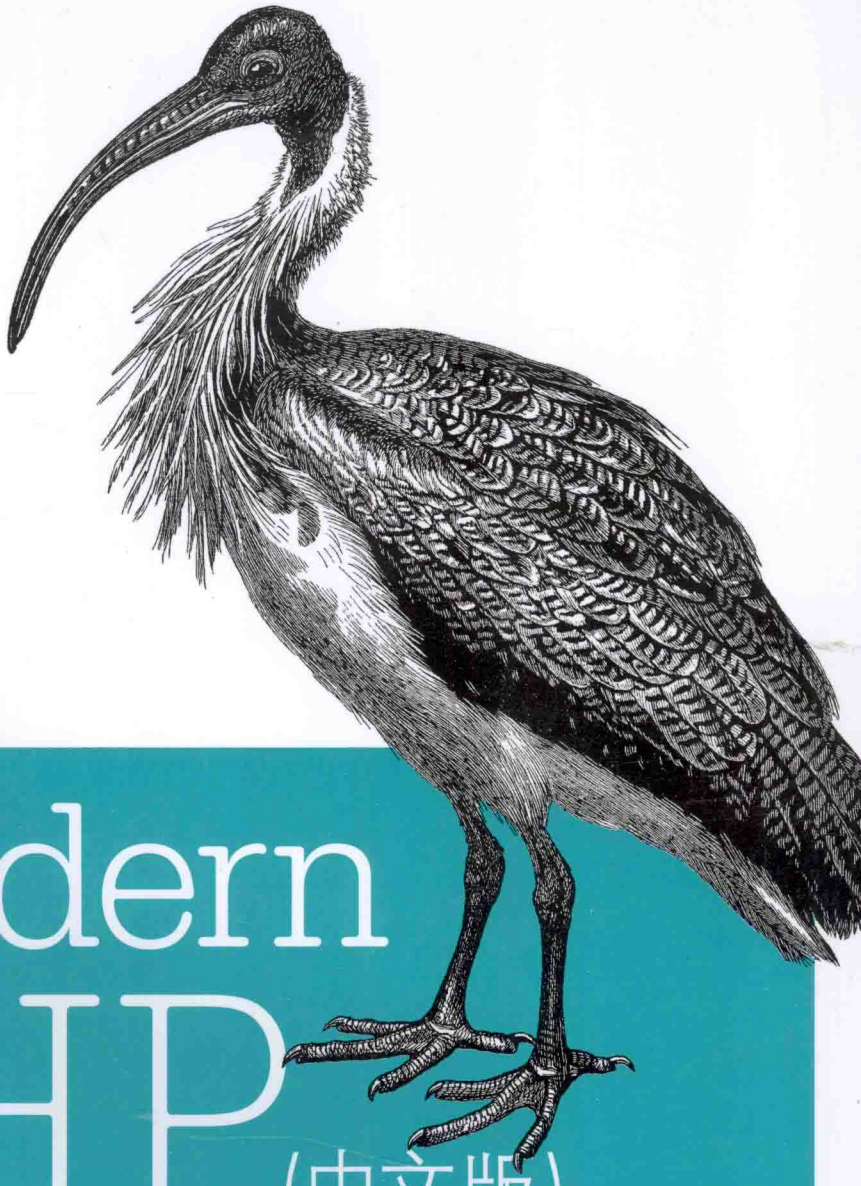
O'REILLY®



Modern PHP（中文版）

Modern PHP

Josh Lockhart 著
安道 译

中国电力出版社

Modern PHP（中文版）

Jash Lockhart 著

安道 译

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

Modern PHP/（美）洛克哈特（Lockhart, J.）著；安道译. —北京：中国电力出版社，2015.9

书名原文：Modern PHP

ISBN 978-7-5123-8093-6

I. ①M… II. ①洛… ②安… III. ①PHP语言－程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2015）第169015号

北京市版权局著作权合同登记

图字：01-2015-4222号

封面设计/ Ellie Volckhausen，张健

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

地　　址/ 北京市东城区北京站西街19号（邮政编码100005）

经　　销/ 全国新华书店

印　　刷/ 北京丰源印刷厂

开　　本/ 787毫米×980毫米　16开本　14.5印张　272千字

版　　次/ 2015年9月第一版　2015年9月第一次印刷

印　　数/ 0001－3000册

定　　价/ 39.00元（册）

O'Reilly Media, Inc.介绍

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

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

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

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

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

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

——Linux Journal

献给Laurel

目录

第二部分 良好实践

第三部分 部署、测试和调优

前言

网上有成千上万的PHP教程，其中大多数都已经过时了，展示的是陈旧的实践方式。可是，谷歌的搜索结果给出的仍是这些教程。过时的信息对马虎的PHP程序员是危险的，他们在不知不觉中就会创建速度慢，且不安全的PHP应用。2013年我意识到了这个问题，发起了“PHP之道”项目（*http://www.phptherightway.com/*）。这个项目由社区成员共同维护，目的是让PHP程序员能轻易找到社区中权威成员最新编写的高质量文档。

本书的目标和“PHP之道”一致。这不是一本参考手册，而是我和你之间亲切友好地谈话，目的是向你介绍现代化的PHP编程语言。我会告诉你我在工作和开源项目中使用的PHP最新技术，让你使用最新的编程标准，以便把你的PHP组件和代码库分享给PHP社区成员使用。

我会不断提到“社区”这个词，PHP社区友好、热情，且乐于助人。不过偶尔也会有负能量。如果你想深入了解书中提到的某个特性，可以到本地的PHP用户组中寻求帮助。我保证附近有PHP开发者愿意帮助你成为一名更好的PHP程序员。本地的PHP用户组是非常宝贵的资源，即使读完这本书很久之后，用户组仍能持续帮助你提升自己的PHP技能。

注意事项

在进入正题之前，我要设定几个前提条件。首先，由于时间仓促，我不可能讲述使用PHP的每种方式。我讲的是我使用PHP的方式。是的，我的方式带有个人色彩，但是很多其他PHP开发者也在使用这些实践方式和标准。你从本书获取的知识能立即应用到自己的项目中。

其次，我假定你熟悉变量、条件判断和循环等概念。你没必要了解PHP，但至少应该能理解这些基本的编程概念。你端杯咖啡就行了（我喜欢喝咖啡），其他东西都由我来提供。

最后，我不限定你必须使用某个操作系统。不过，书中的代码示例是针对Linux编写的，bash命令是为Ubuntu和CentOS提供的，不过应该也能在OS X中执行。如果你使用的是Windows，我强烈建议你在Linux虚拟机中运行书中的示例代码。

本书结构

第一部分说明PHP的新特性，例如命名空间、生成器和性状（trait）。这一部分介绍现代化PHP语言，还会介绍你目前可能还不知道的特性。

第二部分探索应该在PHP应用中使用的良好实践。你是不是听说过PSR，但不完全知道这是什么，也不知道如何使用？你想不想学习如何过滤用户的输入，如何使用安全的数据库查询？这一部分会告诉你答案。

第三部分比前两部分的技术性更强，会说明如何部署、调优、测试和分析PHP应用。我们会学习Capistrano的部署策略，介绍PHPUnit和Travis CI等测试工具，还会探索如何调优PHP，尽量提升应用的性能。

附录A逐步说明如何在你的设备中安装和配置PHP-FPM。

附录B说明如何搭建与生产服务器高度一致的本地开发环境，会介绍Vagrant、Puppet和Chef等工具，帮助你快速上手。

排版约定

本书使用了下述排版约定。

斜体

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

等宽字体（`constant width`）

表示程序代码清单，也表示正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

加粗等宽字体（**`constant width`**）

表示命令或者其他应该由用户输入的内容。

斜体等宽字体（*constant width*）

表示需要使用用户的输入值代替的文本，或者由上下文决定的值。

建议：这个图标表示提示或建议。

注意：这个图标表示一般说明。

警告：这个图标表示警告或提醒。

使用代码示例

本书的补充资料（代码示例和练习等）可以从这个地址下载：*https://github.com/codeguy/modern-php*。

本书的目的是帮助你完成工作。一般来说，你可以在自己的程序或者文档中使用本书附带的示例代码。你无需联系我们获得使用许可，除非你要复制大量的代码。例如，使用本书中的多个代码片段编写程序就无需获得许可。但以CD-ROM的形式销售或者分发O'Reilly书中的示例代码则需要获得许可。回答问题时援引本书内容以及书中示例代码，无需获得许可。在你自己的项目文档中使用本书大量的示例代码时，则需要获得许可。

我们不强制要求署名，但如果你这么做，我们深表感激。署名一般包括书名、作者、出版社和国际标准图书编号。例如：Modern PHP by Josh Lockhart（O'Reilly）. Copyright 2015 Josh Lockhart, 978-1-491-90501-2。

如果你觉得自身情况不在合理使用或上述允许的范围内，请通过邮件和我们联系，地址是*permissions@oreilly.com*。

Safari® Books Online

Safari® Books Online是按需服务的数字图书馆，它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online是技术专家、软件开发人员、Web设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料来源。

Safari Books Online为企业、政府部门、教育机构和个人提供了多种套餐和价格。

订阅者可以在一个完全可搜索的全文数据库中访问上千种图书、培训视频和正式出版之前的书稿。这些内容由以下出版社提供：O'Reilly Media，Prentice Hall Professional，Addison-Wesley Professional，Microsoft Press，Sams，Que，Peachpit Press，Focal Press，Cisco Press，John Wiley & Sons，Syngress，Morgan Kaufmann，IBM Redbooks，Packt，Adobe Press，FT Press，Apress，Manning，New Riders，McGraw-Hill，Jones & Bartlett，Course Technology等。关于Safari Books Online的更多信息，请访问我们的网站。

联系方式

美国：

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

中国：

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

我们为本书提供了网页，该网页上面列出了勘误表、范例和任何其他附加的信息。您可以访问如下网页获得：

http://oreil.ly/HP-Drupal

要询问技术问题或对本书提出建议，请发送电子邮件至：

bookquestions@oreilly.com

要获得更多关于我们的书籍、会议、资源中心和O'Reilly网络的信息，请参见我们的网站：

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

致谢

这是我写的第一本书。O'Reilly公司联系我写这本书时，我既兴奋又害怕得要死。一开始我像沃尔特·休斯顿那样跳起了舞[译注1]，毕竟O'Reilly公司想让我写本书啊，真是太酷了。但是转念一想，我真的能写这么多页吗？写本书可不是件容易的事。

当然，我立即答应了。我知道我能写出这本书，因为我有家人、朋友、同事、编辑和审稿人的一路支持。我要感谢给我无价反馈的支持者们，没有他们就没有这本书。

首先，我要感谢O'Reilly公司的编辑阿莉森·麦克唐纳（@allyatoreilly）。阿莉森为人和善，判断力强，乐于助人，还很聪明。当我迷失时，她知道在什么时候应该使用什么样的方式把我推上正轨。她是我遇到的最好的编辑。

我还要感谢两位技术审稿人，亚当·费尔霍姆（@adamfairholm）和埃德·芬克尔（@funkatron）。亚当是Newfangled（*https://www.newfangled.com*）的天才Web开发者，流行音乐视频数据库IMVDb（*http://imvdb.com/*）是他最为人熟知的项目。埃德的PHP技能异常出色，在/dev/hell播客（*http://devhell.info*）中的主持风格极具个人特色，在Open Sourcing Mental Illness（*http://funkatron.com/osmi*）活动中的表现备受称赞，因此在PHP社区中很有名。亚当和埃德指出了草稿中每一处愚蠢、不合逻辑和错误的地方。没有他们极其真诚的反馈，单靠我自己的话，不可能把这本书写得这么好。我永远都会感激他们对我的指导和他们的智慧。如果在终稿中还有错误或不准确的地方，全都归咎于我。

New Media Campaigns（*http://www.newmediacampaigns.com/*）的同事一直鼓励我。乔尔、克莱、克里斯、亚历克斯、帕特里克、阿什利、伦尼、克莱尔、托德、帕斯卡莱、亨利和南森，我要向你们脱帽致敬，感谢你们自始至终对我温言鼓励。

我最应该感谢的是家人——劳蕾尔、伊桑、特莎、查理、丽莎、格伦和莉兹。感谢你们的鼓励，否则我不可能写完这本书。感谢我可爱的妻子劳蕾尔，感谢你的耐心，感谢你多次陪我一起去卡里布咖啡馆写作，感谢你让我在周末冷落你，感谢你给我动力，让我按计划写作。我永远爱你。

译注1：指沃尔特·休斯顿在电影《碧血金沙》中那段著名的舞蹈。

第一部分

语言特性

第1章

新时代的PHP

PHP正在重生。得益于命名空间、性状、闭包和内置的操作码缓存等有用的特性，PHP正在变成一门现代化脚本语言。而且现在的PHP生态系统也在演进。PHP开发者较少依赖于庞大的框架了，更多的是使用专门的小型组件。依赖管理程序Composer彻底改变了我们构建PHP应用的方式，把我们从框架的封闭环境中解救出来了，让我们可以根据PHP应用的需求混合搭配最合适的PHP互操作组件。如果没有PHP Framework Interop Group提议并监管的社区标准，根本不可能实现组件互操作性。

本书的目的是引导你使用新时代的PHP，会告诉你如何使用社区标准、良好实践和互操作的组件构建并部署出色的PHP应用。

回顾过去

在探索现代的PHP之前，我们要先了解PHP的起源。PHP是一门解释型服务器端脚本语言，也就是说，编写PHP代码后要上传到Web服务器，让解释器执行这些代码。PHP往往在Apache或nginx等Web服务器中运行，用来伺服动态内容。不过，PHP也能用来构建强大的命令行应用（就像bash、Ruby和Python等语言一样）。很多PHP开发者并不知道这一点，因此而错过了一个十分激动人心的特性。当然，我说的不是你。

PHP的官方历史可以在这个网页中查看：*http://php.net/manual/history.php.php*。拉斯姆斯·勒多夫（PHP的创作者）在这篇文章中对PHP的历史说得很清楚，我就不再重复了。我要告诉你的是，PHP的历史很混乱。起初，PHP是拉斯姆斯·勒多夫编写的一系列CGI脚本，用于跟踪他在线简历的访问情况。勒多夫把这些CGI脚本命名为“Personal Home Page Tools”。这个早期阶段和我们现在所熟知的PHP完全不同。勒多夫早期编写

的PHP Tools不是一门脚本语言，只是一些工具，提供基本的变量，并使用嵌入式HTML句法自动处理表单变量。

1994至1998年间，PHP经过多次修改，甚至还有几次彻底重写。两名来自特拉维夫的开发者（安迪·古曼兹和泽埃夫·苏拉斯基）与拉斯姆斯·勒多夫一起，把PHP从一系列简单的CGI工具变成了一门功能完善的编程语言，句法更一致，而且还提供了基本的面向对象编程支持。他们把最终产品命名为PHP 3，并在1998年年底发布了它。这个新名称是从旧名称中演化而来的，它是“PHP: Hypertext Preprocessor”的递归缩写。PHP 3是与现在我们所熟知的PHP最相近的第一个版本，扩展性卓越，支持多种数据库、协议和API。PHP 3的扩展性吸引了很多新开发者。1998年年底，PHP 3在全世界的Web服务器中安装的比率已经达到了惊人的10%。

审视现在

现在，PHP语言发展迅速，由来自全球的几十名核心开发者提供支持，而且开发方式也发生了变化。过去，常见的做法是编写一个PHP文件，使用FTP上传到生产服务器，然后祈祷它能正常运行。这种开发策略非常可怕，但又必须这么做，因为当时没有可用的本地开发环境。

如今，我们都避免使用FTP，转而使用版本控制。版本控制软件（例如Git）能帮助我们维护一个可审查的代码历史，让我们可以创建代码分支、复刻（fork）代码和合并代码。得益于虚拟化工具（例如Vagrant）以及配置工具（例如Ansible、Chef和Puppet），我们能搭建和生产服务器一样的本地开发环境了。我们通过依赖管理工具Composer使用专门的PHP组件。我们的PHP代码遵循PSR，这是由PHP Framework Interop Group管理的社区标准。我们使用PHPUnit等工具彻底测试编写的代码。我们使用PHP的FastCGI进程管理器部署应用，并且放在nginx这样的Web服务器之后。而且还使用操作码缓存来提升应用的性能。

本书包含很多新实践方式，如果你刚接触PHP，或者是从旧版PHP升级过来的，可能不熟悉这些做法。不要觉得不知所措，在本书后面的内容中我会示范每个概念。

我也很高兴地看到，PHP现在有一份正式的规范草案了，2014年之前一直缺少这样一个规范。

注意： 大多数成熟的编程语言都有规范。通俗地说，规范是规范化的蓝图，定义什么是PHP。这个蓝图的作用是供开发者参考，用来开发解析、解释和执行PHP代码的程序，使用PHP创建应用和网站的开发者则不需要使用。

萨拉·高乐曼和Facebook在2014年的O'Reilly OSCON大会上发布了第一份PHP规范草案。PHP的内部邮件列表中有官方公告（*http://news.php.net/php.internals/75886*），这份PHP规范（*https://github.com/php/php-langspec/blob/master/spec/php-spec-draft.md*）可以在GitHub中阅读。

随着多个PHP引擎的出现，制定一份官方的PHP语言规范越发变得重要了。首个PHP引擎是Zend Engine，这个引擎使用C语言编写，并在PHP 4中引入。Zend Engine由拉斯姆斯·勒多夫、安迪·古曼兹和泽埃夫·苏拉斯基开发。如今，Zend Engine是Zend公司对PHP社区主要的贡献。不过，现在出现了第二个PHP引擎——由Facebook开发的HipHop Virtual Machine。语言规范的作用是确保两种引擎具有一致的兼容性。

注意： PHP引擎是解析、解释和执行PHP代码的程序（例如Zend Engine或Facebook开发的HipHop Virtual Machine）。别把这个概念和PHP搞混了，PHP一般指的是PHP语言。

展望未来

Zend Engine正在迅速改进，提供新的功能和提升性能。这些改进得益于新竞争者的出现，尤其是Facebook开发的HipHop Virtual Machine和Hack编程语言。

Hack是一门建立在PHP之上的编程语言，引入了静态类型、新的数据结构和额外的接口，同时还能向后兼容现有的动态类型PHP代码。Hack针对的是欣赏PHP快速开发特点，而又需要静态类型的可预测性和稳定性的开发者。

注意： 本书后面的内容会比较动态类型和静态类型。二者之间的区别在于何时检查PHP类型。动态类型在运行时检查类型，而静态类型在编译时检查类型。更多内容参见第12章。

HipHop Virtual Machine（简称HHVM）是PHP和Hack的解释器，使用即时（Just In Time，JIT）编译器提升应用的性能，并减少内存用量。

我预计Hack和HHVM不会取代Zend Engine，不过Facebook这项新的贡献在PHP社区中引起了巨大的轰动。日益激烈的竞争促使Zend Engine核心团队宣布了PHP 7的开发计划（*https://wiki.php.net/rfc/php7timeline*），声称优化后的Zend Engine和HHVM水平相当。我们会在第12章进一步讨论这些进展。

对PHP程序员来说，现在是令人激动的时刻。PHP社区从未如此充满活力、乐趣和创新精神。我希望这本书能帮助你牢固掌握现代的PHP实践方式。我们要学习的知识很多，而且还有更多的知识在等待我们发掘。照着这个方向前行吧。现在我们进入正题。

第2章

特性

现代的PHP语言有很多令人兴奋的新特性，其中很多特性对从旧版PHP升级过来的程序员来说是全新的。从其他语言转到PHP的程序员，会对这些新特性感到惊讶。这些新特性让PHP语言变成了一个强大的平台，为构建Web应用和命令行工具提供了愉快的体验。

这些特性中有些不是必不可少的，不过能让我们的开发工作更轻松。而有些特性则非常重要，例如命名空间，这是现代PHP标准的基础，能让现代的PHP开发者顺其自然地使用一些开发实践（例如自动加载）。我在本章会介绍每个新特性，说明这些特性为什么有用，还会展示如何在你的项目中使用这些特性。

建议： 我建议你在自己的计算机中跟着我一起动手实践。书中的代码示例在本书的配套GitHub仓库（*https://github.com/codeguy/modern-php*）中。

命名空间

如果只需知道现代的PHP特性中的一个，我想应该是命名空间。命名空间在PHP 5.3.0中引入，是一个很重要的工具，其作用是按照一种虚拟的层次结构组织PHP代码，这种层次结构类似操作系统中文件系统的目录结构。现代的PHP组件和框架都放在各自全局唯一的厂商命名空间中，以免与其他厂商使用的常见类名冲突。

注意： 打个比方，假如你走进一个咖啡厅，看到一个可恶的人在多个桌子上乱七八糟地摆放着书和线缆等东西，而且还坐在唯一可用的插座旁，可是他根本不用，你是不是觉得这个人很可恶？他没有使用命名空间，浪费了对你有用的宝贵空间。不要做这样的人。

下面我们来看真实的PHP组件是如何使用命名空间的。Symfony框架中的symfony/httpfoundation是一个很受欢迎的PHP组件，用于管理HTTP请求和响应。这个组件用到了常见的PHP类名，例如Request、Response和Cookie。我肯定有很多其他PHP组件也使用了相同名称的类。既然其他PHP代码使用了同名的类，那怎么使用symfony/httpfoundation组件呢？其实我们可以放心使用，因为这个组件的代码放在唯一的厂商命名空间Symfony中。打开symfony/httpfoundation组件在GitHub中的仓库（*https://github.com/symfony/HttpFoundation*），找到*Response.php*文件（*http://bit.ly/response-php*），如图2-1所示。

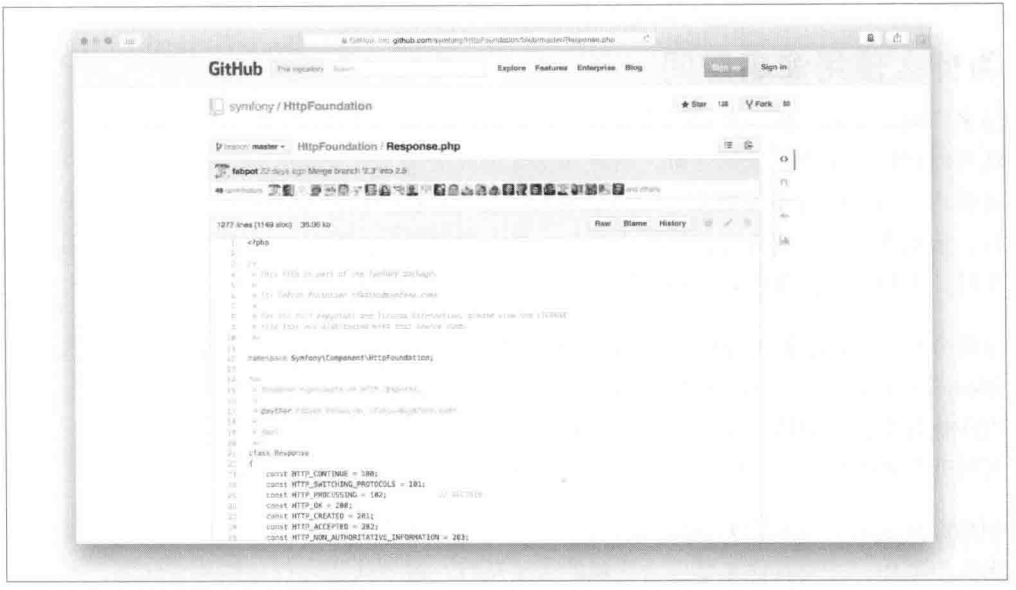


图2-1：symfony/httpfoundation组件的GitHub仓库截图

仔细看第12行，这一行的代码如下：

```
namespace Symfony\Component\HttpFoundation;
```

这一行代码是PHP命名空间声明语句。声明命名空间的代码始终应该放在<?php标签后的第一行。这个命名空间声明语句告诉了我们几件事。首先，我们知道Response类在厂商命名空间Symfony中（厂商命名空间是最顶层命名空间）。我们还知道Response类在子命名空间Component中，而且后面还有一个子命名空间HttpFoundation。你可以看一下和*Response.php*文件在同一层级的其他文件，会发现它们都使用相同的命名空间声明语句。命名空间（或子命名空间）的作用是封装和组织相关的PHP类，就像在文件系统中把相关的文件放在同一个目录中一样。

建议： 子命名空间之间使用一个\符号分隔。

PHP命名空间与操作系统的物理文件系统不同，这是一个虚拟概念，没必要和文件系统中的目录结构完全对应。虽然如此，但是大多数PHP组件为了兼容广泛使用的PSR-4自动加载器标准（在第3章详细介绍），会把子命名空间放到文件系统的子目录中。

注意： 从技术层面来看，命名空间只是PHP语言中的一种记号，PHP解释器会将其作为前缀添加到类、接口、函数和常量的名称前面。

为什么使用命名空间

命名空间很重要，因为代码放在沙盒中，可以和其他开发者编写的代码一起使用。这是现代PHP组件生态系统的基础。组件和框架的作者编写了大量代码，供众多的PHP开发者使用，这些作者不可能知道或控制别人在使用自己的代码时还使用了什么其他类、接口、函数或常量。在你自己的私人项目中也会遇到这种问题，在为项目编写PHP组件或类时，要确保这些代码能和项目的第三方依赖在一起使用。

前面分析symfony/httpfoundation组件时我说过，你的代码可能和其他开发者的代码使用相同的类名、接口名、函数名或常量名，如果不使用命名空间，名称会起冲突，导致PHP执行出错。而使用命名空间，把代码放在唯一的厂商命名空间中的话，你的代码和其他开发者的代码可以使用相同的名称命名类、接口、函数和常量。

如果你开发的是小型个人项目，只有少量的依赖，类名冲突可能不是问题。但是如果在团队中工作，开发有许多第三方依赖的大型项目，就要认真对待命名冲突问题，因为你无法控制项目依赖在全局命名空间中引入的类、接口、函数和常量。这就是为什么一定要在你的代码中使用命名空间的原因。

声明命名空间

每个PHP类、接口、函数和常量都在命名空间（或子命名空间）中。命名空间在PHP文件的顶部，<?php标签之后的第一行声明。命名空间声明语句以namespace开头，随后是一个空格，然后是命名空间的名称，最后以;符号结尾。

注意，命名空间经常用于设定顶层厂商名。下述示例中的命名空间声明语句设定的厂商名是Oreilly：

```
<?php
namespace Oreilly;
```

在这个命名空间声明语句后声明的所有PHP类、接口、函数或常量都在Oreilly命名空间中，而且和O'Reilly Media有某种关系。如果我们想组织本书用到的代码应该怎么做呢？答案是使用子命名空间。

子命名空间的声明方式和前面的示例完全一样。唯一的区别是，我们要使用\符号把命名空间和子命名空间分开。下述示例在最顶层的厂商命名空间Oreilly中声明了一个名为ModernPHP的子命名空间：

```
<?php
namespace Oreilly\ModernPHP;
```

在这个命名空间声明语句后声明的所有类、接口、函数和常量都在Oreilly\ModernPHP子命名空间中，而且和本书有某种关系。

在同一个命名空间或子命名空间中的所有类没必要在同一个PHP文件中声明。你可以在PHP文件的顶部指定一个命名空间或子命名空间，此时，这个文件中的代码就是该命名空间或子命名空间的一部分。因此，我们可以在不同的文件中编写属于同一个命名空间的多个类。

建议： 厂商命名空间是最重要的命名空间。厂商命名空间是最顶层命名空间，用于识别品牌或所属组织，必须具有全局唯一性。子命名空间没那么重要，不过有助于组织项目的代码。

导入和别名

在命名空间出现之前，PHP开发者使用Zend式的类名解决命名冲突问题。这是一种类的命名方案，因Zend框架而流行。这种命名方案在PHP类名中使用下划线表示文件系统的目录分隔符。这种约定有两个作用：其一，确保类名是唯一的；其二，原生的自动加载器会把类名中的下划线替换成文件系统的目录分隔符，从而确定类文件的路径。

例如，Zend_Cloud_DocumentService_Adapter_WindowsAzure_Query类对应的文件是*Zend/Cloud/DocumentService/Adapter/WindowsAzure/Query.php*。可以看出，Zend式命名约定有个缺点——类名特别长。我很懒，根本不想多输入一遍这么长的类名。

现代的PHP命名空间也有类似的问题。例如，symfony\httpfoundation组件中Response类的全名是\Symfony\Component\HttpFoundation\Response。幸好我们可以导入命名空间中的代码，并为其创建别名。

导入的意思是指，在每个PHP文件中告诉PHP想使用哪个命名空间、类、接口、函数和常量。导入后就不用输入全名了。

创建别名，是指告诉PHP我要使用简单的名称引用导入的类、接口、函数或常量。

建议： 从PHP 5.3开始可以导入PHP类、接口和其他命名空间，并为其创建别名。从PHP 5.6开始可以导入PHP函数和常量，并为其创建别名。

示例2-1中的代码创建并发送了一个400 Bad Request HTTP响应，没有导入，也没有创建别名。

示例2-1：使用命名空间，没创建别名

```
<?php
$response = new \Symfony\Component\HttpFoundation\Response('Oops', 400);
$response->send();
```

这不算糟，可是如果要在同一个PHP文件中创建多个Response实例，你的手指很快就会疲惫。现在看一下示例2-2。这个示例使用导入实现了相同的操作。

示例2-2：使用命名空间和默认的别名

```
<?php
use Symfony\Component\HttpFoundation\Response;

$response = new Response('Oops', 400);
$response->send();
```

我们通过use关键字告诉PHP，我们想使用Symfony\Component\HttpFoundation\Response类。我们只需输入一次完全限定的类名，随后实例化Response类时则无需使用完整的类名。这样多棒啊！

如果特别懒，还可以使用别名。我们在示例2-2的基础上做些修改。我可能不想输入Response，只想输入Res。示例2-3展示了如何实现这一需求。

示例2-3：使用命名空间，并自定义别名

```
<?php
use Symfony\Component\HttpFoundation\Response as Res;

$r = new Res('Oops', 400);
$r->send();
```

在这个示例中，我修改了导入Response类的导入语句，在后面加上了as Res。添加的内容告诉PHP，把Res当做Response类的别名。如果不在导入语句中加上as Res，PHP假定别名和导入的类名一样。

建议：应该在PHP文件的顶部使用use关键字导入代码，而且要放在<?php标签或命名空间声明语句之后。

使用use关键字导入代码时无需在开头加上\符号，因为PHP假定导入的是完全限定的命名空间。

use关键字必须出现在全局作用域中（即不能在类或函数中），因为这个关键字在编译时使用。不过，use关键字可以在命名空间声明语句之后使用，导入其他命名空间中的代码。

从PHP 5.6开始还可以导入函数和常量，不过要调整use关键字的句法。如果想导入函数，要把use改成use func：

```
<?php
use func Namespace\functionName;

functionName();
```

如果想导入常量，要把use改成use constant：

```
<?php
use constant Namespace\CONST_NAME;

echo CONST_NAME;
```

函数和常量的别名与类别名的创建方式一样。

实用技巧

多重导入

如果想在一个PHP文件中导入多个类、接口、函数或常量，要在PHP文件的顶部使用多个use语句。PHP允许使用简短的导入句法，把多个use语句写成一行，如下所示：

```
<?php
use Symfony\Component\HttpFoundation\Request,
    Symfony\Component\HttpFoundation\Response,
    Symfony\Component\HttpFoundation\Cookie;
```

别这么做。这样写容易让人困惑，陷入困境。我建议一行写一个use语句，如下所示：

```
<?php
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Cookie;
```

这样要多输入几个字符，但代码更易于阅读和纠错。

一个文件中使用多个命名空间

PHP允许在一个PHP文件中定义多个命名空间，如下所示：

```
<?php
namespace Foo {
    // 在这声明类、接口、函数和常量
}

namespace Bar {
    // 在这声明类、接口、函数和常量
}
```

这么做容易让人困惑，而且违背了“一个文件定义一个类”的良好实践。如果一个文件只使用一个命名空间，代码会更简单，而且更易于纠错。

全局命名空间

如果引用类、接口、函数或常量时没有使用命名空间，PHP假定引用的类、接口、函数或常量在当前命名空间中。如果这个假定不正确，PHP会尝试解析类、接口、函数或常量。如果需要在命名空间中引用其他命名空间中的类、接口、函数或常量，必须使用完全限定的PHP类名（命名空间+类名）。你可以输入完全限定的PHP类名，也可以使用use关键字把代码导入当前命名空间。

有些代码可能没有命名空间，这些代码在全局命名空间中。PHP原生的Exception类就是如此。在命名空间中引用全局命名空间中的代码时，要在类、接口、函数或常量的名称前加上\符号。例如，示例2-4中的\My\App\Foo::doSomething()方法会导致错误，因为PHP会搜索\My\App\Exception类，而这个类不存在。

示例2-4：在命名空间中使用非限定的类名

```
<?php
namespace My\App;

class Foo
{
    public function doSomething()
    {
        $exception = new Exception();
    }
}
```

此时，我们要在Exception类的名称前加上\前缀，如示例2-5所示。这么做的目的是，告诉PHP别在当前命名空间中查找Exception类，要在全局命名空间中查找。

示例2-5：在命名空间中使用限定的类名

```
<?php
namespace My\App;
```

```
class Foo
{
    public function doSomething()
    {
        throw new \Exception();
    }
}
```

自动加载

命名空间还为PHP Framework Interop Group（PHP-FIG）制定的PSR-4自动加载器标准奠定了坚实基础。大多数现代的PHP组件都使用了这种自动加载器模式，使用依赖管理器Composer，可以自动加载项目的依赖。第4章会讨论Composer和PHP-FIG。现在你只需知道，如果没有命名空间，不可能出现现代的PHP生态系统和基于组件的新兴架构。

使用接口

身为PHP程序员，学会如何使用接口改变了我的生活，极大地提升我的能力，让我可以轻易地把第三方PHP组件集成到自己的应用中。接口不是新特性，但非常重要，我们应该了解，并在日常开发中使用。

那么PHP接口是什么呢？接口是两个PHP对象之间的契约，其目的不是让一个对象依赖另一个对象的身份，而是依赖另一个对象的能力。接口把我们的代码和依赖解耦了，而且允许我们的代码依赖任何实现了预期接口的第三方代码。我们不管第三方代码是如何实现接口的，只关心第三方代码是否实现了指定的接口。下面举个例子。

假设我刚到达佛罗里达州迈阿密市，去参加阳光沙滩PHP开发者大会（Sunshine PHP Developer Conference）。我想观光一下，于是找了一家本地汽车租赁行。他们有现代微型轿车，斯巴鲁力狮，还有布加迪威龙（非常让我惊讶）。我知道我需要一种观光工具，这三种汽车都行，但是每一种车的性能有所区别。现代雅绅特是可以，不过我想要性能更好一点的车。我没有小孩，所以力狮的座位太多了。最终我选择了布加迪。

其实，这三种车我都可以开，因为它们实现了相同的接口：都有方向盘、油门、刹车踏板和转向灯，而且燃料都是汽油。布加迪的性能我可能无法驾驭，但驾驶接口和现代是一样的。因为三种车都实现了相同的接口，因此我才能从中选择一辆最喜欢的（老实说，我可能会选择现代）。

在面向对象的PHP中，道理完全一样。如果我编写的代码要处理指定类的对象（从而要使用特定的方式实现），那么代码的功用就完全限定了，因为始终只能使用那个类的对象。可是，如果我编写的代码处理的是接口，那么代码立即就能知道如何处理实现这一接口的任何对象。我的代码不管接口是如何实现的，只关心是否实现了指定的接口。下

面举个例子。

我虚构了一个名为DocumentStore的PHP类，这个类的作用是从不同的源收集文本：可以从远程URL读取HTML，可以读取流资源，也可以收集终端命令的输出。而且，DocumentStore实例中的每个文档都有唯一的标识符。示例2-6是DocumentStore类的代码。

示例2-6：定义DocumentStore类

```
class DocumentStore
{
    protected $data = [];

    public function addDocument(Documentable $document)
    {
        $key = $document->getId();
        $value = $document->getContent();
        $this->data[$key] = $value;
    }

    public function getDocuments()
    {
        return $this->data;
    }
}
```

既然addDocument()方法的参数只能是Documentable类的实例，这样定义DocumentStore类怎么行呢？观察真仔细。其实，Documentable不是类，是接口，其定义如示例2-7所示。

示例2-7：定义Documentable接口

```
interface Documentable
{
    public function getId();

    public function getContent();
}
```

这个接口的定义表明，实现Documentable接口的任何对象都必须提供一个公开的getId()方法和一个公开的getContent()方法。

可是这么做到底有什么用呢？这么做的用处是，我们可以分开定义获取文档的类，而且能使用十分不同的实现方式。示例2-8展示的是一种实现方式，这种方式使用curl从远程URL获取HTML。

示例2-8：定义HtmlDocument类

```
class HtmlDocument implements Documentable
{
    protected $url;
```

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

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

    public function getContent()
    {
        $ch = curl_init();
        curl_setopt($ch, CURLOPT_URL, $this->url);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
        curl_setopt($ch, CURLOPT_CONNECTTIMEOUT, 3);
        curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 1);
        curl_setopt($ch, CURLOPT_MAXREDIRS, 3);
        $html = curl_exec($ch);
        curl_close($ch);

        return $html;
    }
}
```

示例2-9是另一种实现方式，这种方式能读取流资源。

示例2-9：定义StreamDocument类

```
class StreamDocument implements Documentable
{
    protected $resource;
    protected $buffer;

    public function __construct($resource, $buffer = 4096)
    {
        $this->resource = $resource;
        $this->buffer = $buffer;
    }

    public function getId()
    {
        return 'resource-' . (int)$this->resource;
    }

    public function getContent()
    {
        $streamContent = '';
        rewind($this->resource);
        while (feof($this->resource) === false) {
            $streamContent .= fread($this->resource, $this->buffer);
        }

        return $streamContent;
    }
}
```

示例2-10又是一种实现方式，这种方式能获取终端命令的执行结果。

示例2-10：定义CommandOutputDocument类

```
class CommandOutputDocument implements Documentable
{
    protected $command;

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

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

    public function getContent()
    {
        return shell_exec($this->command);
    }
}
```

示例2-11展示了如何借助这三种收集文档的实现方式使用DocumentStore类。

示例2-11：使用DocumentStore类

```
<?php
$documentStore = new DocumentStore();

// 添加HTML文档
$htmlDoc = new HtmlDocument('https://php.net');
$documentStore->addDocument($htmlDoc);

// 添加流文档
$streamDoc = new StreamDocument(fopen('stream.txt', 'rb'));
$documentStore->addDocument($streamDoc);

// 添加终端命令文档
$cmdDoc = new CommandOutputDocument('cat /etc/hosts');
$documentStore->addDocument($cmdDoc);

print_r($documentStore->getDocuments());
```

这太棒了，因为HtmlDocument、StreamDocument和CommandOutputDocument三个类没任何共同点，只是实现了同一个接口。

总之，使用接口编写的代码更灵活，能委托别人实现细节。使用接口后会有越来越多的人（例如办公室里的同事，开源项目的用户，或者从未谋面的开发者）使用你的代码，因为他们只需知道如何实现接口，就可以无缝地使用你的代码。

性状

我的很多PHP开发者朋友都没弄清性状（trait）。这是PHP 5.4.0引入的新概念，既像类又像接口。性状到底是类还是接口呢？哪个都不是。

性状是类的部分实现（即常量，属性和方法），可以混入一个或多个现有的PHP类中。性状有两个作用：表明类可以做什么（像是接口）；提供模块化实现（像是类）。

注意： 你也许熟悉其他语言中的性状，例如，PHP的性状类似Ruby的组合模块或混入（mixin）。

为什么使用性状

PHP语言使用一种典型的继承模型。在这种模型中，我们先编写一个通用的根类，实现基本的功能，然后扩展这个根类，创建更具体的类，从直接父类继承实现。这叫做继承层次结构，很多编程语言都使用了这个模式。

建议： 回想一下曾经学的生物，或许有助于理解这个模式。还记得学过的生物分类体系吗？大自然分成六界，每一界之下有门，门下有纲，纲下有目，目下有科，科下有属，属下有种。在这种分类体系中，每一个层次结构都表示进一步具体化。

大多数时候，这种典型的继承模型能良好运作。可是，如果想让两个无关的PHP类具有类似的行为，应该怎么做呢？例如，`RetailStore`和`Car`两个PHP类的作用十分不同，而且在继承层次结构中没有共同的父类。不过，这两个类都应该能使用地理编码技术转换成经纬度，然后在地图上显示。

性状就是为了解决这种问题而诞生的。性状能把模块化的实现方式注入多个无关的类中。而且性状还能促进代码重用。

为了解决这个问题，我的第一反应是创建一个父类`Geocodable`（这么做不好），让`RetailStore`和`Car`都继承这个类。这种解决方法不好，因为我们强制让两个无关的类继承同一个祖先，而且很明显，这个祖先不属于各自的继承层次结构。

我的第二反应是创建`Geocodable`接口（这么做更好），定义实现地理编码功能需要哪些方法，然后让`RetailStore`和`Car`两个类都实现这个接口。这种解决方法好，因为每个类都能保有自然的继承层次结构。不过，我们要在两个类中重复实现相同的地理编码功能，这不符合DRY原则。

注意： DRY是Don't Repeat Yourself（不要自我重复）的简称，这是个良好实践，表示不要在多个地方重复编写相同的代码。如果需要修改遵守这个原则编写的代码，只需在一处修改，改动就能体现到其他地方。详情参阅维基百科（*http://bit.ly/no-repeat*）。

我的第三反应是创建Geocodable性状（这么做最好），定义并实现地理编码相关的方法，然后把在RetailStore和Car两个类中混入这个性状。这么做不会搅乱这两个类原本自然的继承层次结构。

如何创建性状

PHP性状的定义方式如下：

```
<?php
trait MyTrait {
    // 这里是性状的实现
}
```

建议： 建议与定义类和接口一样，一个文件只定义一个性状。这是良好实践。

我们回到Geocodable那个例子，通过实例更好地演示如何定义性状。我们希望RetailStore和Car两个类都提供地理编码功能，而且认识到继承和接口都不是最佳方案。我们选择的方案是创建Geocodable性状，返回经纬度，然后在地图中绘制。Geocodable性状的完整定义如示例2-12所示。

示例2-12：定义Geocodable性状

```
<?php
trait Geocodable {
    /** @var string */
    protected $address;

    /** @var \Geocoder\Geocoder */
    protected $geocoder;

    /** @var \Geocoder\Result\Geocoded */
    protected $geocoderResult;

    public function setGeocoder(\Geocoder\GeocoderInterface $geocoder)
    {
        $this->geocoder = $geocoder;
    }

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

    public function getLatitude()
```

```
    {
        if (isset($this->geocoderResult) === false) {
            $this->geocodeAddress();
        }

        return $this->geocoderResult->getLatitude();
    }

    public function getLongitude()
    {
        if (isset($this->geocoderResult) === false) {
            $this->geocodeAddress();
        }

        return $this->geocoderResult->getLongitude();
    }

    protected function geocodeAddress()
    {
        $this->geocoderResult = $this->geocoder->geocode($this->address);

        return true;
    }
}
```

Geocodable性状只需定义实现地理编码功能所需的属性和方法，除此之外什么都不需要。

这个Geocodable性状定义了三个类属性：一个表示地址（字符串）；一个是地理编码器对象（\Geocoder\Geocoder类的实例，这个类来自威廉·杜兰德开发的willdurand/geocoder组件，地址是*http://geocoder-php.org*）；一个是地理编码器处理后得到的结果对象（\Geocoder\Result\Geocoded类的实例）。我们还定义了四个公开方法和一个受保护的方法。setGeocoder()方法用于注入Geocoder对象；setAddress()方法用于设定地址；getLatitude()和getLongitude()方法分别返回纬度和经度；geocodeAddress()方法把地址字符串传给Geocoder实例，获取经地理编码器处理得到的结果。

如何使用性状

PHP性状的使用方法很简单，把use MyTrait;语句加到PHP类的定义体中即可。下面是个示例。显然，实际使用时要把MyTrait替换成相应的PHP性状名。

```
<?php
class MyClass
{
    use MyTrait;

    // 这里是类的实现
}
```

建议： 命名空间和性状都使用use关键字导入，可是导入的位置有所不同。命名空间、类、接口、函数和常量在类的定义体外导入，而性状在类的定义体内导入。这个区别虽然小，但很重要。

我们继续看Geocodable那个例子。Geocodable性状的定义如示例2-12所示。下面我们修改RetailStore类，让它使用Geocodable性状（如示例2-13所示）。为了行文简洁，我没有列出RetailStore类的完整实现。

示例2-13：定义RetailStore类

```
<?php
class RetailStore
{
    use Geocodable;

    // 这里是类的实现
}
```

我们只需做这么多。现在，每个RetailStore实例都能使用Geocodable性状提供的属性和方法了，如示例2-14所示。

示例2-14：使用性状

```
<?php
$geocoderAdapter = new \Geocoder\HttpAdapter\CurlHttpAdapter();
$geocoderProvider = new \Geocoder\Provider\GoogleMapsProvider($geocoderAdapter);
$geocoder = new \Geocoder\Geocoder($geocoderProvider);

$store = new RetailStore();
$store->setAddress('420 9th Avenue, New York, NY 10001 USA');
$store->setGeocoder($geocoder);

$latitude = $store->getLatitude();
$longitude = $store->getLongitude();
echo $latitude, ':', $longitude;
```

警告： PHP解释器在编译时会把性状复制粘贴到类的定义体中，但是不会处理这个操作引入的不兼容问题。如果性状假定类中有特定的属性或方法（在性状中没有定义），要确保相应的类中有对应的属性和方法。

生成器

PHP生成器（generator）是PHP 5.5.0引入的功能，往往没被充分利用，其实这是非常有用的功能。我相信很多PHP开发者都不知道生成器，因为生成器的作用不是很明显。生成器是简单的迭代器，仅此而已。

与标准的PHP迭代器不同，PHP生成器不要求类实现Iterator接口，从而减轻了类的负

担。生成器会根据需求计算并产出要迭代的值。这对应用的性能有重大影响。试想一下，假如标准的PHP迭代器经常在内存中执行迭代操作，这要预先计算出数据集，性能低下；如果要使用特定的方式计算大量数据，对性能的影响更甚。此时我们可以使用生成器，即时计算并产出后续值，不占用宝贵的内存资源。

注意： PHP生成器不能满足所有迭代操作的需求，因为如果不查询，生成器永远不知道下一个要迭代的值是什么，在生成器中无法后退或快进。生成器还是一次性的，无法多次迭代同一个生成器。不过，如果需要，可以重建或克隆生成器。

创建生成器

生成器的创建方式很简单，因为生成器就是PHP函数，只不过要在函数中一次或多次使用yield关键字。与普通的PHP函数不同的是，生成器从不返回值，只产出值。示例2-15是一个简单的生成器。

示例2-15：一个简单的生成器

```
<?php
function myGenerator() {
    yield 'value1';
    yield 'value2';
    yield 'value3';
}
```

特别简单，是吧？调用生成器函数时，PHP会返回一个属于`Generator`类的对象。这个对象可以使用`foreach()`函数迭代。每次迭代，PHP会要求`Generator`实例计算并提供下一个要迭代的值。生成器的优雅体现在，每次产出一个值之后，生成器的内部状态都会停顿；向生成器请求下一个值时，内部状态又会恢复。生成器的内部状态会一直在停顿和恢复之间切换，直到抵达函数定义体的末尾或遇到空的`return;`语句为止。我们可以使用下述代码调用并迭代示例2-15中定义的生成器：

```
<?php
foreach (myGenerator() as $yieldedValue) {
    echo $yieldedValue, PHP_EOL;
}
```

上述代码的输出如下：

```
value1
value2
value3
```

使用生成器

下面我要实现一个简单的函数，用于生成一个范围内的数值，以此说明PHP生成器是如何节省内存的。首先，我们使用错误的方式实现（如示例2-16所示）。

示例2-16：生成一个范围内的数值（错误方式）

```
<?php
function makeRange($length) {
    $dataset = [];
    for ($i = 0; $i < $length; $i++) {
        $dataset[] = $i;
    }

    return $dataset;
}

$customRange = makeRange(1000000);
foreach ($customRange as $i) {
    echo $i, PHP_EOL;
}
```

示例2-16中的代码没有善用内存，因为makeRange()函数要为预先创建的一个由一百万个整数组成的数组分配内存。PHP生成器能实现相同的操作，不过一次只会为一个整数分配内存，如示例2-17所示。

示例2-17：生成一个范围内的数值（正确方式）

```
<?php
function makeRange($length) {
    for ($i = 0; $i < $length; $i++) {
        yield $i;
    }
}

foreach (makeRange(1000000) as $i) {
    echo $i, PHP_EOL;
}
```

这是一个虚构的例子。不过，你可以想象一下使用生成器能计算什么样的数据集。数列（例如斐波纳契数列）就是个很好的例子。我们还可以迭代流资源。假设我们想迭代一个大小为4GB的CSV（Comma-Separated Value的简称，由逗号分隔的值）文件，而虚拟私有服务器（Virtual Private Server，VPS）只允许PHP使用1GB内存，因此不能把整个文件都加载到内存中。示例2-18展示了如何使用生成器完成这种操作。

示例2-18：使用生成器处理CSV文件

```
<?php
function getRows($file) {
    $handle = fopen($file, 'rb');
    if ($handle === false) {
        throw new Exception();
```

```
    }
    while (feof($handle) === false) {
        yield fgetcsv($handle);
    }
    fclose($handle);
}

foreach (getRows('data.csv') as $row) {
    print_r($row);
}
```

上述示例一次只会为CSV文件中的一行分配内存，而不会把整个4GB的CSV文件都读取到内存中。而且上述代码还把迭代的实现方式封装到了一个简洁的包中，因此，如果想快速修改获取数据的方式（例如获取CSV、XML或JSON格式的数据），无需改动应用中迭代数据的代码。

生成器是功能多样性和简洁性之间的折中方案。生成器是只能向前进的迭代器，这意味着不能使用生成器在数据集中执行后退、快进或查找操作，只能让生成器计算并产生下一个值。迭代大型数据集或数列时最适合使用生成器，因为这样占用的系统内存量极少。生成器也能完成迭代器能完成的简单任务，而且使用的代码较少。

生成器没为PHP添加新功能，不用生成器也能做生成器能做的事。不过，生成器大大简化了某些任务，而且使用的内存更少。如果需要更多功能，例如在数据集中执行后退、快进或查找操作，最好自己编写类，实现Iterator接口（*http://php.net/manual/class.iterator.php*），或者使用PHP标准库中某个原生的迭代器（*http://php.net/manual/spl.iterators.php*）。

建议：生成器的更多示例参见安东尼·费拉拉（Twitter用户名是@ircmaxell）写的文章“What Generators Can Do For You”（*http://bit.ly/ircmaxwell*）。

闭包

闭包和匿名函数在PHP 5.3.0中引入，这是我最喜欢的两个PHP特性，使用的也最多。这两个特性听起来很吓人（至少我第一次听说时有这种感觉），其实很容易理解。这两个特性非常有用，每个PHP开发者都应该掌握。

闭包是指在创建时封装周围状态的函数。即便闭包所在的环境不存在了，闭包中封装的状态依然存在。这个概念很难理解，不过一旦掌握，将会对你的生活带来巨大变化。

匿名函数其实就是没有名称的函数。匿名函数可以赋值给变量，还能像其他任何PHP对象那样传递。不过匿名函数仍是函数，因此可以调用，还可以传入参数。匿名函数特别

适合作为函数或方法的回调。

注意： 理论上讲，闭包和匿名函数是不同的概念。不过，PHP将其视作相同的概念。所以，我提到闭包时，指的也是匿名函数；反之亦然。

PHP闭包和匿名函数使用的句法与普通函数相同，不过别被这一点迷惑了，闭包和匿名函数其实是伪装成函数的对象。如果审查PHP闭包或匿名函数，会发现它们是Closure类的实例。闭包和字符串或整数一样，也是一等值类型。

创建闭包

既然我们知道闭包和函数很像，那么你应该不会奇怪，我们可以像示例2-19这样创建PHP闭包。

示例2-19：创建简单的闭包

```
<?php
$closure = function ($name) {
    return sprintf('Hello %s', $name);
};

echo $closure("Josh");
// 输出 --> "Hello Josh"
```

就这么简单。示例2-19创建了一个闭包对象，然后将其赋值给$closure变量。闭包和普通的PHP函数很像：使用的句法相同，也接受参数，而且能返回值。不过，匿名函数没有名称。

建议： 我们之所以能调用$closure变量，是因为这个变量的值是一个闭包，而且闭包对象实现了__invoke()魔术方法。只要变量名后有()，PHP就会查找并调用__invoke()方法。

我通常把PHP闭包当做函数和方法的回调使用。很多PHP函数都会用到回调函数，例如array_map()和preg_replace_callback()。这是使用PHP匿名函数的绝佳时机！记住，闭包和其他值一样，可以作为参数传入其他PHP函数。在示例2-20中，我把一个闭包对象当做回调参数，传给array_map()函数。

示例2-20：在array_map()函数中使用闭包

```
<?php
$numbersPlusOne = array_map(function ($number) {
    return $number + 1;
}, [1,2,3]);
print_r($numbersPlusOne);
// 输出 --> [2,3,4]
```

好吧，这并不惊艳。可是要知道，在闭包出现之前，PHP开发者别无选择，只能单独创建具名函数，然后使用名称引用那个函数。这么做，代码执行得稍微慢一点，而且把回调的实现和使用场所隔离开了。传统的PHP开发者会使用类似下面的代码：

```
<?php
// 实现具名回调
function incrementNumber ($number) {
    return $number + 1;
}

// 使用具名回调
$numbersPlusOne = array_map('incrementNumber', [1,2,3]);
print_r($numbersPlusOne);
```

这样的代码虽然可用，但是没有示例2-20简洁。如果只需使用一次回调，没必要单独定义具名函数`incrementNumber()`。把闭包当成回调使用，写出的代码更简洁、更清晰。

附加状态

前面我演示了如何把匿名函数当成回调使用，下面探讨如何为PHP闭包附加并封装状态。JavaScript开发者可能对PHP的闭包感到奇怪，因为PHP闭包不会像真正的JavaScript闭包那样自动封装应用的状态。在PHP中，必须手动调用闭包对象的`bindTo()`方法或者使用`use`关键字，把状态附加到PHP闭包上。

使用`use`关键字附加闭包状态常见得多，因此我们先看这种方式（如示例2-21所示）。使用`use`关键字把变量附加到闭包上时，附加的变量会记住附加时赋给它的值。

示例2-21：使用use关键字附加闭包的状态

```
<?php
function enclosePerson($name) {
    return function ($doCommand) use ($name) {
        return sprintf('%s, %s', $name, $doCommand);
    };
}

// 把字符串"Clay"封装在闭包中
$clay = enclosePerson('Clay');

// 传入参数，调用闭包
echo $clay('get me sweet tea!');
// 输出 --> "Clay, get me sweet tea!"
```

在示例2-21中，具名函数`enclosePerson()`有个名为`$name`的参数，这个函数返回一个闭包对象，而且这个闭包封装了`$name`参数。即便返回的闭包对象跳出了`enclosePerson()`函数的作用域，它也会记住`$name`参数的值，因为`$name`变量仍在闭包中。

建议： 使用use关键字可以把多个参数传入闭包，此时要像PHP函数或方法的参数一样，使用逗号分隔多个参数。

别忘了，PHP闭包是对象。与任何其他PHP对象类似，每个闭包实例都可以使用$this关键字获取闭包的内部状态。闭包对象的默认状态没什么用，不过有一个__invoke()魔术方法和bindTo()方法。仅此而已。

但是，bindTo()方法为闭包增加了一些有趣的潜力。我们可以使用这个方法把Closure对象的内部状态绑定到其他对象上。bindTo()方法的第二个参数很重要，其作用是指定绑定闭包的那个对象所属的PHP类。因此，闭包可以访问绑定闭包的对象中受保护和私有的成员变量。

你会发现，PHP框架经常使用bindTo()方法把路由URL映射到匿名回调函数上。框架会把匿名函数绑定到应用对象上，这么做可以在这个匿名函数中使用$this关键字引用重要的应用对象，如示例2-22所示。

示例2-22：使用bindTo()方法附加闭包的状态

```
01. <?php
02. class App
03. {
04.     protected $routes = array();
05.     protected $responseStatus = '200 OK';
06      protected $responseContentType = 'text/html';
07.     protected $responseBody = 'Hello world';
08.
09.     public function addRoute($routePath, $routeCallback)
10.     {
11.         $this->routes[$routePath] = $routeCallback->bindTo($this, __CLASS__);
12.     }
13.
14.     public function dispatch($currentPath)
15.     {
16.         foreach ($this->routes as $routePath => $callback) {
17.             if ($routePath === $currentPath) {
18.                 $callback();
19.             }
20.         }
21.
22.         header('HTTP/1.1 ' . $this->responseStatus);
23.         header('Content-type: ' . $this->responseContentType);
24.         header('Content-length: ' . mb_strlen($this->responseBody));
25.         echo $this->responseBody;
26.     }
27. }
```

我们要特别注意addRoute()方法。这个方法的参数分别是一个路由路径（例如/users/josh）和一个路由回调。dispatch()方法的参数是当前HTTP请求的路径，它会调用匹

配的路由回调。第11行是重点所在，我们把路由回调绑定到了当前的App实例上。这么做能在回调函数中处理App实例的状态：

```
<?php
$app = new App();
$app->addRoute('/users/josh', function () {
    $this->responseContentType = 'application/json;charset=utf8';
    $this->responseBody = '{"name": "Josh"}';
});
$app->dispatch('/users/josh');
```

Zend OPcache

字节码缓存不是PHP的新特性，有很多独立的扩展可以实现缓存，例如Alternative PHP Cache（APC）、eAccelerator、ionCube和XCache。但是到目前为止，这些扩展都没有集成到PHP核心中。从PHP 5.5.0开始，PHP内置了字节码缓存功能，名为Zend OPcache。

首先我来说明字节码缓存是什么，以及为什么缓存如此重要。PHP是解释型语言，PHP解释器执行PHP脚本时会解析PHP脚本代码，把PHP代码编译成一系列Zend操作码（*http://bit.ly/zend-opcode*），然后执行字节码。每次请求PHP文件都是这样，会消耗很多资源，如果每次HTTP请求PHP都必须不断解析、编译和执行PHP脚本，消耗的资源更多。如果有一种方式能缓存预先编译好的字节码，减少应用的响应时间，降低系统资源的压力，该有多好啊。你很幸运。

字节码缓存能存储预先编译好的PHP字节码。这意味着，请求PHP脚本时，PHP解释器不用每次都读取、解析和编译PHP代码。PHP解释器会从内存中读取预先编译好的字节码，然后立即执行。这样能节省很多时间，极大地提升应用的性能。

启用Zend OPcache

默认情况下，Zend OPcache没有启用，编译PHP时我们必须明确指定启用Zend OPcache。

注意： 如果你选择使用共享的Web主机，一定要选择提供PHP 5.5.0及以上版本，且启用了Zend OPcache的主机商。

如果自己编译PHP（即使用VPS或专用服务器），执行`./configure`命令时必须包含以下选项：

```
--enable-opcache
```

编译好PHP之后，还必须在*php.ini*文件中指定Zend OPcache扩展的路径，如下所示：

```
zend_extension=/path/to/opcache.so
```

PHP编译成功后会立即显示Zend OPcache扩展的文件路径。如果你像我一样，经常忘记看，可以使用下述命令找到这个PHP扩展所在的目录：

```
php-config --extension-dir
```

警告： 如果你使用流行的分析器 Xdebug（*http://xdebug.org*，由大牛德里克·李桑思开发），在*php.ini*文件中必须先加载Zend OPcache扩展，再加载Xdebug。

更新*php.ini*文件后，重启PHP进程就行了。我们可以创建一个PHP文件，写入以下内容，确认Zend OPcache是否正常运行着：

```
<?php
phpinfo();
```

在浏览器中查看这个PHP文件，向下拉滚动条，直到看到Zend OPcache扩展那部分为止，如图2-2所示。如果看不到这部分，说明Zend OPcache没运行。

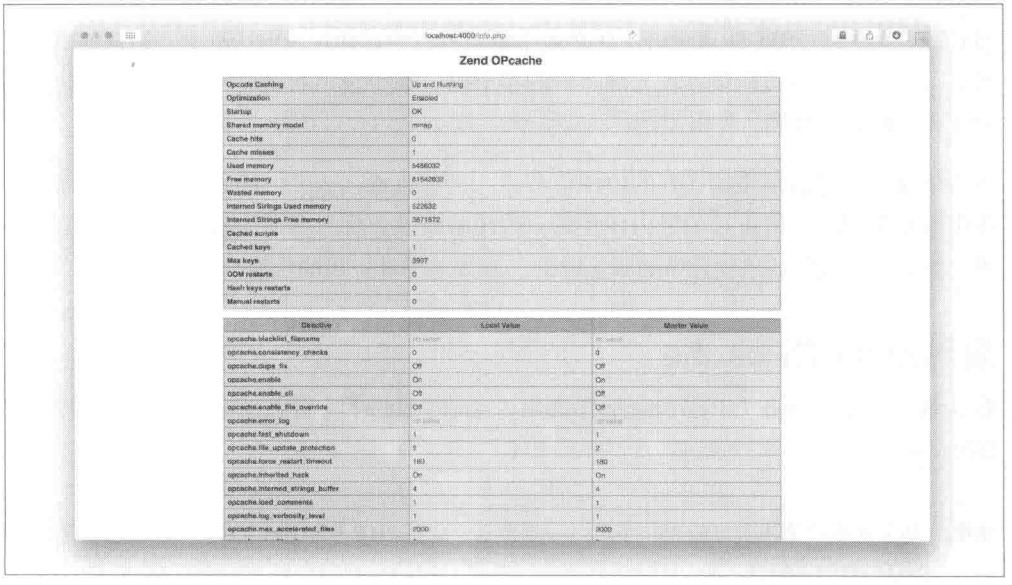


图2-2：Zend OPcache的初始化设置

配置Zend OPcache

启用Zend OPcache后，我们要在*php.ini*文件中配置Zend OPcache的设置。下面是我喜欢使用的OPcache设置：

```
opcache.validate_timestamps = 1 //在生产环境中设为"0"
opcache.revalidate_freq = 0
opcache.memory_consumption = 64
opcache.interned_strings_buffer = 16
opcache.max_accelerated_files = 4000
opcache.fast_shutdown = 1
```

建议： 第8章会进一步介绍Zend OPcache的设置。PHP.net网站中（*http://bit.ly/php-config*）列出了Zend OPcache的全部设置。

使用Zend OPcache

Zend OPcache使用起来很简单，因为启用之后它会自动运行。Zend OPcache会自动在内存中缓存预先编译好的PHP字节码，如果缓存了某个文件的字节码，就执行对应的字节码。

如果*php.ini*文件中`opcache.validate_timestamps`指令的值为0，我们要小心。如果这个设置的值为0，Zend OPcache就察觉不到PHP脚本的变化，我们必须手动清空Zend OPcache缓存的字节码，让它发现PHP文件的变动。这个设置适合在生产环境中设为0，但这么做在开发环境中会带来不便。我们可以在*php.ini*文件中像下面这样配置，启用自动重新验证缓存功能：

```
opcache.validate_timestamps=1
opcache.revalidate_freq=0
```

内置的HTTP服务器

知道吗，从PHP 5.4.0起，PHP内置了Web服务器？这对认为需要Apache或nginx才能预览PHP应用的PHP开发者来说又是一个隐藏的功能。这个内置的Web服务器不应该在生产环境中使用，但对本地开发来说是个极好的工具。

不管写不写PHP代码，我每天都会使用PHP内置的这个Web服务器。我使用这个服务器预览使用Laravel（*http://laravel.com*）和Slim框架（*http://slimframework.com*）开发的应用，使用内容管理框架Drupal架设的网站，以及单纯使用标记语言编写的静态HTML和CSS。

建议： 记住，PHP内置的是一个Web服务器。这个服务器知道怎么处理HTTP协议，能伺服静态资源和PHP文件。使用它，我们无需安装MAMP、WAMP或大型Web服务器，就能在本地编写并预览HTML。

启动这个服务器

这个PHP Web服务器很容易启动。打开终端应用，进入项目文档的根目录，然后执行下述命令即可：

```
php -S localhost:4000
```

上述命令会新启动一个PHP Web服务器，地址是*localhost*。这个服务器监听的端口是4000。当前工作目录是这个Web服务器的文档根目录。

现在，我们可以打开Web浏览器，访问*http://localhost:4000*，这样就能预览应用了。在Web浏览器中浏览应用时，每个HTTP请求的信息都会记录到终端应用的标准输出中，因此，我们可以查看应用是否抛出了400或500响应。

有时，我们需要在同一局域网中的另一台设备中访问这个PHP Web服务器（例如，在iPad或本地Windows虚拟机中预览应用）。为此，我们可以把`localhost`换成`0.0.0.0`，让PHP Web服务器监听所有接口：

```
php -S 0.0.0.0:4000
```

想停止PHP Web服务器时，可以关闭终端应用，也可以按Ctrl+C键。

配置这个服务器

应用经常需要使用专属的PHP初始化配置文件，尤其是对内存用量、文件上传、分析或字节码缓存有特殊要求时，一定要单独配置。我们可以使用`-c`选项，让PHP内置的服务器使用指定的初始化文件：

```
php -S localhost:8000 -c app/config/php.ini
```

建议： 最好把自定义的初始化文件放在应用的根目录中。如果需要和团队中的其他开发者分享，还可以把初始化文件纳入版本控制系统。

路由器脚本

PHP内置的服务器明显遗漏了一个功能：与Apache和nginx不同，它不支持*.htaccess*文

件。因此，这个服务器很难使用多数流行的PHP框架中常见的前端控制器。

注意： 前端控制器是一个PHP文件，用于转发所有HTTP请求（通过*.htaccess*文件或重写规则实现）。前端控制器文件的职责是，分发请求，以及调度适当的PHP代码。Symfony和其他流行的框架都使用了这种模式。

PHP内置的服务器使用路由器脚本弥补了这个遗漏的功能。处理每个HTTP请求前，会先执行这个路由器脚本，如果结果为false，返回当前HTTP请求中引用的静态资源URI。否则，把路由器脚本的执行结果当做HTTP响应主体返回。换句话说，路由器脚本的作用其实和*.htaccess*文件一样。

路由器脚本的用法很简单，只需在启动PHP内置的服务器时指定这个PHP脚本文件的路径：

```
php -S localhost:8000 router.php
```

查明使用的是否为内置的服务器

有时需要知道PHP脚本是使用PHP内置的Web服务器伺服的还是使用传统的Web服务器（例如Apache或nginx）伺服的。之所以想知道这一点，或许是因为想为nginx设定某个首部（例如`Status:`），而不想为PHP内置的Web服务器设置。我们可以使用`php_sapi_name()`函数查明使用的是哪个PHP Web服务器。如果当前脚本由PHP内置的服务器伺服，这个函数会返回字符串`cli-server`：

```
<?php
if (php_sapi_name() === 'cli-server') {
    // PHP 内置的 Web 服务器
} else {
    // 其他 Web 服务器
}
```

缺点

PHP内置的Web服务器不能在生产环境使用，只能在本地开发环境中使用。如果在生产设备中使用PHP内置的Web服务器，会让很多用户失望，还会收到Pingdom（*https://www.pingdom.com*）发出的大量下线通知。

- 内置的服务器性能不是最好的，因为一次只能处理一个请求，其他请求会受到阻塞。如果某个PHP文件必须等待慢速的数据库查询得到结果或远程API返回响应，Web应用会处于停顿状态。

- 内置的服务器只支持少量的媒体类型（*http://bit.ly/built-in-ws*）。
- 内置的服务器通过路由器脚本支持少量的URL重写规则。如果需要更高级的URL重写规则，要使用Apache或nginx。

接下来

现代的PHP语言有很多强大的特性，能改进应用的开发方式。本章讨论了我最喜欢的特性，PHP的网站（*http://php.net/manual/features.php*）对最新的特性有更全面的介绍。

我相信你已经迫不及待地想在自己的应用中使用这些有趣的特性了。不过，我们要遵守PHP的社区标准，正确使用这些特性。下一章会讨论社区标准。

第二部分

良好实践

第3章

标准

PHP组件和框架的数量很多，多得让人难以置信。有Symfony（*http://symfony.com*）和Laravel（*http://laravel.com*）这样的大型框架，也有Silex（*http://silex.sensiolabs.org*）和Slim（*http://slimframework.com*）这样的微型框架，还有在现代的PHP组件出现之前开发的过时框架，例如CodeIgniter（*http://www.codeigniter.com*）。现代的PHP生态系统是个名副其实的大熔炉，有各种各样的代码，帮助我们开发者构建强大的应用。

但是，旧的PHP框架是单独开发的，不能与其他PHP框架共享代码。如果你的项目使用某个旧的PHP框架，你就束缚在这个框架的生态系统中了。如果你喜欢这种框架提供的工具，以它为中心做开发没什么问题。但是，如果使用CodeIgniter框架时想使用Symfony框架中的辅助库，此时应该怎么办呢？你或许只能为项目专门编写一个一次性的适配器。

> 我们完全无法沟通。
>
> ——电影《铁窗喋血》

看出问题了吗？单独开发的框架没有考虑到和其他框架的通信。这么做效率非常低，对开发者（选择的框架限制了创造力）和框架本身（重新编写了其他地方存在的代码）都是如此。不过，我有个好消息。PHP社区已经从中心化的框架模型进化为分布式生态系统了。分布式生态系统中的组件效率高、互操作性好，而且作用专一。

打破旧局面的PHP-FIG

多位PHP框架的开发者认识到了这个问题，在2009年的phpltek（*http://tek.phparch.com*，一个受欢迎的PHP会议）上谈论了这个问题。他们讨论了如何改进框架内的通信，如

何提升效率。例如，如果PHP框架不重新编写紧密耦合的日志类，如何共用**monolog**（*https://github.com/Seldaek/monolog*）这种解耦的日志类？如果PHP框架不各自编写处理HTTP请求和响应的类，如何择优选择Symfony框架中优秀的`symfony/httpfoundation`组件（*http://bit.ly/symf-docs*），用它处理HTTP请求和响应？为了实现这种设想，PHP框架必须使用一种通用方式，让框架之间能通信，能共享代码。也就是说，我们需要一个标准。

这几位在*phpItek*意外碰头的PHP框架开发者后来组建了PHP Framework Interop Group（简称PHP-FIG，*http://www.php-fig.org*）。PHP-FIG由一些PHP框架代表组成，按照PHP-FIG网站中的说法，这些人聚在一起“讨论项目之间的共性，寻找可以合作的方式。”PHP-FIG制定了推荐规范，PHP框架可以自愿实现这些规范，改进与其他框架的通信和共享功能。

PHP-FIG是框架代表自发组织的，其成员不是选举产生的，每位成员的目标都一致，那就是改进PHP社区。任何人都可以申请加入PHP-FIG，而且任何人都能对处于提议阶段的推荐规范向PHP-FIG提交反馈。很多最受欢迎的大型PHP框架通常都会实现定案的PHP-FIG推荐规范。我强烈建议你参与到PHP-FIG中，你可以只提供反馈，帮你最喜欢的PHP框架塑造未来。

注意： 你一定要知道，PHP-FIG发布的是推荐规范，而不是强制规定。推荐规范是谨慎制定的建议，可以让身为PHP开发者（以及PHP框架作者）的我们生活得更轻松。

框架的互操作性

PHP-FIG的使命是实现框架的互操作性。框架的互操作性是指，通过接口、自动加载机制和标准的风格，让框架相互合作。

接口

PHP框架之间通过共用的接口合作。框架通过PHP接口假定第三方依赖提供了什么方法，而不关心依赖是如何实现接口的。

注意： PHP接口的详细说明参见第2章。

例如，假如第三方日志记录器对象实现了`emergency()`、`alert()`、`critical()`、`error()`、`warning()`、`notice()`、`info()`和`debug()`方法，那么框架就可以放心使用这个记录器对象。这些方法是如何实现的无关紧要，框架只关心第三方依赖是否实现了这

些方法。

PHP开发者使用接口可以开发、共享并使用专门的组件，而无需使用庞大的框架。

自动加载

PHP框架之间通过自动加载机制合作。自动加载是指，PHP解释器在运行时按需自动找到并加载PHP类的过程。

在这些PHP标准出现之前，PHP组件和框架会使用魔术方法`__autoload()`或最新的`spl_autoload_register()`方法实现各自特有的自动加载器。因此，我们要学习使用每个组件和框架各自特有的自动加载器。而如今，多数现代的PHP组件和框架都符合同一个自动加载器标准。这意味着，我们只需使用一个自动加载器就能混合搭配多个PHP组件。

风格

PHP框架之间通过标准的代码风格合作。代码风格是指如何使用空格、大小写和括号的位置（等等）。如果PHP框架都使用标准的代码风格，那么每次使用新PHP框架时，PHP开发者对框架所用的风格已经熟悉，就不用学习新风格了。标准的代码风格还能降低项目新贡献者的门槛，让新贡献者把更多的时间用在解决缺陷上，而不用花太多时间学习不熟悉的风格。

标准的代码风格对我们自己的项目也有好处。每个开发者都有一些独特的风格，如果多位开发者在同一个代码基中工作，就会显露问题。使用标准的代码风格，不管作者是谁，团队中的所有成员都能立即理解代码基。

PSR是什么？

PSR是PHP Standards Recommendation（PHP 推荐标准）的简称。如果最近读过关于PHP的博客，你或许见过PSR-1、PSR-2和PSR-3等术语。这些都是PHP-FIG制定的推荐规范。这些规范的名称以PSR-开头，后面跟着一个数字。PHP-FIG制定的每个推荐规范用于解决大多数PHP框架经常会遇到的某个具体问题。PHP框架无需频繁解决相同的问题，它们可以遵守PHP-FIG制定的推荐规范，使用共用的方案来解决。

截至本书出版时，PHP-FIG发布了五个推荐规范：

- PSR-1：基本的代码风格（*http://www.php-fig.org/psr/psr-1/*）。
- PSR-2：严格的代码风格（*http://www.php-fig.org/psr/psr-2/*）。
- PSR-3：日志记录器接口（*http://www.php-fig.org/psr/psr-3/*）。

- PSR-4：自动加载（*http://www.php-fig.org/psr/psr-4/*）。

注意： 如果你数了，发现只有四个推荐规范，你数得对。PHP-FIG废弃了第一份推荐规范——PSR-0（*http://www.php-fig.org/psr/psr-0/*）。第一份推荐规范被新发布的PSR-4（*http://www.php-fig.org/psr/psr-4/*）替代了。

注意，这些PHP-FIG推荐规范与我前面提到的三种实现互操作性的方法（接口、自动加载和代码风格）是一一对应的，这可不是巧合。

PHP-FIG发布的这些推荐规范让我特别高兴，它们是现代PHP生态系统的牢固基石，定义了PHP组件和框架实现互操作性的方式。我承认，PHP标准不是最吸引人的话题，但（我觉得）这些标准是理解现代PHP的前提。

PSR-1：基本的代码风格

如果想编写符合社区标准的PHP代码，首先要遵守PSR-1。这是最容易遵守的PHP标准，简单到你可能已经使用了。PSR-1提出了简单的指导方针，这些方针易于实现，只需要极少的工作量。PSR-1的目的是为遵守这一标准的PHP框架提供代码风格基准。符合PSR-1的代码必须满足以下要求：

PHP标签

必须把PHP代码放在`<?php ?>`或`<?= ?>`标签中。不得使用其他PHP标签句法。

编码

所有PHP文件都必须使用UTF-8字符集编码，而且不能有字节顺序标记（Byte Order Mark，BOM）。这个要求听起来很复杂，其实文本编辑器或IDE都能自动做到这一点。

目的

一个PHP文件可以定义符号（类、性状、函数和常量等），或者执行有副作用的操作（例如，生成结果或处理数据），但不能同时做这两件事。这是一个简单的要求，我们只需稍微深谋远虑一些。

自动加载

PHP命名空间和类必须遵守PSR-4自动加载器标准。我们只需为PHP符号选择合适的名称，并把定义符号的文件放在预期的位置。稍后我们会讨论PSR-4。

类的名称

PHP类的名称必须一直使用驼峰式（CamelCase）。这种格式也叫标题式（TitleCase）。例如`CoffeeGrinder`、`CoffeeBean`和`PourOver`。

常量的名称

PHP常量的名称必须全部使用大写字母。如果需要，可以使用下划线把单词分开。例如`WOOT`、`LET_OUR_POWERS_COMBINE`和`GREAT_SCOTT`。

方法的名称

PHP方法的名称必须一直使用`camelCase`这种驼峰式。也就是说，方法名的首字母是小写的，后续单词的首字母都是大写的。例如`phpIsAwesome`、`iLoveBacon`和`tennantIsMyFavoriteDoctor`。

PSR-2：严格的代码风格

贯彻PSR-1之后，下一步要实施PSR-2。PSR-2使用更严格的指导方针进一步定义PHP的代码风格。

PSR-2制定的代码风格对有多个来自世界各地的贡献者的PHP框架来说，可谓是及时雨。因为每个贡献者都有自己独特的风格和偏好，严格的通用代码风格能让开发者轻易地编写代码，而且其他贡献者能快速理解代码的作用。

与PSR-1不同，PSR-2推荐规范中的指导方针更严格。你可能不喜欢PSR-2中的某些指导方针，可这是很多PHP流行框架选择使用的代码风格。你没必要非得遵守PSR-2，但是如果遵守的话，其他开发者能轻易地理解你的PHP代码，会有更多的人使用你的代码，为代码做贡献。

建议：你应该使用更为严格的PSR-2代码风格。虽然名称中“严格”两字，其实代码写起来十分容易。写得多了，最终会变成第二天性。而且，有工具能把现有的PHP代码自动格式化成符合 PSR-2 风格的代码。

贯彻PSR-1

使用PSR-2代码风格之前先要贯彻PSR-1代码风格。

缩进

这个话题很热门，通常分为两个阵营：第一个阵营选择使用一个制表符缩进，第二个阵营选择使用多个空格缩进（这样酷多了）。PSR-2推荐规范要求PHP代码使用四个空格缩进。

建议：以我个人的经验来看，缩进更适合使用空格，因为空格最可靠，在不同的代码编辑器中渲染的效果基本一致。而制表符的宽度各异，在不同的代码编辑器中渲染的效果也不同。为了得到最好的外观一致性，请使用四个空格缩进代码。

文件和代码行

PHP文件必须使用 UNIX风格的换行符（LF），最后要有一个空行，而且不能使用PHP关闭标签?>。每行代码不能超过80个字符，至少不能超过120个字符。每行末尾不能有空格。这些要求听起来需要做很多工作，其实不然。大多数代码编辑器都能在指定的宽度处换行，删除行尾的空白，并使用UNIX风格的换行符。这些要求都能自动实现，因此你无需担心。

建议： 一开始我觉得不写PHP关闭标签?>很奇怪。其实，最好不写关闭标签，这样能避免意料之外的输出错误。如果加上关闭标签?>，而且在关闭标签后有空行，那么这个空行会被当成输出，导致出错（例如，设定HTTP首部时）。

关键字

我认识很多PHP开发者，他们会使用全部大写的TRUE、FALSE和NULL。如果你也是这么写的，从现在开始摒弃这种写法，只使用小写字母形式。PSR-2推荐规范要求，PHP关键字都应该使用小写字母。

命名空间

每个命名空间声明语句后必须跟着一个空行。类似地，使用use关键字导入命名空间或为命名空间创建别名时，在一系列use声明语句后要加一个空行。下面是一个示例：

```
<?php
namespace My\Component;

use Symfony\Components\HttpFoundation\Request;
use Symfony\Components\HttpFoundation\Response;

class App
{
    // 类的定义体
}
```

类

与缩进方式一样，类定义体的括号位置是另一个引起激烈争论的话题。有些人选择把起始括号和类名放在同一行，另一些人则选择在类名之后新起一行写起始括号。PSR-2推荐规范要求，类定义体的起始括号应该在类名之后新起一行写，如下述示例所示。类定义体的结束括号必须在定义体之后新起一行写。如果你已经这么做了，会觉得这没什么。如果类扩展其他类或实现接口，extends和implements关键字必须和类名写在同一行。

```
<?php
namespace My\App;

class Administrator extends User
```

```
{
    // 类的定义体
}
```

方法

方法定义体的括号位置和类定义体的括号位置一样：方法定义体的起始括号要在方法名之后新起一行写；方法定义体的结束括号要在方法定义体之后新起一行写。要特别注意方法的参数：起始圆括号之后没有空格，结束圆括号之前也没有空格。方法的每个参数（除了最后一个）后面有一个逗号和空格。

```
<?php
namespace Animals;

class StrawNeckedIbis
{
    public function flapWings($numberOfTimes = 3, $speed = 'fast')
    {
        // 方法的定义体
    }
}
```

可见性

类中的每个属性和方法都要声明可见性。可见性由public、protected或private指定，其作用是决定在类的内部和外部如何访问属性和方法。传统的PHP开发者可能习惯在类的属性前加上var关键字，在私有方法的名称前加上下划线（_）。别这么做，我们应该使用前面列出的可见性关键字。如果把类属性或方法声明为abstract或final，这两个限定符必须放在可见性关键字之前。如果把属性或方法声明为static，这个限定符必须放在可见性关键字之后。

```
<?php
namespace Animals;

class StrawNeckedIbis
{
    // 指定了可见性的静态属性
    public static $numberOfBirds = 0;

    // 指定了可见性的方法
    public function __construct()
    {
        static::$numberOfBirds++;
    }
}
```

控制结构

我最常在这条指导方针上犯错。所有控制结构关键字后面都要有一个空格。控制结构关键字包括：if、elseif、else、switch、case、while、do while、for、foreach、try和catch。如果控制结构关键字后面有一对圆括号，起始圆括号后面

不能有空格，结束圆括号之前不能有空格。与类和方法的定义体不同，控制结构关键字后面的起始括号应该和控制结构关键字写在同一行。控制结构关键字后面的结束括号必须单独写在一行。下面是一个简单的示例，展示了上述指导方针：

```
<?php
$gorilla = new \Animals\Gorilla;
$ibis = new \Animals\StrawNeckedIbis;

if ($gorilla->isAwake() === true) {
    do {
        $gorilla->beatChest();
    } while ($ibis->isAsleep() === true);

    $ibis->flyAway();
}
```

建议： 我们可以自动实施PSR-1和PSR-2代码风格，很多代码编辑器都能根据PSR-1和PSR-2自动格式化代码。有些工具还能根据PHP标准，帮你审核并格式化代码，例如PHP Code Sniffer（也叫phpcs，*http://bit.ly/phpsniffer*）。这个工具（可以直接在命令行中使用，也可以集成到IDE中使用）能指出你的代码和指定PHP代码标准之间的差异。大多数包管理器（例如PEAR、Homebrew、Aptitude或Yum）都能安装phpcs。

你还可以使用法比安·博滕斯尔开发的PHP-CS-Fixer（*http://cs.sensiolabs.org/*）自动纠正大多数不符合标准的代码。这个工具并不完美，但是使用这个工具只需做少量工作或无需做任何工作，就能让代码更符合PSR规范。

PSR-3：日志记录器接口

PHP-FIG发布的第三个推荐规范与前两个不同，不是一系列指导方针，而是一个接口，规定PHP日志记录器组件可以实现的方法。

注意： 日志记录器是对象，用于把不同重要程度的消息写入指定的输出。记录的消息用于诊断、检查和排除应用中的操作、稳定性和性能方面的问题。例如，在开发过程中把调试信息写入文本文件；捕获网站的流量统计信息，写入数据库；把致命错误的诊断信息通过电子邮件发给网站管理员。最受欢迎的PHP日志记录器组件是由乔迪·波哥阿诺开发的`monolog/monolog`（*https://packagist.org/packages/monolog/monolog*）。

大多数PHP框架都在某种程度上实现了日志功能。在PHP-FIG出现之前，每个框架使用不同的方式实现日志功能，通常会使用专属的实现方式。为了实现互操作性和专业化，也就是现代PHP提倡的重复利用，PHP-FIG制定了PSR-3日志记录器接口。若想使用符合PSR-3规范的日志记录器，框架要做到两件重要的事：日志功能委托给第三方库实现；最终用户能选择使用他们喜欢的日志记录器组件。这么做对所有人都好。

编写PSR-3日志记录器

符合PSR-3推荐规范的PHP日志记录器组件，必须包含一个实现Psr\Log\LoggerInterface接口的PHP类。PSR-3接口复用了RFC 5424系统日志协议（*http://tools.ietf.org/html/rfc5424*），规定要实现九个方法：

```
<?php
namespace Psr\Log;

interface LoggerInterface
{
    public function emergency($message, array $context = array());
    public function alert($message, array $context = array());
    public function critical($message, array $context = array());
    public function error($message, array $context = array());
    public function warning($message, array $context = array());
    public function notice($message, array $context = array());
    public function info($message, array $context = array());
    public function debug($message, array $context = array());
    public function log($level, $message, array $context = array());
}
```

每个方法对应RFC 5424协议的一个日志级别，而且都接受两个参数。第一个参数$message必须是一个字符串，或者是一个有__toString()方法的对象。第二个参数$context是可选的，这是一个数组，提供用于替换第一个参数中占位标记的值。

建议： $context参数用于构造复杂的日志消息。消息文本中可以使用占位符，例如{placeholder_name}。占位符由{、占位符的名称和}组成，不能包含空格。$context参数的值是一个关联数组，键是占位符的名称（没有花括号），对应的值用于替换消息文本中的占位符。

如果想编写符合PSR-3规范的日志记录器，要创建一个实现Psr\Log\LoggerInterface接口的PHP类，而且要提供这个接口中每个方法的具体实现。

使用PSR-3日志记录器

如果你正在编写自己的PSR-3日志记录器，停下来，想想自己是不是在浪费时间。我强烈不建议你自己编写日志记录器。为什么呢？因为已经有一些十分出色的PHP日志记录器组件了。

如果需要符合PSR-3规范的日志记录器，使用monolog/monolog（*https://packagist.org/packages/monolog/monolog*）即可，别浪费时间找其他组件了。Monolog组件完全实现了PSR-3接口，而且便于使用自定义的消息格式化程序和处理程序扩展功能。Monolog的消息处理程序可以把日志消息写入文本文件、系统日志和数据库，能通过电子邮件发送，还能传给HipChat、Slack、网络中的服务器和远程API。只要你能想到的日志处理

方式，Monolog几乎都提供了。如果非常不巧，Monolog没有提供你所需的处理程序，自己编写消息处理程序并将其集成到Monolog中也特别容易。示例3-1展示了如何设置Monolog，把日志消息写入文本文件。你可以看出这个操作是多么简单。

示例3-1：使用Monolog

```
<?php
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

// 准备日志记录器
$log = new Logger('myApp');
$log->pushHandler(new StreamHandler('logs/development.log', Logger::DEBUG));
$log->pushHandler(new StreamHandler('logs/production.log', Logger::WARNING));

// 使用日志记录器
$log->debug('This is a debug message');
$log->warning('This is a warning message');
```

PSR-4：自动加载器

PHP-FIG发布的第四个推荐规范描述了一个标准的自动加载器策略。自动加载器策略是指，在运行时按需查找PHP类、接口或性状，并将其载入PHP解释器。支持PSR-4自动加载器标准的PHP组件和框架，使用同一个自动加载器就能找到相关代码，然后将其载入PHP解释器。这是了不起的功能，把现代PHP生态系统中很多可互操作的组件联系起来了。

为什么自动加载器很重要

在PHP文件的顶部你是不是经常看到类似下面的代码？

```
<?php
include 'path/to/file1.php';
include 'path/to/file2.php';
include 'path/to/file3.php';
```

太常见了，是吧？你可能熟知require()、require_once()、include()和include_once()函数的作用：把外部PHP文件载入当前脚本。如果只需载入几个PHP脚本，使用这些函数能很好地完成工作。可是，如果需要引入一百个PHP脚本，或者需要引入一千个PHP脚本呢？此时，require()函数和include()函数无法胜任，这就是为什么PHP自动加载器很重要的原因。有了自动加载器，我们无需像前面那样手动引入文件，自动加载器策略能找到PHP类、接口或性状，然后在运行时按需将其载入PHP解释器。

在PHP-FIG发布PSR-4推荐规范之前，PHP组件和框架的作者使用__autoload()和spl_autoload_register()函数注册自定义的自动加载器策略。可是，每个PHP组件和框架

都使用独特的自动加载器，而且每个自动加载器使用不同的逻辑查找并加载PHP类、接口和性状。使用这些组件和框架的开发者，在引导PHP应用时必须调用每个组件各自的自动加载器。我一直使用Sensio Labs开发的Twig（*http://twig.sensiolabs.org*）模板组件，这个组件很棒。可是，如果没有PSR-4，我要阅读Twig的文档，弄清如何在应用的引导文件中注册这个组件自定义的自动加载器，如下所示：

```
<?php
require_once '/path/to/lib/Twig/Autoloader.php';
Twig_Autoloader::register();
```

试想一下，如果要研究每个PHP组件的自动加载器，然后在自己的应用中注册，那该有多麻烦啊！PHP-FIG认识到了这个问题，推荐使用PSR-4自动加载器规范，促进组件实现互操作性。得益于PSR-4，我们只需使用一个自动加载器就能自动加载应用中的所有PHP组件。这太棒了！大多数现代的PHP组件和框架都符合PSR-4规范。如果你自己编写并分发组件，确保你的组件也符合PSR-4规范。符合这一规范的组件包括：Symfony、Doctrine、Monolog、Twig、Guzzle、SwiftMailer、PHPUnit和Carbon等。

PSR-4自动加载器策略

与其他PHP自动加载器一样，PSR-4描述的策略用于在运行时查找并加载PHP类、接口和性状。PSR-4推荐规范不要求改变代码的实现方式，只建议如何使用文件系统目录结构和PHP命名空间组织代码。PSR-4自动加载器策略依赖PHP命名空间和文件系统目录结构查找并加载PHP类、接口和性状。

PSR-4的精髓是把命名空间的前缀和文件系统中的目录对应起来。例如，我可以告诉PHP，`\Oreilly\ModernPHP`命名空间中的类、接口和性状在物理文件系统的*src/*目录中，这样PHP就知道，前缀为`\Oreilly\ModernPHP`的命名空间中的类、接口或性状对应于*src/*目录里的目录和文件。例如，`\Oreilly\ModernPHP\Chapter1`命名空间对应于*src/Chapter1*目录，`\Oreilly\ModernPHP\Chapter1\Example`类对应于*src/Chapter1/Example.php*文件。

建议： PSR-4 会把命名空间的前缀和文件系统中的目录对应起来。命名空间的前缀可以是顶层命名空间，也可以是顶层命名空间加上任意一个子命名空间，相当灵活。

还记得我们在第2章讨论的厂商命名空间吗？PSR-4自动加载器策略对开发组件和框架供其他开发者使用的作者来说最有用。PHP组件的代码在唯一的厂商命名空间中，而且组件的作者会指定厂商命名空间对应于文件系统中的哪个目录，这和我前面演示的做法完全一样。我们在第4章会进一步探讨这个概念。

如何编写PSR-4自动加载器（以及为什么不该这么做）

我们知道，符合PSR-4规范的代码有个命名空间前缀对应于文件系统中的基目录；还知道，这个命名空间前缀中的子命名空间对应于这个基目录里的子目录。示例3-2实现了一个自动加载器，这段代码摘自PHP-FIG网站（*http://bit.ly/php-fig*）。这个自动加载器会根据PSR-4自动加载器策略查找并加载类、接口和性状。

示例3-2：PSR-4自动加载器

```
<?php
/**
 * 举例说明如何实现项目专用的自动加载器。
 *
 * 使用 SPL 注册这个自动加载函数后，遇到下述代码时这个函数
 * 会尝试从 /path/to/project/src/Baz/Qux.php 文件中加载
 * \Foo\Bar\Baz\Qux 类：
 *
 *      new \Foo\Bar\Baz\Qux;
 *
 * @param string $class 完全限定的类名。
 * @return void
 */
spl_autoload_register(function ($class) {

    // 这个项目的命名空间前缀
    $prefix = 'Foo\\Bar\\';

    // 这个命名空间前缀对应的基目录
    $base_dir = __DIR__ . '/src/';

    // 参数传入的类使用这个命名空间前缀吗？
    $len = strlen($prefix);
    if (strncmp($prefix, $class, $len) !== 0) {
        // 不使用，交给注册的下一个自动加载器处理
        return;
    }

    // 获取去掉前缀后的类名
    $relative_class = substr($class, $len);

    // 把命名空间前缀替换成基目录，
    // 在去掉前缀的类名中，把命名空间分隔符替换成目录分隔符，
    // 然后在后面加上.php
    $file = $base_dir . str_replace('\\', '/', $relative_class) . '.php';

    // 如果文件存在，将其导入
    if (file_exists($file)) {
        require $file;
    }
});
```

复制上述代码，将其粘贴到你的应用中，然后修改变量`$prefix`和`$base_dir`的值，这样你就有一个可用的PSR-4自动加载器了。不过，如果你发现自己在编写PSR-4自动加载

器，请停下来，然后问自己有必要这么做吗。为什么呢？因为我们可以使用依赖管理器Composer自动生成的PSR-4自动加载器。很巧，接下来在第4章我们就会讨论这个话题。

第4章

组件

现代的PHP较少使用庞大的框架，而是更多地使用具有互操作性的专门组件制定解决方案。开发新PHP应用时，我很少直接使用Laravel或Symfony，而是思考能把哪些现有的PHP组件结合在一起解决我的问题。

为什么使用组件？

对很多PHP程序员来说，现代的PHP组件是个陌生的概念。我也是几年前才知道。在没有深入理解组件之前，本能驱使我使用巨型框架（例如Symfony或CodeIgniter）开发PHP应用，我从不会考虑其他途径。以前我会花时间研究某个框架的封闭生态系统，使用这个框架提供的工具。如果不幸，框架没有提供我所需的功能，我会自己开发。大型框架也很难集成自定义的库或第三方库，因为这些库之间没有使用相同的接口。现在你可以放心，这样的日子一去不复返了，我们不用再感激庞大的框架，也不用束缚在这些框架筑起的围墙中了。

如今，开发应用时，我们会从不断增多的大量专用组件中选择合适的。既然已经有了guzzle/http组件（*https://packagist.org/packages/guzzle/http*），为什么还要浪费时间自己编写处理HTTP请求和响应的库呢？既然aura/router（*https://packagist.org/packages/aura/router*）和orno/route组件（*https://packagist.org/packages/orno/route*）很好用，为什么还要重新创建路由器呢？既然有aws/aws-sdk-php（*https://packagist.org/packages/aws/aws-sdk-php*）和league/flysystem组件（*https://packagist.org/packages/league/flysystem*）可用，为什么还要花时间为亚马逊S3的在线存储服务编写适配器呢？你应该明白我想表达的意思了。其他开发者用了无数时间创建、优化和测试专门的组件，以便

让组件尽量做好一件事。如果想快速开发更好的应用，不使用这些组件而自己重新发明轮子的话，那就太傻了。

组件是什么？

组件是打包的代码，用于帮你解决PHP应用中某个具体的问题。例如，如果你的PHP应用要收发HTTP请求，可以使用现成的组件实现；如果你的应用要解析逗号分隔的数据，可以使用现成的组件实现。我们使用组件为的是不重新实现已经实现的功能，把更多时间用在实现项目的长远目标上。

注意： 严格来说，PHP组件是一系列相关的类、接口和性状，用于解决某个具体问题。组件中的类、接口和性状通常放在同一个命名空间中。

任何市场中的产品都有好坏之分，PHP组件也是如此。就像在杂货店检查苹果一样，区分PHP组件的好坏也有一些技巧。好的PHP组件具有以下特征：

作用单一

PHP组件的作用单一，能很好地解决一个问题。组件不是万能钥匙，不能杂而不精，要术业有专攻。组件专注于解决一个问题，而且使用简单的接口封装功能。

小型

PHP组件小巧玲珑，只包含解决某个问题所需的最少代码。组件中的代码量各异。一个PHP组件可以只有一个PHP类，也可以有多个PHP类，分别放在不同的子命名空间中。PHP组件中类的数量没有统一限制，根据解决问题的需要，想使用多少个就使用多少个。

合作

PHP组件之间能良好合作。毕竟组件就是为了和其他组件合作，解决更复杂的问题。PHP组件不会让自己的代码搅乱全局命名空间，而会把代码放在自己的命名空间中，防止与其他组件有名称冲突。

测试良好

PHP组件测试良好。因为体型小，因此很容易测试。如果PHP组件体型小，而且作用单一，很可能也易于测试，因为组件关注的东西少，而且依赖易于识别和模拟。最好的PHP组件本身会提供测试，而且有充足的测试覆盖度。

文档完善

PHP组件的文档完善。组件应该能让开发者轻易安装、理解和使用。好的文档可以做到这一点。PHP组件应该有个*README*文件，说明组件的作用，如何安装，以及

如何使用。还可以为组件搭建网站，放上更详细的信息。PHP组件的源码也应该有文档，为组件中的类、方法和属性添加行内文档块，说明参数、返回值和可能抛出的异常。

组件和框架对比

框架（尤其是较旧的框架）的问题是需要很多投入。我们选择框架时，要为这个框架的工具投入很多。框架通常会提供大量工具，可是有时却没提供我们需要的某个具体工具，遇到这种情况时，痛苦就转嫁到我们头上了，我们要寻找并集成自定义的PHP库。把第三方代码集成到框架中是件难事，因为第三方代码和框架可能没使用相同的接口。

选择框架时，我们看中的是这个框架的未来。我们相信框架的核心开发团队，认定他们会持续投入时间开发框架，确保框架的代码能跟上现代标准。可是经常事与愿违。框架很大，需要大量时间和精力维护。维护项目的人有自己的生活、工作和兴趣，而生活、工作和兴趣经常会变。

注意： 说句公道话，大型PHP组件也有中止开发的风险。如果只有一位核心开发者，这种风险更高。

况且，谁能保证某个框架始终是完成某项工作最好的工具呢？存在多年的大型项目必须有好的表现，而且要时刻做好调整。如果选错了PHP框架，可能无法做到这一点。较旧的PHP框架可能由于缺乏社区支持而变慢和过时。这些旧框架通常使用过程式代码编写，而没使用新式的面向对象代码。团队中的新成员可能不熟悉较旧框架的代码基。决定是否使用PHP框架时，要考虑的事情有很多。

框架并非一无是处

目前我说的都是框架的缺点，可是框架并非一无是处。Symfony（*http://symfony.com/*）是个极好的现代PHP框架。这个框架由法比安·博滕斯尔和Sensio Labs（*http://sensiolabs.com/*）开发，由很多解耦的小型组件（*http://symfony.com/components*）构成。这些组件可以放在一起组成框架，也可以在应用中单独使用。

其他较旧的框架也在经历类似的变迁，积极地向现代PHP组件靠拢。内容管理框架Drupal（*https://www.drupal.org*）就是个例子。Drupal 7使用过程式PHP代码编写，而且代码都在全局命名空间中。Drupal舍弃现代的PHP实践，是为了支持陈旧的代码基。可是Drupal 8积极向现代的PHP靠拢了，跨越的幅度之大值得赞赏。Drupal 8使用很多不同的PHP组件构建了一个现代化的内容管理平台。

Laravel（*http://laravel.com*）也是一个流行的PHP框架，由泰勒·奥特威尔编写。与Symfony类似，Laravel构建在自身的Illuminate框架库（*https://github.com/illuminate*）之上。可是，（截至本书出版时）Laravel的组件不能轻易解耦，用于Laravel之外的应用中。Laravel没使用PSR-2社区标准，而且也不遵守语义版本方案（*http://semver.org/*）。不过，别因此而犹豫，Laravel仍是一个出色的框架，能创建非常强大的应用。

建议： 最流行的PHP框架有：

- Aura（*http://auraphp.com/framework*）。
- Laravel（*http://laravel.com/*）。
- Symfony（*http://symfony.com/*）。
- Yii（*http://www.yiiframework.com/*）。
- Zend（*http://framework.zend.com/*）。

使用正确的工具做正确的事

我们应该使用组件还是框架呢？答案是，使用正确的工具做正确的事。大多数现代的PHP框架其实只是构建在小型PHP组件之上的一系列约定。

如果是能通过一些PHP组件准确解决问题的小型项目，那就使用组件。组件非常便于查找并使用现有的工具，这样我们无需过多关注样板代码，有更多的时间处理手头上的大型任务。组件还有助于让代码保持轻量级和灵活性。这样我们只使用自己所需的代码，而且特别容易把一个组件换成另一个更适合项目的组件。

如果是有多个团队成员开发的大型项目，而且能从框架提供的约定、准则和结构中受益，那就使用框架。可是，框架会为我们做很多决定，而且要求我们遵守特殊的约定。框架的灵活性较低，不过较之使用一系列PHP组件，框架为我们提供了很多拿来即用的工具。如果不在意这些，框架是不二之选。使用框架能引导并加速项目的开发。

查找组件

我们可以在PHP组件目录Packagist（*https://packagist.org*，见图4-1）中查找现代PHP组件。这个网站用于收集PHP组件，可以使用关键字搜索组件。最好的PHP组件在Packagist中都有。我要感谢乔迪·波哥阿诺（*http://seld.be/*）和伊戈尔·威德勒（*https://igor.io/archive.html*），感谢他们创造了如此宝贵的社区资源。

建议： 经常有人问我，觉得哪些PHP组件最好。这是一个主观问题。不过，我基本上认为Awesome PHP（*https://github.com/ziadoz/awesome-php*）列出的PHP组件是最好的。Awesome PHP列出了很多优秀的PHP组件，这个项目由杰米·约克（*https://github.com/ziadoz*）维护。

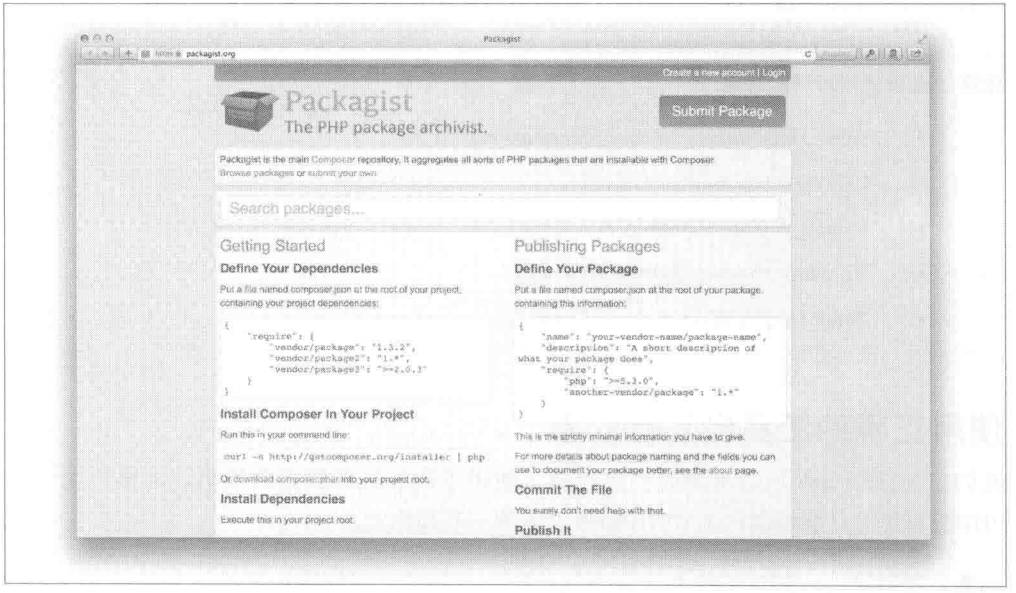


图4-1：Packagist网站

搜索

别浪费时间解决已经解决的问题。需要收发HTTP消息吗？访问Packagist，搜索“`http`”，得到的第一个结果是Guzzle，就用它吧。需要解析CSV文件吗？访问Packagist，搜索“`csv`”，选择并使用某个CSV组件。Packagist可以看成卖PHP组件的杂货店，在这里可以买到最好的材料。Packagist中或许就有能解决你问题的PHP组件。

选择

如果Packagist中有多个符合需求的PHP组件怎么办呢？如何选择最好的那个？Packagist会记录每个PHP组件的统计信息，告诉你每个组件被下载了多少次，以及有多少人喜欢（见图4-2）。下载数和喜欢的人数较多，可能说明那个组件是不错的选择（并非始终如此）。话虽如此，但也别低估下载数较少的新包。每天都会有很多新包添加到Packagist中。

如果搜索某个关键字时返回大量结果，可能难以找到最合适的PHP组件。我们不能始终

靠下载数做决定，因为用的人多并不表明符合自己的需求。随着Packagist的流行，这是必需要解决的问题。我建议你靠口碑和同伴推荐来选择PHP组件。

图4-2：在Packagist中搜索得到的结果

回馈

如果你喜欢某个PHP组件，在Packagist中为这个组件加星号，然后通过Twitter、Facebook、IRC、Slack和其他交流平台将其分享给共事的PHP开发者。这么做能让最好的PHP组件为人熟知，并被其他开发者发现。

使用PHP组件

Packagist是查找PHP组件的地方，Composer（*https://getcomposer.org/*）是安装PHP组件的工具。Composer是PHP组件的依赖管理器，运行在命令行中。你告诉Composer需要哪些PHP组件，Composer会下载并把这些组件自动加载到你的项目中。就这么简单。Composer是依赖管理器，因此还能解析并下载组件的依赖（以及依赖的依赖，如此一直循环下去）。

Composer能和Packagist紧密合作。如果你告诉Composer想使用`guzzlehttp/guzzle`组件，Composer会从Packagist中获取`guzzlehttp/guzzle`组件，找到这个组件的仓库地址，确定要使用哪个版本，还能找出这个组件的依赖，然后把`guzzlehttp/guzzle`组件及其依赖下载到你的项目中。

Composer的作用很重要，因为依赖管理和自动加载是很难处理的问题。自动加载是指，在不使用require()、require_once()、include()或include_once()函数的情况下，按需自动加载PHP类。在较旧的PHP版本中可以使用__autoload()函数自己编写自动加载器；实例化尚未加载的类时，PHP解释器会自动调用这个函数。后来，PHP在SPL库中引入了更灵活的spl_autoload_register()函数。如何自动加载PHP类完全由开发者决定。可是，由于缺少通用的自动加载器标准，各个项目通常会使用独特的方式实现自动加载器。如果每个开发者都使用独特的自动加载器，就很难使用其他开发者创建并分享的代码。

PHP Framework Interop Group认识到了这个问题，因此制定了PSR-0标准（后来被PSR-4标准取代）。PSR-0和PSR-4建议如何使用命名空间和文件系统的目录结构组织代码，让代码符合标准的自动加载器实现方式。我在第3章说过，我们没必要自己编写PSR-4自动加载器，因为依赖管理器Composer会为项目中的所有PHP组件自动生成符合PSR标准的自动加载器。Composer有效抽象了依赖管理和自动加载。

注意： 我认为对PHP社区来说，Composer是最重要的附加工具。Composer改变了我创建PHP应用的方式。在每个PHP项目中我都会使用Composer，因为它大大简化了在应用中集成和使用第三方PHP组件的方式。如果你还没使用Composer，应该从现在开始研究。

如何安装Composer

Composer易于安装。打开终端，然后执行下述命令：

```
curl -sS https://getcomposer.org/installer | php
```

这个命令使用curl下载Composer的安装脚本，然后使用php执行安装脚本，最后在当前工作目录中创建*composer.phar*文件。*composer.phar*文件是Composer的二进制文件。

警告： 千万不要盲目执行从远程URL下载的代码。我们要先查看远程代码，弄清它的作用。而且，一定要通过HTTPS下载远程代码。

我喜欢使用下述命令，重命名下载得到的Composer二进制文件，并将其移到*/usr/local/bin/composer*：

```
sudo mv composer.phar /usr/local/bin/composer
```

记得要执行下述命令，把composer变成可执行的二进制文件：

```
sudo chmod +x /usr/local/bin/composer
```

最后，在*~/.bash_profile*文件中添加下面这行代码，把*/usr/local/bin*目录加入PATH环境变量中：

```
PATH=/usr/local/bin:$PATH
```

现在应该可以在终端应用中执行`composer`命令，查看Composer的选项列表了（见图4-3）。

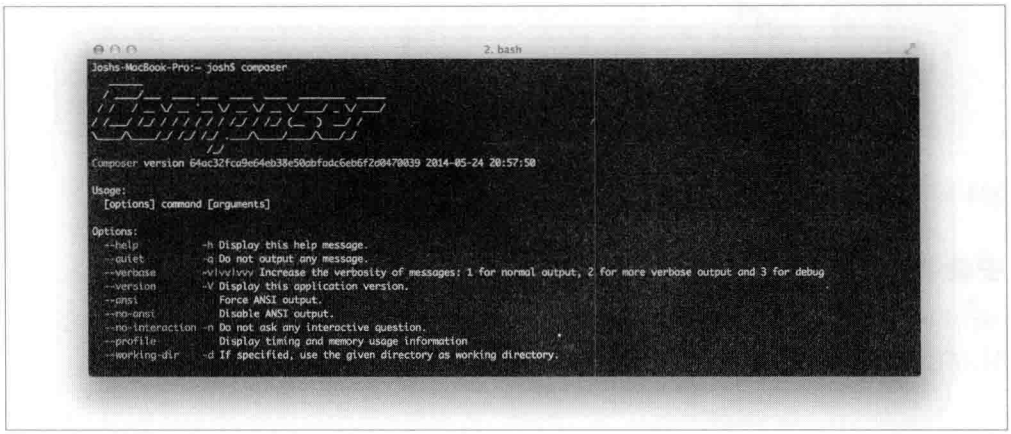


图4-3：Composer的命令行选项

如何使用Composer

现在Composer安装好了，我们来下载一些PHP组件。Composer通常用于下载单个项目所需的PHP组件。

组件的名称

首先，要列出项目所需的组件。此时要特别注意各个组件的厂商名和包名。每个PHP组件的名称都由厂商名和包名组成。例如，对流行的`league/flysystem`组件（*https://packagist.org/packages/league/flysystem*）来说，厂商名是`league`，包名是`flysystem`。厂商名和包名之间由/符号分隔。厂商名和包名在一起组成完整的组件名`league/flysystem`。

厂商名是全局唯一的，这是全局标识符，用于识别名下的包属于谁。包名用于唯一识别指定厂商名下的某个包。Composer和Packagist都使用`vendor/package`这种命名约定，避免不同厂商的PHP组件有名称冲突。Packagist网站中的页面会显示PHP组件的厂商名和包名（见图4-4）。

图4-4：Packagist中显示的厂商名和包名

安装组件

每个PHP组件都可能有多个可用的版本（例如，`1.0.0`、`1.5.0`或`2.15.0`）。Packagist会显示组件的所有可用版本。

建议：语言版本

现代的PHP组件都使用语义版本方案（*http://semver.org/*），版本号由三个点（.）分数字组成（例如，`1.13.2`）。第一个数字是主版本号，如果PHP组件的更新破坏了向后兼容性，会提升主版本号。第二个数字是次版本号，如果PHP组件小幅更新了功能，而且没破坏向后兼容性，会提升次版本号。第三个数字（即最后一个数字）是修订版本号，如果PHP组件修正了向后兼容的缺陷，会提升修订版本号。

幸好，我们自己不用找出每个组件最新稳定版的版本号，Composer会代我们操作。在终端应用中进入项目的最顶层目录，然后为所需的每个PHP组件执行一次下述命令：

```
composer require vendor/package
```

记得要把*vendor/package*换成组件的厂商名和包名。例如，安装Flysystem组件要执行下述命令：

```
composer require league/flysystem
```

这个命令让Composer查找并安装指定PHP组件的最新稳定版。这个命令还能让Composer把组件更新到下一个主版本之前的最新版。上述示例，在2014年10月会安装Flysystem `0.5.9`版。如果之后再执行，会把Flysystem更新到`1.*`之前的最新版。

执行命令的结果可以在项目最顶层目录中新建或更新的*composer.json*文件中查看。执行

这个命令后还会创建一个*composer.lock*文件。这两个文件都要纳入版本控制系统。

示例项目

下面我们开发一个PHP示例应用，以此加强Composer技能。这个应用的作用是扫描一个CSV文件中的URL，找出死链。这个应用会向每个URL发送HTTP请求，如果返回的HTTP状态码大于或等于400，我们就把这个死链发给标准输出。这是一个命令行应用，CSV文件的路径在第一个也是唯一一个命令行参数中指定。开发好之后，我们会执行这个脚本，传入CSV文件的路径，在标准输出中显示死链列表：

```
php scan.php /path/to/urls.csv
```

这个项目的目录结构如图4-5所示。

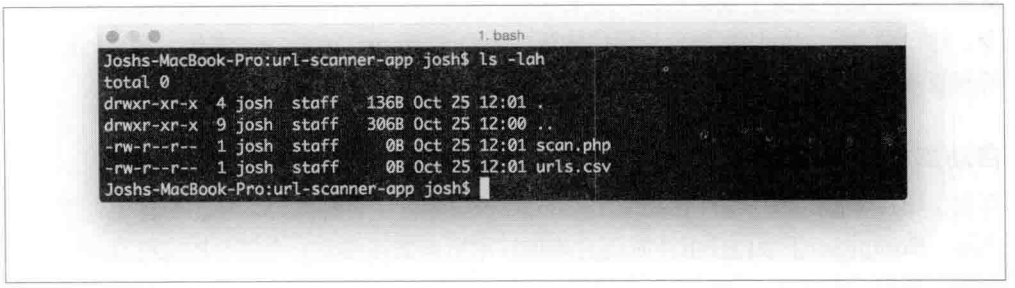


图4-5：组件的目录结构

开始PHP新项目时，首先确定哪些任务可以使用现有的PHP组件解决。*scan.php*脚本会打开并迭代处理一个CSV文件，因此我们需要一个能读取并迭代处理CSV数据的PHP组件。*scan.php*脚本还要向CSV文件中的每个URL发送HTTP请求，因此我们需要一个能发送HTTP请求并检查HTTP响应的PHP组件。我们当然可以自己编写迭代处理CSV文件或发送HTTP请求的代码，可是，既然这些问题已经由他人解决了，我们为什么还要浪费自己的时间呢？记住，我们的目的是扫描一系列URL，而不是开发解析HTTP和CSV的库。

浏览Packagist后，我找到了`guzzlehttp/guzzle`和`league/csv`两个组件。前者用于处理HTTP消息，后者用于解析并迭代处理CSV数据。下面我们在项目的最顶层目录中执行下述命令，使用Composer安装这两个组件：

```
composer require guzzlehttp/guzzle;
composer require league/csv;
```

上述命令让Composer把这两个组件下载到项目最顶层目录中的*vendor/*目录里，而且还会创建*composer.json*文件和*composer.lock*文件。

composer.lock文件

Composer安装项目的依赖后，你会发现Composer创建了一个*composer.lock*文件。这个文件会列出项目使用的所有PHP组件，以及组件的具体版本号（包括主版本号、次版本号和修订版本号）。这其实是锁定了项目，让项目只能使用具体版本的PHP组件。

为什么要这么做呢？如果有*composer.lock*文件，Composer会下载这个文件中列出的具体版本，而不管Packagist中可用的最新版本是多少。你应该把*composer.lock*文件纳入版本控制，把这个文件分发给团队成员，让他们使用的PHP组件版本和你一样。如果团队成员、过渡服务器和生产服务器都使用相同版本的PHP组件，能尽量降低由组件版本差异导致缺陷的风险。

*composer.lock*文件有个缺点，`composer install`命令不会安装比其中列出的版本号新的版本。如果确实需要下载新版组件并更新*composer.lock*文件，要使用`composer update`命令。这个命令会把组件更新到最新稳定版，还会更新*composer.lock*文件，写入PHP组件的新版本号。

自动加载PHP组件

现在，项目所需的PHP组件已经使用Composer安装好了，那么怎么使用组件呢？我们很幸运，Composer下载PHP组件时还会为项目的所有依赖创建一个符合PSR标准的自动加载器。我们只需在*scan.php*文件的顶部使用`require`函数导入Composer创建的自动加载器：

```
<?php
require 'vendor/autoload.php';
```

Composer创建的自动加载器其实就是个名为*autoload.php*的PHP文件，保存在*vendor/*目录中。Composer下载各个PHP组件时会检查每个组件的*composer.json*文件，确定如何加载该组件。得到这个信息后，Composer会在本地为该组件创建一个符合PSR标准的自动加载器。这样，我们就可以实例化项目中的任何PHP组件，这些组件会按需自动加载。特别方便，是吧？

编写scan.php

下面我们使用Guzzle和CSV组件编写*scan.php*脚本。记住，执行这个PHP脚本时，CSV文件的路径由第一个命令行参数（通过`$argv`数组获取）提供。*scan.php*脚本的内容如示例4-1所示。

示例4-1：扫描URL的应用

```
<?php
// 1. 使用Composer自动加载器
```

```
require 'vendor/autoload.php';

// 2. 实例 Guzzle HTTP客户端
$client = new \GuzzleHttp\Client();

// 3. 打开并迭代处理CSV
$csv = new \League\Csv\Reader($argv[1]);
foreach ($csv as $csvRow) {
    try {
        // 4. 发送HTTP OPTIONS请求
        $httpResponse = $client->options($csvRow[0]);

        // 5. 检查HTTP响应的状态码
        if ($httpResponse->getStatusCode() >= 400) {
            throw new \Exception();
        }
    } catch (\Exception $e) {
        // 6. 把死链发给标准输出
        echo $csvRow[0] . PHP_EOL;
    }
}
```

建议： 注意实例化guzzlehttp/guzzle和league/csv组件时，我们使用了\League\Csv和\GuzzleHttp命名空间。我们怎么知道要使用这些命名空间呢？因为我读了guzzlehttp/guzzle和league/csv的文档。记住，好的PHP组件都有文档。

下面我们在*urls.csv*文件中添加一些URL，一行一个，而且至少要有一个是死链。然后，打开终端，执行*scan.php*脚本：

```
php scan.php urls.csv
```

我们执行了php二进制文件，并传入了两个参数。第一个参数是*scan.php*脚本的路径；第二个参数是CSV文件的路径，这个文件中包含一系列URL。如果任何一个URL返回的是表示失败的HTTP响应，这个URL就会在终端界面里输出。

建议： **使用PHP编写命令行脚本**

你知道可以使用PHP编写命令行脚本吗？命令行脚本是在Web应用中自动执行维护任务的好方法。如果想进一步学习如何编写PHP命令行脚本，可以阅读下述资料：

- *http://php.net/manual/wrappers.php.php*。
- *https://php.net/manual/reserved.variables.argv.php*。
- *https://php.net/manual/reserved.variables.argc.php*。

Composer和私有仓库

目前为止，我都假定你使用的是公开可用的开源组件。虽然我也创建并使用开源软件，但我觉得只使用开源的PHP组件或许并不总能解决问题。有时，在同一个应用中我们不得不既使用开源组件也使用私有组件。如果公司要使用内部开发的PHP组件，而基于许可证和安全方面的问题不能将其开源，就一定会遇到这种情况。对Composer来说，这不算问题。

Composer可以管理放在需要认证的仓库中的私有PHP组件。执行`composer install`或`composer update`命令时，如果组件的仓库需要认证凭据，Composer会提醒你。Composer还会询问你是否把仓库的认证凭据保存在本地的*auth.json*文件（和*composer.json*文件放在同一级目录）中。下面是*auth.json*文件的内容示例：

```
{
   "http-basic": {
        "example.org": {
            "username": "your-username",
            "password": "your-password"
        }
    }
}
```

大多数情况下，*auth.json*文件都不能纳入版本控制。我们要让项目的开发者自己创建*auth.json*文件，保存他们自己的认证凭据。

如果不想等待Composer向你询问认证凭据，可以使用下述命令手动告诉Composer远程设备的认证凭据：

```
composer config http-basic.example.org your-username your-password
```

在这个示例中，`http-basic`告诉Composer，我们要为指定的域名添加认证信息。`example.org`是主机名，用于识别存储私有组件仓库的远程设备。最后两个参数是用户名和密码。默认情况下，这个命令会在当前项目中的*auth.json*文件里保存凭据。

你也可以使用`--global`标志，系统全局保存认证凭据。使用这个标志设定认证凭据后，Composer会在本地设备中的所有项目里使用这个凭据。

```
composer config --global http-basic.example.org your-username your-password
```

全局凭据保存在*~/.composer/auth.json*文件中。如果你使用的是Windows系统，全局凭证保存在*%APPDATA%/Composer*文件夹中。

建议： 若想了解在Composer中使用私有仓库的更多信息，请阅读“Authentication management in Composer”（*http://bit.ly/auth-manage*）。

创建PHP组件

现在你应该知道怎么查找并使用PHP组件了。下面换个话题，讨论如何创建PHP组件。具体来说，我们要把这个URL扫描器应用转换成PHP组件，然后提交到Packagist的组件目录中。

创建PHP组件是把工作成果分享给PHP社区成员的好方式。PHP社区乐于分享和帮助他人。如果你在应用中使用了开源组件，创建有创意的新开源组件是回报社区最好的方式。

建议： 注意，别重新编写已经存在的组件。如果你是改进现有的组件，可以在拉取请求中把改进发送给原本的组件。否则，你有可能会创建重复的组件，搅乱PHP组件生态系统。

厂商名和包名

在开发PHP组件之前，首先要选择组件的厂商名和包名。记住，每个PHP组件的名称都由全局唯一的厂商名和包名组成，以防与其他组件有名称冲突。我建议厂商名和包名都只使用小写字母。

厂商名是组件的品牌，用于识别组件属于谁。我开发的很多PHP组件都使用`codeguy`做厂商名，因为这是我在网上使用的昵称。选择的厂商名要能最大限度地表明你的身份或者组件的品牌。

建议： 选择厂商名之前要在Packagist中搜索，确保其他开发者没使用这个名称。

包名用于识别指定厂商名下的PHP组件。一个厂商名下可以有多个组件。对这个示例来说，我使用的厂商名是`modernphp`，包名是`scanner`。

命名空间

第2章说过，每个组件都使用各自的PHP命名空间，以防搅乱全局命名空间，或者与使用同名类的其他组件冲突。

人们经常误以为，组件的命名空间必须与组件的厂商名和包名一致。其实不然，组件使用的命名空间与组件的厂商名和包名无关。厂商名和包名只是为了让Packagist和

Composer识别组件。而组件的命名空间是为了在PHP代码中使用组件。

对本书而言，我们的组件放在`Oreilly\ModernPHP`命名空间中。这个命名空间还不存在，我是为了这个组件专门创建的。

文件系统的组织方式

PHP组件的文件系统结构基本上是定型的：

src/

这个目录包含组件的源码（例如PHP类文件）。

tests/

这个目录包含组件的测试。这个示例用不到这个目录。

composer.json

这是Composer配置文件。这个文件用于描述组件，还会告诉Composer的自动加载器，把组件中符合PSR-4规范的命名空间对应到`src/`目录。

README.md

这个Markdown文件提供关于组件的有用信息，包括组件的名称、说明、作者、用法、贡献者指导方针、软件许可证和要感谢的人。

CONTRIBUTING.md

这个Markdown文件说明别人如何为这个组件做贡献。

LICENSE

这个纯文本文件包含组件的软件许可证。

CHANGELOG.md

这个Markdown文件列出组件在每个版本中引入的改动。

建议： 如果创建PHP组件时遇到问题，可以看一下PHP League优秀的组件样板仓库（*https://github.com/thephpleague/skeleton*）。

---

composer.json文件

PHP组件中必须有*composer.json*文件，而且这个文件的内容必须是有效的JSON。Composer会使用这个文件中的信息查找、安装和自动加载PHP组件。*composer.json*文件还包含组件在Packagist目录中的信息。

示例4-2是这个URL扫描器组件的*composer.json*文件，包含我自己开发PHP组件时最常使用的所有属性。

示例4-2：这个URL扫描器组件的composer.json文件

```
{
    "name": "modernphp/scanner",
    "description": "Scan URLs from a CSV file and report inaccessible URLs",
    "keywords": ["url", "scanner", "csv"],
    "homepage": "http://example.com",
    "license": "MIT",
    "authors": [
        {
            "name": "Josh Lockhart",
            "homepage": "https://github.com/codeguy",
            "role": "Developer"
        }
    ],
    "support": {
        "email": "help@example.com"
    },
    "require": {
        "php" : ">=5.4.0",
        "guzzlehttp/guzzle": "~5.0"
    },
    "require-dev": {
        "phpunit/phpunit": "~4.3"
    },
    "suggest": {
        "league/csv": "~6.0"
    },
    "autoload": {
        "psr-4": {
            "Oreilly\\ModernPHP\\": "src/"
        }
    }
}
```

显然，这里有很多内容要消化。下面我们详细说明每个属性：

`name`

这是组件的厂商名和包名，二者之间使用/符号分隔。这个属性的值会在Packagist中显示。

`description`

这个属性的值是几句话，简要说明组件。这个属性的值会在Packagist中显示。

`keywords`

这个属性的值是几个描述组件的关键字。这些关键字用于帮助别人在Packagist中找到这个组件。

`homepage`

这是组件网站的URL。

`license`

这是PHP组件采用的软件许可证。我喜欢使用MIT Public License。*http://choosealicense.com*对软件许可证做了详细说明。记住，发布代码时一定要使用许可证。

`authors`

这是一个数组，包含项目中每个作者的信息。每个作者的信息至少要包含姓名和网站URL。

`support`

这是组件的用户获取技术支持的方式。我喜欢设为电子邮件地址和支持讨论组的URL。你也可以列出IRC频道等。

`require`

这个属性列出组件自身依赖的组件。我们应该列出每个依赖的厂商名和包名，以及最小版本号。我还喜欢列出组件需要的最小PHP版本号。在开发环境和生产环境都会安装这个属性中列出的全部依赖。

`require-dev`

这个属性的值与`require`属性类似，不过列出的是开发这个组件所需的依赖。例如，我通常会把`phpunit`当做开发依赖，以便组件的其他贡献者能编写和运行测试。这些依赖只在开发时安装，在生产环境中使用时不会安装。

`suggest`

这个属性的值与`require`属性类似，不过只是建议安装的组件，以防与其他组件合作时需要。与`require`属性不同，这个对象的值是自由的文本字段，用于描述每个建议安装的组件。Composer不会安装这些建议安装的组件。

`autoload`

这个属性告诉Composer的自动加载器如何自动加载这个组件。我建议使用符合PSR-4规范的自动加载器，如示例4-2所示。在`psr-4`属性中，我们要把组件的命名空间前缀与相对组件根目录的文件系统路径对应起来。这样，我们的组件就符合PSR-4自动加载器标准了。在示例4-2中，我把`Oreilly\ModernPHP`命名空间与*src/*目录对应了起来。要对应的命名空间必须以两个反斜线（`\\`）结尾，防止与命名空间中有类似字符序列的其他组件冲突。在这个示例中，如果实例化虚构的`Oreilly\ModernPHP\Url\Scanner`类，Composer会自动加载*src/Url/Scanner.php*文件。

建议： *composer.json*文件完整的格式参见*http://getcomposer.org*。

README文件

*README*文件通常是用户最先阅读的文件。对托管在GitHub和Bitbucket中的组件来说，更是如此。因此，组件的*README*文件至少要提供以下信息：

- 组件的名称和描述。
- 安装说明。
- 用法说明。
- 测试说明。
- 贡献方式说明。
- 支持资源。
- 作者信息。
- 软件许可证。

建议： GitHub和Bitbucket都可以使用Markdown格式渲染*README*文件。因此，我们可以使用标题、列表、链接和图像编写格式良好的*README*文件。我们要合理利用这些格式！我们只需把README文件的扩展名设为*.md*或*.markdown*。*CONTRIBUTING*和*CHANGELOG*文件也是如此。Markdown格式的详细说明参见达林·法尔鲍尔的网站（*http://bit.ly/markdown-doc*）。

实现组件

现在我们要实现组件的具体功能了。这一步我们要编写组成PHP组件的类、接口和性状。编写什么类以及编写多少类完全取决于PHP组件的作用。不过，组件中的所有类、接口和性状都要保存在*src/*目录中，而且都要放在*composer.json*文件中设定的命名空间前缀名下。

对这个示例来说，我要只会创建一个PHP类，这个类名为`Scanner`，位于子命名空间`Url`中。这个子命名空间位于*composer.json*文件中设定的`Oreilly\ModernPHP`命名空间中。`Scanner`类保存在*src/Url/Scanner.php*文件中。`Scanner`类实现的逻辑与前面那个URL扫描器示例应用相同，只不过现在我们要把扫描URL的功能封装在一个PHP类中（如示例4-3所示）。

示例4-3：这个URL扫描器组件的类

```
<?php
namespace Oreilly\ModernPHP\Url;

class Scanner
```

```
{
    /**
     * @var array 一个由URL组成的数组
     */
    protected $urls;

    /**
     * @var \GuzzleHttp\Client
     */
    protected $httpClient;

    /**
     * 构造方法
     * @param array $urls一个要扫描的URL数组
     */
    public function __construct(array $urls)
    {
        $this->urls = $urls;
        $this->httpClient = new \GuzzleHttp\Client();
    }

    /**
     * 获取死链
     * @return array
     */
    public function getInvalidUrls()
    {
        $invalidUrls = [];
        foreach ($this->urls as $url) {
            try {
                $statusCode = $this->getStatusCodeForUrl($url);
            } catch (\Exception $e) {
                $statusCode = 500;
            }

            if ($statusCode >= 400) {
                array_push($invalidUrls, [
                    'url' => $url,
                    'status' => $statusCode
                ]);
            }
        }

        return $invalidUrls;
    }

    /**
     * 获取指定URL的HTTP状态码
     * @param string $url远程URL
     * @return int HTTP状态码
     */
    protected function getStatusCodeForUrl($url)
    {
        $httpResponse = $this->httpClient->options($url);

        return $httpResponse->getStatusCode();
```

```
    }
}
```

我们没有解析并迭代处理一个CSV文件，而是把一个URL数组传给`Scanner`了类的构造方法，因为我们要尽量让这个扫描URL的类通用。如果直接处理CSV文件，那就限制了这个组件的用途。可是，如果处理一个URL数组，就能让最终用户决定如何获取URL数组（可以从PHP数组、CSV文件或迭代器中获取）。所以我们才在清单文件*composer.json*的`suggest`属性中设定了`league/csv`组件，建议安装`league/csv`组件，因为它对使用我们这个组件的开发者有所帮助。

`Scanner`类对`guzzlehttp/guzzle`组件有硬性依赖，不过我们在`getStatusCodeForUrl()`方法中隔离了处理HTTP请求的功能，因此可以在组件的单元测试中创建这个方法的桩件（stub，或覆盖这个方法），不让测试依赖于可用的互联网连接。

版本控制

我们做得差不多了。在把组件提交到Packagist之前，我们必须把组件发布到公开的代码仓库中。我喜欢把自己的开源PHP组件发布到GitHub。不过，发布到任何公开的Git仓库都行。这个组件我也发布到GitHub了，仓库的地址是*https://github.com/modern-php/scanner*。

我们最好使用语义版本方案为组件的每个版本创建标签（tag），以便组件的使用者使用组件的某个具体版本（例如`~1.2`）。我为这个URL扫描器组件创建了一个名为`1.0.0`的标签。

提交到Packagist

现在可以把组件提交到Packagist了。如果你不使用GitHub，先去注册一个Packagist账户（*https://packagist.org/register/*）。如果使用GitHub，也可以使用GitHub的凭据登录Packagist。

登录后，单击网页右上角那个大大的绿色“Submit Package”按钮，然后在“Repository URL”文本框中输入完整的Git仓库URL，再单击“Check”按钮。Packagist会验证输入的仓库URL，然后让你确认此次提交。单击“Submit”按钮，结束此次提交。Packagist会创建组件的页面，然后重定向到这个页面，如图4-6所示。

你会发现，Packagist从组件的*composer.json*文件中读取了组件的名称、描述、关键字、依赖和建议。还会发现显示了仓库的分支和标签。Packagist会把仓库的标签和语言版本号对应起来。这就是我建议你使用合理的版本号（例如`1.0.0`和`1.1.0`等）创建仓库标签的原因。可是，我们还会看到一个大大的红色警告消息，内容如下：

图4-6：Packagist中显示的组件页面

```
This package is not auto-updated. Please set up the GitHub Service Hook for
Packagist so that it gets updated whenever you push!
```

我们可以在GitHub或Bitbucket中创建一个钩子，每次更新组件的仓库时通知Packagist。在仓库中设置钩子的方法参阅*https:// packagist.org/profile/*。

使用这个组件

工作做完了！现在任何人都能使用Composer安装这个URL扫描器组件，然后在自己的PHP应用中使用了。在终端执行下述命令，使用Composer安装这个URL扫描器组件：

```
composer require modernphp/scanner
```

然后可以像示例4-4那样使用这个URL扫描器组件。

示例4-4：使用这个URL扫描器组件

```
<?php
require 'vendor/autoload.php';

$urls = [
    'http://www.apple.com',
    'http://php.net',
    'http://sdfssdwerw.org'
];
$scanner = new \Oreilly\ModernPHP\Url\Scanner($urls);
print_r($scanner->getInvalidUrls());
```

第5章

良好实践

本章介绍开发PHP应用时应该运用的各种良好实践。沿用这些良好实践能让应用运行得更快，更安全，也更稳定。PHP语言历史悠久，在过去很长一段时间内不断引入工具，因此积累了很多工具，我们会使用这些工具运用即将介绍的良好实践。随着时间的推移，工具会变化，新版PHP会引入更好的新解决方案。可是，PHP语言中仍有以前遗留下来的过时工具，如果不小心，使用这些过时的工具可能会开发出运行速度慢且不安全的应用。因此，我们要知道哪些工具可以使用，而哪些工具要摒弃。本章的目的就是告诉你如何做到这一点。

我不会站在象牙塔上布道“最佳实践”。本章的内容是一些实用的好建议，我平时在自己的所有项目中都会使用。下面介绍的这些知识可以立即在你的的项目中使用。

注意： 本章说明的良好实践在以前和现在的PHP版本中都可以做到，但是随着PHP语言的发展，实现这些良好实践的方式有所不同，新版PHP会引入一些易于实现的工具。本章说明如何使用PHP 5.3及以上版本提供的最新工具运用这些良好实践。

过滤、验证和转义

福克斯·马尔德[译注1]说的没错，不要相信任何人。同理，不要相信任何来自不受自己直接控制的数据源中的数据，例如下述这些外部源：

- `$_GET`。
- `$_POST`。

译注1： 福克斯·马尔德是科幻电视剧《X档案》中的角色。

- `$_REQUEST`。
- `$_COOKIE`。
- `$argv`。
- `php://stdin`。
- `php://input`。
- `file_get_contents()`。
- 远程数据库。
- 远程API。
- 来自客户端的数据。

所有这些外部数据源都可能是攻击媒介，可能会（有意或无意）把恶意数据注入PHP脚本。编写接收用户输入然后渲染输出的PHP脚本很容易，可是要安全实现的话，需要下一番功夫。我能给你的最基本的建议是：过滤输入、验证数据，以及转义输出。

过滤输入

过滤输入（即来自前面所列数据源中的任何数据）是指，转义或删除不安全的字符。在数据到达应用的存储层（例如Redis或MySQL）之前，一定要过滤输入数据。这是第一道防线。假如网站的评论表单接受HTML，默认情况下，访客可以毫无阻拦地在评论中加入恶意的<script>标签，如下所示：

```
<p>
    This was a helpful article!
</p>
<script>window.location.href='http://example.com';</script>
```

如果不过滤这个评论，恶意代码会存入数据库，然后在网站的标记中渲染。当网站的访客访问包含这个未过滤评论的页面时，会重定向到一个做坏事的网站。这个示例说明了为什么必须要过滤不受自己控制的输入数据。根据我的经验，最需要过滤的输入数据类型有：HTML、SQL查询和用户资料信息（例如电子邮件地址和电话号码）。

HTML

我们要使用`htmlentities()`函数（*http://php.net/manual/function.htmlentities.php*）过滤HTML，把特殊字符（例如&，>，″）转换成对应的HTML实体（如示例5-1所示）。这个函数会转义指定字符串中的所有HTML字符，以便在应用的存储层安全渲染。

可是，htmlentities()函数很傻，不会验证HTML输入。默认情况下，这个函数不会转义单引号。而且也检测不出输入字符串的字符集。htmlentities()函数的正确使用方式是：第一个参数是输入字符串；第二个参数设为ENT_QUOTES常量，转义单引号；第三个参数设为输入字符串的字符集。

示例5-1：使用htmlentities()函数过滤输入

```
<?php
$input = '<p><script>alert("You won the Nigerian lottery!");</script></p>';
echo htmlentities($input, ENT_QUOTES, 'UTF-8');
```

如果需要更多过滤HTML输入的方式，可以使用HTML Purifier库（*http://htmlpurifier.org/*）。HTML Purifier是个很强健且安全的PHP库，作用是使用指定的规则过滤HTML输入。这个库的缺点是，速度慢，而且可能难以配置。

警告： 别使用正则表达式函数过滤HTML，例如preg_replace()、preg_replace_all()和preg_replace_call back()。正则表达式很复杂，可能导致HTML无效，而且出错的几率高。

SQL查询

有时必须根据输入数据构建SQL查询。这些输入数据可能来自HTTP请求的查询字符串（例如?user=1），也可能来自HTTP请求的URI片段（例如/users/1）。如果不小心，不怀好意的人能故意损坏SQL查询，对数据库造成严重破坏。例如，我看到很多初级PHP程序员直接拼接$_GET和$_POST中的原始输入数据，以此构建SQL查询，如示例5-2所示。

示例5-2：构建SQL查询不好的方式

```
$sql = sprintf(
    'UPDATE users SET password = "%s" WHERE id = %s',
    $_POST['password'],
    $_GET['id']
);
```

这么做不好！想想如果有人向PHP脚本发送下述HTTP请求会有什么后果？

```
POST /user?id=1 HTTP/1.1
Content-Length: 17
Content-Type: application/x-www-form-urlencoded

password=abc";--
```

这个HTTP请求会把每个用户的密码都设为abc，因为很多SQL数据库把--视作注释的开头，所以会忽略后续文本。在SQL查询中一定不能使用未过滤的输入数据。如果需要在SQL查询中使用输入数据，要使用PDO预处理语句（prepared statement）。PDO是PHP

内置的数据库抽象层，使用一个接口表示多种数据库。PDO预处理语句是PDO提供的一个工具，用于过滤外部数据，然后把过滤后的数据嵌入SQL查询，避免出现示例5-2那种问题。我觉得PDO和预处理语句特别重要，所以本章后面会分别在单独的一节说明。

用户资料信息

如果应用中有用户账户，可能就要处理电子邮件地址、电话号码和邮政编码等资料信息。PHP预料到会遇到这种情况，因此提供了`filter_var()`和`filter_input()`函数。这两个函数的参数能使用不同的标志，过滤不同类型的输入：电子邮件地址、URL编码字符串、整数、浮点数、HTML字符、URL和特定范围内的ASCII字符。

示例5-3展示了如何过滤电子邮件地址，删除除字母、数字和`!#$%&'*+-/=?^_{|}~@.[]`之外的所有其他字符。

示例5-3：过滤用户资料中的电子邮件地址

```
<?php
$email = 'john@example.com';
$emailSafe = filter_var($email, FILTER_SANITIZE_EMAIL);
```

示例5-4演示了如何过滤用户的个人简介，删除小于ASCII 32的字符，转义大于ASCII 127的字符。

示例5-4：过滤用户资料中的外国字符

```
<?php
$string = "\nIñtërnâtiônàlizætiøn\t";
$safeString = filter_var(
    $string,
    FILTER_SANITIZE_STRING,
    FILTER_FLAG_STRIP_LOW|FILTER_FLAG_ENCODE_HIGH
);
```

注意： `filter_var()`函数能使用的更多标志和选项参见*http://php.net/manual/function.filter-var.php*。

验证数据

验证数据也很重要。与过滤不同，验证不会从输入数据中删除信息，而是只确认输入数据是否符合预期。如果需要的是电子邮件地址，确保输入数据是电子邮件地址；如果需要的是电话号码，确保输入数据是电话号码。验证数据时只需做到这一点。验证是为了保证在应用的存储层保存符合特定格式的正确数据。如果遇到无效数据，要中止数据存储操作，并显示适当的错误消息来提醒应用的用户。验证还能避免数据库出现潜在的错误。例如，如果MySQL期望使用`DATETIME`类型的值，而提供的却是`next year`字符串，

那么MySQL会报错或者使用（不正确的）默认值。不管哪种处理方式，应用的数据完整性都受到了无效数据的破坏。

我们可以把某个FILTER_VALIDATE_*标志传给filter_var()函数，验证用户的输入。PHP提供了用于验证布尔值、电子邮件地址、浮点数、整数、IP地址、正则表达式和URL的标志。示例5-5展示了如何验证电子邮件地址。

示例5-5：验证电子邮件地址

```
<?php
$input = 'john@example.com';
$isEmail = filter_var($input, FILTER_VALIDATE_EMAIL);
if ($isEmail !== false) {
    echo "Success";
} else {
    echo "Fail";
}
```

我们要特别注意filter_var()函数的返回值。如果验证成功，返回值是要验证的值；如果验证失败，返回值是false。

虽然filter_var()函数提供了很多用于验证的标志，但这个函数不能验证所有数据。除了filter_var()函数，我还推荐使用下述提供验证功能的组件：

- aura/filter（*https://packagist.org/packages/aura/filter*）。
- respect/validation（*https://packagist.org/packages/respect/validation*）。
- symfony/validator（*https://packagist.org/packages/symfony/validator*）。

建议：输入数据既要验证也要过滤，以此确保输入数据是安全的，而且符合预期。

转义输出

把输出渲染成网页或API响应时，一定要转义输出。这也是一种防护措施，能避免渲染恶意代码，还能防止应用的用户无意中执行恶意代码。

我们可以使用前面提到的htmlentities()函数转义输出。第二个参数一定要使用ENT_QUOTES，让这个函数转义单引号和双引号。而且，还要在第三个参数中指定合适的字符编码（通常是UTF-8）。示例5-6展示了如何在渲染前转义HTML输出。

示例5-6：使用htmlentities()函数转义输出

```
<?php
$output = '<p><script>alert("NSA backdoor installed");</script>';
echo htmlentities($output, ENT_QUOTES, 'UTF-8');
```

有些PHP模板引擎，例如`twig/twig`（*https://packagist.org/packages/twig/twig*，我的最爱）和`smarty/smarty`（*https://packagist.org/packages/smarty/smarty*），会自动转义输出。以Twig为例，SensioLabs开发的这个模板引擎默认会转义所有输出，除非明确告诉它不要转义。这种默认处理方式很棒，为PHP Web应用提供了有力的安全保障。

密码

随着在线攻击的增多，密码安全越来越重要。因为重要的零售商被黑，你注销过多少张信用卡？很多零售商因为没有使用最好的安全措施而沦为恶意黑客的攻击对象。PHP应用一样，如果没有合适的预防措施，也会受到攻击。

其中一个重要的预防措施是保护密码。作为开发者，我们要担起安全管理、计算哈希和存储用户密码的责任。不管应用是简单的游戏还是绝密商业文件的仓库，都要做到这一点。用户把他们的信息托付给你，是相信你能使用最好的安全措施保护他们的信息。我见过很多不知道如何安全管理密码的PHP开发者。这怨不得他们，毕竟密码很难安全管理。幸好，PHP内置了一些工具，让保护密码变得十分容易。本节说明如何根据现代的安全措施使用这些工具。

绝对不能知道用户的密码

我们绝对不能知道用户的密码，也不能有获取用户密码的方式。如果应用的数据库被黑，你肯定不希望数据库中有纯文本或能解密的密码。一旦密码泄露，用户对你的信任会严重降低，而且你或你的公司要背负大量法律责任。你知道的越少越安全。

绝对不要约束用户的密码

如果某个网站要求账户的密码要符合特定的格式，我会心灰意冷。如果限制账户密码的长度不能超过N个字符，我会更生气。为什么？我知道要求密码使用特定格式有时是为了兼容以前的应用或者数据库，可这不是缺乏安全措施的借口。

绝对不能约束用户的密码。如果要求密码符合特定的模式，其实是为不怀好意的人提供了攻击应用的途径。如果必须约束用户的密码，我建议只限制最小长度。把常用的密码或基于字典创建的密码加入黑名单也是好主意。

绝对不能通过电子邮件发送用户的密码

绝对不能通过电子邮件发送密码。如果你通过电子邮件给我发送密码，我会知道三件事：你知道我的密码；你使用纯文本或能解密的格式存储了我的密码；你没有对通过互联网发送纯文本的密码感到不安。

我们应该在电子邮件中发送用于设定或修改密码的URL。Web应用通常会生成一个唯一的令牌，这个令牌只在设定或修改密码时使用一次。例如，我忘了自己在你应用中的账户密码，我单击登录表单中的“忘记密码”链接，然后转到一个表单，我在这个表单中填写我的电子邮件地址，请求重设密码。你的应用生成一个唯一的令牌，并把这个令牌关联到我的电子邮件地址对应的账户上，然后发送一封电子邮件到账户的电子邮件地址。这封电子邮件中有一个URL，其中某个URL片段或查询字符串的值是这个唯一的令牌。我访问这个URL，你的应用验证令牌，如果令牌有效，就让我为账户重设一个密码。我重设密码之后，你的应用把这个令牌设为失效。

使用bcrypt计算用户密码的哈希值

我们应该计算用户密码的哈希值，而不能加密用户的密码。加密和哈希不是一回事。加密是双向算法，加密的数据以后可以解密。而哈希是单向算法，哈希后的数据不能再还原成原始值，而且相同的数据得到的哈希值始终相同。

在数据库中存储用户的密码时，要先计算密码的哈希值，然后在数据库中存储密码的哈希值。如果黑客攻入了数据库，他们只能看到无意义的密码哈希值，需要花费大量时间和NSA资源才能破解。

哈希算法有很多种（例如MD5、SHA1、bcrypt和scrypt）。有些算法的速度很快，用于验证数据完整性；有些算法的速度则很慢，旨在提高安全性。生成密码和存储密码时需要使用速度慢、安全性高的算法。

目前，经同行审查，最安全的哈希算法是bcrypt。与MD5和SHA1不同，bcrypt是故意设计得很慢。bcrypt算法会自动加盐，防止潜在的彩虹表攻击。bcrypt算法会花费大量时间（以秒计）反复处理数据，生成特别安全的哈希值。在这个过程中，处理数据的次数叫工作因子（work factor）。工作因子的值越高，不怀好意的人破解密码哈希值所需的时间会成指数倍增长。bcrypt算法永不过时，如果计算机的运算速度变快了，我们只需提高工作因子的值。

bcrypt算法得到了同行的大量审查，很多比我聪明多的人都审查过bcrypt算法，试图找出潜在的漏洞，但目前为止一个漏洞都没找到。我们一定要使用经同行审查过的哈希算法，而不要自己创建。群众的眼睛是雪亮的，而你可能并不是加密专家（如果你是的话，给布鲁斯·施奈尔[译注2]带个好）。

译注2：布鲁斯·施奈尔是美国著名的密码学学者、信息安全专家与作家，有很多著作，例如《应用密码学》。

密码哈希API

读过前文可以得知，处理用户的密码时要考虑很多事情。安东尼·费拉拉（*http://blog.ircmaxell.com*）很懂我们的心思，他开发了PHP 5.5.0中原生的密码哈希API（*http://php.net/manual/book.password.php*）。PHP 原生的密码哈希API提供了很多易于使用的函数，大大简化了计算密码哈希值和验证密码的操作。而且，这个密码哈希API默认使用bcrypt哈希算法。

注意： 安东尼·费拉拉（Twitter账号为@ircmaxell）是谷歌的技术推广员，是PHP性能和安全方面的权威。安东尼还是PHP密码哈希API的作者。我建议你在Twitter上关注安东尼，也建议你阅读他的博客（*http://blog.ircmaxell.com*）。我要特别感谢安东尼，他对PHP的贡献让我们能更轻易地使用最好的安全措施，从而提升PHP应用的安全性。

开发Web应用时，有两个地方会用到密码哈希API：注册用户和登录用户。下面我们探讨PHP密码哈希API是如何简化这两个操作的。

注册用户

没有用户，Web应用无法生存下去，而用户需要一种注册账户的方式。假设我们虚构的应用中有个PHP文件，其URL是*/register.php*。这个PHP文件用来接收URL编码的HTTP POST请求，从中获取电子邮件地址和密码。如果电子邮件地址有效，而且密码至少有八个字符，我们就创建一个用户账户。下面是一个HTTP POST请求示例：

```
POST /register.php HTTP/1.1
Content-Length: 43
Content-Type: application/x-www-form-urlencoded

email=john@example.com&password=sekritshhh!
```

示例5-7 是接收这个HTTP POST请求的*register.php*文件。

示例5-7：注册用户的脚本

```
01 <?php
02 try {
03     // 验证电子邮件地址
04     $email = filter_input(INPUT_POST, 'email', FILTER_VALIDATE_EMAIL);
05     if (!$email) {
06         throw new Exception('Invalid email');
07     }
08
09     // 验证密码
10     $password = filter_input(INPUT_POST, 'password');
11     if (!$password || mb_strlen($password) < 8) {
12         throw new Exception('Password must contain 8+ characters');
13     }
```

```
14
15      // 创建密码的哈希值
16      $passwordHash = password_hash(
17          $password,
18          PASSWORD_DEFAULT,
19          ['cost' => 12]
20      );
21      if ($passwordHash === false) {
22          throw new Exception('Password hash failed');
23      }
24
25      // 创建用户账户（注意，这是虚构的代码）
26      $user = new User();
27      $user->email = $email;
28      $user->password_hash = $passwordHash;
29      $user->save();
30
31      // 重定向到登录页面
32      header('HTTP/1.1 302 Redirect');
33      header('Location: /login.php');
34  } catch (Exception $e) {
35      // 报告错误
36      header('HTTP/1.1 400 Bad request');
37      echo $e->getMessage();
38  }
```

在示例5-7中：

- 第4～7行验证用户的电子邮件地址。如果电子邮件地址无效，抛出异常。
- 第10～13行验证从HTTP请求主体中获取的纯文本密码。如果纯文本密码少于八个字符，抛出异常。
- 第16～23行使用PHP密码哈希API中的`password_hash()`函数创建密码的哈希值。这个函数的第一个参数是纯文本密码；第二个参数是`PASSWORD_DEFAULT`常量，告诉PHP使用bcrypt哈希算法；第三个参数是一个数组，指定哈希选项。这个数组中的`cost`键用于设定bcrypt的工作因子。工作因此的默认值是10，不过你应该根据硬件的具体计算能力提高这个值。计算哈希值一般需要0.1～0.5s。如果计算密码的哈希值失败，抛出异常。
- 第26～29行展示了保存虚构的用户账户。这几行是虚构的代码，你应该针对应用的具体需求修改这些代码。我想强调的是，要把密码的哈希值存储到数据库中，而不能直接存储从HTTP请求主体中获取的纯文本密码。我们还存储了用于识别和登录用户账户的电子邮件地址。

建议： 密码的哈希值要存储在`VARCHAR(255)`类型的数据库列中。这样便于以后存储比现在的bcrypt算法得到的哈希值更长的密码。

登录用户

我们虚构的应用还有一个URL为*/login.php*的PHP文件。这个文件用于接收包含电子邮件地址和密码的HTTP POST请求，识别、认证，并登录用户。下面是一个HTTP POST请求示例：

```
POST /login.php HTTP/1.1
Content-Length: 43
Content-Type: application/x-www-form-urlencoded

email=john@example.com&password=sekritshhh!
```

*login.php*文件会找到电子邮件地址对应的用户账户，验证提交的密码与用户账户的密码哈希值匹配后，登录这个用户账户。示例5-8是*login.php*文件的内容。

示例5-8：登录用户的脚本

```
<?php
session_start();
try {
    // 从请求主体中获取电子邮件地址
    $email = filter_input(INPUT_POST, 'email');

    // 从请求主体中获取密码
    $password = filter_input(INPUT_POST, 'password');

    // 使用电子邮件地址查找账户（注意，这是虚构的代码）
    $user = User::findByEmail($email);

    // 验证密码和账户的密码哈希值是否匹配
    if (password_verify($password, $user->password_hash) === false) {
        throw new Exception('Invalid password');
    }

    // 如果需要，重新计算密码的哈希值（参见下面的说明）
    $currentHashAlgorithm = PASSWORD_DEFAULT;
    $currentHashOptions = array('cost' => 15);
    $passwordNeedsRehash = password_needs_rehash(
        $user->password_hash,
        $currentHashAlgorithm,
        $currentHashOptions
    );
    if ($passwordNeedsRehash === true) {
        // 保存新计算得到的密码哈希值（注意，这是虚构的代码）
        $user->password_hash = password_hash(
            $password,
            $currentHashAlgorithm,
            $currentHashOptions
        );
        $user->save();
    }

    // 把登录状态保存到会话中
```

```
37      $_SESSION['user_logged_in'] = 'yes';
38      $_SESSION['user_email'] = $email;
39
40      // 重定向到个人资料页面
41      header('HTTP/1.1 302 Redirect');
42      header('Location: /user-profile.php');
43  } catch (Exception $e) {
44      header('HTTP/1.1 401 Unauthorized');
45      echo $e->getMessage();
46  }
```

在示例5-8中：

- 第5 行和第8行分别从HTTP请求主体中获取电子邮件地址和密码。
- 第11行查找HTTP请求主体中提交的电子邮件地址对应的用户记录。我使用的是虚构的代码，你应该根据应用的具体需求修改这些代码。
- 第14～16行使用`password_verify()`函数比较HTTP请求主体中提交的纯文本密码和用户记录中存储的密码哈希值。如果验证失败，抛出异常。
- 第19～34行调用`password_needs_rehash()`函数，确认用户记录中的密码哈希值是否符合最新的密码算法选项。如果用户记录中的密码哈希值过时了，我们就使用最新的算法选项重新计算哈希值，然后使用新哈希值更新用户记录。

验证密码

我们调用`password_verify()`函数对比从HTTP请求主体中获取的纯文本密码和用户记录中存储的密码哈希值。这个函数有两个参数。第一个参数是纯文本密码，第二个参数是用户记录中现有的密码哈希值。如果`password_verify()`函数返回`true`，说明纯文本密码是正确的，那就登录用户；否则，说明纯文本密码错误，要中止登录过程。

重新计算密码的哈希值

在示例5-8中第17行代码之后，认证已经通过，可以登录用户了。可是，在登录前，一定要检查用户记录中现有的密码哈希值是否过期。如果过期了，要重新计算密码哈希值。

为什么要重新计算密码哈希值呢？假设我们的应用创建于两年前，那时使用的bcrypt工作因子是10。而现在我们使用的工作因子是20，因为黑客更聪明了，而且计算机速度更快了。可是，有些用户账户的密码哈希值仍然是工作因子为10时生成的，因此，登录请求通过认证之后，要使用`password_needs_rehash()`函数检查用户记录中现有的密码哈希值是否需要更新。这个函数能确保指定的密码哈希值是使用最新的哈希算法选项创建的。如果确实需要重新计算密码的哈希值，我们要使用当前的算法选项计算HTTP请求主体中纯文本密码的哈希值，然后使用新哈希值更新用户记录。

建议： 在登录用户的脚本中使用password_needs_rehash()函数最简单，因为能同时获取旧密码哈希值和纯文本密码。

PHP 5.5.0之前的密码哈希API

如果无法使用PHP 5.5.0或以上版本，也不用害怕，可以使用安东尼·费拉拉开发的ircmaxell/password-compat组件（*https://packagist.org/packages/ircmaxell/password-compat*）。这个组件实现了PHP密码哈希API中的所有函数：

- password_hash()
- password_get_info()
- password_needs_rehash()
- password_verify()

费拉拉开发的ircmaxell/password-compat组件可以直接替代现代的PHP密码哈希API，使用Composer把这个组件添加到你的应用中就行了。

日期、时间和时区

日期和时间很难处理，几乎每个PHP开发者在处理日期和时间时都犯过错。所以我建议不要自己处理日期和时间。处理日期和时间时要考虑很多事情，例如日期的格式、时区、夏令时、闰年、闰秒和天数各异的月份，自己处理时太容易出错了。我们应该使用PHP 5.2.0引入的DateTime、DateInterval和DateTimeZone类。这些类很方便，提供了简单的面向对象接口，能准确创建及处理日期、时间和时区。

设置默认时区

首先，我们要为PHP中处理日期和时间的函数设置默认时区。如果不设置默认时区，PHP会显示一个E_WARNING消息。设置默认时区有两种方式。可以像下面这样在*php.ini*文件中设置：

```
date.timezone = 'America/New_York';
```

也可以在运行时使用date_default_timezone_set()函数设置默认时区（如示例5-9所示）。

示例5-9：设置默认时区

```
<?php
date_default_timezone_set('America/New_York');
```

这两种方式都要求使用有效的时区标识符。PHP时区标识符的完整列表可以在这个网页中查看：*http://php.net/manual/timezones.php*。

DateTime类

DateTime类提供了一个面向对象接口，用于管理日期和时间。一个DateTime实例表示一个具体的日期和时间。DateTime类的构造方法（如示例5-10所示）是创建DateTime新实例最简单的方式。

示例5-10：使用DateTime类

```
<?php
$datetime = new DateTime();
```

如果没有参数，DateTime类的构造方法创建的是一个表示当前日期和时间的实例。我们可以把一个字符串传入 DateTime类的构造方法，指定其他日期和时间（如示例5-11所示）。这个字符串参数必须使用某种有效的日期和时间格式，详情参见*http://php.net/manual/datetime.formats.php*。

示例5-11：把参数传入DateTime类的构造方法

```
<?php
$datetime = new DateTime('2014-04-27 5:03 AM');
```

理想情况下，我们会指定PHP能理解的日期和时间格式。可是并不总是如此。有时，我们必须处理其他格式或出乎意料的格式。我每天都会遇到这种问题。我的很多客户会发给我Excel电子表格，让我把其中的数据导入应用，而且每个客户使用的日期和时间格式几乎完全不同。有了DateTime类，这算不上是问题。

通过DateTime::createFromFormat()静态方法，我们可以使用自定义的格式创建DateTime实例。这个方法的第一个参数是表示日期和时间格式的字符串，第二个参数是要使用这种格式的日期和时间字符串（如示例5-12所示）。

示例5-12：使用DateTime类的静态构造方法

```
<?php
$datetime = DateTime::createFromFormat('M j, Y H:i:s', 'Jan 2, 2014 23:04:12');
```

注意：DateTime::createFromFormat()静态方法使用的日期和时间格式与date()函数一样。可用的日期和时间格式参见*http://php.net/manual/datetime.createfromformat.php*。

DateInterval类

处理DateTime实例之前，基本上要先了解DateInterval类。DateInterval实例表示

长度固定的时间段（例如“两天”），或者相对而言的时间段（例如“昨天”）。DateInterval实例用于修改DateTime实例。例如，DateTime类提供了用于处理DateTime实例的add()和sub()方法。这两个方法的参数都是一个DateInterval实例，指定要添加到DateTime实例中的时间量，或者要从DateTime实例中减去的时间量。

实例化DateInterval类的方式是使用构造方法。DateInterval类构造方法的参数是一个字符串，表示间隔规约。间隔规约看起来很难理解，其实不难。间隔规约是一个以字母P开头的字符串，后面跟着一个整数，最后是一个周期标志符，限定前面的整数。有效的周期标志符如下：

- Y（年）
- M（月）
- D（日）
- W（周）
- H（时）
- M（分）
- S（秒）

间隔规约中既可以有日期也可以有时间。如果有时间，要在日期和时间两部分之间加上字母T。例如，间隔规约P2D表示两天，间隔规约P2DT5H2M表示两天五小时两分钟。

示例5-13展示了如何使用add()方法把DateTime实例表示的日期和时间向后移一段时间。

示例5-13：使用DateInterval类

```
<?php
// 创建DateTime实例
$datetime = new DateTime('2014-01-01 14:00:00');

// 创建长度为两周的间隔
$interval = new DateInterval('P2W');

// 修改DateTime实例
$datetime->add($interval);
echo $datetime->format('Y-m-d H:i:s');
```

我们还可以创建反向的DateInterval实例（如示例5-14所示）。通过这种方式，我们可以把DatePeriod实例倒序向前推移。

示例5-14：创建反向的DateInterval实例

```
$dateStart = new \DateTime();
```

```
$dateInterval = \DateInterval::createFromDateString('-1 day');
$datePeriod = new \DatePeriod($dateStart, $dateInterval, 3);
foreach ($datePeriod as $date) {
    echo $date->format('Y-m-d'), PHP_EOL;
}
```

这段代码的输出为：

```
2014-12-08
2014-12-07
2014-12-06
2014-12-05
```

DateTimeZone类

如果应用要迎合国际客户，可能要和时区斗争。时区很难处理，而且很多PHP开发者都不理解时区。

PHP使用DateTimeZone类表示时区。我们只需把有效的时区标识符传给DateTimeZone类的构造方法：

```
<?php
$timezone = new DateTimeZone('America/New_York');
```

注意：全部有效的时区标识符参见*http://php.net/manual/timezones.php*。

创建DateTime实例时经常要使用DateTimeZone实例。DateTime类构造方法可选的第二个参数是一个DateTimeZone实例。传入这个参数后，DateTime实例的值，以及对这个值的所有修改都相对指定的时区而言。如果不传入第二个参数，使用的是设置的默认时区。

```
<?php
$timezone = new DateTimeZone('America/New_York');
$datetime = new DateTime('2014-08-20', $timezone);
```

实例化之后，可以使用`setTimezone()`方法修改DateTime实例的时区（如示例5-15所示）。

示例5-15：使用DateTimeZone类

```
<?php
$timezone = new DateTimeZone('America/New_York');
$datetime = new \DateTime('2014-08-20', $timezone);
$datetime->setTimezone(new DateTimeZone('Asia/Hong_Kong'));
```

我发现一直使用UTC时区最简单。我的服务器使用的时区是UTC，我为PHP设置的默认时区也是UTC。如果要把日期和时间值存入数据库，我保存时区为UTC的日期和时间。把数

据显示给应用的用户查看时，我再把时区为UTC的日期和时间值转换成适当时区的日期和时间。

DatePeriod类

有时我们需要迭代处理一段时间内反复出现的一系列日期和时间，重复在日程表中记事就是个好例子。DatePeriod类可以解决这种问题。DatePeriod类的构造方法接受三个参数，而且都必须提供：

- 一个DateTime实例，表示迭代开始时的日期和时间。
- 一个DateInterval实例，表示到下一个日期和时间的间隔。
- 一个整数，表示迭代的总次数。

DatePeriod实例是迭代器，每次迭代时都会产出一个DateTime实例。示例5-16会产出四个间隔两周的日期和时间。

示例5-16：使用DatePeriod类

```
<?php
$start = new DateTime();
$interval = new DateInterval('P2W');
$period = new DatePeriod($start, $interval, 3);

foreach ($period as $nextDateTime) {
    echo $nextDateTime->format('Y-m-d H:i:s'), PHP_EOL;
}
```

DatePeriod类构造方法的第四个参数是可选的，用于显式指定周期的结束日期和时间。如果迭代时想排除起始日期和时间，可以把构造方法的最后一个参数设为DatePeriod::EXCLUDE_START_DATE常量（如示例5-17所示）。

示例5-17：使用选项设置DatePeriod类

```
<?php
$start = new DateTime();
$interval = new DateInterval('P2W');
$period = new DatePeriod(
    $start,
    $interval,
    3,
    DatePeriod::EXCLUDE_START_DATE
);

foreach ($period as $nextDateTime) {
    echo $nextDateTime->format('Y-m-d H:i:s'), PHP_EOL;
}
```

nesbot/carbon组件

如果经常需要处理日期和时间，应该使用布莱恩·内斯比特开发的`nesbot/carbon`组件（*https://github.com/briannesbitt/Carbon*）。Carbon提供了一个简单的接口，有很多处理日期和时间值的有用方法。

数据库

PHP应用可以在很多种数据库中持久保存信息，例如MySQL、PostgreSQL、SQLite和Oracle。这些数据库都提供了用于在PHP和数据库之间通信的扩展。例如，MySQL使用的是`mysqli`扩展，这个扩展向PHP语言添加了很多`mysqli_*()`函数；SQLite使用的是`SQLite3`扩展，这个扩展向PHP语言添加了`SQLite3`、`SQLite3Stmt`和`SQLite3Result`类。如果在项目中使用多种数据库，需要安装并学习多种PHP数据库扩展和接口。这增加了认知和技术负担。

PDO扩展

正是基于这个原因，PHP原生提供了PDO扩展。PDO（PHP Data Objects的简称，意思是“PHP数据对象”）是一系列PHP类，抽象了不同数据库的具体实现，只通过一个用户界面就能与多种不同的SQL数据库通信。不管使用哪种数据库系统，使用一个接口就能编写和执行数据库查询。

警告： 虽然PDO扩展为不同数据库提供了统一接口，但是我们仍然必须自己编写SQL语句。这是PDO的劣势所在。各种数据库都会提供专属的特性，而这些特性通常需要独特的SQL句法。我建议使用PDO时编写符合ANSI/ISO标准的SQL语句，这样如果更换数据库系统，SQL语句不会失效。如果确实必须使用专属的数据库特性，记住，更换数据库系统时要更新SQL语句。

数据库连接和DSN

首先，我们要选择最适合应用的数据库系统；然后，安装数据库，创建模式，还可以加载初始的数据集；最后，在PHP中实例化`PDO`类。`PDO`实例的作用是把PHP和数据库连接起来。

`PDO`类的构造方法有一个字符串参数，用于指定DSN（Data Source Name的简称，意思是“数据源名称”），提供数据库连接的详细信息。DSN的开头是数据库驱动器的名称（例如`mysql`或`sqlite`），后面跟着:符号，然后是剩下的内容。不同数据库使用的DSN连接字符串有所不同，不过一般都包含以下信息：

- 主机名或IP地址。
- 端口号。
- 数据库名。
- 字符集。

注意： 各种数据库使用的DNS格式参见*http://php.net/manual/pdo.drivers.php*。

PDO类构造方法的第二个和第三个参数分别是数据库的用户名和密码。如果数据库需要认证，要提供这两个参数。

示例5-18使用PDO连接到了一个名为acme的MySQL数据库。这个数据库的IP地址是127.0.0.1，监听的是MySQL使用的标准端口3306。这个数据库的用户名是josh，密码是sekrit。连接使用的字符集是utf8。

示例5-18：PDO类的构造方法

```
<?php
try {
    $pdo = new PDO(
        'mysql:host=127.0.0.1;dbname=books;port=3306;charset=utf8',
        'USERNAME',
        'PASSWORD'
    );
} catch (PDOException $e) {
    // 连接数据库失败
    echo "Database connection failed";
    exit;
}
```

PDO类构造方法的第一个参数是DSN。这个DSN以mysql:开头，因此PDO会使用PDO扩展中的MySQL驱动器连接MySQL数据库。在:符号之后，我们指定了几个使用分号分开的键值对，设置host、dbname、port和charset。

建议： 如果连接数据库失败，PDO构造方法会抛出PDOException异常。创建PDO连接时要预期会出现这种异常，并捕获这种异常。

保证数据库凭据的安全

示例5-18只是用于演示，这么做并不安全。绝对不能在PHP文件中硬编码数据库凭据，尤其不能在可公开访问的PHP文件中这么做。如果由于缺陷或服务器配置出错，让HTTP客户端看到了原始的PHP代码，那么数据库凭据就暴露了，所有人都能看到。我们应该

把数据库凭据保存在一个位于文档根目录之外的配置文件中，然后在需要使用凭据的PHP文件中导入。

建议： 凭据也不能纳入版本控制。我们要使用*.gitignore*文件保护凭据。否则，机密凭据会公开出现在代码仓库中，别人就能看到。如果使用的是公开仓库，后果更严重。

在下面这个例子中，我们在*settings.php*文件中保存数据库连接凭据。这个文件保存在项目的根目录中，但在文档根目录之外。*index.php*文件保存在文档根目录中，通过Web服务器可以公开访问。*index.php*文件使用了保存在*settings.php*文件中的凭据。

```
[项目根目录]
    settings.php
    public_html/ <-- 文档根目录
        index.php
```

*settings.php*文件的内容如下：

```
<?php
$settings = [
    'host' => '127.0.0.1',
    'port' => '3306',
    'name' => 'acme',
    'username' => 'USERNAME',
    'password' => 'PASSWORD',
    'charset' => 'utf8'
];
```

示例5-19是*index.php*文件的内容。这个文件导入了*settings.php*文件，然后建立了一个PDO数据库连接。

示例5-19：在PDO构造方法中使用外部设置

```
<?php
include('../settings.php');

$pdo = new PDO(
    sprintf(
        'mysql:host=%s;dbname=%s;port=%s;charset=%s',
        $settings['host'],
        $settings['name'],
        $settings['port'],
        $settings['charset']
    ),
    $settings['username'],
    $settings['password']
);
```

这样做更安全。如果*index.php*文件泄露了，数据库凭据仍然安全。

预处理语句

现在我们建立了到一个数据库的PDO连接，通过这个连接可以使用SQL语句从数据库中读取数据，或者把数据写入数据库。不过这还不算完事。开发PHP应用时，我经常需要使用从当前HTTP请求中获取的动态信息定制SQL语句。例如，使用/*user?email=john@example.com*这个URL显示具体用户账号的资料信息。这个URL使用的SQL语句可能是：

```
SELECT id FROM users WHERE email = "john@example.com";
```

初级PHP开发者可能会像下面这样构建这个SQL语句：

```
$sql = sprintf(
    'SELECT id FROM users WHERE email = "%s"',
    filter_input(INPUT_GET, 'email')
);
```

这么做不好，因为SQL语句使用了HTTP请求查询字符串中的原始输入数据。这么做等于是为黑客打开了大门，让他们对你的PHP应用做坏事。你听说过Bobby Tables的故事没（*http://xkcd.com/327/*）？在SQL语句中使用用户的输入时，一定要过滤。我们很幸运，PDO扩展通过预处理语句和参数绑定把过滤输入这项操作变得特别简单。

预处理语句是PDOStatement实例。不过，我很少直接实例化PDOStatement类，而是通过PDO实例的`prepare()`方法获取预处理语句对象。这个方法的第一个参数是一个SQL语句字符串，返回值是一个PDOStatement实例：

```
<?php
$sql = 'SELECT id FROM users WHERE email = :email';
$statement = $pdo->prepare($sql);
```

注意，在这个SQL语句中，`:email`是具名占位符，可以安全地绑定任何值。在示例5-20中，我在`$statement`实例上调用`bindValue()`方法，把HTTP请求查询字符串的值绑定到`:email`占位符上。

示例5-20：在预处理语句上绑定电子邮件地址

```
<?php
$sql = 'SELECT id FROM users WHERE email = :email';
$statement = $pdo->prepare($sql);

$email = filter_input(INPUT_GET, 'email');
$statement->bindValue(':email', $email);
```

预处理语句会自动过滤`$email`的值，防止数据库受到SQL注入攻击。一个SQL语句字符串中可以有多个具名占位符，然后在预处理语句上调用`bindValue()`方法绑定各个占位符的值。

在示例5-20中，具名占位符:email的值是字符串。如果修改SQL语句，想使用数值ID查找用户该怎么办呢？此时，我们必须向预处理语句的bindValue()方法传入第三个参数，指定占位符要绑定的数据是什么类型。如果不传入第三个参数，预处理语句假定要绑定的数据是字符串。

示例5-21对示例5-20做了修改，现在我们不使用电子邮件地址查找用户，而是使用数值ID。数值ID从HTTP查询字符串中名为id的参数中获取。

示例5-21：在预处理语句上绑定ID

```
<?php
$sql = 'SELECT email FROM users WHERE id = :id';
$statement = $pdo->prepare($sql);

$userId = filter_input(INPUT_GET, 'id');
$statement->bindValue(':id', $userId, PDO::PARAM_INT);
```

在这个示例中，bindValue()方法的第三个参数是PDO::PARAM_INT常量，告诉PDO要绑定的数据是整数。指定数据类型的PDO常量还有：

```
PDO::PARAM_BOOL
PDO::PARAM_NULL
PDO::PARAM_INT
PDO::PARAM_STR （默认值）
```

注意： 全部PDO常量参见*http://php.net/manual/pdo.constants.php*。

查询结果

现在有了预处理语句，可以在数据库中执行SQL查询了。调用预处理语句的execute()方法后会使用绑定的所有数据执行SQL语句。如果执行的是INSERT、UPDATE或DELETE语句，调用execute()方法后工作就结束了。如果执行的是SELECT语句，我们可能期望数据库能返回匹配的记录。我们可以调用预处理语句的fetch()、fetchAll()、fetchColumn()和fetchObject()方法，获取查询结果。

PDOStatement实例的fetch()方法用于获取结果集合中的下一行。我会使用这个方法迭代大型结果集合，如果可用内存放不下整个结果集合，特别适合使用这个方法。

示例5-22：把预处理语句获取的结果当成关联数组处理

```
<?php
// 构建并执行SQL查询
$sql = 'SELECT id, email FROM users WHERE email = :email';
$statement = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$statement->bindValue(':email', $email, PDO::PARAM_INT);
```

```
$statement->execute();

// 迭代结果
while (($result = $statement->fetch(PDO::FETCH_ASSOC)) !== false) {
    echo $result['email'];
}
```

在这个示例中，在预处理语句实例上调用fetch()方法时，我把这个方法的第一个参数设为PDO::FETCH_ASSOC常量。这个参数决定fetch()和fetchAll()方法如何返回查询结果。可以使用的常量如下：

PDO::FETCH_ASSOC
: 让fetch()和fetchAll()方法返回一个关联数组。数组的键是数据库的列名。

PDO::FETCH_NUM
: 让fetch()和fetchAll()方法返回一个键为数字的数组。数组的键是数据库列在查询结果中的索引。

PDO::FETCH_BOTH
: 让fetch()和fetchAll()方法返回一个既有键为列名又有键为数字的数组。这等于是把PDO::FETCH_ASSOC和PDO::FETCH_NUM合并。

PDO::FETCH_OBJ
: 让fetch()和fetchAll()方法返回一个对象，对象的属性是数据库的列名。

注意：获取PDO语句结果的详细说明参见*http://php.net/manual/pdostatement.fetch.php*。

如果处理的是小型结果集合，可以使用预处理语句的fetchAll()方法获取所有查询结果（如示例5-23所示）。除非十分确定可用内存能放得下整个查询结果，否则，我通常不建议使用这个方法。

示例5-23：让预处理语句获取所有结果，把结果保存到关联数组中

```
<?php
// 构建并执行SQL查询
$sql = 'SELECT id, email FROM users WHERE email = :email';
$statement = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$statement->bindValue(':email', $email, PDO::PARAM_INT);
$statement->execute();

// 迭代结果
$results = $statement->fetchAll(PDO::FETCH_ASSOC);
foreach ($results as $result) {
    echo $result['email'];
}
```

如果只关心查询结果中的一列，可以使用预处理语句的`fetchColumn()`方法。这个方法的作用与`fetch()`方法类似，返回查询结果中下一行的某一列（如示例5-24所示）。`fetchColumn()`方法只有一个参数，用于指定所需列的索引。

建议： 查询结果中列的顺序与SQL查询语句中指定的列顺序一致。

示例5-24：让预处理语句获取一列，且一次获取一行，把结果保存到关联数组中

```
<?php
// 构建并执行SQL查询
$sql = 'SELECT id, email FROM users WHERE email = :email';
$statement = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$statement->bindValue(':email', $email, PDO::PARAM_INT);
$statement->execute();

// 迭代结果
while (($email = $statement->fetchColumn(1)) !== false) {
    echo $email;
}
```

在示例5-24中，`email`列出现在SQL查询语句的第二位，因此在查询结果的行中是第二列，所以我传入`fetchColumn()`方法的参数是数字1（列的索引从零开始）。

我们还可以使用预处理语句的`fetchObject()`方法获取查询结果中的行，这个方法把行当成对象，对象的属性是SQL查询结果中的列（如示例5-25所示）。

示例5-25：把预处理语句获取的行当成对象

```
<?php
// 构建并执行SQL查询
$sql = 'SELECT id, email FROM users WHERE email = :email';
$statement = $pdo->prepare($sql);
$email = filter_input(INPUT_GET, 'email');
$statement->bindValue(':email', $email, PDO::PARAM_INT);
$statement->execute();

// 迭代结果
while (($result = $statement->fetchObject()) !== false) {
    echo $result->name;
}
```

事务

PDO扩展还支持事务。事务是指把一系列数据库语句当成单个逻辑单元（具有原子性）执行。也就是说，事务中的一系列SQL查询要么都成功执行，要么根本不执行。事务的原子性能保证数据的一致性、安全性和持久性。事务还有个很好的副作用——提升性能，因为事务其实是把多个查询排成队列，一次全部执行。

注意： 不是所有数据库都支持事务。详细信息参见数据库的文档和相应的PHP PDO驱动器。

在PDO扩展中使用事务很容易。构建和执行SQL语句的方式完全和示例5-25展示的一样，不过唯有一处区别：要把想执行的SQL语句放在PDO实例的`beginTransaction()`方法和`commit()`方法之间。`beginTransaction()`方法的作用是让PDO把后续SQL查询语句排入队列，而不是立即执行这些SQL语句。`commit()`方法的作用是执行原子事务队列中的查询。如果事务中有一个查询失败了，事务中的所有查询都无效。记住，事务中的SQL查询要么都成功执行，要么根本不执行。

原子性对数据完整性至关重要。下面举个处理银行账户交易的例子。这段代码可以把钱存入账户，如果账户中有足够的余额，也可以取钱。示例5-26从一个账户中转50美元到另一个账户，没有使用数据库事务。

示例5-26：执行数据库查询时没使用事务

```
<?php
require 'settings.php';

// PDO 连接
try {
    $pdo = new PDO(
        sprintf(
            'mysql:host=%s;dbname=%s;port=%s;charset=%s',
            $settings['host'],
            $settings['name'],
            $settings['port'],
            $settings['charset']
        ),
        $settings['username'],
        $settings['password']
    );
} catch (PDOException $e) {
    // 连接数据库失败
    echo "Database connection failed";
    exit;
}

// 查询语句
$stmtSubtract = $pdo->prepare('
    UPDATE accounts
    SET amount = amount - :amount
    WHERE name = :name
');
$stmtAdd = $pdo->prepare('
    UPDATE accounts
    SET amount = amount + :amount
    WHERE name = :name
');

// 从账户1中取钱
$fromAccount = 'Checking';
```

```
$withdrawal = 50;
$stmtSubtract->bindParam(':name', $fromAccount);
$stmtSubtract->bindParam(':amount', $withDrawal, PDO::PARAM_INT);
$stmtSubtract->execute();

// 把钱存入账户2
$toAccount = 'Savings';
$deposit = 50;
$stmtAdd->bindParam(':name', $toAccount);
$stmtAdd->bindParam(':amount', $deposit, PDO::PARAM_INT);
$stmtAdd->execute();
```

这么写看起来可以，其实不然。如果从账户1中取了50美元后，还没把50美元存入账户2，此时服务器突然停机了怎么办？主机商可能会出现电力故障，受到火灾或水灾等灾难的侵害。从账户1中取出的这50美元怎么办？这50美元不会存入账户2，而是就这么消失了。我们可以使用数据库事务保证数据的完整性（如示例5-27所示）。

示例5-27：使用事务执行数据库查询

```
<?php
require 'settings.php';

// PDO连接
try {
    $pdo = new PDO(
        sprintf(
            'mysql:host=%s;dbname=%s;port=%s;charset=%s',
            $settings['host'],
            $settings['name'],
            $settings['port'],
            $settings['charset']
        ),
        $settings['username'],
        $settings['password']
    );
} catch (PDOException $e) {
    // 连接数据库失败
    echo "Database connection failed";
    exit;
}

// 查询语句
$stmtSubtract = $pdo->prepare('
    UPDATE accounts
    SET amount = amount - :amount
    WHERE name = :name
');
$stmtAdd = $pdo->prepare('
    UPDATE accounts
    SET amount = amount + :amount
    WHERE name = :name
');

// 开始事务
```

```
$pdo->beginTransaction();

// 从账户1中取钱
$fromAccount = 'Checking';
$withdrawal = 50;
$stmtSubtract->bindParam(':name', $fromAccount);
$stmtSubtract->bindParam(':amount', $withDrawal, PDO::PARAM_INT);
$stmtSubtract->execute();

// 把钱存入账户2
$toAccount = 'Savings';
$deposit = 50;
$stmtAdd->bindParam(':name', $toAccount);
$stmtAdd->bindParam(':amount', $deposit, PDO::PARAM_INT);
$stmtAdd->execute();

// 提交事务
$pdo->commit();
```

示例 5-27 在一个数据库事务中处理取钱和存钱操作，这样能保证两个操作都成功或都失败，从而保持数据的一致性。

多字节字符串

PHP假设字符串中的每个字符都是八位字符，占一个字节的内存。可是，这种假设很天真，一旦处理非英文字符就失效了。你可能要为多国用户本地化PHP应用，你的博客中有些评论可能是使用西班牙语、德语或挪威语写的，用户的名字可能包含重音字符。总之，你经常会遇到多字节字符，因此必须充分考虑到这种情况。

我所说的多字节字符是指，不在传统的128个ASCII字符集中的字符，例如ñ、ë、â、ô、à、æ和ø，除此之外还有很多。PHP中处理字符串的函数默认假设所有字符串都只使用8位字符，如果使用这些PHP原生的字符串函数处理包含多字节字符的Unicode字符串，会得到意料之外的错误结果。

注意：Unicode是国际标准，为很多不同语言中的每个字符都指定了唯一的数值。这个标准由Unicode联盟（*http://www.unicode.org/*）维护。

为了避免处理多字节字符串时出错，可以安装`mbstring`扩展（*http://php.net/manual/book.mbstring.php*）。这个扩展提供了知道如何处理多字节字符串的函数，能替代大多数PHP原生的处理字符串的函数。例如，知道如何处理多字节字符串的`mb_strlen()`函数用于替代PHP原生的`strlen()`函数。

时至今日，我仍在训练自己使用`mbstring`扩展提供的多字节字符串函数替代PHP 默认的

字符串函数。这个习惯很难养成，但是处理Unicode字符串时必须使用这些能处理多字节字符串的函数，否则很容易损坏多字节Unicode数据。

建议： 我会使用Iñtërnâtiônàlizætiøn这个字符串测试我的PHP应用是否支持多字节字符。

字符编码

如果读完本节你只能记住一条建议，应该是使用UTF-8。所有现代的Web浏览器都能理解UTF-8字符编码。字符编码是打包Unicode数据的方式，以便把数据存储在内存中，或者通过线缆在服务器和客户端之间传输。UTF-8只是很多可用字符编码中的一种。不过，UTF-8是最流行的字符编码，所有现代的Web浏览器都支持。

建议： 解说Unicode和UTF-8

> 汤姆·斯科特对Unicode和UTF-8的解说（*http://bit.ly/ts-unicode*）是我见过最好的。乔尔·斯波尔斯基在他的网站中也写了一篇文章（*http://bit.ly/jspolsky*），很好地解说了字符编码。

对很多开发者来说，字符编码很复杂，而且不易理解。处理多字节字符串时，要记住以下建议：

1. 一定要知道数据的字符编码。
2. 使用UTF-8字符编码存储数据。
3. 使用UTF-8字符编码输出数据。

mbstring扩展不仅能处理Unicode字符串，还能在不同的字符编码之间转换多字节字符串。如果客户使用Windows专用的字符编码导出Excel电子表格，而我真正需要的是UTF-8编码的数据，此时就用得到这种功能。使用mb_detect_encoding()和mb_convert_encoding()函数可以把Unicode字符串从一种字符编码转换成另一种字符编码。

输出UTF-8数据

处理多字节字符时，一定要告诉PHP使用UTF-8字符编码。在*php.ini*文件中设置最简单，如下所示：

```
default_charset = "UTF-8";
```

很多PHP函数都会使用这个默认的字符集，例如htmlentities()、html_entity_decode()和htmlspecialchars()，以及mbstring扩展提供的函数。如果没像下面这样使

用header()函数明确指定字符集，在PHP返回的响应中，Content-Type首部默认也使用这个默认值：

```
<?php
header('Content-Type: application/json;charset=utf-8');
```

警告： 只要PHP已经返回了输出，就不能使用header()函数。

我还建议在HTML文档的头部加入下述meta标签：

```
<meta charset="UTF-8"/>
```

流

在现代的PHP特性中，流或许是最出色但最少使用的。虽然PHP 4.3.0就引入了流，但是很多开发者不知道流的存在，因为人们很少提及流，而且流的文档也匮乏。

> 流在PHP 4.3.0中引入，作用是使用统一的方式处理文件、网络和数据压缩等共用同一套函数和用法的操作。简单而言，流是具有流式行为的资源对象。因此，流可以线性读写，或许还能使用fseek()函数定位到流中的任何位置。
>
> ——PHP手册

这段定义很难理解是吧，下面我们简化一下，以便更易于理解。流的作用是在出发地和目的地之间传输数据。就这么简单。出发地和目的地可以是文件、命令行进程、网络连接、ZIP或TAR压缩文件、临时内存、标准输入或输出，或者是通过PHP流封装协议（*http://php.net/manual/wrappers.php*）实现的任何其他资源。

如果你读写过文件，就使用过流；如果你从php://stdin读取过数据，或者把数据写入过php://stdout，就使用过流。流为PHP的很多IO函数提供了底层实现，例如file_get_contents()、fopen()、fgets()和fwrite()。PHP的流函数提供了处理不同流资源（出发地和目的地）的统一接口。

警告： 我把流理解为管道，相当于把水从一个地方引到另一个地方。在水从出发地流到目的地的过程中，我们可以过滤水，可以改变水质，可以添加水，也可以排出水（提示：水是数据的隐喻）。

流封装协议

流式数据的种类各异，每种类型需要独特的协议，以便读写数据。我们称这些协议为流

封装协议（*http://php.net/manual/wrappers.php*）。例如，我们可以读写文件系统，可以通过HTTP、HTTPS或SSH（安全的shell）与远程Web服务器通信，还可以打开并读写ZIP、RAR或PHAR压缩文件。这些通信方式都包含下述相同的过程：

1. 开始通信。
2. 读取数据。
3. 写入数据。
4. 结束通信。

虽然过程是一样的，但是读写文件系统中文件的方式与收发HTTP消息的方式有所不同。流封装协议的作用是使用通用的接口封装这种差异。

每个流都有一个协议和一个目标。指定协议和目标的方法是使用流标识符，其格式如下所示，我们对此已经熟悉了：

```
<scheme>://<target>
```

其中，`<scheme>`是流的封装协议，`<target>`是流的数据源。示例5-28使用HTTP流封装协议创建了一个与Flickr API通信的PHP流。

示例5-28：使用HTTP流封装协议与Flickr API通信

```
<?php
$json = file_get_contents(
    'http://api.flickr.com/services/feeds/photos_public.gne?format=json'
);
```

不要误以为这是普通的网页URL，`file_get_contents()`函数的字符串参数其实是一个流标识符。`http`协议会让PHP使用HTTP流封装协议。在这个参数中，`http`之后是流的目标。流的目标之所以看起来像是普通的网页URL，是因为HTTP流封装协议就是这样规定的。其他流封装协议可能不是这样。

注意： 前面一段要多读几遍，直到熟记为止。很多PHP开发者不知道普通的URL其实是PHP流封装协议标识符的伪装。

---

file://流封装协议

我们使用`file_get_contents()`、`fopen()`、`fwrite()`和`fclose()`函数读写文件系统。因为PHP默认使用的流封装协议是`file://`，所以我们很少认为这些函数使用的是PHP流。我们在不经意间就使用了PHP流！示例5-29使用`file://`流封装协议创建了一个读写*/etc/hosts*文件的流。

示例5-29：隐式使用file://流封装协议

```
<?php
$handle = fopen('/etc/hosts', 'rb');
while (feof($handle) !== true) {
    echo fgets($handle);
}
fclose($handle);
```

示例5-30的作用一样，不过这一次我们在流标识符中明确指定了file://流封装协议。

示例5-30：显式使用file://流封装协议

```
<?php
$handle = fopen('file:///etc/hosts', 'rb');
while (feof($handle) !== true) {
    echo fgets($handle);
}
fclose($handle);
```

我们通常会省略file://封装协议，因为这是PHP使用的默认值。

php://流封装协议

编写命令行脚本的PHP开发者会感激php://流封装协议。这个流封装协议的作用是与PHP脚本的标准输入、标准输出和标准错误文件描述符通信。我们可以使用PHP提供的文件系统函数打开、读取或写入下述四个流：

php://stdin
: 这是个只读PHP流，其中的数据来自标准输入。例如，PHP脚本可以使用这个流接收命令行传入脚本的信息。

php://stdout
: 这个PHP流的作用是把数据写入当前的输出缓冲区。这个流只能写，无法读或寻址。

php://memory
: 这个PHP流的作用是从系统内存中读取数据，或者把数据写入系统内存。这个PHP流的缺点是，可用内存是有限的。使用php://temp流更安全。

php://temp
: 这个PHP流的作用和php://memory类似，不过，没有可用内存时，PHP会把数据写入临时文件。

其他流封装协议

PHP和PHP扩展还提供了很多其他流封装协议，例如，与ZIP和TAR压缩文件、FTP

服务器、数据压缩库、亚马逊API等通信的流封装协议。开发者经常误以为PHP中的fopen()、fgets()、fputs()、feof()和fclose()等文件系统函数只能用来处理文件系统中的文件。事实并非如此。PHP的文件系统函数能在所有支持这些函数的流封装协议中使用。例如，我们可以使用fopen()、fgets()、fputs()、feof()和fclose()函数处理ZIP压缩文件和亚马逊S3服务（通过自定义的S3封装协议，*http://bit.ly/streamwrap*），甚至还能处理Dropbox中的文件（通过自定义的Dropbox封装协议，*http://www.dropbox-php.com/*）。

注意： 关于php://流封装协议的更多信息，请查看PHP的网站（*http://bit.ly/s-wrapper*）。

自定义流封装协议

我们还可以自己编写PHP流封装协议。PHP提供了一个示例streamWrapper类，演示如何编写自定义的流封装协议，支持部分或全部PHP文件系统函数。关于如何编写自定义的PHP流封装协议，更多信息参见：

- *http://php.net/manual/class.streamwrapper.php*
- *http://php.net/manual/stream.streamwrapper.example-1.php*

流上下文

有些PHP流能接受一系列可选的参数，这些参数叫流上下文，用于定制流的行为。不同的流封装协议使用的上下文参数有所不同。流上下文使用stream_context_create()函数创建。这个函数返回的上下文对象可以传入大多数文件系统和流函数。

例如，你知道可以使用file_get_contents()函数发送HTTP POST请求吗？如果想这么做，可以使用一个流上下文对象（如示例5-31所示）。

示例5-31：流上下文

```
<?php
$requestBody = '{"username":"josh"}';
$context = stream_context_create(array(
    'http' => array(
        'method' => 'POST',
        'header' => "Content-Type: application/json;charset=utf-8;\r\n" .
                    "Content-Length: " . mb_strlen($requestBody),
        'content' => $requestBody
    )
));
$response = file_get_contents('https://my-api.com/users', false, $context);
```

流上下文是个关联数组，最外层键是流封装协议的名称。流上下文数组中的值针对每个具体的流封装协议。可用的设置参见各个PHP流封装协议的文档。

流过滤器

目前为止我们讨论了如何打开流，从流中读取数据，以及把数据写入流。可是，PHP流真正强大的地方在于过滤、转换、添加或删除流中传输的数据。例如，我们可以打开一个流处理Markdown文件，在把文件内容读入内存的过程中自动将其转换成HTML。

注意： PHP内置了几个流过滤器：`string.rot13`、`string.toupper`、`string.tolower`和`string.strip_tags`。这些过滤器没什么用，我们要使用自定义的过滤器。

若想把过滤器附加到现有的流上，要使用`stream_filter_append()`函数。示例5-32从本地文件系统中的文本文件里读取数据时使用了**`string.toupper`**过滤器，目的是把文件中的内容转换成大写字母。我不建议使用这个过滤器，这里只是演示如何把过滤器附加到流上。

示例5-32：演示使用流过滤器string.toupper

```
<?php
$handle = fopen('data.txt', 'rb');
stream_filter_append($handle, 'string.toupper');
while(feof($handle) !== true) {
    echo fgets($handle); // <-- 输出的全是大写字母
}
fclose($handle);
```

我们还可以使用`php://filter`流封装协议把过滤器附加到流上。不过，使用这种方式之前必须先打开PHP流。示例5-33的作用和前一个示例一样，可是这里我们使用`php://filter`方式附加过滤器。

示例5-33：演示使用php://filter附加流过滤器string.toupper

```
<?php
$handle = fopen('php://filter/read=string.toupper/resource=data.txt', 'rb');
while(feof($handle) !== true) {
    echo fgets($handle); // <-- 输出的全是大写字母
}
fclose($handle);
```

我们要特别注意`fopen()`函数的第一个参数。这个参数的值是`php://`流封装协议的流标识符。这个流标识符中的目标如下所示：

```
    filter/read=<filter_name>/resource=<scheme>://<target>
```

这种方式和`stream_filter_append()`函数相比较为繁琐。可是，PHP的某些文件系统函

数在调用后无法附加过滤器，例如file()和fpassthru()。所以，使用这些函数时只能使用php://filter流封装协议附加流过滤器。

下面看个更实际的流过滤器示例。在New Media Campaigns（*http://www.newmediacampaigns.com*），我们内部的内容管理系统会把nginx访问日志保存到rsync.net（*http://rsync.net*）。我们把一天的访问情况保存在一个日志文件中，而且会使用bzip2压缩每个日志文件。日志文件的名称使用*YYYY-MM-DD.log.bz2*格式。领导让我提取过去30天某个域名的访问数据，这听起来有很多事要做，对吧？我要计算日期范围，确定日志文件的名称，通过FTP连接rsync.net，下载文件，解压缩文件，逐行迭代每个文件，把相应的行提取出来，然后把访问数据写入一个输出目标。你可能不相信，使用PHP流，不到20行代码就能做完所有这些事情（如示例5-34所示）。

示例5-34：使用DateTime类和流过滤器迭代bzip压缩的日志文件

```
01 <?php
02 $dateStart = new \DateTime();
03 $dateInterval = \DateInterval::createFromDateString('-1 day');
04 $datePeriod = new \DatePeriod($dateStart, $dateInterval, 30);
05 foreach ($datePeriod as $date) {
06     $file = 'sftp://USER:PASS@rsync.net/' . $date->format('Y-m-d') . '.log.bz2';
07     if (file_exists($file)) {
08         $handle = fopen($file, 'rb');
09         stream_filter_append($handle, 'bzip2.decompress');
10         while (feof($handle) !== true) {
11             $line = fgets($handle);
12             if (strpos($line, 'www.example.com') !== false) {
13                 fwrite(STDOUT, $line);
14             }
15         }
16         fclose($handle);
17     }
18 }
```

在示例5-34中：

- 第2～4行创建一个持续30天的DatePeriod实例，一天一天反向向前推移。
- 第6行使用每次迭代DatePeriod实例得到的DateTime实例创建日志文件的文件名。
- 第8～9行使用SFTP流封装协议打开位于rsync.net上的日志文件流资源。我们把bzip2.decompress流过滤器附加到日志文件流资源上，实时解压缩bzip2格式的日志文件。
- 第10～15行使用PHP原生的文件系统函数迭代解压缩后的日志文件。
- 第12～14行检查各行日志，看访问的是不是指定域名。如果是，把这一行日志写入标准输出。

使用bzip2.decompress流过滤器可以在读取日志文件的同时自动解压缩。除此之外，我们还可以使用shell_exec()或bzdecompress()函数，手动把日志文件解压缩到临时目录中，然后迭代解压缩后的文件，等PHP脚本完成任务后再清理这些解压缩后的文件。不过，使用PHP流更简单，也更优雅。

自定义流过滤器

我们还可以编写自定义的流过滤器。其实，大多数情况下都要使用自定义的流过滤器。自定义的流过滤器是个PHP类，扩展内置的php_user_filter类（*http://php.net/manual/en/class.php-user-filter.php*）。这个类必须实现filter()、onCreate()和onClose()方法。而且，必须使用stream_filter_register()函数注册自定义的流过滤器。

注意：桶排成一排流过来了！

PHP流会把数据分成按次序排列的桶，一个桶中盛放的流数据量是固定的（例如4096字节）。如果还用管道比喻，就是把水放在一个个水桶中，顺着管道从出发地漂流到目的地，在漂流的过程中会经过流过滤器。流过滤器一次能接收并处理一个或多个桶。一定时间内过滤器接收到的桶叫做桶队列。

下面我们自定义一个流过滤器，在把流中的数据读入内存时审查其中的脏字（如示例5-35所示）。首先，我们必须创建一个PHP类，让它扩展php_user_filter类。这个类必须实现filter()方法，这个方法是个筛子，用于过滤流经的桶。这个方法的参数是上游漂来的桶队列，处理过队列中的每个桶对象后，再把桶排成一排，向下游的目的地漂去。我们自定义的DirtyWordsFilter流过滤器如下所示。

建议： 桶队列中的每个桶对象都有两个公开属性：data和datalen。这两个属性的值分别是桶中的内容和内容的长度。

示例5-35：自定义的DirtyWordsFilter流过滤器

```
class DirtyWordsFilter extends php_user_filter
{
    /**
     * @param resource $in       流来的桶队列
     * @param resource $out      流走的桶队列
     * @param int      $consumed 处理的字节数
     * @param bool     $closing  是流中最后一个桶队列吗？
     */
    public function filter($in, $out, &$consumed, $closing)
    {
        $words = array('grime', 'dirt', 'grease');
        $wordData = array();
        foreach ($words as $word) {
```

```
            $replacement = array_fill(0, mb_strlen($word), '*');
            $wordData[$word] = implode('', $replacement);
        }
        $bad = array_keys($wordData);
        $good = array_values($wordData);

        // 迭代流来的桶队列中的每个桶
        while ($bucket = stream_bucket_make_writeable($in)) {
            // 审查桶数据中的脏字
            $bucket->data = str_replace($bad, $good, $bucket->data);

            // 增加已处理的数据量
            $consumed += $bucket->datalen;

            // 把桶放入流向下游的队列中
            stream_bucket_append($out, $bucket);
        }

        return PSFS_PASS_ON;
    }
}
```

filter()方法的作用是接收、处理再转运桶中的流数据。在filter()方法中，我们迭代桶队列$in中的桶，把脏字替换成审查后的值。这个方法的返回值是PSFS_PASS_ON常量，表示操作成功。这个方法接收四个参数：

$in
: 上游流来的一个队列，有一个或多个桶，桶中是从出发地流来的数据。

$out
: 由一个桶或多个桶组成的队列，流向下游的流目的地。

&$consumed
: 自定义的过滤器处理的流数据总字节数。

$closing
: filter()方法接收的是最后一个桶队列吗？

然后，我们必须使用stream_filter_register()函数注册这个自定义的DirtWordsFilter流过滤器（如示例5-36所示）。

示例5-36：注册自定义的DirtWordsFilter流过滤器

```
<?php
stream_filter_register('dirty_words_filter', 'DirtyWordsFilter');
```

第一个参数是用于识别这个自定义过滤器的过滤器名，第二个参数是这个自定义过滤器的类名。现在可以使用这个自定义的流过滤器了（如示例5-37所示）。

示例5-37：使用DirtWordsFilter流过滤器

```
<?php
$handle = fopen('data.txt', 'rb');
stream_filter_append($handle, 'dirty_words_filter');
while (feof($handle) !== true) {
    echo fgets($handle); // <-- 输出审查后的文本
}
fclose($handle);
```

建议：如果想进一步学习PHP流，请观看Nomad PHP网站中伊丽莎白·史密斯的演讲（*http://bit.ly/nomad-php*）。这个视频不是免费的，但值那个价。你还可以阅读PHP文档（*http://php.net/manual/en/book.stream.php*），进一步学习PHP流。

错误和异常

事情会出错，这是不争的事实。不管我们多么努力，在项目中倾注多少时间，总会有忽略的缺陷和错误存在。例如，你用过的PHP应用是不是显示过空白页面？你访问PHP开发的网站时是不是见过难以理解的堆栈跟踪？出现这些不幸的状况，说明应用有错误或未捕获的异常。

错误和异常是强大的工具，能帮助我们预期意料之外的事，使用优雅的方式捕获问题和不足。不过，错误和异常之间很相似，容易让人混淆。错误和异常都表明出问题了，都会提供错误消息，而且都有错误类型。然而，错误出现的时间比异常早。错误会导致程序脚本停止执行，如果可能，错误会委托给全局错误处理程序处理。有些错误是无法恢复的。如今我们基本上只需处理异常，不用管错误，但我们仍然必须做好防御准备，因为PHP中很多较旧的函数（例如`fopen()`）遇到问题时仍会触发错误。

注意：我们可以在可能会触发错误的函数前加上@符号（例如`@fopen()`），不让PHP触发错误。这有悖于常规做法，我建议不要这么做，我们应该修改代码。

异常是PHP的错误处理系统向面向对象演进后得到的产物。异常要先实例化，然后抛出，最后再捕获。异常是预期并处理问题更为灵活的方式，可以就地处理，无需停止执行脚本。异常进可攻退可守。我们必须使用`try {} catch {}`代码块预测第三方厂商的代码可能抛出的异常。我们还可以主动出击，抛出异常，把我们不知道如何处理的特定情况交给其他开发者处理。

异常

异常是`Exception`类的对象，在遇到无法修复的状况时抛出（例如，远程API无响应，数

据库查询失败，或者无法满足前置条件）。我称这些状况为异常状况。出现问题时，异常用于主动出击，委托职责；异常还可用于防守，预测潜在的问题，减轻其影响。

Exception对象与其他任何PHP对象一样，使用new关键字实例化。Exception对象有两个主要的属性：一个是消息，另一个是数字代码。消息用于描述出现的问题；数字代码是可选的，用于为指定的异常提供上下文。实例化Exception对象时可以像下面这样设定消息和可选的数字代码：

```
<?php
$exception = new Exception('Danger, Will Robinson!', 100);
```

我们可以使用公开的实例方法getCode()和getMessage()获取Exception对象的这两个属性，如下所示：

```
<?php
$code = $exception->getCode(); // 100
$message = $exception->getMessage(); // 'Danger...'
```

抛出异常

实例化时可以把异常赋值给变量，不过一定要把异常抛出。如果你编写的代码是提供给其他开发者使用的，遇到异常状况时要主动出击，也就是说，如果代码遇到了异常状况，或者在当前条件下无法操作，要抛出异常。PHP组件和框架的作者尤其无法确定如何处理异常状况，因此要抛出异常，把异常委托给使用组件和框架的开发者处理。

抛出异常后代码会立即停止执行，后续的PHP代码都不会运行。抛出异常的方式是使用throw关键字，后面跟着要抛出的Exception实例：

```
<?php
throw new Exception('Something went wrong. Time for lunch!');
```

我们必须抛出Exception类（或其子类）的实例。PHP内置的异常类和其子类如下：

- Exception（*http://php.net/manual/class.exception.php*）
- ErrorException（*http://php.net/manual/class.errorexception.php*）

PHP标准库（*http://php.net/manual/book.spl.php*）提供了下述额外的Exception子类，扩充了PHP内置的异常类：

- LogicException（*http://php.net/manual/class.logicexception.php*）
  - BadFunctionCallException（*http://php.net/manual/class.badfunctioncallexception.php*）

  - `BadMethodCallException`（*http://php.net/manual/class.badmethodcallexception.php*）
  - `DomainException`（*http://php.net/manual/class.domainexception.php*）
  - `InvalidArgumentException`（*http://php.net/manual/class.invalidargumentexception.php*）
  - `LengthException`（*http://php.net/manual/class.lengthexception.php*）
  - `OutOfRangeException`（*http://php.net/manual/class.outofrangeexception.php*）
- `RuntimeException`（*http://php.net/manual/class.runtimeexception.php*）
  - `OutOfBoundsException`（*http://php.net/manual/class.outofboundsexception.php*）
  - `OverflowException`（http://php.net/manual/class.overflowexception.php）
  - `RangeException`（*http://php.net/manual/class.rangeexception.php*）
  - `UnderflowException`（*http://php.net/manual/class.underflowexception.php*）
  - `UnexpectedValueException`（*http://php.net/manual/class.unexpectedvalueexception.php*）

各个子类针对特定的状况，而且提供了上下文，说明为什么抛出异常。例如，如果PHP组件中的方法应该使用至少有五个字符的字符串参数，但是传入的字符串只有两个字符，那么这个方法可以抛出`InvalidArgumentException`实例。PHP中的异常是类，因此可以轻易扩展`Exception`类，使用定制的属性和方法创建自定义的异常子类。使用哪个异常子类由主观决定，不过选择或创建的异常子类要能最好地回答“为什么抛出异常”这个问题，而且还要说明为什么这么选择。

捕获异常

我们应该捕获抛出的异常，然后使用优雅的方式处理。使用其他开发者编写的PHP组件和框架时必须做好防范。好的PHP组件和框架会在文档中说明什么时候以及什么情况下会抛出异常。预测、捕获并处理异常是我们自己的责任。未捕获的异常会导致PHP应用终止运行，显示致命错误信息。而更糟的是，可能会暴露敏感的调试详细信息，让应用的用户看到。我们都见过这样的错误信息。因此，一定要捕获异常，然后使用优雅的方式处理。

拦截并处理潜在异常的方式是，把可能抛出异常的代码放在`try/catch`块中。在示例5-38中，使用PDO连接数据库失败时会抛出`PDOException`对象。`catch`块会捕获这个异常，然后显示一个友好的错误消息，而不是丑陋的堆栈跟踪。

示例5-38：捕获抛出的异常

```
<?php
try {
    $pdo = new PDO('mysql://host=wrong_host;dbname=wrong_name');
} catch (PDOException $e) {
    // 获取异常的属性，以便输出信息
    $code = $e->getCode();
    $message = $e->getMessage();

    // 给用户显示一个友好的消息
    echo 'Something went wrong. Check back soon, please.';
    exit;
}
```

我们可以使用多个catch块拦截多种异常。如果要使用不同的方式处理抛出的不同异常类型，可以这么做。我们还可以使用一个finally块，在捕获任何类型的异常之后运行一段代码（如示例5-39所示）。

示例5-39：捕获抛出的多个异常

```
<?php
try {
    throw new Exception('Not a PDO exception');
    $pdo = new PDO('mysql://host=wrong_host;dbname=wrong_name');
} catch (PDOException $e) {
    // 处理PDOException异常
    echo "Caught PDO exception";
} catch (Exception $e) {
    // 处理所有其他类型的异常
    echo "Caught generic exception";
} finally {
    // 这里的代码始终都会运行
    echo "Always do this";
}
```

在示例5-39中，第一个catch块会拦截PDOException异常，第二个catch块则会捕获所有其他类型的异常。捕获某种异常时只会运行其中一个catch块。如果PHP没找到适用的catch块，异常会向上冒泡，直到PHP脚本由于致命错误而终止运行。

异常处理程序

你可能会想，应该如何捕获每个可能抛出的异常呢？这是个好问题。PHP允许我们注册一个全局异常处理程序，捕获所有未被捕获的异常。我们一定要设置一个全局异常处理程序。异常处理程序是最后的安全保障，如果没有成功捕获并处理异常，通过这个措施可以给PHP应用的用户显示合适的错误消息。在我自己开发的PHP应用中，我会使用异常处理程序在开发环境中显示调试信息，而在生产环境则显示对用户友好的消息。

异常处理程序是任何可调用的代码。我喜欢使用匿名函数，不过你也可以使用类方法。

不管选择使用什么，异常处理程序都必须接收一个类型为Exception的参数。异常处理程序使用set_exception_handler()函数注册，如下所示：

```
<?php
set_exception_handler(function (Exception $e) {
    // 处理并记录异常
});
```

建议： 我强烈建议在异常处理程序中记录异常。这样，出问题时日志记录器能提醒你，而且还能保存异常细节，供以后查看。

某些情况下，我们可能要使用自定义的异常处理程序替换现有的异常处理程序。代码执行完毕后，PHP会礼貌性地建议你还原现有的异常处理程序。还原成前一个异常处理程序的方式是调用restore_exception_handler()函数（如示例5-40所示）。

示例5-40：设置全局异常处理程序

```
<?php
// 注册异常处理程序
set_exception_handler(function (Exception $e) {
    // 处理并记录异常
});

// 我们编写的其他代码……

// 还原成之前的异常处理程序
restore_exception_handler();
```

错误

除了异常之外，PHP还提供了用于报告错误的函数。这让很多PHP开发者产生了困惑。PHP能触发不同类型的错误，例如致命错误、运行时错误、编译时错误、启动错误和用户触发的错误（很少见）。我们最常见的PHP错误是由句法错误或未捕获的异常导致的。

错误和异常之间的差别很小，如果PHP脚本由于某种原因（例如有句法错误）根本无法运行，通常会触发错误。我们还可以使用trigger_error()函数自己触发错误，然后使用自定义的错误处理程序处理，不过，编写运行在用户空间里的代码时最好使用异常。与错误不同的是，PHP异常可以在PHP应用的任何层级抛出和捕获。异常提供的上下文信息比PHP错误多。而且，我们可以扩展最顶层的Exception类，创建自定义的异常子类。异常加上一个好的日志记录器（例如Monolog）比PHP错误能解决更多的问题。不过，现代的PHP开发者必须能预测并处理PHP错误以及PHP异常。

我们可以使用error_reporting()函数，或者在*php.ini*文件中使用error_reporting指

令，告诉PHP哪些错误要报告，哪些错误要忽略。这两种方式都使用E_*常量确定要报告和忽略哪些错误。

注意： PHP报告错误的详细说明参见*http://php.net/manual/function.error-reporting.php*。

PHP报告错误的方式可以灵敏，也可以迟钝，具体情况由你设定。在开发环境中，我喜欢让PHP显示并记录所有错误消息；而在生产环境中，我会让PHP记录大多数错误消息，但不显示出来。不管怎么做，一定要遵守下述四个规则：

- 一定要让PHP报告错误。
- 在开发环境中要显示错误。
- 在生产环境中不能显示错误。
- 在开发环境和生产环境中都要记录错误。

我在*php.ini*文件中为开发环境设置的错误报告方式如下：

```
;显示错误
display_startup_errors = On
display_errors = On

; 报告所有错误
error_reporting = -1

; 记录错误
log_errors = On
```

我在*php.ini*文件中为生产环境设置的错误报告方式如下：

```
; 不显示错误
display_startup_errors = Off
display_errors = Off

; 除了注意事项之外，报告所有其他错误
error_reporting = E_ALL & ~E_NOTICE

; 记录错误
log_errors = On
```

这两个环境之间的主要区别是，在开发环境执行PHP脚本时要在输出中显示错误，而在生产环境则不显示。不过，在两个环境中我都记录了错误。如果生产环境中的PHP应用有缺陷（我一定不会让这种情况发生），可以查看PHP日志文件中的详情。

错误处理程序

我们可以设置全局异常处理程序，类似地，也可以设置全局错误处理程序，使用自己的

逻辑方式拦截并处理PHP错误。在错误处理程序中可以在终止执行PHP脚本之前清理残局，使用优雅的方式处理错误。

与异常处理程序一样，错误处理程序可以是任何可调用的代码（例如函数或类方法）。我们要在错误处理程序中调用`die()`或`exit()`函数。如果在错误处理程序中不手动终止执行PHP脚本，PHP脚本会从出错的地方继续向下执行。注册全局错误处理程序的方式是使用`set_error_handler()`函数，我们要把一个可调用的参数传入这个方法：

```
<?php
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    // 处理错误
});
```

可调用的错误处理程序接收五个参数：

`$errno`
: 错误等级（对应于一个`E_*`常量）。

`$errstr`
: 错误消息。

`$errfile`
: 发生错误的文件名。

`$errline`
: 发生错误的行号。

`$errcontext`
: 一个数组，指向错误发生时可用的符号表。这个参数是可选的，做高级的调试时才用得到。我一般不用这个参数。

使用自定义的错误处理程序时一定要知道一个重要的注意事项：PHP会把所有错误都交给错误处理程序处理，甚至包括错误报告设置中排除的错误。因此，我们要检查每个错误代码（第一个参数），然后做适当的处理。我们可以通过`set_error_handler()`函数的第二个参数，让错误处理程序只处理一部分错误类型。这个参数的值是使用`E_*`常量组合的位掩码（例如`E_ALL | E_STRICT`）。

我自己以及和我一起开发PHP应用的很多其他开发者都喜欢把PHP错误转换成`ErrorException`对象。`ErrorException`类是`Exception`类的子类，而且是PHP内置的类。因此我可以把PHP错误转换成异常，使用处理异常的现有流程处理错误。

注意：并不是所有错误都能转换成异常！不能转换成异常的错误有：`E_ERROR`、`E_PARSE`、`E_`

CORE_ERROR、E_CORE_WARNING、E_COMPILE_ERROR、E_COMPILE_WARNING和大多数E_STRICT错误。

PHP错误的转换有点棘手，而且我们必须小心，只能转换满足*php.ini*文件中error_reporting指令设置的错误。下面是个错误处理函数示例，在这个函数中，我们把PHP错误转换成了ErrorException对象：

```
<?php
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    if (!(error_reporting() & $errno)) {
        // error_reporting指令没有设置这个错误，所以将其忽略
        return;
    }

    throw new \ErrorException($errstr, $errno, 0, $errfile, $errline);
});
```

这个错误处理函数把适当类型的PHP错误转换成ErrorException对象，然后抛出这种异常，交给现有的异常处理系统处理。当我们自己编写的错误处理代码执行完毕后，还原成之前的错误处理程序（如果有的话）是个好习惯。还原成前一个错误处理程序的方式是调用restore_error_handler()函数（如示例5-41所示）。

示例5-41：设置全局错误处理程序

```
<?php
// 注册错误处理程序
set_error_handler(function ($errno, $errstr, $errfile, $errline) {
    if (!(error_reporting() & $errno)) {
        // error_reporting 指令没有设置这个错误，所以将其忽略
        return;
    }

    throw new ErrorException($errstr, $errno, 0, $errfile, $errline);
});

// 我们编写的其他代码……

// 还原成之前的错误处理程序
restore_error_handler();
```

在开发环境中处理错误和异常

我们知道在开发环境中要显示错误，但是，PHP默认显示的错误消息很难看，而且经常穿插在PHP脚本的输出里，难以阅读。我们可以使用Whoops（*https://github.com/filp/whoops*）改变这种状况。Whoops是个现代的PHP组件，为PHP错误和异常提供了设计精美且易于阅读的诊断页面。Whoops由菲利佩·多布雷拉（*https://github.com/filp*）和丹尼斯·索科洛夫（*https://github.com/denis-sokolov*）开发并维护，界面如图5-1所示。

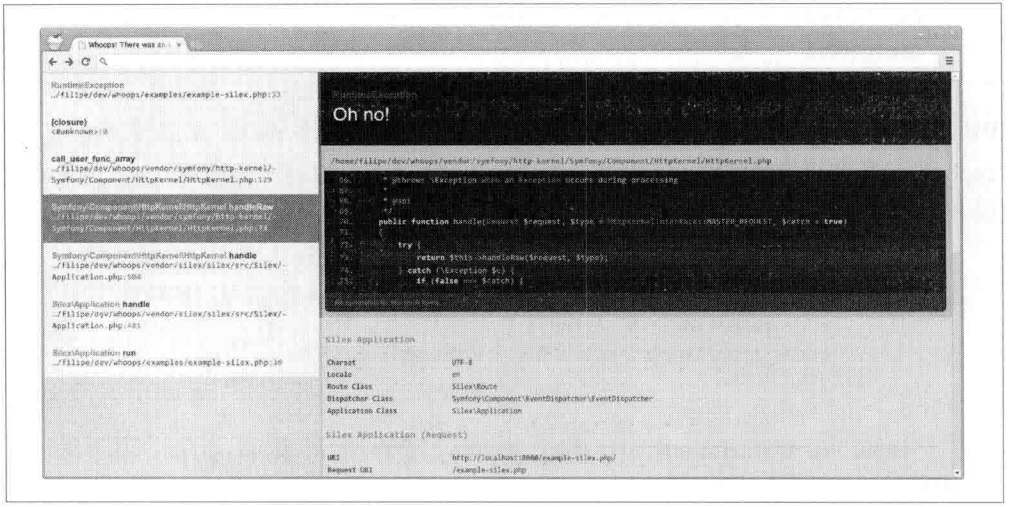


图5-1：Whoops的界面截图

Whoops的界面比PHP默认的错误和异常输出好多了。

Whoops也易于集成。把下述代码写入*composer.json*文件，然后执行`composer install`或`composer update`命令：

```
{
    "require": {
        "filp/whoops": "~1.0"
    }
}
```

然后，在PHP应用的引导文件中注册Whoops提供的错误和异常处理程序，如示例5-42所示。

示例5-42：注册Whoops提供的处理程序

```
<?php
// 使用Composer自动加载器
require 'path/to/vendor/autoload.php';

// 设置Whoops提供的错误和异常处理程序
$whoops = new \Whoops\Run;
$whoops->pushHandler(new \Whoops\Handler\PrettyPageHandler);
$whoops->register();
```

就这么简单。如果PHP脚本触发了PHP错误，或者应用没有捕获异常，就会看到Whoops诊断页面。

示例5-42使用的是Whoops提供的`PrettyPageHandler`处理程序，这个处理程序会创建如图5-1所示的诊断页面。Whoops还提供了其他处理程序，例如纯文本处理程序、回调处

理程序、JSON响应处理程序、XML响应处理程序和SOAP响应处理程序（如果你的尖头发老板[译注3]经常把“企业”这个词挂在嘴边，可能需要这个处理程序）。我开发每个应用时都会在开发环境中使用Whoops。

在生产环境中处理错误和异常

我们知道，在生产环境要记录错误。PHP提供了error_log()函数，使用这个函数可以把错误消息写入文件系统或syslog，还可以通过电子邮件发送错误消息。不过我们有更好的选择——Monolog（*https://github.com/Seldaek/monolog*）。Monolog这个组件非常好，专注于做一件事，即记录日志。使用Composer可以轻易地把这个组件集成到PHP应用中。

首先，在*composer.json*文件中加入monolog/monolog包：

```
{
    "require": {
        "monolog/monolog": "~1.11"
    }
}
```

然后，执行composer install或composer update命令，安装这个组件。再把示例5-43中的代码添加到PHP应用引导文件的顶部。

示例5-43：在生产环境中使用Monolog记录日志

```
<?php
// 使用Composer自动加载器
require 'path/to/vendor/autoload.php';

// 导入Monolog的命名空间
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

// 设置Monolog提供的日志记录器
$log = new Logger('my-app-name');
$log->pushHandler(new StreamHandler('path/to/your.log', Logger::WARNING));
```

就这么简单。现在，Monolog的日志记录器会把Logger::WARNING及以上等级的日志消息写入*path/to/your.log*文件。

Monolog的扩展性很好，我们可以编写多个处理程序，让每个处理程序只处理一个日志等级。例如，我们再添加一个Monolog处理程序，把重要的提醒或突发错误通过电子邮件发给管理员。为此，我们需要使用SwiftMailer组件。下面我们把这个组件添加到*composer.json*文件中，然后执行composer update命令：

译注3：尖头发老板是连载漫画《呆伯特》（http://zh.wikipedia.org/wiki/呆伯特）中呆伯特的老板。这个老板半秃顶，两边的头发尖尖的，爱说大话且富有向物理现实挑战的精神。

```
{
    "require": {
        "monolog/monolog": "~1.11",
        "swiftmailer/swiftmailer": "~5.3"
    }
}
```

接下来我们要修改代码，添加一个新的Monolog处理程序，让SwiftMailer通过电子邮件发送错误消息（如示例5-44所示）。

示例5-44：在生产环境中使用Monolog记录日志

```
<?php
// 使用Composer自动加载器
require 'vendor/autoload.php';

// 导入Monolog的命名空间
use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Monolog\Handler\SwiftMailerHandler;

date_default_timezone_set('America/New_York');

// 设置Monolog和基本的处理程序
$log = new Logger('my-app-name');
$log->pushHandler(new StreamHandler('logs/production.log', Logger::WARNING));

// 添加SwiftMailer处理程序，让它处理重要的错误
$transport = \Swift_SmtpTransport::newInstance('smtp.example.com', 587)
             ->setUsername('USERNAME')
             ->setPassword('PASSWORD');
$mailer = \Swift_Mailer::newInstance($transport);
$message = \Swift_Message::newInstance()
           ->setSubject('Website error!')
           ->setFrom(array('daemon@example.com' => 'John Doe'))
           ->setTo(array('admin@example.com'));
$log->pushHandler(new SwiftMailerHandler($mailer, $message, Logger::CRITICAL));

// 使用日志记录器
$log->critical('The server is on fire!');
```

现在，如果记录了重要的提醒或突发错误，Monolog会使用SwiftMailer创建的`$mailer`和`$message`对象，通过电子邮件发送记录的消息，电子邮件的正文是记录的消息文本。

第三部分

部署、测试和调优

第6章

主机

PHP应用开发好了？恭喜你！不过，现在还没什么用，因为用户无法使用。你要把应用存储到服务器中，让预期受众能访问。一般来说，存储PHP应用有四种方式：共享服务器、虚拟私有服务器、专用服务器和平台即服务。每种方式都有自己的优点，都有适用的应用类型和一定的预算。

而且主机商也有很多，如果你刚接触Web主机领域，要考虑的事情有很多。有些主机商只提供共享服务器，有些主机商除此之外还提供虚拟私有服务器和专用服务器。本章不会过多讨论主机商，我们要重点讨论的是主机方案。

共享服务器

共享服务器是最便宜的主机方案，每月1～10美元。我们不应该选择共享主机方案。我这么说不是因为我对共享主机提供商的服务质量或客户支持有意见，好的共享主机提供商有很多。我这么说只是因为共享主机方案对开发者不友好。

如其名所示，共享服务器意味着要和其他人共享服务器资源。如果选择购买共享主机，你的主机账户会与很多其他顾客的账户在同一个物理设备中。假如你使用的设备有2GB内存，那么你的PHP应用或许只能使用全部内存的一小部分，具体是多少取决于这台设备中有多少账户。如果同一台设备中的其他账户运行一个编写拙劣的脚本，会对你的应用产生负面影响。有些共享主机提供商会超卖共享服务器，导致你的PHP应用始终要在拥挤的设备中争夺系统资源。

而且，共享主机还很难定制。例如，你的应用可能需要使用Memcached（*http://memcached.org*）或Redis（*http://redis.io*）在内存中缓存；可能想安装Elasticsearch

（*http://www.elasticsearch.org*），为应用添加搜索功能。可是，共享服务器使用的软件难以定制（可能根本无法定制），最终受影响的是你的应用。

共享服务器很少提供远程SHH访问功能，通常只能使用(S)FTP访问。这个缺陷有严重的限制，妨碍了我们自动部署PHP应用。

如果预算非常少，或者需求很简单，共享服务器也许够用了。然而，如果你开发的是商业网站或较受欢迎的PHP应用，最好使用虚拟私有服务器、专用服务器或PaaS。

虚拟私有服务器

虚拟私有服务器（Virtual Private Server，VPS）看起来、感觉起来以及表现都像是裸机服务器，但其实不是裸机服务器。VPS由一系列系统资源组成，分布在一台或多台物理设备中，不过仍有自己的文件系统、根用户、系统进程和IP地址。VPS的内存、CPU和带宽是固定的，而且都只属于你一个人。

VPS的系统资源比共享服务器多，会提供根SSH访问功能，而且不限制能安装什么软件。不过，功能强意味着责任大。VPS会提供根权限，让你访问底层操作系统。我们要根据PHP应用的需求，自己动手配置和保护操作系统。对大多数PHP应用来说，VPS是最好的选择。VPS提供了足够的系统资源（例如，CPU、内存和硬盘空间），而且能按需增减。VPS每月需要10～100美元，具体多少取决于PHP应用所需的系统资源量。如果你的PHP应用特别受欢迎（每月有几十万访问量），觉得VPS太贵，或许应该考虑升级，使用专用服务器。

建议： 我几乎都使用VPS，因为VPS能在费用、功能和灵活性之间平衡。我最喜的主机商是Linode（*https://linode.com*），它提供有VPS和专用主机方案。Linode虽然不是最便宜的，但是根据我的经验，Linode的主机速度快且稳定，而且提供了很多有用的教程。

专用服务器

专用服务器是机架式设备，由主机商代你安装、运行和维护。我们可以根据自己制定的规格配置专用服务器。专用服务器是真实的设备，必须搬运、安装和监控，设置和配置的速度没有VPS快。话虽如此，但是专用服务器能为要求高的PHP应用提供最好的性能。

专用服务器和VPS非常类似，有根权限，能通过SSH访问底层操作系统，而且必须根据PHP应用的需求保护和配置操作系统。专用服务器的优点是成本效益高。随着所需的系统资源越来越多，最终你会觉得VPS太贵，而自己投资基础设施能省钱。

专用服务器每月要花几百美元，具体多少取决于服务器的规格。我们可以托管专用服务器（额外付钱给主机商，让它们管理服务器），也可以不托管（自己管理服务器）。

PaaS

使用平台即服务（Platforms as a Service， PaaS）能快速发布PHP应用。与虚拟私有服务器和专用服务器不同，我们无需管理PaaS。我们要做的只是登录PaaS提供商的控制面板，单击几个按钮。有些PaaS提供商会提供命令行工具或HTTP API，让我们部署和管理存储的PHP应用。流行的PHP PaaS提供商有：

- AppFog（*https://appfog.com/*）
- AWS Elastic Beanstalk（*http://aws.amazon.com/elasticbeanstalk/*）
- Engine Yard（*https://www.engineyard.com/products/cloud*）
- Fortrabbit（*http://fortrabbit.com/*）
- Google App Engine（*http://bit.ly/g-app-engine*）
- Heroku（*https://devcenter.heroku.com/categories/php*）
- Microsoft Azure（*http://www.windowsazure.com/*）
- Pagoda Box（*https://pagodabox.com/*）
- Red Hat OpenShift（*http://openshift.com/*）
- Zend Developer Cloud（*http://bit.ly/z-dev-cloud*）

各个PaaS提供商的价格有所不同，不过与虚拟私有服务器差不多：美元10～100美元。我们要为PHP应用使用的系统资源买单。系统资源可以按需增减。我推荐不想自己管理服务器的开发者使用PaaS主机方案。

选择主机方案

我们要根据自己的需求选择合适的主机方案。任何时候，只要需要都可以升级或降级主机基础设施。对小型的PHP应用或原型来说，PaaS提供商（例如Engine Yard或Heroku）或许是最好最省事的方案。如果想更多地控制服务器的配置，那就使用VPS。如果应用特别受欢迎，VPS被几百万的访问量击垮了（顺便恭喜你），那就换用专用服务器。不管选择哪种主机方案，都要保证主机中有最新稳定版PHP，以及PHP应用所需的扩展。

第7章

配置

选好主机后，我们要配置PHP应用的服务器了。说实话，配置服务器是一门艺术，而不是学科。如何配置应用完全取决于应用的需求。

注意： 如果使用PaaS，服务器基础设施由PaaS提供商管理。我们只需按照提供商的说明，把PHP应用传到PaaS平台，这样就行了。

如果使用的不是PaaS，必须先配置VPS或专用服务器才能运行PHP应用。配置服务器不像听起来那么难（先别高兴），不过确实需要熟悉命令行。如果不熟悉命令行，最好使用PaaS，例如Engine Yard或Heroku。

我不认为自己是系统管理员，可是基本的系统管理知识对应用开发者特别有用，掌握这些知识后能开发出更灵活和强健的应用。本章我会分享我掌握的系统管理知识，让你能自如地在终端里配置PHP应用的服务器。然后，我会推荐一些额外资源，让你继续提升系统管理技能。

注意： 在本章，我假设你知道如何使用命令行编辑器编辑文本文件，例如nano（*http://www.nano-editor.org*）或vim（*http://www.vim.org*）（大多数Linux发行版都提供了这两个编辑器）。如果不会用，要使用其他方法访问和编辑服务器中的文件。

我们的目标

首先，我们要购买虚拟私有服务器或者专用服务器。然后，我们要安装Web服务器，以

便接收HTTP请求。最后，我们要设置并管理一组PHP进程，用于处理PHP请求，这些进程必须与Web服务器通信。

几年前，一般都会安装Apache Web服务器和Apache mod_php模块。Apache Web服务器会为每个HTTP请求派生一个专门的子进程。Apache mod_php模块会在派生的每个子进程中嵌入专门的PHP解释器，即使进程只用来伺服静态资源，例如JavaScript、图像和样式表，也会嵌入PHP解释器。这么做消耗很大，会浪费系统资源。如今，使用Apache的PHP开发者越来越少了，因为有了更高效的解决方案。

如今，我们使用nginx（*http://nginx.org/*）Web服务器。这个服务器位于一系列PHP-FPM进程的前面，会把PHP请求转发给这些进程处理。本章我就演示这种解决方案。

设置服务器

首先，我们来设置虚拟私有服务器（Virtual Private Server，VPS）。我特别喜欢Linode（*http://linode.com/*）。它虽然不是最便宜的VPS提供商，但却是最稳定的提供商之一。打开Linode（或者你选择的提供商）的网站，新购买一台VPS。提供商会让你为新买的服务器选择一个Linux发行版，还会让你设置根用户的密码。

建议： 很多VPS提供商，例如Linode（*http://linode.com/*）和Digital Ocean（*https://www.digitalocean.com*），是按小时收费的。这意味着，架设一台VPS的成本几乎为零。

首次登录

首先，我们应该登录新买的服务器。下面我们就登录。在本地设备中打开终端，执行ssh命令，登录服务器。记住，要把下面的IP地址换成你的服务器使用的IP地址。

```
ssh root@123.456.78.90
```

这个命令可能会让你确认新服务器的可靠性，此时输入yes再按回车键即可：

```
The authenticity of host '123.456.78.90 (123.456.78.90)' can't be established.
RSA key fingerprint is 21:eb:37:f3:a5:d3:c0:77:47:c4:15:3d:3c:dc:3c:d1.
Are you sure you want to continue connecting (yes/no)?
```

接下来会要求你输入根用户的密码。输入密码，然后按回车键：

```
root@123.456.78.90's password:
```

现在已经登录你的新服务器了！

升级软件

紧接着，我们应该执行下述命令，升级操作系统中的软件：

```
# Ubuntu
apt-get update;
apt-get upgrade;

# CentOS
yum update
```

执行这些命令后会升级操作系统中的软件，在这个过程中会输出很多信息。这是重要的第一步，因为能保证操作系统中默认的软件安装了最新的更新和安全修补。

非根用户

现在你的新服务器还不安全。下面是一些能加强新服务器安全性的良好实践。

我们要创建非根用户。以后我们都应该使用非根用户登录服务器。根用户在服务器中拥有无限权力，它是神，毫无疑问，它能执行任何命令。我们应该尽量让别人不能使用根用户访问服务器。

Ubuntu

使用示例7-1中的命令创建一个名为deploy的非根用户。见到提示时，输入用户密码，然后按照屏幕上显示的说明做。

示例7-1：在Ubuntu中创建非根用户

```
adduser deploy
```

接下来，执行下述命令，把deploy用户加入sudo用户组：

```
usermod -a -G sudo deploy
```

这么做是为了让deploy用户拥有sudo权限（即通过密码认证后可以执行需要特殊权限的任务）。

```
CentOS
```

执行下述命令，创建一个名为deploy的非根用户：

```
adduser deploy
```

然后执行下述命令，为deploy用户设置密码。根据提示输入新密码，然后确认密码。

```
passwd deploy
```

接下来，执行下述命令，把deploy用户加入wheel用户组：

```
usermod -a -G wheel deploy
```

这么做是为了让deploy用户拥有sudo权限（即通过密码认证后可以执行需要特殊权限的任务）。

SSH密钥对认证

在本地设备可以执行下述命令，以非根用户deploy的身份登录服务器：

```
ssh deploy@123.456.78.90
```

这个命令会要求你输入deploy用户的密码，然后登录服务器。我们可以禁用密码认证，加强安全。密码认证有漏洞，会受到暴力攻击，不怀好意的人会不断尝试猜测你的密码。使用ssh登录服务器时应该使用SSH密钥对认证。

密钥对认证是个复杂的话题。简单来说，我们在本地设备中创建一对“密钥”，其中一个是私钥（保存在本地设备中），另一个是公钥（传到远程服务器中）。之所以叫密钥对，是因为使用公钥加密的消息只能使用对应的私钥解密。

使用SSH密钥对认证方式登录远程设备时，远程设备会随机创建一个消息，使用公钥加密，然后把密文发给本地设备。本地设备收到密文后使用私钥解密，然后把解密后的消息发给远程服务器。远程服务器验证解密后的消息之后，再赋予你访问服务器的权限。我极大地简化了这个过程，不过相信你已经掌握要点了。

如果要在多台电脑中登录远程服务器，或许不应该使用SSH密钥对认证。如果想这么做，要在每台本地电脑中生成SSH密钥对，然后再把每个密钥对中的公钥复制到远程服务器中。遇到这种情况，或许最好使用安全的密码进行密码认证。然而，如果只通过一台本地电脑访问远程服务器（很多开发者都是这样），SSH密钥对认证是最好的方式。创建SSH密钥对的方法是，在本地设备中执行下述命令：

```
ssh-keygen
```

然后按照屏幕上显示的内容，按照提示输入所需的信息。这个命令会在本地设备中创建两个文件：*~/.ssh/ id_rsa.pub*（公钥）和*~/.ssh/id_rsa*（私钥）。私钥应该保存在本地电脑中，而且要保密。不过，公钥必须复制到服务器中。我们可以使用scp（安全复制）命令复制公钥：

```
scp ~/.ssh/id_rsa.pub deploy@123.456.78.90:
```

一定要在末尾加上:符号！这个命令会把公钥复制到远程服务器中deploy用户的家目录

里。接下来，以deploy用户的身份登录远程服务器。登录后，确认~/.*ssh*目录是否存在。如果不存在，执行下述命令新建~/.*ssh*目录：

```
mkdir ~/.ssh
```

然后执行下述命令，创建~/.*ssh*/*authorized_keys*文件：

```
touch ~/.ssh/authorized_keys
```

这个文件的内容是一系列允许登录这台远程服务器的公钥。执行下述命令，把刚上传的公钥添加到~/.*ssh*/*authorized_keys*文件中：

```
cat ~/id_rsa.pub >> ~/.ssh/authorized_keys
```

最后，我们要修改几个目录和文件的访问权限，只让deploy用户访问~/.*ssh*目录和~/.*ssh*/*authorized_keys*文件。这个目录和文件的访问权限由下述命令设置：

```
chown -R deploy:deploy ~/.ssh;
chmod 700 ~/.ssh;
chmod 600 ~/.ssh/authorized_keys;
```

工作完成了！现在，在你的本地设备中应该无需输入密码就能通过ssh登录远程服务器了。

注意： 只有在存放私钥的本地设备中通过ssh登录远程服务器时才不用输入密码。

禁用密码，禁止根用户登录

下面我们让远程服务器再安全一些。我们要禁止所有用户通过密码认证登录，还要禁止根用户登录。记住，根用户能做任何事，所以我们要尽量不让根用户访问服务器。

以deploy用户的身份登录远程服务器，然后在你喜欢的文本编辑器中打开/*etc*/*ssh*/*sshd_config*文件。这是SSH服务器软件的配置文件。找到PasswordAuthentication设置，将其值改为no；如果需要，去掉这个设置的注释。然后，找到PermitRootLogin设置，将其值改为no；如果需要，去掉这个设置的注释。保存改动，然后执行下述命令重启SSH服务器，让改动生效：

```
# Ubuntu
sudo service ssh restart

# CentOS
sudo systemctl restart sshd.service
```

这样就行了。我们的服务器安全了，下面该安装运行PHP应用所需的额外软件了。从现在开始，所有命令都要在远程服务器中执行，而且要以非根用户deploy的身份执行。

注意： 服务器的安全是长久性任务，应该一直关注。除了上述说明，我还建议安装防火墙。Ubuntu用户可以使用UFW（*https://help.ubuntu.com/community/UFW*），CentOS用户可以使用iptables（*http://wiki.centos.org/HowTos/Network/IPTables*）。

PHP-FPM

PHP-FPM（PHP FastCGI Process Manager的简称，意思是“PHP FastCGI进程管理器”）是用于管理PHP进程池的软件，用于接收和处理来自Web服务器（例如nginx）的请求。PHP-FPM软件会创建一个主进程（通常以操作系统中根用户的身份运行），控制何时以及如何把HTTP请求转发给一个或多个子进程处理。PHP-FPM主进程还控制着什么时候创建（处理Web应用更多的流量）和销毁（子进程运行时间太久或不再需要了）PHP子进程。PHP-FPM进程池中的每个进程存在的时间都比单个HTTP请求长，可以处理10、50、100、500或更多的HTTP请求。

安装

安装PHP-FPM最简单的方式是使用操作系统原生的包管理器，如下述命令所示。

建议： PHP-FPM的详细安装说明参见附录A。

```
# Ubuntu
sudo apt-get install python-software-properties;
sudo add-apt-repository ppa:ondrej/php5-5.6;
sudo apt-get update;
sudo apt-get install php5-fpm php5-cli php5-curl \
     php5-gd php5-json php5-mcrypt php5-mysqlnd;
# CentOS
sudo rpm -Uvh \
  http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm;
sudo rpm -Uvh \
  http://rpms.famillecollet.com/enterprise/remi-release-7.rpm;
sudo yum -y --enablerepo=epel,remi,remi-php56 install php-fpm php-cli php-gd \
  php-mbstring php-mcrypt php-mysqlnd php-opcache php-pdo php-devel;
```

建议： 如果使用rpm安装EPEL失败，打开浏览器，访问*http://dl.fedoraproject.org/pub/epel/7/x86_64/e/*，找到最新版 EPEL 的下载地址，使用这个地址安装。

全局配置

在Ubuntu中，PHP-FPM的主配置文件是*/etc/php5/fpm/php-fpm.conf*；在CentOS中，PHP-FPM的主配置文件是*/etc/php-fpm.conf*。在你喜欢的文本编辑器中打开配置文件。

注意： PHP-FPM的配置文件使用INI文件格式。INI格式的详细信息参见维基百科（*https://en.wikipedia.org/wiki/INI_file*）。

下面是PHP-FPM最重要的全局设置，我建议把默认值改为下面列出的值。默认情况下，这两个设置可能被注释掉了，如果需要，去掉注释。这两个设置的作用是，如果在指定的一段时间内有指定个子进程失效了，让PHP-FPM主进程重启。这是PHP-FPM进程的基本安全保障，能解决简单的问题，但是不能解决由拙劣的PHP代码引起的重大问题。

`emergency_restart_threshold = 10`

在指定的一段时间内，如果失效的PHP-FPM子进程数超过这个值，PHP-FPM主进程就优雅重启。

`emergency_restart_interval = 1m`

设定`emergency_restart_threshold`设置采用的时间跨度。

注意： PHP-FPM全局设置的详细信息参见*http://php.net/manual/en/install.fpm.configuration.php*。

配置进程池

PHP-FPM配置文件其余的内容是一个名为`Pool Definitions`的区域。这个区域里的配置用于设置每个PHP-FPM进程池。PHP-FPM进程池中是一系列相关的PHP子进程。通常，一个PHP应用有自己的一个PHP-FPM进程池。

在Ubuntu中，`Pool Definitions`区域只有下面这一行内容：

```
include=/etc/php5/fpm/pool.d/*.conf
```

CentOS则在PHP-FPM主配置文件的顶部使用下面这行代码引入进程池定义文件：

```
include=/etc/php-fpm.d/*.conf
```

这行代码的作用是让PHP-FPM加载*/etc/php5/fpm/pool.d/*目录（Ubuntu）或*/etc/php-fpm.d/*目录（CentOS）中的各个进程池定义文件。进入这个目录，应该会看到一个名为*www.conf*的文件。这是名为`www`的默认PHP-FPM进程池的配置文件。在你喜欢的文本编辑器中打开这个文件。

注意：每个PHP-FPM进程池的配置文件开头都是[符号，后跟进程池的名称，然后是]符号。例如，在默认的PHP-FPM进程池的配置文件中，开头是[www]。

各个PHP-FPM进程池都以指定的操作系统用户和用户组的身份运行。我喜欢以单独的非根用户身份运行各个PHP-FPM进程池，这样在命令行中使用top或ps aux命令时便于识别每个PHP应用的PHP-FPM进程池。这是个好习惯，因为每个PHP-FPM进程池中的进程都受相应的操作系统用户和用户组的权限限制在沙盒中。

我们要配置默认的www PHP-FPM进程池，让它以deploy用户和用户组的身份运行。如果还没打开www PHP-FPM进程池的配置文件，现在使用你喜欢的文本编辑器打开这个文件。我建议把以下设置的默认值改为下面列出的值：

`user = deploy`

拥有这个PHP-FPM进程池中子进程的系统用户。要把这个设置的值设为运行PHP应用的非根用户的用户名。

`group = deploy`

拥有这个PHP-FPM进程池中子进程的系统用户组。要把这个设置的值设为运行PHP应用的非根用户所属的用户组名。

`listen = 127.0.0.1:9000`

PHP-FPM进程池监听的IP地址和端口号，让PHP-FPM只接受nginx从这里传入的请求。127.0.0.1:9000让指定的PHP-FPM进程池监听从本地端口9000进入的连接。我使用的端口是9000，不过你可以使用任何不需要特殊权限（大于1024）且没被其他系统进程占用的端口号。配置nginx虚拟主机时会再次讨论这个设置。

`listen.allowed_clients = 127.0.0.1`

可以向这个PHP-FPM进程池发送请求的IP地址（一个或多个）。为了安全，我把这个设置设为127.0.0.1，即只有当前设备能把请求转发给这个PHP-FPM进程池。默认情况下，这个设置可能被注释掉了，如果需要，去掉这个设置的注释。

`pm.max_children = 51`

这个设置设定任何时间点PHP-FPM进程池中最多能有多少个进程。这个设置没有绝对正确的值，你应该测试你的PHP应用，确定每个PHP进程需要使用多少内存，然后把这个设置设为设备可用内存能容纳的PHP进程总数。对大多数中小型PHP应用来说，每个PHP进程要使用5～15MB内存（具体用量可能有差异）。假设我们使用的设备为这个PHP-FPM进程池分配了512MB可用内存，那么我们可以把这个设置的值设为(512MB总内存)/(每个进程使用10MB) = 51个进程。

`pm.start_servers = 3`

PHP-FPM启动时PHP-FPM进程池中立即可用的进程数。同样地，这个设置也没有绝对正确的值。对大多数中小型PHP应用来说，我建议设为2或3。这么做是为了先准备好两到三个进程，等待请求进入，不让PHP应用的头几个HTTP请求等待PHP-FPM初始化进程池中的进程。

`pm.min_spare_servers = 2`

PHP应用空闲时PHP-FPM进程池中可以存在的进程数量最小值。这个设置的值一般与`pm.start_servers`设置的值一样，用于确保新进入的HTTP请求无需等待PHP-FPM在进程池中重新初始化进程。

`pm.max_spare_servers = 4`

PHP应用空闲时PHP-FPM进程池中可以存在的进程数量最大值。这个设置的值一般比`pm.start_servers`设置的值大一点，用于确保新进入的HTTP请求无需等待PHP-FPM在进程池中重新初始化进程。

`pm.max_requests = 1000`

回收进程之前，PHP-FPM进程池中各个进程最多能处理的HTTP请求数量。这个设置有助于避免PHP扩展或库因编写拙劣而导致不断泄露内存。我建议设为1000，不过你应该根据应用的需求做调整。

`slowlog = /path/to/slowlog.log`

这个设置的值是一个日志文件在文件系统中的绝对路径。这个日志文件用于记录处理时间超过*n*秒的HTTP请求信息，以便找出PHP应用的瓶颈，进行调试。记住，PHP-FPM进程池所属的用户和用户组必须有这个文件的写权限。`/path/to/slowlog.log`只是示例，请替换成真正的文件路径。

`request_slowlog_timeout = 5s`

如果当前HTTP请求的处理时间超过指定的值，就把请求的回溯信息写入`slowlog`设置指定的日志文件。把这个设置的值设为多少，取决于你认为多长时间算久。一开始可以设为`5s`。

编辑并保存PHP-FPM的配置文件后，要执行下述命令重启PHP-FPM主进程：

```
# Ubuntu
sudo service php5-fpm restart

# CentOS
sudo systemctl restart php-fpm.service
```

注意： PHP-FPM进程池的配置详情参见*http://php.net/manual/install.fpm.configuration.php*。

---

nginx

nginx（读作*in gen ex*）是Web服务器，类似Apache，不过更容易配置，而且使用的系统内存通常更少。我没有时间详细介绍nginx，不过我想告诉你如何在你的服务器中安装nginx，以及如何把相应的请求转发给PHP-FPM进程池。

安装

安装nginx最简单的方式是使用操作系统原生的包管理器。

Ubuntu

在Ubuntu中可以使用PPA包安装nginx。PPA是Ubuntu专用的术语，是指由nginx社区维护且预先打包好的档案。

```
sudo add-apt-repository ppa:nginx/stable;
sudo apt-get update;
sudo apt-get install nginx;
```

CentOS

在CentOS中可以使用前面添加的第三方软件仓库EPEL安装nginx。CentOS默认使用的软件仓库可能没有最新版nginx。

```
sudo yum install nginx;
sudo systemctl enable nginx.service;
sudo systemctl start nginx.service;
```

虚拟主机

接下来，我们要为PHP应用配置一个nginx虚拟主机。虚拟主机是一系列设置，用于告诉nginx应用的域名、PHP应用在文件系统的什么地方，以及如何把HTTP请求转发给PHP-FPM进程池。

首先，我们必须决定把应用放在文件系统的什么位置。非根用户`deploy`必须拥有PHP应用所在文件系统目录的读写权限。这里，我要把应用的文件放在*/home/deploy/apps/example.com/current*目录中。我们还需要一个保存应用日志文件的目录。我把日志文件放在*/home/deploy/apps/logs*目录中。执行下述命令，创建所需的目录，并赋予正确的权限：

```
mkdir -p /home/deploy/apps/example.com/current/public;
mkdir -p /home/deploy/apps/logs;
chmod -R +rx /home/deploy;
```

然后把PHP应用放到*/home/deploy/apps/example.com/current*目录中。nginx虚拟主机假设PHP应用有个*public/*目录，这是虚拟主机的文档根目录。

每个虚拟主机都有各自的配置文件。如果使用Ubuntu，请创建*/etc/nginx/sites-available/example.conf*配置文件；如果使用CentOS，请创建*/etc/nginx/conf.d/example.conf*文件。然后在你喜欢的文本编辑器中打开*example.conf*配置文件。

nginx虚拟主机的设置在`server {}`块中。下面是虚拟主机配置文件的完整内容：

```
server {
    listen 80;
    server_name example.com;
    index index.php;
    client_max_body_size 50M;
    error_log /home/deploy/apps/logs/example.error.log;
    access_log /home/deploy/apps/logs/example.access.log;
    root /home/deploy/apps/example.com/current/public;

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location ~ \.php {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param SCRIPT_NAME $fastcgi_script_name;
        fastcgi_index index.php;
        fastcgi_pass 127.0.0.1:9000;
    }
}
```

复制上述代码，粘贴到*example.conf*虚拟主机配置文件中。记得要修改`server_name`设置，还要把`error_log`、`access_log`和`root`改为适当的路径。下面简要说明每个虚拟主机设置：

`listen`

设置nginx监听哪个端口进入的HTTP请求。大多数情况下，HTTP流量从80端口进入，HTTPS流量从443端口进入。

`server_name`

用于识别虚拟主机的域名。这个设置要设为你的应用使用的域名，而且域名要指向服务器的IP地址。如果HTTP请求中`Host`首部的值和虚拟主机中`server_name`的值匹配，nginx就会把这个HTTP请求发给这个虚拟主机。

`index`

HTTP请求URI没指定文件时伺服的默认文件。

`client_max_body_size`

对这个虚拟主机来说，nginx接受HTTP请求主体长度的最大值。如果请求主体的长度超过这个值，nginx会返回HTTP 4xx响应。

`error_log`

这个虚拟主机错误日志文件在文件系统中的路径。

`access_log`

这个虚拟主机访问日志文件在文件系统中的路径。

`root`

文档根目录的路径。

除了上述设置之外，server {}块中还有两个location块。这两个location块的作用是，告诉nginx如何处理匹配指定URL模式的HTTP请求。`location / {}`块使用`try_files`指令查找匹配所请求URI的文件；如果未找到相应的文件，再查找匹配所请求URI的目录；如果也未找到相应的目录，把HTTP请求的URI重写为*/index.php*，如果有查询字符串的话，还会把查询字符附加到URI的末尾。这个重写的URL，以及所有以*.php*结尾的URI，都由location ~ \.php {}块管理。

`location ~ \.php {}`块把HTTP请求转发给PHP-FPM进程池处理。还记得吗，前面我们设置PHP-FPM进程池监听端口9000。在这个块中，我们把PHP请求转发到端口9000，交给PHP-FPM进程池处理。

注意：`location ~ \.php {}`块中其他几行的作用是避免潜在的远程代码执行攻击（*http://bit.ly/remote-ex*）。

在Ubuntu中，我们必须执行下述命令，在*/etc/nginx/sites-enabled/*目录中创建虚拟主机配置文件的符号链接：

```
sudo ln -s /etc/nginx/sites-available/example.conf \
     /etc/nginx/sites-enabled/example.conf;
```

最后，执行下述命令，重启nginx：

```
# Ubuntu
sudo service nginx restart

# CentOS
sudo systemctl restart nginx.service
```

现在服务器可以运行PHP应用了！nginx的配置方式有很多种，本章只说明了最基本的设置，因为这是一本关于PHP的书，而不是关于nginx的书。你可以参阅下述有用的资源，

进一步学习nginx配置：

- *http://nginx.org/*
- *https://github.com/h5bp/server-configs-nginx*
- *https://serversforhackers.com/editions/2014/03/25/nginx/*

自动配置服务器

配置服务器是个漫长的过程，而且很枯燥。如果手动配置很多服务器，这种感觉会更强烈。幸好，有些工具能帮我们自动配置服务器。下面是几个流行的服务器配置工具：

- Puppet（*http://puppetlabs.com/*）
- Chef（*https://www.getchef.com/chef/*）
- Ansible（*http://www.ansible.com/home*）
- SaltStack（*http://www.saltstack.com/*）

各个工具之间有所差别，但目标是一致的——根据精确的规格自动配置新服务器。如果要管理多台服务器，我强烈建议研究使用配置工具，这样能节省大量时间。

委托别人配置服务器

还有些在线服务能代你配置服务器，例如泰勒·奥特威尔创建的Forge（*https://forge.laravel.com/*）。我是Forge测试版的用户，我觉得这是特别有用的服务。Forge可以在Linode、Digital Ocean和其他流行的VPS提供商中配置多台服务器。

Forge配置的每台服务器都会自动运用前面所说的安全实践。Forge会自动安装nginx和PHP-FPM软件栈。Forge还简化了部署PHP应用、安装SSL证书、创建CRON任务和其他单调或难以理解的管理任务。如果你不喜欢管理系统，我强烈建议你使用Forge。

延伸阅读

我觉得系统管理很吸引人。我虽然不想全职做这个工作，但是喜欢沉浸在命令行中。我觉得，对开发者来说最好的系统管理学习资源是克里斯·菲道写的“Servers for Hackers”（*https://book.serversforhackers.com/*）。

接下来

本章讨论了如何配置运行PHP应用的服务器。接下来我们要讨论如何调优服务器，让PHP应用的性能维持在最高水平上。

第8章

调优

现在，你的PHP应用应该在nginx和相应的PHP-FPM进程池中运行着了。不过，我们的工作还没做完。我们要调优PHP的配置，使用适用于应用和生产服务器的设置。默认安装的PHP就像是在本地百货商店购买的普通套装，虽然合身，但并不完美。调优的PHP相当于定做的套装，完全匹配你的尺寸。

别太高兴，调优PHP不是提升应用性能的通用措施。拙劣的代码依然拙劣。例如，调优PHP不能解决编写拙劣的SQL查询语句，也不能解决无响应的API调用。不过，调优PHP是提升PHP效率和应用性能的简单措施。

php.ini文件

PHP解释器在*php.ini*文件中配置和调优。这个文件在操作系统中的位置有所不同，如果像我前面演示的那样使用PHP-FPM运行PHP，那么*php.ini*配置文件在*/etc/php5/fpm/*目录中。说来也怪，在命令行中执行php命令时使用的PHP解释器不受这个*php.ini*文件的控制，而是由专门的*php.ini*文件控制。这个*php.ini*文件通常在*/etc/php5/cli/*目录中。如果从源码构建PHP，*php.ini*文件很可能在配置PHP源码文件时指定的$PREFIX目录中。我假设你按照前文所述，使用PHP-FPM运行PHP。不过，我要讲的优化措施适用于所有*php.ini*文件。

注意： 我们应该使用克里斯·科耐特开发的PHP Iniscan工具（*https://github.com/psecio/iniscan*）扫描*php.ini*文件，检查是否使用了安全方面的最佳实践。

*php.ini*文件使用INI格式，这个格式的详情参见维基百科（*https://en.wikipedia.org/wiki/INI_file*）。

内存

运行PHP时我最关心的是每个PHP进程要使用多少内存。*php.ini*文件中的`memory_limit`设置用于设定单个PHP进程可以使用的系统内存最大值。

这个设置的默认值是`128M`，这对大多数中小型PHP应用来说或许合适。可是，如果运行的是微型PHP应用，可以降低这个值，例如设为`64M`，节省系统资源。如果运行的是内存集中型PHP应用（例如使用Drupal搭建的网站），可以增加这个值，例如设为`512M`，提升性能。这个设置的值由可用的系统内存决定。确定给PHP分配多少值是一门艺术，而不是学科。决定给PHP分配多少内存，以及能负担得起多少个PHP-FPM进程时，我会问自己以下几个问题：

一共能分配给PHP多少内存？

首先，我会确定能分配给PHP多少系统内存。例如，我可能会使用一个Linode虚拟设备，这个设备一共有2GB内存。可是，这台设备中可能还有其他进程（例如，nginx、MySQL或memcache），而这些进程也要消耗内存。我觉得留512MB给PHP就足够了。

单个PHP进程平均消耗多少内存？

然后，我会确定单个PHP进程平均消耗多少内存。为此，我要监控进程的内存使用量。如果使用命令行，可以执行top命令，查看运行中的进程的实时统计数据。除此之外，还可以在PHP脚本的最后调用`memory_get_peak_usage()`函数，输出当前脚本消耗的最大内存量。不管使用哪种方式，都要多次运行同一个PHP脚本，然后取内存消耗量的平均值。我发现PHP进程一般会消耗5～20MB内存（具体消耗可能有差异）。如果要上传文件、处理图像，或者运行的是内存集中型应用，得到的平均值显然会高些。

能负担得起多少个PHP-FPM进程？

假设我给PHP分配了512MB内存，每个PHP进程平均消耗15MB内存，我拿内存总量除以每个PHP进程消耗的内存量，从而确定我能负担得起34个PHP-FPM进程。这是个估值，应该再做实验，得到精确值。

有足够的系统资源吗？

最后我会问自己，确信有足够的系统资源运行PHP应用并处理预期的流量吗？如果答案是肯定的，那太好了。如果答案是否定的，就需要升级服务器，添加更多的内存，然后再回到第一个问题。

注意： 我们应该使用Apache Bench（*http://bit.ly/apache-bench*）或Seige（*http://www.joedog.org/siege-home/*），在类似生产环境的条件下对PHP应用做压力测试，因为最好在把应用部署到生产环境之前确定是否有足够的资源可用。

Zend OPcache

确定要分配多少内存后，我会配置PHP的Zend OPcache扩展。这个扩展用于缓存操作码。为什么要这么做呢？我们先来分析每次HTTP请求时通常是如何处理PHP脚本的。首先，nginx把HTTP请求转发给PHP-FPM，PHP-FPM再把请求交给某个PHP子进程处理。PHP进程找到相应的PHp脚本后，读取脚本，把PHP脚本编译成操作码（或字节码）格式，然后执行编译得到的操作码，生成响应。最后，把HTTP响应发给nginx，nginx再把响应发给HTTP客户端。可以看出，每次HTTP请求都要消耗很多资源。

我们可以缓存编译每个PHP脚本得到的操作码，加速这个处理过程。缓存后，我们可以从缓存中直接读取并执行预先编译好的操作码，不用每次处理HTTP请求时都查找、读取和编译PHP脚本。PHP 5.5.0+中内置了Zend OPcache扩展。下面是我在*php.ini*文件中配置和优化Zend OPcache扩展所用的设置：

```
opcache.memory_consumption = 64
opcache.interned_strings_buffer = 16
opcache.max_accelerated_files = 4000
opcache.validate_timestamps = 1
opcache.revalidate_freq = 0
opcache.fast_shutdown = 1
```

`opcache.memory_consumption = 64`

为操作码缓存分配的内存量（单位是MB）。分配的内存量应该够保存应用中所有PHP脚本编译得到的操作码。如果是小型PHP应用，脚本数较少，可以设为较低的值，例如16MB；如果是大型PHP应用，有很多脚本，那就使用较大的值，例如64MB。

`opcache.interned_strings_buffer = 16`

用来存储驻留字符串（interned string）的内存量（单位是MB）。那么驻留字符串是什么呢？我首先也会想到这个问题。PHP解释器在背后会找到相同字符串的多个实例，把这个字符串保存在内存中，如果再次使用相同的字符串，PHP解释器会使用指针。这么做能节省内存。默认情况下，PHP驻留的字符串会隔离在各个PHP进程中。这个设置能让PHP-FPM进程池中的所有进程把驻留字符串存储到共享的缓冲区中，以便在PHP-FPM进程池中的多个进程之间引用驻留字符串。这样能节省更多内存。这个设置的默认值是4MB，不过我喜欢设为16MB。

`opcache.max_accelerated_files = 4000`

操作码缓存中最多能存储多少个PHP脚本。这个设置的值可以是200到100000之间的任何数。我使用的是4000。这个值一定要比PHP应用中的文件数量大。

`opcache.validate_timestamps = 1`

这个设置的值为1时，经过一段时间后PHP会检查PHP脚本的内容是否有变化。检查的时间间隔由`opcache.revalidate_freq`设置指定。如果这个设置的值为0，PHP不会检查PHP脚本的内容是否有变化，我们必须自己动手清除缓存的操作码。我建议在开发环境中设为1，在生产环境中设为0。

`opcache.revalidate_freq = 0`

设置PHP多久（单位是秒）检查一次PHP脚本的内容是否有变化。缓存的好处是，不用每次请求都重新编译PHP脚本。这个设置用于确定在多长时间内认为操作码缓存是新的。在这段时间之后，PHP会检查PHP脚本的内容是否有变化。如果有变化，PHP会重新编译脚本，再次缓存。我使用的值是0秒。仅当`opcache.validate_timestamps`设置的值为1时，这么设置会在每次请求时都重新验证PHP文件。因此，在开发环境中，每次请求都会重新验证PHP文件（这是好事）。这个设置在生产环境中没有任何意义，因为生产环境中`opcache.validate_timestamps`的值始终为0。

`opcache.fast_shutdown = 1`

这么设置能让操作码使用更快的停机步骤，把对象析构和内存释放交给Zend Engine的内存管理器完成。这个设置缺少文档，你只需知道要把它设为1。

文件上传

你的PHP应用允许上传文件吗？如果不允许，为了增强应用的安全性，应该禁止文件上传功能。如果你的应用允许上传文件，最好设置最大能上传的文件大小。除此之外，最好还要设置最多能同时上传多少个文件。下面是我在*php.ini*文件中为我的应用所做的设置：

```
file_uploads = 1
upload_max_filesize = 10M
max_file_uploads = 3
```

默认情况下，PHP允许在单次请求中上传20个文件，上传的每个文件最大为2MB。你可能不想允许同时上传20个文件，我只允许单次请求上传3个文件。不过，你应该设为对你的应用来说合理的值。

如果我的PHP应用允许上传文件，通常都会允许上传大于2MB的文件。我把`upload_max_`

filesize设置的值增加到了10MB，或许要根据应用的需求设为更高的值。但是别设为太大的值，如果这个值太大，Web服务器会抱怨HTTP请求的主体太大，或者请求会超时。

注意： 如果需要上传非常大的文件，Web服务器的配置要做相应调整。除了在*php.ini*文件中设置之外，可能还要调整nginx虚拟主机配置中的client_max_body_size设置（*http://bit.ly/max-body-size*）。

最长执行时间

*php.ini*文件中的max_execution_time设置用于设定单个PHP进程在终止之前最长可以运行多少时间。这个设置的默认值是30秒。我们可不想让PHP进程运行30秒，因为我们想让应用运行得特别快（以毫秒计）。我建议把这个设置改为5秒：

```
max_execution_time = 5
```

注意： 在 PHP 脚本中可以调用set_time_limit()函数（*http://php.net/manual/function.set-time-limit.php*）覆盖这个设置。

你可能会问，如果PHP脚本需要运行更长的时间怎么办？答案是，PHP脚本不能长时间运行。PHP运行的时间越长，Web应用的访问者等待响应的时间就会越长。如果有长时间运行的任务（例如，调整图像尺寸或生成报告），要在单独的职程中运行。

建议： 我会使用PHP中的exec()函数调用bash的at命令。这个命令的作用是派生单独的非阻塞进程，不耽误当前的PHP进程。使用PHP中的exec()函数时，要使用escapeshellarg()函数（*http://php.net/manual/function.escapeshellarg.php*）转义shell参数。

假设我们要生成报告，并把结果制作成PDF文件。这个任务可能要花10分钟才能完成，而我们肯定不想让PHP请求等待10分钟。我们应该单独编写一个PHP文件，假如将其命名为*create-report.php*，让这个文件运行10分钟，最后生成报告。其实，Web应用只需几毫秒就能派生一个单独的后台进程，然后返回HTTP响应，如下所示：

```
<?php
exec('echo "create-report.php" | at now');
echo 'Report pending...';
```

*create-report.php*脚本在单独的后台进程中运行，运行完毕后可以更新数据库，或者通过电子邮件把报告发给收件人。可以看出，我们完全没有理由让长时间运行的任务拖延PHP主脚本，影响用户体验。

建议： 如果发现自己派生了很多后台进程，或许最好使用专门的作业队列。PHPResque（*https://github.com/chrisboulton/php-resque*）是个不错的作业队列管理器，它是基于GitHub的作业队列管理器Resque（*https://github.com/blog/542-introducing-resque*）开发的。

处理会话

PHP默认的会话处理程序会拖慢大型应用，因为这个处理程序把会话数据存储在硬盘中，需要创建不必要的文件I/O，浪费时间。我们应该把会话数据保存在内存中，例如可以使用Memcached（*http://memcached.org*）或Redis（*http://redis.io*）。这么做还有个额外好处，以后便于伸缩。如果会话数据存储在硬盘中，不便于增加额外的服务器。如果把会话数据存储在Memcached或Redis中央数据存储区里，任何一台分布式PHP-FPM服务器都能访问会话数据。

若想在PHP中访问Memcached存储的数据，要安装连接Memcached的PECL扩展（*http://pecl.php.net/package/memcached*）。然后再把下面两行添加到*php.ini*文件中，把PHP默认的会话存储方式改为Memcached：

```
session.save_handler = 'memcached'
session.save_path = '127.0.0.2:11211'
```

缓冲输出

如果在较少的块中发送更多的数据，而不是在较多的块中发送较少的数据，那么网络的效率会更高。也就是说，在较少的片段中把内容传递给访问者的浏览器，能减少HTTP请求总数。

因此，我们要让PHP缓冲输出。默认情况下，PHP已经启用了输出缓冲功能（不过没在命令行中启用）。PHP缓冲4096字节的输出之后才会把其中的内容发给Web服务器。下面是我推荐在*php.ini*文件中使用的设置：

```
output_buffering = 4096
implicit_flush = false
```

建议： 如果想修改输出缓冲区的大小，确保使用的值是4（32位系统）或8（64位系统）的倍数。

真实路径缓存

PHP会缓存应用使用的文件路径，这样每次包含或导入文件时就无需不断搜索包含路径了。这个缓存叫真实路径缓存（realpath cache）。如果运行的是大型PHP文件（例如

Drupal和Composer组件等），使用了大量文件，增加PHP真实路径缓存的大小能得到更好的性能。

真实路径缓存的默认大小为16k。这个缓存所需的准确大小不容易确定，不过可以使用一个小技巧。首先，增加真实路径缓存的大小，设为特别大的值，例如256k。然后，在一个PHP脚本的末尾加上`print_r(realpath_cache_size());`，输出真实路径缓存的真正大小。最后，把真实路径缓存的大小改为这个真正的值。我们可以在*php.ini*文件中设置真实路径缓存的大小：

```
realpath_cache_size = 64k
```

接下来

现在服务器马力开足，可以把PHP应用部署到生产环境了。下一章会讨论几个自动部署PHP应用的策略。

第9章

部署

我们已经配置好了运行nginx和PHP-FPM的服务器，现在要把我们的PHP应用部署到生产服务器了。把代码推送到生产环境有多种方式。以前，PHP开发者喜欢使用FTP部署PHP代码。现在仍然可以使用FTP，不过有更安全且能预知结果的部署策略。本章说明如何使用现代化工具自动部署，这种方式简单、可预知结果，而且可逆。

版本控制

我假设你使用了版本控制。你用了吗？如果用了，做得好；如果没用，停下手头的事情，把代码纳入版本控制。我喜欢使用Git（*http://git-scm.com*）对代码做版本控制，不过使用其他版本控制软件也行，例如Mercurial（*http://mercurial.selenic.com*）。我之所以使用Git，是因为我知道怎么使用，而且可以与流行的在线仓库无缝集成，例如Bitbucket（*https://bitbucket.org*）和GitHub（*https://github.com*）。

对PHP应用的开发者来说，版本控制是特别有用的工具，因为能记录代码基的变化。我们可以把一个时间点的代码标记为发布版，可以回滚到之前的状态，还可以在单独的分支中实验新功能，而不影响生产环境使用的代码。更重要的是，版本控制有助于自动部署PHP应用。

自动部署

为了让部署过程变得简单、可预知结果和可逆，一定要自动部署应用。一旦实现自动化，我们无需再担心复杂的部署过程。复杂的部署过程让人害怕，而且人们很少会触碰让人害怕的事情。

让部署变得简单

我们应该只执行一个简单的命令就能部署。简单的部署过程不再让人害怕，因此更愿意把代码推送到生产环境。

让部署的结果可预知

我们要让部署的结果可预知。可预知结果的过程更不让人害怕，因为我们确切地知道要做什么。部署不应该有意料之外的副作用。如果出错了，部署过程会中止，现有的代码基不会受到影响。

让部署可逆

我们要让部署过程可逆。如果不小心把拙劣的代码推送到生产环境，应该有个简单的命令可以回滚到之前稳定的代码基。这是我们的安全保障。可逆的部署过程让我们不再害怕，而是乐于把代码推送到生产环境。如果搞砸了，回滚到之前的版本即可。

Capistrano

Capistrano（*http://capistranorb.com/*）是用于自动部署应用的软件，能让部署变得简单、可预知结果和可逆。Capistrano运行在本地设备中，通过SSH与远程服务器通信。Capistrano本来是为了部署Ruby应用而开发的，不过对任何编程语言开发的应用都有用，包括PHP。

Capistrano的工作方式

Capistrano安装在本地设备中。部署PHP应用时，Capistrano会在本地设备中执行SSH命令，与远程服务器通信。Capistrano会在远程服务器中保存之前部署的应用，而且每次部署的版本放在各自的目录中。Capistrano会维护五个或更多之前部署的应用，以防需要回滚到早前版本。Capistrano还会创建一个*current/*目录，通过符号链接指向当前部署的应用所在的目录。在生产服务器中，Capistrano管理的目录结构可能像示例9-1这样。

示例9-1：目录结构示例

```
/
    home/
        deploy/
            apps/
                my_app/
                    current/
                    releases/
                        release1/
```

```
release2/
release3/
release4/
release5/
```

把新版应用部署到生产环境时，Capistrano首先从应用的Git仓库获取最新版代码，然后把应用的代码放到*releases/*目录中的一个新子目录中，最后把*current/*目录的符号链接指向这个新目录。让Capistrano回滚到之前的版本时，Capistrano会把*current/*目录的符号链接指向*releases/*目录中存放之前版本的子目录。Capistrano是一种优雅且简单的部署方案，能让PHP应用的部署过程变得简单、可预知结果和可逆。

安装

Capistrano应该安装在本地设备中，别在远程服务器中安装。安装时还需要`ruby`和`gem`。OS X系统已经有了。Linux用户可以使用相应的包管理器安装`ruby`和`gem`。安装好之后，执行下述命令安装Capistrano：

```
gem install capistrano
```

配置

安装Capistrano之后，为了使用Capistrano，必须初始化项目。打开终端，进入项目的最顶层目录，然后执行下述命令：

```
cap install
```

这个命令会创建一个名为*Capfile*的文件，一个名为*config/*的目录，以及一个名为*lib/*的目录。现在，项目的最顶层目录应该有下述文件和目录：

```
Capfile
config/
    deploy/
        production.rb
        staging.rb
    deploy.rb
lib/
    capistrano/
        tasks/
```

*Capfile*是Capistrano的中央配置文件，会聚合*config/*目录中的配置文件。*config/*目录中存放的是各个远程服务器环境（例如，测试环境、过渡环境或生产环境）的配置文件。

注意： Capsitrano的配置文件使用Ruby语言编写。尽管如此，仍然易于编辑和理解。

默认情况，Capistrano假设你为应用搭建了多个环境。例如，可能有单独的过渡环境和生产环境。Capsitrano在*config/deploy/*目录中为每个环境都提供了单独的配置文件。Capistrano还提供了*config/deploy.rb*配置文件，这个文件用于保存所有环境通用的设置。

在每个环境中，Capistrano会区分服务器的角色。例如，生产环境可能有前置Web服务器（web角色）、应用服务器（app角色）和数据库服务器（db角色）。只有最大规模的应用才有必要使用这种架构，小型PHP应用一般在同一台设备中运行Web服务器（nginx）、应用服务器（PHP-FPM）和数据库服务器（MariaDB）。

此次演示，我只会使用Capistrano的web角色，而不会使用app和db角色。在Capistrano中，角色的作用是把相关的任务组织在一起，只在属于指定角色的服务器中执行这些任务。这里我们无需关心这个问题。不过，我尊重Capistrano对服务器环境的处理方式。此次演示，我使用的是生产环境，不过下述步骤也适用于其他环境（例如，过渡环境或测试环境）。

config/deploy.rb文件

下面说明*config/deploy.rb*文件。这个配置文件包含所有环境（例如，过渡环境和生产环境）通用的设置。此次演示，我们的大多数Capistrano设置都保存在这个文件中。在你喜欢的文本编辑器中打开*config/deploy.rb*文件，然后更新下述设置：

`:application`

这是PHP应用的名称。只能包含字母、数字和下划线。

`:repo_url`

这是Git仓库的URL。这个URL必须指向一个Git仓库，而且远程服务器必须能访问这个仓库。

`:deploy_to`

这是远程服务器中一个目录的绝对路径，我们部署的PHP应用就存放在这个目录中。对示例9-1所示的目录结构来说，这个设置的值是*/home/deploy/apps/my_app*。

`:keep_releases`

保留多少个旧版，以防想把应用回滚到之前的版本。

config/deploy/production.rb文件

这个文件只包含生产环境的设置。这个文件用于定义生产环境的角色，列出属于各个角色的服务器。我们只使用web角色，而且只有一个服务器属于这个角色。我们要使用第7章配置的那个服务器。把*config/deploy/production.rb*中的全部内容替换成下述代码。记得要替换示例IP地址。

```
role :web, %w{deploy@123.456.78.90}
```

认证

使用Capistrano部署应用之前，我们必须在本地电脑和远程服务器之间，以及远程服务器和Git仓库之间建立认证。前面已经讨论了如何使用SSH密钥对在本地电脑和远程服务器之间建立认证。在远程服务器和Git仓库之间也要使用SSH密钥对建立认证。

使用前面讨论的方法在每台远程服务器中生成SSH公钥和密钥。Git仓库应该能访问每台远程服务器的公钥。GitHub和Bitbucket都允许在用户账户中添加多个SSH公钥。总之，我们必须不使用密码就能把Git仓库克隆到远程服务器。

准备远程服务器

就快能部署应用了。不过在部署之前，我们需要准备远程服务器。我们要通过SSH登录远程服务器，创建一个目录，存放部署的PHP应用。`deploy`用户必须有这个目录的读写权限。我喜欢在`deploy`用户的家目录中创建这个目录，如下所示：

```
/
    home/
        deploy/
            apps/
                my_app/
```

虚拟主机

Capistrano会创建符号链接，把*current/*目录指向存放当前应用版本的目录。因此，我们要更新Web服务器的虚拟主机文档根目录，指向Capistrano的*current/*目录。根据上述文件系统结构图，要把虚拟主机的文档根目录改为*/home/deploy/apps/my_app/current/public/*。这么设置的前提是，假设PHP应用中有个*public/*目录，把它当做文档根目录。然后重启Web服务器，加载修改后的虚拟主机配置。

依赖的软件

远程服务器不需要Capistrano，但是需要Git。而且还需要运行PHP应用所需的全部软件。我们可以执行下述命令安装Git：

```
# Ubuntu
sudo apt-get install git;

# CentOS
sudo yum install git;
```

Capistrano的钩子

Capistrano允许在部署应用过程中的特定时刻执行我们指定的命令。很多PHP开发者都使用Composer管理应用的依赖。每次使用Capistrano部署应用时，我们可以使用Capistrano的钩子安装Composer依赖。在你喜欢的文本编辑器中打开*config/deploy.rb*文件，添加下述Ruby代码：

```
namespace :deploy do
  desc "Build"
  after :updated, :build do
    on roles(:web) do
      within release_path  do
        execute :composer, "install --no-dev --quiet"
      end
    end
  end
end
```

建议： 如果项目使用Composer依赖管理器，要在远程服务器中安装Composer。

现在，每次部署到生产环境时都会安装应用的依赖。Capistrano钩子的更多信息参见Capistrano的网站（*http://bit.ly/cap-flow*）。

部署应用

现在到有趣的环节了！先确保提交了应用的最新代码，并且已经推送到Git仓库了。然后，在本地电脑中打开终端，进入应用的最顶层目录。如果一切顺利，执行下面这一个命令就能部署你的PHP应用：

```
cap production deploy
```

回滚应用

如果不慎把拙劣的代码部署到生产环境了，可以执行下面这个命令回滚到之前的版本：

```
cap production deploy:rollback
```

延伸阅读

我只介绍了一点皮毛。Capistrano还有很多功能，能进一步简化部署流程。Capistrano是我最喜欢的部署工具，不过也有很多其他工具可用，例如：

- Deployer（*http://deployer.in/*）

- Magallanes（*http://magephp.com/*）
- Rocketeer（*http://rocketeer.autopergamene.eu/*）

接下来

我们配置了服务器，还使用Capistrano自动部署了PHP应用。接下来我们要讨论如何确保PHP应用能按预期运行。为此，我们要测试和分析。

第10章

测试

测试是开发PHP应用过程中的重要一步，但往往被忽视了。我相信很多PHP开发者都不测试，因为他们觉得测试是不必要的负担，投入的时间多而收益却很少。有些开发者可能不知道如何测试，因为测试工具太多，学习曲线太陡。

我希望通过本章消除这些误区。我要让你感觉测试很亲切，愿意测试你编写的PHP代码。我要让你把测试看成整个工作流程不可分割的一部分，在开发应用开始、开发的过程中和开发结束后都要测试。

为什么测试？

我们编写测试是为了确保PHP应用始终能按照我们预期的方式运行。就这么简单。你是不是经常害怕把应用部署到生产环境？以前我不测试自己编写的代码，我害怕把发布版推送到生产环境。我会想，我的代码正确吗，会不会有缺陷。我只能十指交叉，祈祷代码能正常运行。除此之外别无他法。这么做让人害怕，有压力，而且结果往往都不好。然而，测试能降低这种不确定性，还能让我们带着自信编写和部署代码。

你的尖头发老板可能不同意这么做，觉得没有足够的时间编写测试。毕竟时间就是金钱。这是鼠目寸光的想法。安装测试所需的基础设施以及编写测试是要花时间，但这是明智的投资，未来会得到回报。测试能协助我们编写一开始可以正常运行的代码，而且在持续迭代的过程中还能确保没有破坏之前的代码。编写测试可能会让进度慢下来，但是有了测试，以后我们不用浪费大量时间排查和重构以前忽略的缺陷。从长远来看，测试能省钱，能减少停机时间，还能鼓舞人心。

何时测试？

我发现很多PHP开发者开发完成之后才编写测试。这些开发者知道测试很重要，不过他们觉得测试是被逼无奈，而不是心甘情愿去做的事情。这些开发者往往把测试推延到应用开发过程的最后才做，而且是为了满足管理层的要求，匆匆编写一些测试，然后就收工了。这么做不对。测试很重要，开发之前、开发的过程中和开发完成之后都要关注。

开发之前

开发应用之前，我们要安装和配置测试工具。选择使用什么工具无关紧要，重要的是要把这些工具当成应用的重要依赖。这样在开发应用的过程中，无论是生理上还是心理上都更愿意编写测试。这个阶段还是和项目经理交流的好时机，你们要定义应用的整体行为。

开发的过程中

在开发应用的每个功能时都要编写并运行测试。你刚才是不是新加了一个PHP类？现在就测试吧，防止以后忘了。在开发的过程中测试能增强自信，写出稳定的代码，而且还能帮助我们快速找出并重构破坏现有功能的新代码。

开发完成之后

在开发的过程中你可能不会先期测试应用的全部行为。如果发布应用后发现了缺陷，要编写新测试，确保修补缺陷的方式是正确的。测试不是一劳永逸的事情，和应用本身一样，我们要不断修改和改进。如果更新了应用的代码，一定也要更新受影响的测试。

测试什么？

我们应该测试应用的最小组成部分。从微观的角度来看，应用由PHP类、方法和函数组成。因此，我们应该隔离测试每个公开的类、方法和函数，确保表现符合预期。如果我们知道各个部分能单独正常运行，就可以确信集成在一起组成整个应用时也能正常运行。这种测试叫单元测试。

可是，单独测试每个部分并不能保证各个部分在整个应用中能正常运行。因此我们还要使用自动化测试工具从宏观上验证应用的整体行为。这种测试叫功能测试。

如何测试？

我们知道了为什么测试，何时测试，以及测试什么。更重要的一点是，我们要知道如何

测试。对PHP开发者来说，有几种流行的测试方式可选择。有些开发者喜欢单元测试，有些开发者喜欢测试驱动开发（Test-Driven Development，TDD），还有些开发者喜欢行为驱动开发（Behavior-Driven Development，BDD）。这些测试方式之间不是互斥的。

单元测试

测试PHP应用最流行的方式是单元测试。前面说过，单元测试能单独证实应用中的各个类、方法和函数能正常运行。PHP开发者通常都使用由塞巴斯蒂安·伯格曼（*https://sebastian-bergmann.de/*）开发的单元测试框架PHPUnit（*https://phpunit.de/*）。PHPUnit框架遵守xUnit测试架构。

除了PHPUnit之外还有其他PHP单元测试框架可以使用，例如PHPSpec。不过，大多数流行的PHP框架都使用PHPUnit测试。如果想为PHP组件做贡献，或者自己发布PHP组件，一定要知道如何阅读、编写和运行PHPUnit测试。本章后面会告诉你如何安装PHPUnit，以及如何使用它编写和运行PHP单元测试。

测试驱动开发（TDD）

测试驱动开发的意思是在编写应用代码之前先写测试。我们故意让测试失败，以此描述应用应该具有怎样的表现。开发好应用的功能后，最终测试会成功通过。TDD能帮助我们按照目标开发应用，因此我们要先规划好要开发的功能，想好怎么实现。

不过，并不是说必须在编写代码之前写好所有测试。我们应该先编写一些测试，然后开发相关的功能；再编写一些测试，然后开发功能。像这样一直循环下去。TDD是一种迭代开发方式，小步向前，直到开发完整个应用。

行为驱动开发（BDD）

行为驱动开发的意思是编写故事，描述应用的表现。BDD分为两种类型：SpecBDD和StoryBDD。

SpecBDD是一种单元测试，使用人类能读懂的流畅语言描述应用的实现方式。SpecBDD的作用和PHPUnit这样的单元测试工具一样。不过，PHPUnit使用xUnit架构，而SpecBDD使用人类能读懂的故事描述行为。例如，可能会把某个PHPUnit测试命名为`testRenderTemplate()`，把等价的SpecBDD测试命名为`itRendersTheTemplate()`，而且这个SpecBDD测试可能会使用一些辅助方法，例如`$this->shouldReturn()`、`$this->shouldBe()`和`$this- >shouldThrow()`。和xUnit工具相比，SpecBDD使用的语言更易于阅读和理解。最流行的SpecBDD测试工具是PHPSpec（*http://www.phpspec.net/*）。

StoryBDD和SpecBDD一样，也使用人类能读懂的故事，不过StoryBDD关注更多的是整体行为，而不是低层实现。例如，可能会使用StoryBDD测试确认代码可以生成PDF格式的报告，然后通过电子邮件发送报告；而SpecBDD测试则用于确认有某个类方法，向这个方法传入指定的参数时能正确生成PDF格式的报告。StoryBDD和SpecBDD之间的区别是测试范围不同。StoryBDD测试类似于项目经理的要求（例如，“要能生成报告，然后通过电子邮件发给我”），而SpecBDD测试类似于开发者的要求（例如，“这个类方法要能接收一个数据数组，把数据写入PDF文件”）。StoryBDD和SpecBDD测试工具不是互斥的，而且通常把二者结合在一起使用，编写更全面的测试。我们通常会和项目经理坐在一起编写StoryBDD测试，定义应用的一般行为，设计和开发应用的具体功能时自己再编写SpecBDD测试。最流行的StoryBDD测试工具是Behat（*http://behat.org/*）。

建议： StoryBDD测试用于描述业务逻辑，而非具体的实现方式。好的StoryBDD测试会确认“把物品添加到购物车之后，购物车中的物品总数增加了”，而不好的StoryBDD测试会确认“向*/cart*这个URL发起一个HTTP PUT请求，并把请求主体设为`uct_id=1&quantity=2`，此时购物车中的物品总数增加了”。前一个测试更具一般性，只描述了抽象的业务逻辑；而后一个测试太具体，描述了特定的实现方式。

PHPUnit

下面讨论如何安装PHPUnit，以及如何使用PHPUnit编写和运行测试。安装基础设施要花点时间，不过安装好之后，使用PHPUnit编写和运行测试都特别简单。在详细说明PHPUnit之前，我们先简单介绍一些术语。PHPUnit测试在一起组成测试用例（test case），测试用例在一起组成测试组件（test suite）。PHPUnit会使用测试运行程序（test runner）运行测试组件。

一个测试用例是一个PHP类，这个类扩展自`PHPUnit_Framework_TestCase`类。测试用例中有一些以test开头的公开方法，一个方法是一个测试，在方法中我们断言会发生特定的事情。断言可能通过，也可能失败。我们的目标是让所有断言都通过。

建议： 测试用例的类名必须以`Test`结尾，而且所在的文件名必须以*Test.php*结尾。例如，测试用例的类名为`FooTest`，保存这个类的文件名为*FooTest.php*。

测试组件由一系列相关的测试用例组成。如果测试一个PHP组件，通常只会有一个测试组件；如果测试一个由很多不同的子系统或组件构成的大型PHP应用，最好使用多个测试组件组织测试。

测试运行程序的作用正如其名所示，是PHPUnit运行测试组件并输出结果的工具。

PHPUnit默认使用的是命令行运行程序，这个运行程序在终端应用中使用phpunit命令调用。

目录结构

我喜欢按照下述目录结构组织PHP项目。项目的最顶层目录中有个*src/*目录，用于保存源码；最顶层目录中还有个*tests/*目录，用于保存测试。下面是这种目录结构的示例：

```
src/
tests/
    bootstrap.php
composer.json
phpunit.xml
.travis.yml
```

src/

这个目录中保存的是PHP项目的源码（即各个PHP类）。

tests/

这个目录中保存的是PHP项目的PHPUnit测试。这个目录中有个*bootstrap.php*文件，PHPUnit运行单元测试之前要引入这个文件。

composer.json

这个文件用于列出使用Composer管理的PHP项目依赖，其中包含PHPUnit测试框架。

phpunit.xml

这个文件用于配置PHPUnit的测试运行程序。

.travis.yml

这个文件用于配置持续测试Web服务Travis CI。

注意： 如果在GitHub中查看你最喜欢PHP组件或框架的源码，会发现它们用了类似的组织方式。

安装PHPUnit

首先我们要安装PHPUnit和Xdebug分析器。PHPUnit用于运行测试，Xdebug用于生成有用的覆盖度信息。安装PHPUnit测试框架最简单的方式是使用Composer。打开终端应用，进入项目的最顶层目录，然后执行下述命令：

```
composer require --dev phpunit/phpunit
```

这个命令会把PHPUnit测试框架下载到项目的*vendor/*目录中，然后更新项目的*composer.*

*json*文件，把phpunit/phpunit作为一个项目依赖列出来。二进制文件phpunit安装在项目的*vendor/bin/*目录中。我们可以把这个目录添加到环境路径中，或者调用PHPUnit的命令行测试运行程序时引用vendor/bin/phpunit。PHPUnit框架的类会与项目中其他使用Composer管理的依赖一起，自动加载到PHP应用中。

安装Xdebug

Xdebug是个PHP扩展，有点难安装。如果你是使用包管理器安装PHP的，可以使用相同的方式安装Xdebug（如示例10-1所示）。

示例10-1：安装Xdebug的方式

```
# Ubuntu
sudo apt-get install php5-xdebug

# CentOS
sudo yum -y --enablerepo=epel,remi,remi-php56 install php-xdebug
```

如果PHP是编译源码安装的，要使用pecl命令安装Xdebug扩展：

```
pecl install xdebug
```

然后要更新*php.ini*配置文件，设定编译后Xdebug扩展的路径。

建议： 可以使用php-config --extension-dir或php -i | grep extension_dir命令找出PHP扩展的目录。

我们要在*php.ini*文件中添加下面这行设置。记得要把PHP扩展的路径改为真实的路径。

```
zend_extension="/PATH/TO/xdebug.so"
```

最后重启PHP，这样就行了。我们会在第11章讨论Xdebug分析器。

配置PHPUnit

现在我们要在项目的*phpunit.xml*文件中配置PHPUnit。

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="tests/bootstrap.php">
    <testsuites>
        <testsuite name="whovian">
            <directory suffix="Test.php">tests</directory>
        </testsuite>
    </testsuites>

    <filter>
        <whitelist>
```

```
            <directory>src</directory>
        </whitelist>
    </filter>
</phpunit>
```

PHPUnit的测试运行程序在XML根元素<phpunit>的属性中设置。我觉得最重要的设置是bootstrap，这个设置指定一个PHP文件的路径（相对于*phpunit.xml*文件），PHPUnit的测试运行程序在运行测试之前会引入这个文件。我们要在这个*bootstrap.php*文件中自动加载应用中使用Composer管理的依赖，以便在PHPUnit测试中使用。我们还要在*bootstrap.php*文件中指定测试组件的路径（即保存相关测试用例的目录），PHPUnit会运行这个目录中所有文件名以*Test.php*结尾的PHP文件。最后，在这个配置文件的<filter>元素中要列出代码覆盖度分析涵盖的目录。在上述示例XML中，<whitelist>元素的作用是告诉PHPUnit只分析*src/*目录中代码的覆盖度。

这个配置文件的目的是让我们在一处设置PHPUnit。这样在本地开发时，不用每次调用phpunit命令行运行程序都指定这些设置。有了这个配置文件，我们还能把相同的PHPUnit设置应用于远程持续测试服务中，例如Travis CI。更新*phpunit.xml*配置文件之后还要更新*tests/bootstrap.php*文件，写入下述代码：

```
<?php
// 导入Composer自动加载器
require dirname(__DIR__) . '/vendor/autoload.php';
```

建议： 运行PHPUnit测试之前要先安装使用Composer管理的依赖。

Whovian类

编写单元测试之前要有可测试的代码。下面我们虚构一个名为Whovian的PHP类。这个类特别钟爱某个BBC电视节目。我们把这个类的定义保存在*src/Whovian.php*文件中。

```
<?php
class Whovian
{
    /**
     * @var string
     */
    protected $favoriteDoctor;

    /**
     * Constructor
     * @param  string $favoriteDoctor
     */
    public function __construct($favoriteDoctor)
    {
        $this->favoriteDoctor = (string)$favoriteDoctor;
```

```
    }

    /**
     * Say
     * @return string
     */
    public function say()
    {
        return 'The best doctor is ' . $this->favoriteDoctor;
    }

    /**
     * Respond to
     * @param  string $input
     * @return string
     * @throws \Exception
     */
    public function respondTo($input)
    {
        $input = strtolower($input);
        $myDoctor = strtolower($this->favoriteDoctor);

        if (strpos($input, $myDoctor) === false) {
            throw new Exception(
                sprintf(
                    'No way! %s is the best doctor ever!',
                    $this->favoriteDoctor
                )
            );
        }

        return 'I agree!';
    }
}
```

Whovian类的构造方法会设置实例最喜欢的医生。say()方法返回一个字符串，说出实例最喜欢的医生。respondTo()方法从另一个Whovian实例中获取最喜欢的医生，然后做出相应的回应。

WhovianTest测试用例

Whovian类的单元测试保存在*test/WhovianTest.php*文件中。我们把一系列相关的测试叫测试组件。在这个例子中，*test/*目录中的所有测试都属于同一个测试组件，这个目录中每个文件里的类是一个测试用例，类中每个以test开头的方法（例如testThis或testThat）是单独的测试，每个测试都使用断言验证指定的条件。断言可能会通过，也可能会失败。

注意： PHPUnit的网站（*http://bit.ly/php-unit*）中列出了一些断言。有些断言没有文档，我们可以在GitHub中查看源码（*http://bit.ly/phpu-gh*），找到所有可用的断言。

一个PHPUnit测试用例是一个类，这个类扩展自PHPUnit_Framework_TestCase类。下面我们在*test/WhovianTest.php*文件中声明一个名为WhovianTest的测试用例：

```
<?php
require dirname(__DIR__) . '/src/Whovian.php';

class WhovianTest extends PHPUnit_Framework_TestCase
{
    // 这里是各个测试
}
```

记住，单元测试的目的是验证公开接口的预期行为。因此，我们要测试Whovian类的三个公开方法。我们要先编写一个单元测试，确认有没有把__construct()方法的参数设为实例喜欢的医生。然后，编写一个单元测试确认say()方法的返回值有没有提到实例喜欢的医生。最后，为respondTo()方法编写两个测试：一个测试确认输入值和喜欢的医生一样时，返回值是不是字符串"I agree!"；另一个测试确认输入值和喜欢的医生不一样时，会不会抛出异常。

测试1：__construct()方法

我们编写的第一个测试用于确认构造方法是否设定了Whovian实例最喜欢的医生：

```
public function testSetsDoctorWithConstructor()
{
    $whovian = new Whovian('Peter Capaldi');
    $this->assertAttributeEquals('Peter Capaldi', 'favoriteDoctor', $whovian);
}
```

在这个测试中，我们使用一个字符串参数"Peter Capa ldi"实例化了一个Whovian实例，然后使用PHPUnit提供的断言方法assertAttributeEquals()判断$whovian实例的favoriteDoctor属性是不是等于字符串"Peter Capa ldi"。

注意： PHPUnit提供的断言方法assertAttributeEquals()接收三个参数。第一个参数是期望值，第二个参数是属性名，第三个参数是要检查的对象。assertAttributeEquals()方法的精妙之处在于，可以使用PHP的反射功能检查并验证受保护的属性。

为什么检查最喜欢的医生时我们使用断言方法assertAttributeEquals()，而不使用获取方法（例如getFavoriteDoctor()）呢？因为我们一次只隔离测试一个指定的方法。理想情况下，测试不能依赖其他方法。对这个例子来说，我们要测试的是__construct()方法，我们要验证这个方法能不能把参数的值赋予对象的$favoriteDoctor属性。assertAttributeEquals()这个断言方法能检查对象的内部状态，而且不用依赖某个未测试的获取方法。

测试2：say()方法

接下来，我们要编写测试确认Whovian实例的say()方法会返回一个包含最喜欢的医生名字的字符串：

```
public function testSaysDoctorName()
{
    $whovian = new Whovian('David Tennant');
    $this->assertEquals('The best doctor is David Tennant', $whovian->say());
}
```

比较两个值时，我们使用的是PHPUnit提供的assertEquals()断言方法。这个方法的第一个参数是期望值，第二个参数是要检查的值。

测试3：表示认同的respondTo()方法

现在测试一个Whovian实例认同另一个Whovian实例的喜好时，respondTo()方法的行为：

```
public function testRespondToInAgreement()
{
    $whovian = new Whovian('David Tennant');

    $opinion = 'David Tennant is the best doctor, period';
    $this->assertEquals('I agree!', $whovian->respondTo($opinion));
}
```

这个测试会成功通过，因为Whovian实例的respondTo()方法接收的参数是一个包含它最喜欢的医生名字的字符串。

测试4：表示反对的respondTo()方法

可是，如果Whovian实例反对怎么办？我们要尽快做出反应，因为粉丝就要愤怒了。好吧，其实我们只是抛出一个异常。下面测试这种行为：

```
/**
 * @expectedException Exception
 */
public function testRespondToInDisagreement()
{
    $whovian = new Whovian('David Tennant');

    $opinion = 'No way. Matt Smith was awesome!';
    $whovian->respondTo($opinion);
}
```

如果这个测试抛出异常，测试就会通过；否则，测试失败。我们可以使用@expectedException注解测试这种情况。

注意： PHPUnit提供了几个用于控制指定测试的注解。PHPUnit注解的详细信息参见PHPUnit的文档（*http://bit.ly/phpunit-docs*）。

运行测试

每编写一个测试，我们都应该运行测试组件，确认测试能通过。运行测试的方式很简单：打开终端应用，进入项目的最顶层目录（*phpunit.xml*配置文件所在的目录），调用Composer安装的PHPUnit二进制文件。执行下述命令启动PHPUnit的测试运行程序：

```
vendor/bin/phpunit -c phpunit.xml
```

-c选项的作用是指定PHPUnit配置文件的路径。执行这个命令后，终端里会显示PHPUnit命令行测试运行程序得到的结果，如图10-1所示。

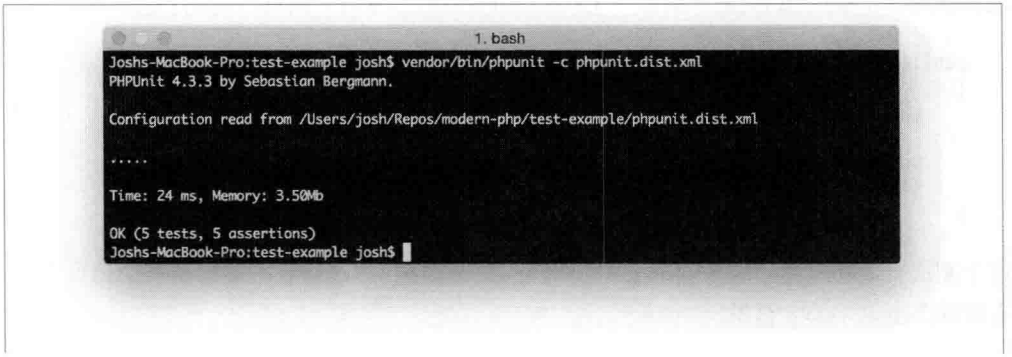


图10-1：PHPUnit的测试结果

这个结果告诉我们：

1. PHPUnit读取了指定的配置文件。
2. PHPUnit完成测试用了24毫秒。
3. PHPUnit用了3.5MB内存。
4. PHPUnit成功运行了五个测试和五个断言。

代码覆盖度

我们知道PHPUnit测试通过了，可是我们能确定做了足够的测试了吗？我们可能会忘了测试什么。我们可以通过PHPUnit的代码覆盖度报告（见图10-2）查看具体测试了哪些代码（以及没测试的代码）。我们在PHPUnit配置文件中指定了源码文件的路径，PHPUnit的代码覆盖度报告会涵盖白名单中的所有PHP文件。我们可以像下面这样调用PHPUnit的测试运行程序，让PHPUnit每次都生成代码覆盖度报告：

```
vendor/bin/phpunit -c phpunit.xml --coverage-html coverage
```

这个命令和前面使用的一样，不过我们新加了`--coverage-html`选项。这个选项的值是保存代码覆盖度报告的目录路径。执行这个命令后，在Web浏览器中打开新生成的*coverage/index.html*文件，就能看到代码覆盖度报告。理想情况下，我们希望看到100%的覆盖度。然而，100%的覆盖度并不现实，而且绝不是我们追求的目标。多少覆盖度算好呢？每个人心中都有一杆秤，而且不同的项目也有不同的要求。

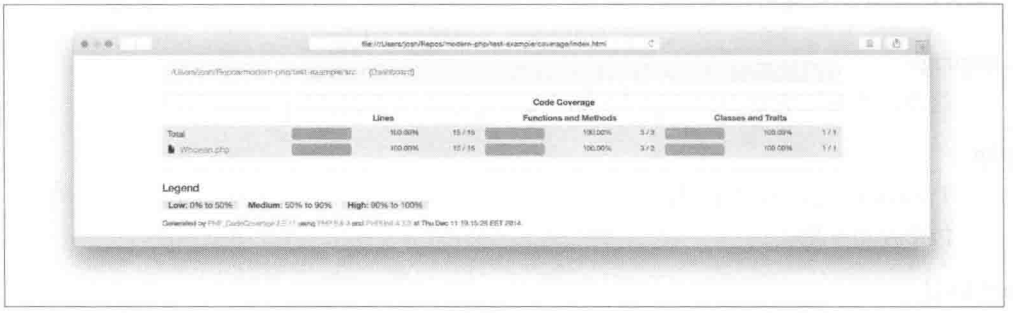


图10-2：PHPUnit生成的代码覆盖度报告

建议： 我们应该把PHPUnit生成的代码覆盖度报告当成一个参考，以此改进代码，而不能一味追求更高的代码覆盖度。

使用Travis CI持续测试

有时，最好的PHP开发者也会忘记编写测试。因此我们要自动测试。最好的测试和好的备份策略一样，眼不见心不烦。测试应该自动运行。我最喜欢的持续测试服务是Travis CI（*https://travis-ci.org/*），因为这个服务原生提供了钩子，可以集成GitHub仓库，每次把代码推送到GitHub，都能在Travis CI中运行应用的测试。而且，Travis CI还能在多个PHP版本中运行测试。

设置

如果你以前没用过Travis CI，先访问*https://travis-ci.org*（针对公开仓库）或*https://travis-ci.com*（针对私有仓库），使用GitHub账户登录。然后按照屏幕上的说明，选择要在Travis CI中测试哪个仓库。

接下来，在应用的最顶层目录中创建Travis CI的配置文件，*.travis.yml*。注意，文件名前面有个.号。然后保存这个文件，将其提交到仓库中，再推送到GitHub仓库。下面是Travis CI配置文件的示例：

```
language: php
php:
  - 5.4
  - 5.5
  - 5.6
  - hhvm
install:
  - composer install --no-dev --quiet
script: phpunit -c phpunit.xml --coverage-text
```

Travis CI配置文件使用YAML格式编写，包含下述设置：

`language`

这是应用使用的语言。这里我们设为`php`。这个设置的值区分大小写！

`php`

Travis CI可以在多个PHP版本中运行应用的测试。我们一定要在应用支持的所有PHP版本中测试。

`install`

这是Travis CI运行应用测试之前执行的bash命令。我们使用这个设置让Travis CI安装项目的Composer依赖。我们要使用`--no-dev`选项，不让Composer安装不必要的开发依赖。

`script`

这是Travis CI用来运行应用测试的bash命令。这个设置的默认值是`phpunit`。我们可以通过这个设置覆盖Travis CI默认使用的命令。在这个示例中，我们告诉Travis CI使用我们自定义的PHPUnit配置文件，还要生成纯文本格式的覆盖度报告。

运行

每次把代码推送到GitHub仓库中，Travis CI都会自动运行应用的测试，而且会通过电子邮件把测试结果发给你。很棒吧？当然，还有很多Travis CI设置可以进一步定制Travis CI的测试环境（例如，安装自定义的PHP扩展，使用自定义的初始化设置等）。关于如何配置测试PHP的Travis CI环境，详细信息参见Travis CI的网站（*http://bit.ly/build-php*）。

延伸阅读

下面列出几个链接，以便你进一步学习PHP应用测试：

- *https://phpunit.de/*
- *http://www.phpspec.net/docs/introduction.html*

- *http://behat.org/*
- *https://leanpub.com/grumpy-phpunit*
- *https://leanpub.com/grumpy-testing*
- *http://www.littlehart.net/atthekeyboard/*

接下来

本章我们学习了为什么测试，何时测试，以及如何编写测试。测试应用能增加自信，写出更能预知结果的代码。不过，测试无法分析应用的性能。因此，除了测试之外，我们还要分析应用。下一章会讨论如何分析应用。

第11章

分析

我所说的分析是指分析应用的性能，目的是调试性能问题，确定应用代码的瓶颈所在。也就是说，如果应用运行的慢，我们可以使用分析器找出原因。分析器可以遍历整个PHP调用堆栈，指出调用了哪个函数或方法，以什么顺序调用，调用了多少次，调用时传入了什么参数，以及运行了多长时间。我们还能看到整个应用请求生命周期使用了多少内存和CPU。

什么时候使用分析器

我们无需立即分析PHP应用，遇到难以诊断的性能问题时才需要分析。我们怎么知道有性能问题呢？有些问题很明显（例如，数据库查询用时太长），而其他问题则不那么明显。

我们可以使用基准测试工具，例如Apache Bench（*http://bit.ly/apache-bench*）和Siege（*http://www.joedog.org/siege-home/*），找出性能问题。基准测试工具的作用是从外部测试应用的性能，就像是用户在Web浏览器中访问应用一样。基准测试工具可以设置同时有多少用户和多少请求访问应用中指定的URL。测试结束后，基准测试工具会告诉你应用每秒能承受多少请求（除此之外还有其他统计数据）。如果发现某个URL每秒能承受的请求数很少，可能表明有性能问题。如果性能问题不是特别明显，就要使用分析器。

分析器的种类

分析器分为两类，一种只应该在开发环境中使用，另一种则可以在生产环境中使用。

Xdebug（*http://xdebug.org*）是一个流行的PHP分析工具，由德里克·李桑思开发，不过这个分析器只应该在开发环境中使用，因为分析应用时它会消耗大量系统资源。Xdebug的分析结果人类读不懂，所以需要使用其他应用解析并显示结果。在这方面，KCacheGrind（*http://kcachegrind.sourceforge.net/*）和WinCacheGrind（*http://sourceforge.net/projects/wincachegrind/*）是不错的选择，这两个工具都能形象化地显示Xdebug的分析结果。

XHProf（*http://xhprof.io*）也是一个流行的PHP分析器，由Facebook开发。这个工具在开发环境和生产环境都能使用。XHProf的分析结果人类也读不懂，不过Facebook提供了一个配套的Web应用——XHGUI，用于形象化显示和比较分析结果。本章后面会进一步讨论XHGUI。

注意： Xdebug和XHProf都是PHP扩展，可以使用操作系统的包管理器安装，也可以使用`pecl`安装。

Xdebug

Xdebug是最流行的PHP分析器之一，使用它分析应用的调用堆栈，能轻易找出瓶颈和性能问题。Xdebug的安装方法参见第10章的示例10-1。

配置

Xdebug的配置保存在*php.ini*文件中。下面是我推荐使用的Xdebug配置。记得要修改分析结果的输出目录。保存这些设置后，要重启PHP进程。

```
xdebug.profiler_enable = 0
xdebug.profiler_enable_trigger = 1
xdebug.profiler_output_dir = /path/to/profiler/results
```

`xdebug.profiler_enable = 0`

这么设置是为了不让Xdebug自动运行。我们不想让Xdebug在每次请求时都自动运行，因为这会极大地降低性能，还会阻碍开发。

`xdebug.profiler_enable_trigger = 1`

这么设置是为了在需要时启动Xdebug。我们可以在PHP应用的任何一个URL中加上XDEBUG_PROFILE=1查询参数，在单个请求中启动Xdebug。Xdebug检测到这个查询参数时，会分析当前请求，然后生成报告，将其保存到`xdebug.profiler_output_dir`设置指定的输出目录。

```
xdebug.profiler_output_dir = /path/to/profiler/results
```

这是一个目录的路径，这个目录用于保存分析器生成的报告。如果是复杂的PHP应用，分析器生成的报告可能很大（例如，500MB或更大）。记得要把这个设置的值改为正确的文件系统路径。

建议： 我建议把分析器生成的结果保存在PHP应用的最顶层目录中。这样在开发过程中便于找到并查看分析结果。

触发运行

因为我们把`xdebug.profiler_enable`的值设为了0，所以Xdebug不会自动运行。我们可以在PHP应用的任何一个URL中加上*XDEBUG_PROFILE=1*查询参数，例如*/users/show/1?XDEBUG_PROFILE=1*，在单个请求中触发运行Xdebug。Xdebug检测到`XDEBUG_PROFILE`查询参数时，会在当前请求中启动Xdebug。分析器生成的结果会转储到`xdebug.profiler_output_dir`设置指定的目录中。

分析

Xdebug生成的结果是CacheGrind格式，因此我们要使用兼容CacheGrind的应用查看分析结果。下面是几个查看CacheGrind格式文件的优秀应用：

- 在Windows中运行的WinCacheGrind（*http://sourceforge.net/projects/wincachegrind/*）。
- 在Linux中运行的KCacheGrind（*http://kcachegrind.sourceforge.net/*）。
- 在Web浏览器中运行的WebGrind（*http://code.google.com/p/webgrind/*）。

Mac OS X用户可以执行下述命令，使用Homebrew安装KCacheGrind：

```
brew install qcachegrind
```

建议： Homebrew（*http://brew.sh*）是OS X系统的包管理器。附录A会讨论Homebrew。

XHProf

XHProf是个较新的PHP应用分析器，由Facebook开发，在开发环境和生产环境中都能使用。XHProf收集的信息没有Xdebug多，不过消耗的系统资源较少，因此适合在生产环境中使用。

安装

XHProf最简单的安装方式是使用操作系统的包管理器（假设PHP也是使用这种方式安装的）：

```
# Ubuntu
sudo apt-get install build-essential;
sudo pecl install mongo;
sudo pecl install xhprof-beta;

# CentOS
sudo yum groupinstall 'Development Tools';
sudo pecl install mongo;
sudo pecl install xhprof-beta;
```

然后把下面两行添加到*php.ini*文件，再重启PHP进程，加载这两个新扩展：

```
extension=xhprof.so
extension=mongo.so
```

XHGUI

XHProf和XHGUI结合在一起使用效果最好。XHGUI是Facebook为XHProf开发的配套Web应用，用于查看和比较XHProf的分析结果。XHGUI是使用PHP开发的Web应用，需要以下软件的支持：

- Composer
- Git
- MongoDB
- PHP 5.3+
- PHP mongo扩展

假设你已经在系统中安装了所需的这些软件，而且把XHGUI应用放在*/var/sites/xhgui/*目录中。记住，你可能会把XHGUI放到服务器的其他目录中。

```
cd /var/sites;
git clone https://github.com/perftools/xhgui.git;
cd xhgui;
php install.php;
```

XHGUI应用中有个*webroot/*目录，我们要更新Web服务器虚拟主机的文档根目录，将其设为这个目录的路径。

配置

在文本编辑器中打开XHGUI的*config/config.default.php*文件。默认情况下，XHProf只会分析全部HTTP请求的1%，收集这些请求的数据。在生产环境中这个比例没问题，不过你可能想在开发环境中收集更多请求的数据。为此，我们可以编辑*config/config.default.php*文件中的下面几行，让XHProf收集更多请求的数据。我们可以把下面几行：

```
'profiler.enable' => function() {
    return rand(0, 100) === 42;
},
```

改为：

```
'profiler.enable' => function() {
    return true; // <-- 每次请求都运行
},
```

建议： XHProf假设PHP应用运行在一个服务器中，还假设MongoDB数据库无需认证。如果MongoDB服务器需要认证的话，要更新*config/config.default.php*文件中Mongo数据库的连接设置。

触发执行

我们必须在PHP应用执行的第一个文件的最顶部引入XHGUI应用的*external/header.php*文件。实现这一要求最简单的方法是使用PHP的`auto_prepend_file`初始化设置。我们可以在*php.ini*配置文件中做下述设置：

```
auto_prepend_file = /var/sites/xhgui/external/header.php
```

或者在nginx虚拟主机的配置中做下述设置：

```
fastcgi_param PHP_VALUE "auto_prepend_file=/var/sites/xhgui/external/header.php";
```

或者在Apache虚拟主机的配置中做下述设置：

```
php_admin_value auto_prepend_file "/var/sites/xhgui/external/header.php"
```

设置好之后重启PHP，现在XHProf会开始收集信息，然后把信息保存到MongoDB数据库中。我们可以打开为XHGUI设置的虚拟主机URL，查看和比较XHProf的分析结果。

New Relic的分析器

另一个流行的PHP分析器是New Relic（*https://newrelic.com/*）。这其实是一个Web服

务，New Relic提供了一个自定义的操作系统守护进程和一个PHP扩展，将其挂在PHP应用上之后，会把收集的数据发给这个Web服务。与Xdebug和XHProf不同的是，New Relic的PHP分析器不是免费的。虽然如此，我依然喜欢New Relic。如果你有足够的预算，推荐你也使用。与XHProf一样，New Relic的PHP分析器适合在生产环境中使用。New Relic提供了特别精美的在线控制面板，几乎可以实时查看应用的性能。更多信息参见New Relic的网站（*http://bit.ly/new-relic-php*）。

Blackfire分析器

我写作本书时，Symfony正在测试一个新的PHP分析器，叫Blackfire（*https://blackfire.io*）。这个分析器提供了独特的可视化工具，以便我们找出应用的瓶颈。我听说这个分析器很好，可以替代Xdebug和XHProf。你可以关注一下。

延伸阅读

希望我在本章介绍了足够的PHP分析知识，让你能轻松地找到、安装并使用对你的应用来说最适合的PHP分析器。下面给出几个链接，以便你进一步学习PHP分析：

- *http://www.sitepoint.com/the-need-for-speed-profiling-with-xhprof-and-xhgui/*
- *https://blog.engineyard.com/2014/profiling-with-xhprof-xhgui-part-1*
- *https://blog.engineyard.com/2014/profiling-with-xhprof-xhgui-part-2*
- *https://blog.engineyard.com/2014/profiling-with-xhprof-xhgui-part-3*

接下来

目前为止，我们对现代的PHP做了很多讨论，包括新特性、良好实践、配置、调优、部署、测试和分析。希望你的脑中已经有了大量有趣的创意，准备在你的下一个PHP应用中来实现。

现在我要花几分钟讨论PHP的未来。PHP生态系统正在发生很多事情，就在我说这话的时候，PHP的未来也在演化着，这得益于一些有远见的项目，例如PHP 7（*https://wiki.php.net/rfc/php7timeline*）、HHVM（*http://hhvm.com*）、Hack（*http://hacklang.org*）和PHP-FIG（*http://www.php-fig.org*）。下面我们要探讨HHVM和Hack，看看二者对PHP的未来有什么影响。

第12章

HHVM和Hack

不管你对Facebook有什么看法，反正我觉得他们很成功。在过去的几年中，Facebook开源组（*https://code.facebook.com/projects/*）开发了几个重要的项目，其中两个项目在PHP社区中引起了重大反响。

第一个项目是HHVM（*http://hhvm.com*），全称是Hip Hop Virtual Machine。HHVM是一个PHP引擎，于2013年10月发布。HHVM的即时（Just in Time，简称JIT）编译器性能比PHP-FPM好很多倍。WP Engine最近迁移到HHVM了，他们发现WordPress的运行速度提高了3.9倍（*http://bit.ly/engine-box*）。MediaWiki也换用了HHVM，他们发现响应时间和吞吐量都有了巨大的提升（*http://www.mediawiki.org/wiki/HHVM*）。

第二个项目是Hack（*http://hacklang.org*），这是一门新的服务器端语言，在PHP语言的基础上修改而来。Hack基本上兼容PHP代码，不过扩展了PHP语言，增加了一些特性：严格类型，新的数据结构和实时类型检查服务器。不过，Hack的开发者喜欢把Hack称为PHP的一种方言，而不把它看做一门新语言。

HHVM

自1994年开始，我们说的PHP解释器都是指Zend Engine（*http://www.zend.com/en/community/php*）。以前，Zend Engine就是PHP，当时只有这么一个PHP解释器。2004年2月4日，马克·扎克伯格创建了Thefacebook。扎克伯格先生和这个不断发展的公司主要使用PHP编写Facebook，因为这门语言易于学习，部署也简单。Facebook借助PHP语言迅速聚集了一些新开发者，这些开发者不断壮大、改革和迭代Facebook的平台。

时间飞逝，最终Facebook变成了名符其实的帝国。Facebook的基础设置很庞大，大到传

统的Zend Engine变成了开发者的瓶颈。Facebook的用户基数特别大，而且一直在增长（截至2007年，地球上每10个人中就有一个人在使用Facebook）。因此，Facebook要找到一种提升性能的方式，而不能只靠建造更多的数据中心和购买更多的服务器。

Facebook对PHP的改进

PHP是传统意义上的解释型语言，而不是编译型语言。因此，在命令行或Web服务器调用解释器解释PHP代码之前，PHP代码就是PHP代码。PHP解释器会解释PHP脚本，把代码转换成一系列Zend操作码（机器码指令，*http://php.net/manual/internals2.opcodes.php*），再把这些操作码交给Zend Engine执行。不过，解释型语言执行的速度比编译型语言慢很多，因为每次执行解释型语言编写的代码时都要将其转换成机器码，消耗额外的系统资源。Facebook意识到了这个性能瓶颈，于2010年开始开发一个叫HPHPc的编译器，把PHP代码编译成C++代码。

HPHPc编译器先把PHP代码编译成C++代码，再把C++代码编译成可执行文件，最后把这个可执行文件部署到生产服务器。HPHPc基本上是成功的，它提升了Facebook的性能，降低了Facebook服务器的负担。可是，HPHPc对性能的提升已经到顶了，而且不能完全兼容PHP语言，还需要花时间编译，因此对开发者来说，反馈回路太长。Facebook需要一种混合解决方案，既要进一步提升性能，又要让开发速度更快，省去昂贵的编译时间。

于是，Facebook开始开发下一代HPHPc，即HHVM。HHVM先把PHP代码转换成一种字节码中间格式，而且会缓存转换得到的字节码，然后使用JIT编译器转换并优化缓存的字节码，将其变成x86_64机器码。HHVM的JIT编译器提供了很多低层性能优化措施，这些优化措施是把PHP代码直接编译成C++代码的HPHPc所不具备的。HHVM还为开发者提供了快速的反馈回路，因为只有Web服务器请求PHP脚本时，HHVM才会即时把字节码编译成机器码，这一点和传统的解释型语言很像。更惊人的是，2012年10月，HHVM的性能超过了HPHPc（*http://bit.ly/hhvm-evo*），而且仍在不断提升（见图12-1）。

HHVM的性能超越HPHPc之后不久，HPHPc就被废弃了。现在，Facebook的首选PHP解释器是HHVM。

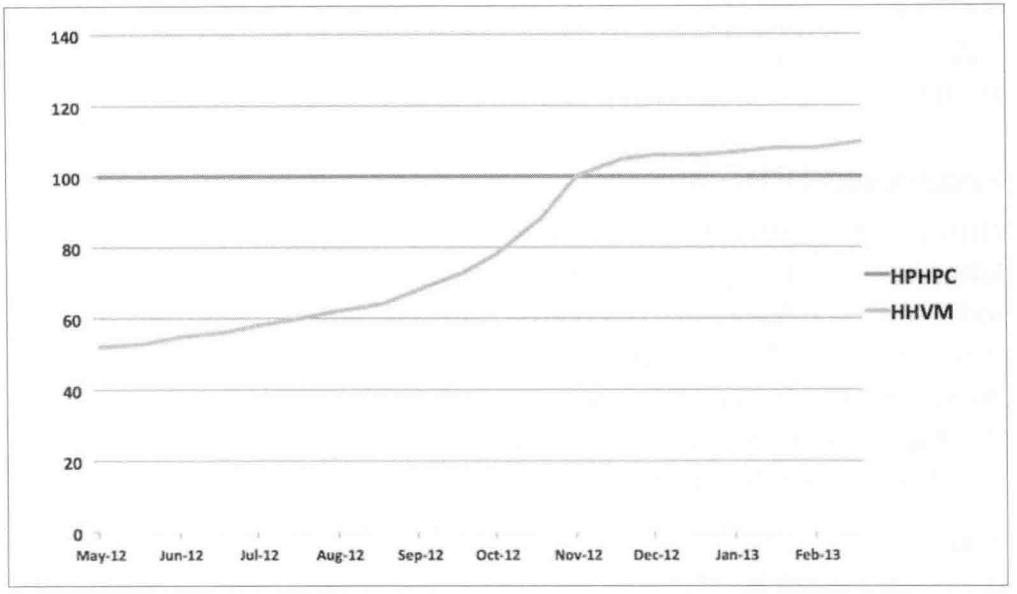


图12-1：HHVM和HPHPc的性能比较（http://bit.ly/hhvm-evo）

注意：别被HHVM吓住了！HHVM的实现可能很复杂，可是说到底，HHVM只不过是我们熟悉的php和php-fpm二进制文件的替代品：

- 我们在命令行使用hhvm二进制文件执行PHP脚本，就像php二进制文件一样。
- 我们使用hhvm二进制文件创建FastCGI服务器，就像php-fpm二进制文件一样。
- HHVM与传统的Zend Engine一样，也是用*php.ini*配置文件，而且使用的初始化指令都一样。
- HHVM原生支持很多常用的PHP扩展。

HHVM和Zend Engine是等价的

Facebook一开始开发的HPHPc编译器不完全兼容PHP语言（即Zend Engine）。Facebook追求的是完全等价，因此HHVM可以直接替代Zend Engine。

Facebook使用大多数流行的PHP框架测试了HHVM，确保HHVM能兼容使用PHP 5编写的真实代码。Facebook已经实现了几乎100%的兼容性。如今，Facebook把注意力转移到用户报告的问题上了，用户可以在HHVM的问题追踪平台（*http://bit.ly/fb-hhvm*）中报告尚未解决的边缘问题。HHVM还没100%兼容传统的Zend Engine，不过每天都在向这个目标靠近。Facebook、百度和维基百科都已经在生产环境使用HHVM了。HHVM还能运行WordPress、Drupal和其他很多流行的PHP框架。

HHVM适合我使用吗？

HHVM不适合所有人使用。提升性能有更容易的方式，例如，减少HTTP请求数和优化数据库查询都是易于实现的方式，能显著提升应用的性能，减少响应时间。如果你还没使用这些优化措施，考虑使用HHVM之前先做这些优化吧。Facebook的HHVM是为已经做了这些优化措施之后仍想进一步加速应用的开发者准备的。如果你觉得自己需要HHVM，可以参考下面给出的资源，再做最后决定：

扩展（http://bit.ly/fb-extensns）
: 查看HHVM兼容的PHP扩展。

框架的等价性（http://hhvm.com/frameworks/）
: 追踪HHVM对最流行的PHP框架的兼容性。

问题追踪平台（http://bit.ly/fb-hhvm）
: 追踪未解决的HHVM问题。

常见问题（https://github.com/facebook/hhvm/wiki/FAQ）
: 阅读HHVM的常见问题。

博客（http://hhvm.com/blog）
: 关注最近的HHVM新闻。

安装

在大多数流行的Linux发行版中，HHVM都易于安装。HHVM起初是为Ubuntu（我喜欢这个Linux发行版）开发的，所以在后面的示例中我都使用Ubuntu。

注意： Facebook为其他Linux发行版提供了预先构建好的包，例如Debian和Fedora。在其他Linux发行版中可以构建源码安装HHVM。

按照Facebook的安装说明（http://bit.ly/fb-prebuilt），在最新版Ubuntu中可以使用Aptitude包管理器安装HHVM，如下所示：

```
wget -O - \
  http://dl.hhvm.com/conf/hhvm.gpg.key |
  sudo apt-key add -;
echo deb \
  http://dl.hhvm.com/ubuntu trusty main | sudo tee /etc/apt/sources.list.d/hhvm.list;
sudo apt-get update;
sudo apt-get install hhvm;
```

如果觉得自己运气不错，可以把最后一行替换成下面这行命令，安装最新的每日构建版：

```
sudo apt-get install hhvm-nightly;
```

上述命令首先添加HHVM的GNU隐私卫士（GNU Privacy Guard，GPG）公钥，用来验证包。然后把HHVM的包仓库添加到本地仓库列表中。最后，像安装其他软件包一样，使用Aptitude安装HHVM。HHVM二进制文件的路径是*/usr/bin/hhvm*。

配置

HHVM和Zend Engine一样，也使用*php.ini*配置文件。这个文件默认的路径是*/etc/hhvm/php.ini*，其中有很多初始化设置与Zend Engine使用的一样。HHVM在*php.ini*文件中使用的全部指令参见*http://docs.hhvm.com/manual/ini.list.php*。

如果把HHVM当成FastCGI服务器使用，要把服务器相关的初始化指令添加到*/etc/hhvm/server.ini*文件中。HHVM服务器的全部指令参见*https://github.com/facebook/hhvm/wiki/INI-Settings*。HHVM的维基缺少细节，因此你可能需要HHVM社区的支持：

- StackOverflow（*http://stackoverflow.com/questions/tagged/hhvm*）。
- IRC频道（*http://webchat.freenode.net/?channels=hhvm*）。
- Facebook粉丝页（*https://www.facebook.com/hhvm*）。

默认的*/etc/hhvm/server.ini*文件足够我们开始动手设置了。这个文件的内容如下：

```
; php options

pid = /var/run/hhvm/pid

; hhvm specific

hhvm.server.port = 9000
hhvm.server.type = fastcgi
hhvm.server.default_document = index.php
hhvm.log.use_log_file = true
hhvm.log.file = /var/log/hhvm/error.log
hhvm.repo.central.path = /var/run/hhvm/hhvm.hhbc
```

最需要注意的设置是`hhvm.server.port = 9000`和`hhvm.server.type = fastcgi`。这两个设置的作用是让HHVM作为FastCGI服务器运行，并且让这个服务器运行在端口9000上。

执行`hhvm`二进制文件时，要使用`-c`选项指定配置文件的路径。如果使用`hhvm`执行命令行脚本，只需要*/etc/hhvm/php.ini*配置文件：

```
hhvm -c /etc/hhvm/php.ini my-script.php
```

如果使用`hhvm`二进制文件启动FastCGI服务器，*/etc/hhvm/php.ini*和*/etc/hhvm/server.ini*两

个文件都需要：

```
hhvm -m server -c /etc/hhvm/php.ini -c /etc/hhvm/server.ini
```

扩展

HHVM不能使用为Zend Engine编译的PHP扩展，除非扩展使用了Facebook的Zend扩展源码兼容层（*http://bit.ly/ext-zen-comp*）。不过，幸好我们使用的大多数PHP扩展原本都支持HHVM。其他第三方PHP扩展（例如GeoIP扩展）则可以单独编译，然后作为动态扩展载入HHVM。HHVM在GitHub中的维基（*http://bit.ly/int-extension*）列出了兼容HHVM的PHP扩展。

使用Supervisord监控HHVM

HHVM可以作为生产服务器使用，但并不完美。我建议使用Supervisord（*http://supervisord.org*）监控HHVM的主进程。Supervisord是进程监控程序，引导系统时会启动HHVM进程，HHVM崩溃后还会自动重启HHVM进程。

建议：如果不熟悉Supervisord，可以阅读克里斯·菲道（*http://fideloper.com*）写的优秀教程（*http://bit.ly/c-fidao*）。

如果还没安装Supervisord，执行下述命令安装：

```
sudo apt-get install supervisor
```

然后，确保*/etc/supervisor/supervisord.conf*配置文件中有下面两行：

```
[include]
files = /etc/supervisor/conf.d/*.conf
```

有了这两行，我们可以在*/etc/supervisor/conf.d/*目录中为每个被监控的应用创建配置文件。接下来，创建*/etc/supervisor/conf.d/hhvm.conf*文件，写入下述内容：

```
[program:hhvm]
command=/usr/bin/hhvm -m server -c /etc/hhvm/php.ini -c /etc/hhvm/server.ini
directory=/home/deploy
autostart=true
autorestart=true
startretries=3
stderr_logfile=/home/deploy/logs/hhvm.err.log
stdout_logfile=/home/deploy/logs/hhvm.out.log
user=deploy
```

其中最重要的设置有：

`command`

Supervisord使用这个命令启动HHVM进程。我们使用-m选项的目的是，让HHVM运行在服务器模式中。我们还使用了-c选项，指定HHVM的php.ini文件和server.ini文件的路径。

`autostart`

这个设置让Supervisord进程启动时（例如引导系统时）启动HHVM进程。

`autorestart`

这个设置让Supervisord在HHVM崩溃时重启HHVM进程。

`startretries`

Supervisord认定HHVM进程崩溃之前尝试启动HHVM进程的次数。

`user`

这是拥有HHVM进程的用户。为了安全，我建议使用没有特殊权限的用户。在这个示例中，我使用的是示例7-1中创建的deploy用户，这个用户没有特殊权限。

警告： 我们要手动创建*/home/deploy/logs*目录，因为Supervisord不会代我们创建。

编辑好Supervisord的配置文件之后，执行下述两个命令，重新加载配置文件，让改动生效：

```
sudo supervisorctl reread;
sudo supervisorctl update;
```

我们可以执行下述命令，查看Supervisord监控的所有进程：

```
sudo supervisorctl
```

如下述代码所示，我们可以启动、停止或重启Supervisord监控的单个程序。在下述代码中，`hhvm`是*/etc/supervisor/conf.d/hhvm.conf*文件顶部指定的程序名。

```
sudo supervisorctl start hhvm;
sudo supervisorctl stop hhvm;
sudo supervisorctl restart hhvm;
```

目前，我们安装了HHVM，还使用Supervisord对HHVM做了监控。我们还需要一个Web服务器，把请求转发给HHVM。记住，HHVM运行的FastCGI服务器，其作用与第7章讨论的PHP-FPM完全一样。我们要使用HHVM的FastCGI服务器处理发自nginx的PHP请求。

HHVM、FastCGI和Nginx

HHVM通过FastCGI协议与Web服务器（例如nginx）通信。我们要创建一个nginx虚拟主机，把PHP请求转发给HHVM的FastCGI服务器。下面是这样一个nginx虚拟主机的配置示例：

```
server {
    listen 80;
    server_name example.com;
    index index.php;
    client_max_body_size 50M;
    error_log /home/deploy/apps/logs/example.error.log;
    access_log /home/deploy/apps/logs/example.access.log;
    root /home/deploy/apps/example.com/current/public;

    location / {
        try_files $uri $uri/ /index.php$is_args$args;
    }

    location ~ \.php {
        include fastcgi_params;
        fastcgi_index  index.php;
        fastcgi_param  SCRIPT_FILENAME ument_root$fastcgi_script_name;
        fastcgi_pass 127.0.0.1:9000;
    }
}
```

注意： 从这里开始，我假设服务器中安装并运行着nginx。nginx的安装说明参见第7章。

假设你按照第7章的说明安装了nginx，现在创建*/home/deploy/apps/example.com/current/public/index.php*文件，写入如下代码：

```
<?php
phpinfo();
```

`example.com`域名要指向服务器的IP地址，然后在Web浏览器中访问*http://example.com/index.php*。我们应该看到浏览器窗口中有“HipHop”这个单词。

建议： 我们可以在本地电脑的*/etc/hosts*文件中强制让电脑把任何域名指向任何IP地址。例如，下面这行代码让域名`example.com`指向IP地址`192.168.33.10`：

```
192.168.33.10 example.com
```

恭喜！作为FastCGI服务器安装的HHVM可以运行PHP应用了。不过，FastCGI服务器不算什么。你知道什么很酷吗？Hack语言。HHVM也可以运行Hack语言。

Hack语言

Hack（*http://hacklang.org*）是一门服务器端语言，类似PHP，而且可以和PHP无缝集成。Hack的开发者其实把Hack当做PHP的一种方言。为什么Facebook要创造类似于PHP的语言呢？原因有如下几个。Hack语言添加了PHP中没有的新数据结构和接口，有利于节省时间。更重要的是，Hack引入了静态类型，这能帮助我们编写更能预知结果也更稳定的代码。静态类型使用几乎实时的类型检查服务器，能尽早发现开发过程中的问题。

新数据结构和接口，以及静态类型，这些值得我们投入时间学习一门新语言和工具链吗？也许值得。要知道，Facebook有上千名开发者，而且代码基巨大，如果Facebook能优化开发过程中哪怕最小的一部分，也能得到大量回报，不仅能提升开发者的效率，还能让代码基更稳定、性能更好。

我不建议你放下手头的工作，立即把现有项目从PHP移植到Hack。然而，如果你刚开始一个新项目，而且有时间安装和学习Hack，那么尽情去做吧，你肯定会从Hack的数据结构和静态类型中受益的。

从PHP转到Hack

若想把代码从PHP转到Hack，把`<?php`改成`<?hh`即可，就这么简单。下面是PHP代码：

```
<?php
echo "I'm PHP";
```

下面则是等效的Hack代码：

```
<?hh
echo "I'm Hack";
```

Facebook让PHP转换成Hack的过程变得特别简单，因为Facebook知道，转换现有的大型代码基不是件轻松的事。迁移代码基时，我们可以先把`<?php`改成`<?hh`，再引入一些静态类型，然后探索Hack的数据结构。转到Hack的过程是渐进且无痛的，可以一步一步做，Facebook就是使用这种思想设计Hack的。

什么是类型？

在比较动态类型和静态类型之前，或许最好先弄明白什么是类型。大多数PHP程序员把类型理解为赋值给变量的数据形式。例如，表达式`$foo = "bar"`表明，`$foo`变量的值是一个字符串；表达式`$bar = 14`表明，`$bar`变量的值是一个整数。当然了，这两个例子展示了什么是类型，不过没有准确定义类型。

类型是模糊的标签，我们把它赋予应用的属性，为的是表明应用有特定的行为，也证明我们的预期是完全正确的。这个定义改述自克里斯·史密斯对编程语言类型的精彩解说（*http://bit.ly/prog-types*）。

我们可以把类型进一步定义成句法上的注解，这种注解是为了表明程序中变量、参数或返回值的身份。PHP和Hack都有类型注解（或提示）。你或许见过类似下面的代码：

```
<?php
class WidgetContainer
{
    protected $widgets;

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

    public function addWidget(Widget $widget)
    {
        $this->widgets[] = $widget;

        return this;
    }

    public function getWidget($index)
    {
        if (isset($this->widgets[$index]) === false) {
            throw new OutOfRangeException();
        }

        return $this->widgets[$index];
    }
}
```

这是个虚构的例子，使用句法提示强制使用指定的应用属性。例如，在`addWidget()`方法的签名中，我们在`$widget`参数前使用`Widget`提示PHP，期望这个方法参数的值是Widget类的实例。PHP解释器会强制实现这个期望，如果传入的参数不是`Widget`类的实例，代码会出错。在这个示例中，注解的期望值就是类型，即期望`addWidget()`方法的参数只能属于`Widget`类。

前面两个简单的示例（例如`$foo = "bar"`），以及这个`WidgetContainer`示例都展示了什么是类型。第一个示例说明，类型能表明变量的值是一个字符串。不过我们没有明确表明自己的期望，PHP解释器很聪明，能从代码的句法中推导出是字符串类型。第二个示例使用注解创建了一个类型，明确定义`addWidget()`方法的预期行为。此时，PHP解释器不会推导，而是根据明确的提示，强制实现预期的行为。

建议： 类型可不仅仅是身份推导和注解。不过，这是编写PHP和Hack代码时最常见到和最常使用的两个特性。你可以阅读本杰明·C·皮尔斯写的《Types and Programming Languages》一书（*http://bit.ly/tpl-pierce*），进一步学习编程语言的类型。

如果你之前以为PHP的类型提示是静态类型，现在可能正在挠头，因为我刚刚打破了这种想法。静态类型和动态类型都能协助我们编写代码，让代码的表现符合我们的预期，而且二者各有一套类型系统。静态类型和动态类型之间的主要区别在于何时检查程序中的类型，以及如何测试程序使用的类型是否正确。

静态类型

使用静态类型语言编写的程序，其正确的行为通过推导、注解或其他语言专用的方式蕴藏在代码中。如果使用静态类型语言编写的程序能成功编译，我们就能确信这表明程序的表现和编写时预期的一样。程序中使用的类型变成了我们的测试，用于确保程序能满足基本的预期。

注意到我用了“编译”这个词吗？静态类型语言通常都要编译。类型检查和错误报告都委托给语言的编译器完成。这么做很好，因为把应用部署到生产环境之前，编译器在编译时能发现程序中与类型有关的问题。可是，编译型语言的反馈回路太长，因为必须编译程序才能发现问题，而且复杂的程序要花很长时间编译，这会降低开发速度。

使用静态类型语言编写的程序有个优点：通常更稳定，因为编译器的类型检查程序已经证明程序的行为符合预期了。不过，我们仍然要单独编写测试，验证程序的行为是否正确。如果程序编译通过了，只说明程序的行为符合代码的预期，并不说明程序做了本该做的事。不过，使用静态类型语言编写程序的话，不用像动态类型语言那样编写类型相关的单元测试。

动态类型

动态类型与静态类型不同，不能在编译时强制代码的行为，因为程序中的类型直到运行时才会检查。动态类型语言编写的程序通常都是解释执行的。PHP就是动态类型的解释型语言，因此每次执行PHP脚本时，不管是直接在命令行中执行，还是通过Web服务器间接执行，解释器都要准备PHP代码，将其转换成一系列预先定义好的操作码，而后再执行。

既然PHP无需编译，那怎么发现错误呢？答案是，错误在运行时发现。这既是好事也是祸根。说好是因为这样迭代速度快，编写代码后直接运行即可，反馈几乎是即时的。不过，这么做缺少内在的准确性，也没有静态类型检查功能提供的测试。因此单元测试特

别重要，不仅要确认类型正确，还要确认符合预期的行为。我们必须编写测试，涵盖所有可能的行为。如果行为是我们能预料的，这不算什么；可是如果我们不能预料，那么测试就会悲惨地失败。未预料到的行为在运行时会导致PHP出错，我们必须使用友好的消息和合适的日志优雅处理这些错误。

Hack二者兼具

静态类型是Hack最大的卖点。更有趣的是，Hack既支持静态类型，也支持动态类型。记住，Hack基本上能向后兼容普通的PHP。这意味着，Hack支持你预期的所有PHP动态类型特性。Hack实现这一点的基础是HHVM的JIT编译器。JIT编译器中有单独的类型检查程序，会检查Hack代码中的类型。HHVM读取Hack代码后会优化和缓存中间字节码，只在需要时才把Hack文件转换成x86_64机器码。Hack充分利用了两种类型系统的特性，我们通过Hack的类型检查程序得到了静态类型的准确性和安全性（下一节详述），又通过HHVM的JIT编译器得到了动态类型的灵活性和快速迭代。

注意： Hack不支持某些PHP特性，详情参见*http://docs.hhvm.com/manual/hack.unsupported.php*。不过，HHVM执行普通的PHP代码时支持这些特性。

Hack的类型检查功能

Hack自带一个单独的类型检查服务器，这个服务器在后台运行，会实时对代码做类型检查。这个功能太重要了，也是Facebook创造Hack语言的主要原因。Hack的即时类型检查功能提供了静态类型的准确性和安全性，而且反馈回路很短。如果使用Hack，却不使用它的类型检查程序，那就错了。

Hack类型检查程序的设置方式如下。我假设你已经安装了HHVM，而且HHVM正在运行中。如果没有，参见前面介绍HHVM的那一节，安装HHVM。然后在项目的最顶层目录中创建一个空文件，命名为*.hhconfig*。这个文件的作用是告诉Hack的类型检查程序分析哪个目录中的代码。HHVM的类型检查程序会监视这个目录中的文件，如果发现文件系统有变化，会对相应文件中的代码做类型检查。启动Hack类型检查程序的方法是，在项目的最顶层目录或其子目录中执行`hh_client`命令。

Hack的类型检查程序有些缺点。根据Hack的在线文档（*http://bit.ly/hack-hhvm*）：

> 类型检查程序假设有一个全局自动加载器，按需加载所需的类。也就是说，类型检查程序要求所有类和函数的名称都是唯一的，不需要检查导入功能。而且，这个类型检查程序也不支持根据条件定义函数或类，必须不经处理就知道定义了什么，没定义什么。当然，没有全局自动加载器的项目也完全可以使用静态类型检查程序，

不过，使用自动加载器的项目才是预定的使用场景。

这个类型检查程序不支持混用HTML和Hack代码，也无法启用这种复杂的静态分析模式，毕竟很多现代的代码都不使用这种功能了。Hack代码可以通过回送功能或者模板引擎把标记输出给浏览器，遇到更复杂的情况时还可以使用XHP[译注1]。

Hack的模式

Hack代码有三种编写模式：严格模式（`strict`）、局部模式（`partial`）和声明模式（`decl`）。如果使用Hack创建新项目，我建议使用严格模式。如果把现有的PHP代码迁移到Hack，或者项目既使用了PHP，也使用了Hack，可能要使用局部模式。声明模式的作用是在严格模式的Hack代码基中集成之前没有使用类型的PHP代码。模式在文件的最顶部声明，放在Hack或PHP起始标签之后（如下所示）。模式的名称区分大小写。

`<?hh // strict`

严格模式要求所有代码都有合适的注解。Hack的类型检查程序会捕获所有可能与类型有关的错误。使用这种模式后，在Hack代码中不能有Hack之外的代码（例如，以前的PHP代码）。使用严格模式之前一定要熟知Hack的类型注解。严格模式有很多要求，例如，所有Hack数组都必须指定类型，不能有未指定类型的数组存在。除此之外，还必须注解函数和方法的返回值类型。

`<?hh // partial`

局部模式（默认的模式）允许在Hack代码中使用还没转换成Hack的PHP代码。局部模式不要求注解函数或方法的所有参数，如果只注解部分参数，Hack的类型检查程序也不会报错。如果刚开始使用Hack，或者转换现有的PHP代码基，使用这个模式或许最好。

`<?php // decl`

声明模式允许严格模式的Hack代码调用未指定类型的代码。如果新编写的Hack代码要依赖以前未指定类型的PHP类，通常会使用这个模式。遇到这种情况时，必须把以前的PHP代码声明为使用这个模式，然后才能在新编写的Hack代码中使用。

Hack的句法

Hack支持为类的属性、方法的参数和返回值注解类型。Hack单独的类型检查程序会根据各个文件的模式检查这些注解。

译注1：XHP是PHP的扩展，由Facebook开发，目的是在PHP中使用XML句法，创建可重用的HTML元素。详情参见https://github.com/facebook/xhp-lib。

建议： 所有可用的类型注解参见Hack的文档（*http://docs.hhvm.com/manual/en/hack.annotations.types.php*）。

我们还以前面的`WidgetContainer`类为例，说明如何使用类型注解。这个类改为Hack代码之后，如下所示：

```
01. <?hh // strict
02. class WidgetContainer
03. {
04.     protected Vector<Widget> $widgets;
05.
06.     public function __construct(array<Widget> $widgets = array())
07.     {
08.         foreach ($widgets as $widget) {
09.             $this->addWidget($widget);
10.         }
11.     }
12.
13.     public function addWidget(Widget $widget) : this
14.     {
15.         $this->widgets[] = $widget;
16.
17.         return this;
18.     }
19.
20.     public function getWidget(int $index) : Widget
21.     {
22.         if ($this->widgets->containsKey($index) === false) {
23.             throw new OutOfRangeException();
24.         }
25.
26.         return $this->widgets[$index];
27.     }
28. }
```

属性的注解

在第4行，我们使用`Vector<Widget>`注解声明这个类的`$widgets`属性。这个注解告诉我们两件事：

- 这个属性的值是一个`Vector`实例（类似于索引为数字的数组）。
- 这个`Vector`实例中的元素只能是`Widget`实例。

参数的注解

如果你已经使用PHP类型提示，可能熟悉这种注解。在第6行，我们使用`array<Widget>`注解`__construct()`方法的参数。这个注解告诉我们两件事：

- 这个参数的值必须是一个数组。
- 这个数组中的元素只能是Widget实例。

与第4行那个属性的注解不同，这个参数的值可以是索引为数字的数组，也可以是关联数组。我们在__construct()方法中迭代这个数组参数中的元素，把各个元素添加到Vector数据结构中。如果希望这个参数是索引为数字的数据或关联数组，可以分别使用array<int, Widget>或array<string, Widget>注解。

返回值类型的注解

在第13行和第20行，我们注解的是方法的返回值类型。addWidget()方法返回的是实例本身（稍后详述）。getWidget()方法返回的是一个新Widget实例。返回值类型的注解在方法签名的结束圆括号之后、方法主体的起始花括号之前声明。

警告： 这个规则有个例外，是__construct()方法。有些人可能觉得构造方法的返回值是void，其实不然。我们不应该注解构造方法的返回值类型。

有些开发者喜欢使用方法链。为此，类中的方法要返回实例本身，以便把多个方法链接在一起调用，如下所示：

```
$object->methodOne()->methodTwo();
```

在Hack中注解这种行为的方式是，把方法的返回值类型声明为this。我们在第13行就使用this注解了addWidget()方法。

Hack的数据结构

Hack语言最重要的特性是静态类型。不过，Hack还提供了PHP没有的新数据结构和接口。相比使用普通的PHP变通实现类似的数据结构和接口，使用这些原生的新数据结构和接口可能会节省开发时间。Hack提供的部分新数据结构和接口如下：

- 集合（矢量，映射，集和值对。*http://docs.hhvm.com/manual/en/hack.collections.php*）
- 泛型（*http://docs.hhvm.com/manual/en/hack.generics.php*）
- 枚举（*http://docs.hhvm.com/manual/en/hack.enums.php*）
- 形状（*http://docs.hhvm.com/manual/en/hack.shapes.php*）
- 元组（*http://docs.hhvm.com/manual/en/hack.tuples.php*）

这些数据结构中有一些补充了PHP的功能，有一些让PHP的功能更清晰易懂。例如，Hack的集合接口让PHP含混不清的数组更清晰易懂；使用泛型创建的数据结构类型固定，用于处理同质的值，但是具体的类型在创建泛型类的实例时才能推导出来，从而避免在类中使用PHP的`instanceof`方法强制检查类型；有了枚举，无需借助抽象类就能方便地创建一系列具名常量；形状用于检查键固定的数据结构中的类型；元组则用于创建长度不变的数组。

不要觉得非得一股脑使用所有这些数据结构。我承认，有些是小众数据结构，作用有限；有些数据结构之间还有功能重复（和增强）。我建议你先熟知有哪些可用的数据结构，在真正需要时再使用。

建议： 我觉得最有用的Hack数据结构是各种集合接口。这些接口的行为比PHP的数组更恰当，也更能预知结果。所以，最好使用集合，别使用PHP的数组。

HHVM/Hack与PHP的比较

既然HHVM和Hack如此出色，为什么还要使用PHP呢？经常有人问我这个问题。还有人问我，PHP会退出历史舞台吗，什么时候退出？这些问题的答案不能一概而论，要具体问题具体分析。

HHVM是Zend Engine这个传统的PHP运行时第一个真正的竞争对手。自PHP 5.x开始，很多真实的基准测试都表明，HHVM的性能比Zend Engine好，而且HHVM更能合理使用内存。我想这让PHP核心开发团队非常吃惊。其实，HHVM的出现或许只是为了让PHP提起兴趣，关注提升性能和减少内存使用量。PHP核心开发团队已经着手开发PHP 7（*https://wiki.php.net/rfc/php7timeline*），按计划，这个版本很快就会发布。开发团队承诺，PHP 7的性能即便不比HHVM好，也能与之持平。任何人都不能保证PHP 7能做到这一点，可是，HHVM的真正作用是带来的竞争，而竞争对所有人都有好处。HHVM和Zend Engine都会改进，从中受益的是PHP开发者。HHVM和Zend Engine没必要争个输赢，我相信二者会共存，不断竞争。

我觉得Hack语言比PHP好很多，原因有如下几个。首先，Hack语言是Facebook开发出来解决具体需求的，有针对性，目标明确，而不是由委员会开发的。相比之下，PHP语言是在很长一段时间内不断进化而来的。PHP能解决很多需求，而且由一个委员会控制，但是这个委员会是出了名的不和谐。从PHP 5.x起，Hack语言借助严格的类型检查和对旧PHP代码的支持，成为更好的选择。我相信，Hack中很多最好的功能最终都会出现在PHP中，反之亦然。事实上，Hack语言的开发团队说过，他们有意以后继续保持与Zend

Engine的兼容性。还是那句话，我相信竞争对这两个语言都有好处，而且它们会享受这种共存的关系。

例如，这种共存关系衍生出了官方的PHP规范。不久之前，除了Zend Engine之外，PHP语言没有其他实现。而HHVM出现之后，几位Facebook的开发者制定了一份PHP语言规范（*http://bit.ly/fb-spec*）。这份规范是PHP社区的重大进展，而且能确保现在和将来的PHP实现（Zend Engine，HHVM等）都支持相同的基本语言特性。

注意：官方的PHP 规范在GitHub中，地址是*https://github.com/php/php-langspec*。

延伸阅读

我们在短时间内接触了很多HHVM和Hack语言的知识，可惜本书没有足够的篇幅详细说明二者。不过，我会给出一些有用的资源，如下所示：

- *http://hhvm.com*
- *http://hacklang.org*
- Twitter 中的 @ptarjan
- Twitter 中的 @SaraMG
- Twitter 中的 @HipHopVM
- Twitter 中的 @HackLang

第13章

社区

PHP社区是我们最宝贵的资源。PHP社区多种多样，生气勃勃，遍布全球。我建议你参与到PHP社区中，从其他PHP开发者身上学习，也把自己的经验分享给别人。我们始终有更多的知识要学习，而融入PHP社区是不断学习的最好方式。这也是结识和帮助其他开发者的好方式。

本地PHP用户组

我建议先找到并加入本地PHP用户组（PHP User Group，PUG）。很多城市都有PHP用户组。你可以访问*http://php.ug*查找本地PUG。参加本地PUG是结识和联系本地社区中同辈PHP开发者的最好机会。

如果附近没有PUG，还有几个其他选择。你可以自己组织PUG。除非你生活在丛林中，否则我敢说附近一定有志趣相投的PHP开发者愿意加入PUG。除此之外，还可以加入NomadPHP（*https://nomadphp.com*）。这是一个在线用户组，每月都会邀请讲师，还会组织微型演讲，涵盖各种PHP特性和实践。

会议

每年都有众多PHP会议举办。会议是结识并与PHP社区中牛人交流的绝好机会。在会议中，你可以倾听并与PHP讲师和思想领袖交流，还能学到新兴的特性和现代的实践方式。参加会议还可以趁机简短地休假。你可以访问*http://php.net/conferences/*，查看即将举办的PHP会议。

辅导

如果你是初级PHP开发者，需要建议或帮助，可以访问*http://phpmentoring.org*，找一个导师。很多经验丰富的PHP开发者愿意奉献自己的时间，帮助PHP开发新手成长。如果你已经是经验丰富的PHP开发者，可以考虑注册为PHP导师。有很多初级PHP开发者不知道如何开始，或者从哪开始，你的辅导对这些人有极大的价值。

与时俱进

PHP语言经常变化。下面列出一些资源，以便你跟进最新的PHP特性和现代的实践方式。

网站

- *http://php.net*
- *http://php.net/docs.php*
- *http://www.php-fig.org*
- *http://www.phptherightway.com*

邮件列表

- *http://php.net/mailing-lists.php*

Twitter账户

- @official_php
- @phpc

播客

- *http://voicesoftheelephpant.com*
- *http://looselycoupled.info*
- *http://elephantintheroom.io*
- *http://phptownhall.com*
- *http://devhell.info*
- *http://www.phpclasses.org/blog/category/podcast/*

- *http://threedevsandamaybe.com/*

幽默

- @phpbard
- @phpdrama

附录A

安装PHP

Linux

Linux是我最喜欢的开发环境。我有一个安装OS X的Macbook Pro，不过都在Linux虚拟机中开发。在Linux中安装PHP的方法很简单，使用包管理器即可，例如Ubuntu Server中的`aptitude`，CentOS中的`yum`。

现在我们暂且只关注在命令行中使用PHP。我们在第7章讨论如何设置PHP-FPM和nginx Web服务器。

包管理器

大多数Linux发行版都提供了包管理器，例如，Ubuntu使用`aptitude`管理器，CentOS和Red Hat Enterprise Linux（简称RHEL）使用`yum`包管理器。在Linux操作系统中，包管理器是查找、安装、更新和删除软件的最简单方式。

警告： 有时，Linux包管理器会安装过时的软件。例如，Ubuntu 14.04 LTS提供的是PHP 5.5.9，落后于最新版PHP 5.6.3。

幸好我们可以使用第三方仓库补充Linux包管理器的默认软件源，使用社区维护的较新版软件包。我们会在Ubuntu和CentOS中使用自定义的软件仓库安装最新版PHP。在正式安装之前，请确保你是系统的根用户，或者是拥有sudo权限的用户。使用Linux包管理器安装软件必须满足这个条件。

Ubuntu 14.04 LTS

Ubuntu的默认软件仓库中没有提供最新版PHP，因此我们要添加社区维护的个人软件包档案（Personal Package Archive，PPA）。PPA这个术语是Ubuntu专用的，不过这个概念可以延伸到其他发行版，其作用是使用第三方软件仓库扩展Ubuntu挑选的默认软件。翁德雷·叙里维护着一个优秀的PPA，提供了PHP最新稳定的每日构建版。这个PPA名为ppa:ondrej/php5-5.6。

1. 安装依赖的软件

添加翁德雷·叙里的PPA之前，我们必须确保操作系统中有add-apt-repository二进制文件。这个二进制文件包含在Ubuntu包python-software-properties中。在终端应用中输入下述命令，然后按回车键。如果有提示，要输入账户密码。

```
sudo apt-get install python-software-properties
```

这个命令会安装包含add-apt-repository二进制文件的python-software-properties包。现在我们可以添加自定义的PPA了。

2. 添加ppa:ondrej/php5-5.6 PPA

这个PPA在Ubuntu默认的软件仓库的基础上扩充Ubuntu可以使用的软件。在终端应用中输入下述命令，然后按回车键。如果有提示，要输入账户密码。

```
sudo add-apt-repository ppa:ondrej/php5-5.6
```

这个命令会把翁德雷·叙里的PPA添加到Ubuntu的软件源列表中，还会下载这个PPA的GPG公钥，并将其添加到本地GPG密钥环中。GPG公钥的作用是让Ubuntu验证PPA中的包，确保在原始作者构建并签名之后没有被篡改。

Ubuntu会缓存所有可用的软件，因此添加新软件源之后，要刷新这个缓存。在终端应用中输入下述命令，然后按回车键。如果有提示，要输入账户密码。

```
sudo apt-get update
```

3. 安装PHP

现在可以使用Ubuntu的aptitude包管理器从翁德雷·叙里的PPA中安装最新稳定版PHP了。在安装之前，我们要知道有哪些可用的PHP包，以及各自的作用。PHP以两种形式分发，一种是CLI包，目的是让我们在命令行中使用PHP（我们会使用这种形式）；另外还有几个PHP包，把PHP与Apache或nginx Web服务器集成在一起（我们在第7章讨论这两个服务器）。现在我们只关注如何安装PHP CLI包。

首先，我们来安装PHP CLI包。在终端应用中输入下述命令，然后按回车键。如果有提示，要输入账户密码。

```
sudo apt-get install php5-cli
```

Linux包管理器还包含各个PHP扩展的包，可以单独安装。现在我们来安装几个PHP扩展。在终端应用中输入下述命令，然后按回车键。如果有提示，要输入账户密码。

```
sudo apt-get install php5-curl php5-gd php5-json php5-mcrypt php5-mysqlnd
```

然后在终端执行下述命令，确认成功安装了PHP：

```
php -v
```

这个命令应该输出类似下面的内容：

```
PHP 5.5.11-3+deb.sury.org~trusty+1 (cli) (built: Apr 23 2014 12:15:16)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.5.0, Copyright (c) 1998-2014 Zend Technologies
    with Zend OPcache v7.0.4-dev, Copyright (c) 1999-2014, by Zend Technologies
```

CentOS 7

与Ubuntu一样，CentOS和RHEL也没在默认的软件仓库中提供最新稳定版PHP。RHEL特别在意官方发行版中有哪些软件包，因为RHEL以更好的安全性和稳定性而自豪，为了安全，软件更新添加得很慢。

我们不是财富500强公司，能承担得起在CentOS/RHEL Linux发行版中安装最新稳定版PHP可能导致的后果。为了安装最新版PHP，我们要使用EPEL（Extra Packages for Enterprise Linux的简称，意思是“企业版Linux的额外包”，*https://fedoraproject.org/wiki/EPEL*）。EPEL像下面这样描述自己：

> ……Fedora特别兴趣小组（Fedora Special Interest Group）致力于创建、维护并管理企业版Linux使用的一系列高质量的包，可以使用这些包的发行版包括但不限于：Red Hat Enterprise Linux（RHEL）、CentOS、Scientific Linux（SL）和Oracle Enterprise Linux（OEL）。

EPEL和官方的CentOS/RHEL Linux发行版没有关系，不过仍然可以用来扩充CentOS/RHEL默认的软件仓库。接下来我们就会这么做。

1. 添加EPEL仓库

下面我们告诉CentOS/RHEL系统，让它使用EPEL软件仓库。在终端应用中逐个输入下述命令，而且每输入一个命令之后都要按回车键。如果有提示，要输入账户密码。

```
sudo rpm -Uvh \
  http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm;
sudo rpm -Uvh \
  http://rpms.famillecollet.com/enterprise/remi-release-7.rpm;
```

这两个命令会把第三方软件仓库EPEL和remi添加到CentOS/RHEL系统中。现在，*/etc/yum.repos.d*目录中应该出现了epel.repo和*remi.repo*两个文件。

2. 安装PHP

现在，我们要从EPEL和remi仓库中安装最新版PHP。说明如何在Ubuntu中安装PHP时我说过，PHP以两种形式分发，其中一种形式是CLI包，目的是在命令行中使用PHP。现在我们只关注如何安装PHP CLI包。

首先，我们安装PHP CLI包。在终端应用中输入下述命令，然后按回车键。如果有提示，要输入账户密码。

```
sudo yum -y --enablerepo=epel,remi,remi-php56 install php-cli
```

然后，我们安装几个额外的PHP扩展。我们可以使用yum包管理器搜索PHP扩展的完整列表。在终端应用中输入下述命令，然后按回车键：

```
yum search php
```

找到PHP扩展列表之后，按照下面演示的方法安装。你安装的包或许有所不同。

```
sudo yum -y --enablerepo=epel,remi,remi-php56 \
     install php-gd php-mbstring php-mcrypt php-mysqlnd php-opcache php-pdo
```

在这个命令中要特别注意`--enablerepo`选项。这个选项的作用是，告诉yum从EPEL、remi和remi-php56仓库中安装指定的软件包。没有这个选项的话，yum只使用默认的软件源。

下面确认是否成功安装了PHP。在终端应用中输入下述命令，然后按回车键：

```
php -v
```

这个命令应该输出类似下面的内容：

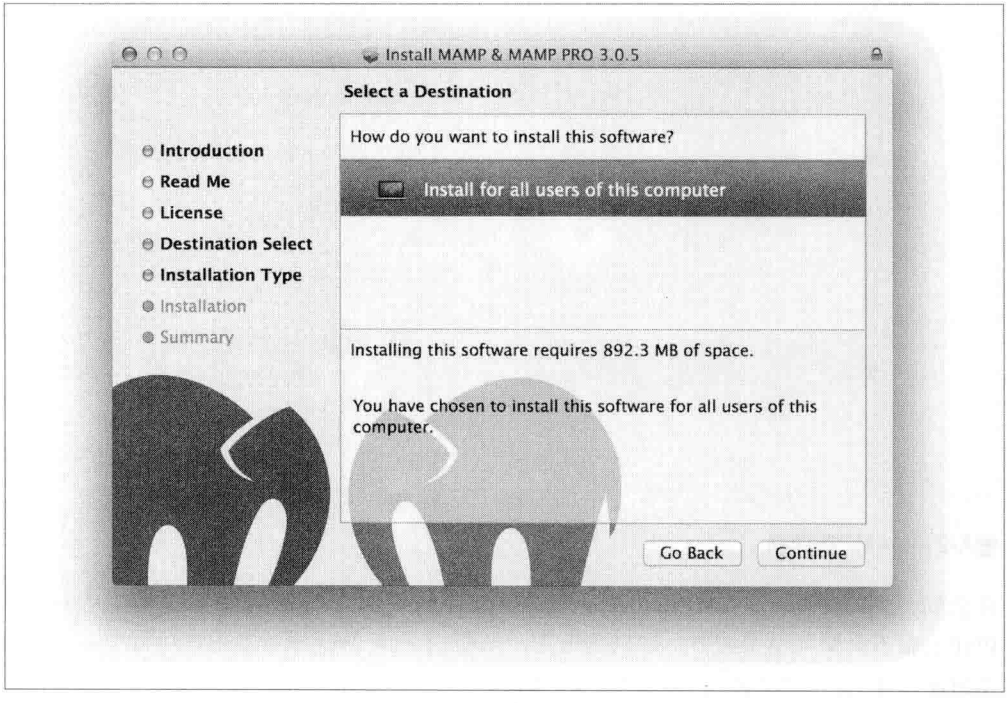
```
PHP 5.6.3 (cli) (built: Nov 16 2014 08:32:30)
Copyright (c) 1997-2014 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2014 Zend Technologies
    with Zend OPcache v7.0.4-dev, Copyright (c) 1999-2014, by Zend Technologies
```

OS X

OS X自带了PHP，不过可能不是最新版，或许也没有所需的PHP扩展。我建议别用OS X自带的PHP，而要使用自己安装的版本。在OS X中安装PHP有很多方式，我建议使用的两种方式是：MAMP和Homebrew。

MAMP

如果畏惧命令行终端，在OS X中安装PHP的最好方式是使用MAMP。MAMP（Mac-Apache-MySQL-PHP的简称）提供了Web开发所需的传统软件栈，包括Apache Web服务器、MySQL数据库服务器和PHP。MAMP是运行在OS X中的应用，有GUI。很多用户喜欢使用GUI界面，因为在界面中移动鼠标单击几下就能安装并配置MAMP软件（见图A-1）。MAMP在*/Applications*文件夹中，双击应用的图标后就启动了。MAMP提供了一个简单的OS X安装包（*.pkg*格式），安装和使用都特别简单。如果想快速访问，甚至还可以把MAMP拖曳到Dock中。

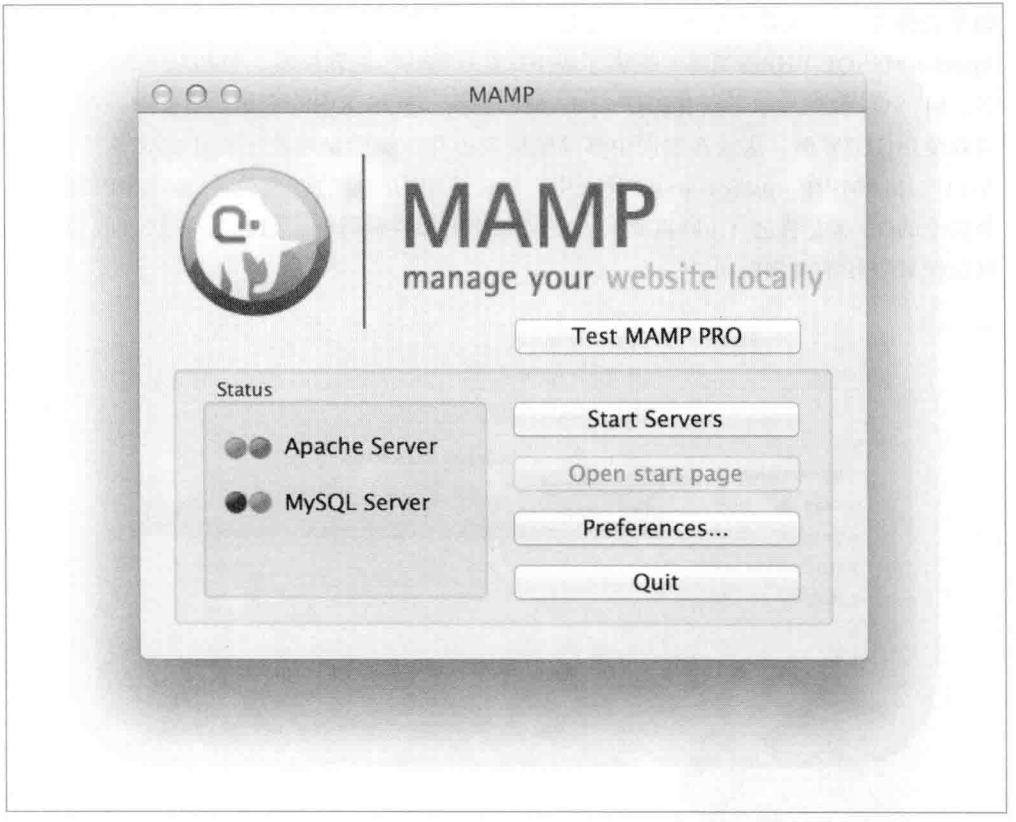


图A-1：安装MAMP

安装

从*http://www.mamp.info*下载MAMP的安装包（*.pkg*格式），然后双击，按照屏幕上的说明安装。

安装完成后，在*/Applications*文件夹中找到MAMP应用，然后双击应用的图标启动MAMP。打开MAMP后，单击“Start Servers”（启动服务器）按钮，启动Apache和MySQL服务器（见图A-2）。就这么简单。



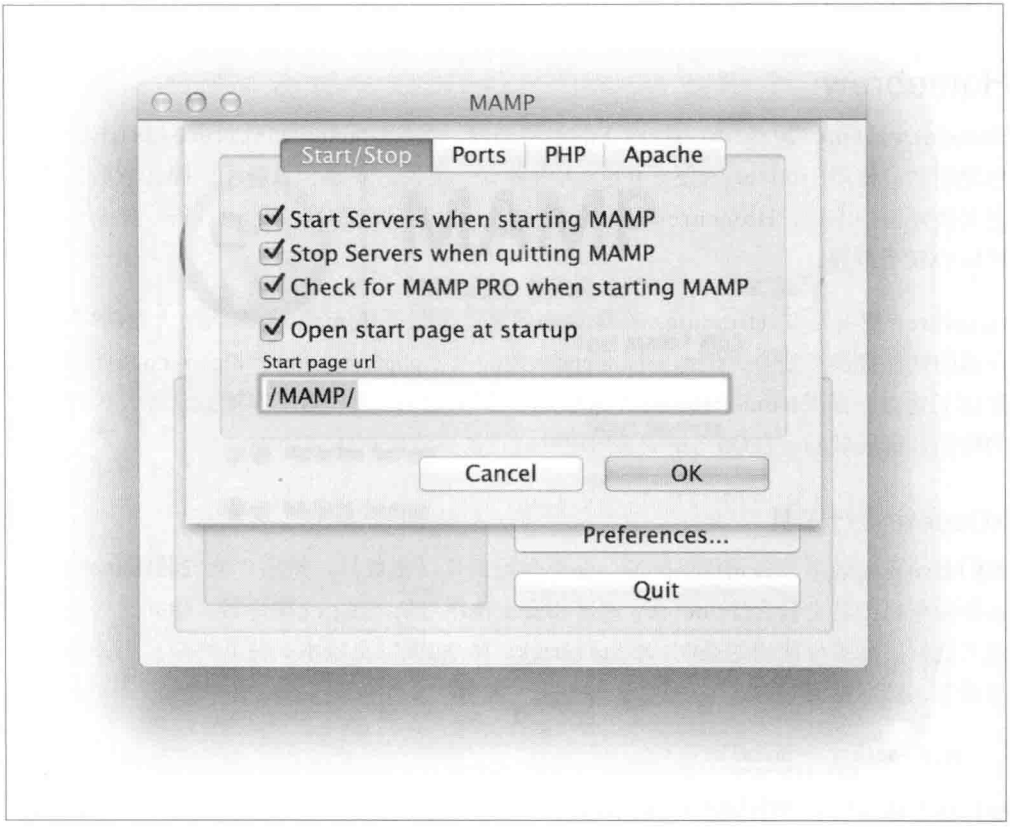
图A-2：MAMP的界面

你会问，那PHP呢？MAMP使用Apache的`mod_php`模块在Apache Web服务器中内嵌了PHP。我不会深入说明细节，你只需记住，只要Apache Web服务器正在运行中，就能使用PHP。我们在第7章讨论了PHP的部署策略。

启动Apache和MySQL服务器之后，打开Web 浏览器，访问*http://localhost:8888*。如果MAMP安装成功，应该会看到一个欢迎页面。

Apache Web服务器一般监听端口80上的连接，而MAMP在端口8888上运行Apache。类似地，MySQL 一般监听端口3306上的连接，而MAMP在端口8889上运行MySQL。我们可以在 MAMP 应用的偏好设置中修改 MAMP 默认使用的端口。在MAMP中，Apache Web服务器的文档根目录是*/Applications/MAMP/htdocs*。这个目录中的任何一个PHP文件都能在 Web 浏览器中通过*http://localhost:8888*访问。

如果经常使用MAMP，可以在MAMP应用的偏好设置（见图 A-3）中勾选“Start Servers when starting MAMP”（启动MAMP时启动服务器），然后把MAMP应用添加到OS X账户的开机自启动列表中。这么做，登录OS X后会自动启动MAMP中的Apache和MySQL服务器。



图A-3：MAMP应用的偏好设置

扩展

我们可以下载MAMP扩展，为本地安装的MAMP提供不同的PHP版本。MAMP更新频

繁，很有可能已经打包了最新版PHP。不过，如果基于某些原因打包的不是最新版，或者你需要使用旧版PHP，可以到MAMP的网站中下载所需的PHP版本。

局限

MAMP免费版只提供了一个Apache虚拟主机，而且也不易于修改PHP的配置或扩展。MAMP很基础，只为在OS X中做PHP开发提供了基本的必需品。

MAMP有收费的“专业”版，可以创建多个Apache虚拟主机，便于修改*php.ini*配置文件，还提供了调优好的PHP扩展。MAMP专业版很好，我们也应该有付费使用专业版的习惯。不过别误解了我的意思，与其花钱，还不如学一些命令行基础知识，使用优秀的包管理器Homebrew（*http://brew.sh*）。

Homebrew

Homebrew（*http://brew.sh*）是OS X的包管理器，类似Ubuntu的`aptitude`和RHEL的`yum`包管理器。使用Homebrew便于在OS X中浏览、查找、安装、更新和删除任意数量的自定义软件包。不过，Homebrew是命令行应用。如果不熟悉OS X的命令行，你会觉得使用MAMP更舒服。

Homebrew使用处方（formula）在电脑中安装软件包。Homebrew默认为大量OS X没有自带的软件提供了处方，例如，Homebrew为`wget`、`phploc`、`phpmd`和`php-code-sniffer`提供了处方。如果Homebrew默认的处方不够用，可以接入第三方处方仓库，扩充可用的软件。毫无疑问，在OS X中安装PHP时，我最喜欢使用Homebrew。

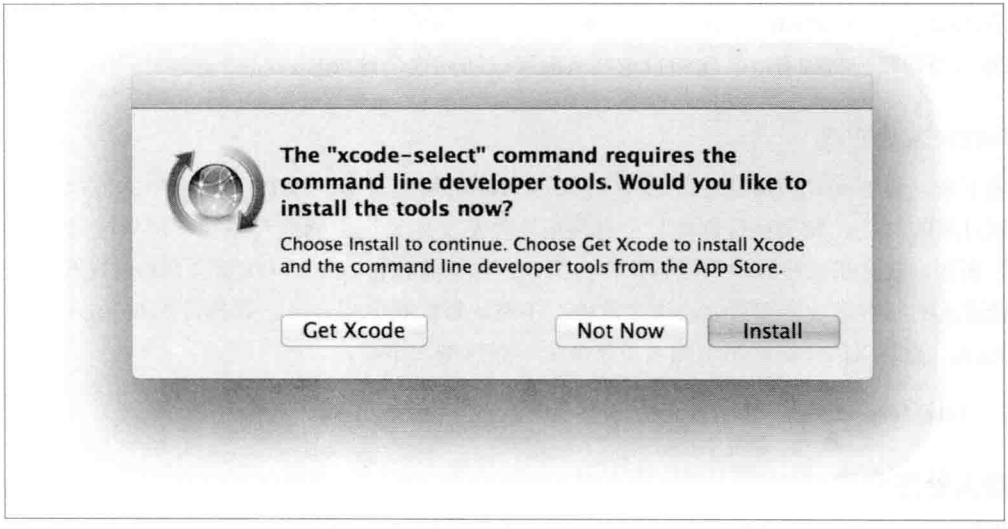
XCode命令行工具

我们必须先安装苹果公司提供的XCode命令行工具（免费），然后才能安装Homebrew。这些命令行工具中包含Homebrew构建和安装软件包所需的`gcc`编译器（除此之外还有其他工具）。如果你使用的是OS X Mavericks 10.9.2或以上版本，打开终端应用，输入下述命令，然后按回车键：

```
xcode-select --install
```

执行这个命令后会弹出如图A-4所示的窗口。

单击“Install”（安装）按钮，开始安装XCode命令行工具。出现软件许可协议时，单击“Agree”（同意）按钮。XCode命令行工具安装好之后，单击“Done”（完成）按钮，然后继续下一步。



图A-4：安装XCode命令行工具

如果你使用的是旧版OS X，必须登录苹果开发者门户网站（https://*developer.apple.com/*），下载并运行单独的XCode命令行工具安装包（*.pkg*格式）。

安装

安装好XCode命令行工具之后，在OS X的终端应用中输入下述命令，然后按回车键：

```
ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
```

警告： 这个命令的作用是，执行从远程URL下载来的Ruby代码。不管远程的源码多么合情合理，执行之前一定要先检查代码。

目录的访问权限

Homebrew会把软件下载并安装到*/usr/local/Cellar*目录中，然后为安装的软件二进制文件创建符号链接，存放在*/usr/local*目录中。因此，你的OS X用户账户必须能访问*/usr/local*目录，才能使用Homebrew包管理器安装的软件。

下面我们确保你的OS X用户账户拥有*/usr/local*目录。在OS X的终端应用中输入下述命令，然后按回车键。如果有提示，要输入管理员的密码。

```
sudo chown -R `whoami` /usr/local
```

`chown`命令的作用是修改指定目录的属主；`-R`标志的意思是，递归修改指定目录中所有

子目录的属主；whoami参数会动态替换成你的OS X用户账户名。执行这个命令后，你的OS X用户账户将拥有*/usr/local*目录（从而可以访问这个目录）。

PATH环境变量

接下来，要把*/usr/local*目录添加到OS X的PATH环境变量中。PATH环境变量的值是一系列目录的路径，如果执行软件时没有使用软件在文件系统中的绝对路径，而只使用软件的名称，就会在这些目录中搜索软件。例如，如果我执行wget，OS X会在PATH环境变量设定的所有目录中搜索wget这个软件。不然，每次想使用wget，我都得输入*/usr/local/wget*。在OS X的终端应用中输入下述命令，然后按回车键：

```
echo 'export PATH="/usr/local/bin:$PATH"' >> ~/.bash_profile
```

接入处方仓库

使用Homebrew安装PHP之前，我们必须接入额外的仓库，提供Homebrew默认仓库中没有的PHP相关的处方。

首先，我们要接入homebrew/dupes仓库。这个仓库中包含OS X中已有软件的处方，不过版本比OS X自带的新。在OS X的终端应用中输入下述命令，然后按回车键：

```
brew tap homebrew/dupes
```

然后，我们要接入homebrew/versions仓库。这个仓库包含OS X中现有软件的多个版本。在OS X的终端应用中输入下述命令，然后按回车键：

```
brew tap homebrew/versions
```

最后，我们要接入homebrew/php仓库。这个仓库包含Homebrew默认仓库中可能没有的PHP相关的处方。Homebrew默认的仓库不是由PHP开发者维护的，而这个仓库是。这个仓库中包含适合PHP开发者使用的软件。在OS X的终端应用中输入下述命令，然后按回车键：

```
brew tap homebrew/php
```

安装PHP

目前，我们安装了Homebrew包管理器，配置了文件系统的访问权限，更新了PATH环境变量，还接入了额外的处方仓库。现在该安装PHP了。每个PHP版本和每个PHP版本的扩展都有各自的Homebrew处方。使用Homebrew搜索可用的处方非常简单，在OS X的终端应用中输入下述命令，然后按回车键：

```
brew search php
```

你会看到一个很长的列表，这个列表列出了可用的PHP处方。在这个处方列表中找到最新稳定版PHP（PHP 5.5.x的处方名为php55，PHP 5.6.x的处方名为php56，以此类推）。我选择使用php56，因为PHP 5.6.x是最新稳定版。在OS X的终端应用中输入下述命令，然后按回车键：

```
brew install php56
```

安装可能要一段时间，所以你可以去泡杯咖啡，几分钟之后再回来看看。PHP软件包安装好之后，可以在OS X的终端应用中执行php -v命令，确认安装是否成功。这个命令应该输出Homebrew安装的PHP解释器的全名和版本号。

安装PHP扩展

在Homebrew中，PHP扩展是和PHP解释器分开安装的。我们可以按照前面搜索PHP的方式搜索PHP扩展。假设你选择使用php56，在OS X的终端应用中输入下述命令，然后按回车键：

```
brew search php56
```

你应该会看到一个很长的列表，这个列表列出了前缀为php56-的PHP 5.6扩展。找到所需的扩展后，在OS X的终端应用中输入下述命令，然后按回车键。记得把下述命令中的处方换成你想安装的扩展。

```
brew install php56-intl php56-mcrypt php56-xhprof
```

Homebrew包管理器的功能很强大，我在这里只演示了一部分。在OS X的终端应用中输入brew，然后按回车键，你会看到完整的Homebrew命令列表。你还可以阅读Homebrew完整的在线文档，地址是*http://brew.sh*。

从源码构建

操作系统的包管理器编译好的PHP二进制文件不一定总是最新版，或者不是你想使用的版本。遇到这种情况时，最好自己从源码构建PHP。是的，这听起来很吓人。第一次编译PHP之前，我用了很长时间才鼓足勇气。不过我可以保证，自己构建并没有听起来那么吓人。

构建的过程很简单：先下载并提取PHP源码；然后使用configure命令配置源码，确保安装了依赖的所有软件；最后使用make命令构建，得到所需的PHP二进制文件。下载、配置和构建，就这简单的三步。

从源码编译PHP能根据自己的具体要求灵活调整PHP。配置PHP的方式很多，为了节省时间，我会演示我为自己的项目构建PHP时喜欢使用的方式。除了PHP默认的特性之外，我一般还希望PHP支持：

- OpenSSL。
- 字节码缓存。
- FPM（FastCGI进程管理）。
- PDO数据库抽象API。
- 加密。
- 多字节字符串。
- 图像处理。
- 网络套接字。
- curl。

看过这个列表之后，下面我们开始构建PHP。你要在自己的电脑中试着跟我一起做。如果这是你第一次从源码构建PHP，我强烈建议你在虚拟机中构建。你可以使用VMware、Parallels或VirtualBox在本地架设虚拟机，还可以买一个特别便宜的远程虚拟机，例如DigitalOcean、Linode或其他按小时收费的Web主机。如果出错了，可以销毁虚拟机，重新架设，然后再试着构建。

现在，深呼吸，打开终端应用，别害怕会犯错（这是最重要的）。

获取源码

首先，我们要下载PHP源码。访问*http://www.php.net/downloads.php*，找到最新稳定版PHP的源码下载地址。我找到的最新稳定版是5.6.3，你找到的可能和我不一样。在终端应用中输入下述命令，每输入一个命令之后都要按回车键。

src/目录

首先，在家目录中创建*src/*目录。这个文件夹用于存放从PHP.net下载的源码。我们要使用cd命令进入src/目录，让它变成当前工作目录：

```
mkdir ~/src;
cd ~/src;
```

下载源码

然后，使用wget下载*tar.gz*存档格式的PHP源码。下载得到的文件路径是*~/src/php.tar.gz*。

```
wget -O php.tar.gz http://www.php.net/get/php-5.6.3.tar.gz/from/this/mirror
```

使用tar命令从存档文件中提取PHP源码，然后使用cd命令进入提取出来的源码目录：

```
tar -xzvf php.tar.gz;
cd php-*;
```

配置PHP

我们下载了PHP的源码，现在需要配置PHP。在配置之前，我们必须先安装几个软件依赖。我是怎么知道需要安装什么依赖的呢？我会执行./configure命令（参见下一节），直到能成功执行为止。如果./configure命令执行失败时提示缺少软件依赖，就说明缺少软件。安装缺少的依赖，然后再执行./configure命令。像这样一直重复，直到./configure命令能成功执行为止。

我很幸运，已经知道./configure命令需要哪些软件依赖了。下面我们来安装这些软件依赖。我列出了Ubuntu/Debian和CentOS/RHEL两种Linux发行版所需的命令，请根据你使用的Linux发行版选择适当的命令。

注意： 如果基于某些原因，./configure命令还是报告缺少依赖，可以在线搜索缺少依赖的软件包：Ubuntu的软件包在*http://packages.ubuntu.com/*中搜索，CentOS的软件包在*https://fedoraproject.org/wiki/EPEL*中搜索。

构建必需的工具

我们需要这些基本的软件二进制文件才能在操作系统中构建PHP。所需的二进制文件包括：gcc、automake和其他基本的开发软件。

```
# Ubuntu
sudo apt-get install build-essential;

# CentOS
sudo yum groupinstall "Development Tools";
```

libxml2

我们需要libxml2库。PHP中XML相关的函数会用到这个库。

```
# Ubuntu
sudo apt-get install libxml2-dev;

# CentOS
sudo yum install libxml2-devel;
```

OpenSSL

我们需要openssl库。在PHP中使用HTTPS流封装协议时需要这个库，所以这个库算是很重要的，对吧？

```
# Ubuntu
sudo apt-get install libssl-dev;

# CentOS
sudo yum install openssl-devel;
```

curl

我们需要libcurl库。PHP中的curl函数需要这个库。

```
# Ubuntu
sudo apt-get install libcurl4-dev;

# CentOS
sudo yum install libcurl-devel;
```

图像处理

我们需要GD、JPEG、PNG和其他与图像有关的系统库。幸运的是，这些库已经打包到一个包中了。使用PHP处理图像时需要这些库。

```
# Ubuntu
sudo apt-get install libgd-dev;

# CentOS
sudo yum install gd-devel;
```

Mcrypt

我们需要mcrypt系统库，这样才能在PHP中使用Mcrypt加密和解密函数。不知何故，CentOS的默认包仓库中没有这个库，所以我们要使用第三方EPEL包仓库补充CentOS的默认包仓库，这样才能安装Mcrypt。

```
# Ubuntu
sudo apt-get install libmcrypt-dev;

# CentOS
wget http://dl.fedoraproject.org/pub/epel/6/x86_64/epel-release-6-8.noarch.rpm;
sudo rpm -Uvh epel-release-6*.rpm;
sudo yum install libmcrypt-devel;
```

./configure命令

我们已经安装了软件依赖，下面要配置PHP。在终端应用中输入下述./configure命令，然后按回车键：

```
./configure
    --prefix=/usr/local/php5.6.3
    --enable-opcache
    --enable-fpm
```

```
--with-gd
--with-zlib
--with-jpeg-dir=/usr
--with-png-dir=/usr
--with-pdo-mysql=mysqlnd
--enable-mbstring
--enable-sockets
--with-curl
--with-mcrypt
--with-openssl;
```

这个命令很长，有大量参数。别被吓住了，每个选项都有特定的作用。你可以执行./configure --help命令，查看全部可用的选项。下面逐一介绍这个./configure命令中的选项，告诉你每个选项的作用：

--prefix=/usr/local/php5.6.3

--prefix选项的值是文件系统中一个目录的路径，这个目录用于存放编译得到的PHP二进制文件、要引入的文件、库和配置文件。为了组织合理，我喜欢把自己构建的PHP和相关的文件放在同一个父目录中。你使用的用户账户要有这个目录的写权限。如果你没有*/usr/local*目录的写权限，可以把--prefix选项的值设为用户账户家目录中的一个目录（例如，*~/local/php-5.5.13*）。不管怎样，执行./configure命令前，--prefix选项设定的目录要存在。

--enable-opcache

--enable-opcache选项的作用是启用PHP内置的字节码缓存系统。这个缓存系统基本上都要启用。启用这个缓存系统后性能会有极大的提升。

--enable-fpm

--enable-fpm选项的作用是启用PHP内置的FastCGI进程管理器。使用这个选项构建得到的PHP能作为FastCGI进程运行，可以通过TCP端口或本地Linux套接字访问。FPM已经逐渐成为运行PHP的首选方式（尤其是在nginx Web服务器中）。如果拿不准，我建议启用这个选项。

--with-gd

--with-gd选项的作用是让PHP能与操作系统中的GD图像处理库交互。如果计划使用PHP处理图像，要启用这个选项。

--with-zlib

--with-zlib选项的作用是让PHP能与操作系统中的Zlib库交互。Zlib是数据压缩库，GD图像库创建和处理PNG图像数据时需要使用这个库。如果使用了--with-gd选项，必须使用这个选项。

`--with-jpeg-dir`

`--with-jpeg-dir`选项指定JPEG库在文件系统中的目录路径。如果使用了`--with-gd`选项，必须使用这个选项。

`--with-png-dir`

`--with-png-dir`选项指定PNG库在文件系统中的目录路径。如果使用了`--with-gd`选项，必须使用这个选项。

`--with-pdo-mysql=mysqlnd`

`--with-pdo-mysql`选项的作用是让PHP使用原生的MySQL驱动为MySQL数据库启用PDO数据库抽象API。如果使用MySQL，就得启用这个选项。

`--enable-mbstring`

`--enable-mbstring`选项的作用是让PHP支持多字节（Unicode）字符串。这个选项基本上都要启用。

`--enable-sockets`

`--enable-sockets`选项的作用是让PHP支持网络套接字，以便通过TCP套接字与远程设备通信。这个选项基本上都要启用。

`--with-curl`

`--with-curl`选项的作用是让PHP与操作系统中的`curl`库交互，以便使用PHP中的`curl`函数收发HTTP请求。这个选项基本上都要启用。

`--with-mcrypt`

`--with-mcrypt`选项的作用是让PHP与操作系统中的`mcrypt`库交互，用于加密和解密数据。虽然这个选项不是必须的，不过有越来越多的PHP组件使用`mcrypt`库，所以我强烈建议启用这个选项。

`--with-openssl`

`--with-openssl`选项的作用是让PHP与操作系统中的`openssl`库交互。若想使用PHP的HTTPS流封装协议，必须启用这个选项。虽然严格来说这个选项是可选的，其实不然。

构建并安装PHP

配置PHP和安装软件依赖是最难的部分，从现在开始就简单了。假设`./configure`命令执行成功了，接下来要在终端应用中输入下述命令，然后按回车键：

```
make && make install
```

这个命令会编译 PHP，可能要花一段时间，趁机我们可以喝杯咖啡。这个命令执行完毕后，PHP就安装好了。其实不怎么难，对吧？

编译得到的PHP二进制文件在--prefix选项指定的目录中的bin/目录里。php-fpm二进制文件在--prefix选项指定的目录中的*sbin/*目录里。记得把*bin/*和*sbin/*两个目录添加到系统的PATH环境变量中，这样直接使用名称就能引用php二进制文件，而不用使用绝对路径。

创建php.ini文件

别忘了*php.ini*文件。构建过程中可能不会自动创建这个文件。GitHub中的PHP仓库里有为本地开发环境配置好的*php.ini*文件。*php.ini*文件应该放在--prefix选项指定的目录中的*lib/*目录里。现在我们来创建这个文件。在终端应用中输入下述命令，每输入一个命令之后都要按回车键。

首先，使用cd命令进入PHP所在目录中的*lib/*目录。如果执行./configure命令时为--prefix选项指定了不同的路径，你的路径可能和下面的不一样：

```
cd /usr/local/php5.6.3/lib
```

然后，从PHP在GitHub中的仓库下载*php.ini-development*文件，把下载得到的文件命名为*php.ini*：

```
curl -o php.ini \
 https://raw.githubusercontent.com/php/php-src/master/php.ini-development
```

到此结束。现在可以使用新安装的PHP解释器执行PHP文件了。第7章讨论PHP的部署策略时会进一步介绍php-fpm二进制文件。

Windows

是的，在Windows中可以运行PHP。不过，我建议你使用Linux虚拟机。生产服务器很有可能使用某个Linux发行版，因此我们要让本地开发环境与生产环境高度一致。然而，如果非得在本地使用Windows的话，PHP的安装方法如下。

二进制文件

PHP.net的开发者心肠很好，为Windows提供了构建好的PHP二进制文件，地址是*http://php.net/windows*。下载合适的PHP版本（是个ZIP存档文件），然后解压到一个文件夹中。我把ZIP存档文件解压到*C:*文件夹中。然后把*php.ini-production*文件重命名为*php.ini*。在Windows的命令行中使用PHP无需再做其他修改了。我们可以像下面这样使用可选的参数执行PHP脚本：

```
C:\PHP\php.exe -f "C:\path\to\script.php" -- -arg1 -arg2 -arg3
```

建议： 你应该把 PHP 的可执行文件添加到Windows的PATH变量中（*http://bit.ly/addtopath*），还应该把.php扩展名添加到Windows的PATHEXT变量中，这样能少输入很多字符。

WAMP

我们还可以下载并安装WAMP（*http://www.wampserver.com/en/*），快速搭建本地PHP开发环境。与OS X中的MAMP一样，WAMP也是一体化的软件包，提供了开箱即用的传统Web开发软件栈。WAMP包含Apache Web服务器，MySQL数据库服务器和PHP。WAMP提供了Windows软件安装程序，安装过程中的每一步都会指导你怎么做。WAMP还在Windows任务栏的通知区域提供了配置菜单，可以快速且轻易地启动、停止或重启Apache和MySQL服务器。与MAMP一样，WAMP使用Apache的mod_php模块把PHP嵌入Apache Web服务器。因此，只要Apache服务器运行着，就能使用PHP。

WAMP是在Windows中快速搭建本地PHP开发环境的最好方式。可是，就像MAMP一样，我们受限于WAMP提供的软件和扩展。我们可以从WAMP的网站中下载其他PHP版本。详情参见*http://www.wampserver.com/*。

Zend Server

另一个一体化解决方案是Zend Server。Zend Server既有免费版，也有收费版。与WAMP一样，Zend Server也提供了Apache Web服务器、最新版PHP解释器、流行的PHP扩展和MySQL数据库服务器。除此之外还提供了一个易于安装的包，用于安装Zend自己的调试工具。Zend Server的安装方法很简单：下载安装程序（*.exe*文件），运行安装程序，然后按照屏幕上显示的说明即可。详情参见*http://www.zend.com/en/products/server*。

附录B

本地开发环境

我们讨论了很多关于配置生产服务器和开发应用的知识，但是还没讨论如何在本地计算机中开发应用。我们可以使用什么工具？如何让开发环境和生产环境保持一致？这一篇附录会解答这些问题。

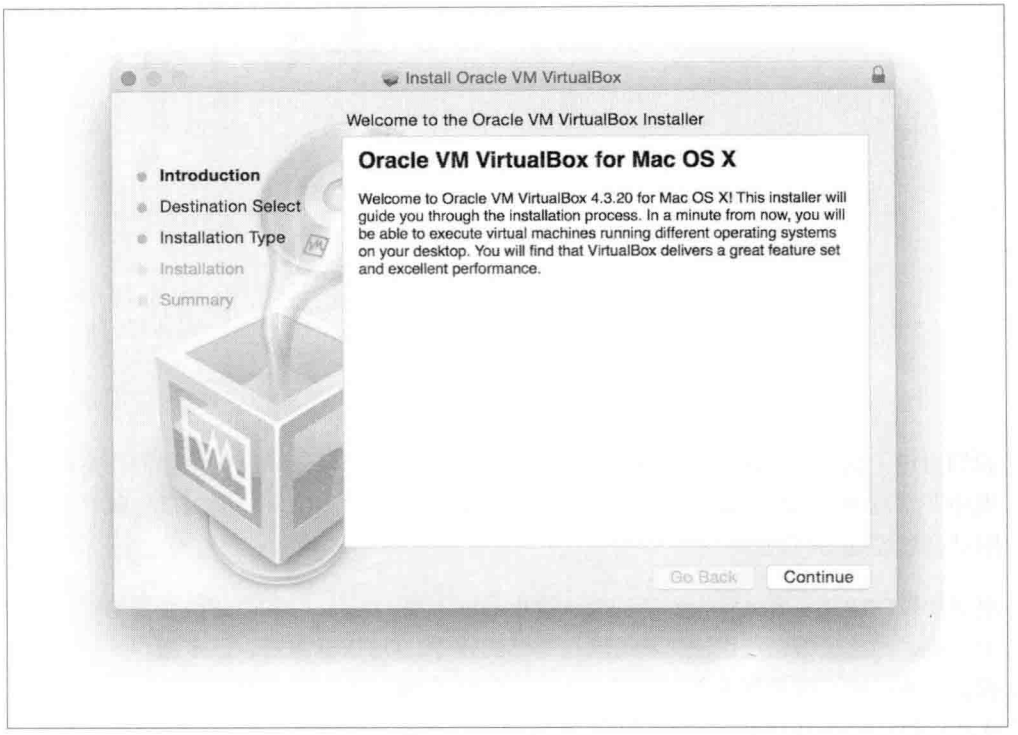
很多初级PHP开发者依赖操作系统默认提供的软件栈，Apache和PHP往往都是较旧的版本。我强烈建议你别使用操作系统默认提供的软件。OS X升级后，很多用户（包括我）都会极为震惊，因为高度定制的Apache配置文件不翼而飞了。使用系统内置的软件时要小心，这些软件往往是过时的，而且系统升级后可能会被覆盖。我们应该使用虚拟机搭建本地开发环境，安全地与本地操作系统隔离开。虚拟机是使用软件模拟的操作系统。例如，可以在OS X中创建运行Ubuntu或CentOS的虚拟机。虚拟机的表现与单独的计算机完全一样。

建议： 虚拟机使用的操作系统（我喜欢用Ubuntu Server）要和生产服务器一样。这么做很重要，因为能避免由操作系统所用软件不同而导致意料之外的开发和运行时错误。

VirtualBox

创建和管理虚拟机的软件程序有很多，有些是商用的，例如VMWare Fusion（*http://www.vmware.com/products/fusion*）和Parallels（*http://www.parallels.com/products/desktop/*），有些则是开源的，例如VirtualBox（*https://www.virtualbox.org*）。说实话，VirtualBox是相当不错的软件，像宣传口号所说的一样强大，而且免费。VirtualBox虽然不像同类商用软件那样强大，但足够使用了。我们可以从*https://www.virtualbox.org*中下

载运行在OS X或Windows中的VirtualBox。VirtualBox提供了适用于各种操作系统的GUI安装程序（见图B-1）。



图B-1：VirtualBox的安装程序

Vagrant

VirtualBox虽然能创建虚拟机，但没有提供用户友好的界面，无法轻易地启动、配置、停止和销毁虚拟机。不过，我们可以使用Vagrant（*https://www.vagrantup.com*），这是一个虚拟化工具，使用用户友好的命令行接口补足（并抽象）了VirtualBox，我们只需执行一个命令就能创建、启动、停止和销毁VirtualBox虚拟机。我们可以从*https://www.vagrantup.com*中下载运行在OS X或Windows中的Vagrant。Vagrant还提供了适用于各种操作系统的GUI安装程序。

命令

安装好之后，我们可以在终端应用中执行`vagrant`命令创建、配置、启动、停止和销毁VirtualBox虚拟机。下面是最常使用的Vagrant命令：

`vagrant init`

这个命令在当前工作目录中新建Vagrantfile配置脚本。我们使用这个脚本配置虚拟机的属性，保存所有配置细节。

`vagrant up`

这个命令用于创建或启动虚拟机。

`vagrant provision`

这个命令用于使用指定的配置脚本配置虚拟机。后文会讨论如何配置。

`vagrant ssh`

这个命令用于通过SSH登录虚拟机。

`vagrant halt`

这个命令用于停止虚拟机。

`vagrant destroy`

这个命令用于销毁虚拟机。

建议： 我建议为这些Vagrant命令创建别名，因为要经常输入。把下面的代码添加到`~/.bash_profile`文件中，然后重启终端应用：

```
alias vi="vagrant init"
alias vu="sudo echo 'Starting VM' && vagrant up"
alias vup="sudo echo 'Starting VM' && vagrant up --provision"
alias vp="vagrant provision"
alias vh="vagrant halt"
alias vs="vagrant ssh"
```

容器

我们已经安装了VirtualBox和Vagrant。下面该做什么呢？我们要选择一个Vagrant容器（box），在此基础上架设虚拟机。Vagrant容器是预先配置好的虚拟机，我们可以在此基础上进一步配置，用来开发PHP应用。有些容器是空盒，像一张空白画布一样；有些容器则为特定类型的应用提供了完整的软件栈。我们可以访问*https://vagrantcloud.com*，浏览可用的容器。

我一般会选择ubuntu/trusty64（*https://vagrantcloud.com/ubuntu/boxes/trusty64*）这个空盒，然后使用Puppet配置，安装具体应用所需的软件。如果发现有Vagrant容器提供了你所需的工具，直接拿来用即可，这样能节省时间。

初始化

找到所需的Vagrant容器后，在终端应用中进入相应的工作目录，然后执行下述命令初始化，新建Vagrantfile文件：

```
vagrant init
```

在你喜欢的文本编辑器中打开新建的Vagrantfile文件。这个文件中的内容使用Ruby编写，不过易于理解。找到config.vm.box设置，将其值改为你使用的Vagrant容器名。例如，如果我选择使用Ubuntu容器，会把这个设置的值改为ubuntu/trusty64。更新后，Vagrantfile文件中应该有下面这行：

```
config.vm.box = "ubuntu/trusty64"
```

然后，去掉下面这行的注释，以便通过本地网络中192.168.33.10这个IP地址在Web浏览器中访问虚拟机：

```
config.vm.network "private_network", ip: "192.168.33.10"
```

最后，执行下述命令，创建这个虚拟机：

```
vagrant up
```

这个命令会下载远程Vagrant容器（如果需要下载的话），然后基于这个容器新建VirtualBox虚拟机。

配置

如果你使用的Vagrant容器没有提供预先配置好的软件栈，那么虚拟机现在没什么用。我们要配置虚拟机，安装运行PHP应用所需的软件。我们至少需要安装Web服务器和PHP，可能还要安装数据库。配置虚拟机涉及很多知识，本书无法一一详述。不过，我可以为你指明方向。你可以使用Puppet或Chef配置Vagrant虚拟机。Puppet和Chef都可以在Vagrantfile配置文件中启用和配置。

建议：埃里卡·海蒂（*http://erikaheidi.com*）在NomadPHP中做了很棒的演讲（*http://www.bit.ly/1zUJmqb*），介绍了Vagrant以及Puppet和Chef等配置工具。她还写了本《Vagrant Cookbook》（*https://leanpub.com/vagrantcookbook*），在LeanPub上销售。

Puppet

在Vagrantfile文件的底部有一部分如下所示的代码。默认情况下，这部分代码可能被注释掉了。

```
config.vm.provision "puppet" do |puppet|
  puppet.manifests_path = "manifests"
  puppet.manifest_file  = "default.pp"
end
```

如果把这部分代码的注释去掉，Vagrant会使用Puppet清单配置虚拟机。如果想进一步学习Puppet，可以访问*http://puppetlabs.com*。

Chef

如果想使用Chef这个配置工具，可以把Vagrantfile文件中下面这部分代码的注释去掉：

```
config.vm.provision "chef_solo" do |chef|
  chef.cookbooks_path = "../my-recipes/cookbooks"
  chef.roles_path = "../my-recipes/roles"
  chef.data_bags_path = "../my-recipes/data_bags"
  chef.add_recipe "mysql"
  chef.add_role "web"

  # 还可以设定自定义的 JSON属性
  chef.json = { mysql_password: "foo" }
end
```

Vagrant会根据你提供的食谱、角色和配方配置虚拟机。如果想进一步学习Chef，请访问*https://www.chef.io/chef/*。

同步文件夹

不管使用什么配置工具，通常都会把本地设备中的项目目录与虚拟机中的一个目录对应起来。例如，可以把本地的项目目录与虚拟机中的`/var/www`目录对应起来。如果虚拟机中Web服务器的虚拟主机文档根目录是`/var/www/public`，那么虚拟机中的Web服务器就会伺服本地项目的`public/`目录。本地的任何变动都会立即同步到虚拟机。我们可以去掉Vagrantfile文件中下面这行的注释，在本地和虚拟机之间同步目录。

```
config.vm.synced_folder ".", "/vagrant_data"
```

第一个参数（`.`）是相对`Vagrantfile`配置文件的本地路径；第二个参数（`/vagrant_data`）是虚拟机中的绝对路径，这个目录会和本地目录对应起来。虚拟机中的路径很大程度上取决于虚拟机中Web服务器的虚拟主机配置。OS X用户应该使用NFS方式同步文件夹，方法是把`config.vm.synced_folder`那一行设置改为：

```
config.vm.synced_folder ".", "/vagrant_data", type: "nfs"
```

然后再去掉下面这几行的注释，把VirtualBox虚拟机使用的内存增加到1024MB：

```
config.vm.provider "virtualbox" do |vb|
```

```
    # 不以无界面模式引导
    # vb.gui = true

    # 使用VBoxManage定制虚拟机。例如，修改内存大小：
    vb.customize ["modifyvm", :id, "--memory", "1024"]
                                                          end
```

快速上手

Puppet和Chef都不易于学习，对Vagrant新手来说更难。下面几个工具能帮助你快速上手，无需自己编写Puppet和Chef清单。

Laravel Homestead

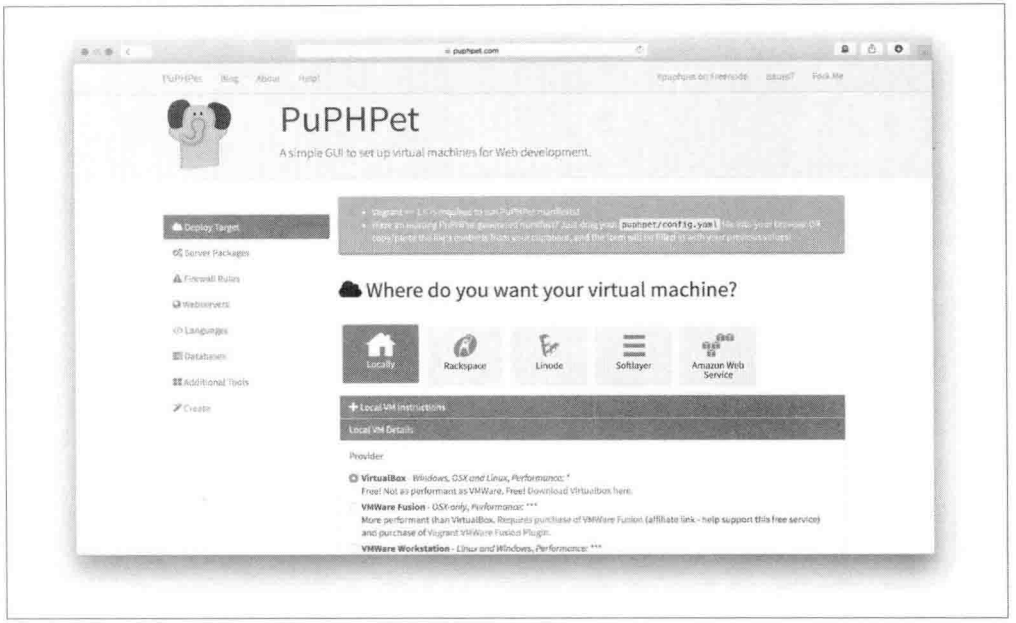
Homestead（*http://laravel.com/docs/4.2/homestead*）是建立在Vagrant之上的抽象，也是个Vagrant容器，预先配置好了完整的软件栈，包括以下软件：

- Ubuntu 14.04
- PHP 5.6
- HHVM
- Nginx
- MySQL
- Postgres
- Node（包括 Bower、Grunt 和 Gulp）
- Redis
- Memcached
- Beanstalkd
- Laravel Envoy

Homestead适用于任何PHP应用。我在本地设备中也使用Homestead开发Slim和Symfony应用。若想进一步学习Homestead，请访问*http://laravel.com/docs/4.2/homestead*。

PuPHPet

如果不知道如何编写Puppet清单，PuPHPet（*https://puphpet.com*）是最好的工具。在这个网站中移动鼠标单击几下就能自动创建Puppet配置（见图B-2），然后下载得到的Puppet配置，再执行`vagrant up`命令即可。就这么简单。



图B-2：PuPHPet

Vaprobash

Vaprobash（*http://fideloper.github.io/Vaprobash/*）类似PuPHPet。虽然没有提供使用鼠标单击的网站，但也特别易于使用。我们只需下载Vaprobash的`Vagrantfile`文件，去掉所需工具的注释即可。想使用nginx吗？去掉nginx那行的注释即可；想使用MySQL吗？去掉MySQL那行的注释即可；想使用Elasticsearch吗？去掉Elasticsearch那行的注释即可。准备好之后，在终端应用中执行vagrant up命令，Vagrant就会配置虚拟机了。

作者介绍

Josh Lockhart开发了Slim框架（*http://slimframework.com/*），这是一个PHP微型框架，适合快速开发Web应用和API。Josh还是“PHP之道”（*http://www.phptherightway.com/*）的发起人和当前的维护者。“PHP之道”在PHP社区中很受欢迎，目的在于鼓励全世界的PHP开发者使用一些良好实践，传播一些高质量的资源。

Josh是New Media Campaigns（*http://www.newmediacampaigns.com/*）的开发者，这个机构位于北卡罗来纳州卡勃罗市，提供全方位的服务，包括Web设计、开发和营销。他热衷于使用HTML、CSS、PHP、JavaScript、Bash和各种内容管理框架开发应用。

他2008年毕业于坐落在教堂山的北卡罗来纳大学，专业是资讯与图书馆学（http://*sils.unc.edu/*）。他现在和妻子Laurel及两条狗一起居住在北卡罗来纳州教堂山。

你可以在Twitter上关注Josh（*https://twitter.com/codeguy*），阅读他的博客（*https://joshlockhart.com*），还可以在GitHub中跟踪他的开源项目（*https://github.com/codeguy*）。

封面介绍

本书封面上的动物是蓑颈白鹮（学名是Threskiornis spinicollis），生活在澳大利亚和新几内亚全境，以及印度尼西亚部分地区。

蓑颈白鹮是大型鸟类，能长到30英寸高。这种鸟以颈部有特色的硬毛而得名，这些硬毛在成年时出现。这种鸟的嘴很长，而且是弯曲的，这有利于在水中筛滤昆虫、软体动物和青蛙。农民喜欢让蓑颈白鹮进入田地，因为这种鸟以危害农作物的昆虫、草蜢、蟋蟀和蝗虫为食。

这种鸟居无定所，喜欢成群在不同的栖息地之间迁徙。这种鸟喜欢生活在浅滩淡水湿地、人工草地、沼泽、泻湖和草原中。在繁殖期，这种鸟会在水中的树上使用木棍和芦苇筑造杯状的鸟巢，而且知道群居，通常会和澳大利亚白鹮生活在一起。因此，人们经常发现这种鸟站在光秃秃的树干高处，在长空中留下引人注目的身影。

O'Reilly出版的图书，封面上很多动物都濒临灭绝。这些动物都是地球的至宝。如果你想知道如何保护这些动物，请访问*http://animals.oreilly.com/*。

封面图片出自《Woods Illustrated Natural History》。

Modern PHP（中文版）

PHP正在重生，不过所有PHP在线教程都过时了，很难体现这一点。通过这本实用的指南，你会发现，借助面向对象、命名空间和不断增多的可重用的组件库，PHP已经成为一门功能完善的成熟语言。

本书作者Josh Lockhart是“PHP 之道”的发起人，这是个受欢迎的新方案，鼓励开发者使用PHP最佳实践。Josh通过实践揭示了PHP语言的这些新特性。你会学到关于应用架构、规划、数据库、安全、测试、调试和部署方面的最佳实践。如果你具有PHP基础知识，想提高自己的技能，绝对不能错过这本书。

- 学习现代的PHP特性，例如命名空间、性状、生成器和闭包。
- 探索如何查找、使用和创建PHP组件。
- 遵从应用安全方面的最佳实践，将其运用在数据库、错误和异常处理等方面。
- 学习部署、调优、测试和分析PHP应用的工具和技术。
- 探索Facebook开发的HHVM和Hack语言。
- 搭建与生产服务器高度一致的本地开发环境。

Josh Lockhart 开发了Slim框架，这是一个PHP微型框架，适合快速开发Web应用和API。他还是“PHP 之道”的发起人和当前的维护者。“PHP 之道”在PHP社区中很受欢迎，目的在于鼓励全世界的PHP开发者使用一些良好实践，传播一些高质量的资源。他是New Media Campaigns的开发者，这个机构位于北卡罗来纳州卡勃罗市。

“这些年我一直想推荐一本关于PHP的书，把这门语言和社区的当前状态说清楚，可是始终找不到合适的书。本书出版后，我终于有无需犹豫就能推荐的书了。”

——**Ed Finkler**

开发者和作者，

就职于 Funkatron.com

“编程领域一直在变化。PHP正在变化，开发应用的方式也在变化。Josh罗列了编写现代化PHP应用所需掌握的工具和概念。”

——**Cal Evans**

PHP

刮开涂层
查询真伪

O'Reilly Media, Inc.授权中国电力出版社出版

ISBN 978-7-5123-8093-6

定价：39.00元