PHP 7 Programming Cookbook

PHP 7编程实战

通过讲解80多个实践案例，使你的PHP网页开发技巧更上一层楼

Nomad PHP开发者群World Wide Herd研究组首席指导
Cal Evans为本书作序

[美] Doug Bierer 著
苏宝龙 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

PHP 7 Programming Cookbook

PHP 7编程实战

[美] Doug Bierer 著
苏宝龙 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Web 网站的专门化和多元化发展趋势，要求网页编程语言必须满足编写出运行速度快、节省资源且具有较高安全性动态网页的需求。本书介绍的 PHP 7 网页编程语言就是其中的佼佼者。

本书由 13 章构成，详细介绍了 PHP 7 的新增功能和中高级 PHP 技术，包括安装和配置 PHP 7 开发环境的入门知识、PHP 中的函数式编程功能、PHP 面向对象编程功能的基础知识、使用命名空间和特性（trait）的方式、从数据库读取数据和向数据库中写入数据的方式、创建 HTML 表单元素的类的方式、过滤与验证数据的方式、实现 SOAP 和 REST 客户端与服务器的手段、PHP 最新的国际化网页功能、遵循 PSR-7 编程规范编写中间件的方式、使用 PHP 实现高级算法（如链表、冒泡排序程序、堆栈和二分查找程序）的方式、软件设计模式、当前互联网中常见的攻击手段和相应的防护手段、加密/解密技术，以及可帮助你编写出优质代码的最佳编程习惯和调试技巧。此外，附录还介绍了 PSR-7 编程规范。

软件架构师、技术主管、中高级软件开发者和爱好者均适合阅读本书。

版权贸易合同登记号　图字：01-2016-9184

图书在版编目（CIP）数据

PHP 7 编程实战 /（美）道格·比尔（Doug Bierer）著；苏宝龙译. —北京：电子工业出版社，2017.10
书名原文：PHP 7 Programming Cookbook
ISBN 978-7-121-32772-8
Ⅰ. ①P… Ⅱ. ①道… ②苏… Ⅲ. ①PHP 语言—程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字（2017）第 236975 号

责任编辑：张春雨
印　　刷：
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编　100036
开　　本：787×980　1/16　　印张：37.5　　字数：640 千字
版　　次：2017 年 10 月第 1 版
印　　次：2017 年 10 月第 1 次印刷
定　　价：119.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

译者序

互联网（internet，又称为因特网）始于 20 世纪 60 年代末的美国，最初仅用于军事和科研。1989 年 3 月 12 日万维网（World Wide Web，无数个 Web 服务器和客户端的集合）在 internet 中诞生。大批富豪风起云涌般随之崛起，如 Amazon 的创始人杰夫·贝佐斯、Yahoo 的创始人杨致远、百度的创始人李彦宏以及阿里巴巴的创始人马云等。人们对计算机网络的运用有了翻天覆地的变化。了解新闻不再需要买报纸、看电视，上网（即浏览网站，也就是访问 Web 服务器）就可以；了解百科知识（如查字典上的解释、医药信息、电子信息、机械知识甚至出行路线），上网就可以；买东西，上网就可以。上网还可以娱乐，如看电视剧、看电影、玩电子游戏。简言之，万维网已经成为人类生活必不可少的组成部分。

万维网诞生之初，Web 网站的内容非常简单：文字和图片。因此对网页编程语言的要求也非常简单——能够美观地为浏览网站的用户显示出文字和图片即可。简单的 HTML 语言足以胜任这项工作。但随着万维网的不断发展，网站的功能也变得越来越专门化（例如专门提供娱乐的视频、游戏网站，以及专门提供网页搜索服务的网站）和多元化（例如购物网站、聚集各种爱好者的网站），各种网站的内容和功能也不断丰富。因而对网页编程语言的要求也越来越高。

当前的 Web 网站要求网页编程语言必须能够编写出运行速度快、节省资源且具有较高安全性的动态网页。PHP 符合所有这些要求，下面是它的优点：

- PHP 属于开源软件，源代码完全公开。
  - 开源软件是免费的，非常省钱。
  - 任何程序员都能非常容易地为 PHP 扩展附加功能。
  - 可跨平台，能够和很多免费的平台结合。
- PHP 语法简单，非常容易上手，可快速实现从设计一个网页到编写一个 Web 应用程序的飞越。使用 PHP，刚刚成为软件工程师甚至还不是软件工程师的人都能够提交新功能。PHP 是为了快速制造新东西而生的（这是维基百科和

Facebook 选择 PHP 的原因)。

- PHP 将程序嵌入到 HTML 文档中去执行，执行效率比完全生成 HTML 标记的 CGI 程序高很多。PHP 可以执行编译后代码，编译可以达到加密和优化代码的目的，使代码运行得更快。
- PHP 支持十多种主流与非主流数据库，如：dBase、Informix、mSQL、MySQL、Microsoft SQL Server、Sybase、ODBC、PostgreSQL、Oracle 等。其中，PHP 与 MySQL 是绝佳的组合，可以跨平台运行。
- 目前主流技术（如 WebService、AJAX、XML 等）都支持 PHP，因此能够满足应用需求。
- 很多大型门户网站都使用 PHP，如淘宝网、网易、新浪等。
- PHP 提供了类和对象，实现了面向对象编程功能。
- 有成熟的开发者社区支持 PHP 的研究和发展。

本书的作者 Doug Bierer 先生是一位拥有三十多年经验的资深软件开发者，他在本书中使用了大量的实践案例。这些范例程序既有实用性也有启迪性，深入浅出地讲解了 PHP 7 的新增功能和中高级 PHP 技术。

翻译前沿计算机科学书籍的工作并不轻松，也不是单独一个人能够完成的。在此我要感谢电子工业出版社张春雨等编辑对本书提供的帮助。此外，苏连印、刘桂英、艾玉林、孙召景、张纪悦、张纪华、孙德林、马佳妮、尹晓婷、徐雯、郭昕、陆迎明和孙召恒等也参与了本书的翻译工作，在此对他们表示感谢。

因时间仓促，译者水平有限，本书难免有错漏之处，欢迎广大读者朋友们批评指正。

苏宝龙

推荐序

PHP 7 为我们带来了大量的新功能和改进，如抽象语法树（Abstract Syntax Tree，AST）、可被捕捉的错误、标量类型提示功能、返回值数据类型声明以及大幅度的性能提升等。

PHP 开发者当前面临的问题不是“我应该使用这些新功能吗”，而是“我怎样才能使用这些新功能开发出品质更好、速度更快的应用程序呢”。

对于使用 PHP 4 开发应用程序的情景，我记忆犹新。对于 PHP 开发者来说，那是一段更为纯真的时光，因为可以将 PHP 代码和 HTML 代码混合在一起，而且能够将这些代码都放在一个文件中。那时候我们使用导入的函数库，而不使用框架。PHP 应用程序基本上只是具有简单基本功能的桌面应用程序，而我们则想方设法把它塞到网页中。

从那以后，PHP 应用程序开发经历了多次变革。AJAX、PHPUnit、composer 和 API-First 等新兴框架相继涌现出来。

所有框架（包括上面提到的和其他许多框架）都对 PHP 开发者们开发应用程序的方式产生了影响。现在，如果你使用将 PHP 和 HTML 代码混合到一起的方式来编写以分页式设计模式为基础的应用程序，就会受到嘲笑。那么应该怎样开发程序呢？怎样开发新式 PHP 应用程序和 API 呢？怎样利用 PHP 新增的工具开发出速度更快、品质更优、健壮性更高的应用程序呢？我非常赞赏你能够想到并提出这些问题。

我的好朋友 Doug Bierer 对这些问题做出了回答。他撰写的这本书不是华而不实的大部头著作，不是你买来放在书架上就不会再碰一下的书。你捧在手中的这本书注定会成为你的参考资料库的一部分。

本书不仅像其他编程书籍一样介绍新增功能的理论知识，而且还会介绍使用这些新增功能解决真实问题的实践方法。你不仅可以从本书汲取知识，而且还能够立刻使用学到的知识解决问题。

如果只是简单使用一下 PHP 中的高级概念，你无须成为 PHP 专家。然而，如果你想要成长为一名专业的 PHP 开发者，就必须学习、理解和掌握这些高级概念。本书会帮助你在自己的专业道路上不断前进。

Cal Evans
Nomad PHP 开发者群 World Wide Herd 研究组首席指导

关于作者

在 1971 年，**Doug Bierer** 使用 Dartmouth BASIC（BASIC 语言的原始版本）在一台 DEC PDP-8 型计算机上编写出了自己的第一个程序，从那以后，他就再也无法与计算机分离了。经过一段体验非常丰富的职业生涯历练后，他在 1978 年成为了专业的合约程序员，此后他一直在使用 BASIC、PL/I、汇编、FORTH、C、C++、dBase/FoxBase/Clipper、Pascal、Perl、Java 和 PHP 语言编写应用程序。Doug Bierer 会说四种语言，喜欢在世界各地旅游，在法国、荷兰、英格兰、瑞典、苏格兰和泰国都居住过一段时间。他曾经花数年时间从事 Linux 系统管理和 TCP/IP 网络工作。他多才多艺，喜欢音乐和小说，写过 60 多首歌。他有一个笔名叫作 Douglas Alan。

Doug Bierer 拥有自己的公司（unlikelysource.com），该公司的主营业务包括咨询、PHP 程序开发、网站架设和培训（主要面向 Zend Technologies 和 Rogue Wave Software 公司的员工）。

Doug Bierer 在 `https://www.lulu.com/` 上发表过小说 *The End, And Then? and Further Indications*。他在 O'Reilly Media 出版社出版的计算机科学著作包括 *Learning PHP and MySQL*、*Learning PHP Security*、*Learning MongoDB* 和 *Learning Doctrine*。

首要的是，我要将这本书献给我的母亲 Betty Bierer，她在 2016 年 5 月永远离开了我们。她在我的整个生命历程中不断鼓励我，为我的每次进步欢呼鼓掌（不论这些进步多么微小）。她参加了我举办的所有音乐会、购买了我录制的所有 CD，而且即便她对计算机科学毫无兴趣，也仍旧阅读了我撰写的所有书。我还要感谢能够与我同甘共苦的妻子 Siri，当我因撰写本书而没有时间陪伴她时，她也无怨无悔（不过她也对我声明下不为例）。最后，我要感谢许多 PHP 社区的活跃人士，他们帮助我迸发出新式思路和灵感。这些人包括：Matthew Weir O'Phinney、Cal Evans、Daryl Wood、Susie Pollock、Salvatore Pappalardo、Slavey Karadzhov 和 Clark Everetts。

关于审校者

Salvatore Pappalardo 是一位技术天才，在 2002 年成为了一名软件工程师。他喜欢“从零开始”的开发形式。他爱好广泛，除了喜欢研究计算机技术，还喜欢阅读科幻小说，观赏电影和 TED 演讲。

Vincenzo Provenza 是一位网页开发者，具有使用多种技术和编程语言（主要是 PHP 和 JavaScript）的从业经验。他喜欢旅游和读书。

前言

PHP 7 以破纪录的速度席卷了开源社区，吸引了许多人的注意力。核心开发团队对这门编程语言做出了很大的改进，但仍然为该语言保留了高度的向下兼容性。这些改进几乎能够为 PHP 程序提高 200%的速度，并能够大幅度地减少内存占用量。从开发的观点看，对命令和统一变量语法（uniform variable syntax）解析方式的改进引入了多种编写代码的新方式，这些方式在以前的 PHP 版本中是无法使用的。同时，如果开发者不了解 PHP 7 解析命令的方式，就会掉到隐藏的陷阱中，导致程序发生故障。因此，本书的使命是介绍这些新的令人激动的代码编写方式，以及这些新方式与以前的 PHP 版本不兼容的地方。需要着重指出的一点是，本书既介绍 PHP 7.0 也介绍 PHP 7.1。

本书主要内容

第 1 章介绍入门知识，帮助你安装和配置 PHP 7 的开发环境。还介绍了几个代表性很强的示例程序，通过它们来展示 PHP 7 的几个新功能。

第 2 章深入介绍了这门语言的新增功能，其中包括抽象语法树（Abstract Syntax Tree）和统一变量等语法，还介绍了这些新增功能是怎样对常规编程方式产生影响的。本章还通过几个示例介绍了 PHP 7 在性能方面的提升，包括在 `foreach()` 循环中的大幅度改进。

第 3 章着重介绍 PHP 一直拥有的使用程序员定义的函数库（而不是类库）的功能，当然 PHP 7 也不会例外。本章会详细介绍对函数处理方式的改进，其中包括类型提示（type hint）功能，而且可以将该功能应用于基础数据类型（如整型、浮点型、布尔型和字符型）。还介绍了 PHP 标准库（Standard PHP Library，SPL）中的许多迭代器，以及利用已改进的生成器自己编写迭代器的方式。

第 4 章介绍 PHP 面向对象编程功能的基础知识。快速掌握这些基础知识后，就可以学习使用 PHP 命名空间和特性（trait）的方式。本章还会介绍需要考虑的软件架构问题，例如怎样以最佳方式使用接口。最后会通过几个实践范例介绍 PHP 7 令人激动的新

增功能：匿名类。

第 5 章介绍从数据库读取数据和向数据库中写入数据的方式，这是现代网站的关键功能。许多人对 PDO（PHP Data Objects，PHP 数据对象）扩展的用法有误解，本章会详细介绍 PDO 扩展，使你不必学习额外命令集就能够编写出能与绝大多数数据库（如 MySQL、Oracle、PostgreSQL、IBM DB2 和 Microsoft SQL Server）进行交互的应用程序。此外，本章还会介绍一些高级技巧，如使用领域模型实体实现嵌入式二次查询，以及使用 PHP 7 程序实现 jQuery DataTable 插件查询操作。

第 6 章详细分析架设交互式网站的 PHP 开发者们所面对的一个典型问题——通过硬编码方式创建表单，之后又不得不对表单进行维护。本章介绍一种简捷、高效的面向对象处理方式，使用最少量的代码生成全部 HTML 表单，而且可以通过初始配置轻松修改这些表单。PHP 开发者会面对的另一个难度不相上下的问题是，怎样过滤和验证用户通过表单提交的数据。通过学习本章，你可以开发出易于配置的过滤器和验证器，它们可应用于任何收到的数据。

第 7 章介绍在网页开发中变得越来越重要的功能——提供和访问 Web 服务。本章会介绍两种重要的提供和访问 Web 服务的方式：SOAP 和 REST。通过阅读本章，你可以学会实现 SOAP 和 REST 客户端与服务器的手段。此外，本章会介绍几个使用适配器设计模式开发的示例程序，这些程序具有非常大的自定义空间，这意味着你不会被局限在特定的设计范式中。

第 8 章会帮助你应对互联网迅猛的增长势头，让客户能够通过你编写的国际化网页将业务扩展到国际化市场中。本章会介绍所有最新的国际化网页功能，其中包括表情符号的用法、字符绘画和将网页内容翻译成多种语言的手段。本章还会介绍获取和处理用户所在地信息的方式，这些信息包括浏览网页的用户的语言设置、数字和货币格式，以及日期和时间格式。本章会通过一些示例介绍创建国际化日历的方式，该日历可处理反复出现的事件。

第 9 章介绍当前在开源社区中最火爆的话题——中间件。顾名思义，中间件是一种能够在不更改应用程序源代码的情况下，为已开发出的应用程序增加价值的“嵌入式”软件。本章会通过一系列示例来介绍在遵循 PSR-7 编程规范（附录详细介绍了 PSR-7 类）的前提下编写中间件的方式，这些中间件可以执行验证、访问控制、缓存和路由操作。

第 10 章介绍重要的高级算法。如果有许多程序员和公司竞争同一项业务，那么掌握这些高级算法对你来说就是至关重要的。本章会介绍如何使用 PHP 7 的理论知识和实用技巧来编写应用程序的读取器和设置器、链表、冒泡排序程序、堆栈和二分查找程序。本章还会介绍使用这些技巧实现搜索引擎和处理多维数组的方式。

第 11 章介绍面向对象程序设计理论的重要组成部分——软件设计模式。如果没有掌握这些知识，那么在尝试获取新职位或客户时，就会处于劣势地位。本章会介绍多种重要的软件设计模式，其中包括水合器（hydrator）、策略、映射器、对象关联映射和发布/订阅等设计模式。

第 12 章介绍当前互联网中常见的攻击手段。随着网络攻击事件越来越多，网络攻击造成的经济损失和泄露的私人信息也越来越多。本章会通过一些示例介绍实用技巧，使用这些技巧可以使网站的安全性和保险性成倍增加。本章还会介绍过滤和验证操作、为 PHP 会话提供安全防护的方式、提高表单提交操作安全性的方式、生成具有安全性的密码的技巧，以及使用验证码的手段。此外，本章会通过示例介绍在不使用 PHP 加密扩展库 mcrypt 的情况下（mcrypt 扩展已被 PHP 7.1 弃用，而且最终会被从该语言中移除），实现加密/解密功能的方式。

第 13 章介绍可帮助你编写出优质代码的最佳编程习惯和调试技巧。本章还会介绍创建和设置单元测试、处理预料之外的错误和异常，以及创建测试数据的方式。本章也介绍了 PHP 7 的几个新增功能，其中包括 PHP 7 解释程序抛出错误的功能和方式。请注意，本书通篇都会介绍最佳编程习惯，而不仅限于这一章！

附录介绍最近被普遍接受的第 7 号 PHP 推荐标准，即 PSR-7，也可称为 PSR-7 编程规范，该编程规范定义了与中间件联合使用的接口。本附录介绍实现 PSR-7 类的可靠方式，这些 PSR-7 类包括值对象（如 URI、流式请求和回应的主体，以及上传文件操作）和代表请求与回应的对象。

做书中实验所需的硬件条件

要做本书中介绍的示例实验，你的计算机需要拥有 100MB 以上的硬盘可用空间，并安装一个文本或代码编辑器（注意这不是指文字处理软件）。第 1 章介绍了设置 PHP 7

开发环境的方式。你可以自己搭建 Web 服务器，也可以不搭建 Web 服务器，因为 PHP 7 解释程序自带了一个在开发环境中使用的 Web 服务器。是否联网也不做要求，但是如果联网就可以下载代码（如 PSR-7 接口），以及阅读 PHP 7.x 的说明文档。

本书面向的读者

软件架构师、技术主管、中高级软件开发者和爱好者都适合阅读本书。在阅读本书之前，你需要先具备 PHP 程序设计的基础知识（OOP 方面的尤为重要）。

本书各章标题结构

本书会反复出现“准备工作”“具体处理过程”“具体运行情况”“补充说明”和“扩展”等标题。

这些标题都用于说明完成示例实验的步骤，下面是它们的具体含义。

准备工作

该标题下方是介绍示例实验的主要内容，以及该实验的初步设置和所有软件设置。

具体处理过程

该标题下方是做示例实验的具体步骤。

具体运行情况

该标题下方是前面具体步骤所介绍代码的详细运行情况。

补充说明

该标题下方是与前面介绍的示例程序有关的补充资料，以便使你进一步了解这些示例程序。

扩展

该标题下方是与前面介绍的示例程序有关的网站链接。

本书约定

本书使用多种字体区分多种类型的信息，下面详细说明这些字体和相应的信息类型。

代码、数据库表的名称、文件夹的名称、文件名、文件的扩展名、目录的名称、虚拟的 URL、用户输入的信息和 Twitter 微博的用户名将使用等宽字体，例如：“将步骤 3 介绍的 `LotsProps` 类添加到独立文件 `chap_10_oop_using_getters_and_setters_magic_call.php` 文件中。”

代码段部分使用等宽字体，例如：

```
protected static function loadFile($file)
{
    if (file_exists($file)) {
        require_once $file;
        return TRUE;
    }
    return FALSE;
}
```

当着重介绍一部分代码时，会使用粗体将其标出，例如：

```
$params = [
  'db' => __DIR__ . '/../data/db/php7cookbook.db.sqlite'
];
$dsn = sprintf('sqlite:' . $params['db']);
```

在命令行界面中输入和输出的信息使用下面的字体：

```
cd /path/to/recipes
php -S localhost:8080
```

新术语和**重要词汇**使用黑体（中文）或加粗（英文）表示。菜单和对话框中的英文界面词也会加粗，例如：“当 **Purchases**（购物信息）按钮被单击后，初始的购物信息就会显示出来”。

这部分内容介绍警告和需要注意的重要信息。

这部分内容介绍提示和小窍门。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 提交勘误：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- 交流互动：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32772

目录

第 1 章　基础知识

本章包括以下要点：

- 安装 PHP 7 的注意事项
- 内置 PHP Web 服务器的使用方法
- 定义用于测试的 MySQL 数据库
- 安装 PHPUnit
- 实现类自动加载
- 扫描网站
- 创建深层次的网页扫描器
- 创建将 PHP 5 代码转换为 PHP 7 代码的代码转换器

本章主要内容简介

本章介绍了 PHP 7 语言的快捷入门知识，掌握了这些内容，你就能够开始编写 PHP 7 程序了。在学习本章内容前，你需要先具备足够的 PHP 语言和程序设计基础知识。尽管本书不会事无巨细地介绍 PHP 的安装过程，但考虑到 PHP 7 还是一个比较新的版本，所以我们会尽最大努力详细介绍在安装 PHP 7 的过程中可能出现的难点和注意事项。

安装 PHP 7 的注意事项

一般可以通过 3 种方式获得 PHP 7：

- 下载源代码并直接通过源代码进行安装
- 安装已编译好的二进制文件
- 安装*AMP（如 XAMPP、WAMP、LAMP 和 MAMP 等）软件包

具体处理过程

上面的 3 种方式是按照由难至易的顺序介绍的。第一种方式最难操作，但这种方式会让你在增加和减少 PHP 7 组件时获得最大的自由度。

通过源代码直接安装

要使用这种安装方式，首先需要拥有一个 C 语言编译器。如果你使用的操作系统是 Windows，那么就可以使用经过考验并广受欢迎的免费编译器 **MinGW**。MinGW 是以通过 **GNU** 计划开发出的 **GNU Compiler Collection**（**GCC**）编译器集合为基础的。你也可以选择购买需付费的编译器（如 Borland 公司出品的 Turbo C），当然，Windows 环境中的开发者可能会更偏爱 **Visual Studio** 编译器。然而，Visual Studio 专门用于开发 C++程序，因此在编译 PHP 代码时，就需要将其设置为 C 语言模式。

如果你使用的是苹果公司的 Mac 机，那么最佳解决方案是安装 **Apple Developer Tools** 开发工具集。你可以使用 **Xcode 集成开发环境**（Integrated Development Environment，IDE）编译 PHP 7 代码，也可以通过终端窗口运行 `gcc`。如果你使用的是 Linux 操作系统，同样可以通过终端窗口运行 `gcc`。

当通过终端窗口或命令行界面编译源代码时，通常应执行下列处理步骤：

1. 配置编译器
2. 添加源代码
3. 进行测试
4. 进行安装

要详细了解各个配置选项（即在运行 `configure` 命令时会出现的选项），可使用 `help` 选项：

```
configure --help
```

下表列出了配置阶段可能出现的错误：

错误提示	解决方法
`configure: error: xml2-config not found. Please check your libxml2 installation`	只需安装 `libxml2` 函数库就可以解决这个问题。要详细了解这个错误，请浏览 `http://superuser.com/questions/740399/how-to-fix-phpinstallation-when-xml2-config-ismissing`

续表

错误提示	解决方法
configure: error: Please reinstall readline - I cannot find readline.h	安装 libreadline-dev 软件包可解决该问题
configure: WARNING: unrecognized options: --enable-spl, --enable reflection, --with-libxml	这个不算问题，无须理会该错误提示。这些选项是默认配置，而且无须专门对它们进行设置。要详细了解该错误提示，请浏览 http://jcutrer.com/howto/linux/howto-compile-php7-on-ubuntu-14-04

通过已编译好的二进制文件安装 PHP 7

此处的**已编译好**的二进制文件是指，由一些富有分享精神的开发者使用 PHP 7 的源代码编译出的一系列二进制文件。

如果你使用的是 Windows，可浏览 http://windows.php.net/。在该页面的左侧可以看到一些帮助进行版本选择的提示，如 **thread safe**、**non-read safe** 等。可以单击下载链接并找到与你的开发环境匹配的.zip 文件。下载了.zip 文件后，可以将这些文件解压到你选择的文件夹中，将 php.exe 文件添加到你的安装路径中，并使用 php.ini 文件配置 PHP 7。

要在 Mac OS X 系统中安装已编译好的二进制文件，最好使用软件包管理系统。在安装 PHP 时，我们推荐使用下列软件包管理系统：

- MacPorts
- Liip
- Fink
- Homebrew

如果你使用 Linux 操作系统，应根据你使用的 Linux 版本选择软件包管理系统。下表列出了各个 Linux 版本中安装 PHP 7 的不同位置。

Linux 版本	安装 PHP 7 的位置	注释
Debian	packages.debian.org/stable/php repos-source.zend.com/zendserver/early-access/php7/php-7*DEB*	可使用下面的命令： **sudo apt-get installphp7** 也可以使用图形界面的软件包管理工具，如 **Synaptic**。 应确保你选择了 **php7**（而没有错选 php5）

续表

Linux 版本	安装 PHP 7 的位置	注释
Ubuntu	packages.ubuntu.com repos-source.zend.com/zendserver/early-access/php7/php-7*DEB*	可使用下面的命令： sudo apt-get installphp7 应确保你选择了正确的 Ubuntu 版本。 也可以使用图形界面的软件包管理工具，如 **Synaptic**
Fedora / Red Hat	admin.fedoraproject.org/pkgdb/packages repos-source.zend.com/zendserver/early-access/php7/php-7*RHEL*	应确保你正在使用的是 root 用户： **su** 可使用下面的命令： **dnf install php7** 也可以使用图形界面的软件包管理工具，如 GNOME 中的 Package Manager
OpenSUSE	software.opensuse.org/package/php7	可使用下面的命令： **yast -i php7** 也可以使用 zypper 管理工具，或使用图形界面工具（如 **YaST**）

安装*AMP 软件包

AMP 是 **Apache**、**MySQL** 和 **PHP**（也包括 **Perl** 和 **Python**）的首字母缩写词。*AMP 中的*代表与 Linux、Windows 和 Mac 等操作系统对应的版本，即 LAMP、WAMP 和 MAMP。这种安装方式通常是最简单的，但在安装 PHP 时获得的自由度也是最小的。从另一方面看，你也可以随时根据需要通过修改 php.ini 文件增加安装内容。下表列出了一些广受欢迎的*AMP 软件包：

软件包	下载地址	是否免费	可在哪些操作系统中使用
XAMPP	www.apachefriends.org/download. html	是	Linux、Windows 和 Mac
AMPPS	www.ampps.com/downloads	是	Linux、Windows 和 Mac
MAMP	www.mamp.info/en	是	Windows 和 Mac
WampServer	sourceforge.net/projects/wampserver	是	Windows
EasyPHP	www.easyphp.org	是	Windows
Zend Server	www.zend.com/en/products/zend_server	否	Linux、Windows 和 Mac

上表介绍了一些*AMP 软件包，其中的*分别代表 **W**（即 Windows 的首字母）、**M**

（即 Mac 的首字母）和 **L**（即 Linux 的首字母）。[1]

补充说明

当你通过软件包安装已编译好的二进制文件时，仅能安装 PHP 的核心扩展。PHP 的非核心扩展必须单独安装。

值得注意的是，在云计算平台上安装 PHP 7 时，通常会根据已编译好的二进制文件确定大致的安装过程。因此，你应该先查明你所处云环境中的虚拟机运行的是 Linux 还是 Mac，又或者是 Windows，然后根据本章前面介绍知识，执行适当的安装步骤。

如果你无法获得 PHP 7 已编译好的二进制文件，可以随时通过 PHP 7 源代码安装，也可以选择某一个*AMP 软件包进行安装（请参阅下一节介绍的内容），还可以使用 Personal Package Archive（**个人软件包档案**，PPA）方式替代在 Linux 中使用的安装方式。因为 PPA 源代码没有经过严格的检验筛选过程，所以其安全性是一个大问题。你可以通过浏览 `http://askubuntu.com/questions/35629/are-ppas-safe-to-add-to-mysystem-and-what-are-some-red-flags-to-watch-out-fo`，详细了解使用 PPA 源代码的安全注意事项。

扩展

通过浏览 `http://php.net/manual/en/install.general.php`，可以了解综合的 PHP 安装注意事项，以及与 3 种主流操作系统平台（Windows、Mac 和 Linux）对应的操作指南。

通过浏览 `http://www.mingw.org/`，可以了解 MinGW。

通过浏览 `https://msdn.microsoft.com/en-us/library/bb384838`，可以获得使用 Visual Studio 编译 C 程序的操作指南。

另一种测试 PHP 7 的方式是使用虚拟机。下面列出了这类颇为有用的工具和获取这些工具的网址：

- **Vagrant**：`https://github.com/rlerdorf/php7dev`（该页面提供的 php7dev 文件是一个 Debian 8 版本的 Vagrant 镜像，该镜像已经被配置好，可以

1 因为 PHP 是运行在服务器端的语言，所以只安装 PHP 7 解释器还不够。只有安装好相应的 Web 服务器和数据库，才能使用 PHP 程序提供网页服务。——译者注

用于测试 PHP 应用和跨多个 PHP 版本开发扩展）。

- **Docker**：`https://hub.docker.com/r/coderstephen/php7/`（该页面提供了 PHP 7 Docker 容器）。

内置 PHP Web 服务器的使用方法

除了进行单元测试和通过命令行界面直接运行 PHP 外，使用 Web 服务器测试应用程序也是一种显而易见的方式。对于开发周期较长的项目来说，为了逼真地模拟真实的运行环境，配置在虚拟主机上运行的 Web 服务器是有好处的。为各种 Web 服务器（如 Apache、Nginx 等）配置虚拟主机的内容已经超出了本书涵盖的范围。另一种简单易用且快捷的方式是使用内置的 PHP Web 服务器，本节将详细介绍这种方式。

具体处理过程

1. 要启动 PHP Web 服务器，需要先切换到存储 PHP 源代码的根目录。

2. 然后，必须设置主机名称或 IP 地址，但端口号的设置不是必要的。使用下面的示例可以运行本书介绍的 PHP Web 服务器：

```
cd /path/to/recipes
php -S localhost:8080
```

屏幕上会显示下图所示的内容：

3. 当这个内置的 Web 服务器不断对服务请求做出回应时，可以查看访问它的信息、HTTP 状态码和请求信息。

4. 如果你需要将该 Web 服务器的文件根目录设置为另一个目录，可使用参数`-t`。该参数的后面必须紧跟一个合法的目录路径。这个内置的 Web 服务器会将该目录视为该 Web 服务器的文件根目录，这样做有助于提高安全性。为了获得更高的安全性，某些框架（如 Zend Framework）要求将 Web 服务器的文件根目录设置为不同于真正存储网站源代码的目录。

下面是一个使用`-t`参数的例子：

```
php -S localhost:8080 -t source/chapter01
```

下图是这个示例的输出结果：

定义用于测试的 MySQL 数据库

为了帮助你测试本书介绍的源代码，我们在`https://github.com/dbierer/php7cookbook`提供了 SQL 文件和示例数据。本书示例使用的数据库名称为`php7cookbook`。

具体处理过程

1. 定义一个 MySQL 数据库并将之命名为 php7cookbook。新建一个数据库管理员账号,将该用户账号的名称设置为cook并将登录密码设置为book。下表列出了这些设置：

需设置的各项配置	设置值
数据库名称	php7cookbook
数据库管理员账号	cook
数据库管理员账号的登录密码	book

2. 下面是用于创建数据库的 SQL 语句示例：

```
CREATE DATABASE IF NOT EXISTS dbname DEFAULT CHARACTER SET utf8
COLLATE utf8_general_ci;
CREATE USER 'user'@'%' IDENTIFIED WITH mysql_native_password;
SET PASSWORD FOR 'user'@'%' = PASSWORD('userPassword');
GRANT ALL PRIVILEGES ON dbname.* to 'user'@'%';
GRANT ALL PRIVILEGES ON dbname.* to 'user'@'localhost';
FLUSH PRIVILEGES;
```

3. 将示例数据导入到新建的数据库中。这些示例数据包含在 php7cookbook.sql 文件中，在 https://github.com/dbierer/php7cookbook/blob/master/php7cookbook.sql 可以下载该文件。

安装 PHPUnit

单元测试可能是最流行的测试 PHP 代码的方式。大多数开发者都赞同这一观点：以适当方式开发出的项目都必须经过一系列严格的测试。但很少有开发者会实实在在地编写这些测试内容。只有极少数幸运的开发者拥有专门为他们编写测试程序的团队！然而，经过数月的与测试团队的摩擦冲突后，这些开发者中的幸运儿们也会牢骚满腹。无论如何，任何介绍 PHP 的书籍都不会对测试只字不提。

在 https://phpunit.de/上可以获得 PHPUnit 的最新版本。PHPUnit 5.1 及以上版本都支持 PHP 7。单击你想要下载的版本的链接后，就能够下载一个 phpunit.phar 文件，然后就可以使用这个文件执行命令：

```
php phpunit.phar <命令>
```

phar 命令是 PHP archive 的缩写词。这项技术源于 UNIX 中的 tar 命令。一个 phar 文件是一个含有多个 PHP 文件的集合，为了便于使用，这些 PHP 文件被封装到了一个文件中。

实现类自动加载

当使用面向对象的编程方式（OOP）开发 PHP 软件时，我们建议将每个类放在它本身的文件中。这样做的好处是有利于长期维护并能够提高可读性。这样做的坏处是必须将每个类的定义文件都包括到主程序文件中（即在主程序文件中添加 include 语句或该语句的各种变体）。为了解决这个问题，开发者们为 PHP 语言添加了一种机制，通过该机制可以在不专门使用包含命令的情况下实现自动加载任何类。

准备工作

实现 PHP 类自动加载功能的最低要求是定义一个全局的__autoload()函数。这个函数拥有神奇的功能，当需要加载某个类而该类没有被包含到主程序中时，该函数就会被 PHP 引擎[1]自动调用。当__autoload()函数被调用时，需要加载的类的名称会被用作调用该函数的参数（就像你手动定义了它们一样）。如果你使用了 PHP 中的命名空间，那么这个类的完整的命名空间名称都会被作为参数传递。因为__autoload()是一个函数，所以它必须处于全局命名空间之中，因而，该函数的使用会有一些局限性。因此，在下面的示例中我们会使用 spl_autoload_register()函数，它会为我们提供更多的灵活性。

具体处理过程

1. 本例介绍的类是 Application\Autoload\Loader。为了同时利用 PHP 命名空间和类自动加载功能的优点，我们创建了 Loader.php 文件并将其放置在/path/to/cookbook/files/Application/Autoload 文件夹中。

1 本书介绍的 PHP 引擎是指 PHP 处理程序（解释器），如 PHP 7、Zend、HHVM 和 mod_php 模块等。——译者注

2. 下面先介绍仅加载一个文件的方式。在调用 `require_once()` 函数前，我们先使用 `file_exists()` 函数进行检查。这样做是因为如果要加载的文件不存在，`require_once()` 函数会生成一个致命错误，而 PHP 7 新增的错误处理功能无法检测到这类错误：

```
protected static function loadFile($file)
{
    if (file_exists($file)) {
        require_once $file;
        return TRUE;
    }
    return FALSE;
}
```

3. 我们可以在执行调用操作的程序中测试函数 `loadFile()` 返回的值，如果在检查完一组备选目录后没有找到可加载文件，则抛出一个异常。

你可能已经注意到这个类中的方法和属性都是静态的。这会使我们在注册执行自动加载操作的方法时获得更多自由，因此让我们像处理**单例对象**（Singleton）一样处理 `Loader` 类。

4. 下面定义调用 `loadFile()` 函数的方法，并为了执行根据类的完整命名空间名称查找文件的操作，定义真正执行该操作的逻辑。该方法会通过将 PHP 命名空间的分隔符\转换为与当前所处服务器对应的目录分隔符，并加上 `.php` 后缀，获取文件的名称：

```
public static function autoLoad($class)
{
    $success = FALSE;
    $fn = str_replace('\\', DIRECTORY_SEPARATOR, $class)
          .'.php';
    foreach (self::$dirs as $start) {
        $file = $start . DIRECTORY_SEPARATOR . $fn;
        if (self::loadFile($file)) {
            $success = TRUE;
            break;
        }
    }
    if (!$success) {
        if (!self::loadFile(__DIR__
```

```
                . DIRECTORY_SEPARATOR . $fn)) {
                throw new \Exception(
                    self::UNABLE_TO_LOAD . ' ' . $class);
            }
        }
        return $success;
    }
```

5. 该方法会使用 `self::$dirs` 语句检查一组目录，将每个目录用作推导文件名称的起点。如果没有找到目标文件，该方法最终会尝试从当前目录加载目标文件。如果该操作还是没有成功，该方法就会抛出一个异常。

6. 下面我们要创建一个能够向测试目录列表中添加更多目录的方法。注意，如果该方法使用数组提供值，那么就会用到 `array_merge()` 函数。否则，只需将目录字符串添加到 `self::$dirs` 数组中：

```
    public static function addDirs($dirs)
    {
        if (is_array($dirs)) {
            self::$dirs = array_merge(self::$dirs, $dirs);
        } else {
            self::$dirs[] = $dirs;
        }
    }
```

7. 本步骤是最重要的一步：要把我们编写的 `autoload()` 方法注册为 PHP 标准库（Standard PHP Library，SPL）自动加载器。通过 `init()` 方法使用 `spl_autoload_register()` 函数可以做到这一点：

```
    public static function init($dirs = array())
    {
        if ($dirs) {
            self::addDirs($dirs);
        }
        if (self::$registered == 0) {
            spl_autoload_register(__CLASS__ . '::autoload');
            self::$registered++;
        }
    }
```

8. 现在我们可以定义 `__construct()` 函数，该函数会调用 `init()` 方法，即 `self::init($dirs)`。这使我们能够根据自己的意愿创建一个 `Loader` 类的实例：

```
    public function __construct($dirs = array())
```

```
{
    self::init($dirs);
}
```

具体运行情况

为了使用我们刚刚编写的类自动加载器，需要执行 `require Loader.php` 命令。如果你的命名空间文件没有放在当前的目录中，还需要执行 `Loader::init()` 命令，并需要提供额外的目录路径。

为了确保这个自动加载器工作正常，还需要使用一个类进行测试。这个测试类是 `/path/to/cookbook/files/Application/Test/TestClass.php`，下面是它的定义：

```
<?php
namespace Application\Test;
class TestClass
{
    public function getTest()
    {
        return __METHOD__;
    }
}
```

下面创建示例代码文件 `chap_01_autoload_test.php`，以便测试这个自动加载器：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
```

然后，获取一个还没有被加载的类的实例：

```
$test = new Application\Test\TestClass();
echo $test->getTest();
```

最后，尝试获取一个不存在的类。注意，这样做会使 PHP 抛出一个错误：

```
$fake = new Application\Test\FakeClass();
echo $fake->getTest();
```

扫描网站

通常，扫描网站并通过特定的标签提取信息是一件有趣的事情。使用这项基本功能

可以在页面中进行搜索，以便获取有用的信息。有时需要获取一系列<IMG>标签和 SRC 属性（或者<A>标签和 HREF 属性），有时需要获取的信息五花八门，一切皆有可能。

具体处理过程

1. 获取目标网站的内容。乍看之下，我们好像需要发送 cURL 请求，或者直接使用 file_get_contents() 函数。使用这两种方式都有一个问题，那就是归根结底会迫使我们进行大量的字符串操作，这非常像滥用令人恐惧的正则表达式。为了从根本上避免该问题，我们只需利用已经在 PHP 7 中出现的 DOMDocument 类。因此，可以创建一个 DOMDocument 实例，并将它设置为 UTF-8 格式。我们无须考虑空格，使用便利的 loadHTMLFile() 方法就可以将网站的内容加载到该对象中：

```
public function getContent($url)
{
    if (!$this->content) {
        if (stripos($url, 'http') !== 0) {
            $url = 'http://' . $url;
        }
        $this->content = new DOMDocument('1.0', 'utf-8');
        $this->content->preserveWhiteSpace = FALSE;
        // @符号用于过滤掉配置错误的网页所生成的警告
        @$this->content->loadHTMLFile($url);
    }
    return $this->content;
}
```

注意，我们在调用 loadHTMLFile() 方法的代码前面添加了一个@符号。这样做是为了避免像 PHP 5 那样经常提取难以理解的错误代码（即错误警告的内容）。更确切地说，当解析器遇到编写错误的 HTML 代码时，@符号会过滤掉解析器生成的报错警告。如果我们提取这些错误警告并将它们记录下来，那么也可以为我们编写的这个 Hoover 类添加诊断功能。

2. 提取感兴趣的标签。可以使用 getElementsByTagName() 方法实现这一点。如果你想要提取所有标签，可以将*用作该方法的参数：

```
public function getTags($url, $tag)
{
```

```
        $count = 0;
        $result = array();
        $elements = $this->getContent($url)
                        ->getElementsByTagName($tag);
        foreach ($elements as $node) {
            $result[$count]['value'] = trim(
               preg_replace('/\s+/', ' ', $node->nodeValue));
            if ($node->hasAttributes()) {
               foreach ($node->attributes as $name => $attr)
               {
                  $result[$count]['attributes'][$name] =
                     $attr->value;
               }
            }
            $count++;
        }
        return $result;
    }
```

3. 提取指定的属性（而不是标签）也会很有趣。因此，我们定义了另一个方法以便实现该目的。在本例中，我们需要解析所有标签，并使用 `getAttribute()` 方法。你会看到我们需要用一个参数代表 DNS 域名，添加该参数是为了在同一个域中（例如在扫描使用树形结构的网站时）进行扫描：

```
    public function getAttribute($url, $attr, $domain = NULL)
    {
        $result = array();
        $elements = $this->getContent($url)
                    ->getElementsByTagName('*');
        foreach ($elements as $node) {
            if ($node->hasAttribute($attr)) {
                $value = $node->getAttribute($attr);
                if ($domain) {
                    if (stripos($value, $domain) !== FALSE) {
                        $result[] = trim($value);
                    }
                } else {
                    $result[] = trim($value);
                }
            }
        }
        return $result;
    }
```

具体运行情况

为了使用新编写的 Hoover 类，需要初始化自动加载器（请参阅前面介绍的内容）并创建一个 Hoover 对象。这样就可以通过运行 Hoover::getTags()方法从使用参数设定的 URL 中获取标签，并将这些标签添加到数组中。

下面是 chap_01_vacuuming_website.php 文件中的一段代码，这段代码使用 Hoover 类扫描了 O'Reilly 网站中的<A>标签：

```
<?php
// 你可以根据自己的实际情况修改这部分内容
define('DEFAULT_URL', 'http://oreilly.com/');
define('DEFAULT_TAG', 'a');

require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/. .');

// 获取用于执行扫描操作的类
$vac = new Application\Web\Hoover();

// 注意：下面使用了 PHP 7 中的 null 合并运算符(??)
$url = strip_tags($_GET['url'] ?? DEFAULT_URL);
$tag = strip_tags($_GET['tag'] ?? DEFAULT_TAG);

echo 'Dump of Tags: ' . PHP_EOL;
var_dump($vac->getTags($url, $tag));
```

下面是这段代码的输出结果：

扩展

要详细了解 DOM 类,请浏览 http://php.net/manual/en/class.domdocument.php。

创建深层次的网页扫描器

有时我们需要更深入地扫描网站，例如，需要为网站绘制网页的树形结构图时。通过查找所有的<A>标签和指向下一个页面的 HREF 属性，可以做到这一点。一旦获得了子页面的内容后，就可以按照树形结构继续执行扫描操作。

具体处理过程

1. 深层次网页扫描器的核心组件之一是前面介绍过的基本的 Hoover 类。本节介绍的基础处理过程包括扫描目标网站，并获取其中所有的 HREF 属性。为了做到这一点，我们定义了一个 Application\Web\Deep 类。下面添加一个代表 DNS 域名的属性：

```
namespace Application\Web;
class Deep
{
    protected $domain;
```

2. 定义一个方法，以便获取扫描列表中所有网站的标签。为了防止这个扫描器扫描整个万维网，应将其限定为仅扫描目标域。添加 yield from 语句的目的是使用 Hoover::getTags()方法添加数组的全部内容。通过 yield from 语法，我们可以将这个数组用作子生成器：

```
public function scan($url, $tag)
{
    $vac = new Hoover();
    $scan = $vac->getAttribute($url, 'href',
        $this->getDomain($url));
    $result = array();
    foreach ($scan as $subSite) {
        yield from $vac->getTags($subSite, $tag);
    }
    return count($scan);
}
```

使用 yield from 语句可以使 scan() 方法变成 PHP 7 中的标准生成器。通常，我们需要将执行扫描操作获得的结果存储到数组中。这种方法的问题是获得的信息可能是海量的。因此，为了节省内存和获得实时响应，最好以实时方式生成结果。否则，就可能需要长时间地等待，还会出现内存溢出错误。

3. 为了仅搜索指定的域，需要创建一个方法，以便通过 URL 获取域名。使用便捷的 parse_url() 函数可以做到这一点：

```
public function getDomain($url)
{
    if (!$this->domain) {
        $this->domain = parse_url($url, PHP_URL_HOST);
    }
    return $this->domain;
}
```

具体运行情况

首先，应定义前面介绍过的 Application\Web\Deep 类和 Application\Web\Hoover 类。

其次，编写一段用于实现自动加载功能的代码，请参阅 chap_01_deep_scan_website.php 文件和前面介绍的内容：

```
<?php
// 你可以根据自己的实际情况修改这部分内容
define('DEFAULT_URL', unlikelysource.com');
define('DEFAULT_TAG', 'img');

require __DIR__ . '/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/../..');
```

为我们新编写的这个类创建一个实例：

```
$deep = new Application\Web\Deep();
```

这样就可以通过设置 URL 的参数，检索 URL 和标签信息。使用 PHP 7 中的空合并运算符（??）有助于创建返回值：

```
$url = strip_tags($_GET['url'] ?? DEFAULT_URL);
$tag = strip_tags($_GET['tag'] ?? DEFAULT_TAG);
```

一些简单的 HTML 代码可以显示这类结果：

```
foreach ($deep->scan($url, $tag) as $item) {
    $src = $item['attributes']['src'] ?? NULL;
    if ($src && (stripos($src, 'png') || stripos($src, 'jpg'))) {
        printf('<br><img src="%s"/>', $src);
    }
}
```

扩展

要详细了解生成器和 `yield from` 语句，请浏览 `http://php.net/manual/en/language.generators.syntax.php`。

创建将 PHP 5 代码转换为 PHP 7 代码的代码转换器

大多数情况中，PHP 5.x 代码可以不经修改地在 PHP 7 引擎上运行。然而，PHP 7 中的几个功能不能向下兼容。换言之，如果你使用特殊的方式编写 PHP 5 程序或者在其中使用了被废弃掉的函数，那么在 PHP 7 引擎上运行该程序时就会出现严重错误并无法继续运行。

准备工作

将 PHP 5 代码转换成 PHP 7 代码的代码转换器会做两件事：

- 扫描你编写的代码文件并将已被废弃的 PHP 5 功能转换为 PHP 7 中的等价功能。
- 在已被更改语法但无须修改旧代码的位置，添加以 `//WARNING` 开头的注释。

请注意，运行过转换器后，也无法保证你编写的旧代码一定能在 PHP 7 引擎上运行。你仍然需要仔细检查转换器添加的以 `//WARNING` 开头的注释。不过本节介绍的内容至少能为你将 PHP 5 代码转换为 PHP 7 代码的工作指明正确的方向。

本节的要点是介绍 PHP 7 新增的 `preg_replace_callback_array()` 函数。这个函数的强大之处在于，能够将用正则表达式数组代表的键与独立回调函数代表的值对

应起来。这样就可以通过传递字符串进行一系列转换。而且回调函数数组中的单元本身也可以是数组。

具体处理过程

1. 在新建的 Application\Parse\Convert 类中，先创建一个 scan() 方法，它会接收文件名，并将它用作参数。该方法用于检查文件是否存在。如果该文件确实存在，那么 scan() 方法就会调用 PHP 中的 file() 函数，该函数会将这个文件加载到一个数组中，每个数组元素用于存储一行内容：

```
public function scan($filename)
{
    if (!file_exists($filename)) {
        throw new Exception(
          self::EXCEPTION_FILE_NOT_EXISTS);
    }
    $contents = file($filename);
    echo 'Processing: ' . $filename . PHP_EOL;

    $result = preg_replace_callback_array( [
```

2. 下面传递一系列键/值对。这些键是由正则表达式代表的，是对照字符串处理的。所有与标准语法字符串匹配的正则表达式都会被传递给回调函数，这些回调函数代表键/值对中的值。我们要检查的是已经在 PHP 7 中被废弃的开始标签和结束标签：

```
// 替换已经被废弃的开始标签
'!^\<\%(\n| )!' =>
    function ($match) {
        return '<?php' . $match[1];
    },

//替换已经被废弃的开始标签
'!^\<\%=(\n| )!' =>
    function ($match) {
        return '<?php echo ' . $match[1];
    },

//替换已经被废弃的结束标签
'!\%\>!' =>
    function ($match) {
```

```
        return '?>';
    },
```

3. 下面是当检测到特殊的操作时，PHP 7 会显示的一些警告，以及 PHP 7 与 PHP 5 之间可能会导致程序运行中断的差异。在所有这些情况中，代码不会被更改，但会被插入以//WARNING 开头的警告注释：

```
// 对$$xxx 变量替换方式的更改（其中 xxx 为含有变量名的字符串）
'!(.*?)\$\$!' =>
    function ($match) {
       return '// WARNING: variable interpolation
           . ' now occurs left-to-right' . PHP_EOL
           . '// see: http://php.net/manual/en/'
           . '// migration70.incompatible.php'
            . $match[0];
    },
```

释义

//警告：现在变量替换的操作按从左到右的次序执行（PHP_EOL 是 PHP 7 中的换行符），要了解更多信息，请浏览 http://php.net/manual/ en/ migration。

```
// 对 list()运算符处理方式的更改
'!(.*?)list(\s*?)?\(!' =>
    function ($match) {
        return '// WARNING: changes have been made '
             . 'in list() operator handling.'
             . 'See: http://php.net/manual/en/'
             . 'migration70.incompatible.php'
             . $match[0];
    },
```

释义

//警告：处理 list()运算符的方式已经改变，要了解更多信息，请浏览 http://php. et/nmanual/en/migration70.incompatible.php。

```
// u 类的实例
'!(.*?)\\\u\{!' =>
    function ($match) {
    return '// WARNING: \\u{xxx} is now considered '
        . 'unicode escape syntax' . PHP_EOL
        . '// see: http://php.net/manual/en/'
        . 'migration70.new-features.php'
        . '#migration70.new-features.unicode-'
        . 'codepoint-escape-syntax' . PHP_EOL
        . $match[0];
},
```

释义

//警告：现在 u{xxx}（其中 xxx 代表代码）已经被视为 unicode escape 语法，要了解更多信息，请浏览 http://php.net/manual/en/migration70.new-features.php。

```
// 取决于你自己对 set_error_handler()函数的设置
'!(.*?)set_error_handler(\s*?).*\(!' =>
```

```
    function ($match) {
        return '// WARNING: might not '
            . 'catch all errors'
            . '// see: http://php.net/manual/en/'
            . '// language.errors.php7.php'
             . $match[0];
    },

// session_set_save_handler(xxx)函数
'!(.*?)session_set_save_handler(\s*?)?\((.*?)\)!' =>
    function ($match) {
        if (isset($match[3])) {
            return '// WARNING: a bug introduced in'
                . 'PHP 5.4 which '
                . 'affects the handler assigned by '
                . 'session_set_save_handler() and '
                . 'where ignore_user_abort() is TRUE
                . 'has been fixed in PHP 7.'
                . 'This could potentially break '
                . 'your code under '
                . 'certain circumstances.' . PHP_EOL
                . 'See: http://php.net/manual/en/'
                . 'migration70.incompatible.php'
                . $match[0];
        } else {
            return $match[0];
        }
},
```

释义

//警告：该函数可能无法检测出所有错误，要了解更多信息，请浏览 http://php.net/ manual/en/language.errors.php7.php。

释义

//警告：PHP 5.4 中有一个 bug，该 bug 在 ignore_user_abort()函数的返回值为 TRUE 的情况下，会影响由 session_set_save_handler()函数分配的处理程序。在 PHP 7 中该 bug 已经被修复。在某些特殊环境中，你编写的代码有可能会因此而中断运行。要了解更多信息，请浏览 http://php.net/manual/en/migration70.incompatible.php。

4. 将负号运算符与<<或>>一起使用时，或者使用超过 64 位的数据时，这些代码都会被封装在 try{xxx}catch(){xxx}代码块中，以便找到被抛出的 ArithmeticError 异常：

```
// 将位移操作封装在try { xxx } catch() { xxx }代码块中
'!^(.*?)(\d+\s*(\<\<|\>\>)\s*-?\d+)(.*?)$!' =>
    function ($match) {
        return '// WARNING: negative and '
            . 'out-of-range bitwise '
            . 'shift operations will now
            . 'throw an ArithmeticError' . PHP_EOL
            . 'See: http://php.net/manual/en/'
            . 'migration70.incompatible.php'
```

释义

//警告：现在负数和超出范围的位移操作都会抛出 ArithmeticError 异常。要了解更多信息，请浏览 http://php. net/manual/en/migration70.incompatible.php。

```
            . 'try {' . PHP_EOL
            . "\t" . $match[0] . PHP_EOL
            . '} catch (\\ArithmeticError $e) {'
            . "\t" . 'error_log("File:"
            . $e->getFile()
            . " Message:" . $e->getMessage());'
            . '}' . PHP_EOL;
    },
```

PHP 7 已经改变了处理错误的方式。在某些情况中，一些类似的错误会被划分到同一个异常类别中，而且这些异常能够被捕捉到！Error 和 Exception 类都实现了 Throwable 接口。如果你想要捕捉 Error 或 Exception 异常，可捕捉 Throwable 接口。

5. 该转换器重写所有 call_user_method*() 函数（这类函数已经被 PHP 7 废弃了）。该类函数会被 call_user_func*() 函数替代：

```
    //使用函数 call_user_func() 替换函数 call_user_method()
    '!call_user_method\((.*?),(.*?)(,.*?)\)(\b|;)!' =>
        function ($match) {
            $params = $match[3] ?? '';
            return '// WARNING: call_user_method() has '
                        . 'been removed from PHP 7'
                        . PHP_EOL
                        . 'call_user_func([' . trim($match[2]) . ','
                        . trim($match[1]) . ']' . $params . ');';
        },

    // 使用函数 call_user_func_array() 替换 call_user_method_array()
    '!call_user_method_array\((.*?),(.*?),(.*?)\)(\b|;)!' =>
        function ($match) {
            return '// WARNING: call_user_method_array()'
                        . 'has been removed from PHP 7'
                        . PHP_EOL
                        . 'call_user_func_array(['
                        . trim($match[2]) . ','
                        . trim($match[1]) . '], '
                        . $match[3] . ');';
        },
```

释义
//警告：PHP 7 已经废弃了 call_user_method() 函数。

释义
//警告：PHP 7 已经废弃了 call_user_method_array() 函数。

6. 使用 preg_replace_callback()函数替换带有/e 修饰符的 preg_replace()函数：

```
'!^(.*?)preg_replace.*?/e(.*?)$!' =>
function ($match) {
    $last = strrchr($match[2], ',');
    $arg2 = substr($match[2], 2, -1 * (strlen($last)));
    $arg1 = substr($match[0],
                   strlen($match[1]) + 12,
                   -1 * (strlen($arg2) + strlen($last)));
     $arg1 = trim($arg1, '(');
     $arg1 = str_replace('/e', '/', $arg1);
     $arg3 = '// WARNING: preg_replace() "/e" modifier
                 . 'has been removed from PHP 7'
                 . PHP_EOL
                 . $match[1]
                 . 'preg_replace_callback('
                 . $arg1
                 . 'function ($m) { return '
                 . str_replace('$1','$m', $match[1])
                 .      trim($arg2, '"\'') . '; }, '
                 .      trim($last, ',');
        return str_replace('$1', '$m', $arg3);
    },

        // 结束数组
        ],

        // 这是转换操作的目标
        $contents
    );
    // 将结果转换为字符串返回
    return implode('', $result);
}
```

释义
// 警告：PHP 7 已经废弃了带/e 修饰符的 preg_replace()。

具体运行情况

要使用这个转换器，可通过命令行界面运行下面的代码。你需要通过命令行界面将要转换的 PHP 5 代码的文件名称设置为参数。

这段代码存储在 chap_01_php5_to_php7_code_converter.php 文件中，通

过命令行界面运行这段代码可以调用前面介绍的转换器：

```
<?php
//通过命令行界面设置含有要转换的 PHP 5 代码的文件的名称
$filename = $argv[1] ?? '';

if (!$filename) {
    echo 'No filename provided' . PHP_EOL;
    echo 'Usage: ' . PHP_EOL;
    echo __FILE__ . ' <filename>' . PHP_EOL;
    exit;
}

// 设置类自动加载功能
require __DIR__ . '/../Application/Autoload/Loader.php';

// 将当前目录添加到自动加载路径中
Application\Autoload\Loader::init(__DIR__ . '/..');

// 获取执行深层次扫描操作的类
$convert = new Application\Parse\Convert();
echo $convert->scan($filename);
echo PHP_EOL;
```

扩展

要详细了解 PHP 7 不向下兼容的情况，请浏览 http://php. net/manual/en/migration70.incompatible.php。

第 2 章　PHP 7 中的高效功能

本章介绍 PHP 5 与 PHP 7 之间的语法差异，下面是本章要点：

- 了解抽象语法树
- 了解语法分析中的差异
- 了解 `foreach()` 处理过程中的差异
- 使用 PHP 7 中的增强功能提高性能
- 遍历含有大量数据的文件
- 将电子表格上传到数据库中
- 递归式目录迭代器

本章主要内容简介

本章将介绍 PHP 7 中新增的高效功能。为了展示 PHP 7 在处理参数解析、语法和 `foreach()` 函数等方面的改进，本章会举几个小例子介绍 PHP 7 与 PHP 5 差异。在深入探讨本章的内容前，让我们先了解 PHP 7 与 PHP 5 之间的几个主要差异。

PHP 7 引入了一种名为**抽象语法树**（Abstract Syntax Tree，AST）的新处理层，该层有效地降低了语法分析处理过程与伪编译处理过程之间的耦合性。尽管这个新增的处理层为性能带来的提升很小（甚至完全没有），但是它为这门编程语言带来了前所未有的语法一致性。

AST 的另一个优点是复引用（dereferencing）处理过程。简言之，复引用[1]是指立刻获得对象中的某个属性、运行对象中的某个方法、访问数组中的某个元素和调用某个回调函数的操作。PHP 5 对这类操作的支持既不方便也不彻底，例如，要调用某个回调函数，通常需要先将回调函数或匿名函数赋予一个变量，然后才能运行该函数。而在 PHP 7 中，你可以立刻运行该函数。

1 复引用（dereferencing）源自 C 语言中的指针间接引用操作，是指获取指针指向地址所存储的值。——译者注

了解抽象语法树

作为开发者，你可能会对解除 PHP 5 及更早版本中的某些语法限制感兴趣。除了前面提到的语法一致性（PHP 语法中的最大进步）外，还可以仅通过添加一组圆括号来实现对任何返回值的调用。而且，在返回值为数组的情况下，你可以直接访问数组中的任何元素。

具体处理过程

1. 仅通过添加圆括号 ()（可带参数也可不带参数），就可以立刻运行任何返回回调函数的函数或方法。仅通过方括号指明要获取的元素，就可以从返回数组的函数或方法中立刻获取该元素；在下面短小（但琐碎）的示例中，函数 `test()` 返回了一个数组。该数组含有 6 个匿名函数。变量 `$a` 拥有变量 `$t` 的值。`$$a` 表达式被解析为 `$test`：

```
function test()
{
    return [
        1 => function () { return [
            1 => function ($a) { return 'Level 1/1:' . ++$a; },
            2 => function ($a) { return 'Level 1/2:' . ++$a; },
        ];},
        2 => function () { return [
            1 => function ($a) { return 'Level 2/1:' . ++$a; },
            2 => function ($a) { return 'Level 2/2:' . ++$a; },
        ];}
    ];
}

$a = 't';
$t = 'test';
echo $$a()[1]()[2](100);
```

2. AST 使我们能够使用 `echo $$a()[1]()[2](100)` 命令。该命令是以从左到右的次序解析的：

- `$$a()` 被解析为 `test()`，`test()` 函数会返回一个数组。
- `[1]` 会获取这个数组中的元素 1，该元素会返回一个回调函数。

- ()会运行这个回调函数，该回调函数会返回一个数组中的两个元素。
- [2]会获取这个数组中的元素 2，该元素会返回一个回调函数。
- (100)会通过参数 100 运行这个回调函数,该回调函数会返回 Level 1/2:101。

phar 不能在 PHP 5 中使用这样的语句，因为会遇到解析错误。

3. 下面的示例更为真实，它利用 AST 语法定义了一个数据过滤和验证类。先定义 Application\Web\Securityclass 类，然后在下面的构造器中创建并定义两个数组。第一个数组用于存储执行过滤操作的回调函数，第二个数组用于存储执行验证操作的回调函数：

```
public function __construct()
  {
    $this->filter = [
      'striptags' => function ($a) { return strip_tags($a); },
      'digits' => function ($a) { return preg_replace(
      '/[^0-9]/', '', $a); },
      'alpha' => function ($a) { return preg_replace(
      '/[^A-Z]/i', '', $a); }
    ];
    $this->validate = [
      'alnum' => function ($a) { return ctype_alnum($a); },
      'digits' => function ($a) { return ctype_digit($a); },
      'alpha' => function ($a) { return ctype_alpha($a); }
    ];
}
```

4. 我们想要通过对开发者友好的方式调用这个功能。因此，如果想要过滤数字，最好运行下面这类命令：

```
$security->filterDigits($item));
```

5. 为了做到这一点，我们定义了一个功能强大的方法__call()，该方法使我们能够访问还不存在的方法：

```
public function __call($method, $params)
{
  preg_match('/^(filter|validate)(.*?)$/i', $method, $matches);
  $prefix = $matches[1] ?? '';
  $function = strtolower($matches[2] ?? '');
```

```
    if ($prefix && $function) {
      return $this->$prefix[$function]($params[0]);
    }
    return $value;
  }
```

我们使用 preg_match()函数将$method 参数与单词 filter 和 validate 作比对。第二个子匹配操作的结果会被转化为通过执行$this->filter 或$this->validate 语句获得的数组的下标（关联数组的键）。如果两个子匹配操作都获得了结果，我们就将第一个结果赋予结果$prefix，将第二个变量赋予$function。在执行相应的回调函数时，这两个变量最终会被用作参数。

不要滥用这种处理方式！

如果你正得意于 AST 带来的表达式自由处理方式，那么请不要忘记，在运行周期较长的项目中，你编写的代码最终可能会变得极难理解。这将会变成长期的维护问题。

具体运行情况

我们先创建示例文件 chap_02_web_filtering_ast_example.php，以便利用第 1 章介绍过的类自动加载功能来获取 Application\Web\Security 类的实例：

```
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
$security = new Application\Web\Security();
```

然后，定义一个用于测试数据的代码块：

```
$data = [
    '<ul><li>Lots</li><li>of</li><li>Tags</li></ul>',
    12345,
    'This is a string',
    'String with number 12345',
];
```

最后，使用过滤器和验证器处理测试数据的每一项：

```
foreach ($data as $item) {
  echo 'ORIGINAL: ' . $item . PHP_EOL;
```

```
    echo 'FILTERING' . PHP_EOL;
    printf('%12s : %s' . PHP_EOL,'Strip Tags',
      $security->filterStripTags($item));
    printf('%12s : %s' . PHP_EOL, 'Digits',
      $security->filterDigits($item));
    printf('%12s : %s' . PHP_EOL, 'Alpha',
      $security->filterAlpha($item));

    echo 'VALIDATORS' . PHP_EOL;
    printf('%12s : %s' . PHP_EOL, 'Alnum',
    ($security->validateAlnum($item)) ? 'T' : 'F');
    printf('%12s : %s' . PHP_EOL, 'Digits',
    ($security->validateDigits($item)) ? 'T' : 'F');
    printf('%12s : %s' . PHP_EOL, 'Alpha',
    ($security->validateAlpha($item)) ? 'T' : 'F');
}
```

下面是部分输出结果：

```
aed@aed: ~/Repos/php7_recipes/source/chapter02
ORIGINAL: <ul><li>Lots</li><li>of</li><li>Tags</li></ul>
FILTERING
  Strip Tags : LotsofTags
       Digits :
        Alpha : ulliLotsliliofliliTagsliul
VALIDATORS
        Alnum : F
       Digits : F
        Alpha : F
ORIGINAL: 12345
FILTERING
  Strip Tags : 12345
       Digits : 12345
        Alpha :
VALIDATORS
        Alnum : T
       Digits : T
        Alpha : F
ORIGINAL: This is a string
FILTERING
  Strip Tags : This is a string
       Digits :
        Alpha : Thisisastring
VALIDATORS
        Alnum : F
       Digits : F
        Alpha : F
ORIGINAL: String with number 12345
FILTERING
  Strip Tags : String with number 12345
       Digits : 12345
        Alpha : Stringwithnumber
VALIDATORS
        Alnum : F
       Digits : F
        Alpha : F
aed@aed:~/Repos/php7_recipes/source/chapter02$
```

扩展

要获取更多关于 AST 的信息，请浏览 `https://wiki.php.net/rfc/abstract_syntax_tree` 中介绍 Abstract Syntax Tree 的 RFC 文件。

了解语法分析中的差异

在 PHP 5 中，赋值操作右侧的表达式会按照从*右至左*的次序被解析。在 PHP 7 中，解析的次序始终是从*左至右*。

具体处理过程

1. 变量套变量是一种间接引用值的方式。在下面的例子中，第一个`$$foo` 语句会被解析为`${$bar}`。因此，最终返回的值会是变量`$bar` 中保存的值，而不是变量`$foo` 中保存的值（字符串`'bar'`）：

```
$foo = 'bar';
$bar = 'baz';
echo $$foo; // 会显示字符串'baz'
```

2. 下面的例子中有一个变量套变量的表达式`$$foo`，该表达式引用了一个多维数组，该数组将字符串`'bar'`和`'bat'`用作它的关联键（即数组下标）：

```
$foo = 'bar';
$bar = ['bar' => ['baz' => 'bat']];
// 会显示字符串'bat'
echo $$foo['bar']['baz'];
```

3. 在 PHP 5 中，解析的次序为从右至左，这意味着 PHP 引擎会寻找以字符串`'bar'`和`'bat'`为关联键的数组`$foo`。那么，该表达式的返回值会被解析为`${$foo['bar']['baz']}`。

4. 然而在 PHP 7 中，解析次序始终为从左至右，这意味着该表达式会被解析为`($$foo)['bar']['baz']`。

5. 对于下面例子中的表达式`$foo->$bar['bada']`，PHP 5 与 PHP 7 的解析方式大相径庭。在下面的示例中，PHP 5 会先解析`$bar['bada']`，然后根据变量`$foo` 中存储的对象引用`$bar['bada']`的返回值。另一方面，PHP 7 会先解析`$foo->$bar`，然后将字符串`'bada'`视为该数组的关联键。还应注意，本例使用 PHP 7 中匿名类功能：

```
// 在 PHP 5 中会将该表达式解析为$foo->{$bar['bada']}
// 在 PHP 7 中会将该表达式解析为($foo->$bar)['bada']
$bar = 'baz';
// 在变量$foo 中存储一个新建的对象
```

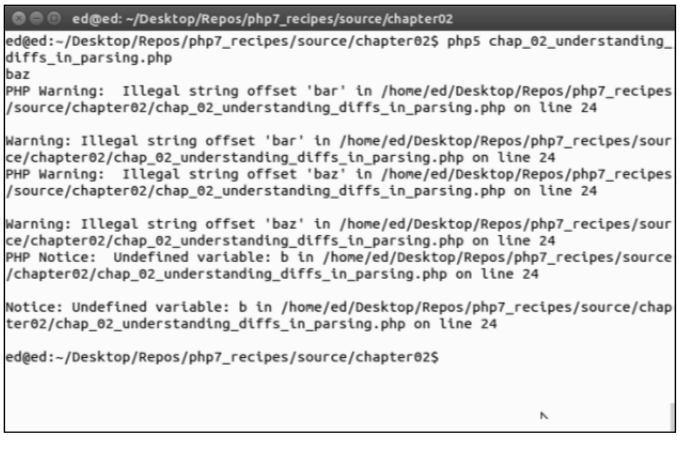
```
$foo = new class
{
    public $baz = ['bada' => 'boom'];
};
// 会显示字符串'boom'
echo $foo->$bar['bada'];
```

6. 最后一个示例与上面的示例相同，只是将返回值设置为回调函数，该回调函数会被立刻执行：

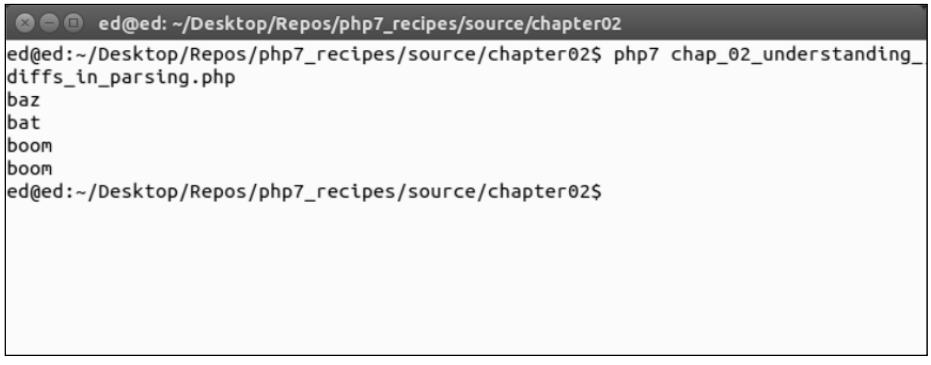
```
//在 PHP 5 中会将该表达式解析为$foo->{$bar['bada']}()
//在 PHP 7 中会将该表达式解析为['bada']()
$bar = 'baz';
// 注意：本例中使用了 PHP 7 新增的匿名类功能
$foo = new class
{
    public function __construct()
    {
        $this->baz = ['bada' => function () { return 'boom'; }];
    }
};
// 会显示字符串'boom'
echo $foo->$bar['bada']();
```

具体运行情况

请将步骤 1 和 2 步骤中介绍的代码放置到一个 PHP 文件中（可以将其命名为 chap_02_understanding_diffs_in_parsing.php）。先使用 PHP 5 运行该脚本，你会看到如下所示的一系列错误提示：



出现这些错误的原因是，PHP 5 的解析处理过程不具有一致性，由于多变的变量表达式状态（如前所述），最终导致得出错误结论。你还可以将步骤 5 和步骤 6 介绍的代码添加到该文件中。如果在 PHP 7 运行这个脚本文件，就能得到清晰的结果：



扩展

要了解更多关于解析过程的信息，请浏览 `https://wiki.php.net/rfc/uniform_variable_syntax` 中介绍 Uniform Variable Syntax 的 RFC 文件。

了解 foreach()处理过程中的差异

在某些不太引人注目的情况中，PHP 5 的 `foreach()` 循环中的操作与 PHP 7 的 `foreach()` 循环中的操作有差异。PHP 7 的 `foreach()` 循环有了许多改进，这意味着与 PHP 5 相比，PHP 7 的 `foreach()` 循环中的操作会以快得多的速度被执行。大家都已经注意到在 PHP 5 中，在 `foreach()` 循环中对数组使用 `current()` 和 `unset()` 函数会引发问题。而且，在使用 `foreach()` 循环操作数组时，通过引用操作传送值也会引发问题。

具体处理过程

1. 请查看下面的代码：

```
$a = [1, 2, 3];
foreach ($a as $v) {
  printf("%2d\n", $v);
```

```
  unset($a[1]);
}
```

2. 在 PHP 5 和 PHP 7 中，这段代码的输出结果如下所示：

```
1
2
3
```

3. 然而，如果你在这个循环的前面添加赋值语句，那么 `foreach()` 循环中的操作就会改变：

```
$a = [1, 2, 3];
$b = &$a;
foreach ($a as $v) {
  printf("%2d\n", $v);
  unset($a[1]);
}
```

4. 请对比 PHP 5 和 PHP 7 中的输出结果：

PHP 5	PHP 7
1	1
3	2
	3

5. 在 PHP 5 中使用引用数组内部指针的函数时，也会导致出现不一致的操作。请看下面的代码示例：

```
$a = [1,2,3];
foreach($a as &$v) {
    printf("%2d - %2d\n", $v, current($a));
}
```

每个数组都拥有一个内部指针，该指针会指向当前元素（该指针最初指向的是数组中的第一个元素），`current()` 函数会返回数组中的当前元素。

6. 注意，这段代码在 PHP 7 中的输出结果是标准的并且具有一致性：

PHP 5	PHP 7
1 - 2	1 - 1
2 - 3	2 - 1
3 - 0	3 - 1

7. 一旦该数组通过引用操作执行的迭代过程结束，在 PHP 5 中，在 `foreach()` 循

环中添加一个新元素也会出问题。PHP 7 已经解决了这个问题，请看下面的示例代码：

```
$a = [1];
foreach($a as &$v) {
    printf("%2d -\n", $v);
    $a[1]=2;
}
```

8. 运行这段代码会得到下面的输出结果：

PHP 5	PHP 7
1 -	1 -
	2 -

9. PHP 7 还处理了另一个在 PHP 5 中通过引用操作处理数组迭代的问题：使用函数修改数组，这些函数包括 `array_push()`、`array_pop()`、`array_shift()` 和 `array_unshift()`。

请看下面的示例：

```
$a=[1,2,3,4];
foreach($a as &$v) {
    echo "$v\n";
    array_pop($a);
}
```

10. 运行这段代码会得到下面的输出结果：

PHP 5	PHP 7
1	1
2	2
1	
1	

11. 接下来的例子中使用嵌套的 `foreach()` 循环（嵌套的内层 `foreach()` 循环也使用引用操作迭代该数组），通过引用操作迭代一个数组。在 PHP 5 中这种结构的代码根本无法运行。PHP 7 已经修复了这个问题。下面的代码展示了这类操作：

```
$a = [0, 1, 2, 3];
foreach ($a as &$x) {
     foreach ($a as &$y) {
       echo "$x - $y\n";
       if ($x == 0 && $y == 1) {
         unset($a[1]);
         unset($a[2]);
```

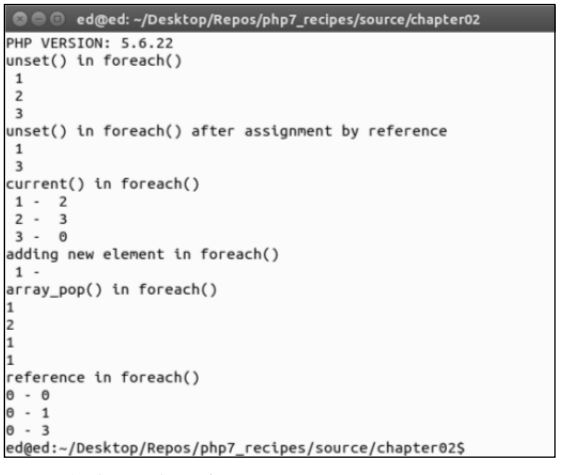
```
        }
    }
}
```

12. 下面是运行这段代码得到的输出结果：

PHP 5	PHP 7
0 - 0	0 - 0
0 - 1	0 - 1
0 - 3	0 - 3
	3 - 0
	3 - 3

具体运行情况

可以将上述代码添加到一个 PHP 文件中，并将它命名为 `chap_02_foreach.php`。通过在 PHP 5 的命令行界面中运行这个脚本，会得到下面的输出结果：



在 PHP 7 中运行这个脚本，并观察与前面输出结果之间的差异：

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter02
PHP VERSION: 7.0.7
unset() in foreach()
 1
 2
 3
unset() in foreach() after assignment by reference
 1
 2
 3
current() in foreach()
 1 -  1
 2 -  1
 3 -  1
adding new element in foreach()
 1 -
 2 -
array_pop() in foreach()
1
2
reference in foreach()
0 - 0
0 - 1
0 - 3
3 - 0
3 - 3
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter02$
```

扩展

要了解更多信息，请参考已被大家公认的讨论该问题的 RFC 文件（`https://wiki.php.net/rfc/php7_foreach`）。

使用 PHP 7 中的增强功能提高性能

程序设计的一个趋势是开发者们喜欢使用**匿名函数**。匿名函数处理中的一个经典问题是使任何对象都能够与`$this` 变量绑定，而且匿名函数仍旧能够运行。PHP 5 使用`bindTo()`方法解决该问题。PHP 7 新增了 `call()`方法，该方法在实现类似功能的基础上，极大地提高了性能。

具体处理过程

要使用 `call()`方法，需要在较长的循环中执行匿名函数。本例会介绍一个匿名函数，该函数会扫描一个日志文件，能够根据出现频率识别 IP 地址的排列。

1. 我们先定义 `Application\Web\Access` 类。这个构造器会将文件名作为参数来接收。该日志文件会被打开并被设置为 `SplFileObject` 对象，然后将其赋予

$this->log 变量：

```
Namespace Application\Web;

use Exception;
use SplFileObject;
class Access
{
  const ERROR_UNABLE = 'ERROR: unable to open file';
  protected $log;
  public $frequency = array();
  public function __construct($filename)
  {
    if (!file_exists($filename)) {
      $message = __METHOD__ . ' : ' . self::ERROR_UNABLE . PHP_EOL;
      $message .= strip_tags($filename) . PHP_EOL;
      throw new Exception($message);
    }
    $this->log = new SplFileObject($filename, 'r');
  }
```

2. 定义一个生成器，以便逐行迭代这个日志文件：

```
public function fileIteratorByLine()
{
  $count = 0;
  while (!$this->log->eof()) {
    yield $this->log->fgets();
    $count++;
  }
  return $count;
}
```

3. 定义一个执行查找操作的方法，通过匹配条件提取 IP 地址：

```
public function getIp($line)
{
  preg_match('/(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/',
    $line, $match);
  return $match[1] ?? '';
  }
}
```

具体运行情况

先定义用于执行调用操作的程序：chap_02_performance_using_php7_enchancement_call.php，在该文件中添加第 1 章介绍过的类自动加载功能，以便获取 Application\Web\Access 类的实例：

```
define('LOG_FILES', '/var/log/apache2/*access*.log');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
```

然后定义匿名函数，使用该函数处理日志文件中的一行内容。如果检测到了 IP 地址，那么该 IP 地址就会成为$frequency 数组中的一个键，而且与该键对应的数组元素含有的当前值也会增加：

```
// 定义函数
$freq = function ($line) {
  $ip = $this->getIp($line);
  if ($ip) {
    echo '.';
    $this->frequency[$ip] =
    (isset($this->frequency[$ip])) ? $this->frequency[$ip] + 1 : 1;
  }
};
```

接下来，循环处理每个日志文件中每一行的迭代过程，以便处理 IP 地址：

```
foreach (glob(LOG_FILES) as $具体文件名) {
  echo PHP_EOL . $具体文件名 . PHP_EOL;
  // 访问日志文件对象
  $access = new Application\Web\Access($具体文件名);
  foreach ($access->fileIteratorByLine() as $line) {
    $freq->call($access, $line);
  }
}
```

实际上，你可以在 PHP 5 中使用相同代码。但必须使用下面的代码替换其中的 call()方法：

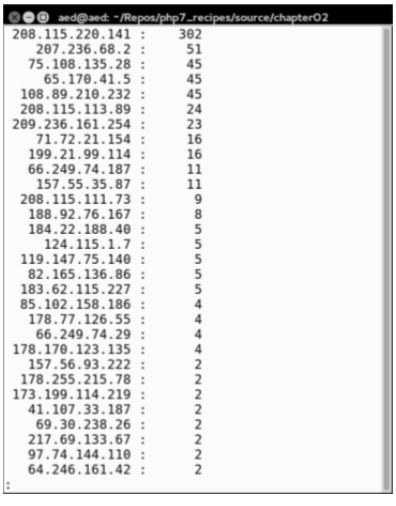
```
$func = $freq->bindTo($access);
$func($line);
```

不过，性能会比 PHP 7 中的 call()方法低 20%至 50%。

最后，将这个数组反向排序，但保持其中的键不变。使用一个简单的 `foreach()` 循环输出结果：

```
arsort($access->frequency);
foreach ($access->frequency as $key => $value) {
  printf('%16s : %6d' . PHP_EOL, $key, $value);
}
```

输出结果会依据不同的被扫描日志文件而改变，下面是输出结果示例：



补充说明

PHP 7 中的许多性能提升都与新增的功能和函数无关。更确切地说，这些性能提升是以内部调整的形式出现的，在运行你编写的程序前它们都是隐形的。下表列出了这类性能提升：

功能	参考资料	注释
快速参数解析	https://wiki.php.net/rfc/fast_zpp	在 PHP 5 中，必须在每次执行函数调用操作时解析。这些参数会被作为字符串传递，而且会通过与 `scanf()` 函数类似的方式解析提供给函数的参数。在 PHP 7 中该解析过程已经被优化，并实现了高得多的效率，从而实现了大幅度的性能提升。该提升的确切幅度难以测量，但可估算为 6%左右
PHP NG	https://wiki.php.net/rfc/phpng	PHP NG（PHP Next Generation）首次重写了大部分 PHP 语言。它保留了已有的功能，增加了许多省时和高效的功能。它压缩了数据结构，并提高了使用内存的效率。例如，仅仅是对数组处理方式的一项更改，就获得了大幅度的性能提升，同时还大幅度减少了占用的内存

续表

功能	参考资料	注释
减轻自重	https://wiki.php.net/rfc/removal_of_dead_sapis_and_exts	被判定为废弃、不再维护、非维护依赖关系和不再在 PHP 7 中包含的扩展有 20 多个。根据核心开发者小组的投票决定，去除 PHP 中约 2/3 的扩展（进入裁减名单中的扩展）。这减少了运行 PHP 程序时的额外开销，并提高了 PHP 项目的整体开发速度

遍历含有大量数据的文件

file_get_contents()和 file()之类函数的运行速度快且易于使用，但由于它们受内存限制，在处理含有大量数据的文件时，这些函数很快会引发问题。php.ini memory_limit 配置的默认设置是 128MB。因此，所有大于该容量的文件都无法加载。

解析含有大量数据的文件时需要考虑的另一个问题是，函数或类方法是否能够快速执行输出操作？例如，在处理用户的输出操作时，尽管乍看之下将输出内容存储在数组中比较好，而且为了提高效率，最好将这些内容一次性全部输出。但令人遗憾的是，这样做可能会对用户体验产生较坏的影响。因此，最好创建一个构造器，并使用 yield 关键字生成能够立刻输出的结果。

具体处理过程

如前所述，file*函数（即 file_get_contents()等函数）不适合处理较大的文件。原因很简单，因为这类函数会一次将一个文件的全部内容都塞入内存中。因此，本节将着重介绍 f*函数（即 fopen()之类的函数）。

但我们需要略微迂回一下，不直接使用 f*函数，而使用 PHP 标准库（Standard PHP Library，SPL）中的 SplFileObject 类：

1. 先定义 Application\Iterator\LargeFile 类，并为它添加适当的属性和常量：

```
namespace Application\Iterator;

use Exception;
use InvalidArgumentException;
use SplFileObject;
use NoRewindIterator;
```

```
class LargeFile
{
  const ERROR_UNABLE = 'ERROR: Unable to open file';
  const ERROR_TYPE = 'ERROR: Type must be "ByLength",
    "ByLine" or "Csv"';
  protected $file;
  protected $allowedTypes = ['ByLine', 'ByLength', 'Csv'];
```

2. 然后定义__construct()方法，使该方法将文件名接收为参数，并将SplFileObject 对象赋予$file 属性。如果要处理的文件不存在，那么这段代码就会抛出异常：

```
public function __construct($filename, $mode = 'r')
{
  if (!file_exists($filename)) {
    $message = __METHOD__ . ' : ' . self::ERROR_UNABLE . PHP_EOL;
    $message .= strip_tags($filename) . PHP_EOL;
    throw new Exception($message);
  }
  $this->file = new SplFileObject($filename, $mode);
}
```

3. 定义 fileIteratorByLine()方法，并使用该方法通过 fgets()函数每次从要处理的文件中读取一行内容。也可以创建 fileIteratorByLength()方法，该方法会执行相同的操作，但会用 fread()函数代替 fgets()函数。使用 fgets()函数的方法适合于处理含有换行符的文本文件，使用 fread()函数的方法适合于处理较大的二进制文件：

```
protected function fileIteratorByLine()
{
  $count = 0;
  while (!$this->file->eof()) {
    yield $this->file->fgets();
    $count++;
  }
  return $count;
}

protected function fileIteratorByLength($numBytes = 1024)
{
  $count = 0;
  while (!$this->file->eof()) {
```

```
        yield $this->file->fread($numBytes);
        $count++;
      }
      return $count;
    }
```

4. 定义 getIterator()方法，使该方法返回 NoRewindIterator()实例。该方法将 ByLine 或 ByLength 变量接收为参数，这两个变量与上一步骤定义的 fileIteratorByLine 和 fileIteratorByLength 函数有关。这个方法还需要接收$numBytes 变量，以便在使用 ByLength 参数调用该方法时执行操作。使用 NoRewindIterator()实例的作用是确保这个程序仅以一个方向读取文件中的内容：

```
public function getIterator($type = 'ByLine', $numBytes = NULL)
{
  if(!in_array($type, $this->allowedTypes)) {
    $message = __METHOD__ . ' : ' . self::ERROR_TYPE . PHP_EOL;
    throw new InvalidArgumentException($message);
  }
  $iterator = 'fileIterator' . $type;
  return new NoRewindIterator($this->$iterator($numBytes));
}
```

具体运行情况

我们先利用第 1 章介绍的类自动加载功能，通过调用程序 chap_02_iterating_through_a_massive_file.php 获取 Application\Iterator\LargeFile 类的实例：

```
define('MASSIVE_FILE', '/../data/files/war_and_peace.txt');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
```

然后在 try {...}catch(){...}代码块中获取 ByLine 迭代器的实例：

```
try {
  $largeFile = new Application\Iterator\LargeFile(__DIR__ . MASSIVE_
FILE);
  $iterator = $largeFile->getIterator('ByLine');
```

执行一个具有实际意义的操作，本例选用获取每行含有单词的平均数量：

```
$words = 0;
foreach ($iterator as $line) {
```

```
    echo $line;
    $words += str_word_count($line);
}
echo str_repeat('-', 52) . PHP_EOL;
printf("%-40s : %8d\n", 'Total Words', $words);
printf("%-40s : %8d\n", 'Average Words Per Line',
($words / $iterator->getReturn()));
echo str_repeat('-', 52) . PHP_EOL;
```

结束 catch 代码块：

```
} catch (Throwable $e) {
    echo $e->getMessage();
}
```

这段程序查明古腾堡版的《战争与和平》中含有 566 095 个单词（因内容较多，此处就不列出输出结果了）。而且，这本小说中每行含有的单词数量平均为 8 个。

将电子表格上传到数据库中

尽管 PHP 没有直接读取电子表格格式文件（如 XLSX 和 ODS 等）的功能，但它确实能够读取逗号分隔型取值格式（Comma Separated Values，CSV）文件。因此，为了处理用户的电子表格，你需要请他们使用 CSV 格式保存文件，或者自己将其他格式的文件转换为 CSV 格式文件。

准备工作

当将电子表格（即 CSV 文件）上传到数据库中时，应注意下列三点：

- 遍历（含有大量数据的）文件
- 获取电子表格中每行中的数据并将它们存储到 PHP 的数组中
- 将 PHP 的数组插入数据库中

可使用前面介绍的方式遍历含有大量数据的文件。我们将使用 fgetcsv()函数将 CSV 文件中的一行数据转换成 PHP 的数组，最后使用 PHP 数据对象（PHP Data Objects，PDO）与数据库相连，并执行该插入操作。

具体处理过程

1. 先定义 Application\Database\Connection 类，这个该类会根据为下面构造器提供的一系列参数创建一个 PDO 实例：

```
<?php
  namespace Application\Database;

  use Exception;
  use PDO;

  class Connection
  {
    const ERROR_UNABLE = 'ERROR: Unable to create database
      connection';
    public $pdo;

    public function __construct(array $config)
    {
      if (!isset($config['driver'])) {
        $message = __METHOD__ . ' : ' . self::ERROR_UNABLE
            . PHP_EOL;
        throw new Exception($message);
  }
  $dsn = $config['driver']
  . ':host=' . $config['host']
  . ';dbname=' . $config['dbname'];
  try {
      $this->pdo = new PDO($dsn,
      $config['user'],
      $config['password'],
      [PDO::ATTR_ERRMODE => $config['errmode']]);
    } catch (PDOException $e) {
      error_log($e->getMessage());
    }
  }

}
```

2. 然后创建 Application\Iterator\LargeFile 类的实例。向这个专门用于

遍历 CSV 文件的类中添加一个新方法：

```
protected function fileIteratorCsv()
{
  $count = 0;
  while (!$this->file->eof()) {
    yield $this->file->fgetcsv();
    $count++;
  }
  return $count;
}
```

3. 还应将 Csv 参数添加到用于执行遍历操作的方法列表中：

```
const ERROR_UNABLE = 'ERROR: Unable to open file';
const ERROR_TYPE = 'ERROR: Type must be "ByLength",
"ByLine" or "Csv"';

protected $file;
protected $allowedTypes = ['ByLine', 'ByLength', 'Csv'];
```

具体运行情况

先定义配置文件/path/to/source/config/db.config.php，并在其中添加用于连接数据库的参数：

```
<?php
return [
  'driver'   => 'mysql',
  'host'     => 'localhost',
  'dbname'   => 'php7cookbook',
  'user'     => 'cook',
  'password' => 'book',
  'errmode'  => PDO::ERRMODE_EXCEPTION,
];
```

然后使用第 1 章介绍的类自动加载功能，获取 Application\ Database\ Connection 和 Application\Iterator\LargeFile 类的实例，定义调用程序 chap_02_uploading_csv_to_database.php：

```
define('DB_CONFIG_FILE', '/../data/config/db.config.php');
define('CSV_FILE', '/../data/files/prospects.csv');
require __DIR__ . '/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
```

设置 try{...}catch(){...}代码块,以便捕捉拥有 Throwable 接口的对象。这使我们既能够捕捉异常，又能够捕捉部分错误：

```
try {
  // 在此处添加执行具体操作的代码
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

在 try {...}catch(){...}代码块内部，我们可以获取用于建立连接的实例和用于遍历大型文件的类：

```
$connection = new Application\Database\Connection(
include __DIR__ . DB_CONFIG_FILE);
$iterator = (new Application\Iterator\LargeFile(__DIR__ . CSV_FILE))
->getIterator('Csv');
```

这样就可以利用 PDO 中的预处理/执行（prepare/execute）功能了。SQL 中的预处理语句使用符号？代表在循环中提供的值：

```
$sql = 'INSERT INTO `prospects` '
  . '(`id`,`first_name`,`last_name`,`address`,`city`,`state_
province`,'
  . '`postal_code`,`phone`,`country`,`email`,`status`,`budget`,
    `last_updated`) '
  . ' VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?)';
$statement = $connection->pdo->prepare($sql);
```

这样我们就可以使用 foreach()循环处理文件迭代器了。每一条 yield 语句都会生成代表数据库中某行数据的一组值。这样我们就可以通过 PDOStatement::execute()语句使用这些值，执行经过预处理的语句，以便将这些值插入数据库：

```
foreach ($iterator as $row) {
  echo implode(',', $row) . PHP_EOL;
  $statement->execute($row);
}
```

之后应检查数据库的内容，以查明这些数据是否已经被成功插入。

递归式目录迭代器

从一个目录中获取一系列文件很容易。从传统意义上讲，开发者们已经使用 glob()函数做到了这一点。要通过递归方式从目录树中的某个指定位置，获取一系列文件和目

录会更复杂一些。本节介绍 SPL 中的 RecursiveDirectoryIterator 类，使用该类可以极好地达到这个目的。

使用这个类可以解析目录树、查找第一级子目录，而且是沿着分支查找至最底层目录为止。令人遗憾的是，这不是我们想要获得的操作结果。有时我们需要使 RecursiveDirectoryIterator 对象从指定的起点解析整个目录树中的所有分支。功能强大的 RecursiveIteratorIterator 类恰好就能做到这一点。通过将 RecursiveDirectoryIterator 对象封装在 RecursiveIteratorIterator 对象中，就可以遍历整个目录树。

警告！

在遍历文件系统时应加倍小心。如果你从根目录开始执行遍历操作，那么最终就会使你的服务器崩溃，因为只有当所有文件和目录都被遍历后，递归处理过程才会停止。

具体处理过程

1. 先定义 Application\Iterator\Directory 类，为其定义适当的属性、常量以及导入外部类的方式（必须以加载类文件或启用类自动加载机制为前提）：

```
namespace Application\Iterator;

use Exception;
use RecursiveDirectoryIterator;
use RecursiveIteratorIterator;
use RecursiveRegexIterator;
use RegexIterator;

class Directory
{

  const ERROR_UNABLE = 'ERROR: Unable to read directory';

  protected $path;
  protected $rdi;
  // 以递归方式处理目录迭代器
```

2. 下面的构造器根据目录路径，在 RecursiveIteratorIterator 对象内部创

建了一个 RecursiveDirectoryIterator 实例：

```
public function __construct($path)
{
  try {
    $this->rdi = new RecursiveIteratorIterator(
      new RecursiveDirectoryIterator($path),
    RecursiveIteratorIterator::SELF_FIRST);
  } catch (\Throwable $e) {
    $message = __METHOD__ . ' : ' . self::ERROR_UNABLE . PHP_EOL;
    $message .= strip_tags($path) . PHP_EOL;
    echo $message;
    exit;
  }
}
```

3. 确定在迭代过程中执行哪些具体操作。可选择将该操作设置为模仿 Linux 中 `ls -l -R` 命令的输出结果。注意，可使用 `yield` 关键字高效地将这个方法变成**生成器**（Generator），从而使该方法能够被其所处对象外部的代码调用。在这个目录迭代处理过程中生成的所有对象，都是 SPL 中的 Spl `FileInfo` 对象，通过这类对象我们可以获取文件中的有用信息。下面是该方法的具体代码：

```
public function ls($pattern = NULL)
{
  $outerIterator = ($pattern)
  ? $this->regex($this->rdi, $pattern)
  : $this->rdi;
  foreach($outerIterator as $obj){
    if ($obj->isDir()) {
      if ($obj->getFileName() == '..') {
        continue;
      }
      $line = $obj->getPath() . PHP_EOL;
    } else {
      $line = sprintf('%4s %1d %4s %4s %10d %12s %-40s' . PHP_EOL,
      substr(sprintf('%o', $obj->getPerms()), -4),
      ($obj->getType() == 'file') ? 1 : 2,
      $obj->getOwner(),
      $obj->getGroup(),
      $obj->getSize(),
      date('M d Y H:i', $obj->getATime()),
      $obj->getFileName());
```

```
    }
    yield $line;
  }
}
```

4. 你可能已经注意到，这个方法的调用语句中含有文件类型参数。我们需要使用某种方式过滤迭代结果，以便获取符合条件的文件。SPL 提供了一种非常适合完成这个任务的迭代器（RegexIterator 类）：

```
protected function regex($iterator, $pattern)
{
  $pattern = '!^.' . str_replace('.', '\\.', $pattern) . '$!';
  return new RegexIterator($iterator, $pattern);
}
```

5. 使用下面的方法可以模仿 dir /s 命令：

```
public function dir($pattern = NULL)
{
  $outerIterator = ($pattern)
  ? $this->regex($this->rdi, $pattern)
  : $this->rdi;
  foreach($outerIterator as $name => $obj){
     yield $name . PHP_EOL;
    }
  }
}
```

具体运行情况

先利用第 1 章介绍的类自动加载功能获取 Application\Iterator\Directory 类的实例，并定义调用程序 chap_02_recursive_directory_iterator.php：

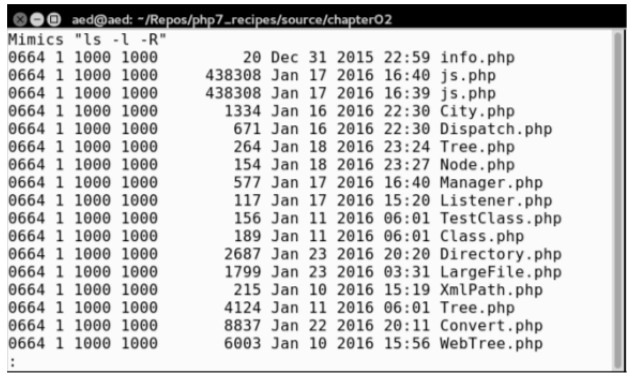
```
define('EXAMPLE_PATH', realpath(__DIR__ . '/../'));
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
$directory = new Application\Iterator\Directory(EXAMPLE_PATH);
```

然后，在 try {...} catch(){...}代码块中通过示例目录路径添加调用前面介绍的两个方法的语句：

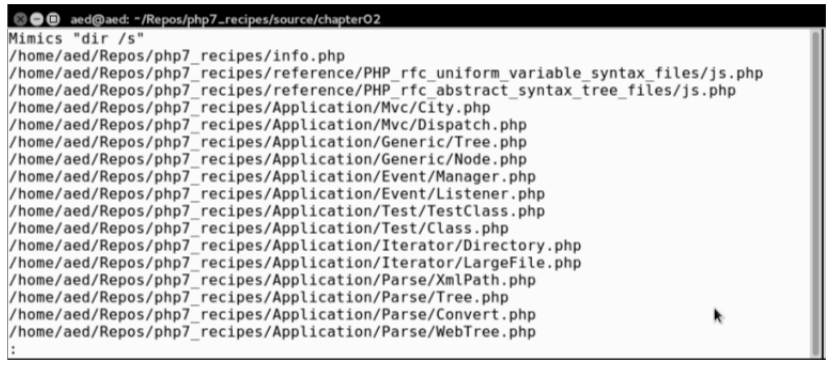
```
try {
  echo 'Mimics "ls -l -R" ' . PHP_EOL;
```

```
    foreach ($directory->ls('*.php') as $info) {
      echo $info;
    }

echo 'Mimics "dir /s" ' . PHP_EOL;
foreach ($directory->dir('*.php') as $info) {
  echo $info;
    }

  } catch (Throwable $e) {
    echo $e->getMessage();
  }
```

下面是 `ls()` 方法的输出结果：



下面是 `dir()` 方法的输出结果：



第 3 章　PHP 中的函数式编程功能

本章包括以下要点：

- 开发函数
- 提示数据类型
- 设置函数返回值的数据类型
- 使用迭代器
- 使用生成器编写自己的迭代器

本章主要内容简介

本章介绍 PHP 中的**函数式编程**功能。函数式（过程式）编程是 PHP 中传统的代码编写方式，该方式比在 PHP 4 中引入的**面向对象编程**（Object-Oriented Programming，OOP）方式出现得更早。在使用函数式编程方式时，程序逻辑会封装在一系列离散函数中。通常这些函数会被存储在一个独立的 PHP 文件中，这样就可以将这个 PHP 文件包含到任何脚本中，从而能自由地调用这些函数。

开发函数

函数式程序设计中的最难点是，确定将程序逻辑分解成函数的方式。在 PHP 中创建函数很简单。只需使用 `function` 关键字为函数起个名字，然后加上圆括号就行了。

具体处理过程

1. 函数的代码应放在花括号中：

```
function someName ($参数)
{
  $result = 'INIT';
```

```
    // 这里应添加执行具体操作的一条或多条语句,
    // 以便处理$result 变量
    $result .= ' and also ' . $参数;
    return $result;
}
```

2. 可以在函数中定义一个或多个**参数**。要将某个参数设置为可选项，只需为其赋予默认值。如果你无法确定为参数赋予怎样的默认值，可为其赋予 NULL：

```
function someOtherName ($必选参数, $可选参数 = NULL)
  {
    $result = 0;
    $result += $必选参数;
    $result += $可选参数 ?? 0;
    return $result;
  }
```

不能重复定义函数。唯一的例外是可以在不同的命名空间中定义相同的函数。下面的函数定义会产生错误：

```
function someTest()
{
 return 'TEST';
}
function someTest($a)
{
 return 'TEST:' . $a;
}
```

3. 如果你不知道你编写的函数需要接收多少个参数，或者你希望自己编写的函数能够接收无数个参数，可使用...和一个变量名。函数接收的所有参数都会成为该变量中的数组元素：

```
function someInfinite(...$params)
{
  // 任何接收到的参数都会存储到$params 数组中
  return var_export($params, TRUE);
}
```

4. 函数可以调用自己，这种调用方式称为**递归**。下面的函数执行了以递归方式扫描目录的操作：

```
function someDirScan($dir)
```

```
{
  // 使用关键字 static 将$list 设置为静态变量，以便保留该变量的值
  static $list = array();
  // 获取当前路径的文件和目录列表
  $list = glob($dir . DIRECTORY_SEPARATOR . '*');
  // 通过循环方式执行操作
  foreach ($list as $item) {
    if (is_dir($item)) {
      $list = array_merge($list, someDirScan($item));
    }
  }
  return $list;
}
```

在 PHP 中，在函数的内部使用关键字 static 的历史已经超过 12 年了。static 关键字的作用是将变量初始化一次（在声明静态变量的时候），然后在同一请求处理过程中执行不同函数调用操作时保留该值。

如果你需要在处理不同 HTTP 请求的过程中保留某个变量的值，应确保已经启动了 PHP 会话（session），并将该值保存在了$_SESSION 变量中。

5. 在 PHP 的**命名空间**中定义的函数是被约束的。这个特点有助于在函数库之间添加额外的逻辑分隔。要使用命名空间，需要添加 use 关键字。下面的示例被放置在不同的命名空间中。注意，即使函数名称相同，也不会引发冲突，就像这些函数相互看不到对方一样。

6. 我们在命名空间 Alpha 中定义了函数 someFunction()，将它保存在一个独立的 PHP 文件中（chap_03_developing_functions_namespace_alpha.php）：

```
<?php
namespace Alpha;

function someFunction()
{
  echo __NAMESPACE__ . ':' . __FUNCTION__ . PHP_EOL;
}
```

7. 然后，在命名空间 Beta 中定义函数 someFunction()，将它保存到另一个独立的 PHP 文件（chap_03_developing_functions_namespace_beta.php）中：

```
<?php
namespace Beta;

function someFunction()
{
  echo __NAMESPACE__ . ':' . __FUNCTION__ . PHP_EOL;
}
```

8. 这样就可以通过在函数名称前添加由命名空间名称构成的前缀来调用函数 someFunction()：

```
include (__DIR__ . DIRECTORY_SEPARATOR
         . 'chap_03_developing_functions_namespace_alpha.php');
include (__DIR__ . DIRECTORY_SEPARATOR
         . 'chap_03_developing_functions_namespace_beta.php');
    echo Alpha\someFunction();
    echo Beta\someFunction();
```

最佳编程习惯

将函数库（和类）放在独立的文件中是公认的最佳编程习惯：一个文件使用一个命名空间，将具有唯一性的类或函数库保存在不同的文件中。可以在同一个命名空间中定义多个类或函数库。只有需要在逻辑上对功能进行分隔时，才需要使用命名空间。

具体运行情况

将所有逻辑上相互关联的函数放置到一个独立的 PHP 文件中是公认的最佳编程习惯。创建一个文件并将其命名为 chap_03_developing_functions_library.php，然后将下面的函数（前面已详细介绍过）放入其中：

- someName()
- someOtherName()
- someInfinite()
- someDirScan()
- someTypeHint()

可以将该文件包含到使用这些函数的代码中。

```
include (__DIR__ . DIRECTORY_SEPARATOR . 'chap_03_developing_
functions_library.php');
```

要调用函数 someName()，可使用该函数的名称并附上参数。

```
echo someName('TEST'); // 返回字符串"INIT and also TEST"
```

可以使用一个或两个参数调用函数 someOtherName()：

```
echo someOtherName(1); // 返回 1
echo someOtherName(1, 1); // 返回 2
```

someInfinite()函数能够接收无数（或任意数量）个参数。下面是一些调用该函数的例子：

```
echo someInfinite(1, 2, 3);
echo PHP_EOL;
echo someInfinite(22.22, 'A', ['a' => 1, 'b' => 2]);
```

下面是调用前面介绍的函数所获得的结果：

```
aed@aed: ~/Repos/php7_recipes/source/chapter03
--------------------------------
OUTPUT FROM: someName()
--------------------------------
INIT and also TEST

--------------------------------
OUTPUT FROM: someOtherName()
--------------------------------
1
2

--------------------------------
OUTPUT FROM: someInfinite()
--------------------------------
array (
  0 => 1,
  1 => 2,
  2 => 3,
)
array (
  0 => 22.219999999999999,
  1 => 'A',
  2 =>
  array (
    'a' => 1,
    'b' => 2,
  ),
)
:
```

提示数据类型

在开发函数时，很可能出现在其他项目中重用以前开发过的函数库的情况。如果你是在一个团队中工作，团队中的其他开发者也可能会使用你编写的函数。为了规范化地使用不同开发人员编写的函数，最好使用**类型提示**（type hint）功能，这包括指明你编写的函数需要接收哪种数据类型的参数。

具体处理过程

1. 可以在函数的参数前面添加类型提示前缀。下面的类型提示既可以在 PHP 5 中使用，也可以在 PHP 7 中使用：

- Array
- Class
- Callable

2. 如果在调用函数时将参数的数据类型设置错了，PHP 引擎就会抛出 TypeError 异常。下面的示例函数需要接收的参数类型分别为数组、DateTime 类的实例和匿名函数：

```
function someTypeHint(Array $a, DateTime $t, Callable $c)
{
  $message = '';
  $message .= 'Array Count: ' . count($a) . PHP_EOL;
  $message .= 'Date: ' . $t->format('Y-m-d') . PHP_EOL;
  $message .= 'Callable Return: ' . $c() . PHP_EOL;
  return $message;
}
```

无须为每个参数都提供类型提示。只有当使用错误的参数类型会对函数处理过程产生负面影响时，才应该使用这个技巧。例如，假设你在函数中使用了一个 foreach() 循环，如果你为该函数提供的参数不是数组或其他可实现 Traversable 接口的数据类型，那么就会产生错误。

3. 在 PHP 7 中，如果使用了适当的 declare() 语句，那么就可以使用**标量**（即整型、浮点型、布尔型和字符型）类型提示。下面通过示例介绍具体步骤。在含有你想要添加标量类型提示的函数的库文件顶部，紧跟 PHP 开始标记添加下面的 declare() 语句：

```
declare(strict_types=1);
```

4. 这样就可以在函数中添加标量类型提示了：

```
function someScalarHint(bool $b, int $i, float $f, string $s)
{
  return sprintf("\n%20s : %5s\n%20s : %5d\n%20s " .
                 ": %5.2f\n%20s : %20s\n\n",
                 'Boolean', ($b ? 'TRUE' : 'FALSE'),
                 'Integer', $i,
```

```
                'Float', $f,
                'String', $s);
}
```

5. 在 PHP 7 中，如果已经声明了严格的类型提示，那么布尔型类型提示与其他三种标量类型（即整型、浮点型和字符型）提示的作用略有不同。可以将任何标量数据类型用作参数，而且程序也不会抛出 TypeError 异常！然而，这些参数一旦被传入函数，就会被自动转换为布尔型数据。如果传输的是标量数据类型之外的数据类型（即数组或对象），那么程序就会抛出 TypeError 异常。下面的示例定义了布尔型的参数，注意，返回值会被自动转换为布尔型：

```
function someBoolHint(bool $b)
{
  return $b;
}
```

具体运行过程

将三个函数（someTypeHint()、someScalarHint()和 someBoolHint()）放入将要被包含进来的独立文件中。本例将该文件命名为 chap_03_developing_functions_type_hints_ library.php。不要忘记将 declare(strict_ types=1) 语句放在该文件的顶部！

本例使用下面的代码包含这个文件：

```
include (__DIR__ . DIRECTORY_SEPARATOR . 'chap_03_developing_
functions_type_hints_library.php');
```

要测试 someTypeHint()函数，可调用该函数两次，第一次使用数据类型正确的参数，第二次使用数据类型错误的参数。这会使程序抛出 TypeError 异常，因此需要将调用该函数的语句放在 try {...} catch(){...}代码块中：

```
try {
   $callable = function () { return 'Callback Return'; };
   echo someTypeHint([1,2,3], new DateTime(), $callable);
   echo someTypeHint('A', 'B', 'C');
} catch (TypeError $e) {
    echo $e->getMessage();
    echo PHP_EOL;
}
```

如本节末尾的输出结果所示，当使用数据类型正确的参数时不会出问题，而当使用

数据类型错误的参数时，程序就会抛出 TypeError 异常。

在 PHP 7 中，某些错误已经被转换为 Error 类，Error 对象被处理的方式与 Exception 对象类似。这意味着你可以在自己编写的程序中捕捉 Error 对象。TypeError 是 Error 类的一个子类。当数据类型不正确的参数被传输给函数时，程序就会抛出 TypeError 对象。

在 PHP 7 中，就像 Exception 类一样，所有的 Error 类都实现了 Throwable 接口。如果你无法确定你需要捕捉 Error 对象还是 Exception 对象，可以添加捕捉带 Throwable 接口的对象的代码块。

要测试 someScalarHint() 函数，可使用数据类型正确和错误的参数分别调用该函数，将调用该函数的语句放在 try {...}catch(){...}代码块中：

```
try {
     echo someScalarHint(TRUE, 11, 22.22, 'This is a string');
     echo someScalarHint('A', 'B', 'C', 'D');
} catch (TypeError $e) {
     echo $e->getMessage();
}
```

正如我们预想的那样，第一次调用该函数运行正常，第二次调用该函数会使程序抛出 TypeError 异常。

当设置布尔型的类型提示时，任何传输给函数的标量参数都不会使程序抛出 TypeError 异常。更确切地说，这些值会被转换为等价的布尔型数据。如果你使函数返回了这些值，那么这些值的数据类型就会变为布尔型。

要验证该结论，可调用前面介绍过的 someBoolHint() 函数，并为它的参数添加标量数据类型提示。var_dump() 方法展示了 someBoolHint() 函数的返回值都是布尔型的：

```
try {
     // 使用第一批数据类型 positive 的参数和标量参数获得的结果
     $b = someBooleanHint(TRUE);
     $i = someBooleanHint(11);
     $f = someBooleanHint(22.22);
     $s = someBooleanHint('X');
     var_dump($b, $i, $f, $s);
     // 使用第二批数据类型 negative 的参数和标量获得的结果
```

```
    $b = someBooleanHint(FALSE);
    $i = someBooleanHint(0);
    $f = someBooleanHint(0.0);
    $s = someBooleanHint(' ');
    var_dump($b, $i, $f, $s);
} catch (TypeError $e) {
  echo $e->getMessage();
}
```

如果你使用相同的函数调用语句，但使用了非标量数据类型的参数，那么程序就会抛出 TypeError 异常：

```
try {
  $a = someBoolHint([1,2,3]);
  var_dump($a);
} catch (TypeError $e) {
  echo $e->getMessage();
}
try {
    $o = someBoolHint(new stdClass());
    var_dump($o);
} catch (TypeError $e) {
    echo $e->getMessage();
}
```

下面是全部输出结果：

扩展

PHP 7.1 引入了新的类型提示 iterable，该类型提示允许将参数的数据类型设置为数据、迭代器或生成器。要详细了解这方面的信息请浏览：

- https://wiki.php.net/rfc/iterable

要了解标量类型提示的基本原理，请浏览：

- https://wiki.php.net/rfc/scalar_type_hints_v5

设置函数返回值的数据类型

PHP 7 允许你指定函数返回值的数据类型。然而，与标量类型提示不同，这样做无须添加任何特殊的声明语句。

具体处理过程

1. 本例介绍设置函数返回值的数据类型的方式。要为函数的返回值设置数据类型，需要先定义函数。在圆括号的后面添加一个空格，然后再添加冒号和数据类型：

```
function returnsString(DateTime $date, $format) : string
{
  return $date->format($format);
}
```

PHP 7.1 引入了另一种设置函数返回值数据类型的功能：**可空值类型**（nullable type）。你需要做的仅是将 string 更改为？string。这就会使函数能够返回字符串或者 NULL。

2. 不论函数的返回值在函数内部拥有何种数据类型，只要它被函数返回了，就会被转换为已经声明的数据类型。注意，本例中$a、$b 和$c 变量的值都被加到一起求和，求和得到的值就是函数的返回值。通常，你会认为该返回值应该为数值型数据。然而，本例将返回值的数据类型声明为字符型，这重写了 PHP 的类型魔术处理过程（type-juggling process）：

```
function convertsToString($a, $b, $c) : string
```

```
  return $a + $b + $c;
}
```

3. 还可以将函数返回值的数据类型设置为某个类。本例将函数返回值的数据类型设置为 DateTime 类，该类属于 PHP 中的 DateTime 扩展：

```
function makesDateTime($year, $month, $day) : DateTime
{
  $date = new DateTime();
  $date->setDate($year, $month, $day);
  return $date;
}
```

也可以对 makesDateTime()函数使用标量数据类型提示。如果$year、$month 或$day 变量不是整型的，那么调用 setDate()函数时就会显示 Warning（警告）信息。如果你使用标量数据类型提示，并传输了数据类型错误的参数，那么程序就会抛出 TypeError 异常。尽管显示 Warning 信息和抛出 TypeError 异常并没有实质性的差别，但至少 TypeError 异常会让那些以错误方式使用你编写的代码的开发人员认真起来，并多加注意！

4. 如果一个函数已经被设置了返回值的数据类型，而且你在该函数中返回了数据类型错误的值，那么在运行程序时就会抛出 TypeError 异常。下面函数的返回值被设置为 DateTime 类，但返回的值是字符型的。程序会抛出 TypeError 异常，但只有当 PHP 引擎检测到这个错误时，该异常才会被抛出：

```
function wrongDateTime($year, $month, $day) : DateTime
{
  return date($year . '-' . $month . '-' . $day);
}
```

如果用于设置函数返回值数据类型的类不是 PHP 内置的类（即 SPL 中的类），应确保这个类已经被自动加载或包含了。

具体运行情况

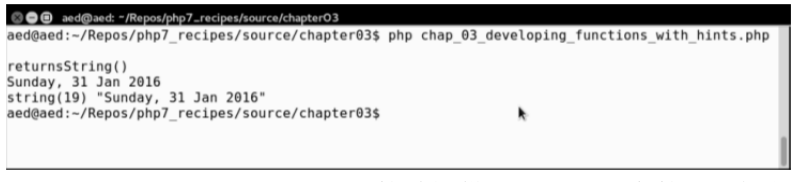
将前面介绍的函数放入名为 chap_03_developing_functions_return_types_library.php 的库文件中。还需要将该文件包含到 chap_03_developing_functions_return_types.php 脚本中，该脚本会调用这些函数：

```
include (__DIR__ . '/chap_03_developing_functions_return_types_
library.php');
```

现在就可以调用 returnsString()函数，将 DateTime 实例和一个带格式的字符串用作参数：

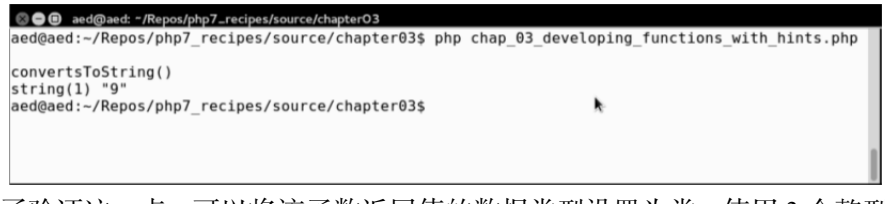
```
$date = new DateTime();
$format = 'l, d M Y';
$now = returnsString($date, $format);
echo $now . PHP_EOL;
var_dump($now);
```

正如我们预想的，输出结果会是一个字符串：



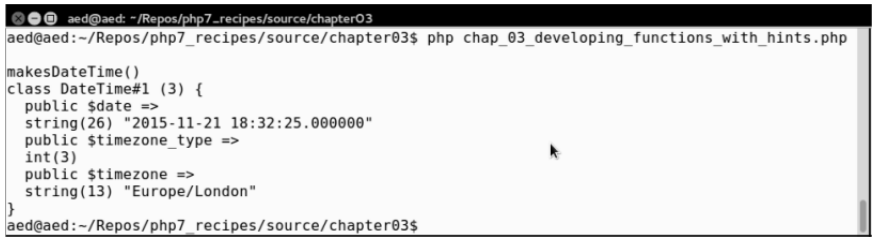
可以调用 convertsToString()函数并提供 3 个整型的参数。注意该函数的返回值也是字符型：

```
echo "\nconvertsToString()\n";
var_dump(convertsToString(2, 3, 4));
```



为了验证这一点，可以将该函数返回值的数据类型设置为类，使用 3 个整型参数调用 makesDateTime()函数：

```
echo "\nmakesDateTime()\n";
$d = makesDateTime(2015, 11, 21);
var_dump($d);
```



最后，使用 3 个整型参数调用 wrongDateTime()函数：

```
try {
    $e = wrongDateTime(2015, 11, 21);
    var_dump($e);
} catch (TypeError $e) {
    echo $e->getMessage();
}
```

注意，程序运行时会抛出 TypeError 异常：

补充说明

PHP 7.1 中增加了新的返回值类型：void。当你不希望函数返回任何值时，可使用该类型。要了解更多信息，请浏览：https://wiki.php.net/rfc/void_ return_ type。

扩展

要了解更多关于返回值声明的信息，请参考下面的文章：

- http://php.net/manual/en/functions.arguments.php#functions.arguments.type-declaration.Strict

- https://wiki.php.net/rfc/return_types

要详细了解可空值类型，请参考下面的文章：

- https://wiki.php.net/rfc/nullable_types

使用迭代器

迭代器是一种特殊类型的类，使用它可以**遍历**容器或列表。此处的关键是*遍历*。这意味着迭代器提供了访问列表中所有内容的方式，但它本身并不执行访问操作。

SPL 提供了许多能够遍历各种内容的通用和专用迭代器。例如，ArrayIterator 类就可以专门用于通过面向对象的方式遍历数组。DirectoryIterator 类专门用于扫描文件系统。

某些 SPL 迭代器专门用于辅助其他迭代器和增强其他迭代器的功能，如 FilterIterator 和 LimitIterator。使用 FilterIterator 对象可以从父迭代器中去除你不想要的值。使用 LimitIterator 对象能够限定遍历范围。

SPL 还提供了一系列递归迭代器，从而能够反复调用父迭代器，如 RecursiveDirectoryIterator 类。使用 RecursiveDirectoryIterator 对象可以扫描目录树中的全部内容。

具体处理过程

1. 我们先使用 ArrayIterator 类做实验。这个类的使用方式非常简单。你需要做的仅是为该构造器提供数组类型的参数。做到这一点后，你就可以使用基于 SPL 的迭代器的任何标准方法，如 current()、next()等。

```
$iterator = new ArrayIterator($array);
```

使用 ArrayIterator 对象可以将 PHP 中的标准数组转换为迭代器。从某种意义上讲，这在过程化程序设计和 OOP 之间搭起了一座桥梁。

2. 下面介绍一个使用迭代器的实用范例。这个函数会接收一个迭代器并生成一系列 HTML 中的<ul>和<li>标签：

```
function htmlList($iterator)
```

```
{
  $output = '<ul>';
  while ($value = $iterator->current()) {
    $output .= '<li>' . $value . '</li>';
    $iterator->next();
  }
  $output .= '</ul>';
  return $output;
}
```

3. 还可以将 ArrayIterator 实例封装到 foreach() 循环中：

```
function htmlList($iterator)
{
  $output = '<ul>';
  foreach($iterator as $value) {
    $output .= '<li>' . $value . '</li>';
  }
  $output .= '</ul>';
  return $output;
}
```

4. 使用 CallbackFilterIterator 实例为已经存在的迭代器添加值，是一种很好的处理方式。这使你能够封装任何已经存在的迭代器并显示它们的输出结果。本例定义 fetchCountryName() 函数，该函数会遍历一条能产生一组国家名称的数据库查询命令。先通过使用 Application\Database\Connection 类（第 1 章介绍过的）执行的查询操作，定义一个 ArrayIterator 实例：

```
function fetchCountryName($sql, $connection)
{
  $iterator = new ArrayIterator();
  $stmt = $connection->pdo->query($sql);
  while($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    $iterator->append($row['name']);
  }
  return $iterator;
}
```

5. 定义一个起过滤作用的方法 nameFilterIterator()，该方法会将部分国家名称和 ArrayIterator 实例作为其参数来接收：

```
function nameFilterIterator($innerIterator, $name)
{
```

```
    if (!$name) return $innerIterator;
    $name = trim($name);
    $iterator = new CallbackFilterIterator($innerIterator,
      function($current, $key, $iterator) use ($name) {
        $pattern = '/' . $name . '/i';
        return (bool) preg_match($pattern, $current);
      }
    );
    return $iterator;
}
```

6. LimitIterator 对象能够为应用程序添加基本的页码。要使用该迭代器，只需提供父迭代器、偏移量和限定范围。这样 LimitIterator 实例仅会在偏移量设定的起始位置上，从整个数据集合中提取一个子集。我们使用步骤 2 中介绍过的示例程序，为数据库查询操作的输出结果分页。要做到这一点非常简单，只需封装 LimitIterator 实例中 fetchCountryName()方法生成的迭代器：

```
$pagination = new LimitIterator(fetchCountryName(
$sql, $connection), $offset, $limit);
```

在使用 LimitIterator 对象时应小心谨慎。为了获取限定范围中的结果，该实例会通过将整个数据集合都加载到内存中来遍历这些内容。因此，在遍历较大的数据集合时，这不是一种好工具。

7. 迭代器是可以分层的。在本例中，ArrayIterator 对象是由 FilterIterator 对象处理的，而 FilterIterator 实例又是由 LimitIterator 实例限定的。下面先设置一个 ArrayIterator 实例：

```
$i = new ArrayIterator($a);
```

8. 然后将 ArrayIterator 实例插入到 FilterIterator 实例中。注意，我们将使用 PHP 7 新增的匿名类功能。在本例中，匿名类扩展了 FilterIterator 类并重写了 accept()方法，使该方法仅接收 ASCII 码为偶数的字母：

```
$f = new class ($i) extends FilterIterator {
  public function accept()
  {
    $current = $this->current();
    return !(ord($current) & 1);
```

```
  }
};
```

9. 将 FilterIterator 实例作为参数提供给 LimitIterator 实例，并提供一个偏移量（本例选 2）和一个限定范围（本例将其设置为 6）：

```
$l = new LimitIterator($f, 2, 6);
```

10. 这样我们就可以通过定义一个简单的函数来显示输出结果，并轮流调用每个迭代器，以查看由 range('A', 'Z')数组生成的简单数组中的结果：

```
function showElements($iterator)
{
  foreach($iterator as $item) echo $item . ' ';
  echo PHP_EOL;
}

$a = range('A', 'Z');
$i = new ArrayIterator($a);
showElements($i);
```

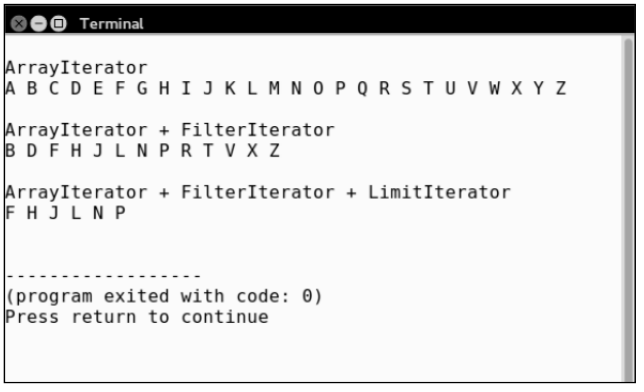
11. 下面是这个程序的另一个更改版本，它通过将 FilterIterator 实例放置在 ArrayIterator 实例上面，从而获得 ASCII 码为奇数的字母：

```
$f = new class ($i) extends FilterIterator {
public function accept()
  {
    $current = $this->current();
    return !(ord($current) & 1);
  }
};
showElements($f);
```

12. 下面是另一个版本，该程序仅会获取字符 F、H、J、L、N 和 P，它演示了另一种层次结构，即由 LimitIterator 对象处理 FilterIterator 对象，而由 FilterIterator 对象处理 ArrayIterator 对象。

```
$l = new LimitIterator($f, 2, 6);
showElements($l);
```

下面是三个示例程序的输出结果：



13. 让我们回到生成国家名称列表的示例。如果我们不仅想获得国家名称，还想遍历含国家名称和 ISO 代码的多维数组，应该怎样做呢？前面介绍的简单迭代器不足以完成该任务。更确切地说，我们需要使用**递归**迭代器。

14. 我们需要定义前面介绍过的数据库连接类的方法，以便从数据库获取所有字段。像之前一样，返回含有通过执行查询操作获得的数据的 `ArrayIterator` 实例：

```
function fetchAllAssoc($sql, $connection)
{
  $iterator = new ArrayIterator();
  $stmt = $connection->pdo->query($sql);
  while($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    $iterator->append($row);
  }
  return $iterator;
}
```

15. 乍看之下，简单地将标准 `ArrayIterator` 实例封装在 `RecursiveArrayIterator` 实例中是不错的选择。令人遗憾的是，这种方式仅会执行**浅**迭代操作，而且无法获得我们想要的结果：遍历含有数据库查询结果的多维数据中的所有元素。

```
$iterator = fetchAllAssoc($sql, $connection);
$shallow = new RecursiveArrayIterator($iterator);
```

16. 尽管这样做可以通过数据库查询操作获得以记录为单位的数组元素，但我们希望的是遍历查询结果中的每条记录中的每个字段。为了做到这一点，需要使用 `RecursiveIteratorIterator` 类拆分每条记录。

17. 虽然这个类的名称有点怪异（含有两个 `Iterator`），但它还是会使用前面介绍过的 `RecursiveArrayIterator` 类增加工作量，通过执行**深度**迭代操作，遍历数

组中的各个层面：

```
$deep = new RecursiveIteratorIterator($shallow);
```

具体运行情况

为了进行实践练习，你可以使用迭代器开发实现过滤和分页功能的测试脚本。可将这些测试代码存储在 chap_03_developing_functions_filtered_and_paginated.php 文件中。

首先根据最佳编程惯例，将前面介绍的函数放在名为 chap_03_developing_functions_iterators_library.php 的库文件中。应确保在测试脚本中包含了该文件。

源数据是一个名为 iso_country_codes 的表，其中含有 ISO2、ISO3 和国家的名称。执行连接数据库操作的类位于 config/db.config.php 文件中。还可以将上一章介绍的 Application\Database\Connection 类包含到测试脚本中：

```
define('DB_CONFIG_FILE', '/../config/db.config.php');
define('ITEMS_PER_PAGE', [5, 10, 15, 20]);
include (__DIR__ . '/chap_03_developing_functions_iterators_library.
php');
include (__DIR__ . '/../Application/Database/Connection.php');
```

在 PHP 7 中，可以将常量定义为数组。在本例中，常量 ITEMS_PER_PAGE 被定义为数组，并被用于生成 HTML 网页文件中的 SELECT 元素。

然后，处理代表国家名称和每页记录条数的输入参数。将当前起始页设置为 0，并且设置页码的增量（翻到下一页）和减量（翻到上一页）：

```
$name   = strip_tags($_GET['name'] ?? '');
$limit  = (int) ($_GET['limit'] ?? 10);
$page   = (int) ($_GET['page'] ?? 0);
$offset = $page * $limit;
$prev   = ($page > 0) ? $page - 1 : 0;
$next   = $page + 1;
```

这样就可以连接数据库并运行一条简单的 SELECT 查询命令了。应该将这些代码放在 try{}catch{}代码块中。因而应该将嵌套在一起的迭代器放在 try{}代码块中：

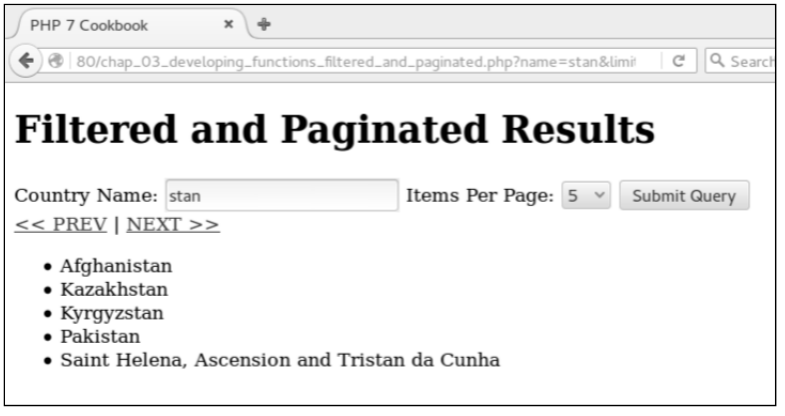
```
try {
```

```
    $connection = new Application\Database\Connection(
      include __DIR__ . DB_CONFIG_FILE);
      $sql = 'SELECT * FROM iso_country_codes';
      $arrayIterator = fetchCountryName($sql, $connection);
      $filteredIterator = nameFilterIterator($arrayIterator, $name);
      $limitIterator = pagination(
      $filteredIterator, $offset, $limit);
} catch (Throwable $e) {
    echo $e->getMessage();
}
```

这样我们就可以编写 HTML 代码了。在这个简单的例子中，我们会制作一个表单，使用户能够设置在每个页面中显示国家名称的数量并查看它们：

```
<form>
  Country Name:
  <input type="text" name="name"
         value="<?= htmlspecialchars($name) ?>">
  Items Per Page:
  <select name="limit">
    <?php foreach (ITEMS_PER_PAGE as $item) : ?>
      <option<?= ($item == $limit) ? ' selected' : '' ?>>
      <?= $item ?></option>
    <?php endforeach; ?>
  </select>
  <input type="submit" />
</form>
  <a href="?name=<?= $name ?>&limit=<?= $limit ?>
    &page=<?= $prev ?>">
  << PREV</a>
  <a href="?name=<?= $name ?>&limit=<?= $limit ?>
    &page=<?= $next ?>">
  NEXT >></a>
<?= htmlList($limitIterator); ?>
```

下面是输出结果：



为了测试对含有国家信息的数据库执行递归式迭代操作，还需要包含迭代器的库文件和 `Application\Database\Connection` 类：

```
define('DB_CONFIG_FILE', '/../config/db.config.php');
include (__DIR__ . '/chap_03_developing_functions_iterators_library.
php');
include (__DIR__ . '/../Application/Database/Connection.php');
```

像前面一样，应该将查询数据库的语句封装在 `try {} catch {}`代码块中。这样就可以在 `try {}`代码块中测试递归式迭代操作：

```
try {
     $connection = new Application\Database\Connection(
     include __DIR__ . DB_CONFIG_FILE);
     $sql = 'SELECT * FROM iso_country_codes';
     $iterator = fetchAllAssoc($sql, $connection);
     $shallow = new RecursiveArrayIterator($iterator);
     foreach ($shallow as $item) var_dump($item);
     $deep = new RecursiveIteratorIterator($shallow);
     foreach ($deep as $item) var_dump($item);
} catch (Throwable $e) {
     echo $e->getMessage();
}
```

下面是 `RecursiveArrayIterator` 实例的输出结果：

```
aed@aed: ~/Repos/php7_recipes/source/chapter03

RecursiveArrayIterator
array(5) {
  'name' =>
  string(11) "Afghanistan"
  'iso2' =>
  string(2) "AF"
  'iso3' =>
  string(3) "AFG"
  'iso_numeric' =>
  string(1) "4"
  'iso_3166' =>
  string(13) "ISO 3166-2:AF"
}
array(5) {
  'name' =>
  string(7) "Albania"
  'iso2' =>
  string(2) "AL"
  'iso3' =>
  string(3) "ALB"
  'iso_numeric' =>
  string(1) "8"
  'iso_3166' =>
  string(13) "ISO 3166-2:AL"
}
:
```

下面是使用 `RecursiveIteratorIterator` 对象后获得的输出结果：

```
aed@aed: ~/Repos/php7_recipes/source/chapter03

RecursiveIteratorIterator
string(11) "Afghanistan"
string(2) "AF"
string(3) "AFG"
string(1) "4"
string(13) "ISO 3166-2:AF"
string(7) "Albania"
string(2) "AL"
string(3) "ALB"
string(1) "8"
string(13) "ISO 3166-2:AL"
string(10) "Antarctica"
string(2) "AQ"
string(3) "ATA"
string(2) "10"
string(13) "ISO 3166-2:AQ"
string(7) "Algeria"
string(2) "DZ"
string(3) "DZA"
string(2) "12"
string(13) "ISO 3166-2:DZ"
string(14) "American Samoa"
string(2) "AS"
string(3) "ASM"
string(2) "16"
:
```

使用生成器编写自己的迭代器

前面的示例介绍了 PHP 7 中 SPL 迭代器的用法。但如果这些标准迭代器无法满足特定项目的需求，应该怎么办呢？一种解决方案是编写一个函数，通过迭代方式使用 yield 关键字以递增方式返回值（而不是通过创建数组返回值）。该函数被称为**生成器**。事实上，后台的 PHP 引擎会自动将你编写的函数转换为名为 `Generator` 的特殊内置类。

这种方式有多个优点。其中一个主要的优点是可以遍历较大的容器（即解析含有大量数据的文件）。传统处理方式是先创建一个数组，返回该数组中存储的值，但这种处理方式会占用双倍的内存容量，而且性能也比较差（只有当获得了最后一个数组时，才能得到结果）。

具体处理过程

1. 本例会创建基于迭代器的函数库，添加我们自己编写的生成器。本例会复制上一节介绍的功能，将 `ArrayIterator`、`FilterIterator` 和 `LimitIterator` 实例嵌套起来。

2. 因为我们需要访问源数据数组、合适的过滤器、页码和每个页面显示的条目数，所以应该在 `filteredResultsGenerator()` 函数中包含适当的参数。这样就可以根据页码和限定范围（即每个页面显示的条目数）计算偏移量。然后遍历数组，应用过滤器，并在没有到达偏移量限定值的情况下继续执行循环操作，以及在到达偏移量限定值时停止执行循环操作：

```
function filteredResultsGenerator(array $array, $filter,
                                  $limit = 10, $page = 0)
  {
    $max = count($array);
    $offset = $page * $limit;
    foreach ($array as $key => $value) {
      if (!stripos($value, $filter) !== FALSE) continue;
      if (--$offset >= 0) continue;
      if (--$limit <= 0) break;
      yield $value;
  }
}
```

3. 你会发现这个函数与其他函数的主要区别在于 yield 关键字。这个关键字的作用是命令 PHP 引擎生成 Generator 实例并将这段代码封装起来。

具体运行情况

为了了解 filteredResultsGenerator() 函数的用法，我们将实现一个网页应用程序，该程序会扫描网页，并通过扫描 HREF 属性生成经过过滤和分页的 URL 列表。

首先需要将 filteredResultsGenerator() 函数添加到上一节介绍过的库文件中，然后将前面介绍过的几个函数添加到 chap_03_developing_functions_iterators_library.php 库文件中。

然后定义测试脚本 chap_03_developing_functions_using_generator.php，使该脚本包含这两个函数库和第 1 章介绍的用于定义 Application\Web\Hoover 类的文件：

```
include (__DIR__ . DIRECTORY_SEPARATOR . 'chap_03_developing_
functions_iterators_library.php');
include (__DIR__ . '/../Application/Web/Hoover.php');
```

此时你需要收集用户输入的数据，以判定需要扫描哪些 URL、将哪个字符串用作过滤器、每个页面显示多少条目和当前的页码。

```
$url    = trim(strip_tags($_GET['url'] ?? ''));
$filter = trim(strip_tags($_GET['filter'] ?? ''));
$limit  = (int) ($_GET['limit'] ?? 10);
$page   = (int) ($_GET['page'] ?? 0);
```

空合并运算符（??）最适合通过网页获取输入数据。在没有定义的情况下，该运算符不会生成任何警告。如果参数没有收到用户输入的数据，你可以为其设置一个默认值。

最佳编程习惯

网页安全应该永远放在第一位。在本例中，你可以使用 strip_tags() 函数，并且为了规范化用户输入的数据，还可以将数据类型强制规定为整型（int）。

接下来，定义在分页列表的上一页和下一页链接中使用的变量。注意，还应该创建是否到达数据集合末尾的检查，以确保下一页没有超出数据集合的末尾。为简便起见，本例没有使用是否超出末尾检查：

```
$next = $page + 1;
$prev = $page - 1;
$base = '?url=' . htmlspecialchars($url)
      .'&filter=' . htmlspecialchars($filter)
      .'&limit=' . $limit
      .'&page=';
```

现在需要创建一个 Application\Web\Hoover 实例，并从目标 URL 中提取出 HREF 属性：

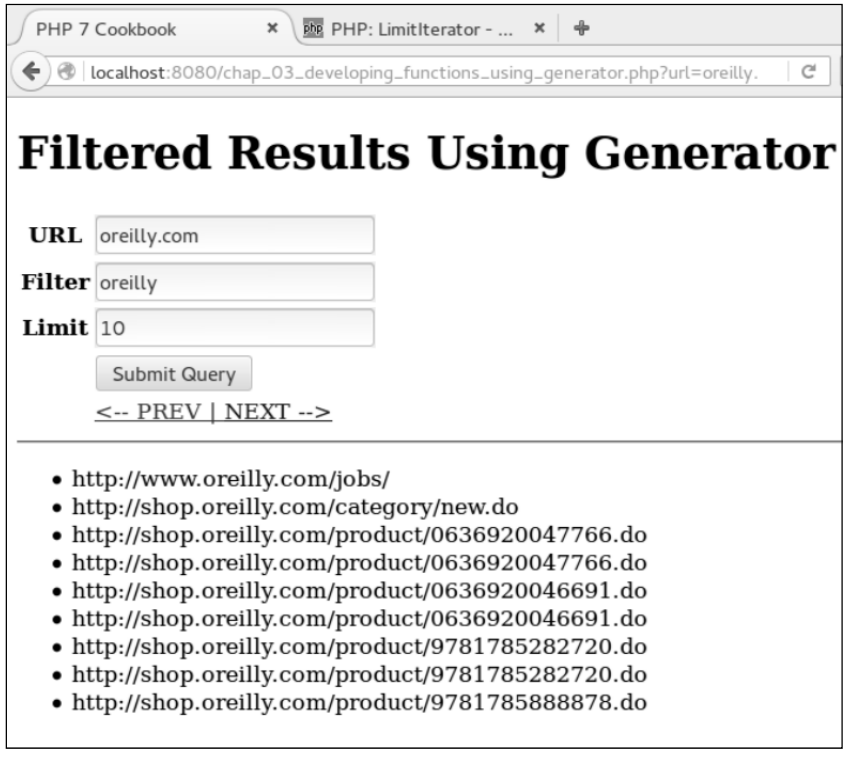
```
$vac  = new Application\Web\Hoover();
$list = $vac->getAttribute($url, 'href');
```

最后定义 HTML 输出结果，生成用于输入数据的表单，并通过前面介绍的 htmlList()函数运行我们编写的生成器：

```
<form>
<table>
<tr>
<th>URL</th>
<td>
<input type="text" name="url"
  value="<?= htmlspecialchars($url) ?>"/>
</td>
</tr>
<tr>
<th>Filter</th>
<td>
<input type="text" name="filter"
  value="<?= htmlspecialchars($filter) ?>"/></td>
</tr>
<tr>
<th>Limit</th>
<td><input type="text" name="limit" value="<?= $limit ?>"/></td>
</tr>
<tr>
<th> </th><td><input type="submit" /></td>
</tr>
<tr>
```

```
<td> </td>
<td>
<a href="<?= $base . $prev ?>"><-- PREV |
<a href="<?= $base . $next ?>">NEXT --></td>
</tr>
</table>
</form>
<hr>
<?= htmlList(filteredResultsGenerator(
$list, $filter, $limit, $page)); ?>
```

下面是输出结果：



第 4 章　PHP 中的面向对象编程功能

本章包括以下要点：

- 开发类
- 扩展类
- 使用静态属性和方法
- 使用命名空间
- 定义可见性
- 使用接口
- 使用特性（trait）
- 实现匿名类

本章主要内容简介

本章介绍 PHP 7.0、PHP 7.1 及以上版本中的**面向对象编程**（OOP）功能。PHP 5.6 中也含有大多数 PHP 7.0 及以上版本提供的 OOP 功能。PHP 7.0 中引入的新功能是**匿名类**。在 PHP 7.1 中，你可以更改类常量的可见性。

另一种引起质变的新增功能是**捕捉**特定类型的错误。第 13 章将详细介绍这部分内容。

开发类

传统的开发方式是将类放在其本身的文件中。通常，类中含有实现某个独立目标的逻辑。可以进一步将类分解为自包含函数，这类函数也称为**方法**。类的内部定义的变量称为**属性**。建议你现在就编写一个测试类，第 13 章详细介绍了这方面内容。

具体处理过程

1. 创建用于保存类定义的文件。为了能够自动加载该类，最好为该文件取与类名称相同的文件名。在该文件的顶部，在关键字 class 前面添加**文档性注释**（DocBlock）。然后就可以为这个类定义属性和方法了。本例定义了 Test 类。该类有$test 属性和 getTest()方法：

```
<?php
declare(strict_types=1);
/**
 * 这是一个示例类。
 *
 * 这个类的作用是获取和设置
 * 受保护的属性（即私有变量）$test
 *
 */
class Test
{

  protected $test = 'TEST';

  /**
   * 该方法会返回变量$test 的当前值
   *
   * @返回变量$test 中的值，如果该值不是字符型的，会将其转换为字符型
   */
  public function getTest() : string
  {
    return $this->test;
  }
  /**
   * 这个方法会设置变量$test 的值
   *
   * @使参数$test 仅接收字符型值
   * @返回 Test 实例的值
   */
  public function setTest(string $test)
  {
    $this->test = $test;
    return $this;
  }
}
```

最佳编程习惯

在保存类的文件的名称中包含类的名称，是一种最佳编程习惯。尽管 PHP 中的类名称是不区分大小写的，但公认的最佳编程习惯是将类名称中的第一个字母大写。此外，不应该在类的定义文件中放置可执行代码。

每个类都应该在关键字 `class` 前面添加 DocBlock 注释。在 DocBlock 注释中，应该添加关于该类的作用的简短介绍。空一行后，再添加更详细的介绍。还可以在 DocBlock 注释中添加@标签，如`@author`、`@license` 等等。同理，每个方法的前面也应该添加介绍该方法的用途（及该方法接收的参数和返回的值）的 DocBlock 注释。

2. 可以在一个文件中定义一个以上的类，但这并非最佳编程习惯。本例创建了一个文件（`NameAddress.php`），该文件中含有两个类（`Name` 和 `Address`）：

```
<?php
declare(strict_types=1);
class Name
{

  protected $name = ' ';

  public function getName() : string
  {
    return $this->name;
  }

  public function setName(string $name)
  {
    $this->name = $name;

    return $this;
  }
}

class Address
{

  protected $address = ' ';
```

```
  public function getAddress() : string
  {
    return $this->address;
  }

  public function setAddress(string $address)
  {
    $this->address = $address;
    return $this;
  }
}
```

尽管可以在一个文件中定义不止一个类（如上面的例子所示），但这不是最佳编程习惯。因为，这不仅会在文件中造成逻辑混乱，还会使实现类自动加载的操作变得更加困难。

3. 类的名称是不区分大小写的。相同的类名称会导致错误。本例在文件 TwoClass.php 中定义了两个类（TwoClass 和 twoclass）：

```
<?php
class TwoClass
{
  public function showOne()
  {
    return 'ONE';
  }
}

// 当第二个类定义被解析时，会出现致命错误
class twoclass
{
  public function showTwo()
  {
    return 'TWO';
  }
}
```

4. PHP 7.1 处理了在使用关键字$this 时的不一致操作。尽管在 PHP 7.0 和 PHP 5.x

中允许将$this变量用作下列角色，但如果以这样的方式在PHP 7.1中使用$this变量，那么就会产生错误：

- 参数
- 静态变量
- 全局变量
- 在try…catch代码块中使用的变量
- 在foreach()循环中使用的变量
- unset()函数的参数
- 字符串转换变量（如$a = 'this'; echo $$a）
- 间接引用

5. 如果你不想为了创建实例而严谨地创建一个类，那么可以使用 PHP 内置的通用类 stdClass。stdClass 类能够使你极快速地定义属性，而无须严谨地通过扩展stdClass类定义新的类：

```
$obj = new stdClass();
```

6. 该功能可以在PHP程序中的许多地方使用。例如，当你使用PHP数据对象（PDO）执行数据库查询操作时，数据提取模式之一是 PDO::FETCH_OBJ。这种模式会返回stdClass实例，该stdClass实例中的属性代表数据库中表的字段：

```
$stmt = $connection->pdo->query($sql);
$row  = $stmt->fetch(PDO::FETCH_OBJ);
```

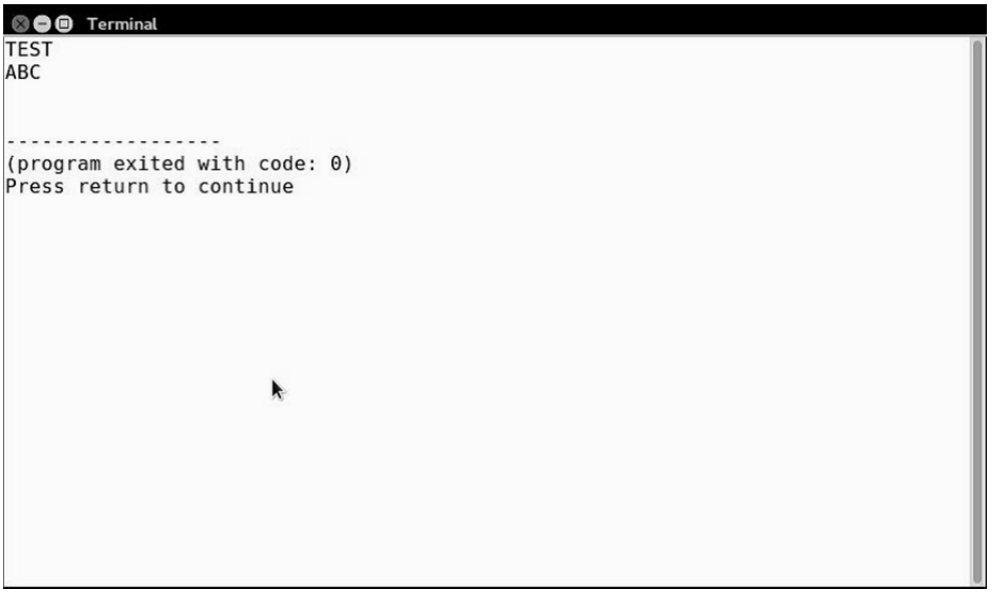
具体运行情况

将前面介绍的 Test 类放入 Test.php 文件中。创建另一个文件并将之命名为chap_04_oop_defining_class_test.php，在其中添加下面的代码：

```
require __DIR__ . '/Test.php';

$test = new Test();
echo $test->getTest();
echo PHP_EOL;
$test->setTest('ABC');
echo $test->getTest();
echo PHP_EOL;
```

下面的输出结果展示了$test属性的初始值，然后显示了通过调用setTest()函

数为$test 属性设置的新值：

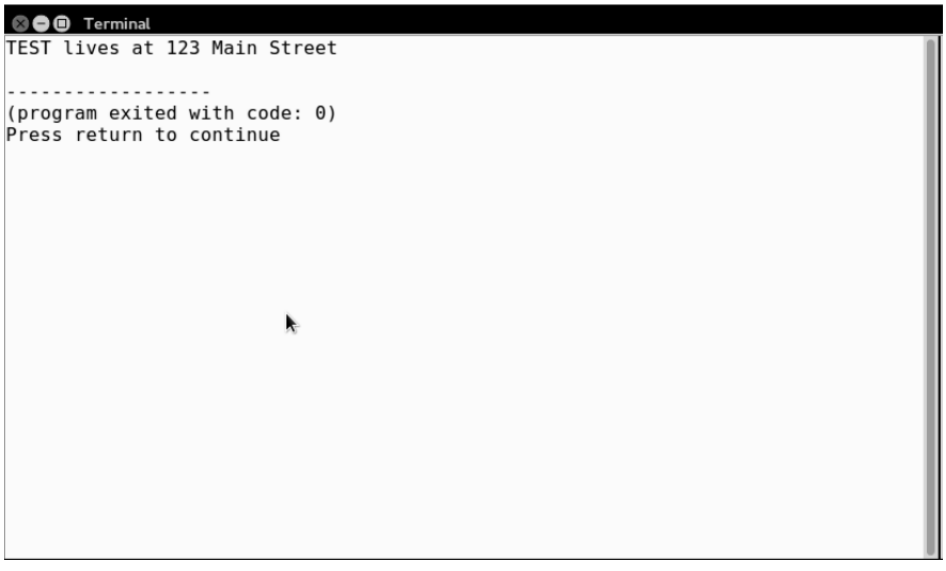


下面的例子在一个文件（NameAddress.php）中定义了两个类（Name 和 Address）。可使用下面的代码调用这两个类：

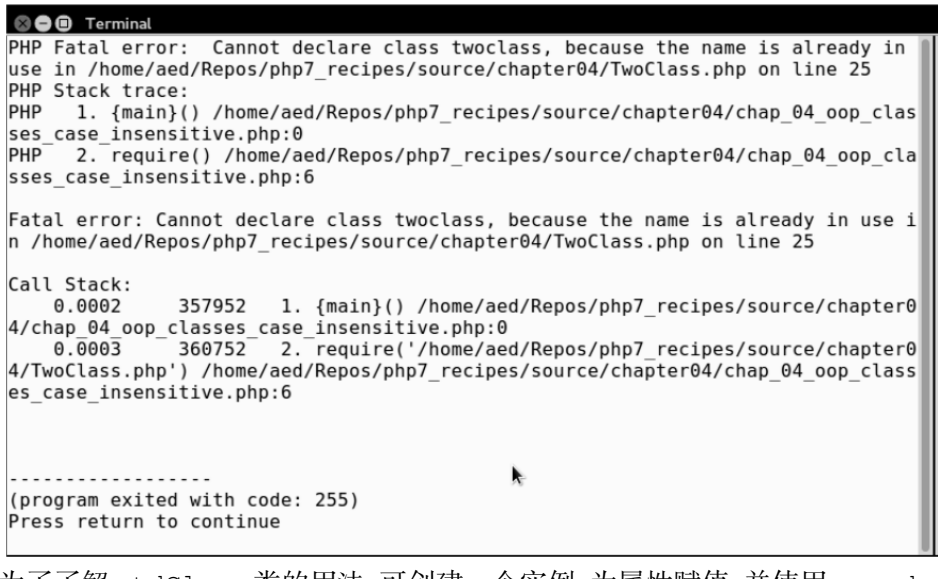
```
require __DIR__ . '/NameAddress.php';

$name = new Name();
$name->setName('TEST');
$addr = new Address();
$addr->setAddress('123 Main Street');

echo $name->getName() . ' lives at ' . $addr->getAddress();
```

尽管 PHP 解析程序没有因在一个文件中定义多个类而生成错误，但该文件中的逻辑纯净度已经受到了影响。而且文件的名称也与类的名称不同，这可能会影响执行类自动加载操作。

下图是这个例子的输出结果：



步骤 3 也展示了在一个文件中保存两个类定义的情况。但这样做的目的是展示 PHP 中的类名称是不区分大小写的。这些代码被放入一个文件（`TwoClass.php`），当你尝试包含这个文件时，就会出现错误，如下图所示：



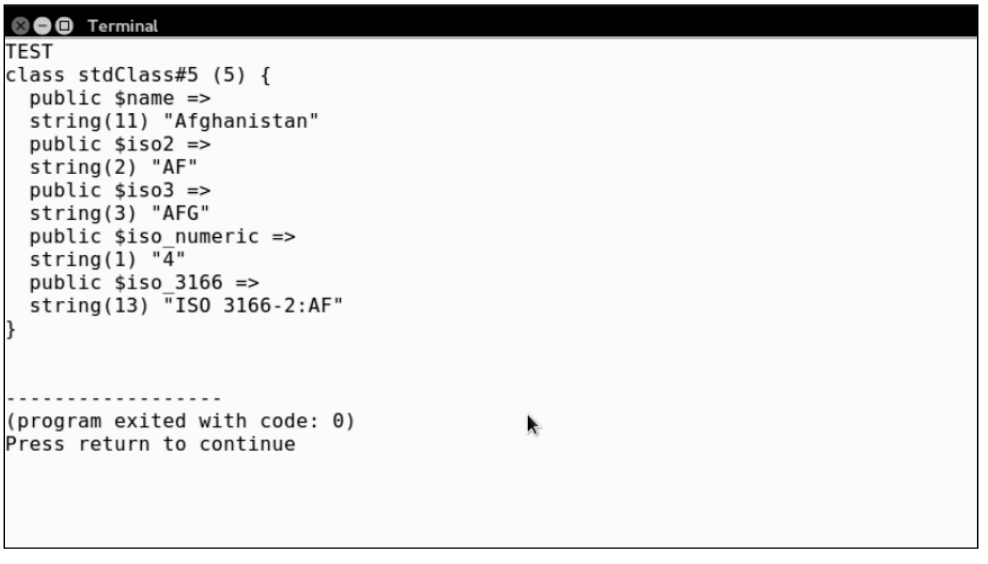
为了了解 `stdClass` 类的用法，可创建一个实例，为属性赋值，并使用 `var_dump()` 方法输出结果。要查看 `stdClass` 类内部的使用情况，可使用 `var_dump()` 方法输出

PDO 查询操作（数据提取模式为 FETCH_OBJ）的结果。

添加下面的代码：

```
$obj = new stdClass();
$obj->test = 'TEST';
echo $obj->test;
echo PHP_EOL;

include (__DIR__ . '/../Application/Database/Connection.php');
$connection = new Application\Database\Connection(
  include __DIR__ . DB_CONFIG_FILE);

$sql  = 'SELECT * FROM iso_country_codes';
$stmt = $connection->pdo->query($sql);
$row  = $stmt->fetch(PDO::FETCH_OBJ);
var_dump($row);
```

下面是输出结果：



扩展

要详细了解 PHP 7.1 对关键字$this 用法的改进，请浏览 https://wiki.php.net/rfc/this_var。

扩展类

开发者们使用 OOP 的主要原因之一是可以重复使用以前编写的代码，同时还可以添加或改写功能。在 PHP 中，关键字 extends 用于在类之间建立父/子关系。

具体处理过程

1. 在 child 类中，使用关键字 extends 设置继承关系。在本例中，Customer 类扩展了 Base 类。Customer 类的所有实例都会继承 Base 类的可见方法和属性（在本例中为$id 变量、getId()方法和 setId()方法）：

```
class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
}

class Customer extends Base
{
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}
```

2. 通过抽象化该类定义方法，可以让任何开发者使用你编写的类。本例通过抽象

Base 类定义了 validate() 方法。之所以必须将类抽象化，是因为从父类 Base 的观点看，无法确定子类需要具体执行哪些操作：

```
abstract class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
  public function validate();
}
```

如果某个类中含有**抽象方法**，那么就必须将该类声明为抽象类。

3. PHP 仅支持单一继承。下面的示例展示了继承（扩展）Customer 类的 Member 类，而 Customer 类继承（扩展）了 Base 类：

```
class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
}

class Customer extends Base
{
  protected $name;
  public function getName()
```

```
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}

class Member extends Customer
{
  protected $membership;
  public function getMembership()
  {
     return $this->membership;
  }
  public function setMembership($memberId)
  {
    $this->membership = $memberId;
  }
}
```

4. 只要父类满足某个数据类型设置，那么其任何一个子类都会满足该数据类型设置。下面示例中的 test() 函数，被设置为只能接收 Base 类类型的参数。处于 Base 类的继承关系中的所有类，都可以用作 test() 函数的参数，而将其他数据类型的数据用作 test() 函数的参数，会使程序抛出 TypeError 异常：

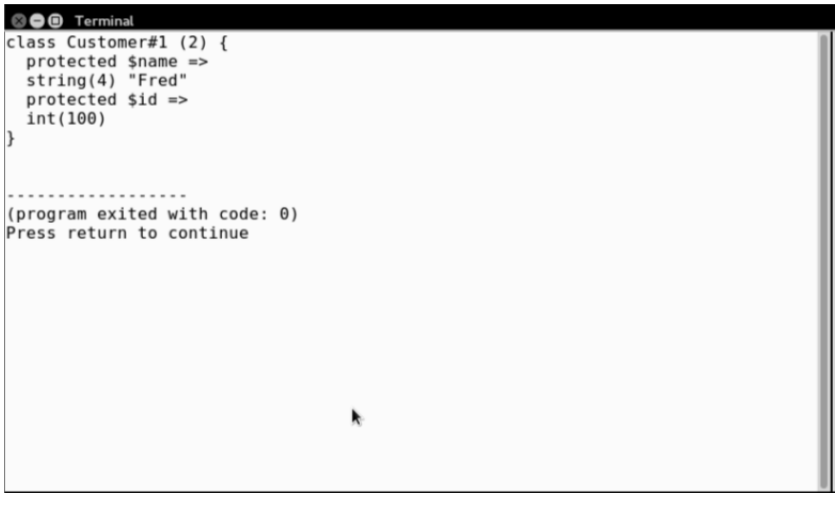
```
function test(Base $object)
{
  return $object->getId();
}
```

具体运行情况

先定义 Base 类和 Customer 类。为了进行对比，将这两个类放入一个文件（chap_04_oop_extends.php）中，然后添加下面的代码：

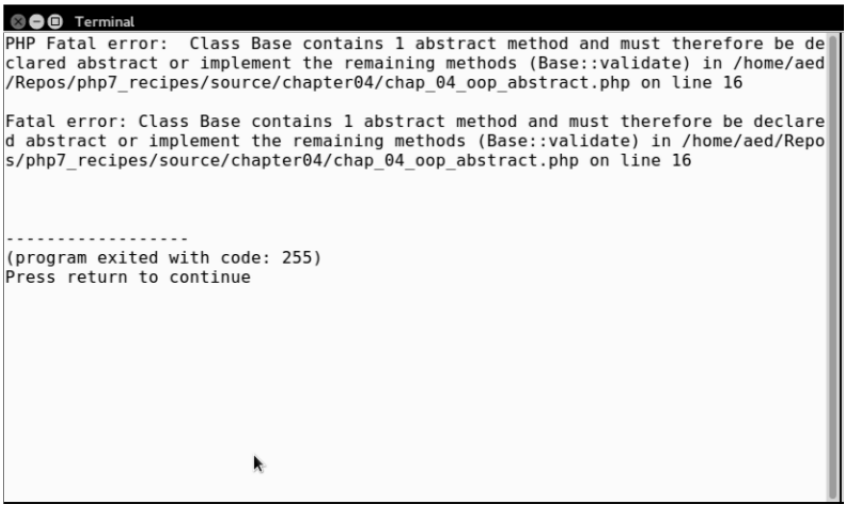
```
$customer = new Customer();
$customer->setId(100);
$customer->setName('Fred');
var_dump($customer);
```

注意，$id 属性、getId() 方法和 setId() 方法，是子类 Customer 从父类 Base 那里通过继承获得的：



为了理解抽象方法的用法，你可以为任意一个扩展 Base 类的类添加一些新功能。此处的难点在于，事先无法知道在这些子类中具体需要添加哪些功能。唯一能够确定的是肯定需要为这些类添加新功能。

使用上一节介绍的 Base 类并在其中添加一个新方法：validate()。将该方法设置为抽象方法，并且不为该方法编写任何执行具体操作的代码。请注意当子类 Customer 扩展父类 Base 时出现的情况。



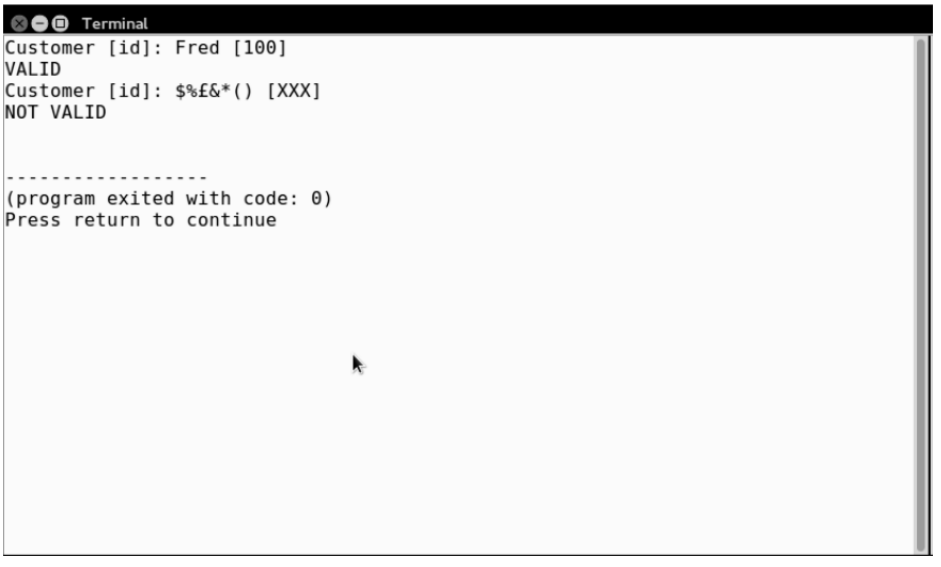
如果你将 Base 类设置为抽象类，但同时没有在 Base 类的子类中为 validate() 方法编写执行具体操作的代码，那么也会出现同样的错误。最后，在子类 Customer 中为 validate()方法添加执行具体操作的代码：

```
class Customer extends Base
{
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
  public function validate()
  {
    $valid = 0;
    $count = count(get_object_vars($this));
    if (!empty($this->id) &&is_int($this->id)) $valid++;
    if (!empty($this->name)
    &&preg_match('/[a-z0-9 ]/i', $this->name)) $valid++;
    return ($valid == $count);
  }
}
```

这样就可以添加下面的代码测试结果：

```
$customer = new Customer();

$customer->setId(100);
$customer->setName('Fred');
echo "Customer [id]: {$customer->getName()}" .
     ."[{$customer->getId()}]\n";
echo ($customer->validate()) ? 'VALID' : 'NOT VALID';
$customer->setId('XXX');
$customer->setName('$%£&*()');
echo "Customer [id]: {$customer->getName()}"
     ."[{$customer->getId()}]\n";
echo ($customer->validate()) ? 'VALID' : 'NOT VALID';
```

下面是输出结果：



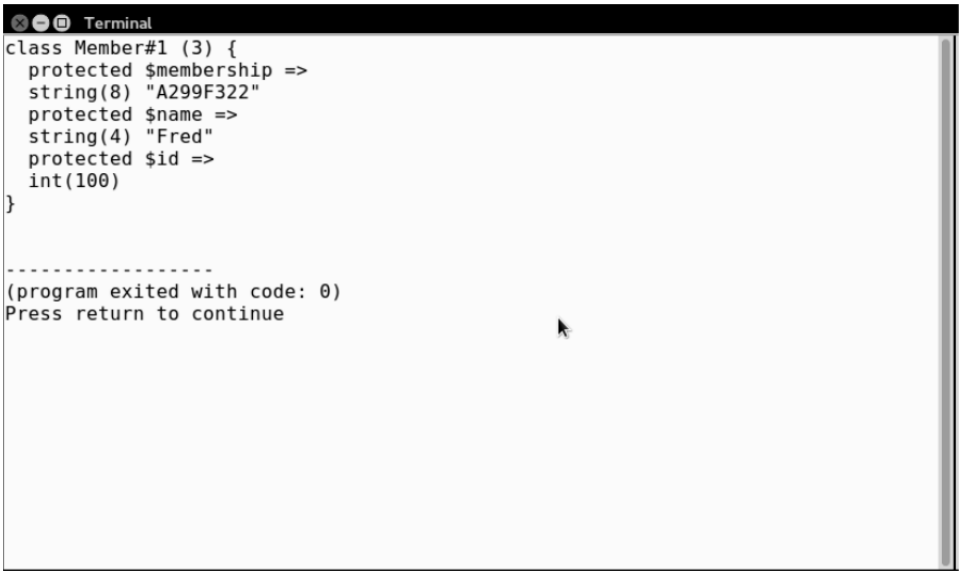
为了理解单一继承关系，可通过前面步骤 1 中介绍的 Base 和 Customer 类来创建新的 Member 类：

```
class Member extends Customer
{
  protected $membership;
  public function getMembership()
  {
    return $this->membership;
  }
  public function setMembership($memberId)
  {
    $this->membership = $memberId;
  }
}
```

创建一个 Member 类的实例，同时应注意，同一继承血脉中的所有类都拥有这些属性和方法（即使该类与 Member 类不是直接继承关系），如下面的代码所示：

```
$member = new Member();
$member->setId(100);
$member->setName('Fred');
$member->setMembership('A299F322');
var_dump($member);
```

下图是输出结果：



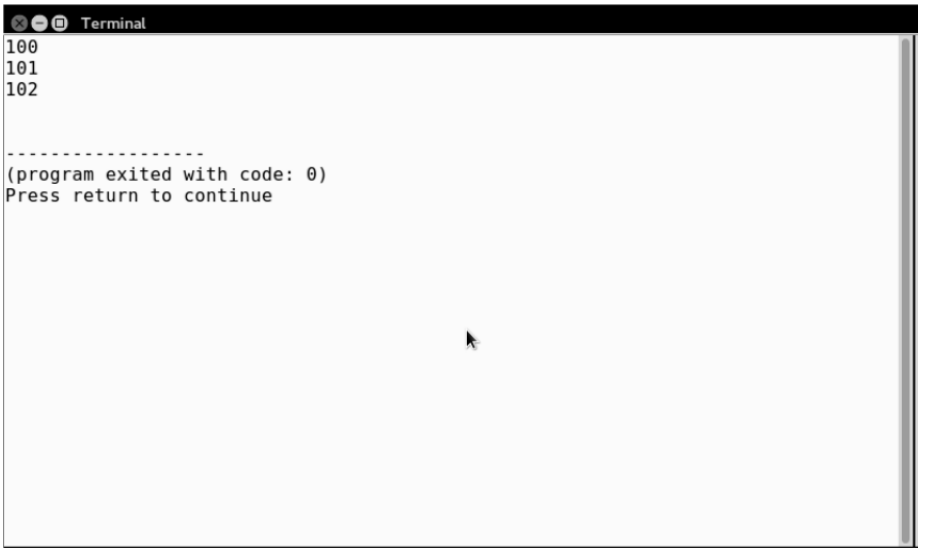
定义一个函数并将其命名为 test()，使该函数将 Base 类的实例接收为参数：

```
function test(Base $object)
{
  return $object->getId();
}
```

注意，Base、Customer 和 Member 类的实例都能够被该函数作为参数接收：

```
$base = new Base();
$base->setId(100);
$customer = new Customer();
$customer->setId(101);

$member = new Member();
$member->setId(102);
// 这 3 个类都能够在 test()函数中使用
echo test($base)     . PHP_EOL;
echo test($customer) . PHP_EOL;
echo test($member)   . PHP_EOL;
```

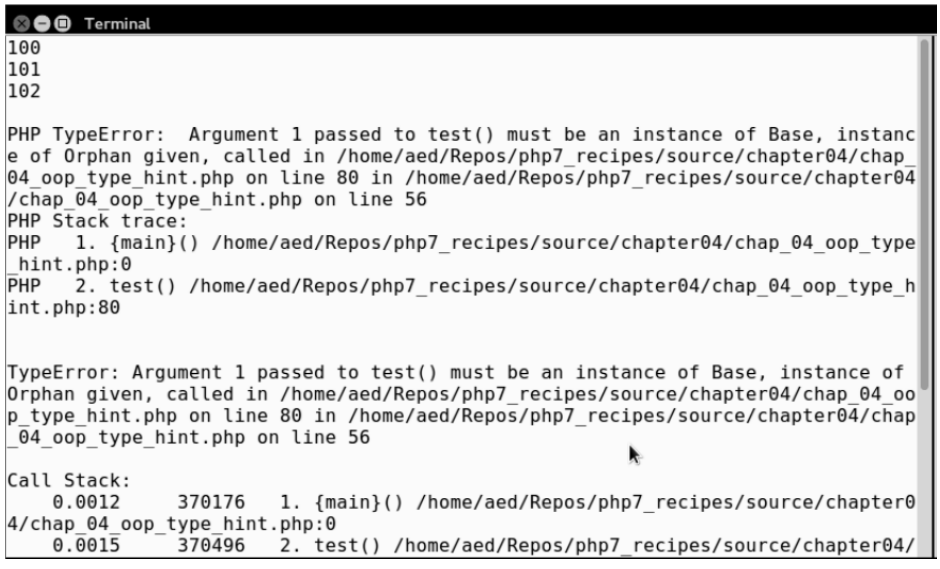
下图是输出结果：



然而，如果你尝试将该继承血脉之外的其他实例用作 `test()` 函数的参数，那么程序就会抛出 `TypeError` 异常：

```
class Orphan
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
}
try {
    $orphan = new Orphan();
    $orphan->setId(103);
    echo test($orphan) . PHP_EOL;
} catch (TypeError $e) {
    echo 'Does not work!' . PHP_EOL;
    echo $e->getMessage();
}
```

下图展示了这种情况：



使用静态属性和方法

PHP 允许在不为类创建实例的情况下访问属性和方法。使用关键字 `static` 可以做到这一点。

具体处理过程

1. 简言之，要做到这一点，只需在声明属性或方法时，在设置可见等级的关键字后面添加 `static` 关键字，然后在内部使用 `self` 关键字引用该属性：

```
class Test
{
  public static $test = 'TEST';
  public static function getTest()
  {
    return self::$test;
  }
}
```

2. `self` 关键字的绑定时间比较早，因而会导致在访问子类中的静态信息时出问题。如果你确实需要通过子类访问信息，可使用 `static` 关键字代替 `self` 关键字。这种处

理过程称为**延迟静态绑定**（Late Static Binding）。

3. 在下面的示例中，如果你使用 `Child::getEarlyTest()` 函数，那么输出结果就会是 TEST。另一方面，如果你使用 `Child::getLateTest()` 函数，那么输出结果就会是 CHILD。出现这种情况的原因是，在使用 `self` 关键字时 PHP 会绑定最早的定义，而在使用 `static` 关键字时 PHP 会绑定最迟的定义：

```
class Test2
{
  public static $test = 'TEST2';
  public static function getEarlyTest()
  {
    return self::$test;
  }
  public static function getLateTest()
  {
    return static::$test;
  }
}
class Child extends Test2
{
  public static $test = 'CHILD';
}
```

4. 在许多情况中，可将**工厂**设计模式与静态方法配合使用，以生成能够接收各种参数的实例。下面的示例定义了 `factory()` 方法，该方法会返回一个 PDO 连接：

```
public static function factory(
  $driver,$dbname,$host,$user,$pwd,array $options = [])
  {
    $dsn = sprintf('%s:dbname=%s;host=%s',
    $driver, $dbname, $host);
    try {
         return new PDO($dsn, $user, $pwd, $options);
    }    catch (PDOException $e) {
         error_log($e->getMessage);
  }
}
```

具体运行情况

可使用范围解析操作符（::）引用静态属性和方法。使用前面介绍的 Test 类运行下面的代码：

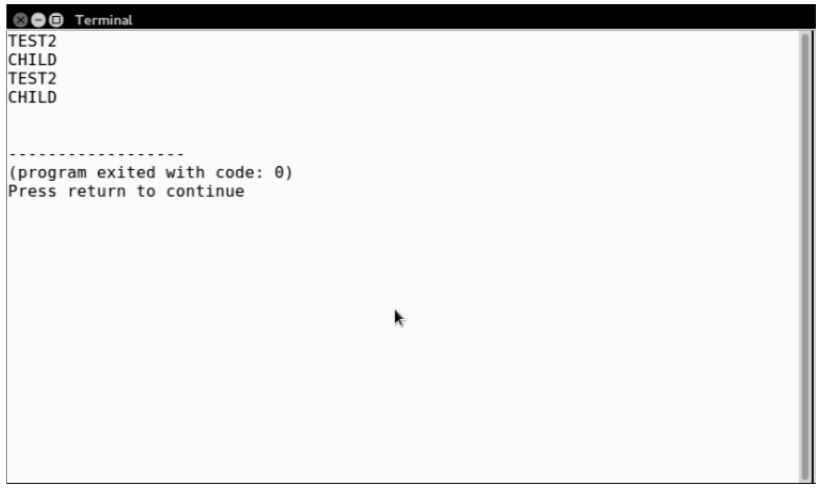
```
echo Test::$test;
echo PHP_EOL;
echo Test::getTest();
echo PHP_EOL;
```

会得到下面的输出结果：

为了理解延迟静态绑定的概念，可利用前面介绍的 Test2 和 Child 类运行下面的代码：

```
echo Test2::$test;
echo Child::$test;
echo Child::getEarlyTest();
echo Child::getLateTest();
```

下图所示的输出结果展示了 self 和 static 关键字之间的差别：



为了测试前面介绍的 `factory()`方法，可将这些代码保存到 `Application\Database` 文件夹中的 `Connection.php` 文件中的 `Application\Database\Connection` 类中。然后可运行下面的代码：

```
include __DIR__ . '/../Application/Database/Connection.php';
use Application\Database\Connection;
$connection = Connection::factory(
'mysql', 'php7cookbook', 'localhost', 'test', 'password');
$stmt = $connection->query('SELECT name FROM iso_country_codes');
while ($country = $stmt->fetch(PDO::FETCH_COLUMN))
echo $country . ' ';
```

你将看到从示例数据库中提取出的国家列表：

扩展

要了解更多关于延迟静态绑定的信息，请浏览 `http://php.net/manual/en/language.oop5.late-static-bindings.php`。

使用命名空间

PHP 开发中的一个关键性进步，是对命名空间的使用。任何已定义的命名空间都会变为类名称的前缀，从而避免因类名称意外相同引发的问题，同时还能够在开发过程中为你提供非常大的自由度。使用命名空间的另一个好处是，在与目录结构相匹配的情况下，命名空间有助于实现类自动加载功能（请参阅第 1 章）。

具体处理过程

1. 要使用命名空间定义类，只需在代码文件的顶部添加关键字 `namespace`：

```
namespace Application\Entity;
```

最佳编程习惯

就像一个文件仅保存一个类的最佳编程习惯，一个文件中也应该仅含有一个命名空间。

2. 在关键字 `namespace` 前面的 PHP 代码只能是分号和/或关键字 `declare`：

```
<?php
declare(strict_types=1);
namespace Application\Entity;
/**
 * Address 类
 *
 */
class Address
{
  // 具体代码
}
```

3. 在 PHP 5 中，如果你需要访问外部命名空间中的某个类，可以预先设定在 use 语句中仅包含这个命名空间。这样就可以使用该命名空间名称的最后一个组成部分，来引用该命名空间中的任何类：

```
use Application\Entity;
$name = new Entity\Name();
$addr = new Entity\Address();
$prof = new Entity\Profile();
```

4. 也可以确切地指明这三个类：

```
use Application\Entity\Name;
use Application\Entity\Address;
use Application\Entity\Profile;
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

5. PHP 7 引入了一种名为 group use 声明的语法改进(即对多个类起作用的 use 语句)，该语法大幅度地提高了代码可读性：

```
use Application\Entity\ {
  Name,
  Address,
  Profile
};
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

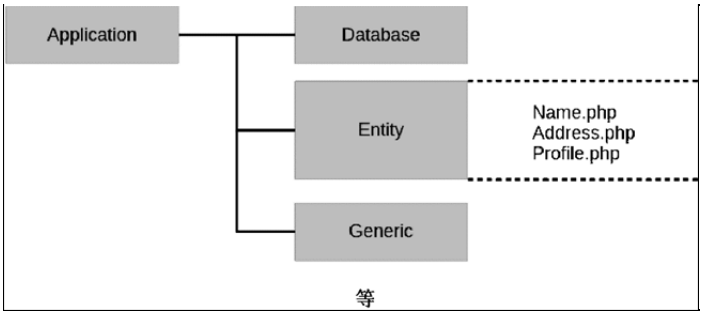
6. 如第 1 章所述，命名空间是类自动加载处理过程中不可或缺的组成部分。下面的示例展示了一个类**自动加载**器，该自动加载器会显示已经接收的参数，然后根据命名空间和类的名称来包含文件。本例假定目录结构是与命名空间相匹配的：

```
function __autoload($class)
{
  echo "Argument Passed to Autoloader = $class\n";
  include __DIR__ . '/../' . str_replace(
              '\\',DIRECTORY_SEPARATOR, $class) . '.php';
}
```

具体运行情况

为了运行该示例程序，可定义与 `Application\*`命名空间相匹配的目录结构。我们来创建基础文件夹 `Application` 和子文件夹 `Entity`。你还可以根据需要创建其他

子文件夹（如 Database 和 Generic 这些在其他章节介绍过的文件夹）：



创建三个 entity 类，分别使用 Application/Entity 文件夹中的三个文件（Name.php、Address.php 和 Profile.php）来保存它们。本例仅展示了 Application\Entity\Name 类。Address 类拥有$address 属性，Profile 类拥有$profile 属性，而且这两个类都拥有相应的 get 和 set 方法——除此之外，Application\Entity\Address 和 Application\Entity\Profile 类的处理方式都与 Application\Entity\Name 类相同的。

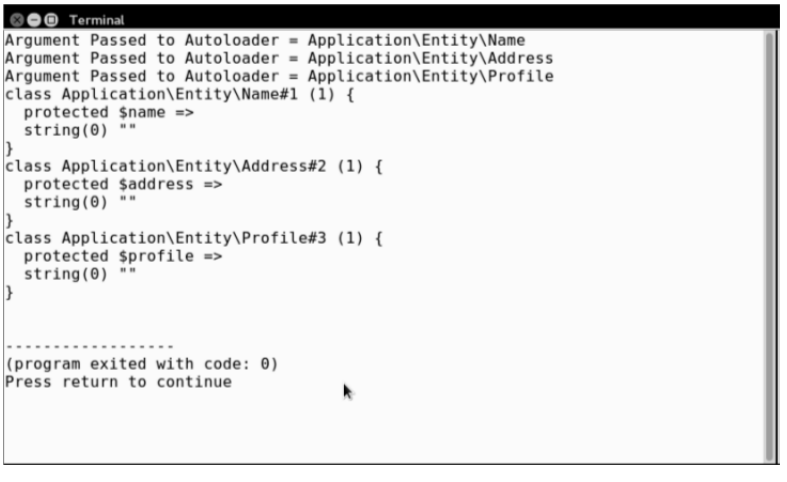
```
<?php
declare(strict_types=1);
namespace Application\Entity;
/**
 * Name 类
 *
 */
class Name
{

  protected $name = '';

  /**
   * 该方法会返回变量$name 的当前值
   *
   * @返回字符型$name 变量值
   */
  public function getName() : string
  {
    return $this->name;
  }

  /**
   * 该方法会设置变量$name 的值
```

```
     *
     * @接收字符型的$name 参数
     * @返回本对象的 name 属性
     */
    public function setName(string $name)
    {
      $this->name = $name;
      return $this;
    }
}
```

这样你既可以使用第 1 章介绍的自动加载器，也可以使用前面介绍的简化的自动加载器。在 chap_04_oop_namespace_example_1.php 文件中添加设置自动加载功能的命令。这样我们就可以在该文件中设定仅引用该命名空间（而不是类的名称）的 use 语句。通过将该命名空间的最后一部分（Entity）添加为类名称的前缀，我们为 Name、Address 和 Profile 类创建实例：

```
use Application\Entity;
$name = new Entity\Name();
$addr = new Entity\Address();
$prof = new Entity\Profile();

var_dump($name);
var_dump($addr);
var_dump($prof);
```

下面是输出结果：



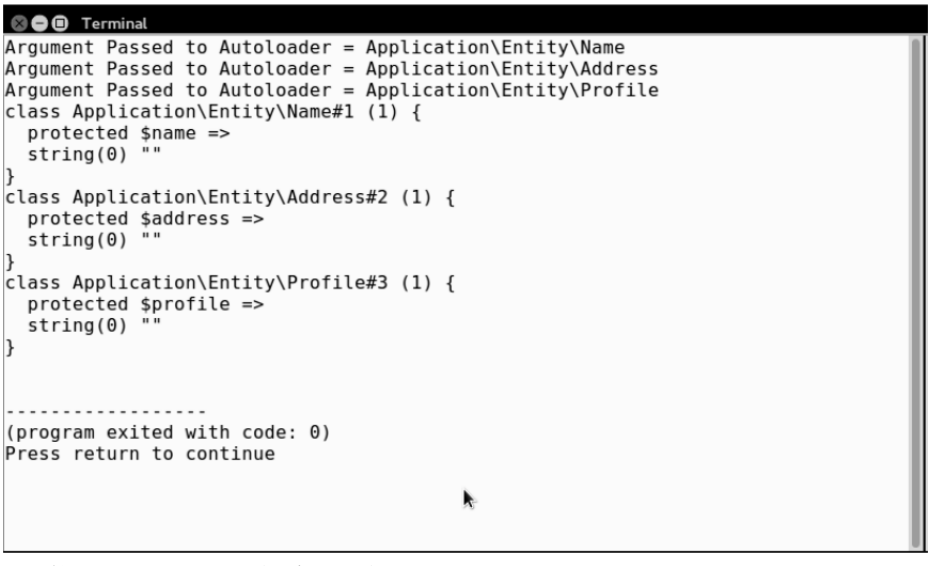
使用 Save as 选项将该文件复制到名为 chap_04_oop_namespace_example_2.php 的新文件中。将 use 语句更改为下面的内容：

```
use Application\Entity\Name;
use Application\Entity\Address;
use Application\Entity\Profile;
```

这样就可以仅使用类名称创建类的实例了：

```
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

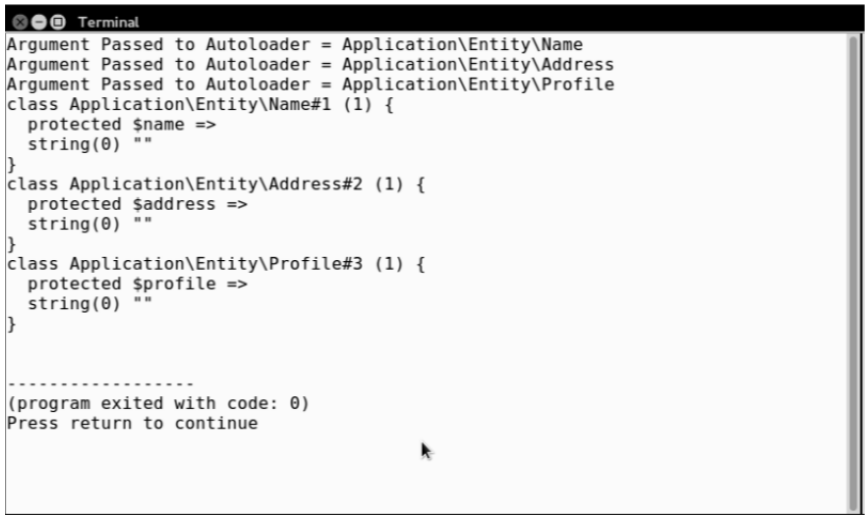
运行这个脚本会得到下面的输出结果：



再次使用 Save as 选项创建新文件 chap_04_oop_namespace_example_3.php。现在可以测试 PHP 7 新增的 group use 功能：

```
use Application\Entity\ {
  Name,
  Address,
  Profile
};
$name = new Name();
$addr = new Address();
$prof = new Profile();
```

运行这段代码后，会获得与上一段示例代码相同的输出结果：



定义可见性

“可见性”这个词容易让我们联想到安全性，但实际上它与应用程序的安全性没有任何关系！确切地说，它仅是一种控制代码用途的机制。通过它，缺乏经验的开发人员可以避免以公用方式使用仅应在类定义内部调用的方法。

具体处理过程

1. 在属性或方法定义的前面添加关键字 public、protected 或 private，可以设置可见性等级。通过将属性设置为 protected（受保护的）或 private（私有的），可以强制这些属性只能通过公用的（public）读取和赋值函数被访问。

2. 在本例中，Base 类定义了一个受保护的属性$id。为了访问该属性，该类还定义了公用方法 getId() 和 setId()。受保护的方法 generateRandId() 可以在这个类的内部使用，而且会被子类 Customer 继承。该方法无法直接在 Base 类的定义之外被调用。注意，使用 PHP 7 新增的 random_bytes() 函数可以创建随机 ID。

```
class Base
{
  protected $id;
```

```
  private $key = 12345;
  public function getId()
  {
    return $this->id;
  }
  public function setId()
  {
    $this->id = $this->generateRandId();
  }
  protected function generateRandId()
  {
    return unpack('H*', random_bytes(8))[1];
  }
}

class Customer extends Base
{
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
}
```

最佳编程习惯

将属性设置为 protected 并定义公用的读取属性名称和设置属性名称的方法，可以控制对属性执行的访问操作。这类方法称为读取器和设置器。

3. 将属性或方法设置为 private，可以防止这些属性或方法被继承，也可以防止从它们所在的类定义之外访问它们。这是一种将类创建为单例对象的好方式。

4. 下面的代码示例展示了 Registry 类，该类仅有一个实例。因为构造器被设置为 private，所以创建实例的唯一途径是使用静态方法 getInstance()：

```
class Registry
{
  protected static $instance = NULL;
```

```
    protected $registry = array();
    private function __construct()
    {
      // 谁都不能为这个类创建实例
    }
    public static function getInstance()
    {
      if (!self::$instance) {
        self::$instance = new self();
      }
      return self::$instance;
    }
    public function __get($key)
    {
      return $this->registry[$key] ?? NULL;
    }
    public function __set($key, $value)
    {
      $this->registry[$key] = $value;
    }
}
```

为方法添加 final 关键字，可以防止该方法被重写。为类添加 final 关键字，可以防止类被扩展。

5. 类的常量通常被视为拥有公用级的可见性。从 PHP 7.1 以后，可以将类的常量声明为 protected 或 private。在下面的示例中，类常量 TEST_WHOLE_WORLD 的行为与在 PHP 5 中的行为完全相同。后面两个常量(TEST_INHERITED 和 TEST_LOCAL) 遵守受保护的和私有的属性和方法规则：

```
class Test
{

  public const TEST_WHOLE_WORLD = 'visible.everywhere';

  // 注意：只能在 PHP 7.1 及以上版本中起作用
  protected const TEST_INHERITED = 'visible.in.child.classes';

  // 注意：只能在 PHP 7.1 及以上版本中起作用
```

```
    private const TEST_LOCAL= 'local.to.class.Test.only';

    public static function getTestInherited()
    {
      return static::TEST_INHERITED;
    }

    public static function getTestLocal()
    {
      return static::TEST_LOCAL;
    }
}
```

具体运行情况

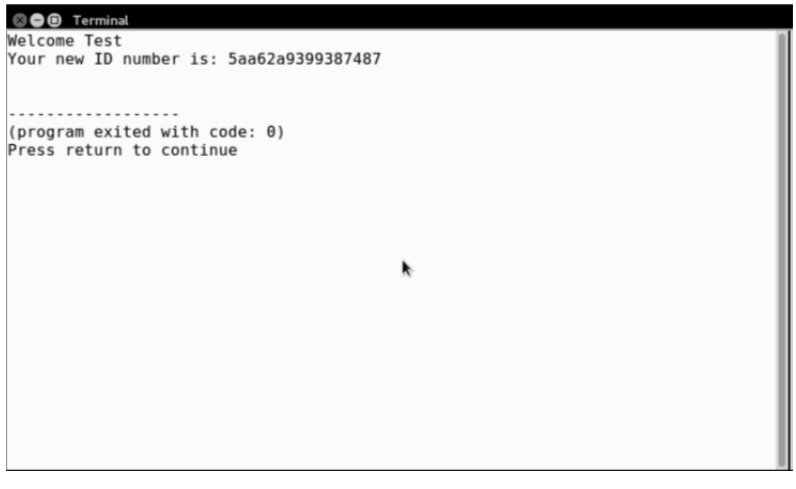
创建文件 chap_04_basic_visibility.php 并定义两个类：Base 和 Customer。然后添加为这两个类创建实例的代码：

```
$base     = new Base();
$customer = new Customer();
```

注意，下面的代码可以正常运行，而且符合最佳编程习惯：

```
$customer->setId();
$customer->setName('Test');
echo 'Welcome ' . $customer->getName() . PHP_EOL;
echo 'Your new ID number is: ' . $customer->getId() . PHP_EOL;
```

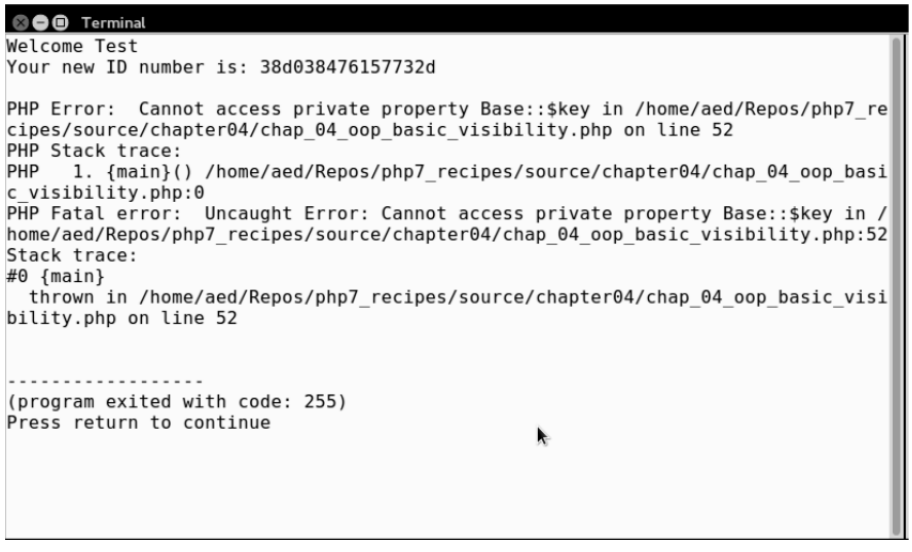
即使变量$id 是受保护的，但相应的方法 getId()和 setId()都是公用的，因此能够从该类定义的外部访问变量$id。下面是输出结果：



然而，下面代码无法运行，因为私有的和受保护的属性是无法从类定义的外部被访问的：

```
echo 'Key (does not work): ' . $base->key;
echo 'Key (does not work): ' . $customer->key;
echo 'Name (does not work): ' . $customer->name;
echo 'Random ID (does not work): ' . $customer->generateRandId();
```

下面的输出结果展示了我们预料之中的错误：



扩展

要详细了解读取器和设置器，请参阅本章后面的内容。要详细了解 PHP 7.1 中类常量可见性的设置，请浏览 `https://wiki.php.net/rfc/class_const_visibility`。

使用接口

接口是非常有用的系统构建工具，通常用于规范化**应用程序编程接口**（API）。接口不含有执行实际操作的代码，但是会含有方法的名称和签名。

在接口中确定的所有方法，都会拥有公用的可见性等级。

具体处理过程

1. 接口确定的方法无法含有执行实际操作的实现代码，但是可以在接口中设置方法参数的数据类型。

2. 在下面的示例中，ConnectionAwareInterface 接口确定了 setConnection() 方法，该方法接收 Connection 类类型的参数：

```
interface ConnectionAwareInterface
{
  public function setConnection(Connection $connection);
}
```

3. 要使用这个接口，可在定义类的代码行中添加 implements 关键字。定义两个类(CountryList 和 CustomerList)，这两个类都需要通过 setConnection() 方法访问 Connection 类。为了确定这个依赖关系，这两个类都实现了 ConnectionAwareInterface 接口：

```
class CountryList implements ConnectionAwareInterface
{
  protected $connection;
  public function setConnection(Connection $connection)
  {
    $this->connection = $connection;
  }
  public function list()
  {
    $list = [];
    $stmt = $this->connection->pdo->query(
      'SELECT iso3, name FROM iso_country_codes');
    while ($country = $stmt->fetch(PDO::FETCH_ASSOC)) {
      $list[$country['iso3']] = $country['name'];
    }
    return $list;
  }

}
class CustomerList implements ConnectionAwareInterface
```

```
{
  protected $connection;
  public function setConnection(Connection $connection)
  {
    $this->connection = $connection;
  }
  public function list()
  {
    $list = [];
    $stmt = $this->connection->pdo->query(
      'SELECT id, name FROM customer');
    while ($customer = $stmt->fetch(PDO::FETCH_ASSOC)) {
      $list[$customer['id']] = $customer['name'];
    }
    return $list;
  }

}
```

4. 可以使用接口实现类型提示功能。在下面的示例代码中，ListFactory 类含有 factory()方法，该方法可以初始化所有实现了 ConnectionAwareInterface 接口的类。该接口确保了 setConnection()方法会被定义。在接口中（而不是在具体的类实例中）设置类型提示功能，能够使 factory 方法拥有更高的通用性：

```
namespace Application\Generic;

use PDO;
use Exception;
use Application\Database\Connection;
use Application\Database\ConnectionAwareInterface;
class ListFactory

{
  const ERROR_AWARE = 'Class must be Connection Aware';
  public static function factory(
    ConnectionAwareInterface $class, $dbParams)
  {
    if ($class instanceofConnectionAwareInterface) {
        $class->setConnection(new Connection($dbParams));
        return $class;
    } else {
        throw new Exception(self::ERROR_AWARE);
    }
    return FALSE;
  }
}
```

5. 如果一个类实现了多个接口，在方法签名不匹配的情况下就会出现**命名冲突**（naming collision）。下面的示例代码中有两个接口（DateAware 和 TimeAware）。除了定义 setDate() 和 setTime() 方法外，它们还都定义了 setBoth() 方法。尽管这不是最佳编程习惯，但拥有相同的方法名称不算问题。问题在于这两个方法的签名不同：

```
interface DateAware
{
  public function setDate($date);
  public function setBoth(DateTime $dateTime);
}

interface TimeAware
{
  public function setTime($time);
  public function setBoth($date, $time);
}

class DateTimeHandler implements DateAware, TimeAware
{
  protected $date;
  protected $time;
  public function setDate($date)
  {
    $this->date = $date;
  }
  public function setTime($time)
  {
    $this->time = $time;
  }
  public function setBoth(DateTime $dateTime)
  {
    $this->date = $date;
  }
}
```

6. 如果运行这段代码，就会产生一个致命错误（且是无法捕捉的）。要解决这个问题，推荐方式是从两个接口中的任意一个删除 setBoth() 方法。可选方式是调整这两个方法的签名，以使它们变得匹配。

最佳编程习惯

不应使用相同或重叠的方法定义接口。

具体运行情况

在 Application/Database 文件夹中，创建文件 ConnectionAwareInterface.php。将前面步骤 2 中介绍的代码添加到该文件中。

在 Application/Generic 文件夹中，创建两个文件：CountryList.php 和 CustomerList.php。将前面步骤 3 介绍的两个类的代码分别添加到这两个类文件中。

在任意一个与 Application 目录同级的目录中，创建源代码文件 chap_04_oop_simple_interfaces_example.php，该文件用于初始化类自动加载器和保存数据库参数：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
$params = include __DIR__ . DB_CONFIG_FILE;
```

本例中的数据库参数位于由常量 DB_CONFIG_FILE 设定的数据库配置文件中。

现在可以使用 ListFactory::factory() 方法生成 CountryList 和 CustomerList 对象（分别代表国家名单和客户名单）了。注意，如果这些类没有实现 ConnectionAwareInterface 接口，那么程序会抛出错误：

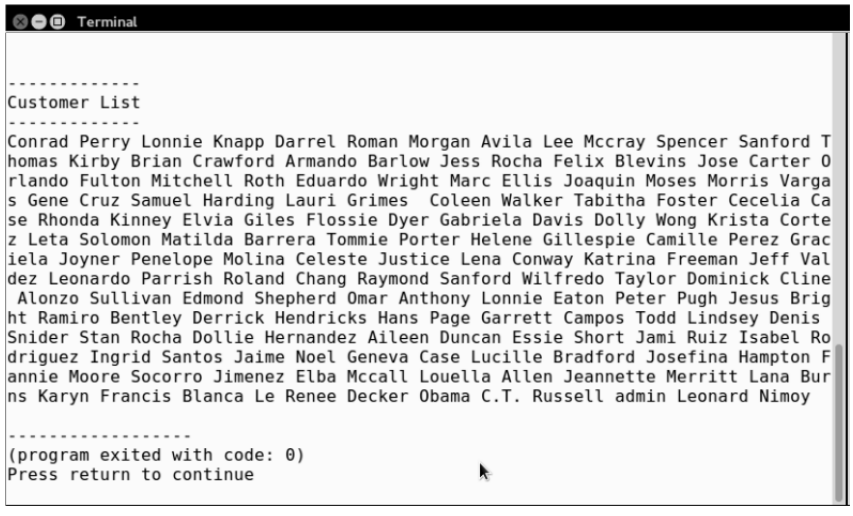
```
$list = Application\Generic\ListFactory::factory(
  new Application\Generic\CountryList(), $params);
foreach ($list->list() as $item) echo $item . ' ';
```

下面是通过 CountryList 对象获得的输出结果：

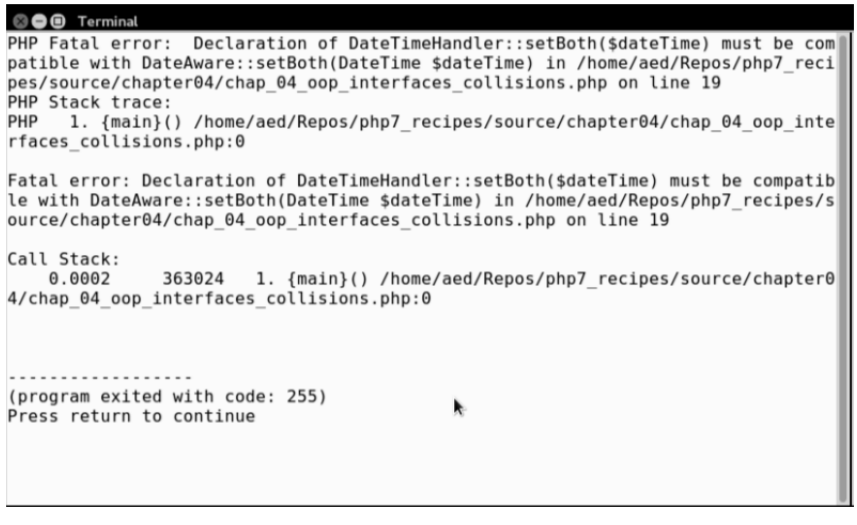
还可以使用 factory 方法生成 CustomerList 对象并使用该示例：

```
$list = Application\Generic\ListFactory::factory(
  new Application\Generic\CustomerList(), $params);
foreach ($list->list() as $item) echo $item . ' ';
```

下面是通过 CustomerList 示例获得的输出结果：



如果你想通过实验查明当实现了多个接口时，方法签名各不相同会出现怎样的情况，可将前面步骤 4 介绍的代码添加到文件 chap_04_oop_interfaces_collisions.php 中。运行该文件会出现如下所示的错误：



如果你对 TimeAware 接口进行如下所示的调整，就不会出现错误：

```
interface TimeAware
{
  public function setTime($time);
  // 这会产生一个错误
  public function setBoth(DateTime $dateTime);
}
```

使用特性

如果你使用过 C 语言，那么就会熟悉宏。宏是一种预先定义好的代码块，它会在指定的位置展开。与此类似，特性也是一种代码块，这种代码块可以被复制和粘贴到类中由 PHP 解释器指定的位置。

具体处理过程

1. 使用关键字 trait 可以定义特性，特性中可以含有属性和方法。在测试上一节介绍的 CountryList 和 CustomerList 类的功能时，你可能已经注意到通过复制添加的代码。本例会重新构建这两个类，并将 list() 方法迁移到一个特性中。注意，这两个类中的 list() 方法是相同的。

2. 当两个或更多个类中含有一部分相同的代码时，就可以使用特性。但请注意，对这类情况的常规处理方式是创建抽象类并扩展它，与特性相比抽象类具有一定优势。特性无法用于确定继承关系，而抽象父类可以做到这一点。

3. 下面将 list() 方法复制到名为 ListTrait 的特性中：

```
trait ListTrait
{
  public function list()
  {
    $list = [];
    $sql = sprintf('SELECT %s, %s FROM %s',
      $this->key, $this->value, $this->table);
    $stmt = $this->connection->pdo->query($sql);
    while ($item = $stmt->fetch(PDO::FETCH_ASSOC)) {
      $list[$item[$this->key]] =
      $item[$this->value];
```

```
        }
        return $list;
      }
    }
```

4.这样就可以将 ListTrait 特性中的代码插入到新建的类 CountryListUsingTrait 中了，如下面的代码所示。现在可以将 list()方法的全部代码从 CountryListUsingTrait 类中删除了：

```
class CountryListUsingTrait implements ConnectionAwareInterface
{

  use ListTrait;
  protected $connection;
  protected $key = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';

  public function setConnection(Connection $connection)
  {
    $this->connection = $connection;
  }

}
```

如果某些部分的代码有副本，那么每次需要做修改时就会出现一个潜在的问题，那就是你必须做非常多的全局搜索和替换操作（或剪切和粘贴代码操作），而且经常会引发灾难性的后果。特性是可以避免这些维护噩梦的好方式。

5. 特性会受到命名空间的影响。在步骤 1 介绍的示例中，如果将新建的类 CountryListUsingTrait 放置到命名空间 Application\Generic 中，还需要将 ListTrait 特性也迁移到这个命名空间中：

```
namespace Application\Generic;

use PDO;

trait ListTrait
{
  public function list()
  {
```

```
    // 具体代码与前面介绍过的相同
  }
}
```

6. 特性中的方法会重写继承的方法。

7. 如下面的示例所示，父类 Base 中 setId()方法的返回值与特性 Test 中 setId()方法的返回值不相同。Customer 类从父类 Base 那里继承了 setId()方法，但 Customer 类也使用了 Test 特性。在这种情况中，在特性中定义的 setId()方法会重写从父类 Base 那里继承的 setId()方法：

```
trait Test
{
  public function setId($id)
  {
    $obj = new stdClass();
    $obj->id = $id;
    $this->id = $obj;
  }
}

class Base
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
}

class Customer extends Base
{
  use Test;
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
```

```
    $this->name = $name;
  }
}
```

在 PHP 5 中，特性也会重写属性。在 PHP 7 中，如果特性中属性的初始化值与父类中属性的初始化值不相同，那么就会产生致命错误。

8. 在使用了特性的类中直接定义的方法，会重写在特性中定义的方法。

9. 在下面的示例中，Test 特性定义了$id 属性、getId()方法和 setId()方法。该特性还定义了 setName()方法，该方法的名称与 Customer 类中定义的一个方法相同。在这类情况下，在 Customer 类中直接定义的 setName()方法会重写在 Test 特性中定义的 setName()方法：

```
trait Test
{
  protected $id;
  public function getId()
  {
    return $this->id;
  }
  public function setId($id)
  {
    $this->id = $id;
  }
  public function setName($name)
  {
    $obj = new stdClass();
    $obj->name = $name;
    $this->name = $obj;
  }
}

class Customer
{
  use Test;
  protected $name;
  public function getName()
  {
    return $this->name;
  }
  public function setName($name)
  {
```

```
    $this->name = $name;
  }
}
```

10. 在使用多个特性时，可使用 insteadof 关键字解决方法名称冲突的问题。同时还可以使用关键字 as 为方法设置别名。

11. 下面的示例创建了两个特性：IdTrait 和 NameTrait。这两个特性都定义了 setKey()方法，但是这两个 setKey()方法的内容并不相同。Test 类使用了这两个特性。请注意 insteadof 关键字的用法，使用这个关键字可以解决方法名称冲突的问题。因此，当通过 Test 类调用 setKey()方法时，PHP 引擎会从 NameTrait 特性中提取 setKey()方法的源代码。此外，IdTrait 特性中的 setKey()方法仍旧可以起作用，但是它的别名会被设置为 setKeyDate()：

```
trait IdTrait
{
  protected $id;
  public $key;
  public function setId($id)
  {
    $this->id = $id;
  }
  public function setKey()
  {
    $this->key = date('YmdHis')
    . sprintf('%04d', rand(0,9999));
  }
}

trait NameTrait
{
  protected $name;
  public $key;
  public function setName($name)
  {
    $this->name = $name;
  }
  public function setKey()
  {
    $this->key = unpack('H*', random_bytes(18))[1];
  }
}

class Test
```

```
{
  use IdTrait, NameTrait {
    NameTrait::setKeyinsteadofIdTrait;
    IdTrait::setKey as setKeyDate;
  }
}
```

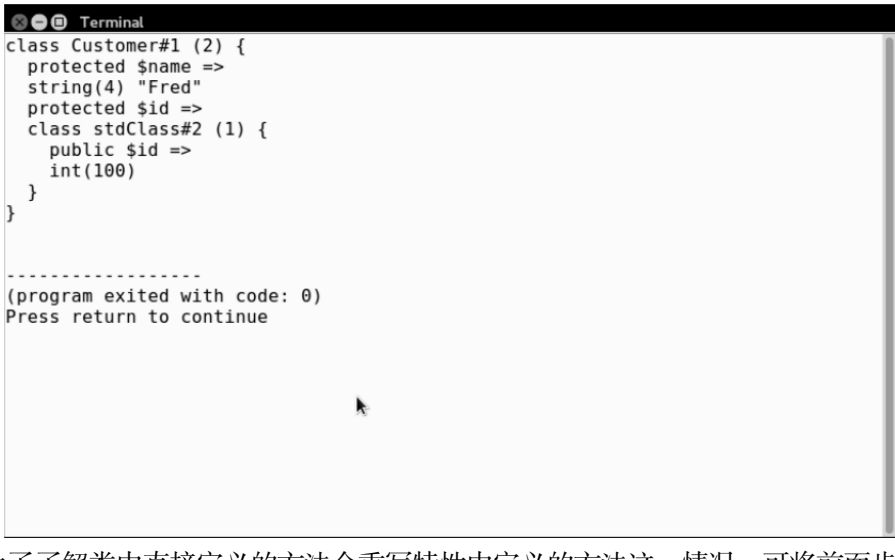
具体运行情况

前面的步骤 1 介绍了使用特性处理代码重复部分的方式。在开发自己的项目时，你需要权衡一下是定义一个基类并扩展它比较好，还是使用特性比较好。在处理多个没有逻辑关系的类中含有重复代码的情况时，特性尤为有用。

为了理解特性方法重写继承方法的方式，可将前面步骤 7 介绍的代码复制到文件 chap_04_oop_traits_override_inherited.php 中，并添加下面的代码：

```
$customer = new Customer();
$customer->setId(100);
$customer->setName('Fred');
var_dump($customer);
```

如下面的输出结果所示，属性$id 存储了 stdClass()类的实例，这就是在特性 Test 中 setId()方法定义的操作：

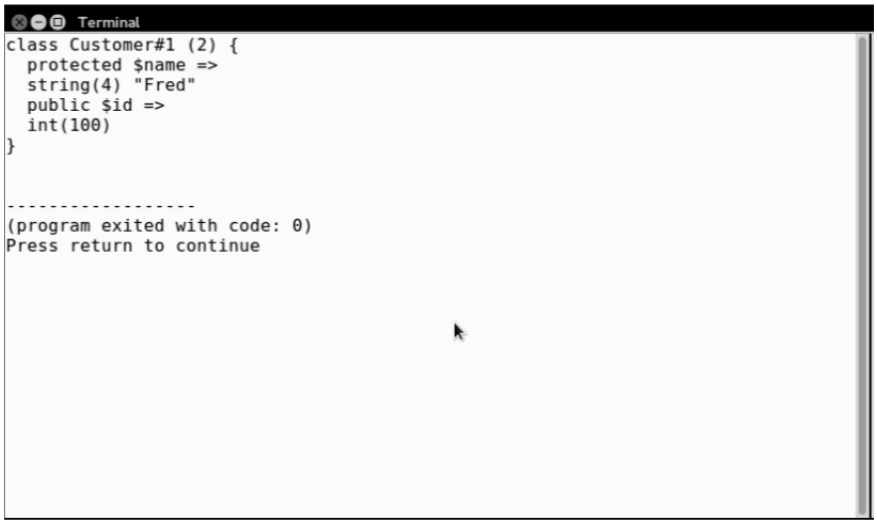


为了了解类中直接定义的方法会重写特性中定义的方法这一情况，可将前面步骤 9 介绍的代码添加到文件 chap_04_oop_trait_methods_do_not_override_class_

methods.php 中，并添加下面的代码：

```
$customer = new Customer();
$customer->setId(100);
$customer->setName('Fred');
var_dump($customer);
```

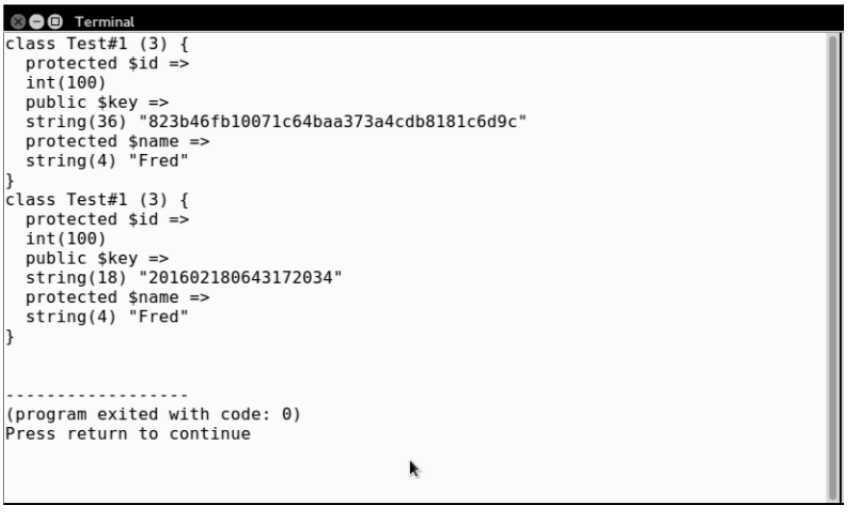
如下面的输出结果所示，属性$id 存储了一个整型的数值（正如在 Customer 类中 setName()方法定义的那样），而特性 Test 中的 setName()方法将属性$id 定义为存储 stdClass 类的示例：



前面的步骤 10 介绍了如何解决在使用多个特性时方法名称冲突的问题。将前面步骤 11 介绍的代码复制到文件 chap_04_oop_trait_multiple.php 中，并添加下面代码：

```
$a = new Test();
$a->setId(100);
$a->setName('Fred');
$a->setKey();
var_dump($a);
$a->setKeyDate();
var_dump($a);
```

注意，在 NameTrait 特性中定义的 setKey()方法，通过 PHP 7 中新增的函数 random_bytes()生成了下面的输出结果。而在特性 IdTrait 中定义的 setKey()方法，在 Test 类中被设置了一个别名（setKeyDate()），该方法通过函数 date()和 rand()生成了一个关键值：



实现匿名类

PHP 7 引入了一种新功能：**匿名类**。与匿名函数很像，匿名类可以被定义为表达式的组成部分，从而创建没有名称的类。可以在需要快速创建对象（用过后便丢弃）的情况中使用匿名类。

具体处理过程

1. 可以将 stdClass 类用作定义匿名类的模板。

在匿名类的定义中，你可以自由地定义任何属性和方法（包括魔术方法（magic method））。下面的示例定义了一个匿名类，该匿名类中含有两个属性和魔术方法 __construct()：

```
$a = new class (123.45, 'TEST') {
  public $total = 0;
  public $test = '';
  public function __construct($total, $test)
  {
    $this->total = $total;
    $this->test = $test;
  }
};
```

2. 通过匿名类可以扩展任何类。

下面的示例通过匿名类扩展了 FilterIterator 类，并重写了__construct()和 accept()方法。FilterIterator 类将 ArrayIterator 类类型的变量$b 接收为参数，该变量代表一组数值（从 10 至 100，增量幅度为 10）。FilterIterator 类的第二个参数用于限定输出结果的范围：

```
$b = new ArrayIterator(range(10,100,10));
$f = new class ($b, 50) extends FilterIterator {
  public $limit = 0;
  public function __construct($iterator, $limit)
  {
    $this->limit = $limit;
    parent::__construct($iterator);
  }
  public function accept()
  {
    return ($this->current() <= $this->limit);
  }
};
```

3. 匿名类可以实现接口。

在下面的示例中，一个匿名类被用于生成 HTML 颜色代码表。这个类实现了 PHP 内置的 Countable 接口。该类还定义了 count()方法，通过要求被处理对象必须拥有 Countable 接口的方法或函数来使用该类时，count()方法就会被调用：

```
define('MAX_COLORS', 256 ** 3);

$d = new class () implements Countable {
  public $current = 0;
  public $maxRows = 16;
  public $maxCols = 64;
  public function cycle()
  {
    $row = '';
    $max = $this->maxRows * $this->maxCols;
    for ($x = 0; $x < $this->maxRows; $x++) {
      $row .= '<tr>';
      for ($y = 0; $y < $this->maxCols; $y++) {
        $row .= sprintf(
          '<td style="background-color: #%06X;"',
          $this->current);
        $row .= sprintf(
```

```
            'title="#%06X"> </td>',
            $this->current);
          $this->current++;
          $this->current = ($this->current >MAX_COLORS) ? 0
              : $this->current;
        }
        $row .= '</tr>';
      }
      return $row;
    }
    public function count()
    {
      return MAX_COLORS;
    }
};
```

4. 匿名类可以使用特性。

5. 下面示例中的部分代码与介绍特性示例中的部分代码相同。但本例没有定义 Test 类，而是使用一个匿名类替代了 Test 类：

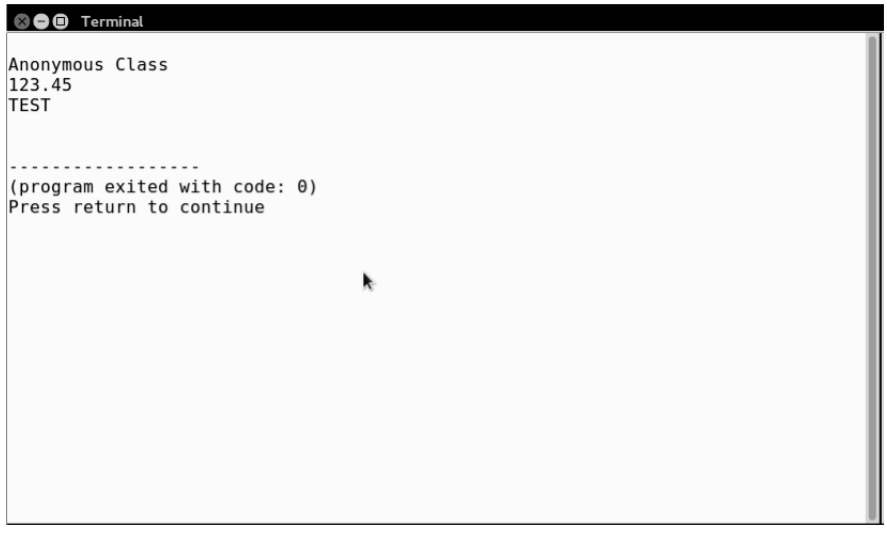
```
$a = new class() {
  use IdTrait, NameTrait {
    NameTrait::setKeyinsteadofIdTrait;
    IdTrait::setKey as setKeyDate;
  }
};
```

具体运行情况

你可以在匿名类中定义任何属性和方法。如前面的示例所示，可以定义接收构造器参数的匿名类，以及访问属性的位置。将前面步骤 2 介绍的代码添加到文件 chap_04_oop_anonymous_class.php 中，并添加下面的 echo 语句：

```
echo "\nAnonymous Class\n";
echo $a->total .PHP_EOL;
echo $a->test . PHP_EOL;
```

下面是这个匿名类的输出结果：



要使用 `FilterIterator` 类，就必须重写 `accept()` 方法。你需要在该方法中规定迭代中的哪些元素应被用作输出结果。将前面步骤 4 介绍的代码添加到一个测试脚本文件中，然后在该文件中添加下面的 `echo` 语句，以便测试这个匿名类：

```
echo "\nAnonymous Class Extends FilterIterator\n";
foreach ($f as $item) echo $item . '';
echo PHP_EOL;
```

本例中输出结果的限定值为 50。初始的 `ArrayIterator` 对象含有一组值（从 10 到 100，增量幅度为 10），如下面的输出结果所示：

为了了解实现接口的匿名类，可使用前面步骤 5 和步骤 6 介绍的代码。将这些代码添加到文件 chap_04_oop_anonymous_class_interfaces.php 中。

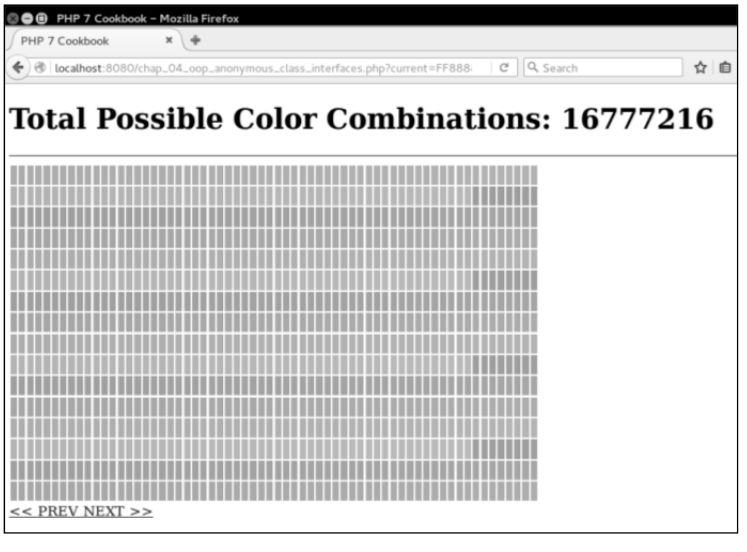
然后添加下面的代码便可以对 HTML 颜色表进行分页了：

```
$d->current = $_GET['current'] ?? 0;
$d->current = hexdec($d->current);
$factor = ($d->maxRows * $d->maxCols);
$next = $d->current + $factor;
$prev = $d->current - $factor;
$next = ($next <MAX_COLORS) ? $next : MAX_COLORS - $factor;
$prev = ($prev>= 0) ? $prev : 0;
$next = sprintf('%06X', $next);
$prev = sprintf('%06X', $prev);
?>
```

最后，通过网页显示 HTML 颜色表：

```
<h1>Total Possible Color Combinations: <?= count($d); ?></h1>
<hr>
<table>
<?= $d->cycle(); ?>
</table>
<a href="?current=<?= $prev ?>"><<PREV</a>
<a href="?current=<?= $next ?>">NEXT >></a>
```

注意，可以通过将匿名类的实例传送到 count() 函数（位于两个<H1>标签之间）中，来利用 Countable 接口提供的功能。下面是在浏览器中显示的输出结果：



为了了解在匿名类中使用特性的情况，可将上一节介绍的 chap_04_oop_trait_multiple.php 文件的内容复制到新建文件 chap_04_oop_trait_nonymous_class.php 中。使用匿名类替换 Test 类：

```
$a = new class() {
  use IdTrait, NameTrait {
    NameTrait::setKeyinsteadofIdTrait;
    IdTrait::setKey as setKeyDate;
  }
};
```

删除下面的代码：

```
$a = new Test();
```

运行这些代码后，你会发现除了对匿名类的引用外，其他输出结果与上一节获得的输出结果完全相同：

第 5 章　与数据库进行交互

本章包括以下要点：

- 使用 PDO 连接数据库
- 创建 OOP 式的 SQL 语句生成器
- 处理分页
- 定义与数据库表匹配的实体
- 将实体类的数据类型设置为与 RDBMS（关系型数据库管理系统）查询操作匹配的数据类型
- 在查询结果中嵌入二次查询操作
- 实现 jQuery DataTables 插件的 PHP 查询

本章主要内容简介

本章介绍一系列使用 **PHP 数据对象**（PDO）扩展连接数据库的方式，还会介绍一些常见的编程问题，如**结构化查询语言**（SQL）的生成、分页，以及将对象的数据类型与数据库表匹配。本章末尾会介绍以嵌入匿名函数的方式处理二次查询的手段，以及使用 jQuery DataTables 插件创建 AJAX 请求的方式。

使用 PDO 连接数据库

PDO 是一种具有高性能并得到了积极维护的数据库扩展，与特定于供应商的扩展相比具有独特的优势。PDO 拥有通用的**应用程序编程接口**（API），该接口能够与十多种**关系型数据库管理系统**（RDBMS）兼容。学会使用这种扩展，可以省去学习特定于供应商的数据库扩展中的命令集的时间。

如下表所示，PDO 主要分为 4 类：

类别	功能
PDO	维护与数据库的实际连接，还能够实现一些低级功能，如事务支持
PDOStatement	处理结果
PDOException	实现数据库专用的异常
PDODriver	与实际的供应商专用数据库通信

具体处理过程

1. 通过创建 PDO 实例设置数据库连接。

2. 还需要创建**数据源名称**（Data Source Name，DSN）。DSN 中包含的信息会随应用的不同数据库驱动程序而不同。例如，下面就是一个用于与 MySQL 数据库连接的 DSN：

```
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd' => 'password',
  'db' => 'php7cookbook'
];

try {
  $dsn = sprintf('mysql:host=%s;dbname=%s',
  $params['host'], $params['db']);
  $pdo = new PDO($dsn, $params['user'], $params['pwd']);
} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

3. 从另一方面看，SQLite 数据库的扩展比较简单，仅需要使用下面的命令：

```
$params = [
  'db' => __DIR__ . '/../data/db/php7cookbook.db.sqlite'
];
$dsn = sprintf('sqlite:' . $params['db']);
```

4. 用于与 PostgreSQL 数据库相连的 DSN 直接包含了用户名和密码：

```
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd' => 'password',
```

```
  'db' => 'php7cookbook'
];
$dsn = sprintf('pgsql:host=%s;dbname=%s;user=%s;password=%s',
          $params['host'],
          $params['db'],
          $params['user'],
          $params['pwd']);
```

5. 如下面示例所示，DSN 中还可以包含特定于服务器的指令，如 unix_socket：

```
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd' => 'password',
  'db' => 'php7cookbook',
  'sock' => '/var/run/mysqld/mysqld.sock'
];

try {
  $dsn = sprintf('mysql:host=%s;dbname=%s;unix_socket=%s',
            $params['host'], $params['db'], $params['sock']);
  $opts = [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION];
  $pdo = new PDO($dsn, $params['user'], $params['pwd'], $opts);
} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

最佳编程习惯

应将创建 PDO 实例的语句封装在 try {}catch{}代码块中，这样在出现失效情况时，可以捕捉特定于数据库信息的 PDOException 异常，并捕捉拥有 Throwable 接口的错误或其他异常。最好将 PDO 的错误处理模式设置为 PDO::ERRMODE_EXCEPTION。

在 PHP 5 中，如果无法创建 PDO 对象（例如使用了非法参数），那么该实例就会被赋值为 NULL。在 PHP 7 中出现这种情况时，程序会抛出异常。如果将创建 PDO 对象的语句封装在 try {}catch{}代码块中，并将 PDO::ATTR_ERRMODE 属性设置为 PDO::ERRMODE_EXCEPTION，那么无须测试 NULL，就能够捕捉和记录这类错误。

6. 使用 PDO::query()方法发送 SQL 命令，会得到 PDOStatement 实例，该对象中含有你想要的查询结果。下面的示例会找出客户列表（按照 ID 排序）中的前 20 位客户：

```
$stmt = $pdo->query(
'SELECT * FROM customer ORDER BY id LIMIT 20');
```

PDO 还提供了一种便于使用的方法：PDO::exec()，该方法不会返回查询结果，仅会返回受操作影响的记录的数量。该方法最适用于执行管理操作，如 ALTER TABLE、DROP TABLE 等。

7. 通过遍历 PDOStatement 实例可以处理查询到的结果。将获取数据的模式设置为 PDO::FETCH_NUM 或 PDO::FETCH_ASSOC，可以返回数字索引数组或关联数组形式的结果。下面的示例使用 while()循环处理获得的结果。得到最后一个结果后，返回的结果为布尔型的 FALSE，该循环会结束运行：

```
while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
  printf('%4d | %20s | %5s' . PHP_EOL, $row['id'],
  $row['name'], $row['level']);
}
```

PDO 的 fetch 操作含有定义迭代方向（即正向或反向）的游标。PDOStatement::fetch()方法的第二个参数可以是任意 PDO::FETCH_ORI_*常量。游标方向包括优先、第一个、最后一个、绝对和相对。默认的游标方向是 PDO::FETCH_ORI_NEXT（代表下一个）。

8. 将获取数据的模式设置为 PDO::FETCH_OBJ，可以使返回结果为 stdClass 实例。你会发现设置为 PDO::FETCH_OBJ 的获取数据模式会为 while()循环提供益处。注意，本例中的 printf()语句用于显示对象的属性，而上一个例子使用 printf()函数显示数组元素。

```
while ($row = $stmt->fetch(PDO::FETCH_OBJ)) {
  printf('%4d | %20s | %5s' . PHP_EOL,
  $row->id, $row->name, $row->level);
}
```

9. 如果你想要在处理查询结果时为指定的类创建实例，可将获取数据的模式设置为

PDO::FETCH_CLASS。同时还必须保证可加载这个类的定义，而且应使用PDO::query()方法设置该类的名称。下面的代码定义了一个名为 Customer 的类，该类含有公用属性$id、$name 和$level。只有将这些属性定义为公用的(public)，获取数据的操作才能正常执行：

```
class Customer
{
  public $id;
  public $name;
  public $level;
}

$stmt = $pdo->query($sql, PDO::FETCH_CLASS, 'Customer');
```

10. 当获取的数据为对象时，可使用 PDOStatement::fetchObject()方法，该技巧比前面步骤 7 介绍的技巧更简单：

```
while ($row = $stmt->fetchObject('Customer')) {
  printf('%4d | %20s | %5s' . PHP_EOL,
  $row->id, $row->name, $row->level);
}
```

11. 还可以使用 PDO::FETCH_INTO 设置，它基本上与 PDO::FETCH_CLASS 设置相同，但是会使用活动的实例而不使用类的定义。每次执行遍历操作的循环，都会向同一个实例中添加当前的信息集合。下面的示例使用了前面步骤 9 介绍的 Customer 类，以及前面步骤 2 定义的数据库参数和 PDO 连接：

```
$cust = new Customer();
while ($stmt->fetch(PDO::FETCH_INTO)) {
  printf('%4d | %20s | %5s' . PHP_EOL,
  $cust->id, $cust->name, $cust->level);
}
```

12. 如果你没有设置 PDO 错误处理模式，那么默认的 PDO 错误处理模式就是PDO::ERRMODE_SILENT。可以将错误处理模式设置为 PDO::ATTR_ERRMODE，将该属性的值设置为 PDO::ERRMODE_WARNING 或 PDO::ERRMODE_EXCEPTION。可通过关联数组形式将错误处理模式设置为 PDO 构造器的第 4 个参数，也可以对已创建好的实例使用 PDO::setAttribute()方法。

13. 假设我们使用了下面的DSN和SQL语句(请注意，下面的SQL语句无法运行！)：

```
$params = [
```

```
    'host' => 'localhost',
    'user' => 'test',
    'pwd' => 'password',
    'db' => 'php7cookbook'
];
$dsn = sprintf('mysql:host=%s;dbname=%s', $params['host'],
$params['db']);
$sql = 'THIS SQL STATEMENT WILL NOT WORK';
```

14. 如果你使用默认的错误处理模式创建 PDO 连接，那么出现错误时能够用于分析出错原因的唯一线索就是 PDO::query()方法返回的布尔型值 FALSE：

```
$pdo1 = new PDO($dsn, $params['user'], $params['pwd']);
$stmt = $pdo1->query($sql);
$row = ($stmt) ? $stmt->fetch(PDO::FETCH_ASSOC) : 'No Good';
```

15. 下面的示例通过构造器方式将错误处理模式设置为 PDO::ATTR_ERRMODE 模式（代表显示错误提示），并将该模式的值设置为 PDO::ERRMODE_WARNING（代表显示错误警告）：

```
$pdo2 = new PDO(
  $dsn,
  $params['user'],
  $params['pwd'],
  [PDO::ATTR_ERRMODE => PDO::ERRMODE_WARNING]);
```

16. 如果你想要将数据库查询操作的准备阶段和执行阶段彻底分隔开，可使用 PDO::prepare()和 PDOStatement::execute()方法。这样就可以将数据库查询语句发送给数据库服务器进行预编译，然后在经检查没有发现语法错误的情况下，重复执行这些数据库查询语句。[1]

17. PDO::prepare()方法的第一个参数是使用占位符替代了实际值的 SQL 语句。这样就会为 PDOStatement::execute()方法提供一组值。PDO 会自动提供数据库

1 每次将数据库查询语句发送给数据库服务器时，都必须检查该查询语句的语法，以确保该查询语句的结构正确并能够执行。这是一个必要的步骤，但也确实增加了一些开销。在执行查询操作时做一次检查是必要的，但如果反复地执行相同的查询操作，仅改变部分参数时还这样做就会带来过多的开销！预处理语句（prepare()和 execute()方法）会在服务器上缓存数据库查询语句的语法和执行过程，而只在服务器和客户端之间传输有变化的参数值，以此来消除这些额外的开销。——译者注

引证，从而确保防止 **SQL 注入**（SQL Injection）。

最佳编程习惯

将外部输入信息与 SQL 语句相结合的任何应用程序都可能会受到 SQL 注入攻击。所有外部输入信息都应先进行适当的过滤和验证，即对这些信息进行消毒。不应直接将外部输入信息放入 SQL 语句中。更确切地说，应在执行阶段使用占位符并提供实际值（经过消毒的信息）。

18. 要反向遍历查询到的结果，可以更改**可滚动游标**（scrollable cursor）的方向。也可以采用更简单的方式，将 ORDER BY 子句中的关键字 ASC 更改为关键字 DESC。下面的代码设置了一个 PDOStatement 对象，并使该对象请求获取一个可滚动的游标：

```
$dsn = sprintf('pgsql:charset=UTF8;host=%s;dbname=%s',
$params['host'], $params['db']);
$opts = [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION];
$pdo = new PDO($dsn, $params['user'], $params['pwd'], $opts);
$sql = 'SELECT * FROM customer '
    . 'WHERE balance > :min AND balance < :max '
    . 'ORDER BY id LIMIT 20';
$stmt = $pdo->prepare($sql, [PDO::ATTR_CURSOR =>
  PDO::CURSOR_SCROLL]);
```

19. 在执行获取数据的操作时还需要设置游标指令。下面的示例获取了查询结果集合中的最后一条记录，然后使游标向后滚动：

```
$stmt->execute(['min' => $min, 'max' => $max]);
$row = $stmt->fetch(PDO::FETCH_ASSOC, PDO::FETCH_ORI_LAST);
do {
  printf('%4d | %20s | %5s | %8.2f' . PHP_EOL,
      $row['id'],
      $row['name'],
      $row['level'],
      $row['balance']);
} while ($row = $stmt->fetch(PDO::FETCH_ASSOC,
  PDO::FETCH_ORI_PRIOR));
```

20. MySQL 和 SQLite 数据库都不支持可滚动的游标！要实现可滚动游标的效果，可对上面的代码做如下更改：

```
$dsn = sprintf('mysql:charset=UTF8;host=%s;dbname=%s',
```

```
$params['host'], $params['db']);
$opts = [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION];
$pdo = new PDO($dsn, $params['user'], $params['pwd'], $opts);
$sql = 'SELECT * FROM customer '
    . 'WHERE balance > :min AND balance < :max '
    . 'ORDER BY id DESC
       . 'LIMIT 20';
$stmt = $pdo->prepare($sql);
while ($row = $stmt->fetch(PDO::FETCH_ASSOC));
printf('%4d | %20s | %5s | %8.2f' . PHP_EOL,
    $row['id'],
    $row['name'],
    $row['level'],
    $row['balance']);
}
```

21. PDO 支持事务。下面的代码将一组 INSERT 命令封装在了一个事务代码块中：

```
try {
    $pdo->beginTransaction();
    $sql = "INSERT INTO customer ('"
    . implode("','", $fields) . "') VALUES (?,?,?,?,?,?)";
    $stmt = $pdo->prepare($sql);
    foreach ($data as $row) $stmt->execute($row);
    $pdo->commit();
} catch (PDOException $e) {
    error_log($e->getMessage());
    $pdo->rollBack();
}
```

22. 为了使代码模块化并具有可重用性，可将 PDO 连接封装在一个独立的类 Application\Database\Connection 中。下面让我们通过构造器创建连接。还可以添加一个静态的 factory() 方法，使我们能够批量创建 PDO 实例：

```
namespace Application\Database;
use Exception;
use PDO;
class Connection
{
   const ERROR_UNABLE = 'ERROR: no database connection';
   public $pdo;
   public function __construct(array $config)
   {
```

```
        if (!isset($config['driver'])) {
           $message = __METHOD__ . ' : '
           . self::ERROR_UNABLE . PHP_EOL;
           throw new Exception($message);
        }
        $dsn = $this->makeDsn($config);
    try {
        $this->pdo = new PDO(
           $dsn,
           $config['user'],
           $config['password'],
           [PDO::ATTR_ERRMODE => $config['errmode']]);
        return TRUE;
    } catch (PDOException $e) {
        error_log($e->getMessage());
        return FALSE;
    }
}

public static function factory(
  $driver, $dbname, $host, $user,
  $pwd, array $options = array())
{
    $dsn = $this->makeDsn($config);

    try {
        return new PDO($dsn, $user, $pwd, $options);
    } catch (PDOException $e) {
        error_log($e->getMessage);
    }
}
```

23. 这个 Connection 类的一个重要组件是一个用于创建 DSN 的泛型方法。要使该方法起作用，只需将 PDODriver 用作前缀，后面添加:，然后通过配置数组添加键/值对。键/值对之间由分号进行分隔。因此还需要使用带截取长度(一个负数)的 substr() 方法去掉键/值对尾部的分号。

```
public function makeDsn($config)
{
    $dsn = $config['driver'] . ':';
    unset($config['driver']);
    foreach ($config as $key => $value) {
      $dsn .= $key . '=' . $value . ';';
```

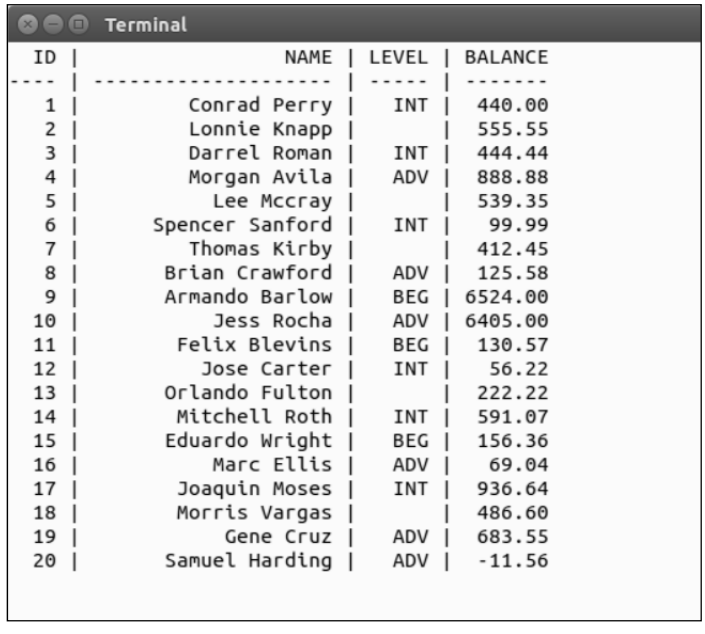
```
    }
    return substr($dsn, 0, -1);
  }
}
```

具体运行情况

先将前面步骤 2 介绍的代码添加到 chap_05_pdo_connect_mysql.php 文件中。为了做这个实验，我们已经创建了一个名为 php7cookbook 的 MySQL 数据库，并创建了一个名为 cook、登录密码为 book 的数据库用户。然后，使用 PDO::query()方法向这个数据库发送一条简单的 SQL 语句。最后，使用存储 SQL 语句处理结果的对象，通过关联数组获取处理结果。注意，应该将这些代码封装在 try {}catch{}代码块中：

```
<?php
$params = [
  'host' => 'localhost',
  'user' => 'test',
  'pwd' => 'password',
  'db' => 'php7cookbook'
];
try {
  $dsn = sprintf('mysql:charset=UTF8;host=%s;dbname=%s',
    $params['host'], $params['db']);
  $pdo = new PDO($dsn, $params['user'], $params['pwd']);
  $stmt = $pdo->query(
    'SELECT * FROM customer ORDER BY id LIMIT 20');
  printf('%4s | %20s | %5s | %7s' . PHP_EOL,
    'ID', 'NAME', 'LEVEL', 'BALANCE');
  printf('%4s | %20s | %5s | %7s' . PHP_EOL,
    '----', str_repeat('-', 20), '-----', '-------');
  while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    printf('%4d | %20s | %5s | %7.2f' . PHP_EOL,
    $row['id'], $row['name'], $row['level'], $row['balance']);
  }
} catch (PDOException $e) {
  error_log($e->getMessage());
} catch (Throwable $e) {
  error_log($e->getMessage());
}
```

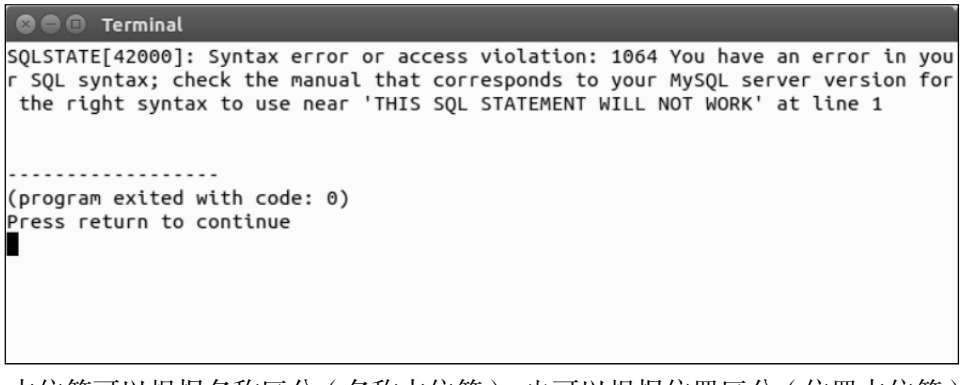
下面是输出结果：



在 PDO 构造器中添加选项，通过该选项将错误处理模式设置为 ERRMODE_EXCEPTION。更改发送给数据的 SQL 语句，并观察出现的错误提示：

```
$opts   = [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION];
$pdo    = new PDO($dsn, $params['user'], $params['pwd'], $opts);
$stmt   = $pdo->query('THIS SQL STATEMENT WILL NOT WORK');
```

下面是输出结果：



占位符可以根据名称区分（名称占位符），也可以根据位置区分（位置占位符）。在准备好的（即经检查未发现语法错误的）SQL 语句中，名称占位符前面会带有:，而且

会在传入 execute() 方法的关联数组中来引用键。在准备好的 SQL 语句中，位置占位符是?。

下面的示例使用名称占位符代表 WHERE 子句中的值：

```
try {
  $dsn = sprintf('mysql:host=%s;dbname=%s',
                 $params['host'], $params['db']);
  $pdo = new PDO($dsn,
                 $params['user'],
                 $params['pwd'],
                 [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
  $sql = 'SELECT * FROM customer '
      . 'WHERE balance < :val AND level = :level '
      . 'ORDER BY id LIMIT 20'; echo $sql . PHP_EOL;
  $stmt = $pdo->prepare($sql);
  $stmt->execute(['val' => 100, 'level' => 'BEG']);
  while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    printf('%4d | %20s | %5s | %5.2f' . PHP_EOL,
          $row['id'], $row['name'], $row['level'], $row['balance']);
  }
} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

下面的示例在 INSERT 操作中使用了位置占位符。注意，将要插入的代表第 4 位用户的数据中可能含有 SQL 注入攻击。还应注意，如果要使用数据库，就必须对相应的 SQL 语法有所了解。本例使用单引号（'）将 MySQL 数据库中的字段名称括起来。

```
$fields = ['name', 'balance', 'email',
           'password', 'status', 'level'];
$data = [
  ['Saleen',0,'saleen@test.com', 'password',0,'BEG'],
  ['Lada',55.55,'lada@test.com', 'password',0,'INT'],
  ['Tonsoi',999.99,'tongsoi@test.com','password',1,'ADV'],
  ['SQL Injection',0.00,'bad','bad',1,
    'BEG\';DELETE FROM customer;--'],
];

try {
  $dsn = sprintf('mysql:host=%s;dbname=%s',
    $params['host'], $params['db']);
```

```
    $pdo = new PDO($dsn,
                   $params['user'],
                   $params['pwd'],
                   [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
    $sql = "INSERT INTO customer ('"
      . implode("','", $fields)
      . "') VALUES (?,?,?,?,?,?)";
    $stmt = $pdo->prepare($sql);
    foreach ($data as $row) $stmt->execute($row);
} catch (PDOException $e) {
    echo $e->getMessage();
} catch (Throwable $e) {
    echo $e->getMessage();
}
```

要使用已命名参数的准备好的语句做实验，可在上例使用的 SQL 语句中添加一个 WHERE 子句，以便找出 balance 字段（代表余额）的值低于某个值，且 level 字段（代表等级）的值为 BEG、INT 或 ADV（分别代表初级、中级和高级）的记录（客户）。本例没有使用 PDO::query()方法，而使用了 PDO::prepare()方法。在获取结果前，必须先调用 PDOStatement::execute()方法，以便提供 balance 字段和 level 字段的实际值：

```
$sql = 'SELECT * FROM customer '
    . 'WHERE balance < :val AND level = :level '
    . 'ORDER BY id LIMIT 20';
$stmt = $pdo->prepare($sql);
$stmt->execute(['val' => 100, 'level' => 'BEG']);
```

下面是输出结果：

```
Terminal
 ID |                 NAME | LEVEL | BALANCE
---- | -------------------- | ----- | -------
 25 |        Rhonda Kinney |   BEG |   46.61
 45 |      Wilfredo Taylor |   BEG |   25.11
 57 |       Garrett Campos |   BEG |    9.47
 88 |               Obama  |   BEG |    0.00
 92 |         C.T. Russell |   BEG |    0.00


-----------------
(program exited with code: 0)
Press return to continue
```

如果你不想在调用 PDOStatement::execute()方法时提供参数，也可以绑定参

数。这能够将变量分配给占位符。在执行程序时，变量的当前值会被用作参数。

下面的示例将变量$min、$max 和$level 与准备好的 SQL 语句绑定起来：

```
$min = 0;
$max = 0;
$level = ' ';

try {
  $dsn = sprintf('mysql:host=%s;dbname=%s', $params['host'],
    $params['db']);
  $opts = [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION];
  $pdo  = new PDO($dsn, $params['user'], $params['pwd'], $opts);
  $sql  = 'SELECT * FROM customer '
       . 'WHERE balance > :min '
       . 'AND balance < :max AND level = :level '
       . 'ORDER BY id LIMIT 20';
  $stmt = $pdo->prepare($sql);
  $stmt->bindParam('min', $min);
  $stmt->bindParam('max', $max);
  $stmt->bindParam('level', $level);

  $min = 5000;
  $max = 10000;
  $level = 'ADV';
  $stmt->execute();
  showResults($stmt, $min, $max, $level);

  $min = 0;
  $max = 100;
  $level = 'BEG';
  $stmt->execute();
  showResults($stmt, $min, $max, $level);

} catch (PDOException $e) {
  echo $e->getMessage();
} catch (Throwable $e) {
  echo $e->getMessage();
}
```

当这些变量的值发生改变时，下次调用 execute()方法时会反映出对 SQL 语句执行结果产生的影响。

最佳编程习惯

将 PDO::query()方法用于仅会被执行一次的数据库命令。当需要使用不同的值反复执行同一条 SQL 语句时，应使用 PDO::prepare()和 PDOStatement::execute()方法。

扩展

- 要了解各种特定于供应商的 PDO 驱动程序的语法和独特操作，请浏览 http://php.net/manual/en/pdo.drivers.php。
- 要全面了解预定义的 PDO 常量（包括获取数据的模式、游标的方向和属性），请浏览 http://php.net/manual/en/pdo.constants.php。

创建 OOP 式的 SQL 语句生成器

PHP 7 实现了**上下文敏感词法分析器**（context sensitive lexer）。该分析器的作用是在上下文允许的情况下，允许开发者使用 PHP 7 的保留字。因此，在创建面向对象的 SQL 语句生成器时，可以将 and、or、not 等保留字用作方法的名称。

具体处理过程

1. 本例定义了 Application\Database\Finder 类。在这个类中，我们定义了与我们最喜欢使用的 SQL 操作相匹配的方法：

```
namespace Application\Database;
class Finder
{
  public static $sql        = ' ';
  public static $instance   = NULL;
  public static $prefix     = ' ';
  public static $where      = array();
  public static $control    = [' ', ' '];

    // $a == 表的名称
    // $cols = 字段的名称
```

```
public static function select($a, $cols = NULL)
{
  self::$instance = new Finder();
  if ($cols) {
       self::$prefix = 'SELECT ' . $cols . ' FROM ' . $a;
  } else {
    self::$prefix = 'SELECT * FROM ' . $a;
  }
  return self::$instance;
}

public static function where($a = NULL)
{
   self::$where[0] = ' WHERE ' . $a;
   return self::$instance;
}

public static function like($a, $b)
{
   self::$where[] = trim($a . ' LIKE ' . $b);
   return self::$instance;
}

public static function and($a = NULL)
{
   self::$where[] = trim('AND ' . $a);
   return self::$instance;
}

public static function or($a = NULL)
{
   self::$where[] = trim('OR ' . $a);
   return self::$instance;
}

public static function in(array $a)
{
   self::$where[] = 'IN ( ' . implode(',', $a) . ' )';
   return self::$instance;
}

public static function not($a = NULL)
{
    self::$where[] = trim('NOT ' . $a);
    return self::$instance;
}
```

```
    public static function limit($limit)
    {
        self::$control[0] = 'LIMIT ' . $limit;
        return self::$instance;
    }

    public static function offset($offset)
    {
        self::$control[1] = 'OFFSET ' . $offset;
        return self::$instance;
    }

    public static function getSql()
    {
      self::$sql = self::$prefix
         . implode('  ', self::$where)
                  . '  '
                  . self::$control[0]
                  . '  '
                  . self::$control[1];
      preg_replace('/  /', '  ', self::$sql);
      return trim(self::$sql);
    }
}
```

2. 每个用于生成 SQL 语句的函数都会返回相同的属性：`$instance`。这使我们能够对这段代码使用连贯接口（fluent interface）：

```
$sql = Finder::select('project')->where('priority > 9') … etc.
```

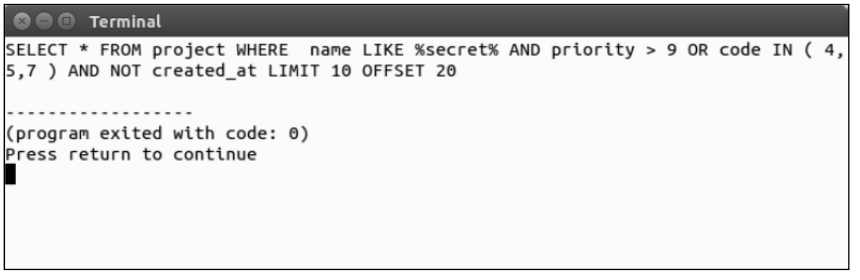
具体运行情况

将前面介绍的代码添加到 `Application\Database` 文件夹中的 `Finder.php` 文件中。这样就可以创建调用程序 `chap_05_oop_query_builder.php` 了，该程序会初始化第 1 章介绍过的类自动加载器。这样就可以通过调用 `Finder::select()` 方法生成能够产生 SQL 语句字符串的对象：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Finder;

$sql = Finder::select('project')
  ->where()
  ->like('name', '%secret%')
```

```
    ->and('priority > 9')
    ->or('code')->in(['4', '5', '7'])
    ->and()->not('created_at')
    ->limit(10)
    ->offset(20);

echo Finder::getSql();
```

下面是输出结果：



扩展

要详细了解上下文敏感词法分析器，请浏览 `https://wiki.php.net/rfc/context_sensitive_lexer`。

处理分页

通过分页可以将数据库查询结果划分为多个限定范围的子集。这样做的目的通常是为了便于查看，但也可以轻松地将之应用于其他情况。乍看之下，`LimitIterator` 类非常适用于分页。然而，在查询结果的内容非常多的情况下，`LimitIterator` 类就不太适用了。因为这需要将查询结果的全部内容存储到一个内部迭代器中，因而很可能会超出可用内存的上限。`LimitIterator` 类构造器的第二个和第三个参数分别为要处理的记录数量和开始执行处理操作的位置。这与我们想要的分页解决方案相符，即与 SQL 语句中的 `LIMIT` 和 `OFFSET` 子句类似。

具体处理过程

1. 先创建 `Application\Database\Paginate` 类，并使用该类保存分页逻辑。

在这个类中添加$sql、$page 和$linesPerPage 属性，并使用这些属性代表与分页操作有关的设置：

```
namespace Application\Database;

class Paginate
{

  const DEFAULT_LIMIT = 20;
  const DEFAULT_OFFSET = 0;
  protected $sql;
  protected $page;
  protected $linesPerPage;

}
```

2. 然后定义__construct()方法，使用该方法将基础的 SQL 语句、当前页码和每页显示的记录数量接收为参数。然后需要重构 SQL 语句的字符串，或者在 SQL 语句中添加 LIMIT 和 OFFSET 子句。

3. 在这个构造器方法中,需要使用页数和每页显示的记录数量计算出要处理的记录数量。还需要检查 LIMIT 和 OFFSET 子句是否已经被包含到了 SQL 语句中。最后，我们需要将每页显示的记录数量用作 LIMIT 子句设置（要处理的记录数量）和重新计算的 OFFSET 子句设置（开始执行处理操作的位置），以修改 SQL 语句：

```
public function __construct($sql, $page, $linesPerPage)
{
  $offset = $page * $linesPerPage;
  switch (TRUE) {
    case (stripos($sql, 'LIMIT') && strpos($sql, 'OFFSET')) :
      // 无须执行具体操作
      break;
    case (stripos($sql, 'LIMIT')) :
      $sql .= ' LIMIT ' . self::DEFAULT_LIMIT;
      break;
    case (stripos($sql, 'OFFSET')) :
      $sql .= ' OFFSET ' . self::DEFAULT_OFFSET;
      break;
    default :
      $sql .= ' LIMIT ' . self::DEFAULT_LIMIT;
      $sql .= ' OFFSET ' . self::DEFAULT_OFFSET;
```

```
            break;
        }
        $this->sql = preg_replace('/LIMIT \d+.*OFFSET \d+/Ui',
            'LIMIT ' . $linesPerPage . ' OFFSET ' . $offset,
            $sql);
    }
```

4. 这样我们就可以使用前面介绍的 Application\Database\Connection 类，执行数据库查询操作了。

5. 在我们刚刚创建的 Paginate 类中添加 paginate() 方法，该方法会接收 Connection 实例并将其用作参数。还需要为这个 PDO 设置获取数据的模式，以及在使用 prepare() 和 execute() 方法执行该 SQL 语句时使用的参数：

```
use PDOException;
public function paginate(
  Connection $connection,
  $fetchMode,
  $params = array())
  {
  try {
    $stmt = $connection->pdo->prepare($this->sql);
    if (!$stmt) return FALSE;
    if ($params) {
      $stmt->execute($params);
    } else {
      $stmt->execute();
    }
    while ($result = $stmt->fetch($fetchMode)) yield $result;
  } catch (PDOException $e) {
    error_log($e->getMessage());
    return FALSE;
  } catch (Throwable $e) {
    error_log($e->getMessage());
    return FALSE;
  }
}
```

6. 为前面介绍的查询操作生成器提供支持也是不错的思路。这会使更新 LIMIT 和 OFFSET 子句设置变得更容易得多。要为 Application\Database\Finder 类提供支持，只需使用该类并修改__construct() 方法，从而检查收到的 SQL 语句是否为该

类的实例：

```
if ($sql instanceof Finder) {
  $sql->limit($linesPerPage);
  $sql->offset($offset);
  $this->sql = $sql::getSql();
} elseif (is_string($sql)) {
  switch (TRUE) {
    case (stripos($sql, 'LIMIT')
    && strpos($sql, 'OFFSET')) :
        // 此处的具体代码已经在前面的步骤 3 中介绍过
    }
}
```

7. 需要做的最后一件事情，是在 Finder 类中添加 getSql()方法，以便检查收到的 SQL 语句的语法是否正确：

```
public function getSql()
{
  return $this->sql;
}
```

具体运行情况

将前面介绍的代码添加到 Application/Database 文件夹中的 Paginate.php 文件中。然后创建调用程序 chap_05_pagination.php，该程序会初始化第 1 章介绍过的类自动加载器：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
define('LINES_PER_PAGE', 10);
define('DEFAULT_BALANCE', 1000);
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
```

然后，使用 use 语句为 Application\Database\Finder、Connection 和 Paginate 类起别名，创建 Application\Database\Connection 类的实例，并使用 Finder 类生成 SQL 语句：

```
use Application\Database\ { Finder, Connection, Paginate};
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
$sql = Finder::select('customer')->where('balance < :bal');
```

这样我们就可以通过$_GET 参数获得页数和余额，并创建 Paginate 对象，结束

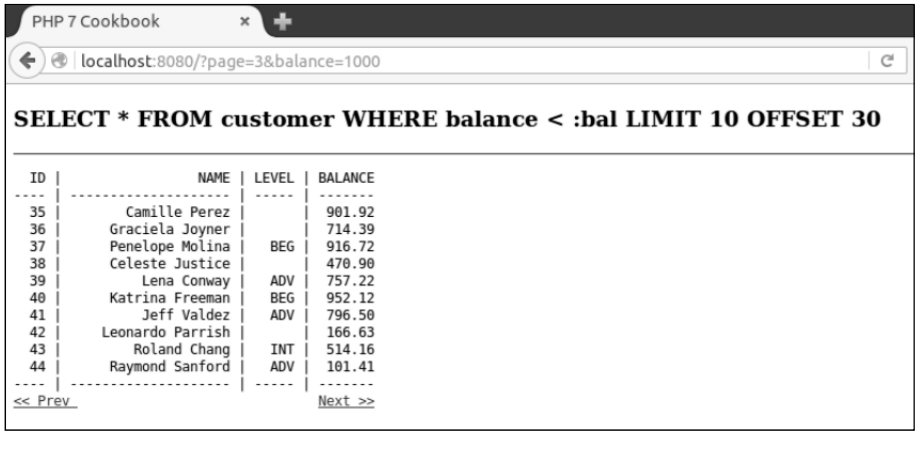
这段 PHP 代码：

```
$page = (int) ($_GET['page'] ?? 0);
$bal = (float) ($_GET['balance'] ?? DEFAULT_BALANCE);
$paginate = new Paginate($sql::getSql(), $page, LINES_PER_PAGE);
?>
```

在这段脚本的输出部分中，只需使用一个简单 `foreach()` 循环遍历每页中的内容：

```
<h3><?= $paginate->getSql(); ?></h3>
<hr>
<pre>
<?php
printf('%4s | %20s | %5s | %7s' . PHP_EOL,
  'ID', 'NAME', 'LEVEL', 'BALANCE');
printf('%4s | %20s | %5s | %7s' . PHP_EOL,
  '----', str_repeat('-', 20), '-----', '-------');
foreach ($paginate->paginate($conn, PDO::FETCH_ASSOC,
  ['bal' => $bal]) as $row) {
  printf('%4d | %20s | %5s | %7.2f' . PHP_EOL,
      $row['id'],$row['name'],$row['level'],$row['balance']);
}
printf('%4s | %20s | %5s | %7s' . PHP_EOL,
  '----', str_repeat('-', 20), '-----', '-------');
?>
<a href="?page=<?= $page - 1; ?>&balance=<?= $bal ?>">
<< Prev </a>  
<a href="?page=<?= $page + 1; ?>&balance=<?= $bal ?>">
Next >></a>
</pre>
```

下面是输出结果中第 3 页的内容，其中每条记录的余额字段值都小于 1000：



扩展

要详细了解 LimitIterator 类，请浏览 http://php.net/manual/en/class.limititerator.php。

定义与数据库表匹配的实体

PHP 开发者中非常常见的一个编程习惯就是创建用于代表数据库中表的类。这些类通常称为**实体**类，并且构成了**领域模型**软件设计模式的核心。

具体处理过程

1. 先为一组实体类创建一些常见的功能，其中包括一些常见的属性和方法。我们会将这些属性和方法放入 Application\Entity\Base 类中，以后创建的所有实体类都会是 Base 类的子类。

2. 为了理解实体类的概念，我们为每个实体类都定义两个属性：$mapping（稍后介绍）和$id（包括与该属性对应的读取器和设置器）：

```
namespace Application\Entity;

class Base
{
  protected $id = 0;
  protected $mapping = ['id' => 'id'];

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

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

3. 定义 arrayToEntity()方法是不错的思路，该方法可以将数组转换为实体类的实例。同理，定义 entityToArray()方法也是很好的思路，该方法可以将实体类的

实例转换为数组。这些方法实现的处理过程通常称为水合。因为这些方法应该具有通用性，所以最好将它们放置在 Base 类中。

4. 在下面的方法中，$mapping 属性用于将数据库中字段的名称与对象的属性的名称对应起来。arrayToEntity()方法用于通过数组为实例赋值。可以将 arrayToEntity()方法定义为静态方法，以便在不创建实例的情况下也能够调用它：

```
public static function arrayToEntity($data, Base $instance)
{
  if ($data && is_array($data)) {
    foreach ($instance->mapping as $dbColumn => $propertyName) {
      $method = 'set' . ucfirst($propertyName);
      $instance->$method($data[$dbColumn]);
    }
    return $instance;
  }
  return FALSE;
}
```

5. 使用 entityToArray()方法可以通过实例属性的当前值创建数组：

```
public function entityToArray()
{
  $data = array();
  foreach ($this->mapping as $dbColumn => $propertyName) {
    $method = 'get' . ucfirst($propertyName);
    $data[$dbColumn] = $this->$method() ?? NULL;
  }
  return $data;
}
```

6. 要创建指定的实体，还需要了解你想要为之建模的数据库表的结构。按照数据库中的字段创建类的属性，为这些属性赋予的初始值应与数据库字段的数据类型对应。

7. 下面的示例使用了 customer 表。下面是在 MySQL 数据库中创建该表的 CREATE 语句，它展示了这个表的数据结构：

```
CREATE TABLE 'customer' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'name' varchar(256) CHARACTER SET latin1 COLLATE
    latin1_general_cs NOT NULL,
  'balance' decimal(10,2) NOT NULL,
  'email' varchar(250) NOT NULL,
  'password' char(16) NOT NULL,
  'status' int(10) unsigned NOT NULL DEFAULT '0',
```

```
  'security_question' varchar(250) DEFAULT NULL,
  'confirm_code' varchar(32) DEFAULT NULL,
  'profile_id' int(11) DEFAULT NULL,
  'level' char(3) NOT NULL,
  PRIMARY KEY ('id'),
  UNIQUE KEY 'UNIQ_81398E09E7927C74' ('email')
);
```

8. 现在我们应该创建类的属性，这也是确定相应表结构的好时机。本例使用 TABLE_NAME 类常量（代表数据库表的名称）：

```
namespace Application\Entity;

class Customer extends Base
{
  const TABLE_NAME = 'customer';
  protected $name = ' ';
  protected $balance = 0.0;
  protected $email = ' ';
  protected $password = ' ';
  protected $status = ' ';
  protected $securityQuestion = ' ';
  protected $confirmCode = ' ';
  protected $profileId = 0;
  protected $level = ' ';
}
```

9. 将这些属性定义为受保护的（protected）是一种最佳编程习惯。为了访问这些属性，我们需要设计公用方法，以便读取和设置这些属性。这也是使用 PHP 7 新增功能处理返回值数据类型的好时机。

10. 下面的代码为$name 和$balance 变量（分别代表 name 和 balance 字段）定义了读取器和设置器。请尝试补全这些方法中缺少的代码：

```
  public function getName() : string
  {
    return $this->name;
  }
  public function setName($name)
  {
    $this->name = $name;
  }
  public function getBalance() : float
  {
    return $this->balance;
```

```
  }
  public function setBalance($balance)
  {
    $this->balance = (float) $balance;
  }
}
```

不应在设置器中检查收到数据的数据类型，因为 RDBMS 数据库查询操作返回的数据都是字符型的。

11. 如果属性的名称没有精确地与相应的数据库字段的名称相匹配，就应该考虑创建用于描述对应关系的属性（即存储在数组中的键/值对），其中的键代表数据库字段的名称，而值代表类中的属性名称。

12. 注意，有三个属性（$securityQuestion、$confirmCode 和 $profileId）没有精确地与数据库字段（security_question、confirm_code 和 profile_id）相匹配。$mapping 属性会确保转换过程正确地执行：

```
protected $mapping = [
  'id'                  => 'id',
  'name'                => 'name',
  'balance'             => 'balance',
  'email'               => 'email',
  'password'            => 'password',
  'status'              => 'status',
  'security_question'   => 'securityQuestion',
  'confirm_code'        => 'confirmCode',
  'profile_id'          => 'profileId',
  'level'               => 'level'
];
```

具体运行情况

将步骤 2、4 和 5 介绍的代码添加到 Application/Entity 文件夹中的 Base.php 文件中。将步骤 8 至步骤 12 结合的代码添加到 Application/Entity 文件夹中的 Customer.php 文件中。你还需要为步骤 10 中没有介绍的属性（email、password、status、securityQuestion、confirmCode、profileId 和 level）创建读取器和设置器。

这样就可以创建调用程序 create a chap_05_matching_entity_to_table.php 了，该程序会初始化第 1 章介绍的类自动加载器，使用 use 语句为 Application\Database\Connection 类和新建的 Application\Entity\Customer 类起别名：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Connection;
use Application\Entity\Customer;
```

然后，创建数据库连接，使用这个连接随机获取一条客户记录的数据：

```
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
$id = rand(1,79);
$stmt = $conn->pdo->prepare(
  'SELECT * FROM customer WHERE id = :id');
$stmt->execute(['id' => $id]);
$result = $stmt->fetch(PDO::FETCH_ASSOC);
```

最后，通过获取的数组创建新的 Customer 实体类实例，并使用 var_dump()函数查看结果：

```
$cust = Customer::arrayToEntity($result, new Customer());
var_dump($cust);
```

下面是这段代码的输出结果：

扩展

介绍领域模型的著作很多。Martin Fowler 撰写的 *Patterns of Enterprise Application Architecture*（`http://martinfowler.com/books/eaa.html`）可能是其中最具影响力的著作。InfoQ 网站的 *Domain Driven Design Quickly* 也很优秀，而且可以免费下载（`http://www.infoq.com/minibooks/domain-driven-design-quickly`）。

将实体类的数据类型设置为与 RDBMS 查询操作匹配的数据类型

大多数应用于商业的 RDBMS 都是在过程式程序设计大行其道的时代发展起来的。可以将 RDBMS 世界视为是二维的、方形的，并且是面向过程的。与此相对应，可将实体视为是三维的、圆形的，并且是面向对象的。这样应该能帮助你理解我们为何将 RDBMS 查询操作结果的数据类型设置到实体类实例的迭代中。

关系模型是现代 RDBMS 的基础，这一理论最早于 1969 年由埃德加·弗兰克·科德提出。第一个商业 RDBMS 发展于 20 世纪 70 年代中后期。换言之，RDBMS 技术已经有超过 40 年的历史！

具体处理过程

1. 先创建一个类，使用该类存储用于执行查询操作的逻辑。如果你使用领域模型，可以将该类命名为 repository（代表资源库）；为了简洁和通用起见，也可以将这个新建的类命名为 `Application\Database\CustomerService`。这个类将会接收 `Application\Database\Connection` 类的实例作为其参数：

```
namespace Application\Database;

use Application\Entity\Customer;

class CustomerService
{
    protected $connection;
```

```
    public function __construct(Connection $connection)
    {
      $this->connection = $connection;
    }
}
```

2. 定义 fetchById()方法，使该方法接收客户的 ID 作为参数，并返回单个的 Application\Entity\Customer 实例或布尔型值 FALSE(代表获取客户 ID 的操作失败)。乍看之下，使用 PDOStatement::fetchObject()方法并将实体类设置为一个参数可以轻松做到这一点：

```
public function fetchById($id)
{
  $stmt = $this->connection->pdo
              ->prepare(Finder::select('customer')
              ->where('id = :id')::getSql());
  $stmt->execute(['id' => (int) $id]);
  return $stmt->fetchObject('Application\Entity\Customer');
}
```

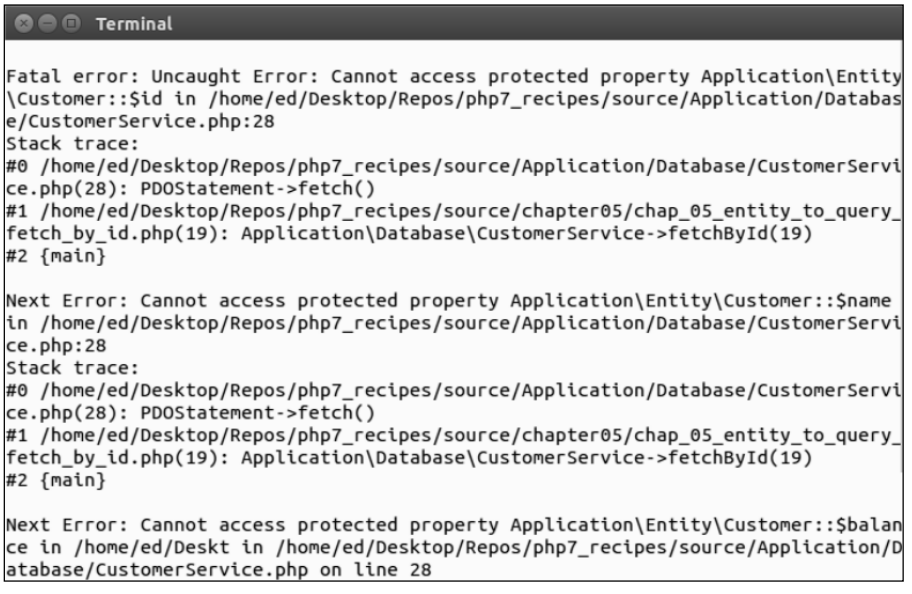
然而，这个操作是包含危险的，即 fetchObject()方法实际上会在构造器被调用前为属性赋值(即使是在将属性设置为受保护的情况下)! 因此，构造器可能会无意中重写这些值。如果你没有定义构造器或者你能够承受这个危险带来的损失，就没有问题。否则，由此开始在 RDBMS 查询操作和 OOP 结果之间创建合适关联纽带的工作会变得更加艰难。

3. 另一种定义 fetchById()方法的方式是，先创建实体类的实例，从而运行它的构造器，并将获取数据的模式设置为 PDO::FETCH_INTO：

```
public function fetchById($id)
{
  $stmt = $this->connection->pdo
              ->prepare(Finder::select('customer')
              ->where('id = :id')::getSql());
  $stmt->execute(['id' => (int) $id]);
  $stmt->setFetchMode(PDO::FETCH_INTO, new Customer());
  return $stmt->fetch();
}
```

4. 然而我们又遇到了一个问题：与 fetchObject()方法不同，fetch()方法无

法重写受保护的属性；如果你尝试使用该方法这样做，就会看到下面的错误提示。这意味着我们要么将所有属性都定义为公用的，要么就使用其他方式处理该问题。



5. 我们可考虑的最后一种方式，是通过数组获取操作结果，然后通过手动方式将其融合到实体对象中。虽然该方式的性能稍差，但它允许所有潜在的实体构造器都正常运行，并能够使属性被安全地定义为私有的或受保护的：

```
public function fetchById($id)
{
  $stmt = $this->connection->pdo
              ->prepare(Finder::select('customer')
              ->where('id = :id')::getSql());
  $stmt->execute(['id' => (int) $id]);
  return Customer::arrayToEntity(
    $stmt->fetch(PDO::FETCH_ASSOC));
}
```

6. 要处理能够生成多条结果的查询操作，只需迭代已创建好的实体对象。下面的示例实现了 `fetchByLevel()` 方法，该方法会以 `Application\Entity\Customer` 实例的形式返回指定等级中的所有客户：

```
public function fetchByLevel($level)
{
  $stmt = $this->connection->pdo->prepare(
```

```
            Finder::select('customer')
            ->where('level = :level')::getSql());
    $stmt->execute(['level' => $level]);
    while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
      yield Customer::arrayToEntity($row, new Customer());
    }
  }
```

7. 我们想要实现的下一个方法是 save()。但在编写该方法前应思考一下，如果执行了插入操作应该返回哪个值。

8. 通常来说，执行插入操作后，应返回新建的实体对象。乍看之下，便捷的 PDO::lastInsertId()方法可以做到这一点。然而，仔细阅读 PHP 文档后，你会发现并非所有数据库扩展都支持该功能，而且支持该功能的数据库扩展对该功能的实现也不一致。因此，使用 ID 之外的另一种具有唯一性的字段识别新建的客户，是一种很好的思路。

9. 下面的示例选择使用 email 字段，因此需要实现 fetchByEmail()方法以提供相应的服务：

```
public function fetchByEmail($email)
{
  $stmt = $this->connection->pdo->prepare(
    Finder::select('customer')
    ->where('email = :email')::getSql());
  $stmt->execute(['email' => $email]);
  return Customer::arrayToEntity(
    $stmt->fetch(PDO::FETCH_ASSOC), new Customer());
}
```

10. 现在可以定义 save()方法了。无须使该方法识别出插入操作和更新操作，只需使该方法在查明 ID 已经存在的情况后执行更新操作，否则执行插入操作。

11. 先编写 save()方法的基础代码，该方法会将 Customer 实体对象（代表客户）接收为参数，然后使用 fetchById()方法查明这条客户记录是否已经存在。如果这条记录已经存在，那么它就会调用 doUpdate()方法执行更新操作；否则，它就会调用 doInsert()方法执行插入操作：

```
public function save(Customer $cust)
{
  // 检查客户的 ID 是否大于 0，以及客户记录是否存在
  if ($cust->getId() && $this->fetchById($cust->getId())) {
    return $this->doUpdate($cust);
```

```
    } else {
      return $this->doInsert($cust);
    }
  }
```

12. 然后定义 doUpdate()方法，应使该方法将 Customer 实体对象中的属性保存到一个数组中，创建初始的 SQL 语句并调用 flush()方法，SQL 语句会将这些数据添加到数据库中。我们不想更新 ID 字段的内容，因为该字段是主键。我们还需要设定需要更新的记录所处的位置，这意味着需要在 SQL 语句中添加 WHERE 子句：

```
  protected function doUpdate($cust)
  {
    // 获取属性值并将这些数据保存在数组中
    $values = $cust->entityToArray();
    // 创建 SQL 语句
    $update = 'UPDATE ' . $cust::TABLE_NAME;
    $where = ' WHERE id = ' . $cust->getId();
    // 使 ID 字段值复位，因为我们不想更新该字段的值
    unset($values['id']);
    return $this->flush($update, $values, $where);
  }
```

13. 除了初始的 SQL 语句需要以 INSERT INTO 开头，且对数组元素 id 执行复位操作的位置不同外，定义 doInsert()方法的方式与定义 doUpdate()方法的方式类似。之所以也在 doInsert()方法中对数组元素 id 执行复位操作，是因为我们希望该属性由数据库自动生成。如果执行插入操作的 SQL 语句成功执行了，我们就可以使用刚刚定义的 fetchByEmail()方法，插入新的客户记录并返回创建好的实例：

```
  protected function doInsert($cust)
  {
    $values = $cust->entityToArray();
    $email = $cust->getEmail();
    unset($values['id']);
    $insert = 'INSERT INTO ' . $cust::TABLE_NAME . ' ';
    if ($this->flush($insert, $values)) {
      return $this->fetchByEmail($email);
    } else {
      return FALSE;
    }
  }
```

14. 最后，我们来定义 flush()方法，该方法完成了 SQL 语句语法检查和真正执行的工作：

```
protected function flush($sql, $values, $where = ' ')
{
  $sql .= ' SET ';
  foreach ($values as $column => $value) {
    $sql .= $column . ' = :' . $column . ',';
  }
  // 去掉 SQL 语句尾部的逗号 ( , )
  $sql = substr($sql, 0, -1) . $where;
  $success = FALSE;
  try {
    $stmt = $this->connection->pdo->prepare($sql);
    $stmt->execute($values);
    $success = TRUE;
  } catch (PDOException $e) {
    error_log(__METHOD__ . ':' . __LINE__ . ':'
    . $e->getMessage());
    $success = FALSE;
  } catch (Throwable $e) {
    error_log(__METHOD__ . ':' . __LINE__ . ':'
    . $e->getMessage());
    $success = FALSE;
  }
  return $success;
}
```

15. 为了使该程序完美无缺，还需要定义 remove()方法，使该方法能够从数据库中删除客户的记录。像先前介绍的 save()方法一样，可使用 fetchById()方法确保删除操作的成功执行：

```
public function remove(Customer $cust)
{
  $sql = 'DELETE FROM ' . $cust::TABLE_NAME . ' WHERE id = :id';
  $stmt = $this->connection->pdo->prepare($sql);
  $stmt->execute(['id' => $cust->getId()]);
  return ($this->fetchById($cust->getId())) ? FALSE : TRUE;
}
```

具体运行过程

将前面步骤 1 至步骤 5 介绍的代码添加到 Application/Database 文件夹中的 CustomerService.php 文件中。定义调用程序 chap_05_entity_to_query.php。使该调用程序初始化类自动加载器，以便能够使用合适的类：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Connection;
use Application\Database\CustomerService;
```

现在可以创建一个代表服务的实例，并以随机方式获取一位客户的数据。这样该服务会将一个代表客户的实体对象作为结果返回：

```
// 获取代表服务的实例
$service = new CustomerService(new Connection(
                        include __DIR__ . DB_CONFIG_FILE));

echo "\nSingle Result\n";
var_dump($service->fetchById(rand(1,79)));
```

下面是输出结果：

将前面步骤 6 至步骤 15 中介绍的代码添加到这个代表服务的类中。将下面的数据添加到调用程序 chap_05_entity_to_query.php 中，然后使用这些数据创建一个 Customer 实体对象：

```
// 示例数据
$data = [
  'name'                => 'Doug Bierer',
  'balance'             => 326.33,
  'email'               => 'doug' . rand(0,999) . '@test.com',
  'password'            => 'password',
  'status'              => 1,
  'security_question'   => 'Who\'s on first?',
  'confirm_code'        => 12345,
  'level'               => 'ADV'
];

// 创建新的 Customer 对象
$cust = Customer::arrayToEntity($data, new Customer());
```

然后，在调用 save()方法之前和之后分别检查这条客户记录的 ID：

```
echo "\nCustomer ID BEFORE Insert: {$cust->getId()}\n";
$cust = $service->save($cust);
echo "Customer ID AFTER Insert: {$cust->getId()}\n";
```

最后，修改这条客户记录中 balance 字段的值，然后再次调用 save()方法并观察得到的结果：

```
echo "Customer Balance BEFORE Update: {$cust->getBalance()}\n";
$cust->setBalance(999.99);
$service->save($cust);
echo "Customer Balance AFTER Update: {$cust->getBalance()}\n";
var_dump($cust);
```

下面是运行该调用程序获得的结果：

```
Terminal

Customer ID BEFORE Insert: 0
Customer ID AFTER Insert: 111
Customer Balance BEFORE Update: 326.33
Customer Balance AFTER Update: 999.99
object(Application\Entity\Customer)#7 (12) {
  ["name":protected]=>
  string(11) "Doug Bierer"
  ["balance":protected]=>
  float(999.99)
  ["email":protected]=>
  string(26) "doug176@unlikelysource.com"
  ["password":protected]=>
  string(8) "password"
  ["status":protected]=>
  int(1)
  ["securityQuestion":protected]=>
  string(15) "Who's on first?"
  ["confirmCode":protected]=>
  string(5) "12345"
  ["profileId":protected]=>
  int(0)
  ["level":protected]=>
  string(3) "ADV"
```

扩展

要详细了解关系模型，请浏览 https://en.wikipedia. org/wiki/Relational_model。要详细了解 RDBMS，请浏览 https://en.wikipedia.org/wiki/Relational_database_management_system。要了解 PDOStatement::fetchObject()方法为属性赋值的方式（以及是如何在构造器被调用前为属性赋值的），请参阅 php.net 网站介绍 fetchObject()方法的参考文档（http://php.net/manual/en/pdostatement.fetchobject.php）。

在查询结果中嵌入二次查询操作

在为实体类之间建立联系的过程中，让我们先将目光投向通过嵌入必要代码执行二次查询操作的方式。执行这类查询操作的一个典型案例是，在显示某位客户的信息时，为了获取这位客户购买商品的列表，使查看逻辑执行第二次查询操作。

这种方式的好处是，在实际的查看逻辑被运行前，二次查询操作会一直被推迟执行。这最终会使程序的性能曲线变得更加平滑，因为这会使查询客户信息操作造成的工作负荷，与查询客户已购买商品信息造成的工作负荷更均匀地分布。这种方式的另一个优点是，避开了大量的 JOIN 子句以及该子句带来的冗余数据。

具体处理过程

1. 先定义一个函数，使该函数能够根据客户的 ID 找到客户的记录。因为本例只是进行演示，所以仅会通过将获取数据的模式设置为 PDO::FETCH_ASSOC 来获取一个数组。本例会继续使用第 2 章介绍的 Application\Database\Connection 类：

```
function findCustomerById($id, Connection $conn)
{
  $stmt = $conn->pdo->query(
    'SELECT * FROM customer WHERE id = ' . (int) $id);
  $results = $stmt->fetch(PDO::FETCH_ASSOC);
  return $results;
}
```

2. 然后分析 purchases 表（其中存储了购买商品的信息）的数据结构，查明 customer 表和 product 表是以怎样的方式关联的。通过下面用于创建 purchases 表的 CREATE 语句，可以查明 customer_id 和 product_id 这两个外键建立了 customer 表和 product 表之间的联系：

```
CREATE TABLE 'purchases' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'transaction' varchar(8) NOT NULL,
  'date' datetime NOT NULL,
  'quantity' int(10) unsigned NOT NULL,
  'sale_price' decimal(8,2) NOT NULL,
  'customer_id' int(11) DEFAULT NULL,
  'product_id' int(11) DEFAULT NULL,
  PRIMARY KEY ('id'),
  KEY 'IDX_C3F3' ('customer_id'),
  KEY 'IDX_665A' ('product_id'),
  CONSTRAINT 'FK_665A' FOREIGN KEY ('product_id')
  REFERENCES 'products' ('id'),
  CONSTRAINT 'FK_C3F3' FOREIGN KEY ('customer_id')
  REFERENCES 'customer' ('id')
);
```

3. 扩展前面创建的 findCustomerById() 函数，以匿名函数的形式定义二次查询操作，从而使该匿名函数能够在查看脚本中执行。这个匿名函数被赋予了数组元素 $results['purchases']：

```
function findCustomerById($id, Connection $conn)
{
  $stmt = $conn->pdo->query(
      'SELECT * FROM customer WHERE id = ' . (int) $id);
  $results = $stmt->fetch(PDO::FETCH_ASSOC);
  if ($results) {
    $results['purchases'] =
      // 定义二次查询操作
      function ($id, $conn) {
        $sql = 'SELECT * FROM purchases AS u '
          . 'JOIN products AS r '
          . 'ON u.product_id = r.id '
          . 'WHERE u.customer_id = :id '
          . 'ORDER BY u.date';
        $stmt = $conn->pdo->prepare($sql);
```

```
            $stmt->execute(['id' => $id]);
            while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
              yield $row;
            }
        };
    }
    return $results;
}
```

4. 如果成功获取了客户的信息并将它们存储到查看逻辑中的$results 数组中，那么只需遍历该匿名函数的返回值。下面的示例会以随机方式获取客户的信息：

```
$result = findCustomerById(rand(1,79), $conn);
```

5. 下面的查看逻辑遍历了二次查询操作返回的结果。在下面的代码中，用于调用嵌入的匿名函数的语句被加粗了：

```
<table>
  <tr>
<th>Transaction</th><th>Date</th><th>Qty</th>
<th>Price</th><th>Product</th>
  </tr>
<?php
foreach ($result['purchases']($result['id'], $conn) as $purchase)
: ?>
  <tr>
    <td><?= $purchase['transaction'] ?></td>
    <td><?= $purchase['date'] ?></td>
    <td><?= $purchase['quantity'] ?></td>
    <td><?= $purchase['sale_price'] ?></td>
    <td><?= $purchase['title'] ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

具体运行情况

创建调用程序 chap_05_secondary_lookups.php，并在其中添加代码，以创建 Application\Database\Connection 类的实例：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
include __DIR__ . '/../Application/Database/Connection.php';
```

```
use Application\Database\Connection;
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
```

向该调用程序中添加前面步骤 3 介绍的 `findCustomerById()` 函数。这样就可以通过随机方式获取某位客户的信息，并结束该调用程序中 PHP 部分的编写工作：

```
function findCustomerById($id, Connection $conn)
{
  // 具体代码请参阅前面步骤 3
}
$result = findCustomerById(rand(1,79), $conn);
?>
```

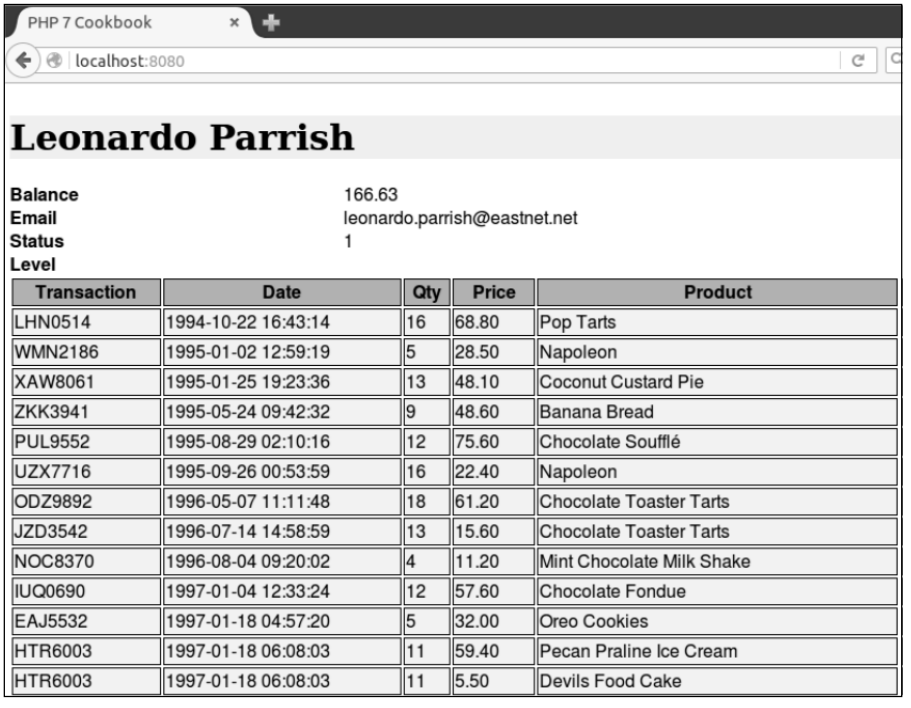
为了了解查看逻辑，你可以像观察前面的几个示例那样观察核心的客户信息：

```
<h1><?= $result['name'] ?></h1>
<div class="row">
<div class="left">Balance</div>
<div class="right"><?= $result['balance']; ?></div>
</div>
<!-- etc.l -->
```

还可以观察客户购买商品的信息：

```
<table>
<tr><th>Transaction</th><th>Date</th><th>Qty</th>
<th>Price</th><th>Product</th></tr>
  <?php
  foreach ($result['purchases']($result['id'], $conn)
           as $purchase) : ?>
  <tr>
    <td><?= $purchase['transaction'] ?></td>
    <td><?= $purchase['date'] ?></td>
    <td><?= $purchase['quantity'] ?></td>
    <td><?= $purchase['sale_price'] ?></td>
    <td><?= $purchase['title'] ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

这段程序的关键之处在于，二次查询操作是通过调用嵌入的匿名函数（使用 `$result['purchases']($result['id'], $conn)` 语句），作为查看逻辑的组成部分被执行的。下面是输出结果：



Transaction	Date	Qty	Price	Product
LHN0514	1994-10-22 16:43:14	16	68.80	Pop Tarts
WMN2186	1995-01-02 12:59:19	5	28.50	Napoleon
XAW8061	1995-01-25 19:23:36	13	48.10	Coconut Custard Pie
ZKK3941	1995-05-24 09:42:32	9	48.60	Banana Bread
PUL9552	1995-08-29 02:10:16	12	75.60	Chocolate Soufflé
UZX7716	1995-09-26 00:53:59	16	22.40	Napoleon
ODZ9892	1996-05-07 11:11:48	18	61.20	Chocolate Toaster Tarts
JZD3542	1996-07-14 14:58:59	13	15.60	Chocolate Toaster Tarts
NOC8370	1996-08-04 09:20:02	4	11.20	Mint Chocolate Milk Shake
IUQ0690	1997-01-04 12:33:24	12	57.60	Chocolate Fondue
EAJ5532	1997-01-18 04:57:20	5	32.00	Oreo Cookies
HTR6003	1997-01-18 06:08:03	11	59.40	Pecan Praline Ice Cream
HTR6003	1997-01-18 06:08:03	11	5.50	Devils Food Cake

实现 jQuery DataTables 插件的 PHP 查询

实现二次查询的另一种方式是使客户端生成二次查询请求。本节会对上一节介绍的实现二次查询的代码稍做修改，将二次查询操作嵌入 QueryResults 中。在上一节介绍的示例中，尽管查询操作是由查看逻辑执行的，但所有的处理工作都是在服务器上完成的。然而，当使用 jQuery DataTables 插件时，实际上二次查询操作是通过浏览器发出的异步 JavaScript 及 XML（AJAX）请求直接在客户端执行的。

具体处理过程

1. 先将实现二次查询操作的逻辑（请参阅上一节）存储到一个独立的 PHP 文件中。这个新建脚本的作用是执行二次查询操作并返回 JSON 数组。

2. 将这个新建的脚本命名为 `chap_05_jquery_datatables_php_lookups_ajax.php`。该脚本会用`$_GET` 参数代表客户的 `id`。注意，其中的 `SELECT` 语句非常

特殊，因为它用于规定获取哪些字段中的数据。获取数据的模式已经切换到 PDO::FETCH_NUM。这个脚本的最后一行代码获取了查询结果，并将它们赋予了 JSON 编码数组中的 data 键。

在使用未进行配置的 jQuery DataTables 插件获取记录时，只能获得完整的（即包含了表中所有字段的）记录信息，牢记这一点极为重要。

```
$id = $_GET['id'] ?? 0;
sql = 'SELECT u.transaction,u.date,
  u.quantity,u.sale_price,r.title '
   . 'FROM purchases AS u '
   . 'JOIN products AS r '
   . 'ON u.product_id = r.id '
   . 'WHERE u.customer_id = :id';
$stmt = $conn->pdo->prepare($sql);
$stmt->execute(['id' => (int) $id]);
$results = array();
while ($row = $stmt->fetch(PDO::FETCH_NUM)) {
  $results[] = $row;
}
echo json_encode(['data' => $results]);
```

3. 使用上一示例介绍的函数，该函数会通过客户的 ID 获取客户的信息，但会从该函数中去除执行二次查询操作的代码：

```
function findCustomerById($id, Connection $conn)
{
  $stmt = $conn->pdo->query(
    'SELECT * FROM customer WHERE id = ' . (int) $id);
  $results = $stmt->fetch(PDO::FETCH_ASSOC);
  return $results;
}
```

4. 在查看逻辑中导入最小化的 jQuery 库、DataTables 插件和样式表，以通过零配置方式实现 jQuery DataTables 插件。在最简实现方式中，我们只需添加 jQuery 库本身（本例使用 jquery-1.12.0.min.js）和 DataTables 插件（jquery.dataTables.js）。还可以添加与 DataTables 关联的便捷样式表 jquery.dataTables.css：

```
<!DOCTYPE html>
<head>
```

```
  <script src="https://code.jquery.com/jquery-1.12.0.min.js">
  </script>
    <script type="text/javascript"
      charset="utf8"
      src="//cdn.datatables.net/1.10.11/js/jquery.dataTables.js">
    </script>
  <link rel="stylesheet"
    type="text/css"
    href="//cdn.datatables.net/1.10.11/css/jquery.dataTables.css">
</head>
```

5. 定义一个 jQuery 库函数：`$(document).ready()`，该函数通过 DataTables 插件与表关联。本例为页面元素 `customerTable` 的 `id` 标签属性赋予将会被分配给 DataTables 插件的表元素。还应注意本例将 AJAX 数据源设置为前面步骤 2 介绍的脚本 `chap_05_jquery_datatables_php_lookups_ajax.php`。一旦使`$id` 变量可用了，就可以将其附到数据源 URL 中：

```
<script>
$(document).ready(function() {
  $('#customerTable').DataTable(
    { "ajax": '/chap_05_jquery_datatables_php_lookups_ajax.
      php?id=<?= $id ?>'
  });
} );
</script>
```

6. 我们在查看逻辑的主体部分中定义了这个表，以确保 `id` 属性与上面代码设置的 `$id` 变量相对应。还需要定义这个表的表头，以便使该表的数据结构与 AJAX 请求的数据结构相匹配：

```
<table id="customerTable" class="display" cellspacing="0"
width="100%">
  <thead>
    <tr>
      <th>Transaction</th>
      <th>Date</th>
      <th>Qty</th>
      <th>Price</th>
      <th>Product</th>
    </tr>
  </thead>
</table>
```

7. 加载上述页面，选择客户的 ID（本例通过随机方式进行选择），并使 jQuery 库生成执行二次查询操作的请求。

具体运行情况

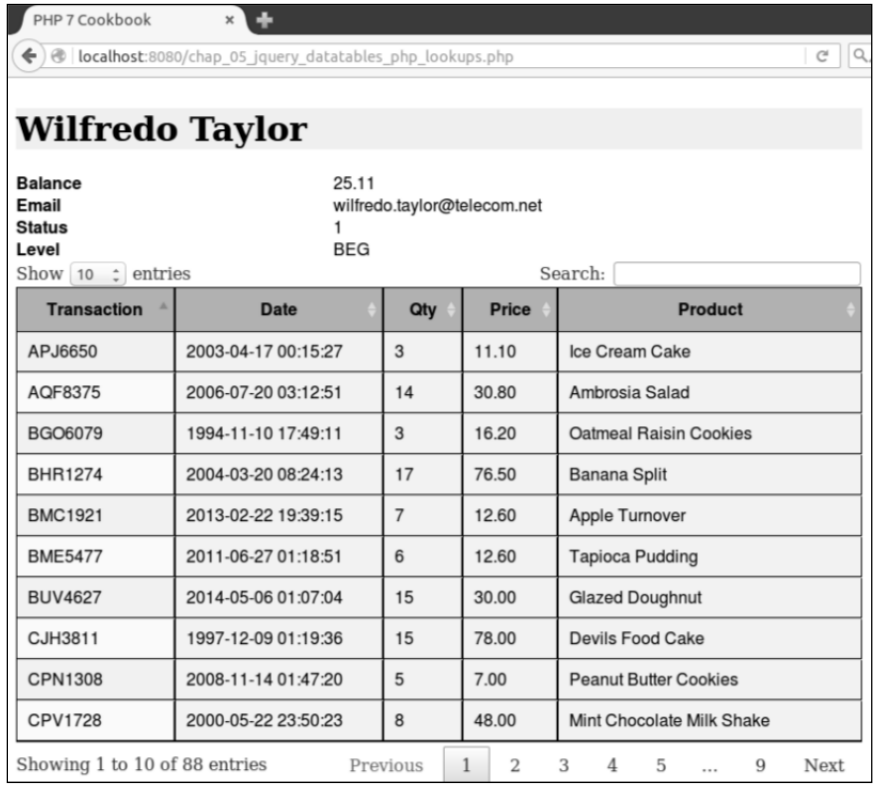
创建 chap_05_jquery_datatables_php_lookups_ajax.php 脚本，该脚本应能够对 AJAX 请求做出回应。在该脚本中添加初始化类自动加载功能的代码，并创建一个 Connection 实例。然后，将前面步骤 2 介绍的代码添加到该脚本中：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
include __DIR__ . '/../Application/Database/Connection.php';
use Application\Database\Connection;
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
```

创建调用程序 chap_05_jquery_datatables_php_lookups.php，该程序应能够通过随机方式获取客户的信息。将前面步骤 3 介绍的代码添加到该程序中：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
include __DIR__ . '/../Application/Database/Connection.php';
use Application\Database\Connection;
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
// 在此处添加函数 findCustomerById()
$id = random_int(1,79);
$result = findCustomerById($id, $conn);
?>
```

这个调用程序还应该含有查看逻辑，该查看逻辑用于导入实现最简 jQuery DataTables 插件的 JavaScript 脚本。先将前面步骤 3 介绍的代码添加到该程序中，然后将前面步骤 5 和步骤 6 介绍的$(document).ready()函数和显示逻辑也添加到该程序中。下图是输出结果：



扩展

要详细了解 jQuery 库，请浏览：`https://jquery.com/`。要详细了解 jQuery 库中的 DataTables 插件，请浏览：`https:// www.datatables.net/`。要详细了解零配置数据表，请浏览：`https:// datatables.net/examples/basic_init/ zero_configuration.html`。要详细了解 AJAX 源数据，请浏览：`https://datatables.net/examples/data_sources/ajax.html`。

第 6 章　创建可伸缩的网站

本章包括以下要点：

- 创建通用表单元素生成器
- 创建 HTML radio 元素生成器
- 创建 HTML select 元素生成器
- 实现表单工厂
- 关联$_POST 过滤器
- 关联$_POST 验证器
- 将验证操作与表单关联起来

本章主要内容简介

本章介绍用于创建 HTML 表单元素的类。这些通用的元素生成器可用于创建文本、文本区域、密码和类似的 HTML 输入元素类型。本章还会介绍通过一系列值预先配置页面元素的多种方式。表单工厂解决方案会将所有这些生成器整合到一起，使你能够通过单个配置数组显示整个表单。最后，本章会介绍过滤和验证收到的$_POST 数据的方式。

创建通用表单元素生成器

创建用于生成简单的表单输入标签（如<input type="text" name="whatever">）的函数非常容易。然而，为了使一个表单生成器能够通用，我们需要做更大的规划。下面是使用输入标签的注意事项：

- 表单输入标签 input 和与之相关的 HTML 标签属性
- 将用户输入的信息反馈给用户的标记
- 在执行了验证操作（稍后详细介绍）后显示输入错误的能力

- 一些封装标签，如 和 HTML 表标签<td>

具体处理过程

1. 先定义 Application\Form\Generic 类。该类稍后会被用作创建专用表单元素的基类：

```
namespace Application\Form;

class Generic
{
  // 在此处添加具体代码
}
```

2. 定义一些类常量，它们会在表单元素生成过程中通用。

3. 前三个常量会变成与表单元素中主要组件关联起来的键。后面的常量用于定义表单支持的输入类型和输入框中的默认值：

```
const ROW = 'row';
const FORM = 'form';
const INPUT = 'input';
const LABEL = 'label';
const ERRORS = 'errors';
const TYPE_FORM = 'form';
const TYPE_TEXT = 'text';
const TYPE_EMAIL = 'email';
const TYPE_RADIO = 'radio';
const TYPE_SUBMIT = 'submit';
const TYPE_SELECT = 'select';
const TYPE_PASSWORD = 'password';
const TYPE_CHECKBOX = 'checkbox';
const DEFAULT_TYPE = self::TYPE_TEXT;
const DEFAULT_WRAPPER = 'div';
```

4. 定义属性和用于设置它们的构造器。

5. 本例需要创建两个属性：$name 和$type，因为没有这两个属性就无法高效地使用表单元素。其他构造器参数都是可选的。此外，为了能够根据一个表单元素创建另一个表单元素，可规定第二个参数（$type）可以是 Application\Form\Generic 类的实例。如果$type 参数传递的是 Generic 类的实例，那么只需运行读取器（稍后详细介绍），就可以为多个属性赋值：

```
protected $name;
protected $type     = self::DEFAULT_TYPE;
protected $label    = '';
protected $errors   = array();
protected $wrappers;
protected $attributes; // HTML 表单属性
protected $pattern = '<input type="%s" name="%s" %s>';

public function __construct($name,
           $type,
           $label = '',
           array $wrappers = array(),
           array $attributes = array(),
           array $errors = array())
{
  $this->name = $name;
  if ($type instanceof Generic) {
      $this->type = $type->getType();
      $this->label = $type->getLabelValue();
      $this->errors = $type->getErrorsArray();
      $this->wrappers = $type->getWrappers();
      $this->attributes = $type->getAttributes();
   } else {
      $this->type = $type ?? self::DEFAULT_TYPE;
      $this->label = $label;
      $this->errors = $errors;
      $this->attributes = $attributes;
      if ($wrappers) {
          $this->wrappers = $wrappers;
      } else {
         $this->wrappers[self::INPUT]['type'] =
           self::DEFAULT_WRAPPER;
         $this->wrappers[self::LABEL]['type'] =
           self::DEFAULT_WRAPPER;
         $this->wrappers[self::ERRORS]['type'] =
           self::DEFAULT_WRAPPER;
    }
  }
  $this->attributes['id'] = $name;
}
```

变量$wrappers 拥有三个主要的子选项：INPUT、LABEL 和 ERRORS，这样我们就能够为 label、input 和 errors 标签定义独立的封装器。

6. 在定义用于生成 HTML 中的 label、input 和 errors 标签的核心方法前，应该先定义 getWrapperPattern()方法，使该方法为 label、input 和 errors 标签生成适当的封装标签。

7. 例如，如果将封装标签定义为 ，而且该封装器的标签属性中包含['class'=> 'label']，那么 getWrapperPattern()方法就会返回 sprintf()模式的封装器，如 %s 。在这个例子中，为 label 标签生成的 HTML 代码最终会替换%s。

8. 下面是 getWrapperPattern()方法的代码：

```
public function getWrapperPattern($type)
{
  $pattern = '<' . $this->wrappers[$type]['type'];
  foreach ($this->wrappers[$type] as $key => $value) {
    if ($key != 'type') {
      $pattern .= ' ' . $key . '="' . $value . '"';
    }
  }
  $pattern .= '>%s</' . $this->wrappers[$type]['type'] . '>';
  return $pattern;
}
```

9. 现在可以定义 getLabel()方法了。该方法的全部作用在于使用 sprintf()函数将 label 标签插入到封装器中：

```
public function getLabel()
{
  return sprintf($this->getWrapperPattern(self::LABEL),
                 $this->label);
}
```

10. 为了生成核心的 input 标签，我们需要能够将标签属性组装起来的手段。幸运的是，只要通过关联数组的形式将这些标签属性提供给构造器，就可以轻松地将它们组装起来。在本例中，我们需要做的只是定义 getAttribs()方法，使该方法生成以空格分隔的键/值对字符串。然后，在返回最终值时使用 trim()函数将键/值对中的空格去掉。

11. 如果页面元素中包含了 value 或 href 标签属性，为安全起见，应避免使用这

些值，以防这些值是由用户提供的（即这些值有可能具有攻击性）。因此，我们需要使用 if 语句进行检查，然后调用 htmlspecialchars() 或 urlencode() 函数：

```
public function getAttribs()
{
  foreach ($this->attributes as $key => $value) {
    $key = strtolower($key);
    if ($value) {
      if ($key == 'value') {
        if (is_array($value)) {
            foreach ($value as $k => $i)
            $value[$k] = htmlspecialchars($i);
        } else {
            $value = htmlspecialchars($value);
        }
      } elseif ($key == 'href') {
          $value = urlencode($value);
      }
      $attribs .= $key . '="' . $value . '" ';
    } else {
      $attribs .= $key . ' ';
    }
  }
  return trim($attribs);
}
```

12. 对于核心的 input 标签，可将其中的内容拆分为两个独立的方法。其中，主要方法 getInputOnly() 应仅生成 HTML 中的 input 标签，而次要方法 getInputWithWrapper() 应生成嵌入到封装器中的 input 标签。进行这种拆分后，在创建衍生型的类（如生成单选按钮的类）时，就不需要为这些标签添加封装器了：

```
public function getInputOnly()
{
  return sprintf($this->pattern, $this->type, $this->name,
            $this->getAttribs());
}

public function getInputWithWrapper()
{
  return sprintf($this->getWrapperPattern(self::INPUT),
            $this->getInputOnly());
}
```

13. 定义一个方法，使该方法能够显示页面元素验证的错误提示信息。可假定这些错误提示是通过数组的形式提供的。如果没有出现错误，就使该方法返回一个空字符串。否则就通过
error 1、
error 2
子类的形式显示这些错误提示：

```
public function getErrors()
{
  if (!$this->errors || count($this->errors == 0)) return '';
  $html = '';
  $pattern = '<li>%s</li>';
  $html .= '<ul>';
  foreach ($this->errors as $error)
  $html .= sprintf($pattern, $error);
  $html .= '</ul>';
  return sprintf($this->getWrapperPattern(self::ERRORS), $html);
}
```

14. 对于某些特殊的标签属性，还应通过对象属性的各个方面对它们增加控制。例如，我们可能需要在已经存在的存储错误提示信息的数组中添加一条错误提示。而且，设置单个标签属性的手段也很有用：

```
public function setSingleAttribute($key, $value)
{
  $this->attributes[$key] = $value;
}
public function addSingleError($error)
{
  $this->errors[] = $error;
}
```

15. 定义读取器和设置器，以便读取和设置属性的值。例如，你可能已经注意到属性$pattern 的默认值是<input type="%s" name="%s" %s>。对于某些标签（如 select 和 form）来说，我们需要将$pattern 属性设置为其他值：

```
public function setPattern($pattern)
{
  $this->pattern = $pattern;
}
public function setType($type)
{
  $this->type = $type;
```

```
}
public function getType()
{
  return $this->type;
}
public function addSingleError($error)
{
  $this->errors[] = $error;
}
// 为名称、标记、封装器、错误提示信息和标签属性定义类似的读取和设置方法
```

16. 还应定义生成表头（不是 HTML 代码）和存储错误提示信息的数组：

```
public function getLabelValue()
{
  return $this->label;
}
public function getErrorsArray()
{
  return $this->errors;
}
```

具体运行情况

将前面介绍的代码都存储到 `Application\Form\Generic` 类中。然后定义调用脚本 `chap_06_form_element_generator.php`，为该脚本设置类自动加载功能并引用 `Application\Form\Generic` 类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Form\Generic;
```

为了了解封装器，我们使用 HTML 表格数据和 header 标签定义封装器。注意，表头使用 `TH` 标签设置，而输入内容和错误提示信息使用 `TD` 标签设置：

```
$wrappers = [
  Generic::INPUT => ['type' => 'td', 'class' => 'content'],
  Generic::LABEL => ['type' => 'th', 'class' => 'label'],
  Generic::ERRORS => ['type' => 'td', 'class' => 'error']
];
```

现在可以通过向这个 HTML 生成器（即 `Application\Form\Generic` 对象）传

输参数来定义电子邮件页面：

```
$email = new Generic('email', Generic::TYPE_EMAIL, 'Email', $wrappers,
                     ['id' => 'email',
                     'maxLength' => 128,
                     'title' => 'Enter address',
                     'required' => '']);
```

也可以使用设置器定义登录页面：

```
$password = new Generic('password', $email);
$password->setType(Generic::TYPE_PASSWORD);
$password->setLabel('Password');
$password->setAttributes(['id' => 'password',
                          'title' => 'Enter your password',
                          'required' => '']);
```

最后不要忘记定义提交按钮：

```
$submit = new Generic('submit',
  Generic::TYPE_SUBMIT,
  'Login',
  $wrappers,
  ['id' => 'submit','title' => 'Click to login','value' =>
   'Click Here']);
```

下面是通过运行上述 PHP 代码生成的 HTML 代码：

```
<div class="container">
  <!-- 登录表单 -->
  <h1>Login</h1>
  <form name="login" method="post">
  <table id="login" class="display"
    cellspacing="0" width="100%">
    <tr><?= $email->render(); ?></tr>
    <tr><?= $password->render(); ?></tr>
    <tr><?= $submit->render(); ?></tr>
    <tr>
      <td colspan=2>
        <br>
        <?php var_dump($_POST); ?>
      </td>
    </tr>
  </table>
  </form>
</div>
```

下面是这些 HTML 代码在浏览器中生成的效果：

创建 HTML radio 元素生成器

单选按钮（radio 元素）生成器与通用型 HTML 表单元素生成器有一些相似之处。像处理其他通用型元素一样，一组单选按钮也需要能够显示选项标题和错误提示信息。这两种生成器的差异主要有两方面：

- 通常需要创建两个或多个单选按钮
- 每个单选按钮本身需要有选项标题

具体处理过程

1. 先通过扩展 `Application\Form\Generic` 类创建新的 `Application\Form\Element\Radio` 类：

```
namespace Application\Form\Element;
use Application\Form\Generic;
class Radio extends Generic
{
  // 在此处添加具体代码
}
```

2. 然后根据创建一组单选按钮的具体需要定义类常量和属性。

3. 本例需要创建隔离物，将之放在单选按钮和它的选项标题之间。我们还需要决定将单选按钮的选项标题放在单选按钮的前面还是后面。因此，需要使用`$after` 标记。

在需要设置默认值和重新显示已经存在表单数据的情况中，还需要有分辨出已选中选项的手段。最后，我们需要设置一系列选项，以便将这些按钮与它们组装到一起：

```
const DEFAULT_AFTER = TRUE;
const DEFAULT_SPACER = '&nbps;';
const DEFAULT_OPTION_KEY = 0;
const DEFAULT_OPTION_VALUE = 'Choose';

protected $after = self::DEFAULT_AFTER;
protected $spacer = self::DEFAULT_SPACER;
protected $options = array();
protected $selectedKey = DEFAULT_OPTION_KEY;
```

4. 如果通过扩展 Application\Form\Generic 类创建生成单选按钮的类就可以扩展__construct()方法，也可以仅定义用于设置特定选项的方法。为了了解具体操作，我们选择后者。

5. 为了确保$this->options 属性就位，下面方法的第一个参数（$options）被定义为必选项（没有默认值）。该方法的其他参数都是可选的。

```
public function setOptions(array $options,
  $selectedKey = self::DEFAULT_OPTION_KEY,
  $spacer = self::DEFAULT_SPACER,
  $after = TRUE)
{
  $this->after = $after;
  $this->spacer = $spacer;
  $this->options = $options;
  $this->selectedKey = $selectedKey;
}
```

6. 这样我们就可以重写核心的 getInputOnly()方法。

7. 将标签属性 id 保存到独立变量$baseId 中，然后将其与$count 变量组合起来，这样每个 id 标签属性就拥有了唯一性。如果与已被选中的键关联的选项被定义了，那么该选项的值就会被赋予相应变量；否则，该选项的默认值就会被赋予相应变量：

```
public function getInputOnly()
{
  $count = 1;
  $baseId = $this->attributes['id'];
```

8. 可在 foreach()循环中检查哪个键被选中了。找到该键后，将 checked 标签属

性添加到与该键对应的单选按钮中。然后调用父类方法 getInputOnly()，为每个按钮生成 HTML 代码。注意，input 标签中的 value 标签属性就是与选项按钮对应的键（保存在关联数组中）。按钮的标题也保存在关联数组中，但它们是键/值对中的值。[1]

具体运行情况

将前面介绍的代码添加到 Application/Form/Element 文件夹中的 Radio.php 文件中。然后定义调用脚本 chap_06_form_element_radio.php，为该脚本设置类自动加载功能并引用新建的 Application\Form\Element\Radio 类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Form\Generic;
use Application\Form\Element\Radio;
```

使用上一示例介绍的$wrappers 数组定义封装器。

定义$statusList 数组，并通过将下面介绍的参数传递给 Radio 类创建用于生成 HTML 代码的实例：

1 在 PHP 中，数组的键就是指数组的下标。

```
foreach ($this->options as $key => $value) {
  $this->attributes['id'] = $baseId . $count++;
  $this->attributes['value'] = $key;
  if ($key == $this->selectedKey) {
      $this->attributes['checked'] = '';
  } elseif (isset($this->attributes['checked'])) {
        unset($this->attributes['checked']);
  }
  if ($this->after) {
      $html = parent::getInputOnly() . $value;
  } else {
     $html = $value . parent::getInputOnly();
  }
  $output .= $this->spacer . $html;
  }
  return $output;
}
```

——译者注

```
$statusList = [
  'U' => 'Unconfirmed',
  'P' => 'Pending',
  'T' => 'Temporary Approval',
  'A' => 'Approved'
];

$status = new Radio('status',
          Generic::TYPE_RADIO,
          'Status',
          $wrappers,
          ['id' => 'status']);
```

现在可以查看通过$_GET 变量是否获取了代表状态的输入数据，并设置这些选项。所有输入数据都会变为选中的数组键。否则，选中的数组键就会被设置为默认值：

```
$checked = $_GET['status'] ?? 'U';
$status->setOptions($statusList, $checked, '<br>', TRUE);
```

最后，不要忘记定义提交按钮：

```
$submit = new Generic('submit',
          Generic::TYPE_SUBMIT,
          'Process',
          $wrappers,
          ['id' => 'submit','title' =>

          'Click to process','value' => 'Click Here']);
```

下面是这段 PHP 程序生成的 HTML 代码：

```
<form name="status" method="get">
<table id="status" class="display" cellspacing="0" width="100%">
  <tr><?= $status->render(); ?></tr>
  <tr><?= $submit->render(); ?></tr>
  <tr>
    <td colspan=2>
      <br>
      <pre><?php var_dump($_GET); ?></pre>
    </td>
  </tr>
</table>
</form>
```

下面是这些 HTML 代码在浏览器中生成的效果：

补充说明

复选框元素生成器与 HTML 单选按钮生成器几乎完全相同。它们之间的主要差异在于，一组复选按钮中可能会有多个按钮同时被选中。因此，应使用 PHP 数组处理复选按钮的标题。复选按钮元素的类型应设置为 `Generic::TYPE_CHECKBOX`。

创建 HTML select 元素生成器

创建单个的 HTML select 元素（下拉菜单中的一个选项）的方式与创建单选按钮的方式类似。然而，两者的创建标签不相同，要创建下拉菜单，既需要使用 `SELECT` 标签，还需要使用一组 `OPTION` 标签。

具体处理过程

1. 通过扩展 `Application\Form\Generic` 类创建新的 `Application\Form\Element\Select` 类。

2. 通过扩展 `Generic` 类（而不是 `Radio` 类）创建 `Select` 类，是因为这两种页面元素的构建方式完全不同：

```
namespace Application\Form\Element;

use Application\Form\Generic;

class Select extends Generic
{
  // 在此添加具体代码
}
```

3. 要创建 Select 类，只需在 Application\Form\Generic 类的基础上少量增加常量和属性。与单选按钮和复选按钮不同，这里无须设置按钮与其标题之间的空白区域的尺寸，以及按钮标题中可选区域的位置：

```
const DEFAULT_OPTION_KEY = 0;
const DEFAULT_OPTION_VALUE = 'Choose';

protected $options;
protected $selectedKey = DEFAULT_OPTION_KEY;
```

4. 下面让我们将注意力集中到选项设置上。因为可在 HTML select 元素中选中一个或多个选项，所以$selectedKey 属性中可能会保存一个字符串，也可能会保存一个数组。因此，我们没有为该属性设置**类型提示**。然而，非常重要的一点是需要分辨出标签属性 multiple 是否已经被设置。通过继承父类的$this->attributes 属性可以做到这一点。

5. 如果标签属性 multiple 已经被设置，那么将标签属性 name 保存到数组中就非常重要。因此，在这种情况中应在 name 属性后添加[]：

```
public function setOptions(array $options, $selectedKey =
                           self::DEFAULT_OPTION_KEY)
{
  $this->options = $options;
  $this->selectedKey = $selectedKey;
  if (isset($this->attributes['multiple'])) {
    $this->name .= '[]';
  }
}
```

在 PHP 程序中，如果 HTML select 页面元素的标签属性 multiple 已经被设置，而且没有使用数组存储 name 标签属性，那么 PHP 程序仅会返回一个值（即只能选中下拉菜单中的一个选项）!

6. 在定义核心方法 getInputOnly()前，需要先定义用于生成 select 标签的方法。这样我们就能够使用 sprintf()方法、$pattern 变量、$name 变量和 getAttribs()方法的返回值（用作参数）生成最终的 HTML 代码。

7. 使用<select name="%s" %s>替换$pattern 变量的默认值。然后遍历这些标签属性，以通过空格分隔的键/值对方式添加它们：

```
protected function getSelect()
{
  $this->pattern = '<select name="%s" %s> ' . PHP_EOL;
  return sprintf($this->pattern, $this->name,
  $this->getAttribs());
}
```

8. 定义生成option标签的方法，这些option标签将会与select标签组合使用。

9. 前面介绍过，$this->options 数组的键代表标签的 value 属性，而该数组的值代表在屏幕上显示的菜单选项的标题。如果$this->selectedKey 属性是一个数组，就需要通过执行检查操作来查明哪些 value 标签属性被保存到了$this->selectedKey 数组中。否则我们就将$this->selectedKey 属性视为一个字符串，这时只需确定该字符串是否与$this->options 数组的键相等。如果选中的键匹配成功（value 标签属性被保存到了$this->selectedKey 数组中，或者$this->selectedKey 字符串与$this->options 数组的键相等），就在相应的 OPTION 标签中添加 selected 标签属性：

```
protected function getOptions()
{
  $output = ' ';
  foreach ($this->options as $key => $value) {
    if (is_array($this->selectedKey)) {
        $selected = (in_array($key, $this->selectedKey))
        ? ' selected' : ' ';
    } else {
        $selected = ($key == $this->selectedKey)
        ? ' selected' : ' ';
    }
      $output .= '<option value=" ' . $key . ' " '
      . $selected . '>'
      . $value
      . '</option>';
  }
  return $output;
}
```

10. 现在可以重写核心的 getInputOnly()方法了。

11. 你可能已经注意到，该方法仅需要捕捉前面介绍的 getSelect()和 getOptions()方法的返回值。当然，我们还需要添加</select>结束标签：

```
public function getInputOnly()
{
  $output = $this->getSelect();
  $output .= $this->getOptions();
  $output .= '</' . $this->getType() . '>';
  return $output;
}
```

具体运行情况

将前面介绍的代码添加到 Application/Form/Element 文件夹中的 Select.php 文件中。定义调用脚本 chap_06_form_element_select.php，为该脚本设置类自动加载功能并引用新建的 Application\Form\Element\Select 类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Form\Generic;
use Application\Form\Element\Select;
```

可以使用本章第一个示例介绍的$wrappers 数组定义封装器，也可以使用“创建 HTML radio 元素生成器”一节介绍过的$statusList 数组来定义。然后就可以创建用于生成 SELECT 页面元素的实例了。第一个实例用于创建单选下拉菜单，第二个实例用于创建多选下拉菜单：

```
$status1 = new Select('status1',
           Generic::TYPE_SELECT,
           'Status 1',
           $wrappers,
           ['id' => 'status1']);
$status2 = new Select('status2',
           Generic::TYPE_SELECT,
           'Status 2',
           $wrappers,
           ['id' => 'status2',
           'multiple' => ' ',
           'size' => '4']);
```

如果用户通过单击下拉菜单输入了代表状态的数据，并通过$_GET 变量设置了下拉菜单选项，那么所有输入的数据都会变成选中的键的值。否则，选中的键就会是默认值。

如前所述，第二个实例用于生成多选下拉菜单，因此通过$_GET 变量获取的 value 标签属性和默认值设置都应该保存在数组中：

```
$checked1 = $_GET['status1'] ?? 'U';
$checked2 = $_GET['status2'] ?? ['U'];
$status1->setOptions($statusList, $checked1);
$status2->setOptions($statusList, $checked2);
```

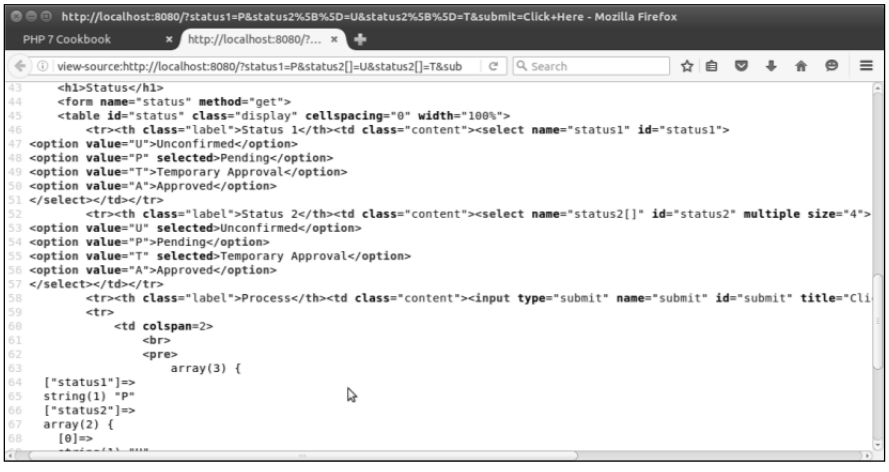
最后不要忘记定义提交按钮（如“创建通用表单元素生成器”一节所示）。

除了需要显示两个独立的 HTML select 页面元素外，本例的显示逻辑与单选按钮示例的显示逻辑相同：

```
<form name="status" method="get">
<table id="status" class="display" cellspacing="0" width="100%">
  <tr><?= $status1->render(); ?></tr>
  <tr><?= $status2->render(); ?></tr>
  <tr><?= $submit->render(); ?></tr>
  <tr>
    <td colspan=2>
      <br>
      <pre>
        <?php var_dump($_GET); ?>
      </pre>
    </td>
  </tr>
</table>
</form>
```

下面是这些 HTML 代码在浏览器中生成的效果：

你还可以在源代码页面查看这些页面元素：



实现表单工厂

使用表单工厂的目的是通过配置数组生成可用的表单对象。这些表单对象应具有获取它本身含有的单个元素的功能，从而能够生成输出结果。

具体处理过程

1. 先创建 `Application\Form\Factory` 类，使用该类存储表单工厂的代码。这个类仅拥有一个属性（`$elements`）和一个读取器：

```
namespace Application\Form;

class Factory
{
  protected $elements;
  public function getElements()
  {
    return $this->elements;
  }
  // 在此处添加其余代码
}
```

2. 在定义生成表单的主要方法前，应先考虑接受配置数据的格式，以及生成表单的确切形式。为了了解具体步骤，我们使用带有`$elements` 属性的 `Factory` 实例生成

表单。该属性会保存一系列 Application\Form\Generic 或 Application\Form\Element 类。

3. 现在可以定义 generate() 方法了。该方法会遍历配置数组，以创建适当的 Application\Form\Generic 或 Application\Form\Element*对象，这些对象会被存储到$elements 数组中。generate() 应将配置数组作为参数接收。将该方法定义为静态方法会更加方便，因为这样我们就可以使用各种配置代码块，根据需要生成任意数量的实例。

4. 为 Application\Form\Factory 类创建一个实例，然后遍历配置数组：

```
public static function generate(array $config)
{
  $form = new self();
  foreach ($config as $key => $p) {
```

5. 检查 Application\Form\Generic 类的构造器中，哪些参数是可选的：

```
$p['errors'] = $p['errors'] ?? array();
$p['wrappers'] = $p['wrappers'] ?? array();
$p['attributes'] = $p['attributes'] ?? array();
```

6. 所有构造器参数都就位后，创建用于生成表单元素的实例，这些实例都存储在 $elements 数组中：

```
$form->elements[$key] = new $p['class']
(
  $key,
  $p['type'],
  $p['label'],
  $p['wrappers'],
  $p['attributes'],
  $p['errors']
);
```

7. 下面让我们来处理下拉菜单中的选项。如果 options 参数被设置了，应使用 list() 方法从$elements 数组提取值，并将这些值赋予下面的变量。然后使用 switch() 方法测试这些页面元素的类型，并使用适当数量的参数调用 setOptions() 方法：

```
if (isset($p['options'])) {
  list($a,$b,$c,$d) = $p['options'];
  switch ($p['type']) {
    case Generic::TYPE_RADIO :
```

```
        case Generic::TYPE_CHECKBOX :
          $form->elements[$key]->setOptions($a,$b,$c,$d);
          break;
        case Generic::TYPE_SELECT :
          $form->elements[$key]->setOptions($a,$b);
          break;
        default :
          $form->elements[$key]->setOptions($a,$b);
          break;
        }
      }
    }
```

8. 返回表单对象并结束该方法：

```
    return $form;
  }
```

9. 从理论上讲，此时我们就可以仅通过遍历存储了页面元素的数组并运行 `render()` 方法，来轻松地在查看逻辑中显示这个表单。下面是查看逻辑的代码：

```
<form name="status" method="get">
  <table id="status" class="display" cellspacing="0" width="100%">
    <?php foreach ($form->getElements() as $element) : ?>
      <?php echo $element->render(); ?>
    <?php endforeach; ?>
  </table>
</form>
```

10. 返回表单对象并结束该方法。

11. 在 `Application\Form\Element` 文件夹中定义独立的 `Form` 类：

```
namespace Application\Form\Element;
class Form extends Generic
{
  public function getInputOnly()
  {
    $this->pattern = '<form name="%s" %s> ' . PHP_EOL;
    return sprintf($this->pattern, $this->name,
              $this->getAttribs());
  }
  public function closeTag()
  {
    return '</' . $this->type . '>';
  }
}
```

12. 再次处理 `Application\Form\Factory` 类，现在需要定义一个简单的方法，

使该方法返回用于封装输入数据的 sprintf()模式的封装器。例如，如果该封装器是带有 class="test"标签属性的 div 标签，那么就应该生成 %s 格式的封装器。这样%s 处的内容就会被 sprintf()函数替换：

```
protected function getWrapperPattern($wrapper)
{
  $type = $wrapper['type'];
  unset($wrapper['type']);
  $pattern = '<' . $type;
  foreach ($wrapper as $key => $value) {
    $pattern .= ' ' . $key . '="' . $value . '"';
  }
  $pattern .= '>%s</' . $type . '>';
  return $pattern;
}
```

13. 现在可以定义用于显示整个表单的方法了。使用 sprintf()模式的封装器显示表单中的每一行内容。然后遍历这些页面元素，显示这些元素，并将输出结果以行为单位封装。接下来，创建 Application\Form\Element\Form 实例。检索 sprintf()模式的表单封装器，检查 form_tag_inside_wrapper 标记，该标记会表明是否应将 form 标签放置在表单封装器的内部：

```
public static function render($form, $formConfig)
{
  $rowPattern = $form->getWrapperPattern(
  $formConfig['row_wrapper']);
  $contents = '';
  foreach ($form->getElements() as $element) {
    $contents .= sprintf($rowPattern, $element->render());
  }
  $formTag = new Form($formConfig['name'],
                  Generic::TYPE_FORM,
                  ' ',
                  array(),
                  $formConfig['attributes']);

  $formPattern = $form->getWrapperPattern(
  $formConfig['form_wrapper']);
  if (isset($formConfig['form_tag_inside_wrapper'])
      && !$formConfig['form_tag_inside_wrapper']) {
      $formPattern = '%s' . $formPattern . '%s';
      return sprintf($formPattern, $formTag->getInputOnly(),
```

```
            $contents, $formTag->closeTag());
        } else {
            return sprintf($formPattern, $formTag->getInputOnly()
            . $contents . $formTag->closeTag());
        }
    }
```

具体运行情况

使用前面介绍的代码创建Application\Form\Factory和Application\Form\Element\Form类。

定义调用脚本 chap_06_form_factor.php，为该脚本设置类自动加载功能，并引用 Application\Form\Factory 和 Application\Form\Element\Form 类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Form\Generic;
use Application\Form\Factory;
```

使用本章第一个示例介绍的$wrappers 数组定义封装器。也可以使用前面介绍的$statusList 数组来定义。

如果代表状态的数据通过$_POST 变量被输入，则所有输入的数据都会变成选中的键的值。否则，选中的键就会是默认值。

```
$email    = $_POST['email'] ?? '';
$checked0 = $_POST['status0'] ?? 'U';
$checked1 = $_POST['status1'] ?? 'U';
$checked2 = $_POST['status2'] ?? ['U'];
$checked3 = $_POST['status3'] ?? ['U'];
```

现在可定义整个表单的配置。参数 name 和 attributes 用于配置 form 标签本身。其他两个参数用于代表表单级和行级的封装器。最后，我们用 form_tag_inside_wrapper 标记指明，form 标签不应出现在封装器（即<table>）的内部。如果该封装器为 ，就可以将表单放置其中，即将 form_tag_inside_wrapper 标记设置为 TRUE：

```
$formConfig = [
  'name'            => 'status_form',
  'attributes'      => ['id'=>'statusForm','method'=>'post',
                        'action'=>'chap_06_form_factory.php'],
```

```
  'row_wrapper'     => ['type' => 'tr', 'class' => 'row'],
  'form_wrapper'    => ['type'=>'table','class'=>'table',
                        'id' =>'statusTable',
                        'class'=>'display','cellspacing'=>'0'],
                        'form_tag_inside_wrapper' => FALSE,
];
```

定义一个数组，使用该数组保存通过 factory 类创建的所有表单元素的参数。该数组的键就是表单元素的名称，而且必须具有唯一性：

```
$config = [
  'email'       => [
    'class'     => 'Application\Form\Generic',
    'type'      => Generic::TYPE_EMAIL,
    'label'     => 'Email',
    'wrappers'  => $wrappers,
    'attributes'=> ['id'=>'email','maxLength'=>128,
                    'title'=>'Enter address',
                    'required'=>'','value'=>strip_tags($email)]
  ],
  'password'    => [
    'class'          => 'Application\Form\Generic',
    'type'           => Generic::TYPE_PASSWORD,
    'label'          => 'Password',
    'wrappers'       => $wrappers,
    'attributes'     => ['id'=>'password',
    'title'          => 'Enter your password',
    'required'       => '']
  ],
  // 其余代码依此类推
];
```

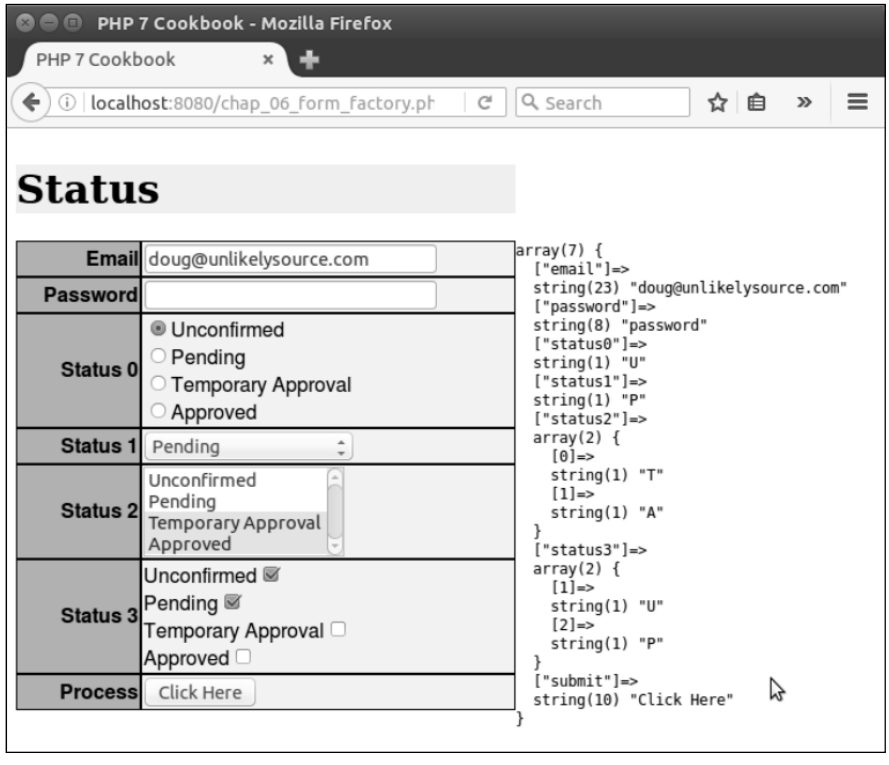
确保使用下面的代码生成表单：

```
$form = Factory::generate($config);
```

实际的显示逻辑非常简单，因为我们仅调用了表单级的 `render()`方法：

```
<?= $form->render($form, $formConfig); ?>
```

下面是这段程序在浏览器中生成的效果：



关联$_POST 过滤器

在处理用户通过网络提交的表单数据时，过滤和验证是需要妥善处理的常见问题。它们很可能是网站最大的安全隐患。此外，将过滤器和验证器覆盖整个应用程序也会使程序变得非常笨拙。关联机制可以灵活地解决这些问题，还能够让我们控制处理过滤器和验证器的次序。

具体处理过程

1. PHP 中的 `filter_input_array()` 函数鲜为人知，乍看之下它很适合完成过滤任务。然而，深入了解该函数的功能后，你会很快明白它仅适用于以前的时代，而没有跟上现代对于攻击防护和灵活性的要求。因此，我们要使用以一系列回调函数为基础的、更灵活的机制来执行过滤和验证操作。

过滤和验证操作之间的差别，是过滤操作可能会除掉或转换某些值。而验证操作会使用与数据特点匹配的标准测试数据，并返回布尔型的结果。

2. 为了增加灵活性，应减轻过滤器和验证器基类的量级。因此，不应在其中定义专用的过滤或验证方法。更确切地说，我们会以回调函数的配置数组为单位执行操作。为了确保过滤和验证结果的兼容性，还应定义专门用于存储结果的对象：`Application\Filter\Result`。

3. `Result` 类的主要函数会含有`$item` 变量，该变量中会保存经过过滤的值或执行验证操作得到的布尔型结果。另一个属性`$messages` 会保存执行过滤和验证操作过程中生成的消息。在这个构造器中，为`$messages` 变量提供的值是以数组的形式组织的。你会发现这两个属性都被定义为了公用的（`public`），这会使访问它们的操作变得更简单：

```
namespace Application\Filter;

class Result
{

  public $item; // （混合型） 经过过滤的数据 | （布尔型） 执行
                   验证操作后得到的结果
  public $messages = array(); // [（字符型） 消息,
                                  （字符型） 消息 ]
  public function __construct($item, $messages)
  {
    $this->item = $item;
    if (is_array($messages)) {
        $this->messages = $messages;
    } else {
        $this->messages = [$messages];
    }
  }
```

4. 还应定义一个方法，通过该方法合并 `Result` 实例。这个方法很重要，因为有时我们需要使用多个过滤器处理同一个值。在这种情况中，需要将新得到的经过过滤的值覆盖已经保存的旧值，同时还需要将新旧消息合并到一起：

```
public function mergeResults(Result $result)
{
  $this->item = $result->item;
```

```
    $this->mergeMessages($result);
  }

  public function mergeMessages(Result $result)
  {
    if (isset($result->messages) && is_array($result->messages)) {
      $this->messages = array_merge($this->messages,
                                    $result->messages);
    }
  }
```

5. 为了使该类中的方法能够构成完整的处理过程，还要向该类中添加一个用于将验证结果合并到一起的方法。此处的要点是只要出现值为 FALSE 的验证结果（不论在验证过程的哪个部分出现），就会导致最终的验证结果为 FALSE：

```
  public function mergeValidationResults(Result $result)
  {
    if ($this->item === TRUE) {
      $this->item = (bool) $result->item;
    }
    $this->mergeMessages($result);
    }
  }
```

6.为了确保这些回调函数生成具有兼容性的结果，应定义一个 Application\Filter\CallbackInterface 接口。注意，我们应利用 PHP 7 的新增功能约束返回值的数据类型，以确保返回值为 Result 实例：

```
  namespace Application\Filter;
  interface CallbackInterface
  {
    public function __invoke ($item, $params) : Result;
  }
```

7. 每个回调函数都应该引用同一组消息。因此，应定义拥有一系列静态属性的 Application\Filter\Messages 类。还应定义用于设置所有消息或仅一条消息的类。应将$messages 属性设置为 public（公用的），以便能够更轻松地访问它：

```
  namespace Application\Filter;
  class Messages
  {
    const MESSAGE_UNKNOWN = 'Unknown';
    public static $messages;
```

```
    public static function setMessages(array $messages)
    {
      self::$messages = $messages;
    }
    public static function setMessage($key, $message)
    {
      self::$messages[$key] = $message;
    }
    public static function getMessage($key)
    {
      return self::$messages[$key] ?? self::MESSAGE_UNKNOWN;
    }
}
```

8. 现在可定义 Application\Web\AbstractFilter 类，使用该类增强这段程序的核心功能。如前所述，应使用较轻的量级定义这个类，无须考虑定义专用的过滤器或验证器，因为它们都是通过配置定义的。可使用 PHP 7 的 PHP 标准库（Standard PHP Library，SPL）中的 UnexpectedValueException 类，在某个回调函数没有实现 CallbackInterface 接口时，抛出描述情况的异常：

```
namespace Application\Filter;
use UnexpectedValueException;
abstract class AbstractFilter
{
  // 稍后介绍此处的具体代码
```

9. 先定义用于保存各种常用值的类常量。下面列出常量中的后 4 个，用于控制显示消息的格式，以及描述缺失数据的方式：

```
const BAD_CALLBACK = 'Must implement CallbackInterface';
const DEFAULT_SEPARATOR = '<br>' . PHP_EOL;
const MISSING_MESSAGE_KEY = 'item.missing';
const DEFAULT_MESSAGE_FORMAT = '%20s : %60s';
const DEFAULT_MISSING_MESSAGE = 'Item Missing';
```

10. 定义核心属性$separator，使用该属性将过滤操作的消息和验证操作的消息结合到一起。$callbacks 变量代表执行过滤和验证操作的一系列回调函数。$assignments 变量用于将需进行过滤和/或验证的数据与过滤器和/或验证器对应起来。$missingMessage 变量代表可被覆盖的属性（用于处理使用多种语言显示页面的网址）。$results 变量用于存储一组 Application\Filter\Result 对象，而且该

变量是通过过滤或验证操作赋值的：

```
protected $separator; // 用于显示消息
protected $callbacks;
protected $assignments;
protected $missingMessage;
protected $results = array();
```

11. 此刻可以定义__construct()方法。该方法的主要功能是设置一系列回调函数和执行赋值操作。它还会设置分隔符（用于控制显示消息的格式）的值，或者接收默认的分隔符，以及接收代表消息缺失情况的值：

```
public function __construct(array $callbacks, array $assignments,
                            $separator = NULL, $message = NULL)
{
  $this->setCallbacks($callbacks);
  $this->setAssignments($assignments);
  $this->setSeparator($separator ?? self::DEFAULT_SEPARATOR);
  $this->setMissingMessage($message
                          ?? self::DEFAULT_MISSING_MESSAGE);
}
```

12. 定义一些方法，使用这些方法设置和去除回调函数。注意，应能够做到读取和设置单个的回调函数。在处理一组通用的回调函数，且仅需要修改其中一个时，这一点非常重要。还应注意，应使用 setOneCall()函数检查回调函数是否实现了CallbackInterface接口。如果回调函数没有实现CallbackInterface接口，那么就应通过setOneCall()函数抛出UnexpectedValueException异常：

```
public function getCallbacks()
{
  return $this->callbacks;
}

public function getOneCallback($key)
{
  return $this->callbacks[$key] ?? NULL;
}

public function setCallbacks(array $callbacks)
{
  foreach ($callbacks as $key => $item) {
    $this->setOneCallback($key, $item);
  }
```

```
public function setOneCallback($key, $item)
{
  if ($item instanceof CallbackInterface) {
     $this->callbacks[$key] = $item;
  } else {
     throw new UnexpectedValueException(self::BAD_CALLBACK);
  }
}

public function removeOneCallback($key)
{
  if (isset($this->callbacks[$key]))
  unset($this->callbacks[$key]);
}
```

13. 用于处理结果的方法非常简单。为简便起见，本例使用了 `getItemsAsArray()` 方法，否则 `getResults()` 方法会返回一组 `Result` 对象：

```
public function getResults()
{
  return $this->results;
}

public function getItemsAsArray()
{
  $return = array();
  if ($this->results) {
    foreach ($this->results as $key => $item)
    $return[$key] = $item->item;
  }
  return $return;
}
```

14. 要检索消息，只需遍历`$this->results` 数组，并提取其中的`$messages` 属性。为简便起见，本例仅使用带有一些格式选项的 `getMessageString()` 方法。为了轻松地生成一组消息，本例使用了 PHP 7 的 `yield from` 语法，这可以将 `getMessages()` 方法变成一种授权型生成器。存储消息的数组则变成了子生成器：

```
public function getMessages()
{
  if ($this->results) {
       foreach ($this->results as $key => $item)
       if ($item->messages) yield from $item->messages;
  } else {
```

```
        return array();
    }
}

public function getMessageString($width = 80, $format = NULL)
{
  if (!$format)
  $format = self::DEFAULT_MESSAGE_FORMAT . $this->separator;
  $output = ' ';
  if ($this->results) {
    foreach ($this->results as $key => $value) {
      if ($value->messages) {
        foreach ($value->messages as $message) {
          $output .= sprintf(
            $format, $key, trim($message));
        }
      }
    }
  }
  return $output;
}
```

15. 定义一些各式各样的读取器和设置器：

```
  public function setMissingMessage($message)
  {
    $this->missingMessage = $message;
  }
  public function setSeparator($separator)
  {
    $this->separator = $separator;
  }
  public function getSeparator()
  {
    return $this->separator;
  }
  public function getAssignments()
  {
    return $this->assignments;
  }
  public function setAssignments(array $assignments)
  {
```

```
    $this->assignments = $assignments;
  }
  // 这是 AbstractFilter 类的结束括号
}
```

16. 尽管过滤和验证操作经常一起执行，但有时也会分开执行。因此，应为每种操作单独定义类。下面先定义 Application\Filter\Filter 类。为了实现前面介绍的核心功能，可通过 AbstractFilter 类扩展 Filter 类：

```
namespace Application\Filter;
class Filter extends AbstractFilter
{
  // 此处添加具体代码
}
```

17. 在这个类中定义核心的 process() 方法，使该方法能够扫描数据，并根据将数据与过滤器关联起来的数组应用过滤器。如果某一组数据没有分配到过滤器，那么只需使 process() 方法返回 NULL：

```
public function process(array $data)
{
  if (!(isset($this->assignments)
     && count($this->assignments))) {
       return NULL;
  }
```

18. 如果这组数据分配到过滤器了，应使 process() 方法初始化$this->results 数组，使用该数组存储一组 Result 对象，使这些 Result 对象中的$item 属性保存通过 $data 参数获得的初始值，而在$messages 属性中存储一个空数组：

```
foreach ($data as $key => $value) {
  $this->results[$key] = new Result($value, array());
}
```

19. 复制$this->assignments 数组中存储的值，检查其中是否含有全局过滤器（用*键标识）。如果该值中确实含有全局过滤器，就运行 processGlobal() 方法，然后复位与*键对应的值：

```
$toDo = $this->assignments;
if (isset($toDo['*'])) {
  $this->processGlobalAssignment($toDo['*'], $data);
  unset($toDo['*']);
}
```

20. 通过调用 processAssignment() 方法遍历经过全局过滤的数据：

```
processAssignment():
foreach ($toDo as $key => $assignment) {
  $this->processAssignment($assignment, $key);
}
```

21. 前面介绍过，每个$assignment 参数都通过键与需执行过滤操作的数据段对应，并代表应用于该数据段的一组回调函数。因此，应在 processGlobalAssignment() 方法中遍历这组回调函数。然而，因为在本例中这些$assignments 变量都是全局的，所以需要遍历整个数据集合，并依次应用每个全局过滤器：

```
protected function processGlobalAssignment($assignment, $data)
{
  foreach ($assignment as $callback) {
    if ($callback === NULL) continue;
    foreach ($data as $k => $value) {
      $result = $this->callbacks[$callback['key']]
                                ($this->results[$k]->item,
      $callback['params']);
      $this->results[$k]->mergeResults($result);
    }
  }
}
```

下面的代码有一点难度：

```
$result = $this->callbacks[$callback['key']]($this
 ->results[$k]->item, $callback['params']);
```

注意，每个回调函数实际上是一个定义了 PHP 中__invoke() 魔术方法的匿名类。它们的参数是需要执行过滤操作的数据和一个参数数组。通过运行$this->callbacks[$callback['key']]()方法，我们实际上巧妙地调用了__invoke()方法。

22. 在定义 processAssignment() 方法时（与定义 processGlobalAssignment() 方法的方式类似），需要执行已经分配给每个数据键的其余回调函数：

```
protected function processAssignment($assignment, $key)
{
  foreach ($assignment as $callback) {
```

```
        if ($callback === NULL) continue;
        $result = $this->callbacks[$callback['key']]
                                  ($this->results[$key]->item,
                                   $callback['params']);
        $this->results[$key]->mergeResults($result);
      }
    }
} // 这是 Application\Filter\Filter 类的结束括号
```

任何更改了用户提供的原始数据的过滤操作，都应通过显示消息的方式指明更改情况，这一点非常重要。在这种更改操作没有得到用户的了解和同意的情况下，这些消息能够变成审计线索，免除你的法律责任。

具体运行情况

创建 Application\Filter 文件夹。在该文件夹中，通过前面介绍的步骤，创建下列类文件：

Application\Filter*类文件	对应的步骤
Result.php	3～5
CallbackInterface.php	6
Messages.php	7
AbstractFilter.php	8～15
Filter.php	16～22

使用前面步骤 5 介绍的代码，在 chap_06_post_data_config_messages.php 文件中配置一组消息。每个回调函数都应引用 Messages::$messages 属性。下面是配置示例：

```
<?php
use Application\Filter\Messages;
Messages::setMessages(
  [
    'length_too_short'  => 'Length must be at least %d',
    'length_too_long'   => 'Length must be no more than %d',
    'required'          => 'Please be sure to enter a value',
    'alnum'             => 'Only letters and numbers allowed',
    'float'             => 'Only numbers or decimal point',
    'email'             => 'Invalid email address',
```

```
    'in_array'          => 'Not found in the list',
    'trim'              => 'Item was trimmed',
    'strip_tags'        => 'Tags were removed from this item',
    'filter_float'      => 'Converted to a decimal number',
    'phone'             => 'Phone number is [+n] nnn-nnn-nnnn',
    'test'              => 'TEST',
    'filter_length'     => 'Reduced to specified length',
  ]
);
```

创建回调函数配置文件 chap_06_post_data_config_callbacks.php，将执行过滤操作的回调函数的配置数据保存在该文件中（参阅前面介绍的步骤 4）。每个回调函数都应遵循下面的通用模板：

```
'callback_key' => new class () implements CallbackInterface
{
  public function __invoke($item, $params) : Result
  {
    $changed = array();
    $filtered = /* perform filtering operation on $item */
    if ($filtered !== $item)
        $changed = Messages::$messages['callback_key'];
    return new Result($filtered, $changed);
  }
}
```

这些回调函数本身必须实现 CallbackInterface 接口并且返回 Result 实例。通过使这些回调函数返回实现 CallbackInterface 接口的匿名类，可以利用 PHP 7 中的匿名类功能。下面是执行过滤操作的回调函数示例：

```
use Application\Filter\ { Result, Messages, CallbackInterface };
$config = [ 'filters' => [
  'trim' => new class () implements CallbackInterface
  {
    public function __invoke($item, $params) : Result
    {
      $changed = array();
      $filtered = trim($item);
      if ($filtered !== $item)
      $changed = Messages::$messages['trim'];
      return new Result($filtered, $changed);
    }
```

```
    },
    'strip_tags' => new class ()
    implements CallbackInterface
    {
      public function __invoke($item, $params) : Result
      {
        $changed = array();
        $filtered = strip_tags($item);
        if ($filtered !== $item)
        $changed = Messages::$messages['strip_tags'];
        return new Result($filtered, $changed);
      }
    },
    // 诸如此类的回调函数
  ]
  ];
```

为了进行测试,可将一个数据结构预览表作为目标。不通过$_POST 变量提供数据,而是构建含有合法和非法数据的数组:

Field name	Type	Allow nulls?
id	int(11)	No
first_name	varchar(128)	No
last_name	varchar(128)	No
address	varchar(256)	Yes
city	varchar(64)	Yes
state_province	varchar(32)	Yes
postal_code	char(16)	No
phone	varchar(16)	No
country	char(2)	No
email	varchar(250)	No
status	char(8)	Yes
budget	decimal(10,2)	Yes
last_updated	datetime	Yes

创建 chap_06_post_data_filtering.php 脚本并为其设置类自动加载功能,并在其中导入消息和回调函数的配置文件:

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
include __DIR__ . '/chap_06_post_data_config_messages.php';
include __DIR__ . '/chap_06_post_data_config_callbacks.php';
```

定义能够将需执行过滤操作的数据段和执行过滤操作的回调函数关联起来的$assignments数组。使用*键定义应用于所有数据的全局数组：

```
$assignments = [
  '*' => [ ['key' => 'trim', 'params' => []],
     ['key' => 'strip_tags', 'params' => []] ],
  'first_name' => [ ['key' => 'length',
   'params' => ['length' => 128]] ],
  'last_name' => [ ['key' => 'length',
   'params' => ['length' => 128]] ],
  'city' => [ ['key' => 'length',
   'params' => ['length' => 64]] ],
  'budget' => [ ['key' => 'filter_float', 'params' => []] ],
];
```

定义合法和非法的测试数据：

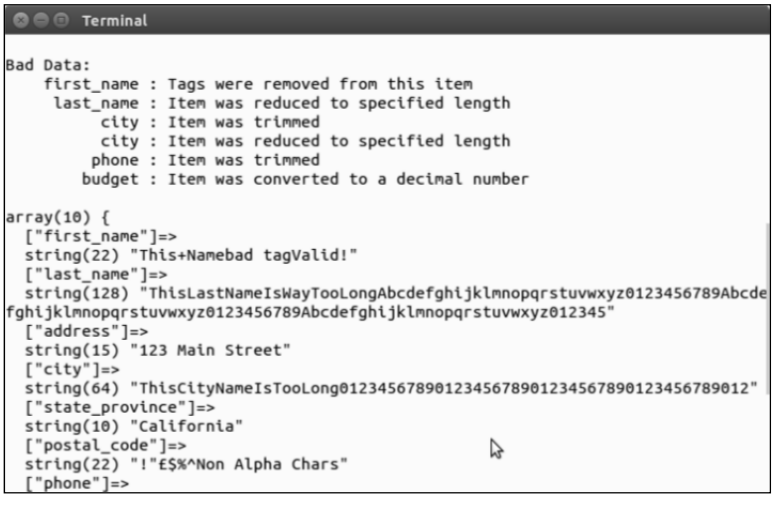
```
$goodData = [
  'first_name'          => 'Your Full',
  'last_name'           => 'Name',
  'address'             => '123 Main Street',
  'city'                => 'San Francisco',
  'state_province'      => 'California',
  'postal_code'         => '94101',
  'phone'               => '+1 415-555-1212',
  'country'             => 'US',
  'email'               => 'your@email.address.com',
  'budget'              => '123.45',
];
$badData = [
  'first_name'          => 'This+Name<script>bad tag</script>Valid!',
  'last_name'           => 'ThisLastNameIsWayTooLong
                        Abcdefghijklmnopqrstuvwxyz0123456789
                        Abcdefghijklmnopqrstuvwxyz0123456789
                        Abcdefghijklmnopqrstuvwxyz0123456789
                        Abcdefghijklmnopqrstuvwxyz0123456789',
  //'address'           => ' ', // 缺失
  'city'                => '
ThisCityNameIsTooLong012345678901234567890123456789012345678901234
5678901234567890123456789 ',
  //'state_province'    => ' ', // 缺失
  'postal_code'         => '!"£$%^Non Alpha Chars',
  'phone'               => ' 12345 ',
  'country'             => 'XX',
```

```
  'email'                => 'this.is@not@an.email',
  'budget'               => 'XXX',
];
```

最后，创建 Application\Filter\Filter 实例并测试上述数据：

```
$filter = new Application\Filter\Filter(
$config['filters'], $assignments);
$filter->setSeparator(PHP_EOL);
  $filter->process($goodData);
echo $filter->getMessageString();
  var_dump($filter->getItemsAsArray());

$filter->process($badData);
echo $filter->getMessageString();
var_dump($filter->getItemsAsArray());
```

在处理合法数据时，会生成一条消息，表明浮点型字段的值是通过字符型数据转换为浮点型数据得到的，除此之外不会生成其他消息。而在处理非法数据时会生成如下输出结果：



上述输出结果中的消息表明，从用户输入的 first_name 字段的值中去除了 HTML 标签，而且为了不超过设定的数据范围，还对用户输入的 last_name 和 city 字段的值执行了截断操作。

补充说明

filter_input_array() 函数接收两个参数：输入的源数据（以预定义常量的形

式，通过$_* PHP 超级全局变量（即$_POST）设置），和一个关联数组（该数组的键为需执行过滤和/或验证操作的数据段，该数组的值为过滤器和/或验证器）。filter_input_array()函数不仅可以执行过滤操作，还可以执行验证操作。其中名称里含有 *sanitize* 的回调函数才是过滤器。

扩展

要查看filter_input_array()函数的文档和示例，可浏览http://php.net/manual/en/function.filter-input-array.php。要了解其他类型的过滤器，请浏览 http://php.net/manual/en/filter.filters.php。

关联$_POST 验证器

上一节已经介绍了这类操作中难度较大的部分。Application\Filter\AbstractFilter 类定义了这类操作中的核心功能。实际的验证操作可通过一组回调函数执行。

具体处理过程

1. 除了特别说明外，本例将使用上一示例中介绍的所有类和配置文件。

2. 先定义用于保存执行验证操作的回调函数的数组。与上一示例一样，所有回调函数都应该实现 Application\Filter\CallbackInterface 接口，而且应该返回 Application\Filter\Result 实例。这些验证器应拥有下面的通用模式：

```
use Application\Filter\ { Result, Messages, CallbackInterface };
$config = [
  // 执行验证操作的回调函数
  'validators' => [
    'key' => new class () implements CallbackInterface
    {
      public function __invoke($item, $params) : Result
      {
        // 在此处添加验证逻辑
        return new Result($valid, $error);
      }
    },
   // 依此类推
```

3. 定义 Application\Filter\Validator 类，使用该类遍历执行赋值操作的数组，使用相应的回调函数验证器检验输入的每一项数据。为了实现前面介绍过的核心功能，可通过扩展 AbstractFilter 类实现 Application\Filter\Validator 类：

```
namespace Application\Filter;
class Validator extends AbstractFilter
{
  // 此处添加具体代码
}
```

4. 在 Validator 类中定义核心的 process() 方法，使该方法能够扫描数据，并根据将数据与验证器关联起来的数组应用验证器。如果某一组数据没有分配到验证器，那么只需返回 $valid 变量的当前值（即 TRUE）：

```
public function process(array $data)
{
  $valid = TRUE;
  if (!(isset($this->assignments)
       && count($this->assignments))) {
         return $valid;
  }
```

5. 如果这组数据分配到验证器了，应使 process() 方法初始化 $this->results 数组，使用该数组存储一组 Result 对象，这些 Result 对象中的 $item 属性保存为 TRUE，而 $messages 属性存储为一个空数组：

```
foreach ($data as $key => $value) {
  $this->results[$key] = new Result(TRUE, array());
}
```

6. 复制 $this->assignments 数组中存储的值，检查其中是否含有全局过滤器（用 * 键标识）。如果该值中确实含有全局过滤器，就运行 processGlobal() 方法，然后复位与 * 键对应的值：

```
$toDo = $this->assignments;
if (isset($toDo['*'])) {
  $this->processGlobalAssignment($toDo['*'], $data);
  unset($toDo['*']);
}
```

7. 通过调用 processAssignment() 方法，遍历经过全局过滤的数据。此时是查看是否有数据被过滤掉的最佳时机。注意，如果某个回调函数验证器返回了 FALSE，就

要将相应对象中的$valid 属性设置为 FALSE：

```
foreach ($toDo as $key => $assignment) {
  if (!isset($data[$key])) {
      $this->results[$key] =
      new Result(FALSE, $this->missingMessage);
  } else {
      $this->processAssignment(
        $assignment, $key, $data[$key]);
  }
  if (!$this->results[$key]->item) $valid = FALSE;
  }
  return $valid;
}
```

8. 如前所述，每个$assignment 参数都通过键与需执行过滤操作的数据段对应，并代表应用于该数据段的一组回调函数。因此，应在 processGlobalAssignment() 方法中遍历这组回调函数。然而，因为在本例中这些$assignments 变量都是全局的，所以需要遍历整个数据集合，并依次应用每个全局过滤器。

9. 与上例介绍的 Application\Filter\Filter::processGlobalAssignment() 方法相比，还需要调用 mergeValidationResults() 方法。这样做的原因是，如果$result->item 属性的值已经被设置为 FALSE，那么需要确保该值不会被后续操作覆盖为 TRUE。如果验证处理过程中的任何一个验证器返回 FALSE，那么其他验证器返回的结果就都必须被覆盖为 FALSE：

```
protected function processGlobalAssignment($assignment, $data)
{
  foreach ($assignment as $callback) {
    if ($callback === NULL) continue;
    foreach ($data as $k => $value) {
      $result = $this->callbacks[$callback['key']]
      ($value, $callback['params']);
      $this->results[$k]->mergeValidationResults($result);
    }
  }
}
```

10. 在定义 processAssignment() 方法时（与定义 processGlobalAssignment() 方法的方式类似），需要执行已经分配给每个数据键的其余回调函数，并再次调用

mergeValidationResults()方法：

```
protected function processAssignment($assignment, $key, $value)
{
  foreach ($assignment as $callback) {
    if ($callback === NULL) continue;
        $result = $this->callbacks[$callback['key']]
      (  $value, $callback['params']);
        $this->results[$key]->mergeValidationResults($result);
  }
}
```

具体处理过程

像上一个示例一样，应确保定义了下面这些类：

- Application\Filter\Result
- Application\Filter\CallbackInterface
- Application\Filter\Messages
- Application\Filter\AbstractFilter

可使用上一示例介绍过的 chap_06_post_data_config_messages.php 文件。

在 Application\Filter 文件夹中创建 Validator.php 文件，在该文件中添加前面步骤 3 至步骤 10 介绍的代码。

创建回调函数配置文件 chap_06_post_data_config_callbacks.php，在该文件中添加前面步骤 2 中介绍的回调函数验证器的代码。所有回调函数都应使用下面的通用模板：

```
'validation_key' => new class () implements CallbackInterface
{
  public function __invoke($item, $params) : Result
  {
    $error = array();
    $valid = /* perform validation operation on $item */
    if (!$valid)
    $error[] = Messages::$messages['validation_key'];
    return new Result($valid, $error);
  }
}
```

创建调用脚本 chap_06_post_data_validation.php，为该脚本初始化类自动加载功能，并导入下面的配置脚本：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
include __DIR__ . '/chap_06_post_data_config_messages.php';
include __DIR__ . '/chap_06_post_data_config_callbacks.php';
```

定义$assignments 数组，将需要进行验证的数据与回调函数验证器关联起来：

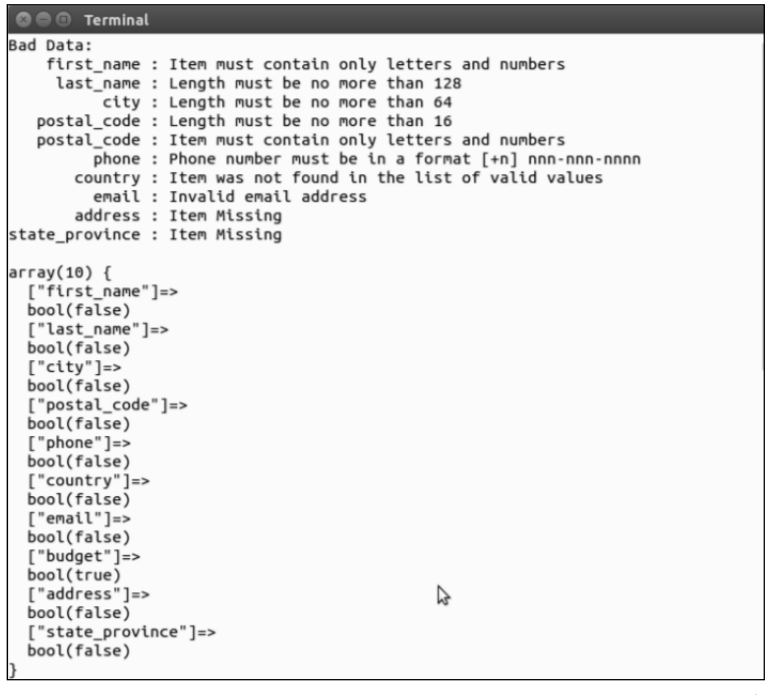
```
$assignments = [
  'first_name'      => [ ['key' => 'length',
  'params'          => ['min'   => 1, 'max' => 128]],
                       ['key'   => 'alnum',
  'params'          => ['allowWhiteSpace'  => TRUE]],
                       ['key'   => 'required','params' => []] ],
  'last_name'       => [ ['key' => 'length',
  'params'          => ['min'   => 1, 'max' => 128]],
                       ['key' => 'alnum',
  'params'          => ['allowWhiteSpace'  => TRUE]],
                       ['key'   => 'required','params' => []] ],
  'address'         => [ ['key' => 'length',
  'params'          => ['max'   => 256]] ],
  'city'            => [ ['key' => 'length',
  'params'          => ['min'   => 1, 'max' => 64]] ],
  'state_province'  => [ ['key' => 'length',
  'params'          => ['min'   => 1, 'max' => 32]] ],
  'postal_code'     => [ ['key' => 'length',
  'params'          => ['min'   => 1, 'max' => 16] ],
                       ['key'   => 'alnum',
  'params'          => ['allowWhiteSpace' => TRUE]],
                       ['key'   => 'required','params' => []] ],
  'phone'           => [ ['key' => 'phone', 'params'   => []] ],
  'country'         => [ ['key' => 'in_array',
  'params'          => $countries ],
                       ['key'   => 'required','params' => []] ],
  'email'           => [ ['key' => 'email', 'params'   => [] ],
                       ['key'   => 'length',
  'params'          => ['max'   => 250] ],
                       ['key'   => 'required','params' => [] ] ],
  'budget'          => [ ['key' => 'float', 'params'   => []] ]
];
```

为了进行测试，还是使用上一示例介绍的 chap_06_post_data_filtering.

php 文件，该文件中含有一组合法数据和一组非法数据。然后创建 Application\Filter\Validator 实例并进行测试：

```
$validator = new Application\Filter\Validator($config['validators'],
$assignments);
$validator->setSeparator(PHP_EOL);
$validator->process($badData);
echo $validator->getMessageString(40, '%14s : %-26s' . PHP_EOL);
var_dump($validator->getItemsAsArray());
$validator->process($goodData);
echo $validator->getMessageString(40, '%14s : %-26s' . PHP_EOL);
var_dump($validator->getItemsAsArray());
```

正如我们所预期的，合法数据不会产生验证错误，而非法数据会生成下面的输出结果：



注意，用户没有填写的 address 和 state_province 字段的验证结果都是 FALSE，而且验证器会返回消息，以表明这两个字段的信息缺失。

将验证操作与表单关联起来

当表单在浏览器中第一次显示时，表单类（如前面介绍过的 Application\Form\Factory）与执行过滤和/或验证操作的类（如前面介绍的 Application\Filter*）几乎没有任何关系。然而，一旦表单数据被提交，这两种类之间的关系就会变得更加紧密。如果表单数据没有通过验证，那么这些数据就会被过滤掉，然后重新显示表单。验证错误消息会与表单元素绑定，并在表单输入框的旁边显示。

具体处理过程

1. 实现表单工厂类，将该类与$_POST 过滤器和$_POST 验证器关联起来。

2. 先创建 Application\Form\Factory 类，在其中添加属性和设置器，使之能够与 Application\Filter\Filter 和 Application\Filter\Validator 类的实例关联起来。还需要定义$data 属性，使用该变量保存经过过滤和/或验证的数据：

```
const DATA_NOT_FOUND = 'Data not found. Run setData()';
const FILTER_NOT_FOUND = 'Filter not found. Run setFilter()';
const VALIDATOR_NOT_FOUND = 'Validator not found.
  Run setValidator()';

protected $filter;
protected $validator;
protected $data;
public function setFilter(Filter $filter)
{
  $this->filter = $filter;
}

public function setValidator(Validator $validator)
{
  $this->validator = $validator;
}

public function setData($data)
{
  $this->data = $data;
}
```

3. 定义 validate()方法,使该方法能够调用 Application\Filter\Validator 实例中的 process()方法。应使该方法检查$data 和$validator 变量中是否存在值。如果这两个属性不存在，应使 validate()方法抛出相应的异常，以指明在 Factory 类中调用方法的次序：

```
public function validate()
{
  if (!$this->data)
  throw new RuntimeException(self::DATA_NOT_FOUND);

  if (!$this->validator)
  throw new RuntimeException(self::VALIDATOR_NOT_FOUND);
```

4. 调用了 process()方法后,应将验证结果消息和表单元素消息关联起来。注意，process()方法会返回布尔型的值，该值代表数据集合的整体验证情况。如果因表单数据没有通过验证而导致表单在浏览器中重新显示，那么错误提示消息会在表单元素的旁边显示。

```
$valid = $this->validator->process($this->data);

foreach ($this->elements as $element) {
  if (isset($this->validator->getResults()
       [$element->getName()])) {
       $element->setErrors($this->validator->getResults()
       [$element->getName()]->messages);
    }
  }
  return $valid;
}
```

5. 可使用类似的方式定义 filter()方法，使该方法调用 Application\ Filter\ Filter 实例中的 process()方法。像处理前面步骤 3 介绍的 validate()方法一样，还需要使 filter()方法检查$data 和$filter 属性中是否存在值。如果这两个属性都没有被赋值，可使 filter()方法抛出带有相应消息的 RuntimeException 异常：

```
public function filter()
{
  if (!$this->data)
  throw new RuntimeException(self::DATA_NOT_FOUND);
```

```
  if (!$this->filter)
  throw new RuntimeException(self::FILTER_NOT_FOUND);
```

6. 调用 process() 方法，该方法会生成一组 Result 对象，这些 Result 对象中的$item 属性中保存了过滤处理过程的最终结果。遍历这些结果，在相应的$elements 数组的键匹配的情况下，将标签属性 value 的值设置为经过过滤的数据。还应添加过滤处理过程中生成的所有消息。当表单被再次显示时，所有 value 标签属性都会显示经过过滤的结果：

```
$this->filter->process($this->data);
foreach ($this->filter->getResults() as $key => $result) {
  if (isset($this->elements[$key])) {
    $this->elements[$key]
    ->setSingleAttribute('value', $result->item);
    if (isset($result->messages)
        && count($result->messages)) {
     foreach ($result->messages as $message) {
        $this->elements[$key]->addSingleError($message);
      }
    }
  }
}
}
```

具体运行情况

可通过修改前面介绍的 Application\Form\Factory 类创建新的表单工厂类。为进行测试，可使用前面介绍$_POST 过滤器时使用的数据库表。可按照该表中各个字段的设置定义表单元素、过滤器和验证器。

可创建 chap_06_tying_filters_to_form_definitions.php 文件，在该文件中保存表单封装器、页面元素和过滤器关联数组的定义，下面是部分示例代码：

```
<?php
use Application\Form\Generic;

define('VALIDATE_SUCCESS', 'SUCCESS: form submitted ok!');
define('VALIDATE_FAILURE', 'ERROR: validation errors detected');

$wrappers = [

  Generic::INPUT    => ['type' => 'td', 'class' => 'content'],
```

```
  Generic::LABEL    => ['type' => 'th', 'class' => 'label'],
  Generic::ERRORS   => ['type' => 'td', 'class' => 'error']
];

$elements = [
  'first_name'      => [
     'class'        => 'Application\Form\Generic',
     'type'         => Generic::TYPE_TEXT,
     'label'        => 'First Name',
     'wrappers'     => $wrappers,
     'attributes'   => ['maxLength'=>128,'required'=>'']
],
     'last_name'    => [
     'class'        => 'Application\Form\Generic',
     'type'         => Generic::TYPE_TEXT,
     'label'        => 'Last Name',
     'wrappers'     => $wrappers,
     'attributes'   => ['maxLength'=>128,'required'=>'']
  ],
    // 依此类推
];

// 整个表单的配置
$formConfig = [
  'name' => 'prospectsForm',
  'attributes'      => [
  'method'=>'post',
  'action'=>'chap_06_tying_filters_to_form.php'
  ],
    'row_wrapper'   => ['type' => 'tr', 'class'=> 'row'],
    'form_wrapper'  => [
      'type'        =>'table',
      'class'       =>'table',
      'id'          =>'prospectsTable',
      'class'       =>'display','cellspacing'  =>'0'
  ],
  'form_tag_inside_wrapper' => FALSE,
];

$assignments = [
  'first_name'      => [ ['key'     => 'length',
  'params'          => ['min'   => 1, 'max' => 128]],
                       ['key'   => 'alnum',
  'params'          => ['allowWhiteSpace'  => TRUE]],
                       ['key'   => 'required','params' => []] ],
  'last_name'       => [ ['key' => 'length',
```

```
    'params'          => ['min'   => 1, 'max' => 128]],
                         ['key'   => 'alnum',
    'params'          => ['allowWhiteSpace'  => TRUE]],
                          ['key'  => 'required','params' => []] ],
    'address'         => [ ['key' => 'length',
    'params'          => ['max'   => 256]] ],
    'city'            => [ ['key' => 'length',
    'params'          => ['min'   => 1, 'max' => 64]] ],
    'state_province'  => [ ['key' => 'length',
    'params'          => ['min'   => 1, 'max' => 32]] ],
    'postal_code'     => [ ['key' => 'length',
    'params'          => ['min'   => 1, 'max' => 16] ],
                         ['key'   => 'alnum',
    'params'          => ['allowWhiteSpace'  => TRUE]],
                         ['key'   => 'required','params' => []] ],
    'phone'           => [ ['key' => 'phone', 'params'   => []] ],
    'country'         => [ ['key' => 'in_array',
    'params'          => $countries ],
                         ['key'   => 'required','params' => []] ],
    'email'           => [ ['key' => 'email', 'params'   => [] ],
                         ['key'   => 'length',
    'params'          => ['max'   => 250] ],
                         ['key'   => 'required','params' => [] ] ],
    'budget'          => [ ['key' => 'float', 'params'   => []] ]
];
```

可在本例中继续使用前面介绍过的 chap_06_post_data_config_callbacks.php 和 chap_06_post_data_config_messages.php 文件。创建 chap_06_tying_filters_to_form.php 文件，为该文件设置类自动加载功能，并导入下列配置文件：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
include __DIR__ . '/chap_06_post_data_config_messages.php';
include __DIR__ . '/chap_06_post_data_config_callbacks.php';
include __DIR__ . '/chap_06_tying_filters_to_form_definitions.php';
```

为表单工厂类、过滤器和验证器创建实例：

```
use Application\Form\Factory;
use Application\Filter\ { Validator, Filter };
$form = Factory::generate($elements);
$form->setFilter(new Filter($callbacks['filters'],
$assignments['filters']));
```

```
$form->setValidator(new Validator($callbacks['validators'],
$assignments['validators']));
```

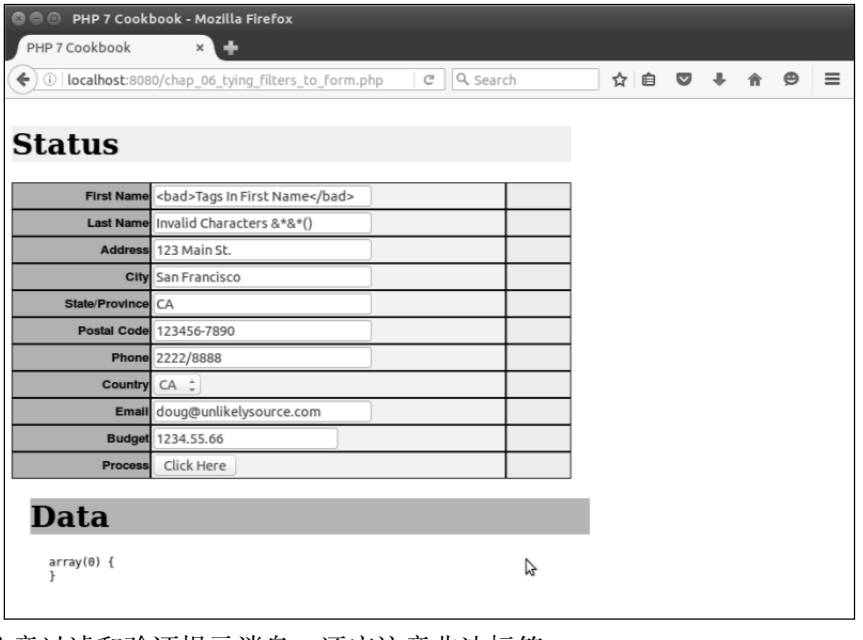
检查$_POST 变量是否传入了数据。如果$_POST 变量确实传入了数据，就执行验证和过滤操作：

```
$message = '';
if (isset($_POST['submit'])) {
  $form->setData($_POST);
  if ($form->validate()) {
    $message = VALIDATE_SUCCESS;
  } else {
    $message = VALIDATE_FAILURE;
  }
  $form->filter();
}
?>
```

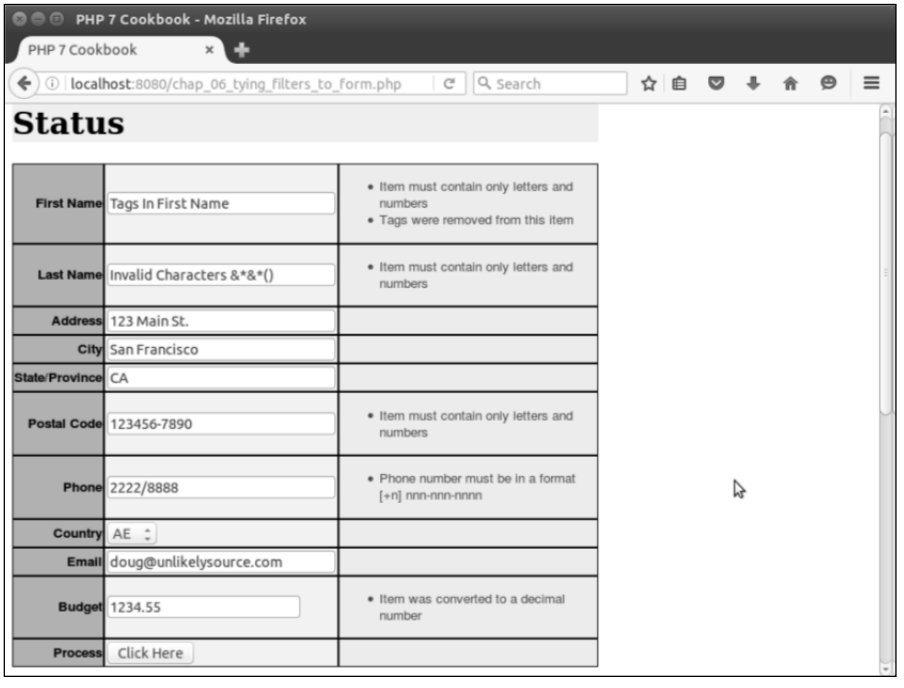
查看逻辑非常简单：只需显示表单。为各种表单元素提供的任何消息和值都会作为验证和过滤处理过程的组成部分被分配：

```
<?= $form->render($form, $formConfig); ?>
```

下面是使用非法数据测试表单的示例：



请注意过滤和验证提示消息，还应注意非法标签：


PHP 7 Cookbook - Mozilla Firefox
PHP 7 Cookbook
localhost:8080/chap_06_tying_filters_to_form.php
Search
Status
First Name
Tags In First Name
• Item must contain only letters and numbers
• Tags were removed from this item
Last Name
Invalid Characters &*&*()
• Item must contain only letters and numbers
Address
123 Main St.
City
San Francisco
State/Province
CA
Postal Code
123456-7890
• Item must contain only letters and numbers
Phone
2222/8888
• Phone number must be in a format [+n] nnn-nnn-nnnn
Country
AE
Email
doug@unlikelysource.com
Budget
1234.55
• Item was converted to a decimal number
Process
Click Here

第 7 章　访问 Web 服务

本章包括以下要点：

- 在 PHP 和 XML 之间转换
- 创建简单的 REST 客户端
- 创建简单的 REST 服务器
- 创建简单的 SOAP 客户端
- 创建简单的 SOAP 服务器

本章主要内容简介

为外部的 Web 服务提供后台查询操作支持，正成为所有 PHP Web 实践应用中不断增长的组成部分。拥有及时提供适当和充足的数据的能力，意味着你可以从客户那里获得更多的业务，以及搭建更多的网站。本章先介绍一些示例，在**可扩展标记语言**（eXtensible Markup Language，XML）和 PHP 之间进行数据转换。然后介绍创建简单的**表述性状态转移**（Representational State Transfer，REST）客户端和服务器的方式。最后介绍创建 SOAP 客户端和服务器的方式。

在 PHP 和 XML 之间转换

当考虑在 PHP 和 XML 之间转换数据类型时，我们通常会将数组视为主要目标。在此前提下，将 PHP 数组转换为 XML 数据的处理过程，与将 XML 数据转换为 PHP 数组的处理过程有本质区别。

也可以考虑对对象进行转换，然而，这在 XML 中很难实现方法。不过使用 get_object_vars()函数可以提取对象中的属性——该函数能够读取对象中的属性，并将这些值保存到数组中。

具体运行情况

1. 先定义 Application\Parse\ConvertXml 类。在该类中创建一些方法，使用这些方法将 XML 数据转换为 PHP 数组，或者将 PHP 数组转换为 XML 数据。这段程序需要使用 SPL 中的 SimpleXMLElement 和 SimpleXMLIterator 类：

```
namespace Application\Parse;
use SimpleXMLIterator;
use SimpleXMLElement;
class ConvertXml
{
}
```

2. 定义 xmlToArray()方法，并将 SimpleXMLIterator 实例作为参数接收。该方法将通过递归方式被调用，而且会使用 XML 文档生成 PHP 数组。我们可以利用 SimpleXMLIterator 类的功能（使用 SimpleXMLIterator 类中的 key()、current()、next()和 rewind()方法）遍历 XML 文档：

```
public function xmlToArray(SimpleXMLIterator $xml) : array
{
  $a = array();
  for( $xml->rewind(); $xml->valid(); $xml->next() ) {
    if(!array_key_exists($xml->key(), $a)) {
      $a[$xml->key()] = array();
    }
    if($xml->hasChildren()){
      $a[$xml->key()][] = $this->xmlToArray($xml->current());
    }
    else{
      $a[$xml->key()] = (array) $xml->current()->attributes();
      $a[$xml->key()]['value'] = strval($xml->current());
    }
  }
  return $a;
}
```

3. 为了将 PHP 数组转换为 XML 数据，可定义两个方法。使用第一个方法 arrayToXml() 设置初始的 SimpleXMLElement 实例，然后调用第二个方法 phpToXml()：

```
public function arrayToXml(array $a)
{
  $xml = new SimpleXMLElement(
  '<?xml version="1.0" standalone="yes"?><root></root>');
  $this->phpToXml($a, $xml);
  return $xml->asXML();
}
```

4. 注意，我们在第二个方法中使用了 get_object_vars() 方法，以应对某个数组元素中存储了对象的情况。纯数字无法用作 XML 的标签，这意味着需要在数字前面添加一些文本：

```
protected function phpToXml($value, &$xml)
{
  $node = $value;
  if (is_object($node)) {
    $node = get_object_vars($node);
  }
  if (is_array($node)) {
    foreach ($node as $k => $v) {
       if (is_numeric($k)) {
         $k = 'number' . $k;
      }
      if (is_array($v)) {
          $newNode = $xml->addChild($k);
          $this->phpToXml($v, $newNode);
      } elseif (is_object($v)) {
          $newNode = $xml->addChild($k);
          $this->phpToXml($v, $newNode);
      } else {
          $xml->addChild($k, $v);
      }
    }
  } else {
     $xml->addChild(self::UNKNOWN_KEY, $node);
  }
}
```

具体处理过程

可以将美国国家气象局的 Web 服务定义语言（Web Services Definition Language，WSDL）文档用作进行测试的 XML 文档。这是一种描述 SOAP 服务的 XML 文档，可从 `http://graphical.weather.gov/xml/SOAP_server/ndfdXMLserver. php?wsdl` 获取它。

可使用 `SimpleXMLIterator` 类提供迭代机制。然后配置类自动加载功能，创建 `Application\Parse\ConvertXml` 实例，使用 `xmlToArray()` 方法将 WSDL 数据转换为 PHP 数组：

```
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Parse\ConvertXml;
$wsdl = 'http://graphical.weather.gov/xml/'
. 'SOAP_server/ndfdXMLserver.php?wsdl';
$xml = new SimpleXMLIterator($wsdl, 0, TRUE);
$convert = new ConvertXml();
var_dump($convert->xmlToArray($xml));
```

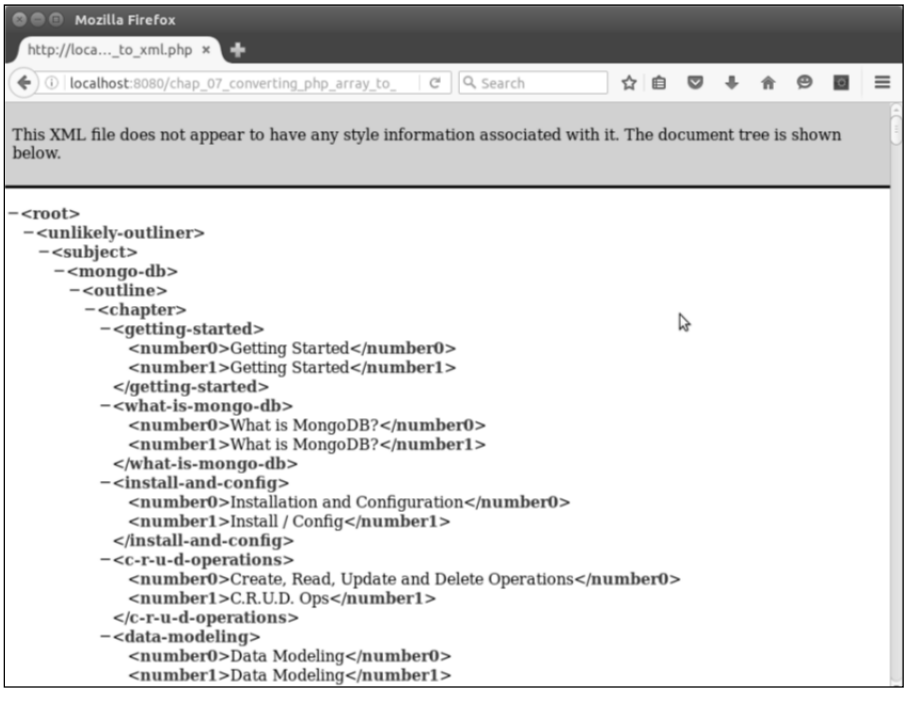
下面是通过执行转换操作得到的数组：

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter07
array(5) {
  ["types"]=>
  array(1) {
    [0]=>
    array(0) {
    }
  }
  ["message"]=>
  array(24) {
    [0]=>
    array(1) {
      ["part"]=>
      array(2) {
        ["@attributes"]=>
        array(2) {
          ["name"]=>
          string(17) "weatherParameters"
          ["type"]=>
          string(25) "tns:weatherParametersType"
        }
        ["value"]=>
        string(0) ""
      }
    }
    [1]=>
    array(1) {
:
```

要将 PHP 数组转换为 WSDL 数据，可使用前面介绍过的 `arrayToXml()`方法。可以将 `source/data/mongo.db.global.php` 文件用作源数据，该文件中含有 MongoDB 数据库视频培训教程（由 O'Reilly Media 公司出品）的大纲。可使用相同的类自动加载配置和 `Application\Parse\ConvertXml` 实例，下面是示例代码：

```
$convert = new ConvertXml();
header('Content-Type: text/xml');
echo $convert->arrayToXml(include CONFIG_FILE);
```

下面是在浏览器中显示的输出结果：



创建简单的 REST 客户端

REST 客户端使用**超文本传输协议**（HTTP）生成发送给外部 Web 服务的请求。通过更改 HTTP 方法，可以使外部服务执行不同操作。尽管可以选用的方法（或操作）不少，但我们将主要使用 `GET` 和 `POST` 方法。本节通过**适配器**软件设计模式介绍两种实现 REST 客户端的方式。

具体处理过程

1. 在定义 REST 客户端的适配器前，需要先定义代表请求和回应信息的通用类。定义一个抽象类，在这个类中定义请求和回应信息都需要用到的方法和属性：

```
namespace Application\Web;

class AbstractHttp
{
```

2. 然后，在该类中定义代表 HTTP 信息的类常量：

```
const METHOD_GET = 'GET';
const METHOD_POST = 'POST';
const METHOD_PUT = 'PUT';
const METHOD_DELETE = 'DELETE';
const CONTENT_TYPE_HTML = 'text/html';
const CONTENT_TYPE_JSON = 'application/json';
const CONTENT_TYPE_FORM_URL_ENCODED =
  'application/x-www-form-urlencoded';
const HEADER_CONTENT_TYPE = 'Content-Type';
const TRANSPORT_HTTP = 'http';
const TRANSPORT_HTTPS = 'https';
const STATUS_200 = '200';
const STATUS_401 = '401';
const STATUS_500 = '500';
```

3. 定义请求和回应信息都需要的属性：

```
protected $uri; // 即 http://xxx.com/yyy
protected $method; // 即 GET、PUT、POST 和 DELETE 方法
protected $headers; // HTTP 报头
protected $cookies; // cookies
protected $metaData; // 与传输有关的信息
protected $transport; // 即 http 或 https 协议
protected $data = array();
```

4. 按顺序为这些属性定义读取器和设置器：

```
public function setMethod($method)
{
  $this->method = $method;
}
public function getMethod()
```

```
{
  return $this->method ?? self::METHOD_GET;
}
// 依此类推
```

5. 一些属性需要通过键进行访问。为了做到这一点，可定义 getXxxByKey()和 setXxxByKey()方法：

```
public function setHeaderByKey($key, $value)
{
  $this->headers[$key] = $value;
}
public function getHeaderByKey($key)
{
  return $this->headers[$key] ?? NULL;
}
public function getDataByKey($key)
{
  return $this->data[$key] ?? NULL;
}
public function getMetaDataByKey($key)
{
  return $this->metaData[$key] ?? NULL;
}
```

6. 有时需要在请求信息中添加参数。可使用在$data 属性中存储 PHP 数组的形式实现这些参数。这样就可以使用 http_build_query()函数创建要请求浏览的 URL：

```
public function setUri($uri, array $params = NULL)
{
  $this->uri = $uri;
  $first = TRUE;
  if ($params) {
    $this->uri .= '?' . http_build_query($params);
  }
}
public function getDataEncoded()
{
  return http_build_query($this->getData());
}
```

7. 根据初始的请求设置$transport 属性：

```
public function setTransport($transport = NULL)
```

```
{
  if ($transport) {
      $this->transport = $transport;
  } else {
      if (substr($this->uri, 0, 5) == self::TRANSPORT_HTTPS) {
          $this->transport = self::TRANSPORT_HTTPS;
      } else {
          $this->transport = self::TRANSPORT_HTTP;
    }
  }
}
```

8. 本例定义 Application\Web\Request 类，使该类能够在我们想要生成请求信息时接收参数，并且能够在实现接收请求信息的服务器时，使用收到的请求信息为属性赋值：

```
namespace Application\Web;
class Request extends AbstractHttp
{
  public function __construct(
    $uri = NULL, $method = NULL, array $headers = NULL,
    array $data = NULL, array $cookies = NULL)
    {
      if (!$headers) $this->headers = $_SERVER ?? array();
      else $this->headers = $headers;
      if (!$uri) $this->uri = $this->headers['PHP_SELF'] ?? '';
      else $this->uri = $uri;
      if (!$method) $this->method =
        $this->headers['REQUEST_METHOD'] ?? self::METHOD_GET;
      else $this->method = $method;
      if (!$data) $this->data = $_REQUEST ?? array();
      else $this->data = $data;
      if (!$cookies) $this->cookies = $_COOKIE ?? array();
      else $this->cookies = $cookies;
      $this->setTransport();
    }
}
```

9. 现在我们可以将注意力转向代表收到信息的类（response class）。可定义 Application\Web\Received 类从名字可以看出，这个类将对从外部 Web 服务收到的数据进行重新封装：

```
namespace Application\Web;
class Received extends AbstractHttp
{
```

```
  public function __construct(
    $uri = NULL, $method = NULL, array $headers = NULL,
    array $data = NULL, array $cookies = NULL)
  {
    $this->uri = $uri;
    $this->method = $method;
    $this->headers = $headers;
    $this->data = $data;
    $this->cookies = $cookies;
    $this->setTransport();
  }
}
```

创建基于流的 REST 客户端

可使用两种方式实现 REST 客户端。第一种方式是使用基础的 PHP I/O 层（也称为**流**）。该层提供了一系列封装器，通过这些封装器可以访问外部的以流的形式存在的资源。默认情况下，所有 PHP 文件命令都会使用这些文件封装器，通过这些封装器可以访问本地文件系统。我们将使用 `http://`或 `https://`封装器实现 `Application\Web\Client\Streams` 适配器。

1. 先定义 `Application\Web\Client\Streams` 类：

```
namespace Application\Web\Client;
use Application\Web\ { Request, Received };
class Streams
{
  const BYTES_TO_READ = 4096;
```

2. 然后定义一个方法，使该方法能够向外部的 Web 服务发送请求。在通过 `GET` 方法处理请求信息时，应向 URI 中添加参数。在通过 `POST` 方法处理请求信息时，应创建流上下文，并在其中包含用于指示接收请求的远程服务执行哪些操作的元数据。在使用 PHP 流时，要生成请求信息只需构造出 URI，而在使用 `POST` 方法时，还需要设置流上下文。之后可使用简单的 `fopen()`方法：

```
public static function send(Request $request)
{
  $data = $request->getDataEncoded();
  $received = new Received();
  switch ($request->getMethod()) {
```

```
        case Request::METHOD_GET :
          if ($data) {
            $request->setUri($request->getUri() . '?' . $data);
          }
          $resource = fopen($request->getUri(), 'r');
          break;
        case Request::METHOD_POST :
          $opts = [
            $request->getTransport() =>
            [
              'method' => Request::METHOD_POST,
              'header' => Request::HEADER_CONTENT_TYPE
              . ': ' . Request::CONTENT_TYPE_FORM_URL_ENCODED,
              'content' => $data
            ]
          ];
          $resource = fopen($request->getUri(), 'w',
          stream_context_create($opts));
          break;
       }
       return self::getResults($received, $resource);
    }
```

3. 最后，获取外部服务回复的结果并将它们封装到 `Received` 对象中。你将看到我们会将收到的数据都解码为 JSON 格式：

```
    protected static function getResults(Received $received, $resource)
    {
      $received->setMetaData(stream_get_meta_data($resource));
      $data = $received->getMetaDataByKey('wrapper_data');
      if (!empty($data) && is_array($data)) {
        foreach($data as $item) {
          if (preg_match('!^HTTP/\d\.\d (\d+?) .*?$!',
              $item, $matches)) {
              $received->setHeaderByKey('status', $matches[1]);
          } else {
              list($key, $value) = explode(':', $item);
              $received->setHeaderByKey($key, trim($value));
          }
        }
      }
      $payload = '';
```

```
  while (!feof($resource)) {
    $payload .= fread($resource, self::BYTES_TO_READ);
  }
  if ($received->getHeaderByKey(Received::HEADER_CONTENT_TYPE)) {
    switch (TRUE) {
       case stripos($received->getHeaderByKey(
                    Received::HEADER_CONTENT_TYPE),
                    Received::CONTENT_TYPE_JSON) !== FALSE:
        $received->setData(json_decode($payload));
        break;
      default :
        $received->setData($payload);
        break;
          }
    }
    return $received;
}
```

定义基于 cURL 的 REST 客户端

下面使用第二种方式实现 REST 客户端，这种方式是以 cURL 扩展为基础的。

1. 本例继续使用前面介绍过的代表请求和收到数据的类。初始的类定义与介绍流客户端时使用的类非常相似：

```
namespace Application\Web\Client;
use Application\Web\ { Request, Received };
class Curl
{
```

2. 本例的 `send()` 方法比流式客户端使用的 `send()` 方法简单很多。我们所需要做的仅是定义一组选项，然后让 cURL 扩展完成其余的工作：

```
public static function send(Request $request)
{
  $data = $request->getDataEncoded();
  $received = new Received();
  switch ($request->getMethod()) {
    case Request::METHOD_GET :
      $uri = ($data)
        ? $request->getUri() . '?' . $data
        : $request->getUri();
```

```
        $options = [
          CURLOPT_URL => $uri,
          CURLOPT_HEADER => 0,
          CURLOPT_RETURNTRANSFER => TRUE,
          CURLOPT_TIMEOUT => 4
        ];
        break;
```

3. POST 方法需要使用略微不同的 cURL 参数：

```
case Request::METHOD_POST :
  $options = [
    CURLOPT_POST => 1,
    CURLOPT_HEADER => 0,
    CURLOPT_URL => $request->getUri(),
    CURLOPT_FRESH_CONNECT => 1,
    CURLOPT_RETURNTRANSFER => 1,
    CURLOPT_FORBID_REUSE => 1,
    CURLOPT_TIMEOUT => 4,
    CURLOPT_POSTFIELDS => $data
  ];
  break;
}
```

4. 调用一系列 cURL 函数并通过 getResults() 方法处理收到的回复数据：

```
$ch = curl_init();
curl_setopt_array($ch, ($options));
if( ! $result = curl_exec($ch))
{
  trigger_error(curl_error($ch));
}
$received->setMetaData(curl_getinfo($ch));
curl_close($ch);
return self::getResults($received, $result);
}
```

5. 使 getResults() 方法将收到的数据封装到 Received 对象中：

```
protected static function getResults(Received $received, $payload)
{
  $type = $received->getMetaDataByKey('content_type');
  if ($type) {
    switch (TRUE) {
      case stripos($type,
```

```
            Received::CONTENT_TYPE_JSON) !== FALSE):
            $received->setData(json_decode($payload));
            break;
        default :
            $received->setData($payload);
            break;
      }
    }
    return $received;
  }
```

具体运行情况

将前面介绍的代码添加到下列类中：

- Application\Web\AbstractHttp
- Application\Web\Request
- Application\Web\Received
- Application\Web\Client\Streams
- Application\Web\Client\Curl

为了了解处理过程，可向谷歌地图的 API 发送一条 REST 请求，以获取两个地点之间的行车路线。还需要创建 API 键，可按照，https://developers.google.com/maps/documentation/directions/get-api-key 上的指示来做。

定义调用脚本 chap_07_simple_rest_client_google_maps_curl.php,使该脚本能够通过 Curl 客户端发出请求。还应定义 chap_07_simple_rest_client_google_maps_streams.php 调用脚本，使该脚本能够使用流式客户端发送请求：

```
<?php
define('DEFAULT_ORIGIN', 'New York City');
define('DEFAULT_DESTINATION', 'Redondo Beach');
define('DEFAULT_FORMAT', 'json');
$apiKey = include __DIR__ . '/google_api_key.php';
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Web\Request;
use Application\Web\Client\Curl;
```

设置行车的起点和目的地：

```
$start  = $_GET['start'] ?? DEFAULT_ORIGIN;
$end    = $_GET['end'] ?? DEFAULT_DESTINATION;
$start  = strip_tags($start);
$end    = strip_tags($end);
```

现在可以为 Request 对象赋值，然后使用该对象生成请求：

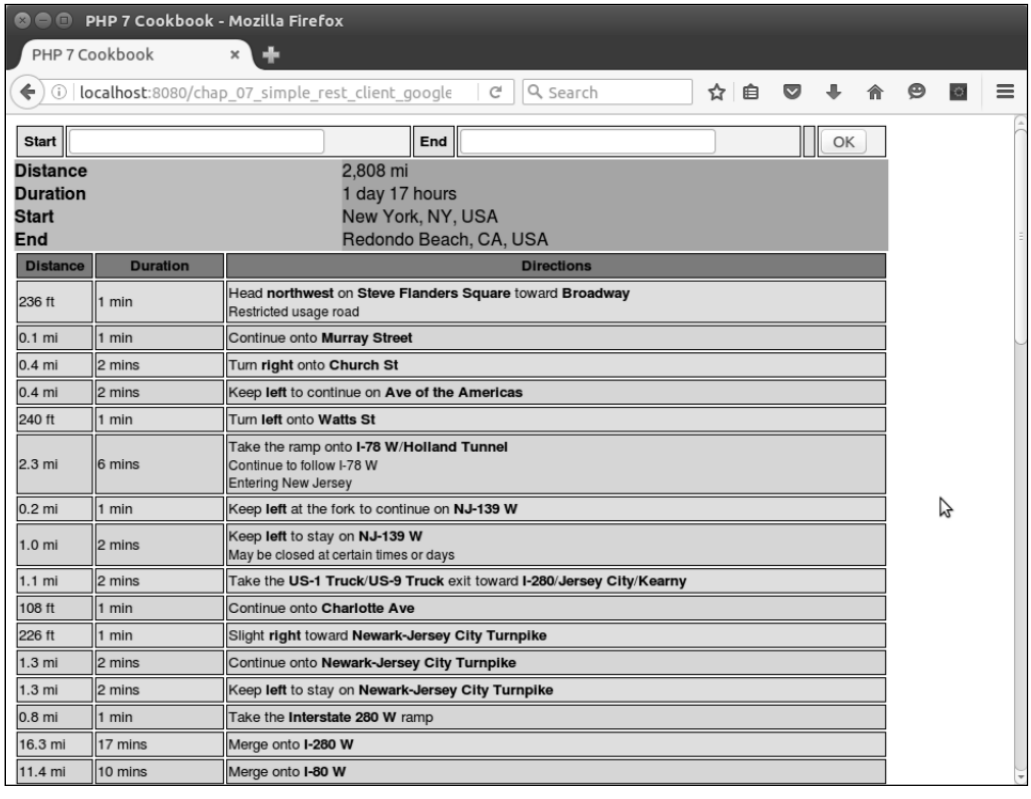
```
$request = new Request(
  'https://maps.googleapis.com/maps/api/directions/json',
  Request::METHOD_GET,
  NULL,
  ['origin' => $start, 'destination' => $end, 'key' => $apiKey],
  NULL
);

$received   = Curl::send($request);
$routes     = $received->getData()->routes[0];
include __DIR__ . '/chap_07_simple_rest_client_google_maps_template.
php';
```

为了了解处理过程，还可以定义一个代表查看逻辑的模板，以显示外部服务根据请求做出的回复：

```
<?php foreach ($routes->legs as $item) : ?>
  <!-- 行车路线信息 -->
  <br>Distance: <?= $item->distance->text; ?>
  <br>Duration: <?= $item->duration->text; ?>
  <!-- 行车方向 -->
  <table>
    <tr>
    <th>Distance</th><th>Duration</th><th>Directions</th>
    </tr>
    <?php foreach ($item->steps as $step) : ?>
    <?php $class = ($count++ & 01) ? 'color1' : 'color2'; ?>
    <tr>
    <td class="<?= $class ?>"><?= $step->distance->text ?></td>
    <td class="<?= $class ?>"><?= $step->duration->text ?></td>
    <td class="<?= $class ?>">
    <?= $step->html_instructions ?></td>
    </tr>
    <?php endforeach; ?>
  </table>
```

```
<?php endforeach; ?>
```

下面是在浏览器中显示的外部服务根据请求回复的结果：



补充说明

PHP 编程规范 7（PSR-7）精确定义了在 PHP 应用程序之间发送请求时应使用的请求和回应对象。本书的附录详细介绍了这方面的内容。

扩展

要详细了解流，请浏览 `http://php.net/manual/en/book.stream.php`。要了解常见问题“HTTP 协议中的 PUT 方法和 POST 方法有哪些区别”的答案，请浏览 `http://stackoverflow.com/questions/107390/whats-the- difference-`

between-apost-and-a-put-http-request。要详细了解从谷歌获取 API 键的方式，请浏览 https://developers.google.com/maps/documentation/directions/get-apikey 和 https://developers.google.com/maps/documentation/directions/intro#Introduction。

创建简单的 REST 服务器

在实现 REST 时需要注意一些事项。回答好下面的问题就可以定义 REST 服务：

- 怎样捕捉到原始请求？
- 发布哪种**应用程序编程接口**（Application Programming Interface，API）？
- 通过何种方式将 HTTP 操作（如 GET、PUT、POST 和 DELETE）与 API 对应起来？

具体处理过程

1. 可根据上一节介绍的代表请求和收到数据的类实现 REST 服务器。下面列出的是上一节介绍过的类：

- Application\Web\AbstractHttp
- Application\Web\Request
- Application\Web\Received

2. 还需要根据 AbstractHttp 类定义代表正式回应的 Application\Web\Response 类。Response 类与其他类的主要差别是，Response 类会将 Application\Web\Request 实例接收为参数。Response 类的主要工作是在__construct()方法中完成的。注意，设置 Content-Type 报头（代表回应主体的数据类型）和状态也很重要：

```
namespace Application\Web;
class Response extends AbstractHttp
{

  public function __construct(Request $request = NULL,
                              $status = NULL, $contentType = NULL)
  {
    if ($request) {
      $this->uri = $request->getUri();
      $this->data = $request->getData();
```

```
        $this->method = $request->getMethod();
        $this->cookies = $request->getCookies();
        $this->setTransport();
      }
      $this->processHeaders($contentType);
      if ($status) {
        $this->setStatus($status);
      }
    }
    protected function processHeaders($contentType)
    {
      if (!$contentType) {
        $this->setHeaderByKey(self::HEADER_CONTENT_TYPE,
          self::CONTENT_TYPE_JSON);
      } else {
        $this->setHeaderByKey(self::HEADER_CONTENT_TYPE,
          $contentType);
      }
    }
    public function setStatus($status)
    {
      $this->status = $status;
    }
    public function getStatus()
    {
      return $this->status;
    }
}
```

3. 现在可以定义 Application\Web\Rest\Server 类了。这个类非常简单，甚至可能简单得让你感到惊奇。实际的工作是在相关的 API 类中完成的：

下面是在 PHP 7 中以批量形式引用类的语法：

```
use Application\Web\ { Request,Response,Received }
namespace Application\Web\Rest;
use Application\Web\ { Request, Response, Received };
class Server
{
  protected $api;
  public function __construct(ApiInterface $api)
  {
    $this->api = $api;
  }
```

4. 定义 listen()方法，使该方法成为请求的目标。实现服务器核心的是下面这行代码：

```
$jsonData = json_decode(file_get_contents('php://input'),true);
```

5. 使 listen()方法能够捕捉原始的输入数据（假定这些数据是 JSON 格式的）：

```
public function listen()
{
  $request  = new Request();
  $response = new Response($request);
  $getPost  = $_REQUEST ?? array();
  $jsonData = json_decode(
    file_get_contents('php://input'),true);
  $jsonData = $jsonData ?? array();
  $request->setData(array_merge($getPost,$jsonData));
```

还需要添加身份验证机制。否则任何人就都能够做出请求并有可能获得敏感数据。我们没有在代表服务器的类中执行身份验证操作，确切地说，是将这项工作留给了 API 类：

```
if (!$this->api->authenticate($request)) {
    $response->setStatus(Request::STATUS_401);
    echo $this->api::ERROR;
    exit;

}
```

6. 将 API 方法与主要的 HTTP 协议方法（GET、PUT、POST 和 DELETE）对应起来：

```
$id = $request->getData()[$this->api::ID_FIELD] ?? NULL;
switch (strtoupper($request->getMethod())) {
  case Request::METHOD_POST :
    $this->api->post($request, $response);
    break;
  case Request::METHOD_PUT :
    $this->api->put($request, $response);
    break;
  case Request::METHOD_DELETE :
    $this->api->delete($request, $response);
    break;
  case Request::METHOD_GET :
```

```
      default :
        // 在没有参数的情况下返回全部结果
      $this->api->get($request, $response);
    }
```

7. 将回应信息封装起来并以 JSON 编码的形式将之发送出去：

```
      $this->processResponse($response);
      echo json_encode($response->getData());
    }
```

8. 应让 processResponse()方法设置回应信息的报头，并确保将结果封装为 Application\Web\Response 对象：

```
    protected function processResponse($response)
    {
      if ($response->getHeaders()) {
        foreach ($response->getHeaders() as $key => $value) {
          header($key . ': ' . $value, TRUE,
                $response->getStatus());
        }
      }
      header(Request::HEADER_CONTENT_TYPE
      . ': ' . Request::CONTENT_TYPE_JSON, TRUE);
      if ($response->getCookies()) {
        foreach ($response->getCookies() as $key => $value) {
          setcookie($key, $value);
        }
      }
    }
```

9. 如前所述，服务器的实际工作是由 API 类完成的。可先定义一个抽象类，确保能够使 get()、put()等方法代表主要的 HTTP 协议方法，并确保所有这些方法都能够将代表请求和回应的对象接收为参数。你可能已经注意到，我们还在这个抽象类中添加了 generateToken()方法，该方法会使用 PHP 7 中的 random_bytes()函数随机生成一组 16 字节的数据：

```
    namespace Application\Web\Rest;
    use Application\Web\ { Request, Response };
    abstract class AbstractApi implements ApiInterface
    {
      const TOKEN_BYTE_SIZE = 16;
      protected $registeredKeys;
```

```
  abstract public function get(Request $request,
                    Response $response);
  abstract public function put(Request $request,
                    Response $response);
  abstract public function post(Request $request,
                    Response $response);
  abstract public function delete(Request $request,
                    Response $response);
  abstract public function authenticate(Request $request);
  public function __construct($registeredKeys, $tokenField)
  {
    $this->registeredKeys = $registeredKeys;
  }
  public static function generateToken()
  {
    return bin2hex(random_bytes(self::TOKEN_BYTE_SIZE));
  }
}
```

10. 还应定义相应的接口，使之起到设计、规范化以及代码开发管理的作用：

```
namespace Application\Web\Rest;
use Application\Web\ { Request, Response };
interface ApiInterface
{
  public function get(Request $request, Response $response);
  public function put(Request $request, Response $response);
  public function post(Request $request, Response $response);
  public function delete(Request $request, Response $response);
  public function authenticate(Request $request);
}
```

11. 下面是以 AbstractApi 类为基础创建的 API 类。这个类利用了第 5 章介绍过的一些用于访问数据库的类：

```
namespace Application\Web\Rest;
use Application\Web\ { Request, Response, Received };
use Application\Entity\Customer;
use Application\Database\ { Connection, CustomerService };

class CustomerApi extends AbstractApi
{
  const ERROR = 'ERROR';
```

```
const ERROR_NOT_FOUND = 'ERROR: Not Found';
const SUCCESS_UPDATE  = 'SUCCESS: update succeeded';
const SUCCESS_DELETE  = 'SUCCESS: delete succeeded';
const ID_FIELD        = 'id'; // field name of primary key
const TOKEN_FIELD     = 'token'; // field used for authentication
const LIMIT_FIELD     = 'limit';
const OFFSET_FIELD    = 'offset';
const DEFAULT_LIMIT   = 20;
const DEFAULT_OFFSET  = 0;

protected $service;

public function __construct($registeredKeys,
                            $dbparams, $tokenField = NULL)
{
  parent::__construct($registeredKeys, $tokenField);
  $this->service = new CustomerService(
    new Connection($dbparams));
}
```

12. CustomerApi 类中的所有方法都能够将请求和回应接收为参数。你可能注意到了，数据是通过 getDataByKey() 方法获得的。但实际的数据库交互操作是由代表服务的类执行的。还应注意，不论出现何种情况，都应该设置 HTTP 状态码，指明客户端的请求是成功了还是失败了。在处理 get() 方法时，应该让它寻找 ID 参数。如果收到了 ID 参数，则发送单个客户的信息。否则，应通过起始 ID 和限定范围发送一组客户的信息：

```
public function get(Request $request, Response $response)
{
  $result = array();
  $id = $request->getDataByKey(self::ID_FIELD) ?? 0;
  if ($id > 0) {
      $result = $this->service->
        fetchById($id)->entityToArray();
  } else {
    $limit = $request->getDataByKey(self::LIMIT_FIELD)
      ?? self::DEFAULT_LIMIT;
    $offset = $request->getDataByKey(self::OFFSET_FIELD)
      ?? self::DEFAULT_OFFSET;
    $result = [];
```

```
    $fetch = $this->service->fetchAll($limit, $offset);
    foreach ($fetch as $row) {
      $result[] = $row;
    }
  }
  if ($result) {
     $response->setData($result);
     $response->setStatus(Request::STATUS_200);
  } else {
     $response->setData([self::ERROR_NOT_FOUND]);
     $response->setStatus(Request::STATUS_500);
  }
}
```

13. 使用 put() 方法插入客户数据：

```
public function put(Request $request, Response $response)
{
  $cust = Customer::arrayToEntity($request->getData(),
                                  new Customer());
  if ($newCust = $this->service->save($cust)) {
      $response->setData(['success' => self::SUCCESS_UPDATE,
                          'id' => $newCust->getId()]);
      $response->setStatus(Request::STATUS_200);
  } else {
      $response->setData([self::ERROR]);
      $response->setStatus(Request::STATUS_500);
  }
}
```

14. 使用 post() 方法更新已经存在的客户记录：

```
public function post(Request $request, Response $response)
{
  $id = $request->getDataByKey(self::ID_FIELD) ?? 0;
  $reqData = $request->getData();
  $custData = $this->service->
    fetchById($id)->entityToArray();
  $updateData = array_merge($custData, $reqData);
  $updateCust = Customer::arrayToEntity($updateData,
  new Customer());
  if ($this->service->save($updateCust)) {
      $response->setData(['success' => self::SUCCESS_UPDATE,
                          'id' => $updateCust->getId()]);
      $response->setStatus(Request::STATUS_200);
```

```
    } else {
        $response->setData([self::ERROR]);
        $response->setStatus(Request::STATUS_500);
    }
}
```

15. 顾名思义，使用 delete()方法可以删除客户记录：

```
public function delete(Request $request, Response $response)
{
    $id = $request->getDataByKey(self::ID_FIELD) ?? 0;
    $cust = $this->service->fetchById($id);
    if ($cust && $this->service->remove($cust)) {
        $response->setData(['success' => self::SUCCESS_DELETE,
                            'id' => $id]);
        $response->setStatus(Request::STATUS_200);
    } else {
        $response->setData([self::ERROR_NOT_FOUND]);
        $response->setStatus(Request::STATUS_500);
    }
}
```

16. 最后，定义 authenticate()方法，通过该方法提供确保 API 被正确使用的低等级机制：

```
public function authenticate(Request $request)
{
    $authToken = $request->getDataByKey(self::TOKEN_FIELD)
      ?? FALSE;
    if (in_array($authToken, $this->registeredKeys, TRUE)) {
        return TRUE;
    } else {
        return FALSE;
    }
}
}
```

具体运行情况

使用上一节介绍的代码创建下列类：

- Application\Web\AbstractHttp
- Application\Web\Request
- Application\Web\Received

使用本节具体处理过程中介绍的代码定义下表列出的类：

Class Application\Web*	对应的步骤
Response	2
Rest\Server	3 ~ 8
Rest\AbstractApi	9
Rest\ApiInterface	10
Rest\CustomerApi	11 ~ 16

现在你可以自由地创建自己的 API 类了。但如果你想创建前面介绍的 Application\Web\Rest\CustomerApi 类，就应确保创建了第 5 章介绍过的下列类：

- Application\Entity\Customer
- Application\Database\Connection
- Application\Database\CustomerService

定义 chap_07_simple_rest_server.php 脚本，使用该脚本调用 REST 服务器：

```
<?php
$dbParams = include __DIR__ . '/../../config/db.config.php';
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Web\Rest\Server;
use Application\Web\Rest\CustomerApi;
$apiKey = include __DIR__ . '/api_key.php';
$server = new Server(new CustomerApi([$apiKey], $dbParams, 'id'));
$server->listen();
```

然后可使用 PHP 7 内置的 Web 服务器监听 8080 端口，以接收 REST 请求：

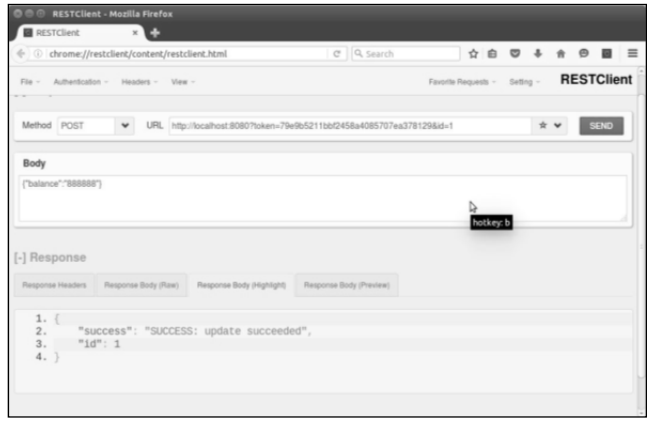
```
php -S localhost:8080 chap_07_simple_rest_server.php
```

为了测试我们编写的 API，可使用 Application\Web\Rest\AbstractApi::generateToken() 方法生成身份验证令牌（如下面的例子所示），并将其存储在 api_key.php 文件中：

```
<?php return '79e9b5211bbf2458a4085707ea378129';
```

这样就可以使用通用的 API 客户端（如前面介绍的）或者浏览器插件（如由 Chao Zhou 开发的 RESTClient，要了解详细信息请浏览 http://restclient.net/）生成请求样例。应确保在请求中包含了身份验证令牌，否则 API 会根据定义拒绝接收请求。

下面是处理 ID 为 1 的客户记录的 POST 请求的例子，该请求会将 balance 字段（代表余额）设置为 888888：



扩展

要实现 REST 服务器，有许多软件库可供选用。我最喜欢的一个 REST 服务器实现范例的网址是 `https://www.leaseweb.com/labs/2015/10/creating-a-simple-rest-api-in-php/`。

各种框架（如 CodeIgniter 和 Zend Framework）也都实现了 REST 服务器。

创建简单的 SOAP 客户端

与实现 REST 客户端或服务器相比，使用 SOAP 客户端或服务器非常简单，因为 PHP 中的 SOAP 扩展提供了这两项功能。

经常有人问：“SOAP 客户端和服务器与 REST 客户端和服务器有什么区别？”SOAP 客户端和服务器在内部使用 XML 的数据格式，SOAP 客户端和服务器也使用 HTTP 协议，但仅会在传输数据时使用，不会用其他 HTTP 方法。REST 客户端和服务器会直接使用 HTTP 协议，可以使用任何数据格式，但 JSON 是首选的数据格式。它们之间另一个主要区别是，SOAP 客户端和服务器可以与 WSDL 语言协同运行，因为 WSDL 语言可以使服务自我描述，所以更易于为大众所接受。因此，公共机构（如国家卫生组织）通常会提供 SOAP 服务。

具体处理过程

本例将创建一个 SOAP 请求，将该请求发送给美国国家气象局网站，要求获取该网站提供的 SOAP 服务。

1. 需要考虑的第一件事是识别 WSDL 文档。**WSDL** 文档是一种描述服务的 XML 文档：

```
$wsdl = 'http://graphical.weather.gov/xml/SOAP_server/'
  . 'ndfdXMLserver.php?wsdl';
```

2. 使用 WSDL 语言创建 `SoapClient` 实例：

```
$soap = new SoapClient($wsdl, array('trace' => TRUE));
```

3. 为获取天气预报信息的请求，初始化一些变量：

```
$units = 'm';
$params = ' ';
$numDays = 7;
$weather = ' ';
$format = '24 hourly';
$startTime = new DateTime();
```

4. 通过 `LatLonListCityNames()` 方法创建 SOAP 请求，使用 WSDL 语言将该请求描述为获取天气预报服务支持的城市名单列表的操作。应使用 XML 格式创建该请求，使用 `SimpleXLMElement` 实例代表该请求：

```
$xml = new SimpleXMLElement($soap->LatLonListCityNames(1));
```

5. 遗憾的是，城市列表和这些城市的经纬度存储在独立的 XML 节点中。因此，应使用 PHP 函数 `array_combine()` 创建关联数组，将城市的经/纬度设置为关联数组的键，将城市的名称设置为关联数组的值。可使用该关联数组实现 HTML 中的下拉菜单，使用 `asort()` 函数按字母表顺序为列表中的城市名称排序：

```
$cityNames = explode('|', $xml->cityNameList);
$latLonCity = explode(' ', $xml->latLonList);
$cityLatLon = array_combine($latLonCity, $cityNames);
asort($cityLatLon);
```

6. 通过 Web 请求获取城市数据：

```
$currentLatLon = (isset($_GET['city'])) ? strip_tags(
                  urldecode($_GET['city'])) : ' ';
```

7. 我们希望使用 `NDFDgenByDay()` 方法创建 SOAP 请求。通过检查 WSDL 描述，

可以确定向 SOAP 服务器发送的参数应具有哪些特征：

```
<message name="NDFDgenByDayRequest">
<part name="latitude" type="xsd:decimal"/>
<part name="longitude" type="xsd:decimal"/>
<part name="startDate" type="xsd:date"/>
<part name="numDays" type="xsd:integer"/>
<part name="Unit" type="xsd:string"/>
<part name="format" type="xsd:string"/>
</message>
```

8. 如果$currentLatLon 变量的值被设置了，我们就可以设置这个请求了。可以将该请求封装在 try {}catch{}代码块中，以便处理抛出异常的情况：

```
if ($currentLatLon) {
  list($lat, $lon) = explode(',', $currentLatLon);
  try {
      $weather = $soap->NDFDgenByDay($lat,$lon,
          $startTime->format('Y-m-d'),$numDays,$unit,$format);
  } catch (Exception $e) {
      $weather .= PHP_EOL;
      $weather .= 'Latitude: ' . $lat . ' | Longitude: ' . $lon;
      $weather .= 'ERROR' . PHP_EOL;
      $weather .= $e->getMessage() . PHP_EOL;
      $weather .= $soap->__getLastResponse() . PHP_EOL;
  }
}
?>
```

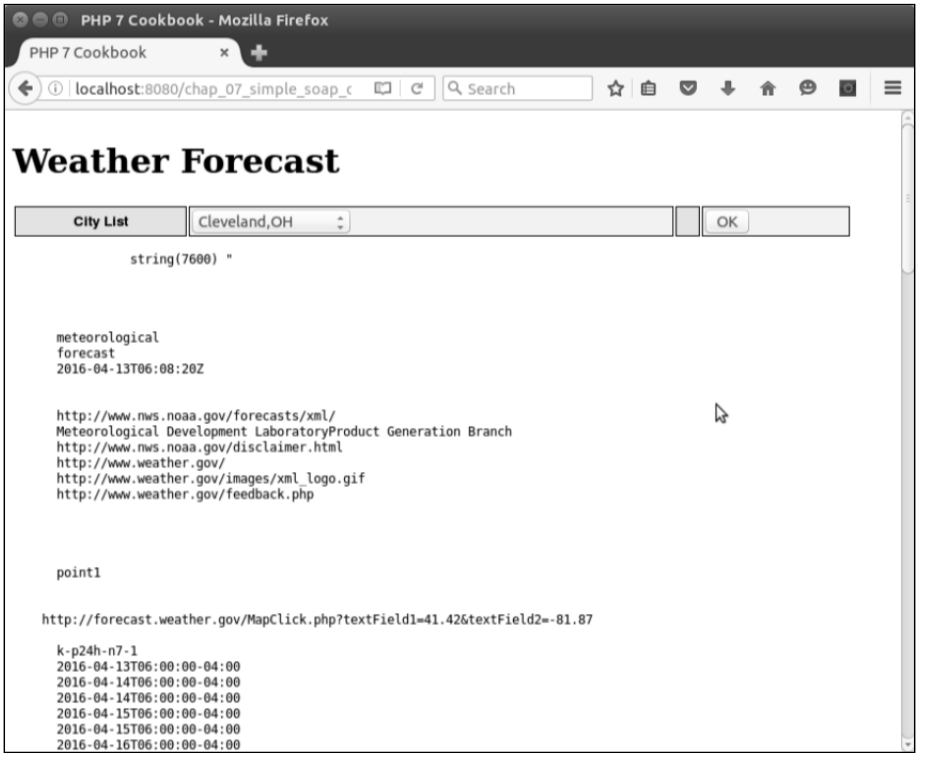
具体运行情况

将前面介绍的代码添加到 chap_07_simple_soap_client_weather_service.php 文件中。然后添加查看逻辑，使该逻辑显示含有城市名称列表的表单和天气预报结果：

```
<form method="get" name="forecast">
<br> City List:
<select name="city">
<?php foreach ($cityLatLon as $latLon => $city) : ?>
<?php $select = ($currentLatLon == $latLon) ? ' selected' : ''; ?>
<option value="<?= urlencode($latLon) ?>" <?= $select ?>>
<?= $city ?></option>
```

```
<?php endforeach; ?>
</select>
<br><input type="submit" value="OK"></td>
</form>
<pre>
<?php var_dump($weather); ?>
</pre>
```

我们请求了俄亥俄州克利夫兰市的天气预报信息，下面是在浏览器中显示的结果：



扩展

要详细了解 SOAP 客户端和服务器与 REST 客户端和服务器之间的区别，请浏览 http://stackoverflow.com/questions/209905/representational-statetransfer-rest-and-simple-object-access-protocol-soap?lq=1。

创建简单的 SOAP 服务器

像创建 SOAP 客户端一样，可使用 PHP 的 SOAP 扩展实现 SOAP 服务器。实现过程中最困难的部分是通过 API 类生成 WSDL 代码。本书没有介绍这个处理过程，因为可供选择的优秀 WSDL 代码生成器有许多。

具体处理过程

1. 先创建通过 SOAP 服务器控制的 API。本例将 `Application\Web\Soap\ProspectsApi` 类定义为 API，通过这个类可以创建、读取、更新和删除 `prospects` 表：

```
namespace Application\Web\Soap;
use PDO;
class ProspectsApi
{
  protected $registerKeys;
  protected $pdo;

  public function __construct($pdo, $registeredKeys)
  {
    $this->pdo = $pdo;
    $this->registeredKeys = $registeredKeys;
  }
}
```

2. 然后定义用于执行创建、读取、更新和删除操作的方法。本例将这些方法命名为 `put()`、`get()`、`post()` 和 `delete()`。还应使这些方法调用通过 PDO 实例执行 SQL 操作的方法。下面是 `get()` 方法的示例：

```
public function get(array $request, array $response)
{
  if (!$this->authenticate($request)) return FALSE;
  $result = array();
  $id = $request[self::ID_FIELD] ?? 0;
  $email = $request[self::EMAIL_FIELD] ?? 0;
  if ($id > 0) {
      $result = $this->fetchById($id);
```

```
        $response[self::ID_FIELD] = $id;
    } elseif ($email) {
        $result = $this->fetchByEmail($email);
        $response[self::ID_FIELD] = $result[self::ID_FIELD] ?? 0;
    } else {
        $limit = $request[self::LIMIT_FIELD]
          ?? self::DEFAULT_LIMIT;
        $offset = $request[self::OFFSET_FIELD]
          ?? self::DEFAULT_OFFSET;
        $result = [];
        foreach ($this->fetchAll($limit, $offset) as $row) {
          $result[] = $row;
        }
    }
    $response = $this->processResponse(
      $result, $response, self::SUCCESS, self::ERROR);
      return $response;
    }

    protected function processResponse($result, $response,
                            $success_code, $error_code)
    {
      if ($result) {
          $response['data'] = $result;
          $response['code'] = $success_code;
          $response['status'] = self::STATUS_200;
      } else {
          $response['data'] = FALSE;
          $response['code'] = self::ERROR_NOT_FOUND;
          $response['status'] = self::STATUS_500;
      }
      return $response;
    }
```

3. 这样就可以通过 API 生成 WSDL 代码。可供选用的基于 PHP 的 WSDL 代码生成器相当多（请参阅本章末尾的内容）。在发布这些方法前，最重要的是添加 phpDocumentor 标签。本例中的两个参数都是数组。下面是应在之前介绍的 API 中添加的全部 WSDL 代码：

```
<?xml version="1.0" encoding="UTF-8"?>
  <wsdl:definitions xmlns:tns="php7cookbook"
```

```
targetNamespace="php7cookbook"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
<wsdl:message name="getSoapIn">
  <wsdl:part name="request" type="tns:array" />
  <wsdl:part name="response" type="tns:array" />
</wsdl:message>
<wsdl:message name="getSoapOut">
  <wsdl:part name="return" type="tns:array" />
</wsdl:message>
<!–为节省篇幅此处略去了一些节点 -->
<wsdl:portType name="CustomerApiSoap">
<!–为节省篇幅此处略去了一些节点 -->
<wsdl:binding name="CustomerApiSoap" type="tns:CustomerApiSoap">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http"
  style="rpc" />
  <wsdl:operation name="get">
    <soap:operation soapAction="php7cookbook#get" />
      <wsdl:input>
         <soap:body use="encoded" encodingStyle=
           "http://schemas.xmlsoap.org/soap/encoding/"
           namespace="php7cookbook" parts="request response" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="encoded" encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/"
          namespace="php7cookbook" parts="return" />
      </wsdl:output>
    </wsdl:operation>
  <!–为节省篇幅此处略去了一些节点-->
  </wsdl:binding>
  <wsdl:service name="CustomerApi">
    <wsdl:port name="CustomerApiSoap"
      binding="tns:CustomerApiSoap">
    <soap:address location="http://localhost:8080/" />
    </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

4. 创建 chap_07_simple_soap_server.php 文件，在该文件中保存 SOAP 服务器的代码。先定义 WSDL 文件和其他必要文件（如数据库配置文件）的位置。如果

设置了 wsdl 参数，就发送 WSDL 描述（而不是尝试处理请求）。本例使用简单的 API 键对请求执行验证操作。创建 SOAP 服务器的实例，为其赋予 API 类的实例，并调用 handle()方法：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
define('WSDL_FILENAME', __DIR__ . '/chap_07_wsdl.xml');

if (isset($_GET['wsdl'])) {
    readfile(WSDL_FILENAME);
    exit;
}
$apiKey = include __DIR__ . '/api_key.php';
require __DIR__ . '/../Application/Web/Soap/ProspectsApi.php';
require __DIR__ . '/../Application/Database/Connection.php';
use Application\Database\Connection;
use Application\Web\Soap\ProspectsApi;
$connection = new Application\Database\Connection(
  include __DIR__ . DB_CONFIG_FILE);
$api = new Application\Web\Soap\ProspectsApi(
  $connection->pdo, [$apiKey]);
$server = new SoapServer(WSDL_FILENAME);
$server->setObject($api);
echo $server->handle();
```

根据 php.ini 文件中的设置，可能需要禁用 WSDL 缓存：

```
ini_set('soap.wsdl_cache_enabled', 0);
```

如果在接收 POST 数据时出了问题，可如下调整这个参数：

```
ini_set('always_populate_raw_post_data', -1);
```

具体运行情况

先创建服务器的 API 类，然后生成 WSDL 代码，就可以轻松对本节介绍的代码进行测试。然后就可以使用 PHP 内置的 Web 服务器，通过下面的命令发送 SOAP 服务：

```
php -S localhost:8080 chap_07_simple_soap_server.php
```

可使用上一节介绍的 SOAP 客户端，通过发送请求测试这个 SOAP 服务：

```
<?php
define('WSDL_URL', 'http://localhost:8080?wsdl=1');
$clientKey = include __DIR__ . '/api_key.php';
try {
  $client = new SoapClient(WSDL_URL);
  $response = [];
  $email = some_email_generated_by_test;
  $email = 'test5393@unlikelysource.com';
  echo "\nGet Prospect Info for Email: " . $email . "\n";
  $request = ['token' => $clientKey, 'email' => $email];
  $result = $client->get($request,$response);
  var_dump($result);

} catch (SoapFault $e) {
  echo 'ERROR' . PHP_EOL;
  echo $e->getMessage() . PHP_EOL;
} catch (Throwable $e) {
  echo 'ERROR' . PHP_EOL;
  echo $e->getMessage() . PHP_EOL;
} finally {
  echo $client->__getLastResponse() . PHP_EOL;
}
```

下面是向电子邮箱 test5393@unlikelysource.com 发送的输出结果：

扩展

在谷歌中可以轻松搜索到十多个可在 PHP 程序中使用的 WSDL 代码生成器。要获得为 `ProspectsApi` 类生成 WSDL 代码的生成器，可浏览 `https:// code.google.com/archive/p/php-wsdl-creator/`。要详细了解 `phpDocumentor` 文档自动生成工具，请浏览 `https://www.phpdoc.org/`。

第 8 章　使用 date/time 数据类型和国际化功能

本章包括以下要点：

- 在查看脚本中使用表情图示或表情符号
- 转换复杂的字符
- 通过浏览器数据获取用户所在地信息
- 根据用户所在地使用适当的格式显示数字
- 根据用户所在地处理货币数据
- 根据用户所在地对日期/时间（date/time）数据类型进行格式化处理
- 创建 HTML 式的国际化日历生成器
- 创建循环事件生成器
- 在不使用 gettext 工具集的情况下处理翻译工作

本章主要内容简介

本章会先通过两个示例介绍 **PHP 7** 引入的 **Unicode** escape 语法，然后介绍通过浏览器数据查明用户**所在地**的方式。本章还会介绍创建代表地区的类的方式，使用这些类可以根据具体地区规定的格式来存储数字、货币、日期和时间信息。最后，本章会介绍创建国家化日历、处理循环事件，以及在不使用 `gettext` 工具集的情况下处理翻译工作的方式。

在查看脚本中使用表情图示或表情符号

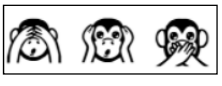
单词 **emoticon**（表情图示）是将单词 *emotion*（情绪）和单词 *icon*（图标）合并到一起生成的词。**Emoji**（表情符号）一词源自日本，是另一种被广泛使用的图形标记。

这些图标（包括笑脸、忍者头像和笑得在地上打滚的表情符号）在网上非常流行，以至于已经成为了社交网络中的一种景致。然而在 PHP 7 出现之前，无法使用 PHP 程序生成这些可爱的小东西。

具体处理过程

1. 首要的是，你需要了解你想要使用哪些代表图标的 Unicode 编码。通过在 Internet 上搜索可以快速找到 Emoji 表情的 Unicode 编码表。下面是非礼勿听、非礼勿视和非礼勿言小猴图标和相应的 Unicode 编码：

U+1F648、U+1F649 和 U+1F64A



2. 任何传输给浏览器的 Unicode 编码都必须通过正确的方式识别。下面是通过 meta 标签实现的最常见的识别方式。应该将字符集设置为 UTF-8：

```
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
    content="text/html;charset=utf-8" />
</head>
```

3. 传统处理方式仅会使用 HTML 代码显示图标。因此，可使用下面的方式：

```
<table>
  <tr>
    <td>&#x1F648;</td>
    <td>&#x1F649;</td>
    <td>&#x1F64A;</td>
  </tr>
</table>
```

4. 从 PHP 7 出现以后，就可以使用语法\u{xxx}创建完整的 Unicode 字符。因此可使用下面的代码显示步骤 1 中介绍的 3 个小猴图标：

```
<table>
  <tr>
    <td><?php echo "\u{1F648}"; ?></td>
    <td><?php echo "\u{1F649}"; ?></td>
    <td><?php echo "\u{1F64A}"; ?></td>
  </tr>
</table>
```

要显示出 emoji 表情符号，你的操作系统和浏览器必须都支持 Unicode 编码，而且还必须拥有正确的字体集。例如，如果你使用的操作系统是 Ubuntu Linux，那么要在浏览器中显示 emoji 表情符号就需要安装 `ttf-ancient-fonts` 软件包。

具体运行情况

PHP 7 引入了一种新语法，使用该语法可以显示任何 Unicode 字符。与其他编程语言不同，这条新的 PHP 语法允许使用数量可变的十六进制数，下面是它的基本语法：

```
\u{xxxx}
```

整条语句必须使用双引号（或 **heredoc** 技术）封装起来。其中的 `xxxx` 可以是任何十六进制数的组合，如 2、4、F 等。

创建 `chap_08_emoji_using_html.php` 文件。确保在该文件中添加了 `meta` 标签，该标签可以通知浏览器，PHP 程序中使用了 UTF-8 字符编码：

```
<!DOCTYPE html>
<html>
  <head>
    <title>PHP 7 Cookbook</title>
    <meta http-equiv="content-type"
      content="text/html;charset=utf-8" />
  </head>
```

设置基础的 HTML 表，并显示一行表情图示或表情符号：

```
<body>
  <table>
    <tr>
      <td>&#x1F648;</td>
      <td>&#x1F649;</td>
      <td>&#x1F64A;</td>
    </tr>
  </table>
</body>
</html>
```

使用 PHP 代码显示一行表情图示或表情符号：

```
<tr>
  <td><?php echo "\u{1F648}"; ?></td>
```

```
  <td><?php echo "\u{1F649}"; ?></td>
  <td><?php echo "\u{1F64A}"; ?></td>
</tr>
```

下面是在 Firefox 浏览器中的显示效果：

扩展

要了解 emoji 代码表，请浏览 `http://unicode.org/emoji/charts/fullemoji-list.html`。

转换复杂的字符

获取了访问整个 Unicode 字符集的能力后，就能够显示许多复杂的字符，尤其是 Latin-1 字符集之外的英文字母表中的字符。

具体处理过程

1. 一些语言使用从右至左(而不是从左至右)的读写顺序，如希伯来语和阿拉伯语。本例将使用 Unicode 字符 `U+202E`，以从右至左的顺序显示文本。下面的代码会显示文本 `txet desreveR`（即反向显示文本 `Reversed text`）：

```
echo "\u{202E}Reversed text";
echo "\u{202D}"; // 以从左至右的顺序返回输出结果
```

在执行了反向显示语句后，不要忘记调用恢复从左至右的显示顺序的字符 U+202D!

2. 另一个注意事项是合成字符的使用。ñ 就是这样的例子，它由字母 n 及其上方的波浪号~组合而成。该字符用在 *mañana*（在西班牙语中为早晨或明天的意思）之类的单词中。这个字符的 Unicode 编码为 U+00F1。下面是显示单词 mañana 的示例：

```
echo "ma\u{00F1}ana"; // 显示单词 mañana
```

3. 然而，这样的处理方式会影响网页被搜索到的可能性。假设用户使用的键盘上没有用于输入这个字符的按键，如果用户尝试使用 man 来搜索单词 mañana，就不会成功找到你编写的网页。

4. 拥有使用整个 Unicode 字符集的能力可以使你拥有更多处理字符的能力。在不使用合成字符的情况下，你可以使用原始字母 n 和 Unicode 合成编码，自己合成出字符。下面 echo 命令的输出结果与前面 echo 命令的输出结果相同，只是合成字符的合成方式不同：

```
echo "man\u{0303}ana"; // 也会显示单词 mañana
```

5. 可使用类似的应用程序显示带音调的字符。在处理法语单词 élève（学生）时，可以使用合成字符显示它，也可以使用 Unicode 合成编码在字符的上方添加音调符号。请观察下面两个示例，这两个示例会生成相同的输出结果，但是它们的处理方式不同：

```
echo "\u{00E9}l\u{00E8}ve";
echo "e\u{0301}le\u{0300}ve";
```

具体运行情况

创建 chap_08_control_and_combining_unicode.php 文件。确保在该文件中添加 meta 标签，该标签可以通知浏览器，PHP 程序中使用了 UTF-8 字符编码：

```
<!DOCTYPE html>
<html>
  <head>
    <title>PHP 7 Cookbook</title>
    <meta http-equiv="content-type"
      content="text/html;charset=utf-8" />
  </head>
```

编写基础的 PHP 和 HTML 程序，以显示前面介绍的示例：

```
<body>
  <pre>
    <?php
      echo "\u{202E}Reversed text"; // 反向显示文本 Reversed text
      //echo "\u{202D}"; // 停止使用反向显示模式
      echo "mañana"; // 显示预先合成的字符
      echo "ma\u{00F1}ana"; // 使用 Unicode 合成编码显示合成字符
      echo "man\u{0303}ana"; // 使用"n"与 Unicode 合成编码
                                (U+0303)生成合成字符
      echo "élève";
      echo "\u{00E9}l\u{00E8}ve"; // 使用 Unicode 合成编码显示合成字符
      echo "e\u{0301}le\u{0300}ve"; // 使用字符"e"与 Unicode 合成编码生成合成
                                      字符
     ?>
    </pre>
</body>
</html>
```

下面是在浏览器中显示的效果：

通过浏览器数据获取用户所在地信息

为提高网站用户的用户体验，以用户所在地区的格式显示信息是很重要的一点。这里的地区泛指世界上的某个具体区域。I.T.社区为地区编码做出的贡献之一是用代表语言的符号和代表国家的符号来组成地区编码。但是当某个用户访问你的网站时，你怎样

才能知道他在哪个地区呢？也许最有用的技巧是检查 HTTP 的语言报头。

具体处理过程

1. 为了封装辨别用户所在地区的功能，可创建 `Application\I18n\Locale` 类。可以通过扩展已经存在的 `Locale` 类（该类是 PHP 的 `Intl` 扩展的组成部分，使用 use 语句为该类起一个别名：`PhpLocale`）创建我们自定义的 `Locale` 类。

I18n 是 **Internationalization**（国际化）的缩写词（其中的 18 代表字母 I 和字母 n 之间的字符个数）。

```
namespace Application\I18n;
use Locale as PhpLocale;
class Locale extends PhpLocale
{
  const FALLBACK_LOCALE = 'en';
  // 在此处添加具体代码
}
```

2. 可使用 `phpinfo(INFO_VARIABLES)` 函数查看收到的请求是什么样的。做完这个实验后，不要忘记立刻禁用这个函数，因为该函数会为潜在的攻击者提供过多的信息：

```
<?php phpinfo(INFO_VARIABLES); ?>
```

3. 地区信息存储在`$_SERVER['HTTP_ACCEPT_LANGUAGE']`超级全局变量中。该变量保存的值会使用一种通用格式：`ll-CC,rl;q=0.n, ll-CC,rl;q=0.n`。下表介绍了这些缩写词的含义：

缩写词	含义
`ll`	用于代表语言的两个小写字母代码
`-`	用于在地区代码 `ll-CC` 中分隔代表语言和国家的代码
`CC`	用于代表国家的两个大写字母代码
`,`	用于将地区代码和备用的**基础地区**代码（通常与语言代码相同）分隔开
`rl`	两个小写字母代码，用于代表推荐的基础地区
`;`	将地区信息与首选语言可能性信息分隔开。如果首选语言可能性信息缺省了，那么其默认值为 q=1 （代表首选语言可能性为 100%），即该语言为首选设置
`q`	首选语言可能性
`0.n`	代表 0.00 至 1.0 之间的某个值。将该值与 100 相乘得到的数值就是实际使用的语言为用户首选语言的概率

4. 我们获得的用户所在地区信息很可能不止一条，例如，用户在自己的计算机上安装多种语言的情况。PHP 的 Locale 类恰好就拥有 acceptFromHttp()方法，使用该方法可以读取用户可接受语言报头字符串（存储在$acceptLangHeader 变量中），从而使我们能够根据获得的信息进行正确的设置：

```
protected $localeCode;
public function setLocaleCode($acceptLangHeader)
{
  $this->localeCode =
    $this->acceptFromHttp($acceptLangHeader);
}
```

5. 接下来定义适当的读取器。getAcceptLanguage()方法将获取存储在$_SERVER['HTTP_ACCEPT_LANGUAGE']变量中的值：

```
public function getAcceptLanguage()
{
  return $_SERVER['HTTP_ACCEPT_LANGUAGE'] ??
    self::FALLBACK_LOCALE;
}
public function getLocaleCode()
{
  return $this->localeCode;
}
```

6. 定义一个构造器，使我们能够以手动方式（即硬编码方式）设置地区代码。并且能够在我们不以手动方式设置地区代码的情况下，从用户的浏览器获取地区代码信息：

```
public function __construct($localeString = NULL)
{
  if ($localeString) {
    $this->setLocaleCode($localeString);
  } else {
    $this->setLocaleCode($this->getAcceptLanguage());
  }
}
```

7. 那么获得了用户所在地的信息后应该做什么呢？请参阅下一个示例。

即使某位用户看起来能够接受一种或多种语言，但他不一定想让浏览器决定使用哪种（些）语言显示他想要看的内容。因此，尽管你肯定可以根据用户的所在地信息设置用于显示网页内容的语言，但仍旧应该为用户提供一个可选语言下拉菜单，使他们能够根据自己的意愿选择语言。

具体运行情况

我们将做下列 3 个实验：

- 从用户的浏览器获取信息
- 使用预设的地区代码 `fr-FR`
- 使用从 RFC 2616 文档中获取的字符串：`da, en-gb;q=0.8, en;q=0.7`

将前面步骤 1 至步骤 6 介绍的代码添加到 `Application\I18n` 文件夹中的 `Locale.php` 文件中。

创建 `chap_08_getting_locale_from_browser.php` 文件，为其设置类自动加载功能并引用 `Application\I18n\Locale` 类：

```
<?php
  require __DIR__ . '/../Application/Autoload/Loader.php';
  Application\Autoload\Loader::init(__DIR__ . '/..');
  use Application\I18n\Locale;
```

可通过在实验中使用的地区代码定义一个字符串数组：

```
$locale = [NULL, 'fr-FR', 'da, en-gb;q=0.8, en;q=0.7'];
```

遍历这 3 个地区代码字符串，以便为新建的类创建实例。显示 `getLocaleCode()` 方法返回的值，查看最终选中的语言设置：

```
echo '<table>';
foreach ($locale as $code) {
  $locale = new Locale($code);
  echo '<tr>
    <td>' . htmlspecialchars($code) . '</td>
    <td>' . $locale->getLocaleCode() . '</td>
  </tr>';
}
echo '</table>';
```

下面是输出结果（其中添加了一点修饰效果）：

扩展

要详细了解PHP的Intl扩展中的`Locale`类，请参阅`http://php.net/manual/en/class.locale.php`。

要详细了解可接受语言报头，请参阅RFC 2616文档中的第14.4小节（`https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html`）。

根据用户所在地使用适当的格式显示数字

世界各地显示数字的格式各不相同。举个简单的例子，在英国人们使用下面的格式书写3 080 512.92：

```
3,080,512.92.
```

然而，在法国人们使用下面的格式书写这个数字：

```
3 080 512,92
```

具体处理过程

在你能够根据用户的所在地以适当的方式显示数字前，你必须先查明用户在哪个地区。使用上一节介绍的`Application\I18n\Locale`类可以做到这一点。可以通过手动方式，也可以根据从用户的浏览器获取的可接受语言报头信息来设置显示网页内容的语言。

1. 可使用`NumberFormatter`类中的`format()`方法，以指定地区的格式输出和解析数字。为`Application\I18n\Locale`类添加一个属性，使用该属性存储`NumberFormatter`类的实例：

```
use NumberFormatter;
protected $numberFormatter;
```

我们最初考虑使用 PHP 函数 setlocale()，通过设置地区代码信息设置显示数字的格式。然而，这种处理方式的问题是，在考虑任何事物时都必须以这个地区代码设置为前提。这就会在处理数据时产生问题，因为数据是根据数据库的规范存储的。setlocale()函数的另一个问题是，它是根据过时的标准（包括 RFC 1766 和 ISO 639）开发的。最后，setlocale()函数需要高度依赖操作系统中地区设置的支持，这会降低代码的可移植性。

2. 通常来讲，现在应该在构造器中设置$numberFormatter 变量。但这种处理方式的问题是，可能会使 Application\I18n\Locale 成为量级最重的类，因为还需要使用它处理货币和日期格式。因此，应创建一个读取器，使用它查看 NumberFormatter 实例是否已经创建。如果 NumberFormatter 实例还没有被创建，就使该读取器创建并返回 NumberFormatter 实例。创建 NumberFormatter 实例时，使用的第一个参数是地区代码。第二参数 NumberFormatter::DECIMAL 代表我们需要的显示格式：

```
public function getNumberFormatter()
{
  if (!$this->numberFormatter) {
     $this->numberFormatter =
       new NumberFormatter($this->getLocaleCode(),
       NumberFormatter::DECIMAL);
  }
  return $this->numberFormatter;
}
```

3. 定义一个方法，使该方法能够根据接收到的数值和地区代码生成代表该数值的字符串：

```
public function formatNumber($number)
{
  return $this->getNumberFormatter()->format($number);
}
```

4. 创建一个方法，使该方法能够根据地区代码解析数字，从而生成 PHP 内置的数值。请注意，根据服务器 ICU 国际化组件的版本，该方法在解析数字失败时可能不会返回 FALSE：

```
public function parseNumber($string)
{
  $result = $this->getNumberFormatter()->parse($string);
  return ($result) ? $result : self::ERROR_UNABLE_TO_PARSE;
}
```

具体运行情况

依据前面介绍的步骤，为 Application\I18n\Locale 类添加代码。创建 chap_08_formatting_numbers.php 文件，为其设置类自动加载功能，并引用 Application\I18n\Locale 类：

```
<?php
  require __DIR__ . '/../Application/Autoload/Loader.php';
  Application\Autoload\Loader::init(__DIR__ . '/..');
  use Application\I18n\Locale;
```

为了了解 Locale 类，可为其创建两个实例，一个用于地区为英国时的语言设置，另一个用于地区为法国时的语言设置。还可以使用一个较大的数值进行测试：

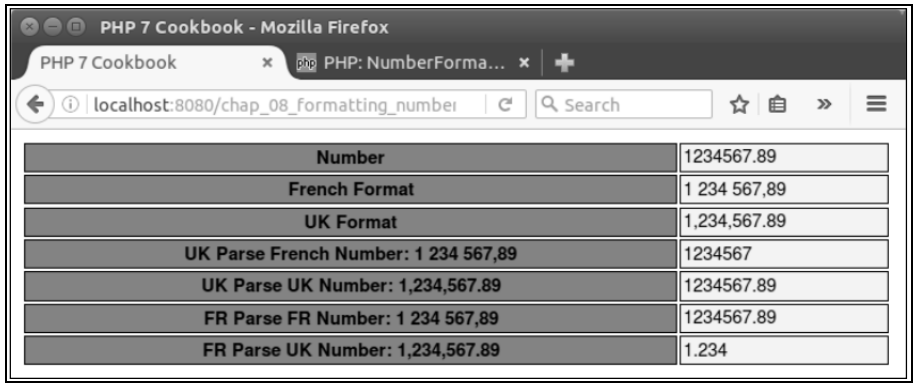
```
  $localeFr = new Locale('fr_FR');
  $localeUk = new Locale('en_GB');
  $number = 1234567.89;
?>
```

最后，可以将 formatNumber() 和 parseNumber() 方法封装在适当的 HTML 显示逻辑中，并查看显示结果：

```
<!DOCTYPE html>
<html>
  <head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
    content="text/html;charset=utf-8" />
  <link rel="stylesheet" type="text/css"
    href="php7cookbook_html_table.css">
  </head>
  <body>
    <table>
      <tr>
        <th>Number</th>
        <td>1234567.89</td>
      </tr>
```

```
      <tr>
        <th>French Format</th>
        <td><?= $localeFr->formatNumber($number); ?></td>
      </tr>
      <tr>
        <th>UK Format</th>
        <td><?= $localeUk->formatNumber($number); ?></td>
      </tr>
      <tr>
        <th>UK Parse French Number:
         <?= $localeFr->formatNumber($number) ?></th>
        <td><?= $localeUk->
         parseNumber($localeFr->formatNumber($number)); ?></td>
      </tr>
      <tr>
        <th>UK Parse UK Number:
         <?= $localeUk->formatNumber($number) ?></th>
        <td><?= $localeUk->
         parseNumber($localeUk->formatNumber($number)); ?></td>
      </tr>
      <tr>
        <th>FR Parse FR Number:
         <?= $localeFr->formatNumber($number) ?></th>
        <td><?= $localeFr->
         parseNumber($localeFr->formatNumber($number)); ?></td>
      </tr>
      <tr>
        <th>FR Parse UK Number:
         <?= $localeUk->formatNumber($number) ?></th>
        <td><?= $localeFr->
         parseNumber($localeUk->formatNumber($number)); ?></td>
      </tr>
    </table>
  </body>
</html>
```

下面是浏览器中的显示效果：



如果地区代码被设置为 `fr_FR`，那么在解析英国格式的数字时，该程序将无法返回正确的数值。同理，如果将地区代码设置为 `en_GB`，在解析法国格式的数字时，该程序也无法返回正确的数值。因此，在执行解析数字操作前，可考虑添加一个验证操作。

扩展

要详细了解 `setlocale()`函数，请浏览 `http://php.net/manual/en/function.setlocale.php`。

要了解不同数字格式会在一些服务器上引发错误，而在另一些服务器上不会引发错误的原因，应检查服务器上安装的 ICU（Unicode 国际化组件）的版本，并浏览 `http://php.net/manual/en/numberformatter.parse.php`。要详细了解 ICU 格式，请浏览 `http://userguide.icuproject.org/formatparse`。

根据用户所在地处理货币数据

处理货币数据的技巧与处理数字的技巧类似，可以继续使用前面介绍过的 `NumberFormatter` 类。然而，这两种技巧的主要差异，同时也是处理货币数据的一个障碍是，为了以适当的格式显示货币数据，需要使用货币代码。

具体处理过程

1. 本处理过程中的第一步是通过某种格式获取货币代码。一种简单的处理方式是将货币代码用作 Application\I18n\Locale 类中构造器方法的参数：

```
const FALLBACK_CURRENCY = 'GBP';
protected $currencyCode;
public function __construct($localeString = NULL,
  $currencyCode = NULL)
{
  // 将这些代码添加到前面介绍的 Locale 类中
  $this->currencyCode = $currencyCode ??
    self::FALLBACK_CURRENCY;
}
```

尽管这种处理方式明显稳妥和可行，但很可能变为折中办法或最简处理方式！这种处理方式还可能彻底消除自动化功能，因为货币代码是无法从 HTTP 报头中获取的。考虑到你已经从本书前面介绍的示例中积累了一定的知识，我们这里将大胆地介绍一个更复杂的解决方案。

2. 首先创建一种查询机制，从而使我们能够通过地区（国家）代码获得相应国家中使用的主要货币的代码。为了了解该处理过程，可使用适配器软件设计模式。我们应该根据这种设计模式创建不同的类，使这些类能够以完全不同的方式运转，但同时又能生成相同的结果。因此，我们需要定义我们想要的结果。为了做到这一点，可创建一个新的类 Application\I18n\IsoCodes。如下所示，这个类拥有所有相关属性，和一个通用型构造器：

```
namespace Application\I18n;
class IsoCodes
{
  public $name;
  public $iso2;
  public $iso3;
  public $iso_numeric;
  public $iso_3166;
  public $currency_name;
```

```
  public $currency_code;
  public $currency_number;
  public function __construct(array $data)
  {
    $vars = get_object_vars($this);
    foreach ($vars as $key => $value) {
      $this->$key = $data[$key] ?? NULL;
    }
  }
}
```

3. 定义一个接口，在该接口中添加方法，使该方法能够执行通过国家代码查找货币代码的查询操作。本例使用 Application\I18n\IsoCodesInterface 接口：

```
namespace Application\I18n;

interface IsoCodesInterface
{
  public function getCurrencyCodeFromIso2CountryCode($iso2)
    : IsoCodes;
}
```

4. 现在可以创建代表执行查询操作的适配器的类，可将之命名为 Application\I18n\IsoCodesDb。在这个类中实现步骤 3 介绍的接口，并使该类将 Application\Database\Connection 实例（请参阅第 1 章）接收为参数，Connection 实例用于执行查询操作。使 IsoCodesDb 类中的构造器方法设置必要的信息，其中包括连接、被执行查询操作的表的名称，以及代表 ISO2 代码的字段。IsoCodesInterface 接口所必需的执行查询操作的方法将提交一条 SQL 语句并返回一个数组，该数组用于创建 IsoCodes 实例：

```
namespace Application\I18n;

use PDO;
use Application\Database\Connection;

class IsoCodesDb implements IsoCodesInterface
{
  protected $isoTableName;
  protected $iso2FieldName;
  protected $connection;
```

```
  public function __construct(Connection $connection,
    $isoTableName, $iso2FieldName)
  {
    $this->connection = $connection;
    $this->isoTableName = $isoTableName;
    $this->iso2FieldName = $iso2FieldName;
  }
  public function getCurrencyCodeFromIso2CountryCode($iso2)
    : IsoCodes
  {
    $sql = sprintf('SELECT * FROM %s WHERE %s = ?',
      $this->isoTableName,
      $this->iso2FieldName);
    $stmt = $this->connection->pdo->prepare($sql);
    $stmt->execute([$iso2]);
    return new IsoCodes($stmt->fetch(PDO::FETCH_ASSOC);
  }
}
```

5. 将注意力重新转到 `Application\I18n\Locale` 类。先向该类中添加一些新属性和类常量：

```
const ERROR_UNABLE_TO_PARSE = 'ERROR: Unable to parse';
const FALLBACK_CURRENCY = 'GBP';

protected $currencyFormatter;
protected $currencyLookup;
protected $currencyCode;
```

6. 在该类中添加一个新方法，使该方法能够通过地区代码字符串获取国家代码。我们可以利用`getRegion()`方法，它来自前面介绍过的PHP的Intl扩展中的`Locale`类。以防万一，还应添加 `getCurrencyCode()`方法：

```
public function getCountryCode()
{
  return $this->getRegion($this->getLocaleCode());
}
public function getCurrencyCode()
{
  return $this->currencyCode;
}
```

7. 与对数字格式的处理方式类似，我们定义 `getCurrencyFormatter()`方法，该

方法与前面介绍的 getNumberFormatter()方法很相似。注意，$currencyFormatter 变量是使用 NumberFormatter 实例定义的，但在初始化该实例时使用了不同的第二参数：

```
public function getCurrencyFormatter()
{
  if (!$this->currencyFormatter) {
    $this->currencyFormatter =
      new NumberFormatter($this->getLocaleCode(),
      NumberFormatter::CURRENCY);
  }
  return $this->currencyFormatter;
}
```

8. 如果执行查询操作的类已经定义好，那么就可以向 Locale 类的构造器中添加货币代码查询操作：

```
public function __construct($localeString = NULL,
  IsoCodesInterface $currencyLookup = NULL)
{
  // 在原有代码的基础上添加这些代码
  $this->currencyLookup = $currencyLookup;
  if ($this->currencyLookup) {
    $this->currencyCode =
      $this->currencyLookup
      ->getCurrencyCodeFromIso2CountryCode($this
      ->getCountryCode())
      ->currency_code;
  } else {
    $this->currencyCode = self::FALLBACK_CURRENCY;
  }
}
```

9. 定义适当的货币格式处理和解析方法。注意，解析货币数据的操作与解析数字的操作不同，如果解析操作执行失败，执行解析操作的方法将返回 FALSE：

```
public function formatCurrency($currency)
{
  return $this->getCurrencyFormatter()
    ->formatCurrency($currency, $this->currencyCode);
}
public function parseCurrency($string)
{
  $result = $this->getCurrencyFormatter()
```

```
        ->parseCurrency($string, $this->currencyCode);
      return ($result) ? $result : self::ERROR_UNABLE_TO_PARSE;
    }
```

具体运行情况

按照前面介绍的步骤，创建下列类：

类	对应的步骤
Application\I18n\IsoCodes	3
Application\I18n\IsoCodesInterface	4
Application\I18n\IsoCodesDb	5

为了了解该处理过程，假定我们已经在 MySQL 数据库中创建好了 iso_country_codes 表，该表应具有下列结构：

```
CREATE TABLE `iso_country_codes` (
  `name` varchar(128) NOT NULL,
  `iso2` varchar(2) NOT NULL,
  `iso3` varchar(3) NOT NULL,
  `iso_numeric` int(11) NOT NULL AUTO_INCREMENT,
  `iso_3166` varchar(32) NOT NULL,
  `currency_name` varchar(32) DEFAULT NULL,
  `currency_code` char(3) DEFAULT NULL,
  `currency_number` int(4) DEFAULT NULL,
  PRIMARY KEY (`iso_numeric`)
) ENGINE=InnoDB AUTO_INCREMENT=895 DEFAULT CHARSET=utf8;
```

按照前面介绍的步骤 6 至步骤 9，向 Application\I18n\Locale 类中添加新代码。创建 chap_08_formatting_currency.php 文件，为其设置类自动加载功能，并引用相应的类：

```
<?php
define('DB_CONFIG_FILE', __DIR__ . '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\I18n\Locale;
use Application\I18n\IsoCodesDb;
use Application\Database\Connection;
```

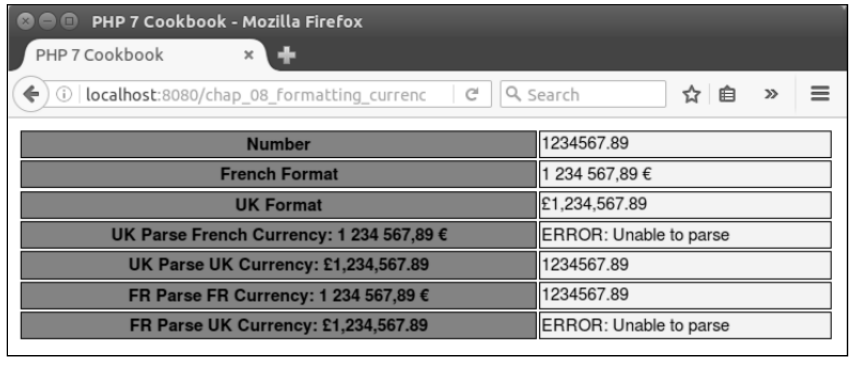
创建 Connection 和 IsoCodesDb 类的实例：

```
$connection = new Connection(include DB_CONFIG_FILE);
$isoLookup = new IsoCodesDb($connection,
  'iso_country_codes', 'iso2');
```

为了了解该处理过程，可创建两个 `Locale` 实例，一个用于处理英国货币，一个用于处理法国货币。还可以使用一个较大的数字进行测试：

```
$localeFr = new Locale('fr-FR', $isoLookup);
$localeUk = new Locale('en_GB', $isoLookup);
$number = 1234567.89;
?>
```

最后，可将 `formatCurrency()` 和 `parseCurrency()` 方法封装到相应的 HTML 显示逻辑中，并查看显示效果。根据上一示例的查看逻辑（为节省篇幅此处不再重复），可得到下列输出结果：



扩展

最新的货币代码是由 **ISO**（**国际标准化组织**）维护的。浏览：`http://www.currency-iso.org/en/home/tables/table-a1.html` 可以获取 **XML** 或 **XLS**（即 **Microsoft Excel** 电子表格格式）格式的货币代码。

根据用户所在地对日期/时间（date/time）数据类型进行格式化处理

世界各地的日期和时间的表示方式各不相同。例如，2016 年 4 月 15 日晚上 7 点 23 分在美国表示为 7:23 PM, 4/15/2016，而在中国会表示为 2016-04-15 19:23。与前面介绍

的数字和货币格式一样，以适当的方式解析日期和时间数据，并以适当的方式对浏览网站的用户显示它们也非常重要。

具体处理过程

1. 先修改 Application\I18n\Locale 类，添加 use 语句，以便引用执行日期格式化操作的类：

```
use IntlCalendar;
use IntlDateFormatter;
```

2. 在 Locale 类中添加一个用于存储 IntlDateFormatter 实例的属性，和一系列预定义常量：

```
const DATE_TYPE_FULL   = IntlDateFormatter::FULL;
const DATE_TYPE_LONG   = IntlDateFormatter::LONG;
const DATE_TYPE_MEDIUM = IntlDateFormatter::MEDIUM;
const DATE_TYPE_SHORT  = IntlDateFormatter::SHORT;

const ERROR_UNABLE_TO_PARSE    = 'ERROR: Unable to parse';
const ERROR_UNABLE_TO_FORMAT   = 'ERROR: Unable to format date';
const ERROR_ARGS_STRING_ARRAY  =
  'ERROR: Date must be string YYYY-mm-dd HH:ii:ss
  or array(y,m,d,h,i,s)';
const ERROR_CREATE_INTL_DATE_FMT =
  'ERROR: Unable to create international date formatter';

protected $dateFormatter;
```

3. 现在可以定义 getDateFormatter()方法，使该方法返回一个 IntlDateFormatter 实例。$type 参数的值与前面介绍的 DATE_TYPE_*常量其中之一相同：

```
public function getDateFormatter($type)
{
  switch ($type) {
    case self::DATE_TYPE_SHORT :
      $formatter = new IntlDateFormatter($this
        ->getLocaleCode(),
        IntlDateFormatter::SHORT,
        IntlDateFormatter::SHORT);
      break;
    case self::DATE_TYPE_MEDIUM :
      $formatter = new IntlDateFormatter($this
        ->getLocaleCode(),
```

```
        IntlDateFormatter::MEDIUM,
        IntlDateFormatter::MEDIUM);
      break;
    case self::DATE_TYPE_LONG :
      $formatter = new IntlDateFormatter($this
        ->getLocaleCode(),
        IntlDateFormatter::LONG,
        IntlDateFormatter::LONG);
      break;
    case self::DATE_TYPE_FULL :
      $formatter = new IntlDateFormatter($this
        ->getLocaleCode(),
        IntlDateFormatter::FULL,
        IntlDateFormatter::FULL);
      break;
    default :
      throw new
InvalidArgumentException(self::ERROR_CREATE_INTL_DATE_FMT);
  }
  $this->dateFormatter = $formatter;
  return $this->dateFormatter;
}
```

4. 定义一个方法，使该方法根据地区代码通过适当的格式显示日期。定义收到的 `$date` 参数的格式有一点难度。不能使用专用的地区代码定义它的格式，否则就需要根据地区编码规则解析它，这会导致不可预测的结果。一个更好的策略是将一组代表年、月、日等日历元素作为整型数值接收。也可以使用一种备用处理机制，就是使该方法仅接收 `YYYY-mm-dd HH:ii:ss` 格式的字符串。可将时区设置为可选参数，单独设置它。下面先初始化这些变量：

```
public function formatDate($date, $type, $timeZone = NULL)
{
  $result   = NULL;
  $year     = date('Y');
  $month    = date('m');
  $day      = date('d');
  $hour     = 0;
  $minutes  = 0;
  $seconds  = 0;
```

5. 然后将代表年、月、日等日历元素的值拆分开：

```
if (is_string($date)) {
```

```
    list($dateParts, $timeParts) = explode('  ', $date);
    list($year,$month,$day) = explode('-',$dateParts);
    list($hour,$minutes,$seconds) = explode(':',$timeParts);
} elseif (is_array($date)) {
    list($year,$month,$day,$hour,$minutes,$seconds) = $date;
} else {
    throw new InvalidArgumentException(self::ERROR_ARGS_STRING_
ARRAY);
}
```

6. 创建一个 `IntlCalendar` 实例，在调用 `format()` 方法时该实例将会被用作参数。应谨慎地使用整数值设置日期和时间：

```
$intlDate = IntlCalendar::createInstance($timeZone,
  $this->getLocaleCode());
$intlDate->set($year,$month,$day,$hour,$minutes,$seconds);
```

7. 获取执行日期格式化操作的实例，并生成结果：

```
  $formatter = $this->getDateFormatter($type);
  if ($timeZone) {
    $formatter->setTimeZone($timeZone);
  }
  $result = $formatter->format($intlDate);
  return $result ?? self::ERROR_UNABLE_TO_FORMAT;
}
```

8. `parseDate()` 方法实际上比执行格式化操作的方法简单。其中唯一的复杂之处是，在日期的数据类型没有确定的情况（出现这种情况的可能性很大）下应执行哪些操作。我们需要做的是遍历所有可能使用的数据类型（仅有 4 种），直到生成结果为止：

```
public function parseDate($string, $type = NULL)
{
  if ($type) {
   $result = $this->getDateFormatter($type)->parse($string);
  } else {
   $tryThese = [self::DATE_TYPE_FULL,
     self::DATE_TYPE_LONG,
     self::DATE_TYPE_MEDIUM,
     self::DATE_TYPE_SHORT];
   foreach ($tryThese as $type) {
   $result = $this->getDateFormatter($type)->parse($string);
     if ($result) {
       break;
     }
   }
```

```
        }
        return ($result) ? $result : self::ERROR_UNABLE_TO_PARSE;
    }
```

具体运行过程

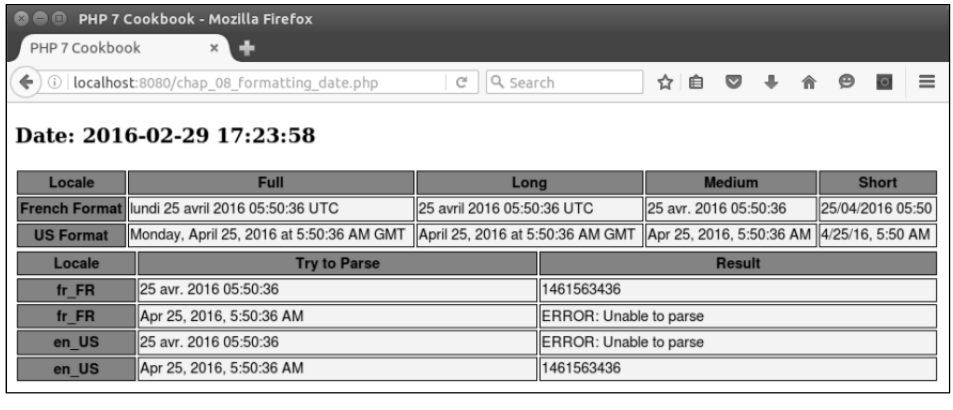
将前面介绍的代码添加到已经存在的 `Application\I18n\Locale` 类中。创建用于进行测试的文件 `chap_08_formatting_date.php`，为其设置类自动加载功能，并创建两个 `Locale` 实例，一个用于处理美国日期格式，一个用于处理法国日期格式：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\I18n\Locale;

$localeFr = new Locale('fr-FR');
$localeUs = new Locale('en_US');
$date = '2016-02-29 17:23:58';
?>
```

使用适当的样式，通过 `formatDate()` 和 `parseDate()` 方法进行测试：

```
echo $localeFr->formatDate($date, Locale::DATE_TYPE_FULL);
echo $localeUs->formatDate($date, Locale::DATE_TYPE_MEDIUM);
$localeUs->parseDate($localeFr->formatDate($date, Locale::DATE_TYPE_
MEDIUM));
// 依此类推
```

下面是输出结果：



扩展

ISO 8601 国际标准详细规定了日期和时间的表示方法。请浏览 https://tools.ietf.org/html/rfc3339 来了解 ISO 8601 国际标准对 Internet 的影响。要详细了解各国不同的日期格式，请浏览 https://en.wikipedia.org/wiki/Date_ format_ by_country。

创建 HTML 式的国际化日历生成器

编写显示日历的程序非常像中学生的作业—— 一个能够生成星期一至星期日的嵌套 for() 循环就足够用了。辨别一个月中具体天数的问题也很容易，通过简单的数组可以轻松解决。难度开始上升的拐点是，需要查明每年的一月一号是星期几。而且，还需要根据用户的所在地以适当的语言和格式显示日期。如你所料，我们仍然会使用前面介绍过的 Application\I18n\Locale 类解决这个问题。

具体处理过程

1. 先创建一个通用类，使用该类存储单独某天的信息。最开始时该类的实例仅会含有一个整数值（存储在$dayOfMonth 属性中）。在下一个示例中，我们将扩展这个类，使其包含事件。因为这个类的主要作用是为$dayOfMonth 变量赋值，所以应该将该变量添加到该类的构造器方法中，并定义__invoke()方法，以便返回$dayOfMonth 变量中保存的值：

```
namespace Application\I18n;

class Day
{
  public $dayOfMonth;
  public function __construct($dayOfMonth)
  {
    $this->dayOfMonth = $dayOfMonth;
  }
  public function __invoke()
  {
    return $this->dayOfMonth ?? '';
  }
}
```

2. 创建一个新类，在该类中包含用于生成日历的方法。使该类将 Application\I18n\Locale 类的实例接收为参数，并定义一系列类常量和属性。格式编码（如 EEEEE 和 MMMM）采用 ICU 日期格式：

```
namespace Application\I18n;

use IntlCalendar;

class Calendar
{

  const DAY_1 = 'EEEEE'; // 代表 T
  const DAY_2 = 'EEEEEE'; // 代表 Tu
  const DAY_3 = 'EEE'; // 代表 Tue
  const DAY_FULL = 'EEEE'; // 代表 Tuesday（星期二）
  const MONTH_1 = 'MMMMM'; // 代表 M
  const MONTH_3 = 'MMM'; // 代表 Mar
  const MONTH_FULL = 'MMMM'; // 代表 March（三月）
  const DEFAULT_ACROSS = 3;
  const HEIGHT_FULL = '150px';
  const HEIGHT_SMALL = '60px';
  protected $locale;
  protected $dateFormatter;
  protected $yearArray;
  protected $height;

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

     // 其余方法会在下面的步骤中介绍
}
```

3. 定义一个方法，使该方法通过 locale 类返回 IntlDateFormatter 实例。这个实例会被存储在一个类属性中，因为该实例会被频繁使用：

```
protected function getDateFormatter()
{
  if (!$this->dateFormatter) {
    $this->dateFormatter =
      $this->locale->getDateFormatter(Locale::DATE_TYPE_FULL);
  }
  return $this->dateFormatter;
}
```

4. 定义核心方法 buildMonthArray()，使该方法创建一个多维数组，嵌套外层数组代表一年中的星期，嵌套内层数组代表一个星期中的天。该方法会将$year、$month、和$timeZone 变量（分别代表年、月和时区）接收为参数，并将$timeZone 变量设置为可选参数。注意，作为变量初始化工作的组成部分，应从$month 变量中减去 1。因为 IntlCalendar::set()方法使用以 0 为起始值的数值代表月份，即 0 代表 1 月，1 代表 2 月，依此类推：

```
public function buildMonthArray($year, $month, $timeZone =
  NULL)
{
$month -= 1;
// Intl 扩展中的 IntlCalendar 类，使用以 0 为起始值的数值代表月份，即 0 代表 1 月，1
   代表 2 月，依此类推
  $day = 1;
  $first = TRUE;
  $value = 0;
  $monthArray = array();
```

5. 创建 IntlCalendar 实例，使用该实例查明当前月份中含有多少天：

```
$cal = IntlCalendar::createInstance(
  $timeZone, $this->locale->getLocaleCode());
$cal->set($year, $month, $day);
$maxDaysInMonth = $cal
  ->getActualMaximum(IntlCalendar::FIELD_DAY_OF_MONTH);
```

6. 使用 IntlDateFormatter 实例查明本月份的第一天是星期几。然后将处理模式设置为 e，从而使我们得到代表星期中各个天的数值：

```
$formatter = $this->getDateFormatter();
$formatter->setPattern('e');
$firstDayIsWhatDow = $formatter->format($cal);
```

7. 通过嵌套循环遍历一个月中的每一天。外层的 while()循环可以确保遍历范围不会超出一个月，内层循环用于遍历星期中的各天。注意，我们使用了 IntlCalendar::get()方法，该方法能够从较大的预定义范围中获取值。当一年中含有的星期数量（由$weekOfYear 变量代表）超过 52 时，还应将$weekOfYear 变量设置为 0：

```
while ($day <= $maxDaysInMonth) {
  for ($dow = 1; $dow <= 7; $dow++) {
    $cal->set($year, $month, $day);
```

```
    $weekOfYear = $cal
      ->get(IntlCalendar::FIELD_WEEK_OF_YEAR);
    if ($weekOfYear > 52) $weekOfYear = 0;
```

8. 查明变量$first是否仍旧被设置为了TRUE。如果该变量仍旧被设置为TRUE，就可以开始向二维数组中添加代表天的数值。否则，就将数组的值设置为NULL。然后，结束所有判断语句并返回该数组。注意，还应确保内层循环的次数不超过一个月含有的最多天数，因此应在嵌套外层if语句的else子句中增加一条if语句。

二维数组中存储的不是代表一个月中的天的简单数值，而是我们刚刚定义的Application\I18n\Day类的实例。

```
if ($first) {
   if ($dow == $firstDayIsWhatDow) {
     $first = FALSE;
     $value = $day++;
  } else {
     $value = NULL;
  }
} else {
       if ($day <= $maxDaysInMonth) {
          $value = $day++;
       } else {
          $value = NULL;
       }
     }
     $monthArray[$weekOfYear][$dow] = new Day($value);
    }
  }
  return $monthArray;
}
```

改进国际化输出结果

1. 先编写一系列较小的方法，可从根据数据类型获取格式化日期的方法开始编写。数据类型决定了是否根据用户所在地发送某一天的日期全称、缩写词或是单个字母（代表该天的）：

```
protected function getDay($type, $cal)
```

```
{
  $formatter = $this->getDateFormatter();
  $formatter->setPattern($type);
  return $formatter->format($cal);
}
```

2. 创建一个方法，使它通过调用刚刚创建的 `getDay()`方法返回由日期构成的 HTML 代码行。如前所述，数据类型指明了显示这些日期的格式：

```
protected function getWeekHeaderRow($type, $cal, $year, $month,
$week)
{
  $output = '<tr>';
  $width = (int) (100/7);
  foreach ($week as $day) {
    $cal->set($year, $month, $day());
    $output .= '<th style="vertical-align:top;"
      width="' . $width . '%">'
      . $this->getDay($type, $cal) . '</th>';
  }
  $output .= '</tr>' . PHP_EOL;
  return $output;
}
```

3. 定义一个非常简单的方法来返回一个星期中每一天的名称。注意，可通过`$day()`变量调用 `Day::__invoke()`方法：

```
protected function getWeekDaysRow($week)
{
  $output = '<tr style="height:' . $this->height . ';">';
  $width = (int) (100/7);
  foreach ($week as $day) {
    $output .= '<td style="vertical-align:top;"
      width="' . $width . '%">'
      . $day() . '</td>';
  }
  $output .= '</tr>' . PHP_EOL;
  return $output;
}
```

4. 定义一个方法，将前面介绍的较小的方法整合到一起，并生成单个月份的日历。先创建代表月份的数组，不过只应该在`$yearArray`数组不可用的情况下才这样做：

```
public function calendarForMonth($year,
    $month,
    $timeZone = NULL,
```

```
    $dayType = self::DAY_3,
    $monthType = self::MONTH_FULL,
    $monthArray = NULL)
{
  $first = 0;
  if (!$monthArray)
    $monthArray = $this->yearArray[$year][$month]
    ?? $this->buildMonthArray($year, $month, $timeZone);
```

5. 应将代表月份的数值减 1，因为 IntlCalendar 类中使用以 0 为起始值的数值代表月份，即 0 代表 1 月，1 代表 2 月，依此类推。然后，使用时区（在存在的情况下）和地区代码创建 IntlCalendar 实例。创建一个 IntlDateFormatter 实例，以便根据地区代码获取月份的名称和其他信息：

```
$month--;
$cal = IntlCalendar::createInstance(
  $timeZone, $this->locale->getLocaleCode());
$cal->set($year, $month, 1);
$formatter = $this->getDateFormatter();
$formatter->setPattern($monthType);
```

6. 遍历代表月份的数组，调用前面介绍的几个较小的方法生成最终的输出结果：

```
$this->height = ($dayType == self::DAY_FULL)
   ? self::HEIGHT_FULL : self::HEIGHT_SMALL;
$html = '<h1>' . $formatter->format($cal) . '</h1>';
$header = ' ';
$body = ' ';
foreach ($monthArray as $weekNum => $week) {
    if ($first++ == 1) {
      $header .= $this->getWeekHeaderRow(
        $dayType, $cal, $year, $month, $week);
    }
    $body .= $this->getWeekDaysRow($dayType, $week);
  }
  $html .= '<table>' . $header . $body .
    '</table>' . PHP_EOL;
  return $html;
}
```

7. 要生成一整年的日历，只需从 1 月遍历到 12 月。为了降低执行外部访问操作的难度，可先定义一个方法以创建代表一整年的数组：

```
public function buildYearArray($year, $timeZone = NULL)
```

```
{
  $this->yearArray = array();
  for ($month = 1; $month <= 12; $month++) {
    $this->yearArray[$year][$month] =
      $this->buildMonthArray($year, $month, $timeZone);
  }
  return $this->yearArray;
}

public function getYearArray()
{
  return $this->yearArray;
}
```

8. 为生成一整年的日历，我们定义 calendarForYear()方法。如果代表年的数组还没有被创建，可调用 buildYearArray()方法。我们考虑希望在一行中显示多少个月的日历，然后调用 calendarForMonth()方法：

```
public function calendarForYear($year,
  $timeZone = NULL,
  $dayType = self::DAY_1,
  $monthType = self::MONTH_3,
  $across = self::DEFAULT_ACROSS)
{
  if (!$this->yearArray) $this->buildYearArray($year,
    $timeZone);
  $yMax = (int) (12 / $across);
  $width = (int) (100 / $across);
  $output = '<table>' . PHP_EOL;
  $month = 1;
  for ($y = 1; $y <= $yMax; $y++) {
    $output .= '<tr>';
    for ($x = 1; $x <= $across; $x++) {
      $output .= '<td style="vertical-align:top;"
        width="' . $width . '%">'
        . $this->calendarForMonth($year, $month,
        $timeZone, $dayType, $monthType,
        $this->yearArray[$year][$month++]) . '</td>';
    }
    $output .= '</tr>' . PHP_EOL;
  }
```

```
    $output .= '</table>';
    return $output;
}
```

具体运行情况

首先，确保你已经创建了前面步骤介绍的 Application\I18n\Locale 类，然后在 Application\I18n 文件夹中创建 Calendar.php 文件，在该文件中添加本节介绍的所有方法。

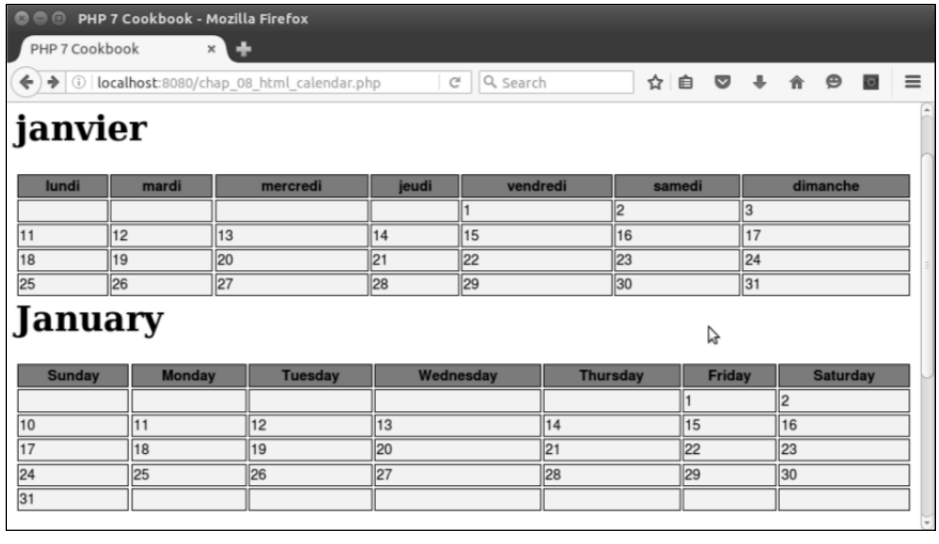
定义调用程序 chap_08_html_calendar.php，为其设置类自动加载功能，并创建 Locale 和 Calendar 实例。同时不要忘记定义代表年和月份的变量：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\I18n\Locale;
use Application\I18n\Calendar;

$localeFr = new Locale('fr-FR');
$localeUs = new Locale('en_US');
$localeTh = new Locale('th_TH');
$calendarFr = new Calendar($localeFr);
$calendarUs = new Calendar($localeUs);
$calendarTh = new Calendar($localeTh);
$year = 2016;
$month = 1;
?>
```

然后，编写适当的查看逻辑，以便显示不同的日历。例如，通过添加参数，显示月份和星期中某天的完整名称：

```
<!DOCTYPE html>
<html>
  <head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
    content="text/html;charset=utf-8" />
  <link rel="stylesheet" type="text/css"
    href="php7cookbook_html_table.css">
  </head>
  <body>
```

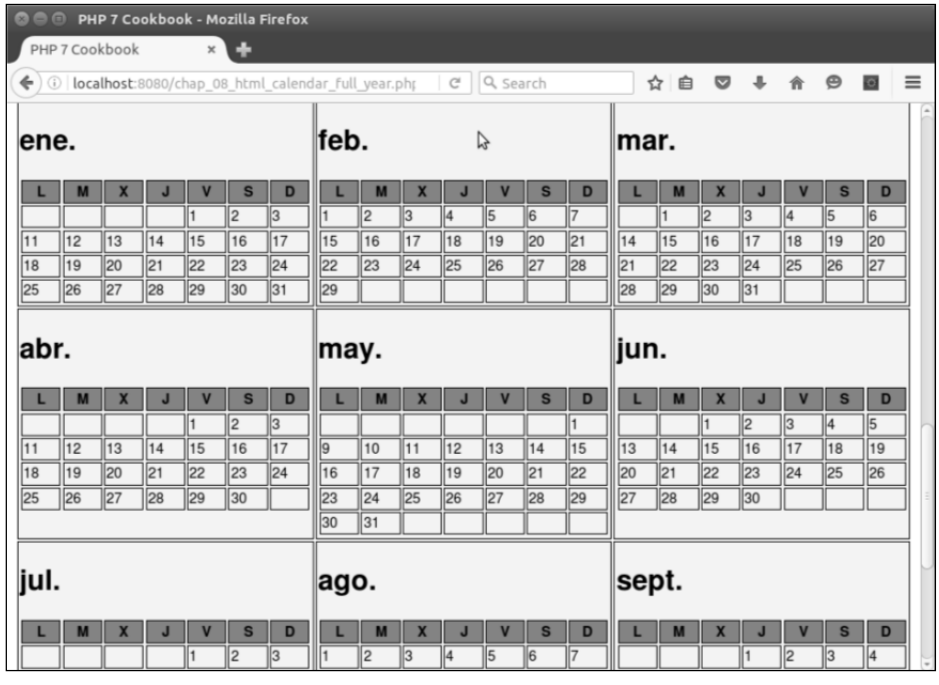
```
    <h3>Year: <?= $year ?></h3>
    <?= $calendarFr->calendarForMonth($year, $month, NULL,
      Calendar::DAY_FULL); ?>
    <?= $calendarUs->calendarForMonth($year, $month, NULL,
      Calendar::DAY_FULL); ?>
    <?= $calendarTh->calendarForMonth($year, $month, NULL,
      Calendar::DAY_FULL); ?>
  </body>
</html>
```



进行一些修改后，还可以使用这段程序显示一整年的日历：

```
$localeTh = new Locale('th_TH');
$localeEs = new Locale('es_ES');
$calendarTh = new Calendar($localeTh);
$calendarEs = new Calendar($localeEs);
$year = 2016;
echo $calendarTh->calendarForYear($year);
echo $calendarEs->calendarForYear($year);
```

下面是在浏览器中使用西班牙语显示一整年日历的效果：



扩展

要详细了解 `IntlDateFormatter::setPattern()`方法中使用的代码，请浏览 `http://userguide.icu-project.org/formatparse/datetime`。

创建循环事件生成器

电子日历的一个非常常见的功能是制定时间表。时间表中列出的事件可能是一次性事件，即某天或某个周末发生了一次。然而，时间表功能还需要起更大的作用，即跟踪反复出现的事件。这就需要记录开始处理事件的日期、事件重复出现的频率（每天、每周、每月），以及事件出现的次数或处理事件所需的时间。

具体处理步骤

1. 在做其他事情之前，最好先创建一个代表事件的类（Event）。使用该类存储的数

据最终会被存储在数据库中。然而，为了将我们的注意力集中在处理过程上，此处不打算介绍将数据存储到数据库中的具体步骤。我们将使用 DateTime 扩展中的许多类，这些类非常适合生成事件：

```
namespace Application\I18n;

use DateTime;
use DatePeriod;
use DateInterval;
use InvalidArgumentException;

class Event
{
  // 此处添加具体代码
}
```

2. 定义一系列类常量和属性。为了有效地利用许多必要的读取器和设置器，应将大多数属性设置为 public（公用的）。将事件出现的时间间隔定义为 sprintf() 格式的字符串；%d 适合处理代表事件出现频率的值（%d 是 sprintf() 方法中的格式转换参数，用于将 sprintf() 方法中的参数转换为带符号的十进制数）：

```
const INTERVAL_DAY = 'P%dD';
const INTERVAL_WEEK = 'P%dW';
const INTERVAL_MONTH = 'P%dM';
const FLAG_FIRST = 'FIRST'; // 每月 1 号
const ERROR_INVALID_END = 'Need to supply either # occurrences or
an end date';
const ERROR_INVALID_DATE = 'String i.e. YYYY-mm-dd or DateTime
instance only';
const ERROR_INVALID_INTERVAL = 'Interval must take the form "P\
d+(D | W | M)"';

public $id;
public $flag;
public $value;
public $title;
public $locale;
public $interval;
public $description;
public $occurrences;
public $nextDate;
```

```
protected $endDate;
protected $startDate;
```

3. 让我们将注意力转向 Event 类的构造器。我们需要收集并设置与事件相关的所有信息。存储这些信息的变量的名称应是自解释的（即变量名称能够反映该变量代表的含义）。

$value 变量的名称不能很好地解释本身代表的意义。这个值最终会被代表事件出现时间间隔的格式化字符串中的值替代。例如，如果用户选择将 INTERVAL_DAY 常量的值赋予$interval 变量，将 2 赋予$value 变量，那么最后生成的代表事件出现时间间隔的字符串就会是"P2D"，这代表该事件每隔一天出现一次（即每 2 天出现一次）。

```
public function __construct($title,
    $description,
    $startDate,
    $interval,
    $value,
    $occurrences = NULL,
    $endDate = NULL,
    $flag = NULL)
{
```

4. 初始化前面介绍的变量。注意，$id 变量（代表事件的 ID）的值是通过伪随机方式生成的，最终$id 变量的值会成为数据库中 events 表中的主键。此处我们使用 md5()方法不是从安全方面考虑的，而是为了能够快速生成一个散列值，以便使事件的 ID 拥有一致的外观：

```
$this->id = md5($title . $interval . $value) . sprintf('%04d',
rand(0,9999));
$this->flag = $flag;
$this->value = $value;
$this->title = $title;
$this->description = $description;
$this->occurrences = $occurrences;
```

5. 如前所述，代表事件出现时间间隔的参数是用于构建适当 DateInterval 实例的 sprintf()格式：

```
try {
  $this->interval = new DateInterval(sprintf($interval, $value));
  } catch (Exception $e) {
```

```
    error_log($e->getMessage());
    throw new InvalidArgumentException(self::ERROR_INVALID_
INTERVAL);
  }
```

6. 要初始化$startDate 变量，可调用 stringOrDate()方法。然后，可通过调用 stringOrDate()或 calcEndDateFromOccurrences()方法为$endDate 变量赋值。如果没有设置完成事件的日期或者事件出现的次数，这段程序就会抛出异常：

```
    $this->startDate = $this->stringOrDate($startDate);
    if ($endDate) {
      $this->endDate = $this->stringOrDate($endDate);
    } elseif ($occurrences) {
      $this->endDate = $this->calcEndDateFromOccurrences();
    } else {
    throw new InvalidArgumentException(self::ERROR_INVALID_END);
    }
    $this->nextDate = $this->startDate;
  }
```

7. stringOrDate()方法中含有几行代码，这些代码用于检查$date 变量的数据类型，然后返回 DateTime 实例或 NULL：

```
  protected function stringOrDate($date)
  {
    if ($date === NULL) {
      $newDate = NULL;
    } elseif ($date instanceof DateTime) {
      $newDate = $date;
    } elseif (is_string($date)) {
      $newDate = new DateTime($date);
    } else {
      throw new InvalidArgumentException(self::ERROR_INVALID_END);
    }
    return $newDate;
  }
```

8. 设置好$occurrences 属性（这样我们就能知道完成事件的日期）后，可通过构造器调用 calcEndDateFromOccurrences()方法。可使用 DatePeriod 类根据开始处理事件的日期执行迭代操作、获取 DateInterval 实例及事件出现的次数：

```
  protected function calcEndDateFromOccurrences()
  {
```

```
    $endDate = new DateTime('now');
    $period = new DatePeriod(
$this->startDate, $this->interval, $this->occurrences);
    foreach ($period as $date) {
      $endDate = $date;
    }
    return $endDate;
}
```

9. 定义魔术方法__toString()，使该方法仅返回事件的标题：

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

10. 可将Event类的最后一个方法定义为getNextDate()，使用该方法生成日历：

```
public function getNextDate(DateTime $today)
{
    if ($today > $this->endDate) {
      return FALSE;
    }
    $next = clone $today;
    $next->add($this->interval);
    return $next;
}
```

11. 让我们将注意力转向上一个示例介绍的 Application\I18n\Calendar 类（代表日历）。稍微处理后，就可以将新定义的 Event 类添加到这个日历中。先在Calendar 类中添加一个新属性$events，然后添加一个通过数组模式添加事件的方法。可使用 Event::$id 属性确保将事件合并起来，而不是使事件被覆盖：

```
protected $events = array();
public function addEvent(Event $event)
{
    $this->events[$event->id] = $event;
}
```

12. 在 Calendar 类中添加 processEvents()方法，该方法能够在创建年日历时将 Event 实例添加到 Day 对象（代表一天）中。先查明某一天中是否有事件，以及 Day 对象的值是否为 NULL。如前所述，一个月的第一天未必是一个星期的第一天，因此需要将 Day 对象的值设置为 NULL。我们不希望将事件添加到非工作日中，因此可调用

Event::getNextDate()方法并查明当前日期是否是工作日。如果当前日期是工作日，可将 Event 实例存储到 Day::$events[]数组中，然后设置下一日期中的 Event 对象：

```
protected function processEvents($dayObj, $cal)
{
  if ($this->events && $dayObj()) {
    $calDateTime = $cal->toDateTime();
    foreach ($this->events as $id => $eventObj) {
      $next = $eventObj->getNextDate($eventObj->nextDate);
      if ($next) {
        if ($calDateTime->format('Y-m-d') ==
            $eventObj->nextDate->format('Y-m-d')) {
          $dayObj->events[$eventObj->id] = $eventObj;
          $eventObj->nextDate = $next;
        }
      }
    }
  }
  return $dayObj;
}
```

我们不会直接比较 DateTime 实例和 IntlCalendar 实例。原因有两个：首先，这两个实例分属不同的类。其次，获取这两个实例的时间有先后之别，而 DateTime 实例中包含了代表小时、分钟、秒数的属性，这会导致这两个实例中的实际值有差异。

13. 在 buildMonthArray()方法中添加一条调用 processEvents()方法的语句：

```
while ($day <= $maxDaysInMonth) {
  for ($dow = 1; $dow <= 7; $dow++) {
    // 在原有代码的基础上添加这些代码
    $dayObj = $this->processEvents(new Day($value), $cal);
    $monthArray[$weekOfYear][$dow] = $dayObj;
  }
}
```

14. 修改 getWeekDaysRow()方法，添加必要的代码，以便在显示日期的表格中显示事件信息：

```
protected function getWeekDaysRow($type, $week)
{
  $output = '<tr style="height:' . $this->height . ';">';
```

```
    $width = (int) (100/7);
    foreach ($week as $day) {
      $events = '';
      if ($day->events) {
        foreach ($day->events as $single) {
          $events .= '<br>' . $single->title;
          if ($type == self::DAY_FULL) {
            $events .= '<br><i>' . $single->description . '</i>';
          }
        }
      }
      $output .= '<td style="vertical-align:top;"
          width="' . $width . '%">'
    . $day() . $events . '</td>';
    }
    $output .= '</tr>' . PHP_EOL;
    return $output;
  }
```

具体运行情况

要将事件与日历绑定到一起，应先在 Application\I18n\Event 类中添加前面步骤 1 至步骤 10 介绍的代码。然后，创建测试脚本 chap_08_recurring_events.php，为其设置类自动加载功能，并引用 Locale 和 Calendar 实例。为了了解该处理过程，我们将'es_ES'用作地区代码：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\I18n\ { Locale, Calendar, Event };

try {
  $year = 2016;
  $localeEs = new Locale('es_ES');
  $calendarEs = new Calendar($localeEs);
```

现在可以定义事件并将它们添加到日历中。我们添加的第一个示例事件需要从 2016 年 1 月 8 日开始处理，总共需要处理 3 天：

```
// 添加一个需要处理 3 天的事件
$title = 'Conf';
```

```
$description = 'Special 3 day symposium on eco-waste';
$startDate = '2016-01-08';
$event = new Event($title, $description, $startDate,
                   Event::INTERVAL_DAY, 1, 2);
$calendarEs->addEvent($event);
```

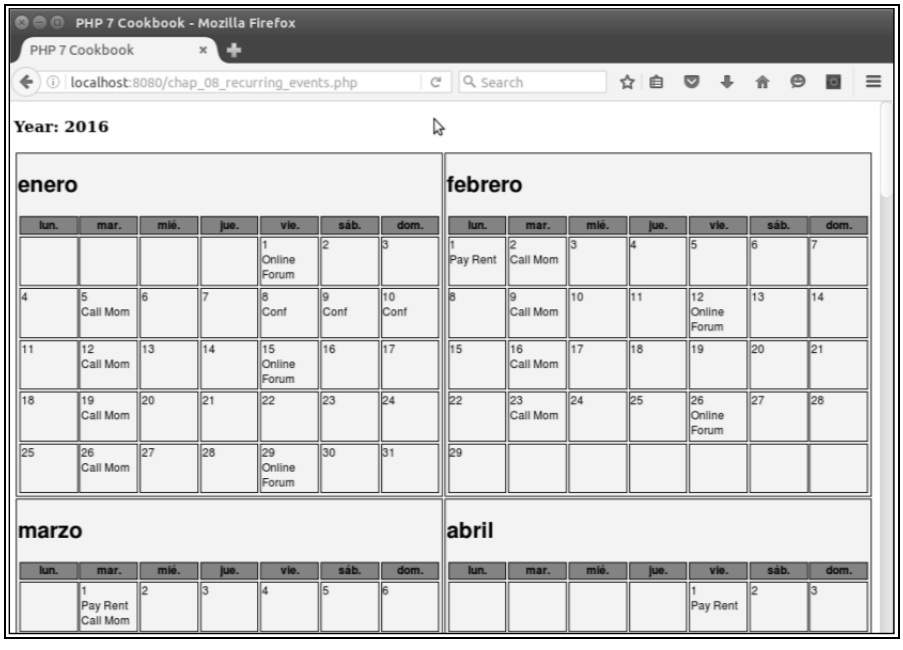
下面是另一个示例，该事件在每个月的第一天都会出现，直到 2017 年 9 月为止：

```
$title = 'Pay Rent';
$description = 'Sent rent check to landlord';
$startDate = new DateTime('2016-02-01');
$event = new Event($title, $description, $startDate,
  Event::INTERVAL_MONTH, 1, '2017-09-01', NULL, Event::FLAG_FIRST);
$calendarEs->addEvent($event);
```

你可以根据自己的需要添加每周、每两周、每月需要处理的事件。然后结束 `try...catch` 代码块，并编写适当的显示逻辑：

```
} catch (Throwable $e) {
  $message = $e->getMessage();
}
?>
<!DOCTYPE html>
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />
  <link rel="stylesheet" type="text/css" href="php7cookbook_html_
table.css">
</head>
<body>
<h3>Year: <?= $year ?></h3>
<?= $calendarEs->calendarForYear($year, 'Europe/Berlin',
    Calendar::DAY_3, Calendar::MONTH_FULL, 2); ?>
<?= $calendarEs->calendarForMonth($year, 1 , 'Europe/Berlin',
    Calendar::DAY_FULL); ?>
</body>
</html>
```

下面是一年中前几个月的日历的显示效果：



扩展

要详细了解 `IntlCalendar` 类中能够与 `get()` 方法一起使用的常量，请浏览 `http://php.net/manual/en/class.intlcalendar.php#intlcalendar.constants`。

在不使用 gettext 工具集的情况下处理翻译工作

翻译是网站国际化工作的重要组成部分。完成翻译工作的一种方式是使用 PHP 的 `gettext` 工具集中的函数，这些函数以安装在本地服务器上的 **GNU** `gettext` 操作系统工具为基础。介绍 `gettext` 工具集的文档很多，而且该工具集也得到了很好的支持，但是它使用较旧的处理方式，并且有明显的缺点。因此，本节介绍另一种在创建适配器时完成翻译工作的方式。

需要注意的一个要点是，可应用于 PHP 代码的程序式翻译工具只能完成有限的单词或短语（也称为 **msgid**，**消息 ID**）的翻译工作。翻译出的目标语言的单词或短语称为 **msgstr**（**消息字符串**）。因此，合成翻译结果通常仅包含相对固定的成分，如菜单、表

单、错误提示或成功提示消息等。为了了解处理翻译工作的步骤，我们假定已经拥有以文本方式存储的网页译文。

如果你需要翻译整个页面的内容，可考虑使用 *Google Translate API*。但应注意要获得这项服务需要付费。也可以通过 *Amazon Mechanical Turk* 网站将翻译工作以较低的价格外包给通晓多种语言的个人。请参阅本节末尾的内容来了解具体的 URL。

具体处理过程

1. 我们将继续使用适配器软件设计模式，以便通过多种方式处理翻译工作。本例将通过适配器方式处理 `.ini` 文件、`.csv` 文件和数据库。

2. 先定义一个接口，稍后我们会使用该接口识别翻译适配器。翻译适配器的必要条件非常简单，只需为指定的消息 ID 返回消息字符串：

```
namespace Application\I18n\Translate\Adapter;
interface TranslateAdapterInterface
{
  public function translate($msgid);
}
```

3. 定义一个与该接口匹配的特性。该特性中会含有实际的必需代码。注意，如果无法找到消息字符串，那么只需使该特性返回消息 ID：

```
namespace Application\I18n\Translate\Adapter;

trait TranslateAdapterTrait
{
  protected $translation;
  public function translate($msgid)
  {
    return $this->translation[$msgid] ?? $msgid;
  }
}
```

4. 现在可以定义第一个适配器。本例介绍的适配器将一个 `.ini` 文件用作需翻译的原文。需要注意的第一件事情是这里使用了前面介绍过的特性（trait）。不同的适配器中含有不同的构造器方法。本例使用 `parse_ini_file()` 函数生成一组键/值对，其中的

键为消息 ID。注意，可使用$filePattern 参数代替地区代码，从而使我们能够加载正确的存储译文的文件：

```
namespace Application\I18n\Translate\Adapter;

use Exception;
use Application\I18n\Locale;

class Ini implements TranslateAdapterInterface
{
  use TranslateAdapterTrait;
  const ERROR_NOT_FOUND = 'Translation file not found';
  public function __construct(Locale $locale, $filePattern)
  {
    $translateFileName = sprintf($filePattern,
                                 $locale->getLocaleCode());
    if (!file_exists($translateFileName)) {
       error_log(self::ERROR_NOT_FOUND . ':' . $translateFileName);
       throw new Exception(self::ERROR_NOT_FOUND);
    } else {
       $this->translation = parse_ini_file($translateFileName);
    }
  }
}
```

5. 除了需要打开译文文件并使用 fgetcsv()函数遍历其中的内容,以检索消息 ID/和消息字符串键/值对外，可通过相同的方式创建适配器 Application\I18n\Translate\Adapter\Csv。下面的代码仅展示了构造器中的差异：

```
public function __construct(Locale $locale, $filePattern)
{
  $translateFileName = sprintf($filePattern,
    $locale->getLocaleCode());
  if (!file_exists($translateFileName)) {
    error_log(self::ERROR_NOT_FOUND . ':' . $translateFileName);
    throw new Exception(self::ERROR_NOT_FOUND);
  } else {
    $fileObj = new SplFileObject($translateFileName, 'r');
    while ($row = $fileObj->fgetcsv()) {
      $this->translation[$row[0]] = $row[1];
    }
  }
}
```

这两个适配器中的一大缺点是需要预先加载整个翻译文件集合，如果翻译文件较多，就会占用很多内存。而且因为需要对翻译文件执行打开和解析操作，所以会降低性能。

6. 现在我们制作第三个适配器，该适配器可以执行数据库查询操作，并且能够避免前两个适配器出现的问题。可使用一条 PDO 准备语句，该语句仅会在开始执行程序时被发送给数据库一次。我们可根据需要执行该语句任意次数，并将消息 ID 用作它的参数。还应注意，需要重写该特性中定义的 `translate()`方法。最后，应使用 `PDOStatement::fetchColumn()`函数获取我们需要的唯一值：

```
namespace Application\I18n\Translate\Adapter;

use Exception;
use Application\Database\Connection;
use Application\I18n\Locale;

class Database implements TranslateAdapterInterface
{
  use TranslateAdapterTrait;
  protected $connection;
  protected $statement;
  protected $defaultLocaleCode;
  public function __construct(Locale $locale,
                              Connection $connection,
                              $tableName)
  {
    $this->defaultLocaleCode = $locale->getLocaleCode();
    $this->connection = $connection;
    $sql = 'SELECT msgstr FROM ' . $tableName
        . ' WHERE localeCode = ? AND msgid = ?';
    $this->statement = $this->connection->pdo->prepare($sql);
  }
  public function translate($msgid, $localeCode = NULL)
  {
    if (!$localeCode) $localeCode = $this->defaultLocaleCode;
    $this->statement->execute([$localeCode, $msgid]);
    return $this->statement->fetchColumn();
  }
}
```

7. 现在可以定义核心类 Translation，该类会与一个或多个适配器绑定。使用一个类常量代表默认的地区代码，使用不同属性代表地区代码、适配器和文本文件格式（稍后会详细介绍）：

```
namespace Application\I18n\Translate;

use Application\I18n\Locale;
use Application\I18n\Translate\Adapter\TranslateAdapterInterface;

class Translation
{
  const DEFAULT_LOCALE_CODE = 'en_GB';
  protected $defaultLocaleCode;
  protected $adapter = array();
  protected $textFilePattern = array();
```

8. 应在 Translation 类的构造器中确定地区代码，并将初始适配器设置为与该地区对应的适配器。通过这种方式，我们可以管理多个适配器：

```
public function __construct(TranslateAdapterInterface $adapter,
              $defaultLocaleCode = NULL,
              $textFilePattern = NULL)
{
  if (!$defaultLocaleCode) {
    $this->defaultLocaleCode = self::DEFAULT_LOCALE_CODE;
  } else {
    $this->defaultLocaleCode = $defaultLocaleCode;
  }
  $this->adapter[$this->defaultLocaleCode] = $adapter;
  $this->textFilePattern[$this->defaultLocaleCode] =
$textFilePattern;
}
```

9. 下面定义一系列设置器，以获取更多灵活性：

```
public function setAdapter($localeCode, TranslateAdapterInterface
$adapter)
{
  $this->adapter[$localeCode] = $adapter;
}
public function setDefaultLocaleCode($localeCode)
{
  $this->defaultLocaleCode = $localeCode;
}
```

```
public function setTextFilePattern($localeCode, $pattern)
{
  $this->textFilePattern[$localeCode] = $pattern;
}
```

10. 定义 PHP 中的魔术方法__invoke()，从而使我们能够直接调用处理翻译工作的实例，通过消息 ID 返回对应的消息字符串：

```
public function __invoke($msgid, $locale = NULL)
{
  if ($locale === NULL) $locale = $this->defaultLocaleCode;
  return $this->adapter[$locale]->translate($msgid);
}
```

11. 添加一个方法，使该方法能够通过存储译文的文件返回翻译出的目标文本。注意，可使用数据库查询操作代替这个处理步骤。我们没有在适配器中包含查询数据库的功能，因为这与适配器的功能南辕北辙；我们的目的仅是通过指定的键（如存储译文的文本文件的名称）返回一大段代码：

```
public function text($key, $localeCode = NULL)
{
  if ($localeCode === NULL) $localeCode =
       $this->defaultLocaleCode;
  $contents = $key;
  if (isset($this->textFilePattern[$localeCode])) {
    $fn = sprintf($this->textFilePattern[$localeCode],
                  $localeCode, $key);
    if (file_exists($fn)) {
      $contents = file_get_contents($fn);
    }
  }
  return $contents;
}
```

具体运行情况

应先定义用于存储翻译文件的目录结构。为了了解该处理过程，可创建/path/to/project/files/data/languages 目录。在这个目录结构中，创建代表不同地区的子目录。可创建 de_DE、fr_FR、en_GB 和 es_ES 子目录，使用它们分别代表德国、法国、英国和西班牙。

然后，应创建不同的翻译文件。如在 data/languages/es_ES/translation.ini 文件中存储西班牙语译文：

```
Welcome=Bienvenido
About Us=Sobre Nosotros
Contact Us=Contáctenos
Find Us=Encontrarnos
click=clic para más información
```

同理，要使用 CSV 适配器，可将译文存储在 CSV 文件中，如 data/languages/es_ES/translation.csv：

```
"Welcome","Bienvenido"
"About Us","Sobre Nosotros"
"Contact Us","Contáctenos"
"Find Us","Encontrarnos"
"click","clic para más información"
```

创建数据库表 translation，在该表中添加相同的译文。这两种处理方式的主要差别是，数据库的表中含有 3 个字段：msgid（代表消息 ID）、msgstr（代表消息字符串）和 locale_code（代表地区代码）：

```
CREATE TABLE `translation` (
  `msgid` varchar(255) NOT NULL,
  `msgstr` varchar(255) NOT NULL,
  `locale_code` char(6) NOT NULL DEFAULT '',
  PRIMARY KEY (`msgid`,`locale_code`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

使用前面介绍的代码定义下列类：

- Application\I18n\Translate\Adapter\TranslateAdapterInterface
- Application\I18n\Translate\Adapter\TranslateAdapterTrait
- Application\I18n\Translate\Adapter\Ini
- Application\I18n\Translate\Adapter\Csv
- Application\I18n\Translate\Adapter\Database
- Application\I18n\Translate\Translation

创建测试文件 chap_08_translation_database.php，测试基于数据库的翻译适配器。应为该程序实现类自动加载功能，引用适当的类并创建 Locale 和 Connection 实例。注意，TEXT_FILE_PATTERN 常量是 sprintf()格式的，在这种格式中地区代

码和文件名会被替换：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
define('TEXT_FILE_PATTERN', __DIR__ . '/../data/languages/%s/%s.txt');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\I18n\Locale;
use Application\I18n\Translate\ { Translation, Adapter\Database };
use Application\Database\Connection;

$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
$locale = new Locale('fr_FR');
```

创建翻译适配器实例并使用该实例创建 `Translation` 实例：

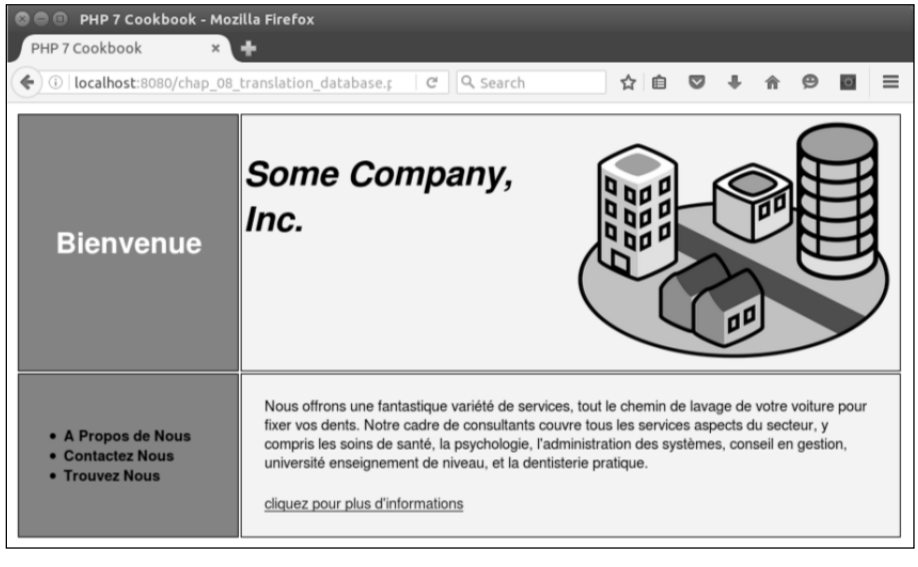
```
$adapter = new Database($locale, $conn, 'translation');
$translate = new Translation($adapter, $locale->getLocaleCode(), TEXT_
FILE_PATTERN);
?>
```

使用`$translate` 变量中存储的 `Translation` 实例编写显示逻辑：

```
<!DOCTYPE html>
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />
  <link rel="stylesheet" type="text/css" href="php7cookbook_html_
table.css">
</head>
<body>
<table>
  <tr>
  <th><h1 style="color:white;"><?= $translate('Welcome') ?></h1></th>
  <td>
    <div style="float:left;width:50%;vertical-align:middle;">
    <h3 style="font-size:24pt;"><i>Some Company, Inc.</i></h3>
    </div>
    <div style="float:right;width:50%;">
    <img src="jcartier-city.png" width="300px"/>
    </div>
  </td>
</tr>
```

```
<tr>
  <th>
    <ul>
      <li><?= $translate('About Us') ?></li>
      <li><?= $translate('Contact Us') ?></li>
      <li><?= $translate('Find Us') ?></li>
    </ul>
  </th>
  <td>
    <p>
    <?= $translate->text('main_page'); ?>
    </p>
    <p>
    <a href="#"><?= $translate('click') ?></a>
    </p>
    </td>
</tr>
</table>
</body>
</html>
```

你可以自己进行其他类似的实验，如修改地区代码以获得其他语言的译文、使用另一个适配器测试不同的数据源。下面是使用地区代码 `fr_FR` 和数据库翻译适配器获得的输出结果：



扩展

要详细了解 Google Translation API，请浏览 https://cloud.google. com/translate/v2/translating-text-with-rest。

要详细了解 Amazon Mechanical Turk 网站，请浏览 https://www.mturk.com/mturk/welcome。要详细了解 gettext 工具集，请浏览 http://www.gnu.org/software/gettext/manual/gettext.html。

第 9 章　开发中间件

本章包括以下要点：

- 通过中间件执行验证操作
- 使用中间件实现访问控制
- 使用缓存提高性能
- 实现路由功能
- 实现框架系统间的相互调用
- 使用中间件实现跨编程语言功能

本章主要内容简介

术语被发明出来，继而被使用，继后术语的概念范围被大幅度扩展，这种情形在 IT 界中已经司空见惯。术语“中间件”也不会例外。该术语很可能是在 2000 年互联网工程任务组（Internet Engineering Task Force，IETF）举行的会议中第一次出现的。最初，该术语是指在传输层（即 TCP/IP）和应用层之间运行的软件。最近，尤其是第 7 号 PHP 推荐标准（PHP Standard Recommendation number 7，PSR-7）出现后，中间件在 PHP 的世界中是指 Web 客户端-服务器环境。

本节中的示例会使用本书附录中介绍的具体类。

通过中间件执行验证操作

中间件的一个非常重要的用途是提供验证功能。大多数基于网页的应用程序都需要

通过用户名和密码核实用户的身份。通过将 PSR-7 标准（编程规范）融入到执行验证操作的类中，可以提高这些类的通用性，也就是说，在能够提供兼容 PSR-7 标准的请求和回应对象的框架中，这些类可以提供足够的安全性。

具体处理过程

1. 先定义 `Application\Acl\AuthenticateInterface` 接口。使用这个接口可以支持适配器软件设计模式，以便通过与各种适配器（每个适配器都能够从不同数据源（如文件、通过 OAuth2 标准访问受保护的数据等）获取验证信息）组合使用，使 `Authenticate` 类（用于执行验证操作）获得更高的通用性。注意，应使用 PHP 7 提供的功能定义返回值的数据类型：

```
namespace Application\Acl;
use Psr\Http\Message\ { RequestInterface, ResponseInterface };
interface AuthenticateInterface
{
  public function login(RequestInterface $request) :
    ResponseInterface;
}
```

通过定义一个方法，使该方法接收符合 PSR-7 标准的请求，并生成符合 PSR-7 标准的回应，就能够使这个接口具有通用性。

2. 定义适配器，以实现该接口必须具备的 `login()` 方法。确定需使用哪些类，并定义适当的常量和属性。下面的构造器使用了 `Application\Database\Connection` 类，请参阅第 5 章：

```
namespace Application\Acl;
use PDO;
use Application\Database\Connection;
use Psr\Http\Message\ { RequestInterface, ResponseInterface };
use Application\MiddleWare\ { Response, TextStream };
class DbTable implements AuthenticateInterface
{
  const ERROR_AUTH = 'ERROR: authentication error';
  protected $conn;
  protected $table;
```

```
  public function __construct(Connection $conn, $tableName)
  {
    $this->conn = $conn;
    $this->table = $tableName;
  }
```

3. 核心方法 `login()`会从代表请求的对象中提取用户名和密码。这样我们就可以直接执行数据库查询操作。如果数据库中有相符的记录，就可以将该用户的信息存储到回应对象的主体中（以 JSON 编码的形式）：

```
public function login(RequestInterface $request) :
  ResponseInterface
{
  $code = 401;
  $info = FALSE;
  $body = new TextStream(self::ERROR_AUTH);
  $params = json_decode($request->getBody()->getContents());
  $response = new Response();
  $username = $params->username ?? FALSE;
  if ($username) {
      $sql = 'SELECT * FROM ' . $this->table
        . ' WHERE email = ?';
      $stmt = $this->conn->pdo->prepare($sql);
      $stmt->execute([$username]);
      $row = $stmt->fetch(PDO::FETCH_ASSOC);
      if ($row) {
        if (password_verify($params->password,
            $row['password'])) {
              unset($row['password']);
              $body =
              new TextStream(json_encode($row));
              $response->withBody($body);
              $code = 202;
              $info = $row;
          }
      }
    }
    return $response->withBody($body)->withStatus($code);
  }
}
```

最佳编程习惯

永远不要以纯文本的形式存储密码。在需要执行密码比对操作时，应使用 password_verify()方法，该方法无须重新生成密码散列值。

4. Authenticate 类用于封装实现 AuthenticationInterface 接口的适配器类。因此，下面的构造器将一个适配器类接收为参数，并将一个字符串用作键，验证信息就存储在该键中的$_SESSION 变量中：

```
namespace Application\Acl;
use Application\MiddleWare\ { Response, TextStream };
use Psr\Http\Message\ { RequestInterface, ResponseInterface };
class Authenticate
{
  const ERROR_AUTH = 'ERROR: invalid token';
  const DEFAULT_KEY = 'auth';
  protected $adapter;
  protected $token;
  public function __construct(
  AuthenticateInterface $adapter, $key)
  {
    $this->key = $key;
    $this->adapter = $adapter;
  }
```

5. 此外，还应该提供带安全令牌的登录表单，这有助于预防跨网站伪造请求（Cross Site Request Forgery，CSRF）攻击：

```
public function getToken()
{
  $this->token = bin2hex(random_bytes(16));
  $_SESSION['token'] = $this->token;
  return $this->token;
}
public function matchToken($token)
{
  $sessToken = $_SESSION['token'] ?? date('Ymd');
  return ($token == $sessToken);
}
public function getLoginForm($action = NULL)
```

```
{
  $action = ($action) ? 'action=" ' . $action . ' "  ' : ' ';
  $output = '<form method="post" ' . $action . '>';
  $output .= '<table><tr><th>Username</th><td>';
  $output .= '<input type="text" name="username" /></td>';
  $output .= '</tr><tr><th>Password</th><td>';
  $output .= '<input type="password" name="password" />';
  $output .= '</td></tr><tr><th> </th>';
  $output .= '<td><input type="submit" /></td>';
  $output .= '</tr></table>';
  $output .= '<input type="hidden" name="token" value="';
  $output .= $this->getToken() . '" />';
  $output .= '</form>';
  return $output;
}
```

6. 最后，在 Authenticate 类中定义 `login()` 方法，用来查明令牌是否失效。如果令牌已经失效，就返回状态码 400；否则，就调用适配器中的 `login()` 方法：

```
public function login(
RequestInterface $request) : ResponseInterface
{
  $params = json_decode($request->getBody()->getContents());
  $token = $params->token ?? FALSE;
  if (!($token && $this->matchToken($token))) {
    $code = 400;
    $body = new TextStream(self::ERROR_AUTH);
    $response = new Response($code, $body);
  } else {
      $response = $this->adapter->login($request);
  }
  if ($response->getStatusCode() >= 200
      && $response->getStatusCode() < 300) {
      $_SESSION[$this->key] =
      json_decode($response->getBody()->getContents());
  } else {
      $_SESSION[$this->key] = NULL;
  }
  return $response;
}

}
```

具体运行情况

应先参阅附录中定义的类，然后根据前面的具体步骤定义下表列出的类：

类	对应的步骤
Application\Acl\AuthenticateInterface	1
Application\Acl\DbTable	2、3
Application\Acl\Authenticate	4～6

定义调用程序 chap_09_middleware_authenticate.php，为其设置类自动加载功能，并引用适当的类：

```
<?php
session_start();
define('DB_CONFIG_FILE', __DIR__ . '/../config/db.config.php');
define('DB_TABLE', 'customer_09');
define('SESSION_KEY', 'auth');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');

use Application\Database\Connection;
use Application\Acl\ { DbTable, Authenticate };
use Application\MiddleWare\ { ServerRequest, Request, Constants,
TextStream };
```

设置执行验证操作的适配器和核心类：

```
$conn   = new Connection(include DB_CONFIG_FILE);
$dbAuth = new DbTable($conn, DB_TABLE);
$auth   = new Authenticate($dbAuth, SESSION_KEY);
```

确保初始化收到的请求，并设置发送给执行验证操作的类的请求：

```
$incoming = new ServerRequest();
$incoming->initialize();
$outbound = new Request();
```

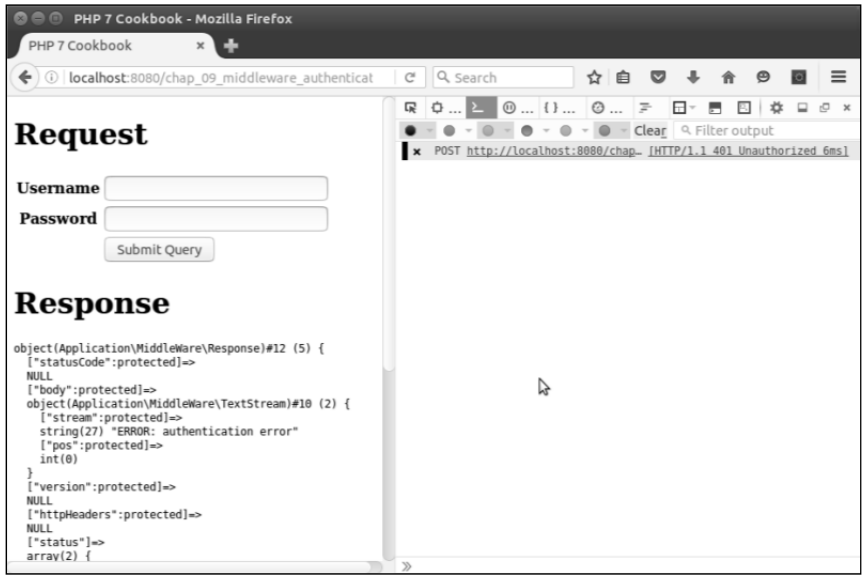
检查收到对象的方法，查明它是否为 POST。如果该方法是 POST，就向执行验证操作的类发送一个请求：

```
if ($incoming->getMethod() == Constants::METHOD_POST) {
  $body = new TextStream(json_encode(
  $incoming->getParsedBody()));
  $response = $auth->login($outbound->withBody($body));
}
$action = $incoming->getServerParams()['PHP_SELF'];
?>
```

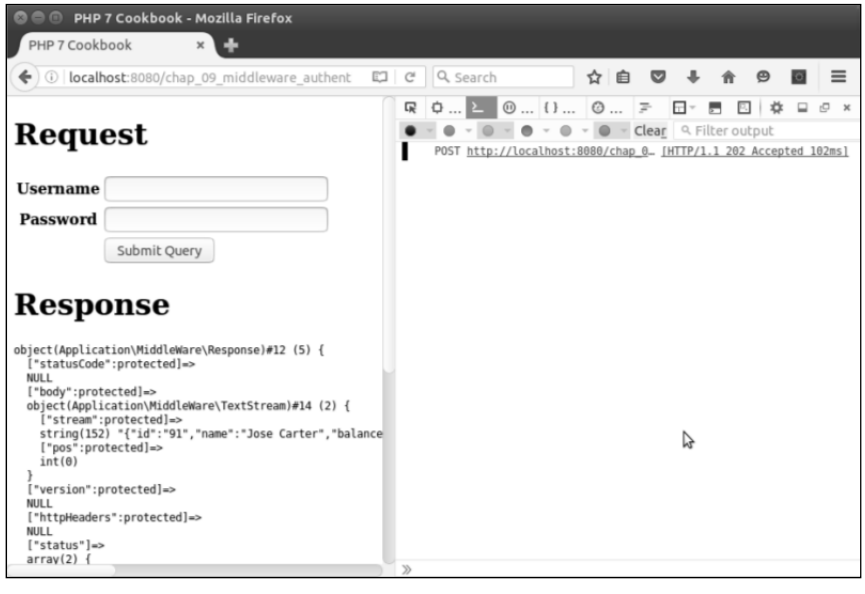
添加下面的显示逻辑：

```
<?= $auth->getLoginForm($action) ?>
```

下面是未通过验证的输出结果。注意，状态码 `401` 显示在右侧。在本例中，可在代表回应的对象中添加 `var_dump()` 函数：



下面是成功通过验证的输出结果：



补充说明

要详细了解预防 CSRF 等攻击的技巧，请参阅第 12 章。

使用中间件实现访问控制

顾名思义，中间件位于一系列函数和方法调用语句之间。因此，中间件非常适合扮演“门卫”的角色。通过使用中间件类读取访问控制列表（Access Control List，ACL），并允许或拒绝对调用序列中下一个函数或方法的访问，可以轻松实现 ACL 机制。

具体处理过程

1. 这个处理过程中最困难的部分是确定应在 ACL 中包含哪些因素。为了演示具体处理过程，可为我们的用户都分配等级（`level`）和状态（`status`）。下面是本例中划分的等级：

```
'levels' => [0, 'BEG', 'INT', 'ADV']
```

2. 状态可以指明用户处于注册过程中的哪个阶段。例如，状态 `0` 代表用户刚开始注册过程，还没有被确认；状态 `1` 代表用户的电子邮箱地址已经被确认，但是用户还没有付每月的会员费；依此类推。

3. 应定义我们想要控制的资源。本例需要控制用户对一个网站中一组页面的访问。因此，我们需要定义一组资源，这样就可以在 ACL 中为这些资源设置键：

```
'pages' => [0 => 'sorry', 'logout' => 'logout',
            'login' => 'auth',
            1 => 'page1', 2 => 'page2', 3 => 'page3',
            4 => 'page4', 5 => 'page5', 6 => 'page6',
            7 => 'page7', 8 => 'page8', 9 => 'page9']
```

4. 配置中最重要之处是根据 `level` 和 `status` 信息分配页面。下面是一个常见的配置数组模板：

```
status => ['inherits' => <key>, 'pages' => [level =>
          [pages allowed], etc.]]
```

5. 现在可以定义 `Acl` 类。如前所述，我们可使用几个类和适当的常量与属性实现访问控制：

```
namespace Application\Acl;

use InvalidArgumentException;
use Psr\Http\Message\RequestInterface;
use Application\MiddleWare\ { Constants, Response, TextStream };

class Acl
{
  const DEFAULT_STATUS  = '';
  const DEFAULT_LEVEL   = 0;
  const DEFAULT_PAGE    = 0;
  const ERROR_ACL = 'ERROR: authorization error';
  const ERROR_APP = 'ERROR: requested page not listed';
  const ERROR_DEF =
    'ERROR: must assign keys "levels", "pages" and "allowed"';
  protected $default;
  protected $levels;
  protected $pages;
  protected $allowed;
```

6. 在__construct()方法中，我们将执行分配操作的数组分解为$pages 变量$levels 变量和$allowed 变量，其中$pages 变量存储了需要控制的资源，$allowed 变量中存储了允许访问的资源。如果该数组缺失了上述 3 个成分中的任意一个，那么程序就会抛出异常：

```
public function __construct(array $assignments)
{
  $this->default = $assignments['default']
    ?? self::DEFAULT_PAGE;
  $this->pages = $assignments['pages'] ?? FALSE;
  $this->levels = $assignments['levels'] ?? FALSE;
  $this->allowed = $assignments['allowed'] ?? FALSE;
  if (!($this->pages && $this->levels && $this->allowed)) {
      throw new InvalidArgumentException(self::ERROR_DEF);
  }
}
```

7. 你可能已经注意到我们使用了继承功能。在$allowed 数组中，inherits 键可以被设置为该数组中的另一个键。如果这样做了，那么就需要将 inherits 键对应的值与另一个键对应的值合并。反向循环遍历$allowed 数组，在每次循环中合并所有 inherits 键相同的值。该方法偶尔还会仅分割应用于特定等级和状态的规则：

```
protected function mergeInherited($status, $level)
{
```

```
    $allowed = $this->allowed[$status]['pages'][$level]
      ?? array();
    for ($x = $status; $x > 0; $x--) {
      $inherits = $this->allowed[$x]['inherits'];
      if ($inherits) {
          $subArray =
            $this->allowed[$inherits]['pages'][$level]
            ?? array();
          $allowed = array_merge($allowed, $subArray);
      }
    }
    return $allowed;
}
```

8. 在处理验证操作时，可初始化几个变量，然后通过原始的请求 URI 提取被请求的页面。如果页面参数不存在，就将状态码设置为 400：

```
public function isAuthorized(RequestInterface $request)
{
  $code = 401;     // 代表未验证
  $text['page'] = $this->pages[$this->default];
  $text['authorized'] = FALSE;
  $page = $request->getUri()->getQueryParams()['page']
    ?? FALSE;
  if ($page === FALSE) {
      $code = 400; // 代表错误的请求
```

9. 否则，就对请求的主体内容进行解码，并获取 status 和 level 信息。现在可调用 mergeInherited()方法，该方法会返回一组可访问的页面（具体是否可访问是根据该用户的 status 和 level 信息判定的）：

```
} else {
    $params = json_decode(
      $request->getBody()->getContents());
    $status = $params->status ?? self::DEFAULT_LEVEL;
    $level = $params->level ?? '*';
    $allowed = $this->mergeInherited($status, $level);
```

10. 如果被请求的页面存储在$allowed 数组中，就可以将状态码设置为 200，并返回已验证设置，以及与被请求的页面代码对应的页面：

```
if (in_array($page, $allowed)) {
    $code = 200;    // 代表允许访问
```

```
        $text['authorized'] = TRUE;
        $text['page'] = $this->pages[$page];
    } else {
        $code = 401; }
    }
```

11. 然后返回 JSON 编码形式的回应，处理工作就完成了：

```
$body = new TextStream(json_encode($text));
return (new Response())->withStatus($code)
->withBody($body);
}

}
```

具体运行情况

使用前面介绍的代码定义Application\Acl\Acl类。切换到/path/to/ source/for/this/chapter 文件夹，并创建两个目录：public 和 pages。在 pages 目录中创建一系列 PHP 文件（用于生成页面），如 page1.php、page2.php 等。下面是这些页面的示例：

```
<?php // page 1 ?>
<h1>Page 1</h1>
<hr>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. etc.</p>
```

还可以定义 menu.php 页面（菜单页面），该页面也会被包含到输出结果中：

```
<?php // menu ?>
<a href="?page=1">Page 1</a>
<a href="?page=2">Page 2</a>
<a href="?page=3">Page 3</a>
// 依此类推
```

logout.php 页面应该具有切断会话的功能：

```
<?php
  $_SESSION['info'] = FALSE;
  session_destroy();
?>
<a href="/">BACK</a>
```

auth.php 页面应显示登录界面（请参阅前面介绍的内容）：

```
<?= $auth->getLoginForm($action) ?>
```

这样就可以创建配置文件，该文件会根据用户的等级和状态信息来限定用户访问哪些网页。为了了解具体处理过程，我们将该配置文件命名为 chap_09_middleware_acl_config.php，并使之返回如下所示的数组：

```
<?php
$min = [0, 'logout'];
return [
  'default' => 0, // 默认页面
  'levels' => [0, 'BEG', 'INT', 'ADV'],
  'pages' => [0 => 'sorry',
  'logout' => 'logout',
  'login' => 'auth',
               1 => 'page1', 2 => 'page2', 3 => 'page3',
               4 => 'page4', 5 => 'page5', 6 => 'page6',
               7 => 'page7', 8 => 'page8', 9 => 'page9'],
'allowed' => [
             0 => ['inherits' => FALSE,
                 'pages' => [ '*' => $min, 'BEG' => $min,
                 'INT' => $min,'ADV' => $min]],
             1 => ['inherits' => FALSE,
                 'pages' => ['*' => ['logout'],
                 'BEG' => [1, 'logout'],
                 'INT' => [1,2, 'logout'],
                 'ADV' => [1,2,3, 'logout']]],
             2 => ['inherits' => 1,
                 'pages' => ['BEG' => [4],
                 'INT' => [4,5],
                 'ADV' => [4,5,6]]],
             3 => ['inherits' => 2,
                 'pages' => ['BEG' => [7],
                 'INT' => [7,8],
                 'ADV' => [7,8,9]]]
    ]
];
```

在 public 文件夹中创建 index.php 文件，该文件可设置类自动加载功能，而且最终会调用 Authenticate 和 Acl 类。像之前做过的练习一样，定义配置文件、设置类自动加载功能，并引用指定的类。而且不要忘记建立会话：

```
<?php
session_start();
session_regenerate_id();
```

```
define('DB_CONFIG_FILE', __DIR__ . '/../../config/db.config.php');
define('DB_TABLE', 'customer_09');
define('PAGE_DIR', __DIR__ . '/../pages');
define('SESSION_KEY', 'auth');
require __DIR__ . '/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/../..');

use Application\Database\Connection;
use Application\Acl\ { Authenticate, Acl };
use Application\MiddleWare\ { ServerRequest, Request, Constants,
  TextStream };
```

最佳编程习惯

下面介绍一种能够为会话提供防护的最佳编程习惯。为会话提供防护的一种简单方式是使用 session_regenerate_id()函数，该函数能够使当前的 PHP 会话标识符作废，并生成新的 PHP 会话标识符。因此，在攻击者想要通过非法途经获取会话标识符时，这种处理方式可以将攻击者实施攻击的时间窗口压缩至最小。

现在可以添加 ACL 配置，并创建 Authenticate 和 Acl 实例：

```
$config = require __DIR__ . '/../chap_09_middleware_acl_config.php';
$acl    = new Acl($config);
$conn   = new Connection(include DB_CONFIG_FILE);
$dbAuth = new DbTable($conn, DB_TABLE);
$auth   = new Authenticate($dbAuth, SESSION_KEY);
```

定义代表收到的和发出的请求的实例：

```
$incoming = new ServerRequest();
$incoming->initialize();
$outbound = new Request();
```

如果收到的请求的方法是 post，就通过调用 login()方法执行验证操作：

```
if (strtolower($incoming->getMethod()) == Constants::METHOD_POST) {
    $body = new TextStream(json_encode(
    $incoming->getParsedBody()));
    $response = $auth->login($outbound->withBody($body));
}
```

如果为验证操作定义的会话键被赋值了，就意味着用户已经成功通过验证。可编写

一个匿名函数，在其中包含登录界面，稍后调用该函数：

```
$info = $_SESSION[SESSION_KEY] ?? FALSE;
if (!$info) {
    $execute = function () use ($auth) {
      include PAGE_DIR . '/auth.php';
   };
```

在会话键没有被赋值的情况下，应继续执行 ACL 检查。但先需要通过原始查询操作查明用户想要访问哪些页面：

```
} else {
    $query = $incoming->getServerParams()['QUERY_STRING'] ?? '';
```

可重新编写$outbound 请求，以便在其中包含下列信息：

```
$outbound->withBody(new TextStream(json_encode($info)));
$outbound->getUri()->withQuery($query);
```

现在可以检查验证操作，将发出的请求用作参数：

```
$response = $acl->isAuthorized($outbound);
```

检查对 authorized 参数返回的回应，并编写一个匿名函数，使该函数在通过验证的情况下返回 page 参数，在没有通过验证的情况下返回 sorry 页面：

```
$params = json_decode($response->getBody()->getContents());
$isAllowed = $params->authorized ?? FALSE;
if ($isAllowed) {
    $execute = function () use ($response, $params) {
      include PAGE_DIR .'/' . $params->page . '.php';
      echo '<pre>', var_dump($response), '</pre>';
      echo '<pre>', var_dump($_SESSION[SESSION_KEY]);
      echo '</pre>';
   };
} else {
     $execute = function () use ($response) {
       include PAGE_DIR .'/sorry.php';
       echo '<pre>', var_dump($response), '</pre>';
       echo '<pre>', var_dump($_SESSION[SESSION_KEY]);
       echo '</pre>';
     };
}
}
```

设置表单操作并将匿名函数封装到 HTML 代码中：

```
$action = $incoming->getServerParams()['PHP_SELF'];
```

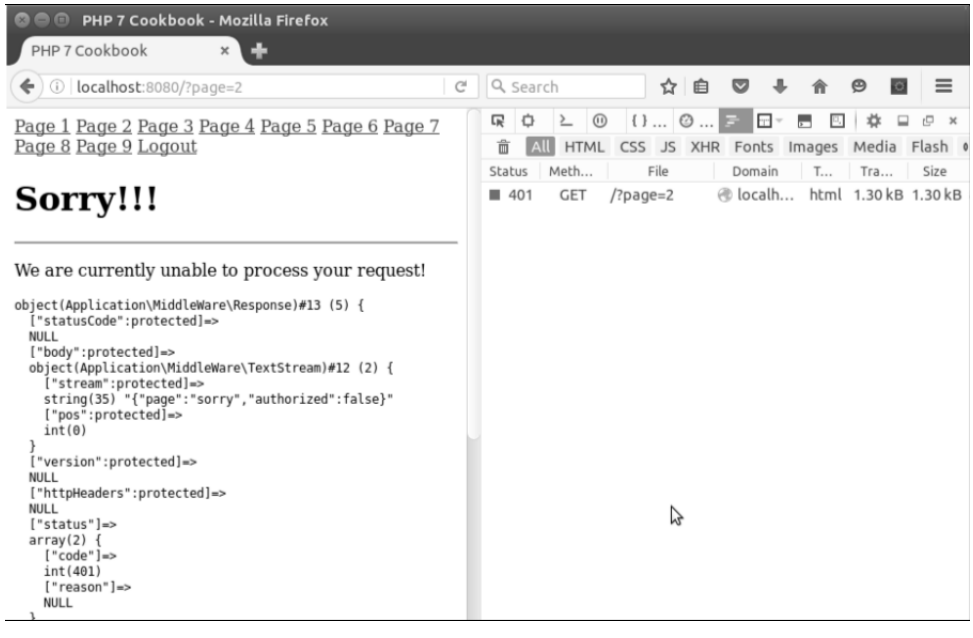
```
?>
<!DOCTYPE html>
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type" content="text/html;charset=utf-8" />
</head>
<body>
  <?php $execute(); ?>
</body>
</html>
```

要测试这个程序，可使用 PHP 内置的 Web 服务器，但是需要使用-t 参数指明网站页面文档的根目录为 public：

```
cd /path/to/source/for/this/chapter
php -S localhost:8080 -t public
```

使用浏览器访问 http://localhost:8080/。

如果你要访问某个页面，都会被导向登录页面来开始访问过程。根据配置，status = 1 且 level = BEG 的用户只能访问 page 1 和登出页面。如果该用户尝试访问 page 2 页面，就会看到下面的输出结果：



补充说明

在用户登录后，本例将超级全局变量$_SESSION 用作用户身份验证的唯一手段。要详细了解保护 PHP 会话的内容，请参阅第 12 章。

使用缓存提高性能

对于需要很长时间才能生成的结果，缓存软件设计模式可以将其缓存起来。这种设计模式会使用冗长的查看脚本或复杂数据库查询操作的形式。当然，如果你想要提高网站浏览者的用户体验，那么存储数据的目的地就需要具有较高的性能。因为不同的装置有不同的潜在存储目标，所以缓存机制也会趋向于适配器模式。典型的潜在存储目的地包括内存、数据库和文件系统。

具体处理过程

1. 因为本章前面介绍了多个示例，所以有许多相同的常量。因此，可定义一个专门定义常量的类 Application\Cache\Constants：

```
<?php
namespace Application\Cache;

class Constants
{
  const DEFAULT_GROUP = 'default';
  const DEFAULT_PREFIX = 'CACHE_';
  const DEFAULT_SUFFIX = '.cache';
  const ERROR_GET = 'ERROR: unable to retrieve from cache';
  // 为了节省篇幅，此处没有列出所有常量
}
```

2. 为了遵循适配器设计模式，可定义一个接口：

```
namespace Application\Cache;
interface CacheAdapterInterface
{
  public function hasKey($key);
  public function getFromCache($key, $group);
  public function saveToCache($key, $data, $group);
```

```
  public function removeByKey($key);
  public function removeByGroup($group);
}
```

3. 下面定义第一个缓存适配器，本例使用 MySQL 数据库。需要定义用于存储字段名称和准备语句的属性：

```
namespace Application\Cache;
use PDO;
use Application\Database\Connection;
class Database implements CacheAdapterInterface
{
  protected $sql;
  protected $connection;
  protected $table;
  protected $dataColumnName;
  protected $keyColumnName;
  protected $groupColumnName;
  protected $statementHasKey       = NULL;
  protected $statementGetFromCache = NULL;
  protected $statementSaveToCache  = NULL;
  protected $statementRemoveByKey  = NULL;
  protected $statementRemoveByGroup= NULL;
```

4. 通过下面的构造器，提供主要的字段名称、`Application\Database\Connection` 实例，以及用于缓存数据的数据库表的名称：

```
public function__construct(Connection $connection,
  $table,
  $idColumnName,
  $keyColumnName,
  $dataColumnName,
  $groupColumnName = Constants::DEFAULT_GROUP)
  {
   $this->connection = $connection;
   $this->setTable($table);
   $this->setIdColumnName($idColumnName);
   $this->setDataColumnName($dataColumnName);
   $this->setKeyColumnName($keyColumnName);
   $this->setGroupColumnName($groupColumnName);
}
```

5. 下面几种方法用于生成准备语句，且会在访问数据库时被调用。此处没有列出所

有方法，希望下面的代码能够让你有一个大概的了解：

```
public function prepareHasKey()
{
  $sql = 'SELECT `' . $this->idColumnName . '` '
  . 'FROM `' . $this->table . '` '
  . 'WHERE `' . $this->keyColumnName . '` = :key ';
  $this->sql[__METHOD__] = $sql;
  $this->statementHasKey =
  $this->connection->pdo->prepare($sql);
}
public function prepareGetFromCache()
{
  $sql = 'SELECT `' . $this->dataColumnName . '` '
  . 'FROM `' . $this->table . '` '
  . 'WHERE `' . $this->keyColumnName . '` = :key '
  . 'AND `' . $this->groupColumnName . '` = :group';
  $this->sql[__METHOD__] = $sql;
  $this->statementGetFromCache =
  $this->connection->pdo->prepare($sql);
}
```

6. 定义一个方法，使用该方法确定是否将数据与现存的键对应起来：

```
public function hasKey($key)
{
  $result = 0;
  try {
       if (!$this->statementHasKey) $this->prepareHasKey();
         $this->statementHasKey->execute(['key' => $key]);
  } catch (Throwable $e) {
      error_log(__METHOD__ . ':' . $e->getMessage());
      throw new Exception(Constants::ERROR_REMOVE_KEY);
  }
  return (int) $this->statementHasKey
  ->fetch(PDO::FETCH_ASSOC)[$this->idColumnName];
}
```

7. 核心方法负责从缓存读取数据和向缓存中写入数据。使用下面的方法可以从缓存读取数据，我们要做的只是执行准备语句，该语句含有 SQL 语句 `SELECT` 和 `WHERE` 子句，`WHERE` 子句会将键和分组合并到一起：

```
public function getFromCache(
```

```
$key, $group = Constants::DEFAULT_GROUP)
{
  try {
       if (!$this->statementGetFromCache)
           $this->prepareGetFromCache();
           $this->statementGetFromCache->execute(
             ['key' => $key, 'group' => $group]);
           while ($row = $this->statementGetFromCache
             ->fetch(PDO::FETCH_ASSOC)) {
             if ($row && count($row)) {
               yield unserialize($row[$this->dataColumnName]);
             }
           }
  } catch (Throwable $e) {
     error_log(__METHOD__ . ':' . $e->getMessage());
     throw new Exception(Constants::ERROR_GET);
  }
}
```

8. 当向缓存中写入数据时，应先查明与缓存键对应的条目是否已经存在。如果与缓存键对应的条目已经存在，就应该执行 UPDATE（更新）操作；否则，应执行 INSERT（插入）操作：

```
public function saveToCache($key, $data,
                    $group = Constants::DEFAULT_GROUP)
{
  $id = $this->hasKey($key);
  $result = 0;
  try {
       if ($id) {
           if (!$this->statementUpdateCache)
               $this->prepareUpdateCache();
               $result = $this->statementUpdateCache
               ->execute(['key' => $key,
               'data' => serialize($data),
               'group' => $group,
               'id' => $id]);
    } else {
        if (!$this->statementSaveToCache)
        $this->prepareSaveToCache();
        $result = $this->statementSaveToCache
```

```
            ->execute(['key' => $key,
            'data' => serialize($data),
            'group' => $group]);
        }
      } catch (Throwable $e) {
          error_log(__METHOD__ . ':' . $e->getMessage());
          throw new Exception(Constants::ERROR_SAVE);
      }
      return $result;
    }
```

9. 定义两个方法，使用它们根据键或分组从缓存中删除数据。根据分组执行删除操作的处理方式很方便，可被用于需删除条目数量很多的情况：

```
    public function removeByKey($key)
    {
      $result = 0;
      try {
           if (!$this->statementRemoveByKey)
           $this->prepareRemoveByKey();
           $result = $this->statementRemoveByKey->execute(
               ['key' => $key]);
      } catch (Throwable $e) {
          error_log(__METHOD__ . ':' . $e->getMessage());
          throw new Exception(Constants::ERROR_REMOVE_KEY);
      }
      return $result;
    }

    public function removeByGroup($group)
    {
      $result = 0;
      try {
           if (!$this->statementRemoveByGroup)
               $this->prepareRemoveByGroup();
               $result = $this->statementRemoveByGroup->execute(
                 ['group' => $group]);
      } catch (Throwable $e) {
          error_log(__METHOD__ . ':' . $e->getMessage());
          throw new Exception(Constants::ERROR_REMOVE_GROUP);
      }
      return $result;
```

```
}
```

10. 为每个属性定义读取器和设置器。为了节省篇幅，此处没有列出全部代码：

```
public function setTable($name)
{
  $this->table = $name;
}
public function getTable()
{
  return $this->table;
}
// 依此类推
}
```

11. 在文件系统缓存适配器中定义的方法与前面步骤中定义的方法相同。注意，使用 `md5()` 函数的目的不是提高安全性，而且为了能够通过键快速地生成文本字符串：

```
namespace Application\Cache;
use RecursiveIteratorIterator;
use RecursiveDirectoryIterator;
class File implements CacheAdapterInterface
{
  protected $dir;
  protected $prefix;
  protected $suffix;
  public function __construct(
    $dir, $prefix = NULL, $suffix = NULL)
  {
    if (!file_exists($dir)) {
        error_log(__METHOD__ . ':' . Constants::ERROR_DIR_NOT);
        throw new Exception(Constants::ERROR_DIR_NOT);
    }
    $this->dir = $dir;
    $this->prefix = $prefix ?? Constants::DEFAULT_PREFIX;
    $this->suffix = $suffix ?? Constants::DEFAULT_SUFFIX;
  }

  public function hasKey($key)
  {
    $action = function ($name, $md5Key, &$item) {
      if (strpos($name, $md5Key) !== FALSE) {
        $item ++;
```

```
    }
  };

  return $this->findKey($key, $action);
}

public function getFromCache($key,
                      $group = Constants::DEFAULT_GROUP)
{
  $fn = $this->dir . '/' . $group . '/'
  . $this->prefix . md5($key) . $this->suffix;
  if (file_exists($fn)) {
      foreach (file($fn) as $line) { yield $line; }
  } else {
      return array();
  }
}

public function saveToCache(
  $key, $data, $group = Constants::DEFAULT_GROUP)
{
  $baseDir = $this->dir . '/' . $group;
  if (!file_exists($baseDir)) mkdir($baseDir);
  $fn = $baseDir . '/' . $this->prefix . md5($key)
  . $this->suffix;
  return file_put_contents($fn, json_encode($data));
}

protected function findKey($key, callable $action)
{
  $md5Key = md5($key);
  $iterator = new RecursiveIteratorIterator(
    new RecursiveDirectoryIterator($this->dir),
    RecursiveIteratorIterator::SELF_FIRST);
    $item = 0;
  foreach ($iterator as $name => $obj) {
    $action($name, $md5Key, $item);
  }
  return $item;
}
public function removeByKey($key)
```

```
    {
      $action = function ($name, $md5Key, &$item) {
        if (strpos($name, $md5Key) !== FALSE) {
          unlink($name);
          $item++;
        }
      };
      return $this->findKey($key, $action);
    }

    public function removeByGroup($group)
    {
      $removed = 0;
      $baseDir = $this->dir . '/' . $group;
      $pattern = $baseDir . '/' . $this->prefix . '*'
      . $this->suffix;
      foreach (glob($pattern) as $file) {
        unlink($file);
        $removed++;
      }
      return $removed;
    }
}
```

12. 现在可以编写核心的缓存机制。在构造器中，我们将实现了 CacheAdapter Interface 接口的类接收为参数：

```
namespace Application\Cache;
use Psr\Http\Message\RequestInterface;
use Application\MiddleWare\ { Request, Response, TextStream };
class Core
{
  public function __construct(CacheAdapterInterface $adapter)
  {
    $this->adapter = $adapter;
  }
```

13. 编写一组封装器方法来调用适配器中与它们同名的方法，这些封装器方法会将 Psr\Http\Message\RequestInterface 实例用作参数，然后将 Psr\Http\ Message\ ResponseInterface 实例返回为参数。可以从简单的 hasKey() 方法开始编写。请注意从请求参数中提取键（存储在 key 变量中）的方式：

```
public function hasKey(RequestInterface $request)
{
  $key = $request->getUri()->getQueryParams()['key'] ?? '';
  $result = $this->adapter->hasKey($key);
}
```

14. 要在缓存中检索信息，需要从请求对象中提取键和分组参数，然后调用适配器中的同名方法。如果没有获得结果，就应该将状态码设置为 204，代表获取数据的请求成功获得回应，但是请求获取的数据不存在。否则，应将状态码设置为 200（代表成功获得请求的数据），并循环遍历获得的结果。得到的所有数据都应该存储到代表回应的对象中，代表回应的对象会被返回：

```
public function getFromCache(RequestInterface $request)
{
  $text = array();
  $key = $request->getUri()->getQueryParams()['key'] ?? '';
  $group = $request->getUri()->getQueryParams()['group']
    ?? Constants::DEFAULT_GROUP;
  $results = $this->adapter->getFromCache($key, $group);
  if (!$results) {
      $code = 204;
  } else {
      $code = 200;
      foreach ($results as $line) $text[] = $line;
  }
    if (!$text || count($text) == 0) $code = 204;
    $body = new TextStream(json_encode($text));
    return (new Response())->withStatus($code)
                           ->withBody($body);
  }
```

15. 奇妙的是，除了得到的结果为数字（代表受影响的记录数量）或布尔值 ，向缓存写入数据的操作与从缓存读取数据的操作几乎相同：

```
public function saveToCache(RequestInterface $request)
{
  $text = array();
  $key = $request->getUri()->getQueryParams()['key'] ?? '';
  $group = $request->getUri()->getQueryParams()['group']
    ?? Constants::DEFAULT_GROUP;
  $data = $request->getBody()->getContents();
  $results = $this->adapter->saveToCache($key, $data, $group);
```

```
  if (!$results) {
      $code = 204;
  } else {
      $code = 200;
      $text[] = $results;
  }
     $body = new TextStream(json_encode($text));
     return (new Response())->withStatus($code)
                            ->withBody($body);
  }
```

16. 执行删除操作的方法相互之间也很相似：

```
public function removeByKey(RequestInterface $request)
{
  $text = array();
  $key = $request->getUri()->getQueryParams()['key'] ?? '';
  $results = $this->adapter->removeByKey($key);
  if (!$results) {
      $code = 204;
  } else {
      $code = 200;
      $text[] = $results;
  }
  $body = new TextStream(json_encode($text));
  return (new Response())->withStatus($code)
                         ->withBody($body);
  }

  public function removeByGroup(RequestInterface $request)
  {
    $text = array();
    $group = $request->getUri()->getQueryParams()['group']
      ?? Constants::DEFAULT_GROUP;
    $results = $this->adapter->removeByGroup($group);
    if (!$results) {
        $code = 204;
    } else {
        $code = 200;
        $text[] = $results;
    }
    $body = new TextStream(json_encode($text));
```

```
        return (new Response())->withStatus($code)
                               ->withBody($body);
        }
    } // 这是 Core 类的结束花括号
```

具体运行情况

为了了解 Acl 类的用法，可使用前面介绍的代码定义下列类：

类	对应的步骤
Application\Cache\Constants	1
Application\Cache\CacheAdapterInterface	2
Application\Cache\Database	3 ~ 10
Application\Cache\File	11
Application\Cache\Core	12 ~ 16

定义测试程序 chap_09_middleware_cache_db.php。和以前一样，在这个程序中需要为必要文件定义常量，设置类自动加载功能，引用合适的类，以及编写用于生成素数的函数（这个问题可能有点难度，不过无须担心，我们一起来解决它）：

```
<?php
define('DB_CONFIG_FILE', __DIR__ . '/../config/db.config.php');
define('DB_TABLE', 'cache');
define('CACHE_DIR', __DIR__ . '/cache');
define('MAX_NUM', 100000);
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Connection;
use Application\Cache\{ Constants, Core, Database, File };
use Application\MiddleWare\ { Request, TextStream };
```

素数生成器是一种需要较长运行时间的函数。数字 1、2 和 3 是指定的素数。可使用 PHP 7 中的 yield from 语法生成这前 3 个素数，然后可直接跳到 5，并继续生成素数，直到达到请求的最大值：

```
function generatePrimes($max)
{
  yield from [1,2,3];
  for ($x = 5; $x < $max; $x++)
  {
    if($x & 1) {
        $prime = TRUE;
```

```
            for($i = 3; $i < $x; $i++) {
                if(($x % $i) === 0) {
                    $prime = FALSE;
                    break;
                }
            }
            if ($prime) yield $x;
        }
      }
    }
```

然后可设置数据库缓存适配器实例，该实例会被用作 Core 实例的参数：

```
$conn   = new Connection(include DB_CONFIG_FILE);
$dbCache= new Database(
  $conn, DB_TABLE, 'id', 'key', 'data', 'group');
$core   = new Core($dbCache);
```

如果你想要使用文件缓存适配器代替数据库缓存适配器，可使用下面的代码：

```
$fileCache= new File(CACHE_DIR);
$core   = new Core($fileCache);
```

如果你想要从缓存中删除数据，可使用下面的方式：

```
$uriString = '/?group=' . Constants::DEFAULT_GROUP;
$cacheRequest = new Request($uriString, 'get');
$response = $core->removeByGroup($cacheRequest);
```

可以使用 `time()` 和 `microtime()` 函数来查明在使用和不使用缓存的情况下，这个脚本分别会运行多长时间：

```
$start = time() + microtime(TRUE);
echo "\nTime: " . $start;
```

生成一个缓存请求。状态码 200 表明可以从缓存中获取一个素数列表：

```
$uriString = '/?key=Test1';
$cacheRequest = new Request($uriString, 'get');
$response = $core->getFromCache($cacheRequest);
$status = $response->getStatusCode();
if ($status == 200) {
    $primes = json_decode($response->getBody()->getContents());
```

其他状态码表明无法从缓存中获取素数列表，这意味着你需要自己生成素数，并将它们存储到缓存中：

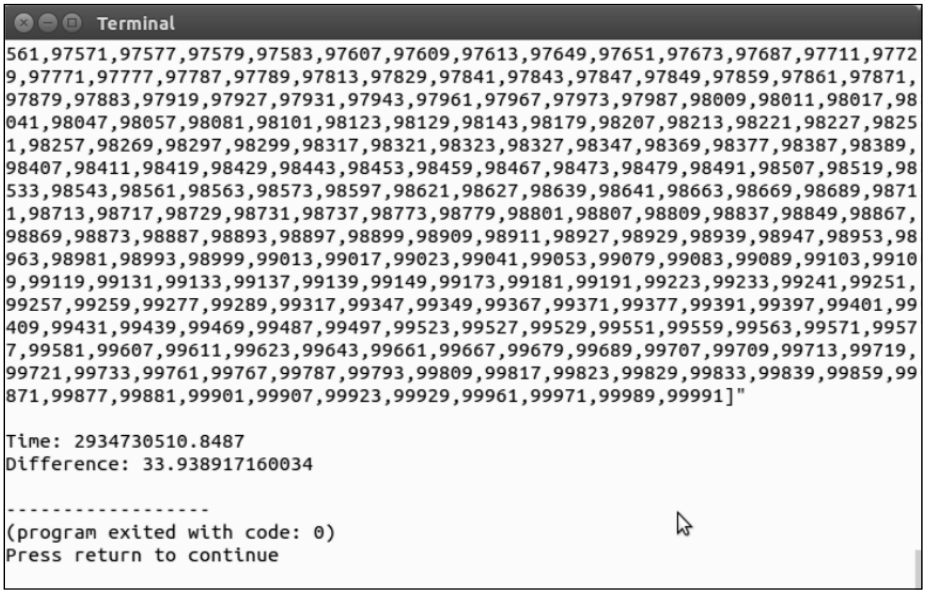
```
} else {
    $primes = array();
```

```
        foreach (generatePrimes(MAX_NUM) as $num) {
            $primes[] = $num;
        }
        $body = new TextStream(json_encode($primes));
        $response = $core->saveToCache(
        $cacheRequest->withBody($body));
    }
```

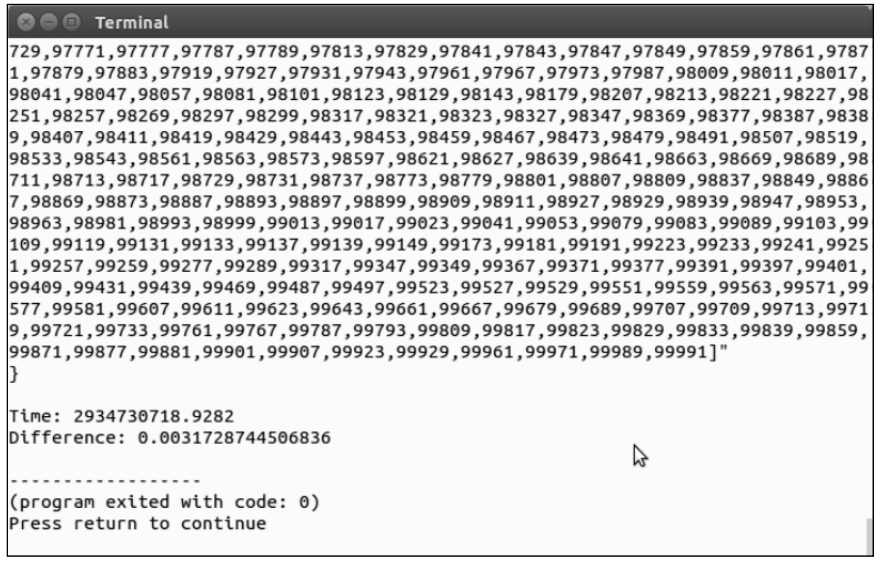
这样就可以查明程序停止运行的时间、计算程序运行时间的差别，以及查看新建的素数列表：

```
$time = time() + microtime(TRUE);
$diff = $time - $start;
echo "\nTime: $time";
echo "\nDifference: $diff";
var_dump($primes);
```

下面是将素数列表存储到缓存前的输出结果：



再次运行这段程序，但这次从缓存中提取素数列表：



尽管我们编写的素数生成器的性能不是很高，而且是在笔记本电脑上进行这个实验的，但使用了缓存后仍然能够节省 30 多秒的时间。

补充说明

使用 Alternate PHP Cache（APC）扩展可以创建另一种缓存适配器。该扩展含有 `apc_exists()`、`apc_store()`、`apc_fetch()`和 `apc_clear_cache()`等函数。这些函数非常适合用于编写我们自己的 `hasKey()`、`saveToCache()`、`getFromCache()`和 `removeBy*()`方法。

扩展

你可能希望略微改动前面介绍的遵循 PSR-6 标准（直接面向缓存处理的推荐标准）的缓存适配器类。因为 PSR-7 标准中没有对缓存处理的描述，所以我们决定遵循 PSR-6 标准。要详细了解 PSR-6 标准，请浏览 `http://www.php-fig.org/psr/psr-6/`。

实现路由功能

路由是指一种处理过程，用于接收用户友好的 URL、解析 URL，并决定将 URL 的

各个组成部分分配给哪些类和方法。这种实现方式不仅能够使你的 URL 拥有搜索引擎优化（Search Engine Optimization，SEO）的特质，还能够创建规则和合并正则表达式模式，从而提取参数的值。

具体处理过程

1. 实现路由功能的最流行的方式可能是利用支持 URL 重写功能的 Web 服务器的方式。使用 `mod_rewrite` 模块配置的 Apache Web 服务器就是一个实际的例子。应定义重写规则，以便能够无损地传输图片文件、CSS 和 JavaScript 请求。否则，这些请求会被路由方法漏掉。

2. 另一种实现路由功能的方式很简单，只需使 Web 服务器的虚拟主机定义指向专用的路由脚本，该脚本会调用执行路由操作的类，做路由决策，以及以适当的方式重定向。

3. 应先编写用于定义路由配置的代码。可创建一个数组，使用该数组中的键指向与 URI 路径对应的正则表达式和某种形式的操作。下面的代码展示了这种配置的示例。本例定义了 3 种路由：`home`、`page` 和默认路由。默认路由应该最后定义，因为只有在前两种路由都不符合条件的情况下，默认路由才会被采用。应使用匿名函数的形式实现操作，该匿名函数会在符合路由条件时被调用：

```
$config = [
  'home' => [
    'uri' => '!^/$!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/page0.php'; }
  ],
  'page' => [
    'uri' => '!^/(page)/(\d+)$!',
      'exec' => function ($matches) {
        include PAGE_DIR . '/page' . $matches[2] . '.php'; }
  ],
  Router::DEFAULT_MATCH => [
    'uri' => '!.*!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/sorry.php'; }
  ],
];
```

4. 定义 `Router` 类。定义在检查和匹配路由时使用的常量和属性：

```
namespace Application\Routing;
```

```
use InvalidArgumentException;
use Psr\Http\Message\ServerRequestInterface;
class Router
{
  const DEFAULT_MATCH  = 'default';
  const ERROR_NO_DEF   = 'ERROR: must supply a default match';
  protected $request;
  protected $requestUri;
  protected $uriParts;
  protected $docRoot;
  protected $config;
  protected $routeMatch;
```

5. 下面的构造器方法接收具有兼容性的 ServerRequestInterface 类、文档根目录的路径和前面介绍的配置文件。注意，如果没有为该程序提供默认配置，那么该程序就会抛出异常：

```
public function __construct(ServerRequestInterface $request,
  $docRoot, $config)
{
  $this->config = $config;
  $this->docRoot = $docRoot;
  $this->request = $request;
  $this->requestUri =
    $request->getServerParams()['REQUEST_URI'];
  $this->uriParts = explode('/', $this->requestUri);
  if (!isset($config[self::DEFAULT_MATCH])) {
      throw new InvalidArgumentException(
        self::ERROR_NO_DEF);
  }
}
```

6. 编写一组读取器，以便获取原始请求、文档根目录和最终的路由匹配条件：

```
public function getRequest()
{
  return $this->request;
}
public function getDocRoot()
{
  return $this->docRoot;
}
public function getRouteMatch()
```

```
{
  return $this->routeMatch;
}
```

7. 编写 isFileOrDir() 方法，使用该方法查明请求的类型是否为 CSS、JavaScript 脚本或图片：

```
public function isFileOrDir()
{
  $fn = $this->docRoot . '/' . $this->requestUri;
  $fn = str_replace('//', '/', $fn);
  if (file_exists($fn)) {
      return $fn;
  } else {
      return '';
  }
}
```

8. 定义 match() 方法，应使该方法循环遍历配置数组，并使用 uri 参数调用 preg_match() 函数。如果路由符合条件，那么配置数组中的键和通过 preg_match() 函数赋值的$matches 数组就会被存储到$routeMatch 变量中，而且回调函数会被返回。如果路由不符合条件，那么默认的回调函数就会被返回：

```
public function match()
{
  foreach ($this->config as $key => $route) {
    if (preg_match($route['uri'],
        $this->requestUri, $matches)) {
        $this->routeMatch['key'] = $key;
        $this->routeMatch['match'] = $matches;
        return $route['exec'];
    }
  }
  return $this->config[self::DEFAULT_MATCH]['exec'];
}
}
```

具体运行情况

先切换到/path/to/source/for/this/chapter 文件夹，并创建 routing 目录。然后定义 index.php 文件，为该文件设置类自动加载功能，并引用合适的类。定

义 PAGE_DIR 常量，使用它指向前面步骤创建的 pages 目录：

```
<?php
define('DOC_ROOT', __DIR__);
define('PAGE_DIR', DOC_ROOT . '/../pages');

require_once __DIR__ . '/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/../..');
use Application\MiddleWare\ServerRequest;
use Application\Routing\Router;
```

添加前面步骤 3 介绍的配置数组。注意，可在格式语句的末尾添加(/)?，将(/)?用作可选的末尾斜杠。而对于主页路由，你也可以提供两个选项：/或/home：

```
$config = [
  'home' => [
    'uri' => '!^(/|/home)$!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/page0.php'; }
  ],
  'page' => [
    'uri' => '!^/(page)/(\d+)(/)?$!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/page' . $matches[2] . '.php'; }
  ],
  Router::DEFAULT_MATCH => [
    'uri' => '!.*!',
    'exec' => function ($matches) {
      include PAGE_DIR . '/sorry.php'; }
  ],
];
```

这样就可以定义 Router 实例（代表路由器），将已初始化的 ServerRequest 实例（代表服务器收到的请求）用作创建 Router 实例的第一个参数：

```
$router = new Router((new ServerRequest())
  ->initialize(), DOC_ROOT, $config);
$execute = $router->match();
$params  = $router->getRouteMatch()['match'];
```

还需要查明请求获取的信息是文件还是目录，以及路由匹配条件是否为/：

```
if ($fn = $router->isFileOrDir()
     && $router->getRequest()->getUri()->getPath() != '/') {
     return FALSE;
} else {
```

```
    include DOC_ROOT . '/main.php';
}
```

定义 main.php 文件：

```
<?php // 使用中间件实现路由功能 ?>
<!DOCTYPE html>
<head>
  <title>PHP 7 Cookbook</title>
  <meta http-equiv="content-type"
  content="text/html;charset=utf-8" />
</head>
<body>
   <?php include PAGE_DIR . '/route_menu.php'; ?>
   <?php $execute($params); ?>
</body>
</html>
```

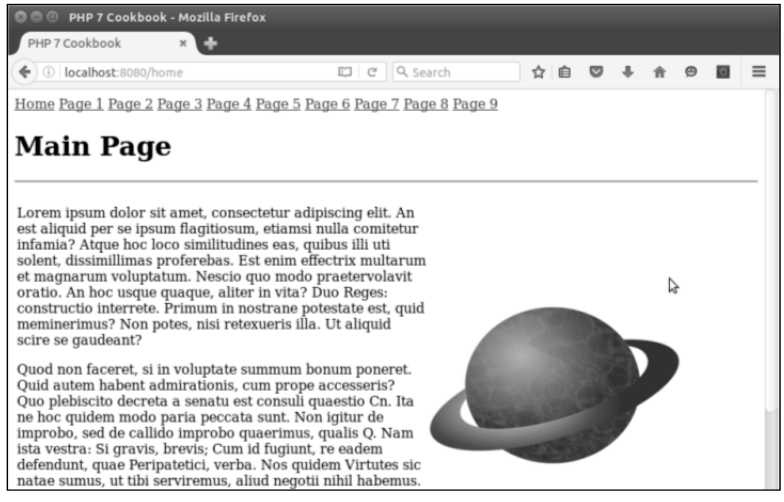
最后，还应改进用户友好的路由菜单：

```
<?php // 用于进行路由的菜单 ?>
<a href="/home">Home</a>
<a href="/page/1">Page 1</a>
<a href="/page/2">Page 2</a>
<a href="/page/3">Page 3</a>
<!-- 依此类推 -->
```

要使用 Apache 服务器测试这些配置，可使虚拟主机定义指向/path/to/source/for/this/chapter/routing 目录。此外，还应定义.htaccess 文件，以便重定向非文件请求、目录或指向 index.php 文件的链接。你也可以使用 PHP 内置的 Web 服务器。在终端窗口或命令行界面中输入下面的命令：

```
cd /path/to/source/for/this/chapter/routing
php -S localhost:8080
```

在浏览器的网址栏中输入 http://localhost:8080/home 并按回车键，可以得到下面的显示效果：



扩展

要详细了解使用 **NGINX** Web 服务器是怎样重写代码的，请浏览 `http://nginx.org/en/docs/http/ngx_http_rewrite_module.html`。提供成熟的路由功能的 PHP 软件库很多，它们所提供的功能比本节介绍的简单路由器的功能多得多，这些软件库包括：Altorouter（`http://altorouter.com/`）、TreeRoute（`https://github.com/baryshev/TreeRoute`）、FastRoute（`https://github.com/nikic/FastRoute`）和 Aura.Router.（`https://github.com/auraphp/Aura.Router`）。此外，大多数框架（如 Zend Framework 2 和 CodeIgniter）都拥有自己的路由功能。

实现框架系统间的相互调用

编写 PSR-7 标准（和中间件）的一个主要原因是，框架之间实现调用功能的需求在不断增长。请注意，PSR-7 标准的主要文档是由 **PHP 框架可互用性小组**（PHP Framework Interop Group，PHP-FIG）管理的。

具体处理过程

1. 对于实现框架间相互调用功能的中间件，我们使用的主要机制是创建以连续方式执行框架调用操作的驱动程序，以便维护通用的请求和回应对象。本例使用 Psr\Http\Message\ServerRequestInterface 和 Psr\Http\ Message\ ResponseInterface 类来代表请求和回应对象。

2. 为了了解具体处理过程，可定义中间件会话验证器。可使用常量和属性反映会话指纹。会话指纹是一个术语，是指网站浏览者的 IP 地址、浏览器型号和语言设置等元素：

```
namespace Application\MiddleWare\Session;
use InvalidArgumentException;
use Psr\Http\Message\ {
  ServerRequestInterface, ResponseInterface };
use Application\MiddleWare\ { Constants, Response, TextStream };
class Validator
{
  const KEY_TEXT = 'text';
  const KEY_SESSION = 'thumbprint';
  const KEY_STATUS_CODE = 'code';
  const KEY_STATUS_REASON = 'reason';
  const KEY_STOP_TIME = 'stop_time';
  const ERROR_TIME = 'ERROR: session has exceeded stop time';
  const ERROR_SESSION = 'ERROR: thumbprint does not match';
  const SUCCESS_SESSION = 'SUCCESS: session validates OK';
  protected $sessionKey;
  protected $currentPrint;
  protected $storedPrint;
  protected $currentTime;
  protected $storedTime;
```

3. 下面的构造器将 ServerRequestInterface 实例和会话接收为参数。如果该会话是一个数组（如$_SESSION），就应将其封装到一个类中。这样做的原因是可能需要发送会话对象，如在 Joomla!系统中使用的 JSession 类。这样我们就可以使用前面介绍过的元素创建会话指纹。如果无法获得已存储的会话指纹，就将该操作视为第一次创建会话指纹的操作，在会话参数被设置好后，存储当前会话指纹和停止运行时间。使用 md5() 函数的原因是该函数可以快速生成散列值，而且不会暴露到外边：

```
public function __construct(
  ServerRequestInterface $request, $stopTime = NULL)
```

```
{
  $this->currentTime  = time();
  $this->storedTime   = $_SESSION[self::KEY_STOP_TIME] ?? 0;
  $this->currentPrint =
    md5($request->getServerParams()['REMOTE_ADDR']
      . $request->getServerParams()['HTTP_USER_AGENT']
      . $request->getServerParams()['HTTP_ACCEPT_LANGUAGE']);
       $this->storedPrint = $_SESSION[self::KEY_SESSION]
     ?? NULL;
  if (empty($this->storedPrint)) {
      $this->storedPrint = $this->currentPrint;
      $_SESSION[self::KEY_SESSION] = $this->storedPrint;
      if ($stopTime) {
         $this->storedTime = $stopTime;
         $_SESSION[self::KEY_STOP_TIME] = $stopTime;
    }
  }
}
```

4. 本来无须定义__invoke()方法，但是在独立的中间件类中使用这个魔术方法会非常方便。按俗成约定，可将 ServerRequestInterface 和 ResponseInterface 实例接收为参数。我们仅使用该方法检查当前会话指纹是否与已存储的会话指纹相同。当然，第一次执行这个检查操作时会得到二者相同的结果。但是在检查后续的请求时可能会发现会话劫持攻击。此外，如果会话时间超出了停止运行时间（在设置了该时间限定值的情况下），状态码 401 就会被发送：

```
public function __invoke(
  ServerRequestInterface $request, Response $response)
{
  $code = 401; // 代表会话未被授权
  if ($this->currentPrint != $this->storedPrint) {
      $text[self::KEY_TEXT] = self::ERROR_SESSION;
      $text[self::KEY_STATUS_REASON] =
      Constants::STATUS_CODES[401];
  } elseif ($this->storedTime) {
     if ($this->currentTime > $this->storedTime) {
         $text[self::KEY_TEXT] = self::ERROR_TIME;
         $text[self::KEY_STATUS_REASON] =
           Constants::STATUS_CODES[401];
     } else {
         $code = 200; // 代表当前会话指纹与已存储的会话指纹相同
```

```
        }
    }
    if ($code == 200) {
        $text[self::KEY_TEXT] = self::SUCCESS_SESSION;
        $text[self::KEY_STATUS_REASON] =
          Constants::STATUS_CODES[200];
    }
    $text[self::KEY_STATUS_CODE] = $code;
    $body = new TextStream(json_encode($text));
    return $response->withStatus($code)->withBody($body);
}
```

5. 现在可以使用新建的中间件了。下面列出了此刻实现框架间调用操作的主要问题。可以看出，我们实现中间件的方式在很大程度上取决于最后一点：

- 并非所有 PHP 框架都符合 PSR-7 标准
- 已实现的 PSR-7 标准并不完全
- 所有框架都想成为调用者

6. 可参阅 **Zend Expressive** 的配置文件 middlewarepipeline.global.php，Zend Expressive 是一个 PSR 7 中间件微框架。在标准的 Zend Expressive 应用程序中，`middlewarepipeline.global.php` 文件位于 `config/autoload` 文件夹中。依赖关系键用于标识将会在管道中被激活的中间件封装器类：

```
<?php
use Zend\Expressive\Container\ApplicationFactory;
use Zend\Expressive\Helper;
return [
  'dependencies' => [
     'factories' => [
        Helper\ServerUrlMiddleware::class =>
        Helper\ServerUrlMiddlewareFactory::class,
        Helper\UrlHelperMiddleware::class =>
        Helper\UrlHelperMiddlewareFactory::class,
        // 在此处添加你自己编写的类
  ],
],
```

7. 在 `middleware_pipline` 键的下方，可添加在路由处理过程之前或之后执行的类。可选的参数包括 `path`、`error` 和 `priority`：

```
'middleware_pipeline' => [
    'always' => [
      'middleware' => [
         Helper\ServerUrlMiddleware::class,
    ],
    'priority' => 10000,
  ],
  'routing' => [
    'middleware' => [
         ApplicationFactory::ROUTING_MIDDLEWARE,
         Helper\UrlHelperMiddleware::class,
         // 在此处插入引用中间件的语句
         ApplicationFactory::DISPATCH_MIDDLEWARE,
       ],
       'priority' => 1,
  ],
  'error' => [
     'middleware' => [
       // 在此处添加用于处理错误的中间件
     ],
     'error' => true,
     'priority' => -10000,
    ],
  ],
];
```

8. 此处的另一个编程技巧是修改现存框架模块的源代码，并向符合 PSR-7 标准的中间件应用程序发送请求。本例会修改 **Joomla!**的安装文件，以便在其中添加中间件会话验证器。

9. 将下面的代码添加到/path/to/joomla 文件夹中 index.php 文件的尾部。因为 Joomla!系统使用 Composer 管理工具，所以我们可以利用 Composer 管理工具中的自动加载器：

```
session_start();      // 为$_SESSION 变量提供支持
$loader = include __DIR__ . '/libraries/vendor/autoload.php';
$loader->add('Application', __DIR__ . '/libraries/vendor');
$loader->add('Psr', __DIR__ . '/libraries/vendor');
```

10.为我们编写的中间件会话验证器创建一个实例，在$app = JFactory::getApplication('site');语句前面创建验证请求：

```
$session = JFactory::getSession();
$request =
```

```
  (new Application\MiddleWare\ServerRequest())->initialize();
$response = new Application\MiddleWare\Response();
$validator = new Application\Security\Session\Validator(
  $request, $session);
$response = $validator($request, $response);
if ($response->getStatusCode() != 200) {
  // 在此处添加具体操作
}
```

具体运行情况

使用步骤 2 至步骤 5 介绍的代码创建用于进行测试的中间件类 Application\MiddleWare\Session\Validator。浏览 https://getcomposer.org/，并按照提示下载 Composer 管理工具。将该工具存储在/path/toz/source/for/this/chapter 文件夹中。使用下面的命令创建一个基础的 Zend Expressive 应用程序。注意，在提示是否选用 minimal skeleton 选项时，应选择 No：

```
cd /path/to/source/for/this/chapter

php composer.phar create-project zendframework/zend-expressive-skeleton
expressive
```

这样就创建了 folder/path/to/source/for/this/chapter/expressive 文件夹。切换到该目录，使用下面的代码修改 public/index.php 文件：

```
<?php
if (php_sapi_name() === 'cli-server'
    && is_file(__DIR__ . parse_url(
$_SERVER['REQUEST_URI'], PHP_URL_PATH))
) {
   return false;
}
chdir(dirname(__DIR__));
session_start();
$_SESSION['time'] = time();
$appDir = realpath(__DIR__ . '/../../..');
$loader = require 'vendor/autoload.php';
$loader->add('Application', $appDir);
$container = require 'config/container.php';
$app = $container->get(\Zend\Expressive\Application::class);
$app->run();
```

创建一个封装器类，使用该类封装我们编写的会话验证器中间件。在/path/to/source/for/this/chapter/expressive/src/App/Action文件夹中创建SessionValidateAction.php文件。为了观察不同的对比结果，可将停止运行时间参数设置得较小。在本例中使用time() + 10语句可以将该参数设置为10秒钟：

```
namespace App\Action;
use Application\MiddleWare\Session\Validator;
use Zend\Diactoros\ { Request, Response };
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;
class SessionValidateAction
{
  public function __invoke(ServerRequestInterface $request,
  ResponseInterface $response, callable $next = null)
  {
    $inbound    = new Response();
    $validator  = new Validator($request, time()+10);
    $inbound    = $validator($request, $response);
    if ($inbound->getStatusCode() != 200) {
        session_destroy();
        setcookie('PHPSESSID', 0, time()-300);
        $params = json_decode(
          $inbound->getBody()->getContents(), TRUE);
          echo '<h1>',$params[Validator::KEY_TEXT],'</h1>';
          echo '<pre>',var_dump($inbound),'</pre>';
          exit;
    }
    return $next($request,$response);
  }
}
```

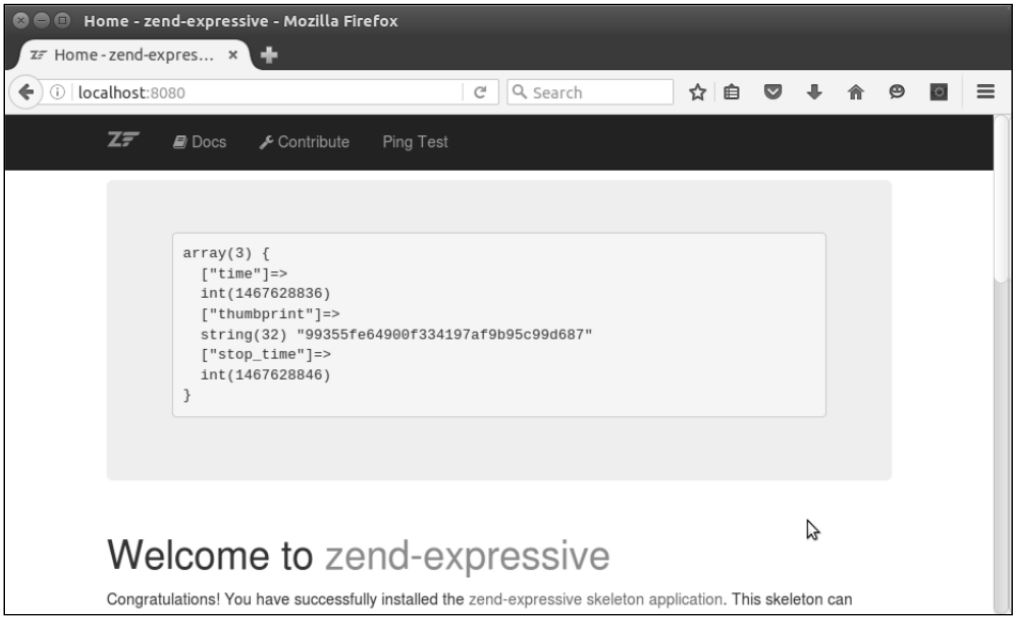
现在可向中间件管道中添加这个新建的类。使用下面的代码修改config/ autoload/middleware-pipeline.global.php文件，需修改的代码以粗体显示：

```
<?php
use Zend\Expressive\Container\ApplicationFactory;
use Zend\Expressive\Helper;
return [
  'dependencies' => [
     'invokables' => [
        App\Action\SessionValidateAction::class =>
```

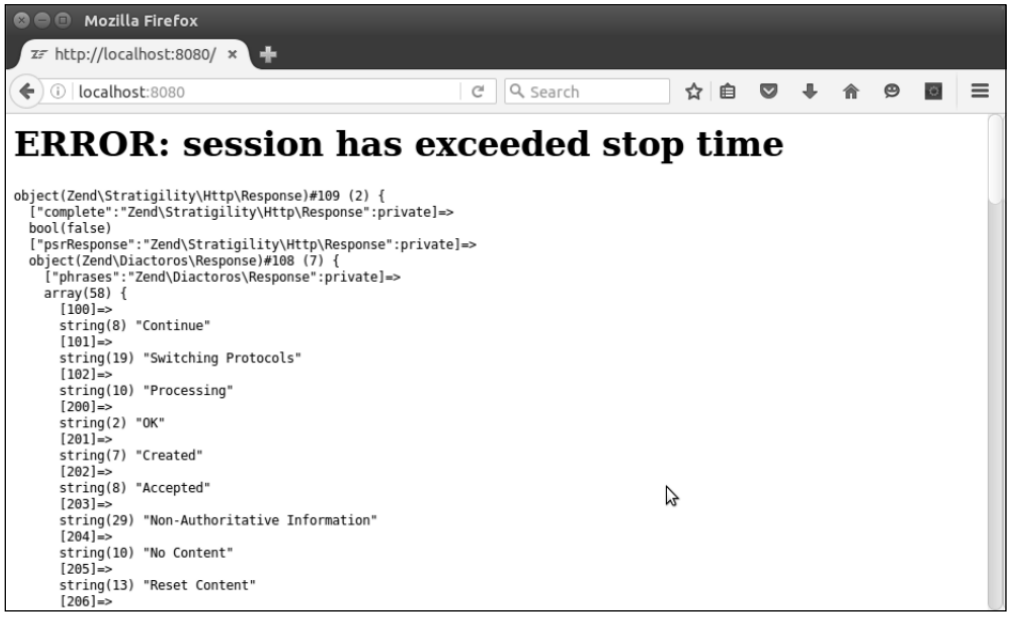
```
            App\Action\SessionValidateAction::class,
        ],
        'factories' => [
            Helper\ServerUrlMiddleware::class =>
            Helper\ServerUrlMiddlewareFactory::class,
            Helper\UrlHelperMiddleware::class =>
            Helper\UrlHelperMiddlewareFactory::class,
        ],
    ],
    'middleware_pipeline' => [
        'always' => [
            'middleware' => [
                Helper\ServerUrlMiddleware::class,
            ],
            'priority' => 10000,
        ],
        'routing' => [
            'middleware' => [
                ApplicationFactory::ROUTING_MIDDLEWARE,
                Helper\UrlHelperMiddleware::class,
                App\Action\SessionValidateAction::class,
                ApplicationFactory::DISPATCH_MIDDLEWARE,
            ],
            'priority' => 1,
        ],
        'error' => [
            'middleware' => [
                // 在此处添加用于处理错误的中间件。
            ],
            'error' => true,
            'priority' => -10000,
        ],
    ],
];
```

还可以通过修改主页模板显示$_SESSION 变量的状态。主页模板文件为/path/to/source/for/this/chapter/expressive/templates/app/home-page.phtml。只需在该文件中添加 var_dump($_SESSION)语句，即可达到该目的。

最初，浏览器中会显示下面的结果：



10 秒钟后刷新浏览器，会得到下面的效果：



使用中间件实现跨编程语言功能

除需要在不同版本的 PHP 程序之间进行通信的情况外，PSR-7 中间件的作用都会维持在最低水准。我们可以回忆一下，PSR 是 PHP Standards Recommendations（**PHP 推荐标准**）的首字母缩写词。因此，如果需要创建向其他语言编写的应用程序发送的请求，就可以像处理其他网页服务的 HTTP 请求一样处理该请求。

具体处理过程

1. PHP 4 对面向对象编程方式仅提供了有限的支持。因此，在使用 PHP 4 时，最佳方式是将前面三个示例中介绍的 PSR-7 类降级。因篇幅所限此处不会介绍所有更改，但会介绍一个 PHP 4 版本的 `Application\MiddleWare\ServerRequest` 类。需要注意的第一件事情是，PHP 4 中没有命名空间！因此，我们在类名称中将下画线（`_`）用作命名空间分隔符（即使用名称前缀代替命名空间的功能）：

```
class Application_MiddleWare_ServerRequest
extends Application_MiddleWare_Request
implements Psr_Http_Message_ServerRequestInterface
{
```

2. 在 PHP 4 中，所有属性都使用关键字 `var` 定义：

```
var $serverParams;
var $cookies;
var $queryParams;
// 此处并没有列出所有属性
```

3. 除了在 PHP 4 中不能使用`$this->getServerParams()['REQUEST_URI']`之类的语法，PHP 4 中的 `initialize()`方法与 PHP 7 中的 `initialize()`方法几乎相同。因此，我们需要将这些值分别存储在一个独立的数组中：

```
function initialize()
{
  $params = $this->getServerParams();
  $this->getCookieParams();
  $this->getQueryParams();
  $this->getUploadedFiles;
  $this->getRequestMethod();
```

```
    $this->getContentType();
    $this->getParsedBody();
    return $this->withRequestTarget($params['REQUEST_URI']);
  }
```

4. 所有$_XXX超级全局变量都是在PHP 4.1.0之后被启用的：

```
  function getServerParams()
  {
    if (!$this->serverParams) {
        $this->serverParams = $_SERVER;
    }
    return $this->serverParams;
  }
  // 为节省篇幅此处没有列出所有 getXXX()方法
```

5. 空值合并操作符（null coalesce operator）是在PHP 7中被引入的。因此需要使用isset(XXX) ? XXX : '';语句代替该操作符：

```
  function getRequestMethod()
  {
    $params = $this->getServerParams();
    $method = isset($params['REQUEST_METHOD'])
      ? $params['REQUEST_METHOD'] : '';
    $this->method = strtolower($method);
    return $this->method;
  }
```

6. 在PHP 5之前，JSON扩展还没有被引入PHP中，因此我们需要使用原始输入数据。可使用 serialize()和 unserialize()函数代替 json_encode()和 json_decode()函数：

```
  function getParsedBody()
  {
    if (!$this->parsedBody) {
        if (($this->getContentType() ==
            Constants::CONTENT_TYPE_FORM_ENCODED
            || $this->getContentType() ==
             Constants::CONTENT_TYPE_MULTI_FORM)
             && $this->getRequestMethod() ==
             Constants::METHOD_POST)
        {
             $this->parsedBody = $_POST;
        } elseif ($this->getContentType() ==
                  Constants::CONTENT_TYPE_JSON
```

```
                || $this->getContentType() ==
                Constants::CONTENT_TYPE_HAL_JSON)
        {
            ini_set("allow_url_fopen", true);
            $this->parsedBody =
              file_get_contents('php://stdin');
        } elseif (!empty($_REQUEST)) {
             $this->parsedBody = $_REQUEST;
        } else {
             ini_set("allow_url_fopen", true);
             $this->parsedBody =
               file_get_contents('php://stdin');
        }
    }
    return $this->parsedBody;
}
```

7. PHP 7 中的 `withXXX()` 方法与 PHP 4 中的 `withXXX()` 方法非常相似：

```
function withParsedBody($data)
{
  $this->parsedBody = $data;
  return $this;
}
```

8. PHP 7 中的 `withoutXXX()` 方法也与 PHP 4 中的 `withoutXXX()` 方法非常相似：

```
function withoutAttribute($name)
{
  if (isset($this->attributes[$name])) {
      unset($this->attributes[$name]);
  }
  return $this;
}

}
```

9. 对于使用其他语言编写网页的网站，我们可使用 PSR-7 类创建请求和回应，但是必须使用 HTTP 客户端与这些网站进行通信。例如，可使用本章前面介绍过的 `Request` 类实现该处理过程：

```
$request = new Request(
  TARGET_WEBSITE_URL,
  Constants::METHOD_POST,
```

```
    new TextStream($contents),
    [Constants::HEADER_CONTENT_TYPE =>
    Constants::CONTENT_TYPE_FORM_ENCODED,
    Constants::HEADER_CONTENT_LENGTH => $body->getSize()]
);

$data = http_build_query(['data' =>
$request->getBody()->getContents()]);

$defaults = array(
    CURLOPT_URL => $request->getUri()->getUriString(),
    CURLOPT_POST => true,
    CURLOPT_POSTFIELDS => $data,
);
$ch = curl_init();
curl_setopt_array($ch, $defaults);
$response = curl_exec($ch);
curl_close($ch);
```

第 10 章　高级算法

本章包括以下要点：

- 使用读取器和设置器
- 实现链表
- 编写冒泡排序程序
- 实现堆栈
- 创建实现二分查找操作的类
- 实现搜索引擎
- 显示多维数组和累加合计

本章主要内容简介

本章介绍实现各种高级算法的方式，如链表、冒泡排序、堆栈和二分查找。此外，本章还会介绍读取器和设置器、实现搜索引擎的方式、显示多维数组中存储的值和这些值的累加合计的方式。

使用读取器和设置器

乍看之下，好像使用 public 关键字定义类中的属性是有道理的，因为这样就可以直接读取类和直接向类中的属性写入数据。然而，使用关键字 protected 定义属性，然后为每个属性定义读取器和设置器，才是大家认为的最佳编程习惯。顾名思义，读取器用于读取属性的值，设置器用于设置属性的值。

最佳编程习惯

将属性定义为 protected，可以通过 public get*和 set*方法访问这些属性，从而预防属性在意外情况下被外部程序访问。这种处理方式不仅使你可以更精确地控制访问操作，还允许使你在获取属性值的时候，对这些数据应用不同的格式和更改这些数据的类型。

具体处理过程

1. 在读取和设置属性值时，读取器和设置器可以提供更多灵活性。在直接读取或设置 public 属性无法实现目的时，可根据需要添加一个额外的逻辑层。使用 get 或 set 前缀创建公用的方法，将属性的名称用作这些方法的后缀。将属性名称中的第一个字母设置为大写是一种惯例。因此，如果某个属性为$testValue，那么该属性的读取器的名称就应该为 getTestValue()。

2. 本例使用受保护的属性$date 定义类。注意，该属性的读取器和设置器方法都应该能够处理 DateTime 对象和字符串。属性$date 的值实际上以 DateTime 实例的形式存储在事件中：

```
$a = new class() {
  protected $date;
  public function setDate($date)
  {
    if (is_string($date)) {
        $this->date = new DateTime($date);
    } else {
        $this->date = $date;
    }
  }
  public function getDate($asString = FALSE)
  {
    if ($asString) {
        return $this->date->format('Y-m-d H:i:s');
    } else {
        return $this->date;
    }
  }
};
```

3. 通过使用读取器和设置器，可以过滤和审查收到或发出的数据。本例中有两个属性：$intVal 和$arrVal，它们都被设置为默认的初始值 NULL。注意，读取器不仅可以用于设置返回值的数据类型，还可以用于提供返回的默认值。不仅可以使用设置器限定接收哪些类型的数据，还可以使用设置器将收到的数据转换为指定的类型：

```
<?php
class GetSet
{
  protected $intVal = NULL;
  protected $arrVal = NULL;
// 注意本例使用空值合并操作符返回默认值
  public function getIntVal() : int
  {
    return $this->intVal ?? 0;
  }
  public function getArrVal() : array
  {
    return $this->arrVal ?? array();
  }
  public function setIntVal($val)
  {
    $this->intVal = (int) $val ?? 0;
  }
  public function setArrVal(array $val)
  {
    $this->arrVal = $val ?? array();
  }
}
```

4. 如果你定义的类中含有非常多的属性，那么单独为每个属性都定义读取器和设置器方法会变成一项冗长乏味的任务。在这种情况下，可以使用魔术方法__call()定义一种回调函数。下面的类定义了 9 个属性。我们不需要定义 9 个读取器和 9 个设置器，只需定义一个方法：__call()，该方法能够判定是使用读取器还是使用设置器。如果要使用读取器，__call()方法就会检索一个内部数组中的键。如果要使用设置器，__call()方法就会将属性的值存储在这个内部数组中。

__call()是一个魔术方法，当应用程序调用不存在的方法时，__call()方法就会被调用。

```
<?php
class LotsProps
{
  protected $firstName = NULL;
  protected $lastName = NULL;
  protected $addr1 = NULL;
  protected $addr2 = NULL;
  protected $city = NULL;
  protected $state = NULL;
  protected $province = NULL;
  protected $postalCode = NULL;
  protected $country = NULL;
  protected $values = array();

  public function __call($method, $params)
  {
    preg_match('/^(get|set)(.*?)$/i', $method, $matches);
    $prefix = $matches[1] ?? '';
    $key = $matches[2] ?? '';
    $key = strtolower($key);
    if ($prefix == 'get') {
        return $this->values[$key] ?? '---';
    } else {
        $this->values[$key] = $params[0];
    }
  }
}
```

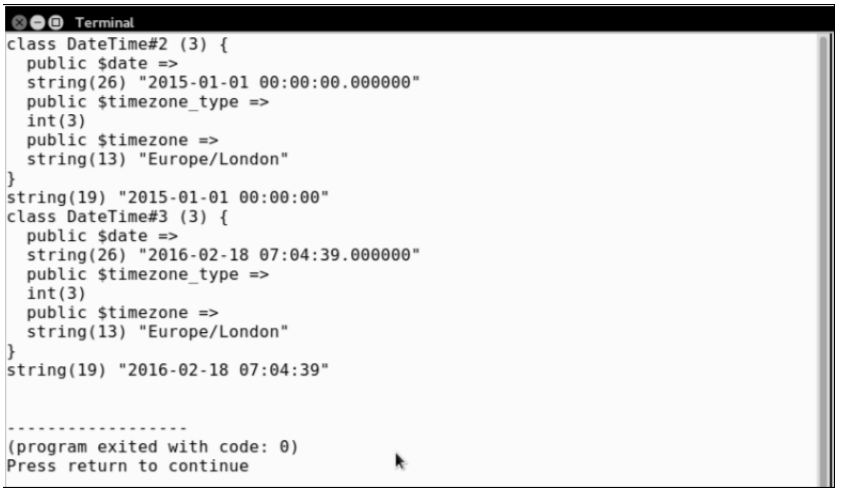
具体运行情况

将前面步骤 2 中介绍的代码添加到 chap_10_oop_using_getters_and_setters.php 文件中。要测试这个匿名类，可在该文件中添加下列代码：

```
// 使用字符串设置日期
$a->setDate('2015-01-01');
var_dump($a->getDate());

// 获取 DateTime 实例
var_dump($a->getDate(TRUE));
```

```
// 使用 DateTime 实例设置日期
$a->setDate(new DateTime('now'));
var_dump($a->getDate());

// 获取 DateTime 实例
var_dump($a->getDate(TRUE));
```

如下面的输出结果所示，既可以使用字符串也可以使用 DateTime 实例来设置$date 属性的值。当 getDate()方法被调用时，可根据$asString 变量中存储的标记值返回字符串或 DateTime 实例：



将前面步骤 3 中介绍的代码添加到 chap_10_oop_using_getters_and_setters_defaults.php 文件中，并在该文件中添加下列代码：

```
// 创建实例
$a = new GetSet();

// 设置适当的值
$a->setIntVal(1234);
echo $a->getIntVal();
echo PHP_EOL;

// 设置伪造的值
$a->setIntVal('some bogus value');
echo $a->getIntVal();
echo PHP_EOL;

// 注意：布尔值 TRUE 等于 1
$a->setIntVal(TRUE);
```

```
echo $a->getIntVal();
echo PHP_EOL;

// 即使没有任何值被设置也返回数组
var_dump($a->getArrVal());
echo PHP_EOL;

// 设置适当的值
$a->setArrVal(['A','B','C']);
var_dump($a->getArrVal());
echo PHP_EOL;

try {
    $a->setArrVal('this is not an array');
    var_dump($a->getArrVal());
    echo PHP_EOL;
} catch (TypeError $e) {
    echo $e->getMessage();
}

echo PHP_EOL;
```

如下面的输出结果所示，为属性设置适当的整型值，程序就能够像我们期望的那样运行。使用非数值型数据设置属性会使属性被设置为默认值 0。有趣的是，如果将布尔值 TRUE 用作 setIntVal()方法的参数,那么该布尔值就会以内插的方式被替换为 1。

如果在没有设置属性值的情况下调用 getArrVal()方法,那么属性的默认值就会是一个空数组。将属性值设置为数组可以使程序按照我们期望的方式运行。然而，如果将非

数组值用作参数，数组的类型提示功能就会使程序抛出 TypeError 异常，该异常可以捕捉，参见上图。

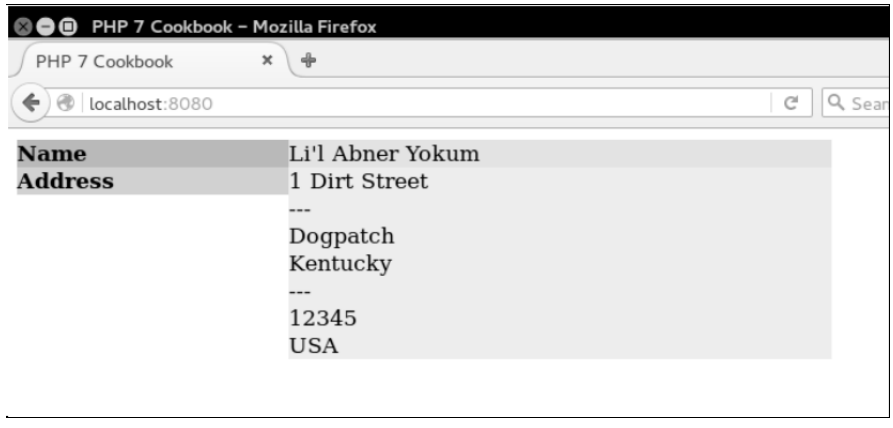
将前面步骤 4 介绍的 LotsProps 类添加到独立文件 chap_10_oop_using_getters_and_setters_magic_call.php 中。将下面的代码也添加到该文件中，以设置 LotsProps 类中各个属性的值。这样做必然会使魔术方法__call()被调用。调用了 preg_match()函数后，以 set 为名称前缀的其余不存在的属性设置器都会变成内部数组$values 中的键：

```
$a = new LotsProps();
$a->setFirstName('Li\'l Abner');
$a->setLastName('Yokum');
$a->setAddr1('1 Dirt Street');
$a->setCity('Dogpatch');
$a->setState('Kentucky');
$a->setPostalCode('12345');
$a->setCountry('USA');
?>
```

这样就可以使用相应的读取器方法来定义用于显示这些属性值的 HTML 代码。反过来，这些读取器方法会从内部数组返回键：

```
<div class="container">
<div class="left blue1">Name</div>
<div class="right yellow1">
<?= $a->getFirstName() . ' ' . $a->getLastName() ?></div>
</div>
<div class="left blue2">Address</div>
<div class="right yellow2">
     <?= $a->getAddr1() ?>
     <br><?= $a->getAddr2() ?>
     <br><?= $a->getCity() ?>
     <br><?= $a->getState() ?>
     <br><?= $a->getProvince() ?>
     <br><?= $a->getPostalCode() ?>
     <br><?= $a->getCountry() ?>
</div>
</div>
```

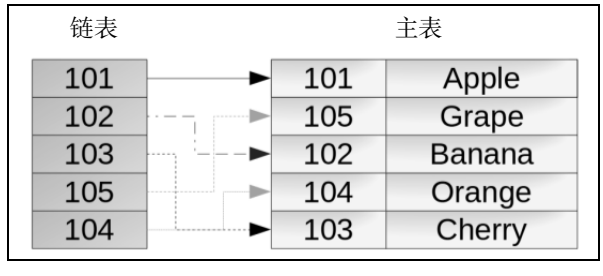
下面是最终的输出结果：



实现链表

链表是指一种特殊的列表，这种列表中含有指向其他列表中键的键。这就像数据库中含有数据的表和指向这些数据的独立索引。一个索引可能会根据 ID 生成一组条目。另一个索引可能会根据标题生成另一组条目，依此类推。链表的显著功能是让你无须接触原始列表的内容。

如下图所示，主表含有水果的 ID 和名称。如果你直接输出主表的内容，水果的名称会以 **Apple**、**Grape**、**Banana**、**Orange**、**Cherry** 的顺序显示。如果将链表用作主表的索引，就可以按英文字母表的顺序显示 **Apple**、**Banana**、**Cherry**、**Grape**、**Orange**：



具体处理过程

1. 链表的主要用途之一是以不同的顺序显示条目。达到该目的的方式之一是创建键/值对迭代，使用键代表新次序，使用值代表主表中的条目。下面是一个示例函数：

```
function buildLinkedList(array $primary,
                         callable $makeLink)
{
  $linked = new ArrayIterator();
  foreach ($primary as $key => $row) {
    $linked->offsetSet($makeLink($row), $key);
  }
  $linked->ksort();
  return $linked;
}
```

2. 可使用匿名函数生成新的键，以便获得额外的灵活性。注意，我们使用键（ksort()函数）执行排序操作，因此链表是按照这些键的次序迭代的。

3. 我们仅需要对列表做循环迭代的操作，本例通过主表$customer生成输出结果：

```
foreach ($linked as $key => $link) {
  $output .= printRow($customer[$link]);
}
```

4. 注意，我们绝不会碰主表的内容。这样我们就可以生成多个链表，每个链表都能够代表一种次序，同时可保护主表中的原始数据。

5. 链表的另一个重要用途是执行过滤操作。该技巧与前面介绍的排序技巧类似。它们之间的唯一差异是，我们扩展了buildLinkedList()函数，在其中添加了过滤字段和过滤值：

```
function buildLinkedList(array $primary,
                         callable $makeLink,
                         $filterCol = NULL,
                         $filterVal = NULL)
{
  $linked = new ArrayIterator();
  $filterVal = trim($filterVal);
  foreach ($primary as $key => $row) {
    if ($filterCol) {
       if (trim($row[$filterCol]) == $filterVal) {
           $linked->offsetSet($makeLink($row), $key);
      }
    } else {
      $linked->offsetSet($makeLink($row), $key);
    }
```

```
    }
    $linked->ksort();
    return $linked;
}
```

6. 我们仅在链表中包含符合条件的主表条目，这些条目在$filterCol 变量中存储的字段值必须与在$filterVal 变量中存储的过滤值相同。这段程序中的迭代逻辑与前面步骤 2 中介绍的迭代逻辑相同。

7. 链表的另一种形式是双向链表，双向链表既可以正向迭代也可以反向迭代。在 PHP 中，我们可以使用 SPL 类 SplDoublyLinkedList 巧妙地实现该功能。下面的函数使用 SplDoublyLinkedList 类创建了一个双向链表：

```
function buildDoublyLinkedList(ArrayIterator $linked)
{
  $double = new SplDoublyLinkedList();
  foreach ($linked as $key => $value) {
    $double->push($value);
  }
  return $double;
}
```

SplDoublyLinkedList 类可能会引发歧义。SplDoublyLinkedList::top()方法实际上指向的是列表的末尾，而 SplDoublyLinkedList::bottom()指向的是列表的开头！

具体运行情况

将步骤 1 介绍的代码添加到 chap_10_linked_list_include.php 文件中。为了做使用链表的实验，我们需要使用数据源。为了了解详细的处理步骤，可将本书前面介绍的 customer.csv 文件用作数据源。这个 CSV 文件含有下列字段：

```
"id","name","balance","email","password","status","security_question",
"confirm_code","profile_id","level"
```

可以将下列函数添加到 chap_10_linked_list_include.php 文件中，以便生成存储客户记录的主表，并显示相关信息。注意，我们将第一个字段 id 用作主键：

```
function readCsv($fn, &$headers)
{
```

```
  if (!file_exists($fn)) {
    throw new Error('File Not Found');
  }
  $fileObj = new SplFileObject($fn, 'r');
  $result = array();
  $headers = array();
  $firstRow = TRUE;
  while ($row = $fileObj->fgetcsv()) {
    // 将第一行记录存储为报头
    if ($firstRow) {
      $firstRow = FALSE;
      $headers = $row;
    } else {
      if ($row && $row[0] !== NULL && $row[0] !== 0) {
        $result[$row[0]] = $row;
      }
    }
  }
  return $result;
}

function printHeaders($headers)
{
  return sprintf('%4s : %18s : %8s : %32s : %4s' . PHP_EOL,
                 ucfirst($headers[0]),
                 ucfirst($headers[1]),
                 ucfirst($headers[2]),
                 ucfirst($headers[3]),
                 ucfirst($headers[9]));
}

function printRow($row)
{
  return sprintf('%4d : %18s : %8.2f : %32s : %4s' . PHP_EOL,
                 $row[0], $row[1], $row[2], $row[3], $row[9]);
}

function printCustomer($headers, $linked, $customer)
{
  $output = ' ';
  $output .= printHeaders($headers);
```

```
  foreach ($linked as $key => $link) {
    $output .= printRow($customer[$link]);
  }
  return $output;
}
```

这样就可以定义调用程序 chap_10_linked_list_in_order.php，在该文件中包含前面定义的文件，并读取 customer.csv 文件：

```
<?php
define('CUSTOMER_FILE', __DIR__ . '/../data/files/customer.csv');
include __DIR__ . '/chap_10_linked_list_include.php';
$headers = array();
$customer = readCsv(CUSTOMER_FILE, $headers);
```

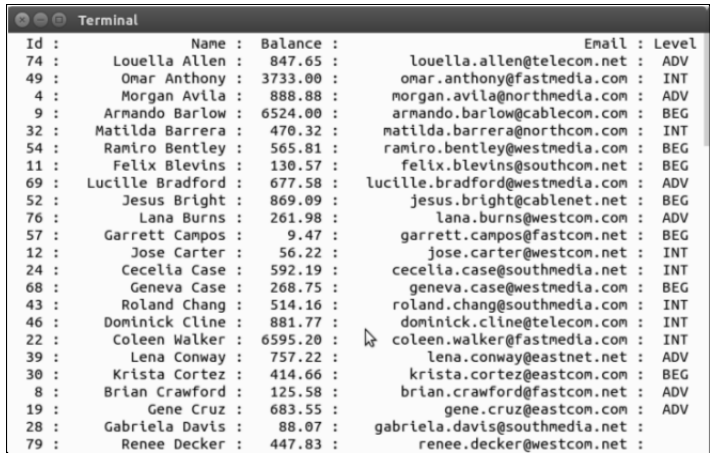
定义一个匿名函数，使该函数生成链表中的键。本例定义的函数会将第一个字段（name）拆分为名字（存储在$first 变量中）和姓氏（存储在$last 变量中）：

```
$makeLink = function ($row) {
  list($first, $last) = explode(' ', $row[1]);
  return trim($last) . trim($first);
};
```

这样就可以通过调用这个函数创建链表，并使用 printCustomer()方法显示结果：

```
$linked = buildLinkedList($customer, $makeLink);
echo printCustomer($headers, $linked, $customer);
```

下面是输出结果：



要过滤这些结果可按照步骤 4 介绍的方式修改 buildLinkedList()方法。可添

加检查逻辑，以查明选定的用于执行过滤操作的字段值是否与设定的过滤值匹配：

```
define('LEVEL_FILTER', 'INT');

$filterCol = 9;
$filterVal = LEVEL_FILTER;
$linked = buildLinkedList($customer, $makeLink, $filterCol,
$filterVal);
```

补充说明

PHP 7.1 引入了 `[ ]`，以替代 `list()` 函数。如果你使用的 PHP 处理程序是 PHP 7.1，可使用下面的代码修改前面介绍的匿名函数：

```
$makeLink = function ($row) {
  [$first, $last] = explode(' ', $row[1]);
  return trim($last) . trim($first);
};
```

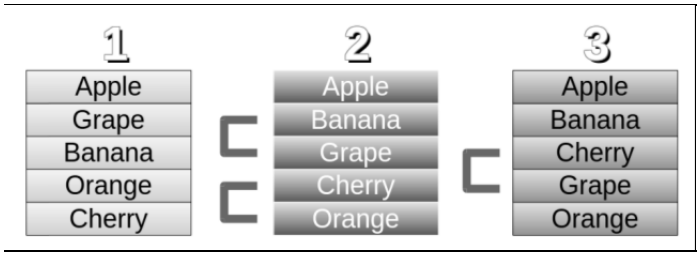
要详细了解这部分语法，请浏览 `https://wiki.php.net/rfc/short_list_syntax`。

编写冒泡排序程序

经典的冒泡排序算法是常见的大学练习题。尽管你可能已经很熟悉了，但还是要强调这个算法是非常重要的，因为有很多无法使用 PHP 内置排序函数的情况。一种典型的情况是，在对多维数组进行排序时，用于执行排序操作的键不是第一个字段。

冒泡排序算法的运行方式是以递归方式遍历列表，并将当前值与下一个值交换位置。如果你想要使用升序排列列表中的条目，就应在下一个值小于当前值的情况下执行交换操作；如果想要使用降序排列列表中的条目，就应在下一个值大于当前值的情况下执行交换操作。当不再有交换操作出现时，排序过程就结束了。

在下面的算法中，第一轮排序遍历过程执行后，**Grape** 和 **Banana** 条目的位置互换了，**Orange** 和 **Cherry** 条目的位置也互换了。第二轮排序遍历过程执行后，**Grape** 和 **Cherry** 条目的位置互换了。第三轮排序遍历过程执行后，没有出现交换操作，冒泡排序处理过程就结束了：



具体处理过程

1. 实际上，我们不打算直接使数组中存储的值交换位置，因为这样做会耗费大量的资源。我们会使用上一节介绍的**链表**。

2. 先使用上一节介绍的 `buildLinkedList()` 函数创建一个链表。

3. 然后定义一个新函数 `bubbleSort()`，使该函数根据引用、主表、排序字段和代表排序类型（升序或降序）的参数接收链表：

```
function bubbleSort(&$linked, $primary, $sortField, $order = 'A')
{
```

4. 需要在该函数中定义的变量包括代表迭代次数的变量、存储交换操作执行次数的变量和存储根据链表创建的迭代器的变量：

```
static $iterations = 0;
$swaps = 0;
$iterator = new ArrayIterator($linked);
```

5. 在 `while()` 循环中，只有在迭代操作合法的情况下才能继续执行循环操作（即继续执行交换位置操作）。这样就可以获得当前的键和值，以及下一个键和值。注意，额外添加的 `if()` 语句用于确保迭代操作的合法性（即确保不会落下列表的末尾）：

```
while ($iterator->valid()) {
  $currentLink = $iterator->current();
  $currentKey = $iterator->key();
  if (!$iterator->valid()) break;
  $iterator->next();
  $nextLink = $iterator->current();
  $nextKey  = $iterator->key();
```

6. 查明排序操作是降序的还是升序的。根据循环遍历的方向，我们来查看下一个值是大于还是小于当前值。将比较结果存储在`$expr` 变量中：

```
if ($order == 'A') {
   $expr = $primary[$linked->offsetGet
```

```
    ($currentKey)][$sortField] >
          $primary[$linked->offsetGet($nextKey)][$sortField];
} else {
   $expr = $primary[$linked->offsetGet
          ($currentKey)][$sortField] <
          $primary[$linked->offsetGet($nextKey)][$sortField];
}
```

7. 如果$expr 变量的值为 TRUE，就说明当前键和下一个键是合法的，而链表中的值会被交换。还应将$swaps 变量（代表执行交换操作的次数）中存储的值加 1：

```
if ($expr && $currentKey && $nextKey
     && $linked->offsetExists($currentKey)
     && $linked->offsetExists($nextKey)) {
     $tmp = $linked->offsetGet($currentKey);
     $linked->offsetSet($currentKey,
     $linked->offsetGet($nextKey));
     $linked->offsetSet($nextKey, $tmp);
     $swaps++;
  }
}
```

8. 如果在一轮循环中出现了交换操作，就需要再次执行迭代操作，直到不再出现交换操作为止。因此，可通过递归方式调用 bubbleSort 函数：

```
if ($swaps) bubbleSort($linked, $primary, $sortField, $order);
```

9. 实际的返回值是一个重新排过序的链表。为了了解具体运行情况，还可以返回执行迭代操作的次数：

```
  return ++$iterations;
}
```

具体运行情况

将前面介绍的 bubbleSort() 函数添加到上一节介绍的 chap_10_linked_list_include.php 文件中。可使用上一节介绍的逻辑读取 customer.csv 文件以生成一个主表：

```
<?php
define('CUSTOMER_FILE', __DIR__ . '/../data/files/customer.csv');
include __DIR__ . '/chap_10_linked_list_include.php';
$headers = array();
$customer = readCsv(CUSTOMER_FILE, $headers);
```

可将第一列用作排序键，从而生成一个链表：

```
$makeLink = function ($row) {
  return $row[0];
};
$linked = buildLinkedList($customer, $makeLink);
```

最后，调用 bubbleSort()函数，将该链表和存储客户记录的列表（通过读取customer.csv文件创建的）用作参数。还可以设定排序字段（本例使用第三个字段，该字段代表客户的账户余额），并使用字母“A”指明该排序操作为升序。可使用printCustomer()函数显示输出结果：

```
echo 'Iterations: ' . bubbleSort($linked,
                                 $customer, 2, 'A') . PHP_EOL;
echo printCustomer($headers, $linked, $customer);
```

下面是输出结果：

```
Terminal
Iterations: 82
  Id :              Name :  Balance :                        Email : Level
 101 :      Leonard Nimoy :   -99.99 :      mrspock788843@starfleet.gov :  ADV
  21 :       Lauri Grimes :   -37.95 :        lauri.grimes@cablecom.com :  ADV
  20 :     Samuel Harding :   -11.56 :    samuel.harding@southmedia.net :  ADV
  88 :              Obama :     0.00 :             obama@president.gov :  BEG
  92 :       C.T. Russell :     0.00 :               ctrussell@jw.org :  BEG
  99 :              admin :     0.00 :         admin@sweetscomplete.com :  ADV
  57 :     Garrett Campos :     9.47 :      garrett.campos@fastcom.net :  BEG
  45 :    Wilfredo Taylor :    25.11 :      wilfredo.taylor@telecom.net :  BEG
  25 :      Rhonda Kinney :    46.61 :      rhonda.kinney@fastmedia.com :  BEG
  58 :       Todd Lindsey :    48.91 :         todd.lindsey@fastnet.net :  ADV
  12 :       Jose Carter :    56.22 :          jose.carter@westcom.net :  INT
  71 :       Fannie Moore :    68.48 :  fannie.moore@cablemedia.net :  ADV
  16 :        Marc Ellis :    69.04 :           marc.ellis@westnet.com :  ADV
  28 :     Gabriela Davis :    88.07 :    gabriela.davis@southmedia.net :
   6 :    Spencer Sanford :    99.99 :     spencer.sanford@cablenet.net :  INT
  44 :    Raymond Sanford :   101.41 :     raymond.sanford@cablenet.net :  ADV
   8 :     Brian Crawford :   125.58 :       brian.crawford@fastcom.net :  ADV
  11 :      Felix Blevins :   130.57 :       felix.blevins@southcom.net :  BEG
  48 :    Edmond Shepherd :   135.29 :   edmond.shepherd@southmedia.com :  ADV
  50 :       Lonnie Eaton :   139.07 :        lonnie.eaton@southcom.net :  ADV
  65 :   Isabel Rodriguez :   142.87 :    isabel.rodriguez@fastcom.net :  BEG
  75 :  Jeannette Merritt :   146.89 : jeannette.merritt@northmedia.com :  BEG
```

实现堆栈

堆栈是一种数据结构，其算法非常简单：**后进先出**（Last In First Out，LIFO），或**先进后出**（First In Last Out，FILO）。堆栈就像堆在图书馆桌子上的一叠书，当图书管理员把这些书重新摆放回书架上时，最先处理的是这堆书中最上面的那本，然后会按顺序继续向下处理，直到处理完为止。这堆书中最上面那本书是最后一个被放在桌上的，

但它是被第一个处理的。

编程术语“堆栈”用于描述暂时存储信息的数据结构。堆栈的检索次序有助于最快速地获取最近存储的记录。

具体处理过程

1. 先定义 Application\Generic\Stack 类。使用 SPL 类 SplStack 实现核心逻辑：

```
namespace Application\Generic;
use SplStack;
class Stack
{
  // 此处添加具体代码
}
```

2. 定义用于代表堆栈的属性，设置 SplStack 实例：

```
protected $stack;
public function __construct()
{
  $this->stack = new SplStack();
}
```

3. 编写向堆栈中写入数据和从堆栈读取数据的方法，即经典的 push() 方法（代表压入操作）和 pop() 方法（代表弹出操作）：

```
public function push($message)
{
  $this->stack->push($message);
}
public function pop()
{
  return $this->stack->pop();
}
```

4. 还应编写__invoke() 方法，使该方法能够返回 stack 属性中存储的实例。这样我们就能够通过直接调用函数的方式使用这个对象：

```
public function __invoke()
{
  return $this->stack;
}
```

具体运行情况

堆栈的一种用途是存储消息。在处理消息时，通常需要先获取最新的消息，因此这也是一种最适合使用堆栈的情况。使用前面介绍的代码定义 `Application\Generic\Stack` 类。然后，定义一个调用程序，为其设置类自动加载功能，并创建代表堆栈的实例：

```
<?php
// 设置类自动加载功能
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Generic\Stack;
$stack = new Stack();
```

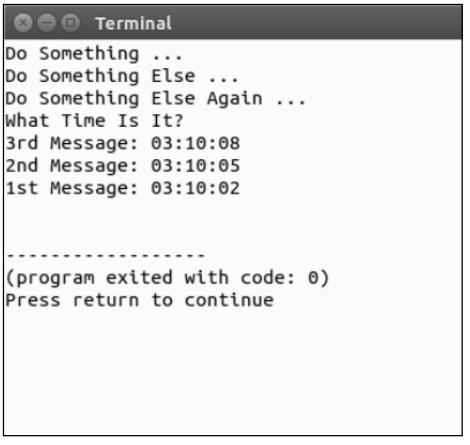
要使用堆栈做实验，可在其中存储一组消息。因为在现实情况中很可能需要在应用程序的不同位置存储消息，所以可使用 `sleep()` 函数模拟应用程序中其他代码部分运行时占用的时间：

```
echo 'Do Something ... ' . PHP_EOL;
$stack->push('1st Message: ' . date('H:i:s'));
sleep(3);

echo 'Do Something Else ... ' . PHP_EOL;
$stack->push('2nd Message: ' . date('H:i:s'));
sleep(3);

echo 'Do Something Else Again ... ' . PHP_EOL;
$stack->push('3rd Message: ' . date('H:i:s'));
sleep(3);
```

循环遍历堆栈就可以获取消息。注意，可以像调用函数那样调用代表堆栈的对象，这样做可以获得返回的 `SplStack` 实例：

```
echo 'What Time Is It?' . PHP_EOL;
foreach ($stack() as $item) {
  echo $item . PHP_EOL;
}
```

下面是输出结果：



创建实现二分查找操作的类

常规搜索操作通常会以顺序方式遍历列表中的条目。这意味着要找到数值最大的条目，很可能需要访问列表的全部内容！这种方式效率不高。如果你需要加快搜索速度，可考虑使用二分查找算法。

这种技巧非常简单：找到列表的中间点，判断被搜索条目的值是小于、等于还是大于位于中间点的条目。如果被搜索条目的值小于中间点条目的值，就将中间点的条目设置为搜索操作的最底端边界，然后仅在该列表的上半部分中执行搜索操作。如果被搜索条目的值大于中间点条目的值，就将中间点的条目设置为搜索操作的最顶端边界，然后仅在该列表的下半部分中执行搜索操作。然后，可以继续将该列表划分为 1/4、1/8、1/16，依此类推，直到找到要搜索的条目（或确定列表中不含有你要搜索的条目）为止。

使用二分查找算法时需要执行比较操作的次数为 $\log n + 1$ 次，其中 n 为列表中含有的条目数，该算式求取的结果为以 2 为底数、以 n 为真数的对数。可以看出，这比使用常规搜索算法时执行比较操作的次数少得多，但在使用二分查找算法执行搜索操作前必须先对列表进行排序，这显然会降低搜索操作的性能。

具体处理过程

1. 先创建用于执行搜索操作的类 Application\Generic\Search，使该类将主表接收为参数。为了控制执行循环操作的次数，可定义$iterations 属性：

```
namespace Application\Generic;
class Search
{
  protected $primary;
  protected $iterations;
  public function __construct($primary)
  {
    $this->primary = $primary;
  }
```

2. 定义 binarySearch()方法，使用该方法设置实现搜索操作的基础代码。第一件需要做的事情是创建独立数组$search，该数组的键由在搜索操作中使用的字段的值构成。这样我们就可以根据这些键执行排序操作：

```
public function binarySearch(array $keys, $item)
{
  $search = array();
  foreach ($this->primary as $primaryKey => $data) {
    $searchKey = function ($keys, $data) {
      $key = '';
      foreach ($keys as $k) $key .= $data[$k];

      return $key;
    };
    $search[$searchKey($keys, $data)] = $primaryKey;
}
ksort($search);
```

3. 将这些键存储到另一个数组（$binary）中，这样我们就可以根据这些数值型键执行二分排序操作了。调用 doBinarySearch()方法，该方法会通过中间数组$search 返回一个键或布尔值 FALSE：

```
  $binary = array_keys($search);
  $result = $this->doBinarySearch($binary, $item);
  return $this->primary[$search[$result]] ?? FALSE;
}
```

4. doBinarySearch()方法会初始化一系列参数。$iterations、$found、

$loop、$done 和$max 变量都用于防止出现无限循环的情况。$upper 和$lower 变量用于在主表中划分搜索范围：

```
public function doBinarySearch($binary, $item)
{
  $iterations = 0;
  $found= FALSE;
  $loop = TRUE;
  $done = -1;
  $max  = count($binary);
  $lower= 0;
  $upper= $max - 1;
```

5. 实现 while() 循环并设置中间点：

```
while ($loop && !$found) {
  $mid = (int) (($upper - $lower) / 2) + $lower;
```

6. 使用 PHP 7 新增的**飞船操作符**（spaceship operator，综合比较运算符），仅通过一次比较就可以获得小于、等于或大于的结果。如果被搜索条目的值小于中间点条目的值，就将中间点的条目设置为搜索操作的最底端边界，然后仅在该列表的上半部分中执行搜索操作。如果被搜索条目的值大于中间点条目的值，就将中间点的条目设置为搜索操作的最顶端边界，然后仅在该列表的下半部分中执行搜索操作。如果被搜索条目的值等于中间点条目的值，那么就可以结束搜索操作，中间点条目就是我们要搜索的记录：

```
switch ($item <=> $binary[$mid]) {
  // $item 变量中存储的被搜索条目的值小于$binary[$mid]数组元素中
  //存储的中间点条目的值
  case -1 :
  $upper = $mid;
  break;
  // $item 变量中存储的被搜索条目的值等于$binary[$mid]数组元素中
  //存储的中间点条目的值
  case 0 :
  $found = $binary[$mid];
  break;
  // $item 变量中存储的被搜索条目的值大于$binary[$mid]数组元素中
  //存储的中间点条目的值
  case 1 :
  default :
  $lower = $mid;
}
```

7. 为了控制执行循环操作的次数，每执行一轮循环就应将$iterations变量中存储的数值加 1，并确保该值不超过列表中条目的总数。如果该值超过了列表中条目的总数，就明确表明出错了，需要进行补救。否则，应检查在一轮循环中最底端边界的值等于最顶端边界的值的次数是否超过了两次，如果出现这种情况，就说明被搜索的条目不在主表中。那么就应该将循环次数存储起来，并返回找到的结果（或代表没有找到结果的布尔值）：

```
        $loop = (($iterations++ < $max) && ($done < 1));
        $done += ($upper == $lower) ? 1 : 0;
      }
      $this->iterations = $iterations;
      return $found;
    }
```

具体运行情况

先使用前面介绍的方法实现Application\Generic\Search类。然后定义调用程序chap_10_binary_search.php，为其设置类自动加载功能，使之读取customer.csv文件（请参阅前面介绍的内容），生成用作搜索范围的主表：

```
<?php
define('CUSTOMER_FILE', __DIR__ . '/../data/files/customer.csv');
include __DIR__ . '/chap_10_linked_list_include.php';
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Generic\Search;
$headers = array();
$customer = readCsv(CUSTOMER_FILE, $headers);
```

创建新的Search实例，并设定该表的中间点。为了进行实验，可根据该表中的第二个字段name（代表客户的姓名）执行搜索操作，将姓名为Todd Lindsey的客户记录设定为搜索目标：

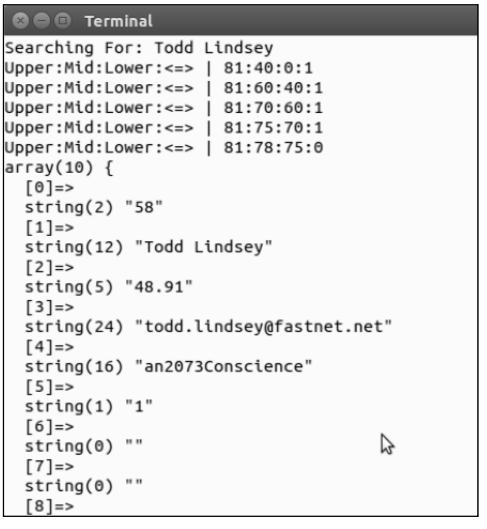
```
$search = new Search($customer);
$item = 'Todd Lindsey';
$cols = [1];
echo "Searching For: $item\n";
var_dump($search->binarySearch($cols, $item));
```

为了了解具体处理过程，可在Application\Generic\Search::doBinary

Search()方法中 switch()语句的前面添加下面的代码：

```
echo 'Upper:Mid:Lower:<=> | ' . $upper . ':' . $mid . ':' .
  $lower . ':' . ($item <=> $binary[$mid]);
```

下面是输出结果。请注意观察，在找到我们想要的客户记录前，最底端边界、中间点和最顶端边界是怎样被调整的：



扩展

要详细了解二分查找算法，请参阅维基百科网站中一篇介绍该算法基础数学原理的文章（https://en.wikipedia.org/wiki/Binary_search_algorithm）。

实现搜索引擎

要实现搜索引擎，需要预先设置在搜索操作中使用的多个字段。此外，目标条目可能是通过多个字段中的某一个字段找到的，并且用户几乎不会为获得精确匹配结果而提供足够的信息——意识到这些情况很重要。因此，我们将在很大程度上依靠 SQL 语言中的 LIKE %value%子句。

具体处理过程

1. 先定义用于存储搜索条件的基类。该对象应含有 3 个属性：键（最终会用于代表数据库中的字段）、操作符（LIKE、<、>等）和一个可选的条目。条目属性为可选的原因是，某些操作符（如 IS NOT NULL）不要求必须带具体数值：

```
namespace Application\Database\Search;
class Criteria
{
  public $key;
  public $item;
  public $operator;
  public function __construct($key, $operator, $item = NULL)
  {
    $this->key = $key;
    $this->operator = $operator;
    $this->item = $item;
  }
}
```

2. 定义 Application\Database\Search\Engine 类，以及必要的类常量和属性。$columns 变量和$mapping 变量之间的区别是，存储在$columns 变量中的信息最终会在 HTML 代码的 SELECT 多项选择列表（或等价的页面元素）中显示。为安全起见，不应暴露数据库字段的真实名称，因此需要创建另一个数组（$mapping）：

```
namespace Application\Database\Search;
use PDO;
use Application\Database\Connection;
class Engine
{
  const ERROR_PREPARE = 'ERROR: unable to prepare statement';
  const ERROR_EXECUTE = 'ERROR: unable to execute statement';
  const ERROR_COLUMN = 'ERROR: column name not on list';
  const ERROR_OPERATOR= 'ERROR: operator not on list';
  const ERROR_INVALID = 'ERROR: invalid search criteria';

  protected $connection;
  protected $table;
  protected $columns;
  protected $mapping;
```

```
  protected $statement;
  protected $sql = ' ';
```

3. 定义一组我们需要使用的操作符。这些键用于代表实际的 SQL 命令。这些值会被显示在表单中：

```
protected $operators = [
    'LIKE'     => 'Equals',
    '<'        => 'Less Than',
    '>'        => 'Greater Than',
    '<>'       => 'Not Equals',
    'NOT NULL' => 'Exists',
];
```

4. 下面的构造器将用于连接数据库的实例接收为参数。为了连接数据库，我们将使用 Application\Database\Connection 类（请参阅第 5 章）。我们还需要提供数据库中表的名称和$columns 数组（$columns 数组会存储用于处理字段的键和标签，这些信息将在 HTML 表单中显示）。为安全起见还需要使用$mapping 数组，$mapping 数组中的键与$columns 数组中的键相同，但$mapping 数组中的值是数据库中真实的字段名称：

```
public function __construct(Connection $connection,
                            $table, array $columns, array $mapping)
{
  $this->connection = $connection;
  $this->setTable($table);
  $this->setColumns($columns);
  $this->setMapping($mapping);
}
```

5. 编写好上面的构造器后，应编写一系列有用的读取器和设置器：

```
public function setColumns($columns)
{
  $this->columns = $columns;
}
public function getColumns()
{
  return $this->columns;
}
// 依此类推
```

6. 这个类中最关键的方法很可能是用于创建预备执行的 SQL 语句的方法。对 SQL 语言中的 SELECT 语句进行初始配置后，可为其添加 WHERE 子句，以便使用$mapping

数组添加真正的数据库字段名称。然后可添加操作符并实现 switch() 语句，根据不同的操作符，switch() 语句可能会添加代表被搜索条目的已命名占位符，也可能不会添加代表被搜索条目的已命名占位符：

```
public function prepareStatement(Criteria $criteria)
{
  $this->sql = 'SELECT * FROM ' . $this->table . ' WHERE ';
  $this->sql .= $this->mapping[$criteria->key] . ' ';
  switch ($criteria->operator) {
    case 'NOT NULL' :
      $this->sql .= ' IS NOT NULL OR ';
      break;
    default :
      $this->sql .= $criteria->operator . '  :'
         . $this->mapping[$criteria->key] . ' OR ';
}
```

7. 定义了核心的 SELECT 语句后，可去除所有尾部的 OR 关键字，并添加一个子句，以便根据执行搜索操作的字段的值对查询结果进行排序。这样该 SELECT 语句就会被发送到数据库预备执行：

```
  $this->sql = substr($this->sql, 0, -4)
    . ' ORDER BY ' . $this->mapping[$criteria->key];
  $statement = $this->connection->pdo->prepare($this->sql);
  return $statement;
}
```

8. 现在编写最重要的 search() 方法（用于执行搜索操作）。应使该方法将 Application\Database\Search\Criteria 对象接收为参数。这至少可以确保我们拥有代表条目的键和操作符。为安全起见，还应添加一条用于检查这些属性的 if() 语句：

```
public function search(Criteria $criteria)
{
  if (empty($criteria->key) || empty($criteria->operator)) {
    yield ['error' => self::ERROR_INVALID];
    return FALSE;
  }
```

9. 在使用 try / catch 代码块捕捉异常的前提下，调用 prepareStatement() 方法：

```
try {
     if (!$statement = $this->prepareStatement($criteria)) {
```

```
            yield ['error' => self::ERROR_PREPARE];
            return FALSE;
    }
```

10. 为 execute()方法创建一个参数数组，该数组中的键代表在准备语句中被用作占位符的数据库字段名称。注意，我们没有使用=，而是使用了 LIKE % value % 结构：

```
    $params = array();
    switch ($criteria->operator) {
      case 'NOT NULL' :
        // 此处不执行任何操作：需要执行的操作已经被包含在语句中
        break;
        case 'LIKE' :
        $params[$this->mapping[$criteria->key]] =
        '%' . $criteria->item . '%';
        break;
        default :
        $params[$this->mapping[$criteria->key]] =
        $criteria->item;
    }
```

11. 执行这条查询语句，并使用 yield 关键字返回结果，这可以高效地将 search()方法变为一个生成器：

```
        $statement->execute($params);
        while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
          yield $row;
        }
      } catch (Throwable $e) {
        error_log(__METHOD__ . ':' . $e->getMessage());
        throw new Exception(self::ERROR_EXECUTE);
      }
      return TRUE;
    }
```

具体运行情况

将前面介绍的代码添加到 Application\Database\Search 文件夹中的 Criteria.php 和 Engine.php 文件中。定义调用脚本 chap_10_search_engine.php，为其设置类自动加载功能，可利用第 5 章介绍的 Application\Database\

Connection 类和第 6 章介绍的表单元素类：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');

use Application\Database\Connection;
use Application\Database\Search\ { Engine, Criteria };
use Application\Form\Generic;
use Application\Form\Element\Select;
```

定义在表单中显示哪些数据库字段，以及与这些字段对应的映射文件：

```
$dbCols = [
  'cname' => 'Customer Name',
  'cbal' => 'Account Balance',
  'cmail' => 'Email Address',
  'clevel' => 'Level'
];

$mapping = [
  'cname' => 'name',
  'cbal' => 'balance',
  'cmail' => 'email',
  'clevel' => 'level'
];
```

配置数据库连接并创建代表搜索引擎的实例：

```
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
$engine = new Engine($conn, 'customer', $dbCols, $mapping);
```

为了以适当方式显示 SELECT 多项选择列表元素，应根据 Application\Form* 类定义封装器和 SELECT 多项选择列表元素：

```
$wrappers = [
  Generic::INPUT => ['type' => 'td', 'class' => 'content'],
  Generic::LABEL => ['type' => 'th', 'class' => 'label'],
  Generic::ERRORS => ['type' => 'td', 'class' => 'error']
];

// 定义 SELECT 多项选择列表元素
$fieldElement = new Select('field',
                Generic::TYPE_SELECT,
                'Field',
                $wrappers,
                ['id' => 'field']);
```

```
$opsElement = new Select('ops',
Generic::TYPE_SELECT,
'Operators',
$wrappers,
['id' => 'ops']);
$itemElement = new Generic('item',
Generic::TYPE_TEXT,
'Searching For ...',
$wrappers,
['id' => 'item','title' => 'If more than one item,
separate with commas']);
$submitElement = new Generic('submit',
Generic::TYPE_SUBMIT,
'Search',
$wrappers,
['id' => 'submit','title' => 'Click to Search',
'value' => 'Search']);
```

获取输入参数（在定义了这些参数的情况下），设置表单元素选项，创建搜索规则并调用 search()方法：

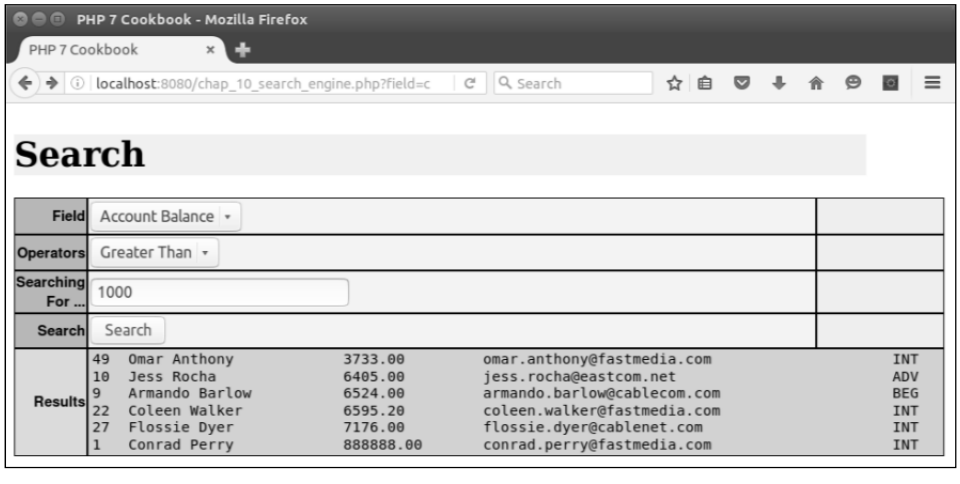
```
$key = (isset($_GET['field']))
? strip_tags($_GET['field']) : NULL;
$op     = (isset($_GET['ops'])) ? $_GET['ops'] : NULL;
$item   = (isset($_GET['item'])) ? strip_tags($_GET['item']) : NULL;
$fieldElement->setOptions($dbCols, $key);
$itemElement->setSingleAttribute('value', $item);
$opsElement->setOptions($engine->getOperators(), $op);
$criteria = new Criteria($key, $op, $item);
$results = $engine->search($criteria);
?>
```

显示逻辑主要用于显示表单。第 6 章详细介绍了这方面的内容，此处我们仅列出了核心逻辑：

```
<form name="search" method="get">
<table class="display" cellspacing="0" width="100%">
  <tr><?= $fieldElement->render(); ?></tr>
  <tr><?= $opsElement->render(); ?></tr>
  <tr><?= $itemElement->render(); ?></tr>
  <tr><?= $submitElement->render(); ?></tr>
  <tr>
  <th class="label">Results</th>
```

```
    <td class="content" colspan=2>
    <span style="font-size: 10pt;font-family:monospace;">
    <table>
    <?php foreach ($results as $row) : ?>
      <tr>
        <td><?= $row['id'] ?></td>
        <td><?= $row['name'] ?></td>
        <td><?= $row['balance'] ?></td>
        <td><?= $row['email'] ?></td>
        <td><?= $row['level'] ?></td>
      </tr>
    <?php endforeach; ?>
    </table>
    </span>
    </td>
  </tr>
</table>
</form>
```

下面是该程序在浏览器中显示的结果：



显示多维数组和累加合计

怎样以适当方式显示多维数组中的数据，是所有网页开发者都需要面对的常见问题。例如，有时需要显示一个含有客户及其购买的商品的列表。在处理每位客户的记录时，

需要显示他们的姓名、电话号码、账户余额等字段。可使用二维数组代表这种结构，其中 *x* 轴代表客户记录，*y* 轴代表每条客户记录的各个字段。添加了客户所购商品的信息后，就为这种数据结构添加了第三个轴！怎样在 2D 屏幕上显示 3D 模型呢？一种解决方案是将 hidden 标签属性与简单的 JavaScript 可见性触发器组合使用。

具体处理过程

1. 先通过带有多个 JOIN 子句的 SQL 语句生成一个 3D（三维）数组。使用第 1 章介绍的 Application/Database/Connection 类生成适当的 SQL 查询命令。保持参数 min 和 max 的开放状态，以便支持分页功能。令人遗憾的是，在本例中无法使用简单的 LIMIT 和 OFFSET 子句，因为记录的数量取决于客户所购商品的数量。通过限定客户 ID 的数量，可大致限定记录的数量。为了使该方式起作用，使用 ORDER BY 子句为客户 ID 排序：

```
define('ITEMS_PER_PAGE', 6);
define('SUBROWS_PER_PAGE', 6);
define('DB_CONFIG_FILE', '/../config/db.config.php');
include __DIR__ . '/../Application/Database/Connection.php';
use Application\Database\Connection;
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
$sql  = 'SELECT c.id,c.name,c.balance,c.email,f.phone, '
  . 'u.transaction,u.date,u.quantity,u.sale_price,r.title '
  . 'FROM customer AS c '
  . 'JOIN profile AS f '
  . 'ON f.id = c.id '
  . 'JOIN purchases AS u '
  . 'ON u.customer_id = c.id '
  . 'JOIN products AS r '
  . 'ON u.product_id = r.id '
  . 'WHERE c.id >= :min AND c.id < :max '
  . 'ORDER BY c.id ASC, u.date DESC ';
```

2. 使用简单的$_GET 参数来限定客户 ID，实现分页式表单。注意，应添加额外的检查语句，以确保$prev 变量的值不小于 0。你还应该考虑添加另一个控制功能，以确保$next 变量的值不会超过最后一条记录的客户 ID。本例仅为$next 变量设置了累加功能：

```
$page = $_GET['page'] ?? 1;
$page = (int) $page;
```

```
$next = $page + 1;
$prev = $page - 1;
$prev = ($prev >= 0) ? $prev : 0;
```

3. 计算$min 和$max 变量的值，并准备和执行 SQL 语句：

```
$min  = $prev * ITEMS_PER_PAGE;
$max  = $page * ITEMS_PER_PAGE;
$stmt = $conn->pdo->prepare($sql);
$stmt->execute(['min' => $min, 'max' => $max]);
```

4. 可使用 while()循环获取查询结果。我们使用简单的 PDO::FETCH_ASSOC 获取数据模式，实现本示例的目标。我们将客户的 ID 用作键，将客户的基本信息存储为数组参数，然后将客户所购商品信息存储在子数组$results[$key]['purchases'][]中。当客户 ID 改变时，应为下一条客户记录存储相同的信息。注意，我们将每位客户消费额的合计存储为一组代表合计的键：

```
  $custId = 0;
  $result = array();
  $grandTotal = 0.0;
  while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    if ($row['id'] != $custId) {
      $custId = $row['id'];
      $result[$custId] = [
      'name'     => $row['name'],
      'balance'  => $row['balance'],
      'email'    => $row['email'],
      'phone'    => $row['phone'],
    ];
    $result[$custId]['total'] = 0;
  }
  $result[$custId]['purchases'][] = [
    'transaction'   => $row['transaction'],
    'date'          => $row['date'],
    'quantity'      => $row['quantity'],
    'sale_price'    => $row['sale_price'],
    'title'         => $row['title'],
  ];
  $result[$custId]['total'] += $row['sale_price'];
  $grandTotal += $row['sale_price'];
}
?>
```

5. 接下来实现显示逻辑。先编写用于显示客户主要信息的代码：

```
<div class="container">
<?php foreach ($result as $key => $data) : ?>
<div class="mainLeft color0">
     <?= $data['name'] ?> [<?= $key ?>]
</div>
<div class="mainRight">
  <div class="row">
    <div class="left">Balance</div>
           <div class="right"><?= $data['balance']; ?></div>
  </div>
  <div class="row">
    <div class="left color2">Email</div>
           <div class="right"><?= $data['email']; ?></div>
  </div>
  <div class="row">
    <div class="left">Phone</div>
           <div class="right"><?= $data['phone']; ?></div>
    </div>
  <div class="row">
        <div class="left color2">Total Purchases</div>
    <div class="right">
<?= number_format($data['total'],2); ?>
</div>
  </div>
```

6. 再编写用于显示客户所购商品的列表的逻辑：

```
<!-- 购物信息 -->
<table>
  <tr>
  <th>Transaction</th><th>Date</th><th>Qty</th>
   <th>Price</th><th>Product</th>
  </tr>
  <?php $count = 0; ?>
  <?php foreach ($data['purchases'] as $purchase) : ?>
  <?php $class = ($count++ & 01) ? 'color1' : 'color2'; ?>
  <tr>
  <td class="<?= $class ?>"><?= $purchase['transaction'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['date'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['quantity'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['sale_price'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['title'] ?></td>
  </tr>
  <?php endforeach; ?>
</table>
```

7. 为了进行分页，应添加用于切换到上一页（*Previous*）和下一页（*Next*）的按钮：

```
<?php endforeach; ?>
<div class="container">
  <a href="?page=<?= $prev ?>">
        <input type="button" value="Previous"></a>
  <a href="?page=<?= $next ?>">
        <input type="button" value="Next" class="buttonRight"></a>
</div>
<div class="clearRow"></div>
</div>
```

8. 到目前为止，我们还没有得到整理好的查询结果！因此我们添加一个简单的JavaScript 函数，以便根据 标签的 id 属性切换 标签的可见或隐藏状态：

```
<script type="text/javascript">
function showOrHide(id) {
  var div = document.getElementById(id);
  div.style.display = div.style.display == "none" ?
    "block" : "none";
}
</script>
```

9. 将用于在 HTML 中显示购物信息的表封装在初始可见状态为隐藏的 标签中。然后，可设置初始时可见的行数，并添加一个用于显示其余购物信息的链接：

```
<div class="row" id="<?= 'purchase' . $key ?>"
style="display:none;">
  <table>
    <tr>
      <th>Transaction</th><th>Date</th><th>Qty</th>
                <th>Price</th><th>Product</th>
    </tr>
  <?php $count = 0; ?>
  <?php $first = TRUE; ?>
  <?php foreach ($data['purchases'] as $purchase) : ?>
    <?php if ($count > SUBROWS_PER_PAGE && $first) : ?>
    <?php $first = FALSE; ?>
    <?php $subId = 'subrow' . $key; ?>
    </table>
    <a href="#" onClick="showOrHide('<?= $subId ?>')">More</a>
    <div id="<?= $subId ?>" style="display:none;">
    <table>
```

```
    <?php endif; ?>
  <?php $class = ($count++ & 01) ? 'color1' : 'color2'; ?>
  <tr>
  <td class="<?= $class ?>"><?= $purchase['transaction'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['date'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['quantity'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['sale_price'] ?></td>
  <td class="<?= $class ?>"><?= $purchase['title'] ?></td>
  </tr>
  <?php endforeach; ?>
  </table>
  <?php if (!$first) : ?></div><?php endif; ?>
</div>
```

10. 添加一个按钮，通过单击该按钮可以显示隐藏的 标签：

```
<input type="button" value="Purchases" class="buttonRight"
    onClick="showOrHide('<?= 'purchase' . $key ?>')">
```

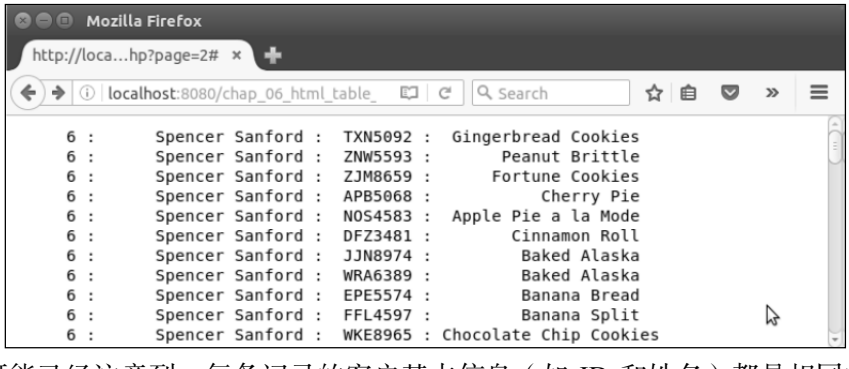
具体运行情况

将步骤 1 至步骤 5 中介绍的代码添加到 chap_10_html_table_multi_array_hidden.php 文件中。

在 while()循环中添加下列代码：

```
printf('%6s : %20s : %8s : %20s' . PHP_EOL,
    $row['id'], $row['name'], $row['transaction'], $row['title']);
```

在 while()循环后面添加 exit 命令，下面是输出结果：



你可能已经注意到，每条记录的客户基本信息（如 ID 和姓名）都是相同的，但购物信息（如交易编号和商品名称，即后两个字段的值）是不同的。继续处理该程序，删除 printf()语句。

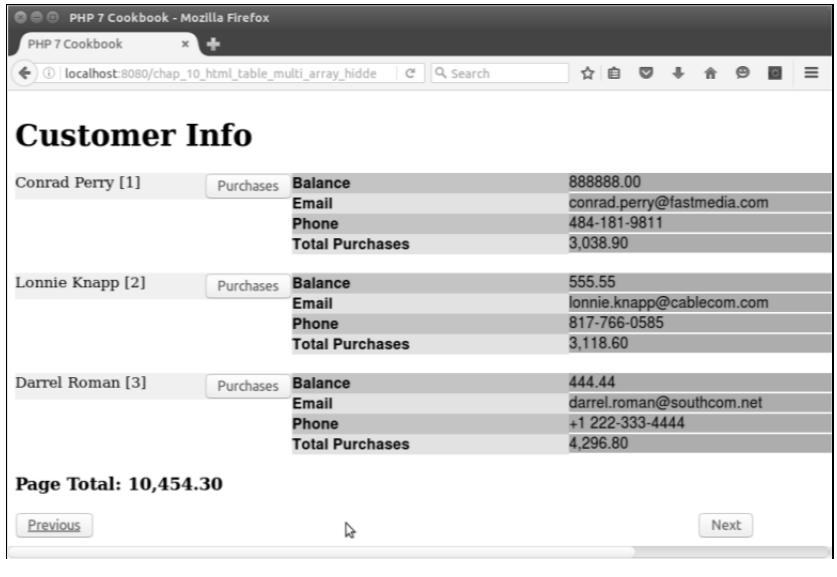
使用下面的代码替换 `exit` 命令：

```
echo '<pre>', var_dump($result), '</pre>'; exit;
```

下面是这个合成的 3D 数组：

```
array(6) {
  [6]=>
  array(5) {
    ["name"]=>
    string(15) "Spencer Sanford"
    ["balance"]=>
    string(5) "99.99"
    ["email"]=>
    string(28) "spencer.sanford@cablenet.net"
    ["phone"]=>
    string(12) "451-815-7386"
    ["purchases"]=>
    array(92) {
      [0]=>
      array(5) {
        ["transaction"]=>
        string(7) "TXN5092"
        ["date"]=>
        string(19) "2016-09-12 05:46:16"
        ["quantity"]=>
        string(2) "44"
        ["sale_price"]=>
        string(5) "10.50"
        ["title"]=>
        string(19) "Gingerbread Cookies"
      }
      [1]=>
      array(5) {
        ["transaction"]=>
        string(7) "ZNW5593"
        ["date"]=>
        string(19) "2015-09-18 03:58:26"
```

现在可以添加步骤 5 至步骤 7 介绍的显示逻辑。如前所述，尽管现在的设置是显示所有数据，但隐藏的数据不会被显示出来。继续处理该程序并将其余步骤介绍的代码也添加进来。下面是该程序初始的输出结果：



当用户单击了 Purchases 按钮后，初始的购物信息会显示出来。如果用户单击 More 链接，那么其余购物信息也会显示出来：

Balance	888888.00
Email	conrad.perry@fastmedia.com
Phone	484-181-9811
Total Purchases	3,038.90

Transaction	Date	Qty	Price	Product
OJM3659	2016-10-27 03:27:28	2	8.40	Peanut Butter Cups
RLI3437	2016-08-10 14:16:22	17	10.20	Neapolitan Ice Cream
EOV1192	2016-05-15 08:56:28	3	17.10	Chocolate Eclair
ZAM6290	2015-10-21 23:11:22	18	72.00	Ice Cream Cake
ZNW5593	2015-09-18 03:58:26	15	75.00	Pumpkin Ice Cream

More

第 11 章　实现多种软件设计模式

本章包括以下要点：

- 创建数组至对象水合器（array to object hydrator）
- 创建对象至数组水合器（object to array hydrator）
- 实现策略模式
- 定义映射器
- 实现对象关联映射功能
- 实现发布/订阅设计模式

本章主要内容简介

将多种**软件设计模式**融入**面向对象程序设计**（OOP）中的思想，是由软件设计领域的四位世界顶级大师（Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides）在他们合著的具有开创性的著作 *Design Patterns: Elements of Reusable Object-Oriented Software*（1994 年出版）中首次提出的。这本书既没有定义标准也没有定义协议，而是介绍了经过多年实践验证的多种常见软件设计模式。这本书介绍的软件设计模式分为 3 类：创建型、结构型和行为型。

本书已经介绍了这些软件设计模式的示例。下表列出了具体章节：

软件设计模式	章节	示例
单例模式	第 2 章	定义可见性
工程模式	第 6 章	实现表单工厂
适配器模式	第 8 章	在不使用 `gettext()` 工具集的情况下处理翻译工作
代理模式	第 7 章	创建简单的 REST 客户端 创建简单的 SOAP 客户端
迭代器模式	第 2 章 第 3 章	递归目录迭代器 使用迭代器

本章将介绍多种软件设计模式，并着重介绍并发和架构模式。

创建数组至对象水合器（array to object hydrator）

水合器模式是**数据传输对象**（Data Transfer Object，DTO）设计模式中的一个分支。其设计原则非常简单：将数据从一个地方移动到另一个地方。为了了解具体处理过程，我们会定义将数据从数组移动到对象中的类。

具体处理过程

1. 定义代表水合器（Hydrator）的类（Application\Generic\Hydrator\GetSet），使该类能够使用读取器和设置器：

```
namespace Application\Generic\Hydrator;
class GetSet
{
  // 此处添加具体代码
}
```

2. 定义 hydrate()方法，它能够将数组和对象作为参数接收。然后该方法会通过调用对象中的 setXXX()方法将存储在数组中的值赋予对象。使用 get_class()函数查明对象所属的类，使用 get_class_methods()函数获取该类含有的所有方法。使用 preg_match()函数匹配方法的前缀和后缀，之后方法的后缀会被设定为数组的键：

```
public static function hydrate(array $array, $object)
{
  $class = get_class($object);
  $methodList = get_class_methods($class);
  foreach ($methodList as $method) {
    preg_match('/^(set)(.*?)$/i', $method, $matches);
    $prefix = $matches[1] ?? '';
    $key = $matches[2] ?? '';
    $key = strtolower(substr($key, 0, 1)) . substr($key, 1);
    if ($prefix == 'set' && !empty($array[$key])) {
      $object->$method($array[$key]);
    }
  }
  return $object;
}
```

具体运行情况

为了做这个实验，可使用前面“具体处理情况”中介绍的代码先定义 Application\Generic\Hydrator\GetSet 类。然后定义用于存储数组数据的类：Application\Entity\Person，并为之定义适当的属性和方法。不要忘记为所有属性都定义读取器和设置器。下面仅列出了该类的部分方法：

```
namespace Application\Entity;
class Person
{
   protected $firstName = ' ';
   protected $lastName = ' ';
   protected $address = ' ';
   protected $city = ' ';
   protected $stateProv = ' ';
   protected $postalCode = ' ';
   protected $country = ' ';

   public function getFirstName()
   {
     return $this->firstName;
   }

   public function setFirstName($firstName)
   {
     $this->firstName = $firstName;
   }

   // 依此类推
 }
```

现在可创建调用程序 chap_11_array_to_object.php，为其设置类自动加载功能，并引用适当的类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Entity\Person;
use Application\Generic\Hydrator\GetSet;
```

定义用于实验的数组，其中存储的值会被赋予新建的 Person 实例：

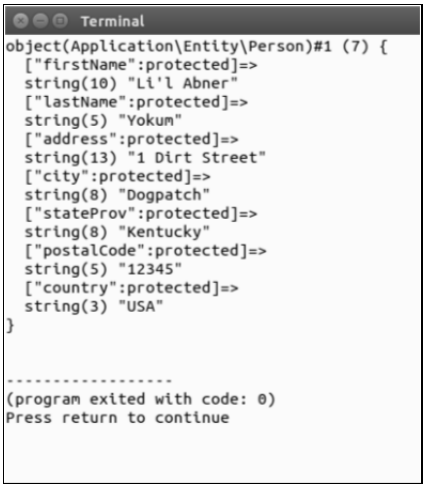
```
$a['firstName'] = 'Li\'l Abner';
$a['lastName']  = 'Yokum';
```

```
$a['address']   = '1 Dirt Street';
$a['city']      = 'Dogpatch';
$a['stateProv'] = 'Kentucky';
$a['postalCode']= '12345';
$a['country']   = 'USA';
```

通过静态方式调用 hydrate() 和 extract() 方法：

```
$b = GetSet::hydrate($a, new Person());
var_dump($b);
```

下面是该程序的输出结果：



创建对象至数组水合器（object to array hydrator）

本节介绍创建数组至对象水合器的相反过程。在处理过程中，我们需要通过对象属性获取值，并返回关联数组，该数组中的键应该是字段的名称。

具体处理过程

1. 本例会在上一示例介绍的 Application\Generic\Hydrator\GetSet 类的基础上来演示：

```
namespace Application\Generic\Hydrator;
class GetSet
{
  // 此处添加具体代码
}
```

2. 在上一示例定义的 hydrate()方法后面定义 extract()方法，它会将对象接收为参数。extract()方法的逻辑与 hydrate()方法的逻辑类似，只是 extract()方法要搜索的是 get×××()方法。preg_match()函数还是被用来匹配方法的前缀和后缀，将来方法的后缀会被设定为数组的键：

```
public static function extract($object)
{
  $array = array();
  $class = get_class($object);
  $methodList = get_class_methods($class);
  foreach ($methodList as $method) {
     preg_match('/^(get)(.*?)$/i', $method, $matches);
     $prefix = $matches[1] ?? '';
     $key    = $matches[2] ?? '';
     $key    = strtolower(substr($key, 0, 1)) . substr($key, 1);
     if ($prefix == 'get') {
       $array[$key] = $object->$method();
     }
   }
   return $array;
 }
 }
```

为简便起见，我们将 hydrate()和 extract()定义为静态方法。

具体运行情况

定义调用程序 chap_11_object_to_array.php，为其设置类自动加载功能，并引用合适的类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Entity\Person;
use Application\Generic\Hydrator\GetSet;
```

定义 Person 实例，为它的属性赋值：

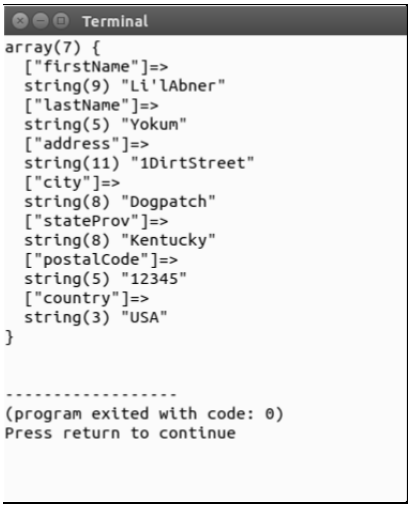
```
$obj = new Person();
```

```
$obj->setFirstName('Li\'lAbner');
$obj->setLastName('Yokum');
$obj->setAddress('1DirtStreet');
$obj->setCity('Dogpatch');
$obj->setStateProv('Kentucky');
$obj->setPostalCode('12345');
$obj->setCountry('USA');
```

以静态方式调用新定义的 `extract()`方法：

```
$a = GetSet::extract($obj);
var_dump($a);
```

下面是该程序的输出结果：



实现策略模式

程序的运行环境迫使开发者为完成同一任务而开发出多个解决方案，是经常出现的情况。传统上，这会用到大量的 `if/elseif/else` 代码块。不是在 `if` 语句中定义较大的逻辑块，就是通过一系列方法或函数实现这些解决方案。策略模式通过使用主类封装一系列代表解决同一问题的多种解决方案的子类，将该处理过程模式化。

具体处理过程

1. 本例使用前面介绍的水合器类 `GetSet` 代表策略。定义一个起主要作用的类

Application\Generic\Hydrator\Any，使用该类处理 Application\Generic\Hydrator\Strategy 命名空间中代表策略的对象，这些对象包括 GetSet、PublicProps 和 Extending。

2. 先定义代表可用策略的类常量：

```
namespace Application\Generic\Hydrator;
use InvalidArgumentException;
use Application\Generic\Hydrator\Strategy\ {
GetSet, PublicProps, Extending };
class Any
{
  const STRATEGY_PUBLIC     = 'PublicProps';
  const STRATEGY_GET_SET    = 'GetSet';
  const STRATEGY_EXTEND     = 'Extending';
  protected $strategies;
  public $chosen;
```

3. 定义构造器，使用该方法将所有策略添加到$strategies 属性中：

```
public function __construct()
{
  $this->strategies[self::STRATEGY_GET_SET] = new GetSet();
  $this->strategies[self::STRATEGY_PUBLIC] = new PublicProps();
  $this->strategies[self::STRATEGY_EXTEND] = new Extending();
}
```

4. 定义 addStrategy()方法,使我们能够在不修改现存策略类的代码的情况下，重写策略或添加新策略：

```
public function addStrategy($key, HydratorInterface $strategy)
{
  $this->strategies[$key] = $strategy;
}
```

5. 修改 hydrate()和 extract()方法，使它们仅调用被选中的方法：

```
public function hydrate(array $array, $object)
{
  $strategy = $this->chooseStrategy($object);
  $this->chosen = get_class($strategy);
  return $strategy::hydrate($array, $object);
}

public function extract($object)
{
  $strategy = $this->chooseStrategy($object);
```

```
    $this->chosen = get_class($strategy);
    return $strategy::extract($object);
  }
```

6. 有点难度的地方是需要判断出哪个策略被选中了。要查明哪个策略被选中了，可定义 chooseStrategy() 方法，使该方法将对象接收为参数。可先通过获取类中含有的方法执行检查工作。通过扫描这些方法，查明其中是否包含 getXXX() 或 setXXX() 方法。如果其中含有这些方法，就将水合器类 GetSet 选作策略：

```
public function chooseStrategy($object)
{
  $strategy = NULL;
  $methodList = get_class_methods(get_class($object));
  if (!empty($methodList) && is_array($methodList)) {
       $getSet = FALSE;
       foreach ($methodList as $method) {
         if (preg_match('/^get|set.*$/i', $method)) {
              $strategy = $this->strategies[self::STRATEGY_GET_SET];
        break;
       }
     }
  }
```

7. 继续编写 chooseStrategy() 方法，如果经过检查后发现策略对象中既没有读取器也没有设置器，就应该使用 get_class_vars() 方法查明策略对象中是否含有可用的属性。如果策略对象中含有可用的属性，就将 PublicProps 选作水合器类：

```
if (!$strategy) {
     $vars = get_class_vars(get_class($object));
     if (!empty($vars) && count($vars)) {
          $strategy = $this->strategies[self::STRATEGY_PUBLIC];
     }
 }
```

8. 如果前面的判断条件都不成立，就将 Extending 选作水合器类，Extending 类会返回仅通过扩展策略对象的类获得的新类，因此能够使所有公用（public）或受保护（protected）的属性可用：

```
if (!$strategy) {
     $strategy = $this->strategies[self::STRATEGY_EXTEND];
}
return $strategy;
}
}
```

9. 现在让我们处理代表策略的类。先定义一个新的命名空间：Application\Generic\Hydrator\Strategy。

10. 在新建的命名空间中定义一个接口，以便使我们能够区分出哪些类是由Application\Generic\Hydrator\Any类处理的策略类：

```
namespace Application\Generic\Hydrator\Strategy;
interface HydratorInterface
{
  public static function hydrate(array $array, $object);
  public static function extract($object);
}
```

11. 在前面两个示例介绍的GetSet水合器类中添加新定义的接口：

```
namespace Application\Generic\Hydrator\Strategy;
class GetSet implements HydratorInterface
{

  public static function hydrate(array $array, $object)
  {
    // 要了解具体代码，
    // 请参阅“创建数组至对象水合器”示例
  }

  public static function extract($object)
  {
    // 要了解具体代码，
    // 请参阅“创建对象至数组水合器”示例
  }
}
```

12. 下面的水合器类仅用于读写公用属性：

```
namespace Application\Generic\Hydrator\Strategy;
class PublicProps implements HydratorInterface
{
  public static function hydrate(array $array, $object)
  {
    $propertyList= array_keys(
      get_class_vars(get_class($object)));
    foreach ($propertyList as $property) {
      $object->$property = $array[$property] ?? NULL;
    }
    return $object;
```

```
  }

  public static function extract($object)
  {
    $array = array();
    $propertyList = array_keys(
      get_class_vars(get_class($object)));
    foreach ($propertyList as $property) {
      $array[$property] = $object->$property;
    }
    return $array;
  }
}
```

13. Extending 是一个具有多种功能的水合器，使用它可以扩展代表策略的类，从而能够直接访问属性。我们可以进一步定义魔术读取器和设置器方法，以便提供访问属性的功能。

14. 处理 hydrate()方法的难度最高，因为我们假定的前提条件是没有定义读取器和设置器，也没有定义可见等级为 public 的属性。因此，我们需要定义一个类，使用该类扩展要进行水合处理（即数组与对象间的数据转换）的对象所属的类。可先定义用作新类创建模板的字符串：

```
namespace Application\Generic\Hydrator\Strategy;
class Extending implements HydratorInterface
{
  const UNDEFINED_PREFIX = 'undefined';
  const TEMP_PREFIX = 'TEMP_';
  const ERROR_EVAL = 'ERROR: unable to evaluate object';
  public static function hydrate(array $array, $object)
  {
    $className  = get_class($object);
    $components = explode('\\', $className);
    $realClass  = array_pop($components);
    $nameSpace  = implode('\\', $components);
    $tempClass  = $realClass . self::TEMP_SUFFIX;
    $template   = 'namespace '
      . $nameSpace . '{'
      . 'class ' . $tempClass
      . ' extends ' . $realClass . '  '
```

15. 继续编写 hydrate()方法，定义$values 属性和一个构造器，使该构造器将通

过水合处理转换为对象的数组接收为参数。循环遍历该数组的值，将每个数组元素中存储的值赋予$values 属性。还应定义 getArrayCopy()方法，使用该方法在有需要时返回$values 属性的值。定义魔术方法__get()方法，以便模拟直接访问属性的操作：

```
.'{ '
.'   protected $values; '
.'   public function __construct($array) '
.'   { $this->values = $array; '
.'     foreach ($array as $key => $value) '
.'          $this->$key = $value; '
.'   } '
.'   public function getArrayCopy() '
.'   { return $this->values; } '
```

16. 为了更方便地访问属性，可定义魔术方法__get()，该方法可以像访问公用变量那样直接访问属性：

```
.' public function __get($key) '
.' { return $this->values[$key] ?? NULL; } '
```

17. 在用于定义新类的模板中，定义魔术方法__call()，使用该方法模拟读取器和设置器：

```
.' public function __call($method, $params) '
.' { '
.'    preg_match("/^(get|set)(.*?)$/i", '
.'        $method, $matches); '
.'    $prefix    = $matches[1] ?? ""; '
.'    $key       = $matches[2] ?? ""; '
.'    $key       = strtolower(substr($key, 0, 1)) '
.'           substr($key, 1); '
.'    if ($prefix == "get") { '
.'        return $this->values[$key] ?? NULL; '
.'    } else { '
.'       $this->values[$key] = $params[0]; '
.'    } '
.'  } '
.' } '
.' } // 结束命名空间 ' . PHP_EOL
```

18. 在创建新类的模板中，在全局命名空间中添加一个函数，创建并返回新建类的实例：

```
.'namespace { '
.'function build($array) '
```

```
.'{ return new ' . $nameSpace . '\\'
.    $tempClass . '($array); } '
.'} // 结束全局命名空间 '
. PHP_EOL;
```

19. 继续编写 hydrate()方法，使用 eval()函数处理已经编写好的新建类模板。然后调用在新建类模板的末尾定义的 build()方法。注意，如果不能确定将新建的类添加到哪个命名空间中，就应该在全局命名空间中定义创建新类的模板，并通过全局命名空间调用 build()方法：

```
try {
     eval($template);
} catch (ParseError $e) {
     error_log(__METHOD__ . ':' . $e->getMessage());
     throw new Exception(self::ERROR_EVAL);
}
return \build($array);
}
```

20. 定义 extract()方法要简单得多，因为我们选择提取的数据极为有限。先扩展一个类再使用魔术方法为该类的实例赋值，都是较容易的操作。但反向的操作就不那么容易。如果先为对象赋值后再扩展类，就会丢失所有属性的值，因为我们扩展的是类而不是对象。因此，我们唯一的选择是组合使用读取器和公用属性：

```
public static function extract($object)
{
    $array = array();
    $class = get_class($object);
    $methodList = get_class_methods($class);
    foreach ($methodList as $method) {
      preg_match('/^(get)(.*?)$/i', $method, $matches);
    $prefix  = $matches[1] ?? '';
    $key     = $matches[2] ?? '';
    $key     = strtolower(substr($key, 0, 1))
    . substr($key, 1);
    if ($prefix == 'get') {
        $array[$key] = $object->$method();
    }
  }
  $propertyList= array_keys(get_class_vars($class));
  foreach ($propertyList as $property) {
```

```
      $array[$property] = $object->$property;
    }
    return $array;
    }
}
```

具体运行情况

可先定义 3 个拥有相同属性（firstName、lastName 等）的类用于测试。第一个类 Person 应该含有受保护的属性以及读取器与设置器。第二个类 PublicPerson 应含有公用属性。第三个类 ProtectedPerson 应含有受保护的属性，但不含有读取器和设置器：

```
<?php
namespace Application\Entity;
class Person
{
  protected $firstName  = ' ';
  protected $lastName   = ' ';
  protected $address    = ' ';
  protected $city       = ' ';
  protected $stateProv  = ' ';
  protected $postalCode = ' ';
  protected $country    = ' ';

    public function getFirstName()
    {
      return $this->firstName;
    }

    public function setFirstName($firstName)
    {
      $this->firstName = $firstName;
    }

  // 不要忘记定义其余读取器和设置器

}

<?php
namespace Application\Entity;
class PublicPerson
```

```
{
  private $id = NULL;
  public $firstName     = ' ';
  public $lastName      = ' ';
  public $address       = ' ';
  public $city          = ' ';
  public $stateProv     = ' ';
  public $postalCode    = ' ';
  public $country       = ' ';
}

<?php
namespace Application\Entity;

class ProtectedPerson
{
  private $id = NULL;
  protected $firstName  = ' ';
  protected $lastName   = ' ';
  protected $address    = ' ';
  protected $city       = ' ';
  protected $stateProv  = ' ';
  protected $postalCode = ' ';
  protected $country    = ' ';
}
```

现在可定义调用程序 chap_11_strategy_pattern.php，为其设置类自动加载功能，并引用适当的类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Entity\ { Person, PublicPerson, ProtectedPerson };
use Application\Generic\Hydrator\Any;
use Application\Generic\Hydrator\Strategy\ { GetSet, Extending,
  PublicProps };
```

创建一个 Person 实例，通过调用设置器，定义 Person 实例的属性值：

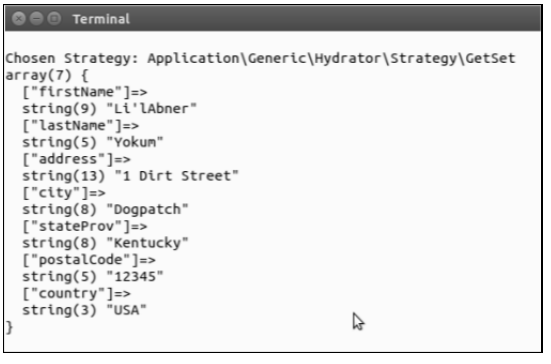
```
$obj = new Person();
$obj->setFirstName('Li\'lAbner');
$obj->setLastName('Yokum');
$obj->setAddress('1 Dirt Street');
$obj->setCity('Dogpatch');
$obj->setStateProv('Kentucky');
```

```
$obj->setPostalCode('12345');
$obj->setCountry('USA');
```

为水合器类 Any 创建一个实例，调用 extract()方法，并使用 var_dump()函数查看结果：

```
$hydrator = new Any();
$b = $hydrator->extract($obj);
echo "\nChosen Strategy: " . $hydrator->chosen . "\n";
var_dump($b);
```

如下面的输出结果所示，显然被选中的策略是 GetSet 类：



id 属性没有被设置，因为它的可见性等级为 private。

可使用相同的值定义一个数组。将新建的 PublicPerson 实例用作参数，调用 Any 实例中的 hydrate()方法：

```
$a = [
  'firstName'   => 'Li\'lAbner',
  'lastName'    => 'Yokum',
  'address'     => '1 Dirt Street',
  'city'        => 'Dogpatch',
  'stateProv'   => 'Kentucky',
  'postalCode'  => '12345',
  'country'     => 'USA'
];

$p = $hydrator->hydrate($a, new PublicPerson());
echo "\nChosen Strategy: " . $hydrator->chosen . "\n";
```

```
var_dump($p);
```

下面是运行结果。注意，本例选中的是 PublicProps 策略：

再次调用 hydrate() 方法，但这次将 ProtectedPerson 实例用作参数。然后通过调用 getFirstName() 和 getLastName() 方法测试魔术读取器。还应通过直接访问变量的方式访问 firstName 和 lastName 属性：

```
$q = $hydrator->hydrate($a, new ProtectedPerson());
echo "\nChosen Strategy: " . $hydrator->chosen . "\n";
echo "Name: {$q->getFirstName()} {$q->getLastName()}\n";
echo "Name: {$q->firstName} {$q->lastName}\n";
var_dump($q);
```

下面是最后一批输出结果，表明这次选中的策略是 Extending。注意，进行水合处理的实例所属的类是 ProtectedPerson_TEMP，而且该实例中受保护的属性都被赋值了：

定义映射器

映射器或**数据映射器**与水合器非常相似：都会将数据从一种类型（如数组或对象）转换为另一种类型。它们之间的主要区别是，水合器是通用的，而且无须预先在代码中添加对象中属性的名称。而映射器恰好相反：需要获得两种数据类型中精确的属性名称信息。本节介绍使用映射器将一个数据库表中的数据转换到另一个数据库表中的方式。

具体处理过程

1. 先定义 Application\Database\Mapper\FieldConfig 类，使用该类存储单个字段的映射关系说明。为该类定义适当的类常量：

```
namespace Application\Database\Mapper;
use InvalidArgumentException;
class FieldConfig
{
  const ERROR_SOURCE =
    'ERROR: need to specify destTable and/or source';
  const ERROR_DEST = 'ERROR: need to specify either '
    . 'both destTable and destCol or neither';
```

2. 定义类常量的同时也要定义主要属性。$key 变量用于识别对象，$source 变量代表源数据库表中的字段，$destTable 和$destCol 变量代表目标数据库中的表和其中的字段。如果$default 变量被定义了，那么它就会被用于存储默认值或能够生成适当值的回调函数：

```
public $key;
public $source;
public $destTable;
public $destCol;
public $default;
```

3. 创建构造器，它能够分配默认值、创建键，并查明$source、$destTable 和$destCol 变量是否已经定义：

```
public function __construct($source = NULL,
                            $destTable = NULL,
                            $destCol = NULL,
                            $default = NULL)
```

```
{
  // 通过$source、$destTable 和$destCol 变量生成键
  $this->key = $source . '.' . $destTable . '.' . $destCol;
  $this->source = $source;
  $this->destTable = $destTable;
  $this->destCol = $destCol;
  $this->default = $default;
  if (($destTable && !$destCol) ||
      (!$destTable && $destCol)) {
       throw new InvalidArgumentException(self::ERROR_DEST);
  }
  if (!$destTable && !$source) {
      throw new InvalidArgumentException(
        self::ERROR_SOURCE);
  }
}
```

我们允许源数据库表和目的数据库表中的字段为 NULL。这样做的原因是，可能出现源数据库表中字段在目的数据库表中没有对应位置的情况。同理，也可能出现目的数据库表中字段在目的数据库表中没有对应位置的情况。

4. 默认情况下，我们需要查明$default 变量存储的值是否为回调函数。如果该变量存储的值是回调函数，那么就应该调用该回调函数。否则，就应该返回$default 变量存储的值。注意，在定义该回调函数时，应使之能够将数据库表中的一条记录作为参数接收：

```
public function getDefault()
{
  if (is_callable($this->default)) {
      return call_user_func($this->default, $row);
  } else {
      return $this->default;
  }
}
```

5. 要圆满地完成这个类的编写工作，应为该类的 5 个属性都定义读取器和设置器：

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

```
}

public function setKey($key)
{
  $this->key = $key;
}

// 依此类推
```

6. 定义用于执行映射操作的类 Application\Database\Mapper\Mapping，使其将源和目的数据库表的名称以及一组 FieldConfig 对象接收为参数。稍后我们会将代表目的表的属性转换为数组，因为一个源表可能会与两个或多个目的表建立映射关系：

```
namespace Application\Database\Mapper;
class Mapping
{
  protected $sourceTable;
  protected $destTable;
  protected $fields;
  protected $sourceCols;
  protected $destCols;

  public function __construct(
    $sourceTable, $destTable, $fields = NULL)
  {
    $this->sourceTable = $sourceTable;
    $this->destTable = $destTable;
    $this->fields = $fields;
  }
```

7. 为 Mapping 类中的属性定义读取器和设置器：

```
public function getSourceTable()
{
  return $this->sourceTable;
}
public function setSourceTable($sourceTable)
{
  $this->sourceTable = $sourceTable;
}
// 依此类推
```

8. 为了配置字段，还需要为 Mapping 类编写添加单个字段的功能。无须单独将键用作参数，因为可以从 FieldConfig 实例中提取键：

```
public function addField(FieldConfig $field)
{
  $this->fields[$field->getKey()] = $field;
  return $this;
}
```

9. 获取源表中字段的名称是极为重要的步骤(使用 getSourceColumns() 方法)。其难点在于，源表字段名称是以属性的形式被存储在 FieldConfig 对象中的。因此，我们在调用 getSourceColumns() 方法时，应使该方法循环遍历一组 FieldConfig 对象，并调用这些对象中的 getSource() 方法，以便获取源表中字段的名称：

```
public function getSourceColumns()
{
  if (!$this->sourceCols) {
       $this->sourceCols = array();
       foreach ($this->getFields() as $field) {
            if (!empty($field->getSource())) {
                $this->sourceCols[$field->getKey()] =
                  $field->getSource();
            }
       }
  }
  return $this->sourceCols;
}
```

10. 可使用类似的方式编写 getDestColumns() 方法。与 getSourceColumns() 方法最大区别是，getDestColumns() 方法只需从指定的目的表获取字段的名称，在定义了多个目的表的情况中，这一点非常关键。我们无须使用 getDestColumns() 方法检查$destCol 变量的值是否已经被设置，因为这项工作已经由 FieldConfig 对象的构造器完成了：

```
public function getDestColumns($table)
{
  if (empty($this->destCols[$table])) {
       foreach ($this->getFields() as $field) {
            if ($field->getDestTable()) {
              if ($field->getDestTable() == $table) {
                   $this->destCols[$table][$field->getKey()] =
                   $field->getDestCol();
              }
            }
       }
```

```
    }
    return $this->destCols[$table];
  }
```

11. 定义一个方法，使该方法将代表源表中一条记录的数组作为其第一个参数接收，将目的表的名称作为其第二个参数接收。使该方法生成可插入目的表中的一组数据。

12. 我们需要为默认值（有可能是一个回调函数）和来自源表的数据设置被选用的优先等级。本例选择先测试默认值。如果返回的默认值为 NULL，那么来自源表的数据就会被采用。注意，如果要进行进一步的处理，就应该将默认值定义为回调函数：

```
  public function mapData($sourceData, $destTable)
  {
    $dest = array();
    foreach ($this->fields as $field) {
      if ($field->getDestTable() == $destTable) {
          $dest[$field->getDestCol()] = NULL;
          $default = $field->getDefault($sourceData);
          if ($default) {
              $dest[$field->getDestCol()] = $default;
          } else {
              $dest[$field->getDestCol()] =
                    $sourceData[$field->getSource()];
          }
      }
    }
    return $dest;
  }
  }
```

请注意，目的表中会出现源表中没有的某些字段。在这种情况中，FieldConfig 对象中的$source 属性会被设置为 NULL，而默认值会以标量值或回调函数的形式被采用。

13. 定义两个用于生成 SQL 语句的方法。使第一个方法生成从源表读取数据的 SQL 语句。该 SQL 语句应该包含用于执行准备操作（例如使用 PDO::prepare()方法）的占位符：

```
  public function getSourceSelect($where = NULL)
  {
    $sql = 'SELECT '
```

```
    . implode(',', $this->getSourceColumns()) . ' ';
    $sql .= 'FROM ' . $this->getSourceTable() . ' ';
    if ($where) {
      $where = trim($where);
      if (stripos($where, 'WHERE') !== FALSE) {
          $sql .= $where;
      } else {
          $sql .= 'WHERE ' . $where;
      }
    }
    return trim($sql);
  }
```

14. 在编写另一个用于生成 SQL 语句的方法时，使其生成为指定的目的表准备的语句。注意，SQL 语句中的占位符与字段名称相同，但前面带冒号（:）：

```
  public function getDestInsert($table)
  {
    $sql = 'INSERT INTO ' . $table . ' ';
    $sql .= '( '
    . implode(',', $this->getDestColumns($table))
    . ' ) ';
    $sql .= ' VALUES ';
    $sql .= '( :'
    . implode(',:', $this->getDestColumns($table))
    . ' ) ';
    return trim($sql);
  }
```

具体运行情况

使用步骤 1 至步骤 5 介绍的代码创建 `Application\Database\Mapper\FieldConfig` 类。使用步骤 6 至步骤 14 介绍的代码，创建 `Application\Database\Mapper\Mapping` 类。

在创建执行映射操作的调用程序前，应考虑源数据库表和目的数据库表的结构，这一点很重要。下面是源表 `prospects_11` 的结构：

```
CREATE TABLE `prospects_11` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(128) NOT NULL,
  `last_name` varchar(128) NOT NULL,
  `address` varchar(256) DEFAULT NULL,
  `city` varchar(64) DEFAULT NULL,
```

```
  `state_province` varchar(32) DEFAULT NULL,
  `postal_code` char(16) NOT NULL,
  `phone` varchar(16) NOT NULL,
  `country` char(2) NOT NULL,
  `email` varchar(250) NOT NULL,
  `status` char(8) DEFAULT NULL,
  `budget` decimal(10,2) DEFAULT NULL,
  `last_updated` datetime DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `UNIQ_35730C06E7927C74` (`email`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

本例使用两个目的表：customer_11 和 profile_11，这两个表中的记录都是一一对应的：

```
CREATE TABLE `customer_11` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(256) CHARACTER SET latin1
     COLLATE latin1_general_cs NOT NULL,
  `balance` decimal(10,2) NOT NULL,
  `email` varchar(250) NOT NULL,
  `password` char(16) NOT NULL,
  `status` int(10) unsigned NOT NULL DEFAULT '0',
  `security_question` varchar(250) DEFAULT NULL,
  `confirm_code` varchar(32) DEFAULT NULL,
  `profile_id` int(11) DEFAULT NULL,
  `level` char(3) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `UNIQ_81398E09E7927C74` (`email`)
  ) ENGINE=InnoDB AUTO_INCREMENT=80 DEFAULT CHARSET=utf8
  COMMENT='Customers';

CREATE TABLE `profile_11` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `address` varchar(256) NOT NULL,
  `city` varchar(64) NOT NULL,
  `state_province` varchar(32) NOT NULL,
  `postal_code` varchar(10) NOT NULL,
  `country` varchar(3) NOT NULL,
  `phone` varchar(16) NOT NULL,
  `photo` varchar(128) NOT NULL,
  `dob` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=80 DEFAULT CHARSET=utf8
COMMENT='Customers';
```

现在可以定义调用程序 chap_11_mapper.php，为其设置类自动加载功能，并引用前面介绍的两个类，以及第 5 章介绍的 Connection 类：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
define('DEFAULT_PHOTO', 'person.gif');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Mapper\ { FieldConfig, Mapping };
use Application\Database\Connection;
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
```

为了了解具体处理步骤，确定两个目的表已经创建好以后，可删减这两个表的内容，以便使这些表变得更加清晰明了：

```
$conn->pdo->query('DELETE FROM customer_11');
$conn->pdo->query('DELETE FROM profile_11');
```

现在可定义 Mapping 实例，并使用 FieldConfig 对象为其赋值。每个 FieldConfig 对象代表一个源表和目的表之间的对应关系。应在 Mapping 类的构造器中，以数组的形式提供源表和两个目的表的名称：

```
$mapper = new Mapping('prospects_11', ['customer_11','profile_11']);
```

在没有默认值的情况下，只需通过将 prospects_11 和 customer_11 表中的字段对应起来，就能执行映射操作：

```
$mapper>addField(new FieldConfig('email','customer_11','email'))
```

注意，addField()方法会返回当前用于执行映射操作的实例，因此无须专门添加 $mapper->addField()语句。这种技巧称为**连贯接口**（fluent interface）。

处理姓名字段（name）有点难度，因为 prospects_11 表使用两个字段（first_name 和 last_name）代表该字段，但 customer_11 表仅含有 name 字段。因此，可将回调函数设置为 first_name 属性的默认值，以便将两个字段合并为一个字段。还需要为 last_name 字段定义一个映射关系，但它在目的表中并没有对应的字段：

```
->addField(new FieldConfig('first_name','customer_11','name',
  function ($row) { return trim(($row['first_name'] ?? '')
.' ' . ($row['last_name'] ?? ''));}))
->addField(new FieldConfig('last_name'))
```

可使用空值合并操作符（??）检查 customer_11::status 字段是否已被设置：

```
->addField(new FieldConZfig('status','customer_11','status',
  function ($row) { return $row['status'] ?? 'Unknown'; }))
```

源表中没有 customer_11::level 字段，因此可将源表字段的映射关系条目设置为 NULL，但同时应确保在该条目中设置目的表和字段。与此类似，源表中没有 customer_11::password 字段（代表登录密码）。在这种情况中，回调函数可将客户的电话号码用作临时的登录密码：

```
->addField(new FieldConfig(NULL,'customer_11','level','BEG'))
->addField(new FieldConfig(NULL,'customer_11','password',
  function ($row) { return $row['phone']; }))
```

可以使用下面的方式为 prospects_11 和 profile_11 表设置映射关系。注意，因为 prospects_11 表中没有 photo（代表照片）和 dob（代表出生日期）字段，所以可以在映射关系中为这些字段设置任何适当的默认值：

```
->addField(new FieldConfig('address','profile_11','address'))
->addField(new FieldConfig('city','profile_11','city'))
->addField(new FieldConfig('state_province','profile_11',
'state_province', function ($row) {
  return $row['state_province'] ?? 'Unknown'; }))
->addField(new FieldConfig('postal_code','profile_11',
'postal_code'))
->addField(new FieldConfig('phone','profile_11','phone'))
->addField(new FieldConfig('country','profile_11','country'))
->addField(new FieldConfig(NULL,'profile_11','photo',
DEFAULT_PHOTO))
->addField(new FieldConfig(NULL,'profile_11','dob',
date('Y-m-d')));
```

为了在 profile_11 和 customer_11 表之间创建一一对应的映射关系，可使用回调函数将 customer_11::id、customer_11::profile_id 和 profile_11::id 字段的值设置为$row['id']数组中存储的值：

```
$idCallback = function ($row) { return $row['id']; };
$mapper->addField(new FieldConfig('id','customer_11','id',
$idCallback))
->addField(new FieldConfig(NULL,'customer_11','profile_id',
$idCallback))
->addField(new FieldConfig('id','profile_11','id',$idCallback));
```

现在可以通过调用适当的方法生成 3 条 SQL 语句，一条用于从源表读取数据，另外两条用于向目的表中插入数据：

```
$sourceSelect   = $mapper->getSourceSelect();
$custInsert     = $mapper->getDestInsert('customer_11');
$profileInsert  = $mapper->getDestInsert('profile_11');
```

这 3 条语句可以立刻被处理为将来执行的准备语句：

```
$sourceStmt     = $conn->pdo->prepare($sourceSelect);
$custStmt       = $conn->pdo->prepare($custInsert);
$profileStmt    = $conn->pdo->prepare($profileInsert);
```

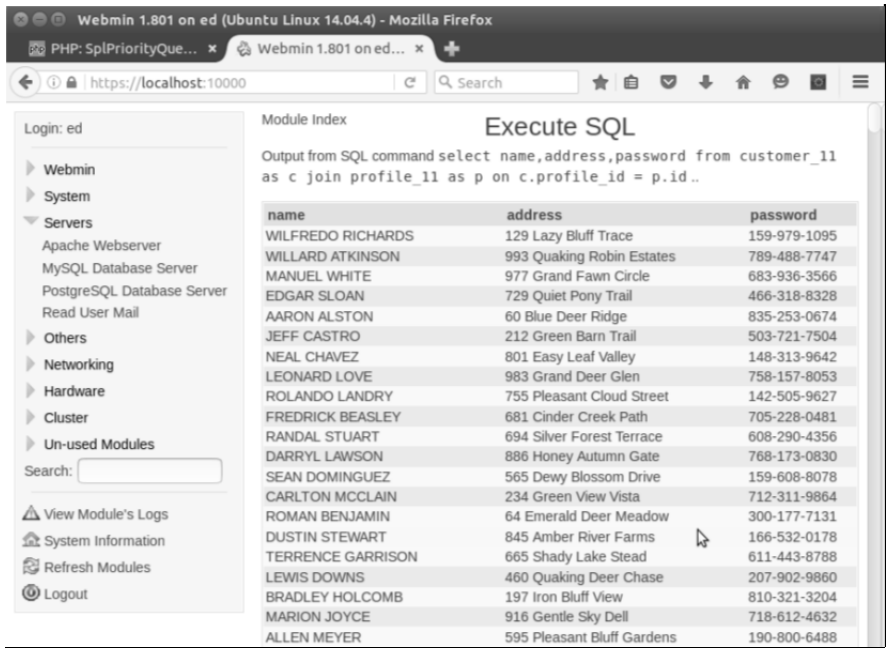
这样就可以执行 SELECT 语句，该语句可以从源表获取记录。可使用循环语句为每个目的表生成插入（INSERT）这些表中的数据，并执行适当的准备语句：

```
$sourceStmt->execute();
while ($row = $sourceStmt->fetch(PDO::FETCH_ASSOC)) {
  $custData = $mapper->mapData($row, 'customer_11');
  $custStmt->execute($custData);
  $profileData = $mapper->mapData($row, 'profile_11');
  $profileStmt->execute($profileData);
  echo "Processing: {$custData['name']}\n";
}
```

下面是通过调用方法生成的 3 条 SQL 语句：

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter11
SQL Statements:
SELECT email,first_name,last_name,status,address,city,state_province,postal_code,phone,
country,id,id FROM prospects_11
INSERT INTO customer_11 ( email,name,status,level,password,id,profile_id )  VALUES ( :e
mail,:name,:status,:level,:password,:id,:profile_id )
INSERT INTO profile_11 ( address,city,state_province,postal_code,phone,country,photo,do
b,id )  VALUES ( :address,:city,:state_province,:postal_code,:phone,:country,:photo,:do
b,:id )

Processing: WILFREDO RICHARDS
Processing: WILLARD ATKINSON
Processing: MANUEL WHITE
Processing: EDGAR SLOAN
Processing: AARON ALSTON
Processing: JEFF CASTRO
Processing: NEAL CHAVEZ
Processing: LEONARD LOVE
Processing: ROLANDO LANDRY
Processing: FREDRICK BEASLEY
Processing: RANDAL STUART
Processing: DARRYL LAWSON
Processing: SEAN DOMINGUEZ
Processing: CARLTON MCCLAIN
:
```

这样就可以使用 SQL 语言中的 JOIN 子句直接从数据库查看数据，以确定映射关系是否已经建立好：



实现对象关联映射功能

在对象之间实现关联映射功能的技巧主要有两种。第一种技巧需要预先将相关的子对象加载到父对象中。这种方式的优点是易于实现，而且可以立刻使用所有父子关系信息。这种方式的缺点是会占用大量的内存，而且会降低性能。

第二种技巧是在父对象中嵌入二次查询操作。在使用这种方式时需要访问子对象，通过调用读取器执行二次查询操作。这种方式的优点是可以将性能要求分布到请求周期的各个阶段中，并使得（或有可能使得）内存管理容易很多。这种方式的缺点是需要执行更多查询操作，这意味着会增加数据库服务器的负担。

我们将使用**准备语句**大幅度地降低该缺点带来的负面影响。

具体处理过程

下面分别使用这两种技巧实现对象关联映射功能。

第一种技巧——预先加载所有子对象的信息

让我们先了解通过将所有子对象的信息预先加载到父对象中，实现对象关联映射功能的方式。为了讲解具体处理过程，我们将使用 3 个关联的数据库表（customer、purchases 和 products）：

1. 可将第 5 章中介绍的 Application\Entity\Customer 类用作示例来开发 Application\Entity\Purchase 类。像以前一样，可将数据库定义用作实体类定义的基础。下面是 purchases 表的数据库定义：

```
CREATE TABLE `purchases` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `transaction` varchar(8) NOT NULL,
  `date` datetime NOT NULL,
  `quantity` int(10) unsigned NOT NULL,
  `sale_price` decimal(8,2) NOT NULL,
  `customer_id` int(11) DEFAULT NULL,
  `product_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `IDX_C3F3` (`customer_id`),
  KEY `IDX_665A` (`product_id`),
  CONSTRAINT `FK_665A` FOREIGN KEY (`product_id`) REFERENCES
     `products` (`id`),
  CONSTRAINT `FK_C3F3` FOREIGN KEY (`customer_id`) REFERENCES
     `customer` (`id`)
);
```

2. 根据 customer 实体类（代表客户）创建 Application\Entity\Purchase 类（代表购物信息）。注意，此处没有列出所有的读取器和设置器：

```
namespace Application\Entity;

class Purchase extends Base
{

  const TABLE_NAME = 'purchases';
  protected $transaction = '';
  protected $date = NULL;
```

```
    protected $Product= NULL;
    protected $quantity = 0;
    protected $salePrice = 0.0;
    protected $customerId = 0;
    protected $productId = 0;

    protected $mapping = [
      'id' => 'id',
      'transaction'   => 'transaction',
      'date'          => 'date',
      'quantity'      => 'quantity',
      'sale_price'    => 'salePrice',
      'customer_id'   => 'customerId',
      'product_id'    => 'productId',
    ];

    public function getTransaction() : string
    {
      return $this->transaction;
    }
    public function setTransaction($transaction)
    {
      $this->transaction = $transaction;
    }
    // 注意：此处没有列出其余读取器和设置器
}
```

3. 现在可以定义 Application\Entity\Product 类（代表商品）。下面是 products 表的数据库定义：

```
CREATE TABLE `products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `sku` varchar(16) DEFAULT NULL,
  `title` varchar(255) NOT NULL,
  `description` varchar(4096) DEFAULT NULL,
  `price` decimal(10,2) NOT NULL,
  `special` int(11) NOT NULL,
  `link` varchar(128) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `UNIQ_38C4` (`sku`)
);
```

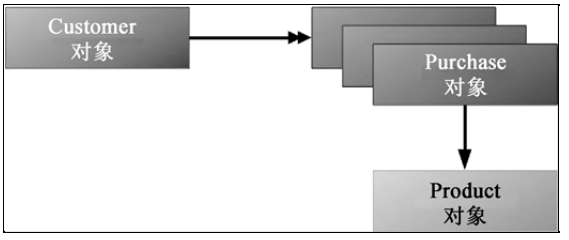
4. 根据 customer 实体类定义 Application\Entity\Product 类：

```
namespace Application\Entity;
```

```
class Product extends Base
{

  const TABLE_NAME = 'products';
  protected $sku = ' ';
  protected $title = ' ';
  protected $description = ' ';
  protected $price = 0.0;
  protected $special = 0;
  protected $link = ' ';

  protected $mapping = [
     'id'           => 'id',
     'sku'          => 'sku',
     'title'        => 'title',
     'description'  => 'description',
     'price'        => 'price',
     'special'      => 'special',
     'link'         => 'link',
  ];

  public function getSku() : string
  {
    return $this->sku;
  }
  public function setSku($sku)
  {
    $this->sku = $sku;
  }
  // 注意：此处没有列出其余读取器和设置器
}
```

5. 下面向父对象中嵌入关联的对象，先处理父类 `Application\Entity\Customer`。本例使用下面关联关系（如下图所示）：

- 在一个 Customer 对象中嵌入多个 Purchase 对象
- 在一个 Purchase 对象中嵌入一个 Product 对象



6. 因此，应定义一对读取器和设置器，以便通过数组批量处理代表多条购物信息的多个 Purchase 对象：

```
protected $purchases = array();
public function addPurchase($purchase)
{
  $this->purchases[] = $purchase;
}
public function getPurchases()
{
  return $this->purchases;
}
```

7. 下面我们将注意力转到 Application\Entity\Purchase 类。在本例中，购物信息与商品是一一对应的关系，因此无须使用数组进行处理：

```
protected $product = NULL;
public function getProduct()
{
  return $this->product;
}
public function setProduct(Product $product)
{
  $this->product = $product;
}
```

在上述两个实体类中，我们没有修改$mapping 数组。这是因为实现对象关联映射功能不会影响实体类中属性与数据库字段之间的对应关系。

8. 因为仍旧需要定义获取客户基本信息的核心功能，所以我们需要做的是扩展第 5 章介绍的 Application\Database\CustomerService 类。通过扩展 Application\Database\CustomerService 类，我们可以创建 Application\

Database\CustomerOrmService_1 类：

```
namespace Application\Database;
use PDO;
use PDOException;
use Application\Entity\Customer;
use Application\Entity\Product;
use Application\Entity\Purchase;
class CustomerOrmService_1 extends CustomerService
{
  // 在此处添加方法的代码
}
```

9. 在 CustomerOrmService_1 类中添加一个方法，使用该方法执行查询操作，并把获得的查询结果以 Product 和 Purchase 实体对象的形式插入核心实体对象 Customer 中。应使该方法以 JOIN 子句的形式执行查询操作，可以这样做的原因是 purchase 和 product 表之间存在一一对应的关系。因为这两个表中都有 id 字段，所以需要以别名的形式添加 purchase 表中的 ID 字段。循环遍历查询结果，以便创建 Product 和 Purchase 实体对象。覆盖了 ID 字段的值后，就可以将 Product 实体对象嵌入到 Purchase 实体对象中，然后将 Purchase 实体对象添加到 Customer 实体对象中的数组中：

```
protected function fetchPurchasesForCustomer(Customer $cust)
{
  $sql = 'SELECT u.*,r.*,u.id AS purch_id '
    . 'FROM purchases AS u '
    . 'JOIN products AS r '
    . 'ON r.id = u.product_id '
    . 'WHERE u.customer_id = :id '
    . 'ORDER BY u.date';
  $stmt = $this->connection->pdo->prepare($sql);
  $stmt->execute(['id' => $cust->getId()]);
  while ($result = $stmt->fetch(PDO::FETCH_ASSOC)) {
    $product = Product::arrayToEntity($result, new Product());
    $product->setId($result['product_id']);
    $purch = Purchase::arrayToEntity($result, new Purchase());
    $purch->setId($result['purch_id']);
    $purch->setProduct($product);
    $cust->addPurchase($purch);
  }
  return $cust;
}
```

10. 为之前介绍过的 fetchById()方法编写封装器。这个代码块不仅需要获取最初的 Customer 实体对象，而且还需要执行查询操作并嵌入 Product 和 Purchase 实体对象。可以调用 fetchByIdAndEmbedPurchases()方法，并将客户的 ID 字段值接收为参数：

```
public function fetchByIdAndEmbedPurchases($id)
{
  return $this->fetchPurchasesForCustomer(
    $this->fetchById($id));
}
```

第二种技巧——嵌入二次查询操作

下面我们会将二次查询操作嵌入到关联的实体类中。本例会继续使用上例插图中介绍的实体类，将这些已定义的实体类与 3 个相互关联的数据库表（customer、purchases 和 products）对应：

1. 本技巧的技术细节与第一种技巧的技术细节非常相似。它们之间的主要区别是，第二种技巧不会执行数据库查询操作并立刻生成实体类，而是通过嵌入一系列匿名函数并使用查看逻辑调用这些函数，实现相同的效果。

2. 需要在 Application\Entity\Customer 类中添加一个新方法，以便能够向 purchases 属性中添加单个条目。这次不通过数组存储 Purchase 实体类，而是使用匿名函数为 purchases 属性赋值：

```
public function setPurchases(Closure $purchaseLookup)
{
  $this->purchases = $purchaseLookup;
}
```

3. 为 Application\Database\CustomerOrmService_1 创建一个副本，将该副本命名为 Application\Database\CustomerOrmService_2：

```
namespace Application\Database;
use PDO;
use PDOException;
use Application\Entity\Customer;
use Application\Entity\Product;
use Application\Entity\Purchase;
class CustomerOrmService_2 extends CustomerService
```

```
{
  // 此处添加具体代码
}
```

4. 定义 fetchPurchaseById()方法，使用该方法根据购物信息记录的 ID 查询单条购物记录，并生成 Purchase 实体对象。因为本例最终会重复请求获取一系列单条购物记录，所以能够通过处理相同的准备语句（本例使用$purchPreparedStmt 变量），提高数据库查询操作的效率：

```
public function fetchPurchaseById($purchId)
{
  if (!$this->purchPreparedStmt) {
       $sql = 'SELECT * FROM purchases WHERE id = :id';
       $this->purchPreparedStmt =
       $this->connection->pdo->prepare($sql);
  }
  $this->purchPreparedStmt->execute(['id' => $purchId]);
  $result = $this->purchPreparedStmt->fetch(PDO::FETCH_ASSOC);
  return Purchase::arrayToEntity($result, new Purchase());
}
```

5. 创建 fetchProductById()方法，使该方法通过商品的 ID 查询单条商品记录，并生成 Product 实体对象。因为一位客户很可能会多次购买同一类商品，所以将获得的商品实体对象（即 Product 实例）存储在$products 数组中可以进一步提高效率。此外，在处理购物信息记录时，可使用相同的准备语句执行查询操作：

```
public function fetchProductById($prodId)
{
  if (!isset($this->products[$prodId])) {
       if (!$this->prodPreparedStmt) {
            $sql = 'SELECT * FROM products WHERE id = :id';
            $this->prodPreparedStmt =
            $this->connection->pdo->prepare($sql);
       }
       $this->prodPreparedStmt->execute(['id' => $prodId]);
       $result = $this->prodPreparedStmt
       ->fetch(PDO::FETCH_ASSOC);
       $this->products[$prodId] =
         Product::arrayToEntity($result, new Product());
  }
  return $this->products[$prodId];
}
```

6. 现在可修改 fetchPurchasesForCustomer()方法，在该方法中嵌入一个调用 fetchPurchaseById()和 fetchProductById()方法的匿名函数，然后将获得的购物信息实体对象（即 Purchase 实例）中的$Product 属性赋予获得的商品实体对象（即 Product 实例）。本例执行初始查询操作来获取某位客户的所有购物信息记录的 ID。然后在 Customer::$purchases 属性中嵌入一系列匿名函数，以便将购物信息记录的 ID 存储为该数组的键，并将这些匿名函数存储为该数组的值：

```
public function fetchPurchasesForCustomer(Customer $cust)
{
  $sql = 'SELECT id '
    . 'FROM purchases AS u '
    . 'WHERE u.customer_id = :id '
    . 'ORDER BY u.date';
  $stmt = $this->connection->pdo->prepare($sql);
  $stmt->execute(['id' => $cust->getId()]);
  while ($result = $stmt->fetch(PDO::FETCH_ASSOC)) {
    $cust->addPurchaseLookup(
    $result['id'],
    function ($purchId, $service) {
      $purchase  = $service->fetchPurchaseById($purchId);
      $product   = $service->fetchProductById(
                   $purchase->getProductId());
      $purchase->setProduct($product);
      return $purchase; }
    );
  }
  return $cust;
}
```

具体运行情况

使用相应步骤介绍的代码定义下列类，第一种技巧中使用的类如下表所示：

类	第一种技巧中对应的步骤
Application\Entity\Purchase	1、2、7
Application\Entity\Product	3、4
Application\Entity\Customer	6、16 以及第 5 章介绍该类的相关步骤
Application\Database\CustomerOrmService_1	8 ~ 10

第二种技巧中使用的类如下表所示：

类	第二种技巧中对应的步骤
Application\Entity\Customer	2
Application\Database\CustomerOrmService_2	3 ~ 6

要使用第一种技巧（嵌入实体对象的处理方式），可定义调用程序 chap_11_orm_embedded.php，为其设置类自动加载功能，并引用合适的类：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Connection;
use Application\Database\CustomerOrmService_1;
```

创建代表服务的实例（CustomerOrmService_1 实例），使用随机 ID 查询客户记录：

```
$service = new CustomerOrmService_1(
           new Connection(include __DIR__ . DB_CONFIG_FILE));
$id      = rand(1,79);
$cust    = $service->fetchByIdAndEmbedPurchases($id);
```

在查看逻辑中，可通过 fetchByIdAndEmbedPurchases()方法得到获取了数据库表 customer 中全部字段值的 Customer 实体对象。现在需要做的是调用正确的读取器以便显示信息：

```
<!-- 客户的信息 -->
<h1><?= $cust->getname() ?></h1>
<div class="row">
  <div class="left">Balance</div><div class="right">
    <?= $cust->getBalance(); ?></div>
</div>
  <!–依此类推 -->
```

下面是用于显示购物信息的 HTML 代码。注意，Customer::getPurchases()方法会返回一组 Purchase 实体对象。要从 Purchase 实体对象中获取商品信息，可在循环中调用 Purchase::getProduct()方法，该方法会生成 Product 实体对象。这样就能够调用 Product 实体类中的任何读取器，本例调用的读取器为 Product::getTitle()方法：

```
<!-- 购物信息 -->
<table>
```

```
<?php foreach ($cust->getPurchases() as $purchase) : ?>
  <tr>
  <td><?= $purchase->getTransaction() ?></td>
  <td><?= $purchase->getDate() ?></td>
  <td><?= $purchase->getQuantity() ?></td>
  <td><?= $purchase->getSalePrice() ?></td>
  <td><?= $purchase->getProduct()->getTitle() ?></td>
  </tr>
  <?php endforeach; ?>
</table>
```

让我们将注意力转向第二种技巧,这种技巧会用到二次查询。定义调用程序 chap_11_orm_secondary_lookups.php，为其设置类自动加载功能，并引用合适的类：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Connection;
use Application\Database\CustomerOrmService_2;
```

创建一个代表服务的实例（CustomerOrmService_2 实例），并使用随机 ID 查询客户记录：

```
$service = new CustomerOrmService_2(new Connection(include __DIR__ .
DB_CONFIG_FILE));
$id = rand(1,79);
```

现在可以检索 Application\Entity\Customer 实例，并调用该实例中的 fetchPurchasesForCustomer()方法，该方法会调用一系列匿名函数：

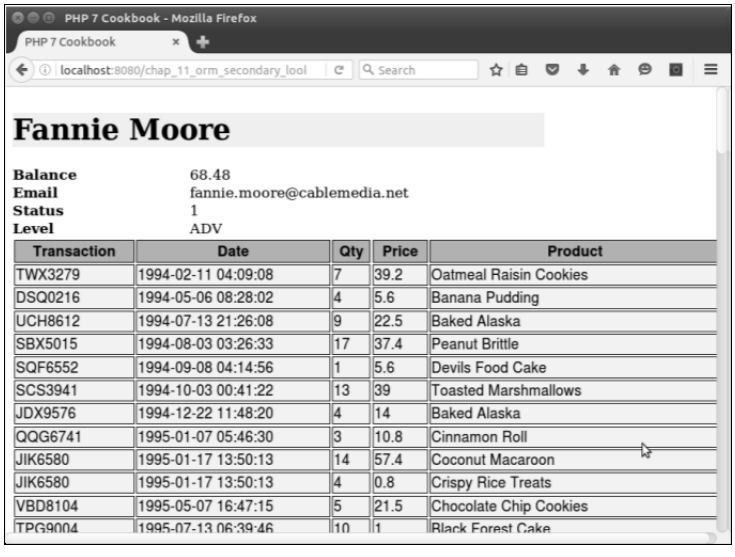
```
$cust = $service->fetchById($id);
$cust = $service->fetchPurchasesForCustomer($cust);
```

用于显示核心客户信息的显示逻辑与上例使用的显示逻辑相同。下面的 HTML 代码是用于显示购物信息的逻辑。注意，Customer::getPurchases()方法会返回一组匿名函数。这些匿名函数每次被调用时，都会返回一条具体购物信息记录和相关的商品记录：

```
<table>
  <?php foreach($cust->getPurchases() as $purchId => $function) : ?>
  <tr>
  <?php $purchase = $function($purchId, $service); ?>
  <td><?= $purchase->getTransaction() ?></td>
  <td><?= $purchase->getDate() ?></td>
  <td><?= $purchase->getQuantity() ?></td>
```

```
  <td><?= $purchase->getSalePrice() ?></td>
  <td><?= $purchase->getProduct()->getTitle() ?></td>
  </tr>
  <?php endforeach; ?>
</table>
```

下面是该程序的输出结果：



最佳编程习惯

尽管循环中的每次迭代都会执行两个独立的数据库查询操作（一个用于获取购物信息，一个用于获取商品信息），但还是可以通过准备语句保证程序性能。预先准备的 SQL 语句有两条：一条用于查询指定的购物信息，另一条用于查询指定的商品信息。然后这些准备语句会被多次执行。因此，尽管每条商品信息记录都以独立方式存储在数组中，但在使用预备语句的情况下依然能够提高性能。

扩展

Doctrine 也许是实现对象关联映射功能的最佳范例软件库。Doctrine 库使用嵌入处理方式（其说明文档称之为代理方式）。要详细了解这方面的内容，请浏览 http://www.doctrine-project.org/projects/orm.html。要观看 Doctrine

库的教学视频，请浏览 http://shop.oreilly.com/product/0636920041382.do。本书作者参与了该视频的制作。

实现发布/订阅设计模式

发布/订阅设计模式通常是事件驱动型软件的基础设计模式。这种设计模式允许在多个应用程序或一个应用程序中的多个模块之间进行异步通信。这样做的目的是允许方法或函数在某个重要操作出现后发布消息。当某个特定消息被发布后，一个或多个对象会订阅这条消息并执行操作。

数据库修改操作和用户登录操作，就是这种设计模式的典型例子。这种设计模式的另一种常见用途是应用程序的新闻推送功能。当出现紧急新闻时，应用程序需要公布此新闻，并能够为订阅新闻的用户刷新新闻列表。

具体处理过程

1. 先定义代表发布者的类：Application\PubSub\Publisher。你将看到我们使用了两个功能较强的 PHP 标准库（Standard PHP Library，SPL）接口 SplSubject 和 SplObserver：

```
namespace Application\PubSub;
use SplSubject;
use SplObserver;
class Publisher implements SplSubject
{
  // 此处添加具体代码
}
```

2. 定义一些属性，用这些属性分别代表发布者、传送给订阅者的数据和一组订阅者（也称为监听者）。注意，我们会使用链表（请参阅第 10 章）实现优先次序：

```
protected $name;
protected $data;
protected $linked;
protected $subscribers;
```

3. 编写构造器，以便初始化这些属性。还应添加魔术函数__toString()，以便能够快速访问发布者的名称：

```
public function __construct($name)
{
  $this->name = $name;
  $this->data = array();
  $this->subscribers = array();
  $this->linked = array();
}

public function __toString()
{
  return $this->name;
}
```

4. 为了将订阅者与发布者关联起来，可定义 attach() 方法，它是按照 SplSubject 接口的规定实现的。该方法会将 SplObserver 实例接收为参数。注意，我们既需要为$subscribers 属性（数组）赋值，也需要为$linked 属性（数组）赋值。然后可使用 arsort() 函数（arsort() 函数使用降序排序，并且不会影响数组中的键），根据$linked 数组中各个元素的值（代表优先等级）对$linked 数组中的元素进行排序：

```
public function attach(SplObserver $subscriber)
{
  $this->subscribers[$subscriber->getKey()] = $subscriber;
  $this->linked[$subscriber->getKey()] =
    $subscriber->getPriority();
  arsort($this->linked);
}
```

5. 我们还需要按照 SplSubject 接口的规定来定义 detach() 方法，使该方法能够从接收者列表中删除条目：

```
public function detach(SplObserver $subscriber)
{
  unset($this->subscribers[$subscriber->getKey()]);
  unset($this->linked[$subscriber->getKey()]);
}
```

6. SplSubject 接口还规定必须实现 notify() 方法，该方法能够调用每个订阅者中的 update() 方法。注意，应通过循环遍历链表的方式，确保按照由高至低的优先等级调用订阅者：

```
public function notify()
{
```

```
    foreach ($this->linked as $key => $value)
    {
      $this->subscribers[$key]->update($this);
    }
  }
```

7. 定义合适的读取器和设置器。为节省篇幅此处没有将这些方法都列出来：

```
public function getName()
{
  return $this->name;
}

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

8. 还需要编写通过键设置数据条目的功能，以便在 `notify()` 方法被调用时能够对订阅者起作用：

```
public function setDataByKey($key, $value)
{
  $this->data[$key] = $value;
}
```

9. 下面编写用于代表订阅者的 `Application\PubSub\Subscriber` 类。通常需要为每个发布者定义多个订阅者。本例实现了 `SplObserver` 接口：

```
namespace Application\PubSub;
use SplSubject;
use SplObserver;
class Subscriber implements SplObserver
{
  // 此处添加具体代码
}
```

10. 每个订阅者都需要拥有一个具有唯一性的标识符。本例使用 `md5()` 函数创建键（通过将日期/时间信息与随机数字混合的方式）。如下面代码所示，构造器应初始化多个属性。由订阅者执行的实际逻辑功能是以回调函数的形式实现的：

```
protected $key;
protected $name;
protected $priority;
protected $callback;
```

```
public function __construct(
  string $name, callable $callback, $priority = 0)
{
  $this->key = md5(date('YmdHis') . rand(0,9999));
  $this->name = $name;
  $this->callback = $callback;
  $this->priority = $priority;
}
```

11. 当发布者中的 notifiy() 方法被调用时，update() 函数就会被调用。可将代表发布者的实例用作 update() 函数的参数，并调用为该订阅者定义的回调函数：

```
public function update(SplSubject $publisher)
{
  call_user_func($this->callback, $publisher);
}
```

12. 为简便起见，还需要定义读取器和设置器。此处没有列出所有方法：

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

public function setKey($key)
{
  $this->key = $key;
}

// 此处省略了其他读取器和设置器
```

具体运行情况

为了做这个实验，可定义调用程序 chap_11_pub_sub_simple_example.php，为其设置类自动加载功能，并引用合适的类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\PubSub\ { Publisher, Subscriber };
```

创建一个代表发布者的实例并为该实例赋值：

```
$pub = new Publisher('test');
$pub->setDataByKey('1', 'AAA');
```

```
$pub->setDataByKey('2', 'BBB');
$pub->setDataByKey('3', 'CCC');
$pub->setDataByKey('4', 'DDD');
```

创建用于进行测试的订阅者，使它们能够从发布者那里读取数据并显示得到的结果。可以将第一个参数设置为名称，将第二个参数设置为回调函数，并将最后一个参数设置为优先级：

```
$sub1 = new Subscriber(
  '1',
  function ($pub) {
    echo '1:' . $pub->getData()[1] . PHP_EOL;
  },
  10
);
$sub2 = new Subscriber(
  '2',
  function ($pub) {
    echo '2:' . $pub->getData()[2] . PHP_EOL;
  },
  20
);
$sub3 = new Subscriber(
  '3',
  function ($pub) {
    echo '3:' . $pub->getData()[3] . PHP_EOL;
  },
  99
);
```

为了进行测试，可按照打乱的次序将订阅者与发布者关联起来，并调用 `notify()` 方法两次：

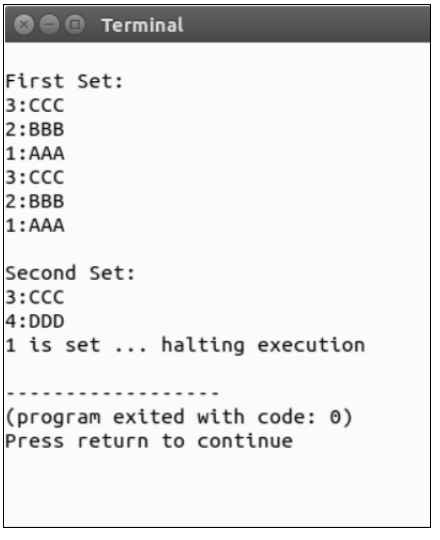
```
$pub->attach($sub2);
$pub->attach($sub1);
$pub->attach($sub3);
$pub->notify();
$pub->notify();
```

定义并关联另一个订阅者，并检查 1 号订阅者（由存储在`$sub1` 中的 Subscriber 实例代表）订阅的数据，如果该实例存储的值为非空，就结束该订阅者的创建过程：

```
$sub4 = new Subscriber(
  '4',
```

```
    function ($pub) {
      echo '4:' . $pub->getData()[4] . PHP_EOL;
      if (!empty($pub->getData()[1]))
        die('1 is set ... halting execution');
    },
    25
);
$pub->attach($sub4);
$pub->notify();
```

下面是输出结果。注意，这些输出结果是按照优先权等级由高至低排列的，而且在输出第二部分结果时程序会中断运行：



补充说明

与发布/订阅模式非常类似的一种软件设计模式是**观察者模式**。但这两种模式之间公认的差异，是观察者模式以同步方式运行，即当收到信号（也称为消息或事件）时，所有观察者方法都会被调用。与此相比，发布/订阅模式以异步方式运行，通常会使用消息队列。这两种模式之间的另一个区别是，发布/订阅模式中的发布者无须知道有哪些订阅者。

扩展

要详细了解观察者模式和发布/订阅模式之间的异同，请浏览 http://stackoverflow.com/questions/15594905/differencebetween-observer-pub-sub-and-data-binding。

第 12 章　提高网页的安全性

本章包括以下要点：

- 过滤通过$_POST 变量获得的数据
- 验证通过$_POST 变量获得的数据
- 为 PHP 会话提供安全防护
- 通过令牌提高表单的安全性
- 创建具有较高安全性的密码生成器
- 通过验证码为表单提供安全防护
- 在不使用 mcrypt 加密扩展库的情况下实现加密/解密功能

本章主要内容简介

本章介绍一种简单且高效的机制，以便过滤和验证通过$_POST 变量获得的数据。然后介绍保护 PHP 会话的技巧，使其免受各种形式的攻击（如会话劫持攻击）。接下来会介绍使用随机生成的令牌，保护表单免受**跨网站伪造请求**（Cross Site Request Forgery，CSRF）攻击的手段。本章还会介绍使用 PHP 7 中真正的随机化功能生成具有安全性的密码的方式。然后会介绍两种**验证码**：一种是基于文本的，另一种是通过扭曲图像实现的。最后，本章会介绍在不使用可信度低并且很快将被弃用的 mcrypt 扩展的情况下，如何实现强加密功能。

过滤通过$_POST 变量获得的数据

下面是过滤数据处理过程中，可能包含的部分或全部步骤：

- 删除不需要的字符（即删除<script>标签）
- 转换数据（将"转换为 HTML 中的特殊符号编码&quot）

- 加密或解密数据

本章最后一节会介绍加密技术。下面让我们先了解一种基础机制，这种机制用于在表单提交操作被执行后，过滤通过超级全局变量$_POST获取的数据。

具体处理过程

1. 我们应该先了解通过$_POST变量获取的数据的结构。也许更为重要的是，我们需要了解向数据库表中存储表单数据的限制。例如，我们来观察prospects表在数据库中的存储结构：

字段	数据类型	是否可为空	默认值
first_name	varchar(128)	否	无（NULL）
last_name	varchar(128)	否	无（NULL）
address	varchar(256)	是	无（NULL）
city	varchar(64)	是	无（NULL）
state_province	varchar(32)	是	无（NULL）
postal_code	char(16)	否	无（NULL）
phone	varchar(16)	否	无（NULL）
country	char(2)	否	无（NULL）
email	varchar(250)	否	无（NULL）
status	char(8)	是	无（NULL）
budget	decimal(10,2)	是	无（NULL）
last_updated	datetime	是	无（NULL）

2. 对于需要在网页中显示和在数据库中存储的数据完成分析后，就可以确定需要进行哪种类型的过滤，以及需要用到哪些PHP函数。

3. 例如，如果需要去掉开头和结尾处的空格（这种情况在用户提供的表单数据中非常常见），可使用PHP的trim()函数。数据库的结构对于所有字符型数据都有长度限制。因此，可使用substr()函数确保字符型数据没有超过长度上限。如果你想要去掉非英文字母表中的字符，可通过适当形式使用preg_replace()函数。

4. 现在可以将你想要使用的一系列PHP函数存储到一个独立的回调函数数组中。下面的回调函数数组示例是根据需要对表单数据执行的过滤操作编写的，经过下列过滤操作后的表单数据最终会被存储到prospects表中：

```
$filter = [
  'trim' => function ($item) { return trim($item); },
```

```
  'float' => function ($item) { return (float) $item; },
  'upper' => function ($item) { return strtoupper($item); },
  'email' => function ($item) {
    return filter_var($item, FILTER_SANITIZE_EMAIL); },
  'alpha' => function ($item) {
    return preg_replace('/[^A-Za-z]/', '', $item); },
  'alnum' => function ($item) {
    return preg_replace('/[^0-9A-Za-z ]/', '', $item); },
  'length' => function ($item, $length) {
    return substr($item, 0, $length); },
  'stripTags' => function ($item) { return strip_tags($item); },
];
```

5. 根据$_POST 变量中可能存储的表结构（字段名称）定义一个数组。我们使用该数组和相关参数，根据不同需要调用$filter 数组中的回调函数。请注意该数组中的第一个键（*），它可以被用作对所有字段进行过滤操作的通配符：

```
$assignments = [
  '*' => ['trim' => NULL, 'stripTags' => NULL],
  'first_name' => ['length' => 32, 'alnum' => NULL],
  'last_name' => ['length' => 32, 'alnum' => NULL],
  'address' => ['length' => 64, 'alnum' => NULL],
  'city' => ['length' => 32],
  'state_province'=> ['length' => 20],
  'postal_code' => ['length' => 12, 'alnum' => NULL],
  'phone' => ['length' => 12],
  'country' => ['length' => 2, 'alpha' => NULL,
                'upper' => NULL],
  'email' => ['length' => 128, 'email' => NULL],
  'budget' => ['float' => NULL],
];
```

6. 循环遍历数据集合（即通过$_POST 变量获得的数据），并依次调用回调函数。先调用分配给通配符键（*）的所有回调函数。

实现通配符过滤器非常重要，因为这样可以避免出现冗余的设置。在前面的示例中，我们希望将通过 PHP 函数 strip_tags()和 trim()实现的过滤器应用于每个字段值。

7. 调用所有分配给指定字段数据的回调函数。调用了这些回调函数后，$data 变

量中存储的所有字段值就都会被过滤：

```
foreach ($data as $field => $item) {
  foreach ($assignments['*'] as $key => $option) {
    $item = $filter[$key]($item, $option);
  }
  foreach ($assignments[$field] as $key => $option) {
    $item = $filter[$key]($item, $option);
  }
}
```

具体运行情况

将步骤4至步骤6介绍的代码添加到chap_12_post_data_filtering_ basic.php 文件中。还需要定义一个数组，以模拟通过$_POST 变量获得的数据。实际上在做这个实验时需要定义两个数组，一个数组用于模拟符合数据库限定条件的数据，另一个数组用于模拟不符合数据库限定条件的数据：

```
$testData           = [
  'goodData'        => [
    'first_name'    => 'Doug',
    'last_name'     => 'Bierer',
    'address'       => '123 Main Street',
    'city'          => 'San Francisco',
    'state_province'=> 'California',
    'postal_code'   => '94101',
    'phone'         => '+1 415-555-1212',
    'country'       => 'US',
    'email'         => 'doug@unlikelysource.com',
    'budget'        => '123.45',
  ],
  'badData' => [
    'first_name' => 'This+Name<script>bad tag</script>Valid!',
    'last_name' =>
       'ThisLastNameIsWayTooLongAbcdefghijklmnopqrstuvwxyz0123456789
       Abcdefghijklmnopqrstuvwxyz0123456789Abcdefghijklmnopqrstuvwxyz
       0123456789Abcdefghijklmnopqrstuvwxyz0123456789',
    //'address' => ' ', // 用户没有填写这个表单项
   'city'       => 'ThisCityNameIsTooLong01234567890123456
      7890123456789012345678901234567890123456789 ',
    //'state_province'=> ' ', // 用户没有填写这个表单项
```

```
    'postal_code'    => '!"£$%^Non Alpha Chars',
    'phone'          => ' 12345 ',
    'country'        => '12345',
    'email'          => 'this.is@not@an.email',
    'budget'         => 'XXX',
  ]
];
```

应使用循环方式通过过滤器执行赋值操作，将对符合和不符合数据库限定条件的数据进行过滤后的结果都显示出来：

```
foreach ($testData as $data) {
  foreach ($data as $field => $item) {
    foreach ($assignments['*'] as $key => $option) {
      $item = $filter[$key]($item, $option);
    }
    foreach ($assignments[$field] as $key => $option) {
      $item = $filter[$key]($item, $option);
    }
    printf("%16s : %s\n", $field, $item);
  }
}
```

下面是本例的输出结果：

注意，不符合数据库限定条件数据的姓名字段中的字符串被截断了，而且其中的标签也被去除了。还应注意，尽管电子邮件地址已经被过滤，但它仍然不是一个合法的电子邮件地址。还需注意的一个要点是，在处理数据的过程中，验证和过滤的重要性是相同的。

补充说明

对于本节介绍的基础过滤概念，第 6 章讲述了将其应用到综合过滤关联机制中的方式。

验证通过$_POST 变量获得的数据

过滤和验证之间的主要区别是，验证操作无须更改原始数据。它们之间的另一个不同点是处理目的。执行验证操作的目的是，确定数据符合根据用户需求所制定的标准。

具体处理过程

1. 本节介绍的基础验证机制与上一节介绍的基础过滤机制非常相似。在实现过滤操作时，必须了解即将被验证的数据有什么特点，如这些数据是通过怎样的方式满足用户需求的，以及这些数据是否符合数据库的强制限定标准，举例来说，如果数据库规定字段值的最大长度为 128 个字符，那么执行验证操作的回调函数就可以使用 `strlen()` 函数，确保被提交数据的长度小于或等于 128 个字符；同理，可根据具体情况使用 `ctype_alnum()` 函数来确保数据中仅含有字母和数字。

2. 在实现验证操作时需要注意的另一点是，应显示合适的验证失败消息。从某个方面讲，验证处理过程也是一种确认处理过程，人们需要通过检查执行验证操作的结果来确认验证操作是执行成功了还是失败了。如果验证操作执行失败了，那么就需要了解失败的原因。

3. 本例将继续使用 `prospects` 表。现在我们可以将想要使用的 PHP 函数都存储到一个独立的回调函数数组中。下面的示例程序是根据验证表单数据的需要编写的，这些表单数据最终会被存储到 `prospects` 表中：

```
$validator = [
  'email' => [
    'callback' => function ($item) {
      return filter_var($item, FILTER_VALIDATE_EMAIL); },
    'message' => 'Invalid email address'],
  'alpha' => [
    'callback' => function ($item) {
      return ctype_alpha(str_replace('  ', ' ', $item)); },
```

```
    'message' => 'Data contains non-alpha characters'],
  'alnum' => [
    'callback' => function ($item) {
      return ctype_alnum(str_replace('  ', ' ', $item)); },
    'message' => 'Data contains characters which are '
       . 'not letters or numbers'],
  'digits' => [
    'callback' => function ($item) {
      return preg_match('/[^0-9.]/', $item); },
    'message' => 'Data contains characters which '
      . 'are not numbers'],
  'length' => [
    'callback' => function ($item, $length) {
      return strlen($item) <= $length; },
    'message' => 'Item has too many characters'],
  'upper' => [
    'callback' => function ($item) {
      return $item == strtoupper($item); },
    'message' => 'Item is not upper case'],
  'phone' => [
    'callback' => function ($item) {
      return preg_match('/[^0-9() -+]/', $item); },
    'message' => 'Item is not a valid phone number'],
];
```

我们允许被验证数据中含有空格，这是因为在调用回调函数 alpha 和 alnum 时，可以先使用 str_replace()函数将空格去掉。然后再调用 ctype_alpha()或 ctype_alnum()函数，查明被验证数据中是否含有不允许出现的字符。

4. 定义一个用于执行赋值操作的数组，使该数组中存储的键值与通过$_POST 变量获得的数据的数据库结构相符，即数组的键与数据库表中的字段对应，数组的值与数据库表中的字段值对应。我们使用该数组和相关参数，根据不同需要调用$validator 数组中的回调函数：

```
$assignments = [
  'first_name'      => ['length' => 32, 'alpha' => NULL],
  'last_name'       => ['length' => 32, 'alpha' => NULL],
```

```
    'address'         => ['length' => 64, 'alnum' => NULL],
    'city'            => ['length' => 32, 'alnum' => NULL],
    'state_province'  => ['length' => 20, 'alpha' => NULL],
    'postal_code'     => ['length' => 12, 'alnum' => NULL],
    'phone'           => ['length' => 12, 'phone' => NULL],
    'country'         => ['length' => 2, 'alpha' => NULL,
                          'upper'  => NULL],
    'email'           => ['length' => 128, 'email' => NULL],
    'budget'          => ['digits' => NULL],
];
```

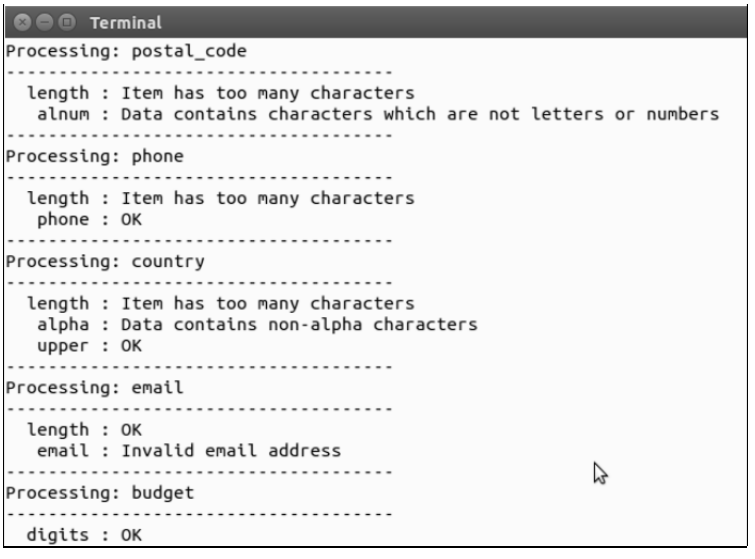
5. 使用嵌套的 `foreach()` 循环遍历被验证数据中的每个字段值。在验证具体字段值时，通过循环调用与该字段对应的回调函数：

```
foreach ($data as $field => $item) {
  echo 'Processing: ' . $field . PHP_EOL;
  foreach ($assignments[$field] as $key => $option) {
    if ($validator[$key]['callback']($item, $option)) {
        $message = 'OK';
    } else {
        $message = $validator[$key]['message'];
    }
    printf('%8s : %s' . PHP_EOL, $key, $message);
  }
}
```

如果你不想像下面那样直接显示输出结果，也可以将验证成功和失败的消息记录下来，以便将来查看。还可以像第 6 章介绍的那样，在表单中添加验证机制，在相应的表单元素旁边显示验证消息。

具体运行情况

将步骤 3 至步骤 5 介绍的代码添加到 `chap_12_post_data_validation_basic.php` 文件中。还应该定义一个数组，用来模拟通过`$_POST` 变量获得的数据。本例使用了上一节提到的两个数组，一个数组用于模拟符合数据库限定条件的数据，另一个数组用于模拟不符合数据库限定条件的数据。下面是这个验证程序的输出结果：



补充说明

对于本节介绍的基础验证概念，第 6 章讲述了将其应用到综合过滤关联机制中的方式。

为 PHP 会话提供安全防护

PHP 的会话机制非常简单。一旦通过 `session_start()` 函数或配置文件 `php.ini` 中的会话自动启动设置（`session.autostart`）启动会话后，PHP 引擎就会生成一个具有唯一性的令牌。默认情况下，该令牌会以 cookie 的形式被发送给用户。在之后的请求处理过程中，当会话仍旧处于活动状态时，为了辨别身份，用户的浏览器（或起相同作用的软件）通常会以 cookie 的形式提供会话标识符。这样 PHP 引擎就会使用这个标识符，找到存储在服务器上的相应文件，将客户端请求获取的信息存储到超级全局变量`$_SESSION` 中。如果将会话标识符用作辨别网站浏览者身份的唯一手段，会存在巨大的安全隐患。本节会介绍几种为会话提供安全防护的技巧，同时这些技巧也能够大幅度地提高网站的整体安全水平。

具体处理过程

1. 首先，应了解将会话标识符用作身份验证的唯一手段有哪些危险，这对于学习后面的知识很重要。假设某个合法用户正在登录你的网站，你就需要在$_SESSION 变量中存储一个布尔型的 loggedIn 标记：

```
session_start();
$loggedIn = $_SESSION['isLoggedIn'] ?? FALSE;
if (isset($_POST['login'])) {
  if ($_POST['username'] == // 查询用户名
       && $_POST['password'] == // 查询登录密码) {
       $loggedIn = TRUE;
       $_SESSION['isLoggedIn'] = TRUE;
  }
}
```

2. 在程序逻辑中，如果数组元素$_SESSION['isLoggedIn']被设置为 TRUE，那么就允许用户查看敏感信息：

```
<br>Secret Info
<br><?php if ($loggedIn) echo // 显示给用户的敏感信息; ?>
```

3. 如果攻击者获得了会话标识符，例如通过**跨站点脚本攻击**（Cross-site scripting，XSS）的方式，那么他只需将 PHPSESSID cookie 设置为非法获得的会话标识符，就能以合法用户的身份浏览你的网站了。

4. 一种能够轻松且快速缩短 PHPSESSID 有效期的方式是使用 session_regenerate_id()函数。这种非常简单的函数可以生成新的会话标识符，使旧会话标识符过期失效，同时保持会话数据完整并且对性能的影响最小。该函数只能在会话启动后被调用：

```
session_start();
session_regenerate_id();
```

5. 另一个经常被忽视的技巧是为网站浏览者提供注销选项。这一点非常重要，因为这不仅可以使用 session_destroy()函数销毁会话，还可以复位$_SESSION 数据并使会话 cookie 过期：

```
session_unset();
session_destroy();
setcookie('PHPSESSID', 0, time() - 3600);
```

6. 另一种能够为会话提供安全防护的简单技巧是为网站浏览者创建指纹。实现途径之一是收集超出会话标识符范围的具有唯一性的网站浏览者信息，包括用户使用的浏览器、可接受的语言和远程 IP 地址。可通过这些信息生成一个简单的散列值，并将该散列值存储在服务器上的独立文件中。当该用户再次浏览你的网站时，可根据会话信息允许该用户登录，然后使用指纹数据对该用户的身份进行二次验证：

```
$remotePrint = md5($_SERVER['REMOTE_ADDR']
                   . $_SERVER['HTTP_USER_AGENT']
                   . $_SERVER['HTTP_ACCEPT_LANGUAGE']);
$printsMatch = file_exists(THUMB_PRINT_DIR . $remotePrint);
if ($loggedIn && !$printsMatch) {
    $info = 'SESSION INVALID!!!';
    error_log('Session Invalid: ' . date('Y-m-d H:i:s'), 0);
    // 采取适当的处置手段
}
```

本例使用了 md5() 函数，这是因为该函数使用了一种快速生成散列值的算法，而且该函数非常适合在程序内部使用。不推荐在程序外部使用 md5() 函数，因为它容易受到穷举攻击（暴力破解，brute-force attack）。

具体运行情况

为了演示一个会话是有风险的，我们来编写一个简单的登录脚本，它在成功登录后会设置$_SESSION['isLoggedIn']标记，我们将此登录脚本命名为 chap_12_session_hijack.php：

```
session_start();
$loggedUser = $_SESSION['loggedUser'] ?? '';
$loggedIn = $_SESSION['isLoggedIn'] ?? FALSE;
$username = 'test';
$password = 'password';
$info = 'You Can Now See Super Secret Information!!!';

if (isset($_POST['login'])) {
  if ($_POST['username'] == $username
      && $_POST['password'] == $password) {
          $loggedIn = TRUE;
```

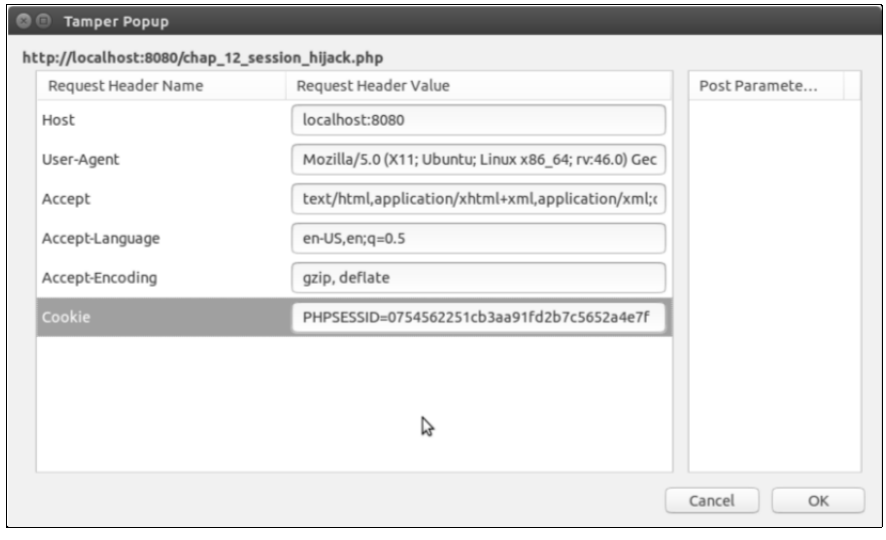
```
            $_SESSION['isLoggedIn'] = TRUE;
            $_SESSION['loggedUser'] = $username;
            $loggedUser = $username;
    }
  } elseif (isset($_POST['logout'])) {
    session_destroy();
  }
```

添加用于显示一个简单的登录表单的代码。要了解会话存在的安全隐患，可对我们刚刚创建的 `chap_12_session_hijack.php` 文件执行下列处理步骤：

1. 切换到含有该文件的目录。
2. 运行 `php -S localhost:8080` 命令。
3. 在浏览器的地址栏中输入 `http://localhost:8080/chap_12_session_hijack.php`，然后按回车键运行该文件。
4. 使用用户 `test` 和密码 `password` 登录。
5. 你将会看到提示信息：**You Can Now See Super Secret Information!!!**。
6. 刷新该页面，每次刷新后，你都会看到新的会话标识符。
7. 复制 `PHPSESSID` cookie 的值。
8. 打开另一个浏览器，并浏览同一页面。
9. 使用复制来的 `PHPSESSID` 值修改该浏览器发送的 cookie 内容。

为了了解具体情况，我们在此处还显示了`$_COOKIE` 和`$_SESSION` 变量的值，如下面 Vivaldi 浏览器的屏幕截图所示：

复制 PHPSESSID 的值，打开 Firefox 浏览器，并使用 Tamper Data 工具修改该 cookie 值：



如下图所示，虽然没有输入用户名和密码，但我们现在已经成为了一名经过身份验证的合法用户：

下面让我们对前面介绍的安全隐患采取一些措施。将 chap_12_session_hijack.

php 文件中的代码复制到 chap_12_session_protected.php 文件中。下面的代码会生成会话 ID：

```
<?php
define('THUMB_PRINT_DIR', __DIR__ . '/../data/');
session_start();
session_regenerate_id();
```

初始化变量并确认用户的登录状态（这部分代码与 chap_12_session_hijack.php 文件中的代码相同）：

```
$username = 'test';
$password = 'password';
$info = 'You Can Now See Super Secret Information!!!';
$loggedIn = $_SESSION['isLoggedIn'] ?? FALSE;
$loggedUser = $_SESSION['user'] ?? 'guest';
```

可使用远程 IP 地址、用户使用的浏览器和语言设置来创建会话指纹数据：

```
$remotePrint = md5($_SERVER['REMOTE_ADDR']
  . $_SERVER['HTTP_USER_AGENT']
  . $_SERVER['HTTP_ACCEPT_LANGUAGE']);
$printsMatch = file_exists(THUMB_PRINT_DIR . $remotePrint);
```

如果用户成功登录了，就将指纹信息和登录状态存储到会话中：

```
if (isset($_POST['login'])) {
  if ($_POST['username'] == $username
      && $_POST['password'] == $password) {
        $loggedIn = TRUE;
        $_SESSION['user'] = strip_tags($username);
        $_SESSION['isLoggedIn'] = TRUE;
        file_put_contents(
          THUMB_PRINT_DIR . $remotePrint, $remotePrint);
}
```

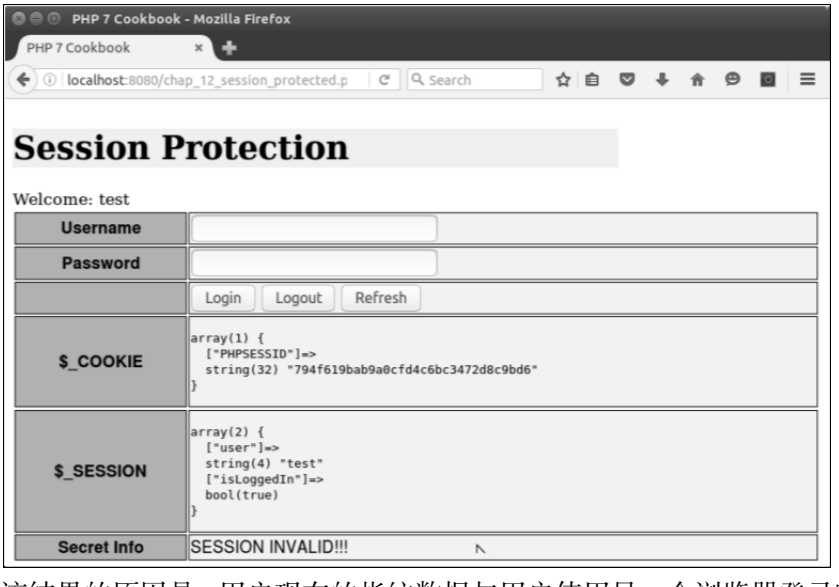
还可以添加注销选项并实现适当的注销过程：复位$_SESSION 变量，使会话作废，使 cookie 过期并失效。还可以删除存储指纹信息的文件并实现重定向：

```
} elseif (isset($_POST['logout'])) {
  session_unset();
  session_destroy();
  setcookie('PHPSESSID', 0, time() - 3600);
  if (file_exists(THUMB_PRINT_DIR . $remotePrint))
    unlink(THUMB_PRINT_DIR . $remotePrint);
    header('Location: ' . $_SERVER['REQUEST_URI'] );
exit;
```

如果用户的操作既不是登录也不是注销，则应核对用户的指纹数据，如果新的指纹数据与原始记录的指纹数据不符，那么就应将该会话判定为非法，并采取适当措施：

```
} elseif ($loggedIn && !$printsMatch) {
  $info = 'SESSION INVALID!!!';
  error_log('Session Invalid: ' . date('Y-m-d H:i:s'), 0);
  // 采取适当措施
}
```

现在可使用相同的步骤运行我们刚刚创建的 `chap_12_session_protected.php` 文件。你会看到的第一件事情是现在该会话已经被判定为非法。下面是该程序的输出结果：



出现该结果的原因是，用户现在的指纹数据与用户使用另一个浏览器登录时的指纹数据不相同。同理，如果你刷新第一个浏览器，该 PHP 程序就会重新生成一个会话标识符，从而使先前被复制的会话标识符都作废。最后，logout（注销）按钮会彻底清除会话信息。

扩展

要详细了解网站漏洞，请浏览 `https://www.owasp.org/index.php/Category:Vulnerability`。

要详细了解会话劫持攻击，请浏览 https://www.owasp.org/index.php/Session_hijacking_attack。

通过令牌提高表单的安全性

本节介绍另一种非常简单的技巧，它可以保护表单免受**跨网站伪造请求**（Cross Site Request Forgery，CSRF）攻击。简言之，当攻击者能够控制你网站中的某个页面（可能通过其他攻击手段）时，他就能够发起 CSRF 攻击。在大多数情况中，被控制的页面会使用已登录的合法用户的凭据发送请求（即使用 JavaScript 脚本购物或更改设置），而应用程序极难检测出这类行为。一种简单的应对措施是生成一种随机令牌，将这种令牌包含在每个会被提交的表单中。因为被控制的页面不会接触令牌，而且也无法生成一块相同的令牌，因此该页面提交的表单就无法通过验证。

具体处理过程

1. 为了了解具体问题，我们创建一个网页来模拟被控制页面，生成向数据库发送的请求。将该页面文件命名为 chap_12_form_csrf_test_unprotected.html：

```
<!DOCTYPE html>
  <body onload="load()">
  <form action="/chap_12_form_unprotected.php"
    method="post" id="csrf_test" name="csrf_test">
    <input name="name" type="hidden" value="No Goodnick" />
    <input name="email" type="hidden" value="malicious@owasp.org" />
    <input name="comments" type="hidden"
       value="Form is vulnerable to CSRF attacks!" />
    <input name="process" type="hidden" value="1" />
  </form>
  <script>
    function load() { document.forms['csrf_test'].submit(); }
  </script>
</body>
</html>
```

2. 创建脚本文件 chap_12_form_unprotected.php，使用该文件回应表单发出的请求。像本书介绍的其他调用程序一样，应为其设置类自动加载功能，并引用第 5

章介绍的 Application\Database\Connection 类：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Connection;
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
```

3. 确认 process 按钮已经被单击，并使用本章前面介绍的方式实现过滤机制。这可以证明 CSRF 攻击能够轻松绕过过滤器：

```
if ($_POST['process']) {
    $filter = [
      'trim' => function ($item) { return trim($item); },
      'email' => function ($item) {
        return filter_var($item, FILTER_SANITIZE_EMAIL); },
      'length' => function ($item, $length) {
        return substr($item, 0, $length); },
      'stripTags' => function ($item) {
        return strip_tags($item); },
  ];

  $assignments = [
    '*' => ['trim' => NULL, 'stripTags' => NULL],
    'email' => ['length' => 249, 'email' => NULL],
    'name' => ['length' => 128],
    'comments'=> ['length' => 249],
  ];

  $data = $_POST;
  foreach ($data as $field => $item) {
    foreach ($assignments['*'] as $key => $option) {
      $item = $filter[$key]($item, $option);
    }
    if (isset($assignments[$field])) {
      foreach ($assignments[$field] as $key => $option) {
        $item = $filter[$key]($item, $option);
      }
      $filteredData[$field] = $item;
  }
}
```

4. 使用准备语句，将经过过滤的数据插入数据库中。然后，重定向到另一个脚本：chap_12_form_view_results.php，该脚本仅会显示 visitors 表的内容：

```
try {
     $filteredData['visit_date'] = date('Y-m-d H:i:s');
     $sql = 'INSERT INTO visitors '
         . ' (email,name,comments,visit_date) '
         . 'VALUES (:email,:name,:comments,:visit_date)';
     $insertStmt = $conn->pdo->prepare($sql);
     $insertStmt->execute($filteredData);
} catch (PDOException $e) {
     echo $e->getMessage();
}
}
header('Location: /chap_12_form_view_results.php');
exit;
```

5. 结果当然是攻击可以成功实施，尽管我们使用了过滤机制和准备语句。

6. 实现表单防护令牌实际上非常简单！首先生成令牌并将之存储到会话中。可利用 PHP 7 新增的 `random_bytes()` 函数生成真正的随机令牌，仿制该令牌的难度非常高：

```
session_start();
$token = urlencode(base64_encode((random_bytes(32))));
$_SESSION['token'] = $token;
```

`random_bytes()` 函数的输出结果为二进制数据。可使用 `base64_encode()` 函数将二进制数据转换为字符串。然后，使用 `urlencode()` 函数进一步处理该字符串，以便使其能够在 HTML 表单中显示出来。

7. 在显示表单时，可将令牌设置为表单的隐藏单元格：

```
<input type="hidden" name="token" value="<?= $token ?>" />
```

8. 修改前面介绍的 `chap_12_form_unprotected.php` 脚本，添加初次检查逻辑，以便查明表单的令牌是否与会话中存储的令牌相同。注意，我们已经复位了当前令牌以使其失效。将新脚本命名为 `chap_12_form_protected_with_token.php`：

```
if ($_POST['process']) {
     $sessToken = $_SESSION['token'] ?? 1;
     $postToken = $_POST['token'] ?? 2;
     unset($_SESSION['token']);
     if ($sessToken != $postToken) {
          $_SESSION['message'] = 'ERROR: token mismatch';
     } else {
```

```
            $_SESSION['message'] = 'SUCCESS: form processed';
            // 继续处理表单
        }
    }
```

具体运行情况

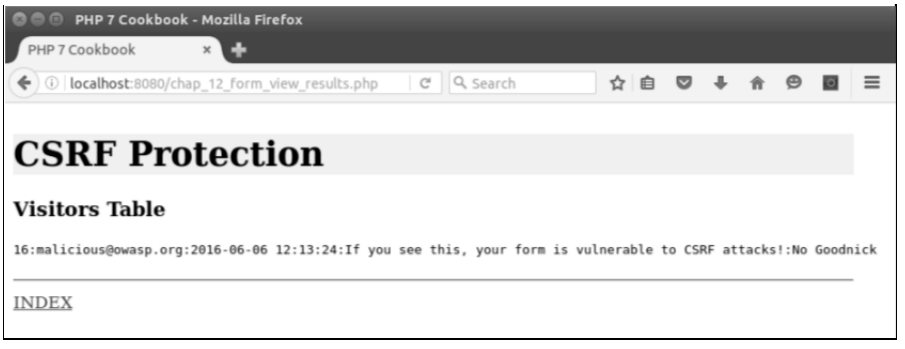
为了了解通过受控制网页实施 CSRF 攻击的方式，可使用前面介绍的代码创建下列文件：

- chap_12_form_csrf_test_unprotected.html
- chap_12_form_unprotected.php

定义 chap_12_form_view_results.php 文件，使用该文件显示 visitors 表：

```
<?php
session_start();
define('DB_CONFIG_FILE', '/../config/db.config.php');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Database\Connection;
$conn = new Connection(include __DIR__ . DB_CONFIG_FILE);
$message = $_SESSION['message'] ?? '';
unset($_SESSION['message']);
$stmt = $conn->pdo->query('SELECT * FROM visitors');
?>
<!DOCTYPE html>
<body>
<div class="container">
  <h1>CSRF Protection</h1>
  <h3>Visitors Table</h3>
  <?php while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) : ?>
  <pre><?php echo implode(':', $row); ?></pre>
  <?php endwhile; ?>
  <?php if ($message) : ?>
  <b><?= $message; ?></b>
  <?php endif; ?>
</div>
</body>
</html>
```

使用浏览器打开 chap_12_form_csrf_test_unprotected.html 文件，下面是输出结果：



如上图所示,尽管我们使用了过滤机制和准备语句,但CSRF攻击仍旧成功实施了!

将 chap_12_form_unprotected.php 文件中的代码复制到 chap_12_form_protected.php 文件中。按照步骤 8 介绍的方式修改这些代码。将用于测试的 HTML 文件 chap_12_form_csrf_test_unprotected.html 中的代码复制到 chap_12_form_csrf_test_protected.html 文件中。更改 FORM 标签中 action 参数的值:

```
<form action="/chap_12_form_protected_with_token.php"
  method="post" id="csrf_test" name="csrf_test">
```

当你使用浏览器运行新的 HTML 文件时,该文件会调用 chap_12_form_protected.php 脚本,该脚本会查找一个不存在的令牌。下面是输出结果:

定义 called chap_12_form_protected.php 文件,使该文件生成令牌并将其设置为隐藏的表单元素:

```
<?php
session_start();
$token = urlencode(base64_encode((random_bytes(32))));
$_SESSION['token'] = $token;
?>
<!DOCTYPE html>
```

```
<body onload="load()">
<div class="container">
<h1>CSRF Protected Form</h1>
<form action="/chap_12_form_protected_with_token.php"
   method="post" id="csrf_test" name="csrf_test">
<table>
<tr><th>Name</th><td><input name="name" type="text" /></td></tr>
<tr><th>Email</th><td><input name="email" type="text" /></td></tr>
<tr><th>Comments</th><td>
<input name="comments" type="textarea" rows=4 cols=80 />
</td></tr>
<tr><th> </th><td>
<input name="process" type="submit" value="Process" />
</td></tr>
</table>
<input type="hidden" name="token" value="<?= $token ?>" />
</form>
<a href="/chap_12_form_view_results.php">
   CLICK HERE</a> to view results
</div>
</body>
</html>
```

当通过表单显示和提交数据时，如果令牌通过了验证，那么插入数据的操作就会被允许执行，如下所示：

扩展

要详细了解 CSFR 攻击，请浏览 https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)。

创建具有较高安全性的密码生成器

一个常见的错误观点是，攻击者破解散列化密码的唯一方式是使用**穷举攻击**（brute-force attack）和**彩虹表**。尽管这通常是攻击过程中的第一步，但攻击者会在攻击过程的第 2 步、第 3 步或第 4 步中使用更复杂的攻击手段。其他攻击手段包括字典、组合、掩码和基于规则的攻击。字典攻击使用存储了词典内容的数据库，猜测密码。组合攻击会组合使用词典中的单词。掩码攻击与穷举攻击类似，但是其选择性更强，因此能够减少破解密码的时间。基于规则的攻击利用编码规则破解密码（如使用数字 0 代替字母 o）。

好消息是只需增加密码长度，使其超过魔幻的 6 位字符限制，就能够以指数形式增加破解散列化密码所需的时间。此外，在密码中无规则地交叉使用大小写字母、随机数字和特殊字符，也能够以指数形式增加破解密码所需的时间。最后，我们还应牢记密码是要被输入到计算机中的，因此这些密码至少应该能够被记住。

最佳编程习惯

密码应以散列值的形式存储，而且永远不应将密码存储为普通文本。MD5 和 SHA*算法已经被公认为是不再安全的算法（尽管 SHA*算法的安全性比 MD5 算法的安全性高得多）。对于使用 MD5 算法散列化的密码，攻击者可以使用密码破解工具（如 `oclHashcat`）对其执行平均每秒 55 亿次的破解尝试（通过成功的 SQL 注入攻击）。

具体处理过程

1. 定义 `Application\Security\PassGen` 类，使用该类存储用于生成密码的方法。还应定义相关的类常量和属性：

```
namespace Application\Security;
class PassGen
{
  const SOURCE_SUFFIX = 'src';
  const SPECIAL_CHARS =
    '\`¬|!"£$%^&*()_-+={}[]:@~;\'#<>?,./|\\';
  protected $algorithm;
```

```
  protected $sourceList;
  protected $word;
  protected $list;
```

2. 定义在密码生成过程中使用的低等级方法。顾名思义，digits()方法用于生成随机数字，special()方法用于通过 SPECIAL_CHARS 类常量生成单个字符：

```
public function digits($max = 999)
{
  return random_int(1, $max);
}

public function special()
{
  $maxSpecial = strlen(self::SPECIAL_CHARS) - 1;
  return self::SPECIAL_CHARS[random_int(0, $maxSpecial)];
}
```

本例会频繁使用 PHP 7 的新增函数 random_int()。尽管该函数稍微慢一点，但与较旧的 rand()函数相比，该函数提供了真正的**具有密码学安全性的伪随机数生成器**（Cryptographically Secure Pseudo Random Number Generator，CSPRNG）功能。

3. 下面介绍有难度的部分：生成一个难以被猜到的单词。这是构造器参数 $wordSource 负责完成的任务。我们通过一系列网站创建我们的词库。因此，首先需要创建一个方法，使该方法从指定的数据源获取具有唯一性的单词列表，并把得到的结果存储到一个文件中。应使该方法将$wordSource 数组接收为参数，并循环遍历每个 URL。可使用 md5()函数通过网站的名称生成散列值，然后使用该散列值创建文件名。将新生成的文件名存储到$sourceList 变量中：

```
public function processSource(
$wordSource, $minWordLength, $cacheDir)
{
  foreach ($wordSource as $html) {
    $hashKey = md5($html);
    $sourceFile = $cacheDir . '/' . $hashKey . '.'
    . self::SOURCE_SUFFIX;
    $this->sourceList[] = $sourceFile;
```

4. 如果该文件不存在或者为空，那么就可以为该文件处理内容。如果数据源是

HTML 文件，那么只应获取<body>标签内部的内容，然后使用 str_word_count() 函数从$contents 变量存储的字符串中获取单词,同时不要忘记先使用 strip_tags() 函数去掉该字符串中的标记：

```
if (!file_exists($sourceFile) || filesize($sourceFile) == 0) {
    echo 'Processing: ' . $html . PHP_EOL;
    $contents = file_get_contents($html);
    if (preg_match('/<body>(.*)<\/body>/i',
        $contents, $matches)) {
        $contents = $matches[1];
    }
    $list = str_word_count(strip_tags($contents), 1);
```

5. 去掉过短的单词，并使用 array_unique() 函数去掉重复的单词。将获得的最终结果存储到一个文件中：

```
foreach ($list as $key => $value) {
  if (strlen($value) < $minWordLength) {
    $list[$key] = 'xxxxxx';
  } else {
    $list[$key] = trim($value);
  }
}
     $list = array_unique($list);
     file_put_contents($sourceFile, implode("\n",$list));
    }
  }
  return TRUE;
}
```

6. 定义一个方法，使用该方法在单词中随机选定一些字母，并将这些字母转换为大写形式：

```
public function flipUpper($word)
{
  $maxLen = strlen($word);
  $numFlips = random_int(1, $maxLen - 1);
  $flipped = strtolower($word);
  for ($x = 0; $x < $numFlips; $x++) {
       $pos = random_int(0, $maxLen - 1);
       $word[$pos] = strtoupper($word[$pos]);
  }
  return $word;
}
```

7. 定义从数据源选择单词的方法。通过随机方式选中单词库，然后使用 file() 函数从适当的词库文件读取数据：

```
public function word()
{
  $wsKey    = random_int(0, count($this->sourceList) - 1);
  $list     = file($this->sourceList[$wsKey]);
  $maxList  = count($list) - 1;
  $key      = random_int(0, $maxList);
  $word     = $list[$key];
  return $this->flipUpper($word);
}
```

8. 这样我们就不会总是用同一种模式生成密码了。定义一个方法，用来调整密码各个组成部分的位置。将生成密码各个部分的算法定义为该类中的一组方法。例如，以 ['word','digits','word','special'] 结构（即[单词，数字，单词，特殊字符]结构）生成密码的算法，会生成类似 hElLo123aUTo!的密码：

```
public function initAlgorithm()
{
  $this->algorithm = [
    ['word', 'digits', 'word', 'special'],
    ['digits', 'word', 'special', 'word'],
    ['word', 'word', 'special', 'digits'],
    ['special', 'word', 'special', 'digits'],
    ['word', 'special', 'digits', 'word', 'special'],
    ['special', 'word', 'special', 'digits',
    'special', 'word', 'special'],
  ];
}
```

9. 应使下面的构造器将存储词库的数组、单词长度下限和存储词库文件的目录接收为参数。使该构造器能够处理词库文件和初始化算法：

```
public function __construct(
  array $wordSource, $minWordLength, $cacheDir)
{
  $this->processSource($wordSource, $minWordLength, $cacheDir);
  $this->initAlgorithm();
}
```

10. 定义执行生成密码操作的方法。该方法只需以随机方式选择算法，然后循环调用合适的方法：

```
public function generate()
{
  $pwd = ' ';
  $key = random_int(0, count($this->algorithm) - 1);
  foreach ($this->algorithm[$key] as $method) {
    $pwd .= $this->$method();
  }
  return str_replace("\n", ' ', $pwd);
}

}
```

具体运行情况

将前面介绍的代码添加到 Application\Security 文件夹中的 PassGen.php 文件中。创建调用程序 chap_12_password_generate.php，并为其设置类自动加载功能，引用 PassGen 类，并定义存储词库文件的目录：

```
<?php
define('CACHE_DIR', __DIR__ . '/cache');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Security\PassGen;
```

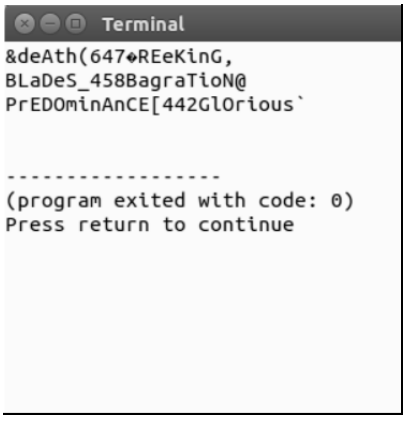
定义一组网站，将这些网站用作获取密码制作材料（词库）的数据源。本例选择古登堡计划提供的《尤利西斯》（詹姆斯·乔伊斯著）、《战争与和平》（列夫·托尔斯泰著）和《傲慢与偏见》（简·奥斯丁著）电子版小说，作为创建词库的数据源：

```
$source = [
  'https://www.gutenberg.org/files/4300/4300-0.txt',
  'https://www.gutenberg.org/files/2600/2600-h/2600-h.htm',
  'https://www.gutenberg.org/files/1342/1342-h/1342-h.htm',
];
```

创建 PassGen 实例并调用 generate() 方法：

```
$passGen = new PassGen($source, 4, CACHE_DIR);
echo $passGen->generate();
```

下面是通过 PassGen 实例生成的几个密码：



扩展

要详细了解攻击者破解密码的手段，请浏览 http://arstechnica.com/security/2013/05/how-crackers-make-mincedmeat-out-of-your-passwords/。要详细了解穷举攻击，请浏览 https://www.owasp.org/index.php/Brute_force_attack。要详细了解密码破解工具 oclHashcat，请浏览 http://hashcat.net/oclhashcat/。

通过验证码为表单提供安全防护

验证码（CAPTCHA）实际上是 Completely Automated Public Turing Test to Tell Computers and Humans Apart（**全自动区分计算机和人类的图灵测试**）的首字母缩写词。该技巧与前面介绍的令牌技巧类似。它们之间的区别是，验证码技巧不会将令牌存储在隐藏的表单单元格中，而是通过图片的方式显示令牌，而这种图片是自动化攻击系统难以识别的。而且，使用验证码的目的与表单令牌也略有不同：验证码专门用于确认浏览网页的是人类，而不是自动化系统。

具体处理过程

1. 可通过多种方式实现验证码：根据只有人类才能掌握的知识提出问题、字谜和需要拥有抽象思维才能理解的图像。

2. 通过图像实现验证码的方式是指，为网页浏览者显示含有被严重扭曲的字母和/或数字的图像。然而，该方式可能会很复杂，而且需要依靠 GD 扩展才能实现，然而并非所有服务器上都有 GD 扩展。GD 扩展难以编译，而且严重依赖各种软件库。

3. 通过字谜实现验证码的方式是指，为网页浏览者显示一组字母和/或数字，并下达一个简单的指令，如请反向键入这些内容。另一种字谜验证码的实现形式是使用 ASCII 码创作只有人类才能认出的字符绘画。

4. 也可以通过问答方式实现验证码，如头部与身体相连的部位叫什么，并列出备选答案：胳膊、腿和脖子。这种方式的缺点，是自动化攻击系统有三分之一的机会能够通过测试。

生成字谜验证码

1. 为了了解具体处理步骤，让我们先做字谜验证码实验，然后做图像验证码实验。不论做哪个实验，都需要先定义用于生成验证码内容的类。本例定义 `Application\Captcha\Phrase` 类，同时还应定义相关的属性和类常量：

```
namespace Application\Captcha;
class Phrase
{
  const DEFAULT_LENGTH  = 5;
  const DEFAULT_NUMBERS = '0123456789';
  const DEFAULT_UPPER   = 'ABCDEFGHJKLMNOPQRSTUVWXYZ';
  const DEFAULT_LOWER   = 'abcdefghijklmnopqrstuvwxyz';
  const DEFAULT_SPECIAL =
      '¬\`|!"£$%^&*()_-+={}[]:;@\'~#<,>.?/|\\';
  const DEFAULT_SUPPRESS = ['O','l'];

  protected $phrase;
  protected $includeNumbers;
  protected $includeUpper;
  protected $includeLower;
  protected $includeSpecial;
  protected $otherChars;
  protected $suppressChars;
  protected $string;
  protected $length;
```

2. 应使下面的构造器接收各种属性值，从而在获得已分配默认值的情况下，无须设

置任何参数就能够创建实例。$include*变量用于选择字符集合，以便创建用于生成验证码的基础字符串(例如,如果你只想使用数字创建密码,那么就可以将$includeUpper（代表大写字母）和$includeLower 变量（代表小写字母）都设置为 FALSE)；$otherChars 变量用于提供更多灵活性。$suppressChars 变量代表将要从基础字符串中移除的字符。默认设置为从基础字符串中移除大写字母 O 和小写字母 l：

```
public function __construct(
  $length = NULL,
  $includeNumbers = TRUE,
  $includeUpper= TRUE,
  $includeLower= TRUE,
  $includeSpecial = FALSE,
  $otherChars = NULL,
  array $suppressChars = NULL)
  {
    $this->length = $length ?? self::DEFAULT_LENGTH;
    $this->includeNumbers = $includeNumbers;
    $this->includeUpper = $includeUpper;
    $this->includeLower = $includeLower;
    $this->includeSpecial = $includeSpecial;
    $this->otherChars = $otherChars;
    $this->suppressChars = $suppressChars
      ?? self::DEFAULT_SUPPRESS;
    $this->phrase = $this->generatePhrase();
}
```

3. 为每个属性都设置一对设置器和读取器。为节省篇幅此处仅列出了一个属性的设置器和读取器。

```
public function getString()
{
  return $this->string;
}

public function setString($string)
{
  $this->string = $string;
}

// 此处没有列出其他属性的设置器和读取器
```

4. 定义一个用于初始化基础字符串的方法。应在该方法中包含一系列简单的 if 语句，使用这些 if 语句检查各种$include*变量，并根据需要增加基础字符串的内容。

使用 str_replace()函数去掉基础字符串中与$suppressChars 字符串相同的内容，即在基础字符串$string 中查找与$suppressChars 变量中存储的字符串相同的字符串，找到这些字符串后使用空格替换它们：

```
public function initString()
{
  $string = ' ';
  if ($this->includeNumbers) {
      $string .= self::DEFAULT_NUMBERS;
  }
  if ($this->includeUpper) {
      $string .= self::DEFAULT_UPPER;
  }
  if ($this->includeLower) {
      $string .= self::DEFAULT_LOWER;
  }
  if ($this->includeSpecial) {
      $string .= self::DEFAULT_SPECIAL;
  }
  if ($this->otherChars) {
      $string .= $this->otherChars;
  }
  if ($this->suppressChars) {
      $string = str_replace(
          $this->suppressChars, '', $string);
  }
  return $string;
}
```

最佳编程习惯

应去除外形与数字非常相似的字母，如字母 O 经常会被误认为是数字 0，而小写字母 l 常常会被误认为数字 1。

5. 定义核心方法来生成随机短语，即显示给网站浏览者的验证码。可设置一个简单的 for()循环，使用 PHP 7 新增的 random_int()函数，以随机方式在基础字符串中选取字符：

```
public function generatePhrase()
{
```

```
    $phrase = ' ';
    $this->string = $this->initString();
    $max = strlen($this->string) - 1;
    for ($x = 0; $x < $this->length; $x++) {
      $phrase .= substr(
        $this->string, random_int(0, $max), 1);
    }
    return $phrase;
  }
  }
```

6. 让我们将注意力从验证码短语转向用于生成字谜验证码的类上。应先定义一个接口，这样将来创建其他用于生成验证码的类时，就都可以利用 `Application\Captcha\Phrase` 类了。注意，`getImage()` 方法应根据我们选择使用的类返回字符、字符绘画或真正的图像：

```
namespace Application\Captcha;
interface CaptchaInterface
{
  public function getLabel();
  public function getImage();
  public function getPhrase();
}
```

7. 为了创建字谜验证码，可定义 `Application\Captcha\Reverse` 类。将该类命名为 `Reverse`（代表反向）的原因是，这个类不仅会生成验证码的文本，而且会以反转的方向生成该文本。该类中的 `__construct()` 会创建 `Phrase` 实例。注意，`getImage()` 方法会以反转的方向返回这个短语（通过 `Phrase` 实例生成的）：

```
namespace Application\Captcha;
class Reverse implements CaptchaInterface
{
  const DEFAULT_LABEL = 'Type this in reverse';
  const DEFAULT_LENGTH = 6;
  protected $phrase;
  public function __construct(
    $label = self::DEFAULT_LABEL,
    $length = self:: DEFAULT_LENGTH,
    $includeNumbers = TRUE,
    $includeUpper   = TRUE,
    $includeLower   = TRUE,
```

```
    $includeSpecial = FALSE,
    $otherChars     = NULL,
    array $suppressChars = NULL)
  {
    $this->label    = $label;
    $this->phrase   = new Phrase(
      $length,
      $includeNumbers,
      $includeUpper,
      $includeLower,
      $includeSpecial,
      $otherChars,
      $suppressChars);
    }

  public function getLabel()
  {
    return $this->label;
  }

  public function getImage()
  {
    return strrev($this->phrase->getPhrase());
  }

  public function getPhrase()
  {
    return $this->phrase->getPhrase();
  }
}
```

生成图像验证码

1. 无须赘言，生成图像验证码的方式比生成字谜验证码的方式要复杂得多。生成验证码短语的过程是相同的，主要区别是，图像验证码不仅需要将验证码短语嵌入图像中，而且还需要通过不同的方式扭曲每个字母，并使用圆点增加图像干扰效果。

2. 定义实现了 `CaptchaInterface` 接口的 `Application\Captcha\Image` 类。该类中的常量和属性不仅用于生成验证码短语，而且还需要能够用于生成图像：

```
namespace Application\Captcha;
use DirectoryIterator;
class Image implements CaptchaInterface
```

```
{

  const DEFAULT_WIDTH = 200;
  const DEFAULT_HEIGHT = 50;
  const DEFAULT_LABEL = 'Enter this phrase';
  const DEFAULT_BG_COLOR = [255,255,255];
  const DEFAULT_URL = '/captcha';
  const IMAGE_PREFIX = 'CAPTCHA_';
  const IMAGE_SUFFIX = '.jpg';
  const IMAGE_EXP_TIME = 300; // 单位为秒
  const ERROR_REQUIRES_GD = 'Requires the GD extension + '
    . ' the JPEG library';
  const ERROR_IMAGE = 'Unable to generate image';
  protected $phrase;
  protected $imageFn;
  protected $label;
  protected $imageWidth;
  protected $imageHeight;
  protected $imageRGB;
  protected $imageDir;
  protected $imageUrl;
```

3. 该类的构造器需要接收所有用于生成验证码短语的参数（请参阅前面的内容），此外，还应该能够接收用于生成图像的参数。两个必要参数是`$imageDir` 和`$imageUrl`，第一个参数用于存储图像的位置，第二个参数是基础 URL，我们可以在其中添加生成的文件名。当需要使用 TrueType 字体时可增加`$imageFont` 参数，TrueType 字体可以生成安全性更高的验证码。如果不使用`$imageFont` 参数，就仅能使用默认字体，那么得到的图像就不会很漂亮：

```
public function __construct(
  $imageDir,
  $imageUrl,
  $imageFont = NULL,
  $label = NULL,
  $length = NULL,
  $includeNumbers = TRUE,
  $includeUpper= TRUE,
  $includeLower= TRUE,
  $includeSpecial = FALSE,
  $otherChars = NULL,
  array $suppressChars = NULL,
  $imageWidth = NULL,
```

```
    $imageHeight = NULL,
    array $imageRGB = NULL
)
{
```

4. 在这个构造器中，还应该检查 imagecreatetruecolor 函数是否存在。如果检查结果为 FALSE（代表该函数不存在），就说明无法使用 GD 扩展。否则，就应将参数值赋予属性，生成验证码短语，删除旧图像，并创建验证码图像：

```
if (!function_exists('imagecreatetruecolor')) {
    throw new \Exception(self::ERROR_REQUIRES_GD);
}
$this->imageDir    = $imageDir;
$this->imageUrl    = $imageUrl;
$this->imageFont   = $imageFont;
$this->label       = $label ?? self::DEFAULT_LABEL;
$this->imageRGB    = $imageRGB ?? self::DEFAULT_BG_COLOR;
$this->imageWidth  = $imageWidth ?? self::DEFAULT_WIDTH;
$this->imageHeight = $imageHeight ?? self::DEFAULT_HEIGHT;
if (substr($imageUrl, -1, 1) == '/') {
    $imageUrl  = substr($imageUrl, 0, -1);
}
$this->imageUrl = $imageUrl;
if (substr($imageDir, -1, 1) == DIRECTORY_SEPARATOR) {
    $imageDir = substr($imageDir, 0, -1);
}

$this->phrase = new Phrase(
  $length,
  $includeNumbers,
  $includeUpper,
  $includeLower,
  $includeSpecial,
  $otherChars,
  $suppressChars);
$this->removeOldImages();
$this->generateJpg();
}
```

5. 删除旧图像的处理过程极为重要，因为如果不删除旧图像，那么存储验证码图像的目录就会被过期的验证码图像塞满！可使用 DirectoryIterator 类扫描指定的目

录并检查访问时间。将当前时间减去由 IMAGE_EXP_TIME 常量设置的有效期，我们据此差值来判定一幅图像是否为过期的旧图像：

```
public function removeOldImages()
{
  $old = time() - self::IMAGE_EXP_TIME;
  foreach (new DirectoryIterator($this->imageDir)
           as $fileInfo) {
    if($fileInfo->isDot()) continue;
    if ($fileInfo->getATime() < $old) {
      unlink($this->imageDir . DIRECTORY_SEPARATOR
             . $fileInfo->getFilename());
    }
  }
}
```

6. 下面创建主要方法。先将$imageRGB 数组分解为$red、$green 和$blue 变量，再使用核心的 imagecreatetruecolor()函数生成指定宽度和高度的基础图像，并使用 RGB 值为背景添加颜色：

```
public function generateJpg()
{
  try {
       list($red,$green,$blue) = $this->imageRGB;
       $im = imagecreatetruecolor(
         $this->imageWidth, $this->imageHeight);
       $black = imagecolorallocate($im, 0, 0, 0);
       $imageBgColor = imagecolorallocate(
         $im, $red, $green, $blue);
       imagefilledrectangle($im, 0, 0, $this->imageWidth,
         $this->imageHeight, $imageBgColor);
```

7. 根据图像的宽度和高度，定义 *x* 轴和 *y* 轴方向的页边空白。然后初始化用于将验证码短语写入图像中的变量。根据验证码短语的长度，循环执行向图像中写入字符的操作：

```
$xMargin = (int) ($this->imageWidth * .1 + .5);
$yMargin = (int) ($this->imageHeight * .3 + .5);
$phrase = $this->getPhrase();
$max = strlen($phrase);
$count = 0;
$x = $xMargin;
$size = 5;
for ($i = 0; $i < $max; $i++) {
```

8. 如果$imageFont 变量的值被设定了，我们就可以通过不同的尺寸和扭曲角度写入字符。而且还需要根据尺寸调整 *x* 轴（即水平）方向的值：

```
if ($this->imageFont) {
    $size = rand(12, 32);
    $angle = rand(0, 30);
    $y = rand($yMargin + $size, $this->imageHeight);
    imagettftext($im, $size, $angle, $x, $y, $black,
      $this->imageFont, $phrase[$i]);
    $x += (int) ($size + rand(0,5));
```

9. 如果$imageFont 变量的值没有被设定就只能使用默认字体了。我们使用 5 号字号，因为更小字号的字符难以阅读。可通过交替使用 imagechar() 和 imagecharup() 函数获得等级较低的扭曲效果，imagechar() 函数会以正常方式向图像中写入字符，而 imagecharup() 函数会以倾斜方式向图像中写入字符：

```
  } else {
      $y = rand(0, ($this->imageHeight - $yMargin));
      if ($count++ & 1) {
          imagechar($im, 5, $x, $y, $phrase[$i], $black);
      } else {
          imagecharup($im, 5, $x, $y, $phrase[$i], $black);
      }
      $x += (int) ($size * 1.2);
    }
  } // for ($i = 0; $i < $max; $i++)循环结束
```

10. 通过随机向图像中添加圆点增加干扰效果。这是一个必要处理步骤，用于使图像更难以被自动化系统识别。在图像中添加一些线条会有更好的干扰效果：

```
$numDots = rand(10, 999);
for ($i = 0; $i < $numDots; $i++) {
  imagesetpixel($im, rand(0, $this->imageWidth),
    rand(0, $this->imageHeight), $black);
}
```

11. 使用我们的老朋友 md5() 函数和日期以及一个 0 至 9999 之间的随机数字，创建随机的图像文件名。注意，此处我们可以安全地使用 md5() 函数，因为我们没有将该函数用于隐藏机密信息；我们使用该函数仅是为了快速地生成具有唯一性的文件名。创建了图像文件后，就可以销毁图像对象，从而释放它占用的内存：

```
$this->imageFn = self::IMAGE_PREFIX
. md5(date('YmdHis') . rand(0,9999))
. self::IMAGE_SUFFIX;
```

```
imagejpeg($im, $this->imageDir . DIRECTORY_SEPARATOR
. $this->imageFn);
imagedestroy($im);
```

12. 生成图像的所有操作都应放在 try/catch 代码块中。如果程序抛出了错误或异常，就应该将该消息记录下来，并采取适当措施：

```
} catch (\Throwable $e) {
    error_log(__METHOD__ . ':' . $e->getMessage());
    throw new \Exception(self::ERROR_IMAGE);
}
}
```

13. 最后，定义 CaptchaInterface 接口所必需的方法。注意，getImage()方法应返回 HTML 中的<img>标签，从而使图像能够立刻显示：

```
public function getLabel()
{
  return $this->label;
}

public function getImage()
{
  return sprintf('<img src="%s/%s" />',
    $this->imageUrl, $this->imageFn);
}

public function getPhrase()
{
  return $this->phrase->getPhrase();
}
}
```

具体运行情况

使用前面介绍的代码定义下列类：

类	描述	对应的步骤
Application\Captcha\Phrase	生成验证码短语	1 ~ 5
Application\Captcha\CaptchaInterface		6
Application\Captcha\Reverse		7
Application\Captcha\Image	生成图像验证码	2 ~ 13

定义调用程序 chap_12_captcha_text.php，使用该程序生成字谜验证码。先为其设置类自动加载功能，然后引用合适的类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Captcha\Reverse;
```

确保启动了会话。当然，还应该使用适当的手段为会话提供保护。为节省篇幅，此处仅介绍一种简单的手段（session_regenerate_id()函数）：

```
session_start();
session_regenerate_id();
```

定义用于创建验证码短语的函数；获取短语、字符绘画和图像（在本例中为字符反向排列的短语）；并将验证码短语存储到会话中：

```
function setCaptcha(&$phrase, &$label, &$image)
{
  $captcha  = new Reverse();
  $phrase   = $captcha->getPhrase();
  $label    = $captcha->getLabel();
  $image    = $captcha->getImage();
  $_SESSION['phrase'] = $phrase;
}
```

初始化变量并确定$loggedIn 变量的状态：

```
$image      = ' ';
$label      = ' ';
$phrase     = $_SESSION['phrase'] ?? ' ';
$message    = ' ';
$info       = 'You Can Now See Super Secret Information!!!';
$loggedIn   = $_SESSION['isLoggedIn'] ?? FALSE;
$loggedUser = $_SESSION['user'] ?? 'guest';
```

这样就可以查明 login 按钮（登录按钮）是否已经被单击过。如果该按钮已经被单击过，应检查用户是否已经键入验证码。如果用户没有键入验证码，就应该显示一条消息，提示用户必须输入验证码：

```
if (!empty($_POST['login'])) {
  if (empty($_POST['captcha'])) {
    $message = 'Enter Captcha Phrase and Login Information';
```

如果用户已经键入了验证码，应检查用户输入的验证码是否与会话中存储的验证码

相同。如果二者不同，就将表单提交内容视为非法。否则，就按照正常步骤处理登录过程。为了演示具体步骤，我们使用硬编码的用户名和密码模拟登录过程：

```
    } else {
        if ($_POST['captcha'] == $phrase) {
            $username = 'test';
            $password = 'password';
            if ($_POST['user'] == $username
                && $_POST['pass'] == $password) {
                $loggedIn = TRUE;
                $_SESSION['user'] = strip_tags($username);
                $_SESSION['isLoggedIn'] = TRUE;
            } else {
                $message = 'Invalid Login';
            }
        } else {
            $message = 'Invalid Captcha';
        }
}
```

还需要编写代码，以便实现注销选项，请参阅“为 PHP 会话提供安全防护”一节的内容：

```
} elseif (isset($_POST['logout'])) {
  session_unset();
  session_destroy();
  setcookie('PHPSESSID', 0, time() - 3600);
  header('Location: ' . $_SERVER['REQUEST_URI'] );
  exit;
}
```

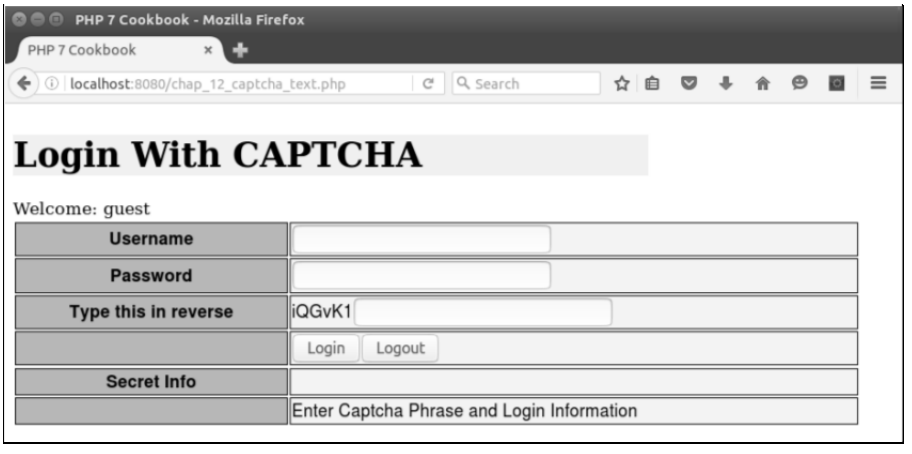
调用 `setCaptcha()` 方法：

```
setCaptcha($phrase, $label, $image);
```

最后，不要忘记添加查看逻辑，本例使用查看逻辑显示简单的登录表单。在 HTML 的 form 标签中，应添加查看逻辑以显示验证码和字符绘画：

```
<tr>
  <th><?= $label; ?></th>
  <td><?= $image; ?><input type="text" name="captcha" /></td>
</tr>
```

下面是输出结果：



为了了解使用图像验证码的方式，可将 chap_12_captcha_text.php 文件中的代码复制到 cha_12_captcha_image.php 文件中。定义代表目录的常量，该目录用于存储验证码图像（应确保创建了该目录！）。cha_12_captcha_image.php 文件的类自动加载功能和引用语句结构与 chap_12_captcha_text.php 文件类似。注意，我们还定义了 TrueType 字体，而且使用了**加粗效果**：

```
<?php
define('IMAGE_DIR', __DIR__ . '/captcha');
define('IMAGE_URL', '/captcha');
define('IMAGE_FONT', __DIR__ . '/FreeSansBold.ttf');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Captcha\Image;

session_start();
session_regenerate_id();
```

重要提示

某些字体会受版权、商标权、专利权或其他知识产权法律的保护。如果你在没有获得许可的情况下使用了某个受法律保护的字体，那么你和你的客户可能会被告上法庭！可以使用开源字体或者你已取得字体使用许可证的字体。

当然，在 setCaptcha()方法中应使用 Image 类替换 Reverse 类：

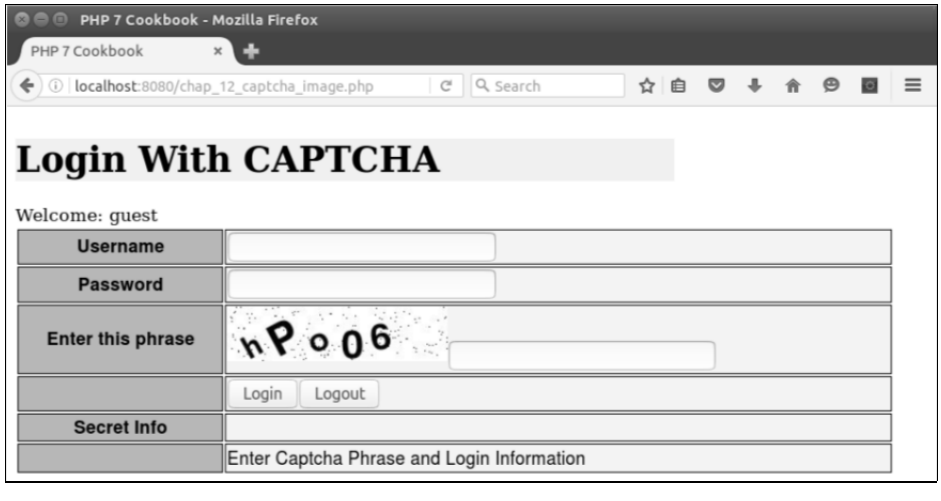
```
function setCaptcha(&$phrase, &$label, &$image)
```

```
{
  $captcha  = new Image(IMAGE_DIR, IMAGE_URL, IMAGE_FONT);
  $phrase   = $captcha->getPhrase();
  $label    = $captcha->getLabel();
  $image    = $captcha->getImage();
  $_SESSION['phrase'] = $phrase;
  return $captcha;
}
```

初始化变量的方式和登录处理过程与上一个示例相同：

```
$image      = ' ';
$label      = ' ';
$phrase     = $_SESSION['phrase'] ?? ' ';
$message    = ' ';
$info       = 'You Can Now See Super Secret Information!!!';
$loggedIn   = $_SESSION['isLoggedIn'] ?? FALSE;
$loggedUser = $_SESSION['user'] ?? 'guest';

if (!empty($_POST['login'])) {

  // 此处的代码与 chap_12_captcha_text.php 文件中处理登录过程的代码相同
```

因为我们使用了 `getImage()` 方法，在生成图像验证码的过程中，该方法会返回可直接使用的 HTML 代码，因此两个文件中的查看逻辑也相同。下面是使用了 TrueType 字体的输出结果：



补充说明

如果你不想使用前面介绍的代码生成自己制造的验证码，可使用其他用于生成验证码的软件库。大多数流行的框架都具有该功能。例如，Zend Framework 框架就拥有 Zend\Captcha 组件类。也可以通过 reCAPTCHA 计划获取验证码，通常可以将 reCAPTCHA 计划视为一种可调用的服务，即让你编写的应用程序向生成验证码和令牌的外部网站发出请求，然后获得具有安全性的验证码和令牌。要详细了解 reCAPTCHA 计划，请浏览 `http://www.captcha.net/`。

扩展

要详细了解知识产权法律对字体的保护，请浏览 `https://en.wikipedia.org/wiki/Intellectual_property_protection_of_typefaces`。

在不使用 mcrypt 加密扩展库的情况下实现加密/解密功能

大多数基于 PHP 的加密功能的核心——`mcrypt` 扩展——一点也不安全，这是不为 PHP 社区的成员所熟知的事实。从安全的角度来说，一个最大的问题是，要成功地操作 `mcrypt` 扩展就需要掌握高级的密码学知识，然而极少有程序员能够做到这一点。这就导致了 `mcrypt` 扩展被大量滥用，并最终导致密码被破解的可能性大幅度增加。此外，开发者们支持的 `libmcrypt` 库已经于 2007 年被废弃，`libmcrypt` 是 `mcrypt` 扩展的核心库，这意味着 `mcrypt` 扩展的基础代码过时了，其中含有很多 bug 而且没有应用补丁的机制。因此，应在不使用 `mcrypt` 扩展的情况下实现强力加密/解密功能，认识到这一点极为重要！

具体处理过程

1. 上述问题的解决方案是使用加密扩展 `openssl`。该扩展得到了很好的维护，并且拥有新式且非常强大的加密/解密功能。

重要提示

要使用 `openssl*`函数，必须先编译和启用 PHP 的 `openssl` 扩展！此外，还需要在你的 Web 服务器上安装最新的 OpenSSL 软件包。

2. 在安装 OpenSSL 软件包时，需要决定安装哪些生成密码的方法。为了做到这一点，可使用 `openssl_get_cipher_methods()` 命令。应安装的常见算法包括：**高级加密标准**（Advanced Encryption Standard，AES）、BlowFish（BF）、CAMELLIA、CAST5、**数据加密标准**（Data Encryption Standard，DES）、Rivest Cipher（RC，也称为 Ron's Code）和 SEED。使用 `openssl_get_cipher_methods()` 函数获取的方法列表中，会含有生成密码的方法所用算法的名称和具体模式。

3. 需要确定哪种加密方法（算法）最能满足你的需要。下表概括介绍了各种加密方法：

加密方法	发布时间	密钥长度（单位为位）	被加密数据的分组长度（单位为字节）	注释
`camellia`	2000 年	128、192、256	16	由三菱公司和日本电信电话株式会社共同发明
`aes`	1998 年	128、192、256	16	由比利时密码学家 Joan Daemen 和 Vincent Rijmen 发明。最初该算法被命名为 Rijndael
`seed`	1998 年	128	16	由韩国信息安全局（KISA）发明
`cast5`	1996 年	40 ~ 128	8	由加拿大密码学家 Carlisle Adams 和 Stafford Tavares 发明
`bf`	1993 年	1 ~ 448	8	由美国密码学家 Bruce Schneier 设计
`rc2`	1987 年	8 ~ 1024（默认长度为 64）	8	由美国密码学家 Ron Rivest（RSA 算法的核心发明者之一）设计
`des`	1977 年	56（可增加 8 位奇偶性密钥）	8	由 IBM 在德国密码学家 Horst Feistel 研究成果的基础上发明

4. 另一个需要考虑的要点是，你喜欢使用哪种数据分组加密模式。下面列出了一些常见的加密模式：

加密模式	全称	注释
ECB	电码本	**无须初始向量**（IV）；既能够加密也能够解密；简单快捷；不隐藏数据模式；不推荐使用
CBC	密码分组链接	需要使用 IV；前面的分组数据会与后面的分组数据进行异或运算，因此即使后面分组的明文内容与前面分组的明文内容完全相同，也能够获得更高的加密强度；如果 IV 被破解，那么第一个分组数据就会被破解，因而其余数据也会被破译；明文数据会被填充数据，从而使其能够根据加密方法的规定被划分为多个数据分组；仅在解密时支持并行处理方式
CFB	密码反馈	除了反向执行加密处理操作，与 CBC 模式非常相似
OFB	输出反馈	非常对称：加密处理过程与解密处理过程相同；完全不支持并行处理方式

续表

加密模式	全称	注释
CTR	计数	运行方式与 OFB 模式类似；在加密和解密处理过程中都支持并行处理方式
CCM	CBC-MAC 计数	源自 CTR 模式；专门用于处理分组长度为 128 位的明文数据；提供了验证和认证功能;CBC-MAC 是 Cipher Block Chaining - Message Authentication Code（**密码分组链接-消息认证码的首字母缩写词**）
GCM	迦罗瓦/计数（Galois/Counter）	以 CTR 模式为基础；应为每个要加密的数据流应用不同的 IV;（与其他模式相比）特别适合处理高吞吐量情况；支持以并发方式处理加密和解密过程
XTS	基于 XEX 模式的通过借用密文调整分组长度的电码本（XEX-based Tweaked-codebook mode with ciphertext Stealing）	较新（2010 年获得 NIST 证明）并且速度较快；使用两个密钥；通过借用倒数第二个分组数据的密文，填充最后一个数据分组，使该分组具有正常分组的长度，从而能够作为正常的分组被加密

5. 在选择加密方法和模式前，还需要确定是否需要在你编写的 PHP 应用程序的外部解密已经加密的内容。例如，如果你要将加密的数据库凭证存储到单个文本文件中，是否需要能够通过命令行界面解密这些内容？如果需要具有这种能力，就应确保你选择的加密方法和操作模式能够得到目标操作系统的支持。

6. 为 **IV** 提供的字节数会因你选择的加密方法的不同而不同。为了获得最佳效果，可使用 PHP 7 新增的 `random_bytes()` 函数，该函数能够返回真正的 **CSPRNG** 字节序列。IV 长度变化的幅度会非常大。在初始时可以将其设置为 16 字节。如果 PHP 引擎显示了警告信息，那么警告信息中也会显示该算法正确的 IV 长度，可根据该信息调整 IV 长度：

```
$iv = random_bytes(16);
```

7. 要执行加密操作，可使用 `openssl_encrypt()` 函数。下面列出了该函数的参数：

参数	注释
数据	需要加密的明文
方法	通过 `openssl_get_cipher_methods()` 选择的加密方法。该方法应使用下面的方式表示： 方法-密钥长度-加密模式 例如，如果你选择了一个 AES 加密方法、256 位的密钥长度和 GCM 加密模式，那么该参数就应该为 `aes-256-gcm`
密码	尽管将该参数称为密码，但实际上该参数是指密钥。应使用 `random_bytes()` 函数按照相关加密方法的密钥长度限制生成这个密钥
选项	在你熟悉 `openssl` 扩展前，我们建议你将该参数设置为默认值 `0`
IV	使用 `random_bytes()` 函数按照加密方法的规定生成指定长度的 IV

8. 如果你选择 AES 加密方法、256 位的密钥长度和 XTS 模式，就可以使用下面的

代码执行加密操作：

```
$plainText = 'Super Secret Credentials';
$key = random_bytes(16);
$method = 'aes-256-xts';
$cipherText = openssl_encrypt($plainText, $method, $key, 0, $iv);
```

9. 要解密密文可使用相同的密钥（`$key` 变量）和 IV（`$iv` 变量），调用 `openssl_decrypt()` 函数：

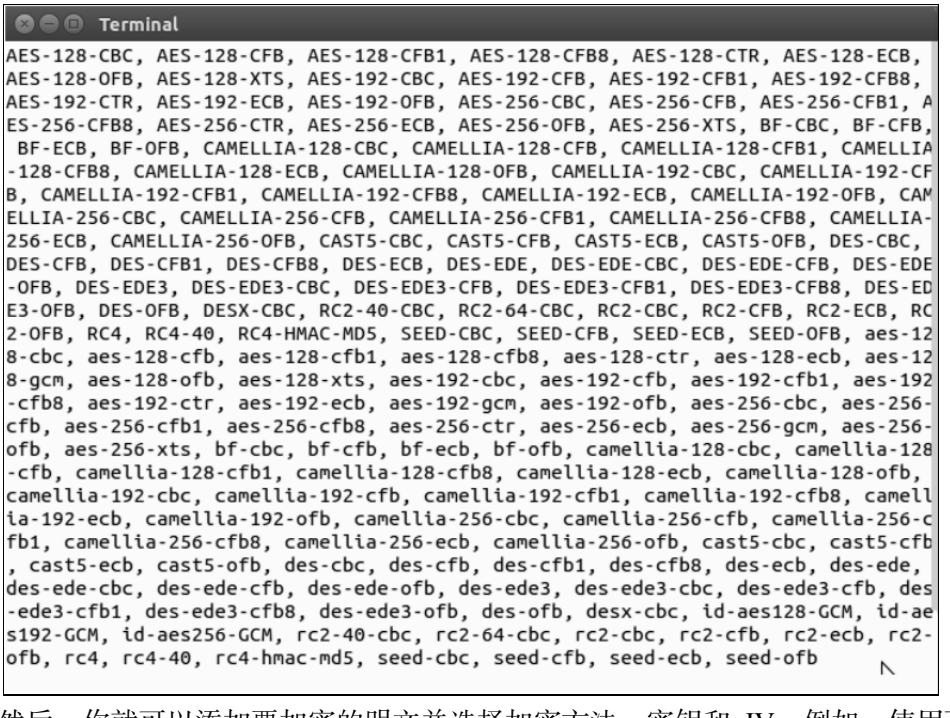
```
$plainText = openssl_decrypt($cipherText, $method, $key, 0, $iv);
```

具体运行情况

要了解 openssl 扩展提供了哪些加密方法，可创建 PHP 脚本 `chap_12_openssl_encryption.php` 并运行下面的命令：

```
<?php
echo implode(', ', openssl_get_cipher_methods());
```

下面是输出结果：



然后，你就可以添加要加密的明文并选择加密方法、密钥和 IV。例如，使用下面

的代码可选择 AES 加密方法、256 位长的密钥和 XTS 操作模式：

```
$plainText = 'Super Secret Credentials';
$method = 'aes-256-xts';
$key = random_bytes(16);
$iv = random_bytes(16);
```

要执行加密操作，可使用上述配置好的参数调用 `openssl_encrypt()` 函数：

```
$cipherText = openssl_encrypt($plainText, $method, $key, 0, $iv);
```

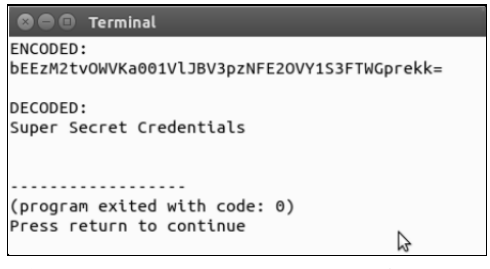
还可以使用 `base64_encode()` 函数对密文进行编码，使二进制数据可以通过非纯 8 位的传输层进行传输，从而使密文更易于使用：

```
$cipherText = base64_encode($cipherText);
```

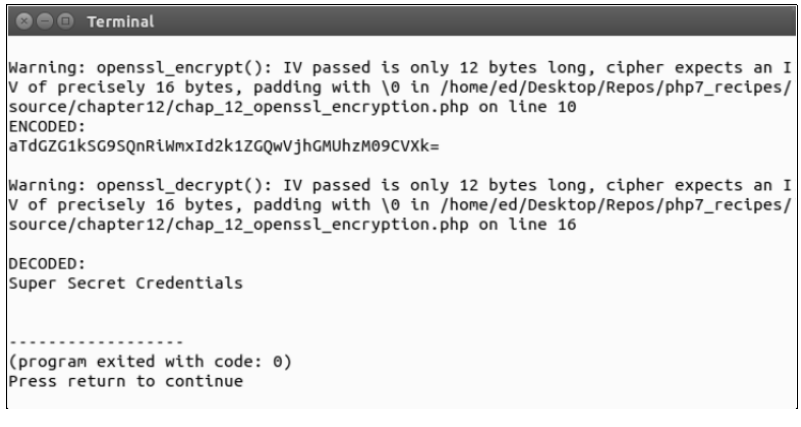
要执行解密操作，可使用相同的密钥（`$key` 变量）和 IV（`$iv` 变量）。不要忘记先解码使用 `base64_encode()` 函数获得 MIME BASE64 编码：

```
$plainText = openssl_decrypt(base64_decode($cipherText),
$method, $key, 0, $iv);
```

下面是密文的 MIME BASE64 编码和经过解密得到的明文：



如果你选择了具体的加密方法后设置错了 IV 的长度，PHP 引擎会显示警告消息：



补充说明

在 PHP 7 中，当使用 open_ssl_encrypt() 和 open_ssl_decrypt() **通过带关联数据的认证加密模式**（GCM 和 CCM）执行加密操作时会出问题。因此，PHP 7.1 为这两个函数增加了 3 个参数：

参数	注释
标签（$tag）	通过引用传输的认证标签；如果认证操作失败了，该变量的值会保持不变
$aad	额外的认证数据
标签长度（$tag_length）	在 GCM 模式中标签长度为 4 至 16 位；CCM 模式对标签长度没有限制；只能在 open_ssl_encrypt() 函数中使用

要详细了解这方面的内容，请浏览 https://wiki.php.net/rfc/openssl_aead。

扩展

要详细了解 PHP 7.1 弃用 mcrypt 扩展的原因，请浏览 https://wiki.php.net/rfc/mcrypt-viking-funeral。要详细了解分组密码（各种加密方法的基础），请浏览 https://en.wikipedia.org/wiki/Block_cipher。要详细了解加密算法 AES，请浏览 https://en.wikipedia.org/wiki/Advanced_Encryption_Standard。要详细了解加密操作模式，请浏览 https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation。

在使用某些较新的模式时，如被加密数据的长度小于分组长度，那么 openssl_decrypt() 函数不会返回值。向被加密数据中填充数据，使其达到分组长度的限定值，就可以解决这个问题。大多数模式都会在内部自行执行填充操作，因此这不算是问题。在使用较新的模式（即 xts）时会遇到这个问题。在将程序投入使用前，应先使用少于 8 个字符的字符串进行测试。

第 13 章　最佳编程习惯、测试和调试

本章包括以下要点：

- 使用特性和接口
- 通用异常处理程序
- 通用错误处理程序
- 编写简单测试
- 编写测试套件
- 生成模拟测试数据
- 使用 session_start 参数自定义会话

本章主要内容简介

本章介绍组合使用特性和接口的方式，还会介绍后备机制——当我们无法（或忘记）定义 try/catch 代码块时，可使用后备机制捕捉错误和异常。然后介绍单元测试，其中包括编写简单测试的方式，以及将这些简单测试组合成测试套件的方式。接下来会介绍用于生成任意数量通用测试数据的类。最后会讲述使用 PHP 7 的新增功能轻松管理会话的方式。

使用特性和接口

将接口用作创建一组类的定义规范（以确保实现特定方法）是公认的最佳编程习惯。特性和接口通常被组合使用，而且是实现代码的重要方面。如果你有一个经常使用的接口，而它定义了一个不会被修改的方法（如设置器和读取器），那么通常也需要定义含有实现代码的特性。

具体处理过程

1. 本例将使用第 4 章介绍过的 ConnectionAwareInterface 接口。该接口定义了用于设置$connection 属性的 setConnection()方法。Application\Generic 命名空间中含有两个类：CountryList 和 CustomerList，这两个类中含有因按照接口规定定义方法而产生的冗余代码。

2. 下面是未修改前的 CountryList 类：

```
class CountryList
{
  protected $connection;
  protected $key = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';

  public function setConnection(Connection $connection)
  {
    $this->connection = $connection;
  }
  public function list()
  {
    $list = [];
    $sql = sprintf('SELECT %s,%s FROM %s', $this->key,
                   $this->value, $this->table);
    $stmt = $this->connection->pdo->query($sql);
    while ($item = $stmt->fetch(PDO::FETCH_ASSOC)) {
      $list[$item[$this->key]] = $item[$this->value];
    }
    return $list;
  }

}
```

3. 现在让我们将 list()方法移动到 ListTrait 特性中：

```
trait ListTrait
{
  public function list()
  {
    $list = [];
```

```
        $sql = sprintf('SELECT %s,%s FROM %s',
                       $this->key, $this->value, $this->table);
        $stmt = $this->connection->pdo->query($sql);
        while ($item = $stmt->fetch(PDO::FETCH_ASSOC)) {
                $list[$item[$this->key]] = $item[$this->value];
        }
        return $list;
    }
}
```

4. 然后，通过 ListTrait 特性将这些代码插入到新建的 CountryListUsingTrait 类中：

```
class CountryListUsingTrait
{
  use ListTrait;
  protected $connection;
  protected $key = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';
  public function setConnection(Connection $connection)
  {
    $this->connection = $connection;
  }

}
```

5. 我们发现许多类都需要设置用于与数据库连接的实例，这也会用到特性。然而，这次我们要在 Application\Database 命名空间中创建新特性：

```
namespace Application\Database;
trait ConnectionTrait
{
  protected $connection;
  public function setConnection(Connection $connection)
  {
    $this->connection = $connection;
  }
}
```

6. 特性通常用于避免重复编写相同的代码。通常需要将使用了特性的类都标示出来，一种不错的实现方式是开发与特性对应的接口。本例定义 Application\Database\ConnectionAwareInterface 接口：

```
namespace Application\Database;
use Application\Database\Connection;
interface ConnectionAwareInterface
{
  public function setConnection(Connection $connection);
}
```

7. 然后定义 CountryListUsingTrait 类，该类应使用与 ConnectionAware Interface 接口对应的特性。注意，因为新建的特性会受到它在命名空间中所处位置的影响，所以需要在 CountryListUsingTrait 类的顶部添加 use 语句。我们实现 ConnectionAwareInterface 接口的目的是标示出需要使用特性定义方法的类。还应注意，我们利用了 PHP 7 新增的批量引用语法：

```
namespace Application\Generic;
use PDO;
use Application\Database\ {
Connection, ConnectionTrait, ConnectionAwareInterface
};
class CountryListUsingTrait implements ConnectionAwareInterface
{
  use ListTrait;
  use ConnectionTrait;

  protected $key = 'iso3';
  protected $value = 'name';
  protected $table = 'iso_country_codes';

}
```

具体运行情况

首先，确保第 4 章介绍的类已经创建，包括 Application\Generic\Country List 类和 Application\Generic\CustomerList 类。将这两个类分别存储到 Application\Generic 文件夹中的 CountryListUsingTrait.php 和 Customer ListUsingTrait.php 文件中。不要忘记更改这两个类的名称，使之与存储它们的文件同名！

按照步骤 3 的介绍，在 CountryListUsingTrait.php 和 CustomerList UsingTrait.php 中移除 list() 方法。在该方法原来的位置上添加 use ListTrait;

语句。将移除的代码添加到同一文件夹中的 ListTrait.php 文件中。

你会发现 CountryListUsingTrait 类和 CustomerListUsingTrait 类含有一部分相同的代码（即 setConnection() 方法），这就会用到另一个特性！

在 CountryListUsingTrait.php 和 CustomerListUsingTrait.php 中剪切 setConnection() 方法，然后将这些代码粘贴到 ConnectionTrait.php 文件中。因为该特性与 ConnectionAwareInterface 接口和 Connection 类有逻辑关系，所以应将 ConnectionTrait.php 文件放在 Application\Database 文件夹中，并为 ConnectionTrait 特性设置相应的命名空间。

最后，按照步骤 6 的介绍定义 Application\Database\ConnectionAwareInterface 接口。下面是完成了全部修改后的 Application\Generic\CustomerListUsingTrait 类：

```
<?php
namespace Application\Generic;
use PDO;
use Application\Database\Connection;
use Application\Database\ConnectionTrait;
use Application\Database\ConnectionAwareInterface;
class CustomerListUsingTrait implements ConnectionAwareInterface
{

  use ListTrait;
  use ConnectionTrait;

  protected $key    = 'id';
  protected $value  = 'name';
  protected $table  = 'customer';
}
```

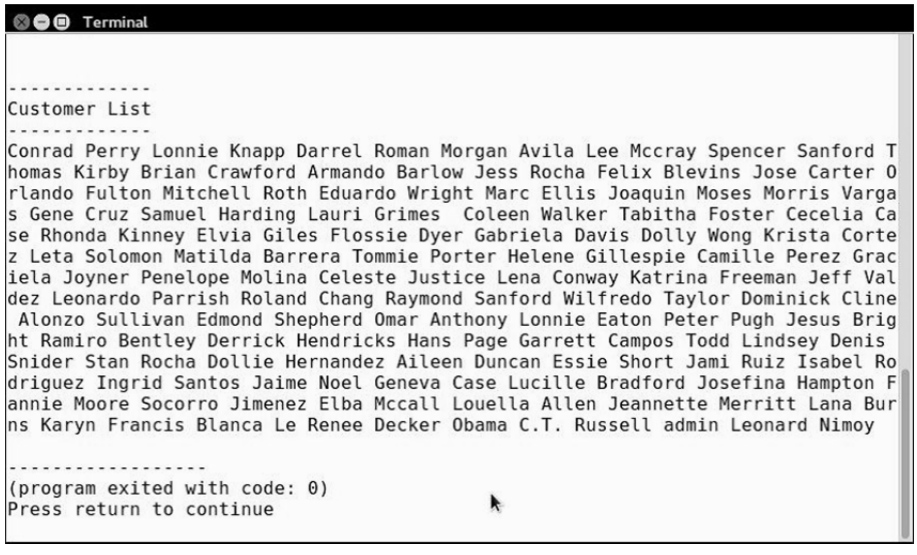
将第 4 章介绍的 chap_04_oop_simple_interfaces_example.php 文件的内容，复制到新建的 chap_13_trait_and_interface.php 文件中。将引用 CountryList 类的语句的，更改为引用 CountryListUsingTrait 类的。同理，将引用 CustomerList 类的语句更改为引用 CustomerListUsingTrait 类的。其他部分保持不变，下面是更改后的调用程序：

```
<?php
define('DB_CONFIG_FILE', '/../config/db.config.php');
```

```
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
$params = include __DIR__ . DB_CONFIG_FILE;
try {
     $list = Application\Generic\ListFactory::factory(
        new Application\Generic\CountryListUsingTrait(), $params);
     echo 'Country List' . PHP_EOL;
     foreach ($list->list() as $item) echo $item . ' ';
     $list = Application\Generic\ListFactory::factory(
       new Application\Generic\CustomerListUsingTrait(),
       $params);
     echo 'Customer List' . PHP_EOL;
     foreach ($list->list() as $item) echo $item . ' ';

} catch (Throwable $e) {
     echo $e->getMessage();
}
```

该程序的输出结果与第 4 章介绍的程序的输出结果完全相同。下图是输出结果中的国家列表部分：

下面是输出结果中的客户列表部分：



通用异常处理程序

当与 try/catch 代码块组合使用时，异常特别有用。然而，在某些情况中使用这种结构会使程序变得笨拙，无形中使代码变得难以理解。这种结构的另一个不足之处是许多类会以我们无法预料的方式抛出异常。在这种情况中，迫切需要后备的异常处理程序。

具体处理过程

1. 定义用于处理常见异常的 Application\Error\Handler 类：

```
namespace Application\Error;
class Handler
{
  // 在此处添加具体代码
}
```

2. 定义用于代表日志文件的属性。在没有专门为日志文件设置文件名的情况下，使用具体的年月日命名该文件。下面的构造器中，我们使用 set_exception_handler() 函数将 exceptionHandler() 方法作为后备异常处理程序（在 handler 类中）来分配：

```
protected $logFile;
public function __construct(
  $logFileDir = NULL, $logFile = NULL)
```

```
{
  $logFile = $logFile ?? date('Ymd') . '.log';
  $logFileDir = $logFileDir ?? __DIR__;
  $this->logFile = $logFileDir . '/' . $logFile;
  $this->logFile = str_replace('//', '/', $this-
    >logFile);
  set_exception_handler([$this,'exceptionHandler']);
}
```

3. 定义 exceptionHandler()方法，它会将 Exception 对象作为参数接收。将日期、时间、异常所属类的名称和异常消息记录到日志文件中：

```
public function exceptionHandler($ex)
{
  $message = sprintf('%19s : %20s : %s' . PHP_EOL,
    date('Y-m-d H:i:s'), get_class($ex), $ex->getMessage());
  file_put_contents($this->logFile, $message, FILE_APPEND);
}
```

4. 如果你专门在代码中添加了 try/catch 代码块，那么该代码块就会覆盖你自己编写的通用异常处理程序。如果你没有使用 try/catch 代码块且程序抛出了异常，那么你自己编写的通用异常处理程序就会发挥作用。

最佳编程习惯

在绝大多数情况中都应该使用 try/catch 代码块捕捉异常，以便程序能够继续运行。此处介绍的异常处理程序仅用于在无法使用 try/catch 代码块捕捉异常的情况中，这时使用异常处理程序能够使程序以优雅的方式结束运行。

具体运行情况

将前面介绍的代码添加到 Application\Error 文件夹中的 Handler.php 文件中。定义一个能够抛出异常的测试类。为了了解具体处理过程，可创建会抛出异常的 Application\Error\ThrowsException 类。例如，使用错误模式（PDO::ERRMODE_EXCEPTION）设置 PDO 实例，然后，创建一个肯定会出错的 SQL 语句：

```
namespace Application\Error;
use PDO;
```

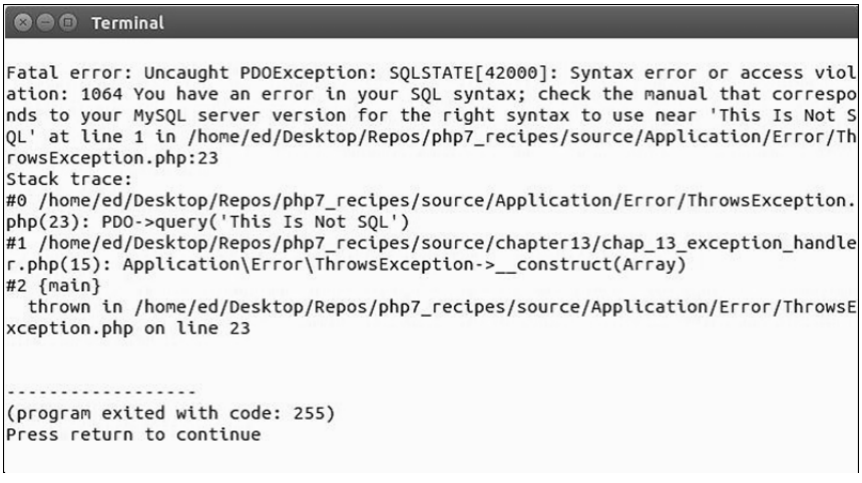
```
class ThrowsException
{
  protected $result;
  public function __construct(array $config)
  {
    $dsn = $config['driver'] . ':';
    unset($config['driver']);
    foreach ($config as $key => $value) {
      $dsn .= $key . '=' . $value . ';';
    }
    $pdo = new PDO(
      $dsn,
      $config['user'],
      $config['password'],
      [PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
      $stmt = $pdo->query('This Is Not SQL');
      while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
        $this->result[] = $row;
      }
  }
}
```

定义调用程序 chap_13_exception_handler.php，为其设置类自动加载功能，并引用合适的类：

```
<?php
define('DB_CONFIG_FILE', __DIR__ . '/../config/db.config.php');
$config = include DB_CONFIG_FILE;
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Error\ { Handler, ThrowsException };
```

此刻，如果你在创建 ThrowsException 实例时没有实现通用异常处理程序，那么程序就会将一个 Fatal Error（致命错误）作为没有被捕捉到的异常抛出：

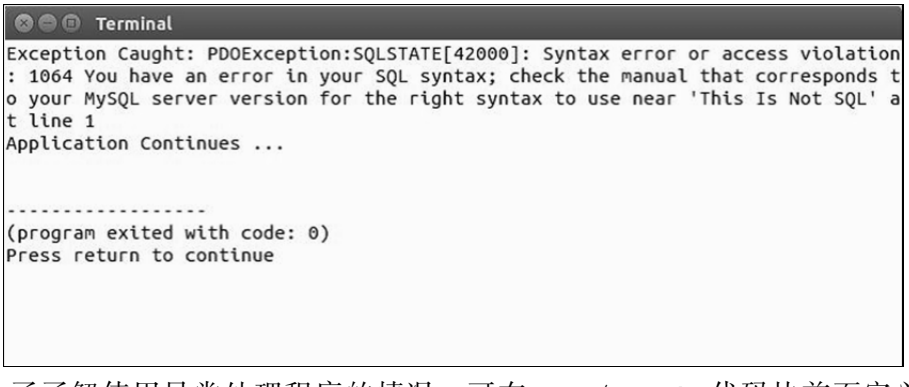
```
$throws1 = new ThrowsException($config);
```



如果你使用了 try/catch 代码块，该异常就会被捕捉，而且你的应用程序（在具有足够稳定性的前提下）也能够继续运行：

```
try {
    $throws1 = new ThrowsException($config);
} catch (Exception $e) {
    echo 'Exception Caught: ' . get_class($e) . ':' . $e->getMessage()
    . PHP_EOL;
}
echo 'Application Continues ...' . PHP_EOL;
```

你会得到下面的输出结果：

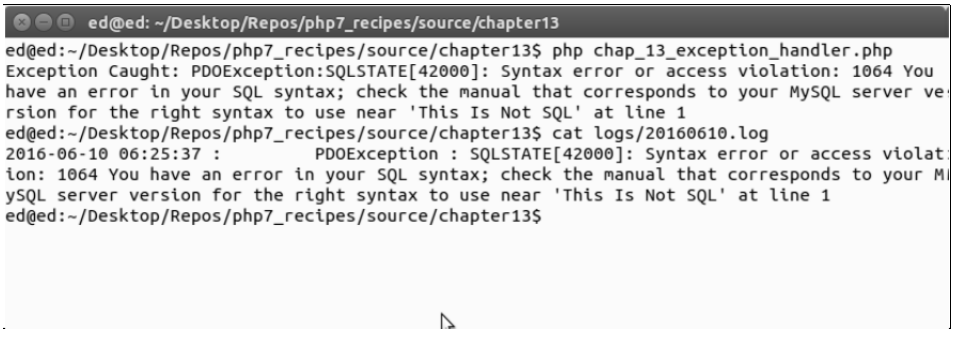


为了了解使用异常处理程序的情况，可在 try/catch 代码块前面定义一个 Handler 实例，并给该实例传输一个代表含有日志文件的目录的参数。在 try/catch 代码块后面创建一个 ThrowsException 实例。运行这个程序，你会发现第一个异常

被 try/catch 代码块捕捉了，而第二个异常被异常处理程序捕捉了。还应注意在异常处理程序被调用后，该程序结束运行了：

```
$handler = new Handler(__DIR__ . '/logs');
try {
    $throws1 = new ThrowsException($config);
} catch (Exception $e) {
    echo 'Exception Caught: ' . get_class($e) . ':'
      . $e->getMessage() . PHP_EOL;
}
$throws1 = new ThrowsException($config);
echo 'Application Continues ...' . PHP_EOL;
```

下面是完整示例程序的输出结果和日志文件的内容：



扩展

再次阅读 set_exception_handler() 的说明文档，这个函数会对异常处理工作有很大帮助。要详细了解该函数的运行方式，请浏览 http:// php.net/manual/en/function.set-exception-handler.php。

通用错误处理程序

编写通用错误处理程序的过程与编写通用异常处理程序的过程非常相似。当然，它们之间也存在差异。首先，在 PHP 7 中，某些错误是可以被抛出的而且是可以被捕捉的，而某些错误仅会使程序中断运行。简言之，某些错误可以像处理异常那样被处理，而另一些错误属于 PHP 7 中新增的 Error 类。幸运的是，在 PHP 7 中，Error 类（代表错

误)和 Exception 类(代表异常)都实现了新增的 Throwable 接口(代表可被抛出)。因此，如果你无法确定你的程序会抛出 Exception 对象还是 Error 对象，那么只需捕捉带 Throwable 接口的实例，就可以既捕捉异常也捕捉错误了。

具体处理过程

1. 修改上一节介绍的 Application\Error\Handler 类。在下面的构造器中定义新的 errorHandler() 方法，将该方法用作默认的错误处理程序：

```
public function __construct($logFileDir = NULL, $logFile = NULL)
{
  $logFile = $logFile ?? date('Ymd') . '.log';
  $logFileDir = $logFileDir ?? __DIR__;
  $this->logFile = $logFileDir . '/' . $logFile;
  $this->logFile = str_replace('//', '/', $this->logFile);
  set_exception_handler([$this,'exceptionHandler']);
  set_error_handler([$this, 'errorHandler']);
}
```

2. 编写这个新方法，使用具有自说明性质的名称为该方法定义参数。像异常处理程序一样，使该方法能够将错误信息写入日志文件：

```
public function errorHandler($errno, $errstr, $errfile, $errline)
{
  $message = sprintf('ERROR: %s : %d : %s : %s : %s' . PHP_EOL,
     date('Y-m-d H:i:s'), $errno, $errstr, $errfile, $errline);
  file_put_contents($this->logFile, $message, FILE_APPEND);
}
```

3. 为了将错误和异常区分开，可通过 exceptionHandler() 方法在写入日志文件的消息中添加 EXCEPTION 一词：

```
public function exceptionHandler($ex)
{
  $message = sprintf('EXCEPTION: %19s : %20s : %s' . PHP_EOL,
    date('Y-m-d H:i:s'), get_class($ex), $ex->getMessage());
  file_put_contents($this->logFile, $message, FILE_APPEND);
}
```

具体运行情况

按照前面介绍的步骤修改 Application\Error\Handler 类。创建用于抛出错

误的类，这里称之为 Application\Error\ThrowsError。例如，可使一个方法执行以 0 为除数的除法操作，使另一个方法尝试解析非 PHP 代码（如 JavaScript 的 eval() 函数）：

```
<?php
namespace Application\Error;
class ThrowsError
{
  const NOT_PARSE = 'this will not parse';
  public function divideByZero()
  {
    $this->zero = 1 / 0;
  }
  public function willNotParse()
  {
    eval(self::NOT_PARSE);
  }
}
```

定义调用程序 chap_13_error_throwable.php，为其设置类自动加载功能，引用合适的类，并创建一个 ThrowsError 实例：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Error\ { Handler, ThrowsError };
$error = new ThrowsError();
```

如果在没有使用 try/catch 代码块且没有定义通用错误处理程序的情况下调用下面两个方法，那么第一个方法会生成警告（Warning），而第二个方法会抛出 ParseError 错误：

```
$error->divideByZero();
$error->willNotParse();
echo 'Application continues ... ' . PHP_EOL;
```

因为这是一个错误，所以会使程序中断运行，你不会看到 Applicationcontinues：

如果你将调用这两个方法的语句封装在 try/catch 代码块中，并捕捉到了带 Throwable 接口的实例，那么程序就能够继续运行：

```
try {
    $error->divideByZero();
} catch (Throwable $e) {
    echo 'Error Caught: ' . get_class($e) . ':'
      . $e->getMessage() . PHP_EOL;
}
try {
    $error->willNotParse();
} catch (Throwable $e) {
    echo 'Error Caught: ' . get_class($e) . ':'
    . $e->getMessage() . PHP_EOL;
}
echo 'Application continues ... ' . PHP_EOL;
```

如下面的输出结果所示，程序停止运行的提示是 program exits with code:0，这代表程序是以正常方式结束运行的：

在 try/catch 代码块的后面，再次调用生成错误的方法，将 echo 语句移动到程

序末尾。如下面的输出结果所示，错误被捕捉到了。但是通过查看日志文件可以了解到，将 0 用作除数的错误（Division By Zero Error）是被异常处理程序捕捉到的，而语法解析错误（ParseError）是由错误处理程序捕捉到的：

```
$handler = new Handler(__DIR__ . '/logs');
$error->divideByZero();
$error->willNotParse();
echo 'Application continues ... ' . PHP_EOL;
```

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter13
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ php chap_13_error_throwable.php

Warning: Division by zero in /home/ed/Desktop/Repos/php7_recipes/source/Application/Error/T
hrowsError.php on line 11
Error Caught: ParseError:syntax error, unexpected 'will' (T_STRING)
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ cat logs/20160610.log
ERROR    : 2016-06-10 07:16:00 : 2 : Division by zero : /home/ed/Desktop/Repos/php7_recipes
/source/Application/Error/ThrowsError.php : 11
EXCEPTION: 2016-06-10 07:16:00 :          ParseError : syntax error, unexpected identifier
 (T_STRING)
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$
```

补充说明

PHP 7.1 允许在 catch() 子句中设置一个以上的类。因此，无须使用捕捉带 Throwable 接口的实例的语句，使用 catch (Exception | Error $e) {xxx }语句，就能够既捕捉代表异常的对象，也捕捉代表错误的对象。

编写简单测试

测试 PHP 代码的主要手段是使用 **PHPUnit** 测试框架，该框架以**单元测试**方法论为基础。单元测试的基础理论非常简单：将程序分解为最小的逻辑单元，然后单独测试每个单元，以确保每个单元都能够得到预期结果。这些预期结果被编写为一系列**断言**（assertion）。如果所有断言都返回 TRUE，那么该单元就通过了测试。

在过程式 PHP 程序中，一个单元就是一个函数。在 OOP 式 PHP 程序中，单元就是类中的方法。

具体处理过程

1. 需要做的第一件事情是安装 PHPUnit。可将 PHPUnit 直接安装在你的开发服务器上，也可以下载 PHPUnit 的源代码，这些源代码会存储在一个 phar 文件（PHP 归档文件）中。可在 PHPUnit 的官网（`https://phpunit.de/`），下载它。

2. 使用软件包管理器安装和维护 PHPUnit 是最佳编程习惯。因此，我们选择使用名为**Composer**的软件包管理程序。要安装 Composer 可浏览 `https://getcomposer.org/`，按照下载页面的说明操作即可。在笔者撰写本书时，使用的是下面的安装命令。注意，当你安装 Composer 时，应使用正确的散列值替换下面命令中的`<hash>`：

```
php -r "copy('https://getcomposer.org/installer',
  'composer-setup.php');"
php -r "if (hash_file('SHA384', 'composer-setup.php')
  === '<hash>') {
    echo 'Installer verified';
} else {
    echo 'Installer corrupt'; unlink('composer-setup.php');
} echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

最佳编程习惯

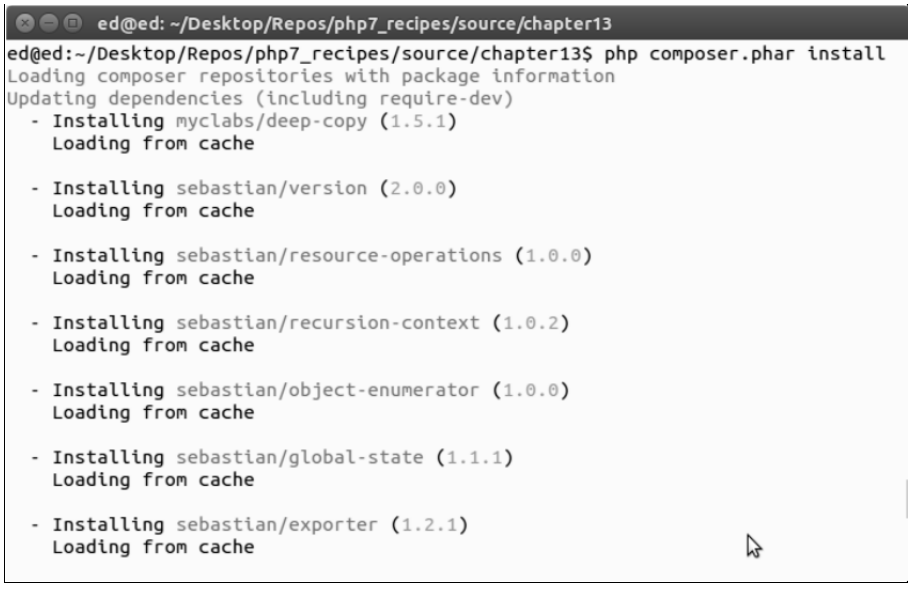
使用软件包管理程序（如 Composer）的好处是，软件包管理程序不仅可以安装程序，还可以更新你编写的 PHP 程序所使用的所有外部软件（如 PHPUnit）。

3. 让我们使用 Composer 安装 PHPUnit。通过创建 `composer.json` 文件并在其中添加一系列命令（用于设置项目参数和依赖关系）就可以做到这一点。对这些命令的详细介绍已经超出了本书涵盖的范围，但为了做本节的实验，我们还是会使用关键参数 `require` 创建一个较小的命令集。注意，该文件的格式是 JavaScript 对象表示法（JSON）：

```
{
  "require-dev": {
    "phpunit/phpunit": "*"
  }
}
```

4. 在命令行界面中执行下面的安装命令，下图是输出结果：

```
php composer.phar install
```



5. PHPUnit 和它的依赖关系都存储在 `vendor` 文件夹中，如果之前不存在该文件，Composer 会自动创建它。这样调用 PHPUnit 的主要命令就会象征性地指向 `vendor/bin` 文件夹。如果你使用的是类 UNIX 操作系统（如 Linux），那么只需将 vendor 文件夹放置在 `PATH` 目录中，就能够以全局方式使用 PHPUnit 的命令了。运行下面的命令可以查看 PHPUnit 的版本，并顺便确认它的安装过程已成功完成：

```
phpunit --version
```

运行简单的测试

1. 为了做这个实验，我们来创建 `chap_13_unit_test_simple.php` 文件并在其中添加 `add()` 函数：

```
<?php
function add($a = NULL, $b = NULL)
{
  return $a + $b;
}
```

2. 通过扩展 `PHPUnit\Framework\TestCase` 类，创建用于执行测试操作的类。

如果要测试一组函数，就应在执行测试操作的类的前面使用 require 语句，导入含有这些函数的文件。然后，就可以为测试类编写测试方法，测试方法的名称应以单词 test 开头，其后通常是被测试函数的名称，还可以带一些驼峰式单词（大写小写字母掺杂到一起的）来进一步描述该测试。为了做这个实验，我们定义了 SimpleTest 测试类：

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/chap_13_unit_test_simple.php';
class SimpleTest extends TestCase
{
  // 在此处添加 testXXX()方法
}
```

3. 断言是所有测试的核心。本节末尾的补充说明列出了详细介绍断言的参考资料。断言是一个 PHPUnit 方法，该方法会将一个已知值与被测试函数生成的值做比较。assertEquals()方法就是一个典型的例子，它用于检查函数的第一个参数是否与它的第二个参数相等。下面的示例测试了一个名为 add()的方法，并确认 add(1,1)的返回值为 **2**：

```
public function testAdd()
{
  $this->assertEquals(2, add(1,1));
}
```

4. 还可以用断言测试两个参数不相符的情况。例如，下面的示例断言 1 + 1 不等于 3：

```
$this->assertNotEquals(3, add(1,1));
```

5. 在测试字符串时，assertRegExp()断言极为有用。本例测试一个函数，该函数通过多维数组生成 HTML 表格：

```
function table(array $a)
{
  $table = '<table>';
  foreach ($a as $row) {
    $table .= '<tr><td>';
    $table .= implode('</td><td>', $row);
    $table .= '</td></tr>';
  }
  $table .= '</table>';
  return $table;
}
```

6. 我们可以编写一个简单的测试，确认 table()方法的输出结果中含有<table>

和</table>标签，以及这两个标签之间的一个或多个字符。此外，我们还需要确认该方法的输出结果中含有<td>B</td>页面元素。在编写该测试时，可创建一个由 3 个子数组构成的二维数组，这 3 个子数组分别含有字符 A ~ C、D ~ F 和 G ~ I。将这个二维数组设置为 table() 方法的参数，然后调用断言方法确认 table() 方法的输出结果：

```
public function testTable()
{
  $a = [range('A', 'C'),range('D', 'F'),range('G','I')];
  $table = table($a);
  $this->assertRegExp('!^<table>.+</table>$!', $table);
  $this->assertRegExp('!<td>B</td>!', $table);
}
```

7. 要测试一个类（而不是一组方法），只需导入定义该类的文件。为了了解具体处理步骤，可将前面介绍的一些函数添加到 Demo 类中：

```
<?php
class Demo
{
  public function add($a, $b)
  {
    return $a + $b;
  }

  public function sub($a, $b)
  {
    return $a - $b;
  }
  // 为节省篇幅此处没有列出所有函数
}
```

8. 在 SimpleClassTest 测试类中，无须导入存储函数的文件，只需包含（即使用 require 语句导入）存储 Demo 类的文件。要进行测试，需要先创建一个 Demo 实例。为了做到这一点，可使用专门编写的 setup() 方法，该方法会在所有测试操作执行前运行。还应注意 teardown() 方法，该方法紧跟在每次测试操作后运行：

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/Demo.php';
class SimpleClassTest extends TestCase
{
```

```
    protected $demo;
    public function setup()
    {
      $this->demo = new Demo();
    }
    public function teardown()
    {
      unset($this->demo);
    }
    public function testAdd()
    {
      $this->assertEquals(2, $this->demo->add(1,1));
    }
    public function testSub()
    {
      $this->assertEquals(0, $this->demo->sub(1,1));
    }
    // 依此类推
}
```

在每个测试操作之前和之后分别运行 setup()和 teardown()方法是为了使每次测试操作都能够有崭新的测试环境。这样测试结果就不会受到其他测试操作的影响。

测试数据库模型类

1. 在测试执行数据库访问操作的类（如模型类）时需要考虑更多问题。主要的一个问题是，需要使用测试数据库进行测试，而不是使用软件产品中真正的数据库进行测试。可以确定的一点是，通过使用测试数据库可以提前将合适的、可控制的数据添加到测试数据库中。setup()和 teardown()方法也可以被用来添加或移除测试数据。

2. 我们定义 VisitorOps 类，使用它访问数据库。这个新建的类中含有多个方法，使用这些方法可以添加、删除和查找浏览我们网站的注册用户。注意，我们还额外添加了一个方法，使用该方法能够获得最近被执行的 SQL 语句：

```
<?php
require __DIR__ . '/../Application/Database/Connection.php';
```

```
use Application\Database\Connection;
class VisitorOps
{

const TABLE_NAME = 'visitors';
protected $connection;
protected $sql;

public function __construct(array $config)
{
  $this->connection = new Connection($config);
}

public function getSql()
{
  return $this->sql;
}

public function findAll()
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  $stmt = $this->runSql($sql);
  while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {
    yield $row;
  }
}

public function findById($id)
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  $stmt = $this->runSql($sql, [$id]);
  return $stmt->fetch(PDO::FETCH_ASSOC);
}

public function removeById($id)
{
  $sql = 'DELETE FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  return $this->runSql($sql, [$id]);
}
```

```
public function addVisitor($data)
{
  $sql = 'INSERT INTO ' . self::TABLE_NAME;
  $sql .= ' (' . implode(',',array_keys($data)) . ') ';
  $sql .= ' VALUES ';
  $sql .= ' ( :' . implode(',:',array_keys($data)) . ') ';
  $this->runSql($sql, $data);
  return $this->connection->pdo->lastInsertId();
}

public function runSql($sql, $params = NULL)
{
  $this->sql = $sql;
  try {
    $stmt = $this->connection->pdo->prepare($sql);
    $result = $stmt->execute($params);
  } catch (Throwable $e) {
    error_log(__METHOD__ . ':' . $e->getMessage());
    return FALSE;
  }
  return $stmt;
}
}
```

3. 如果测试涉及数据库，那么推荐使用测试数据库，而不是软件产品中真正的数据库。相应地，你将需要使用另一套数据库连接参数，以便使用 `setup()` 方法连接测试数据库。

4. 应创建具有一致性的测试数据。然后使用 `setup()` 方法将这些数据插入测试数据库中。

5. 每次做完测试后应重置测试数据库，通过 `teardown()` 方法可以做到这一点。

使用模拟类

1. 在某些情况中，测试需要访问复杂的组件，而这些组件只能由外部资源提供——例如，一个提供服务的类需要访问数据库。因此，在测试套件中将数据库访问操作最小化就是一个最佳实践。另一个需要考虑的问题是，我们的目的不是测试数据库访问操作，我们只是想要测试一个特定类的某项功能。因此，有时需要定义**模拟**类，使用模拟类模拟被测试类的父类的操作，但限制模拟类访问外部资源。

最佳编程习惯

在进行测试时，应限制模型类（或其他具有相同作用的类）访问真正的数据库。否则，在测试这些类时会花费过多的时间。

2. 本例定义的提供服务的类是 VisitorService，该类会用到前面介绍过的 VisitorOps 类：

```
<?php
require_once __DIR__ . '/VisitorOps.php';
require_once __DIR__ . '/../Application/Database/Connection.php';
use Application\Database\Connection;
class VisitorService
{
  protected $visitorOps;
  public function __construct(array $config)
  {
    $this->visitorOps = new VisitorOps($config);
  }
  public function showAllVisitors()
  {
    $table = '<table>';
    foreach ($this->visitorOps->findAll() as $row) {
      $table .= '<tr><td>';
      $table .= implode('</td><td>', $row);
      $table .= '</td></tr>';
    }
    $table .= '</table>';
    return $table;
  }
```

3. 为了进行测试，应为$visitorOps 属性编写读取器和设置器，这样我们就能够在 VisitorOps 类的位置插入一个模拟类：

```
public function getVisitorOps()
{
  return $this->visitorOps;
}

public function setVisitorOps(VisitorOps $visitorOps)
{
```

```
    $this->visitorOps = $visitorOps;
  }
  } // 这是 VisitorService 类的结束花括号
```

4. 定义 VisitorOpsMock 类，使用该类模拟它的父类（VisitorOps）的功能。它的类常量和属性都是从VisitorOps类那里继承的。我们可以添加模拟的测试数据，以及一个设置器，以便将来能够继续添加测试数据：

```
<?php
require_once __DIR__ . '/VisitorOps.php';
class VisitorOpsMock extends VisitorOps
{
  protected $testData;
  public function __construct()
  {
    $data = array();
    for ($x = 1; $x <= 3; $x++) {
      $data[$x]['id'] = $x;
      $data[$x]['email'] = $x . 'test@unlikelysource.com';
      $data[$x]['visit_date'] =
         '2000-0' . $x . '-0' . $x . ' 00:00:00';
      $data[$x]['comments'] = 'TEST ' . $x;
      $data[$x]['name'] = 'TEST ' . $x;
    }
    $this->testData = $data;
  }
  public function getTestData()
  {
    return $this->testData;
  }
```

5. 重写 findAll()方法，以使用 yield 协程返回测试数据，就像在父类中那样，。注意，我们仍旧创建了 SQL 字符串，因为这就是父类执行的操作：

```
public function findAll()
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  foreach ($this->testData as $row) {
    yield $row;
  }
}
```

6. 要模拟 findById()方法，只需通过$this->testData 语句返回 testData 数组的键。要模拟 removeById()方法，可通过$this->testData 语句复位被用作参数的 testData 数组的键。

```
public function findById($id)
{
  $sql = 'SELECT * FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  return $this->testData[$id] ?? FALSE;
}
public function removeById($id)
{
  $sql = 'DELETE FROM ' . self::TABLE_NAME;
  $sql .= ' WHERE id = ?';
  if (empty($this->testData[$id])) {
      return 0;
  } else {
      unset($this->testData[$id]);
      return 1;
  }
}
```

7. 添加测试数据的操作更复杂一点，因为需要模拟 id 参数未被提供的情况（数据库通常会自动生成该参数）。要解决这个问题，可检查 id 参数。如果该参数没有被设置，就找到 testData 数组中最大的键并将它的值加 1：

```
  public function addVisitor($data)
  {
    $sql = 'INSERT INTO ' . self::TABLE_NAME;
    $sql .= ' (' . implode(',',array_keys($data)) . ') ';
    $sql .= ' VALUES ';
    $sql .= ' ( :' . implode(',:',array_keys($data)) . ') ';
    if (!empty($data['id'])) {
        $id = $data['id'];
    } else {
        $keys = array_keys($this->testData);
        sort($keys);
        $id = end($keys) + 1;
        $data['id'] = $id;
    }
    $this->testData[$id] = $data;
    return 1;
  }

} // 这是 VisitorOpsMock 类的结束花括号
```

将匿名类用作模拟对象

1. 除了使用正常的类定义模拟功能外，使用 PHP 7 中新增的匿名类来创建模拟对象也是不错的选择。使用匿名类的优点是可以扩展已经存在的类，从而使对象合乎规范。在仅需要重写一两个方法的情况中，这种方式尤为有用。

2. 本例将修改前面介绍过的 `VisitorServiceTest.php` 文件，将它重命名为 `VisitorServiceTestAnonClass.php`：

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorService.php';
require_once __DIR__ . '/VisitorOps.php';
class VisitorServiceTestAnonClass extends TestCase
{
  protected $visitorService;
  protected $dbConfig = [
    'driver'    => 'mysql',
    'host'      => 'localhost',
    'dbname'    => 'php7cookbook_test',
    'user'      => 'cook',
    'password'  => 'book',
    'errmode'   => PDO::ERRMODE_EXCEPTION,
  ];
    protected $testData;
```

3. 注意，我们在 `setup()` 方法中通过扩展 `VisitorOps` 方法定义了一个匿名类。此处仅需要重写 `findAll()` 方法：

```
public function setup()
{
  $data = array();
  for ($x = 1; $x <= 3; $x++) {
    $data[$x]['id'] = $x;
    $data[$x]['email'] = $x . 'test@unlikelysource.com';
    $data[$x]['visit_date'] =
       '2000-0' . $x . '-0' . $x . ' 00:00:00';
    $data[$x]['comments'] = 'TEST ' . $x;
    $data[$x]['name'] = 'TEST ' . $x;
  }
  $this->testData = $data;
```

```
    $this->visitorService =
      new VisitorService($this->dbConfig);
    $opsMock =
      new class ($this->testData) extends VisitorOps {
      protected $testData;
      public function __construct($testData)
      {
        $this->testData = $testData;
      }
      public function findAll()
      {
        return $this->testData;
      }
    };
    $this->visitorService->setVisitorOps($opsMock);
  }
```

4. 注意，当 testShowAllVisitors()方法中的$this->visitorService->showAllVisitors()语句被执行时，该匿名类就会被 visitorservice 对象调用，从而使已经被重写的 findAll()方法被调用：

```
  public function teardown()
  {
    unset($this->visitorService);
  }
  public function testShowAllVisitors()
  {
    $result = $this->visitorService->showAllVisitors();
    $this->assertRegExp('!^<table>.+</table>$!', $result);
    foreach ($this->testData as $key => $value) {
      $dataWeWant = '!<td>' . $key . '</td>!';
      $this->assertRegExp($dataWeWant, $result);
    }
  }
  }
```

使用模拟类生成器

1. 另一种测试技巧是使用 getMockBuilder()方法。尽管这种创建方式无法对所生成的模拟对象实现大量的细微控制，但在仅需要确定会返回指定类的对象，以及调用

了指定方法，而该方法会返回某些预期值的情况中，这种创建方式极为有用。

2. 在下面的示例中，我们复制了 VisitorServiceTestAnonClass 类，唯一的差别是在 setup() 方法中提供 VisitorOps 实例的方式，本例使用了 getMockBuilder() 方法。注意，尽管本例没有使用 with() 方法，但该方法可用于将受控的参数提供给模拟方法：

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorService.php';
require_once __DIR__ . '/VisitorOps.php';
class VisitorServiceTestAnonMockBuilder extends TestCase
{
  // 此处的代码与 VisitorServiceTestAnon 类中的代码相同
  public function setup()
  {
    $data = array();
    for ($x = 1; $x <= 3; $x++) {
      $data[$x]['id'] = $x;
      $data[$x]['email'] = $x . 'test@unlikelysource.com';
      $data[$x]['visit_date'] =
        '2000-0' . $x . '-0' . $x . ' 00:00:00';
      $data[$x]['comments'] = 'TEST ' . $x;
      $data[$x]['name'] = 'TEST ' . $x;
}
$this->testData = $data;
  $this->visitorService =
    new VisitorService($this->dbConfig);
  $opsMock = $this->getMockBuilder(VisitorOps::class)
               ->setMethods(['findAll'])
               ->disableOriginalConstructor()
               ->getMock();
               $opsMock->expects($this->once())
               ->method('findAll')
               ->with()
               ->will($this->returnValue($this->testData));
               $this->visitorService
               ->setVisitorOps($opsMock);
  }
  // 其余代码相同
}
```

前面介绍了创建简单的一次性测试的方式。然而在大多数情况中，需要测试的类有很多，而且最好一次将这些类都测试完。这就需要编写测试套件，下一节将详细介绍这方面的内容。

具体运行情况

首先，应按照前面步骤 1 至步骤 5 的介绍安装 PHPUnit。如果你使用的是类 UNIX 操作系统（如 Linux），应确保将 `vendor/bin` 文件夹放在 PATH 目录中，这样就可以通过命令行界面在任何目录下运行 PHPUnit 了。

运行简单测试

使用步骤 1 介绍的一组简单函数（如 `add()`、`sub()`等）创建 `chap_13_unit_test_simple.php` 文件。然后使用步骤 2 和步骤 3 介绍的简单测试类（SimpleTest）创建 `SimpleTest.php` 文件。

将 `phpunit` 的安装文件放在 `PATH` 目录中后，就可以通过终端窗口切换到存储本节介绍的代码的目录，并运行下面的命令：

```
phpunit SimpleTest SimpleTest.php
```

下面是输出结果：

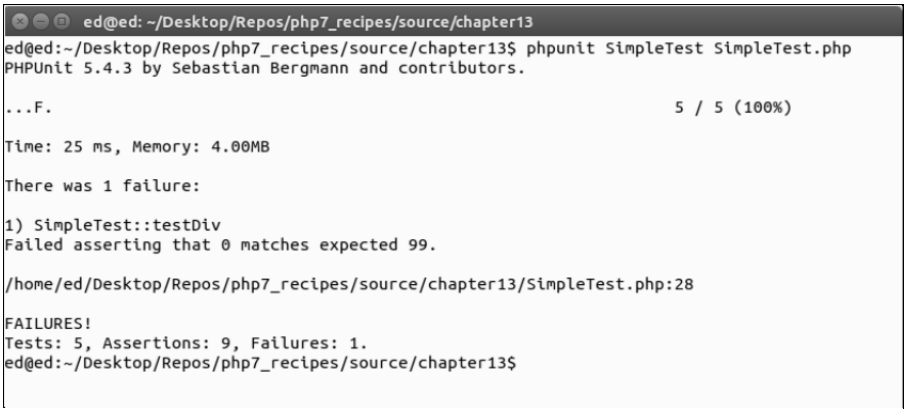
```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter13
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ phpunit SimpleTest SimpleTest.php
PHPUnit 5.4.3 by Sebastian Bergmann and contributors.

.....                                                               5 / 5 (100%)

Time: 33 ms, Memory: 4.00MB

OK (5 tests, 9 assertions)
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$
```

修改 `SimpleTest.php` 文件中的代码，使被测试方法无法通过测试（请参阅前面的步骤 4）：

```
public function testDiv()
{
  $this->assertEquals(2, div(4, 2));
```

```
  $this->assertEquals(99, div(4, 0));
}
```

下面是输出结果：



在 chap_13_unit_test_simple.php 文件中添加 table() 函数（请参阅前面的步骤 5），在 SimpleTest.php 文件中添加 testTable() 方法（请参阅前面的步骤 6）。重新运行这个单元测试并观察得到的结果。

要测试某个类，可将 chap_13_unit_test_simple.php 文件中的函数复制到 Demo 类中（请参阅前面的步骤 7）。按照前面步骤 8 的介绍修改 SimpleTest.php 文件，重新运行这个简单测试并观察得到的结果。

测试数据库模型类

按照前面步骤 2 的介绍创建将要被测试的 VisitorOps 类。定义 SimpleDatabaseTest 类，使用该类测试 VisitorOps 类。先使用 require_once 语句加载要被测试的类（下一节介绍相应的类自动加载功能），然后定义关键属性，其中包括用于测试数据库配置和测试数据的属性。可将 php7cookbook_test 用作测试数据库：

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorOps.php';
class SimpleDatabaseTest extends TestCase
{
  protected $visitorOps;
  protected $dbConfig = [
    'driver' => 'mysql',
```

```
    'host'      => 'localhost',
    'dbname'    => 'php7cookbook_test',
    'user'      => 'cook',
    'password'  => 'book',
    'errmode'   => PDO::ERRMODE_EXCEPTION,
  ];
  protected $testData = [
    'id' => 1,
    'email' => 'test@unlikelysource.com',
    'visit_date' => '2000-01-01 00:00:00',
    'comments' => 'TEST',
    'name' => 'TEST'
  ];
}
```

定义 setup()方法，用来插入测试数据，并确定最近被执行的一条 SQL 语句是 INSERT。还应检查该方法的返回值是否为 TRUE：

```
public function setup()
{
  $this->visitorOps = new VisitorOps($this->dbConfig);
  $this->visitorOps->addVisitor($this->testData);
  $this->assertRegExp('/INSERT/', $this->visitorOps->getSql());
}
```

定义 teardown()方法，用来移除测试数据，并确定查询 id=1 的记录的方法会返回 FALSE：

```
public function teardown()
{
  $result = $this->visitorOps->removeById(1);
  $result = $this->visitorOps->findById(1);
  $this->assertEquals(FALSE, $result);
  unset($this->visitorOps);
}
```

第一个测试用于检查 findAll()方法。首先，确定该方法返回结果的数据类型。应使用 current()方法获取第一个字段。我们预期 findAll()方法会使用 5 个字段执行查询操作，其中之一为 name 字段，其值与测试数据中 name 字段的值相同：

```
public function testFindAll()
{
  $result = $this->visitorOps->findAll();
  $this->assertInstanceOf(Generator::class, $result);
```

```
  $top = $result->current();
  $this->assertCount(5, $top);
  $this->assertArrayHasKey('name', $top);
  $this->assertEquals($this->testData['name'], $top['name']);
}
```

第二个测试用于检查 `findById()` 方法。这个测试方法几乎与 `testFindAll()` 方法相同：

```
public function testFindById()
{
  $result = $this->visitorOps->findById(1);
  $this->assertCount(5, $result);
  $this->assertArrayHasKey('name', $result);
  $this->assertEquals($this->testData['name'], $result['name']);
}
```

无须专门为 `removeById()` 方法编写测试方法，因为在检查 `teardown()` 方法时已经完成了对它的测试。同理，也无须测试 `runSql()` 方法，因为在测试其他方法的同时就会完成对该方法的检查。

使用模拟类

使用前面步骤 2 和步骤 3 介绍的代码定义代表服务的 `VisitorService` 类。然后，使用前面步骤 4 至步骤 7 介绍的代码定义模拟类 `VisitorOpsMock`。

为测试 `VisitorService` 类，我们编写测试类 `VisitorServiceTest`。注意，你需要提供自己的数据库配置，因为使用测试数据库（而不是在软件产品中使用的真正数据库）进行测试是最佳编程习惯：

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/VisitorService.php';
require_once __DIR__ . '/VisitorOpsMock.php';

class VisitorServiceTest extends TestCase
{
  protected $visitorService;
  protected $dbConfig = [
    'driver' => 'mysql',
    'host'       => 'localhost',
    'dbname'     => 'php7cookbook_test',
```

```
    'user'      => 'cook',
    'password'  => 'book',
    'errmode'   => PDO::ERRMODE_EXCEPTION,
  ];
}
```

在 setup()方法中创建一个代表服务的实例，并将 VisitorOpsMock 类插入其模拟原型类（VisitorService）的位置：

```
public function setup()
{
  $this->visitorService = new VisitorService($this->dbConfig);
  $this->visitorService->setVisitorOps(new VisitorOpsMock());
}
public function teardown()
{
  unset($this->visitorService);
}
```

下面的测试会通过网站浏览者列表生成一个 HTML 表。这样我们就可以在该表中查找特定的元素，提前了解调整测试数据的方向了：

```
public function testShowAllVisitors()
{
  $result = $this->visitorService->showAllVisitors();
  $this->assertRegExp('!^<table>.+</table>$!', $result);
  $testData = $this->visitorService->getVisitorOps()->getTestData();
  foreach ($testData as $key => $value) {
    $dataWeWant = '!<td>' . $key . '</td>!';
    $this->assertRegExp($dataWeWant, $result);
  }
}
}
```

你可以按照前面介绍的内容，自己做匿名类模拟对象和模拟生成器的测试实验。

补充说明

下面列出了一些方法，使用这些方法可以对数字、字符串、数组、对象、文件、JSON 数据和 XML 数据执行断言测试操作：

类别	断言方法
常规	assertEquals()、assertFalse()、assertEmpty()、assertNull()、assertSame()、assertThat()、assertTrue()
数字	assertGreaterThan()、assertGreaterThanOrEqual()、assertLessThan()、assertLessThanOrEqual()、assertNan()、assertInfinite()
字符串	assertStringEndsWith()、assertStringEqualsFile()、assertStringStartsWith()、assertRegExp()、assertStringMatchesFormat()、assertStringMatchesFormatFile()
数组/迭代器	assertArrayHasKey()、assertArraySubset()、assertContains()、assertContainsOnly()、assertContainsOnlyInstancesOf()、assertCount()
文件	assertFileEquals()、assertFileExists()
对象	assertClassHasAttribute()、assertClassHasStaticAttribute()、assertInstanceOf()、assertInternalType()、assertObjectHasAttribute()
JSON 数据	assertJsonFileEqualsJsonFile()、assertJsonStringEqualsJsonFile()、assertJsonStringEqualsJsonString()
XML 数据	assertEqualXMLStructure()、assertXmlFileEqualsXmlFile()、assertXmlStringEqualsXmlFile()、assertXmlStringEqualsXmlString()

扩展

- 要详细了解单元测试，请浏览 https://en.wikipedia.org/wiki/Unit_testing。
- 要详细了解 composer.json 文件指令，请浏览 https:// getcomposer.org/doc/04-schema.md。
- 要查看完整的断言方法列表，请参阅 PHPUnit 的说明文档 https://phpunit.de/manual/current/en/phpunit-book.html#appendixes.assertions。
- PHPUnit 的说明文档还详细介绍了 getMockBuilder() 方法，请参阅 https://phpunit.de/manual/current/en/phpunit-book.html#testdoubles.mock-objects。

编写测试套件

你可能已经注意到了，如果以手动方式运行 phpunit 并设置测试类和 PHP 文件的名称将会很麻烦，而且很快就会变得低效。在处理含有数十个乃至数百个类和文件的应用程序时，这个问题尤为突出。PHPUnit 测试框架拥有内置的批处理功能，使用该功能可以仅通过一条命令执行多个测试。这样的测试分组称为**测试套件**。

具体处理过程

1. 在最简单的情况中，你所需要做的仅是将所有测试文件移动到一个文件夹中：

```
mkdir tests
cp *Test.php tests
```

2. 如果外部文件改变了位置，应对导入这些文件的命令做相应的调整。本示例中的 SimpleTest 类已经在前面介绍了：

```
<?php
use PHPUnit\Framework\TestCase;
require_once __DIR__ . '/../chap_13_unit_test_simple.php';

class SimpleTest extends TestCase
{
  // 请参阅前面的内容
```

3. 这样只需将目录路径用作参数运行 phpunit 命令，PHPUnit 就会自动执行该文件夹中的所有测试。本例执行了 tests 子文件夹中的所有测试：

```
phpunit tests
```

4. 可使用--bootstrap 选项指定某文件夹中的内容先于其他测试执行。该选项常用于测试类自动加载功能：

```
phpunit --boostrap tests_with_autoload/bootstrap.php tests
```

5. 下面使用示例文件 bootstrap.php 实现类自动加载功能：

```
<?php
require __DIR__ . '/../../Application/Autoload/Loader.php';
Application\Autoload\Loader::init([__DIR__]);
```

6. 可使用 XML 配置文件定义一组或多组测试。下面是仅运行 Simple*测试文件的示例：

```
<phpunit>
  <testsuites>
    <testsuite name="simple">
      <file>SimpleTest.php</file>
      <file>SimpleDbTest.php</file>
      <file>SimpleClassTest.php</file>
    </testsuite>
  </testsuites>
</phpunit>
```

7. 下面的示例也以目录为单位，批量执行测试，同时也设置了先于其他测试执行的测试：

```
<phpunit bootstrap="bootstrap.php">
  <testsuites>
    <testsuite name="visitor">
      <directory>Simple</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

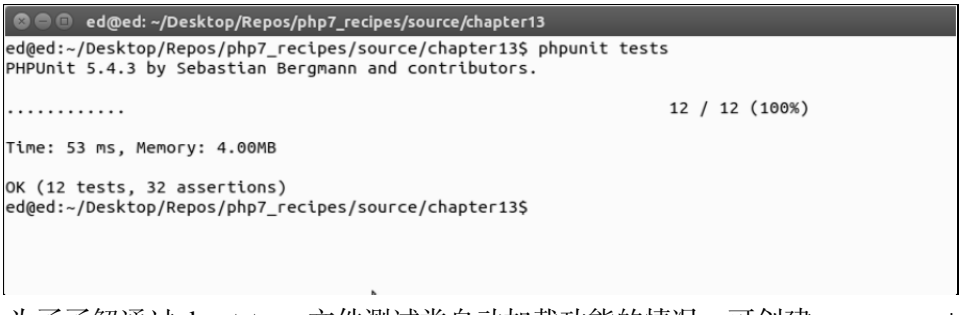
具体运行情况

确保“编写简单测试”一节中介绍的所有简单测试都定义好了，然后创建 `tests` 文件夹并将`*Test.php` 文件都移动或复制到该文件夹中。还需要调整 `require_once` 语句中的路径设置，参见步骤 2 中的介绍。

为了演示 PHPUnit 是怎样运行一个文件夹中的所有测试的，我们在存储本章介绍的所有测试文件的目录中运行下面的命令：

```
phpunit tests
```

你会看到下面的输出结果：



为了了解通过 bootstrap 文件测试类自动加载功能的情况，可创建 `tests_with_`

autoload 目录。使用步骤 5 介绍的代码在该文件夹中定义 bootstrap.php 文件。在 tests_with_autoload 目录中创建两个子目录：Demo 和 Simple。

将 Demo.php 文件（请参阅前面的内容）从存储本章源代码文件的目录复制到 tests_with_autoload/Demo 目录中。在<?php 开始标签后面添加下面的代码：

```
namespace Demo;
```

然后将 SimpleTest.php 文件复制到 tests_with_autoload/Simple 目录，并将其重命名为 ClassTest.php。按照下面的代码，修改该文件中的前几行代码：

```
<?php
namespace Simple;
use Demo\Demo;
use PHPUnit\Framework\TestCase;

class ClassTest extends TestCase
{
  protected $demo;
  public function setup()
  {
    $this->demo = new Demo();
  }
// 其余代码无须修改
```

创建 tests_with_autoload/phpunit.xml 文件，以便将所有测试整合到一起：

```
<phpunit bootstrap="bootstrap.php">
  <testsuites>
    <testsuite name="visitor">
      <directory>Simple</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

切换到存储本章源代码文件的目录。现在可以在使用 bootstrap 文件、类自动加载功能和命名空间的情况下，运行单元测试了：

```
phpunit -c tests_with_autoload/phpunit.xml
```

下面是输出结果：

```
ed@ed: ~/Desktop/Repos/php7_recipes/source/chapter13
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$ phpunit -c tests_with_autoload/phpunit.xml
PHPUnit 5.4.3 by Sebastian Bergmann and contributors.

....                                                                4 / 4 (100%)

Time: 33 ms, Memory: 4.00MB

OK (4 tests, 6 assertions)
ed@ed:~/Desktop/Repos/php7_recipes/source/chapter13$
```

扩展

要详细了解编写 PHPUnit 测试套件的方式，请浏览 `https://phpunit.de/manual/current/en/phpunitbook.html#organizing-tests.xml-configuration`。

生成模拟测试数据

部分测试和调试过程会涉及具有真实性的测试数据。在某些情况中（尤其是测试数据库访问操作和生成标准检查程序的情况中），需要使用大量的测试数据。获取大量测试数据的一条途径是从各个网站收集数据，然后按照现实情况将这些数据随机组合到一起，最后将这些数据存储到数据库中。

具体处理过程

1. 首先，确定需要使用哪种数据测试你的应用程序。需要考虑的另一个问题是，你的网站是面向国际用户还是仅面向某个国家的用户的？

2. 要生成具有一致性的模拟数据池，转换数据的格式极为重要。最佳选择是将数据存储到数据库表中，次之的选择是将数据存储到 CSV 文件中。

3. 可将转换数据的过程分为多个阶段。例如，先将从其他网站获取的国家编码和名称存储到文本文件中。

AF Afghanistan	AG Algeria	AJ Azerbaijan
AL Albania	AM Armenia	AN Andorra
AO Angola	AR Argentina	AS Australia
AT Ashmore & Cartier Islands	AU Austria	AV Anguilla
AX Akrotiri	AY Antarctica	BA Bahrain
BB Barbados	BC Botswana	BD Bermuda
BE Belgium	BF Bahamas, The	BG Bangladesh
BH Belize	BK Bosnia & Herzegovina	BL Bolivia
BM Burma	BN Benin	BO Belarus
BP Soloman Islands	BR Brazil	BS Bassas Da India
BT Bhutan	BU Bulgaria	BV Bouvet Island
BX Brunei	BY Burundi	CA Canada
CB Cambodia	CD Chad	CE Sri Lanka
CF Congo	CG Congo (Dem. Republic of The)-(Zaire)	CH China
CI Chile	CJ Cayman Islands	CK Cocos (Keeling) Islands
CM Cameroon	CN Comoros	CO Colombia
CR Coral Sea Islands	CS Costa Rica	CT Central African Republic
CU Cuba	CV Cape Verde	CW Cook Islands

4. 因为这个列表很短，所以可以轻松将它们复制粘贴到文本文件中。

5.在这些数据中搜索空格并使用\n 符号（代表换行）替换它们，就可以获得下面的列表：

```
1   AA Aruba
2   AC Antigua & Barbuda
3   AE United Arab Emirates
4   AF Afghanistan
5   AG Algeria
6   AJ Azerbaijan
7   AL Albania
8   AM Armenia
9   AN Andorra
10  AO Angola
11  AR Argentina
12  AS Australia
13  AT Ashmore & Cartier Islands
14  AU Austria
15  AV Anguilla
16  AX Akrotiri
17  AY Antarctica
```

6. 然后可以将这些数据导入电子表格中，进而将这些数据导出为 CSV 文件。使用电子表格将这些数据导入数据库中也很简单。例如，使用 phpMyAdmin 数据库管理工具就可以实现。

7. 为了了解具体处理步骤，我们将测试数据存储到 `prospects` 表中。下面是创建该表的 SQL 语句：

```
CREATE TABLE 'prospects' (
  'id' int(11) NOT NULL AUTO_INCREMENT,
  'first_name' varchar(128) NOT NULL,
  'last_name' varchar(128) NOT NULL,
  'address' varchar(256) DEFAULT NULL,
```

```
  'city' varchar(64) DEFAULT NULL,
  'state_province' varchar(32) DEFAULT NULL,
  'postal_code' char(16) NOT NULL,
  'phone' varchar(16) NOT NULL,
  'country' char(2) NOT NULL,
  'email' varchar(250) NOT NULL,
  'status' char(8) DEFAULT NULL,
  'budget' decimal(10,2) DEFAULT NULL,
  'last_updated' datetime DEFAULT NULL,
  PRIMARY KEY ('id'),
  UNIQUE KEY 'UNIQ_35730C06E7927C74' ('email')
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

8. 创建用于生成模拟测试数据的类。为该类创建生成上述字段数据（id 字段除外，该字段是由数据库自动生成的）的方法：

```
namespace Application\Test;

use PDO;
use Exception;
use DateTime;
use DateInterval;
use PDOException;
use SplFileObject;
use InvalidArgumentsException;
use Application\Database\Connection;

class FakeData
{
  // 在此处添加用于生成模拟测试数据的方法
}
```

9. 为该类定义常量和属性：

```
const MAX_LOOKUPS       = 10;
const SOURCE_FILE       = 'file';
const SOURCE_TABLE      = 'table';
const SOURCE_METHOD     = 'method';
const SOURCE_CALLBACK   = 'callback';
const FILE_TYPE_CSV     = 'csv';
const FILE_TYPE_TXT     = 'txt';
const ERROR_DB          = 'ERROR: unable to read source table';
const ERROR_FILE        = 'ERROR: file not found';
```

```
const ERROR_COUNT      = 'ERROR: unable to ascertain count or ID
                         column missing';
const ERROR_UPLOAD = 'ERROR: unable to upload file';
const ERROR_LOOKUP = 'ERROR: unable to find any IDs in the
                     source table';

protected $connection;
protected $mapping;
protected $files;
protected $tables;
```

10. 定义用于生成随机字符、街道名称和电子邮箱地址的属性。可以将这些数组视为积木，根据你的需要增加和减少它们。例如，可以为法国的浏览者提供巴黎的街道名称：

```
protected $alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
protected $street1 = ['Amber','Blue','Bright','Broad','Burning',
  'Cinder','Clear','Dewy','Dusty','Easy']; // 诸如此类的街道名称
protected $street2 = ['Anchor','Apple','Autumn','Barn','Beacon',
  'Bear','Berry','Blossom','Bluff','Cider','Cloud']; // 诸如此类的街道名称
protected $street3 = ['Acres','Arbor','Avenue','Bank','Bend',
  'Canyon','Circle','Street'];
protected $email1 = ['northern','southern','eastern','western',
  'fast','midland','central'];
protected $email2 = ['telecom','telco','net','connect'];
protected $email3 = ['com','net'];
```

11. 应使该类的构造器接收 Connection 对象（用于访问数据库），它是一个与模拟测试数据对应的数组：

```
public function __construct(Connection $conn, array $mapping)
{
  $this->connection = $conn;
  $this->mapping = $mapping;
}
```

12. 要生成街道名称（而不是创建数据库表），使用一组数组生成随机组合的字符串，可以获得更高的效率。请看下面的示例：

```
public function getAddress($entry)
{
  return random_int(1,999)
   . ' ' . $this->street1[array_rand($this->street1)]
   . ' ' . $this->street2[array_rand($this->street2)]
   . ' ' . $this->street3[array_rand($this->street3)];
}
```

13. 根据需要达到的模拟现实程度，还可以创建存储各城市邮政编码的数据库表。邮政编码也可以通过随机方式生成。下面的示例生成了英国的邮政编码：

```
public function getPostalCode($entry, $pattern = 1)
{
  return $this->alpha[random_int(0,25)]
    . $this->alpha[random_int(0,25)]
    . random_int(1, 99)
    . ' '
    . random_int(1, 9)
    . $this->alpha[random_int(0,25)]
    . $this->alpha[random_int(0,25)];
}
```

14. 同理，可使用可组合数组以随机方式生成模拟电子邮箱地址。可以使该方法接收已经定义的$entry数组和一些参数,并使用这些参数创建电子邮箱地址的名称部分：

```
public function getEmail($entry, $params = NULL)
{
  $first = $entry[$params[0]] ?? $this->alpha[random_int(0,25)];
  $last = $entry[$params[1]] ?? $this->alpha[random_int(0,25)];
  return $first[0] . '.' . $last
   . '@'
   . $this->email1[array_rand($this->email1)]
   . $this->email2[array_rand($this->email2)]
   . '.'
   . $this->email3[array_rand($this->email3)];
}
```

15. 生成日期数据的一个途径是接收已定义的$entry 数组和其他参数。这些参数中的第一个元素将存储了起始日期，第二元素将存储最大天数，以便与起始日期一起设置生成随机日期的范围。注意，我们会使用 DateTime::sub()语句减去随机天数。sub()方法需要接收 DateInterval 实例，该实例是由字符 P、一个随机天数和字符 D 生成的：

```
public function getDate($entry, $params)
{
  list($fromDate, $maxDays) = $params;
  $date = new DateTime($fromDate);
  $date->sub(new DateInterval('P' . random_int(0, $maxDays) . 'D'));
  return $date->format('Y-m-d H:i:s');
}
```

16. 如前所述，我们可以使用多种数据源创建模拟测试数据。在某些情况中，可使用可组合的数据创建模拟测试数据。在另一些情况中，需要将 CSV 文件用作数据源。下面是一个示例方法：

```
public function getEntryFromFile($name, $type)
{
  if (empty($this->files[$name])) {
      $this->pullFileData($name, $type);
  }
  return $this->files[$name][
  random_int(0, count($this->files[$name]))];
}
```

17. 注意，首先需要将文件中的数据添加到用于构成返回值的数组中。下面代码就是实现方法，如果没有找到指定的文件，该方法会抛出异常。可使用类常量 FILE_TYPE_TEXT 或 FILE_TYPE_CSV 标识文件的类型，并根据文件的类型选择使用 fgetcsv() 函数还是 fgets() 函数：

```
public function pullFileData($name, $type)
{
  if (!file_exists($name)) {
      throw new Exception(self::ERROR_FILE);
  }
  $fileObj = new SplFileObject($name, 'r');
  if ($type == self::FILE_TYPE_CSV) {
      while ($data = $fileObj->fgetcsv()) {
        $this->files[$name][] = trim($data);
      }
  } else {
      while ($data = $fileObj->fgets()) {
        $this->files[$name][] = trim($data);
      }
  }
```

18. 数据转换过程中最复杂的部分可能是从数据库表提取随机数据。应使提取数据的方法将数据库表的名称、构成主键的字段名、用于将数据库表中的字段与查询操作中的目标字段名对应起来的数组，都接收为参数：

```
public function getEntryFromTable($tableName, $idColumn, $mapping)
{
  $entry = array();
```

```
  try {
       if (empty($this->tables[$tableName])) {
         $sql = 'SELECT ' . $idColumn . ' FROM ' . $tableName
            . ' ORDER BY ' . $idColumn . ' ASC LIMIT 1';
         $stmt = $this->connection->pdo->query($sql);
         $this->tables[$tableName]['first'] =
           $stmt->fetchColumn();
         $sql = 'SELECT ' . $idColumn . ' FROM ' . $tableName
           . ' ORDER BY ' . $idColumn . ' DESC LIMIT 1';
         $stmt = $this->connection->pdo->query($sql);
         $this->tables[$tableName]['last'] =
           $stmt->fetchColumn();
}
```

19. 设置数据库准备语句并初始化一组起关键作用的变量：

```
$result = FALSE;
$count = self::MAX_LOOKUPS;
$sql = 'SELECT * FROM ' . $tableName
  . ' WHERE ' . $idColumn . ' = ?';
$stmt = $this->connection->pdo->prepare($sql);
```

20. 应将实际的查询语句放在 do…while 循环中，因为我们至少需要执行一次数据库查询操作才能获得结果。只有当没有获得我们想要的结果时，该循环才会继续执行。可在最小 ID 和最大 ID 的范围内生成随机 ID，然后将它用作查询操作的参数。注意，还应设置一个递减的计数器，以防止出现无限循环。无限循环可能会出现在 ID 为非连续数字的情况中，这时可能会生成不存在的 ID。如果执行循环的次数达到了最大次数，却仍然没有获得结果，那么就应使该方法抛出异常：

```
do {
  $id = random_int($this->tables[$tableName]['first'],
    $this->tables[$tableName]['last']);
  $stmt->execute([$id]);
  $result = $stmt->fetch(PDO::FETCH_ASSOC);
} while ($count-- && !$result);
  if (!$result) {
      error_log(__METHOD__ . ':' . self::ERROR_LOOKUP);
      throw new Exception(self::ERROR_LOOKUP);
  }
} catch (PDOException $e) {
    error_log(__METHOD__ . ':' . $e->getMessage());
    throw new Exception(self::ERROR_DB);
}
```

21. 使用实现对应关系的数组以及目的表中的预测键从源表检索值：

```
foreach ($mapping as $key => $value) {
  $entry[$value] = $result[$key] ?? NULL;
}
return $entry;
}
```

22. 这个类的核心是 getRandomEntry()方法，该方法能够生成一组模拟数据。循环遍历$mapping 数组中存储的记录，并检查各种参数：

```
public function getRandomEntry()
{
  $entry = array();
  foreach ($this->mapping as $key => $value) {
    if (isset($value['source'])) {
      switch ($value['source']) {
```

23. source 参数用于实现策略模式（Strategy Pattern）。应为 source 参数提供 4 种行为支持，并使用类常量定义它们。第一种行为是文件型数据源（由类常量 SOURCE_FILE 代表）行为。在本例中，我们可使用前面介绍过的 getEntryFromFile()方法：

```
case self::SOURCE_FILE :
    $entry[$key] = $this->getEntryFromFile(
    $value['name'], $value['type']);
  break;
```

24. 第二种行为是回调函数行为（由类常量 SOURCE_CALLBACK 代表），它会根据 $mapping 数组提供的回调函数返回一个值：

```
case self::SOURCE_CALLBACK :
    $entry[$key] = $value['name']();
  break;
```

25. 第三种行为是数据库表行为（由 SOURCE_TABLE 常量代表），它会将$mapping 数组中定义的数据库表用作数据源。注意，也可以使用前面介绍过的 getEntryFromTable()方法返回一组值，但要使用这些数据，还需使用 array_merge()函数将这些值合并到一起：

```
case self::SOURCE_TABLE :
    $result = $this->getEntryFromTable(
    $value['name'],$value['idCol'],$value['mapping']);
    $entry = array_merge($entry, $result);
  break;
```

26. 第四种行为是方法行为（由 SOURCE_METHOD 常量代表），它也是默认行为，该行为使用本类中已定义的方法获取源数据。应检查这些方法是否需要设置参数，如果需要设置参数，就在调用方法的语句中添加这些参数。使用{}花括号会影响插入的值。如果在 PHP 7 中使用$this->$value['name']()语句调用方法，由于抽象语法树（Abstract Syntax Tree，AST）的重写功能，该语句会被篡改为${$this->$value}['name']()，而这不是我们想要的结果：

```
        case self::SOURCE_METHOD :
        default :
          if (!empty($value['params'])) {
                $entry[$key] = $this->{$value['name']}(
                $entry, $value['params']);
           } else {
                $entry[$key] = $this->{$value['name']}($entry);
            }
          }
      }
    }
    return $entry;
  }
```

27. 定义一个方法，使该方法以循环方式调用 getRandomEntry()方法，以生成多条模拟测试数据。还应为该方法添加一个选项，使其能够将模拟测试数据插入目的表中。在启用了该选项的情况下，可设置用于执行插入操作的准备语句，并检查是否需要截断要插入的数据：

```
  public function generateData(
  $howMany, $destTableName = NULL, $truncateDestTable = FALSE)
  {
    try {
         if ($destTableName) {
            $sql = 'INSERT INTO ' . $destTableName
              . ' (' . implode(',', array_keys($this->mapping))
              . ') '. ' VALUES ' . ' (:'
              . implode(',:', array_keys($this->mapping)) . ')';
            $stmt = $this->connection->pdo->prepare($sql);
            if ($truncateDestTable) {
              $sql = 'DELETE FROM ' . $destTableName;
              $this->connection->pdo->query($sql);
```

```
            }
        }
    } catch (PDOException $e) {
        error_log(__METHOD__ . ':' . $e->getMessage());
        throw new Exception(self::ERROR_COUNT);
    }
```

28. 调用 `getRandomEntry()`方法，以便获得指定条数的源数据。如果需要执行向数据库插入数据的操作，就应在 `try/catch` 代码块中执行准备语句。在任何情况下都可以使用 `yield` 关键字将该方法变成数据生成器：

```
for ($x = 0; $x < $howMany; $x++) {
  $entry = $this->getRandomEntry();
  if ($insert) {
    try {
        $stmt->execute($entry);
    } catch (PDOException $e) {
        error_log(__METHOD__ . ':' . $e->getMessage());
        throw new Exception(self::ERROR_DB);
    }
  }
  yield $entry;
}
}
```

最佳编程习惯

如果需要转换的数据量非常大，最好使用 yield 协程处理数据，以节省数组占用的内存空间。

具体运行情况

在进行数据转换前，应确保已经准备好用于进行转换的源数据。本例将 `prospects` 数据库表用作目的表，步骤 7 介绍了该表的 SQL 数据库定义。

处理用户姓名的源数据时，可将姓氏和名字存储在文本文件中。本例会使用 `data/files` 目录中的 `first_names.txt` 和 `surnames.txt` 文件。在处理城市、州或省、邮政编码和国家源数据时，最好从 `http://www.geonames.org/`之类的网站下载数据，并上传一个 `world_city_data` 表格。处理其余字段（如住址、电子邮

箱、状态）的源数据时，可使用 FakeData 类中的方法或编写回调函数获取这些数据。

使用步骤 8 至步骤 28 介绍的代码定义 Application\Test\FakeData 类。创建调用程序 chap_13_fake_data.php，为其设置类自动加载功能，并引用适当的类。还应定义一些常量，使用这些常量匹配数据库配置文件和存储用户姓名的文件的路径：

```
<?php
define('DB_CONFIG_FILE', __DIR__ . '/../config/db.config.php');
define('FIRST_NAME_FILE', __DIR__ . '/../data/files/first_names.txt');
define('LAST_NAME_FILE', __DIR__ . '/../data/files/surnames.txt');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Test\FakeData;
use Application\Database\Connection;
```

定义代表目的表和源表对应关系的数组，将目的表（prospects）的字段名称设置为该数组的键，然后为 source、name 等必要参数定义子键。在$mapping 数组中，键的名称与源表中字段的名称相同，代表 first_name（名字）和 last_name（姓氏）字段的数组元素会将文件用作数据源，代表 name 字段（姓名）的数组元素会将存储用户名字的文件用作数据源，而代表 type 字段（类型）的数组元素会将文本文件用作数据源：

```
$mapping = [
  'first_name'  => ['source' => FakeData::SOURCE_FILE,
  'name'        => FIRST_NAME_FILE,
  'type'        => FakeData::FILE_TYPE_TXT],
  'last_name'   => ['source' => FakeData::SOURCE_FILE,
  'name'        => LAST_NAME_FILE,
  'type'        => FakeData::FILE_TYPE_TXT],
```

代表 address（地址）、email（电子邮箱）和 last_updated（最近一次更新的时间）字段的数组元素都使用内置方法获取源数据，其中后两个数组元素还需要传输参数：

```
'address'      => ['source' => FakeData::SOURCE_METHOD,
'name'         => 'getAddress'],
'email'        => ['source' => FakeData::SOURCE_METHOD,
'name'         => 'getEmail',
'params'       => ['first_name','last_name']],
'last_updated' => ['source' => FakeData::SOURCE_METHOD,
'name'         => 'getDate',
'params'       => [date('Y-m-d'), 365*5]]
```

代表 phone（电话）、status（状态）和 budget（预算）字段的数组元素可以通过调用回调函数获取模拟测试数据：

```
'phone'     => ['source' => FakeData::SOURCE_CALLBACK,
'name'      => function () {
               return sprintf('%3d-%3d-%4d', random_int(101,999),
               random_int(101,999), random_int(0,9999)); }],
'status'    => ['source' => FakeData::SOURCE_CALLBACK,
'name'      => function () { $status = ['BEG','INT','ADV'];
               return $status[rand(0,2)]; }],
'budget'    => ['source' => FakeData::SOURCE_CALLBACK,
                'name' => function() { return random_int(0, 99999)
                + (random_int(0, 99) * .01); }]
```

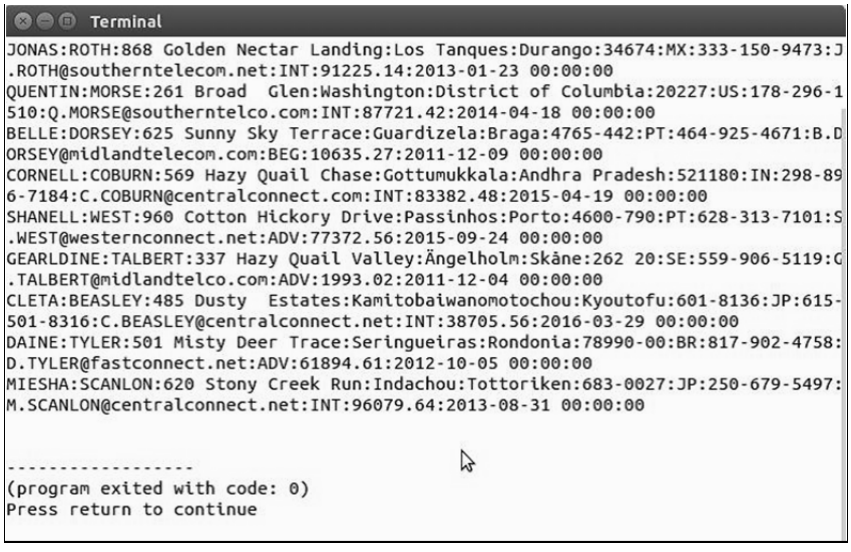
代表 city（城市）字段的数组元素通过查找表获取源数据，该查找表也为 mapping 子数组中代表各个字段的子数组元素提供数据。你可以先不为这些数组元素赋值，但应注意，还需要为源表设置代表主键的字段：

```
'city' => ['source' => FakeData::SOURCE_TABLE,
'name' => 'world_city_data',
'idCol' => 'id',
'mapping' => [
'city' => 'city',
'state_province' => 'state_province',
'postal_code_prefix' => 'postal_code',
'iso2' => 'country']
],
  'state_province'=> [],
  'postal_code' => [],
  'country' => [],
];
```

定义目的表（由 Connection 实例代表），并创建 FakeData 实例。使用一个 foreach()循环就足以显示指定数量的记录了：

```
$destTableName = 'prospects';
$conn = new Connection(include DB_CONFIG_FILE);
$fake = new FakeData($conn, $mapping);
foreach ($fake->generateData(10) as $row) {
  echo implode(':', $row) . PHP_EOL;
}
```

下面的输出结果显示了 10 条记录的情况：



补充说明

下列网站中含有各种各样的列表，可以使用这些列表中的数据生成模拟测试数据：

数据类型	URL	注释
姓名	http://nameberry.com/	
	http://www.babynamewizard.com/international-names-listspopular-names-from-around-theworld	
未经润饰的名字	http://deron.meranda.us/data/census-dist-female-first.txt	美国女性的名字
	http://deron.meranda.us/data/census-dist-male-first.txt	美国女性的名字
	http://www.avss.ucsb.edu/NameFema.HTM	美国女性的名字
	http://www.avss.ucsb.edu/namemal.htm	美国女性的名字
姓氏	http://names.mongabay.com/data/1000.html	美国人口普查统计出的姓氏
	http://surname.sofeminine.co.uk/w/surnames/most-commonsurnames-in-great-britain.html	英国姓氏
	https://gist.github.com/subodhghulaxe/8148971	使用 PHP 数组存储的美国姓氏列表

续表

数据类型	URL	注释
姓氏	http://www.dutchgenealogy.nl/tng/surnames-all.php	荷兰姓氏
	http://www.worldvitalrecords.com/browsesurnames.aspx?l=A	各国姓氏；只需更改该网址中的最后一（几）个字母，就可以获得以这一（几）个字母开头的姓氏
城市	http://www.travelgis.com/default.asp?framesrc=/cities/	世界各地的城市
	https://www.maxmind.com/en/freeworld-cities-database	
	https://github.com/David-Haim/CountriesToCitiesJSON	
	http://www.fallingrain.com/world/index.html	
邮政编码	https://boutell.com/zipcodes/	仅限于美国；数据包含城市、邮政编码、经度和纬度
	http://www.geonames.org/export/	各国的邮政编码；数据包含城市、邮政编码等非常多的内容，可免费下载

使用 session_start 参数自定义会话

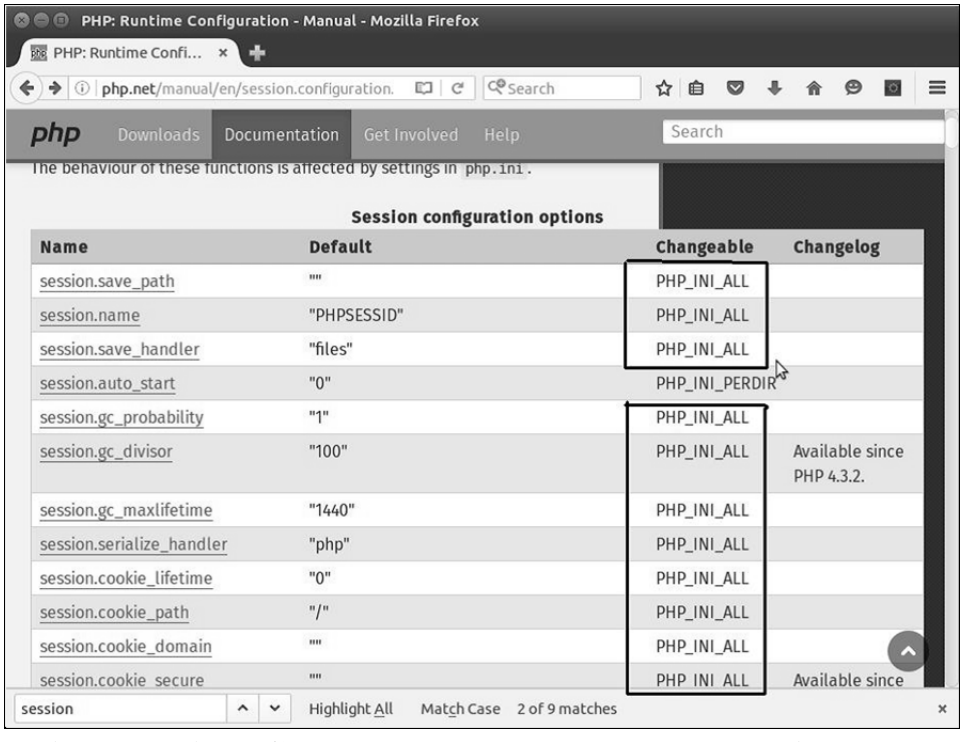
在 PHP 7 出现前，为了以安全的方式管理会话而重写 php.ini 设置时，必须使用一系列 ini_set() 命令。这种设置方式特别麻烦，你需要了解哪些设置是可用的，而且难以为其他应用程序重用相同的设置。然而，自从 PHP 7 出现后，就可以在 session_start() 命令中设置一组参数了，因此我们可以使用该命令迅速完成会话的设置。

具体处理过程

1. 首先，编写 Application\Security\SessOptions 类，它将会存储会话参数，并实现启动会话的功能。还应定义类常量，以处理出现非法会话设置的情况：

```
namespace Application\Security;
use ReflectionClass;
use InvalidArgumentsException;
class SessOptions
{
  const ERROR_PARAMS = 'ERROR: invalid session options';
```

2. 扫描 php.ini 文件中的会话设置（要详细了解会话设置方面的内容，请浏览 http://php.net/manual/en/session.configuration.php）。我们要找的是 Changeable 字段的值为 PHP_INI_ALL 的指令。这类指令可以在程序运行时重写，因此可以被用作 session_start() 函数的参数：



3. 定义下列类常量，使 SessOptions 类在开发过程中拥有更高的通用性。大多数正规的代码编辑器都能够扫描这个类，并显示常量列表，这简化了管理会话设置的工作。但请注意，为节省篇幅此处没有列出全部设置：

```
const SESS_OP_NAME = 'name';
const SESS_OP_LAZY_WRITE = 'lazy_write'; // PHP 7.0.0 以上的版本
  // 可以使用该设置
const SESS_OP_SAVE_PATH = 'save_path';
const SESS_OP_SAVE_HANDLER = 'save_handler';
// 为节省篇幅，此处没有列出所有设置
```

4. 定义这个类的构造器，应使该构造器将一组 php.ini 会话设置接收为参数。可使用 ReflectionClass 函数获取这个类的常量，然后使用循环处理$options 参数

（代表会话设置），以便确认这些设置是可以使用的。还应注意 array_flip()函数的用法，该函数会将数组中的键和值对调位置，这样就会将 SessOptions 类中常量的值设置为$allowed 数组（代表可用的设置）中的键，而将 SessOptions 类中常量的名称设置为$allowed 数组中的值：

```
protected $options;
protected $allowed;
public function __construct(array $options)
{
  $reflect = new ReflectionClass(get_class($this));
  $this->allowed = $reflect->getConstants();
  $this->allowed = array_flip($this->allowed);
  unset($this->allowed[self::ERROR_PARAMS]);
  foreach ($options as $key => $value) {
    if(!isset($this->allowed[$key])) {
      error_log(__METHOD__ . ':' . self::ERROR_PARAMS);
      throw new InvalidArgumentsException(
      self::ERROR_PARAMS);
    }
  }
  $this->options = $options;
}
```

5. 为 SessOptions 类编写两个方法，一个方法用于访问允许使用的参数，另一个方法用于启动会话：

```
public function getAllowed()
{
  return $this->allowed;
}

public function start()
{
  session_start($this->options);
}
```

具体运行情况

将前面介绍的代码添加到 Application\Security 目录中的 SessOptions.php 文件中。定义调用程序 chap_13_session_options.php，以便测试我们刚刚

编写的类，为该调用程序设置类自动加载功能，并引用 SessOptions 类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\Security\SessOptions;
```

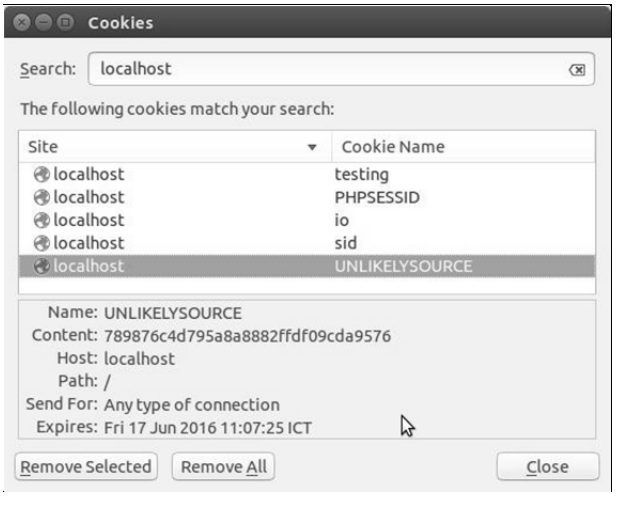
定义一个数组，将 SessOptions 类中的常量设置为该数组的键，将用于管理会话的参数设置为该数组的值。注意，下面的示例将会话信息存储在 session 子目录中，你需要创建该文件夹：

```
$options = [
  SessOptions::SESS_OP_USE_ONLY_COOKIES => 1,
  SessOptions::SESS_OP_COOKIE_LIFETIME => 300,
  SessOptions::SESS_OP_COOKIE_HTTPONLY => 1,
  SessOptions::SESS_OP_NAME => 'UNLIKELYSOURCE',
  SessOptions::SESS_OP_SAVE_PATH => __DIR__ . '/session'
];
```

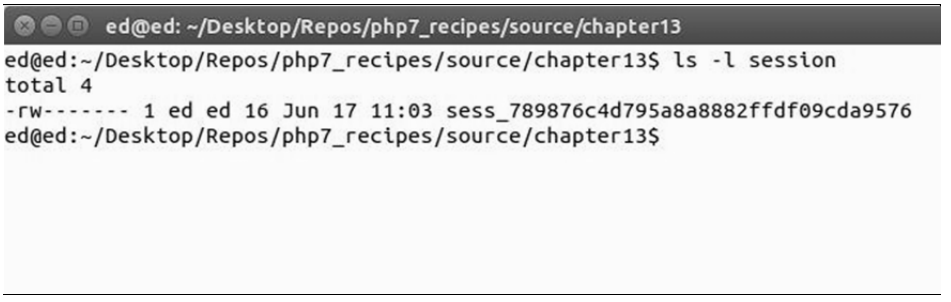
创建 SessOptions 实例并运行 start() 函数，以启动会话。可以使用 phpinfo() 函数查看一些会话信息：

```
$sessOpt = new SessOptions($options);
$sessOpt->start();
$_SESSION['test'] = 'TEST';
phpinfo(INFO_VARIABLES);
```

如果你使用浏览器中的开发者工具查看了 cookie 中的信息，就会发现 cookie 的名称（name）被设置为了 UNLIKELYSOURCE，且该 cookie 的有效期为 5 分钟：



如果你查看了存储会话文件的目录，就会发现该会话的信息已经被存储在这里了：



扩展

要详细了解 php.ini 配置文件中的会话设置，请浏览 http://php.net/manual/en/session.configuration.php。

附录　定义 PSR–7 类

下面是本附录包括的要点：

- 实现 PSR-7 值对象类
- 开发 PSR-7 请求类
- 定义 PSR-7 回应类

本附录主要内容简介

PHP 编程规范 7（PSR-7 推荐编程标准）定义了许多接口，但没有提供这些接口的实现代码。因此，为了创建自定义的中间件，我们需要编写具体的实现代码。

实现 PSR-7 值对象类

为了使用 PSR-7 请求和回应，首先需要定义一系列值对象。这些值对象类代表基于网页的行为（如 URI、上传文件，以及流式请求和回应的主体）中的逻辑对象。

准备工作

通过 `Composer` 软件包可以获得 PSR-7 接口的源代码。使用 `Composer` 依赖关系管理工具管理外部软件（如 PSR-7 接口）是公认的最佳编程习惯。

具体处理过程

1. 首先，在 `https://github.com/php-fig/http-message` 上获取最新版本的 PSR-7 接口定义，也可以获得 PSR-7 接口的源代码。在撰写本书时，可获得下列定义：

接口	所属扩展	注释	规定必须实现的方法
MessageInterface		定义用于处理 HTTP 消息的通用方法	处理报头、消息主体（即内容）和协议的方法
RequestInterface	MessageInterface	代表由客户端生成的请求的类	处理 URI、HTTP 和请求目标的方法
ServerRequestInterface	RequestInterface	代表服务器从客户端接收到的请求	处理服务器和查询操作参数、cookie、上传的文件和解析内容的方法
ResponseInterface	MessageInterface	代表从服务器发送到客户端的回应	处理 HTTP 状态码和出错原因的方法
StreamInterface		代表数据流	实现流式操作的方法，这些操作包括查找、通知、读取和写入等
UriInterface		代表 URI	处理协议（即 HTTP 和 HTTPS）、主机、端口、用户名、登录密码（FTP 服务器中的）、查询操作的参数、路径以及范围的方法
UploadedFileInterface		用于处理上传的文件	处理文件大小、媒体类型、移动文件操作和文件名的方法

2. 令人遗憾的是，要使用 PSR-7 规范，我们就必须自己动手编写实现这些接口的具体代码。幸运的是，介绍这些接口的材料非常多，通过这些说明材料能够详细了解这些接口。下面先创建一个含有多个常量的类：

使我们利用了在 PHP 7 中引入的新功能，使用该功能可以将常量定义为数组。

```
namespace Application\MiddleWare;
class Constants
{
  const HEADER_HOST = 'Host'; // 主机的报头
  const HEADER_CONTENT_TYPE = 'Content-Type';
  const HEADER_CONTENT_LENGTH = 'Content-Length';

  const METHOD_GET      = 'get';
  const METHOD_POST     = 'post';
  const METHOD_PUT      = 'put';
  const METHOD_DELETE   = 'delete';
```

```
    const HTTP_METHODS = ['get','put','post','delete'];

    const STANDARD_PORTS  = [
      'ftp' => 21, 'ssh'  => 22, 'http' => 80, 'https' => 443
    ];

    const CONTENT_TYPE_FORM_ENCODED  =
      'application/x-www-form-urlencoded';
    const CONTENT_TYPE_MULTI_FORM    = 'multipart/form-data';
    const CONTENT_TYPE_JSON          = 'application/json';
    const CONTENT_TYPE_HAL_JSON      = 'application/hal+json';

    const DEFAULT_STATUS_CODE    = 200;
    const DEFAULT_BODY_STREAM    = 'php://input';
    const DEFAULT_REQUEST_TARGET = '/';

    const MODE_READ = 'r';
    const MODE_WRITE = 'w';

    // 注意：为节省篇幅，此处没有列出所有用于处理错误的常量
    const ERROR_BAD = 'ERROR: ';
    const ERROR_UNKNOWN = 'ERROR: unknown';

    // 注意：此处没有列出全部状态码！
    const STATUS_CODES = [
      200 => 'OK',
      301 => 'Moved Permanently',
      302 => 'Found',
      401 => 'Unauthorized',
      404 => 'Not Found',
      405 => 'Method Not Allowed',
      418 => 'I_m A Teapot',
      500 => 'Internal Server Error',
    ];
  }
```

要查看全部HTTP状态码，请浏览:https://tools.ietf.org/html/rfc7231#section-6.1。

3. 定义代表由其他 PSR-7 类使用的值对象的类。让我们先处理代表 URI 的类，使该类的构造器将 URI 字符串作为参数接收，然后使用 `parse_url()` 函数将 URI 字符串

分解为多个部分：

```
namespace Application\MiddleWare;
use InvalidArgumentException;
use Psr\Http\Message\UriInterface;
class Uri implements UriInterface
{
  protected $uriString;
  protected $uriParts = array();

  public function __construct($uriString)
  {
    $this->uriParts = parse_url($uriString);
    if (!$this->uriParts) {
      throw new InvalidArgumentException(
        Constants::ERROR_INVALID_URI);
    }
    $this->uriString = $uriString;
  }
```

URI 是 Uniform Resource Indicator（**统一资源标识符**）的首字母缩写词。当你使用浏览器访问网页时，URI 会显示在浏览器的顶部。要详细了解 URI 的各个组成部分，请浏览 `http://tools.ietf.org/html/rfc3986`。

4. 编写好构造器后，应定义用于访问 URI 的各个组成部分的方法。键为 **scheme** 的 `uriParts` 数组元素代表 PHP 封装器（即 HTTP、FTP 等）：

```
public function getScheme()
{
  return strtolower($this->uriParts['scheme']) ?? '';
}
```

5. 定义 `getAuthority()` 方法，使用该方法获取 URI 中的用户名（如果存在用户名）、主机名称（host）和端口号（port）：

```
public function getAuthority()
{
  $val = ' ';
  if (!empty($this->getUserInfo()))
  $val .= $this->getUserInfo() . '@';
  $val .= $this->uriParts['host'] ?? '';
```

```
  if (!empty($this->uriParts['port']))
  $val .= ':' . $this->uriParts['port'];
  return $val;
}
```

6. 编写 `getUserInfo()`方法，使用该方法获取用户名（如果存在用户名）和登录密码。使用登录密码的一个典型例子是访问 FTP 网站（如 `ftp://username:password@website.com:/path`）：

```
public function getUserInfo()
{
  if (empty($this->uriParts['user'])) {
    return ' ';
  }
  $val = $this->uriParts['user'];
  if (!empty($this->uriParts['pass']))
    $val .= ':' . $this->uriParts['pass'];
  return $val;
}
```

7. 定义 `getHost()`方法，使用该方法获取主机名称（host），该主机名称就是 URI 中的 DNS 地址：

```
public function getHost()
{
  if (empty($this->uriParts['host'])) {
    return ' ';
  }
  return strtolower($this->uriParts['host']);
}
```

8. 定义 `getPort()`方法，使用该方法获取端口号（port）。该端口号指的是 HTTP 服务的端口号（如果存在端口号）。注意，如果某个端口号出现在 `STANDARD_PORTS` 常量的端口列表中，那么根据 PSR-7 编程规范，`getPort()`方法就应该返回 `NULL`。

```
public function getPort()
{
  if (empty($this->uriParts['port'])) {
      return NULL;
  } else {
      if ($this->getScheme()) {
        if ($this->uriParts['port'] ==
        Constants::STANDARD_PORTS[$this->getScheme()]) {
```

```
            return NULL;
            }
          }
          return (int) $this->uriParts['port'];
    }
}
```

9. 编写 getPath()方法，使用该方法获取 URI 中位于 DNS 地址之后的路径信息（path）。根据 PSR-7 编程规范，这部分信息必须编码。我们使用 PHP 函数 rawurlencode()进行编码，因为该函数符合 RFC 3986 标准。然而，我们无法仅对路径信息进行编码，因为路径分隔符（即/）也会被编码！因此，需要先使用 explode()函数将路径信息分解为各个组成部分，再对各个组成部分进行编码，然后将它们组装到一起：

```
public function getPath()
{
  if (empty($this->urlParts['path'])) {
    return ' ';
  }
  return implode('/', array_map("rawurlencode",
                 explode('/', $this->urlParts['path'])));
}
```

10. 定义 getQueryParams 方法，使用该方法提取 query 字符串（通过$_GET 变量）。query 字符串也必须进行 URL 编码。先分解 query 字符串，将各个组成部分存储到关联数组中。注意，应设置重置选项，以便能够刷新执行查询操作的参数（即 query 字符串）。然后定义 getQuery()方法，它会将该数组接收为参数，并生成合适的 URL 编码：

```
public function getQueryParams($reset = FALSE)
{
  if ($this->queryParams && !$reset) {
    return $this->queryParams;
  }
  $this->queryParams = [];
  if (!empty($this->uriParts['query'])) {
    foreach (explode('&', $this->uriParts['query']) as $keyPair) {
      list($param,$value) = explode('=',$keyPair);
      $this->queryParams[$param] = $value;
    }
  }
```

```
    return $this->queryParams;
  }

  public function getQuery()
  {
    if (!$this->getQueryParams()) {
      return ' ';
    }
    $output = ' ';
    foreach ($this->getQueryParams() as $key => $value) {
      $output .= rawurlencode($key) . '='
      . rawurlencode($value) . '&';
    }
    return substr($output, 0, -1);
  }
```

11. 编写 `getFragment()` 方法，使用该方法获取分段符号（即 URI 中的#），以及该符号后面的信息：

```
  public function getFragment()
  {
    if (empty($this->urlParts['fragment'])) {
      return ' ';
    }
    return rawurlencode($this->urlParts['fragment']);
  }
```

12. 定义与前面介绍的 `getXXX()` 方法对应的 `withXXX()` 方法。这些方法专门用于添加、替换或移除与请求类有关的属性（如协议、权限、用户信息等）。此外，这些方法还应返回当前实例，以允许我们在连续的调用语句（通常被称为**连贯接口**）中使用这些方法。下面先介绍 `withScheme()` 方法：

应根据 PSR-7 编程规范，使用空参数标识被移除的属性。此外，不允许使用与 `Constants::STANDARD_PORTS` 数组中定义不符的协议。

```
  Public function withscheme($SCHEME)
  {
    if (empty($scheme) && $this->getScheme()) {
        unset($this->uriParts['scheme']);
    } else {
```

```
            if (isset(STANDARD_PORTS[strtolower($scheme)])) {
                $this->uriParts['scheme'] = $scheme;
        } else {
            throw new InvalidArgumentException(
            Constants::ERROR_BAD . __METHOD__);
        }
      }
      return $this;
    }
```

13. 对于实现覆盖、添加或替换用户信息、主机名、端口号、路径、查询操作和分段信息的方法，我们对其应用相同的逻辑。注意，withQuery()方法能够重置存储查询操作参数的数组。我们对 withHost()、withPort()、withPath()和 withFragment()方法应用与 withquery()方法相同的逻辑，为节省篇幅此处没有列出全部方法：

```
    public function withUserInfo($user, $password = null)
    {
      if (empty($user) && $this->getUserInfo()) {
          unset($this->uriParts['user']);
      } else {
          $this->urlParts['user'] = $user;
          if ($password) {
              $this->urlParts['pass'] = $password;
          }
      }
      return $this;
    }
    // 为节省篇幅，此处没有列出 withHost()、withPort()、withPath()
    // 和 withFragment()方法

    public function withQuery($query)
    {
      if (empty($query) && $this->getQuery()) {
          unset($this->uriParts['query']);
      } else {
          $this->uriParts['query'] = $query;
      }
      // 重置存储查询操作参数的数组
      $this->getQueryParams(TRUE);
      return $this;
    }
```

14. 当在字符串上下文中使用 Uri 对象，并通过使用$uriParts 数组组装 URI 的

各个组成部分时，可使用__toString()函数封装 Application\MiddleWare\Uri 类。还可以定义便捷的 getUriString()方法，使该方法仅调用__toString()方法：

```
public function __toString()
{
    $uri = ($this->getScheme())
      ? $this->getScheme() . '://' : ' ';
```

15. 如果 URI 中的 authority 部分存在，就应该添加 authority 信息，包括用户信息、主机名和端口号。如果 URI 中的 authority 部分不存在，就只需添加主机名和端口号：

```
if ($this->getAuthority()) {
     $uri .= $this->getAuthority();
} else {
     $uri .= ($this->getHost()) ? $this->getHost() : ' ';
     $uri .= ($this->getPort())
       ? ':' . $this->getPort() : ' ';
}
```

16. 在添加路径信息前，应先检查其中的第一字符是否为/。如果第一个字符不是/，就需要添加这个分隔符。然后添加查询操作（query）和分段信息（fragment）（如果这两部分信息都存在）：

```
$path = $this->getPath();
if ($path) {
  if ($path[0] != '/') {
       $uri .= '/' . $path;
  } else {
       $uri .= $path;
  }
}
$uri .= ($this->getQuery())
  ? '?' . $this->getQuery() : ' ';
$uri .= ($this->getFragment())
  ? '#' . $this->getFragment() : ' ';
return $uri;
}

public function getUriString()
{
  return $this->__toString();
}

}
```

PHP 7 新增了字符串式非关联化功能（如**$path[0]**）。

17. 下面让我们将注意力转向代表消息主体的类。因为无法事先知道消息主体的尺寸，所以 PSR-7 编程规范建议将消息主体作为**数据流**处理。数据流是一种允许以线性方式输入和输出数据的数据资源。在 PHP 中，所有文件命令都是以 Streams 子系统（一个具有流式行为的资源对象）为基础的，因此 PHP 的自有特点与 PSR-7 编程规范恰好相合。PSR-7 通过 Psr\Http\Message\StreamInterface 接口规定必须实现的 read()、write()、seek() 等方法，以此来实现这种数据流处理特点。我们使用 Application\MiddleWare\Stream 类代表收到或发出的请求和/或回应的主体：

```
namespace Application\MiddleWare;
use SplFileInfo;
use Throwable;
use RuntimeException;
use Psr\Http\Message\StreamInterface;
class Stream implements StreamInterface
{
  protected $stream;
  protected $metadata;
  protected $info;
```

18. 在这个类的构造器中，我们使用简单的 fopen() 命令打开数据流文件。然后使用 stream_get_meta_data() 函数获取数据流中的信息。为了获取其他细节，我们还创建了 SplFileInfo 实例：

```
public function __construct($input, $mode = self::MODE_READ)
{
  $this->stream = fopen($input, $mode);
  $this->metadata = stream_get_meta_data($this->stream);
  $this->info = new SplFileInfo($input);
}
```

我们选择使用 fopen() 函数处理新版本的 SplFileObject 文件对象的原因是，旧版本的 SplFileObject 文件对象不允许直接访问文件内部的资源对象，因而无法为应用程序所用。

19. 创建两个便捷方法，使用它们访问数据资源和 SplFileInfo 实例：

```
public function getStream()
{
  return $this->stream;
}

public function getInfo()
{
  return $this->info;
}
```

20. 定义低层级的处理数据流的核心方法：

```
public function read($length)
{
  if (!fread($this->stream, $length)) {
       throw new RuntimeException(
       self::ERROR_BAD . __METHOD__);
  }
}
public function write($string)
{
  if (!fwrite($this->stream, $string)) {
       throw new RuntimeException(
       self::ERROR_BAD . __METHOD__);
  }
}
public function rewind()
{
  if (!rewind($this->stream)) {
       throw new RuntimeException(
       self::ERROR_BAD . __METHOD__);
  }
}
public function eof()
{
  return eof($this->stream);
}
public function tell()
{
  try {
        return ftell($this->stream);
```

```
    } catch (Throwable $e) {
        throw new RuntimeException(
        self::ERROR_BAD . __METHOD__);
    }
  }
  public function seek($offset, $whence = SEEK_SET)
  {
    try {
        fseek($this->stream, $offset, $whence);
    } catch (Throwable $e) {
        throw new RuntimeException(
        self::ERROR_BAD . __METHOD__);
    }
  }
  public function close()
  {
    if ($this->stream) {
      fclose($this->stream);
    }
  }
  public function detach()
  {
    return $this->close();
  }
```

21. 还应定义用于获取数据流描述信息的方法，以便于了解数据流的基本情况（如尺寸和只读设置等）：

```
  public function getMetadata($key = null)
  {
    if ($key) {
        return $this->metadata[$key] ?? NULL;
    } else {
        return $this->metadata;
    }
  }
  public function getSize()
  {
    return $this->info->getSize();
  }
  public function isSeekable()
  {
```

```
    return boolval($this->metadata['seekable']);
  }
  public function isWritable()
  {
    return $this->stream->isWritable();
  }
  public function isReadable()
  {
    return $this->info->isReadable();
  }
```

22. 遵循 PSR-7 编程规范定义 `getContents()`和`__toString()`方法，以便获取数据流中的数据：

```
  public function __toString()
  {
    $this->rewind();
    return $this->getContents();
  }

  public function getContents()
  {
    ob_start();
    if (!fpassthru($this->stream)) {
      throw new RuntimeException(
      self::ERROR_BAD . __METHOD__);
    }
    return ob_get_clean();
  }
  }
```

23. `TextStream`类是前面介绍的`Stream`类的一个重要修改版本。`TextStream`类专门用于处理主体为字符串（JSON 编码的数组）的数据流，而不是主体为文件的数据流。因为需要确保通过`$input`变量获得的数据必须是字符型数据，所以应在 PHP 程序的开始标签之后启用 PHP 7 严格类型设置模式（`strict_types=1`）。还应设置一个`$pos`属性，使用它模拟文件指针，但实际上它指向的是字符串内部的某个位置：

```
<?php
declare(strict_types=1);
namespace Application\MiddleWare;
use Throwable;
use RuntimeException;
```

```
use SplFileInfo;
use Psr\Http\Message\StreamInterface;

class TextStream implements StreamInterface
{
  protected $stream;
  protected $pos = 0;
```

24. TextStream 类中的大多数方法都非常简单并且易于理解。$stream 属性用于存储输入的字符串：

```
public function __construct(string $input)
{
  $this->stream = $input;
}
public function getStream()
{
  return $this->stream;
}
public function getInfo()
{
  return NULL;
}
public function getContents()
{
  return $this->stream;
}
public function __toString()
{
  return $this->getContents();
}
public function getSize()
{
  return strlen($this->stream);
}
public function close()
{
  // 不做任何操作；毕竟字符串是不需要关闭的!
}
public function detach()
{
  return $this->close(); // 不做任何操作!
}
```

25. 要模拟数据流的行为，可使用 tell()、eof() 和 seek() 等方法处理$pos 变量：

```
public function tell()
{
  return $this->pos;
}
public function eof()
{
  return ($this->pos == strlen($this->stream));
}
public function isSeekable()
{
  return TRUE;
}
public function seek($offset, $whence = NULL)
{
  if ($offset < $this->getSize()) {
      $this->pos = $offset;
  } else {
      throw new RuntimeException(
        Constants::ERROR_BAD . __METHOD__);
  }
}
public function rewind()
{
  $this->pos = 0;
}
public function isWritable()
{
  return TRUE;
}
```

26. 使用 `read()` 和 `write()` 方法处理`$pos` 变量和子字符串：

```
public function write($string)
{
  $temp = substr($this->stream, 0, $this->pos);
  $this->stream = $temp . $string;
  $this->pos = strlen($this->stream);
}

public function isReadable()
{
  return TRUE;
```

```
}
public function read($length)
{
  return substr($this->stream, $this->pos, $length);
}
public function getMetadata($key = null)
{
  return NULL;
}

}
```

27. 下面介绍本节中的最后一个值对象 Application\MiddleWare\Uploaded File。像编写其他类一样，应先定义在文件上传操作中会用到的属性：

```
namespace Application\MiddleWare;
use RuntimeException;
use InvalidArgumentException;
use Psr\Http\Message\UploadedFileInterface;
class UploadedFile implements UploadedFileInterface
{

  protected $field;     // 上传文件的原始名称
  protected $info;      // 用于获取$_FILES[$field]变量的值
  protected $randomize;
  protected $movedName = ' ';
```

28. 在 UploadedFile 类的构造器中，应定义通过$field 变量上传的文件的名称属性，以及$_FILES 变量中的对应数组。我们添加该构造器中的最后一个参数，用它来标识在确定了要上传的文件时，UploadedFile 类是否生成新的随机文件名：

```
public function __construct(
  $field, array $info, $randomize = FALSE)
{
  $this->field = $field;
  $this->info = $info;
  $this->randomize = $randomize;
}
```

29. 创建 Stream 实例，使用该实例代表临时的或已移除的文件：

```
public function getStream()
{
  if (!$this->stream) {
```

```
    if ($this->movedName) {
        $this->stream = new Stream($this->movedName);
    } else {
        $this->stream = new Stream($info['tmp_name']);
    }
  }
  return $this->stream;
}
```

30. 定义 moveTo()方法，使用该方法执行实际的文件移除操作。注意，添加大量的安全检查操作有助于预防注入攻击。如果生成随机文件名的功能没有被启用，就使用用户提供的原始文件名：

```
public function moveTo($targetPath)
{
  if ($this->moved) {
      throw new Exception(Constants::ERROR_MOVE_DONE);
  }
  if (!file_exists($targetPath)) {
      throw new InvalidArgumentException(Constants::ERROR_BAD_DIR);
  }
  $tempFile = $this->info['tmp_name'] ?? FALSE;
  if (!$tempFile || !file_exists($tempFile)) {
      throw new Exception(Constants::ERROR_BAD_FILE);
  }
  if (!is_uploaded_file($tempFile)) {
    throw new Exception(Constants::ERROR_FILE_NOT);
  }
  if ($this->randomize) {
      $final = bin2hex(random_bytes(8)) . '.txt';
  } else {
      $final = $this->info['name'];
  }
  $final = $targetPath . '/' . $final;
  $final = str_replace('//', '/', $final);
  if (!move_uploaded_file($tempFile, $final)) {
      throw new RuntimeException(Constants::ERROR_MOVE_UNABLE);
  }
  $this->movedName = $final;
  return TRUE;
}
```

31. 可通过$info 属性访问使用$_FILES 变量返回的其他参数。请注意，应将getClientFilename()和 getClientMediaType()方法的返回值视为不可信的，因为这些值是通过外部输入的信息生成的。我们还添加了一个方法，通过它返回被移除文件的名称：

```
public function getMovedName()
{
  return $this->movedName ?? NULL;
}
public function getSize()
{
  return $this->info['size'] ?? NULL;
}
public function getError()
{
  if (!$this->moved) {
       return UPLOAD_ERR_OK;
  }
  return $this->info['error'];
}
public function getClientFilename()
{
  return $this->info['name'] ?? NULL;
}
public function getClientMediaType()
{
  return $this->info['type'] ?? NULL;
}

}
```

具体运行情况

浏览 https://github.com/php-fig/http-message/tree/master/src，为 PSR-7 接口下载 GitHub 仓库（开源代码库以及版本控制系统）。在/path/to/source 目录中创建 Psr/Http/Message 目录，并将下载到的这些仓库文件存储到该目录中。也可以浏览 https://packagist.org/packages/psr/http-message，并使用 Composer 安装这些源代码（要详细了解如何获得和使用 Composer，可浏览 https://getcomposer.org/）。

使用前面具体处理过程中介绍的代码定义下列类：

类	对应的步骤
Application\MiddleWare\Constants	2
Application\MiddleWare\Uri	3 ~ 16
Application\MiddleWare\Stream	17 ~ 22
Application\MiddleWare\TextStream	23 ~ 26
Application\MiddleWare\UploadedFile	27 ~ 31

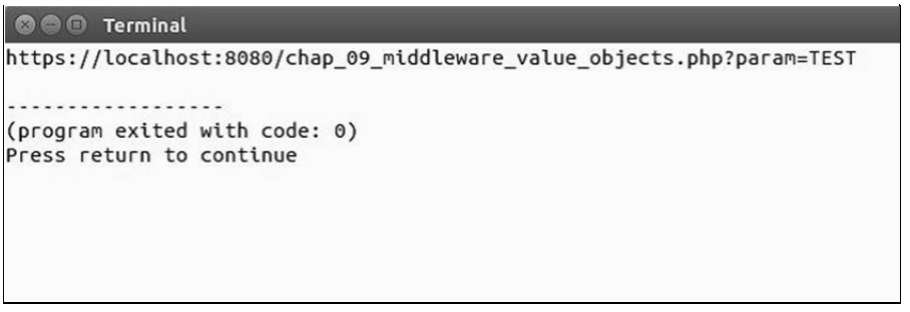
定义调用程序 chap_09_middleware_value_objects_uri.php，为其设置类自动加载功能，并引用合适的类。请注意，如果你使用了 Composer，除非进行了专门的设置，否则 Composer 会自动创建一个名为 vendor 的文件夹。Composer 也会添加本身的自动加载器，你可以随意使用它：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\Uri;
```

创建一个 Uri 实例，并使用 with*方法添加参数。这样就可以像定义了__toString()方法那样直接显示 Uri 实例：

```
$uri = new Uri();
$uri->withScheme('https')
    ->withHost('localhost')
    ->withPort('8080')
    ->withPath('chap_09_middleware_value_objects_uri.php')
    ->withQuery('param=TEST');

echo $uri;
```

下面是输出结果：



在/path/to/source/for/this/chapter 目录中创建 uploads 目录。定义另一个调用程序 chap_09_middleware_value_objects_file_upload.php，为其设置类自动加载功能，并引用合适的类：

```
<?php
define('TARGET_DIR', __DIR__ . '/uploads');
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\UploadedFile;
```

使用 try...catch 代码块检查是否已经有被上传的文件。如果发现了被上传的文件，循环遍历$_FILES 遍历的内容，并在设置 tmp_name 变量的位置创建 UploadedFile 实例。这样就可以使用 moveTo()方法将这些文件移动到 TARGET_DIR 目录中：

```
try {
    $message = ' ';
    $uploadedFiles = array();
    if (isset($_FILES)) {
        foreach ($_FILES as $key => $info) {
          if ($info['tmp_name']) {
              $uploadedFiles[$key] = new UploadedFile(
                                          $key, $info, TRUE);
              $uploadedFiles[$key]->moveTo(TARGET_DIR);
          }
        }
    }
} catch (Throwable $e) {
    $message = $e->getMessage();
}
?>
```

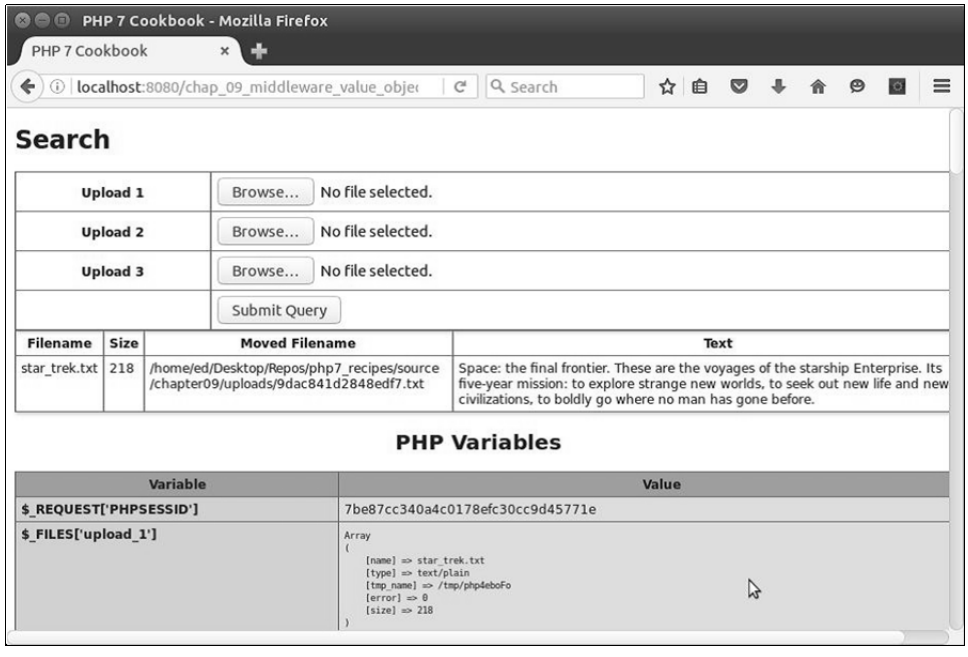
使用查看逻辑显示一个简单的上传文件表单，还可以使用 phpinfo()方法显示被上传文件的信息：

```
<form name="search" method="post"
  enctype="<?= Constants::CONTENT_TYPE_MULTI_FORM ?>">
<table class="display" cellspacing="0" width="100%">
   <tr><th>Upload 1</th><td><input type="file" name="upload_1" /></
td></tr>
   <tr><th>Upload 2</th><td><input type="file" name="upload_2" /></
td></tr>
   <tr><th>Upload 3</th><td><input type="file" name="upload_3" /></
td></tr>
   <tr><th> </th><td><input type="submit" /></td></tr>
</table>
</form>
<?= ($message) ? '<h1>' . $message . '</h1>' : ' '; ?>
```

如果有被上传的文件,我们就可以显示每个文件的信息。还可以使用 getStream() 方法（其后紧接着的是 getcontents() 方法）显示每个文件（如果处理的是较短的文本文件）：

```
<?php if ($uploadedFiles) : ?>
<table class="display" cellspacing="0" width="100%">
   <tr>
      <th>Filename</th><th>Size</th>
    <th>Moved Filename</th><th>Text</th>
   </tr>
   <?php foreach ($uploadedFiles as $obj) : ?>
        <?php if ($obj->getMovedName()) : ?>
        <tr>
          <td><?= htmlspecialchars($obj->getClientFilename()) ?></td>
          <td><?= $obj->getSize() ?></td>
          <td><?= $obj->getMovedName() ?></td>
          <td><?= $obj->getStream()->getContents() ?></td>
          </tr>
          <?php endif; ?>
    <?php endforeach; ?>
</table>
<?php endif; ?>
<?php phpinfo(INFO_VARIABLES); ?>
```

下面是在浏览器中的输出效果：



扩展

- 要详细了解 PSR 编程规范，请浏览 `https://en.wikipedia.org/wiki/PHP_ Standard_Recommendation`。
- 要了解 PSR-7 编程规范的官方说明，请浏览 `http://www.php-fig.org/psr/psr-7/`。
- 要详细了解 PHP 数据流，请浏览 `http://php.net/manual/en/book.stream.php`。

开发 PSR-7 请求类

PSR-7 中间件的重要特点之一是会使用代表**请求**和**回应**的类。通过使用请求和回应类，可以使软件中的各个模块在不建立过多依赖关系的情况下一起执行操作。在这种情况中，请求类应包含用户原始请求的所有成分，其中包括浏览器设置、最初请求访问的 URL 和已经发送的参数等。

具体处理过程

1. 请确保已经定义了代表前面介绍过的 Uri、Stream 和 UploadedFile 值对象的类。

2. 现在可以定义核心的 Application\MiddleWare\Message 类处理 Stream 和 Uri 对象，并实现 Psr\Http\Message\MessageInterface 接口。应先定义用于处理关键值对象的属性，其中包括用于存储消息主体（即 StreamInterface 实例）、消息版本和 HTTP 报头的属性：

```
namespace Application\MiddleWare;
use Psr\Http\Message\ {
  MessageInterface,
  StreamInterface,
  UriInterface
};
class Message implements MessageInterface
{
  protected $body;
  protected $version;
  protected $httpHeaders = array();
```

3. 编写代表 StreamInterface 实例的 getBody()方法（用于获取消息主体）。定义 withBody()方法，使该方法返回代表当前消息的实例，并使我们能够覆盖消息主体的当前值：

```
public function getBody()
{
  if (!$this->body) {
      $this->body = new Stream(self::DEFAULT_BODY_STREAM);
  }
  return $this->body;
}
public function withBody(StreamInterface $body)
{
  if (!$body->isReadable()) {
      throw new InvalidArgumentException(
              self::ERROR_BODY_UNREADABLE);
  }
  $this->body = $body;
  return $this;
}
```

4. PSR-7 编程规范建议使用不区分大小写的形式标识报头。因此，我们定义了 findHeader() 方法（MessageInterface 接口没有直接定义该方法），我们通过该方法使用 stripos() 函数找到报头：

```
protected function findHeader($name)
{
  $found = FALSE;
  foreach (array_keys($this->getHeaders()) as $header) {
    if (stripos($header, $name) !== FALSE) {
        $found = $header;
        break;
    }
  }
  return $found;
}
```

5. 下面的方法不是由 PSR-7 编程规范定义的，该方法专门用于为$httpHeaders 属性赋值。$httpHeaders 属性存储的是一个关联数组，其中的键为报头名称，其中的值为代表报头值的字符串。如果报头中含有一个以上的值，额外的值会被逗号分隔。Apache 扩展提供了一个非常有用的 PHP 函数 apache_request_headers(),使用该函数可以在无法通过$httpHeaders 变量获取报头的情况下生成报头：

```
protected function getHttpHeaders()
{
  if (!$this->httpHeaders) {
      if (function_exists('apache_request_headers')) {
          $this->httpHeaders = apache_request_headers();
      } else {
          $this->httpHeaders = $this->altApacheReqHeaders();
      }
  }
  return $this->httpHeaders;
}
```

6. 如果无法使用 apache_request_headers() 函数(即无法启用 Apache 扩展),也可以使用替代方法 altApacheReqHeaders()：

```
protected function altApacheReqHeaders()
{
  $headers = array();
  foreach ($_SERVER as $key => $value) {
    if (stripos($key, 'HTTP_') !== FALSE) {
```

```
            $headerKey = str_ireplace('HTTP_', '', $key);
            $headers[$this->explodeHeader($headerKey)] = $value;
        } elseif (stripos($key, 'CONTENT_') !== FALSE) {
            $headers[$this->explodeHeader($key)] = $value;
        }
    }
    return $headers;
}
protected function explodeHeader($header)
{
    $headerParts = explode('_', $header);
    $headerKey = ucwords(implode(' ', strtolower($headerParts)));
    return str_replace(' ', '-', $headerKey);
}
```

7. 实现 getHeaders()方法，该方法是 PSR-7 编程规范要求必须实现的方法。现在实现它非常简单，只需循环遍历由前面步骤 5 介绍的 getHttpHeaders()方法生成的$httpHeaders 属性：

```
public function getHeaders()
{
    foreach ($this->getHttpHeaders() as $key => $value) {
        header($key . ': ' . $value);
    }
}
```

8. 编写一系列 with*方法，使用这些方法覆盖或替换报头。因为需要处理的报头数量可能会很多，还应编写用于向已存在报头分组中添加新报头的方法。withoutHeader()方法用于删除报头。注意，在编写这些方法时都应使用前面介绍过的 findHeader()方法，它允许以不区分大小写的形式处理报头：

```
public function withHeader($name, $value)
{
    $found = $this->findHeader($name);
    if ($found) {
        $this->httpHeaders[$found] = $value;
    } else {
        $this->httpHeaders[$name] = $value;
    }
    return $this;
}
```

```
public function withAddedHeader($name, $value)
{
  $found = $this->findHeader($name);
  if ($found) {
      $this->httpHeaders[$found] .= $value;
  } else {
      $this->httpHeaders[$name] = $value;
  }
  return $this;
}

public function withoutHeader($name)
{
  $found = $this->findHeader($name);
  if ($found) {
      unset($this->httpHeaders[$found]);
  }
  return $this;
}
```

9. 接下来遵循 PSR-7 编程规范编写一系列与报头有关的方法，使用这些方法来确定报头确实存在、获取单行报头内容，以及获取存储在数组中的报头：

```
public function hasHeader($name)
{
  return boolval($this->findHeader($name));
}

public function getHeaderLine($name)
{
  $found = $this->findHeader($name);
  if ($found) {
      return $this->httpHeaders[$found];
  } else {
      return ' ';
  }
}

public function getHeader($name)
{
  $line = $this->getHeaderLine($name);
  if ($line) {
```

```
            return explode(',', $line);
        } else {
            return array();
        }
    }
```

10. 为了尽善尽美地处理报头，我们编写了 getHeadersAsString() 方法，该方法会以“报头名称：报头内容”的形式显示报头，并且每行只显示一个报头（通过在报头字符串的头部添加\r\n 符号）。这样通过该方法就能够在 PHP 数据流上下文中直接使用报头：

```
    public function getHeadersAsString()
    {
        $output = ' ';
        $headers = $this->getHeaders();
        if ($headers && is_array($headers)) {
            foreach ($headers as $key => $value) {
                if ($output) {
                    $output .= "\r\n" . $key . ': ' . $value;
                } else {
                    $output .= $key . ': ' . $value;
                }
            }
        }
        return $output;
    }
```

11. 继续编写 Message 类，但请将注意力转向版本处理工作。根据 PSR-7 编程规范，返回的代表传输协议版本（即 HTTP/1.1）的值只能是数字。因此，还应编写 onlyVersion() 方法，使用该方法去掉版本返回值中的所有非数字字符：

```
    public function getProtocolVersion()
    {
        if (!$this->version) {
            $this->version = $this->onlyVersion(
                $_SERVER['SERVER_PROTOCOL']);
        }
        return $this->version;
    }

    public function withProtocolVersion($version)
    {
```

```
    $this->version = $this->onlyVersion($version);
    return $this;
  }

  protected function onlyVersion($version)
  {
    if (!empty($version)) {
        return preg_replace('/[^0-9\.]/', ' ', $version);
    } else {
        return NULL;
    }
  }

  }
```

12. 完成了前面的准备工作，现在可以定义 Request 类了。必须注意，我们既需要考虑向外发送的请求，也需要考虑接收到的请求。换言之，需要使用一个类代表从客户端向服务器发送的请求，再使用另一个类代表服务器从客户端那里接收到的请求。因此，应编写 Application\MiddleWare\Request 类（代表从客户端向服务器发送的请求）和 Application\MiddleWare\ServerRequest 类（代表服务器从客户端接收到的请求）。好消息是大部分工作其实已经完成，注意，Request 类扩展了 Message 类。我们还需要定义代表 URI 和 HTTP 方法的属性：

```
namespace Application\MiddleWare;

use InvalidArgumentException;
use Psr\Http\Message\ { RequestInterface, StreamInterface,
UriInterface };

class Request extends Message implements RequestInterface
{
  protected $uri;
  protected $method; // HTTP 方法
  protected $uriObj; // 实现了 Psr\Http\Message\UriInterface 接口的实例
```

13. 在 Request 类的构造器中，所有属性的默认值都被设置为 NULL，但其中保留了立刻定义适当参数的可能性。可使用通过继承父类得到的 onlyVersion()方法过滤代表传输协议版本的值，从而使该值中仅含有数字。还应定义 checkMethod()方法，以便确保通过$method 参数获得的方法都是在常量数组 Constants 中定义的 HTTP 方法：

```
public function __construct($uri = NULL,
                            $method = NULL,
                            StreamInterface $body = NULL,
                            $headers = NULL,
                            $version = NULL)
{
  $this->uri = $uri;
  $this->body = $body;
  $this->method = $this->checkMethod($method);
  $this->httpHeaders = $headers;
  $this->version = $this->onlyVersion($version);
}
protected function checkMethod($method)
{
  if (!$method === NULL) {
    if (!in_array(strtolower($method), Constants::HTTP_METHODS)) {
        throw new InvalidArgumentException(
        Constants::ERROR_HTTP_METHOD);
    }
  }
  return $method;
}
```

14. 我们在字符串表单中将请求目标解析为被请求的原始 URI。注意，前面介绍过的 `Uri` 类中含有能够解析请求目标并将这类信息吸收到本身实例中的方法，因此可以将 `Uri` 实例赋予`$uriObj` 属性。在处理 `withRequestTarget()`方法时，注意应调用 `getUri()`方法执行上述解析处理过程：

```
public function getRequestTarget()
{
  return $this->uri ?? Constants::DEFAULT_REQUEST_TARGET;
}

public function withRequestTarget($requestTarget)
{
  $this->uri = $requestTarget;
  $this->getUri();
  return $this;
}
```

15. 代表HTTP方法的`get*`和`with*`方法并无特别之处。可使用`checkMethod()`方法（在 Request 类的构造器中使用过的）确保获得我们想要的方法：

```
public function getMethod()
```

```
{
  return $this->method;
}

public function withMethod($method)
{
  $this->method = $this->checkMethod($method);
  return $this;
}
```

16. 为 URI 编写 get*和 with*方法。如前面步骤 14 所述，代表原始请求的字符串存储在$uri 属性和$uriObj 变量中最新解析的 Uri 实例中。注意，添加额外的标记是为了保留所有已存在的 Host 报头：

```
public function getUri()
{
  if (!$this->uriObj) {
       $this->uriObj = new Uri($this->uri);
  }
  return $this->uriObj;
}

public function withUri(UriInterface $uri, $preserveHost = false)
{
  if ($preserveHost) {
    $found = $this->findHeader(Constants::HEADER_HOST);
    if (!$found && $uri->getHost()) {
      $this->httpHeaders[Constants::HEADER_HOST] = $uri->getHost();
    }
  } elseif ($uri->getHost()) {
       $this->httpHeaders[Constants::HEADER_HOST] = $uri->getHost();
  }
  $this->uri = $uri->__toString();
  return $this;
  }
}
```

17. 通过扩展 Request 类定义 ServerRequest 类，并为 ServerRequest 类添加额外的功能，以便使服务器能够通过处理收到的请求，获取它感兴趣的信息。先定义代表收到数据的属性，这些数据是通过读取各种 PHP 超级全局变量（如$_SERVER、$_POST 等）获得的：

```
namespace Application\MiddleWare;
use Psr\Http\Message\ { ServerRequestInterface,
UploadedFileInterface } ;

class ServerRequest extends Request implements
ServerRequestInterface
{

  protected $serverParams;
  protected $cookies;
  protected $queryParams;
  protected $contentType;
  protected $parsedBody;
  protected $attributes;
  protected $method;
  protected $uploadedFileInfo;
  protected $uploadedFileObjs;
```

18. 定义一系列读取器，以获取超级全局变量中存储的信息。为节省篇幅，此处没有列出所有方法：

```
public function getServerParams()
{
  if (!$this->serverParams) {
       $this->serverParams = $_SERVER;
  }
  return $this->serverParams;
}
// getCookieParams()方法应读取$_COOKIE 变量
// getQueryParams()方法应读取$_GET
// getUploadedFileInfo()方法应读取$_FILES

public function getRequestMethod()
{
  $method = $this->getServerParams()['REQUEST_METHOD'] ?? '';
  $this->method = strtolower($method);
  return $this->method;
}

public function getContentType()
{
  if (!$this->contentType) {
       $this->contentType =
       $this->getServerParams()['CONTENT_TYPE'] ?? ' ';
       $this->contentType = strtolower($this->contentType);
```

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

19. 因为被上传的文件应该由独立的 UploadedFile 对象（请参阅前面的内容）代表,所以还应定义处理$uploadedFileInfo 变量和创建 UploadedFile 对象的方法：

```
  public function getUploadedFiles()
  {
    if (!$this->uploadedFileObjs) {
      foreach ($this->getUploadedFileInfo() as $field => $value) {
        $this->uploadedFileObjs[$field] =
        new UploadedFile($field, $value);
      }
    }
    return $this->uploadedFileObjs;
  }
```

20. 像前面定义的其他类一样，也应为 ServerRequest 类编写 with*方法，以便使用这些方法添加或覆盖属性值，以及返回新的实例：

```
  public function withCookieParams(array $cookies)
  {
    array_merge($this->getCookieParams(), $cookies);
    return $this;
  }
  public function withQueryParams(array $query)
  {
    array_merge($this->getQueryParams(), $query);
    return $this;
  }
  public function withUploadedFiles(array $uploadedFiles)
  {
    if (!count($uploadedFiles)) {
        throw new InvalidArgumentException(
        Constant::ERROR_NO_UPLOADED_FILES);
    }
    foreach ($uploadedFiles as $fileObj) {
      if (!$fileObj instanceof UploadedFileInterface) {
          throw new InvalidArgumentException(
          Constant::ERROR_INVALID_UPLOADED);
      }
    }
```

```
    $this->uploadedFileObjs = $uploadedFiles;
}
```

21. PSR-7 消息的一个要点，是在解析方式中也能够获得消息主体。换言之，应使用结构化表达模式处理消息，而不是仅用未加工的数据流代表消息。因此，应定义 `getParsedBody()` 方法和与之对应的 `withParsedBody()` 方法。PSR-7 编程规范对表单提交操作的建议非常明确。注意，下面介绍的一系列 `if` 语句是用于检查报头内容的数据类型和其中含有的方法的：

```
public function getParsedBody()
{
    if (!$this->parsedBody) {
      if (($this->getContentType() ==
           Constants::CONTENT_TYPE_FORM_ENCODED
           || $this->getContentType() ==
           Constants::CONTENT_TYPE_MULTI_FORM)
           && $this->getRequestMethod() ==
           Constants::METHOD_POST)
    {
           $this->parsedBody = $_POST;
    } elseif ($this->getContentType() ==
              Constants::CONTENT_TYPE_JSON
              || $this->getContentType() ==
              Constants::CONTENT_TYPE_HAL_JSON)
    {
      ini_set("allow_url_fopen", true);
      $this->parsedBody =
        json_decode(file_get_contents('php://input'));
    } elseif (!empty($_REQUEST)) {
        $this->parsedBody = $_REQUEST;
    } else {
        ini_set("allow_url_fopen", true);
        $this->parsedBody = file_get_contents('php://input');
    }
  }
  return $this->parsedBody;
}

public function withParsedBody($data)
{
```

```
    $this->parsedBody = $data;
    return $this;
  }
```

22. 还应在该类中添加未在 PSR-7 编程规范中定义的属性。更确切地说，保留这个开放通道的目的是，使其他开发者能够为我们编写的应用程序提供适当的支持。注意，使用 `withoutAttributes()` 方法，可以移除任何附加属性：

```
  public function getAttributes()
  {
    return $this->attributes;
  }
  public function getAttribute($name, $default = NULL)
  {
    return $this->attributes[$name] ?? $default;
  }
  public function withAttribute($name, $value)
  {
    $this->attributes[$name] = $value;
    return $this;
  }
  public function withoutAttribute($name)
  {
    if (isset($this->attributes[$name])) {
        unset($this->attributes[$name]);
    }
    return $this;
  }

  }
```

23. 为了通过收到的请求加载不同的属性，应定义 `initialize()` 方法，这不是 PSR-7 编程规范的要求，但该方法会提供极大的便利：

```
  public function initialize()
  {
    $this->getServerParams();
    $this->getCookieParams();
    $this->getQueryParams();
    $this->getUploadedFiles;
    $this->getRequestMethod();
    $this->getContentType();
```

```
        $this->getParsedBody();
        return $this;
    }
```

具体运行情况

不要忘记使用上一节介绍的代码创建 Uri、Stream 和 UploadedFile 值对象，因为 Message 和 Request 类都会用到这些对象。使用本节具体处理过程中介绍的代码创建下列类：

类	对应的步骤
Application\MiddleWare\Message	2 ~ 9
Application\MiddleWare\Request	10 ~ 14
Application\MiddleWare\ServerRequest	15 ~ 20

定义在服务器上运行的程序 chap_09_middleware_server.php，为其设置类自动加载功能，并引用合适的类。该脚本会初始化 ServerRequest 实例，将收到的请求赋予该实例，然后使用 var_dump() 函数显示收到的信息：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\ServerRequest;

$request = new ServerRequest();
$request->initialize();
echo '<pre>', var_dump($request), '</pre>';
```

要运行这个服务器端程序，应先切换到/path/to/source/for/this/chapter 文件夹，然后运行下面的命令：

```
php -S localhost:8080 chap_09_middleware_server.php
```

要在客户端运行发送请求的程序，可在客户端创建调用程序 chap_09_middleware_request.php，为其设置类自动加载功能，并引用合适的类。然后定义目标服务器和本地的文本文件：

```
<?php
define('READ_FILE', __DIR__ . '/gettysburg.txt');
define('TEST_SERVER', 'http://localhost:8080');
```

```
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\ { Request, Stream, Constants };
```

可将该文本文件用作数据源来创建一个 Stream 实例。Stream 实例将会变为新请求的主体，本例会使用该实例反映通过表单提交的内容：

```
$body = new Stream(READ_FILE);
```

然后可以直接创建 Request 实例，并为之提供合适的参数：

```
$request = new Request(
   TEST_SERVER,
   Constants::METHOD_POST,
   $body,
   [Constants::HEADER_CONTENT_TYPE =>
     Constants::CONTENT_TYPE_FORM_ENCODED,
     Constants::HEADER_CONTENT_LENGTH => $body->getSize()]
);
```

使用连贯接口语法也可以获得完全相同的结果：

```
$uriObj = new Uri(TEST_SERVER);
$request = new Request();
$request->withRequestTarget(TEST_SERVER)
       ->withMethod(Constants::METHOD_POST)
       ->withBody($body)
       ->withHeader(Constants::HEADER_CONTENT_TYPE,
         Constants::CONTENT_TYPE_FORM_ENCODED)
       ->withAddedHeader(
         Constants::HEADER_CONTENT_LENGTH, $body->getSize());
```

设置 cURL 资源来模仿表单提交操作，在这类情况中数据参数就是文本文件的内容，因而就可以使用 curl_init()、curl_exec() 等方法显示结果：

```
$data = http_build_query(['data' =>
   $request->getBody()->getContents()]);
$defaults = array(
   CURLOPT_URL => $request->getUri()->getUriString(),
   CURLOPT_POST => true,
   CURLOPT_POSTFIELDS => $data,
);
$ch = curl_init();
curl_setopt_array($ch, $defaults);
$response = curl_exec($ch);
curl_close($ch);
```

下面是直接显示的输出结果：

扩展

Matthew Weir O 'Phinney 是 PSR-7 编程规范的编辑，也是 Zend Framework1、2 和 3 的首席架构师，他撰写了一篇非常优秀的文章，介绍了应用 PSR-7 编程规范的经典范例，参见 `https://mwop.net/blog/2015-01-26-psr-7-by-example.html`。

定义 PSR-7 回应类

回应类代表向发送原始请求的实体回复的信息。在这种情况中，HTTP 报头扮演着重要的角色，因为我们需要了解客户端请求获取的数据格式（该信息通常封装在接收到的 `Accept` 报头中）。因此还需要在回应类中设置合适的 `Content-Type` 报头（代表数据类型），以便使回复数据的格式与客户端请求的格式相符。如果不考虑客户端请求

的数据格式，那么回应消息的主体（实际提供给客户端的数据）就会使用 HTML、JSON 或之前被请求（发送）的格式。

具体处理过程

1. 实际上，回应类（Response）比请求类容易实现得多，因为我们只需考虑服务器回复给客户端的消息。此外，可以通过扩展刚才编写的 Application\ MiddleWare\ Message 类创建回应类，因此大部分工作其实已经完成。剩余的工作仅是定义 Application\MiddleWare\Response 类。你可能已经注意到，该类仅有一个属性（$statusCode）：

```
namespace Application\MiddleWare;
use Psr\Http\Message\ { Constants, ResponseInterface,
StreamInterface };
class Response extends Message implements ResponseInterface
{
  protected $statusCode;
```

2. PSR-7 编程规范没有定义下面的构造器，但该构造器可以为我们提供便利，进而创建出完整无缺的 Response 实例。应使用 Message 类中的方法和 Constants 类中的常量验证该构造器的参数：

```
public function __construct($statusCode = NULL,
                            StreamInterface $body = NULL,
                            $headers = NULL,
                            $version = NULL)
{
  $this->body = $body;
  $this->status['code'] = $statusCode
    ?? Constants::DEFAULT_STATUS_CODE;
  $this->status['reason'] =
    Constants::STATUS_CODES[$statusCode] ?? '';
  $this->httpHeaders = $headers;
  $this->version = $this->onlyVersion($version);
  if ($statusCode) $this->setStatusCode();
}
```

3. 使用 PHP 5.4 中引入的 http_response_code() 函数，可以在不受任何报头影响的情况下以非常好的方式设置 HTTP 状态码。因为我们使用的是 PHP 7，所以该函数是一定存在的：

```
public function setStatusCode()
{
  http_response_code($this->getStatusCode());
}
```

4. 如果你使用其他 PHP 版本时无法确定 `http_response_code()` 函数是否存在，也可以使用下面的方法获取状态码：

```
public function getStatusCode()
{
  return $this->status['code'];
}
```

5. 像前面介绍的符合 PSR-7 编程规范的类一样，还应为 `Response` 类定义 `with*` 方法，用该方法设置状态码并返回当前的实例。注意，应使用 `STATUS_CODES` 常量确认该验证码确实存在：

```
public function withStatus($statusCode, $reasonPhrase = '')
{
  if (!isset(Constants::STATUS_CODES[$statusCode])) {
    throw new InvalidArgumentException(
    Constants::ERROR_INVALID_STATUS);
  }
  $this->status['code'] = $statusCode;
  $this->status['reason'] = ($reasonPhrase)
    ? Constants::STATUS_CODES[$statusCode] : NULL;
  $this->setStatusCode();
  return $this;
}
```

6. 定义一个方法，使用该方法返回 HTTP 状态码代表的错误的出现原因，这段信息的形式为一个简短的文本短语，在本例中是基于 RFC 7231 标准的。请注意 PHP 7 中空值合并操作符（null coalesce operator，即`??`）的用法，本例使用该操作符在 3 个选项中返回了第一个非空选项：

```
public function getReasonPhrase()
{
  return $this->status['reason']
    ?? Constants::STATUS_CODES[$this->status['code']]
    ?? ' ';
  }
}
```

具体运行情况

不要忘记定义前两节介绍的类。创建另一个在服务器上运行的简单程序：chap_09_middleware_server_with_response.php，为其设置类自动加载功能，并引用合适的类：

```
<?php
require __DIR__ . '/../Application/Autoload/Loader.php';
Application\Autoload\Loader::init(__DIR__ . '/..');
use Application\MiddleWare\ { Constants, ServerRequest, Response,
  Stream };
```

定义一个存储键/值对的数组，将该数组的值设置为存储在当前目录中的、将被用作数据源的文件的名称：

```
$data = [
  1 => 'churchill.txt',
  2 => 'gettysburg.txt',
  3 => 'star_trek.txt'
];
```

在 try...catch 代码块中初始化一些变量，创建一个 ServerRequest 实例（代表服务器从客户端接收到的请求），并设置一个临时文件名：

```
try {

    $body['text'] = 'Initial State';
    $request = new ServerRequest();
    $request->initialize();
    $tempFile = bin2hex(random_bytes(8)) . '.txt';
    $code = 200;
```

检查 ServerRequest 实例中的方法是 GET*方法（代表请求获取数据）还是 POST*方法（代表请求提交数据）。如果 ServerRequest 实例中的方法是 GET*方法，就检查该请求中是否含有 id 参数。如果含有 id 参数，就返回符合条件的文本文件的内容；如果没有 id 参数，就返回一个文本文件列表：

```
if ($request->getMethod() == Constants::METHOD_GET) {
    $id = $request->getQueryParams()['id'] ?? NULL;
    $id = (int) $id;
    if ($id && $id <= count($data)) {
       $body['text'] = file_get_contents(
```

```
            __DIR__ . '/' . $data[$id]);
        } else {
            $body['text'] = $data;
        }
```

如果 ServerRequest 实例中的方法是 POST*方法，就返回一个回应实例，在该实例中包含代表提交操作执行成功的代码 204，以及服务器收到的请求主体的大小：

```
} elseif ($request->getMethod() == Constants::METHOD_POST) {
    $size = $request->getBody()->getSize();
    $body['text'] = $size . ' bytes of data received';
    if ($size) {
        $code = 201;
    } else {
        $code = 204;
    }
}
```

可以在 catch 代码块中捕捉任何异常，并使用状态码 500 代表它们：

```
} catch (Exception $e) {
    $code = 500;
    $body['text'] = 'ERROR: ' . $e->getMessage();
}
```

应将回应封装成数据流，以便将回应的主体写入临时文件，并将其创建为 Stream 实例。还可以将 Content-Type 报头设置为 application/json 字符串（代表数据格式为 JSON），然后调用 getHeaders()方法，该方法会输出当前的报头集合。之后显示回应的主体。为了观察具体情况，你还可以显示 Response 实例，以确定它拥有正确的结构：

```
try {
    file_put_contents($tempFile, json_encode($body));
    $body = new Stream($tempFile);
    $header[Constants::HEADER_CONTENT_TYPE] = 'application/json';
    $response = new Response($code, $body, $header);
    $response->getHeaders();
    echo $response->getBody()->getContents() . PHP_EOL;
    var_dump($response);
```

为了捕捉所有带有 Throwable 接口的错误和异常，应将这些代码都封装在 try...catch 代码块中，而且不要忘记删除临时文件：

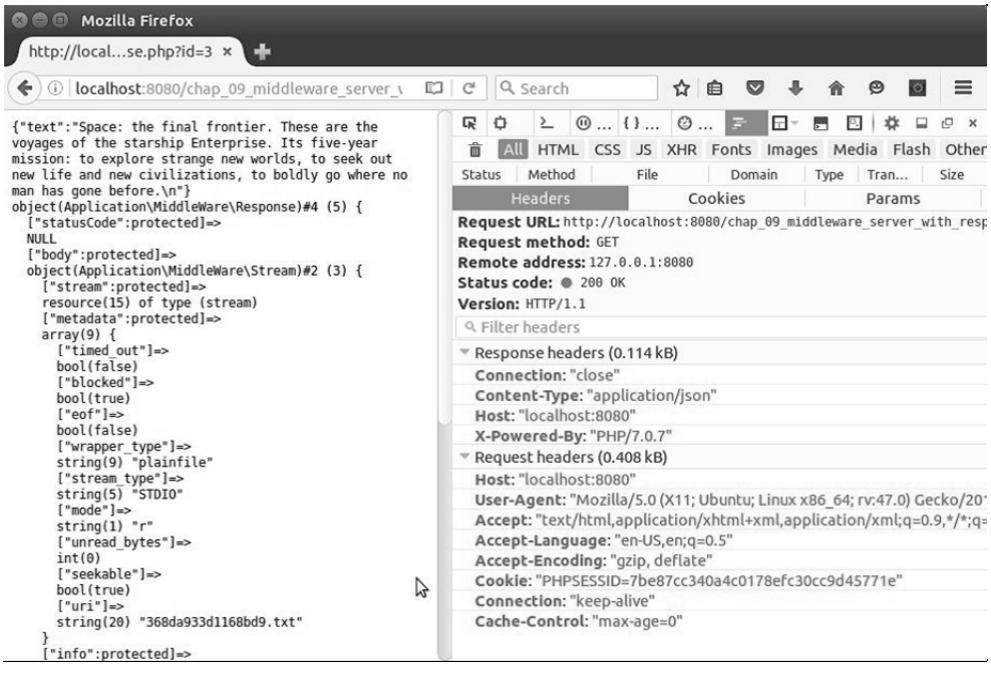
```
} catch (Throwable $e) {
    echo $e->getMessage();
} finally {
```

```
    unlink($tempFile);
}
```

要测试该程序，只需打开终端窗口，切换到/path/to/source/for/this/chapter 目录，并运行下面的命令：

```
php -S localhost:8080
```

可以通过添加 id 参数，使用浏览器运行该程序。你可以打开开发者工具并监控回应的报头。下面是一个输出结果示例，注意回应的数据类型为"application/json"：



扩展

- 要详细了解 PSR 编程规范，请浏览 http://www.php-fig.org/psr/。
- 在我撰写本书时，遵循 PSR-7 编程规范的框架如下。没有被列出的框架可能完全不支持 PSR-7 编程规范，也可能在其说明文档中没有涵盖 PSR-7 编程规范。

框架	下载网址	注释
Slim	http://www.slimframework.com/docs/concepts/valueobjects.html	高度遵循 PSR-7 编程规范
Laravel/Lumen	https://lumen.laravel.com/docs/5.2/requests	高度遵循 PSR-7 编程规范

续表

框架	下载网址	注释
Zend Framework 3/ Expressive	https://framework.zend.com/blog/2016-06-28-zend-framework-3.html 和 https://zendframework.github.io/zendexpressive/	高度遵循 PSR-7 编程规范，Zend-Diactoros 和 Zend-Straigility 框架也遵循了 PSR-7 编程规范
Zend Framework 2	https://github.com/zendframework/zendpsr7bridge	可使用 PSR-7 Bridge 组件
Symfony	http://symfony.com/doc/current/cookbook/psr7.html	可使用 PSR-7 Bridge 组件
Joomla	https://www.joomla.org	提供有限的 PSR-7 支持
Cake PHP	http://mark-story.com/posts/view/psr7-bridge-for-cakephp	计划提供 PSR-7 支持，而且将使用 PSR-7 Bridge 组件

- PSR-7 中间件类的数量很多，下面列出了一些比较流行的：

中间件	下载网址	注释
Guzzle	https://github.com/guzzle/psr7	HTTP 消息库
Relay	http://relayphp.com/	调度器
Radar	https://github.com/radarphp/Radar.Project	行为/域/响应器基础结构
NegotiationMiddleware	https://github.com/rszrama/negotiation-middleware	内容协商
psr7-csrf-middleware	https://packagist.org/packages/schnittstabil/psr7-csrf-middleware	预防跨网站伪造请求攻击
oauth2-server	http://alexbilbie.com/2016/04/league-oauth2-server-version-5-is-out	支持 PSR-7 编程规范的 OAuth2 服务器
zend-diactoros	https://zendframework.github.io/zend-diactoros/	实现符合 PSR-7 编程规范的 HTTP 消息