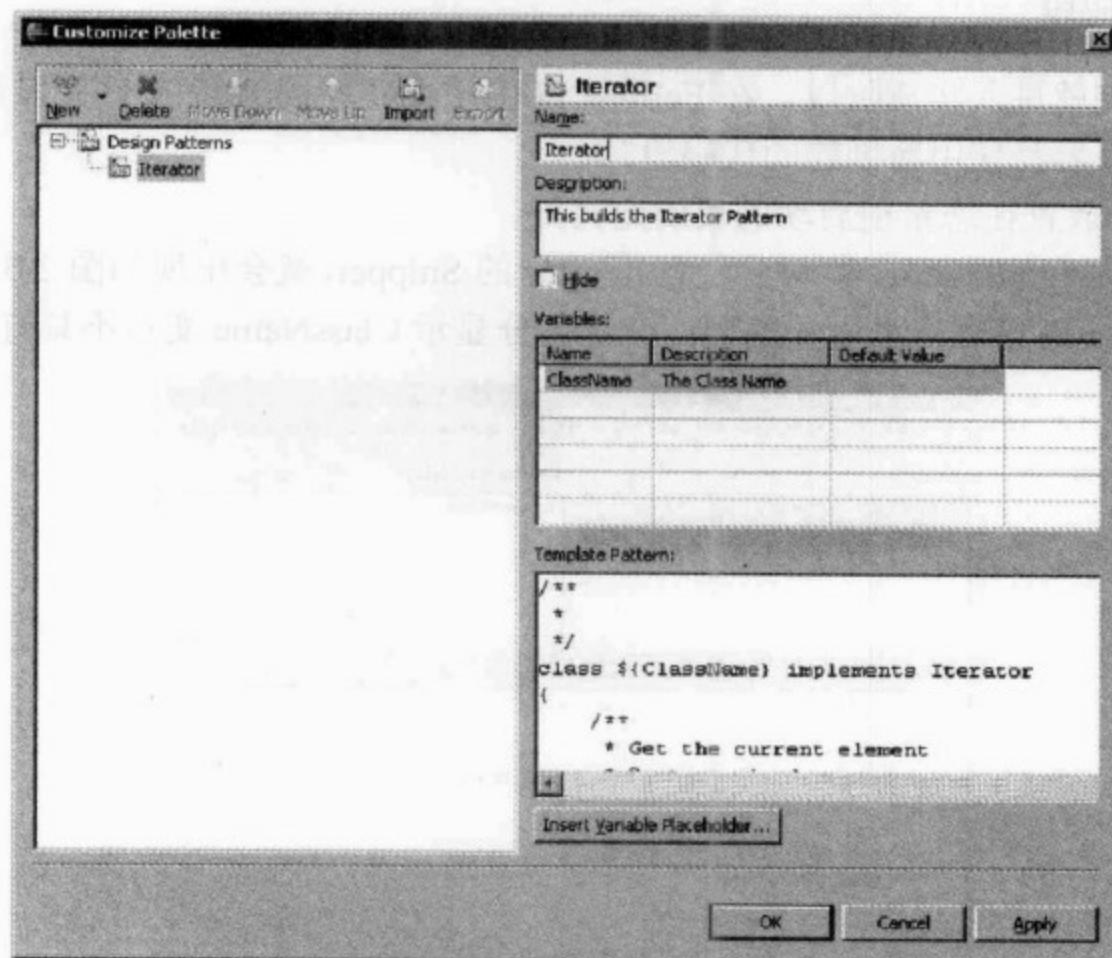


图 2-4

(4) 将前面的代码段粘贴在 Template Pattern 区域内。需要声明这个类，并且确认其可以实现 Iterator 接口。在该示例中，我们并不知道类的名称。不过，代码段的特性决定了允许在代码中插入变量。

(5) 在 Variables 题头下，单击 New 按钮，这样就会在下方的表中插入泛型变量名。

(6) 单击每个题头下方都可以修改相应的值。在我们的示例中，变量命名为 Class Name，相应的说明为 The Class Name，并且默认值保留为空。通过使用\$符号以及将变量名放入花括号内，变量就可以插入模板。

(7) 使用这个类的定义修改模板模式。在模板模式中先前代码段的上方插入下列代码：

```
/**  
 *  
 */  
class ${ClassName} implements Iterator  
{
```

注意：不要忘记在代码段结尾处添加一个右花括号。

(8) 单击 OK 按钮。

### 3. 使用代码段

现在，我们就进入实质阶段。在 Eclipse PDT 中创建新的 PHP 文件并显示 Snippets 标识时，我们就很容易使用新创建的代码段。

- (1) 将光标放置在您希望启动的类的定义处。
- (2) 双击 Design Patterns 类别下名为 Iterator 的 Snippet，就会出现如图 2-5 所示的 Insert Template: Iterator 对话框。可以注意到：变量部分显示 ClassName 变量不具有任何值。

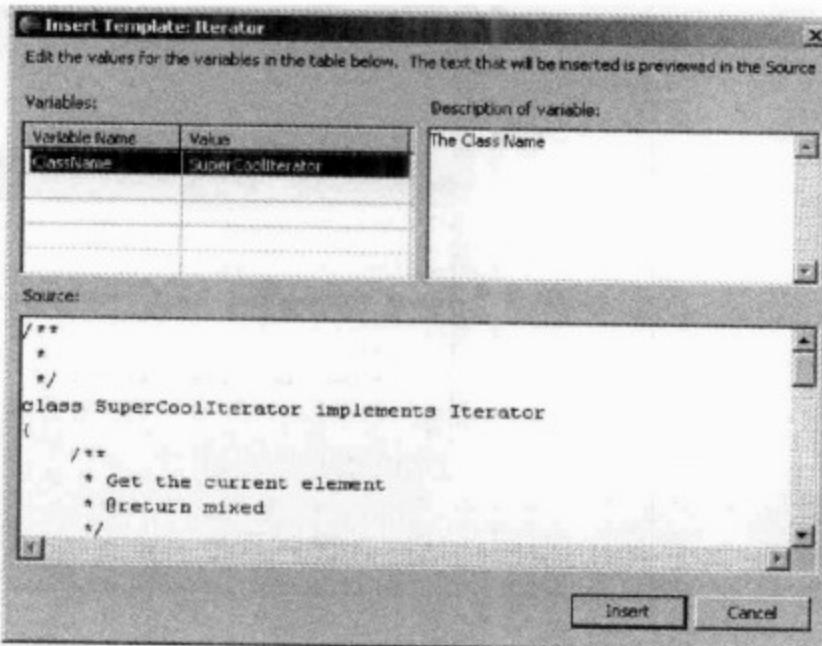


图 2-5

(3) 单击表单元，并且输入新的类名。按下 Enter 键以后，Source 框会采用新类名进行更新。

(4) 单击 Insert 按钮，新模板就被插入源文件内。

## 2.4 本章小结

本章主要介绍在 PHP 和具体 IDE 中已有的一些工具，这些工具有助于启动基于设计模式的编程工作。通过查看不同库中各种类的用法与构造，我们介绍了渗入 PEAR、Zend Framework 和 Doctrine 的设计模式。PHP 标准库(Standard PHP Library)也具有一个非常有用的、接口和类的集合，从而为在新体系结构内应用设计模式提供某些必要的构建块。最后，我们在 Eclipse PDT 中创建了一个代码段，以便减少 Iterator 接口要求重新输入代码的需求，从而更易于继续进行最优编程。

本书接下来的部分是参考章节，该部分将讨论前面提到的一些设计模式方法以及其他一些方法，从而覆盖所有设计模式。

# 第 II 部分 参 考 内 容

- 第 3 章 适配器模式
- 第 4 章 建造者模式
- 第 5 章 数据访问对象模式
- 第 6 章 装饰器模式
- 第 7 章 委托模式
- 第 8 章 外观模式
- 第 9 章 工厂模式
- 第 10 章 解释器模式
- 第 11 章 迭代器模式
- 第 12 章 中介者模式
- 第 13 章 观察者模式
- 第 14 章 原型模式
- 第 15 章 代理模式
- 第 16 章 单元素模式
- 第 17 章 策略模式
- 第 18 章 模板模式
- 第 19 章 访问者模式





# 第3章

## 适配器模式

如果世界是简单的，那么软件需求将会一成不变，大量应用和业务都不会得到革新。编程将会变得简单，不过同时也会令人厌烦。编程人员会遵循数年前的相同技术构建应用程序。他们不再需要引入不同的数据库，也不用实现新的最优方法或者使用不同的 API。然而，所有事物都是变化的。幸运的是，通过适配器设计模式(Adapter Design Pattern)，编程人员能够使用新的代码和功能性来帮助更新原有的系统。

名称：适配器

适配器设计模式只是将某个对象的接口适配为另一个对象所期望的接口。

### 3.1 问题与解决方案

在应用程序中，您也许会使用一个在体系结构上可靠稳定的工作代码库。不过，我们常常会添加新的功能，这些功能要求采用不同的方式使用现有的对象，而不是采用原先设计的方式。此时，障碍可能只是新功能需要一个不同的名字。在较为复杂的场景中，障碍也可能是新功能需要与原始对象稍有不同的行为。

针对上述问题，我们采用的解决方案是使用适配器模式构建另一个对象。这个 Adapter 对象充当了原始应用与新功能之间的中介。适配器模式为已有的对象定义了新的接口，从而能够匹配新对象的要求。

如果采用适配器设计模式，那么在绝大部分情况下，不会丢失现有的功能性，只是使用或消耗的方式不同。读者可以将这种情况视为一个将三相插座转换为两相插座的电源适配器，该适配器透明地发送来自插座管脚的交流电，不过为接地功能提供了不同的接口。在大多数常用的电源适配器中，接地功能并未丢失，不过改为通过与电源插座容器上螺丝钉相连接的接地线来提供这种功能。同样地，适配器设计模式的目标是有助于面向对象的

代码，该模式可以为对象接口创建对话。

虽然可以修改现有代码从而采用新功能所期望的方式运行，但我们最好还是创建一个适配器对象。大家常常认为，要完成这样的任务，快速调整现有对象是最快捷和最划算的方式。我坚决主张下面的观点：在创建适配器对象时，速度和成本基本不会成为争论的问题。实际上，我们不会创建新的功能。到修改原始对象并进行回归测试时，我们可能不必回归就已创建了一个灵活的、具有若干代码行的适配器类。

创建 *Adapter* 对象仍然是最佳的解决方案，这可以提供并行开发新功能和现有代码库的可能性。如果您的工作是集成新功能性，并且要通过编辑基本代码完成该目标，那么就会发现自己与在初始类中开发新功能性的团队之间存在矛盾。开发团队可能添加另外的私有方法，同时期望稳定的最新版本原来所采用的公有方法调用这些私有方法。我们最后想要创建的是复杂的合并场景或叉状代码库。

适配器设计模式也是针对数据源改变的优秀解决方案。下面给出了与数据库引擎改变和数据文件格式改变相关的两个常见问题。

- 因为各种原因，具体项目需要改变数据库引擎。常见的场景涉及将使用 MySQL 创建的应用程序迁移至更大的数据库(如 Oracle)。平时，授权许可限制和成本要求使用不同的引擎，例如产品最终发布时要求使用 Postgres。如果没有使用过数据库抽象层，那么需要创建 *Adapter* 对象，从而截获对原有数据库功能性的调用并使其兼容于新的数据库。非常有趣的是，如果查看某些数据库抽象库的代码，那么会看到它们只是适配器的集合而已。
- 在使用第三方数据进行工作时，被提供的数据文件的格式可能发生变化。供应商可能在过去数年都提供 CSV 格式的数据，但是马上准备迁移至 XML 文档。为此，我们可以创建一个适配器，该适配器用于接收 XML 格式的数据，并且将这些数据以可用的格式交给稳定的 CSV 处理对象。

基本上，只要存在要求主平台持续稳定并且不使现有应用程序流程混乱的问题，在开发解决方案时就可以使用适配器设计模式。

## 3.2 UML

如图 3-1 所示，该 UML 图详细说明了一个使用适配器设计模式的类设计。

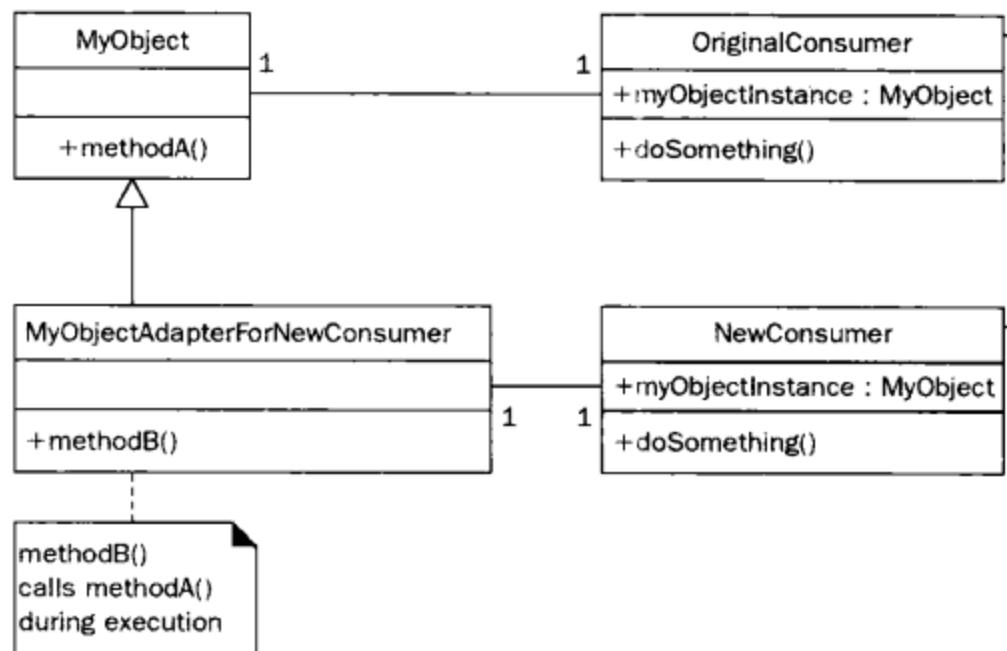


图 3-1

下面是对图 3-1 的说明：

- `MyObject` 类包含一个名为 `methodA()` 的公共方法。`OriginalConsumer` 类获取 `MyObject` 的一个实例，并且在执行其 `doSomthing()` 函数期间调用 `methodA()`。
- 引入 `NewConsumer` 类。这个类的 `doSomthing()` 函数在执行期间期望调用 `MyObject` 实例的一个公共方法：`methodB()`。
- 通过扩展 `MyObject` 类创建 `MyObjectAdapterForNewConsumer` 类。像 `NewConsumer` 期望的一样，新创建的类会提供名为 `methodB()` 的公共方法。在这个简单的示例中，所有 `methodB()` 都会调用 `methodA()`。

### 3.3 代码示例

在项目最初的代码库中，名为 `errorObject` 的对象能够处理所有错误的消息和代码。最初的编程人员并不认为自己的代码会产生任何错误，因此他们设计系统将 `errorObject` 对象的错误消息直接输出至控制台。

下面的示例会生成一个“404: Not Found”错误。我们假定错误消息的内容和代码可能变化，但是文本始终采用相同的格式。

```

class errorObject
{
    private $__error;
  
```

## 第II部分 参考内容

```

public function __construct($error)
{
    $this->_error = $error;
}

public function getError()
{
    return $this->_error;
}

class logToConsole
{
    private $_errorObject;

    public function __construct($errorObject)
    {
        $this->_errorObject = $errorObject;
    }

    public function write()
    {
        fwrite(STDERR,$this->_errorObject->getError());
    }
}

/** create the new 404 error object **/
$error = new errorObject("404:Not Found");

/** write the error to the console **/
$log = new logToConsole($error);
$log->write();

```

在这个场景中，项目中加入了新的网络管理员。建议的最佳做法是安装用于监控软件的网络日志。网络管理员选择的软件包要求将错误记录至一个多列 CSV 文件。具体的 CSV 格式要求第一列是数值错误代码，第二列应当是错误文本。

选择的新软件包精通于 `errorObject` 类，并且软件供应商已提供了实现适当日志记录格式的代码！遗憾的是，这些代码是根据 `errorObject` 的另一版本编写的，该版本与当前项目使用的版本不同。新的 `errorObject` 类具有另外两个名为 `getErrorMessage()` 和 `getErrorText()` 的公共方法，`logToCSV` 类会使用到这两个方法。

```
class logToCSV
```

```

{
    const CSV_LOCATION = 'log.csv';

    private $__errorObject;

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

    public function write()
    {
        $line = $this-> $__errorObject->getErrorMessage();
        $line .= ',';
        $line .= $this-> $__errorObject->getErrorText();
        $line .= "\n";

        file_put_contents(self::CSV_LOCATION, $line, FILE_APPEND);
    }
}

```

针对这个问题，我们可以采用下面两种解决方案：

- 更改现有代码库的 `errorObject` 类。
- 创建一个 `Adapter` 对象。

考虑到保持这些公共接口标准性的需求，因此创建一个 `Adapter` 对象是最佳的解决方案。

新创建的适配器对象中必须存在现有 `errorObject` 的功能性。此外，`getErrorMessage()` 和 `getErrorText()` 公共方法必须有效。在传统的 `logToConsole` 类中，`getError()` 方法用于获取错误消息。适配器对象应当利用该方法从父类中获取错误消息，然后再转化输出供给两个新公共方法使用。

```

class logToCSVAdapter extends errorObject
{
    private $__errorMessage, $__errorText;

    public function __construct($__error)
    {
        parent::__construct($__error);

        $parts = explode(':', $__error->getError());
        $__errorMessage = $parts[0];
        $__errorText = $parts[1];
    }
}

```

## 第II部分 参考内容

```
}

public function getErrorNumber()
{
    return $this->_errorNumber;
}

public function getErrorText()
{
    return $this->_errorText;
}

}
```

最后，为了实现这个适配器，必须通过使用该适配器替代原来的 errorObject 来更新代码。这样，logToCSV 类就能够接收被适配的类(而不是原来的 errorObject 类)，从而使原有的代码能够像 logToCSV 类期望的那样运行。

```
/** create the new 404 error object adapted for csv **/
$error = new logToCSVAdapter("404:Not Found");

/** write the error to the csv file **/
$log = new logToCSV($error);
$log->write();
```

需要记住的是，在需要转化一个对象的接口用于另一个对象时，实现 Adapter 对象不仅是最佳做法，而且也能减少很多麻烦。

# 第 4 章

## 建造者模式

软件复杂性是非常有趣的事情。对软件的需求就像软件包或产品的功能一样复杂。即使是构成软件的代码也同样非常复杂。设计模式方法的重心是提供可维护性、高强度的结构和较低的复杂性。因为具有构成大多数软件储存库的许多对象，所以涉及建造者模式的解决方案能够符合上述要求。

名称：建造者

建造者设计模式定义了处理其他对象的复杂构建的对象设计。

### 4.1 问题与解决方案

从技术上讲，一个对象在被实例化之后就是完整的对象。某些对象可以在这个范围内使用，并且随时可以加入代码流。不过，另外一些对象的性质较为复杂。这些对象可能需要执行其他公共方法才能被视为“完整的”，并且只适用于部分应用程序。

在这里，理解术语“复杂(complex)”的含义十分重要。通常，复杂性是某个类的方法内包含的实际逻辑。但是，当我们提及一个与对象实例化相联系的复杂对象时，复杂性意味着创建这个完整对象所需的步骤等级。每个执行方法中的实际逻辑步骤与这些类是否复杂无关。

创建复杂对象时，构造函数的创建会涉及通用体系结构判定。某些编程人员认为任何构造函数都应当执行适当的逻辑以创建整个对象。另外一些编程人员则认识到将某些逻辑分解入其他方法才是有意义的。采用后一种样式设计的构造函数基本上是一系列请求实例化的方法。上述两种解决方案都不是特别灵活。事实上，它们根本就是错误的解决方案。

我们可以基于一组业务逻辑的结果来构造对象。在本章的具体示例中，基本业务判定规则已完成了编写和测试工作。根据判定结果，对象只有特定的部分必须被创建。实际上，

## 第II部分 参考内容

如果完全定义对象的所有部分，那么可能导致完全预见到的结果。我知道这个说法非常模糊，那么接下来让我们查看一个更好的示例。

在 ABC 公司内，小配件在定价和入库之前就会被载入库存系统。无论是系统首次载入小配件还是生成显示小配件及其说明和价格的 HTML 页面，都会创建相同的小配件对象。因为产品 ID 值为 NULL 的小配件不能被销售，所以 HTML 页面进程不会显示这些小配件，它们也不会进入库存系统。已有库存和新库存的请求在代码中都按照相同的路径处理。当数据到达小配件对象的时候，HTML 页面进程已知道小配件数据是用于新库存项还是用于已有库存项。既然已经了解了 HTML 页面进程的相关功能，那么就必须决定如何创建小配件对象。此外，您不会希望在用于销售的 Web 站点上意外显示任何新的库存小配件。下面列出了各种解决方案：

- 错误的解决方案 1：复制小配件对象构造函数内部的业务逻辑。此刻，这个解决方案只需使用简单的条件语句判断数据的类型，不过却是错误的解决方案。随着时间的推移，判断某个小配件是全新的还是用于销售的业务逻辑可能会发生变化。如果引入第三种状态(例如“入库”)但未定价会怎样？这就要求采用其他代码和对象构造函数来改变进行判定的业务逻辑。
- 错误的解决方案 2：在创建小配件对象时通过调用构造函数中的所有方法来设置所有默认值。这些方法可能包括设置价格、说明和产品 ID 的方法。然而，这个解决方案同样具有潜在的问题。我们知道，HTML 页面进程会查找值不为 NULL 的产品 ID。如果为产品 ID 指派了默认值，那么 HTML 页面进程就会找出这些产品 ID。此外，采用这种方式调用除了产品 ID 设置器之外的所有方法都是非常诱人的。此时，我们只需在相关方法中置入简单的条件语句，并且确保不在创建新小配件时指派属性。不过，这个解决方案也是不适当的。
- 最佳解决方案：基于建造者设计模式创建一个新的对象。这个对象负责解释来自业务逻辑的结果，并且只调用构建完整小配件所需的功能性。即使必要的信息或业务规则发生变化，主代码流仍然采用相同的方式处理 Builder 对象。我们只需要修改 Builder 对象(只在一个位置修改)。这个解决方案不仅节省了时间，而且降低了主代码库的复杂性。

通过查看与第三方应用程序(常见的有数据库包装器)的交互也可以证明建造者设计模式的优点。在具体示例中，如果使用 1.0 版本的数据库包装器类，那么可能需要调用 setUsername()、setPassword() 和 setHostName() 方法得到特定对象的一个完整实例。每当创建与数据库的连接时，相应代码都会包含包装器的实例化，之后则是 3 个函数调用。

在 1.1 版本中，第三方强制使用 setDatabase() 方法。这意味着，代码库中数据库包装器实例化的每个实例现在都需要更改为采用另一个方法调用。

使用上述复杂的创建步骤实现包装器时，最佳的解决方案是创建一个数据库包装器 **Builder** 类。这样一来，在发布 1.1 版本时，就只需要修改 **Builder** 类。

需要记住的是，多个方法调用的复杂性问题在使用之初似乎并非太严重，但该复杂性却是缓慢增长的。如果需要经常调用这些方法，那么就应当创建一个 **Builder** 对象。

## 4.2 UML

如图 4-1 所示，该 UML 图详细说明了一个使用建造者设计模式的类设计。

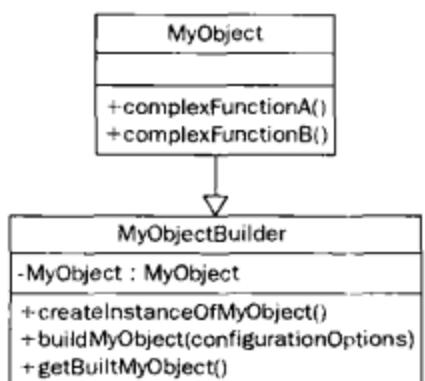


图 4-1

下面是对图 4-1 的说明：

- **MyObject** 类具有能够完全实现对象构造的两个方法。为了具有完整的 **MyObject** 对象，需要执行 `complexFunctionA()` 和 `complexFunctionB()` 方法。
- **MyObjectBuilder** 类包含一个名为 `createInstanceOfMyObject()` 的方法。这个类负责创建 **MyObject** 类的一个简单实例。需要注意没有用于进一步构造的配置选项。这个类还存储 **MyObjectBuilder** 类创建的实例中的私有实例。
- `buildMyObject()` 方法接受参数 `configurationOptions`。这个方法用于调用在 **MyObjectBuilder** 对象中存储的 **MyObject** 对象的 `complexFunctionA()` 和 `complexFunctionB()` 方法。
- `getBuildMyObject()` 方法返回 **MyObjectBuilder** 对象内部 **MyObject** 对象的私有实例，该实例既是完整的，也是正确构建的。

## 4.3 代码示例

具体项目包含一个创建复杂产品对象的类。这个类包含完整构成对象的 3 个方法。如果在创建新产品对象时没有调用其中任何一个方法，那么类的属性便会丢失，并且程序也将中断。这 3 个方法是 `setType()`、`setColor()` 和 `setSize()`。如下面的代码所示，最早版本

## 第II部分 参考内容

的代码被设计为先创建产品对象，然后再执行上述3个方法。

```
class product
{
    protected $_type = '';
    protected $_size = '';
    protected $_color = '';

    public function setType($type)
    {
        $this->_type = $type;
    }

    public function setSize($size)
    {
        $this->_size = $size;
    }

    public function setColor($color)
    {
        $this->_color = $color;
    }
}
```

为了创建完整的产品对象，需要将产品配置分别传递给产品类的每个方法：

```
// our product configuration received from other functionality
$productConfigs = array('type'=>'shirt', 'size'=>'XL', 'color'=>'red');

$product = new product();
$product->setType($productConfigs['type']);
$product->setSize($productConfigs['size']);
$product->setColor($productConfigs['color']);
```

创建对象时分别调用每个方法并不是最佳的做法。此时，我们最好使用基于建造者设计模式的对象来创建这个产品实例。

`productBuilder` 类被设计为接受构建 `product` 对象所需的这些配置选项。它不仅存储配置参数，而且存储一个实例化的新 `product` 实例。`build()` 方法负责调用 `product` 类中的所有方法，从而构建完整的 `product` 对象。最后，`getProduct()` 方法返回完整构建的 `product` 对象。

```
class productBuilder
{
```

```
protected $_product = NULL;
protected $_configs = array();

public function __construct($configs)
{
    $this->_product = new product();
    $this->_xml = $configs;
}

public function build()
{
    $this->_product->setSize($configs['size']);
    $this->_product->setType($configs['type']);
    $this->_product->setColor($configs['color']);
}

public function getProduct()
{
    return $this->_product;
}
}
```

需要注意的是，`build()`方法隐藏了来自请求新 `product` 对象的代码的实际方法调用。如果 `product` 类以后发生改变，那么只需要修改 `productBuilder` 类的 `build()` 方法。下面的代码说明了如何使用 `productBuilder` 类创建 `product` 对象：

```
$builder = new productBuilder($productConfigs);
$builder->build();
$product = $builder->getProduct();
```

建造者设计模式的目的是消除其他对象的复杂创建过程。使用建造者设计模式不仅是最佳的做法，而且在某个对象的构造和配置方法改变时可以尽可能地减少重复更改代码。





# 第 5 章

## 数据访问对象模式

最简单的 Web 部件和最复杂的在线电子商务 Web 站点具有一个共同点：它们都要处理数据。大量的编程都涉及数据的访问和操作。随着 Internet 的发展、廉价存储设备应用数量的大量增长、对分析论的更深入理解以及对信息访问的更大期待，数据以更加有趣和独特的方式受到影响。数据访问对象设计模式的目的是帮助构造出能够容易地(透明地)处理所有这些数据的对象。

名称：数据访问对象

数据访问对象设计模式描述了如何创建提供透明访问任何数据源的对象。

### 5.1 问题与解决方案

对于同时学习过 PHP 和 MySQL 的人来说，数据访问对象设计模式是一个全新的和令人兴奋的概念。这种设计模式的目的是解决下列两种特定的问题：重复与数据源抽象化。

编程通常是大量重复。在更通用的架构被发布之前，情况尤为如此。绝大多数 PHP 编程人员都编写过数十个 CRUD(create[创建]、read[读取]、update[更新]和 delete[删除]的简写)应用程序。在标准的创建/更新应用程序中，重复的一个主要部分是数据源操作。在随后的讨论中，我将停止通用化数据源以及将其称为 SQL。

在应用程序中，必须编写一条 SQL 语句才能在数据库内创建实体。接下来，为了提供对实体任何特性的更新，还需要编写另外一条 SQL 语句。创建这些 SQL 语句所涉及的重复操作不仅令人厌烦，而且也不是最佳的做法。

事实上，我们应当创建一个基于数据访问对象设计模式的对象。这个数据访问对象(Data Access Object, DAO)封装了创建 SQL 调用、减少实体创建的复杂性和重复以及更新进程的智能方式，其编写方式应当是：该对象的使用者不会知道实际使用的表结构或数据库引擎。此外，这个对象引用的方法应当使用逻辑参数，并且应处理 SQL 语句的创建工作。

数据访问对象模式的额外优点是提供数据库抽象层。现在，应用程序的主要处理代码不再需要知道数据库引擎或表关系。调用这种对象的公共方法会返回任何数据类型，并且不用考虑内在 SQL 所需的类型。

使用将非规范化表与另一个表相连接以提供特定结果集的关系数据库结构能够很好地说明这个问题。如果数据库管理员将非规范化表结构修改为完全规范化的，那么遍布应用程序中所有模块的每条 SQL 语句都需要被修改为添加额外的连接表。如果使用数据访问对象，那么就只需要编辑提供该信息的方法。再来看一下实际表结构发生变化的情况：对一个列的命名可能发生变化，或者可能添加额外的列。同样地，需要编辑代码的仍然是数据访问对象。SQL 纯粹主义者会坚决主张添加的表列应当完全不影响查询，他们认为应当使用 SQL 语句中的指定列。我同意上述观点。然而，如果列名发生变化，这种做法将毫无用处。

如果编程人员没有采用数据访问对象设计模式，那么总是会出现问题。将这些对象类型的全部能力与易于使用相对比，无疑添加更多功能的诱惑占了统治地位。不过，我更倾向于数据访问对象中的简单性。千万不要添加未经证明或不需要的额外功能。

管理数据访问对象类中简单性的一个好方法是创建父-子关系。首先，创建一个基本的父对象。这个对象应当负责数据库连接、抽象地执行查询以及与子对象通信。使用数据访问对象设计模式时，最好开始就将一对关系的子类与数据库中的表相关联。这些子类具有必不可少的信息，如表名和主键。此外，子类可能包含一些特定的公共方法，这些方法通过只对子类有意义的方式来执行父类的查询。例如，名为 userAddress 的子类可能包含一个 getAddressesByZip( )方法。将该方法放入父 DAO 类是毫无逻辑意义的，并且会破坏这个父类希望实现的抽象性。

处理引用特定数据库信息的实体时，最好的做法是创建一个数据访问对象。

## 5.2 UML

如图 5-1 所示，该 UML 图详细说明了一个使用数据访问对象设计模式的类设计，进一步的阐述如下所示。

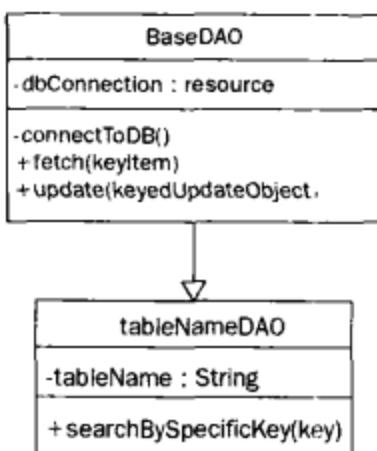


图 5-1

- BaseDAO 类是 tableNameDAO 类扩展的一个抽象类，它具有连接数据源的私有方法 `connectToDB()`。这个方法存储私有实例变量 `dbConnection` 中的连接。BaseDAO 类还包含两个公共方法：`fetch()` 和 `update()`。
- `fetch()` 方法期望接收名为 `keyItem` 的参数，该参数引用希望返回的数据源的主标识符。这个方法会执行正常的数据库调用并返回结果集。
- `update()` 方法期望接收名为 `keyedUpdateObject` 的参数，该参数是一个对象或数组，包含对数据库进行更新的键和值。在这个函数中，列和值被抽取出来并应用更新。
- tableNameDAO 类直接关联数据库中的表。`tableName` 变量存储实际的表名，这个私有变量被用于创建 `fetch()` 和 `update()` 方法中的数据库调用。作为数据访问所具有的、不同于基对象的额外功能示例，`searchBySpecificKey()` 函数也出现在 UML 图中。这个方法期望接收变量 `key`，它会创建对父数据访问对象的适当数据库调用组合，从而获取指定的返回样式。

### 5.3 代码示例

这个示例关注的是一个用户实体。MySQL 数据库拥有一条记录，该记录包含对每个用户来说都是具体和特有的信息。这种功能性必须允许我们通过用户的主键或对用户名的查找返回一个用户。此外，我们必须能够对用户实体记录的任意字段执行更新操作。

考虑到上述需求，我们需要使用两个类。第一个类应当是基本数据访问对象类，它具有获取和更新数据的方法，如下所示：

```

abstract class baseDAO
{
    private $connection;

    public function __construct()

```

## 第II部分 参考内容

```

{
    $this->__connectToDB(DB_USER, DB_PASS, DB_HOST, DB_DATABASE);
}

private function __connectToDB($user, $pass, $host, $database)
{
    $this->__connection = mysql_connect($host, $user, $pass);
    mysql_select_db($database, $this->__connection);
}

public function fetch($value, $key = NULL)
{
    if (is_null($key)) {
        $key = $this->primaryKey;
    }

    $sql = "select * from {$this->tableName} where {$key}='{$value}' ";
    $results = mysql_query($sql, $this->__connection);

    $rows = array();
    while ($result = mysql_fetch_array($results)) {
        $rows[] = $result;
    }

    return $rows;
}

public function update($keyedArray)
{
    $sql = "update {$this->tableName} set ";

    $updates = array();
    foreach ($keyedArray as $column=>$value) {
        $updates[] = "{$column}='{$value}' ";
    }

    $sql .= implode(',', $updates);
    $sql .= "where {$this->primaryKey}='{$keyedArray[$this->primaryKey]}'";

    mysql_query($sql, $this->__connection);
}
}

```

需要注意的第一点是：这个类是一个抽象类。显而易见，这意味着为了能够使用该类首先必须扩展该类。在实例化时，私有方法 `_connectToDB()` 借助正确的证书才能执行。该方法简单地存储了对象内部的数据库连接。只要执行新的查询，就会引用被存储的数据库连接。因为很可能同时打开多个数据库连接，所以在数据访问对象类中存储内部的数据库连接并且每个查询都进行引用是十分重要的。这个数据访问对象类应当唯一地引用其自己的连接。通常，在更多可扩展的模型中，接口被创建用于共享连接。

下一个方法是 `fetch()` 公共方法，它将接受一个必要的参数和一个可选的参数。`select` 语句规范中的 MySQL 查询会使用必要的 `$value` 参数；可选的 `$key` 参数默认是表的主键。不过，如果设置了 `$key` 参数，那么就会执行更灵活的查询，并且可能返回更多的结果。最后，该方法会创建一个结果数组、将结果填入该数组并返回该数组。一定要注意到 `fetch()` 方法的抽象程度：该方法事先并不知道被查询的表名、键或值。这也正是数据访问对象为代码提供的能力。

基本数据访问对象类中的最后一个方法是 `update()` 公共方法。同样地，因为它构建的查询的抽象性，所以其构造也会令人感到有趣。为了成功地更新表记录，这个特定的方法期望键控数组将实体的主键作为数组的元素。

任何子类都可以扩展这个抽象类。通过指向 `userTable` MySQL 表，示例给出的类能够引用用户实体。此外，我们还需要更多只在用户实体具体环境中有意义的特定功能性。

```
class userDAO extends baseDAO
{
    protected $_tableName = 'userTable';
    protected $_primaryKey = 'id';

    public function getUserByFirstName($name)
    {
        $result = $this->fetch($name, 'firstName');
        return $result;
    }
}
```

因为上面给出的类扩展了 `baseDAO` 对象，所以能够访问父类的所有函数。子类定义了表名和主键，它们直接关联数据库中的一个 MySQL 表。为了获得一个起作用的数据访问对象子实体，至少需要定义这两个受保护变量。不过，功能性需求的一部分是通过名称能够查找用户表。为了实现这样的需求，公共方法 `getUserByFirstName()` 会接受一个名称参数。通过调用父类 `fetch()` 方法和定义应当查询的列，就可以得到最后的结果。

## 第II部分 参考内容

下面给出了要使用的数据访问对象的示例。

```
define('DB_USER', 'user');
define('DB_PASS', 'pass');
define('DB_HOST', 'localhost');
define('DB_DATABASE', 'test');

$user = new userDAO();
$userDetailsArray = $user->fetch(1);

$updates=array('id'=>1, 'firstName'=>'aaron');
$user->update($updates);

$allAarons = $user->getUserByFirstName('aaron');
```

代码的第一部分是定义数据库证书。在产品系统中，显然存在更安全和更灵活的数据证书提供方式。新的 userDAO 会被创建。请求信息的第一位(bit)就是第一个用户。此时，用户会得到一个数组，该数组的所有细节都来自主键为 1 的用户实体。接下来的代码定义了更新操作。id 为 1 的用户实体的名称会被更新为 aaron。最后，数组由包含名称 Aaron 的所有用户构建而成。

为了减少重复和抽象化数据，最好的做法是基于数据访问对象创建一个类。

# 第 6 章

## 装饰器模式

编程人员最害怕听到的一句话是：“这是一个不断变化的需求文档。”客户指定在需求收集和规范创建阶段自始至终都需要进行持续开发。即使在这两个阶段完成之后，客户还可能返回上述阶段并要求在很多地方进行一些调整。因为这些修改对于客户来说似乎非常细小，所以他们并不期望在部署时再进行优化处理。如果没有进行正确的处理，那么即使是很小的调整也会让人头疼。如果要对基本功能稍加改动，那么最好选择使用装饰器设计模式。

### 名称：装饰器

如果已有对象的部分内容或功能性发生改变，但是不需要修改原始对象的结构，那么使用装饰器设计模式最适合。

## 6.1 问题与解决方案

我们最开始学习面向对象编程的相关知识时，第一个障碍往往是理解继承当中的父子关系。随着时间的推移，我们会更加熟悉这种编程方法。当面对新的挑战时，经验丰富的面向对象编程人员会立即为某个对象扩展添加更多的功能。不过，正如万物皆有度，只有适度的使用才能保证这种工作的良好开展。

代码库应当对类层次的数量有所限制。如果对象开始要求启用过多的子类，那么相应的代码就会牺牲编程人员的理解力和可维护性。通常，我竭力保证用于一个对象的父-子关系不超过 3 个。我发现：只要创建较多的父-子关系，那么代码就会变得混乱和难以控制。除此之外，使用一般的纸张也无法打印出应用程序中任何对象的 UML 图表示。

生成复杂的类层次时需要小心的另一个原因是：PHP 限制只扩展一个类。如果在 PHP 中存在更全面的扩展模型，那么就很少会创建无法控制的对象关系。

不过，我并不希望阻止类扩展的使用。实际上，我们经常使用适当的解决方案来扩展

对象，甚至本书中的某些设计模式示例都要求将对象进行扩展。但是，对于某些问题来说，使用基于装饰器设计模式的类是更好的解决方案。

装饰器设计模式适用于编程人员花费大量时间所处的下列工作场合：变化是快速和细小的，而且几乎不影响应用程序的其余部分。使用装饰器设计模式设计类的目标是：不必重写任何已有的功能性，而是对某个基对象应用增量变化。装饰器采用这样的构建方式：在主代码流中应当能够直接插入一个或多个更改或“装饰”目标对象的装饰器，同时不影响其他代码流。

某些编程人员建议最好通过扩展现有对象来构建基于装饰器设计模式的对象，该装饰器随后可提供其他方法，或者可能重写现有方法。这不仅仅类似于适配器设计模式的某些基本概念，而且确实撤消了装饰器设计模式的一个主要目的。在不修改对象结构的情况下，装饰器可以提供对对象内容或功能性的快速非侵入性修改。在这个特定的案例中，扩展某个对象以添加其他功能性和修改基本结构。主代码流中需要在新子类中包含新功能的所有位置都必须被修改。父类的实例化需要替换为使用子类的名称。

一个应使用装饰器设计模式的场合是将用户输入传递至外部系统。假设某个进程上传用户的文件，在数据库内部关联适当的用户 ID，最后将这些内容存放至网络存储设备。网络存储设备上的文件系统允许混用大小写文件名，因此这个进程相当简单。稍后，引入采用传统文件系统的网络存储设备。这要求所有文件名只能以大写形式存在。当文件被物理移动至驱动器时，文件名会自动转换为大写形式。不过，对象内部仍然存储混用大小写的文件名。如果这种紧密联系的关系不复存在，那么可能导致系统不稳定。

这种情况下的最佳解决方案是引入一个基于装饰器设计模式的对象。在所有文件被移动至物理存储空间之后，但是在信息记入数据库之前，这个对象可直接修改用户文件管理对象。如果要传递对对象的引用，就要允许装饰器修改内部数据，同时将用大写形式重写每个文件名。随后，用户文件管理对象继续处理数据的插入。需要特别注意的是：通过引入新的需求和装饰器解决方案，基对象结构没有被修改。

适合使用基于装饰器设计模式的对象的另一个实例是处理博客输出。通常，此时会存在一个标记条件的标准集：修改图像链接至实际的图像，找到链接并将其变为可单击的对象，以及应用视觉风格(如粗体或斜体)。如果博客内容更为专业化，那么还可能要求获取或修改其他项。为了形成一个更好的用户体验，我们可以添加额外的功能。这些装饰示例包括：将某个地址变为指向地图的可单击链接；对商标名应用某种样式，从而使其与广告主题融为一体。采用这种方式添加过多的功能，会导致一个类的层次体系结构过大。通过创建基于装饰器设计模式的对象，我们就可以最佳地执行这些细小的更改。

当引入的需求要求在不影响现有基本代码稳定性的前提下对应用程序的内容或功能性进行很小的改动时，最好的做法就是创建装饰器对象。

## 6.2 UML

如图 6-1 所示，该 UML 图详细说明了一个使用装饰器设计模式的类设计。

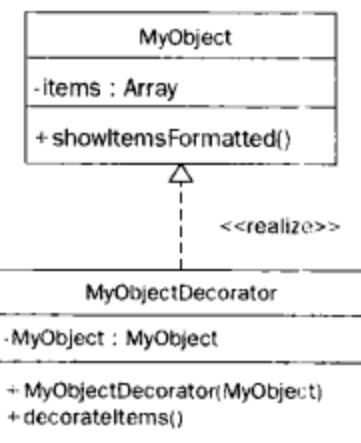


图 6-1

下面是对图 6-1 的说明：

- MyObject 是具有现有功能性的基类。这个类包含名为 items 的公共数组和名为 showItemsFormatted( ) 的公共方法。
- showItemsFormatted( ) 方法负责接受 items 数组，并且使用预定义的功能性格式化该数组后提交输出。
- MyObjectDecorator 类包含 MyObject 的一个私有实例和两个公共方法：MyObjectDecorator( ) 和 decorateItems( )。
- MyObjectDecorator( ) 方法代表构造函数，它接受一个 MyObject 类型参数并将其存储在内部。
- decorateItems( ) 方法可修改 MyObject 实例的 items 数组。

## 6.3 代码示例

在这个示例中，应用程序对光盘(Compact Disc, CD)进行处理。应用程序必须具有为 CD 添加音轨的方法以及显示 CD 音轨列表的方式。客户指定了应当采用单行并且每个音轨都必须以音轨号为前缀的方式显示 CD 音轨列表。

```

class CD
{
    public $trackList;

    public function __construct()
  
```

## 第II部分 参考内容

```

    {
        $this->trackList = array();
    }

    public function addTrack($track)
    {
        $this->trackList[] = $track;
    }

    public function getTrackList()
    {
        $output = '';

        foreach ($this->trackList as $num=>$track) {
            $output .= ($num + 1) . ") {$track}. ";
        }

        return $output;
    }
}

```

CD 类包含一个名为\$trackList 的公共变量，该变量将存储为 CD 对象添加的音轨数组。构造函数初始化了这个变量。addTrack( )方法只是为 CD 对象的 trackList 数组添加一个音轨。最后，getTrackList( )将方法遍历 CD 上的每个音轨，并且将这些音轨按指定格式编辑为单个字符串。

为了使用这个 CD 对象，需要执行下面的代码：

```

$tracksFromExternalSource = array('What It Means', 'Brr', 'Goodbye');

$myCD = new CD();

foreach ($tracksFromExternalSource as $track) {
    $myCD->addTrack($track);
}

print "The CD contains

```

对于该示例来说，上述代码已经可以很好地解决问题了。不过，此时需求又发生了小变化：只针对这个输出实例，输出的每个音轨都需要采用大写形式。对于这么小的变化而言，最佳的做法并非修改基类或创建父—子关系，而是创建一个基于装饰器设计模式的对象。

```

class CDTrackListDecoratorCaps
{

```

```

private $__cd;

public function __construct(CD $cd)
{
    $this->__cd = $cd;
}

public function makeCaps()
{
    foreach ($this->__cd->trackList as &$track)
    {
        $track = strtoupper($track);
    }
}
}

```

CDTrackListDecoratorCaps 类非常简单。\_\_construct()方法只是将 CD 类的实例添加至内部私有变量\$\_\_cd。在最开始的时候，这个变量似乎是隐秘的，甚至不可能通过真正的装饰器来修改基本对象，但是 PHP 通过引用就可以完成对对象的处理工作。即使实例是内部存储的和私有的，主代码流也会立即对其进行任何修改。

makeCaps()方法存在于装饰器内，它可以执行所需的修饰或更改。在这个示例中，该方法会遍历每个音轨并对这些音轨执行 PHP 的 strtoupper()函数。

如下所示，为了在原有示例中加入装饰器，需要添加新的 CDTrackListDecoratorCaps 类：

```

$myCD = new CD();

foreach ($tracksFromExternalSource as $track) {
    $myCD->addTrack($track);
}

$myCDCaps = new CDTrackListDecoratorCaps($myCD);
$myCDCaps->makeCaps();

print "The CD contains the following tracks: " . $myCD->getTrackList();

```

主代码流只需添加两行代码就能够完成这个很小的变化。通过引用已有的 CD 对象实例化 CDTrackListDecoratorCaps，我们就可以创建\$myCDCaps 变量。随后，调用 makeCaps() 函数可以执行这个稍有变化的功能性。

为了在不修改对象结构的前提下对现有对象的内容或功能性稍加修改，就应当使用装饰器设计模式。



# 第 7 章

## 委托模式

面向对象编程的最强大功能之一是其拥有的动态特性。当今世界不断涌现出更多可用的功能、混合搭建结构以及持续发展的标准，动态代码具有了全新的含义。无论是新文件存储标准或流媒体标准，还是社会网站或某些现有 Internet 先锋 API 上的新事物，Web 编程总是在不断地发生突变。如今在面对大量的可用选项时，处理判决的传统方式不再有效。通过将智能化对象移动到适当的位置，委托设计模式能够远离复杂的判决。

### 名称：委托

通过分配或委托至其他对象，委托设计模式能够去除核心对象中的判决和复杂的功能性。

### 7.1 问题与解决方案

大多数 PHP 编程人员最初接触的几乎都是程序化的编程类型。这种编程样式极大地依赖于以条件语句为基础的流控制。面向对象的编程提供了某些不同于传统条件语句的手段，从而创建了更多态的代码流。这个功能的实现方法之一是创建基于委托设计模式的对象。

委托设计模式致力于从核心对象中去除复杂性。此时我们并不设计极大依赖于通过评估条件语句而执行特定功能性的对象，基于委托模式的对象能够将判决委托给不同的对象。委托既可以像使用中间对象处理判决树一样简单，也可以像使用动态实例化对象提供期望的功能一样复杂。

不要将委托设计模式视为条件语句的直接竞争者，这是非常重要的。相反，委托设计模式通过不需要条件语句就可以调用正确功能性的的方式来帮助构成体系结构。条件语句最好驻留在实际方法中，并且在方法中完成对业务规则的处理。

委托设计模式的一个使用示例是为特定数据部分提供多种格式。假设在开放源代码库中存在一个归档。当访问者打算下载部分源代码时，他们可以选择两种不同格式的文件。

## 第II部分 参考内容

指定文件被压缩后将被发送至浏览器。在这个示例中，我打算采用 zip 和 tgz 压缩格式的文件。

通常，我们需要创建一个文件收集和下载对象。这个对象具有收集被请求文件以及再将对这些文件的引用存储在内部的方法。随后，专门用于指定压缩类型的方法会被调用。如果压缩类型为 zip，那么就会调用 generateZip( )方法。

我们应当使用基于委托设计模式的对象来取代上述约定的函数。generateZip( )方法的功能性应当被转移至针对基对象的文件列表执行该功能的 Delegate 类。这不仅可以减少基对象的复杂性，而且也可以为代码提供更好的可维护性。如果要采用新的压缩类型(如.dmg)，那么只需要创建新的 Delegate 对象。稳定的基对象并不需要被编辑。

当一个对象包含复杂但独立的、必须基于判决执行的功能性的若干部分时，最佳的做法是使用基于委托设计模式的对象。

## 7.2 UML

如图 7-1 所示，该 UML 图详细说明了一个使用委托设计模式的类设计。

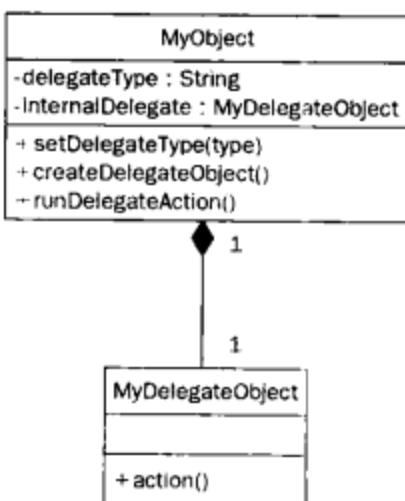


图 7-1

下面是对图 7-1 的说明：

- 基类 MyObject 知道会使用基于委托设计模式的对象。这个类包含私有字符串 delegateType 以及 MyDelegateObject 的私有实例 internalDelegate。
- setDelegateType( )方法接收一个名为 type 的参数，这个参数被存储在 delegateType 字符串中。
- createDelegateObject( )方法会创建委托对象的一个实例，并且根据 delegateType 变量为实例命名。随后，通过将该实例指派给 internalDelegate，这个方法将其存储在内部。

- `runDelegateAction()`方法负责运行 `internalDelegate` 对象的 `action()`方法。
- `MyDelegateObject` 包含负责特定动作的逻辑。`MyObject` 运行 `action()`方法实现具体的功能。

## 7.3 代码示例

示例中的 Web 站点具有创建 MP3 文件播放列表的功能。MP3 文件可以来自访问者的硬盘驱动器，也可以来自 Internet。访问者可以选择以 M3U 或 PLS 格式下载播放列表。为了节省篇幅，具体的代码示例只会显示播放列表的创建部分。

如下所示，第一个步骤是创建 Playlist 类：

```
class Playlist
{
    private $__songs;

    public function __construct()
    {
        $this->__songs = array();
    }

    public function addSong($location, $title)
    {
        $song = array('location'=>$location, 'title'=>$title);
        $this->__songs[] = $song;
    }

    public function getM3U()
    {
        $m3u = "#EXTM3U\n\n";

        foreach ($this->__songs as $song) {
            $m3u .= "#EXTINF:-1,{$song['title']}\\n";
            $m3u .= "{$song['location']}\\n";
        }
        return $m3u;
    }

    public function getPLS()
    {
        $pls = "[playlist]\\nNumberOfEntries=".count($this->__songs)."\n\\n";
    }
}
```

## 第II部分 参考内容

```

        foreach ($this->_songs as $songCount => $song) {
            $counter = $songCount + 1;
            $pls .= "File{$counter}={$song['location']}\\n";
            $pls .= "Title{$counter}={$song['title']}\\n";
            $pls .= "Length{$counter}=-1\\n\\n";
        }
        return $pls;
    }
}

```

Playlist 对象存储一个歌曲数组，这个数组是由构造数组初始化的。

`addSong()`公共方法接受两个参数：MP3 文件的位置和该文件的标题。这两个参数组成一个关联数组，随后被添加至内部的歌曲数组。

具体的需求规定播放列表必须可以任意使用 M3U 和 PLS 格式。为此，Playlist 类具有 `getM3U()` 和 `getPLS()` 两个方法。这两个方法都负责创建适当的播放列表文件头并遍历内部的歌曲数组以完成播放列表。之后，每个方法都会返回采用字符串格式的播放列表。

如下所示，执行上述功能性的当前代码包含了常见的 `if/else` 子句：

```

$playlist = new Playlist();
$playlist->addSong('/home/aaron/music/brr.mp3', 'Brr');
$playlist->addSong('/home/aaron/music/goodbye.mp3', 'Goodbye');

if ($externalRetrievedType == 'pls') {
    $playlistContent = $playlist->getPLS();
}
else {
    $playlistContent = $playlist->getM3U();
}

```

上面的代码创建了 Playlist 对象的一个新实例，添加了两个歌曲位置和标题，接着创建了一个 `if/else` 子句。如果格式类型为 `pls`，那么就执行 `getPLS()` 方法，其输出被置入变量 `$playlistContent`。否则，`$externalRetrievedType` 可能包含 `m3u`，这将进入 `if/else` 子句的 `else` 部分。

这个 Web 站点的销售团队发现至少可以使用 5 种以上播放列表格式。因此，他们在开发之前就开始推销相应的软件功能。那时，编程人员仍然不知道哪种新播放列表格式会被出售。

在这一时期，编程人员可以使用委托设计模式来更改代码。这种做法的目的是消除潜在的、难以控制的 `if/else` 语句。此外，随着添加更多的代码，最初的 Playlist 类会变得极为庞大。

创建 newPlaylist 类是因为意识到要使用委托设计模式的事实。PHP 动态创建基于某个变量的类实例的能力也十分有用。

```
class newPlaylist
{
    private $__songs;
    private $__typeObject;

    public function __construct($type)
    {
        $this->__songs = array();
        $object = "{$type}Playlist";
        $this->__typeObject = new $object;
    }

    public function addSong($location, $title)
    {
        $song = array('location'=>$location, 'title'=>$title);
        $this->__songs[] = $song;
    }

    public function getPlaylist()
    {
        $playlist = $this->__typeObject->getPlaylist($this->__songs);
        return $playlist;
    }
}
```

newPlaylist 对象的构造函数现在接受\$type 参数。除了初始化内部的歌曲数组之外，构造函数还可以根据\$type 动态地创建指定委托的新实例并将该实例内部存储在\$\_\_type Object 变量中。

addSongs( )方法与最初 Playlist 对象所包含的方法相同。getM3U( )和 getPLS( )方法被替换为 getPlaylist( )方法。这个方法执行内部存储的委托对象的 getPlaylist( )方法，它将歌曲数组传递指定对象，从而使该对象能够创建和返回正确的播放列表。

如下所示，作为 Playlist 对象原有的上述两个方法被移动至这些对象自己的委托对象：

```
class m3uPlaylistDelegate
{
    public function getPlaylist($songs)
    {
        $m3u = "#EXTM3U\n\n";
    }
}
```

## 第II部分 参考内容

```

foreach ($songs as $song) {
    $m3u .= "#EXTINF:-1,{$song['title']}\n";
    $m3u .= "{$song['location']} \n";
}

return $m3u;
}

}

class plsPlaylistDelegate
{
    public function getPlaylist($songs)
    {
        $pls = "[playlist]\nNumberOfEntries=" . count($songs) . "\n\n";

        foreach ($songs as $songCount=>$song) {
            $counter = $songCount + 1;
            $pls .= "File{$counter}={$song['location']}\n";
            $pls .= "Title{$counter}={$song['title']}\n";
            $pls .= "Length{$counter}=-1\n\n";
        }

        return $pls;
    }
}

```

每个委托类本质上都只是重新包装基类 Playlist 中的原有方法。每个委托对象都具有完全相同的指定公共方法 getPlaylist( )，该方法接受 songs 参数。这种方式使基对象能够简单、动态地创建和访问任何委托者。

如下所示，执行这个基于委托的新系统的代码更为简单：

```

$externalRetrievedType = 'pls';

$playlist = new newPlaylist($externalRetrievedType);
$playlistContent = $playlist->getPlaylist();

```

当通告其他播放列表格式时，开发人员不必修改上面的代码就能够创建基于委托设计模式的新类。

为了去除核心对象的复杂性并且能够动态添加新的功能，就应当使用委托设计模式。

# 第 8 章

## 外 观 模 式

如果应用程序编程是简单的，那么任何人都能够胜任此项工作。那样，人们就不再需要相关书籍，软件开发将变得无趣，相关行业为了生存则需要开发不同的业务模型。但是，编程并不是简单的，事实上相当复杂。虽然承认这种复杂性，但是高级编程人员仍然致力于简化自己的系统。他们选择使用任何可用的设计模式(包括外观设计模式)，从而抓住每一个机会去除复杂性。听到术语“外观(facade)”时，大多数人脑海中会浮现出旧建筑的装饰性门面。其他一些人可能想到狡猾的人在困境中营造的假象，这些假象的目的是试图迷惑周围的人。人的动作、感知或反应似乎非常简单，它们隐藏了人们可能经历的复杂性。同样地，外观设计模式被设计用于处理人们误认为简单的复杂组件。

### 名称：外观

通过在必需的逻辑和方法的集合前创建简单的外观接口，外观设计模式隐藏了来自调用对象的复杂性。

### 8.1 问题与解决方案

本书中反复出现的一个主题似乎是去除代码中的复杂性。但是，我们没有必要尝试去除代码中的复杂性。不同对象的连接才是简化的目标。项目设计师应当向复杂子系统的编程人员虚心学习，体会子系统的复杂性、质量和执行，然后将其插入整个项目。不过，由于内部组件逻辑变得更为复杂，因此外部交互也是如此。外观设计模式的目标是：控制外部错综复杂的关系，并且提供简单的接口以利用上述组件的能力。外观设计模式的独特性在于被设计为将多个互相联系的组件组合或连接入简单可用的接口内。

按照更为实际的说法，基于外观设计模式的类可能提供执行一个逻辑业务请求的公共接口。为了完成这个单独的业务请求，可能需要执行多个技术逻辑步骤。业务进程并不总

## 第II部分 参考内容

是像它们的名称所指那样简单。例如，具体进程的标题可能为 Make Shared Files Available on Network(使共享文件在网络上可用)。根据这个标题给出的观点，编程人员知道他们必须执行下列技术处理：Create File Share(创建文件共享)、Move File to File Share(移动文件至文件共享)以及 Apply Proper Permissions to File(对文件应用适当的权限)。通过首先调用上述每个技术需求，外观设计模式提供了名为 Make Shared Files Available on Network 的接口。

使用基于外观设计模式的对象的另一个原因是接口第三方解决方案。需要记住的是，我们不断强调面向对象的项目应当只是关联对象的一个集合。鉴于这种体系结构，首席编程人员可能觉得使用第三方对象更为明智。

假设要为某个应用程序提供搜索 Web 页面。该页面首先自己查找符合搜索项的所有数据。如果结果数小于 10，那么就会调用第三方服务(如 Google)检索其他结果，这些结果会被添加至应用程序在内部发现的结果末尾。Search 外观对象将结果返回至调用视图对象。在内部，Search 外观对象会调用查询内部数据库的方法。随后，它会判断是否需要通过 Web 服务来调用 Google。如果需要，那么还会分析所有结果，以便返回一个同类结果集。

如果读者还不能理解这种体系结构的优点，那么可以看看该应用程序的下一步发展。Yahoo! 搜索引擎能够返回比 Google 更好的结果。此时，外部的 Web 服务请求需要被更改为调用 Yahoo! 的 API。

如果采用传统的方式，那么每次创建对结果的请求时，Google API 都需要被替换。不过，如果采用 Search 外观对象，那么就不必对调用视图对象进行任何更改。此时，我们只需要创建一个 Yahoo! Search 类，随后将 Search 外观对象的方法从使用 Google Search 类更改为使用 Yahoo! Search 类。

为了隐藏复杂的、执行业务进程某个步骤所需的方法和逻辑组，就应当使用基于外观设计模式的类。

## 8.2 UML

如图 8-1 所示，该 UML 图详细说明了一个使用外观设计模式的类设计。

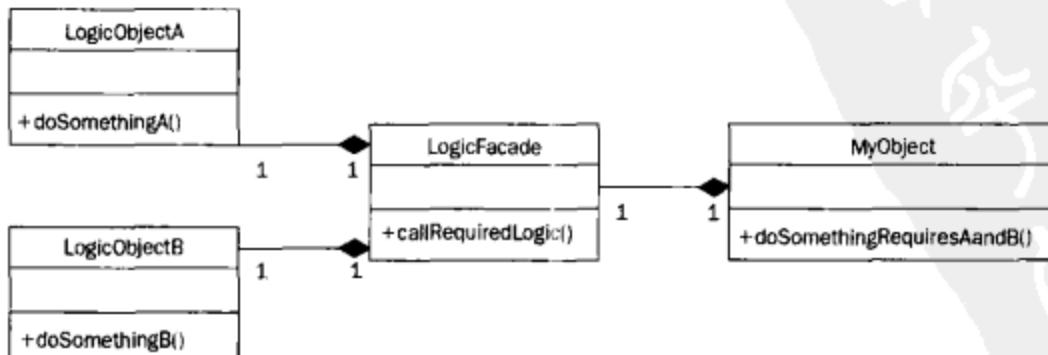


图 8-1

下面是对图 8-1 的说明：

- MyObject 类包含一个名为 doSomethingRequiresAandB() 的公共方法，该方法只是 MyObject 类执行中的一个步骤。doSomethingRequiresAandB() 方法创建对象 LogicFacade 的一个新实例，该实例调用对于 MyObject 来说足够抽象的、名称为 callRequiredLogic() 的公共方法。
- LogicFacade 类内部的 callRequiredLogic() 方法随后负责创建 LogicObjectA 的一个实例以及调用 doSomethingA() 方法，此外还负责创建 LogicObjectB 的一个实例以及调用 doSomethingB() 方法。
- 上述所有动作都通过 LogicFacade 类向回传递，从而使它们能够被 MyObject 使用。

### 8.3 代码示例

为了完全必要的审计，公司的 Web 站点每夜都要将其库存信息传递至公司内的不同系统。这个不同的系统会接受通过提交实现的对其 Web 服务的请求。不过，它是一个较老的系统，并且只处理大写的字符字符串。具体代码需要获取 CD 对象，对其所有属性应用大写形式，并且创建一个要提交给 Web 服务的、格式完整的 XML 文档。

下面给出了 CD 类的一个简单示例：

```
class CD
{
    public $tracks = array();
    public $band = '';
    public $title = '';

    public function __construct($title, $band, $tracks)
    {
        $this->title = $title;
        $this->band = $band;
        $this->tracks = $tracks;
    }
}
```

在实例化新的 CD 对象时，构造函数会为这个 CD 对象添加标题、乐队和曲目列表。如下所示，构建 CD 对象的步骤相当简单：

```
$tracksFromExternalSource = array('What It Means', 'Brrr', 'Goodbye');
$title = 'Waste of a Rib';
$band = 'Never Again';
```

```
$cd = new CD($title, $band, $tracksFromExternalSource);
```

要为外部系统格式化 CD 对象，就需要创建其他两个类。第一个类用于准备 CD 对象的属性。要求的格式是大写形式。另一个类则负责根据 CD 对象构建 XML 文档，这个类会返回由完整文档组成的一个字符串。

注意到这两个类是为了最大可重用性而创建的是十分重要的。用户可能想将上述所有步骤组合入一个类，但是以后您可能会被要求进行分解。

```
class CDUpperCase
{
    public static function makeString(CD $cd, $type)
    {
        $cd->$type = strtoupper($cd->$type);
    }

    public static function makeArray(CD $cd, $type)
    {
        $cd->$type = array_map('strtoupper', $cd->$type);
    }
}

class CDMakeXML
{
    public static function create(CD $cd)
    {
        $doc = new DomDocument();

        $root = $doc->createElement('CD');
        $root = $doc->appendChild($root);

        $title = $doc->createElement('TITLE', $cd->title);
        $title = $root->appendChild($title);

        $band = $doc->createElement('BAND', $cd->band);
        $band = $root->appendChild($band);

        $tracks = $doc->createElement('TRACKS');
        $tracks = $root->appendChild($tracks);

        foreach ($cd->tracks as $track) {
            $track = $doc->createElement('TRACK', $track);
            $track = $tracks->appendChild($track);
        }
    }
}
```

```

    $track = $tracks->appendChild($track);
}

return $doc->saveXML();
}
}

```

CDUpperCase 对象具有两个公共的静态方法。第一个方法 makeString( ) 接受 CD 对象和一个名为\$type 的字符串参数，它只是对 CD 实例的、根据\$type 内容命名的公共变量应用 PHP 的 strtoupper( ) 函数。另一个方法名为 makeArray( )，其功能类似于 makeString( )，它对 CD 对象公共数组中使用 array\_map( ) 函数且根据\$type 内容命名的每个项都应用 strtoupper( ) 方法。因为在 PHP 中按照引用传递 CD 对象，所以没有定义返回变量。上述所有方法的动态执行允许在今后 CD 类扩展包含更多公共属性时仍然能够使用这个类。

CDMakeXML 对象只具有一个公共方法 create( )，该方法接受 CD 对象，并且负责返回来自 CD 内容的、结构完整的 XML 文档。简单说来，这个方法为标题、乐队和曲目列表创建使用大写标记名称的元素。

乍看起来，编程人员可能希望通过下面的方式实现这样的功能：

```

CDUpperCase::makeString($cd, 'title');
CDUpperCase::makeString($cd, 'band');
CDUpperCase::makeArray($cd, 'tracks');
print CDMakeXML::create($cd);

```

虽然这是解决问题的一种方式，但并不是最佳方式。如下所示，我们应当针对具体的 Web 服务调用创建一个 Façade 对象：

```

class WebServiceFacade
{
    public static function makeXMLCall(CD $cd)
    {
        CDUpperCase::makeString($cd, 'title');
        CDUpperCase::makeString($cd, 'band');
        CDUpperCase::makeArray($cd, 'tracks');

        $xml = CDMakeXML::create($cd);

        return $xml;
    }
}

```

## 第II部分 参考内容

WebServiceFacade 对象只具有一个名为 makeXMLCall( )的公共静态方法，该方法接受 CD 对象并返回一个 XML 文档。我们可以将前面介绍的创建 XML 文档所使用的步骤移动至这个外观对象的方法内。此时，前面列出的 4 行代码只需要被替换为如下所示的 1 行代码：

```
print WebServiceFacade::makeXMLCall($cd);
```

在应用程序进程中的下一步骤包含许多复杂的逻辑步骤和方法调用时，最佳的做法是创建一个基于外观设计模式的对象。

# 第 9 章

## 工厂模式

在面向对象的程序中，我曾经见过的最大的 switch/case 语句所具有的条件超过了 20 个。在运行期间，这个代码模块每次都要执行所有条件。每个条件都负责创建一个新的对象，这个对象被用于与应用程序 API 的外部使用者进行通信。对其中一个执行某些常规的故障处理之后，我开始研究应用程序所实现的接口。令我吃惊的是，switch/case 语句中引用的每个类都在实现相同的接口！随后要在 switch/case 语句中添加新的条件语句时，我建议转而使用工厂模式。正如前面所介绍的，设计模式的名称非常重要，它们不仅提供了引用每种设计模式的一致性，而且是模式实际作用的关键所在。在上面所介绍的代码中，工厂设计模式是最适用的。

### 名称：工厂

工厂设计模式提供获取某个对象的新实例的一个接口，同时使调用代码避免确定实际实例化基类的步骤。

### 9.1 问题与解决方案

随着 PHP 发展并演变为一种语言，它的功能不断地通过应用经过验证的设计模式来提供更简单的开发手段。PHP 中一个特别有用的功能是能够创建基于变量内容的类的新实例。这种对象实例化的动态方法是 PHP 中实现工厂设计模式的一种代码块构建途径。

基于工厂设计模式的类有助于减少主代码流中基于条件的复杂性。在整个应用程序中，调用对象的方式众多而且不同。与某个对象创建相关的任何修改都会影响到应用程序的其余部分。假设实例化 5 个对象中的其中一个，从而完成某种功能性。创建条件语句来判断实例化的对象是一种方法。条件语句可能是复杂的 if/else 语句或 switch/case 语句。这种功能性可以在应用程序中的许多地方使用，但是会导致重复的代码。接下来，如果添加

## 第II部分 参考内容

第六个对象或修改现有 5 个对象其中一个的名称，那么程序代码的所有实例都需要被修改和再次测试。通过提供一个创建上述对象的简单接口，工厂设计模式有助于避免这个令人头痛的问题。无论对象被修改或者添加了其他对象，Factory 对象的调用方式仍然是相同的。

显示博客条目就是现实中的一个实例。个性化的博客非常流行，并且具有向使用者提供其内容的多种方式，包括标准的 Web 浏览器、RSS feed、移动传输以及 REST API。实际检索适当博客项的代码流或控制器并不需要关心具体使用的视图，而只是从视图创建工厂中请求新的视图对象。一旦具有该视图对象的实例，文章对象就会被传递至视图内。最后，代码流会调用执行视图对象的渲染对象。在上述整个过程中，由于使用了工厂对象，所以主代码流不必处理找出待创建视图对象的问题，而是笼统地调用工厂对象并提供正确的对象进行处理。

然而，不同对象的创建并不是使用工厂设计模式的唯一目的。使用基于工厂设计模式的类的另一个场合是处理若干项的集合。在这种情况下，对象集合包含相同的基对象，但是每个对象都具有不同的特征。

库存系统是使用工厂类管理对象集合的一个优秀实例。某个音乐店可能具有显示其吉他库存的应用系统。具体的视图最初被创建用于处理单个吉他对象，以便确定其品牌、型号、颜色和字符弦数。为了显示库存中的多个结果，我们可以使用一个 Guitars 工厂对象，该对象接受来自数据库的实例化 ID 集合，随后使用名为 getGuitar( ) 的公共方法返回根据单个 ID 创建的吉他对象。此时，工厂对象能够根据 ID 集合不断地创建新的吉他对象，并且使用一个公共方法统一地返回这些吉他对象。

在需要若干步骤才能确定要创建的对象类型时，我们最好使用基于工厂设计模式的类来检索新的实例。

## 9.2 UML

如图 9-1 所示，该 UML 图详细说明了一个使用工厂设计模式的类设计。

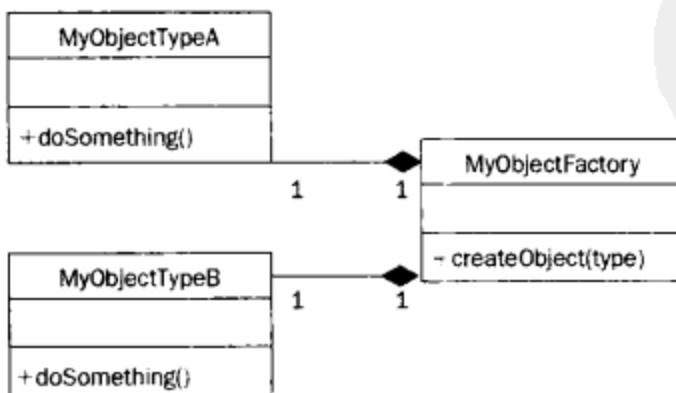


图 9-1

下面是对图 9-1 的说明：

- 现存的两个基类是 MyObjectTypeA 和 MyObjectTypeB。这两个类都具有名为 doSomething() 的公共方法，该方法采用自己独特的方式执行具体对象的逻辑。两个基类的公共接口和返回类型是完全相同的。
- MyObjectFactory 类用于创建上述任意一个基类的实例并将其返回至代码流。它具有一个名为 createObject() 的公共方法，该方法接受参数 type，这有助于判断应当创建哪一个基类的实例。随后，createObject() 方法会返回被请求类型类的一个实例。

## 9.3 代码示例

为了管理控制 CD，应用程序需要将必要的信息编辑入 CD 对象。将 CD 对象传递给外部供应商，这些人会完成实际的 CD 创建工作。CD 对象需要包含标题、乐队名称以及曲目列表。

如下所示，简单的 CD 类包含添加标题、乐队名称和曲目列表的方法。

```
class CD
{
    public $title = '';
    public $band = '';
    public $tracks = array();

    public function __construct()
    {}

    public function setTitle($title)
    {
        $this->title = $title;
    }

    public function setBand($band)
    {
        $this->band = $band;
    }

    public function addTrack($track)
    {
        $this->tracks[] = $track;
    }
}
```

## 第II部分 参考内容

---

为了创建完整的 CD 对象，处理过程总是相同的：首先创建 CD 类的一个实例，然后添加标题、乐队名称和曲目列表。

```
$title = 'Waste of a Rib';
$band = 'Never Again';
$tracksFromExternalSource = array('What It Means', 'Brrr', 'Goodbye');

$cd = new CD();
$cd->setTitle($title);
$cd->setBand($band);
foreach ($tracksFromExternalSource as $track) {
    $cd->addTrack($track);
}
```

如今，某些艺术家在他们的 CD 上发布了在计算机中能够使用的其他内容。这些 CD 称为增强型 CD。写至光盘的第一个音轨是数据音轨。管理控制软件通过其标签 DATA TRACK 识别数据音轨，并且创建相应的 CD 对象。

```
class enhancedCD
{
    public $title = '';
    public $band = '';
    public $tracks = array();

    public function __construct()
    {
        $this->tracks[] = 'DATA TRACK';
    }

    public function setTitle($title)
    {
        $this->title = $title;
    }

    public function setBand($band)
    {
        $this->band = $band;
    }

    public function addTrack($track)
    {
        $this->tracks[] = $track;
    }
}
```

查看上述共性和认识到只可能存在两种 CD 类型之后，似乎我们只需要创建条件语句。如果 CD 类型是增强型 CD，那么就创建 enhancedCD 类的新实例；否则，就应创建通用的 CD 类。然而，还存在更好的解决方案：使用工厂设计模式。

CDFactory 类使用了 PHP 根据变量动态实例化一个类的能力。create()方法接受被请求类的类型并返回该类的一个实例：

```
class CDFactory
{
    public static function create($type)
    {
        $class = strtolower($type) . "CD";

        return new $class;
    }
}
```

现在，类的创建和执行的变化反映了 Factory 类的用法：

```
$type = 'enhanced';

$cd = CDFactory::create($type);
$cd->setBand($band);
$cd->setTitle($title);
foreach ($tracksFromExternalSource as $track) {
    $cd->addTrack($track);
}
```

最后需要考虑的可能是已有 CD 类的名称。为了使其统一，将类名改变为 standardCD 也许更有实际意义。确认这样不会破坏代码中其他位置的其他功能性。此时，我们最好将 CD 的新实例修改为使用 CDFactory 类。

请求需要某些逻辑和步骤才能确定基对象的类实例时，最佳的做法是使用一个基于工厂设计模式的类。



# 第10章

## 解释器模式

无论喜欢还是讨厌算术类，用户都会应用到这些概念。即使是早期的代数概念，其核心思想仍然能够用于具体的编程工作。代数围绕于使用变量保存未知数。在等式中的代入值提供了对变量的解释，从而可以获得最后结果。编程语言(如 PHP)提供了另一种解释形式。因为其核心安装并不提前编辑代码，所以通常认为 PHP 是一种解释语言，也是一种模板语言。因为创建了设计更精细的应用程序，所以这种模板与解释语言变得更为高级。在多数情况下，存在的模板方式允许开发团队通过较少技术和复杂因素较少的途径创建实体。使用解释器设计模式审查这些实体并为模板提供替代品或对实体的解释。

### 名称：解释器

解释器设计模式用于分析一个实体的关键元素，并且针对每个元素都提供自己的解释或相应的动作。

### 10.1 问题与解决方案

解释器设计模式是我们经常使用但未意识到的少数几个常见设计模式之一。并未限制这种设计模式只用于类的创建。在创建大多数编程算法的过程中，我们都会使用到解释器设计模式的基本概念。

为了理解解释器设计模式的工作原理，我们以宏语言的处理为例。为每个宏编写的命令本身是更多命令的集合。简化的宏语言使编程人员不必考虑其他系统命令的语法就能够更容易地完成创建操作。在某些情况下，这也能提升安全性，此时编程人员不直接访问系统命令。相反，编程人员会在某种沙箱内编写包装器方法来执行系统命令。宏语言会被解释和转换为一组要执行的命令。

通过查看模板系统也可以认识解释器设计模式。特定的预定义关键字或符号被定义为

## 第II部分 参考内容

表示其他事物。模板处理器用于接受代码，解释每个关键字以引用特定的指令集，并执行这些代码。

构建基于解释器设计模式的系统允许第三方或用户更灵活地表示和检索系统提供的数据。我们并不通过预定义方法名或特定常量来表示检索数据的类型，而是使用一组关键字检索这些数据。

例如，通过发送一个包含关键字的请求，使用 Web 服务的第三方可以按照希望的顺序规定希望检索的值。这个请求可能按照其构造方式模拟一个 XML 路径语言查询。基于解释器设计模式的类随后会检索按请求顺序符号化的所有键组成的信息的每一位。由于对关键字的解释是一点点完成的，因此并不要求为一个复杂的预定义数据集设置顺序。对于一个 CD 来说，请求可能采用这样的形式：“band title track4”。此时，解释器知道向请求者返回乐队名称、CD 标题和曲目数 4 的名称。下一个请求可能是“track3 title”，解释器则会返回第三音轨的名称和 CD 的标题。

解释器设计模式最常用于 PHP/HTML 模板系统中。创建 HTML 文档时，其主体中使用了特定的占位符。占位符引用了处理对象、其他模板或文件系统中文件的函数或性能。这些模板系统常用于处理类似数据的大型集合(例如用户配置文件)。基本模板的创建使用了引用用户名称、家乡和照片的关键字。处理类随后将每个关键字解释为对相应用户数据和动作的请求。类似地，整个站点都会复制用于一个 Web 站点的标题头、导航条和页脚。使用关键字创建的基本模板会被解释为所请求页面的输出。处理对象负责解释关键字，以便标记当前请求输出的布局。

处理使用关键字引用功能性的请求时，最好借助于一个使用解释器设计模式创建的对象。

## 10.2 UML

如图 10-1 所示，该 UML 图详细说明了一个使用解释器设计模式的类设计。

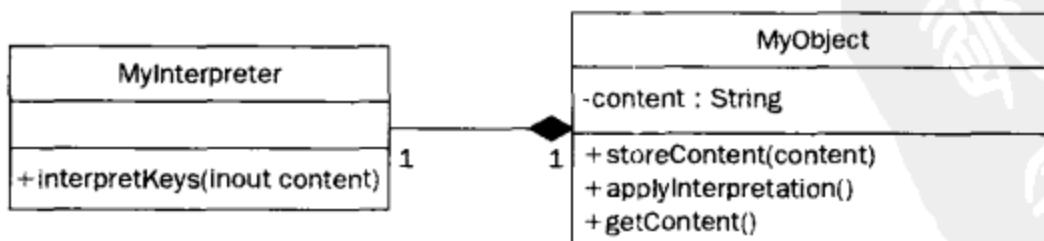


图 10-1

下面是对图 10-1 的说明。

- MyObject 类处理需要被解释的内容。它具有私有字符串 content，这个字符串存储需要处理的内容。
- storeContent( )方法接受一个名为 content 的参数，该方法对处理内容进行预解释，随后将结果存储在 MyObject 对象内部。
- 随后会调用 applyInterpretation( )方法。该方法会创建 MyInterpreter 类的一个实例。MyInterpreter 具有一个名为 interpretKeys( )的公共方法，这个方法接受参数 content. applyInterpretation( )并获取该参数的内部内容。MyInterpreter 类对要处理的内容执行解释，并且将结果返回 MyObject。接下来，applyInterpretation( )方法替换内部的内容变量。
- 最后，MyObject 通过 getContent( )方法提供解释后的内容。

## 10.3 代码示例

为了顺应潮流，这个示例中的 Web 站点决定合并 CD 购买活动和社会网络。注册了该 Web 站点的用户有自己的配置文件页面，他们能够添加高级的功能性，如 HTML、构件以及自己喜爱的 CD 列表。

在第一次迭代中，用户可以创建自己的配置文件，并且能够在配置文件中添加自己喜爱的 CD 标题。如下所示，第一部分功能性是 User 类：

```
class User
{
    protected $_username = '';

    public function __construct($username)
    {
        $this->_username = $username;
    }

    public function getProfilePage()
    {
        //In lieu of getting the info from the DB, we mock here
        $profile = " < h2 > I like Never Again! < /h2 > ";
        $profile .= "I love all of their songs. My favorite CD: < br /> ";
        $profile .= "{{myCD.getTitle}}!!";

        return $profile;
    }
}
```

## 第II部分 参考内容

---

大多数 User 类都模拟了这个示例。创建 User 类的实例时，用户名被指派给保护变量 `$_username`。在未模拟的示例中，某些逻辑可能用于查询数据库和为 User 对象初始化适当的值。`getProfilePage()` 函数也是一种模拟方法，它返回一个硬编码的配置文件。不过，该示例需要注意的重要部分是 `{(myCD.getTitle)}` 字符串，这表示稍后需要解释的模板语言。`getProfilePage()` 只返回用户在其配置文件页面中指定的内容。

为了为用户检索 CD 信息，需要创建一个名为 userCD 的新对象：

```
class userCD
{
    protected $_user = NULL;

    public function setUser($user)
    {
        $this->_user = $user;
    }

    public function getTitle()
    {
        //mock here
        $title = 'Waste of a Rib';

        return $title;
    }
}
```

同样地，这个示例也基于程度极深的模拟。`setUser()` 方法接受用户对象并将其存储在内部。在更健壮的示例中，该方法可以创建 CD 对象的一个实例并将其存储在内部。`getTitle()` 方法会从 CD 中检索和返回标题。

注意到 `getTitle()` 方法的名称与用户配置文件中指定的模板语言之间的相似性，这是十分重要的。如下所示，解释器类使用了这种相似性：

```
class userCDInterpreter
{
    protected $_user = NULL;

    public function setUser($user)
    {
        $this->_user = $user;
    }

    public function getInterpreted()
```

```

{
    $profile = $this->_user->getProfilePage();

    if (preg_match_all('/\{\{myCD\.(.*?)\}\}/', $profile,
        $triggers, PREG_SET_ORDER)) {
        $replacements = array();

        foreach ($triggers as $trigger) {
            $replacements[] = $trigger[1];
        }

        $replacements = array_unique($replacements);

        $myCD = new userCD();
        $myCD->setUser($this->_user);

        foreach ($replacements as $replacement) {
            $profile = str_replace("{{myCD.{$replacement}}}", 
                call_user_func(array($myCD, $replacement)), $profile);
        }
    }

    return $profile;
}
}

```

`userCDInterpreter` 类包含 `setUser()` 方法，该方法接受一个 `User` 对象并将其存储在内部。除此之外，这个类包含的其他方法是名为 `getInterpreted()` 的公共函数。

- 首先，`getInterpreted()` 方法从 `User` 对象中获取内部存储的配置文件。接着，该方法通过解析配置文件查找能够处理的可解释关键字。如果发现这样的关键字，那么就会构造一个替换数组。操作完成后，就会生成一个独特的替换组。
- 下一个步骤是创建基于 `userCD` 对象的新 `CD` 对象。创建这个新对象后，`User` 实例会被传递入其内部。
- 最后，所有替换都会被遍历。根据属于 `userCD` 实例的 `$replacement` 变量的内容而命名的方法会被调用，该方法的输出用于替换配置文件中的被解释占位符。完成所有解释后，则返回配置文件。

现在，实际执行解释和生成模板化输出的代码非常简单，如下所示：

```
$username = 'aaron';
```

## 第II部分 参考内容

```
$user = new User($username);
$interpreter = new userCDInterpreter();
$interpreter->setUser($user);

print "< h1 > {$username}'s Profile < /h1 > ";
print $interpreter->getInterpreted();
```

这样就可以创建用户实例、创建一个基于解释器设计模式的类以及执行解释。  
当使用关键字或宏语言引用一组指令时，最佳的做法是使用基于解释器设计模式的类。

# 第11章

## 迭代器模式

计算机最有价值的工作之一是执行重复的任务。计算机总是“不厌倦”或不知疲惫地以同样的方式执行操作，它可以周而复始地一直重复相同的工作。这也是计算机成为主流、能够普及以及作为家用主要产品的原因之一。计算机重复执行简单操作的能力令人惊叹。例如重复演奏音乐或反复修正拼写错误的词汇时所使用的简单数学运算。计算机编程语言通过被称为循环的构造来管理重复。也许我们还未意识到，目前几乎所有的程序和动作中都使用了循环。然而，从编程的角度出发，在进入循环时，并非所有对象都是相同的。某些对象要求复杂的散列表访问。其他对象能够像数组一样被处理。正因为对象接口中的这种复杂性，所以就建立了一种在很多项之间进行循环的常用方法。此时，使用迭代器设计模式就是最适当的。

### 名称：迭代器

迭代器设计模式可帮助构造特定对象，那些对象能够提供单一标准接口循环或迭代任何类型的可计数数据。

### 11.1 问题与解决方案

这些设计模式有意义的最具说服力的证据之一是不断出现重构编程人员已创建代码的情况。我总是回想自己所创建第一个数据驱动的站点。这个站点使用平面文件来处理数据，并且相当笨重和缓慢。在接触到 MySQL 时，我将所有数据迁移入一些数据库表内，不过仍旧需要编写许多查询返回数据。当深入学习面向对象的重构时，我开始根据这些 MySQL 查询创建数据对象。此时，我仍然手工创建每个对象，接着再对其进行访问。最终，我记得最后的步骤是创建一个对象，该对象能够创建用于页面的各种对象，它具有的方法能够从列表中获取下一个对象。那时，我根本没有意识到，事实上我已经初步实现了一种

设计模式：迭代器设计模式。

迭代器设计模式帮助样式对象处理数据或其他对象的集合。创建基于迭代器设计模式的类时，为了提供管理这些集合的统一方式，我们还会创建一组接口。例如，某些对象将自身的创建描述为对集合来说完整和可用的证明。其他对象可能在其能够作为集合的部分进行处理之前要求另外的构建操作。基于迭代器设计模式的类能够提供统一的访问集合的公共方法。不过，这个类内部会应用逻辑来确定从集合中返回的下一个对象。

有些时候，数据集似乎十分简单。编程人员可能无法预计将会发生变化的情景，因此他们选择不修改代码和不创建迭代器。调用数据库时经常会出现这样的情况。此时，编程人员创建了 MySQL 查询，随后会执行简单的取数组命令。但是，在代码中保留这样的程序性方式并不是最佳的解决方案。

处理 MySQL 结果集时应当创建迭代器对象。比较简单的做法是为类构造函数提供一个 MySQL 查询，然后通过调用迭代器对象的公共方法遍历循环结果集。更复杂的迭代器示例可能还具有要发送给迭代器的其他参数，根据这些条件，就可能执行不同的 MySQL 查询集。不过无论如何，外部的代码流只通过处理相同的公共方法来获取集合中的下一个项。

使用基于迭代器设计模式的类的另一个非常有用示例涉及文件系统的处理。遍历循环文件系统以提供可用文件列表是编程中的常见主题。这种方式的使用示例包括提供可下载的一个文件列表以及对模块化基本代码动态应用插件。首先，创建一个基于迭代器设计模式的类。接下来，这个类访问文件系统并向调用代码返回文件信息，文件信息的形式可以是文件路径的一个字符串，也可以是一个文件对象。迭代器可能具有帮助确定返回文件类型的逻辑。例如，下载页面可能只希望提供 zip 文件下载，同时忽略目录中的其他元数据和文件。插件系统可能只查找迭代器能够透明处理的、以“.inc”为扩展名的文件。在所有情况下，迭代器都向外部代码流提供相同的公共方法来检索文件信息。正如第 2 章所介绍的，PHP 扩展库具有一个很大的基于迭代器设计模式的类集。要了解这些类和接口的更多信息，读者可以访问 <http://www.php.net/spl> 站点。

Web 上可用的信息量越来越大，并且聚集和表示数据集的要求在不断增长，编程人员正面对着新的挑战。现在，我们在一个累积视图中不必只遍历循环和提供一种实体类型，而是可以置入其他各种实体。针对这样的挑战，迭代器设计模式提供了一种实用工具。

假定某个社交站点希望在 Web 页面上公开显示用户状态的变化。该站点每分钟都会运行一个循环，以便从数据库中检索所有的状态变化。迭代器对象被创建用于遍历循环每个状态变化并提供有用的信息(如用户 ID 和状态内容)。

这里引入了一种更新状态的新方式，也就是将 txt 格式的消息添加入一个队列。现在，除了查询数据库之外，txt 消息队列也需要被检查和循环。我们可以创建新的迭代器来处理

txt 消息。当一个迭代器返回内容完成后，新的迭代器就像 txt 消息迭代器一样被创建，并且被提供给外部代码流。因为这种对象是在迭代器设计模式的基础上创建的以及存在通用的公共方法，所以不需要修改其他代码。

处理需要遍历的可计数数据时，最佳的解决方案是创建一个基于迭代器设计模式的对象。

## 11.2 UML

如图 11-1 所示，该 UML 图详细说明了一个使用迭代器设计模式的类设计。

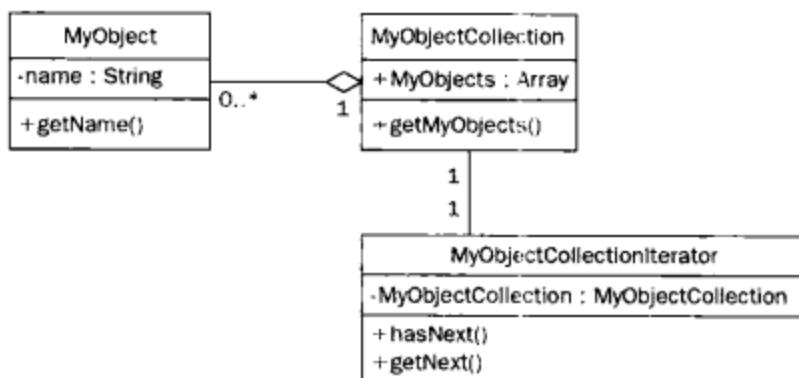


图 11-1

下面是对图 11-1 的说明：

- MyObject 是能够收集入可计数集合的基对象。MyObject 具有一个名为 name 的私有字符串，这个字符串用于代表特定对象的唯一性。公共方法 getName( ) 提供了通过私有名 name 检索对象名称的接口。
- MyObjectCollection 表示一个管理 MyObject 对象集合的类。MyObjects 数组容纳这些对象的集合。getMyObjects( ) 提供的逻辑被用于创建集合和在 MyObjects 数组中存储对象。
- MyObjectCollectionIterator 提供了对 MyObjectCollection 中存储的对象进行迭代的接口。这个类具有两个公共方法。其中，hasNext( ) 方法使调用程序知道 MyObjects 的 MyObjectCollection 集合中是否还有其他项；getNext( ) 方法会返回 MyObjectCollection 中数组内的下一个 MyObject 对象。

## 11.3 代码示例

示例 Web 站点的部分工作是显示特定艺术家或乐队的所有 CD。这些信息存储在一个 MySQL 数据库内。某些访问者可能希望根据乐队名搜索数据库，并且得到特定艺术家已

## 第II部分 参考内容

发布的所有 CD 的概述。这是实际应用迭代器设计模式的一个优秀示例。

首先，半标准 CD 类如下所示：

```
class CD
{
    public $band = '';
    public $title = '';
    public $trackList = array();

    public function __construct($band, $title)
    {
        $this->band = $band;
        $this->title = $title;
    }

    public function addTrack($track)
    {
        $this->trackList[] = $track;
    }
}
```

在这个 CD 类示例中，我们为乐队、标题和曲目列表使用了公共变量。构造函数创建了实例，同时在其内部指派了乐队和标题。`addTrack()` 函数接受 `$track` 变量，并且使用该变量以添加至曲目列表。

下面创建的是迭代器类。这个示例实现了 SPL 迭代器。因此，我们要求其具有 `current()`、`key()`、`rewind()`、`next()` 和 `valid()` 公共方法。

```
class CDSearchByBandIterator implements Iterator
{
    private $__CDs = array();
    private $__valid = FALSE;

    public function __construct($bandName)
    {
        $db = mysql_connect('localhost', 'user', 'pass');
        mysql_select_db('test');

        $sql = "select CD.id, CD.band, CD.title, tracks.tracknum, ";
        $sql = "tracks.title as tracktitle ";
        $sql.= "from CD left join tracks on CD.id=tracks.cid where band=' ";
        $sql.= mysql_real_escape_string($bandName);
        $sql.= "' order by tracks.tracknum";
```

```
$results = mysql_query($sql);

$cdID = 0;
$cd = NULL;

while ($result = mysql_fetch_array($results)) {
    if ($result['id'] !== $cdID) {
        if (!is_null($cd)) {
            $this->_CDs[] = $cd;
        }
        $cdID = $result['id'];
        $cd = new CD($result['band'], $result['title']);
    }

    $cd->addTrack($result['tracktitle']);
}

$this->_CDs[] = $cd;
}

public function next()
{
    $this->_valid = (next($this->_CDs) === FALSE) ? FALSE : TRUE;
}

public function rewind()
{
    $this->_valid = (reset($this->_CDs) === FALSE) ? FALSE : TRUE;
}

public function valid()
{
    return $this->_valid;
}

public function current()
{
    return current($this->_CDs);
}

public function key()
{
    return key($this->_CDs);
```

```

    }
}

```

与各示例中使用的大多数类相比，上面这个迭代器类相当冗长。不过，为了恰当地说明迭代器，尤其是 SPL 迭代器的实现，这是十分必要的。虽然代码很长，但是并不复杂。

`CDSearchByBandIterator` 类被设计为返回一个对象，使用 PHP 数组的某些函数就可以访问这个对象。注意到每个迭代器并不需要实现 SPL 迭代器，这是十分重要的。不过，在该示例中，我们很容易发现这个问题。

迭代器类具有两个私有变量，其中`$_CDs` 是一个包含 CD 对象集合的数组；`$_valid` 是数组访问函数使用的变量，通常，该变量说明了集合中是否存在要处理的可用对象。

`__construct()`方法接受一个名为`$bandName` 的参数。在实例化时，首先会创建一个与数据库的连接，随后创建的查询则返回乐队列与`$bandName` 匹配的所有 CD 和音轨的 MySQL 结果集。

CD 和音轨的存储是规格化的，这意味着检索结果集时存在相同 CD 的许多记录，并且具有相同的标题，但是音轨名是不同的。如果具体的关系是一个数据记录对应于一个 CD 对象，那么新编程人员极有可能不会创建迭代器对象。

因为希望处理 CD 对象，所以构造函数的下一部分就会遍历循环所有结果并创建单独的 CD 对象。只要存储在`$cdID` 变量中的 CD ID 发生变化，当前存储在`$cd` 变量中的 CD 对象就会被添加入内部的结果数组。随后，一个新实例会被创建。判断是否应当创建新 CD 对象之后，结果集的音轨标题会被添加至该对象。构建 `CDSearchByBandIterator` 之后，最终结果是一个完整的类，这个类具有一个 CD 对象数组，其中 CD 对象的乐队名与要搜索的乐队名匹配。

公共方法 `next()` 和 `rewind()` 的功能类似。首先，针对`$_CDs` 的内部实例执行使用 PHP 内建数组方法的匹配操作。如果这些函数无法对内部数组执行匹配操作，那么就会返回 `FALSE`。函数执行时会使用一个简单的条件句，将其结果与内部的`$_valid` 变量相比较并将其实添加至该变量。

公共的 `valid()` 函数相当简单。使用这个方法时，实现迭代器类是必需的。该方法的全部功能是提供内部变量`$_valid` 的值。

最后两个公共方法是 `current()` 和 `key()`。正如预料的一样，它们也针对内部的 CD 对象集合执行相应的 PHP 内建数组方法。

使用迭代器类时所采用的代码十分常见。`CDSearchByBandIterator` 的函数就像一个数组。遍历执行这些方法后，就会返回代码期望的 CD 对象。

```

$queryItem = 'Never Again';

```

```
$cds = new CDSearchByBandIterator($queryItem);

print '<h1>Found the Following CDs</h1>';
print '<table> <tr> <th> Band </th> <th> Title </th> <th> Num Tracks </th> </tr>';
foreach ($cds as $cd) {
    print "<tr><td>{$cd->band}</td><td>{$cd->title}</td><td>";
    print count($cd->trackList). '</td></tr>';
}
print '</table>';
```

处理可计数和可遍历数据的集合时，最佳的做法是创建基于迭代器设计模式的对象。



# 第12章

## 中介者模式

复杂的代码不会受到任何人的赞赏。大量的耦合、易管理的相关性和整体式代码流能够很好地证明编程人员出众的能力、辛勤的工作和高超的智慧。我无法相信复杂的代码会得到恶评。现在，无论是初级的还是经验丰富的编程人员，都应当忘记所有的讽刺并尽快振作起来。在很多情况下，为了在处理软件应用程序时保持清醒的头脑，编程人员必须保持这样的心态。增加的功能、扩张的范围以及相关人员众多都会导致代码库的交织。术语“面向对象(object oriented)”似乎已经消失，这种不可分开的代码一度是非常流行、快速和模块化的系统。不过，系统内部的类似对象相互的查找以及彼此应用的更新过多，并且耦合通常过于紧密。但是，我们还是有希望解决这种问题的。在许多并非紧密耦合的类似对象需要接受改变时，中介者设计模式就可以提供帮助。

### 名称：中介者

中介者设计模式用于开发一个对象，这个对象能够在类似对象相互之间不直接交互的情况下传送或调解对这些对象的集合的修改。

### 12.1 问题与解决方案

当对象的耦合过于紧密时，面向对象编程的优点就会显现出来。虽然仍旧在处理对象，但是这种样式开始向程序化转换。基本代码变得整体化且较为笨重。在能够应用解决方案之前，我们需要深入研究这个特定的问题。理解根本的原因是十分重要的。中介者设计模式解决的相同问题会缓慢地蔓延至具体代码的其他实例中。

在对象并未被专门设计为在开始就意识到相互之间的关系时，问题就会接踵而至。现在，对象时常被创建用于处理子对象或集合。这是一个非常完美的体系结构选项。然而，当可交换的对象或不基于相同架构的对象开始具有相互依赖性时，问题也就随之而来——往往是在对象被更新或对象接口发生变化的时候，或者是需要对某个对象应用更新的时候。

## 第II部分 参考内容

为此，我们似乎只要添加一个新方法，以便使用相同的信息更新类似的对象。但是，如果添加这个方法，指定对象就会理解和联系不需耦合的另一个对象。最直接的后果是由另一对象公共方法的改变造成的。此时，不仅需要修改另一个对象，而且也需要更新与之无关的指定对象的方法。

基于中介者设计模式的对象在这些有关系但不耦合的对象之间提供了一个非常必要的通信中心。一个类似但是无关的对象会受到某个改变的影响，它会向中介者对象提及这个改变。通过将该改变应用于其他所有对象，中介者对象随后进行调解以促成这些对象能够接受该改变。最开始发生改变的对象并不知道有多少其他的对象也进行了相同的改变。除了应当应用改变之外，其他类似的对象并不知道更新的来源。

通过雇员和老板就可以阐明这种行为在现实生活中的实例。某位雇员决定在家自学一项新的综合技能。在学习新的综合技能时，他并未联系其他任何雇员。学习完成后，这位雇员向老板告知了自己所掌握的新技能。老板觉得这种做法很好，并且通告其他所有雇员都应当学习这项新技能。随后，其他雇员就按照自己的方式开始学习新的技能。

通过销售系统，我们也可以了解中介者设计模式的具体表现。假设销售系统被用于出售某个音乐店的吉他。通过创建一个吉他对象就可以访问库存的每把吉他。在销售管理界面上，卖主可以选择对特定的吉他进行打折。因为某些供应商可能想进行特别的推销活动，所以卖主也能够通过选择一个复选标记对特定品牌的所有吉他进行打折。如果选择了复选项，那么更新就能够被正常地应用。不过，中介者(也就是卖主)必须被告知这个改变(也就是举办的推销活动)以及应当对其他吉他对象应用该改变。

在源对象的改变应当被传递给其他有联系但不耦合的对象时，就应当使用基于中介者设计模式的对象来管理这些更新。

## 12.2 UML

如图 12-1 所示，该 UML 图详细说明了一个使用中介者设计模式的类设计。

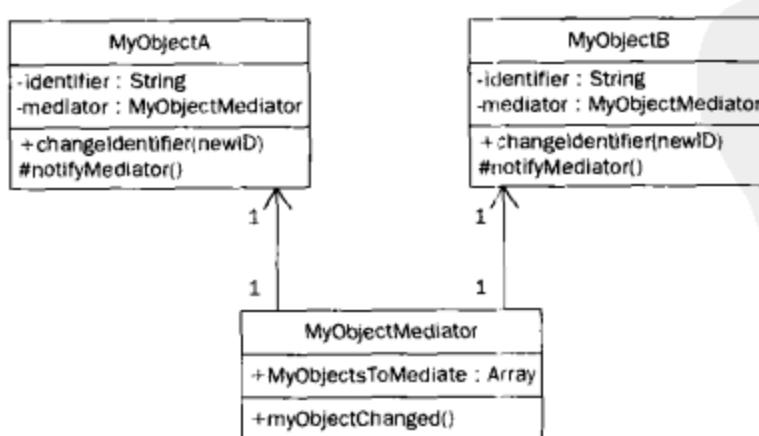


图 12-1

下面是对图 12-1 的说明：

- 图 12-1 中存在两个相似的类：MyObjectA 和 MyObjectB。这两个类显现的外形是相同的。它们的差异可能是用私有变量 `identifier` 表示的标识符。此外，所有函数都是相似的。
- 在对象创建期间，`MyObjectMediator` 的实例被存储在内部。随后，如果通过调用公共方法 `changeIdentifier()` 请求针对某个对象的变化，那么参数 `newID` 通过更新私有标识符字符串就能够被应用于该对象。接下来，调用保护方法 `notifyMediator()` 接可以对其余对象应用中介。
- `MyObjectMediator` 是一系列对象的中心所在。这些对象存储在数组 `MyObjectsToMediate` 中。`MyObjectsToMediate` 接到通知时就会执行 `myObjectChanged()` 方法，该方法负责解析数组 `MyObjectsToMediate` 并对其他所有对象应用指定的改变。

## 12.3 代码示例

示例 Web 站点不仅允许乐队进入和管理他们的音乐合辑，而且还允许乐队更新他们的配置文件、修改乐队相关信息以及更新其 CD 信息。现在，艺术家可上传 MP3 集合并从 Web 站点上撤下 CD。因此，Web 站点需要保持相对应的 CD 和 MP3 彼此同步。

Web 站点的最初版本允许乐队从配置文件页面或单独的 CD 本身修改其乐队名称。如下所示，CD 对象具有一个接受乐队变化以及在数据库中进行更新的方法：

```
class CD
{
    public $band = '';
    public $title = '';

    public function save()
    {
        //stub - writes data back to database - use this to verify
        var_dump($this);
    }

    public function changeBandName($newName)
    {
        $this->band = $newName;
        $this->save();
    }
}
```

上面这个简单的类只是说明了 CD 对象可以具有乐队名和标题。接下来，函数 changeBandName( )接受一个新的乐队名参数。这个函数将该参数插入指定对象并调用 save( )方法。出于演示说明的目的，所以这里没有进一步说明 save( )方法。读者可以自己验证已完成的相应变化。

为了添加 MP3 归档文件，就需要创建另一个类似的对象来处理归档文件。艺术家必须也能够在 MP3 归档文件页面上修改其乐队名。同样，在与之关联的 CD 中也必须能够修改乐队名。

现在应当使用中介者设计模式。首先，为了使用该模式，必须修改 CD 类。随后，创建与 CD 类相似的 MP3 归档类。

```
class CD
{
    public $band = '';
    public $title = '';
    protected $_mediator;

    public function __construct($mediator = null)
    {
        $this->_mediator = $mediator;
    }

    public function save()
    {
        //stub - writes data back to database - use this to verify
        var_dump($this);
    }

    public function changeBandName($newName)
    {
        if (!is_null($this->_mediator)) {
            $this->_mediator-> change($this, array('band'=>$newName));
        }
        $this->band = $newName;
        $this->save();
    }
}

class MP3Archive
{
    public $band = '';
    public $title = '';
```

```

protected $_mediator;

public function __construct($mediator = null)
{
    $this->_mediator = $mediator;
}

public function save()
{
    //stub - writes data back to database - use this to verify
    var_dump($this);
}

public function changeBandName($newName)
{
    if (!is_null($this->_mediator)) {
        $this->_mediator->change($this, array('band'=>$newName));
    }
    $this->band = $newName;
    $this->save();
}
}

```

对 CD 对象的第一个改变被添加入名为 `$_mediator` 的保护变量，该变量用于存储中介对象的实例。MP3Archive 类中添加了构造函数。创建 CD 类的一个实例时，新的中介者对象应当被传递至这个类内。不过需要注意的是，`$_mediator` 变量的默认值为 `null`。这不仅允许我们创建用于只读功能的、实例中没有中介者的对象，而且也确保在使用中介者对象更新其他类时不会创建无限的循环。接下来，`changeBandName()` 方法会被修改。该方法用于查看中介者对象是否存在且不为空。如果存在且不为空，那么就会调用中介者对象的 `change()` 方法，从而传递入自身的实例并修改指定项的键控数组。

上述操作在对中介者对象本身应用更新之前发生。对于中介者来说，在修改指定项前获取其快照是十分重要的。

MP3Archive 对象几乎与 CD 对象完全相同。

如下所示，随后需要创建中介者类：

```

class MusicContainerMediator
{
    protected $_containers = array();

    public function __construct()
    {

```

## 第II部分 参考内容

---

```

    $this->_containers[] = 'CD';
    $this->_containers[] = 'MP3Archive';
}

public function change($originalObject, $newValue)
{
    $title = $originalObject->title;
    $band = $originalObject->band;

    foreach ($this->_containers as $container) {
        if (!$changedObject instanceof $container) {
            $object = new $container;
            $object->title = $title;
            $object->band = $band;

            foreach ($newValue as $key=>$val) {
                $object->$key = $val;
            }

            $object->save();
        }
    }
}

```

中介者对象知道其将要中介调解的所有音乐容器(Music Container)。构造函数用于构建将要中介调解的对象的内部数组。如果今后创建了新的音乐容器，那么中介者类中所需的唯一变化就是在保护数组\$\_containers 内添加一个新的元素。

更改方法将接受原始对象以及将要添加的新值。首先，原始对象中的标题和乐队名会被检索出来。接着，所有音乐容器会被循环遍历。如果\$originalObject 不是这些容器的实例，那么就会创建相应的容器。将指定容器与原始对象进行比较的原因是为了减少重复。完成对中介者的操作后，变化就会应用于原始对象。此时，我们不需要创建和更改原始对象的副本，从而跳过了循环。

如果指定容器被创建为一个新的对象，那么就要根据原始对象设置标题和乐队名。最后，\$newValue 的任何变化都会被循环遍历，然后应用于这个新的对象，接着该对象才会被保存。

上述整个处理过程之后，原始对象就会应用自己的更新并自动保存。

使用新的中介者对象，所采用的代码十分简单，如下所示：

```
$titleFromDB = 'Waste of a Rib';
```

```
$bandFromDB = 'Never Again';

$mediator = new MusicContainerMediator();
$cd = new CD($mediator);
$cd->title = $titleFromDB;
$cd->band = $bandFromDB;

$cd->changeBandName('Maybe Once More');
```

处理具有类似属性并且属性需要保持同步的非耦合对象时，最佳的做法是使用基于中介者设计模式的对象。





# 13 章

## 观察者模式

致力于开放源代码软件的公司是很了不起的，这种公司在十年以前几乎是不可能存在的。如今，许多成功的公司依赖于他们帮助创建的相同开放源代码软件而生存发展。这些公司提供一种能够运转的企业模型。在这种概念最初出现时，可能会遇到下列障碍：

- 这些公司希望继续提供开放源代码软件，而不是用代码库取而代之。
- 如果继续使用特定的代码库，那么这些公司可能会考虑许可限制的问题。
- 这些公司的目标是保持社会公众的密切关系。他们希望提供一个载体，现有的强势集团能够通过这个载体保持密切的联系。

这些公司拥有大量精通代码库的人才。因此，他们选择在应用程序中添加额外功能的方式来产生收益。不过，每当添加新的功能性之后，现有的软件用户不可能被强迫再次更新软件包。但是，这些公司肯定也不希望提供这种软件的分支版本。最终，形成的统一意见是：创建一个插件系统。通过这种方式，公司可以为了收益而继续开发自定义的专有软件，同时鼓励其他团体的成员借助该途径提供更多的功能性。由于只是插件系统，因此终端用户能够决定希望在核心软件包内添加的具体功能。在需要向代码库内添加新功能性而并不知道核心对象的时候，我们可以使用观察者设计模式。

### 名称：观察者

观察者设计模式能够更便利地创建查看目标对象状态的对象，并且提供与核心对象非耦合的指定功能性。

### 13.1 问题与解决方案

插件系统是最显而易见的观察者设计模式示例。因为我参与的应用程序的有效部分是由开发人员核心团队管理的，所以我并不清楚太多创建基于插件的代码库的道理。事实上，

我极不情愿深入地探讨观察者模式。但是，观察者设计模式在应用中最常见的示例只有插件系统。其实，插件系统还有其他许多实现方式。

观察者设计模式的核心概念解决了一个十分常见的问题。为了保持切实有效，软件应用会不断要求添加新功能。然而，如何在不必完全重构某些核心对象的情况下向应用程序添加额外的功能性？观察者设计模式设计了在最初创建对象时解决这个问题的蓝图。

首先，理解观察者设计模式基于状态变化十分重要。状态变化其实就是一个对象的变换。它既可以非常简单，如调用更改对象属性的内部方法；也可以是非常复杂的过程（如一个完整的逻辑路径），例如在销售某些货物之后基于另一个对象平衡某个库存对象。无论是哪一种情况，状态变化都针对对象发生。

基于观察者设计模式的核心对象负责向被指派观察和理解核心对象的其他类传达这些状态变化。根据其被设计的状态通信处理方式，其他观察类既可以采取外部动作，也可以更改核心对象。

设计作为核心对象的可观察类时，极为频繁的通信是十分重要的。每当发生值得注意的状态变化时，可观察类应当通知侦听其变化的任何对象，同时还应当告知所发生变化的类型。虽然现在可能还没有任何“应关注”的观察类，但是软件应用的额外迭代可能添加查找特定状态变化的新功能性。观察类应当被配置为只在匹配其动作标准时才发出通知。

另外需要记住的是，生成所有通知的可观察类并不知道或关心是否会对执行操作。这种松散耦合方法允许可观察类只是从头至尾执行其逻辑过程。在整体非耦合的体系结构中，可观察类不知道观察类执行的任何操作。

前面曾经提到过，插件系统是编程人员想到的第一个观察者设计模式示例。但是，正如第1章所介绍的那样，PHP支持可供企业使用的引擎。考虑到其规模，消息排队应当是较大系统中的一个大部分。在PHP中开发基于观察者设计模式的类还有助于易化体系结构。

在实际应用中，商业Web站点是一个常见的观察者设计模式示例。当访问者购买某委托客户的商品时，销售操作会应用于库存清单和存货。销售进程可以实现等待成功交易的观察者设计模式。完成一个成功的交易时，不仅会应用信用卡的主要处理和存货清单的调整，而且销售对象也要向所有观察者通知被销售的商品项。委托客户可以选择在商品被销售的同时立即接收到电子邮件。这些客户可能希望实际配送商品或了解当前的存货列表。

“交易完成”消息被发送给某个观察类，该观察类接收信息并向客户发送电子邮件。这不仅是处理上述活动的极佳方式，而且还考虑了通知系统的可扩展性。也许在不久以后，委托客户可能希望采用txt消息方式进行通知。这种方式十分简单，此时不必修改销售系统，而只需添加另一个观察类。

实际应用中的另一个观察者设计模式示例涉及缓存器的构建。针对每个新的博客发

布, RSS 源上都会出现 RSS 读者占用的一个入口。对于编程新手来说,为了构建这样的 RSS 源,他们会首先创建一个在每个载入页面上查找顶部入口的脚本,随后在某个 RSS 文档中格式化其结果。如果采用这样的设计,那么高流量站点会受到损害。不过事实正好相反,编程人员应当构建静态的 RSS 文档,这些文档也包含有效期信息。后一种方式能够创建良好的、为 RSS 读者提供数据的可缓存系统。为了更新缓存的 RSS 资源,我们可以使用一个观察类。成功的博客发布应当向所有观察者通知其已完成。观察类会接受这个信息并创建新的 RSS 资源。接下来,缓存器会被更新。这样可以保持缓存器的内容是最新的,同时也减少了服务器的负载。此时,我们不必定期生成缓存器,即使没有变化,缓存器仍然只在观察到需要更新的情况时被更新。

为软件添加由某个动作或状态变化激活的、但是松散耦合的新功能时,就应当创建基于观察者设计模式的对象。

## 13.2 UML

如图 13-1 所示,该 UML 图详细说明了一个使用中介者设计模式的类设计。

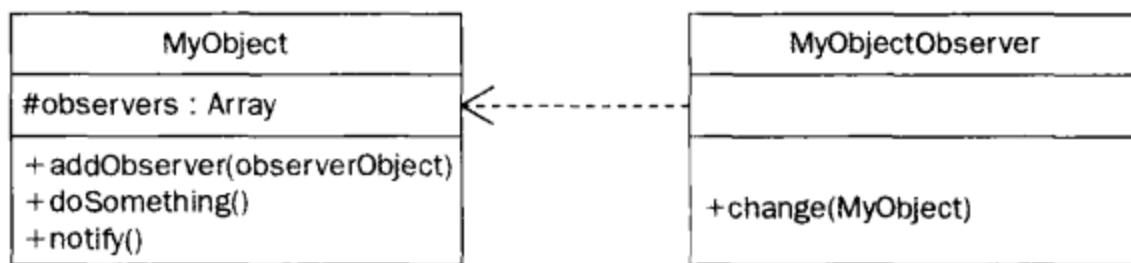


图 13-1

下面是对图 13-1 的说明:

- MyObject 是可观察对象,它包含一个名为 `observers` 的观察者保护数组。公共方法 `addObserver()` 接受一个观察者的实例并将其存储在观察者数组内。
- `doSomething()` 公共方法会被调用,这个方法对 MyObject 应用状态变化。随后, `notify()` 公共方法会被调用,这个方法可遍历循环观察者数组。
- MyObjectObserver 具有一个名为 `change()` 的公共方法,该方法接受 MyObject 的一个实例。这个特定的观察者接下来会对 MyObject 的内容执行某些操作。当在观察者数组中找到特定的观察者时,MyObject 的 `notify()` 方法会直接调用 `change()` 方法。

### 13.3 代码示例

本书的音乐 Web 站点具有站点访问者可用的某些社交类型功能。其中，集成的最新功能是一个活动流(activity stream)，它能够在主页上显示最近的购买情况。这种功能主要是希望人们单击最近出售的商品，从而使其可能购买相同的商品。

如下所示，将 CD 销售情况置入活动流的第一个步骤是创建一个基于观察者设计模式的 CD 对象：

```
class CD
{
    public $title = '';
    public $band = '';
    protected $_observers = array();

    public function __construct($title, $band)
    {
        $this->title = $title;
        $this->band = $band;
    }

    public function attachObserver($type, $observer)
    {
        $this->_observers[$type][] = $observer;
    }

    public function notifyObserver($type)
    {
        if (isset($this->_observers[$type])) {
            foreach ($this->_observers[$type] as $observer) {
                $observer->update($this);
            }
        }
    }

    public function buy()
    {
        //stub actions of buying
        $this->notifyObserver('purchased');
    }
}
```

构造函数只是将标题和乐队名分别指派给公共变量 \$title 和 \$band。

attachObserver() 公共方法接受两个参数。第一个参数是\$type。因为 CD 对象可能存在许多状态变化类型，所以还要通过\$type 参数进一步指定通知类型。第二个参数是将要添加入保护数组 \$\_observers 的观察者类。注意，\$type 变量决定了数组第一层的键。随后，\$type 指定的所有观察者类型都按顺序被添加入特定的层次。

公共方法 notifyObserver() 接受一个名为\$type 的参数，该参数被用于获取保护数组 \$\_observers 的有效键，从而访问相应的每个观察者。在这个方法中执行公共方法 update() 时，需要将指定对象的当前实例作为参数。

buy() 方法只是简单地注明购买 CD 的过程。一旦执行了该方法，就会在类型设置为“purchased”的情况下调用 notifyObserver() 方法。

如下所示，随后要求某个观察者向活动流发布购买信息：

```
class buyCDNotifyStreamObserver
{
    public function update(CD $cd)
    {
        $activity = "The CD named {$cd->title} by ";
        $activity .= "{$cd->band} was just purchased. ";
        activityStream::addNewItem($activity);
    }
}
```

上面这个类具有一个名为 update() 的公共方法，该方法接受 CD 实例。update() 方法只是从 CD 实例中收集信息并构建准备向活动流发布的内容。

最后详细说明两个问题：activityStream 类和使用观察者启动 CD 销售的方式。

```
class activityStream
{
    public static function addNewItem($item)
    {
        //stub functions
        print $item;
    }
}

$title = 'Waste of a Rib';
$band = 'Never Again';
$cd = new CD($title, $band);

$observer = new buyCDNotifyStreamObserver();
```

## 第II部分 参考内容

```
$cd->attachObserver('purchased', $observer);  
$cd->buy();
```

activityStream 类包含公共方法 addNewItem( )，该方法只是将\$item 参数打印至屏幕。在完整的示例中，这个方法还可能将\$item 参数写至数据库或可缓存的 XML 文件。

如下所示，启动销售的代码相当简单：

- (1) 使用\$title 和\$band 创建一个新的 CD 对象。
- (2) 实例化 buyCDNotifyStreamObserver 的一个新观察者实例。使用 attachObserver( ) 方法将该实例添加至 CD 对象。这个实例被规定为观察者的 purchased 类型。
- (3) 指定 CD 对象被购买。

在创建其核心功能性可能包含可观察状态变化的对象时，最佳的做法是基于观察者设计模式创建与目标对象进行交互的其他类。

# 第14章

## 原型模式

在采用面向对象的语言进行编程时，我一再想起一件很特别的事情。这件事情比其他所有事情都更为重要，并且在大多数书籍和文章中被经常提及。面向对象的编程人员到底应当牢记什么？答案就是保持对象小型化、模块化、非耦合以及流线型。不过，在下列两种情况下，我们可能发现与之相抵触的对象：

- 从其他人那里继承代码时。有时候，用户可能没有时间或预算将这些代码重构为更可接受的代码。
- 代码无法进一步被削减时。此时，代码相当简洁。在指定对象只能被唯一创建并且在一个代码流中使用多次时，就会带来一定的开销。

在需要经常创建资源集中型对象的情况下，原型模式可以提供更快速执行的途径。

### 名称：原型

原型设计模式创建对象的方式是复制和克隆初始对象或原型，这种方式比创建新实例更为有效。

### 14.1 问题与解决方案

应用程序变得更为高级，核心对象似乎同样如此。虽然创建标准对象的方式仍然是完全模块化的，但是构建适当对象的途径却可能增加十倍。此外，代码库并不像编程人员希望的那样是完全流线型的。无论是哪一种情况，某些对象的创建需要更大的开销。构造这些对象需要更多的时间，或者耗用更多的资金，这都是应当尽力避免的。解决这种问题的好办法似乎是更为简单地创建一个新类型的对象，这个对象模仿核心对象的行为，并且不具有任何复杂性。如果创建这些对象的成本显然过高，那么可以采用下列两种方式。第一种方式是重构。前面已经暗示过，重构有时不切实际甚至不可能。另一种方式涉及修改对

象，从而使这些对象在采用原型对象模式的情况下能够被复制。

原型对象模式提供了减少创建复杂对象所需开销的方式。这种方式并不是去除对象本身存在的复杂性，而是创建一个通过复制来保持这些属性的接口。原型对象模式十分有用，尤其复杂对象在整个代码流中都不会改变其状态时更是如此。也就是说，这些对象创建时所拥有的初始状态在整个会话期间始终都是相同的。不过，每个单独的副本都能够具有不同和独特的特征。某个对象向其自身的受保护属性提交很大的 XML 文档能够极好地说明上述问题。在具体代码中，某些属性可能被修改，但是在大多数节点都保持相同。

原型对象模式的实现级别范围从简单的创建复制品一直到详细的克隆方法。读者可能会发现，某些对象的初始状态能够容易地变换为该对象的一个独特版本。对于其他对象来说，实现较为困难。假定某个对象具有可以作为基本属性的独特标识符(例如全局唯一标识符(Globally Unique Identifier, GUID))。在复制过程中，我们不需要修改其他任何属性，但是需要获得新的 GUID。

为了理解使用原型设计模式的实际好处，让我们以在人口稀少地区某工厂工作的一群雇员为例。当工作量增加时，就需要新的雇员。为了拥有新的工人，工厂需要培训新招募的人员。技能提升过程的花费相当大。这个过程会积压越来越多的工作量。为什么不能更简单地将新工人复制为某位辛勤工作的雇员？这恰好就是原型设计模式要实现的详细设计。杰出的工人是所有新雇员应当效仿的原型。

公共图书馆书籍目录系统是一个能够很好说明原型设计模式的技术示例。具体的应用程序被设计用于拥有 50 个图书馆的系统中的每一个图书馆。每个软件应用都采用相同的方式进行部署。应用程序的动态部分是具体图书馆的名称和位置，这些信息在部署后被存储在文件系统中的一个 XML 文件内。因为图书馆共享组提供相互出借功能，所以都必须知道哪些书籍被登记借出。最合理的方法是将这些信息存储在一个中央存储系统内(如 MySQL 数据库)。当顾客在系统中查看到某本书时，他们可以选择是否借书，甚至可以选择借出多本书。在顾客借走这些书籍之前，系统需要使用出借信息更新中央数据库。首先，系统会创建一个原型对象，该对象从文件系统中获取图书馆信息。随后，针对队列中出借的每本书籍，都会创建一个初始原型书籍的副本。创建的副本仍然具有最初从 XML 文件中检索到的信息。接着，副本接收客户信息，例如书籍标题或 ISBN。基于原型设计模式的书籍对象保存文件系统的查询，以便从 XML 文件中检索这些信息。我们可以认定信息在登记借出期间不会发生变化，因此能够应用于类似缓存器的原型对象。

理解原型设计模式优点的另一种方式较为抽象。假设某个对象具有数目众多的设置器与读取器。构建这个对象需要执行极为小心细致的步骤。为了完整地创建该对象，需要以特定的顺序调用许多不同的设置器。在需要新的对象时，我们不必完全执行相同的一系列步骤，而是最好构建一个基于原型设计模式的初始对象。随后，第一个实例仍然保持复杂

的构成成分。实际上成为主代码流中逻辑部分的对象只是初始原型的复制品，从而减少了许多需要执行的复杂过程。

在需要多次创建构建成本较高的对象时，最有效的做法是创建基于原型设计模式的对象。

## 14.2 UML

如图 14-1 所示，该 UML 图详细说明了一个使用解释器设计模式的类设计。

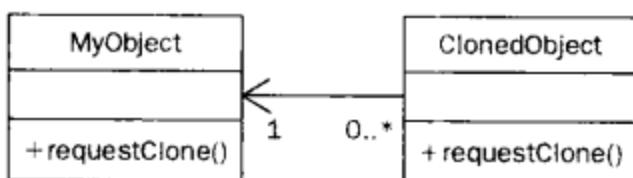


图 14-1

下面是对图 14-1 的说明：

- **MyObject** 类组合使用了原型设计模式。这个类具有名为 `requestClone()` 的公共方法，该方法用于生成 **MyObject** 实例的一个副本。
- **ClonedObject** 对象表示 **MyObject** 的复制实例。需要注意的是，因为该对象实际是一个复制品，所以它也具有 `requestClone()` 方法。
- 通过调用 **MyObject** 的 `requestClone()` 方法，可以创建 **ClonedObject** 的许多实例。

## 14.3 代码示例

音乐销售 Web 站点允许登录站点的艺术家创建许多乐队的音乐“合辑”。目前，这种功能性还限制为所有有效的音轨只能来自一个乐队。为了开始创建 CD “合辑”，我们可以采用多种途径。其中，在访问者查看某乐队的 CD 页面的时候可以使用最通用的选项。CD 页面存在的某个链接能够启动构建新 CD “合辑”的进程，该进程会发送一个 ID，这个 ID 对应于乐队的特定 CD。

上面这个进程的第一个构建代码块是 CD 类。通常，为了构建 CD 对象，需要从数据库中检索出与被请求 ID 匹配的具体信息：

```

class CD
{
    public $band = '';
  
```

## 第II部分 参考内容

```

public $title = '' ;
public $trackList = array();
public function __construct($id)
{
    $handle = mysql_connect('localhost', 'user', 'pass');
    mysql_select_db('CD', $handle);

    $query = "select band, title, from CDs where id={$id}";

    $results = mysql_query($query, $handle);

    if ($row = mysql_fetch_assoc($results)) {
        $this->band = $row['band'];
        $this->title = $row['title'];
    }
}

public function buy()
{
    //cd buying magic here
    var_dump($this);
}
}

```

这个类具有标准的公共属性\$band、\$title 和\$trackList。

构造函数接受形式为\$id 参数的 ID，并且针对数据库执行查询。当发现指定的 ID 时，相对应的乐队和标题就会被分别指派给公共属性\$band 和\$title。

此外，CD 类还增加了一个名为 buy( )的函数。这个很短的方法只是为了显示用于本示例的 CD 实例。在最后的产品代码中，该方法可以处理 CD 对象并提供用于购买。

下一个需要创建的类代表混合 CD 实体。如下所示，这种特定的对象利用了 PHP 的克隆能力：

```

class MixtapeCD extends CD
{
    public function __clone()
    {
        $this->title = 'Mixtape';
    }
}

```

因为 MixtapeCD 实际只是一种特殊化的 CD，所以它扩展了 CD 对象。

执行 PHP 的 clone 命令时，就会对指定的对象执行奇妙的 `_clone()` 方法。在 MixtapeCD 对象中，初始 CD 的标题属性会被重写。这个 MixtapeCD 对象不再对应于一个乐队和标题的 CD 组合。此时，该对象仍然关联 \$band，但是具有新的标题 Mixtape。

下面这个示例展示了某位用户基于指定乐队定制两个混合标题的情况：

```
$externalPurchaseInfoBandID = 12;
$bandMixProto = new MixtapeCD($externalPurchaseInfoBandID);

$externalPurchaseInfo = array();
$externalPurchaseInfo[] = array('brrr', 'goodbye');
$externalPurchaseInfo[] = array('what it means', 'brrr');

foreach ($externalPurchaseInfo as $mixed) {
    $cd = clone $bandMixProto;
    $cd->trackList = $mixed;
    $cd->buy();
}
```

`$bandMixProto` 对象是根据 `MixtapeCD` 的新实例创建的。传递入该对象的参数 `$externalPurchaseInfoBandID` 被用于实际由 `CD` 类构造函数执行的查询。

一旦创建了原型，就能够循环遍历用于特定访问者的 CD 合辑的音轨列表。对于 `foreach()` 循环的每个实例来说，`$cd` 都被指派给原始 `$bandMixProto` 的一个新副本。接下来，特定的曲目列表会被添加入指定的对象。因为使用了克隆技术，所以每个新的循环都不需要针对数据库的新查询。克隆对象已经存储了所有信息，并且可以采用与原始对象相同的方式使用这些信息。最后，通过执行公共方法 `buy()` 就可以购买指定的 `$cd` 对象。

处理创建成本较高和新实例的初始信息保持相对不变的对象时，最佳的做法是使用基于原型设计模式创建的复制类。



# 第 15 章

## 代理模式

最初涉足 AJAX 领域时，我既体会过成功的喜悦，也经历过挫折。我在无意中发现，这种新技术使我的设计和创建 Web 页面的方式发生了彻底的改变。从 Web 页面到 Web 应用程序都能够使用这个工具，这非常令人兴奋。我首先尝试做的是将天气和相应的衣着混搭在一起。我希望知道，如果要参加户外活动，那么是否需要准备一件夹克。我最初设计具体项目在服务器端使用 PHP 读取 XML 源。一旦知道能够完成这样的工作，我就希望将指定的功能性迁移至浏览器端，从而能够利用 AJAX 对象的美观效果。正如读者可能知道的那样，我很快就遇到一个困难：使用 AJAX 请求无法加载 XML 源。XMLHttpRequest 对象不能载入跨站点请求。起初，我并不清楚如何实现检索远程站点中的这些数据。最后，才偶然发现了解决方案，它涉及设计一个代理 PHP 对象，从而检索远程数据并提供给本地。在处理要求特别注意或拥有特别访问需求的对象时，代理设计模式能够有所帮助。

### 名称：代理

代理设计模式构建了透明置于两个不同对象之内的一个对象，从而能够截取或代理这两个对象间的通信或访问。

### 15.1 问题与解决方案

人们总是喋喋不休地反复说要使用代理。无论何时测试某个安全开发，人们总是希望使用代理在测试器和目标系统之间提供一个匿名层。将代理设计为接受和重新发送所接收的数据。在这种情况下，人们希望通过在中间使用代理来遮敝测试的起源。

不过，实际应用中仍然存在其他更为直观和常用的代理实现。某些公司的网络会置入像 Web 资源一样工作的代理服务器。内部的网络浏览器直接与代理服务器通信。代理服务器随后查看其缓存器，从而确定在其本地磁盘是否具有指定资源。如果具有指定的资源，

那么代理服务器就会做出应答。否则，代理服务器就透明地转发请求。其他网络通过安装内容过滤代理来执行进一步的步骤。内容过滤代理能够监视违反公司策略的 Web 站点流量。如果代理发现雇员试图查看被限制访问的 Web 站点，那么就会中断这个操作，其他所有请求则被透明地发送。对于浏览者来说，在发现被限制访问某些网页之前，他们并不知道自己通过代理连接入网络。

代理设计模式构造了一个对象，该对象可以截取其他两个或更多对象之间的通信。Proxy 对象的外部接口与其所属的类完全相同。调用对象无法辨别原始对象和 Proxy 对象之间的差异。代理设计模式的核心存在于下列 3 个目标中：提供访问、只按需引起开销，以及持续数据存储。

准许对特定资源的访问是一项非常复杂的工作。这项工作可能需要应用各种不同的业务规则、需要验证或检索凭证、需要验证和确认身份甚至位置。就良好的面向对象设计而言，基对象应当不关心被谁访问，它应当只是一个完成任务的编程容器。这也正是适合使用代理设计模式的原因。Proxy 对象能够介入并基于其他外部属性确定访问限制。例如，某个 Proxy 对象可能规定当前用户并不具有执行指定命令或拒绝执行的适当权限。不过，只有证书验证不限制访问条件。在分布和分段的网络中，特定资源可能只存在当前对象集能够到达的区域之外。Proxy 对象能够截取这些请求，并且充当了连接远程系统的桥梁。

某些对象的创建需要大量开销。开销可能来自对某个数据库或者只是一大块内存占用的许多远程调用。如果能够不立即实例化这种对象，那么就可以创建一个基于代理设计模式的对象。代理对象只在需要的时候才请求创建开销很大的对象。如此一来，直至整个代码执行结束都可能不必真正地创建这种对象。平时，当释放其他对象的内存空间后再创建较大的对象是更为合理的。

最后，Proxy 对象能够充当持续数据存储的一种方法。只要制定了缓存方法，其基础都是一个使用代理设计模式创建的对象。这个代理对象会判断先前请求的对象是否被存储。如果请求对象已存储，那么就提供这种资源。否则代理对象就会透明地转发初始请求。

与外部设备的 Web 接口是一个特别常见的代理设计模式实例。让我们通过为远程查看而控制分线盒与 TV 的 Web 应用程序为例。从 Web 页面接收到切换到第 5 频道的请求时，指定的 Proxy 对象会截取这个请求。根据 TV 的类型，Proxy 对象可能向远程接收器发送序列“0005 < enter >”或简单的“5 < enter >”。如果不存在截取通信的代理，那么 TV 会因为被发送的信号“5”没有采用期望的格式而忽视该信号。

代理设计模式的另一个应用示例是对发布信息的访问限制。某个社交网络站点允许用户创建简介页面。用户可以使该页面保持保密状态或公开状态。这个系统经过完全测试，并且运转情况良好。所有人都不想对代码进行任何改动。不久以后，公司内部传达了一个新的规定：不允许公司任何雇员创建和公开发布简介。编程人员的任务是创建一个阻止雇

员公开发布公司简介的系统。他们决定创建截取发布简介请求的 `Proxy` 对象。如果具体用户是公司的雇员，那么请求就会被中断。否则，会转发请求。发布对象完全不知道信息已经过代理的过滤。

在需要过滤或增强两个对象之间的通信时，我们可以通过基于代理设计模式的对象来完成这个要求。

## 15.2 UML

如图 15-1 所示，该 UML 图详细说明了一个使用代理设计模式的类设计。

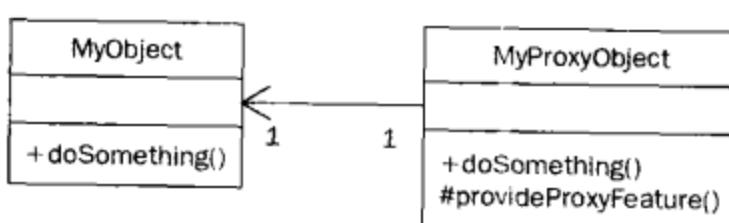


图 15-1

下面是对图 15-1 的说明：

- `MyObject` 是与客户正常交互的基对象，它具有一个名为 `doSomething()` 的公共方法。
- `MyProxyObject` 是用于 `MyObject` 的代理，它包含一个名为 `doSomething()` 的公共方法和一个名为 `provideProxyFeature()` 的受保护方法。
- 当 `MyProxyObject` 代替 `MyObject` 时，`doSomething()` 方法仍然可以被调用。在可能向 `MyObject` 的 `doSomething()` 方法转发请求之前，`MyProxyObject` 会执行 `provideProxyFeature()` 方法。

## 15.3 代码示例

因为示例 Web 站点非常不错，所以 CD 店的销售不断增长。扩展已势在必行，但是 Web 站点每天都要进行正常的销售。良好的工作代码实际十分简单。首先，我们要创建表示访问者能够购买的 CD 的对象。

```

class CD
{
    protected $_title = '';
    protected $_band = '';
}
  
```

## 第II部分 参考内容

```

protected $_handle = null;

public function __construct($title, $band)
{
    $this->_title = $title;
    $this->_band = $band;
}

public function buy()
{
    $this->_connect();

    $query = "update CDs set bought=1 where band=''";
    $query .= mysql_real_escape_string($this->_band, $this->_handle);
    $query .= " and title=''";
    $query .= mysql_real_escape_string($this->_title, $this->_handle);
    $query .= "'";

    mysql_query($query, $this->_handle);
}

protected function _connect()
{
    $this->_handle = mysql_connect('localhost', 'user', 'pass');
    mysql_select_db('CD', $this->_handle);
}
}

```

通过将两个参数\$title 和\$band 分别指派给保护变量\$\_title 和\$\_band, 构造函数构建了 CD 对象。

CD 对象还具有一个名为 buy( )的公共方法, 该方法执行具体的销售。第一个步骤是调用受保护方法\_connect( )。\_connect( )使用适当的凭证创建一个与本地 MySQL 数据库的连接。接下来, buy( )方法创建了一条查询, 该查询会更新 CD 记录并将其设置为已购买状态。最后, 执行这个查询, 完成 CD 购买。

如下所示, 购买 CD 的当前代码拥有相当明星的流线型结构:

```

$externalTitle = 'Waste of a Rib';
$externalBand = 'Never Again';

$cd = new CD($externalTitle, $externalBand);
$cd->buy();

```

上面的代码创建了 CD 对象的一个新实例，随后执行了公共方法 buy()，从而完成了 CD 的购买操作。

因为销售形势喜人，所以我们已经扩展了服务器性能。现在，我们需要访问位于德克萨斯州达拉斯某处的数据。这就要求一个具有访问性能的 Proxy 对象，该对象需要截取与本地数据库的连接，转而连接达拉斯网络运营中心(Network Operations Center)。

Proxy 对象简单地扩展了基对象 CD，不过还替换了功能性：

```
class DallasNOCCDProxy extends CD
{
    protected function _connect()
    {
        $this->_handle = mysql_connect('dallas', 'user', 'pass');
        mysql_select_db('CD');
    }
}
```

受保护方法 \_connect() 会被 Proxy 对象重写。这个方法此时不连接 localhost，而是连接 dallas 主机。调用代码不清楚实际上与代理共同工作。如下所示，调用代码只做了很小的更改：

```
$externalTitle = 'Waste of a Rib';
$externalBand = 'Never Again';

$cd = new DallasNOCCDProxy($externalTitle, $externalBand);
$cd->buy();
```

在需要截取两个对象之间的通信时，最佳的做法是使用一个基于代理设计模式的新对象。



# 第 16 章

## 单元素模式

一天，老板告诉我，客户抱怨 Web 站点的运行速度缓慢。我决定对 Web 站点在高负载时的响应能力进行一些测试。首先，我测试了同时在线 25 位用户的情况。这已经足以使 MySQL 服务器崩溃了。通过反复试验，我能够找出让 Web 站点开始变得缓慢的同时在线用户数。Web 站点只能够完全应付同时在线 6 位用户的情况。很显然，老板要求我找出其中的原因，我也渴望解决这个问题。测试的页面来自某个内容管理系统(Content Management System)的内容，这并不复杂。经过一些调查之后，发现页面会查询两个特定的数据库系统。然而，每个页面都会打开与相同 MySQL 服务器的十多个连接以及与相同 DB2 服务器的两个连接。通过查看用于构建这些页面的查询，我又发现：在整个页面打开期间，这些连接状态不必一直存在。连接被打开、信息被检索之后，连接则被舍弃。以后，新连接只为新查询而打开。唯一的好消息是，代码库使用面向对象的方式连接数据库。经过某些重构后，所有对象只使用一个共享的数据库连接来连接各种类型的服务器。我并没有在调用数据库时生成新连接，而是使对象重用自身。为了完成这个功能，我使用了单元素设计模式。

名称：单元素

通过提供对自身共享实例的访问，单元素设计模式用于限制特定对象只能被创建一次。

### 16.1 问题与解决方案

单元素设计模式是我见过的最常用的设计模式之一。这种特殊的体系结构类型为通常只存在全局和过程编程的面向对象设计提供了平衡作用。根据单元素设计模式创建的对象一般只允许创建自身的一个实例。在最初的实例化时，这种对象将自身存储在内部。随后，该对象会提供对被存储实例的引用。每当出现对新对象的新请求时，单元素对象都会检查其存储器。如果已经具有自身的实例，那么就返回引用；否则，就按照前面介绍的步骤处

理新对象请求。

只允许自身一个实例的单元素最为常见。单元素被认为是一个运行时专有的，它们不被允许生成自身的副本。

设计模式是用于构建对象的基本蓝图。然而，正如反复重申的那样，设计模式确实提供了灵活性。此外，还有一种对象限制了任何时候所允许的自身副本数。初始的体系结构基于单元素设计模式，不过需要稍加修改。例如，这个对象可能最多存储自身的 5 个实例。如果出现第六个请求，那么就不得不等待，或者只是提供对先前创建的 5 个实例之一的引用。在排队请求中，这种体系结构类型特别有用。

不管设计哪种单元素类型，都存在一个需要被应用的基本特征。单元素对象的构造函数应当是一个受保护方法，这样就只允许指定类自身创建它的一个实例。构造函数还具有一个实际创建、存储和提供新创建实例的公共方法。

基于单元素设计模式的对象为全局变量的使用提供了一个友好的面向对象防护。虽然许多 PHP 包不具有强大的面向对象核心，但是仍然使用了 \$GLOBALS 数组。它们实现全局变量用法的最常见情况是用于配置选项。此时，配置对象可以被创建为单元素对象。一旦需要配置某个选项，代码就会请求新的配置对象。如果对象内部不存在一个被存储的实例，那么就会创建新实例。这个构造函数具有能够确定以及使所有配置选项可用的逻辑。随后，请求对象就能够利用对配置选项进行检索的对象。因为这是所有请求都要使用的对一个实例的引用，所以某个对象能够根据要求更新配置选项，并且其他请求者都可以使用该引用。上述的单元素对象完全能够替代用于配置选项的全局变量。此外，因为配置对象是一个对象而不是一个普通的变量，所以能够添加限制特定配置选项在初始实例化之后被更改的安全措施。

单元素设计模式最常用于构建数据库连接对象。数据库访问对象可以负责创建一个与数据库的实例化连接。接下来，只要调用这个对象的特定方法，该对象就会使用已创建的连接。因为创建与数据库服务器的连接开销很大(需要耗用大量的时间和资源)，所以代码应当尽可能不执行这种操作。在大多数情况下，数据库连接的状态在数据被检索之后不再需要保持。了解这两点后，读者就会明白将数据库访问对象创建为单元素对象是非常有意义的。因此，每次新查询请求数据库访问对象的新实例时，我们就能够确信重用该对象，并且不会创建另外的连接。

应当阻止创建某个对象的多个实例时，我们最好使用单元素设计模式来创建这个对象。

## 16.2 UML

如图 16-1 所示，该 UML 图详细说明了一个使用单元素设计模式的类设计。

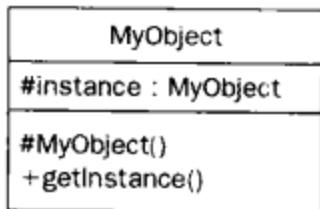


图 16-1

下面是对图 16-1 的说明：

- `MyObject` 被设计为一个单元素对象。名为 `MyObject()` 的构造函数是一个受保护方法。
- 公共方法 `getInstance()` 负责查看保护变量 `instance`。如果 `instance` 包含 `MyObject` 的一个实例，那么就会简单地返回这个实例。不过，如果 `instance` 为 `NULL`，那么 `setInstance()` 就会执行受保护方法 `MyObject()` 来构造一个新的 `MyObject` 实例；接下来，这个新实例存储在 `instance` 变量内；最后，`getInstance()` 方法会返回该实例。

### 16.3 代码示例

CD Web 站点的访问者一次能够购买多张 CD。实际上，我们非常鼓励访问者的这种行为。应当为访问者提供保存其购买物品的购物车。因为是实时处理库存，所以在购买 CD 后立即更新库存是十分重要的。为了实现这个功能，需要连接 MySQL 数据库以及更新指定 CD 的数量。使用面向对象方式时，可能要创建多个不必要的与数据库的连接。如下所示，完全可以选择基于单元素设计模式的库存连接：

```

class InventoryConnection
{
    protected static $_instance = NULL;

    protected $_handle = NULL;

    public static function getInstance()
    {
        if (!self::$_instance instanceof self) {
            self::$_instance = new self;
        }

        return self::$_instance;
    }

    protected function __construct()
  
```

## 第II部分 参考内容

```

    {
        $this->_handle = mysql_connect('localhost', 'user', 'pass');
        mysql_select_db('CD', $this->_handle);
    }

    public function updateQuantity($band, $title, $number)
    {
        $query = "update CDS set amount=amount+" . intval($number);
        $query .= " where band=' " . mysql_real_escape_string($band) . " ' ";
        $query .= " and title=' " . mysql_real_escape_string($title) . " ' ";

        mysql_query($query, $this->_handle);
    }
}

```

InventoryConnection 类的第一个公共方法是名为 getInstance( )的静态方法。这个方法查看受保护静态变量\$\_instance 是否具有类自身的一个实例。如果不具有，那么就会将类自身的一个新实例指派给\$\_instance 变量。接下来，无论是需要创建新实例，还是不必调用特定的方法，最后一个步骤都是返回来自\$\_instance 变量的实例。

上面的构造函数是一个受保护方法。因此，只有指定的对象才能够调用这个构造函数。\_\_construct( )生成与数据库的连接，并且将具体实例本地存储在保护变量\$\_handle 内。

公共方法 updateQuantity( )接受 3 个参数：乐队名、标题以及数量修改数。被创建的 MySQL 查询中会使用上述 3 个参数。最后，指定对象使用内部存储的句柄来执行该查询。

如下所示，只要购买 CD 就会涉及 InventoryConnection 类：

```

class CD
{
    protected $_title = '';
    protected $_band = '';

    public function __construct($title, $band)
    {
        $this->_title = $title;
        $this->_band = $band;
    }

    public function buy()
    {
        $inventory = InventoryConnection::getInstance();
        $inventory->updateQuantity($this->_band, $this->_title, -1);
    }
}

```

```
}
```

这个 CD 对象相当标准。不过，`buy()`方法让人很感兴趣。首先，它通过调用 `InventoryConnection` 的 `getInstance()`方法来获得这个类的一个实例。一旦接收到该实例，就会通过调用 `InventoryConnection` 对象的 `updateQuantity()`方法将指定 CD 的数量减 1。

大家可能非常熟悉使用这些对象的样本代码：

```
$boughtCDs = array();
$boughtCDs[] = array('band'=>'Never Again', 'Waste of a Rib');
$boughtCDs[] = array('band'=>'Therapee', 'Long Road');

foreach ($boughtCDs as $boughtCD) {
    $cd = new CD($boughtCD['title'], $boughtCD['band']);
    $cd->buy();
}
```

在这个示例中，`$boughtCDs` 数组表示来自购物车的商品项。具体代码会遍历循环每个被购买的 CD。代码首先创建一个新的 CD 对象，随后会请求对指定 CD 执行 `buy()`方法。因为这样的操作会发生很多次，所以 `InventoryConnection` 对象最好是单元素对象。针对每个被购买的 CD 都打开一个与数据库的新连接并不是一个好做法。

当某个对象的实例化在整个代码流中只允许发生一次时，最佳的做法是使用单元素设计模式。



# 第 17 章

## 策略模式

在我的工作经历中，最糟糕的那几个月时间里，我疲于应付一个具有庞大代码库的大项目。新的需求不断出现，我完全像一名毫无经验的初级编程人员那样：只知道在项目中添加这些需求。项目中的类越来越多，并且我逐渐地去除了单元测试。幸运的是，我正好才学习了 PHP 中的对象继承。从那时开始，我创建更多相似的类，每个类都通过扩展其他类添加了额外的功能。我的烦恼从拥有一些庞大的类转移到跟踪过多较小的类。问题仍然存在。如果不考虑存储库中对象列表的大小，那么就必须考虑开发命名方案来组织这些对象。我发现各种类中还存在许多臃肿无用之处：也就是在基类的大部分实例中不会调用的方法。接下来的重构步骤涉及从基类中取出这些方法以及将其添加入只在需要时创建的具体类中。这正是策略设计模式的真正魅力。

### 名称：策略

策略设计模式帮助构建的对象不必自身包含逻辑，而是能够根据需要利用其他对象中的算法。

### 17.1 问题与解决方案

面向对象编程的魅力有时可能由于用法错误而使其失色。许多编程人员能够将个别实体的模块化理解为对象。但是，面向对象体系结构并不仅仅停留于实体体系结构。不论是实体的创建还是对指定实体的更改，任何任务都可以被涵盖在某个对象中。策略设计模式就是将面向对象方式转至下一级别的优秀示例。

策略设计模式详细说明了一个对象的构造，从而通过去除对象本身的复杂逻辑而使该对象简洁。此时，这个对象不必在内部包含一组逻辑，而是可以在运行中调用其他类的算法。

最初接触策略设计模式时，编程人员可能将这个功能性的构造误认为是标准对象继承。最常用的一种面向对象编程由构建彼此扩展的对象组成。通过这种方式，核心对象就会变小，同时所有子对象都具有另外的逻辑集。

从功能上看，这会使人头痛。假定某个对象具有多个子对象，每个子对象都采用特定的方式更改基对象。如果只发生单个实例更改，那么扩展某个子类似乎较为容易。当特定对象需要进行多处更改时，问题就会不期而至。因为已经生成了父/子关系，所以需要创建特定对象的一个新实例，以便将新的子功能性添加至基对象。例如，某个基对象具有一个公共属性。一个子对象可能希望将该属性转换为大写字母，另一个子对象可能希望将其转换为不同的语言。将多个子对象一起链接至父基对象显得非常笨重，或者是不可能的。每个子对象的方法名必须有所区别，从而不会导致重写其他子对象功能性的情况。

尽管具有父/子关系，然而策略设计模式采用的方式是从父对象中去除功能性。不过，此时并非构建扩展父对象的对象，而是将指定逻辑更改为自包含算法。随后，这个类就是一个完整的对象，其唯一用法是执行指定的功能性。被正确构建之后，父对象可以选择在运行期间希望使用哪一个逻辑对象来更改自身。除了基对象变得简洁的明显优点之外，其他策略对象的可重用程度更高，而且更为灵活。

各种不同类型的基对象都能够重用 *Strategy* 对象。如果使用这种对象，那么我们不再需要构建一系列复杂的继承类，而且不必在所有不同类型的基对象子类中复制代码。因为 *Strategy* 对象是一种自包含算法，所以应当能够接受一个它毫不了解的对象并应用其变化。在特定算法要求某个指定对象来完成算法时，才会出现唯一的耦合级别。通常，这是一个可接受的依赖性级别。

计算运输费用是策略设计模式的一个优秀示例。在具体代码中，特定的商品项可以用一个对象表示。这个对象具有某些属性，如长度、宽度、高度和重量。调用针对该对象的购买方法时，还可以执行通过调用美国邮政服务(U.S. Postal Service, USPS)API 来确定运输费用的其他受保护方法。如果另一个商品项变为可用于销售，那么似乎肯定要复制这些受保护方法中的代码。实际上，具体的计算方法被取出并创建为 *Strategy* 对象。USPS 运输策略对象接受一个与其物理属性通信的对象。*Strategy* 对象调用 USPS API 并对基对象应用运输费用计算。此时，相同的 *Strategy* 对象能够被用于可销售的任何商品项。其他商品项也会使用这个 *Strategy* 对象。此外，将运输费用计算方法创建为 *Strategy* 对象还为今后的扩展添加了额外的灵活性。如果站点决定通过联合包裹服务(United Parcel Service, UPS)运输，那么就会创建一个新的 *Strategy* 对象。任何原始对象都不需要被更改。主调用代码随后只是判断应当对指定对象应用 USPS 策略还是 UPS 策略来更新运输费用的计算。

策略设计模式的另一种使用方式涉及创建模式替换。假定一个对象保存了指定 Web 站

点某个用户的简介信息。显示用户即时通信客户信息的简介信息内存在占位符。Profile 对象具有设置用户首选即时通信供应商类型的方法。接下来，对应的 Strategy 对象会被应用于 Profile 对象。Profile 对象将识别占位符并将其替换为一个与用户交互的链接。这甚至可以像从该服务的 API 中检索出状态信息一样复杂。因为 Profile 对象使用策略设计模式，所以执行更为快速和便捷。此时，我们不需要围绕于某个即时通信供应商的简单功能性以及显示其他供应商的复杂而庞大的逻辑，Profile 对象并未包含上述内容，而是在运行时期间通过只有 Strategy 对象需要才进行的实例化来检索这些值。

当对象能够自己去除算法并将算法替换为只有需要才被调用的其他自包含对象时，就应当使用策略设计模式。

## 17.2 UML

如图 17-1 所示，该 UML 图详细说明了一个使用策略设计模式的类设计。

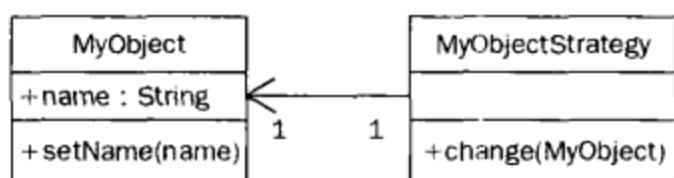


图 17-1

下面是对图 17-1 的说明：

- MyObject 包含一个名为 name 的属性，它表示通常接受更改的 MyObject 属性。公共方法 setName( )接收参数 name，该参数被指派给 MyObject 的名称属性。
- MyObjectStrategy 是自包含算法对象。这个对象拥有一个公共方法 change( )，该方法接受 MyObject.change( )的一个参数，并且通过更改公共的名称变量对 MyObject 执行特定的逻辑。

## 17.3 代码示例

示例 Web 站点大量使用了 AJAX。有时候，CD 对象有必要生成自身的 XML 版本，这就会返回开始处理的 JavaScript。

CD 对象的代码如下所示：

## 第II部分 参考内容

```

class CD
{
    public $title = '';
    public $band = '';

    public function __construct($title, $band)
    {
        $this->title = $title;
        $this->band = $band;
    }

    public function getAsXML()
    {
        $doc = new DomDocument();
        $root = $doc->createElement('CD');
        $root = $doc->appendChild($root);
        $title = $doc->createElement('TITLE', $this->title);
        $title = $root->appendChild($title);
        $band = $doc->createElement('BAND', $this->band);
        $band = $root->appendChild($band);

        return $doc->saveXML();
    }
}

```

构造函数接受两个参数\$title 和\$band，并且将它们指派给 CD 类的公共属性。getAsXML() 公共方法首先创建一个新的 DomDocument，接下来构建每个单独的节点并将其添加至新创建的 DomDocument，最后调用的 saveXML() 方法能够返回 XML 文件的字符串表示形式。

实现 CD 对象的代码相当简单：

```

$externalBand = 'Never Again';
$externalTitle = 'Waste of a Rib';

$cdd = new CD($externalTitle, $externalBand);

print $cdd->getAsXML();

```

编程团队加入了一名新开发人员，他在 AJAX 方面具有丰富的经验，并且对代码提出了改进意见。Web 站点的 AJAX 功能性需要额外的灵活性，包括能够将 CD 对象生成为 JavaScript 对象表示法(JavaScript Object Notation, JSON)实体。

CD 对象的第一个实现不够灵活，因此不能生成这种新的输出类型。而且，我们可能需要针对其他使用(uses)不断创建更多的 CD 对象表示类型。某些 Web 服务会要求 XML

或 JSON 之外的不同格式。此时，我们最好使用策略设计模式。

更改 CD 对象的第一个步骤是去除 XML 功能性。此外，CD 对象还应当能够执行我们所创建的 Strategy 对象：

```
class CDusesStrategy
{
    public $title = '';
    public $band = '';

    protected $_strategy;

    public function __construct($title, $band)
    {
        $this->title = $title;
        $this->band = $band;
    }

    public function setStrategyContext($strategyObject)
    {
        $this->_strategy = $strategyObject;
    }

    public function get()
    {
        return $this->_strategy->get($this);
    }
}
```

CDusesStrategy 类的第一部分与之前的 CD 类较为相似。不过，这个类存在其他两个公共方法和一个受保护属性。`setStrategyContext()`接受一个名为`$strategyObject` 的参数。通过被指派给保护变量`$_strategy`，该对象被存储在指定对象内，这样就会保存 Strategy 对象的一个实例。可将另一个公共方法`get()`认为是替换了 CD 类中的`getAsXML()`方法。因为应用于 Strategy 对象，所以该方法的名称更为抽象。存储在保护变量`$_strategy` 内的 Strategy 对象拥有自己的、在此处执行的`get()`函数。基类的一个实例被传递入 Strategy 对象，注意到这点十分重要。

接下来，我们需要创建用于 XML 和 JSON 格式的 Strategy 对象：

```
class CDAsXMLStrategy
{
    public function get(CDusesStrategy $cd)
    {
```

## 第II部分 参考内容

```

    $doc = new DomDocument();
    $root = $doc->createElement('CD');
    $root = $doc->appendChild($root);
    $title = $doc->createElement('TITLE', $cd->title);
    $title = $root->appendChild($title);
    $band = $doc->createElement('BAND', $cd->band);
    $band = $root->appendChild($band);

    return $doc->saveXML();
}

}

class CDAsJSONStrategy
{
    public function get(CDusesStrategy $cd)
    {
        $json = array();
        $json['CD']['title'] = $cd->title;
        $json['CD']['band'] = $cd->band;

        return json_encode($json);
    }
}

```

CDAsXMLStrategy类只拥有一个名为get( )的公共方法。前面曾经提及该方法是在CDusesStrategy的get( )方法中被调用的。这里的get( )方法接受CDusesStrategy的一个实例。除了使用\$cd代替\$this之外，随后的逻辑与CD对象中getAsXML( )方法的逻辑基本相同。

CDAsJSONStrategy类的设计与CDAsXMLStrategy类的设计几乎完全相同，明显的区别是构造了\$json数组来代替一个DomDocument。指定信息从\$cd变量中检索得到，该变量是CDusesStrategy的实例。最后，返回CD对象的JSON编码形式。

如下所示，执行使用Strategy对象的代码并不复杂：

```

$cd = new CDusesStrategy($externalTitle, $externalBand);

//xml output
$cd->setStrategyContext(new CDAsXMLStrategy());
print $cd->get();

//json output
$cd->setStrategyContext(new CDAsJSONStrategy());
print $cd->get($cd);

```

读者可能已经非常清楚，通过实例化 CDusesStrategy 就可以创建 CD 的一个新实例。为了输出 XML 格式，需要调用 setStrategyContext( )方法，该方法会发送相应 Strategy 对象的一个新实例。任何新功能都会进入使用 setStrategyContext( )指派的新 Strategy 对象的指定方法，注意到这一点十分重要。

在能够创建应用于基对象的、由自包含算法组成的可互换对象时，最佳的做法是使用策略设计模式。





# 18 章

## 模 板 模 式

无论是因为时间限制，还是为了使用开发人员所擅长的技术，开发中常常会出现在编程人员和团队之间分解一系列相似功能性的情况。但是，有时这会导致某些编程的公共接口中的不一致性。体系结构良好的基本代码要求基类从构建单独的功能性开始。为了创建这些实施一组特定接口和行为的基类，我们可以使用模板设计模式。

### 名称：模板

模板设计模式创建了一个实施一组方法和功能的抽象对象，子类通常将这个对象作为模板用于自己的设计。

### 18.1 问题与解决方案

某些编程人员并不能立即认识到实现模版设计模式的需要。一般说来，这些编程人员过度依赖于对象的继承，而且不理解继承的真正意图。事实上，它们对父方法不能实现所需操作的笼统解决方法是简单地覆盖它。在将某些理论用于模板设计模式中时，模板设计模式提供了一种更为健壮的解决方案。

模板设计模式创建的类能够用作另一个类的父类，这个类被设计为实现子类中存在特定的功能性方法。Template 对象具有较少的一组功能性，这略少于其他类。在大多数情况下，我们还会强烈地阻止或限制直接的类自身实例化。通常，使创建的模板类抽象化就能够实现上述目标。

一般来说，Template 对象定义了一个其他任何子类都能共有的共享公共方法。通过请求属于该对象的其他方法，这个方法内部会调用基本的设计步骤。将会属于某个子对象的这些方法既可以是必需的，也可以是可选的。如果是必需的，那么 Template 类就会定义需要这些方法的子类。如果上述方法对于子类来说是可选的，那么为了不禁止对共享公共方

法的操作，Template 类会将这些方法创建为一个分支。实现包括可选方法的另一种方式是查看指定方法是否存在，如果存在则调用该方法。

子类随后扩展 Template 对象使其能够接受上述设计或模板本身。子类负责创建 Template 对象定义的所有必需方法。因为这些方法被预定义在 Template 对象中，所以任何子对象都被认定拥有所有方法，并且在外部以相同的方式执行。每个不同的对象以自己的方式指定了这些方法的内部逻辑。大致说来，Template 对象定义的抽象步骤此时在子对象中进行了详细深入的说明。

模板设计模式与大多数设计方法稍有不同。这种特殊的模式在父对象中定义了一组具体化的步骤，并且指定了对父对象的许多控制。与依赖于子类的其他许多模式相对比，模板设计模式为指定对象给出了附加的结构和方向。这意味着，大多数模式都定义了一种独立的或对等的方式。模板设计模式尤其强调父/子关系。认识模板设计模式的另一个方法是将 Template 对象看作某个橄榄球队的教练。教练确定了比赛的战术与参与不同战术的球员。教练告诉四分卫在中场拦截，并且在得球后马上掷给前场的接球手。接球手是否能够接住球，这取决于四分卫(或者教练对象的子对象)丢掷球的力道和高度。

模板设计模式能够用在某个社交网站中。这个示例描述了一个被视为实体的基 Template 对象。实体能够被创建、共享或删除。为了促进该 Web 站点的社会要素，我们还可以公开创建被站点内所有用户共享的实体。在这个社交网站中，实体可以是链接、博客或图片。基对象 Entity 是一个抽象 Template 类，它包含名为 `create()`、`share()` 和 `delete()` 的 3 个抽象方法，这些方法将定义子类的职责。Entity 类还具有一个公共的最终方法 `generatePublicly()`，该方法使用了两个抽象方法。`generatePublicly()` 首先可能调用 `create()`，然后检索指定所有者的好友列表并针对每位好友都调用 `share()`。子类(如某个 Blog 对象)接下来会扩展基类 Entity，也就是需要定义 `create()`、`share()` 和 `delete()` 方法内部的逻辑。稍后，当为该站点的某位成员创建新博客时，就会调用 Blog 对象继承的 `generatePublicly()` 方法。这个示例极好地阐明了如何为将来在相同函数中添加新条目而设置某种架构。新功能的添加操作并不要求对调用逻辑或 Entity 基类进行其他任何修改。

模板设计模式的另一个应用示例是网上银行 Web 站点。一个银行能够处理许多不同类型的账户。通常，所有账户的处理方式都是相同的。网上银行账户接受存款、提供支付，并且可能具有增加或减少余额的其他逻辑。这些逻辑可以是活期账户的小额资金管理费，也可以是投资账户的存款利息。基本账户被创建为一个 Template 对象。这个对象具有名为 `applyAdjustment()` 的最终方法，这会先调用该对象定义的 `retrieveAdjustmentOnBalance()` 抽象方法，然后再调用 `add()` 或 `subtract()` 抽象方法。在处理活期账户的情况下，子类 Checking 会扩展主模板，如果账户的余额低于 100 美元时，那么该子类的 `retrieveAdjustmentOnBalance()` 方法将产生 10 美元的费用。接下来，父 Template 对象调用 `subtract()` 方法时会对指定账户的总

余额应用所产生的管理费用。生成这些基于模板的子对象的更重要的原因是进行存款账户的认证。在使用 `add()` 方法为指定账户增加利息时，子对象的 `add()` 方法会请求通过电子邮件将新结算单发送给账户所有者。

如果必须针对行为生成严格的指导原则并且使实际逻辑独立和灵活，那么就应当使用模板设计模式。

## 18.2 UML

如图 18-1 所示，该 UML 图详细说明了一个使用模板设计模式的类设计。

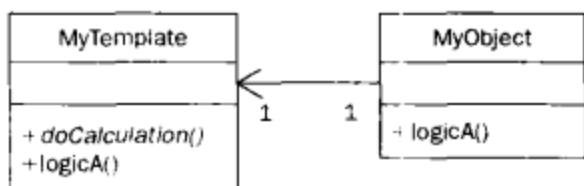


图 18-1

下面是对图 18-1 的说明：

- `MyTemplate` 类具有两个公共方法。第一个方法是名为 `doCalculation()` 的最终方法，它会在计算期间通过调用另一个名为 `logicA()` 的方法来应用逻辑。在 `MyTemplate` 中，这些方法应当是抽象的。
- `MyObject` 类扩展了 `MyTemplate` 类，这个类包含 `MyTemplate` 中引用的 `logicA()` 方法的实际逻辑。

## 18.3 代码示例

示例的电子商务 Web 站点已进行了扩展。现在，该站点的访问者能够定制许多新的商品项，包括划分了等级的谷类食品。这是一个很高的要求。此时，我们可以创建拥有自身价格以及附加税率的类。过去您能够应用统一的税率，然而对于目前的食品销售来说，每个类都具有适用于自身的税率。此外，某些商品项相当大。例如，某些产品还要求涉及附加费用的处理，这些费用会加入最后的购买价。

**注意：**

这个特定示例的主要区别是更为结构化。通常，我们创建最简单形式的类来说明某种设计模式。一般都会略过最佳应用(如使用接口实施访问)。不过，针对这种特定的设计模

## 第II部分 参考内容

式，指定的控制非常严密，我认为：通过使用抽象类和方法的更为结构化的方式，示例能够最好地说明模板设计模式。

如下所示，第一个步骤是定义使用模板设计模式来处理任何销售项的基类：

```
abstract class SaleItemTemplate
{
    public $price = 0;

    public final function setPriceAdjustments()
    {
        $this->price += $this->taxAddition();
        $this->price += $this->oversizedAddition();
    }

    protected function oversizedAddition()
    {
        return 0;
    }

    abstract protected function taxAddition();
}
```

`SaleItemTemplate` 是一个简单类。它是一个抽象类，因此能够实施要扩展的需求。公共属性 `$price` 被设置为 0。这个类定义的公共方法为 `setPriceAdjustment()`，该方法是一个最终方法，所以不允许被任何子类重写。通过调用 `taxAddition()` 和 `oversizedAddition()` 方法，`set Price Adjustment()` 实施了对价格的更改。

因为子类能够选择定义受保护方法 `oversizedAddition()`，所以该方法位于这个模板类内。由于大多数商品项都不会太大，所以没有必要在所有子类中都创建将要返回 0 的 `oversizedAddition()` 方法。

最后，模板类还存在受保护的抽象方法 `taxAddition()`。子对象有可能需要纳税，也可能不需要纳税。虽然某些商品项需要定义由于过大所带来的附加费用，但是更多的商品项需要定义是否纳税。因此，这个方法被创建为在子对象元素中实现其创建的抽象方法。

下一个步骤是更改 CD 对象的常规状态以及创建一个新的谷类食品对象：

```
class CD extends SaleItemTemplate
{
    public $band;
    public $title;

    public function __construct($band, $title, $price)
    {
```

```
$this->band = $band;
$this->title = $title;
$this->price = $price;
}

protected function taxAddition()
{
    return round($this->price * .05, 2);
}
}

class BandEndorsedCaseOfCereal extends SaleItemTemplate
{
    public $band;

    public function __construct($band, $price)
    {
        $this->band = $band;
        $this->price = $price;
    }

    protected function taxAddition()
    {
        return 0;
    }

    protected function oversizedAddition()
    {
        return round($this->price * .20, 2);
    }
}
```

CD 对象扩展了 SaleItemTemplate，其构造函数设置了指定对象的等级、标题和价格。因为 taxAddition() 被定义为 SaleItemTemplate 中的抽象方法，所以也必须被定义在 CD 类中。在这里，taxAddition() 方法计算了指定对象价格的 5% 税率并返回计算值。

BandEndorsedCaseOfCereal 对象也扩展了 SaleItemTemplate，其构造函数设置了与 CD 对象类似的等级和价格。接下来，BandEndorsedCaseOfCereal 对象还定义了在必要时使用的 taxAddition() 方法。此时，因为商品项与食品相关，所以不存在税率，taxAddition() 方法返回 0。不过，谷类食品是一种相当大的商品项。考虑到这个因素，总价格需要添加 20% 的附加费。BandEndorsedCaseOfCereal 中存在的受保护方法 oversizedAddition() 会重写模板中定义的同一方法，最后的返回值是当前价格的 20%。

## 第II部分 参考内容

下面的代码说明了如何使用这些类：

```
$externalTitle = "Waste of a Rib";
$externalBand = "Never Again";
$externalCDPrice = 12.99;
$externalCerealPrice = 90;

$cd = new CD($externalBand, $externalTitle, $externalCDPrice);
$cd->setPriceAdjustments();

print 'The total cost for CD item is: '.$cd->price. '<br/>';

$cereal = new BandEndorsedCaseOfCereal($externalBand, $externalCerealPrice);
$cereal->setPriceAdjustments();

print 'The total cost for the Cereal case is: '.$cereal->price;
```

上面两个示例都创建了一个扩展 `Template` 对象的子类。两个子类都调用可能对公共属性 `$price` 应用变化的 `setPriceAdjustments()` 方法，随后的输出显示了为指定商品项调整的价格。

创建定义了设计常规步骤，但实际逻辑留给子类进行详细说明的对象时，最佳的做法是使用模板设计模式。

# 19

## 第 章

# 访问者模式

社会不断地使所有事物拥有更多的功能和价值。如今，毫不费力地得到一组简单的功能或“经典”的商品项几乎是不可能的。编程和应用同样具有这样的趋势。人们发现，公司对新功能的要求非常强烈，以至于需要额外的编程人员。这些编程人员作为独立的顾问访问主要办公地点并被要求“更好地”构造应用程序。每个顾问都被告知相同的事情——创建新的功能，并且他们提供相同的产物——创建完美的新功能。不过，他们在公司内部都会通过不同的渠道实现这个目标。借助于相同的方式，我们可以设计一组对象来访问应用程序逻辑，并且以自己特有的方式“加以改善”。因此，访问者设计模式变得十分有用。

### 名称：访问者

访问者设计模式构造了包含某个算法的截然不同的对象，在父对象以标准方式使用这些对象时就会将该算法应用于父对象。

## 19.1 问题与解决方案

访问者设计模式有助于从基对象中去除复杂性。一般来说，也就是从基对象中分离出要应用的算法。这似乎是本书所介绍的许多设计模式的共同目标。不过，访问者设计模式以独特的方式解决了这个问题。

访问者设计模式名字的得来有一定原因。其他设计模式采取的方式是在外部设计应用于自身的自包含算法类，而访问者对象被设计为并入将要更改的对象。访问或进入指定对象之后，被发送入访问者类的一个基类实例就会执行标准的方法调用。因此，访问者设计模式实施了对这些来访对象逻辑的统一访问方式。通常，每个访问者都被询问相同的事情，但是他们在内部完成任务所执行的操作是不同的。

访问者设计模式的一个主要优点是能够在不更改对象的情况下就向该对象添加新的

功能性。如果基对象被构建为接受访问者，那么任何新逻辑都能够通过访问者接受方法被应用。根本不需要修改对基对象的单元测试。任何新功能都采用了访问者形式，访问者不同于基对象，并且具有自己的测试方法。任何新功能集都会被构造为具有自己独特操作的访问者，这为将来的开发提供了极大的灵活性。

不过，PHP 中的访问对象只能更改某个对象的公共属性，而且也只能访问公共方法。私有的以及受保护的属性和方法仍然在其修改范围之外。

博客是一个能够使用基于访问者设计模式的对象的示例。确定适当的请求之后，系统就会创建一个表示博客条目的对象。通常，视图类会检索公共属性并将其用于显示特定项。博客条目对象包含作者属性，该属性含有作者的用户名。因为博客 Web 站点的所有者现在希望使用全名，所以会创建一个访问者对象。这个对象访问指定的博客对象，博客对象则通过将对自身的引用传递给访问者来调用访问者的访问方法。接着，访问者执行一个基于用户名的查询，从而检索出全名。全名随后被应用于博客对象的作者属性，访问也随之结束。之后，访问者可能被通知所有条目时间都采用协调世界时间(Coordinated Universal Time, UTC)。我们可以创建一个新的访问者，这个访问者可访问博客对象，并且将公共的时间标记属性改变为作者所来自的时区。在上面两个示例中，基对象的属性都进行了更改。但是，基类本身并未被修改。

访问者设计模式的另一个应用示例是开放源代码服务 Web 站点。某个对象被创建用于表示将要下载的代码文件，它包含了若干公共属性，如项目名、在服务器上的位置以及作者。因为对这些开放源代码文件的需求很高，所以站点启动之后引入了其他镜像。创建的文件对象需要通过 3 个新镜像中的 1 个提供要下载的源代码文件。因为源码文件对象的功能非常完美，所以没必要对其进行修改。取而代之的是，我们可以构造一个新的访问者对象，该对象可以在指定下载源和 3 个附加的镜像之间进行随机的选择。在随后访问文件对象时，访问者对象会修改位置对象，从而将原始链接替换为属于某个镜像的链接。随着时间的推移，现实需求表明最好只接受以 TGZ(Tarred and GZipped)格式压缩的文件。所有镜像中的所有文件都被压缩成这样的格式，被接受的新文件也只能采用 TGZ 格式。现在，我们需要创建一个新的访问者，这个访问者能够查看公共的文件位置属性是否存在。如果不存在，那么就会查看文件是否存在 TGZ 版本。如果存在，那么就会修改文件位置属性指向新近压缩的 TGZ。

设计模式是项目的基础，这是我尽可能常常重申的一件事情。项目经常会被修改和增强。此外，设计本身是一种经过验证的方法，但是应用程序只是由理论得到的。就访问者设计模式而言，标准的方式可以是将访问者视为将一组功能性应用于某个对象的事物。不过，我们也可能因为不同的目的而创建访问者对象。在先前的示例中，访问者可以被创建为在发现新的 TGZ 文件时对数据库应用更新。一旦所有文件都已被转换，访问者就可以被

删除。这里只是想强调设计非常重要，实际用法可能不同。

当创建被封装在外部类中、能够以标准方式应用于主类的功能性时，可以使用访问者设计模式。

## 19.2 UML

如图 19-1 所示，该 UML 图详细说明了一个使用访问者设计模式的类设计。

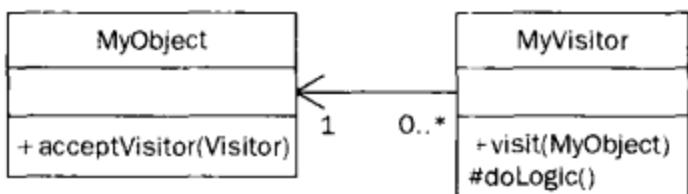


图 19-1

下面是对图 19-1 的说明：

- MyObject 类可以被访问。它是一个基类，并且具有名为 acceptVisitor( )的公共方法，该方法接受参数 Visitor。在内部，acceptVisitor( )根据传递的 Visitor 参数对象调用公共方法 visit( )。
- MyVisitor 类是这个 UML 图中的访问者。所有访问者都必须具有公共方法 visit( )，该方法接受 MyObject 的一个实例。visit( )方法可以调用其他逻辑，例如受保护方法 doSomething( )。这是访问类的特有部分。

## 19.3 代码示例

为了进行审计，电子商务 Web 站点上所有新的 CD 购买情况都必须被记录下来。这些信息随后会被归档。在库存数量不一致的情况下，就可以通过检查日志记录文件来查看指定的 CD 是否已被购买。

为了实现这个功能，CD 对象必须接受访问者：

```

class CD
{
    public $band;
    public $title;
    public $price;

    public function __construct($band, $title, $price)
  
```

## 第II部分 参考内容

```

{
    $this->band = $band;
    $this->title = $title;
    $this->price = $price;

}

public function buy()
{
    //stub
}

public function acceptVisitor($visitor)
{
    $visitor->visitCD($this);
}
}

```

CD 对象接收实例化 CD 的乐队、标题和价格，上面的构造函数将这些信息分别应用于公共属性\$band、\$title 和\$price。

CD 对象具有一个名为 buy( )的公共方法。示例中省略了购买逻辑。这个分支方法只是起到示例说明的目的。

CD 对象还具有另一个公共方法 acceptVisitor( )。为了遵从访问者设计模式，该方法是必需的。这个方法从\$visitor 参数接受访问者的一个实例。acceptVisitor( )方法内部会调用访问者的公共方法 visitCD( )，该方法使用\$this 变量传递 CD 类的一个实例。

日志记录访问者包含下面的代码：

```

class CDVisitorLogPurchase
{
    public function visitCD($cd)
    {
        $logline = "{$cd->title} by {$cd->band} was purchased for {$cd->price}";
        $logline .= " at " . sdate('r') . "\n";

        file_put_contents('/logs/purchases.log', $logline, FILE_APPEND);
    }
}

```

上面的类包含一个从 CD 类调用的公共方法 visitCD( )。这个方法接受名为\$cd 的参数，该参数是 CD 对象的一个实例。通过结合日志记录信息与 CD 对象的某些公共方法，接下来又创建了\$logline 变量。日志记录可以在 buy( )方法的内部进行，注意到这一点是十分重

要的。不过，如果日志记录样式发生变化或者请求其他日志记录，那么 CD 对象就需要被再次修改。这并不是最优的方式。最后，\$logline 变量被写入日志文件。

为了购买指定的 CD 以及记录销售情况，需要使用如下所示的代码：

```
$externalBand = 'Never Again';
$externalTitle = 'Waste of a Rib';
$externalPrice = 9.99;

$cd = new CD($externalBand, $externalTitle, $externalPrice);
$cd->buy();
$cd->acceptVisitor(new CDVisitorLogPurchase());
```

被创建的 CD 对象具有将被购买的 CD 的属性。随即会调用 buy() 方法。最后，CD 对象接受以新 CDVisitorLogPurchase 形式被传递到 CD 类 acceptVisitor() 方法的 Visitor 对象。最后一个调用实际上执行了日志记录。

最近公布的研究结果表明，查看主页的访问者实际上寻找的是打折的 CD，而不是原价 CD。因此，我们决定在标题版显示打折 CD 的实时更新列表。在这里，价格低于 10 美元的 CD 被认为是打折 CD。

为了完成该任务，需要创建一个新的访问者：

```
class CDVisitorPopulateDiscountList
{
    public function visitCD($cd)
    {
        if ($cd->price < 10) {
            $this->_populateDiscountList($cd);
        }
    }

    protected function _populateDiscountList($cd)
    {
        //stub connects to sqlite and logs
    }
}
```

CDVisitorPopulateDiscountList 类也具有公共方法 visitCD()。与创建的所有访问者一样，该方法也通过使用参数 \$cd 接受 CD 对象的一个实例。如果 CD 对象的价格属性小于 10，那么就会调用受保护方法 \_populateDiscountList()。CD 对象的实例会被传递到这个函数。

`_populateDiscountList()`方法只在主访问逻辑确定 CD 被限定为打折商品时才被调用，这个特别的实例将该方法表示为分支算法。不过，在实际的工作类中，打折 CD 的详细信息会被写入某个 SQLite 数据库或 XML 文件，访问者可以通过标题版获取这些信息。

加入新访问者的购买代码如下所示：

```
$cd = new CD($externalBand, $externalTitle, $externalPrice);
$cd->buy();
$cd->acceptVisitor(new CDVisitorLogPurchase());
$cd->acceptVisitor(new CDVisitorPopulateDiscountList());
```

当需要的对象包含以标准方式应用于某个对象的算法时，最佳的做法是使用访问者设计模式。

# 第 III 部分 PHP 设计

## 案例分析

第 20 章 需求分析

第 21 章 选择设计模式与计划

第 22 章 应用程序编程

第 23 章 使用更多设计模式进行改进





# 第20章

## 需求分析

我总是发现这样的情况：开放源代码领域中的编程人员会通过实践教育受益，尤其在涉足 PHP 时更是如此。虽然我喜欢到每位读者的家里坐坐，边喝饮料边编写 PHP 程序，但实际上这是不可行的。事实上，我准备通过一个案例深入地分析 PHP 设计模式。

在本书中，这个特定的案例分析会被分为若干部分进行介绍。首先，本章讲述了需求。我将努力排除任何假设的可能性，并且尽力回答案例分析所涉及的其他问题。第 21 章分析比较了参考章节中详细介绍的各种设计模式。最后两章(第 22 章和第 23 章)涉及代码的创建，其中使用了设计模式以及我开发的 UML 图。

以案例分析来结束本书是非常重要的。除了提供一种“动手”实践工具之外，我还希望读者了解下列 3 个要点。

- 首先，进行需求分析十分重要。我们需要小心地维持迅速开展工作和认真分析项目之间的平衡关系。我的目标是说明需求分析的重要性。
- 其次，需要注意的是编程前的分析。在本书绪论的最后一部分，我提到自己讨厌有人借口没有时间而直接开始编程。与需求分析一样，编程计划也是编程的一个重点。在实际编程之前留出适当的设计时间是至关重要的。我曾经经历过这样的烦恼：由于没有在最初进行规划设计，因此项目的开发时间和预算大大超出原先的设想。
- 再次，我希望说明的最后一个要点为编程人员编写应用程序的实际方式。令我感到沮丧的一件事是其他书籍几乎没有创建完整的应用程序，这些书籍给出了许多代码片段与实例，但是从未将它们合成为一个完整的项目。

我还计划以有些争议的方式给出案例分析最后的代码段：也就是计划出错。我的经验告诉我，最初的基本代码从来就不是完美的。在发布或进一步构建之前，基本代码往往会经过若干重构。大多数书籍所勾勒的完整项目似乎在第一次构建后就是完美的。莫非作者是比我优秀的编程人员？然而，他们很可能在形成最后的文本之前已经重构了其代码。我计划提供非常粗糙的代码，以便在不浪费读者宝贵时间的同时说明重构的价值。我认为，通过分析使用的某个设计模式是否最合理来说明设计模式的灵活性十分重要。在某些情况下，稍微改动代码和变换体系结构从而使用通过不同设计模式构建的某个对象，这可能更有意义。允许代码中存在“错误”的争议方式保证能为读者提供完整和准确的结果。最终版本的代码将是最后的体系结构，并且是有用的和正确的。

既然读者知道了即将学习的内容，那么让我们进一步讨论案例分析的实际内容。在这个前提下，我希望详细说明该实例的一些背景信息。我将为虚构的 ACME 公司编写代码。一位直接主管负责领导我的工作，并且我只能与其联系。刚进入这一行业的人可能忍不住笑起来。在开发一个项目时，有谁可能只联系某一个人？但是这个案例分析只是示例项目，而不是实际应用。请理解我的做法。主管将提出需求，他负责解答我的所有问题并做出最后的决定。我是项目的唯一编程人员。这意味着我不仅要提出所采用的体系结构，而且还要创建具体的代码。这个项目是一个在线联系方式管理器。

案例分析的第一个步骤是查看老板提供的需求文档。

## 20.1 初始需求

下面这些部分是我的老板(来自 ACME 公司)提出的针对联系方式管理器的需求文档。

### 20.1.1 执行纲要

ACME 公司准备将所有雇员的联系方式转移至某个在线 Web 页面。在线联系方式管理器可以通过 Internet Explorer 或移动电话被访问。联系方式管理器将保留公司目前电子邮件系统地址簿的绝大部分功能。公司的任何雇员都可以使用自己的证书访问在线联系方式管理器。

### 20.1.2 作用域

表 20-1 包含了若干作用域项。

表 20-1 若干作用域项

作用域	作用域项
作用域内	<ul style="list-style-type: none"> <li>创建一个在线联系方式系统</li> <li>分离使用不同证书的用户账户</li> <li>为 Internet Explorer 提供一个接口</li> <li>为移动浏览器提供一个接口</li> <li>从现有电子邮件系统导入联系方式的方法</li> <li>允许通过链接直接使用联系方式信息</li> </ul>
作用域外	<ul style="list-style-type: none"> <li>保持新、旧联系方式系统的同步与最新</li> <li>使用与网络应用相同的证书</li> </ul>

### 20.1.3 假设/限制

项目应当具有下面的假设与限制:

- 不会招致出现其他任何授权费用。
- 必须只利用现有的 Web 服务器。

### 20.1.4 详细需求

接下来详细说明项目的特定目标。

#### 1. Web 站点可用性

这个 Web 站点不会被限制于公司的内联网，而是应当对整个 Internet 可用。通过作为 Web 浏览器的 Internet Explorer 应当能够查看该 Web 站点。使用智能电话的雇员应当也能够通过电话浏览方式访问指定的 Web 页面。

#### 2. 联系方式信息

Web 站点上应当能够创建、更新或删除联系方式。每个联系方式都应当具有下列可用信息:

- 名、中名和姓
- 个人电子邮件地址
- 个人住址
- 个人座机/移动电话
- 公司名称
- 工作职位

- 公司电子邮件地址
- 公司地址
- 公司电话/分机/移动电话
- 公司社交网络 URL
- 即时通信软件名
- Web 站点

应用程序中出现的联系方式应当具有包含名、中名和姓的标签。

### 3. 联系方式同步

Web 站点应当从公司的电子邮件系统中取得现有的联系方式信息，并且能够将联系方式信息直接导入这个系统。

## 20.2 初始需求分析

来自 ACME 公司的老板向我提供了先前的需求文档。虽然有了良好的开端，但是我仍然有大量问题。接下来的步骤是考虑具体的实现方式。

对于由需求文档所产生的问题来说，存在下面两种解决方式：

- 创建一个问题列表并将其发送回利益相关方，随后可能发生下列 3 种情况之一：收到回复；只接收到更新后的请求文档；接收到指示自行更新请求文档的回复。
- 收集所有问题并按下列约定将其发送回利益相关方：计划基于所收到的问题回复来更新请求文档。我常常尝试使用第二种方式完成分析。通过这种方式，每个人都被清楚地告知下一步骤的操作。

注意：

在这里必须插入一个提示：本书涉及的是设计模式，而不是商业惯例。不过，在为 PHP 应用程序选择设计模式时，某些步骤是做出良好判断的必要步骤。如果没有阐明这些必要的步骤，那么体系结构可能会受到损害。此时，做出正确的选择会花费更长的时间，更糟的是可能进行过多的假设以及生成完全不够准确的体系结构。为了更为平稳地运行项目的剩余部分，我希望在案例分析的第一个步骤中就确认清楚的需求。此外，详细说明我的思考过程可能有助于读者从中学习如何完善自己的分析过程。

在这个示例中，我已经提醒老板：我准备通过先创建问题列表、然后更新请求文档来完成该项目。在接下来的内容中，我将要审查请求文档，并且找出需要澄清的地方。为了简洁起见，我准备在每个部分都给出来自老板的“回答”或“响应”。

### 20.2.1 大小/用户规模

我想要知道的第一件事情是应用程序的大小以及计划增长的规模。最开始有多少雇员会使用这个应用程序？规模会增长到什么程度？是否会在更多的公司展开？我注意到存在不增加授权费用以及使用当前 Web 服务器的约束。这将带来某些限制。判断现有的硬件能否支持预计的规模，这是我的职责。

- **问题：**公司中总共有多少用户会使用这个应用程序？预计同时会有多少用户使用这个应用程序？公司计划雇用新员工吗？是否有计划立即在 ACME 公司之外的其他公司或个体中使用这个应用程序？
- **回复：**公司雇用的员工被分为两班，每班 25 名雇员。公司员工人数的增长十分稳定：每个季度大约招收两名新雇员。我们没有立即向其他任何公司和个人发布该软件的计划。如果应用程序非常成功，那么毫无疑问会开展销售工作。

查看回复之后，我能够认定一些情况。

首先，公司总共有 50 名雇员。在任何时刻，应当只有一半员工使用该软件。公司还计划每季度招收两名雇员。因为我不知道这两人会被安排在哪一班，所以我计划将他们都放入其中某个班。一年以后，同时使用应用程序的用户最多会达到 33 位。根据我的经验，管理的趋势是处理总雇员人数上下附近的数量。我准备稳妥地假定同时使用软件的用户数为 40，初始用户的总数为 80。

接下来，就初始设计而言，软件计划分布在内部并且保持较小的规模。这个应用程序无疑需要增加规模，但是似乎可被暂定为小规模的。根据经验，即使利益相关方没有要求，我们仍然常常需要计划应用程序的规模。如果他们有一丝扩大规模的想法，那么我们有可能迅速地使多于预计数量的用户能够使用指定的产品。

### 20.2.2 联系方式信息的类型

在联系方式清单中，需求非常详尽。这些需求项被分为个人和公司联系方式信息。某些项似乎是重叠的。我知道这有极大的风险，例如，只有一个电子邮件地址能够作为个人的电子邮件信息。同样地，某些公司的客户除了个人住宅之外还拥有海边别墅。光是查看这些简洁列出的需求我都会小心翼翼。我认为应当将实际需求进行分类。为了说明实际需求需要一定的灵活性(使个人和公司信息都易于使用)，就不得不开发将这样的灵活性分类归入预定义字段的需求。

- **问题：**我注意到需求中列出了公司和个人地址、电子邮件和电话信息。这是否说明希望限制每种信息类型只具有一个信息？如果允许用户指定多个地址信息，那么是否建议首选某个公司和个人地址？同样地，我还注意到诸如个人电子邮件地

址或 IM 客户之类的信息类型也只具有一个信息。是否希望为这些信息类别提供多个信息？

- **回复：**如果用户能够同时建立个人和公司信息，那么最好按照自己的想法去做。联系方式可以具有两个或三个 IM 客户。允许 IM 客户添加多个信息。

借助于问题中所列出的建议选项，我确定了实际需求。应用程序不必限制只接口唯一的个人和公司信息。确认的需求是能够容易地添加多个信息。

回复的另一部分说明利益相关方没有考虑过每种联系方式信息类型可能具有多个信息的标准联系方式，其回复表明他们同意接口中的每个特征可以拥有多个信息。

从设计的角度出发，我已经收集了一些非常重要的信息。现在，我知道能够开发多组联系方式信息。需求建议这些组应当首选以 Business 和 Personal 命名。此外，我准备采用这样的假设：因为允许多个 IM 客户，所以允许组中存在多个字段类型也是没有问题的。

### 20.2.3 应用程序访问

一旦看到需求文档中提及了 Internet Explorer，开放源代码用户就可能不愿意参与其中。既然这个特定的浏览器占有巨大的市场，那么它肯定是一个不可忽视的浏览器。不过，就像人们趋向于使用商标名 Kleenex 表示擦面纸一样，我希望确定需求是否真的是 Internet Explorer，或者是否想说明应当能够使用这种最常见的 Web 浏览器。

另一个被提及的查看方式是移动电话。需求文档考虑使用智能手机访问具体的应用程序。确定提及的智能手机类型十分重要。这个范围可以大到任何移动接入设备，也可以被限制为 Internet Explorer on Windows Mobile 6 及更高版本。这都是我需要澄清的一些问题。

要澄清这些问题，就必须与大部分普通利益相关方进行特别微妙的讨论。业务需求应当尽可能基于应用需要，而不是基于技术规范。不过，通过预先确定更多的技术需求，我期望不让其他人员了解需求文档的内容。通过不从技术方面将事情过度复杂化的方式与利益相关方进行联系时，就会带来一定的微妙性。

我希望知道的最后一件事情是其他查看方式。我所想到的、在这个阶段可能会被忽视的一种查看方式是 Web 服务。考虑到前面曾经提及产品的销售可能性，Web 服务查看方式似乎也是可能出现的。

- **问题：**需求文档提到，将 Internet Explorer 作为查看应用程序的浏览器。我认为您可能最熟悉这种浏览器。我能否修改为使用非常流行的浏览器（当然包括 Internet Explorer）通过 Internet 来访问应用程序？此外，需求文档还提及了智能电话。能否指定应当支持的电话型号？最后，通过使用 Web 服务，应用程序的任何部分能否立即作为某个软件中的服务方法使用？或者留待将来开发？

- **回复：**我们没有意识到在公司内会有人使用 Internet Explorer 以外的浏览器。访问不必限于 Internet Explorer，只是支持其他浏览器不要导致预算的增加。移动电话应当是公司颁发的电话。SaaS 应用程序是不错的选择，但是并非必需的选择。

虽然这些回复不是我真正所想要的，但是有助于对需求文档的更新。我个人对没有优先考虑支持其他浏览器感到失望。不过，通过以自己的方式提出问题，我认为有助于去除需求中的一些特殊性以及将主浏览器支持移动至限制部分。

与移动电话的相关回复，要求我弄清楚公司颁发的标准电话。在案例中，标准电话具有不同的品牌，这些电话都安装了 Windows Mobile OS。

通过回复，我能够明白这个项目的主要目标是帮助提高业务功能性。在公司内联网外部能够使用该应用程序只是额外的目标。确保应用程序的核心业务部署硬件和软件完美无限是非常重要的。其他功能性和性能只是额外的惊喜。

#### 20.2.4 联系方式同步

阅读需求文档的文字表述之后，我还未完全理解联系方式同步处理。它是双向同步吗？是不断进行的吗？应当是自动化的吗？此外，需求文档还提及了公司电子邮件系统。我还应当了解这个系统。

- **问题：**需求要求通过某种方式从公司电子邮件系统导入联系方式。在说到现有的电子邮件系统时，是否意味着每台计算机上都安装了 Outlook 程序？此外，这种导入应当是不断进行的吗？这个应用程序应当支持双向同步吗？
- **回复：**我们构建这个应用程序的主要原因是替换现有的电子邮件和联系方式基础架构。授权费用已变得过高。另一个团队正在开发 Web 邮件应用程序。我们希望将来集成开发的所有应用程序。目前，我们认为最好独立使用这些应用程序。导入应当是从用户计算机上的 Outlook 一次性地导入至应用程序。导入完成后，用户会请求删除他们的 Microsoft Address Book。

老板通过这样的回复方式提出了更多原始需求文档中未指定的目标。现在，我对应用程序如何满足 ACME 公司的开发方案有了初步构想。

我已经了解了导入将是从 Outlook 导入至应用程序的一次性单向过程。幸运的是，我正好知道 Outlook 能够以 CSV 格式导出联系方式。这免除了需求收集阶段中的一个技术问题。

我还了解到自己开发的应用程序将来会与 Web 邮件应用程序集成在一起。在设计阶段，记住这一点十分有用。

## 20.2.5 用户证书

需求文档作用域外的一个需求是使用相同的网络登录证书。我恰好知道公司内联网具有一个 LDAP 服务器。根据经验，我发现集成指定的登录与 LDAP 并不困难。

澄清这个问题也可能特别困难。我想了解与这个作用域外的需求相关的更多信息。从用户的角度出发，使登录证书保持相同似乎是一个极大的优点。对客户进行这样的要求可能被认为带有一丝挖苦的意味，甚至会引起争议。我希望这只是一个需要澄清的问题，而不是对利益相关方判断能力的讨论，也并非试图增加项目的预算。

- **问题：**需求文档作用域外的一个需求是使用相同的网络证书来访问这个应用程序。如果不与网络同步，那么能否详细说明我应当知道的对证书的所有需求？
- **回复：**现在我们还未决定是否应当集成登录系统与 Web 邮件应用程序，或者登录系统是否应当与网络集成在一起。Web 邮件团队提及他们在使用网络时还有一些问题。就目前而言，登录系统可以是独立的。

与实际的需求文档相比，上面的回复为项目的设计部分提供了更多的素材。我知道必须确保身份验证方法是可置换的，该方法应当能够很容易被换出。这个作用域外的选项看起来是切实可行的。

## 20.3 最新需求文档

现在我已完成对需求的分析并收到针对所提出问题的回复，此时就该提供最后确定的需求文档。我会修改原始的需求文档，从而包含对所提出问题的回复以及添加其他被澄清的内容。接下来，我们将对此进行介绍。

### 20.3.1 执行纲要

ACME 公司准备将所有雇员的联系方式转移至某个在线 Web 页面。在线联系方式管理器可以通过大多数常用的浏览器或公司颁发的移动电话被访问。联系方式管理器将直接替换公司当前的地址簿软件。通过提供公司的自己的证书，任何雇员都可以使用 Internet 访问在线联系方式管理器。表 20-2 说明了作用域内和作用域外的功能。

表 20-2 作用域内和作用域外的功能

作用域	作用项
作用域内	<ul style="list-style-type: none"> <li>• 创建一个在线联系方式系统</li> <li>• 使用这个应用程序替换当前的 Outlook 地址簿功能性</li> <li>• 分开使用不同证书的用户账户</li> </ul>

(续表)

作用域	作用项
作用域内	<ul style="list-style-type: none"> <li>为大部分常用的浏览器提供一个接口</li> <li>为公司颁发的移动电话浏览器提供一个接口</li> <li>从 Outlook 地址簿执行联系方式一次性导入的方法</li> <li>允许通过链接直接使用联系方式信息</li> <li>灵活的、允许多种身份验证方式的身份验证系统</li> </ul>
作用域外	<ul style="list-style-type: none"> <li>向原有的 Outlook 系统提供联系方式信息</li> <li>使用与网络应用相同的证书</li> <li>此时对应用程序的 Web 服务进行访问</li> </ul>

### 20.3.2 假设/限制

项目应当具有下面的假设与限制:

- 不会招致其他任何授权费用。
- 必须只利用现有的 Web 服务器。
- 应用程序能够兼容 Internet Explorer。

### 20.3.3 详细需求

接下来详细说明了项目的特定目标。

#### 1. Web 站点可用性

这个 Web 站点不会被限制于公司的内联网，而是应当对整个 Internet 可用。通过使用大多数常见的浏览器(特别是 Internet Explorer)，用户应当能够查看该 Web 站点。使用公司颁发的 Windows Mobile 电话的雇员应当也能够通过其电话的浏览器访问指定的 Web 页面。

#### 2. 联系方式信息

Web 站点上应当能够创建、更新或删除联系方式。联系方式应当保留名、中名和姓。

每个联系方式可以具有无限制的信息分组。建议的第一个分组是 Business。其他分组名可以由用户指定。

每个分组至少具有一个用于联系方式信息的标签。每个标签可以具有一个或多个相关条目。下面列出了指定的标签:

- 电子邮件地址。

- 具体住址，包括城市、州和邮政编码。
- 可以具有分机的电话。
- 移动电话。
- 机构名称。
- 在机构中的职位。
- 社交网络 URL。
- 即时通信软件名。
- Web 站点。

例如，John Smith 可以具有名为 Personal 的一个分组，该分组中包含了两个移动电话、1 个电子邮件地址和 3 个即时通信软件名。

应用程序中出现的联系方式应当具有个人照片。如果存在个人照片，那么接下来是一个包含名、中名和姓的标签。

### 3. 初始联系方式导入

Web 站点应当从 Outlook 中取得现有的联系方式信息，并且能够一次性导入应用程序。

## 20.4 对最新需求的讨论

对需求文档进行的一些修改有必要进行下面的讨论：

- 我们直接提到了 Outlook 应用程序。我认为在整个需求中对此进行详细说明十分重要。执行纲要仍然非常简洁，但是作用域和细节会按名称出现在应用程序中。因为执行纲要只需要提及实际的目标，所以十分重要。具体细节应当确定公司实际采用的“地址簿系统”。并非所有人都使用相同的地址簿软件。既然利益相关方已经将 Outlook 软件确定为我应当考虑的正式地址簿软件，那么在需求中指明这个特定的软件能够维护我的权益。我不希望从开始就开发多个导入序列类型。
- 某些需求中不再提及 Internet Explorer，而是改为应当首先关注 Internet Explorer 兼容性的限制。我并不希望需求处于错误的状态，Internet Explorer 不能被用为查看应用程序的唯一浏览器。
- 就移动浏览器而言，因为只支持公司颁发的移动电话，所以提到这一点十分重要。借助于支持手机的 Java 型浏览器、Safari 型浏览器以及其他操作系统，应用程序能够很好地实现该需求。首要功能性是基于公司业务的，因此我希望继续只关注与应用程序相关联的公司业务。

- 我将表述“联系方式同步”改变为“初始联系方式导入”。发现联系方式处理过程中更多的是导入而不是同步之后，我希望确保这个功能性不存在错误的假设。每个用户的联系方式只能被一次性地导入。
- 联系方式信息部分是我改动最多的内容之一。我并没有将联系方式限制为个人和公司信息，而是改用分组方法。“Business”是建议的首选分组。不过，用户可以具有无限制的分组。这允许拥有多个职位的用户为联系方式指定多个公司名称。
- 所有分组的子部分也进行了修改。我不再只关注被限制于特定分组的信息，而是使信息更为通用。此外，因为用户可以具有数量不受限制的子类型，所以就解决了某人具有多个个人电子邮件地址或多个IM名的问题。
- 我还通过一个小示例详细说明了上面的功能性。根据我的经验，利益相关方在面对修改需求的时候，通过解释而让他们接受是很麻烦的。从收到的对联系方式信息相关问题的回复中，我发现的另一件事情是利益相关方对联系方式细节的理解过于单纯，他们似乎根本没有考虑过联系方式具有多种特定细节类型的情况。为了有助于利益相关方完善需求文档，我觉得在文档中添加一个简单的说明或实例会提供帮助。我知道某些纯粹的项目经理可能不同意我这样做，事实也是这样。我只是想告诉读者自己如何一步步成功地完成项目。

在这个案例分析中，我将最新的需求文档发送给老板。令人惊奇的是，利益相关方完全同意了这个需求文档。

## 20.5 本章小结

本章引入了我的一个案例分析，其中论述了如何计划在整个案例分析中覆盖业务需求、计划编制和应用程序的实际构建，还详细说明了自己完成该案例的独特方式。接下来，我回顾了为虚构的 ACME 公司开发在线联系方式管理器的样本业务需求文档。我说明了如何从初始需求中发现问题、与利益相关方对话以及将反馈信息应用于具体需求。最后，我创建了最新的需求文档，这将是后面章节的基础。第 21 章将介绍如何计划体系结构和选择某些设计模式。



# 第 21 章

## 选择设计模式与计划

在更深入地理解一组需求之后，我们就该进入项目的计划编制部分。我准备逐步跟踪应用程序的每个部分，同时详细说明我的想法并提供 UML 图。我知道可以立即进行编码并使用一些设计模式。然而，随着对每个步骤的分析，预先计划这个应用程序的好处会逐渐显现出来。

一般说来，针对应用程序的这些部分，我会生成一些假定。我准备绘制适用于编程部分的 UML 图。经验告诉我：这些蓝图既可以帮助记忆使用我的设计模式集合，也可以提供开发路线图。另一个优点则是在展开全部设计之后可能提供对项目的最新评估。

下面列出了我要讨论的应用程序部分：

- 主核心
- 用户交互与管理
- 联系方式管理

首先，将介绍应用程序的核心。

### 21.1 设计核心

围绕着应用程序的核心，优秀的编程人员通常会经历下列步骤：

- (1) 使用一个包含集合的程序化编程。
- (2) 接下来构建自己的架构，并将其用为下一个项目的基础。
- (3) 随后放弃自己的架构，同时在项目中调整改变已经过验证的现有架构。

通常，我会使用所学的知识到达步骤(3)并推荐使用第 2 章中提及的一种架构。不过，因为试图从初始计划的观点来说明设计模式的用法，所以我选择创建一个非常简单的架构。由于使用已有架构启动这个应用程序，因此相关代码已编写完成。在这种情况下，我只能够猜想最初的编程人员的思考过程。

### 第 III 部分 PHP 设计案例分析

我偏爱的体系结构类型是模型 - 视图 - 控制器(Model-View-Controller, MVC)。为了动手设计应用程序的核心，我准备构建 3 个对象、1 个控制器、1 个模块(或模型)和 1 个视图对象。

如图 21-1 所示，Controller 对象负责确定应用程序根据 Web 请求而试图完成的动作类型。这个对象具有两个公共方法 `_construct()` 和 `render()`。`_construct()` 方法负责理解 Web 请求，而且还可以处理任何发布的信息。`render()` 方法负责执行适当的调度动作，如果可能的话还会返回指定操作的输出。

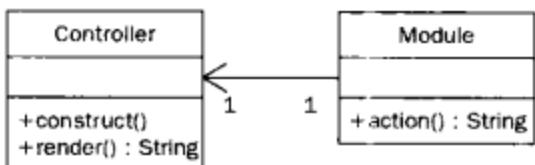


图 21-1

图 21-1 中的 Module 对象只是一个用于 Web 站点每个部分的构建模块。这个对象至少包含一个由 Controller 对象执行的公共方法。Module 类的设计使用了模板设计模式的变化形式。Web 站点的每个部分都会创建自己的、与基类相似的 Module 类型。

用于应用程序核心的最后一个元素是 view 对象(见图 21-2)。

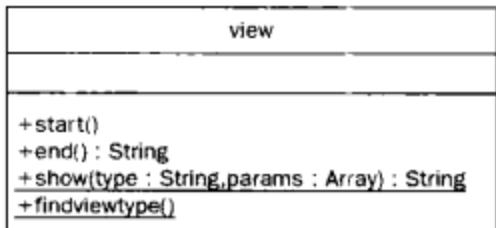


图 21-2

view 对象拥有公共方法 `start()` 和 `end()`。在自己的工作经历中，我已经开发了许多不同样式的视图。在生成视图时，我往往选用 PHP 的 `ob_start()` 方法。假定 `start()` 方法将包含对 `ob_start()` 的调用是十分合理的。`end()` 方法应当能够返回从缓冲区中捕获的数据。

view 对象还具有一个名为 `show()` 的静态公共方法。这个方法不仅接受一个表明显示视图类型的字符串，而且还接受一个参数数组。`show()` 方法需要基于 `type` 字符串找到适当的视图，随后会将数组中的所有参数传递至要显示的视图。

我最初的想法是通过 `show()` 方法创建基于 `Template` 对象的视图对象，随后将参数数组传递给视图对象或者将其直接指派为公共属性。但是经过考虑，我认为这只是为了创建对象而创建对象，并且会引入过多的系统开销。事实上，创建的视图对象包含文件系统中的文件，并且将参数数组作为本地变量。

我还添加了公共静态方法 `findviewtype()`。这个方法被用于设置当前视图类型，因此可以被 `show()` 方法引用。需要记住的是，应用程序必须支持常规的浏览器和移动浏览器。我的想法是，在不必考虑所使用浏览器类型的情况下能够从任何模块调用 `show()` 方法。视图对象应当能够处理这样的透明性。

查看经过图示说明的上述对象之后，我知道在自己面前有许多工作要做。这个编程步骤应当非常有趣。接下来，让我们继续步入应用程序的下一部分。

## 21.2 设计用户交互

既然设计了核心部分，我们就可以构建应用程序的下一部分。用户交互部分是能够为用户提供一个接口的一组模块。应用程序所需的交互如下所示：

- 身份验证/授权
- 创建、编辑和删除用户
- 提供对所有用户的管理访问

### 21.2.1 身份验证和授权

用户访问站点的第一个页面时存在两种可能结果(见图 21-3)。如果用户通过了身份验证，那么就会看到他们的联系方式列表。如果用户没有通过身份验证，那么他们就会被重定向至登录页面。

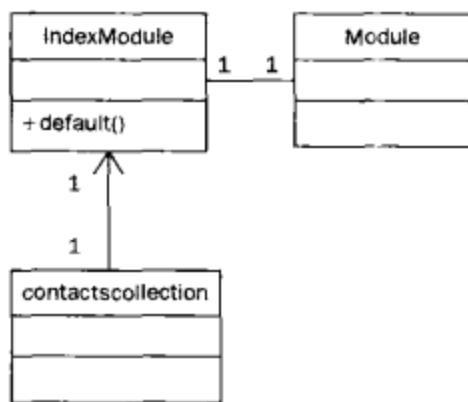


图 21-3

`IndexModule` 对象是在核心部分中详细说明的 `Module` 的一个实例。它具有一个名为 `default()` 的公共方法，该方法是 `Controller` 对象在调用 `render()` 方法期间将要执行的操作。`default()` 方法需要判断是显示联系方式集合还是执行重定向操作。

从逻辑上出发，我准备假定下一步骤是为用户提供登录视图。稍后会讨论 `contactscolle-ction`，但是它只会在当前用户已经过身份验证的情况下使用。

图 21-4 所示的模块提供了登录功能。

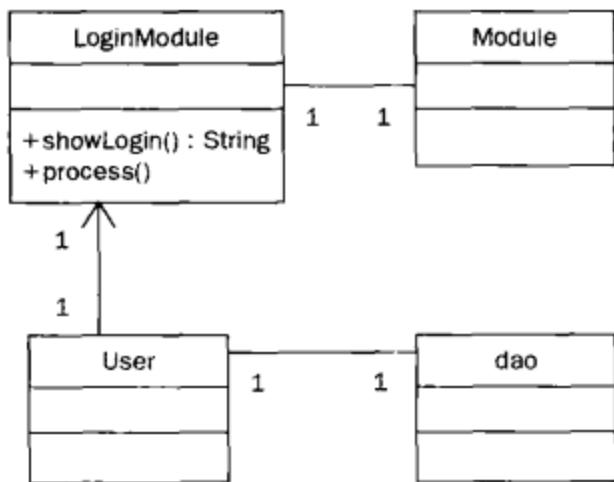


图 21-4

LoginModule 具有两个公共方法。showLogin( )方法会返回一个字符串，它利用了 view 对象的 show( )方法。我计划使任务为进行显示和返回一个字符串的模块的所有公共方法都至少调用一次 view 对象的 show( )方法。以前缀“process”命名的公共方法则可能不会利用 view 对象。

showLogin( )方法所显示的视图会接受一个用户名和密码，它们将支持雇员用户登录 Web 站点。应用程序的其中一个需求是具有灵活的身份验证系统。迄今为止，该系统仍然相当灵活。如果今后需要显示电子邮件地址字段，那么这个模块可以显示不同的视图。更进一步说，诸如单点登录之类的行为可以直接调用 process( )公共方法。

登录界面被提交时，就会执行 LoginModule 对象的 process( )方法。因为 Controller 对象处理被提交的所有数据，所以为了获取来自上一个视图的被提交数据，LoginModule 对象需要与 Controller 对象交互。我期望创建一个具有上述数据的 User 对象，注意到这一点十分重要。我的想法是尝试创建一个 User 对象，然后通过密码对该对象进行身份验证。如果身份验证失败，那么就会丢弃这个 User 对象。我认为 User 对象的身份验证往往能够成功，因此在身份验证系统的剩余部分中能够继续使用该对象。对用户数据库执行查询以便首先确定密码是否匹配会更浪费资源。如果采用这样的方式，那么我无论如何都必须创建一个新的 User 对象(这个新对象很可能会执行另一个查询)。立刻创建用户的方法看上去似乎更好。

User 是基于数据访问对象设计模式的 dao 对象的一个实例。因为 LoginModule 的 process() 方法会创建一个扩展 dao 对象的新 User 对象，所以下个步骤就是图示说明如何试图实现数据访问对象设计模式(见图 21-5)。

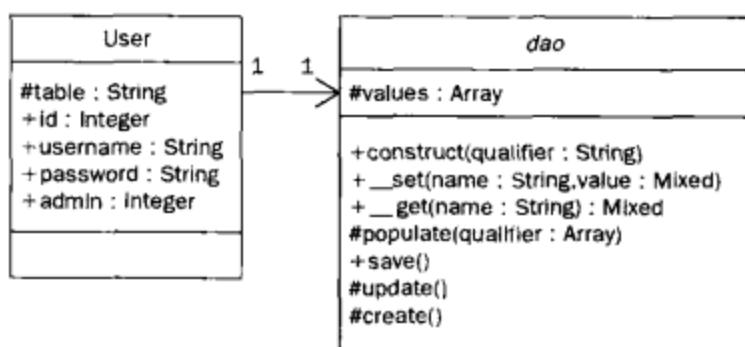


图 21-5

User 类扩展了 dao 类。id、username、password 和 admin 这 4 个公共特性标识了一个 User。dao 类在试图访问数据源的时候会使用受保护属性 table。

dao 对象的构造函数接受一个限定符。如果这个限定符是空位，那么就不会执行任何操作。这允许创建新的空白 dao 对象并在稍后进行保存。不过，为了检索 dao 对象，需要将某些限定符发送至 dao 的构造函数。构造函数随后会调用接受限定符数组的受保护方法 populate()。我准备预测是通过发送用户名还是发送 ID 来检索一个 User。因此，我期望 dao 构造函数接受指定的限定符并基于子类的表变量确定一个映射。最符合逻辑的方式是匹配 dao 类的公共属性与指定表的列。如果所有表都是使用类似模式创建的，那么这个操作几乎是自动化的处理过程。创建映射后，指定信息会被传递至 populate() 方法，从而能够构建一个查询。

populate() 方法负责将指定对象的信息填入受保护数组 values。随后，PHP 的巧妙方法 \_\_set() 和 \_\_get() 会被用于访问这些模拟子对象使用属性的数据。我选择使用上述方法而不是直接使用属性，原因在于将来可能会在 dao 对象中添加自己的公共属性，但是并不希望将其重写为某个子对象的属性。

查看 User 对象 UML 图，读者可能发现难以理解的地方是我已经定义了上面提到的公共属性，但是 User 对象实际上并不具有这些属性。事实上，扩展 dao 的任何对象的公共属性应当是推断属性，并且未被定义。因此，dao 对象的 \_\_set() 和 \_\_get() 方法仍然会被执行。

子对象发生任何变化时，dao 对象的公共方法 save() 会根据需要被调用。这个方法负责确定应当更新某个元素还是创建一个新元素，从而分别调用受保护方法 update() 和 create()。

update() 和 create() 方法都需要与数据库进行通信。因为目前的需求包括使用现有的服务器，所以我知道要使用 MySQL。不过，我希望使之抽象化，从而能够根据需要交换各种不同的数据库系统。有些时候，应用软件可能被移动至不同的平台，因此这种灵活性是有必要的。

基于这个需求，第一个浮现的设计模式是工厂设计模式。此时，我会被请求一个数据库连接的 MySQL 实例。不过在今后，应用程序能够通过工厂模式请求不同的数据库连接。

### 第 III 部分 PHP 设计案例分析

因为我希望保持所有数据连接的一致性，从而在交换至不同的数据时不必进行其他任何更改，所以还要使用模板设计模式是明智的。主类 db 将成为一个用于任何特定数据库对象的模板，并且提供工厂设计模式来构建数据库对象(见图 21-6)。

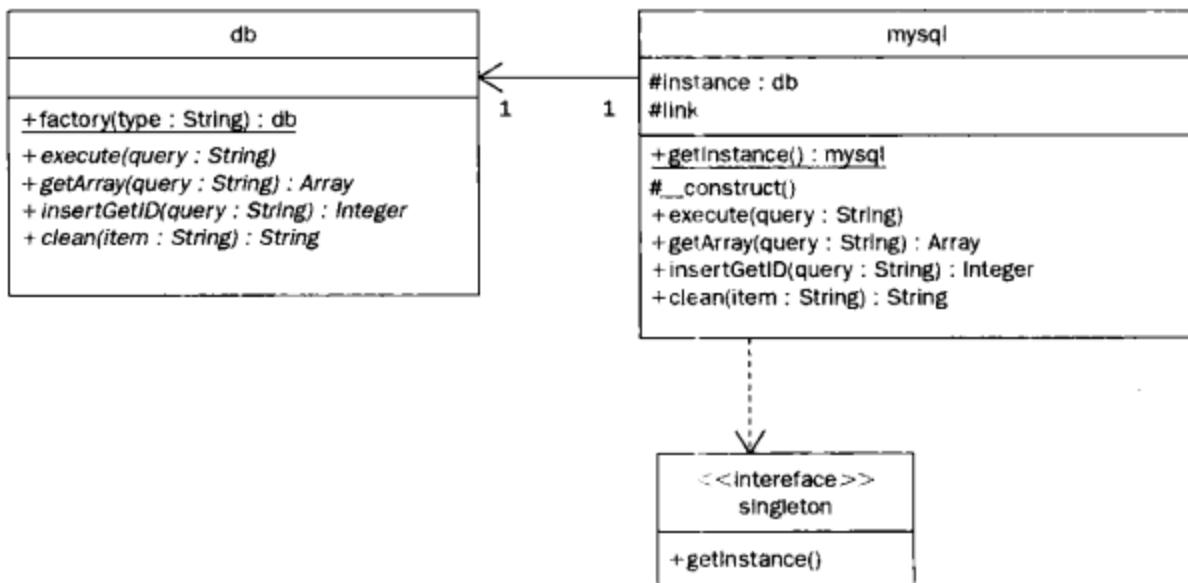


图 21-6

db 对象具有一个名为 `factory()` 的公共方法，该方法接受一个字符串参数，从而确定应当创建的数据库连接对象的类型。随后，`factory()` 方法会返回指定数据库连接对象的一个实例，它实际上继承了 db 的一个实例。

剩下的 4 种方法是抽象声明的公共方法，它们都是这种模板设计模式的构建块。`execute()` 命令会接受并运行一个查询。`getArray()` 方法会接受与运行一个查询，并且将结果作为一个数组返回。`insertGetID()` 方法会接受与运行一个查询，并且返回插入操作的 ID，这个 ID 与主键非常相像。最后，`clean()` 方法负责过滤字符串条目的内容，从而去除任何可能的 SQL 数据库注入。

我计划创建的第一个数据库连接对象是 mysql 对象，它扩展了 db 对象。mysql 对象基于单元素设计模式，并且将实现单元素接口。mysql 的第一个方法是单元素接口需要的公共静态方法 `getInstance()`，这个方法会返回自身的一个实例，该实例被存储在受保护的静态实例变量中。

Web 站点上的每个页面应当与指定数据源只连接一次，这似乎建议设计每个数据库连接对象都使用单元素设计模式。因为采用了单元素设计模式，所以 mysql 的 `__construct()` 方法是受保护的。这样一来，在指定类的外部创建类的实例是不可能的，从而固化了对 `getInstance()` 方法的需求。单元素接口会强制实现这个必要方法的存在。最后，mysql 对象具有了在 db 对象中定义的 4 种抽象方法。

回顾登录模块，我发现该模块基本实现了所有的相关需求。现在应用程序能够显示登录界面、接受用户输入以及创建使用数据访问对象设计类的用户。最后一个步骤是处理用户的身份验证。

在需求的收集和分析期间，我发现身份验证方法并不是固定的(见图 21-7)。目前，身份验证方法与公司中的其他事物毫无关系。不过，我计划采用单点登录的形式，并且在将来集成 Web 邮件系统或利用公司网络身份验证系统。因此，我再次需要工厂方法所提供的灵活性。

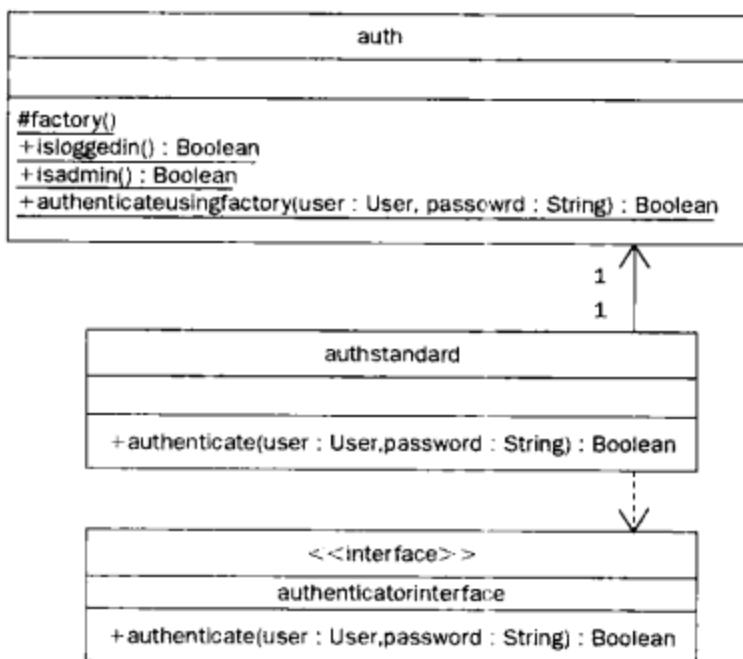


图 21-7

应用程序不会再次提供公共的工厂方法，取而代之的是创建了受保护方法。我的想法是 **auth** 对象实际上只需要静态方法。因此，为了更加简单，我只是会直接调用 `authenticate_usingfactory()` 方法，该方法接受一个 **User** 和一个密码字符串，并且在内部通过调用 `factory()` 方法获得身份验证器的一个实例。

由于没有更好的名称，我只好将当前的身份验证机制称为 **standard**。**authstandard** 对象是一个能够使用的身份验证器的示例。每个身份验证器在外部的运作是相同的，这就要求所有身份验证器都要实现 **authenticatorinterface**。该接口只定义了一个名为 `authenticate()` 的方法。无论其内部的工作方式如何，身份验证器都具有 `authenticateusingfactory()` 方法期望的公共方法 `authenticate()`。

我的想法是使用静态方法 `authenticateusingfactory()` 来进行一个条件比较。通过使用 `factory()` 方法，我们可以获得身份验证器的一个实例。随后，在不知道身份验证器其他相关信息的情况下，这个静态方法会返回指定身份验证器的 `authenticate()` 方法的输出。不过，

`authenticateusingfactory()`方法会将 User 和密码字符串转发至 `authenticate()`方法。

完成上述所有身份验证之后，用户就可以登录并前进至索引模块。索引模块目前会显示 `contactscollection` 对象。我将在后面部分深入讨论该对象。

最后，剩下的两个公共方法能够实现其他两个功能。索引模块需要知道用户是否已登录。`isloggedin()`方法对此进行了判断并返回一个布尔返回类型。考虑到用户管理，Web 站点需要知道当前用户是否是管理员。此时可以使用 `isadmin()`方法，该方法能够基于当前 User 对象的 `admin` 属性返回一个布尔返回类型。

不要忘记所计划事情的对立面(这里登录的反向动作就是登出)，这是十分重要的。登出也由一个简单的模块处理(见图 21-8)。

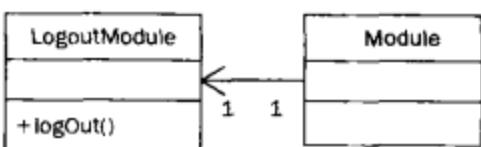


图 21-8

`LogoutModule` 只具有一个对应其唯一动作的公共方法 `logout()`。这个方法会登出当前用户，随后将用户重定向回索引模块。

考虑到 `User` 对象的创建，因此身份验证期间的身份验证用户创建部分有些重叠。不过，在这个步骤中，我准备定义更多的 `User` 对象交互。

### 21.2.2 创建、编辑和删除用户

站点的管理员应当能够创建、编辑和删除系统的授权用户。这个功能由 `UsersModule` 完成(见图 21-9)。

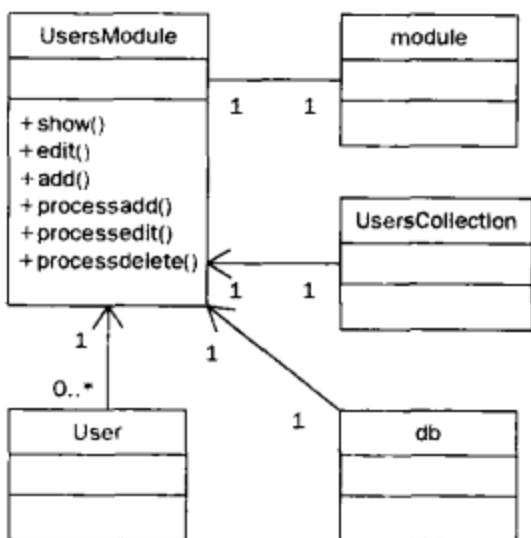


图 21-9

这个模块具有 6 个公共方法，它们对应于我为用户计划的 6 个动作。`add()` 和 `edit()` 方法能够分别显示用于添加用户和编辑用户的视图。`show()` 方法会显示当前 `User` 对象通过使用 `UsersCollection` 对象能够访问的所有 `User` 对象的一个视图。

`processadd()` 和 `processedit()` 方法可以与 `User` 对象交互。这两个方法能够填入指定 `User` 对象的公共属性，随后调用 `dao` 对象的 `save()` 方法。基于 `dao` 对象 `save()` 方法的设计，应用程序能够知道是使用 `create()` 方法添加对象还是调用 `update()` 方法。

`processdelete()` 方法会创建一个删除指定用户的查询。如果只为了进行删除而创建一个 `User` 对象，那么是毫无意义的。这个过程看起来似乎需要采用两个查询，不过我只计划使用一个查询。我期望在这个方法中直接处理一个查询和 `db` 对象。

读者可以看到，这些方法将处理单独的 `User` 对象。下一部分涉及创建处理 `User` 对象的管理接口。创建这样的接口后，应用程序就能够执行上述操作。

### 21.2.3 提供对所有用户的管理访问

因为 `UsersModule` 的 `show()` 方法通过处理 `UsersCollection` 来显示当前 `User` 能够访问的所有用户，所以接下来的步骤是构建这个集合类。

`User` 只是 `dao` 对象的特化实例，因此 `UsersCollection` 作为 `daocollection` 的特殊实例是非常合理的。因为我要创建一个被包含在其他对象内部的对象集合，所以迭代器设计模式是最佳选择。同样，因为存在不同的特化集合，所以模板设计模式也会被应用于 `daocollection` 对象的设计(见图 21-10)。

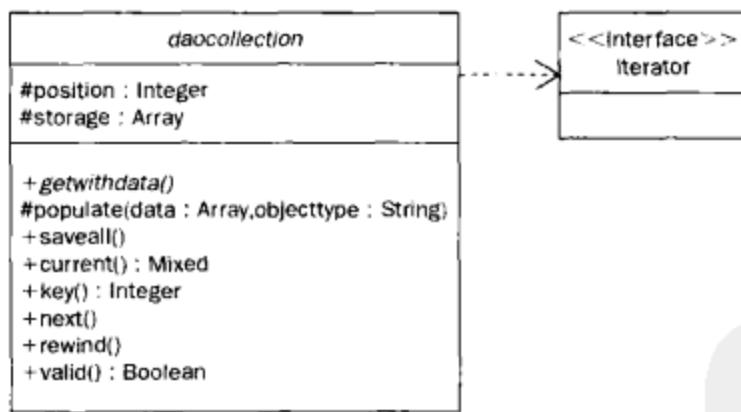


图 21-10

`daocollection` 类是一个抽象对象，而且还实现了 PHP 的 SPL 迭代器接口 `Iterator`。因此，它必须存在下列 5 个公共方法：`current()`、`key()`、`next()`、`rewind()` 以及 `valid()`。为了向这些方法提供特定的功能性，`daocollection` 类需要添加两个受保护属性。第一个受保护属性是存储数组，该数组存储被集合的所有 `dao` 对象；另一个属性是位置整型数，该整数存储当前输入存储数组内部的位置。执行先前的 5 个方法都会更新或访问 `position` 和 `storage` 属性。

受保护方法 `populate()` 接受两个参数：一个数据数组和一个 `dao` 对象类型字符串。通常，我预想该方法接受一个数据数组，这个数组可能使用与 `db` 对象的 `getArray()` 类似的方法收集数据。接下来，针对数组中的所有元素，`populate()` 方法会创建一个以类型字符串参数命名的新 `dao` 对象，并且将数据数组中的所有值都指派给该对象。最后，这个方法会将新创建的 `dao` 对象添加至内部的存储数组。

最后一个方法是名为 `getwithdata()` 的抽象公共方法。子 `daocollection` 的每个特定类型都具有自己的、与数据源交互的查询类型。这种抽象方法在 `daocollection` 对象中依照模板设计模式而存在。

经过进一步分析，我已经决定每个集合都应当具有一个所有者，任何集合都应当是相应所有者的一组 `dao` 子对象。因此，每个子 `daocollection` 在被创建时都需要接受一个 `dao` 对象。为了确定填充指定集合的 `dao` 对象，我将给出所有者的某些属性。

`UsersCollection` 集合(见图 21-11)是我计划创建的第一个集合。这个集合扩展了上面刚刚介绍过的 `daocollection` 对象，并且还实现了 `daocollection` 接口。

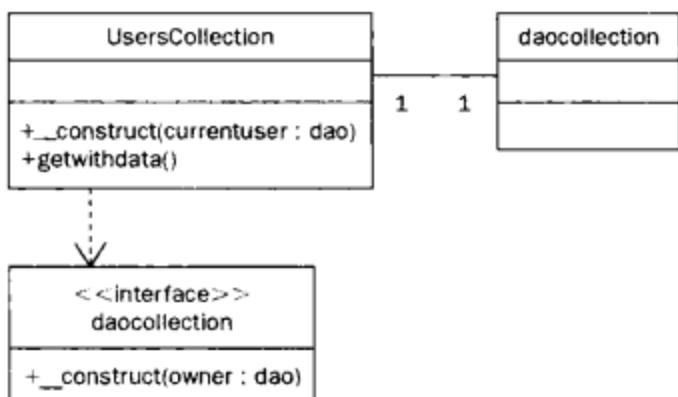


图 21-11

指定子 `daocollection` 的 `daocollection` 接口应当具有接受一个 `dao` 所有者对象的构造函数。

`UsersCollection` 具有两个公共方法。由 `daocollection` 接口实施其存在的第一个方法是构造函数。`__construct()` 接受作为一个参数的当前 `User` 对象。我准备使用当前用户实例来确定是否应当生成指定的集合。只有管理类型用户才能够看到系统中的其他用户。需要记住的是，`User` 对象具有公共属性 `admin`。

第二个公共方法 `getwithdata()` 被定义为 `daocollection` 对象中的抽象方法。这个方法可以接受一个数据库连接，构建获取系统中所有用户的查询，然后调用来自父 `daocollection` 的 `populate()` 方法，同时将检索结果填充入指定集合。此外，字符串 `User` 也会被发送入指定集合，从而指定这些数据应当被用于创建一个 `User` `dao` 对象集合。

当 `UsersCollection` 对象创建完成时，这标志着针对用户交互的计划结束。不过，任何用户在系统中都可以具有联系方式。从逻辑上讲，下一个计划步骤是应用程序的最后一个部分：联系方式管理。

## 21.3 设计联系方式管理

考虑到应用程序将具有高度灵活性，因此这个部分将处理一组更为复杂的关系。除了这些关系之外，还应当创建一些添加、编辑和删除上述对象的接口。另外，我也准备创建另一个接口将 Outlook 中的一组联系方式适配为具体系统内的新联系方式。最后，用户应当能够查看他们自己的联系方式。

这种联系方式管理被分为下列几个步骤：

- 处理联系方式与信息。
- 联系方式信息关系。
- 导入联系方式。
- 查看联系方式。

### 21.3.1 处理联系方式与信息

联系方式交互的基础是 ContactsModule。接下来，我准备开始定义这个模块(见图 21-12)。

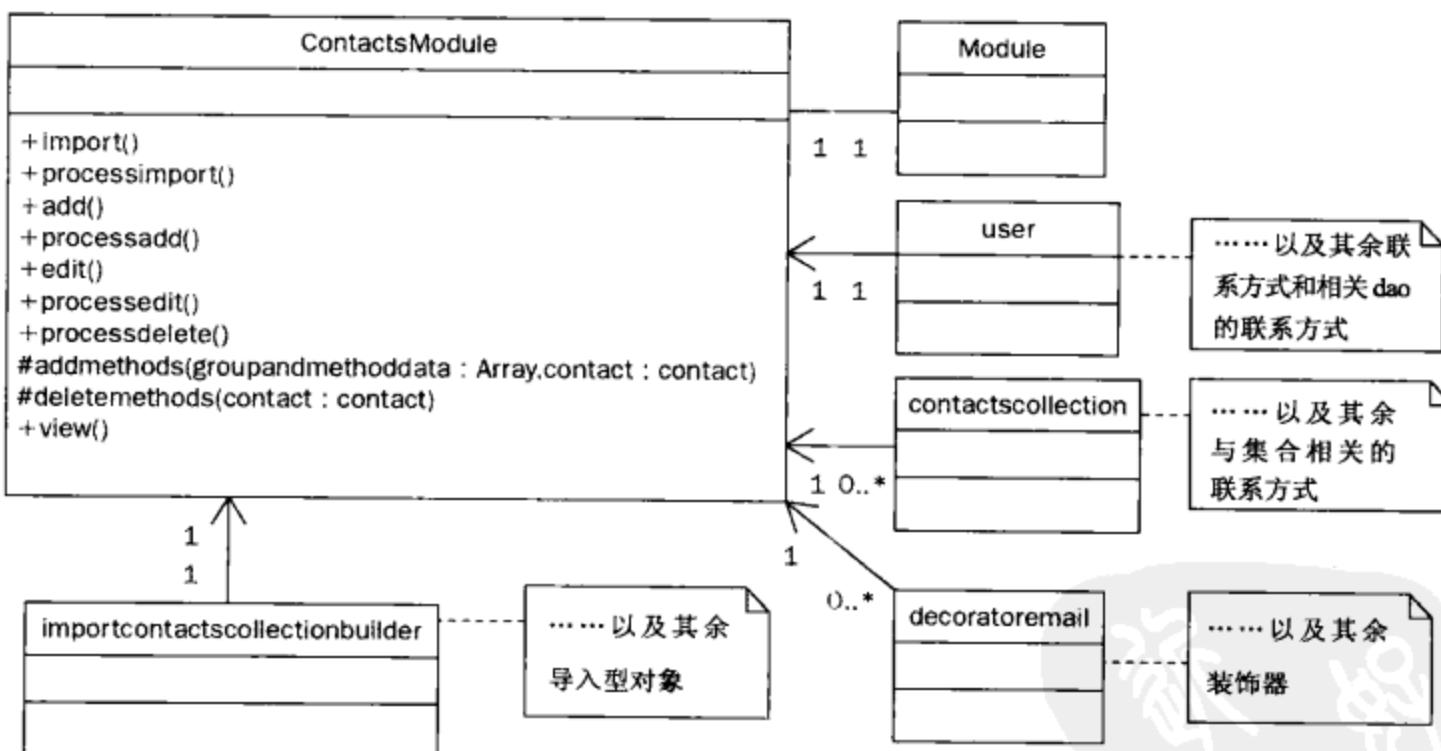


图 21-12

可以预见，ContactsModule 是 Module 对象的一个实例。它具有 8 个公共方法。这些方法能够显示和处理添加、编辑和删除操作；还能够接受和处理要导入的联系方式；最后，view( )方法被用于显示单独的联系方式。这个模块具有两个受保护方法：addmethods( )和 deletemethods( )。

`addmethods()`方法接受两个参数。一个参数是应用程序要为其添加方法的当前 `Contact`，另一个参数是分组与方法信息的数组。我的想法是添加方法界面应当允许用户指定一个分组名以及至少一个方法。因为可能存在一个或多个分组，所以最好的处理方式似乎是数组。HTML 输入字段的名字应当使用一对括号向 PHP 指示一个数组。

`deletemethods()`方法接受一个 `Contact` 对象，该方法会先删除与这个 `Contact` 相关联的所有联系方法，然后再删除所有分组。最初，我计划发送具有请求的某种指示标志或 ID 来专门处理特定方法的删除。但是经过深入分析，我认为这种方式的开销过大。在编辑期间，我会提供所有的现有信息。随后，所有的现有信息可以在更新期间被删除，取而代之的是提交的信息。

此外，其他很多类都与这个模块相关联。指定的 `User dao` 需要知道正在处理的 `Contact` 的所有者。`contactscollection` 对象被用于显示联系方式的完整列表。`Importcontactscollection_builder` 对象也可以被用于处理 Outlook 联系方式导入。最后，为联系方法信息添加功能性的装饰器将会用在 `view()` 方法中。这些装饰器实现了能够通过在应用程序内部直接单击链接来使用联系方式信息的需求。

### 21.3.2 联系方式信息关系

我在前面部分暗示过下列项的复杂关系。现在可通过如图 21-13 所示的 UML 来了解这种关系。

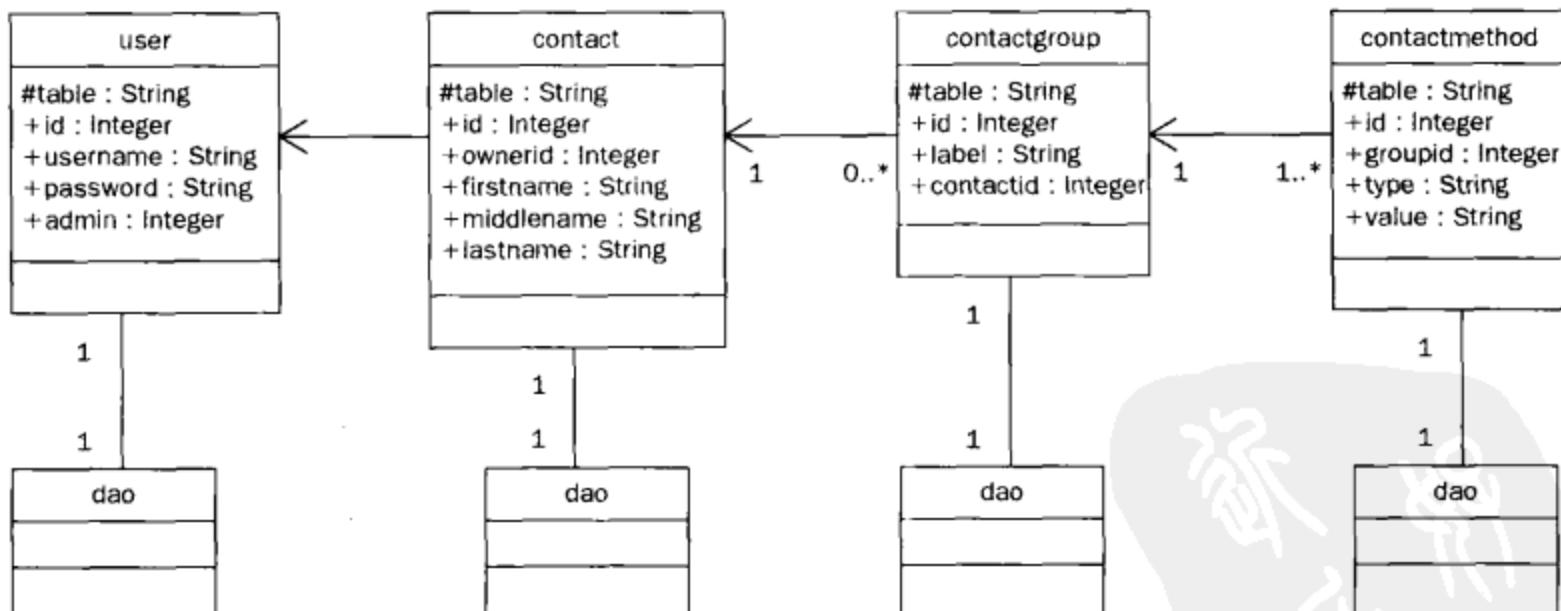


图 21-13

正如我所提到的，这些 `dao` 子类的任何公共特性都应当是推断的。它们会引用数据库中的列，并且不会在相应类自身中被定义。

一个 User 对象能够拥有任意数量的 Contact 对象。只要指定分组至少具有一个 contact-method 对象，那么一个 Contact 对象就能够属有任意数量的 contactgroup 对象。如果指定分组中不能存在联系方法，那么就不会创建 contactgroup。一个 contactgroup 对象能够拥有一个或多个 contactmethod 对象。

### 21.3.3 导入联系方式

在这个应用程序的创建期间，没有任何联系方式能够被立即用于测试。因此，计划中的下一个逻辑步骤是确定如何创建联系方式。

应用程序预先填充的、用于测试的并且适合的联系方式来源是 Outlook 联系方式。Contact-Module 对象的 processimport() 公共方法会处理这个联系方式来源。

因为应用程序处理 dao 子对象的集合，所以我需要通过某个对象将联系方式的导入转变为集合。由于导入格式是复杂的，并且与 dao 集合大不相同，因此我准备使用一个基于生成器模式的对象。导入处理方法不必考虑如何获取 dao 集合，而是只需要知道具有一个 dao 集合。生成器会接受导入的信息（此时是来自 Outlook 的 CSV 文件的一个字符串），并且将信息指派给它的受保护属性 importedstring。buildcollection() 公共方法能够返回联系方式信息的集合。

查看来自 Outlook 的 CSV 文件中的数据之后，我注意到用于独特联系方式信息和联系方式分组信息的存储机制与应用程序的方法是不同的。名字、中名和姓氏只是不同的命名方式，因此可以使用一个基于适配器设计模式的简单对象。不过，分组略为麻烦。如果存在分组，那么可以尝试使用一个解释器来确定具体的分组（见图 21-14）。

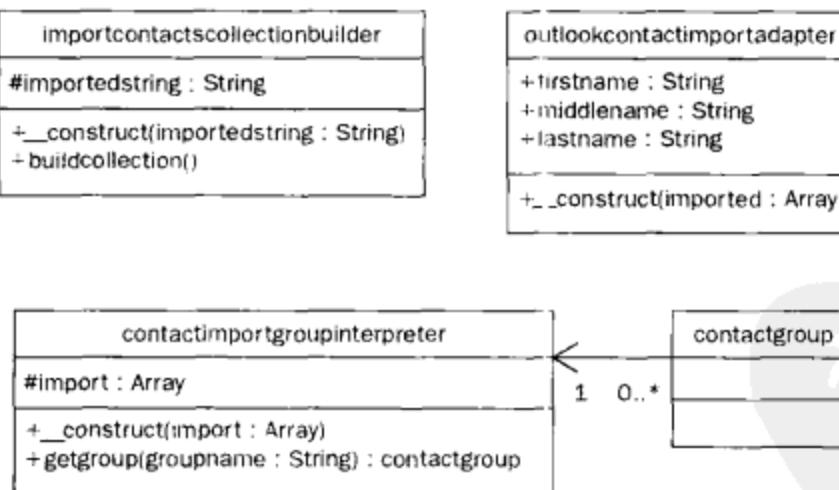


图 21-14

我的想法是将 outlookcontactimportadapter 对象作为针对 User dao 的适配器。因此，这个对象需要 firstname、middlename 和 lastname 公共变量。因为这不是一个真正的 dao 对象，所以会定义这些属性，并且不会推断它们。适配器的构造函数会接受一个表示被导入联系

### 第 III 部分 PHP 设计案例分析

方式的数组，这个数组可能是来自 importcontactsbuilder 的输出。我认为，在没有应用适配器之前，来自 importcontactsbuilder 的联系方式不是真正的联系方式。

通过为信息列添加分组名前缀，CSV 文件存储了分组的信息。因此，对于电话号码来说，存在下列一些变化形式：Home Phone、Business Phone 和 Other Phone。解释器会以数组格式接受这种联系方式信息(与刚才计划的适配器类似)，并且基于为公共方法 getgroup() 提供的名字返回一个 contactgroup。

解释器需要根据联系方式信息和接收的参数做出一些判断。首先，如果传递的参数是 Home，那么就应当查找诸如 Home Phone 之类的字符串，以便确定是否存在可能的 Home contactgroup。如果传递入解释器的参数总是 Home 和 Business，那么由于 CSV 文件中始终存在相关的列，解释器总是会返回两个分组。不过，这些列实际上可能不具有任何联系方式信息。解释器的下一个判决是需要判断是否应当真正创建 contactgroup。这个判决基于一个 contactgroup 必须具有至少一个联系方法的需求。如果“Home”相关列不具有任何指定的信息，那么就不会创建 Home contactgroup。

因为解释器已经理解了各种分组，所以它似乎也最适用于创建联系方法。解释器会根据分组中的定义和创建生成导入方法。

介绍完导入联系方式后，下一个步骤是向用户实际显示联系方式。

#### 21.3.4 查看联系方式

IndexModule 引用了 contactsbuilder 类。为了显示这个信息(见图 21-15)，我准备使用 contactmodule 对象的 view() 方法。因为可能存在 Contact dao 对象的集合，所以需要创建相应的 daocollection 对象。

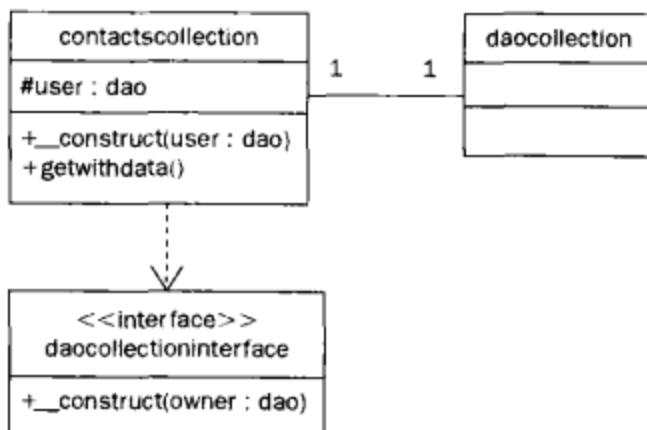


图 21-15

contactsbuilder 对象与 userscollection 对象非常相似，它扩展了 daocollection 对象并实现了 daocollectioninterface。两者的差异是 contactsbuilder 构造函数的参数是一个 User dao 对象，这个 User 是 contact 对象集合的所有者。getwithdata() 方法会使用 User 对象中的

某些标识信息来开发一个查询。通过查看 Contact dao，我发现使用 User dao 对象的 ID 属性来构建这种关系是最合理的。

既然能够处理联系方式的集合，那么就需要关注于单个 Contact 及其分组和方法。前面已经说明了，一个 Contact 能够具有任意数量与之相关联的 contactgroup 对象。因此，必须存在一个 contactgroupscollection 对象(见图 21-16)。

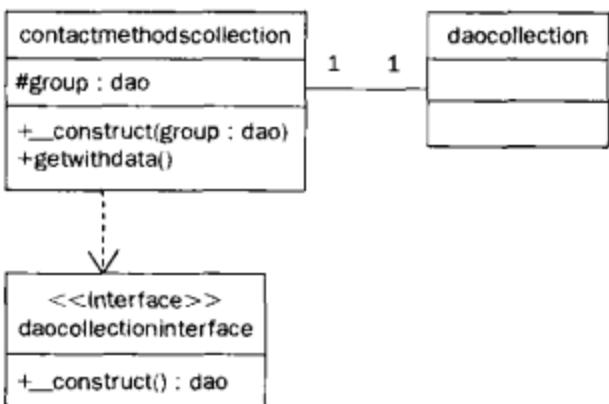


图 21-16

因为扩展了 `daocollection` 对象并实现了 `daocollecitoninterface`，所以这个对象与其他集合是类似的，其特别之处是构造函数的 `dao` 对象。此时，`dao` 对象是一个 `contact dao`，它表示集合中所有 `contactgroup` 对象的所有者。通过查看前一个 UML 图中的关系，我计划添加 `getwithdata()` 方法，从而利用 `contact dao` 对象的 `id` 公共属性(而不是 `contactgroup dao` 对象的 `contactid` 属性)。

这个关系的最后一个构建块是 `contactmethod dao`。与处理的其他所有 `dao` 一样，我也会创建一个相应的集合对象(见图 21-17)。

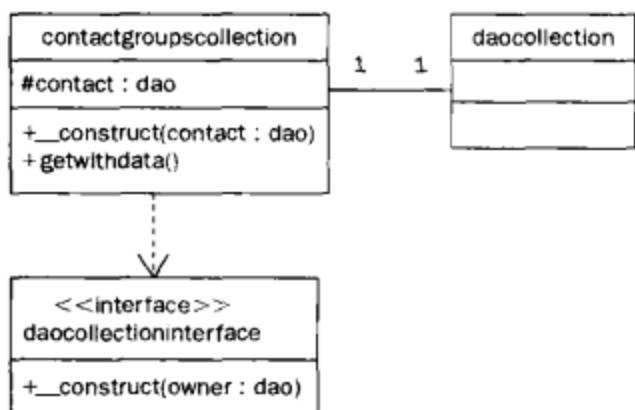


图 21-17

像预计的一样，`contactmethodscollection` 对象扩展了 `daocollection` 对象，并且实现了 `daocolle citoninterface`。构造函数接受一个 `contactgroup dao` 对象。`getwithdata()` 方法会使用 `contactgroup dao` 对象的 `id` 属性(而不是 `contactmethod dao` 对象的 `contactgroupid` 属性)。

回顾第 21.3.1 节，读者应当非常明显的看到：ContactsModule 对象的 processadd()、processedit() 和 processdelete() 方法易于构建适当的关系。因为具体功能被构建入每个 dao 对象，所以插入数据也应当十分容易。最后，管理集合应当非常简单，其原因在于 daocollection 对象 Iterator 接口的实现。

我在说明 ContactsModule 的 UML 图中标记了一个对以装饰器设计模式命名的对象的引用。应用程序的需求是确保来自 Web 页面的交互能够使用这些信息。这意味着用户可以单击到某个 Web 站点的链接，而不必将该站点复制和粘贴入地址栏。同样地，用户也能够通过单击某个电子邮件地址来发送信息。

因为我希望修改显示的信息(但是并非实际的内容)，所以装饰器设计模式是非常不错的选择。这些对象会接受信息，并且添加某种功能性或进行标记。

回过头来，我注意到需求文档没有完全明确所有期望的功能性。我准备进行某些讨论，并且针对所需的功能性给出一些设想。

下面列出了我开发的交互：

- mailto: 用于电子邮件的链接
- 用于 Web 站点的可单击链接
- 用于社交网络的可单击链接
- 用于地址的地图链接
- 使用移动电话的直接拨号

使用移动电话的直接拨号看上去是唯一具有一定复杂性的交互。我必须确保这个装饰器了解当前用户所使用的视图。其他装饰器则只需要向指定数据添加额外的信息。

为了使基于装饰器设计模式的所有对象保持相同，我会创建这些对象都可以实现的接口 decoratorinterface。ContactsModule 对象的 view() 方法负责将图 21-18 中的所有装饰器应用于 contactmethod 对象的输出。

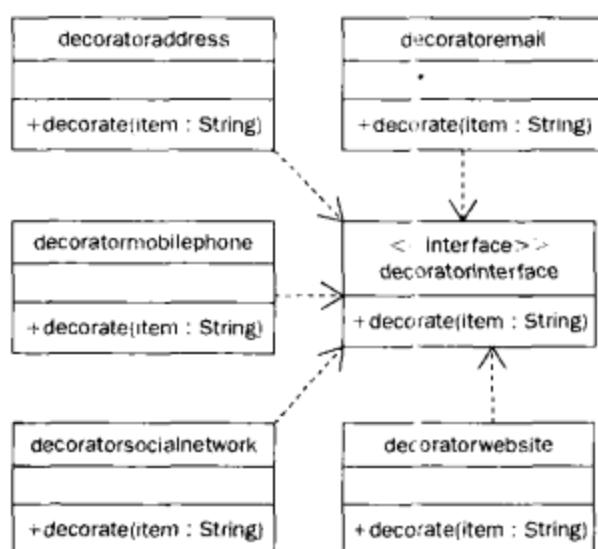


图 21-18

设计完装饰器后，我认为自己已经到达计划的结束部分。正如前面所提到的，我计划将这些 UML 作为实际编程的蓝图和模式使用，这也正是第 22 章将要讨论的内容。回顾这些 UML 图，我知道自己面临着一项复杂的任务。如果所有对象和方法能够按我计划的那样成为现实，那么我会觉得非常惊奇。不过，我将尽力执行自己的计划。如果具体情况有变化，那么我会在编程中进行相应的修改，并且遵循所计划的方式。纯粹主义者可能具有这样的主张：只要在编程期间发现体系基结构上的变化，UML 图就应当被更新。虽然我也看到了这一点，但它并不属于本书所讨论的范畴。

下一个步骤是非常有趣的，也就是从设计进入实际的应用程序。

## 21.4 本章小结

本章探讨了为将需求文档转变为应用程序而开发计划的计划编制步骤。本章讨论了应用程序的 3 个主要部分及其使用。第一个部分是应用程序的核心，通过 UML 图说明了如何创建用于应用程序的逻辑和视图。在应用程序的用户交互和管理部分中，出现了更多的对象。本章用 UML 图说明了数据访问对象、集合与数据库操纵对象，此外还定义了与 Web 访问者交互的主模块模式。最后涉及的是应用程序最复杂的部分。这个部分描述了用户与联系方式的关系，并且通过 UML 图说明了联系方式、分组和联系方法对象关系。此外，本章还介绍了针对公司现有联系方式系统的导入方法，同时定义了提供与联系方法进行必要交互的装饰器。第 22 章将利用上述所有信息逐步完成应用程序的编码过程。





# 第22章

## 应用程序编程

本章的内容是所有编程人员一直期盼的：用于构建应用程序的实际编程。目前，需求已被收集和固化，应用程序计划和 UML 图也已完成。

本章的第一部分简要说明了如何准备应用程序所需的信息。在这个具体的应用程序中，所需的只是收集来自 Outlook 的导出数据。随后，按照第 21 章的计划和 UML 图，我将开始对使用已指定设计模式的每个对象进行编程。除了 PHP 编程之外，本章也会介绍涉及 JavaScript、HTML 和 MySQL 的编程细节。不过，我们关注的重点是 PHP。这实际上意味着只对非 PHP 编程进行简要的阐述，这些支持代码并不是本书的重点。

第 23 章将分析整个程序的 bug 或优化处理。现在，让我们首先进入信息准备阶段。

### 22.1 信息准备

需求文档规定：应用程序的任何用户都需要从公司现有的联系方式管理软件(也就是 Microsoft Outlook)中导入自己的联系方式。在早先的计划中，我将这个过程假定为逗号分割值导出。

在这个案例分析中，公司使用了 Microsoft Windows 平台上的 Outlook 2007。为了生成测试 CSV 文件，我进行了下列操作：

- (1) 启动 Microsoft Outlook 2007。
- (2) 单击 File Import and Export... 按钮。
- (3) 在 Export Wizard 窗口中选择 Export to a file 选项。
- (4) 单击 Next 按钮。
- (5) 选择 Comma Separated Values (DOS) 选项。
- (6) 单击 Next 按钮。
- (7) 选择 Personal Folders | Contacts 命令。

(8) 单击 Next 按钮。

(9) 在 Save exported file as 输入框中输入 CSV 文件的位置和名字。我在这里输入了“c:\test.CSV”。

(10) 单击 Next 按钮。

(11) 单击 Finish 按钮完成操作。

操作完成后系统会创建一个新的测试 CSV 文件。应用程序的用户在能够选择导入其联系方式之前需要完成上述操作。

准备用于测试的实际导出文件具有的列远远多于本书所采用的列。在本书的辅助 Web 站点上可以下载这个文件以及其他程序文件。不过，为了便于编程期间进行参考，我从文件中节选了下面的记录(见表 22-1 和表 22-2)。

表 22-1 第一个记录节选

名 字	姓 氏	家 庭 住 址	居 住 城 市	居 住 州	住 址 邮 政 编 码
John	Smith	123 Fourth Street	Beverly Hills	CA	90210

表 22-2 第二个记录节选

名	中名	姓	公司	职务	公司 地址	公司 所在城市	公司 所在州	公司 邮政编码
William	Henry	Gates	Bill& Melinda Gates Foundation	董事	PO Box 23350	Seattle	WA	98102

节选的记录中存在许多列。不过，这两条记录说明某些联系方式可能具有家庭信息或公司信息。联系方式可以具有上述所有字段。

既然已收集了构建应用程序所需的唯一外部数据，那么接下来就可以进行应用程序编程了。

## 22.2 应用程序编程

就像计划阶段一样，本章的编程部分也会被分为若干步骤。这些步骤几乎与上一章所讨论的步骤完全相同。前面曾经提到过，UML 图为实现应用程序的编程绘制了极好的蓝图。

下面是应用程序编程的 3 个主要部分。

- 主核心
- 用户交互与管理
- 联系方式管理

主核心部分涉及需要创建的对象的 UML 图。此外，该部分还会描述利用这些对象的文件系统布局以及其他所有附加配置选项。用户交互与管理部分继续采用了相同的样式。UML 图会被转换为 PHP 语言。其他测试过程和配置也会有所介绍。最后，采用相同模式的联系方式管理过程将结束本章的应用程序编程小节。联系方式管理部分会利用前面创建的 CSV 文件。

### 22.2.1 应用程序核心的编程

启动用户和联系方式部分中的任何功能都需要应用程序的核心。应用程序的核心必须具有 Web 站点功能，但是并不过多展现基准章节中所指定的任何设计模式。请读者耐心地跟随我创建应用程序的核心，然后再进入说明 PHP 设计模式的部分。

公司具体采用的 Web 服务器是一些使用 PHP 5.2.5、Apache 2.2.11 和 MySQL 5.1.31 的服务页面。应用程序被设计为至少能在先前列出软件的 Major/Minor 版本中运作。提及这一点的主要原因是接下来的步骤涉及一个 Apache 配置文件。

我期望应用程序位于某个 Apache 实例的 Document Root 目录中。如果已有一个 Web 站点运行在该实例中，那么就创建新的 Virtual Host 目录并将具体代码置于此处。如果应用程序位于某个子目录内，那么具体代码就要在任何不能正常运作的地方使用\$\_SERVER['DOCUMENT\_ROOT']值。

#### 1. 为专用 URL 生成.htaccess 文件

在这个应用程序中，我计划采用人工可读的 URL。这种 URL 机制也被用于直接引用特定的模块功能名。计划期间曾经提到过，控制器对象将负责这些调用的实际创建和执行，在应用程序根部，我会添加下面的.htaccess 文件。这个文件的内容如下所示：

```
<IfModule !mod_rewrite.c>
    ErrorDocument 500 'mod_rewrite must be enabled'
</IfModule>
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ /index.php?u=$1
```

这个文件实现了两个功能。首先，它确保 mod\_rewrite 功能在服务器上是活动的。如果不是活动的，那么就会生成一个适当的错误。该文件的另一个功能是接受不直接针对文

### 第 III 部分 PHP 设计案例分析

件或目录的请求并重写请求，从而将该请求作为 Get 参数调用主文件 index.php。index.php 文件可以使用控制器，而且能够获取\$\_GET['u']变量所请求的动作。这个.htaccess 文件位于应用程序的根目录。

**注意：**

如果试图通过浏览器查看应用程序时接收到“404 - Not Found”错误，那么.htaccess 文件的解析可能存在问题。确认这个文件位于应用程序的根目录。此外，验证 Apache 配置允许重写。要查看 Apache 的.htaccess 配置选项的更多信息，读者可以访问 <http://httpd.apache.org/docs/2.0/mod/core.html#allowoverride>。

## 2. 管理包含、异常和控制器的 index.php

从计划阶段开始，我就知道需要使用一些包含文件。针对构建核心所使用的通用功能，我准备创建一个名为 includes 的子目录。我也会创建一个用于 dao 对象的子目录 dataobjects。最后，我还要为所有模块创建一个名为 modules 的子目录。

接下来将生成文件 index.php。.htaccess 文件会将所有请求重定向至这个文件。index.php 文件会使用控制器对象。我计划将这个对象放入 includes 目录内部。我已经决定使用某种自动载入机制来包含刚刚创建的 3 个子目录中的所有文件。index.php 文件需要只请求包含存在自动载入逻辑的文件。此外，我还希望能够使用 PHP 中的会话机制，而 index.php 文件同样可以提供帮助。

目前 index.php 文件的内容如下所示：

```
<?php
require 'includes/autoload.php';
session_start();
```

针对 autoloader.php 文件，我使用了 SPL Autoload 提供的功能。includes/autoload.php 的内容如下所示：

```
<?php
class autoloader
{
    public static function moduleautolcader($class)
    {
        $path = $_SERVER['DOCUMENT_ROOT'] . "/modules/{$class}.php";
        if (is_readable($path)) require $path;
    }

    public static function daoautoloader($class)
```

```

{
    $path = $_SERVER['DOCUMENT_ROOT'] . "/dataobjects/{$class}.php";
    if (is_readable($path)) require $path;
}
public static function includesautoloader($class)
{
    $path = $_SERVER['DOCUMENT_ROOT'] . "/includes/{$class}.php";
    if (is_readable($path)) require $path;
}
}
spl_autoload_register('autoloader::includesautoloader');
spl_autoload_register('autoloader::daoadutoloader');
spl_autoload_register('autoloader::moduleautoloader');

```

autoloader 类包含 3 个静态方法，每个方法都负责尝试定位一种特定类型的包含文件。名为 moduleautoloader( ) 的方法尝试载入模块。这种描述性名字能够自己解释其功能性。

确定某个类所在位置的方法十分简单。具体的类存在于以该类命名的 PHP 文件内部。因为 SPL Autoload 功能性允许在堆栈中添加多个自动载入器功能，那么应当仔细确认每个方法都不会生成致命错误。如果某个方法能够发现所期望的类(无论是包含、模块或数据对象)，那么都会调用 require( ) 方法，并且继续代码执行。只有 3 个方法都执行后仍未发现以指定类命名的文件，才会生成一个致命错误。

在类声明后的文件结束部分，都会使用 spl\_autoload\_register( ) 方法注册 3 个自动载入器静态方法。

回到 index.php 文件，接下来的步骤是创建 controller 对象的一个实例。不过，因为知道这个对象不仅处理被请求的动作，而且还根据具体请求生成可用的其他查询信息，所以我会将该对象存储在任何方法都能够对其进行访问的位置。

我准备创建一个名为 lib 的通用类，这个类会包含一组静态方法。我将使用这些方法设置和获取信息、更改对象，或者只是在不考虑具体环境的情况下多次执行某个动作。被创建的/includes/lib.php 文件具有如下所示的内容：

```

class lib
{
    const SETTING_AN_ARRAY = TRUE;

    const NO_PERSISTENT_STORAGE = FALSE;

    public static function getitem($name, $persist = TRUE)
    {
        $return = NULL;

```

```

        if (isset($_SESSION[$name])) {
            $return = $_SESSION[$name];
            if (!$persist) unset($_SESSION[$name]);
        }

        return $return;
    }

    public static function setitem($name, $value, $array = false)
    {
        if ($array) {
            if (!isset($_SESSION[$name])) {
                $_SESSION[$name] = array();
                $_SESSION[$name][] = $value;
            }
        } else {
            $_SESSION[$name] = $value;
        }
    }
}

```

`getitem()` 和 `setitem()` 方法能够直接处理 `$_SESSION` 变量，因此不必再进行相关操作。要查看的第一个方法是 `setitem()`，该方法接受下列 3 个参数：`$name`(表示用于信息的键)、`$value`(要存储的信息)，以及`$array`(一个布尔变量，它指示具体条目是否确实为要添加入以`$name` 命名的数组的另一个值)。在默认情况下，不需要执行这个方法。不过，有时该方法也是有用的。因此，`$array` 在方法声明中被设置为 `false`。`setitem()` 方法只是简单地将值指派给 PHP 会话中的某个指定键。

`getitem()` 方法被用于检索可能使用 `setitem()` 方法设置的信息，该方法接受下列两个参数：`$name`(表示检索值的名称)，以及`$persist`(一个布尔变量，它指示具体条目被检索后是否应当继续存在)。`getitem()` 方法的返回值初始设置为 `NULL`。如果检索值存在于 PHP 会话中，那么会将返回值设置为这个值。最后，如果`$persist` 为 `false`，那么 `$_SESSION` 变量中的键就会被复位。

这个类特有的另外两点是常量声明。为了努力编写更易理解的代码，应定义常量来替换使用默认设置的布尔参数。假设通过使用某个数组值和进行不保持持久性的检索来实现一个对 `setitem()` 的调用，那么通常可采用下面的形式：

```
lib::setitem('test', 123, TRUE);
```

```
$test_once = lib::getitem('test', FALSE);
```

虽然上面的代码能够实现我们希望的功能，但是并不容易理解。此时，使用定义的常量来实现相同功能的代码如下所示：

```
lib::setitem('test', 123, lib::SETTING_AN_ARRAY);
$test_once = lib::getitem('test', lib::NO_PERSISTENT_STORAGE);
```

我发现这些代码更易于理解和调试。

接下来，在 index.php 文件中添加下列代码行：

```
lib::setitem('controller', new controller($_GET['u']));
```

使用 lib 类的 setitem( )方法来存储新 controller 对象的一个实例。通过访问 htaccess 文件提供的\$\_GET['u']字符串就可以构造这个新的 controller 对象。我期望自动载入器类的 includesAutoloader( )方法能够发现指定文件/includes/controller.php，controller 的构造函数将包含下面的内容：

```
<?php
class controller
{
    protected $parts;
    public $params;

    public function __construct($urlString)
    {
        $urlString = strtolower($urlString);

        if (substr($urlString, -1, 1) == '/') {
            $urlString = substr($urlString, 0, strlen($urlString) - 1);
        }

        $parts = explode('/', $urlString);
        if (empty($parts[0])) {
            $parts[0] = 'index';
        }
        if (empty($parts[1])) {
            $parts[1] = 'defaultaction';
        }

        $this->parts = $parts;
        $this->sectionaction = $parts[0] . '/' . $parts[1];
        array_shift($parts);
    }
}
```

### 第 III 部分 PHP 设计案例分析

```

        array_shift($parts);
        $this->params = $parts;
    }

    public function render()
    {
        if (!class_exists($this->parts[0])) {
            throw new SectionDoesntExistException("{$this->parts[0]} is".
                "not a valid module.");
        }

        if (!method_exists($this->parts[0], $this->parts[1])) {
            throw new ActionDoesntExistException("{$this->parts[1]} of".
                "module {$this->parts[0]} is not a valid action.");
        }

        $called = call_user_func_array(array( new $this->parts[0],
            $this->parts[1]), array($parts));

        if ($called === FALSE) {
            throw new ActionFailedException("{$this->parts[1]} of section".
                "{$this->parts[0]} failed to execute properly.");
        }
    }
}

```

控制器具有图 21-1 中 UML 图所指示的两个公共方法：\_\_construct() 和 render()。

\_\_construct() 接受单个参数\$urlstring。因为该方法负责接受一个字符串并执行某个模块的方法，所以需要对这个字符串进行某些操纵。首先，这个字符串会被转换为小写字母。接下来，任何结束正斜杠都会被去除。最后，这个字符串会通过使用正斜杠分隔符被分割入一个数组。因为存在没有任何字符串进入 Web 站点根部的可能性，所以 index 被定义为处理未给出字符串的请求的默认模块。用户还可能在没有指定动作的时候定位至某个模块的 URL。如果出现这种情况，就会将操作设置为 defaultaction。为了稍后在 render() 方法中加以使用，\$parts 数组会被设置为控制器对象的受保护属性\$params。\$parts 数组的模块和动作部分会被删除，剩余部分则被添加入任何模块的方法都能够使用的公共数组\$params。需要记住的是，因为使用 setitem() 方法将控制器对象的这个实例指派给具体会话，所以这些方法都能够检索该实例并访问公共属性\$params。

render() 方法的工作相当简单。这个方法首先通过针对\$params 数组的第一项调用 PHP 的 class\_exist() 来判断指定模块是否存在。为了试图发现指定类，class\_exist() 会调用每个自动

载入器。如果指定类不存在，那么就会抛出一个定制异常。通过访问表示模块的数组元素与表示动作的数组元素，`method_exists()`方法也会执行相同的处理过程。需要记住的是，每个动作都直接与模块的一个方法相互关联。如果没有发现指定的方法，那么就会抛出一个定制异常。最后，这个模块和方法会被执行。如果执行失败，那么也会抛出一个定制异常。

进行其他介绍之前，我希望通过创建具有下面内容的`/includes/exceptions.php`文件来定义在控制器中使用的所有定制异常：

```
<?php
class SectionDoesntExistException extends Exception {}
class ActionDoesntExistException extends Exception {}
class ActionFailedException extends Exception {}
class InternalException extends Exception {}
```

添加名为`InternalException`的第四种异常，并且计划在编程中出现不期望的情况时使用。当然，我希望根本不使用这种异常。最后，因为这个文件并不遵循标准包含文件的命名约定(只是简单地为上述异常创建 4 个几乎全空的文件)，所以需要在`index.php`文件中添加如下所示的代码行：

```
require 'includes/exceptions.php';
```

这将确保控制器能够使用上述异常，并且其他类都可以利用`InternalException`。

### 3. 创建视图对象

结束控制器对象的编程后，下一个步骤是创建`view`对象。根据图 21-2 中的 UML 图，该对象将具有 4 个公共方法。`start()`方法利用了 PHP 的`ob_start()`功能，`end()`方法则利用了内容收集机制。在这两个步骤中间，控制器需要调用指定的模块和动作。考虑到这点，应在`index.php`文件中添加如下所示的代码行：

```
$view = new view();
lib::getitem('controller')->render();
$content = $view->finish();
```

上面的代码会创建`view`对象的一个新实例，并且访问被存储的控制器对象和调用`render()`方法，最后将捕获的输出置入`$content`变量。`view`对象由包含文件自动载入器载入。`/includes/view.php`文件目前的内容如下所示：

```
class view
{
    public function __construct()
    {
```

### 第 III 部分 PHP 设计案例分析

```

        ob_start();
    }

    public function finish()
{
    $content = ob_get_clean();
    return $content;
}
}

```

这是一种非常简单的模块动作输出收集方式。

下一个添加至视图对象的功能是静态方法 `setviewtype()`。我会使用 `show()` 方法访问名为 `$viewtype` 的受保护静态变量，以便查找所需的包含文件。`setviewtype()` 方法会在首次执行 `show()` 时被执行。将在视图类中添加下面的方法：

```

protected static function setviewtype()
{
    switch (TRUE) {
        case strpos($_SERVER['HTTP_USER_AGENT'], 'Windows CE')
            != FALSE:
            self::$viewtype = 'mobile';
            break;
        default:
            self::$viewtype = 'default';
    }
}

```

这个方法被构建用于确定基于 HTTP User Agent 的视图。因为需要支持 Windows Mobile 平台，所以应寻找特定的标识符。如果发现了特定的标识符，那么就会将视图设置为移动的。否则，就会采用默认的视图类型。这个方法的结构更便于通过在开关语句中加入其他分支来允许添加更多的视图。

#### 注意：

在这个特定情况中，HTTP User Agent 头部提供了标识视图类型的适当信息。不过，实际情况并非往往如此。用户的浏览器可能更改或错误报告 HTTP User Agent。此外，将来的 Windows Mobile 版本可能不会在 HTTP User Agent 头部中包含字符串 Windows CE。目前，因为访问应用程序的浏览器的受控状态(分配给标准移动设备的业务)，所以这个方法能够准确地工作。如果应用程序被分发给其他参与方，那么这个方法可能是需要重访的代码部分。

`show()`方法存在一些必须进行的有趣判断。最后决定尽可能使用复制的视图。在大多数情况下，如果对特有的 HTML 进行编程，那么移动视图与浏览器视图之间的最大差异应当是基于 CSS 的。我也可能在移动版本中不提供某些功能。因此，我的想法是基于应用程序构建一个浏览器。随后，如果请求默认浏览器视图之外的视图，那么应查看视图的类型。如果请求的视图不存在，那么将继续提供默认的试图。通过这种方式，能够减少对视图的不必要的复制。

`show()`方法应当也能够自动基于`$viewtype` 变量在文件系统中定位视图文件。因此，决定在应用程序根部创建一个名为 `views` 的子目录。这个目录包含了用于不同视图类型的子目录。在应用程序的第一个迭代中，应当存在下列两个子目录：`default` 和 `mobile`。

最后，`show()`方法应当能够允许对传递入包含视图文件的变量进行求值。因为求值是即时进行的，所以还应当使用输出缓冲(与 `view` 对象的 `start()` 和 `end()` 方法非常相似)。下面给出了静态方法 `show()` 的代码：

```
public static function show($location, $params = array())
{
    if (empty(self::$viewtype)) {
        self::setviewtype();
    }

    $views = array();

    if (self::$viewtype != 'default') {
        $views[] = $_SERVER['DOCUMENT_ROOT'] . '/views/' .
            self::$viewtype .
            '/' . $location . '.php';
    }
    $views[] = $_SERVER['DOCUMENT_ROOT'] . '/views/default/' .
        $location . '.php';

    $content = '';

    foreach ($views as $viewlocation) {
        if (is_readable($viewlocation)) {
            $view = $params;

            ob_start();
            include $viewlocation;
            $content = ob_get_clean();
            break;
        }
    }
}
```

```

    return $content;
}

```

`show()`方法接受两个参数：视图的位置以及传递至视图的任意参数。位置变量可以只是某个视图的名字，但是我感觉使用文件夹中分组视图的组织结构方法可能更好一些。位置字符串会镜像这个组织结构。例如：

```

echo show('login/form');
// this would reference /views/default/login/form.php

```

`show()`方法的第一个步骤是查看是否设置了视图类型。如果没有设置，那么就会执行先前创建的 `setviewtype()` 方法。

接下来的步骤是创建一个视图位置数组。这个数组将会被循环遍历，以便查找到要使用的包含文件。如果当前视图不是默认的，那么表示视图类型的一个路径就会被添加入该数组。除此之外，数组也会添加默认的路径。

随后，每个可能的视图都会被遍历。具体函数会查看路径是否可读取。如果发现路径和文件是可用的，那么就会打开输出缓冲，`$params` 数组作为一个局部变量被添加和包含，并且 `$content` 接收输出。接下来，指定函数将不再循环任何实例。这种循环类型解决了只为移动视图创建特定视图的判定。

如果没有发现指定的视图，那么 `show()` 方法绝不会生成致命错误或异常，注意到这一点十分重要。因为我觉得可以将其作为粗心编程而导致错误的例子进行讨论，所以决定到最后来阐述这个问题。通过这种方式，我可以添加目前不存在而占位符稍后能够被更改的额外视图。如果企业的开发团队变得更为庞大，那么首席架构师可以生成视图位置，同时指派其他团队成员具体创建指定的视图。最后，这个方法会返回 `$content` 内容。

添加至 `index.php` 文件的最后一个步骤是通过使用 `view` 对象的 `show()` 方法的调用来显示 `controller` 对象 `render()` 方法的内容。因为前面已经不断地逐步构建了 `index.php` 文件，所以该文件最后的内容如下所示：

```

<?php
require 'includes/autoload.php';
require 'includes/exceptions.php';
session_start();

lib::setItem('controller', new controller($_GET['u']));

$view = new view();
lib::getItem('controller')->render();
$content = $view->finish();

```

```
echo view::show('shell', array('body'=>$content));
```

#### 4. 创建主视图

现在我准备创建实际的显示输出。添加一个名为 assets 的新子目录，这个目录是所有 CSS 和 JavaScript 文件的存放位置。

index.php 文件的最后一行引用了名为 shell 的视图位置，它是 Web 站点的主外壳。这个文件具有下列两个位置：用于浏览器的默认位置；用于移动视图的简化版本的位置。

/views/default/shell.php 文件的内容如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" type="text/css" href="/assets/main.css"/>
    <title> Acme Company Contact Manager </title>
    <script type="text/javascript" src="/assets/jquery-1.3.2.min.js">
        </script>
</head>
<body>
    <div id="header"> <?php echo view::show('standard/header');?></div>
    <div id="body">
        <?php echo $view['body'];?>
    </div>
    <div id="footer"> <?php echo view::show('standard/footer');?></div>
</body>
</html>
```

/views/mobile/shell.php 文件的内容如下所示：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <link rel="stylesheet" type="text/css" href="/assets/main.css"/>
    <title> <?php echo view::show('standard/title');?></title>
</head>
<body>
    <div id="header" > < ?php echo view::show('standard/header'); ? > </div>
    <div id="body" >
        <?php echo $view['body'];?>
    </div>
</body>
</html>
```

### 第 III 部分 PHP 设计案例分析

```
</div>
<div id="footer"><?php echo view::show('standard/footer'); ?></div>
</body>
</html>
```

#### 注意：

在这个特定示例中，两个文件之间的唯一区别是移动视图中缺少 JQuery。因此，讨论将基于默认的 shell.php 文件。

这个视图构建了标准的 Web 页面。它包含了用于标题的 HTML 标记(包括 CSS 和 JavaScript)。在 HTML 文件关闭之前，其内容还包括题头、主体和页脚。

读者需要注意的第一点是对\$view['body']的引用。因为使用了输出缓冲，所以这些包含文件能够通过使用回应或打印来显示传递给它们的内容。这些内容可以像标准的 PHP 文件一样被添加入 HTML 中部。\$view 变量是视图对象 show( )方法使指定 PHP 包含文件可用的参数的一个局部实例。

注意对 view 对象 show( )方法的其他调用。我还准备包含一个题头页脚视图。因为 show( )方法的结构中使用了输出缓冲，所以这些对自身的递归调用是可能的。

下面给出了/assets/main.css 文件的内容(在这里可以放置一张书签，将有助于读者随时翻阅并牢记所有 CSS 声明)：

```
body {
    font: 14px tahoma, verdana, sans-serif;
    padding: 0;
    margin: 0;
    background-color: #f9f9f9;
}
table {
    border-collapse: collapse;
}
table td {
    border: 1px solid #ccc;
    padding: 3px;
}
hr {
    clear: both;
    margin: 20px 0;
}
fieldset, legend {
    padding: 3px;
    border: 1px solid #aaa;
```

```
}

legend {
    font-weight: bold;
    background-color: #ddd;
    margin-bottom: 6px;
}

fieldset {
    margin-bottom: 10px;
}

form div.row {
    clear: left;
    display: block;
    margin: 5px 0 0;
    padding: 1px 3px;
}

label {
    display: block;
    float: left;
    width: 150px;
    font-weight: bold;
    text-align: right;
    margin: 0 0 5px;
    padding: 3px 5px;
}

input {
    border: 1px solid #ccc;
    padding: 1px 3px;
    width: 200px;
    float: left;
}

select {
    float: left;
    margin-right: 2px;
}

input:focus, select:focus {
    background-color: #f9f9f9;
}

input.submitbutton {
    border: 2px solid #333;
    background-color: #369;
    color: #fff;
    cursor: pointer;
    font-weight: bold;
}
```



### 第 III 部分 PHP 设计案例分析

```
width: auto;
}
input.radio {
    width: auto;
    border: none;
    float: none;
}
.radiooption {
    margin-right: 10px;
}
.error {
    color: #f00;
}
#header {
    border-bottom: 1px solid #ccc;
}
#footer {
    text-align: center;
    clear: both;
}
#header, #footer {
    background-color: #333;
    padding: 5px;
}
#header ul {
    list-style: none;
    margin: 0px;
    padding: 0px;
}
#header li {
    display: inline;
    margin-right: 10px;
}
#header a, #header a:visited, #footer a, #footer a:visited {
    color: #eee;
}
#body {
    width: 990px;
    margin: 0px auto;
    border: solid #ccc;
    border-width: 0 1px;
    padding: 5px;
    background-color: #fff;
```

```
min-height: 500px;
}
.sidebar {
    padding: 5px;
    float: right;
}
a.featured {
    padding: 10px;
    color: #000;
    font-weight: bold;
    background-color: #6f6;
    margin: 5px;
}
a.featured:hover {
    background-color: #292;
    color: #fff;
}
.sidebar a.removal {
    background-color: #f66;
}
.contactgrouping {
    border: 1px solid #ccc;
    background-color: #f9f9ff;
    margin-bottom: 10px;
}
.contactgrouping a, .contactgrouping a:visited {
    display: block;
    font-size: 10px;
    color: #00f;
}
.contactgrouping a.addcontactgrouping,
.contactgrouping a.deletecontactgrouping {
    padding-top: 20px;
    clear: both;
}
.methodboxvaluebox {
    display: none;
}
#contactgroupingcontainer {
    display: none;
}
.hasvalue {
    display: block;
```



### 第 III 部分 PHP 设计案例分析

```

}

#browsecontacts a {
    padding: 5px;
    border: 1px solid #f4f4f4;
    display: block;
    font-weight: bold;
    color: #000;
}
#browsecontacts a:hover {
    border-color: #69f;
    background-color: #def;
}
#loginbox {
    margin: 0 auto;
    width: 500px;
}

```

因为已经在图 21-7 的 UML 图中计划了 auth 对象，所以能够从头至尾编写题头视图。但是，如果立即着手编程，那么就会遇到一个致命错误。不过，可以为如何使用计划的方法给出相当不错的假设。

/views/default/standard/header.php 文件的内容如下所示：

```

<?php
if (auth::isloggedin()) {
    $links = array('/'=>'Home',
                  '/contacts/add'=>'Add Contact',
                  '/contacts/import'=>'Import Contacts');
    if (auth::isadmin()) {
        $links['/users'] = 'User Admin';
    }
    $links['/logout'] = 'Log Out';
    echo '<ul>';
    foreach ($links as $link=>$title) {
        echo '<li><a href=" ' . $link . '">' . $title . '</a> </li> ';
    }
    echo '</ul>';
}

```

这个视图主要查看用户是否登录。如果没有登录，那么不会执行任何操作。不过，如果用户登录了系统，那么就会构建一个菜单并将其显示为无序列表。如果用户是系统管理员，那么还会在菜单中添加一个另外的链接。

这个视图的移动版本较为简洁。我承诺后面不再有这么多复制视图。/views/mobile/standard/header.php 文件的内容如下所示：

```
<?php
if (auth::isloggedin()) {
    $links = array('/'=>'Home', '/logout'=>'Log Out');

    echo '<ul>';
    foreach ($links as $link=>$title) {
        echo '<li> <a href="' . $link . '">' . $title . '</a></li>';
    }
    echo '</ul>';
}
```

这个移动视图只是没有提供用户或联系方式管理选项，其余内容都是相同的。页脚视图只定位于/views/default/standard/footer.php，该文件的内容如下所示：

```
<a href="http://www.acmecompany.com">Corporate Site </a>
| <a href="mailto:helpdesk@acmecompany.com"> Contact Helpdesk </a>
```

因为应用程序仍然无法载入一个页面，所以还需要进一步加以完善。现在，我们已经构建了应用程序的核心，并且生成了一些视图。下一部分将关注各种模块及其动作。读者能够从这个部分中看到努力工作的成果。

## 22.2.2 用户交互与管理的编程

应用程序的核心已编写完成。UML 图中的下一个步骤是构建 index 模块。尽管第 21 章中的 UML 图说明每个模块都扩展了 Module 类，但是我还未真正发现很好的扩展用法。迄今为止，我仍然无法确定扩展模块需要的一组共享方法或属性。因此，只能独立地创建每个模块。

### 1. 索引和登录模块

modules 子目录创建用于存储所有模块。/modules/index.php 文件的内容如下所示：

```
class index
{
    public function defaultaction()
    {
        if (!auth::isloggedin()) {
            lib::sendto('/login');
        }
        else {
```

第 III 部分 PHP 设计案例分析

需要记住的是，如果没有进行任何指定，那么控制器对象会将 index 模块指派为片段。此外，如果没有指定任何动作，那么具体操作就被定义为 defaultaction。正因为如此，再加上索引视图在 URI 上只采用了一个正斜杠，所以索引模块中需要的唯一方法是 defaultaction()。

`defaultaction( )`方法能够判断是显示 `contactscollection` 对象还是将用户重定向至登录界面。我会再次引用 `auth` 对象的静态方法 `isloggedin( )`。根据我的计划，该方法会返回一个布尔变量。我将在对登录过程的编程完成后再处理这个方法，其原因在于我希望先阐明如何真正登录一个用户。会在编程 `login` 模块时进行判定。完成判定之后，某个模板能够处理经过身份验证的用户，因此 `isloggedin( )` 方法将是一个简单的条件。

如下所示，在 lib 类中创建了能够处理所有重定向的另一个静态方法 sendto( )：

```
public static function sendto($url = '')  
{  
    if (empty($url)) {  
        $url = '/';  
    }  
  
    die(header('Location: ' . $url));  
}
```

这个方法只是接受一个使用 HTTP 题头重定向用户的 URL。如果没有指定任何 URL，那么用户会被重定向至索引页面。

因为发送至 sendto() 的 URL 包含了词汇 login，所以下一个要编写的模块是登录模块。目前，/modules/login.php 文件的内容如下所示：

```
class login
{
    public function defaultaction()
    {
        echo view::show('login/form');
    }
}
```

需要记住的是，因为请求 URI 只包含/login，所以必然包含 defaultaction() 公共方法，并且还会执行该方法。登录模块的默认动作是显示登录窗体。下面的内容构成了/views/default/login/form.php 文件：

```
<div id="loginbox">
    <h1>Login</h1>
    <?php
        echo view::show('standard/errors');
    ?>
    <form action="/login/process" method="post">
        <div class="row">
            <label for="username">Username:</label>
            <input type="text" name="username" id="username"
                   value="<?php echo lib::getitem('username')?> " />
        </div>
        <div class="row">
            <label for="password">Password:</label>
            <input type="password" name="password" id="password"/>
        </div>
        <div class="row">
            <label for="submit"></label>
            <input id="submit" type="submit" value="login"
                   class="submitbutton"/>
        </div>
    </form>
</div>
```

这个视图简单地显示一个登录窗体，该窗体会被提交给 URL /login/process，也就是 login 模块的 process() 方法。用于显示错误的另一个视图会插入至顶部。我能够想象到用户可能会无意输入错误的密码，因此需要向他们提示这种情况。最后，唯一需要注意的项是用户名 HTML input 项的值。lib 类的 getitem() 方法会在这里执行。如果指定用户名之前没有提交用户名，那么就会返回 NULL，并且不会执行任何打印。不过，需要确保使用 process() 方法中的 lib::setitem() 设置用户名。

要插入的视图定位于如下所示的/views/default/standard/errors.php 文件：

```
<?php
$errors = lib::getitem('error', lib::NO_PERSISTENT_STORAGE);
if (is_array($errors)) {
    print '<ul class="error"><li>' .
        implode('</li><li>', $errors) . '</li></ul>';
}
```

### 第 III 部分 PHP 设计案例分析

这个视图会尝试从指定会话中检索出错误键。请注意，某个错误键只允许存在一次。在被检索之后，传递至 lib::getitem( ) 的额外参数会确保错误键被复位。如果结果是一个数组，那么该数组会被遍历循环和显示。

上述编程完成后，读者应当能够看到与图 22-1 非常相似的界面。

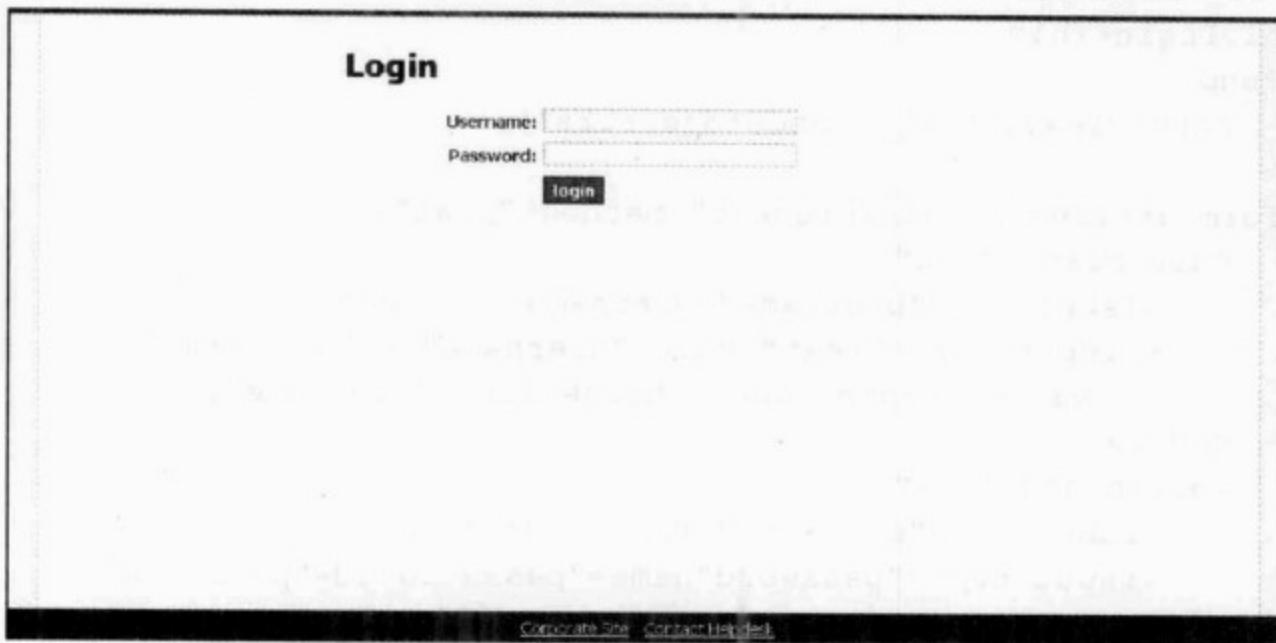


图 22-1

接下来的方法是登录模块的 process( ) 方法。如果没有进行计划，那么编程将非常困难。不过，我已经充分计划了相关的 User 和 Auth 对象，从而能够轻松地从头至尾编写这个方法。下面给出了作为登录模块公共方法的 process( ) 的内容：

```
public function process()
{
    $username = $_POST['username'];
    $password = $_POST['password'];

    if (empty($username)) {
        lib::seterror('Please enter a username.');
        lib::sendto('/login');
    }

    if (empty($password)) {
        lib::setitem('username', $username);
        lib::seterror('Please enter a password. ');
        lib::sendto('/login');
    }

    $user = new user(array('username'=>$username));
}
```

```
if (auth::authenticate($user, $password)) {
    lib::setitem('user', $user);
    lib::sendto();
}
else {
    lib::setitem('username', $username);
    lib::setError('Invalid username or password.');
    lib::sendto('/login');
}
}
```

这个方法也相当简单。它从提交的内容中检索用户名和密码。如果其中任意一项是空的，那么就会置位一个错误，并且用户会被重定向回登录页面。如果密码为空，那么千万不要忘记设置用户名。在这里，最新编辑的代码针对 lib 类的静态方法 setError( )。我希望创建方便置位错误视图将要分析的错误的方法，从而能够节省一些时间。lib::setError( )方法的内容如下所示：

```
public static function setError($error)
{
    self::setItem('error', $error, self::SETTING_AN_ARRAY);
}
```

这个方法只是使用其数组功能快速调用已有的 setItem( )静态方法。

process( )方法的最后几个步骤在前面已进行了一些讨论。我尝试确定下列哪种方式更为合适：是先创建 user 对象再对其进行身份验证，还是先对证书进行身份验证然后再创建 user 对象。根据我的结论，下一个步骤将是生成新的 User 对象。项目计划说明：user 所基于的 dao 对象能够接受一个映射用户表的限定符。此时，我将发送\$username 变量，用户名列(user name column)使用该变量定位某个 user。

auth 对象的 authenticate( )公共方法随后会执行。图 21-7 中的 auth UML 图表名这个方法是我需要调用的唯一方法。authenticate( )方法内部的工厂调用会选择适当的身份验证方案。如果该方法返回 false，那么我将存储用户名并将用户重定向回登录页面。如果返回成功标志，那么就在会话中存储当前 user 对象。通过这种方式，可了解当前用户何时会构建 contactsCollection 对象。同时，这也可提示如何构建 auth::isloggedin( ) 和 auth::isadmin( ) 方法：它们需要通过执行 lib::getitem( ) 方法来访问被存储的 user 对象。一切看起来相当简单。接下来的工作将是阐明如何创建 dao user 对象。

## 2. 数据访问对象和 User 对象

因为 Login 模块必须创建一个新的 User 对象, 所以需要参考图 21-5 中 UML 图的设置。构建的第一个类是 dao 抽象类。因为它不是实际的数据对象而是要构建的包含类, 所以将位于具有下面内容的 includes/dao.php 文件中:

```
class dao
{
    protected $values = array();

    public function __construct($qualifier = NULL)
    {
        if (!is_null($qualifier)) {

            $conditional = array();

            if (is_numeric($qualifier)) {
                $conditional = array('id'=>$qualifier);
            }
            else if (is_array($qualifier)) {
                $conditional = $qualifier;
            }
            else {
                throw new Exception('Invalid type of qualifier given');
            }

            $this->populate($conditional);
        }
    }

    public function __set($name, $value)
    {
        $this->values[$name] = $value;
    }

    public function __get($name)
    {
        if (isset($this->values[$name])) {
            return $this->values[$name];
        }
        else {
            return null;
        }
    }
}
```

```
}

protected function populate($conditional)
{
    $connection = db::factory('mysql');

    $sql = "select * from {$this->table} where";
    $qualifier = '';

    foreach ($conditional as $column=>$value) {
        if (!empty($qualifier)) {
            $qualifier .= ' and ';
        }
        $qualifier .= "{$column}='{$connection->clean(
            $value)}' ";
    }

    $sql .= $qualifier;
    $valuearray = $connection->getArray($sql);
    if (!isset($valuearray[0])) {
        $valuearray[0] = array();
    }

    foreach ($valuearray[0] as $name=>$value) {
        $this->$name = $value;
    }
}

public function save()
{
    if (!$this->id) {
        $this->create();
    }
    else {
        $this->update();
    }
}

protected function create()
{
    $connection = db::factory('mysql');

    $sql = "insert into {$this->table} (";
```

### 第 III 部分 PHP 设计案例分析

```

$sql .= implode('', '', array_keys($this->values));
$sql .= "'") values ('';

$clean = array();
foreach ($this->values as $value) {
    $clean[] = $connection->clean($value);
}

$sql .= implode('', '', $clean);
$sql .= "')";

$this->id = $connection->insertGetID($sql);
}

protected function update()
{
    $connection = db::factory('mysql');

    $sql = "update {$this->table} set ";

    $updates = array();
    foreach ($this->values as $key=>$value) {
        $updates[] = "'{$key}'='". $connection->clean($value) ."'";
    }

    $sql .= implode(', ', $updates);
    $sql .= "where id={$this->id}";

    $connection->execute($sql);
}
}

```

UML 图显示 dao 对象具有许多方法，因此不要对这个代码段的长度感到惊讶。分解该代码段相当容易，尤其在牢记第 21 章所介绍结论和计划的情况下更是如此。

第一个方法是构造函数。它接受一个可选的限定符。如果接受的限定符为 NULL，那么只会进行 dao 对象的初始创建。不过，如果发送的是 NULL，那么必须构建相应的条件。如果发送的是一个数值变量，那么就会认定用户表的主键被标记为 id。否则，限定符数组会按照原样发送。如下所示，这允许以多种方式创建一个新的 dao 对象(例如 user):

```

$u = new user(12); //by primary ID = 12
$u = new user(array('username'=>'ted')); //username column = ted
$u = new user(array('username'=>'ty', 'admin'=>1));

```

```
//username column=aaron,admin=1
```

随后，限定符/条件信息被发送至受保护方法 `populate()`，从而填充 `dao` 对象的各种属性。

`dao` 类的受保护方法 `property()` 直接处理数据源。此时，我会使用 `db` 对象的工厂方法来检索与 MySQL 数据库的连接，然后会构建相应的 SQL 语句。

我们应当注意与查询构建相关的 3 个重要事项。首先，查询引用 `dao` 类的表属性。为了成功地创建这个查询，`dao` 对象的任何子对象都必须定义该属性。接下来要注意的是条件在 SQL 语句中的应用方式。因为 `dao` 类接受限定符或将限定符转换为数组，所以 SQL 语句能够匿名地构建这个查询。最后需要注意的就是对 `db` 对象 `clean()` 方法的调用。我会让数据库对象来决定如何净化处理用户输入。

`populate()` 方法的最后一个步骤是执行查询。如果得到某个结果，那么该数组的每个值都会被指派给 `dao` 对象的公共属性。在 `dao` 对象和子类的设计阶段期间，我确定了 `dao` 类应当利用 PHP 的优秀方法 `_get()` 和 `_set()`。这两个方法都是只针对指定对象的受保护数组 `$values` 执行操作。

公共方法 `save()` 将确定指定对象是新对象还是需要被更新，随后会按计划调用 `create()` 或 `save()` 方法。因为能够获得数据库对象以及自动构建查询，所以这两个方法都与 `populate()` 方法相似。唯一应当注意的是创建指定对象之后第一次为对象指派主键 `id` 的操作。

创建父 `dao` 类之后，我就能够将 `User dao` 对象的创建转移至如下所示的 `/dataobjects/user.php` 文件中：

```
<?php
class user extends dao
{
    protected $table = __CLASS__;
}
```

此时，该对象唯一需要执行的操作是定义 `populate()`、`update()` 和 `create()` 方法所需的受保护属性 `$table`。我已经决定在 MySQL 数据库中将用于这个 `User dao` 对象的表命名为 `user`。使用下面的语句能够创建数据库：

```
CREATE TABLE 'contacts'.'user' (
    'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
    'username' varchar(64) NOT NULL,
    'password' varchar(40) NOT NULL,
    'admin' tinyint(3) unsigned NOT NULL DEFAULT '0',
    PRIMARY KEY ('id')
) ENGINE=InnoDB;
```

### 3. 数据库连接对象

显而易见，dao 对象极大地依赖于数据库。通过图 21-6 中的 UML 图，读者可以看到对 db、mysql 和 singleton 对象的规划。

首先希望创建的是 db 对象。因为这个类最为抽象，所以定义应当十分简短。db 对象位于如下所示的/includes/db.php file 文件：

```
abstract class db
{
    public static function factory($type)
    {
        return call_user_func(array($type, 'getInstance'));
    }

    abstract public function execute($query);
    abstract public function getArray($query);
    abstract public function insertGetID($query);
    abstract public function clean($string);
}
```

正如 UML 图所指定的那样，db 类是抽象的，因此只具有 4 个公共方法：execute()、getArray()、insertGetID() 以及 clean()。因为 db 类基于工厂设计模式，所以在类中定义的唯一方法是 factory()。这个方法简单地返回调用某个类(以\$type 参数命名)getInstance() 方法的结果。

**注意：**

要了解工厂设计模式的更多信息，请参见第 9 章。

在 dao 对象中，会请求 mysql 数据库连接对象的一个实例。这个实例位于/includes /mysql.php 文件，并且具有如下所示的内容：

```
<?php
class mysql extends db implements singletoninterface
{
    protected static $instance = null;
    protected $link;

    public static function getInstance()
    {
        if (is_null(self::$instance)) {
            self::$instance = new self;
```

```
}

    return self::$instance;
}

protected function __construct()
{
    $user = 'user';
    $pass = 'pass';
    $host = 'localhost';
    $db = 'contacts';

    $this->link = mysql_connect($host, $user, $pass);
    mysql_select_db($db, $this->link);
}

public function clean($string)
{
    return mysql_real_escape_string($string, $this->link);
}

public function getArray($query)
{
    $result = mysql_query($query, $this->link);

    $return = array();

    if ($result) {
        while ($row = mysql_fetch_array($result, MYSQL_ASSOC)) {
            $return[] = $row;
        }
    }
}

return $return;
}

public function execute($query)
{
    mysql_query($query, $this->link);
}

public function insertGetID($query)
{
```

```

    $this->execute($query);
    return mysql_insert_id($this->link);
}
}

```

被定义的第一个方法是公共静态方法 `getInstance()`。UML 图定义了该方法，`db` 对象的 `factory()` 方法会对其进行调用，并且 `singleton` 接口(后面将进行讨论)也需要这个方法。`getInstance()` 方法只是查看 `$instance` 的受保护静态属性是否保存 `mysql` 对象的一个实例。如果没有，那么就会创建一个实例并将其指派给 `$instance`。最后，这个方法会通过使用 `$instance` 作为其返回属性来返回对该 `mysql` 对象的引用。

`mysql` 对象的构造函数简单地创建了一个与数据库的连接，接着会将与数据库的链接存储在受保护变量 `$link` 中。最后，构造函数会选择适当的数据库。只要执行查询或清除字符串，就会使用链接属性。

#### 注意：

处理可能存在的多个数据源时，最好显式引用想要处理的链接。否则，就可能使用不同的连接执行查询，从而可能破坏一个缓冲的请求。此外，如果字符编码类型与数据库不同，清除基于某个编码类型的字符串仍然可能会使采用其他连接编码类型的字符串中存在问题字符。

其他 4 个公共方法要求根据其释义被定义为 `db` 对象中的抽象方法。在这里，模块设计模式能够提供帮助。第一个方法是 `clean()`，它接受“脏”字符串、执行 MySQL 字符串净化函数并返回“干净”字符串。下一个方法是 `execute()`，它简单地返回通过 `$query` 参数传递的查询。`insertGetID()` 方法也很简短，它调用 `execute()` 方法，然后通过调用使用 `execute()` 的 MySQL 链接上的 `mysql_insert_id()` 来返回检索的 ID。

#### 注意：

要了解模板设计模式的更多信息，请参见第 18 章。

最后一个方法是 `getArray()`，它执行 MySQL 查询并保持结果指针。接着，该方法遍历循环所有结果，并且根据结果构建关联键控数组的一个数值键控数组。最后，这个数值键控数组通过返回 `$return` 数组向外发送。

UML 图的最后一部分是 `singleton` 接口。这个简单的接口位于如下所示的 `/includes/singletoninterface.php` 文件：

```

<?php
interface singletoninterface

```

```
{
    public static function getInstance();
}
```

上面的接口只是使 db 子对象具有 db::factory() 所期望的 getInstance() 方法。

### 身份验证对象

似乎有很长时间都没有谈到 Login 模块了。但是，我正忙于处理这个模块的 process() 方法。最后一个未定义的部分是 auth 对象的方法。

在基本代码中，已经使用了图 21-7 中 UML 图表示的 auth 对象所具有的 3 个公共方法。下面将创建内容如下所示的 /includes/auth.php 文件：

```
< ?php
class auth
{
    public static function isloggedin()
    {
        return !is_null(lib::getitem('user'));
    }

    public static function isadmin()
    {
        return self::isloggedin() & & (1 == lib::getitem('user') -> admin);
    }

    public static function authenticate(user $user, $password)
    {
        $authenticator = self::factory('standard');
        return $authenticator->authenticate($user, $password);
    }

    protected static function factory($type)
    {
        $class = "auth{$type}";
        if (class_exists($class)) {
            return new $class;
        }
        else {
            throw new InternalException($type . ' is not a defined
                                         auth module.');
        }
    }
}
```

第一个公共静态方法名为 `isloggedin()`。题头视图和 Login 模块中都使用了这个方法。因为已经创建了成功的身份验证代码，并且这部分代码将当前 User 对象存储入指定会话，所以该方法十分简单。`isloggedin()`方法只是查看指定会话中是否存在一个 user 对象。如果存在，那么用户就已成功登录了系统。

下一个公共静态方法名为 `isadmin()`。读者可能记得，这个方法先前在题头视图中被用于添加另外的菜单选项。确认用户登录后，`isadmin()`方法会查看当前 user 对象的管理员属性。如果该属性被设置为 1，那么就认定当前 user 对象是管理员。

前面曾经提到，计划步骤完成后，编程有时会稍有变化。在 UML 图中，将下一个方法的名字标识为 `authenticateusingfactory()`。不过，在针对该对象进行编程时，决定使用另一个名字 `authenticate()`，这个名字更为合理，并且更易于理解。`authenticate()`方法执行 auth 对象的 `factory()`方法，从而获得标准身份验证器的一个实例。如果今后添加更多的身份验证器，那么这条语句会变得更为动态。随后，`authenticate()`方法的参数被转发给新身份验证器的 `authenticate()`方法。这个过程的结果将作为返回属性被发送。

最后一个方法是受保护方法 `factory()`，该方法构建身份验证器类的名字。如果存在指定的身份验证器类，那么就返回该类的一个新实例。否则，就会抛出一个异常。借助于此处定义的格式，能够将用于标准身份验证的类称为 `authstandard`。如果创建了一个使用 LDAP 的类，那么这个类可以被命名为 `authLDAP`。被调用的 `factory()`方法则会使用 LDAP 参数。

因为知道每个身份验证器都必须具有一个 `authenticate()`方法，并且不管怎样 auth 对象的 `authenticate()`方法都会调用这个方法，所以我准备借助一个接口来实施该方法。`Authenticatorinterface` 接口位于内容如下所示的 `/includes/authenticatorinterface.php` 文件：

```
<?php
interface authenticatorinterface
{
    public function authenticate(user $user, $password);
}
```

这个接口会使所有身份验证器都处理一个 User dao 对象和一个密码。

最后，让我们看看 auth 对象中 `factory()` 方法所请求的 `authstandard` 类。这个类位于具有下面内容的 `/includes/authstandard.php` 文件中：

```
<?php
class authstandard implements authenticatorinterface
{
    public function authenticate(user $user, $password)
    {
        if ($user->password == lib::makehashedpassword($user,
```

```
        $password)) {  
  
    return true;  
}  
else {  
    return false;  
}  
}  
}
```

无论是否经过身份验证，User dao 对象被创建时都会从指定表中获得密码。这种方法会生成被提交密码的散列形式，并且能够将其与 user 对象所有的密码进行比较。如果两者匹配，那么指定用户就是通过身份验证的。

我在 lib 类中添加了另外一个名为 makehashedpassword( ) 的方法。最初，我以为能够将它作为一个受保护方法添加入身份验证器。但是用户管理部分也可能需要访问这个方法。目前，最佳的解决方案似乎是在 lib 类中放置如下所示的机制：

```
public static function makehashedpassword(user $user, $password)
{
    return sha1($user->username . $password);
}
```

我选择的机制会使用 SHA-1 对密码进行散列运算。我将指定用户对象的用户名属性和密码字符串组合在一起，并且返回该组合的 SHA-1 散列形式。

注意：

为了测试这个应用，有必要手工插入第一个用户。可以使用如下所示的 MySQL 语句：

```
insert into 'contacts'.'user' ('username', 'password', 'admin')
values ('admin', 'efacc4001e857f7eba4ae781c2932dedf843865e', 1);
```

这条语句将允许使用用户名 admin 和密码 password 登录系统。

完成上述步骤后，用户就可以作为一个 User dao 对象登录。Index 模块此时能够执行创建 contactscollection 对象和显示指定用户联系方式的代码。不过，接下来应当按照计划开始着手处理 LogOut 模块。我准备在 Index 模块对象 defaultaction( )方法中条件语句 else 部分的内部对代码进行相应的注释：可能只是打印输出 You are Here。通过这种方式，能够测试登录过程，并且不必执行未编写完成的代码就可以访问菜单选项。接下来，将继续创建登出的功能。

## 5. 登出模块

如果用户能够登录，那么他们可能希望在某些时刻登出。因此，题头视图包含一个对登出 URL: /logout 的链接。根据已开发的模块模式以及图 21-8 中所计划的 UML 图，我能够创建一个 LogOut 模块。这个模块定位于具有下面内容的/modules/logout.php 文件：

```
<?php
class logout
{
    public function defaultaction()
    {
        lib::setitem('user', NULL);
        lib::sendto();
    }
}
```

这是到目前为止我已创建的最简单的模块。该模块被访问时，一般会撤消当前 user 对象在具体会话中的存储，随即指定用户被重定向至索引页面，同时结束整个操作循环。

## 6. 使用 UserModule 和 Userscollection 的管理

已经为登录过程创建了一个 User dao 对象。因为之前计划这个特定的 User dao 对象能够访问其他用户，所以将会变得更为有用。这是由 User 对象的\$admin 属性决定的。题头视图通过调用 isadmin()方法查看能否访问当前用户。如果当前用户是管理员，那么就能够进入用户管理部分。

添加至题头视图的链接引用了 Users 模块。根据图 21-8 中的 UML 图，UsersModule 对象将具有 6 个公共方法：3 个显示视图；另外 3 个处理针对 User 的动作。第一个方法是 show()。同样地，已对 ContactsModule 进行了计划，UsersModule 的默认动作应当是显示当前 User 能够使用的 User dao 对象集合。show()方法将引用一个集合对象。

因为知道 Contacts 模块会处理一个 UsersCollection 对象，所以准备首先对该对象进行编程。图 21-10 中的 UML 图表明需要创建抽象对象 daocollection。UsersCollection 将扩展这个类。在具有下面内容的/includes/daocollection.php 文件中可以找到 daocollection 类：

```
<?php
abstract class daocollection implements Iterator
{
    protected $position = 0;
    protected $storage = array();

    abstract public function getwithdata();
```

```
protected function populate($array, $dataobject)
{
    foreach ($array as $item) {
        $object = new $dataobject;
        foreach ($item as $key=>$val) {
            $object->$key = $val;
        }
        $this->storage[] = $object;
    }
}

public function saveall()
{
    foreach ($this as $item) {
        $item->save();
    }
}

public function current()
{
    return $this->storage[$this->position];
}

public function key()
{
    return $this->position;
}

public function next()
{
    $this->position++;
}

public function rewind()
{
    $this->position = 0;
}

public function valid()
{
    return isset($this->storage[$this->position]);
}
```

### 第 III 部分 PHP 设计案例分析

按照模板设计模式，除了声明整个类抽象之外，公共方法 `getwithdata()` 也被定义为抽象的。

**注意：**

第 18 章讨论了模板设计模式。

`populate()` 公共方法十分有趣，它接受一个信息数组以及一个表示数据访问对象类的字符串。因为这个 `daocollection` 实际上是 `dao` 对象的集合，所以这是有意义的。接下来，该方法遍历循环数据数组的每个元素，然后基于传递入内的名字创建一个新的数据访问对象，并且将公共值指派给这个 `dao` 对象。这是我为什么定义 `dao` 抽象类选择接受限定符并且不强制其存在的一个原因。在这个特定的案例中，我拥有创建 `dao` 对象所需的所有信息，因此不必要求任何对数据库的额外调用。最后，所有元素都存储至 `dao` 类。

`daocollection` 实现的 PHP 的 `Iterator` 接口需要 5 个公共方法：`current()`、`key()`、`next()`、`rewind()` 以及 `valid()`。正如计划期间所确定的那样，将信息数组存储在受保护数组 `$storage` 中。

在这个类中创建的最后一个方法是公共方法 `saveall()`，它只是遍历循环 `$this` 的每个元素并针对指定对象执行 `save()` 方法。因为 `$this` 是一个 PHP 迭代器，所以将 `$this` 视为数组时，先前提到的各种方法能够返回适当的 `dao` 对象。由于每个数据对象都必须具有 `save()` 方法，因此它们能够被放心地调用。在稍后实现中介者设计模式时，也可以使用这个特定方法。在编程中，将有可能遇到需要对整个集合进行一些更改、随后每个元素都需要被保存的场景。

**注意：**

要想了解中介者设计模式的更多信息，请参见第 12 章。

只需另外几个步骤就能够重新访问 `Users` 模块对象。不过，首先应当回顾图 21-11 中的 UML 图。这个 UML 图表明 `UsersCollection` 扩展了刚刚创建的 `daocollection` 对象。此外，`UsersCollection` 还实现了 `daocollection` 接口，从而确保每个 `daocollection` 子类接受一个所有者 `dao` 对象。

如下所示，`daocollectioninterface` 的内容位于 `/includes/daocollectioninterface.php` 文件中：

```
<?php
interface daocollectioninterface
{
    public function __construct(dao $item);
}
```

最后一个步骤是创建 `userscollection` 类，该类位于具有下面内容的`/includes/userscollection.php` 文件中：

```
<?php
class userscollection extends daocollection implements
daocollectioninterface
{
    public function __construct(dao $currentuser)
    {
        $this->currentuser = $currentuser;
    }

    public function getwithdata()
    {
        $connection = db::factory('mysql');

        $sql = "select * from user order by username";
        $results = $connection->getArray($sql);

        $this->populate($results, 'user');
    }
}
```

构造函数简单地将所有者 `dao` 对象设置为指定集合对象的公共属性。最初，没有进行任何应用限制。如果访问用户是管理员，那么他们就可以检索所有 `User dao` 对象。今后，系统可能引入其他需求或限制。那时，我将满意于这个当前用户被存储并且可供访问。

`userscollection` 类内定义了 `daocollection` 中的唯一抽象方法 `getwithdata()`。`daocollection` 对象的子类通过与 `dao` 类所采用相似的方式为自身填充数据。这个特定的方法会生成一个从用户表系统中检索所有用户对象的查询，随后检索的数组形式信息传入 `daocollection` 类的 `populate()` 方法。因为我期望的是 `User dao` 对象的集合，所以发送字符串中的最后一个参数是 `user`。

借助于构建的集合对象，就能够重新访问 `Users` 模块。下面的内容位于`/modules/users.php` 文件中：

```
<?php
class users
{
    public function defaultaction()
    {
        $users = new userscollection(lib::getitem('user'));
        $users->getwithdata();
```

### 第 III 部分 PHP 设计案例分析

```

        echo view::show('users/show', array('users'=>$users));
    }

    public function add()
    {
        echo view::show('users/add');
    }

    public function edit()
    {
        $controller = lib::getitem('controller');

        if (empty($controller->params[0])) {
            lib::sendto();
        }
        else {
            $user = new user((int) $controller->params[0]);
            echo view::show('users/edit', array('user'=>$user));
        }
    }
}

```

因为题头视图提供一个只包含模块名的 URL，所以 defaultaction( )方法将替代图 21-9 所示 UML 图中的 show( )方法。这个方法将创建 userscollection 模块的一个新实例。我能够使用用于查看用户是否登录或为管理员的相同方法来检索当前的 user dao 对象。接着，通过调用 getwithdata( )方法可以填充 userscollection 对象。最后，某个视图将被显示。对 show()静态方法的调用会将一个数组作为参数发送给该视图。这个参数数组包含刚刚创建的 userscollection 对象。

/views/default/users/show.php 文件的内容如下所示：

```

<div class="sidebar"><br/>
<a class="featured" href="/users/add">Add User</a>
</div>

<h1>User Admin</h1>
<p>
    Create, Edit or Delete users of the Contact System here.
</p>
<table>
<tr>
<th> Username </th>

```

```

<th>Admin? </th>
<th></th>
<th></th>
</tr>
<?php
foreach ($view['users'] as $user) {
    echo view::show('users/row', array('user'=>$user));
}
?>
</table>
<script type="text/javascript" src="/assets/removal.js"></script>

```

这个视图提供了添加新用户的选项，随后还会生成一个显示系统中现有用户的表。通过使用另一个视图在一条记录中显示一个用户，迭代 `userscollection` 对象。需要注意的是，集合中的当前 `user dao` 对象被传递入参数数组中的下一个视图。

`/views/default/users/row.php` 文件的内容如下：

```

<?php
echo '<tr>';
echo "<td>{$view['user']->username}</td><td>";
echo $view['user']->admin == 1 ? 'Yes' : 'No';
echo "</td><td><a href='/users/edit/{$view['user']->id}'>Edit</a></td> ";
echo "<td><a class='removal' href='/users/processdelete/
    {$view['user']->id}'> ";
echo "Delete </a></td> ";
echo '</tr>';

```

这个视图简单地接受当前的 `user dao` 对象，并且显示用户名和管理员状态。此外，该视图还添加了对 `Users` 模块中用于编辑和删除用户的不同方法进行访问的链接。

`users/show` 视图的最后一行是一个包含下列 JavaScript 内容的链接：

```

if (typeof jQuery != 'undefined') {
    $(function() {
        $(".removal").click(function(){
            if (!confirm('You really want to delete this?')) {
                return false;
            }
        });
    });
}

```

如果用户单击任何具有 `removal` 类的链接(如 `Delete User` 链接)，那么上面的代码会提

### 第 III 部分 PHP 设计案例分析

示用户同意继续执行删除操作。我希望为站点的管理员增加一个防护措施，而这是最简单的方式。

完成这个步骤以后，现在就能够登录并单击用户管理链接，从而在一个表中显示系统中所有用户的集合。此时显示的视图大致与图 22-2 相同。



图 22-2

接下来的步骤是创建 Users 模块 add() 公共方法所引用的用户添加视图。

我认为添加用户与编辑用户非常相似。因此，我的想法是创建一个接口，并且定制为只用于添加或编辑功能性。/views/default/users/add.php 文件的内容如下所示：

```
<?php
    echo view::show('users/manage',
                    array('title'=>'Add User',
                          'action'=>'/users/processadd'));
```

所引用的这个共享接口是 users/manage 视图。定制时需要注意该窗体的题头和操作。读者可能很容易发现，用户添加代码希望引用 processadd() 方法，因此窗体动作将反映这个情况。如下所示，/views/default/users/manage.php 文件的内容略为复杂：

```
<?php
    echo '<h1>' . $view['title'] . '</h1>';
    echo view::show('standard/errors');
    echo '<form method="post" action="' . $view['action'] . '">';
    if (!is_null($view['user'])) {
        echo ' < input type="hidden" name="id" value="' .
            $view['user']->id. '">';
```

```
}

echo '<div class="row"><label for="username">Username:</label>';
$value = $view['user']->username;
echo '<input type="text" name="username" id="username" value="' .
    $value . '" /></div>';

echo ' <div class="row"><label for="password">Password:</label>';
    echo ' <input type="password" name="password" id="password"/></div>';

echo ' <div class="row"><label> Is Admin? </label>';

$options = array('No', 'Yes');
$value = (int) $view['user']->admin;

foreach ($options as $key=>$option) {
    echo ' <input class="radio" type="radio" name="admin" value="' .
        . $key . '"';
    if ($value == $key) echo 'checked="checked"';
    echo ' /><span class="radiooption">' . $option . '</span >';
}

echo '</div>';

echo '<div class="row"><label for="submit"></label>
      <input id="submit" type="submit" class="submitbutton" value="' .
        . $view['title'] . '" /></div>';

echo '</form>';
```

第一个步骤是使用参数数组打印要传递入视图的题头。接下来会添加常见的 standard/errors 视图。需要记住的是，standard/errors 视图能够显示 processadd() 或 processedit() 方法决定生成的任何错误。最后可预见的是 HTML 窗体标记的创建，它将从对该视图的调用中接受动作参数。

如果在编辑用户性能中使用 users/manage 视图，那么我还将向参数内传递一个 user dao 对象。因此，这个视图会查看该对象是否存在。如果 user dao 对象参数确实存在，那么这个视图会生成一个隐藏的输入来跟踪 user 对象的 id 属性。执行更新操作的时候需要主键 ID。

此外还会生成用户名和密码字段。如果传递入一个 user dao 对象，那么就会填充用户名字段。绝不预先输入密码字段是很好的安全策略，因此无论是添加用户还是编辑用户，这个字段最初都保留空状态。

最后，管理员选项使用了 HTML 单选输入。如果不存在任何 user 对象，那么管理员属性将被设置为 0。否则，user 对象的管理员公共属性会被计值(仍然可能为 0)。这个值被用于

### 第 III 部分 PHP 设计案例分析

自动选中合适的单选按钮。此时，窗体的设计完成。这个窗体应当如图 22-3 所示。

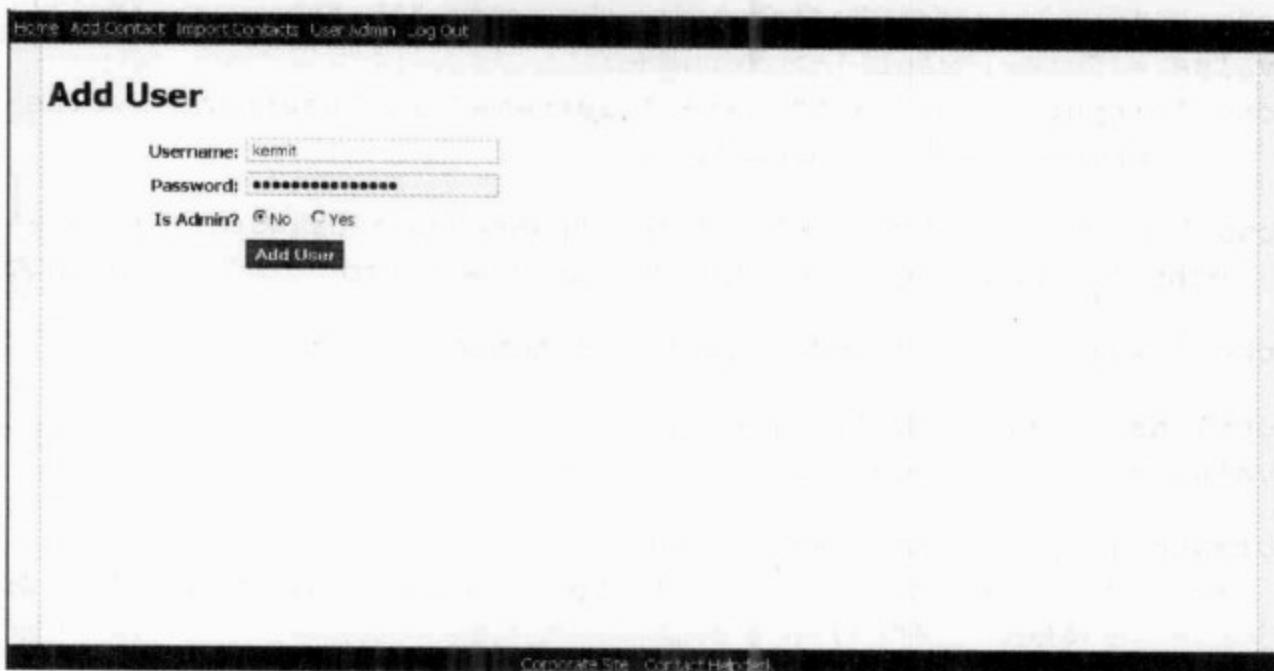


图 22-3

正确构建了创建新用户的代码块之后，就该将公共方法添加至 users 模块(如下所示)：

```
public function processadd()
{
    $username = $_POST['username'];
    $password = $_POST['password'];
    $admin = $_POST['admin'];

    $user = new user(array('username'=>$username));

    if (!is_null($user->id)) {
        lib::setError($username . ' is already in use');
        lib::sendto('/users/add');
    }

    $user->username = $username;
    $user->password = lib::makehashedpassword($user, $password);
    $user->admin = $admin;
    $user->save();

    lib::sendto('/users');
}
```

这个方法接受 3 个提交项，随后尝试使用\$username 变量创建一个新的 user 对象。如果指定用户被成功填充，那么该用户之前就已存在。此时将生成一个错误，并且用户被发

送回添加界面。否则，用户名、管理员状态以及密码散列值会被添加至这个 user dao 对象。最后，使用 dao 父对象的 save( )方法就能够保存和创建 user 对象。新创建的用户被发送回说明上述操作结果的 userscollection 显示界面。

接下来讨论用户对象的编辑。上面 users 模块的 edit( )方法应当非常容易生成。显示 user dao 对象的每条记录都具有一个编辑链接，该链接的 URL 中体现了 user 对象的 ID 属性。ID 属性会被转换为 controller 所存储的第一个参数。edit( )方法会检索 controller 对象并查看这个参数。如果 ID 不存在，那么相应用户就不能被编辑。理想情况下，这些代码不会被执行，因此不必提供错误消息。一旦检索到这个参数，那么就会使用指定 ID 创建新的 user dao 对象。随后，users/edit 视图接受该 user 并进行显示。

除了几处不同之外，这个视图与 users/add 视图的内容类似。如下所示，这个视图位于 /views/users/edit.php 文件中：

```
<?php
    echo view::show('users/manage',
                    array('title'=>'Edit User',
                           'action'=>'/users/processedit',
                           'user'=>$view['user']));
```

此时，窗体的标题和动作是不同的。此外，users/manage 视图现在会得到其寻找的 user dao 对象的实例。

为了处理编辑操作，需要在模块中添加下面的 processedit( )方法：

```
public function processedit()
{
    $id = $_POST['id'];
    $username = $_POST['username'];
    $password = $_POST['password'];
    $admin = $_POST['admin'];

    $user = new user($id);
    $user->username = $username;
    if (!empty($password)) {
        $user->password = lib::makehashedpassword($user, $password);
    }
    $user->admin = $admin;
    $user->save();

    lib::sendto('/users');
}
```

与 processadd( )方法一样，上面的方法也会检索到用户名、密码和管理员状态，此外

还将检索独特的 ID 元素。使用这个 ID 可以创建 user dao 对象的一个新实例。随后，用户名和管理员状态将被更新。如果发送的密码来自前一个界面，那么新的密码散列值会被指派给指定的 user。否则，密码属性不会被更改。最后，user 对象的父 dao 对象的 save( )方法将被执行，该方法会调用受保护方法 update( )。用户被重定向回 userscollection 视图，从而能够在其列表中查看更新的用户。

这部分最后需要做的是构建删除用户的能力。读者可能注意到这里不存在名为 delete( )的方法。我认为 JavaScript 提示足以完成这个功能。我并不需要为删除动作生成新的视图。显示一个用户的每条记录都具有指向 users 模块 processdelete( )方法的删除 URL。与编辑链接一样，删除 URL 以 user dao 对象的 ID 属性结束。被添加入 users 模块的 processdelete( )方法应当具有下面的内容：

```
public function processdelete()
{
    $controller = lib::getitem('controller');

    if (empty($controller->params[0])) {
        lib::sendto();
    }
    else {
        $userid = (int) $controller->params[0];

        $connection = db::factory('mysql');

        $sql = "delete u.* from user u
                where u.id = {$userid}";
        $connection->execute($sql);

        lib::sendto('/users');
    }
}
```

与 processedit( )方法一样，相同的方法用于检索待更改 user 对象的 id。不过，processdelete( )方法的下一条语句与其他两个类似方法有所区别。这个方法并不创建 user dao 对象再进行删除，而是生成一个直接在数据库中执行操作的查询。我感觉这种处理方式是最有效的。使用来自 db 对象 factory( )方法的连接对象就能够创建和执行这个查询。最后，用户被重定向至 userscollection 的视图，以便验证被删除用户不再可用。

到这个时候，应用程序的用户管理部分的编程宣告结束。与联系方式管理相比较，用户管理的实际过程虽然更为简单，但是整个应用程序所需的大量构建代码都在用户管理部分中创建。因此，我期望下一部分的工作同样轻松。

### 22.2.3 联系方式管理的编程

现在，用户能够成功地使用应用程序。不过，应用程序仍然不能令人兴奋，其原因在于它只是一个外壳。接下来的步骤是开始对联系方式管理部分进行编程。这个部分将允许用户实际导出、管理和查看联系方式。

#### 1. 生成关系以及创建数据访问对象

绘制的 UML 图与计划表明接下来需要创建 ContactsModule。不过，我准备先跳过这个模块，转而创建其他所有数据访问对象及相关联的 MySQL 表。有时候，不按照计划的顺序完成创建操作更为合理。

我已经创建了自己的 User dao 对象。一个用户可以不具有或具有多个与之相关联的 contact-dao 对象。contact dao 对象定位于具有下面内容的 /dataobjects/contact.php 文件中：

```
<?php
class contact extends dao
{
    protected $table = __CLASS__;
}
```

contact 数据访问对象的创建方式与 User dao 对象的创建方式完全相同。事实上，关系中的其余数据访问对象也通过相同的方式创建。下面用于创建 MySQL 表的语句与图 21-13 中用于这个对象的 UML 图相匹配：

```
CREATE TABLE 'contacts'.'contact' (
    'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
    'ownerid' int(10) unsigned NOT NULL,
    'firstname' tinytext NOT NULL,
    'middlename' tinytext NOT NULL,
    'lastname' tinytext NOT NULL,
    PRIMARY KEY ('id')
) ENGINE=InnoDB;
```

一个 contact 可以不具有或具有多个称为 contactgroup 对象的联系方式信息分组。下面的内容位于 /dataobjects/contactgroup.php 文件中：

```
<?php
class contactgroup extends dao
{
    protected $table = __CLASS__;
}
```

### 第 III 部分 PHP 设计案例分析

下面的 MySQL 语句被用于根据来自 UML 图的属性创建 MySQL 表：

```
CREATE TABLE 'contacts'.'contactgroup' (
    'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
    'contactid' int(10) unsigned NOT NULL,
    'label' text NOT NULL,
    PRIMARY KEY ('id')
) ENGINE=InnoDB;
```

不过，为了保证其存在，`contactgroup` 必须至少具有一个为其指派的 `contactmethod` 对象。但是，这个完整性是在应用程序中而不是在数据库中实施的。`/dataobjects/contactmethod.php` 文件的内容如下所示：

```
<?php
class contactmethod extends dao
{
    protected $table = __CLASS__;
}
```

下面的 MySQL 语句用于根据来自 UML 图的属性创建 MySQL 表：

```
CREATE TABLE 'contacts'.'contactmethod' (
    'id' int(10) unsigned NOT NULL AUTO_INCREMENT,
    'contactgroupid' int(10) unsigned NOT NULL,
    'type' tinytext NOT NULL,
    'value' text NOT NULL,
    PRIMARY KEY ('id')
) ENGINE=InnoDB;
```

创建数据访问对象后，就能够开始处理这些对象的显示和管理。

## 2. 联系方式集合的编程

在本章的前面部分中，对 `index` 模块 `defaultaction()` 方法内部的代码进行了注释，这部分代码引用了目前还未创建的 `contactscollection` 对象。

创建父 `daocollection` 抽象对象之后，`contactscollection` 对象的创建应当更为容易。此时，大多数构建块都已创建。

`contactscollection` 类位于具有下面内容的 `/includes/contactscollection.php` 文件中：

```
<?php
class contactscollection extends daocollection implements
    daocollectioninterface
{
```

```

protected $user;

public function __construct(dao $user)
{
    $this->user = $user;
}

public function getwithdata()
{
    $connection = db::factory('mysql');

    $sql = "select * from contact where ownerid=" . $this->user->id
        . ' order by firstname';
    $results = $connection->getArray($sql);

    $this->populate($results, 'contact');
}

```

与 userscollection 类似, contactscollection 对象扩展了 daocollection 的功能。daocollection 的模板设计模式性质要求定义公共方法 getwithdata()。daocollectioninterface 也实现了使构造函数接受一个 dao 对象。

### 注意:

要了解模板设计模式的更多信息, 请参见第 18 章。

contactscollection 对象 \_\_construct() 方法指派 user 传递至某个受保护属性, 该属性会被用在 getwithdata() 方法中。

getwithdata() 方法可以得到与数据库的连接, 随即将构建一个从属于受保护对象的 MySQL 联系方式表中收集所有记录的查询。最后, 父 populate() 方法会被执行, 此时 contact 对象作为用于具体集合的目标 dao 对象。很高兴, 围绕 userscollection 对象生成新集合的所有工作和计划非常简单。

### 3. 查看联系方式集合

创建完成 contactscollection 对象后, 我就准备将 index 模块的 defaultaction() 方法还原为下列代码:

```

public function defaultaction()
{
    if (!auth::isloggedin()) {
        lib::sendto('/login');
    }
}

```

### 第 III 部分 PHP 设计案例分析

```

    }
    else {
        $contacts = new contactscollection(lib::getitem('user'));
        $contacts->getwithdata();

        echo view::show('contacts/browse',
                        array('contacts'=>$contacts));
    }
}

```

如果用户已登录系统，那么就要创建 contactscollection 对象的一个新实例。当前登录用户的 user dao 被发送给这个集合对象。通过收集 contact 对象的被请求集合，getwithdata() 方法可以使用该集合对象。最后，这个集合将被传递至名为 contacts/browse 的新视图。

contacts/browse 视图位于具有下面内容的/views/default/contacts/browse.php 文件中：

```

<h1> Your Contacts </h1>
<div id="browsecontacts">
<?php
    foreach ($view['contacts'] as $contact) {
        echo view::show('contacts/small', array('contact'=>$contact));
    }
    echo '</div>';

    if (!isset($contact)) {
        echo view::show('index/welcome');
    }
}

```

上面的视图首先会创建名为 Your Contacts 的标题，然后遍历循环传递进来的 contactscollection。与查看站点中的用户一样，每个 contact 都具有自己的视图。这个名为 contacts/small 的视图接受作为参数的 contact dao。contacts/small 视图位于具有下面内容的/views/default/contacts/small.php 文件中：

```

<?php
echo ' <a href="/contacts/view/' . $view['contact']->id . '" > ';
echo "{$view['contact']->firstname} ";
echo "{$view['contact']->middlename} {$view['contact']->lastname}";
echo ' </a> ';

```

所有这些小视图执行的操作是生成一个对 contacts 模块 view( ) 操作的链接，同时将 contact dao 对象的 id 属性作为参数。这个链接的标记使用了指定联系方式的名、中名和姓。

contacts/browse 视图的最后部分会查看任何联系方式是否能够被真正检索。如果不能检索到任何联系方式，那么就会显示/views/default/index/welcome.php 文件中的友好欢迎消息：

```

<div class="sidebar">
  <br/>
  <a class="featured" href="/contacts/add"> Add Contact </a>
  <a class="featured" href="/contacts/import"> Import Contacts </a>
</div>
<h2> Welcome To Acme Company Contact Manager </h2>
<p>
  Welcome to the replacement system for the Outlook Address book.
</p>
<p>
  Cool things to know:
</p>
<ul>
  <li> You can easily <a href="/contacts/add"> add contacts </a></li>
  <li>
    You can <a href="/contacts/import"> import your address book </a>
    from Outlook
  </li>
  <li> You can view this using your mobile phone </li>
  <li>
    You can send email directly to a contact by clicking their
    email address.
  </li>
</ul>
<p>
  If you have any questions, please contact the help desk.
</p>

```

这条消息只是向可能还未使用过系统的用户告知他们可以选择的操作。

现在，我能够成功地登录并显示自己的所有联系方式。不过，因为没有添加任何联系方式，所以将看到欢迎消息(见图 22-4)。



图 22-4

#### 4. 联系方式模块和导入功能性的编程

联系方式模块是应用程序最重要的部分之一。因此，它也是应用程序中最大的模块。为了清楚起见，准备将这个模块分解为若干步骤。

第一个步骤是创建导入功能性。获取大量数据的最快速方式是导入自己的 Outlook CSV 文件。除此之外，欢迎消息也告知使用/contacts/import 链接来执行这样的操作。

联系方式模块位于/modules/contacts.php 文件中。如下所示，准备开始只创建 import() 和 processimport() 方法：

```
<?php
class contacts
{
    public function import()
    {
        echo view::show('contacts/import');
    }

    public function processimport()
    {
        if (is_uploaded_file($_FILES['CSV']['tmp_name'])) {
            $contents = file_get_contents($_FILES['CSV']['tmp_name']);
            unlink ($_FILES['CSV']['tmp_name']);

            $builder = new importcontactsarraybuilder($contents);
            $imports = $builder->buildarray();

            $currentuser = lib::getitem('user');

            foreach ($imports as $import) {
                $contact = new contact();
                $adaptor = new outlookcontactimportadapter($import);

                $contact->firstname = $adaptor->firstname;
                $contact->middlename = $adaptor->middlename;
                $contact->lastname = $adaptor->lastname;
                $contact->ownerid = $currentuser->id;
                $contact->save();
                $possiblegroups = array('Business', 'Home');

                foreach ($possiblegroups as $groupname) {
                    $groupfinder =
                        new contactimportgroupinterpreter($import);
                    $group = $groupfinder->getgroup($groupname);
                }
            }
        }
    }
}
```

```
        if ($group instanceof contactgroup) {
            $group->contactid = $contact->id;
            $group->save();

            $methods = new contactmethodscollection($group);
            $methods->generateimportmethods($import);
            $methods->saveall();
        }
    }

    lib::sendto();
}
else {
    lib::setError(array('Please upload a file.'));
    lib::sendto('/contacts/import');
}
}
}
```

公共方法 add( )简单地显示了一个新视图，该视图应当包含一个上传文件的 HTML 元素。这个视图位于具有下面内容的/views/default/contacts/import.php 文件中：

```
<h1> Import Your Contacts </h1>
<p>
    Upload a .CSV file from Outlook. Contact helpdesk for assistance.
</p>
<?php
    echo view::show('standard/errors');
?>
<form action="/contacts/processimport" method="post"
    enctype="multipart/form-data">
    <div class="row">
        <label for="CSV"> CSV File: </label><input type="file"
            id="CSV" name="CSV" />
    </div>
    <div class="row">
        <label for="submit"></label>
        <input id="submit" type="submit" class="submitbutton"
            value="Upload"/>
    </div>
</form>
```

上面的视图告诉用户上传他们的 CSV 文件。能够想象，到服务中心以后肯定会有对相关操作的说明。不过，这并不在需求之中，因此只是放置一个与用户关联的有用提醒，从而为他们提供帮助。接下来，标准错误视图会被添加用于显示文件上传期间所发生的任何错误。这些错误包括没有上传任何文件或者文件的格式不正确。

这个视图的最后一个步骤是生成一个具有适当编码类型的窗体，该窗体能够接受文件上传。需要注意的要点是类型为 file 的输入元素的名字是 CSV。将在接下来要介绍的 `process-import()` 方法中使用这个名字。

公共方法 `processingimport()` 执行的第一个操作是确认文件被成功地上传。如果来自 PHP 的 `is_uploaded_file()` 函数的结果为 `false`，那么就会设置一个相应的错误，并且用户会被重定向回前一个视图。错误视图将显示文件没有成功上传的消息。

接下来，上传文件的内容将被检索，而且文件本身会被删除。之后，该方法将创建 `importcontactsarraybuilder` 类的一个新实例。正如在图 21-14 的 UML 图中所定义的那样，这个类基于生成器设计模式。被创建的新实例具有上传文件的完整内容。最后，这个类的 `buildcollection()` 公共方法会被调用，其结果将存储在 `$imports` 变量中。

#### 注意：

要了解生成器设计模式的更多信息，请参见第 4 章。

如下所示，`importcontactsarraybuilder` 类位于 `/includes/importcontactsarraybuilder.php` 文件中：

```
<?php
class importcontactsarraybuilder
{
    protected $importedstring;

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

    public function buildarray()
    {
        $lines = explode("\n", $this->importedstring);
        $keys = explode(',', array_shift($lines));

        $array = array();
        foreach ($lines as $line) {
            $values = explode(',', $line);
            $array[] = array_combine($keys, $values);
        }
    }
}
```

```

        foreach ($lines as $line) {
            if (!empty($line)) {
                $keyed = array_combine($keys, explode(',', $line));
                $array[] = $keyed;
            }
        }

        return $array;
    }
}

```

最初在 UML 图中，这个类的名称及方法名具有词汇 collection。在应用程序的其他部分内，daocollection 对象的实例也使用了相同含义的词汇 collection。考虑到这个类实际上构建的是一个数组而不是 dao 对象的集合，这可能会引起混淆。

importcontactsarraybuilder 类的构造函数将其参数指派为受保护属性字符串 \$imported。名为 buildarray( ) 的公共方法执行实际的操纵。首先，新的分行被用于将指定字符串(也就是 CSV 文件的内容)分为若干行。第一行应当是文档的题头，应从文档中将其去除该题头并将其存储在其自己的数组 \$keys 中。随后，文档的剩余部分将被迭代。\$keys 数组与当前行组合形成新的关联数组。一旦迭代结束，关联数组就被添加至返回参数 \$array。这个方法最后得到的是一个关联键控数组，该数组的键是文件的第一行，内容是文件的剩余部分。

继续 contacts 模块的 processimport( ) 方法，当前 user 将被检索。随后，包含 buildarray( ) 方法返回参数内容的 \$imports 数组会被循环遍历。

针对导入数组的每个实例，系统都会创建一个新的 contact dao。接下来将创建 outlookcontactimportadapter 的一个新实例。在计划期间，导入的数组可能具有 contact dao 无法理解和使用的格式。/includes/outlookcontactimportadapter.php 文件的内容如下所示：

```

<?php
class outlookcontactimportadapter
{
    public function __construct($import)
    {
        $this->firstname = $import['First Name'];
        $this->middlename = $import['Middle Name'];
        $this->lastname = $import['Last Name'];
    }
}

```

这个适配器简单地使用了我们所了解的导入数组相关信息，并且按照 user dao 所采用的相同方式指派公共属性。适配器在 processimport( ) 方法的下一个步骤中十分有用，此时

### 第 III 部分 PHP 设计案例分析

适配器对象的实例指派了 contact dao 对象的属性。

最后, contact 从当前 user 中获取其所有者 ID 属性。contact dao 对象的 save( )方法完成了时 contact 对象的创建。

下个步骤将是确定这个 contact 是否需要创建 contactgroup 对象。Outlook 的 CSV 文件中有两个分组的前缀为 Business 和 Home。针对这些分组名, 需要执行下面将要讲述的操作。

contactimportgroupinterpreter 的一个实例会被创建, 其目的是导入数组将具有一组特定的键, 这些键能够被解释为分组的构建代码块。在计划解决这个问题期间, 该对象基于解释器设计模式。导入数组被传递至新实例之后, \$group 变量会从对 contactimportgroupinterpreter 对象 getgroup( )公共方法的调用中接收到返回参数。在这里, 需要注意循环中的当前分组名如何被发送到 getgroup( )方法。

**注意:**

要了解解释器设计模式的更多信息, 请参看第 10 章。

contactimportgroupinterpreter 类位于/includes/contactimportgroupinterpreter.php 文件中(如下所示):

```
<?php
class contactimportgroupinterpreter
{
    protected $import;

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

    public function getgroup($groupname)
    {
        $contactgroup = NULL;

        if (!empty($this->import["{$groupname} Street"])) {
            $contactgroup = new contactgroup();
            $contactgroup->label = $groupname;
        }

        return $contactgroup;
    }
}
```

与前面创建的适配器类相似，这个方法的构造函数只需要将其接收的数据指派给指定类的受保护属性。`getgroup()`公共方法是开始执行解释的位置。

为了确定某个联系方式分组应当存在，我准备查看导入数组中用于分组“街道”名的内容。在 CSV 文件中，我了解的两个相关列是 Business Street 和 Home Street。我假定如果这个字段中存在一个值，那么整个地址都可能存在，稍后很可能生成一个 `contactmethod dao`，这意味着可以创建指定的 `contactgroup` 对象。因此，如果发现该字段具有内容，那么就会创建并返回一个新的 `contactgroup` 对象，其标签被指派为 `$groupname` 变量的内容。如果该字段为空，那么就返回 `NULL`。

因为可能返回 `NULL`，所以 `processimport()` 方法的下一行将查看 `$group` 变量是否确实为一个 `contactgroup` 对象。如果不是，那么就不执行任何操作，进而开始尝试下一个分组名。

如果发现有效的 `contactgroup` 对象，那么就会通过将 `contact` 对象的 `id` 属性作为其所有者 ID 来更新这个对象。随后，调用 `dao` 父类的 `save()` 方法就可以结束对这个 `contactgroup` 对象的创建。

最后一个步骤是向具体分组添加联系方法。`contactmethodscollection daocollection` 的一个实例会被创建。所有者元素 `contactgroup` 会传递入构造函数。导入数组随后作为参数被发送至对公共方法 `generateimportmethods()` 的调用。最后，将会对 `daocollection` 对象 `saveall()` 公共方法进行首次执行。我的计划是创建一个像 `userscollection` 对象的标准集合对象。名为 `generateimportmethods()` 的额外方法应当能够采用与调用 `getwithdata()` 类似的方式生成 `contactmethod dao` 对象的集合，但是这些对象的内容基于导入数组，而不是基于 MySQL 查询生成的数组。

`/includes/contactmethodscollection.php` 文件的内容如下所示：

```
<?php
class contactmethodscollection extends daocollection
implements daocollectioninterface
{
    protected $group;

    public function __construct(dao $group)
    {
        $this->group = $group;
    }

    public function getwithdata()
    {
        $connection = db::factory('mysql');
```

### 第 III 部分 PHP 设计案例分析

```

    $sql = "select * from contactmethod where contactgroupid="
        . $this->group->id . 'order by type';
    $results = $connection->getArray($sql);

    $this->populate($results, 'contactmethod');
}

public function generateimportmethods($import)
{
    $results = array();

    $addressline = $import["{$this->group->label} Street"] . ' '
        . $import["{$this->group->label} Street 2"] . ' '
        . $import["{$this->group->label} Street 3"] . ' '
        . $import["{$this->group->label} City"] . ' '
        . $import["{$this->group->label} State"] . ' '
        . $import["{$this->group->label} Postal Code"];
    $results[] = array('type'=>'address', 'value'=> $addressline,
                      'contactgroupid'=>$this->group->id);

    $this->populate($results, 'contactmethod');
}
}

```

这个 daocollection 的构造函数按照预计实现，它将所有者 dao 对象指派给一个受保护属性。下一个公共方法 getwithdata( )也通过类似的方式构建，该方法生成一个从分组 dao 对象所有的 MySQL 表中检索记录的查询。对 daocollection 对象 populate( )方法的调用被用于创建新的 contactmethod dao 对象。

generateimportmethods( )公共方法与 getwithdata( )方法非常类似，但是这个公共方法并不从 MySQL 连接中检索结果，而是使用导入数组键的相关信息中构建一个结果类型。接下来，按照 db 对象 getArray( )方法返回 SQL 结果集的相同方式，generateimportmethods( )方法构建的结果会演变为一个数组。随后，populate( )方法被调用，该方法采用与 getwithdata( )方法执行方式相同的方式处理这些信息。

在 contact 中添加所有分组和联系方法之后，contacts 模块会将用户重定向至主页。在主页上，用户能够看到最新导入的所有联系方式。

联系方式导入功能性创建完成后，我就拥有了继续测试所需的良好基础：大量联系方式。图 22-5 显示了迄今为止的测试分组。

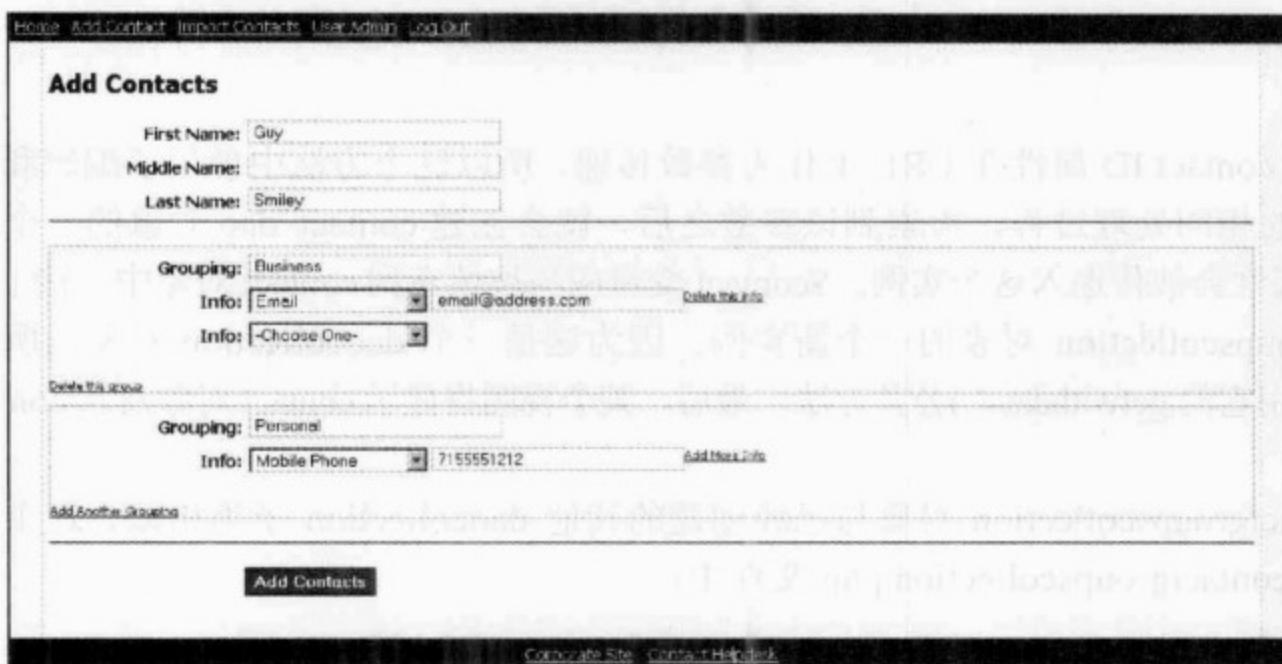


图 22-5

后面的几个步骤将完成 contacts 模块的剩余部分。用户将能够添加和编辑联系方式及其分组和方法，或者删除一个完整的联系方式。因为希望继续测试和验证我的结果，所以接下来要创建的是单独的 contact 视图。下一部分将继续对这个视图进行编程。

## 5. 从联系方式列表移动至单个联系方式视图

index 模块完成了针对 contactscollection 对象的大量工作，并且显示访问每个联系方式的链接。查看不同联系方式的链接从/contacts/view 文件开始。这意味着 contacts 模块的 view() 公共方法将被调用，已经创建了 import() 和 processimport() 方法，接下来则要添加如下所示的 view() 方法：

```
public function view()
{
    $controller = lib::getitem('controller');

    if (empty($controller->params[0])) {
        lib::sendto();
    }
    else {
        $contact = new contact((int) $controller->params[0]);
        $groups = new contactgroupscollection($contact);
        $groups->getwithdata();

        echo view::show('contacts/view',
                        array('contact'=>$contact,
                              'groups'=>$groups));
    }
}
```

### 第 III 部分 PHP 设计案例分析

```

    }
}

```

因为 contact ID 属性在 URL 上作为参数传递，所以这个方法中使用了编辑和删除 user dao 对象的相同处理过程。检索到该参数之后，就会创建 contact dao 对象的一个新实例，同时 ID 属性会被传递入这个实例。\$contact 变量应当被填充到 contact 对象中。接下来将创建 contactgroupscollection 对象的一个新实例。因为这是一个 daocollection 对象，所以下一个步骤是调用它的 getwithdata() 公共方法。最后，某个视图将显示 contact 对象及其 contactgroups 集合。

contactgroupscollection 对象与已经创建的其他 daocollection 子类相似。这个对象位于 /includes/contactgroupscollection.php 文件中：

```

<?php
class contactgroupscollection extends daocollection
implements daocollectioninterface
{
    protected $contact;

    public function __construct(dao $contact)
    {
        $this->contact = $contact;
    }

    public function getwithdata()
    {
        $connection = db::factory('mysql');

        $sql = "select * from contactgroup where contactid="
            . $this->contact->id . ' order by label';
        $results = $connection->getArray($sql);

        $this->populate($results, 'contactgroup');
    }
}

```

构造函数将 contact dao 所有者对象指派给受保护属性 \$contact。随后，contactgroup MySQL 表中 contactid 列与所有者 contact 对象 ID 属性匹配的所有记录将被检索。populate() 方法根据这些数据创建了 contactgroup 对象的集合。

contact 对象和 contactgroups daocollection 对象都传递至 contacts/view 视图。这个视图位于具有下面内容的 /views/default/contacts/view.php 文件中：

```
<?php
    $contactname = "{$view['contact']->firstname}
                    {$view['contact']->middlename}
                    {$view['contact']->lastname}";

    echo view::show('contacts/viewsidebar',
                    array('contactname'=>$contactname,
                          'id'=>$view['contact']->id));

    print " <h1> {$contactname} </h1> ";

    foreach ($view['groups'] as $group) {
        print " <fieldset> <legend> {$group->label} </legend> ";

        $methods = new contactmethodscollection($group);
        $methods->getwithdata();

        print ' <table> ';
        foreach ($methods as $method) {
            $decoratorclass = "decorator{$method->type}";

            if (class_exists($decoratorclass)) {
                $decorator = new $decoratorclass;
                $method->value = $decorator->decorate($method->value);
            }

            print " <tr> <td> {$method->type}: </td> <td>
                            {$method->value} </td> </tr> ";
        }
        print ' </table> ';

        print " </fieldset> ";
    }
?>
<script type="text/javascript" src="/assets/removal.js"> </script>
```

这个视图可能是迄今为止创建的最复杂的视图。首先系统会创建格式不错的联系方式名。随后，将显示另一个名为 contacts/viewsidebar 的视图。这个视图具有编辑和删除特定联系方式的链接。我们将在稍后对这些内容进行介绍。

下面的步骤是循环遍历指定联系方式具有的所有联系方式分组。\$view['groups']变量是对在视图外部创建的 contactgroups 对象的引用。每个分组都将采用接下来介绍的相同处理方式。

group dao 的标签属性将被首先显示。迄今为止，已能够使用 Business 或 Home 标签了。不过，当手动创建添加联系方式信息的接口时，可以看到重复的 Business 标签甚至更多外来的描述说明。接下来的步骤是从当前的分组中获取联系方法的集合。前面已经创建了这个具有导入功能的类，它应当在 \$methods 变量中填充一个充满 contactmethod dao 对象的 daocollection 对象。

针对每个被检索的方法，指定表中都会添加一条记录。不过，在显示具体值之前，会查看是否存在一个装饰器类。在计划过程中，我们讨论了一大堆能够应用于联系方法信息的装饰器。图 21-18 中的 UML 图说明了接下来要创建的装饰器。虽然给出了这样的创建方式，但是如果没有任何装饰器，具体方法仍然能够很好地执行。拥有装饰器并不是一个必要条件。进一步说，如果存在装饰器，那么使用装饰器的 decorate() 方法就能够装饰实例化与具体方法值。这个循环的最后一个步骤是使用方法类型和可能的装饰值打印相应记录。

视图的最底部会再次添加 removal.js 脚本，这是因为 contacts/viewsidebar 将具有用于当前联系方式的删除链接。

contacts/viewsidebar 视图是容易被遗忘的视图，这个视图位于 /views/default/contacts/viewsidebar.php 文件中：

```
<?php
    print ' <div class="sidebar"> <br/> <a class="featured"
        href="/contacts/edit/' . $view['id'] . '"> Edit ' . $view['contactname'] . ' </a> ';
    print ' <a class="featured removal" href="/contacts/processdelete/' . $view['id'] . '"> Delete ' . $view['contactname'] . ' </a> ';
    print ' </div> ';
```

这个视图简单地为当前联系方式生成编辑和删除链接。在后面的部分中，将创建处理这些链接的方法。

因为计划 Add and Edit Contact 页面上具有 JavaScript 扩展，所以决定不为移动浏览器提供这些选项。为了向移动浏览器提供另外的试图，已经在具有下面内容的 /views/mobile/contacts/viewsidebar.php 文件中创建了一个视图：

```
<div class="sidebar"> <br/> Editing is restricted to the browser only </div>
```

生成这些视图之后，可以看到导入过程的运行是成功的。用户能够查看联系方式、分组以及与之相关联的方法。不过，这些方法看上去有些空洞。下面的部分将解决这个问题。

## 6. 装饰器的编程

图 21-18 中的 UML 图表明要创建 5 个装饰器与 1 个装饰器接口。装饰器接口是我想创建的第一个元素，它将简单地实现 contacts 模块 view( )方法期望调用的 decorate( )方法的存在。如下所示，这个接口位于/includes/decoratorinterface.php 文件中：

```
<?php
interface decoratorinterface
{
    public function decorate($item);
}
```

下一个步骤是创建各个装饰器本身。第一个装饰器用于地址。希望使指定地址链接描述其位置的 Google Maps。我对 Google Maps 链接进行过一些研究，并且能够指出 URL 的工作方式。如下所示，decoratoraddress 类存在只具有一个方法的/includes/decoratoraddress.php 文件中：

```
<?php
class decoratoraddress implements decoratorinterface
{
    public function decorate($item)
    {
        $return = ' <a href="http://maps.google.com/maps?q=' .
                  urlencode($item)
                  . '" >' . $item . ' </a> ';
        return $return;
    }
}
```

所有这些装饰器将非常相似。每个装饰器都实现了装饰器接口，它们通过接受指定项来进行一定程度的装饰和更改。最后一个步骤是发送作为返回参数的更改项。上面的装饰器为 Google Maps 链接创建了正确编码的 URL，并且为其给出了地址标签。

下一个装饰器是 decoratoremail 类，它会将电子邮件地址更改为 mailto 链接。这个类位于具有下面内容的/includes/decoratoremail.php 文件中：

```
<?php
class decoratoremail implements decoratorinterface
{
    public function decorate($item)
    {
        $return = ' <a href="mailto:' . $item . '" >' . $item . ' </a> ';
        return $return;
    }
}
```

### 第 III 部分 PHP 设计案例分析

```

        return $return;
    }
}

```

某些移动电话支持无线电话应用接口(Wireless Telephone Applications Interface, WTAI)。因为公司颁发的 Windows Mobile 电话支持 WTAI 协议,所以我已经创建了一个在移动电话号码为联系方式时使用该协议的装饰器。这个装饰器会生成能够使用移动浏览器呼叫电话的链接,并且还将确认当前视图确实是一个移动视图。如果不能通过移动浏览器访问当前视图,那么 WTAI 协议将不起作用,因此不会执行该视图。/includes/decoratormobilephone.php 文件的内容如下所示:

```

<?php
class decoratormobilephone implements decoratorinterface
{
    public function decorate($item)
    {
        $return = $item;

        if (view::$viewtype == 'mobile') {
            $return = ' <a href="wtai://wp/mc;' . $item . '"> ' .
                $item . ' </a> ';
        }

        return $return;
    }
}

```

再下一个装饰器用于社交网络 URL。因为只被期望为简单的超级链接,所以这个装饰器会创建为一个 HTML Anchor 元素。decoratorsocialnetwork 类位于具有下面内容的/includes/decoratorsocialnetwork.php 文件中:

```

<?php
class decoratorwebsite implements decoratorinterface
{
    public function decorate($item)
    {
        $return = ' <a href="' . $item . '"> ' . $item . ' </a> ';
        return $return;
    }
}

```

最后一个装饰器用于站点联系方法。这个装饰器目前与社交网络 URL 完全相同,它位

于/includes/decoratorwebsite.php 文件中：

```
<?php
class decoratorwebsite implements decoratorinterface
{
    public function decorate($item)
    {
        $return = ' <a href="' . $item . '" > ' . $item . ' </a> ';
        return $return;
    }
}
```

此时，已完成计划期间所想到的装饰器集合的创建工作，并且得到了一个健壮的联系方式视图。在图 22-6 中，可以查看一个导入的联系方式。

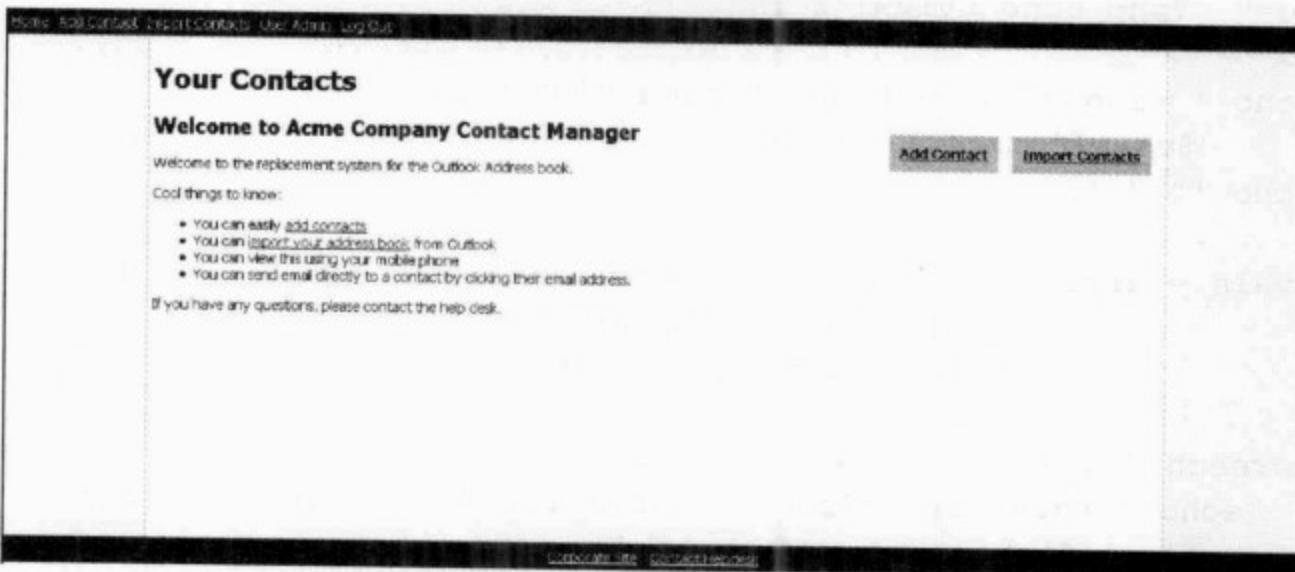


图 22-6

不过，仍然要开发添加、编辑和删除联系方式的方法。接下来的部分将介绍这些内容。

## 7. 添加联系方式

要添加联系方式，可以借鉴用户管理部分的 users/manage 视图。因此，准备尝试在联系方式管理中采用相同的机制。该过程的第一个步骤是显示题头中具有/contacts/add 链接的视图。为了实现这样的功能，将在 contacts 模块中添加下面的方法：

```
public function add()
{
    echo view::show('contacts/add');
}
```

### 第 III 部分 PHP 设计案例分析

这个方法会简单地载入/views/default/contacts/add.php 文件中的内容：

```
<?php
echo view::show('contacts/manage', array('title'=>'Add Contacts',
                                             'action'=>'/contacts/processadd',
                                             'formid'=>'addform',
                                             'type'=>'add'));
```

与添加用户的视图类似，这个视图为名为 contacts/manage 的共享视图构建参数。视图的标题将被定义，窗体动作将被指定，另外两个参数 formid 和 type 也会被转发至该视图。随后，将创建具有下面内容的/views/default/contacts/manage.php 文件：

```
<h2> <?php echo $view['title']?> </h2>
<form action=" <?php echo $view['action']?> " method="post"
      id=" <?php echo $view['formid']?> " >
<?php
    echo ' <input type="hidden" name="id" value="' .
          $view['contact']->id;
    echo '" />';

$vals = array('firstname'=>'First Name',
              'middlename'=>'Middle Name',
              'lastname'=>'Last Name');

foreach ($vals as $name=> $label) {
    echo ' <div class="row"> <label for="' . $name . '"> ' .
        . $label . ': </label> ';
    echo ' <input name="' . $name . '" id="' . $name .
        '" value="';
    echo $view['contact']->$name;
    echo '" /> </div> ';
}
?>
<hr/>

<?php
if (isset($view['groups'])) {
    foreach ($view['groups'] as $counter=> $group) {
        echo view::show('contacts/group',
                        array('group'=>$group,
                              'counter'=>$counter,
                              'type'=>$view['type']));
    }
}
```

```
$counter++;
$group = null;
}
else {
    $counter = 0;
    $group = new stdClass;
    $group->label = 'Business';
}
echo view::show('contacts/group',
    array('group'=>$group, 'counter'=>$counter,
        'type'=>'add'));

?>

<hr id="lastclone" />
<div> <label for="submit"></label>
    <input id="submit" type="submit" value=" <?php echo
        $view['title']?> "
        class="submitbutton"/>
</div>
</form>
<div id="contactgroupingcontainer">
    <?php echo view::show('contacts/group',
        array('label'=>'Business', 'counter'=>0))?>
</div>
<script type="text/javascript" src="/assets/managecontact.js"></script>
<script type="text/javascript">
var groupcount = <?php echo $counter?> ;
</script>
```

在按步骤编程之前，首先阐述一下基本的思路。创建名、中名和姓字段，随后构建一个用于所有联系方式分组的方框。如果不存在联系方式分组(也就是需要创建新联系方式的情况)，那么就会根据需求创建一个名为 Business 的分组。每个分组都具有创建新分组框的链接。这个过程通常会复制空白的隐藏分组框。

现在，每个分组框都具有一个特定的联系方法。不过，相应的值是空的。如果用户选择不添加联系方法，那么处理方法将确认不包含这个分组。每个联系方法也都具有一个创建其他联系方法的链接。这个链接使用了针对分组的类似复制方法。现在，让我们开始查看这个复杂视图的代码。

前几行代码相当简单。这些代码将检索和显示标题。在使用 add( )方法的情况下，标题会反映用户正在添加联系方式。随后，通过使用动作和 id 参数将构建具体的窗体。

编辑联系方式时，我计划传递入一个具有参数的 contact dao。下一行代码视图检索指定 contact 的 id 属性，并且将其存储为一个隐藏的 HTML Input 元素。添加联系方式时，这个字段是空的。

接下来，将 First Name、Last Name 和 Middle Name 元素添加到窗体。在任何一种情况下，contact dao 的公共属性都将被检索并填入成为这些字段的默认值。

窗体的下一个部分支持 contactgroup 体系结构。编辑联系方式时，一个 contactgroups daoco-llection 将被传递入这个视图。接下来的步骤是查看这个 contactgroups daocollection 是否存在。如果确实存在，那么每个集合的 dao 对象会被访问并传递至名为 contacts/group 的新视图。这个新视图期望接收一个 contactgroup、一个计数器变量以及该视图的类型。

在首次没有发送分组和添加联系方式的情况下，将会创建一个标准类。这个类被设计为模拟某个 contactgroup 的公共属性。读者需要注意标签如何被设置为 Business，以便满足第一个分组名为 Business 的需求。这将在条件语句外部实现。

创建 contacts/group 视图之前，希望查看这部分的最后一行代码。这行代码生成对 contacts/group 视图的另一个调用。在用户创建新 contact 的情况下，标签为 Business 的分组对象会被发送入该视图。很显然，这个对象没有其他特性，因此将创建默认空白的 Business 分组。如果用户正在编辑联系方式，那么可能已发生对 contacts/group 视图的多个调用。这个循环的最后一个步骤是增加计数器的值，并且将 group 变量设置为 NULL。随后，最后一个视图被执行时，新的联系方式分组框将被创建，但是标签仍然是空白的，这是因为传递的 \$group 并不具有任何公共属性。

创建 contacts/group 视图之前，还准备结束对这个视图的逐步跟踪。分组框被创建之后还会添加另一个水平条，此时它将具有一个 id 特性。在生成克隆时，JavaScript 会使用该特性。按下提交按钮将完成整个窗体的创建工作。

在窗体下面，生成了一个 id 特性被设置为 contactgroupingcontainer 的新 HTML Div 元素。该元素被用作一个键，JavaScript 可以使用这个键查找要复制的并且至于 lastclone 水平条之前的联系方式分组框的实例。在 div 分段内部将会执行对 contacts/group 视图的另一个调用，这是对联系方式分组框的实际克隆。

最后，还要添加两个 JavaScript 代码段。在/assets/managecontacts.js 中，JavaScript 会使用变量 groupcount。将在完成采用 PHP 的所有视图之后简要介绍这个 JavaScript 代码段。

因为已开发的 contact、contactgroup 和 contactmethod 对象的独特关系，所以视图的嵌套比较好理解。到目前为止，已经创建了 contact 视图。接下来，需要关注位于/views/default/contacts/group.php 文件中的 group 视图：

```
<div class="contactgrouping">
<div class="row"><label> Grouping: </label>
```

```
<input name="type[ <?php echo $view['counter']?> ][label]"  
      value=" <?php echo $view['group']->label?> " /> </div>  
<div>  
    <?php  
      if ($view['group'] instanceof contactgroup) {  
        $methods =  
          new contactmethodscollection($view['group']);  
        $methods->getwithdata();  
        foreach ($methods as $method) {  
          echo view::show('contacts/method',  
                          array('method'=>$method,  
                                'counter'=>$view['counter']));  
        }  
      }  
      echo view::show('contacts/method',  
                      array('method'=> null,  
                            'counter'=> $view['counter']));  
    ?>  
</div>  
  
<?php  
  if ($view['type'] == 'edit') {  
    echo ' <a href="#" class="deletecontactgrouping">  
          Delete this group </a> ';  
  }  
  else {  
    echo ' <a href="#" class="addcontactgrouping">  
          Add Another Grouping </a> ';  
  }  
?>  
</div>
```

每个联系方式分组都被封闭在类为 contactgrouping 的 div 分段中。第一个步骤是定义 group 的标签。编辑一个 contact 时，我会传递入一个完整的 contactgroup 对象，该对象具有标签公共属性。创建新的联系方式分组时，默认的 Business 标签将被提交给某个标准类，这个类将是使用标签的地方。

因为分组参数可能不是一个 contactgroup dao，所以需要使用一条简单的条件语句对此进行判断。这个方法复制框的体系结构与分组框非常相似。在父视图上，我会遍历循环任意分组，然后显示一个空白的分组框。此时，如果分组是一个 contactgroup，那么我知道至少存在一个 contactmethod。应循环遍历每个分组，并且采用适当的方法显示这些分组。接下来，无论如何都会生成一个空的方法框。

### 第 III 部分 PHP 设计案例分析

在判断分组参数是否位于一个 contactgroup 的条件语句内部时，将会创建一个新的 contactmethodsCollection 对象。按照要求，所有者 contactgroup 被发送入构造函数，并且数据通过 getWithdata() 方法被检索。随后，针对集合中的每个 contactmethod dao，都会显示 contacts/method 视图。

在介绍最后一个嵌套方法之前，将完成对当前视图的编程。如果用户正在编辑一个 contact，那么会显示删除分组框的选项。否则，添加新分组的选项将被显示。JavaScript 必须使这些链接能够提供适当的动作。

最后一个嵌套视图位于/views/default/contacts/method.php 文件中：

```
<?php
    echo ' <div class="row"> <label> Info: </label>
        <select name="type[' . $view['counter'] . '][methodtype][]" > ';

    $options = array(
        '' => '-Choose One-',
        'organization' => 'Organization',
        'title' => 'Title',
        'email' => 'Email',
        'website' => 'Website',
        'address' => 'Complete Address',
        'telephone' => 'Telephone',
        'mobilephone' => 'Mobile Phone',
        'socialnetwork' => 'Social Network URL',
        'im' => 'IM Name'
    );
    foreach ($options as $value=>$description) {
        echo ' <option value="' . $value . '" ';
        if ($view['method']->type == $value) echo 'selected="selected"';
        echo '>' . $description . ' </option> ';
    }

    echo ' </select> ';

    echo ' <span class="methodboxvaluebox" ';
    if ($view['method']->value) {
        echo 'hasvalue';
    }

    echo '> <input name="type[' . $view['counter'] .
        '][methodvalue][]" value="' . $view['method']->value .
        '" /> ';
```

```

if ($view['method']->value) {
    echo ' <a href="#" class="deletecontactmethod"> Delete this
          Info </a> ';
}
else {
    echo ' <a href="#" class="addcontactmethod" > Add More Info </a> ';

}

echo ' </span> </div> ';

```

计数器变量用于跟踪相配的联系方法和分组。接下来生成的选择框使用了一个预定义联系方法列表。因为它们全部被添加在选择框中，而且构建了具有括号的 HTML Input 元素的名字(在被提交给 PHP 时会被转化为一个数组)，所以能够满足每个分组允许多个联系方法类型的需求。

与分组视图一样，在添加和编辑方法之间共享了这个嵌套视图。选择框被生成时会不断查看 contactmethod 参数的类型属性。如果某个用户正在编辑一个方法，那么就会使用该参数传递入 contactmethod 对象。如果这是一个新方法，那么就不存在任何相应的对象，匹配也不可能成功。

联系方法类型为空时，并不希望突出显示指定的 HTML Input 元素。因此，我将该元素封装在 methodboxvaluebox 类中。这种特别的包含样式能够使指定元素不可见。不过，如果方法参数具有某个值，那么就要添加 hasvalue 类。这个类取代了 methodboxvaluebox 类，并且具有显示方法框的请求。

被创建的 HTML Input 元素的名称与 HTML Select 元素的名称相似，这有助于它们保持同步。如果 contactmethod 对象存在某个值，那么可将这个值插入该元素。与分组框类似，方法框也会显示两种链接中的某个链接。如果存在联系方法，那么就显示删除该方法的选项；否则，就显示添加其他联系方法的链接。JavaScript 需要理解这些链接并执行适当的功能性。

最后，结束 HTML Span 和 Div 标记完成了对方法框的定义。此时，就完成了添加或编辑一个 contact 所需的 3 个嵌套视图的设计。最后一个步骤是创建执行复制和可视性变化的 JavaScript 代码。这部分代码位于/assets/managecontact.js 文件中，并且被包含在 contacts/manage 视图中，其内容如下所示：

```

if (typeof jQuery != 'undefined') {
    $(function() {
        $("select").change(function(){
            if (this.value) {
                $(this).next().show().children('input').focus();

```

### 第 III 部分 PHP 设计案例分析

```
        }
        else {
            $(this).next().hide().children('input').val('');
        }
    });

    $(".addcontactgrouping").click(function(){
        groupcount++;
        var cloning =
            $("#contactgroupingcontainer").children('div');
        var myclone = cloning.clone(true);

        myclone.find('input,select').each(function() {
            this.value = '';
            this.name = this.name.replace(/\[\d*\]/, [
                ' '+groupcount + ']');
        });

        myclone.insertBefore($("#lastclone"));

        $(this).html('Delete this group').unbind('click').blur();
        $(this).click(function(){
            $(this).parent('div').remove();
        });

        return false;
    });

    $(".deletecontactgrouping").click(function(){
        $(this).parent('div').remove();
        return false;
    });

    $(".addcontactmethod").click(function() {
        var cloning = $(this).parent().parent();
        var myclone = cloning.clone(true);

        myclone.children('select').val('').trigger('change');
        myclone.children('span').children('input').val('');

        myclone.insertAfter(cloning);

        $(this).html('Delete this info').unbind('click').blur();
    });
});
```

```
$ (this).click(function() {
    $(this).parent().parent().remove();
});

return false;
});
$(".deletecontactmethod").click(function() {
    $(this).parent().parent().remove();

    return false;
});
});
});
```

接下来将简要介绍这个功能，从而使读者了解所发生的操作。首先，观察 HTML Select 元素。只要该元素发生变化，就会查看其是否被设置了某个值。如果不存在任何值，那么包含 Input 元素的 HTML Span 元素将被隐藏，Input 元素的内容也将被去除。相反地，如果存在某个设置值，HTML Span 元素会显示，并且 Input 元素将突出显示。

下一个功能是守候基于某个类添加联系方式分组的链接。这个功能性能会增加变量 groupcount 的值，该变量被用于将每个子方法与指定分组关联在一起。随后，作为 HTML Div 元素(被构建为容器)子部分的联系方式分组框将被克隆。在克隆中，为了反映新分组计数器，HTML Input 和 Select 元素的名称会有所变化。最后，克隆被添加至 ID 为 lastclone 的水平条的右侧。对于用户来说，这个克隆看上去就像在驻留最近单击链接的分组框下面添加的新分组框一样。这个功能性的最后一个步骤是将链接的文本修改为 Delete this group 并复位刚执行的动作。此时，指定链接的功能被修改为根据用户请求删除这个分组。

编辑分组时，生成的链接将 deletecontactgrouping 的类添加至 Delete this group。下一个功能将查找这个链接，并且去除保持被单击链接的联系方式分组。

接下来的方法在较小的级别上执行类似的操作过程。该方法复制了方法框，并且通过单击联系方法 HTML Input 元素右侧的链接被执行。这个方法通常生成 HTML Div 元素的克隆，同时将克隆插入相同的 Div 元素之后，此外还会通过将 HTML Select 元素修改为不具有任何值以及触发修改事件来确保复位被克隆元素的值。最后，这个方法会将刚单击的链接更改为具有相反的行为并匹配相关文本。

如果编辑已有方法时显示 Delete this info 链接，那么就会执行最后一个方法。这个方法会简单地删除父 HTML Div 元素，从而成功地从 HTML 窗体中删除指定的联系方法。

图 22-7 说明了添加一个联系方式的某些交互特征。

### 第 III 部分 PHP 设计案例分析

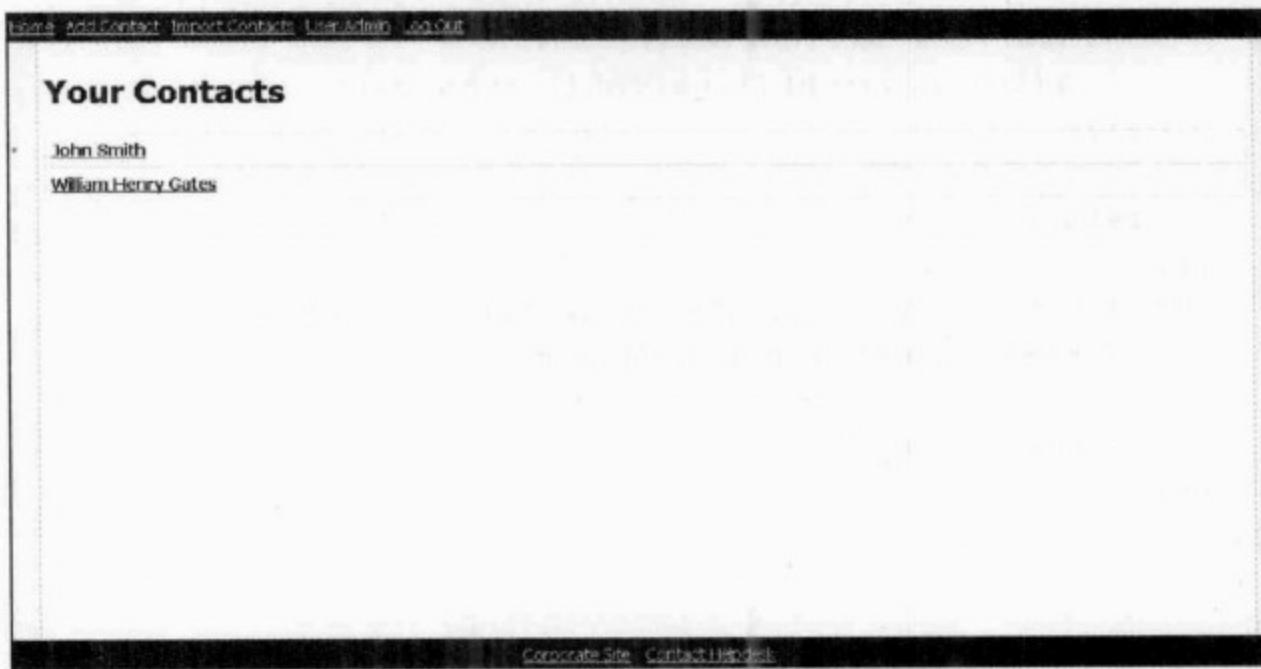


图 22-7

完成所有接口的编码后，接下来的步骤是处理方法。将窗体提交给/contacts/processadd，这意味着 contacts 模块必须具有 processadd() 公共方法。在 contacts 模块中添加下面的方法：

```
public function processadd()
{
    $firstname = $_POST['firstname'];
    $middlename = $_POST['middlename'];
    $lastname = $_POST['lastname'];
    $contact = new contact();
    $contact->firstname = $firstname;
    $contact->lastname = $lastname;
    $contact->middlename = $middlename;

    $currentuser = lib::getItem('user');

    $contact->ownerid = $currentuser->id;
    $contact->save();

    $this->addMethods($_POST['type'], $contact);

    lib::sendto("/contacts/view/{$contact->id}");
}
```

在大多数情况下，上面的方法相当简单。在调用受保护方法 addMethods() 时，才会看到 processadd() 方法真正的巧妙之处。不过，我并不想忽略实现这个方法的重要步骤。被提交的窗体中将会检索出名、中名和姓，并且它们会指派给一个新创建的 contact dao。

根据提交数据创建 contact dao 的最后一个步骤是指派 ownerid 属性。接着，当前 user 将被检索，并且该 user 对象的 id 属性会指派给这个 contact。最后，将调用 contact dao 对象的 save() 方法。这个方法能够填入 contact 对象的 id 属性。指定用户随后被发送至一个被构建用于查看新创建 contact 的 URL。

不过，在结束之前还会调用 addMethods() 方法。这个方法可接收两个参数：被提交的 type 数组与 contact 对象。在 contacts 模块中添加如下所示的 addMethods() 方法，

```
protected function addMethods($types, contact $contact)
{
    foreach ($types as $groupid=>$type) {
        if (!empty($type['label'])) {
            $group = new contactgroup();
            $group->contactid = $contact->id;
            $group->label = $type['label'];

            foreach ($type['methodtype'] as $methodtypekey=>$methodtype) {
                if (!empty($methodtype))
                    && !empty($type['methodvalue'][$methodtypekey])) {
                    if (is_null($group->id)) $group->save();
                    $method = new contactmethod();
                    $method->contactgroupid = $group->id;
                    $method->type = $methodtype;
                    $method->value = $type['methodvalue'][$methodtypekey];
                    $method->save();
                }
            }
        }
    }
}
```

考虑到 HTML Input 和 Select 元素名称的创建方式，因此构建了一个不错的多维数组 types。addMethods() 方法遍历循环 \$types 参数中存在的每个元素。首先，如果标签是空的，那么就会被跳过。标签键是联系方式分组框中的第一个 HTML Input 元素。因为允许生成的元素不需要任何内容，所以此处需要这个步骤。

接下来将创建一个新的 contactgroup。从作为该方法第二个参数的 contact dao 中可以检索出所有者 contact 的 id 属性，并且标签也会被添加。contactgroup 对象仍然未被保存，注意到这一点非常重要。因为 contactgroup 应当至少具有一个 contactmethod 子对象，所以我无法将其写至数据库。此外，并不知道是否还存在联系方法，因此 contactgroup 对象不得不保持未保存状态。

下一个步骤是循环遍历每个联系方法类型键。针对每个类型键，系统都会创建一条确认类型被设置并且被实际指派某些值的条件语句。如果通过条件判断，那么才会考虑生成一个新的 contactmethod 对象。

条件语句内部的下一步骤是检查新创建的 contactmethod 对象。如果该对象不具有 id 公共属性，那么就必须是被传递检测的第一个联系方法，并且需要被保存。需要注意的是，这个计算基于 id 属性的存在，而不是基于变量 \$methodtypekey。在得到一个有效方法之前，很可能存在一组空的方法。所有方法甚至可能都是空的。因此，不会采用 \$methodtypekey 值来认定这是第一个联系方法。

一个新的 contactmethod 对象会被创建。这个对象接收其需要的 contactgroup 所有者 ID 值、类型，并且被指派的值具有所接收的类型。最后，该对象将被保存。

此时，创建一个新 contact 的全过程宣告结束。通过这样的固化，contact 的编辑将更加容易。下面将进入联系方式编辑部分。

## 8. 编辑联系方式

每条联系方式记录都具有一个编辑链接。这个链接将引用 contacts 模块的公共方法 edit()。链接的最后部分包含打算操纵的 user dao 的 id 属性。在 contacts 模块中添加下面的代码：

```
public function edit()
{
    $controller = lib::getitem('controller');

    if (empty($controller->params[0])) {
        lib::sendto();
    }
    else {
        $contact = new contact((int) $controller->params[0]);
        $contactgroups = new contactgroupscollection($contact);
        $contactgroups->getwithdata();
        echo view::show('contacts/edit',
                        array('contact'=>$contact,
                              'groups'=>$contactgroups));
    }
}
```

通用使用存储的 controller 对象，采用与处理 user 编辑相同的方法就能够从指定 URL 中检索 id 属性。如果 id 属性为空，那么用户被简单地发送至其索引页面；否则，就会使用检索到的 id 属性创建一个 contact dao 的新实例。随后，该 contact 将作为所有者对象创建 contactgroupscollection 的一个实例。接下来则是调用填充指定对象的方法。最后，显示

名为 contacts/edit 的视图。contact dao 和 contactgroupscollection 对象作为参数传递。因为已经开发了共享管理视图的系统，所以读者可能能够预计/views/default/contacts/edit.php 文件的内容：

```
<?php
echo view::show('contacts/manage',
    array('title'=>'Edit Contact',
        'action'=>'/contacts/processedit',
        'formid'=>'editform',
        'type'=>'edit',
        'contact'=>$view['contact'],
        'groups'=>$view['groups']));
```

对 contacts/manage 视图的调用填入了其他大量参数。前 4 个参数与 contact 添加过程所使用的参数类似。标题和窗体动作会被设置，此外还会指定 HTML Form id 属性。最后，类型被设置为 edit。

这个视图的较大差异是被传递至视图的 contact dao 和分组 daocollection。目前，管理视图在尝试预填自身时才使用这些参数。

窗体的动作是引用 contacts 模块的公共方法 processedit()。为了创建这个方法，要添加如下所示的代码：

```
public function processedit()
{
    $firstname = $_POST['firstname'];
    $middlename = $_POST['middlename'];
    $lastname = $_POST['lastname'];
    $id = $_POST['id'];

    $contact = new contact($id);
    $contact->firstname = $firstname;
    $contact->lastname = $lastname;
    $contact->middlename = $middlename;
    $contact->save();

    $this->deleteMethods($contact);

    $this->addMethods($_POST['type'], $contact);

    lib::sendto("/contacts/view/{$contact->id}");
}
```

上面的方法与先前创建的 processadd() 方法非常相似。从被提交的窗体中可以检索到名、中名和姓。此外，当前 contact 对象的 id 属性也将被检索，并且被用于创建该对象的一个新实例。随后，几个名字被重写，指定对象将被保存。

删除所有联系方式是非常容易的，这比试图分析和更新当前存在的每个联系方式简单得多。除此之外，用户希望这个联系方式具有的任何联系方法应当被显示在前一个窗体上。对受保护方法 deleteMethods() 的调用会将当前 contact 传递进来。随后，受保护方法 addMethods() 将被调用，并且用户前进至单个联系方式视图页面查看他们所进行的改动。

受保护方法 deleteMethods() 的内容只是一个快速的 MySQL 操作。将在 contacts 模块中添加下面的内容：

```
protected function deleteMethods(contact $contact)
{
    $connection = db::factory('mysql');

    $sql = "delete g.*, m.* from contactgroup g
            left join contactmethod m on g.id=m.contactgroupid
            where contactid={$contact->id}";
    $results = $connection->execute($sql);
}
```

mysql 数据库连接对象的一个新实例将被检索。随后，创建一个使用所有者 contact dao 对象 id 公共属性的 MySQL 查询。这个查询考虑了分组与联系方法之间的关系。最后一个步骤则是执行这条删除语句。

随着这个方法的编码结束，应用程序整个联系方式管理部分的编程就完成了。多种不同设计模式的应用使得编程过程易如反掌。注意到：在整个编程过程中，只要知道计划使用何种设计模式，那么对象本身的创建几乎是本能的，并不需要很多额外的考虑。

这是一个很长的过程。许多地方能够进行完善，还有一些地方可以添加某些额外的安全性和差错检测。不过，在完成对这个项目首次编程之前，需要进行一次立即修改。

## 9. 用户管理中的数据完整性

因为在联系方式管理部分已经开发了一组关系，所以我发现自己在用户管理部分中的一个疏忽：删除某个 user 时，只有 user dao 对象的表记录会被删除。在系统中，这个 user 极有可能具有类似 contact、contactgroup、contactmethod dao 的对象，而我应当也删除上述对象。

在先前的部分中，针对 users 模块进行了编程。创建了一个名为 processdelete() 的公共方法，该方法处理指定 user 对象的删除操作。不过，应当将相应的 MySQL 语句替换为更

健壮的版本。创建应用程序的最终步骤是对 users 模块的 processdelete( )方法进行如下所示的修改：

```
$sql = "
    delete u.*, c.*, g.*, m.* from user u
    left join contact c on c.ownerid = u.id
    left join contactgroup g on g.contactid=c.id
    left join contactmethod m on g.id=m.contactgroupid
    where u.id = {$userid}
";
```

上面的语句不仅将删除指定的 user 对象，而且还会寻找和删除该对象与其他对象的关系。最终，需要确保系统中不存在任何孤立的 dao 对象。只有先完成整个联系方式管理部分之后才能够进行上述的适当操作。

## 22.3 本章小结

祝贺读者完成了对本章案例分析的学习。这个案例分析的目标是说明如何能够使用设计模式创建一个 PHP 应用程序。使用设计模式不仅使编码过程更快，而且有助于开发结构更为合理的应用程序。

应用程序利用了大量设计模式。在 contact、contactgroup、contactmethod 和用户类中使用了数据访问对象设计模式；在 userscollection、contactscollection、contactgroupscollection 和 contactmethodscollection 类中使用了迭代器设计模式；在 auth 和 db 类中使用了工厂设计模式；在 mysql 类中使用了单元素设计模式；在 decoratoraddress、decoratoremail、decoratormobilephone、decoratorsocialnetwork 和 decoratorwebsite 类中使用了单元素设计模式；在 importcontactsarraybuilder 类中使用了建造者设计模式的一个窗体；适配器设计模式则被用于创建 outlookcontactimportadapter 类。

现在，已经完成了对整个 ACME 公司联系方式管理器应用程序的编程。不过，与绝大多数编程工作一样，应用程序的第一个版本并不完美。第 23 章将诊断软件故障、修复安全问题，以及确定能够对基于设计模式的编程进行完善的地方。因为已经拥有基于 PHP 设计模式的、稳定的基本代码，所以我预计接下来最困难的部分将是发现问题，而不是修复问题。



# 第23章

## 使用更多设计模式进行改进

目前，我们已经完成了对应用程序的首次编程。完整的功能原型已被创建，按道理也为软件测试做好了准备。不过，仍然可以借助于更多设计模式对应用程序进行修改。

首先，需要考虑的是联系方式导入的功能。应用程序的这个特定部分与具体需求结合得非常紧密。虽然这在理论上似乎可以被接受，但是需求不会总是像我们希望的那样固定不变。在应用中，需求可能会变化为允许更多的联系方式导入类型。如果需求不进行改变，那么可能很快就会出现更新的请求。能够预见到所使用的应用程序可能被出售给不同的组织机构，因此，应当再创建应用程序的这个更抽象部分。在这里，将介绍一些有助于完成具体任务的设计模式。

其次，打算对视图系统进行讨论。对象的创建与用户接口的显示之间应当完全分离。某个特定联系方式视图的内容难以创建，我错误地允许了某些对象创建和数据检索方法调用进入这个视图。因此，还将使用某种模式来修正这个错误。

接下来，首先解决与联系方式导入功能有关的问题。

### 23.1 处理联系方式的导入

即使具体规范描述只指定一种导入类型，我仍然决定采用更多的导入类型，以免到时候手忙脚乱。目前，具体需求是接受正式的公司 Microsoft Outlook 通信录的联系方式导入。尽管某些雇员觉得这个工具易于使用，不过我预计会出现迁移其他非正式地址簿的需求。就像公司需要开发作为正式地址簿支持工具的软件一样，用户始终需要使用其他软件(IT 从业人员更是如此)。

这个过程存在保持紧密耦合的两个部分。第一个部分是操纵联系方式名字信息的 Adapter 对象。另一个部分则是通过构建某个数组从而遍历文件内容的对象。下面将对它们分别进行讨论。

### 23.1.1 Outlook 联系方式适配器

目前，系统将导入从 Outlook 中导出的 CSV 文件。不过，并不希望由于给出过多不可用选项而使软件用户产生混淆。通过使用某些设计模式，准备将具体代码的处理部分修改得更加灵活。用于导入的视图看上去是相同的。

已经创建了如图 23-1 所示的 UML 图。

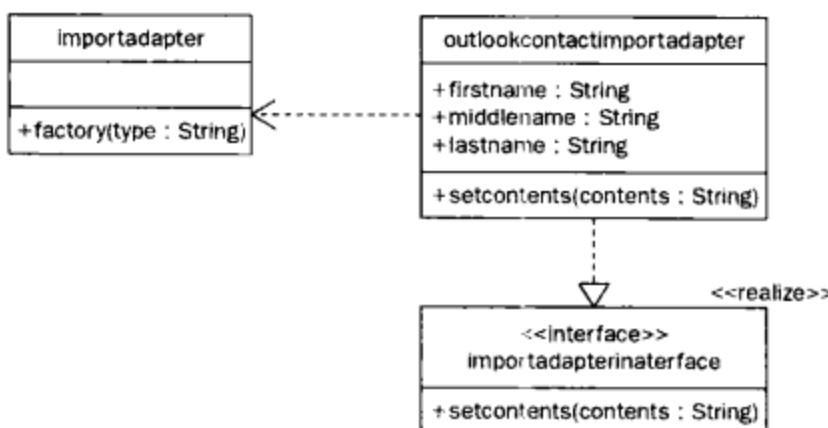


图 23-1

`importadapter` 类具有一个名为 `factory()` 的方法，该方法是工厂设计模式的实现。`factory()` 方法接受字符串参数 `type`，它将被用于创建适配器对象的新实例。

`outlookcontactimportadapter` 类将被更改为使用 `setcontents()` 方法来代替完成构造函数中的所有逻辑。

最后，`outlookcontactimportadapter` 类会实现 `importadapterinterface` 接口。这个接口将使今后创建的任何适配器对象保持一致性。

我准备做的第一件事情是指定用户执行导入的 `type`。通过在窗体中添加一个隐藏字段，就可以完成这个操作。此外，准备较为模糊地命名 HTML File Input 元素。因为以后上传的可能不是 CSV 文件，所以目前不进行明确的命名。

在 `/views/default/contacts/import.php` 文件中，已经进行了如下所示的修改：

```

<h1> Import Your Contacts </h1>
<p>
    Upload a .CSV file from Outlook. Contact helpdesk for assistance.
</p>
<?php
    echo view::show('standard/errors');
?>
<form action="/contacts/processimport" method="post"
    enctype="multipart/form-data">
    <input type="hidden" name="importtype" value="outlook"/>
  
```

```

<div class="row">
    <label for="contactsfile"> Contacts File: </label>
    <input type="file" id="contactsfile" name="contactsfile"/>
</div>
<div class="row">
    <label for="submit"></label>
    <input id="submit" type="submit" class="submitbutton" value="Upload"/>
</div>
</form>

```

添加隐藏的 HTML Input 元素 importtype 是第一个变化。这个元素将包含指定要创建导入类型的值 outlook。

下一个变化是 HTML File Input 元素的 name 和 id 特性。这些值现在变为更抽象的 contactsfile。

既然已经更改了这个视图，那么下一个步骤是执行 contacts 模块的 processimport() 方法。我准备创建一个作为工厂设计模式实现的新类，而不是创建 outlookcontactimportadapter 类的一个新实例。首先，对 processimport() 方法的更改：

```

if (is_uploaded_file($_FILES['contactsfile']['tmp_name'])) {
    $contents = file_get_contents($_FILES['contactsfile']['tmp_name']);
    unlink ($_FILES['contactsfile']['tmp_name']);

    $builder = new importcontactsarraybuilder($contents);
    $imports = $builder->buildarray();

    $currentuser = lib::getitem('user');

    foreach ($imports as $import) {
        $contact = new contact();
        $adaptor = importadapter::factory($_POST['type']);
        $adaptor->setcontents($import);

        $contact->firstname = $adaptor->firstname;
        $contact->middlename = $adaptor->middlename;
    }
}

```

### 注意：

要了解工厂设计模式的更多信息，请参见第 9 章。

检查上传文件的条件语句，这里存在一个小变化。此时，该语句查看文件是否被命名为 contactsfile(而不是 CSV)。这与我应用于视图的变化相匹配。

### 第 III 部分 PHP 设计案例分析

向后移动若干行，读者会看到导入循环的第一行代码仍然相同。现在，不必创建 outlookcontactimportadapter 的新实例，而是调用 importadapter 类的公共静态方法 factory()。被提交的 type 值会被传递入该方法，type 应当包含在先前更改的视图中指定的字符串 outlook。接下来，通过使用导入的内容执行特定适配器的 setcontents() 方法。随后，这个方法像在代码先前迭代中一样继续其操作。

准备执行的下一步骤是创建 importadapter 类。这个类位于具有下面内容的 /includes/importadapter.php 文件：

```
<?php
class importadapter
{
    public static function factory($type)
    {
        $classname = "{$type}contactimportadapter";
        return new $classname;
    }
}
```

importadapter 类具有一个名为 factory() 的公共静态函数。这个函数接受和利用单个参数来构建联系方式导入适配器类的名字。随后，联系方式导入适配器类被创建并返回至调用者函数。

最后，为了能够正确地完成变化，需要更改 outlookcontactimportadapter 类。现在，/includes/outlookcontactimportadapter.php 文件的内容如下所示：

```
<?php
class outlookcontactimportadapter implements importadapterinterface
{
    public function setcontents($import)
    {
        $this->firstname = $import['First Name'];
        $this->middlename = $import['Middle Name'];
        $this->lastname = $import['Last Name'];
    }
}
```

为了实现 importadapter 工厂设计模式的添加，outlookcontactimportadapter 类只需要进行两个简单的修改。首先，构造函数被重命名为 setcontents()。除此之外，该函数不存在其他变化。另一处修改是 importadapterinterface 的实现。因为预计将来会创建更多这样的适配器，并且不想修改 contacts 模块，所以希望实现 setcontents() 方法。

/includes/importadapterinterface.php 文件的内容如下所示：

```

<?php
interface importadapterinterface
{
    public function setcontents($contents);
}

```

完成上述变化后，适配器更为灵活，同时仍然能够提供需求中要求的基于 Outlook 的功能性。

### 23.1.2 构建联系方式数组

延续使联系方式导入更为抽象的思路，我注意到另外一个能够使用各种设计模式的区域。初始文件被上传时，就会根据文件的内容构建一个 PHP 数组。这个逻辑将在 Outlook 的 CSV 文件专用的方法中被执行。我认可构建数组的方式，但是认为该逻辑应当在其他类中。这个时候非常适合使用委托设计模式。我准备让数组构建类把实际负责的将字符串解析至某个数组的操作委托给另一个类。

**注意：**

要了解委托设计模式的更多信息，请参见第 7 章。

在分析期间，生成了如图 23-2 的 UML 图。

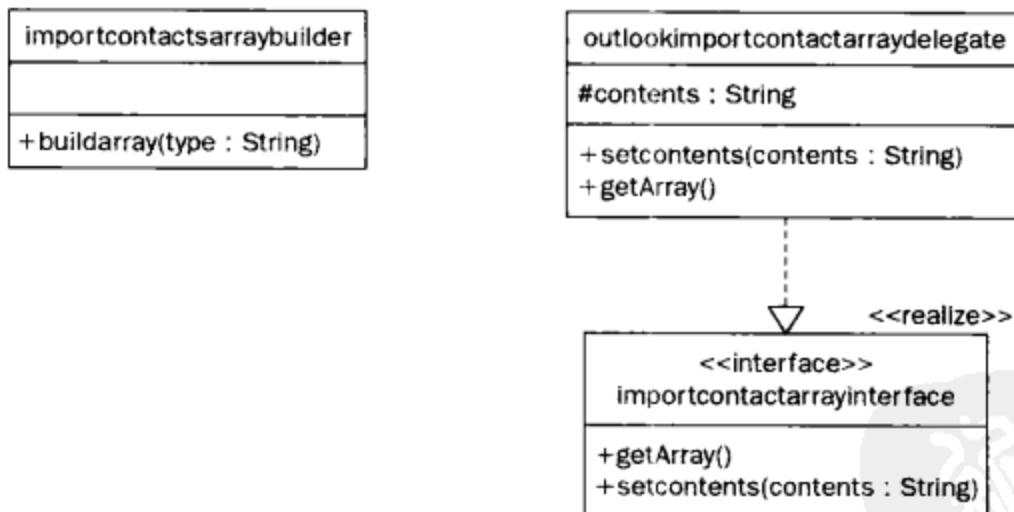


图 23-2

`importcontactsarraybuilder` 类被创建用于根据输入字符串构建数组。这个类具有一个名为 `buildarray()` 的方法，该方法接受一个类型字符串，从而生成适当的调用来构建数组。

`outlookimportcontactarraydelegate` 类被用于处理 Outlook 形式的联系方法导入。这个类在其受保护属性 `contents` 中存储联系方式，并且具有两个公共方法。第一个方法是 `setcontents()`，

### 第 III 部分 PHP 设计案例分析

它负责接受联系方式导入字符串并将其指派给指定对象。第二个方法是 `getArray()`，它负责处理受保护属性 `contents` 的内容并返回一个数组。

最后，还需要 `importcontactarrayinterface` 在所有委托对象中实现一致性。

如下所示，第一个变化将再次应用于 `contacts` 模块的 `processimport()` 方法：

```
unlink ($_FILES['contactsfile']['tmp_name']);

$builder = new importcontactsarraybuilder($contents);
$imports = $builder->buildarray($_POST['type']);

$currentUser = lib::getitem('user');
```

前两行代码是相同的，并且被用于上下文。发生的变化是调用 `importcontactsarraybuilder` 类公共方法 `buildarray()` 的方式。现在，要传送被提交的导入类型。根据对视图的最新修改，导入类型将设置为 `outlook`。

接下来，对 `/includes/importcontactsarraybuilder.php` 文件进行了一些修改。现在，`buildcollection()` 方法具有如下所示的内容：

```
public function buildcollection($type)
{
    $classname = "{$type}importcontactsarraydelegate";

    $delegate = new $classname;
    $delegate-> setcontents($this->importedstring);
    $array = $delegate->getArray();

    return $array;
}
```

`buildcollection()` 方法现在接受单个参数 `$type`。随后，使用该参数能够构建一个新的类名。这个类的新实例被指派给变量 `$delegate`。通过调用将导入内容作为参数的公共方法 `setcontents()`，导入的内容会被传递入这个基于委托设计模式的类。最后，调用指定对象的 `getArray()` 方法可以检索变量 `$array`。这个方法的最后一行代码十分常见：它将变量 `$array` 作为返回参数发送。

查看了变量 `$classname` 的结构，并且知道能够检索的唯一值是 `outlook`，将创建一个新类 `outlookimportcontactarraydelegate`。已经去除了 `buildcollection()` 方法中的逻辑，并且准备将其插入新类。在具有下面内容的 `/includes/outlookimportcontactsarraydelegate.php` 文件中可以找到这个新类：

```
<?php
class outlookimportcontactarraydelegate implements
```

```

import contactarrayinterface

{
    protected $contents;

    public function setcontents($contents)
    {
        $this->contents = $contents;
    }

    public function getArray()
    {
        $lines = explode("\n", $this->contents);
        $keys = explode(',', array_shift($lines));

        $array = array();

        foreach ($lines as $line) {
            if (!empty($line)) {
                $keyed = array_combine($keys, explode(',', $line));
                $array[] = $keyed;
            }
        }
        return $array;
    }
}

```

`outlookimportcontactarraydelegate` 类只包含两个公共方法，`importcontactarraybuilder` 类会调用这两个方法。第一个方法 `setcontents()` 接受名为 `$contents` 的参数，该参数将被指派给这个类中的受保护变量 `$contents`。

读者应当非常熟悉 `getArray()` 方法，它直接复制了 `buildcollection()` 方法先前的内容。只是将第一行代码修改为引用 `$this->contents`，而不是引用 `$this->importedstring`。

最后，因为可能创建多个 `Delegate` 对象，所以外部接口应当相同。`/includes/importcontactarrayinterface.php` 文件包含了如下所示的、实现这个公共方法的接口：

```

<?php
interface importcontactarrayinterface
{
    public function getArray();
    public function setcontents($contents);
}

```

在数组构建类内部使用委托设计模式是一个非常不错的想法。Outlook 的 CSV 文件相当

容易被解析。不过，如果添加了其他类型，那么其解析工作可能不会太容易。对于这样的场景来说，将这个逻辑委托给外部对象是最佳的做法。

## 23.2 去除视图中的逻辑

我努力将逻辑置于应用程序的视图外部。不过，无论是因为疏忽还是因为编程人员的懒惰，某些代码有时会出现在视图内部。这是非常糟糕的：数据应当从不执行任何数据或对象操纵任务。相反，整个信息集应当被提供给视图。视图能够决定所显示的具体信息。

在这个应用程序中，拥有多个视图的需求使得去除视图中的逻辑更为重要。如果默认视图包含构建对象的某些对象，那么移动视图必须包含完全相同的逻辑。如果添加第三个视图，那么这个逻辑也必须被复制到该视图。这样会产生代码复制，而且不是最佳的做法。

在这个应用程序中，我自己知道最佳做法，而是故意错误地允许 contact dao 关系检索处理的复杂性。接下来，将描述存在问题之处以及如何进行修正。

### 更改联系方式的单个视图

为了构建查看单个联系方式的视图，在视图外部提前收集信息的工作变得非常复杂。对于这种复杂性来说，应当放弃使用某种设计模式。相反，会创建一个 contact dao 和一个 contactgroupscollection 对象。在视图内部，会创建单独的 contactmethodcollection 对象。使用这样的方式时，就在视图内部执行逻辑。取而代之的是，所有信息应当被传递至视图，从而能够被显示。此时，应用程序不必要求获得完成上述操作的新对象。

因为收集所有联系方式信息是一个复杂的过程，所以决定使用外观设计模式。不必在执行单个联系方式视图之前创建一堆逻辑，而是可以创建一个执行所有逻辑的新外观对象，随后再将需要的信息传递至输入视图。

**注意：**

要了解外观设计模式的更多信息，请参见第 8 章。

如图 23-3 所示的 UML 图详细描述了我想要进行的修改。

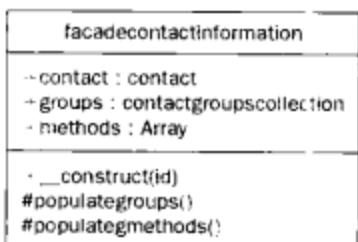


图 23-3

`facadecontactinformation` 类很可能相当复杂。不过，指定对象的实例化相对简单，而这是外观设计模式的迷人之处。公共函数 `_construct()` 接受一个参数，该参数是被请求 `contact dao` 的整型 ID。这个方法首先将 `contact dao` 指派给公共属性 `contact`，然后再调用两个受保护方法。

受保护方法 `populategroups()` 负责为属于 `contactgroupscollection` 对象的 `contact dao` 创建该对象的一个实例，它会将 `contactgroupscollection` 对象指派给公共属性 `groups`。

最后，受保护方法 `populatemethods()` 会被调用。这个方法负责为这个类内存储的 `contactgroupscollection` 对象中的每个 `group dao` 创建所有必要的 `contactmethodscollection` 对象实例。每个 `contactmethodscollection` 对象都会被添加至公共的数组属性 `methods`。

第一个要进行修改的区域是 `contacts` 模块。`view()` 方法具有显示单个联系方式的逻辑。应将其修改为下面的内容：

```
public function view()
{
    $controller = lib::getitem('controller');

    if (empty($controller->params[0])) {
        lib::sendto();
    }
    else {
        $params = new facadecontactinformation((int) $controller->
                                                params[0]);
        echo view::show('contacts/view',
                        array('contact'=>$params->contact,
                              'groups'=>$params->groups,
                              'methods'=>$params->methods));
    }
}
```

在这个方法中，条件语句的第一部分是相同的。不过，现在不会创建一个新的 `contact dao` 和一个新的 `contactgroupscollection` 对象，而是创建 `facadecontactinformation` 类的一个新实例。像先前 `contact dao` 所接受的一样，来自控制器的标识符会传递至这个实例。

显示视图的代码行也进行了修改。参数现在是 `facadecontactinformation` 对象的公共属性，而不是指向局部变量。除了 `contact` 和 `groups` 键之外，`methods` 键也使用相关联的公共特性进行定义。

下一个步骤是查看位于 `/includes/facadecontactinformation.php` 文件的 `Façade` 对象(如下所示)：

### 第 III 部分 PHP 设计案例分析

```

<?php

class facadecontactinformation
{
    public $contact;
    public $groups;
    public $methods = array();

    public function __construct($id)
    {
        $this->contact = new contact($id);
        $this->populategroups();
        $this->populatemethods();
    }

    protected function populategroups()
    {

        $this->groups = new contactgroupscollection($this->contact);
        $this-> groups->getwithdata();
    }

    protected function populatemethods()
    {
        foreach ($this->groups as $group) {
            $this->methods[$group->id] = new
                contactmethodscollection($group);
            $this->methods[$group->id]->getwithdata();
        }
    }
}

```

这个类首先定义了 3 个将被检索用于显示视图的公共属性。接下来，构造函数被定义为接受单个参数\$id，它用于将 contact dao 的一个新实例指派给公共属性\$contact。随后，在实际的 Façade 窗体中，通过调用其他两个受保护方法能够掩蔽其余逻辑。

受保护方法 populategroups() 看上去可能是相似的，它几乎是 contacts 模块中逻辑的完全副本。不过，这个方法没有使用局部变量\$contact，而是使用来自当前实例的\$contact 属性。contactgroupscollection 对象会被指派给公共属性\$groups。

为了替换视图中的逻辑，创建了受保护方法 populatemethods()。与视图中原有循环相似的一个循环将被创建。随后，针对作为当前实例\$group 属性一部分的每个分组，都会生成由\$methods 属性组成的一个条目。当前分组对象的 id 属性可以键控这个数组。接着，具

体值被设置为根据当前分组构建的 contactmethodscollection 对象的一个实例。最后，将会调用 contactmethodscollection 对象的 getwithdata( )方法，它将完全填充与指定联系方式所具有的每个联系方式分组相关联的 \$methods 属性。

利用这个新 Façade 对象所需的最后一个变化是更改 contacts/view 视图。如下所示，修改了 /views/default/contacts/view.php 文件的下列代码行：

```
print "<fieldset> <legend> {$group->label} </legend>";

print ' <table> ';
foreach ($view['methods'][$group->id] as $method) {
```

删除了分组标签之后的两行代码。创建 contactmethodscollection 对象的一个新实例不再需要这些代码。随后，接下来的 foreach 循环被更改为引用发送入视图的 methods 参数的适当键。

完成上述修改之后，这个视图不再直接处理对象的创建。现在，它已成为一个将逻辑与视图分离的优秀示例。

## 23.3 尽力实现设计模式

我不喜欢“准备工作”，而是希望在实践中学习。读者已经看到，通过应用更多的设计模式，我已经进入了应用程序的第二轮编码工作。但是，这个工作并未完成。希望读者阅读下面这些基于设计模式的内容，并且针对前面的基本代码亲自动手实现自己的设计模式形式。

### 23.3.1 设计模式和错误检查

这个应用程序的优点之一是几乎不需要错误检查。只有在用户创建中才需要进行实际的错误检查。这个错误检查过程会简单地确认用户是否存在。我们还能通过什么方式实现使用 PHP 设计模式的错误检查呢？

读者可以仔细研究策略设计模式，以便在各种字符串类型中查看有效的类型。例如，因为知道用户名或密码应当不是空字符串，所以可以创建一个验证指定字符串至少包含一个字符的 Strategy 类。

**注意：**

要了解策略设计模式的更多信息，请参见第 17 章。

另一个被考虑的方式是在调用 `save()` 方法之前可以使用 `Visitor` 对象访问 `dao` 的实例。这个对象能够验证必要的字段，并且标记 `dao` 是否有效。在创建和更改过程中都可以使用这种方式。

读者还可以创建一个名为 `isValid()` 的方法，该方法利用某些 `Delegate` 对象来查看指定对象的各种属性。随后，响应 `isValid()` 方法的布尔值能够被用于确定 `dao` 是否应当调用其 `save()` 方法。

### 23.3.2 设计模式和联系方式管理

第 23.2 节讨论了从应用程序视图中去除逻辑的最佳做法。遗憾的是，这个应用程序仍然具有一个存在对象逻辑的特定视图：`contacts/manage` 视图。怎样才能取出这个视图中的逻辑呢？

类似于在单个联系方式视图中那样创建 `Façade` 对象的一个实例是一个可接受的解决方案。一旦这个 `Façade` 被构建在 `contacts` 模块的 `edit()` 方法中，那么就能够去除 `contacts/manage` 视图中的逻辑。

### 23.3.3 设计模式和视图类型

视图类具有一个名为 `setViewtype()` 的静态方法。目前，这个特定方法相当简单。不过，随着时间的推移，请求的视图类型越来越多，该方法将变得更加复杂。那么，使用哪一种设计模式能够处理这种复杂性呢？

委托设计模式能够被用于创建一个尝试推断视图类型的对象。这个对象可以在 `setViewtype()` 方法之内被调用。

如果请求一个标识视图类型的通用方法（例如分析用于某个标识符的 HTTP User Agent），那么也可以使用解释器设计模式。这个类将被用于基于传递的信息来解释视图类型。

注意：

要了解委托设计模式的更多信息，请参见第 7 章。要了解解释器设计模式的更多信息，请参见第 10 章。

### 23.3.4 设计模式和删除对象

要从数据源中删除某个对象，目前的做法是创建一条复杂的 MySQL 语句。如果应用程序不得不迁移至不同的数据源，那么这些语句可能需要被更改。使用基于哪一种设计模式的对象能够提供这样的灵活性？

中介者设计模式将是一个不错的选择。删除某个用户时，还会通知应当也被删除的所有 contact dao。在类似情况下，中介者能够帮助将某个 contact dao 的删除操作向下转移至该联系方式所有的 contactgroup 和 contactmethod 对象。

注意：

要了解中介者设计模式的更多信息，请参见第 12 章。

### 23.3.5 分享您的设计模式完善工作

请尽力为上面已提及的应用程序部分给出一些基于设计模式的解决方案。这个过程有助于读者练习设计模式的使用，从而能够在今后的应用程序编程中熟练地加以应用。

如果发现使用 PHP 设计模式解决现有问题的新方法，那么请告诉所有人。将这些方法与同事进行分享，或者发布在自己的博客中。这种做法一箭双雕。您不仅为开源社区做出了贡献，而且能够鼓励其他开发人员进行更好的编码实践。

查看这些代码后，读者既可以创建自己的解决方案，也可以只更改这个应用程序，不管怎样都请您告诉我。我的博客地址是 <http://www.aaronsaray.com/blog>。请告诉我您的心得，与所有人分享您的最新成果，甚至可以阐明为了达到最佳效果如何更改我的代码。

## 23.4 本章小结

本章主要阐述了对应用程序体系结构的完善。已有代码会被更新为使用基于某些设计模式的新对象，这些设计模式在基准章节中都进行过介绍。

导入联系方式功能性是被更改的第一个部分。目标是将 Outlook 需求与实际的联系方式导入逻辑相分离。首先，基于工厂设计模式的对象被创建用于生成 Adapter 对象，从而不必生成一个围绕 Outlook 设计的实例。接下来，引入一个 Delegate 对象，该对象被用于根据来自联系方式文件导入的信息来构造一个数组。与 Factory 对象类似，这会将紧密耦合对象中的代码移动到更为专用的 Delegate 对象中。上述变化带来了极大的灵活性。

本章关注的另一个方面是视图系统。在一个特定的视图中，允许了对象创建操作。Facade 对象被创建用于以更隐蔽的方式完成某些现有逻辑。随后，能够将具体代码从视图中迁移至 Facade 对象内。这会生成一种收集联系方式信息的简单方式，并且巩固了视图与逻辑之间的屏障。

本章的最后部分为读者指出了应用程序中能够从设计模式受益的其余方面。可能的解决方案提供了各种机会。我非常鼓励读者根据这些建议继续更新应用程序，从而获得更多的实践经验。

祝贺您阅读完本书。很荣幸有机会能与您分享这些知识。希望读者能够和我一起见证 PHP 的发展和成熟。请继续创建使用 PHP 设计模式的开放源代码软件应用程序，从而使 PHP 社区中出现更多严谨的设计和体系结构。



PHP 是一种引发 Internet 风暴的服务器端编程语言。本书通过详细的示例说明和可供读者免费下载的代码示例，深入探讨了如何配置 PHP、创建已启用 PHP 的 Web 页面，以及如何实践应用每种高级开发工具。本书也详细介绍了如何个性化 PHP 工作空间，定义运算符和变量，操作字符串和数组，部署 HTML 表单和按钮，以及如何处理用户输入等内容。除此之外，本书还将深入探讨 PHP 的下一代 Web 2.0 设计功能，包括 AJAX、XML 和 RSS。

书 名：PHP 完全参考手册

作 者：(美) Steven Holzner

译 者：曹俊 徐剑

ISBN：9787302205609

定 价：68.00 元

出版社：清华大学出版社

### 本书主要内容：

- ◀ 集成 HTML 控件、文本字段、表单、单选按钮和复选框。
- ◀ 使用服务器端文件存储器构建博客、客户留言簿和反馈页面。
- ◀ 编写检索、修改和更新数据库信息的 MySQL 脚本。
- ◀ 设置 cookie，执行 FTP 事务，并从 PHP 会话中发送电子邮件。



通过结合 AJAX 和 PHP 的强大功能，基于 Oracle 的 Web 应用程序可以提供下一代用户功能和客户端服务。本书由 Web 2.0 编程专家编写，详述了如何设计出更快、更轻量、更具交互性的 Web 应用程序，同时又可以减少代码支持和创建时间。在本书中，您将学会如何开发模块化的 AJAX 和 PHP 代码、添加动态更新的内容以及实现单击编辑和拖放功能。此外，本书也详述了 Mashup、报告与通信等这些前沿技术。

书 名：AJAX+PHP+Oracle 数据库 Web 开发 作 者：(美) Lee Barney 等

译 者：赵利通 付勇

ISBN：9787302194743

定 价：58.00 元

出版社：清华大学出版社

### 本书主要内容：

- ◀ 使用最新的 AJAX、PHP 和模块化技术构建健壮的 Web 应用程序。
- ◀ 构建 AJAX 和 PHP 对象、控制器、iframes 和模型。
- ◀ 使用 XMLHttpRequest 对象自动刷新 Web 页面及执行 XML 调用。
- ◀ 使用远程过程调用和基于 Pear 的服务从外部源获取数据。
- ◀ 使用 VOIP、IM、有针对性的报告和文件共享方便客户端通信。



本书由精通 PHP 与 MySQL 应用的 PHP 专家编写。书中一共介绍了 12 个功能完整的程序，这些程序涵盖了在开发基于 Web 的应用程序时所需的 PHP 与 MySQL 基本概念和语法。在按顺序学习每个程序的细节时，您将获得许多应用程序的设计技巧，以及为程序添加安全措施以适应实际需要的方法。无论是希望了解各种程序设计的入门知识，还是想寻找编写程序所需的灵感，或者是想修改并重用书中的代码，都可以通过学习本书提升自己的技能。

书 名：PHP & MySQL 范例精解

作 者：(美) Tim Boronczyk 等

译 者：熊伟

ISBN：9787302195627

定 价：48.00 元

出版社：清华大学出版社

### 本书主要内容：

- ◀ 创建基础的用户注册系统。
- ◀ 扩展用户注册系统来构建一个社区论坛，并支持用户权限和按话题发帖。
- ◀ 创建由 AJAX 技术支撑的文件上传和目录浏览系统。
- ◀ 创建支持评论和 RSS 提要功能的新闻或博客系统。



本书将为读者理解 PHP、Apache 和 MySQL 这 3 个核心组件如何独立工作和协同工作奠定良好的基础，引导读者充分利用它们提供的各种功能。本书首先通过两个项目介绍了如何利用这些技术的重要功能创建完整的网站。这两个项目引导读者了解基础知识，如编写 PHP 代码、建立 MySQL 数据库、用数据填充数据库，以及向访客显示特定的信息等。接着，在完成每个网站的开发过程中，讨论了一些使用 PHP、Apache 和 MySQL 的复杂主题。

书 名：Web 开发入门经典

作 者：(美) Timothy Boronczyk 等

——使用 PHP6、Apache 和 MySQL

译 者：薛焱

ISBN：9787302203759

定 价：88.00 元

出版社：清华大学出版社

### 本书主要内容：

- ◀ 创建、修改和处理图像文件的技巧。
- ◀ 建立内容管理系统。
- ◀ 通过活动日志和错误日志监控网站。
- ◀ 建立电子邮件列表，处理用户的注册。

# PHP设计模式

## Professional PHP Design Patterns

PHP通常被视为构建灵活和功能丰富的Web应用程序的基石。由于学习相对简单，因此初级编程人员能够方便地采用PHP从头开始创建应用程序。然而，考虑到这些应用程序的规模，其维护工作令人畏惧。幸运的是，成熟的编程社区已经提供了包括设计模式在内的一些解决方案。这些针对相同问题的可重复解决方案已在其他编程语言中应用了多年。《PHP设计模式》的目标是：通过将经过实践检验的设计模式应用于PHP应用程序，将PHP与早期的编程语言联系在一起。

《PHP设计模式》首先介绍了设计模式，讲述了设计模式的使用及重要性，并且详细说明了应用设计模式的场合。接下来，本书通过代码示例介绍了许多设计模式。最后，本书通过全面深入的案例分析说明了如何使用设计模式来计划新的应用程序，如何采用PHP语言编写这些模式，以及如何使用书中介绍的设计模式修正和重构已有的代码块。作者采用专业的、便于使用的格式来介绍相关的概念，自学成才的编程人员与经过更多正规培训的编程人员都会非常熟悉这种格式。

### 主要内容

- ◆ 设计模式的含义及其在PHP中的重要性
- ◆ 建立设计模式的PHP库的常规用法
- ◆ 使用设计模式(如建造者、外观、原型和模板模式)更快速、更有效地创建对象
- ◆ 使用数据访问对象和代理设计模式分离体系结构
- ◆ 使用委托、工厂和单元素设计模式改善代码流和控制
- ◆ 在不修改对象核心的情况下实现访问者和装饰器设计模式以增强对象
- ◆ 全面的、使用观察者设计模式的插入式创建技术
- ◆ 使用设计模式计划应用程序体系结构
- ◆ 使用基于设计模式的编码方式从头至尾地完成一个Web站点编程的案例分析
- ◆ 在已有应用程序中发现设计模式的体系结构优势
- ◆ 通过案例分析介绍进一步增强代码的方法

### 读者对象

本书适用于有一定经验的、希望借助设计模式来改善代码体系结构质量的PHP编程人员。

### 源代码下载及技术支持

<http://www.wrox.com>

<http://www.tupwk.com.cn/downpage>

**Wrox Professional guides** are planned and written by working programmers to meet the real-world needs of programmers, developers, and IT professionals. Focused and relevant, they address the issues technology professionals face every day. They provide examples, practical solutions, and expert education in new technologies all designed to help programmers do a better job.

p2p.wrox.com  
The programmer's resource center

**www.wrox.com**



An Imprint of WILEY

图书上架  
分类建议

编程语言

PHP、设计模式

读者信箱：[wkservice@vip.163.com](mailto:wkservice@vip.163.com)  
投稿信箱：[bookservice@263.net](mailto:bookservice@263.net)

ISBN 978-7-302-22843-1



9 787302 228431 >

定价：36.00元