

用Silex开发一个RESTful API（已完成）

taylorr

Published
with GitBook



Table of Contents

Introduction	0
LAMP	1
安装虚拟机	1.1
安装Vagrant	1.2
安装Ubuntu	1.3
安装Apache 2	1.4
安装PHP	1.5
安装MySQL	1.6
最后的微调	1.7
设置一个虚拟主机	1.8
一个趁手的IDE	1.9
Symfony 3和重要构件	2
Symfony 3	2.1
Doctrine	2.2
Twig	2.3
Composer	2.4
Symfony重要概念	3
MVC	3.1
Bundle/包	3.2
Route/路由	3.3
Controller/控制器	3.4
Entity/实体	3.5
Repository/仓库	3.6
Template/模板	3.7
Test/测试	3.8
藏书管理程序的结构	4
创建应用	5
建立版本管理	5.1
建立数据库	5.2
应用结构	5.3

建立数据库实体	5.4
样本数据	5.5
路由	5.6
模板	5.7
开始编写首页	5.8
书籍详情页面	5.9
书籍列表页面	5.10
书籍搜索	5.11
用户和后台	6
结语	7

Introduction 引言

This book is written in Chinese for Chinese PHP programmers (entry to intermediate level¹).

1. This means he/she at least has a basic knowledge in Linux and PHP programming.

这意味着他/她具有Linux和PHP编程的基础知识。 ↵

本书以中文写成，面向中文PHP程序员（初级到中级水平）。

After reading through this book, the readers shall have a better understanding in the following areas:

阅读完此书后，读者应该对如下的领域有一个更好的理解：

- MVC structure of a typical web application
- 一个典型的Web应用所使用的MVC结构
- Symfony 3 installation, bootstrapping
- Symfony 3安装和自举式配置
- Important concepts and terminologies in Symfony 3
- Symfony 3中重要的概念和术语
 - Controller, Routes, Entity, Repository, Templates
 - 控制器，路由，实体，仓库，模板
 - Bundle
 - 包

We will build from scratch a personal book collection site to learn Symfony 3 framework.

This site will have the following features:

我们将学习Symfony 3框架来从头建立一个个人书籍收藏站点。这个站点将有如下特性：

- A secured backend to CRUD a book entry and other related operations.
- 一个安全的后台来CRUD一本书籍和其它相关操作。
- A frontend to display book collections, and other related operations.
- 一个前台来显示书籍收藏和其它相关操作。

The full code repository can be found [here](#).

本教程所有代码可以在[这里](#)下载。

Let's get started!

让我们开始吧！

(All other contents in this book will be written in Chinese.)

(本书其它内容将均以中文写成。)

TR@SOE

2015.1.21 Updated: 2016.3.17

1 一个常规的**LAMP**设置

一个常规的**LAMP**设置将包含如下几个部分：

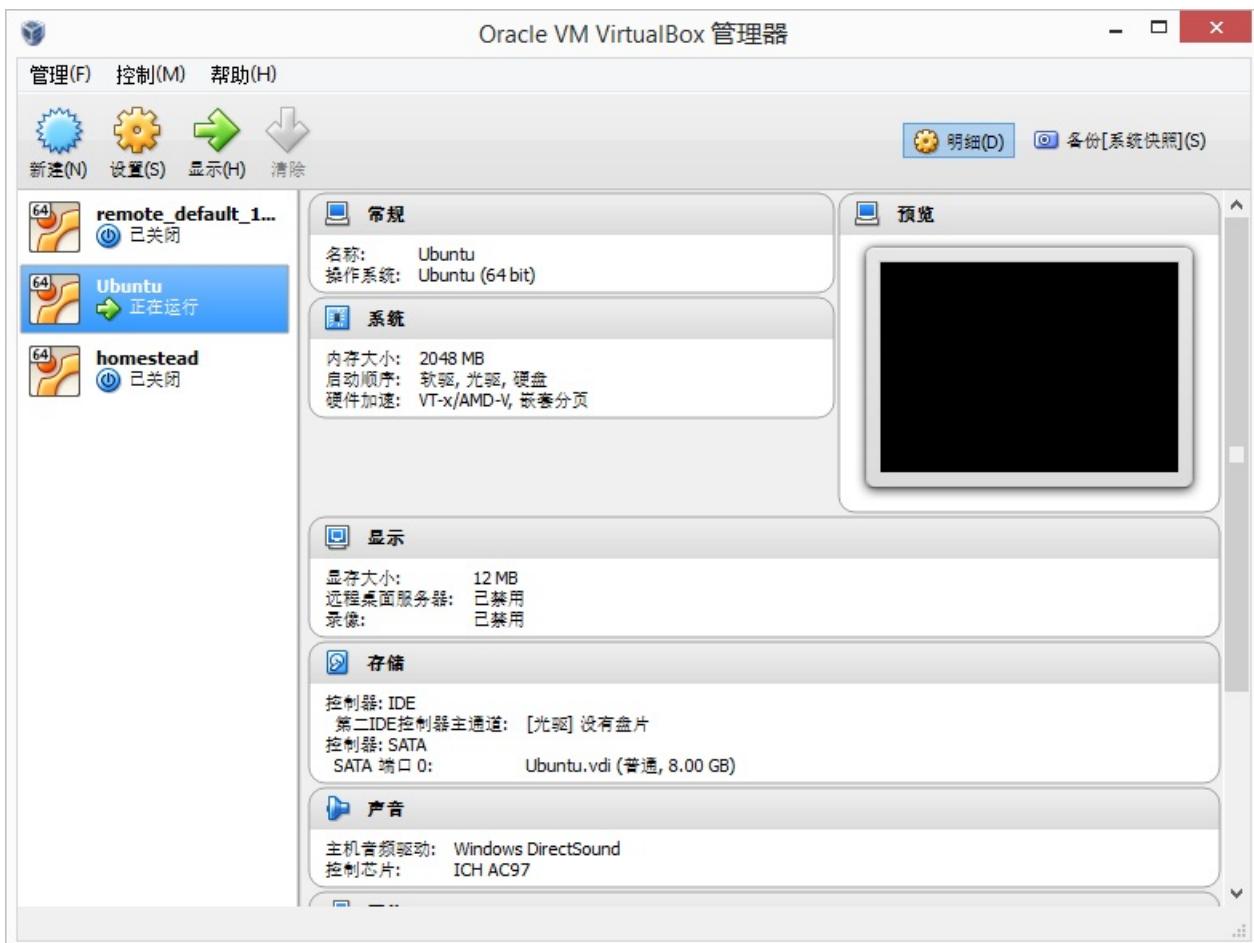
- Linux操作系统。我个人比较喜欢的是[Ubuntu](#)。
- Web服务器。我个人使用的是[Apache](#)，老牌、稳定、文档丰富。另外可选的比如[Nginx](#)。
- 数据库服务器。我使用的是[MySQL](#)。另外可选的比如[Postgre SQL](#)，[SQLite](#)，[MongoDB](#)等。
- Web应用编程语言。本书将只讨论[PHP](#)和[Dart](#)。对其它语言将不做讨论。

1.1 安装VBOX

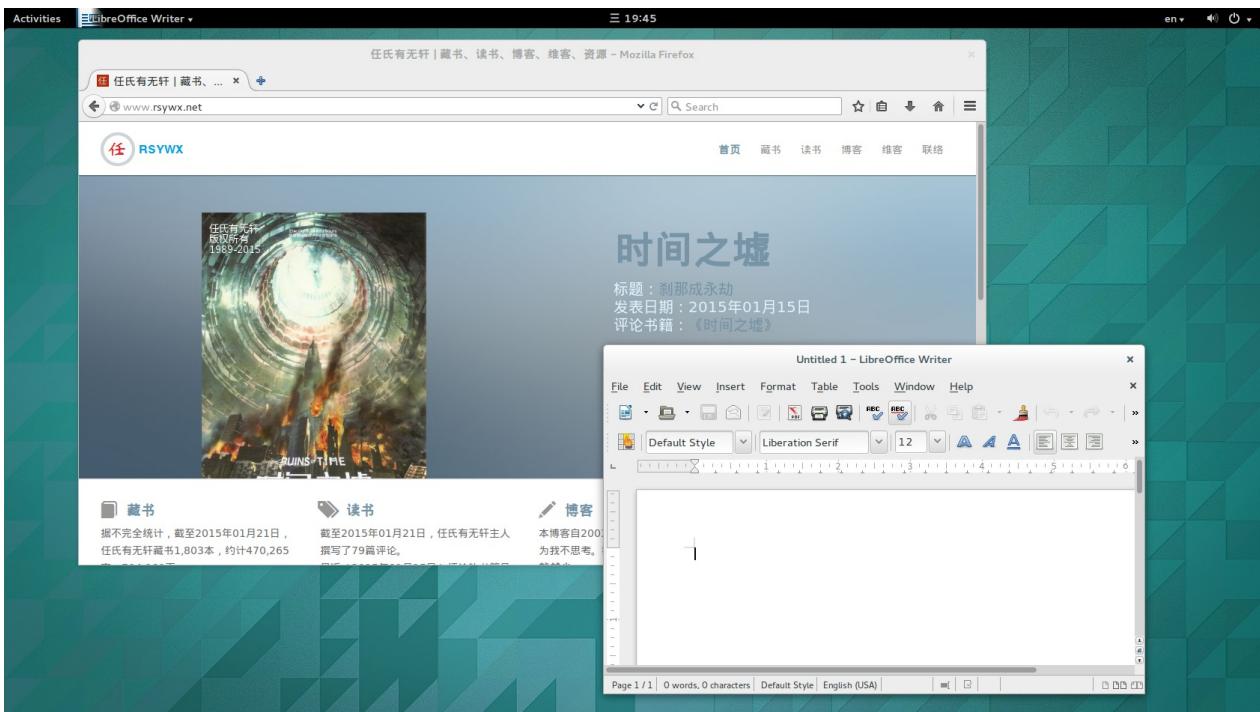
作为开发人员来说，如果能有一台独立的电脑安装Linux操作系统作为开发机当然是很好的。不过，在大部分时候，我们一方面要在Windows系统下工作，一方面要同时进行Linux开发。而且为了方便，我们也许不愿意在两台机器中切换来切换去。

这时，我会推荐安装一个虚拟机，这台虚拟机当然是基于Linux/Ubuntu的，然后在这台虚拟机中继续LAMP其它组件和别的相关软件包的安装。

在Windows上我一般使用[Oracle VirtualBox](#)。运行界面如下：



我安装了一个具有图形界面的Ubuntu并将其运行了起来：



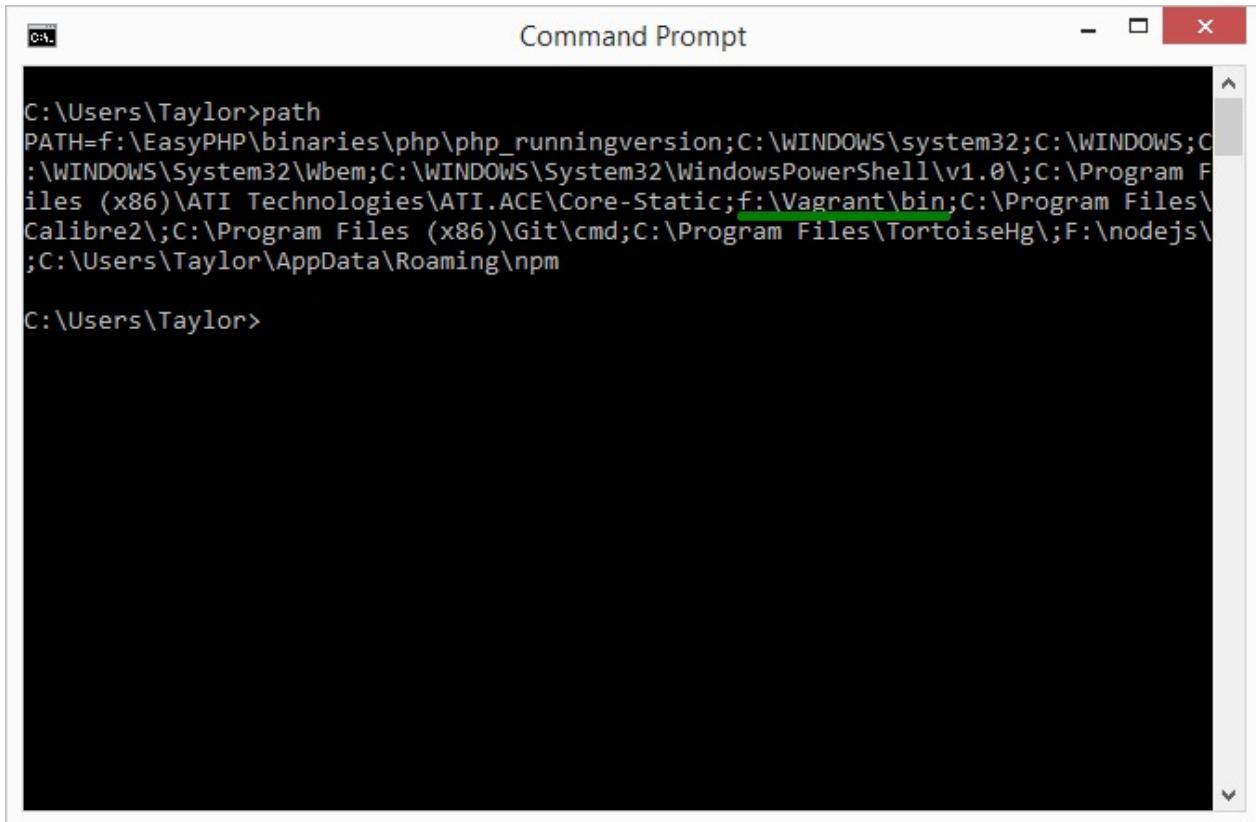
安装好VBOX只是第一步。

接下来我们要安装的是Vagrant这个专门用来搭建虚拟环境的程序。

1.2 安装Vagrant

首先我们要做的，是到Vagrant官方站点去[下载](#)我们要用的Vagrant程序。

Vagrant的安装很直观，不用多做评论。在安装结束后，需要确定一件事情，就是保证你的Vagrant可执行程序的目录是在Windows的path内：



```
Command Prompt -> C:\Users\Taylor>path  
PATH=f:\EasyPHP\binaries\php\php_runningversion;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\Program Files (x86)\ATI Technologies\ATI.ACE\Core-Static;f:\Vagrant\bin;C:\Program Files\Calibre2\;C:\Program Files (x86)\Git\cmd;C:\Program Files\TortoiseHg\;F:\nodejs\;C:\Users\Taylor\AppData\Roaming\npm  
C:\Users\Taylor>
```

这样我们才能在我们所选定的目录中都能执行vagrant相关的命令。

下一节我们要讲述如何安装Ubuntu，配置一个Vagrant的环境（称为一个box）。

1.3 安装Ubuntu

在Vagrant的术语中，安装一个系统（以及相应的应用）是一个创建一个“盒子”（box）的过程。所以，我们说要安装Ubuntu，实际上就是创建一个Ubuntu的盒子，然后在这个盒子中，再进一步地进行配置和应用的安装。

要创建一个盒子，我们需要一个基础。针对Vagrant，已经有太多现成的原始“盒子”存在。在本书中，我们用的是“ubuntu/trusty”。

1.3.1 下载盒子

我们先建立一个目录，用来存放我们以后要用的经过定制的盒子。比如在 `f:\vagrant_env` 这个目录。

然后我们访问<http://www.vagrantbox.es/>这个站点，寻找我们需要使用的原始盒子：

Box Name	Provider	Download Link	Downloads
Ubuntu Server Precise 12.04.3 amd64	Virtualbox	http://nitron-vagrant.s3-website-us-east-1.amazonaws.com/vagrant_ubuntu_12.04.3_amd64_virtualbox.box	380
Ubuntu Server Precise 12.04.4 amd64 (source)	Virtualbox	https://oss-binaries.phusionpassenger.com/vagrant/boxes/latest/ubuntu-12.04-amd64-vbox.box	547
Ubuntu Server Precise 12.04.4 amd64 (source)	VMware	https://oss-binaries.phusionpassenger.com/vagrant/boxes/latest/ubuntu-12.04-amd64-vmwarefusion.box	612
Ubuntu Server Trusty 14.04 amd64 (source)	Virtualbox	https://oss-binaries.phusionpassenger.com/vagrant/boxes/latest/ubuntu-14.04-amd64-vbox.box	547
Ubuntu Server Trusty 14.04 amd64 (source)	VMware	https://oss-binaries.phusionpassenger.com/vagrant/boxes/latest/ubuntu-14.04-amd64-vmwarefusion.box	612
Ubuntu Server Precise 12.04 amd64 (Chef 11.8.0, VMware Tools)	VMware	http://shopify-vagrant.s3.amazonaws.com/ubuntu-12.04_vmware.box	470.7

我们要用的是 `Ubuntu Server Trusty 14.04 (Virtual Box)` 这个盒子。根据给出的下载链接，我们先将这个“盒子”下载，保存在一个临时目录中（比如说：`f:\temp` 下的 `\trusty64.box`）。

然后，进入 `f:\vagrant_env` 目录，建立一个子目录，名字是任意的，但是最好具有指示性，比如：`f:\vagrant_env\remote`，表示这里是一个“远程”的系统。

在 `remote` 目录中，运行如下命令：

```
vagrant box add "trusty64" "f:\temp\trusty64.box"
```

这将我们刚才下载的那个原始盒予以“`trusty64`”的名字注册在Vagrant的环境中了。

然后运行下面这个命令：

```
vagrant init trusty64
```

开始初始化我们自己的Vagrant/Ubuntu环境。

初始完毕后，当前目录（`f:\vagrant_env\remote`）下会多出一些文件，其中最重要的是 `vagrantfile` 文件。它是我们这个定制盒子的配置文件。

1.3.2 配置我们的盒子

通过对 `vagrantfile` 文件的修改，我们可以配置我们的盒子。

1.3.2.1 修改我们盒子的IP

我们可以将盒子认为是一个操作系统，一台虚拟的电脑。我们对这个盒子的操作除了某些特定操作之外，必须通过SSH登陆到该系统后方能进行。所以我们需要给这台“电脑”分配一个IP地址。

Vagrant支持三种方式来进行SSH登录：

- 端口转发。我们可以直接SSH到 `localhost:2222` 来访问虚拟机的 22 端口，从而登录到虚拟机；
- 私有网络。我们分配一个与我们Windows机器所在网段不同的网段IP。这也是本书所用的方法。比如，我的Windows机器的IP是 `10.0.0.2`，那么给虚拟机分配一个 `192.168.2.100` 就是一个不错的选择。这将保证，该虚拟机只有Windows本机可以访问，而内网中别的电脑将无法访问。而我们可以通过SSH到 `192.168.2.100:22` 来完成登录。
- 共有网络。此时我们分配一个与我们Windows机器所在网段相同的网段的IP。

要将我们的虚拟机设置为私有网络，我们修改 `vagrantfile` 中如下的几行：

```
# Create a private network, which allows host-only access to the machine
# using a specific IP.
config.vm.network :private_network, ip: "192.168.2.100"
```

1.3.2.2 映射Windows目录到虚拟机目录

通常，我习惯在Windows下进行代码的开发。如果我们映射这个工作目录到Vagrant虚拟机中，我们就省去了文件拷贝、同步的麻烦。在任何一端对代码进行修改，在另一端就会立刻反应（因为本来就是同一个文件）。

缺省情况下，Vagrant会映射我们在Windows中启动Vagrant的目录（`f:\vagrant_env\remote`）到Vagrant虚拟机中的`/home/vagrant`目录。这两个目录是等效的。

我们还可以增加更多的Windows目录，将其映射到虚拟机中。

1.3.3 启动我们的虚拟机

一切配置完成后，我们就可以启动我们的虚拟机：

```
vagrant up
```

经过一段稍显冗长的过程后，我们的Vagrant虚拟机已经启动。我们可以简单地在我们的Windows机器中`ping 192.168.2.100`，来确定该虚拟机已经启动。

1.3.4 进入虚拟机系统

我们所启动的虚拟机，已经安装好了我们之前选择的`Ubuntu Server Trusty 14.04`。我们可以使用`putty`或其它第三方程序用SSH的方式安全地登录虚拟机。

缺省情况下，Vagrant盒子的超级用户是`vagrant`，登录密码是`vagrant`。

让我们看看登录后的界面吧：

The screenshot shows a terminal window titled "vagrant@vagrant-ubuntu-trusty-64: ~". It displays the following text:

```
vagrant@192.168.2.100's password:  
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-44-generic x86_64)  
  
 * Documentation: https://help.ubuntu.com/  
  
 System information as of Wed Jan 21 21:29:26 CST 2015  
  
 System load: 0.48 Processes: 98  
 Usage of /: 7.8% of 39.34GB Users logged in: 0  
 Memory usage: 22% IP address for eth0: 10.0.2.15  
 Swap usage: 0% IP address for eth1: 192.168.2.100  
  
 Graph this data and manage this system at:  
 https://landscape.canonical.com/  
  
 Get cloud support with Ubuntu Advantage Cloud Guest:  
 http://www.ubuntu.com/business/services/cloud  
  
 1 package can be updated.  
 1 update is a security update.  
  
 Last login: Wed Jan 21 19:32:11 2015  
 vagrant@vagrant-ubuntu-trusty-64:~$
```

1.3.5 在开始下一步之前

现在是一个很好的时机，对我们这个刚建立好的虚拟机进行一些维护工作。比如，更新一下 `apt-get` 的源（将其改为使用国内的仓库），对系统进行一次全面的更新，安装一些必要的支持软件等等。

1.3.6 停止虚拟机

我们可以用 `vagrant halt` 来关闭虚拟机。虚拟机关闭后，将不能接受任何指令。

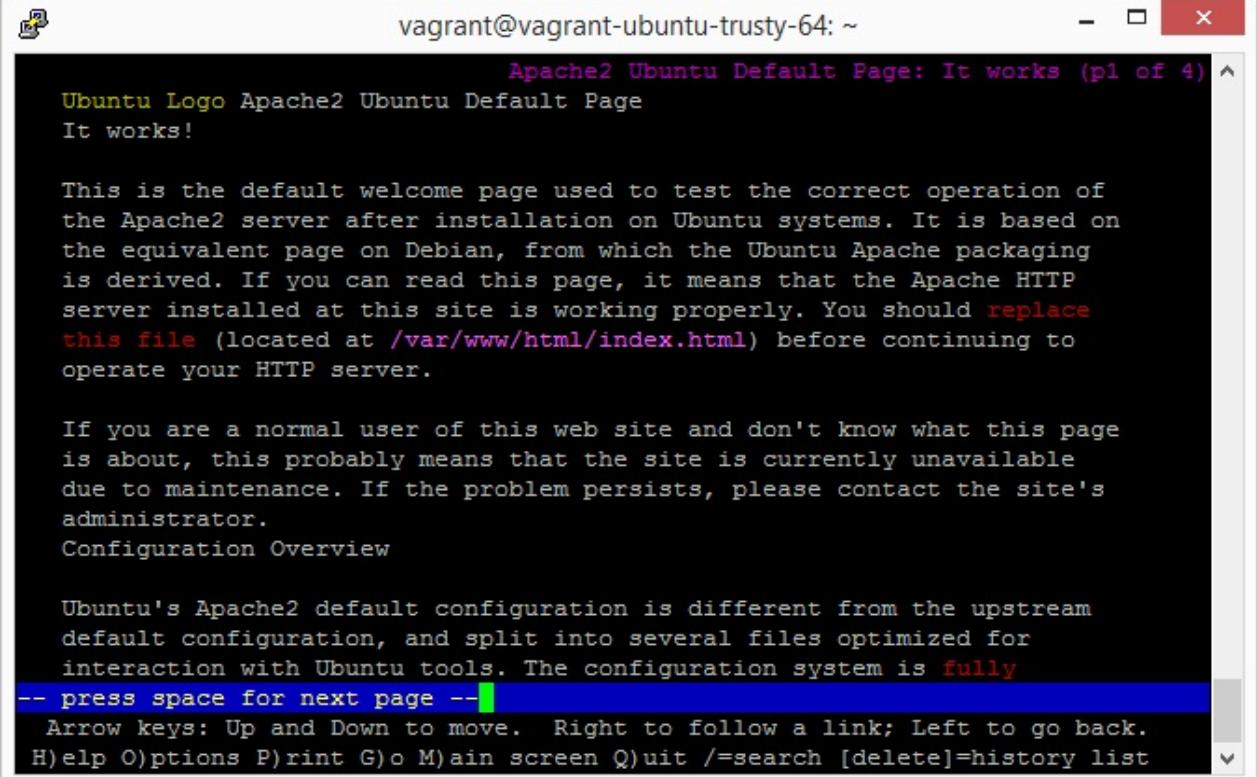
1.4 安装Apache 2

Apache 2的安装请参见相关资料。本书不再赘述。

安装完毕后，可以在虚拟机中运行：

```
lynx localhost
```

终端应该显示一些文字信息，表明Apache 2已经安装完成，运行正常：



vagrant@vagrant-ubuntu-trusty-64: ~

Apache2 Ubuntu Default Page: It works (p1 of 4)

Ubuntu Logo Apache2 Ubuntu Default Page

It works!

This is the default welcome page used to test the correct operation of the Apache2 server after installation on Ubuntu systems. It is based on the equivalent page on Debian, from which the Ubuntu Apache packaging is derived. If you can read this page, it means that the Apache HTTP server installed at this site is working properly. You should [replace this file](#) (located at `/var/www/html/index.html`) before continuing to operate your HTTP server.

If you are a normal user of this web site and don't know what this page is about, this probably means that the site is currently unavailable due to maintenance. If the problem persists, please contact the site's administrator.

Configuration Overview

Ubuntu's Apache2 default configuration is different from the upstream default configuration, and split into several files optimized for interaction with Ubuntu tools. The configuration system is [fully](#)

-- press space for next page --

Arrow keys: Up and Down to move. Right to follow a link; Left to go back.

H)elp O)ptions P)rint G)o M)ain screen Q)uit /=search [delete]=history list

严格地说，我们还没有完全完成Apache 2的安装——特别是模块和配置部分。我们将在以后讨论。

1.4 安装PHP

本书所讨论的Symfony框架最低需要PHP 5.3的支持。一般而言，在我们刚[安装好的Ubuntu盒子](#)中，其提供的PHP安装版本已经可以满足这个要求，但可能不是最新的，一般会是在5.4或者5.5这个版本。

如果你需要安装最新版本的PHP（5.6.x），那么需要对apt源进行一些修改。

完整的步骤可以参见[这个帖子](#)。

简单地说，有如下四个步骤：

1. `sudo apt-get update && sudo apt-get install python-software-properties`，安装 `python-software-properties` 这个应用。
2. `sudo add-apt-repository ppa:ondrej/php5-5.6`，添加PPA来源。
3. `sudo apt-get update && sudo apt-get upgrade`，重新更新apt。
4. `sudo apt-get install php5`。如果之前你没有安装过PHP，就执行这个命令。

如果一切顺利，在终端中执行 `php -v` 会提示如下信息：

```
PHP 5.6.4-1+deb.sury.org-trusty+1 (cli) (built: Dec 21 2014 19:28:16)
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
```

至此，PHP安装告一段落。我们将在稍后的章节继续配置。

1.6 安装MySQL服务器

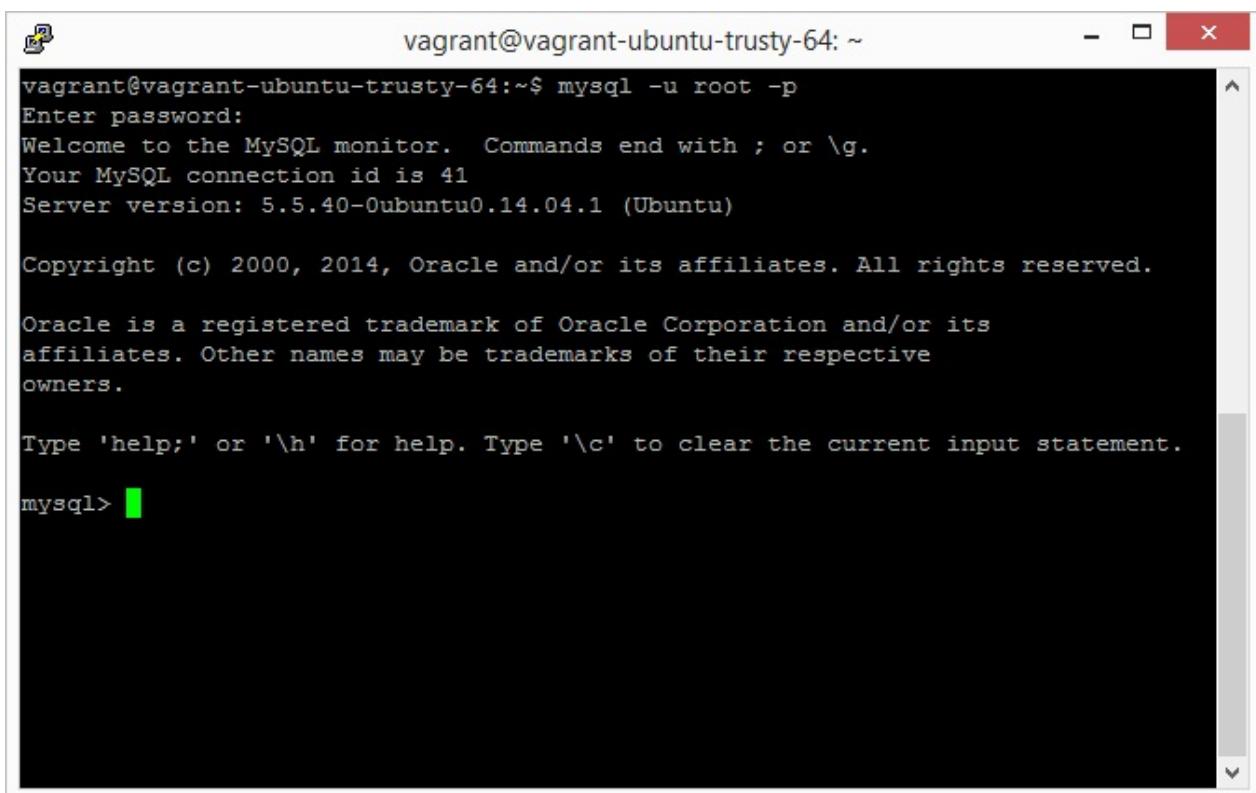
MySQL服务器的安装比较直接。我们不做详细讨论。

本书编写时所使用的MySQL服务器版本是5.5.41。你使用的MySQL版本不应该比这个低。

安装完成后，我们可以用 `mysql -u root -p` 命令，并输入 `root` 的密码来验证MySQL安装是否正确。

注意：这个 `root` 不是Ubuntu的 `root` 用户，而是在安装MySQL服务器时创建的MySQL的 `root` 用户。

一切顺利的话，你会看到这样的一个界面：



A screenshot of a terminal window titled "vagrant@vagrant-ubuntu-trusty-64: ~". The window contains the following text:

```
vagrant@vagrant-ubuntu-trusty-64:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 41
Server version: 5.5.40-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> [redacted]
```

1.7 最后的微调

我们已经安装了Linux（Ubuntu），Apache，PHP和MySQL。至此，一个基本的LAMP结构已经搭建完成。但是我们还缺少最后关键的几步，将这几个单元有机地整合起来。

1.7.1 对Apache的微调

我们需要对Apache进行的微调是使得它支持URL重写规则。这个是我们日后用Symfony编写应用并分发测试时必须的：

```
sudo a2enmod rewrite
```

该命令将使得Apache支持URL重写。

```
sudo service apache2 restart
```

该命令将重启Apache服务，我们刚才的改动从此生效。

1.7.2 对PHP的微调

对PHP的微调分为几个方面。一个是对 `php.ini` 配置文件的修改，一个是添加一些PHP的挂接模块（特别是MySQL模块）。

一般而言，在安装好Apache之后再安装PHP，则PHP的安装过程中会自动挂接一个 `libapache2-mod-php5` 的Apache模块。在重启Apache之后，Apache就开始支持PHP脚本的解析。

如果PHP的安装没有自动挂接该Apache模块，我们可以手动安装：

```
sudo apt-get install libapache2-mod-php5
```

然后重启Apache服务使模块生效。

1.7.2.1 修改 `php.ini`

PHP的配置文件有两个，分别处理PHP在命令行中执行和在Apache中作为模块执行时的表现。由于我们是要进行Web开发，所以我们要修改的是所谓Web下的PHP配置文件。该文件位于`/etc/php5/apache2`，对其修改需要`root`权限（我们登录时使用的`vagrant`用户就具有`root`权限）。

一般而言，我们不用太多地修改这个文件，但是有这么几项还是需要进行改动的：

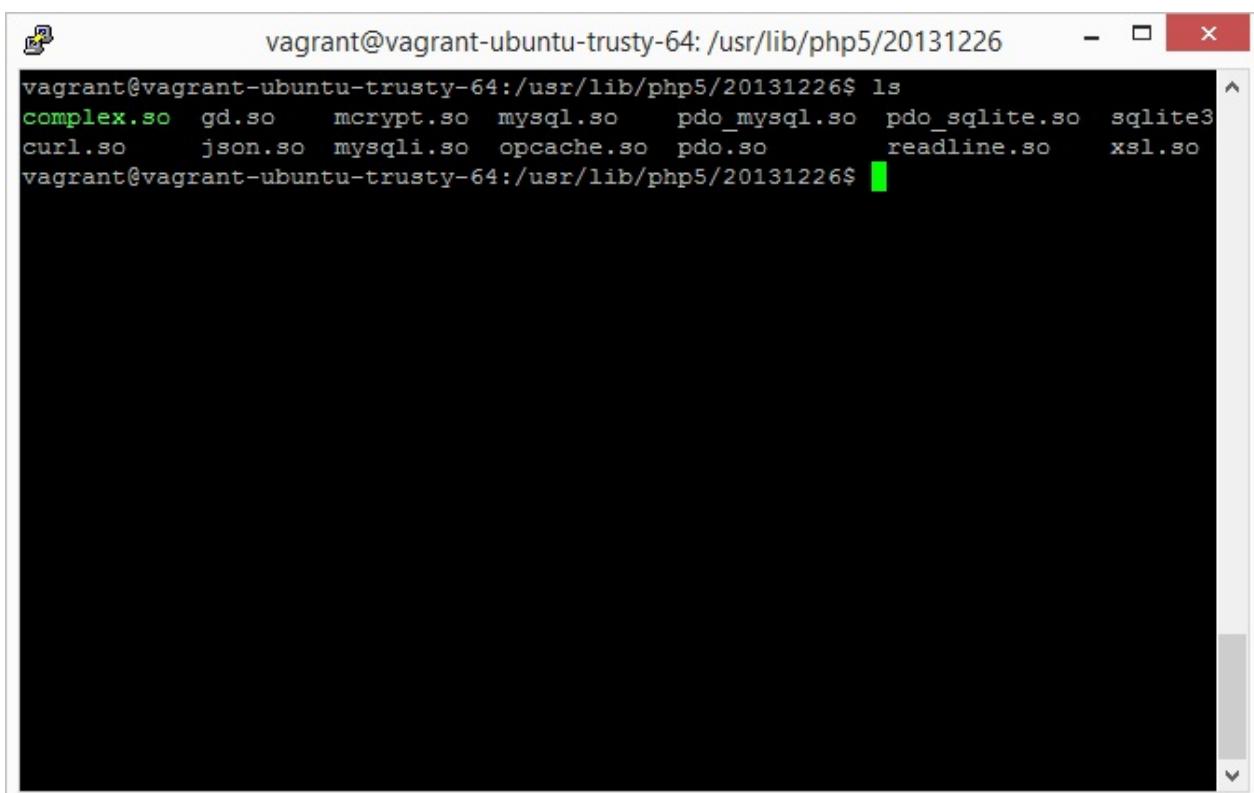
```
date.timezone = Asia/Shanghai  
;或者用UTC；或者你喜欢的时区  
  
upload_max_filesize = 100M  
;设置的这么大，是因为有时需要通过上传大图片  
  
post_max_size = 100M  
;这个值和上面这个值的设置最好保持一致
```

1.7.2.2 PHP模块

PHP模块又称为PHP扩展，是一些可挂载的程序（在Linux系统下是`*.so`），为PHP语言本身提供更多的功能。有关PHP模块的开发，已经超出了本书的范围。有兴趣的读者可以参阅我写的系列教程：[第一部分](#)，[第二部分](#)，[第三部分](#)。

到底要装哪些模块完全是个人喜好和开发要求决定，没有标准。

作为本书所讨论的Web应用，我选择安装了如下扩展：



```
vagrant@vagrant-ubuntu-trusty-64:/usr/lib/php5/20131226$ ls  
complex.so gd.so mcrypt.so mysql.so pdo_mysql.so pdo_sqlite.so sqlite3  
curl.so json.so mysqli.so opcache.so pdo.so readline.so xsl.so  
vagrant@vagrant-ubuntu-trusty-64:/usr/lib/php5/20131226$
```

这些扩展的具体介绍可以参见相关文档。

注意：`complex.so` 是我自己开发的一个复数模块，你在别处找不到，只能根据我的教程自己编译。没有这个模块也不会影响我们后面的开发。

这些模块的安装都可以使用类似的方法。比如，我们要安装 `mcrypt` 这个扩展，那么我们这么做：

```
apt-cache search mcrypt | grep php
```

这个命令会给出如下的信息：

```
php-crypt-blowfish - Allows for quick two-way blowfish encryption without requiring the M  
php5-mcrypt - MCrypt module for php5
```

其中的 `php5-mcrypt` 就是我们要安装的扩展名。然后我们用：

```
sudo apt-get install php5-mcrypt
```

命令就可以完成安装和配置了。

模块安装完成后，建议重新启动Apache服务。

1.7.3 对MySQL的微调

对MySQL的微调也分为两部分。一部分是和PHP的挂接，一部分是MySQL本身的微调。

1.7.3.1 和PHP的挂接

我们安装了PHP，也安装了MySQL，但是我们还没有在PHP中挂接MySQL扩展。只有挂接了这个MySQL扩展，我们才能在我们的PHP程序中使用MySQL数据库的功能。

```
sudo apt-get install php5-mysql
```

如果觉得有必要，你也可以安装一个 `phpmyadmin` 来对MySQL数据库进行管理：

```
sudo apt-get install phpmyadmin
```

但是我们也可以远程来管理MySQL数据库。

1.7.3.2 修改 my.cnf

MySQL的配置文件是：`/etc/mysql/my.cnf`。

我们需要记住：我们在Windows机器上进行开发，通过Vagrant的文件夹共享将我们的修改直接映射到Vagrant盒子中。相对我们的Windows机器，这个盒子是“远程”的（因为我们的盒子设置中，其IP和我们Windows机器的IP不是一个，更不在一个网段）。

缺省情况下，MySQL只对本级的MySQL请求做出响应，因此无法进行远程管理。使用PHPMyAdmin虽然可以解决这个问题，但是我比较喜欢用本地GUI的程序来管理远程MySQL，所以需要设置MySQL以放开远程管理。

```
bind-address = 127.0.0.1
```

修改完毕后，使用如下命令重启MySQL服务：

```
sudo service mysql restart
```

在终端登录MySQL，输入如下命令：

```
grant all privileges on *.* to 'root'@'%' identified by 'password' with grant option
```

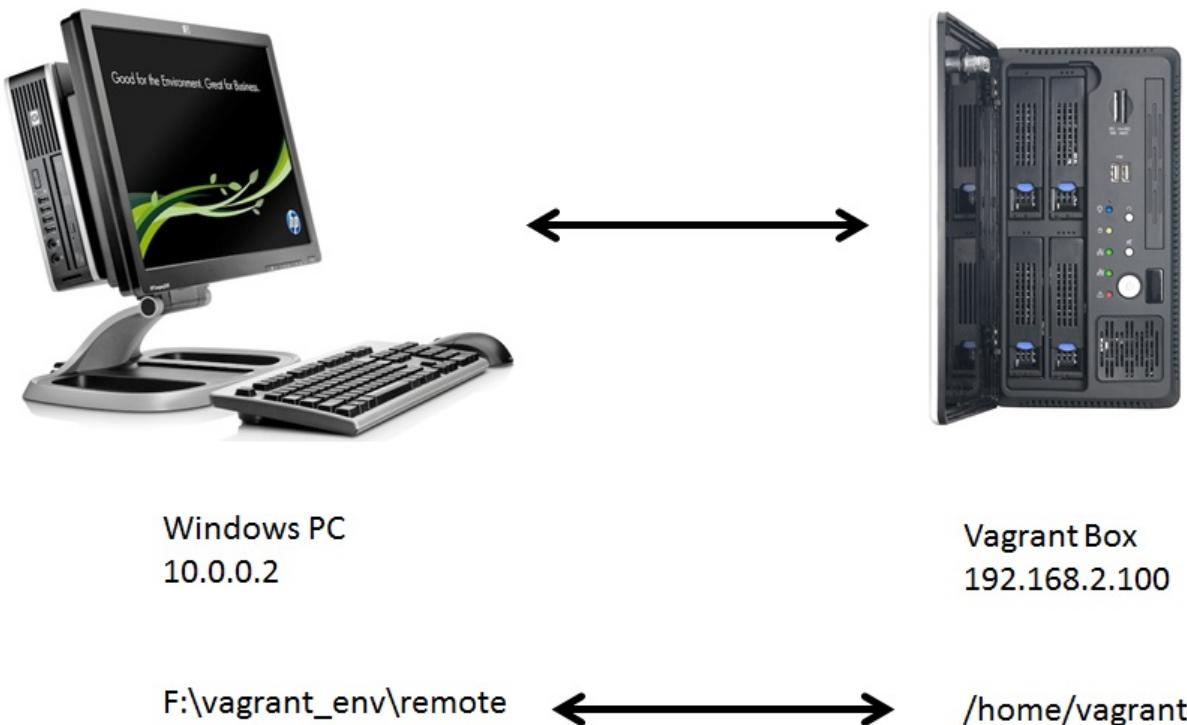
好了。经过这些修改后，我们在Windows机器中也可以用GUI程序来管理在Vagrant运行的MySQL服务器了。我个人使用的是Valentina Studio Pro（我为这个软件写过一篇[测评](#)，所以获得了一个免费的专家版授权）。还有很多免费的GUI程序可以下载使用，比如MySQL官方推荐的[MySQL Workbench](#)。

至此，我们的LAMP服务器设置基本完成——只有最后一步了：我们需要设置一个虚拟服务器来使得我们可以远程访问我们的站点。

1.8 设置一个虚拟主机

让我们再次回顾一下到目前为止我们进行的工作。我们已经有了一个运行正常的LAMP系统运行在一个Vagrant的盒子中。我们可以通过SSH远程登录到这个系统，进行常规的操作。同时，缺省地，我们有一个Windows下的目录（`f:\vagrant_env\remote`）被映射到了虚拟环境中的`/home/vagrant`中。

我的设想是，我们可以在Windows中利用比较丰富的GUI资源和程序进行PHP和Web应用的编程，对目录和文件的操作会即时反应到盒子中，然后我们在盒子中来提供这个Web应用的访问。在这种配置下，我们能最大限度地模拟最终的生产环境。用一张图来说明可能更好：



如果我们在Windows机器中用浏览器访问Vagrant盒子：`http://192.168.2.100`，我们已经能够看到由Vagrant盒子伺服的Web内容（现在当然只是Apache缺省的欢迎页面）。

但是我们想做得更完善一点。我想做到的是，在Windows中可以通过`http://vagrant`这样的方式来访问Vagrant中的Web应用，而且这个应用是放在一个我们指定的目录中，便于管理和分发。

1.8.1 设置Windows的hosts

要能在Windows中用`vagrant`这样的“域名”来访问远程服务器，我们需要修改Windows的系统文件`hosts`，它位于`C:\Windows\System32\drivers\etc`目录下。

注意：这是一个系统文件。你需要有Administrator权限才能对这个文件进行修改。

在这个文件中加入这么一行：

```
192.168.2.100 vagrant
```

保存并退出后，在Windows的命令行中输入：`ping vagrant` 应该会返回正确的PING信息。

如果我们现在用 `http://vagrant` 来访问我们的Vagrant盒子，我们还是会得到同样的Apache欢迎页面。这是因为，我们并没有在Vagrant中设置一个“记录”来处理针对 `vagrant` 的Web访问。

1.8.2 设置Vagrant的虚拟主机

我建议使用虚拟主机的方式来伺服我们的Web应用。自Apache 2.4以来，Apache对虚拟主机的设置和伺服做了一些改动，因此需要一些相关的指令来设置虚拟主机。

登录Vagrant后，我们进入 `/etc/apache2/sites-available` 目录，目前这里应该只有一个名为 `000-default.conf` 的站点配置文件和一个 `default-ssl.conf` 的文件。

要创建我们自己虚拟主机（站点）配置，我们可以从这个 `000-default.conf` 出发。先将这个文件做个拷贝：

```
sudo cp 000-default.conf 001-vagrant.conf
```

然后在终端中用文本编辑器对其进行编辑如下。

```
<VirtualHost *:80>
    ServerName vagrant

    ServerAdmin webmaster@localhost
    DocumentRoot /vagrant
    <Directory "/vagrant">
        Options FollowSymLinks Indexes
        AllowOverride All
        Order allow,deny
        Allow from all
        Require all granted
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined

</VirtualHost>
```

说明如下：

1. `ServerName` 设置为 `vagrant`。这和我们在上一步中对Windows中的 `hosts` 文件的修改所提供的名字是一样的。
2. `DocumentRoot /vagrant`。为了方便站点文件的同步，我简单地将该站点的根目录设置到了 `/vagrant` 这个目录。我们应该还记得，这个目录是映射到Windows下的 `f:\vagrant_env\remote` 目录的。
3. `<Directory "/vagrant">...</Directory>` 这一段基本应该按照样本书写。这样才能保证我们能在远程用 `http://vagrant` 来访问这个站点。

修改完毕后，我们用如下命令使该站点成为可用：

```
sudo a2ensite 001-vagrant.conf
```

这个命令其实就是在 `/etc/apache2/sites-enabled` 中做了一个符号链接到 `001-vagrant.conf` 而已。

最后，我们重启Apache服务。

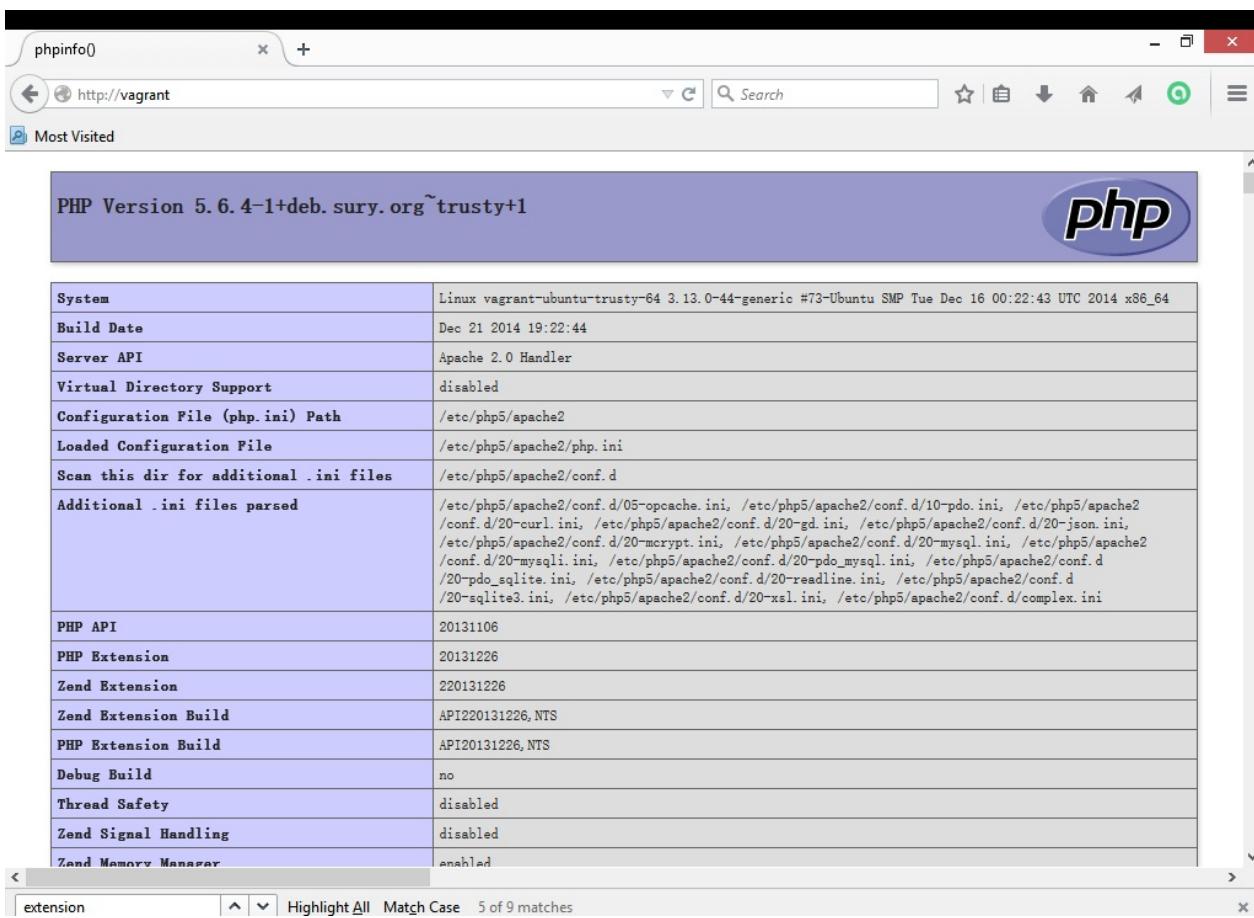
1.8.3 访问虚拟主机

一切就绪，让我们来测试一下。

在 `f:\vagrant_env\remote` 中创建一个 `index.php` 文件，内容很简单：

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

保存文件，然后在浏览器中访问这个地址：`http://vagrant`。浏览器应该显示如下内容：



你需要花一点时间来浏览一下这个非常长的页面。需要关注的地方有：

- Apache调用PHP时所使用的 `php.ini` 。在我的配置中，这个文件是 `/etc/php5/apache2/php.ini` 。
- PHP API版本。我的是 `20131106`，因为我用的是PHP 5.6.4。
- 相应的PHP扩展（模块）是否已经被启用，如MySQL，MySQL-PDO，GD，mcrypt等等.....

1.8.4 可能出现的错误

如果你不能看到这个页面，那么可能出错的地方有：

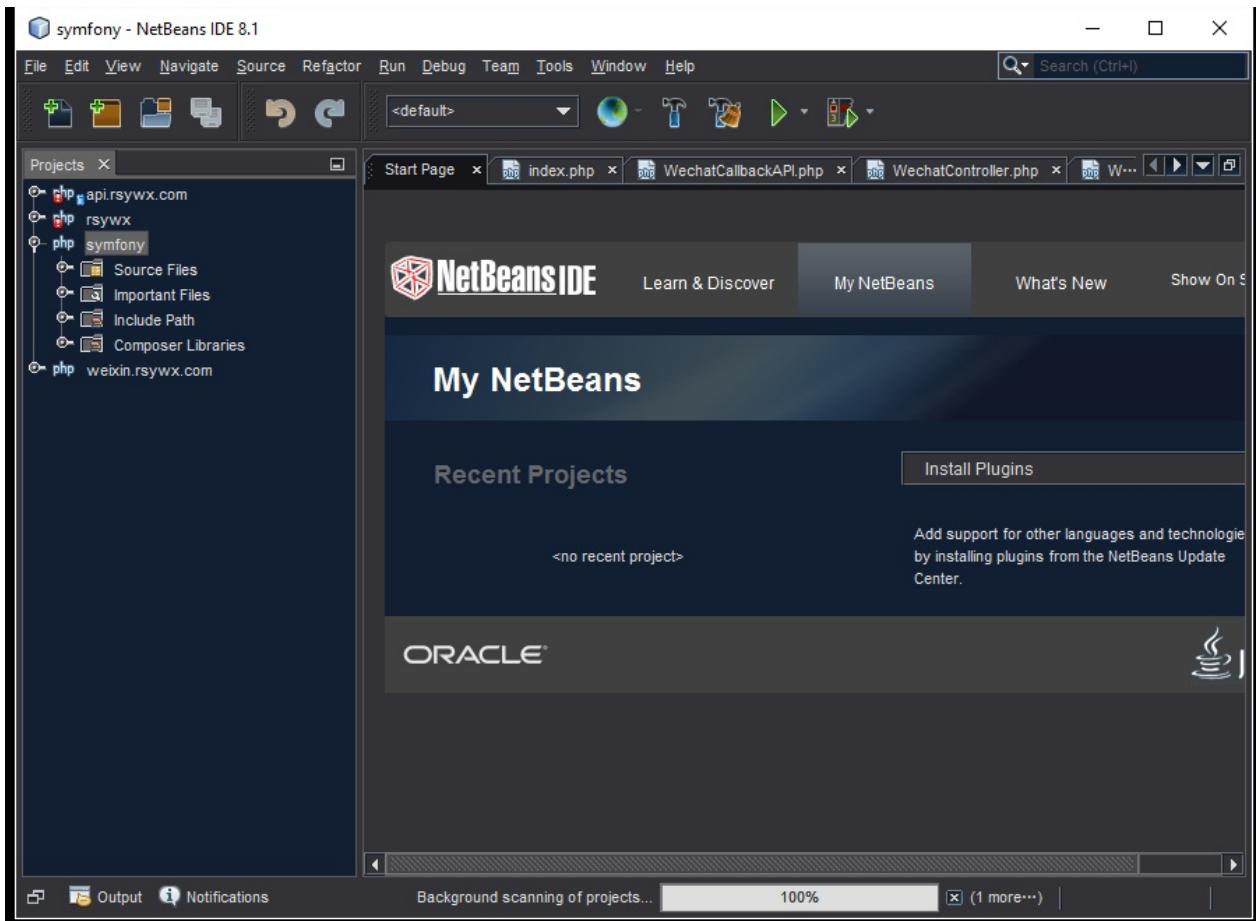
- 确定 `index.php` 文件内容正确，也在正确的位置；
- Apache服务已经启动；
- PHP安装正确；
- Windows下的 `hosts` 文件有 `vagrant` 这个域名的解析；
- Apache中虚拟主机的设置正确。

到此，我们的开发环境已经搭建成功。接下来我们要开始使用Symfony来开发我们的应用。

1.9 一个趁手的IDE

最后，我们需要一个趁手的IDE。

笔者使用的是[NetBeans]出品的NetBeans IDE。当前其版本是8.1。



我使用它的原因是：

1. 免费。
2. 丰富的插件。
3. 强大而实用的编辑器。
4. 速度尚可。
5. 提供一个Chrome插件NetBeans Connector方便我们在Chrome浏览器中进行调试。
6. 用惯了.....

用户可以用自己更趁手的IDE进行本书中应用的开发。

在IDE的选择上，没有更好，只有更舒服！

Symfony 3和重要构件

Symfony是一个全方位、重量级、基于PHP（5.4以上）的MVC框架。自从推出以来，就收到好评，树立了在PHP框架内的领先地位。Symfony不仅只是作为一个框架存在，它的设计理念就是非常的de-coupled（去耦合的），也就是说，Symfony这个框架中所有的构件都可以被独立出来单独使用。这为其它框架（如Laravel）的发展和构件的标准化做出了巨大的贡献。

我们将要使用的是Symfony 3的框架（以后简称为SF3，或者SF），当前的主版本为3.0。Symfony同时维护着一个2.8的版本。

2.1 Symfony 3

按照Symfony官方站点的[列表](#)，SF3提供了如下构件：

- **Asset**: 管理URL的生成以及WEB部件（CSS，JS，图片）的版本控制。
- **BrowserKit**：模拟一个WEB浏览器的行为。
- **ClassLoader**：只要你项目中的类符合PHP标准约定可以将其自动调入。
- **Config**：对配置值的操作
- **Console**：简化优美且可测试的命令行界面的创建。
- **CssSelector**：将CSS选择子转化到XPath表达式。
- **Debug**：提供简化PHP代码调试的工具。
- **DependencyInjection**：允许用户标准化并集约化应用中对象创建的方式。
- **DomCrawler**：简化HTML和XML文档的DOM导航。
- **EventDispatcher**：实现了Mediator模式。
- **ExpressionLanguage**：提供引擎来编译并估值一个表达式。
- **FileSystem**：提供文件系统的基本操作。
- **Finder**：提供一个直观流畅的界面来找到文件和目录。
- **Form**：创建、处理、复用HTML表单的工具。
- **Guard**：整合多重认证层。
- **HttpFoundation**：定义了HTTP规范的面向对象的层。
- **HttpKernel**：为创建可扩展和快速的基于HTTP的框架提供了基础。
- **Icu**：该部件于2014年10月后失效，请改用**Intl**。
- **Intl**：如果**Intl**扩展没有安装，则提供了一个候选库。
- **Ldap**：基于PHP的LDAP扩展，提供了一个LDAP客户端。
- **Locale**：该部件于SF2.3后失效，请改用**Intl**。
- **OptionsResolver**：帮助用户以选项数组来配置对象。
- **Process**：在子进程中执行命令。
- **PropertyAccess**：用一个简化的字符串形式从对象、数组中读取数据或者写入数据。
- **PropertyInfo**：提取PHP类的属性。
- **Routing**：将HTTP请求映射到一系列配置变量。
- **Security**：为复杂授权系统提供基础。
- **Serializer**：将对象转换成一个特定格式（XML，JSON，YAML等）或者反之。
- **Stopwatch**：提供代码调试的方式。
- **Templating**：提供创建任意模板系统需要的工具。
- **Translation**：为应用的国际化提供工具。
- **Validator**：用来验证一个类的工具。
- **VarDumper**：检视一个任意PHP变量的工具。
- **Yaml**：对YAML文件的操作。

需要说明的是，这些构件不是SF3能使用的构件的全部，只是SF3框架本身提供的、可被第三方单独使用的构件。SF还将借助其它第三方构件来建立起一个庞大的支撑系统，从而使开发者能在开放、统一的平台上进行编程。

需要了解这些构件具体的功能和用法的话，可以参考[SF官方文档](#)。

02.02 Doctrine

几乎可以这么说，所有的Web应用都不可避免地要和数据库打交道。

在“上古时代”，我们习惯于用 `mysql_connect()` 这样的语句来连接数据库，`mysql_fetch_array()` 这样的语句来获得数据。但是这些已经是过时的、不再推荐反而要被禁止的做法。即使我们不使用任何框架而编写一些快速的测试代码，我们也应该避免使用这些语句，而至少使用[PDO](#)或者[MySQLi](#)中提供的API。

但是在这一层次之上，我们还有所谓的ORM (Object-relational mapping，对象-关系映射)。

什么是ORM？简单的说，操作数据库的一个最直截了当的方式，是直接使用类似 `select * from a_table` 这样的语句来选择数据，用 `update a_table set field1=value1, field2=value2 where...` 这样的语句来更新数据……

这样做当然也行。但是有很多问题：

1. 不安全。SQL注入最容易发生在这种类型的语句中。
2. 麻烦。每次都要从数据库中获取一堆数据，然后根据用途分配给不同的变量。反过来也是一样，要根据不同的变量（可能是通过计算，或者是用户通过表单提交）来构造一个SQL语句从而使得数据、信息得到存续（persistent）。

鉴于此，ORM的目的就是能让我们用处理对象的方式来处理表中的数据。创建一个对象，用 `$obj->setField1()` 的方式来为对象的属性赋值，然后用类似 `$obj->save()` 这样的方式保存这个对象到数据库中¹，对象的属性映射到表格的字段。

在获取数据时，返回的记录也能自动映射到一个对象（或者对象数组），于是我们就可以用类似 `$obj->getField1()` 的方式来获得记录的字段值。

这是非常直观、非常高效、同时也非常安全的做法。

Symfony 2中使用的ORM是[Doctrine](#)。对这个ORM的详细解说已经超出了本书的范围。我们在后面的章节中会结合实际使用介绍在SF中使用Doctrine的方法。

Doctrine是SF中缺省使用的ORM。在我们[创建一个新的SF应用](#)时，Doctrine是缺省安装的。

由于本应用已经开发到了6.0版本，笔者对应用结构也有了全新的布局，所以在该版本的应用中，M模块其实已经不再存在，而改用RESTful API调用的方式。所以本教程中对Doctrine的描述近乎为0——因为几乎没有用到²！即便是在API接口的开发中，我们也没有用到Doctrine（因为Silex并不支持ORM）而用了更低层的DBAL（ DataBase Abstraction Layer ）操作。

注意：在任何情形下都不推荐使用 `mysql_connect` 等已经过时并马上要被淘汰、而且极不安全的语句！

1. SF中使用的Doctrine在保存对象时不是用的这样的语法，但是这不妨碍我们的理解。

↔

2. 在[建立数据库实体和样本数据](#)这两节中，我们还会用到Doctrine来为我们目前这个还没有任何数据的数据库填充样本数据。但是，从严格意义上说，这不是应用编程本身的一部分，因为样本数据只有在测试的时候才会用到。↔

02.03 Twig

Twig也可以说是Symfony系列产品中的一个，它的开发者也是Fabien Potencier。

Symfony 3缺省安装时也会安装Twig。其官方站点是<http://twig.sensiolabs.org/>。

Twig非常轻量级，语法也十分简明。简单说来，它只有两种语法：

- `{{ say something }}` 表示的是一种输出；
- `{% control something %}` 表示的是一种控制。

它还支持模板的嵌入、扩展、继承，以及一些所谓的过滤器（比如将一个日期型变量以某种格式输出）。同时，用户也可以在SF2中编写自己的过滤器。

Twig采用`obj.member`值这样的方式来访问传递到模板中的变量的属性、成员。所以，如果我们传递一个如下的变量到Twig模板中：

```
$summary['bc']=100;  
$summary['wc']=10000;
```

在Twig模板中，我们就可以通过这样的语法来访问其成员：

```
 {{summary.bc}} {{summary.wc}}
```

注意：Twig中所有用到的变量、对象都必须显式赋值。变量的定义可以在Twig内部进行，但更多的时候是应用通过控制器传递给模板的。

更多Twig的实际使用我们会在后续文章中讲到。

02.04 Composer

如今的PHP社区，Composer已经替代了过往所有的模块/包管理、安装系统。

获得Composer

要获得Composer，需要在命令行输入如下命令：

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

NOTE: 这个命令需要 curl 的支持。

执行上述命令后，一个名为 `composer.phar` 的文件就会下载到当前目录。

安装/更新应用所需的包

`composer.phar` 需要一个 `composer.json` 文件配合，来查找、安装、更新一个应用所需要的包。

一个最简单的 `composer.json` 文件可以只有这么几行：

```
{
  "require": {
    "silex/silex": "~1.2",
    "twig/twig": ">=1.8, <2.0-dev",
    "doctrine/dbal": "2.2.*",
    "symfony/twig-bridge": "~2.3",
    "symfony/form": "~2.3",
    "symfony/config": "~2.3",
    "symfony/translation": "~2.3",
    "symfony/locale": "~2.3"
  }
}
```

这是一个典型的使用 [Silex](#) 框架（另一个由SF2开发者开发的轻量级PHP框架）的应用的包依赖关系描述。

创建好这个文件后，我们可以用：`php composer.phar update` 这个命令开始安装、更新我们这个应用使用到的包。

使用Composer的更多细节，可以参考[官方文档](#)。

03 Symfony重要概念

我们会在后续章节择重点讲述一些SF3中的重要概念。

SF3是一个基于PHP的重量级框架，其学习难度是很高的。决定使用或者尝试使用SF3作为应用开发的人员必须做好心理准备。

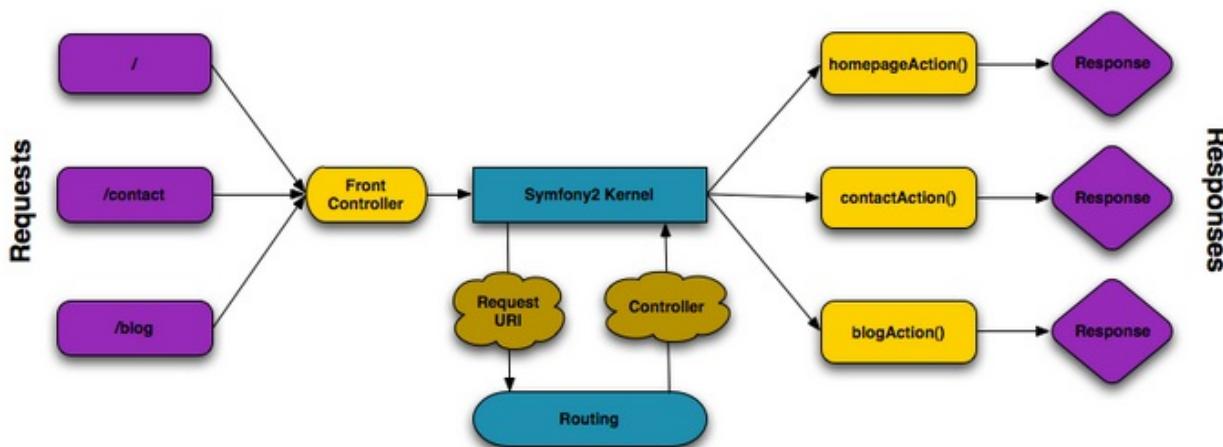
但是一旦掌握了基本的SF开发过程、语法约定和对象结构，用SF3编程是非常令人愉快的一件事。

03.01 MVC

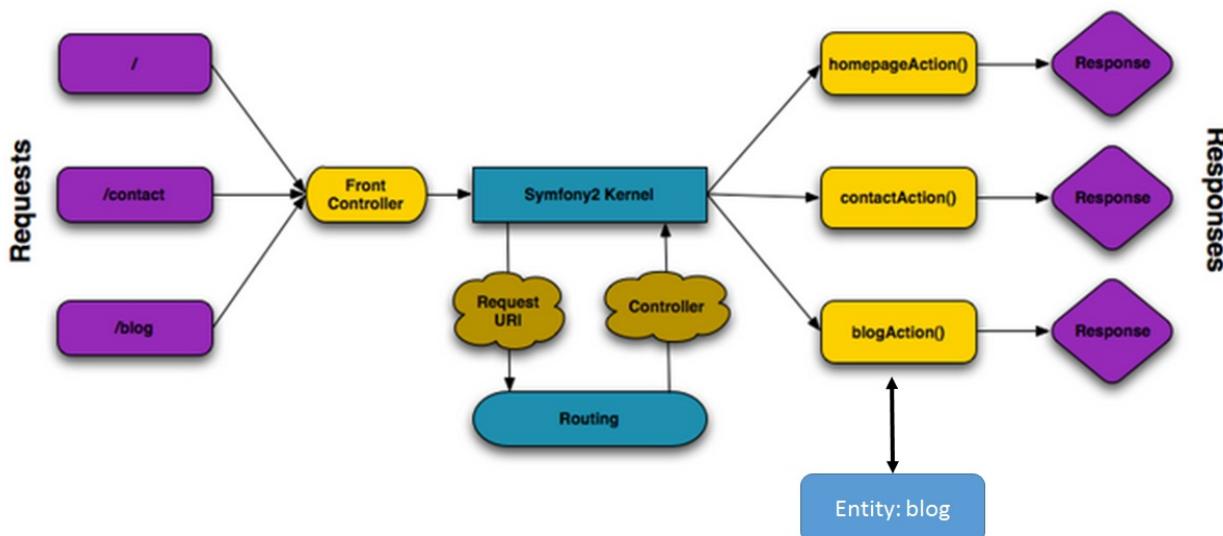
SF3是基于MVC层次的。MVC层次很好地隔离了一个Web应用的各个不同层面：

- **M(odel)**：模型层，也可以理解为数据库接口层。这个层在PHP对象（类、成员）和数据库结构（表格、字段）之间建立起一种映射关系。
- **V(iew)**：呈现层，或者叫展示层。我们可以简单地将其理解为我们在Web浏览器中看到的一个一个页面。
- **C(ontroller)**：控制层。它回到这样一个问题：页面中的内容（特别是动态内容）应该从哪里来？

SF3官方文档中有这么一张图：

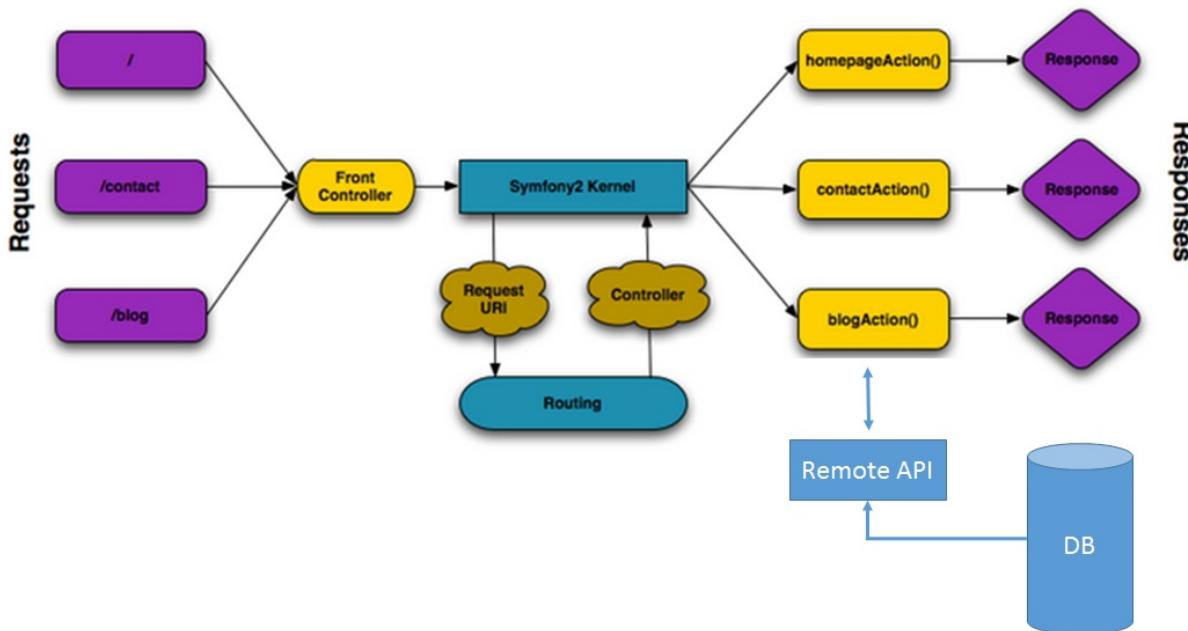


这里出现了C和V的代表，如 Front Controller , Controller , Response 等。没有出现的是M的代表。所以我们可以基于这张图来加以扩展：



这张图中的 `Entity: blog` 就是个M。在实际应用中，它应该是一个表格，保存了所有的blog的记录。某一个控制层的控制器（比如 `blogAction`）向 `blog` 这个实体进行查询和其它操作。根据操作的不同，`blog` 这个实体可能返回所有博客的列表，或者是经过筛选、排序过的列表等，也可能是CRUD中其它的操作。

重要更新：在之前的开发中，笔者都采用了标准的MVC模式。而在最新一次的升级开发中，笔者采用了更灵活、更去耦合化的方式。



在图示中我们看到，一个Controller不再直接和数据库/实体打交道，换句话说，在Controller中不再直接从数据库中获得数据，而是借由一个第三方（当然也是由笔者开发）的RESTful API封装层从数据库中获取数据并返回给调用的Controller。

这么做的好处有不少。

首先，Controller端的代码极为简化，统一为“准备参数、发起调用、获得返回、处理返回、显示模板”。而其中的调用过程变成单纯的调用远程API。

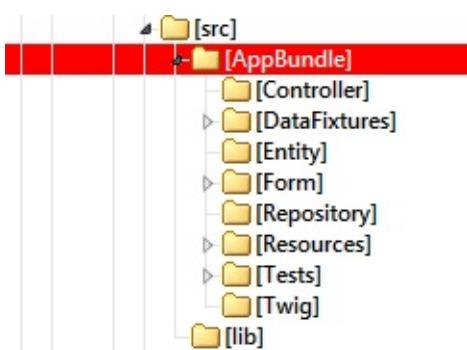
其次，API的开发和应用本身没有太大的关联，基本去耦合。

第三，API本身可以向外开放，获得复用。

03.02 Bundle/包

SF对一个应用的结构安排，是基于bundle（包）。这个应用的MVC结构，所有的配套文件都被集中在一个包里。这么做是有好处的。比如，一个应用不再仅仅是一个独立的应用，只要符合一定的规范，这个应用（包）完全可以嵌入到另外一个更大的应用之中。实际上，我们之前看到过的所谓第三方库，都是一个一个独立的bundle，通过composer安装到我们的应用中，成为我们应用中的一部分，并为我们的应用提供功能。

在实际应用中，我们可以将一个包理解为一个目录。一个典型的包可能包括如下的内容：



其中：

- **Controller**中存放所有控制器代码。
- **DataFixtures**中是样本数据填充。
- **Entity**中是所有的数据实体。可以简单地理解为一张张表格。
- **Form**中存放所有的表单类型，用于生成表单。
- **Repository**中是对数据实体的一些自定义操作。
- **Resources/config**中有一部分是以YML形式定义的Doctrine数据实体；另一部分可以存放针对本包的配置文件，如 `routing.yml` 文件。
- **Resources/views**中会存放所有本包要用到的视图模板，以Twig语法写成。
- **Tests**中可以存放各种测试文件，既可以是单元测试也可是功能测试。
- **Twig**中存放着专为Twig编写的定制过滤器。

在实际应用中，你的应用可能不会有这么多目录。一个典型的SF2安装，是没有 `DataFixtures` , `Form` , `Repository` , `Twig` 等目录的。

事实上，有些目录的命名也完全是任意的。上面截屏中的名字只是列出了一个我所开发的应用中的例子。

03.03 路由

在理解SF的路由之前，我们先了解一下什么是pretty URI。

现在的Web页面，基本都会用形如“`http://www.rsywx.net/books/01805.html`”的URI来表示一个资源。这个名为 `01805.html` 的页面并不物理存在于服务器上。在以前的编写实践中，它很可能就是要这样来表达的：“`http://www.rsywx.net/listbook.php?bookid=01805`”。

现代PHP框架（其实更恰当地说是所有现代框架）都抛弃了第二种很丑陋也极不灵活极不安全、也极不SEO友好的做法。而采用类似第一种这样的pretty URI方式。

和几乎所有现代Web框架一样，SF也是单入口的。所谓单入口，是说整个Web应用都以一个文件作为入口，作为调用其它控制器的总调度。在SF中，这个文件是 `app.php`（生产环境）或者 `app_dev.php`（调试环境）。

那么问题来了，如果我们只有一个PHP文件，我们怎么来定义一个URI的路径呢？比如说：`http://www.rsywx.net/books/01805.html` 需要将我们“带到”一个控制器，这个控制器能识别出参数（`01805`），然后进行相关的后续工作。

换句话说，在V->C的过程中（我们访问一个URI是V层次的动作，而对这个动作进行相应是C层次的操作），我们如何建立起这个映射？

我们需要的是所谓的路由。

我们先看一个典型的路由：

```
homepage:
    pattern: /
    defaults: { _controller: AppBundle:Default:index }
```

这个最简单的路由由三个部分组成：

- 名称：`homepage`。这是一个描述性的名称，可以随意起名，但是最好和其内容有点关联并有指示作用。
- 模式：`pattern: /`。这个模式定义了访问应用的入口。这个入口可以是对外的，也可以是内部的。即以本例来说，它定义的入口就是“`http://somedomain.com/`”，也就是常规意义上的首页。
- 选项：`defaults: { ... }`。这部分进一步定义了该入口的参数。其中，最重要的就是 `_controller` 参数。
 - `_controller`：指明处理该应用入口请求的控制器是哪一个。除了极个别情况，这是必须有的参数。控制器的指定形式是：**Bundle_Name:Controller_Name (or class name):Action_Name**。在本例中个，处理 `/` 这个URI请求的控制器动作是 `index`，

它在 `Default` 控制器中，并位于 `AppBundle` 这个包里。

从理论上说，一个应用可以开放的入口并没有上限。不过在实际应用中，有那么几十个也就差不多了。

参数

路由可以带参数。而参数有两种：一种是没有缺省值而必须提供的，一种是有缺省值而可以省略的。

带有参数的路由举例如下：

```
book_list:
    pattern: /books/list/{type}/{page}/{key}
    defaults:
        page: 1
        key: all
        type: title
    _controller: trrsywxBundle:Book:list
```

路由中的参数用形如 `{param_name}` 的形式定义。一般建议将各个参数用 / 分割以避免参数之间的混淆和最终URI的清晰。

在上面这个路由中，我们定义了三个参数：`type`，`page`，`key`。这些参数具体派什么用途我们会稍后讨论。

在 `defaults` 中，对这三个参数设置了缺省值。因此，如果我们只是简单地访问：`http://mydomain.com/books/list`，那么由于三个参数都没有提供值，就等同于访问：`http://mydomain.com/books/list/title/all/1`。

如果一个路由中的参数没有缺省值，那么必须在访问时提供。否则SF会报错。

路由参数的限定

对类似“`http://www.rsywx.net/books/01805.html`”这样的一个URL，我们可以这样来设置其路由：

```
book_detail:
    pattern: /books/{id}.html
    defaults: { _controller: AppBundle:Book:detail }
```

如果一个用户不小心使用了类似：`/books/abc.html`（一本书的ID不可能是字符）或者`/books/123.html`（一本书的ID必须是5位数字组成）这样的URI，会发生什么？

这个路由中指定的控制器还是会被执行，根据传递进来的参数（abc或者123）进行书籍的选择——当然就找不到了。如果我们能在请求进入控制器之前就对参数加以限定以避免这样的低级错误，不是更好吗？

此时，我们可以对“合法”的参数该是怎样做出限定。

```
book_detail:
  pattern: /books/{id}.html
  defaults: { _controller: AppBundle:Book:detail }
  requirements:
    id: \d{5}
```

这里的 `\d{5}` 是一个正则表达式，匹配5个数字。通过这样的限制，我们可以保证通过该路由传递过来的参数必然是5位数字。当然，这个数字是不是有对应的书籍是另外一个问题。

访问URI的方法

通常，我们输入一个URI或者通过点击一个链接访问一个URI时，都是进行的GET请求。一般来说，GET方法是最常见的，也是足够用的。但是，在处理表单的提交时，我们一般更会偏向于使用POST方法。也就是说，一个URI显示表单，然后提交的数据进入另一个URI进行处理。

因此，通常情形下，我们要设置两个路由：一个用来显示表单(比如 `register`)，一个用来处理表单(比如 `do_register`)。我们当然不希望用户在浏览器中直接访问 `do_register` 所对应的URI，因此有必要对访问路由的方法进行限制：

```
do_register:
  pattern: /create_user
  defaults: { _controller: AppBundle:User:create }
  methods: [POST]
```

通过指定 `do_register` 只能通过POST方法访问，就阻止了用户简单地在浏览器中输入“`/create_user`”来访问这个URI。

创建路由时的常见陷阱

- SF对路由的解析是由上到下的。也就是说，如果有一个路由的模式得到匹配，SF将不再匹配后续的路由。因此，我们必须注意一点，就是路径模式应该遵循“越精确、越特殊的模式越在前定义”的原则。
- 在定义路径模式时，一定要注意会不会出现可能的重复。

比如这两个路由：

```
books_with_tag:  
    pattern: /tag/{tag}  
  
add_tag:  
    pattern: /tag/add  
    methods: [POST]
```

第一个路由可以用来显示那些有着标记为 `tag` 的书籍，第二个路由用来为一本书籍增加一个 `tag`。这两个路由以这样的顺序出现的问题在于，如果我们通过一个表单提交了一些新的 `tag` 准备加到一本书籍上，那么我们期望的动作是 `/tag/add` 这个路由定义的动作，但是由于 `books_with_tag` 这个路由定义的路径模式在前，也匹配形如 `/tag/add` 这样的调用（此时这个路由的 `tag` 参数变成 `add`），且该路由没有说明不可接受 `POST` 方法，于是这个路由将被第一个匹配。于是我们的表单递交动作将不会被执行。

解决方法之一，是调换这两个路由的定义次序；其二，可以限制 `books_with_tag` 的方法为只接受 `GET`；其三，当然也可以修改其中一个路由的路径模式，使其不会产生误解。在实际操作中，我们可以根据需要选择一种方法来避免出现问题。

路由定义往往是应用开发的第一步——因为至少你必须创建一个主页吧？

03.04 控制器

控制器其实就是一个PHP类。

控制器的作用在于，接受来自路由的调度，进行相应的工作（获取请求的参数，进行数据库的查询或操作，对返回的数据加以进一步的处理，显示一个模板并对模板中的变量加以赋值）。

出于管理的需要，我们通常将对某个实体进行操作的工作集中归并到一个类中，这个类的名称、类中方法的命名都遵循一定的规范。

类的名称和类成员的名称

从之前我们讨论的[路由](#)中，我们看到这样一个路由：

```
homepage:
    pattern: /
    defaults: { _controller: AppBundle:Default:index }
```

我们知道这个路由表示的路径是 `/`，也就是一般意义上的站点首页的位置，而它对应的控制器是 `AppBundle:Default:index`，再回想一下我们在[包](#)这一章中展示的包结构图，SF的规定是这样的：

1. 所有的控制器都位于 `AppBundle\Controller` 目录下。
2. 定义控制器类的文件命名规范是：`Name+Controller.php`。而这里的 `name` 和该路由定义的 `AppBundle:Default:index` 的第二部分（也就是 `Default`）一样。所以，针对 `homepage` 这个路由，我们必然要有一个 `DefaultController.php` 的文件与之对应。
3. 这个文件必须定义一个`NameController`的类，且这个类必须派生于 `Controller` 类。
4. 鉴于路由定义的第三部分 `index`，它规定了具体采用什么动作。与之对应的是这个类中的成员函数。这个成员函数必须是 `public`，而且命名为 `indexAction`。也就是说，它的名称是路由定义中的第三部分 `action` 加上 `Action` 这个后缀。
5. 函数参数的定义必须和路由中的要求一致。参数出现的顺序并不是特别重要，但是名字必须和路由中指定的参数名称有对应。我们会在以后再更详细地讨论路由参数和控制器参数的问题。
6. 控制器类必须有自己的 `namespace` 声明。通常它就是该文件所在目录，因此它应该总是 `namespace AppBundle\Controller;`。

一个典型的控制器类的代码可能是这样的：

```
//File: src/AppBundle/Controller/DefaultController.php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Security;

class DefaultController extends Controller
{

    public function indexAction()
    {
        // Code for index action.
    }
    // More codes and more actions
}
```

上述代码中出现的一些 `use` 语句，我们在此不多做解释。只说明一点：它们是根据控制器中代码的需要所引入的命名空间。

关于控制器，本节就描述到这里。控制器是SF中最关键的一个概念，也是我们用SF编写应用时写代码最多、业务逻辑最集中的一个地方。

03.05 实体

在SF的术语中，实体（Entity）是一个对象，有自己的属性和方法。我们都知道，根据面向对象的编程理念，一个所谓的对象是一个封装的实体。这么来回说似乎有点循环定义的味道。但是，确实我们只能这样来理解。

在实际应用中个，我们通常可以将一个实体理解为数据库中某个表格中记录的PHP中的类实现。

我们可以简单地说：有一个 `user` 表格，保存了诸如用户名，密码，主页等用户信息，那么通过某种方式将这个表格映射到一个 `User` 实体，这个实体有着诸如 `username`，`password`，`homepage` 这样的属性，也有类似 `setusername` 这样的方法来设置某个属性。

或者，更SF的想法是，我有一个 `User` 实体，其中定义了诸如 `username`，`password`，`homepage` 这样的属性和一些方法来操作属性，我们可以要求SF和Doctrine根据我们这个实体的结构来创建相应的数据库表格以存续（persist）这样的用户信息。

这是一个很巨大的变动。对我们如何编程，先定义什么后定义什么的顺序以及这个顺序内涵的意义有着不寻常的影响。

以往我们总是先定义一个数据库结构，定义各个表格以及之间的关系，然后才是通过ORM将数据库结构映射回一个类。而现在，SF的推荐方式是先不要管底层数据库会怎样，我们先要关心的是我们的应用需要哪些实体的支持，这些实体怎样互相操作，又各自提供怎样的一些属性和方法。这些工作做完之后，才是数据库结构的映射和将来对象得以存续。

现有数据库再有对象是传统的思路；而先有对象再有数据库是SF提倡的思路。

在本书所讲述的应用开发过程中，我们会用到这两种不同的方式。

在最开始的时候，我们用传统的方法：先定义数据库然后导出，并映射到一个个实体。这是因为我们的应用定义很明确，要处理的对象也非常明确。

在开发过程中，我们会采用SF提倡的方法，对我们的实体进行一些微调，这些微调可能是会影响到数据库表格结构的。我们将会发现，这个反向（或者更应该说是正向？）的操作是无损的，不会影响数据库中现有数据。

也许，在我们的开发中，这样两个方向的调整还是需要进行若干次的。这样的循环没有一个固定的模式，我们要根据实际的需要和自己的经验来确定此时此地用哪个方向的映射最合适。

讲述了这么多实体的理论，我们来看一个典型的实体的代码。这是一个书籍表格的映射，文件位于：`src/AppBundle/Entity/Book.php`：

```
<?php

namespace tr\rsywxBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * BookBook
 */
class BookBook
{
    /**
     * @var integer
     */
    private $id;

    /**
     * @var string
     */
    private $bookid;

    ...

    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set bookid
     *
     * @param string $bookid
     * @return BookBook
     */
    public function setBookid($bookid)
    {
        $this->bookid = $bookid;

        return $this;
    }

    /**
     * Get bookid
     *
     * @return string
     */
    public function getBookid()
    {
        return $this->bookid;
    }
}
```

```
    }  
    ... ...  
}
```

以上列出的只是该实体很小的一部分。一般而言，一个实体中将包含所有属性（全部是 `private` 成员）和针对该属性的R/W操作（对于某些只读或者只写参数，R/W操作会只有一个，但无论如何，都是 `public` 函数）。

对于实体的更多讨论，我们在后续章节会结合编程的过程加以进一步的介绍。

03.06 仓库

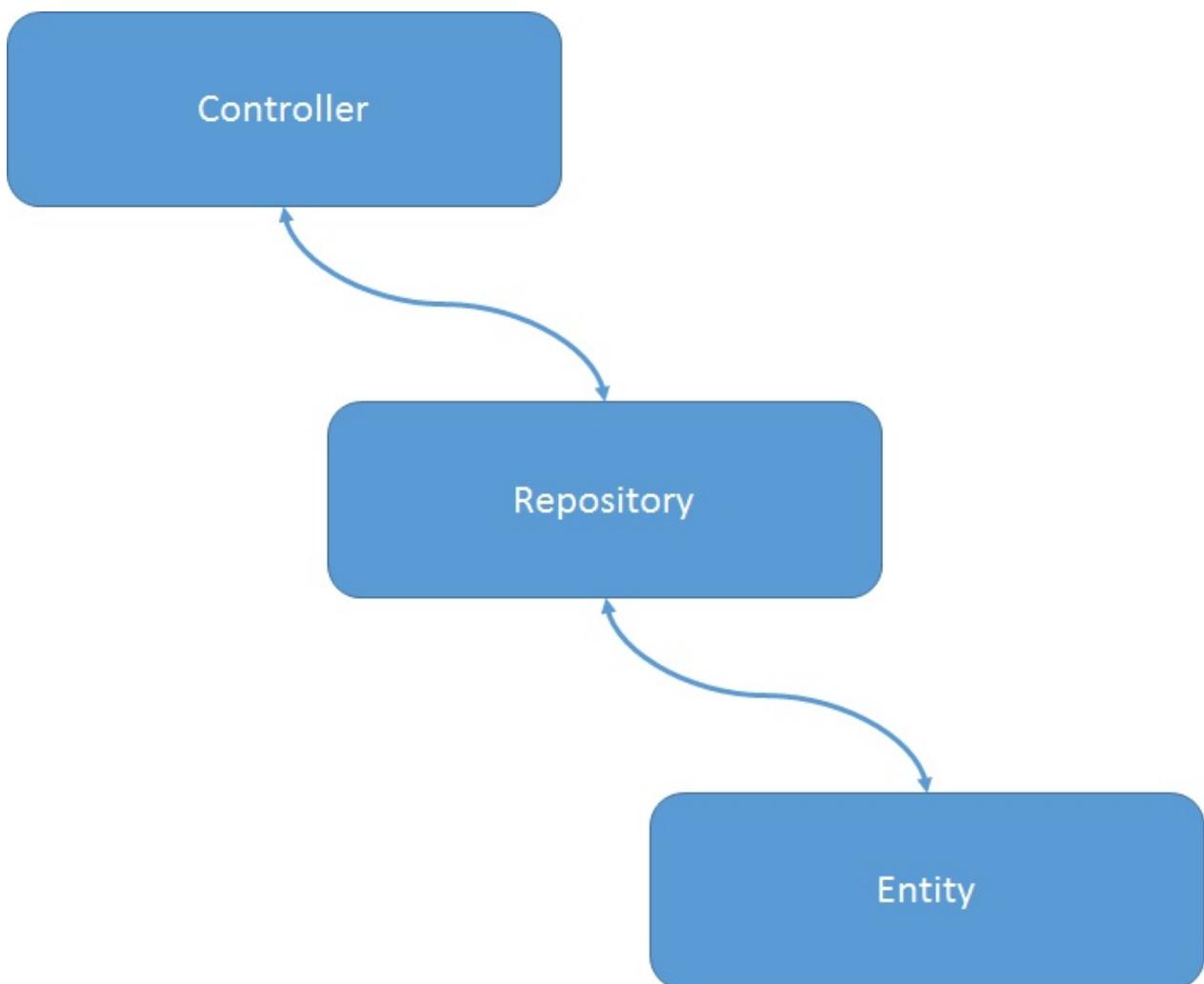
Repository（代码仓库）是SF中的另一个重要概念。

顾名思义，代码仓库就是存放（通常是通用的）代码的地方。在SF中，一般用来堆放进行数据库操作的代码。

SF为什么要增加这么一个额外的层？有两个原因：

1. 虽然一个实体提供了一些基本的数据库操作，如 `findOneBy` 之类的，但是肯定不能满足我们定制搜索的需要；
2. 如上的这个要求既不应该Model的任务，更不应该是Controller的任务（不符合DRY和代码重用原则）。

所以，我们有必要加入一个新的层，形成这样的一个关系图：



1. Controller只提出我要什么数据；
2. Repository只负责选择数据；
3. Model存放真正的数据。

现在要马上透彻理解这里的关系和区别还是比较困难的。我们会在后续文章中更深入地讨论这些。

我们来看一个典型的仓库的代码：

```
<?php

namespace AppBundle\Repository;

use Doctrine\ORM\EntityRepository;
use Symfony\VarDumper;

/**
 * StatusRepo
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class StatusRepo extends EntityRepository
{
    public function getStatusIn($start, $end, $count = 4)
    {
        $inter_s = new \DateInterval($start);
        $inter_e = new \DateInterval($end);

        $cts = new \DateTime();
        $cts->sub($inter_s);

        $cte = new \DateTime();
        $cte->sub($inter_e);

        $em     = $this->getEntityManager();
        $repo   = $em->getRepository('AppBundle>Status');
        $q      = $repo->createQueryBuilder('s')
            ->leftJoin('AppBundle>User', 'u', 'with', 'u=s.author')
            ->where('s.created>=:e')
            ->setParameter('e', $cte)
            ->andWhere('s.created<=:s')
            ->setParameter('s', $cts)
            ->setMaxResults($count)
            ->orderBy('s.created', 'desc')
            ->addOrderBy('s.id', 'desc')
            ->getQuery()
        ;
        $status = $q->getResult();

        return $status;
    }
}
```

`getStatusIn` 方法会提供在开始和结束期间的数据，缺省是提供4个。显然，这个数据的提供（搜索）不是简单地由 `findBy` 之类的实体方法可以完成的。

上面我们看到的构建SQL的方法是两种可用方法中的一种，即通过链接各个函数调用，在查询构造中加入 `join`、`where`、`order by` 等限定而得到我们需要的一个 `select` 语句。

对其的调用在Controller中一般这样进行：

```
public function indexAction()
{
    $em      = $this->getDoctrine()->getManager();
    $repo    = $em->getRepository('AppBundle:Status');

    $day    = $repo->getStatusIn('P0D', 'P1D', 5);
    $week   = $repo->getStatusIn('P1D', 'P7D');
    $month  = $repo->getStatusIn('P7D', 'P1M', 5);

    ...
}
```

这几乎是一个标准的调用模式。

重要更新：如今的一种开发方式将我们从这样的重度耦合中解放了出来。因此不再需要 `Repository`，而直接改用远程RESTful API的调用。详细的讨论可以参见[03.01 MVC](#)。

03.07 模板

SF2缺省使用Twig引擎作为模板的渲染。

一般的应用，都会由不止一个页面构成。

比如我的藏书管理程序包括首页，藏书列表，书评列表和某本书的详情页面，也包括“和我联络”这样的页面。这些页面有些是动态页面，有些是静态页面。但是它们都有类似的布局。我采用的是最常见的“头+主体+尾”的三段式布局。如下两个页面分别是藏书列表和书籍详情页面：

The screenshot shows a web application interface for managing a book collection. At the top, there is a navigation bar with links for 首页 (Home), 藏书 (Books), 读书 (Reading), 博客 (Blog), 维客 (Guest), and 联络我 (Contact Me). On the left, there is a sidebar with a user profile icon labeled '任' and 'RSYWX'. Below the sidebar is a search bar containing the text 'all' and a '搜索' (Search) button. To the right of the search bar is another input field labeled '直接去第几页' (Go to page) with a '直接去' (Go directly) button.

编号	书名	作者	购买/整理日期	位置
01810	白话本国史	【中国大陆】吕思勉	2015年03月24日	n5
01809	千年国门	【中国大陆】卢群	2015年03月24日	n5
01808	纽约客	【中国台湾】白先勇	2015年03月24日	n5
01807	The Death of Caesar	【美国】Strauss	2015年03月06日	z1
01806	洗澡之后	【中国大陆】杨绛	2015年02月05日	c2
01805	Zhou Enlai: The Last Perfect Revolutionary	【中国大陆】Gao Weiqian	2015年02月01日	z1
01804	时间之墟	【中国大陆】宝树	2014年01月14日	n5
01803	黄雀记	【中国大陆】苏童	2014年01月14日	n5
01802	在中国屏风上	【英国】毛姆	2014年01月10日	n5
01801	客厅里的绅士	【英国】毛姆	2015年01月10日	n5

Below the table are four navigation icons: double left, single left, single right, and double right.

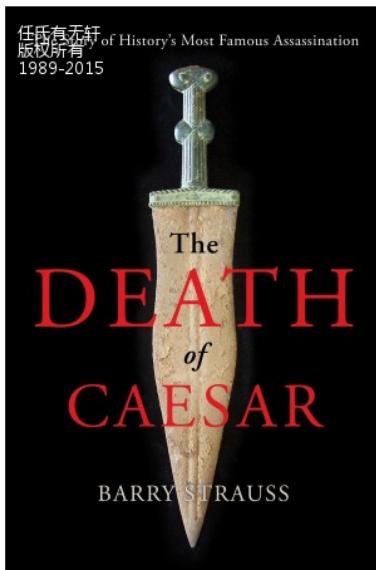
版权说明

任氏有无杆, V5.1, 1989 - 2015, 基于Apache 2 + MySQL 5 + PHP 5.5开发
其它技术: Symfony 2.6, Bootstrap, Dart

除非另有显式声明, 本作品均采用知识共享署名-禁止演绎 3.0 Unported 许可协议进行许可。

推荐使用Chrome, FireFox等现代浏览器浏览, 分辨率1680*1050以上。

Originated from HackerThemes. Customized by RSYWX.net.



The Death of Caesar

ISBN：978-1-4516-6882-7

作者：Strauss (美国)

价格：RMB 120.00

购于：2015/03/06，网络购买/网购



TAG

共和 凯撒 历史 帝国 罗马 [增加更多TAG»](#)

豆瓣TAG

豆瓣给不出任何TAG



豆瓣评分

(豆瓣找不到)



浏览次数

53 (上次访问时间是2015年04月03日17时50分38秒)

简介：

The Death of Caesar: The Story of History's Most Famous Assassination A book review in WJ can be found here:
<http://www.wsj.com/articles/book-review-the-death-of-caesar-by-barry-schiff-1426279394>.

任氏有无轩主人评论：

豆瓣简介：

(豆瓣找不到)

更多信息参见：(豆瓣找不到)

更多书籍信息

出版社：	(未指定)
出版日期：	2015/03/01
印刷日期：	2015/03/01
版次：	1.1
装帧：	电子书
千字数：	0
页数：	670
分类号：	
藏书位置：	z1

版权说明

任氏有无轩，V5.1，1989 - 2015，基于Apache 2 + MySQL 5 + PHP 5.5开发

其它技术：Symfony 2.6, Bootstrap, Dart



除非另有显式声明，本作品均采用知识共享署名-禁止演绎 3.0 Unported 许可协议进行许可。

推荐使用Chrome, FireFox等现代浏览器浏览，分辨率1680*1050以上。

社交网络



很明显，头部的内容基本保持不变（站点标识，导航条），尾部的内容页基本不变（版权申明等）。变化的是中间主体的内容。如果是藏书列表，就应该是分页显示的书籍列表信息；如果是书籍详情，就显示某本书更详细的信息。

针对这样的布局，在模板设计时我们应该遵循DRY原则，将相似的部分抽取出来成为一个独立的部分并使得这些部分可以在不同的页面中重用。

Twig提供了丰富的功能来完成这些工作。

include 一个子模板

在我们刚才讨论的页面布局中，头（`header`）和尾（`footer`）是相对独立和固定的内容，所以可以作为独立的模板存在，并被包含入主模板。

因此，主模板（`layout.html.twig`）的书写大致是这样的：

```
<body>
    <!-- Header Section -->
    {% include "AppBundle:default:header.html.twig" %}
    {% block content %}
    {% endblock %}
    {% include "AppBundle:default:footer.html.twig" %}
</body>
```

是的，就这么简单。特别请注意 `block content` 到 `endblock` 这两行。这两行定义了一个名字为 `content` 的内容块，但是没有任何内容。这部分内容要由基于该主模板派生的各个模板去填充。

`include` 模板还可以传递参数。比如下面的这个例子：

```
{{ include ("AppBundle:default:slider.html.twig", {'random':random}) }}
```

这个指令在包含 `slider.html.twig` 模板的时候，也会同时将本模板中 `random` 这个变量传递到被包含的这个模板中的 `random` 变量，从而达到变量在两个模板中传递的目的。

`include` 严格来说，是个控制结构，但是我们在上面的代码中看到了两种用法都是可行的。

extends 一个模板

上文定义的 `layout` 模板，只是一个框架，一般不会在我们的应用中单独呈现。我们要呈现给用户的页面都是基于这个模板派生而来。

我们来看一个比较简单的书籍列表的页面模板：

```

{% extends 'AppBundle:default:layout.html.twig' %}

{% block meta %}
    <meta name="description" content="任氏有无轩书籍列表，第{{cur}}页，总{{total}}页。">
    <meta name="keyword" content="任氏有无轩, 列表, 索引">
{% endblock %}

{% block title %}任氏有无轩 | 藏书列表 | 第{{cur}}页，总{{total}}页{% endblock %}

{% block content %}
    <div class="widewrapper">
        <div class="container content">
            <div class="row">
                .....
            </div>
            <div class="row">
                <section id="data" class="col-md-12">
                    <table class="table table-striped table-hover">
                        .....
                    </table>

                </section>
                <section id="pagination" class="col-md-12">
                    .....
                </section>
            </div>
        </div>
    </div>
{% endblock %}

```

最关键的是要 `extends` 一个主模板。然后在本模板中，针对主模板中定义的不同块（`block`），如果需要加以覆盖的，就用：`{% block blockname %}...{% endblock %}` 的语法加以重写。如果我们不覆盖主模板中的某个块，那么在本模板中会显示主模板中该快的内容。

在模板中嵌入一个控制器

模板的第三种用法，也是比较高级的用法是在模板中嵌入一个控制器。

我们考虑我的藏书管理程序首页的一个用例。在该页面中，我需要在一个幻灯片效果中显示一本随机挑选的书，在另一处 `<div>` 中显示三本随机挑选的书。

一种常规的做法是，在显示首页的控制器（`AppBundle:Default:indexAction`）中，通过调用相关的仓库方法获得这些书，然后通过变量传递的方式传递给相应的模板，并在模板中显示。这样做是可以达到目的的。

但是这么做可能有问题。如果这样的显示要求是多个页面都要求的，我们会面临在A控制器中调用仓库方法然后获得数据传递给a模板；在B控制器中调用同一个仓库方法获得数据然后传递给b模板……的过程。这里有重复的过程：都要调用一个仓库方法。而且，假定这一方法调用后返回数据的呈现在不同模板中也是一样的，重复性会更大：我们需要在各个模板中 `include` 一个子模板，然后传递给这个子模板以数据。这个过程更繁琐。

因此，比较推荐的方法是，在模板中内嵌一个控制器，这个控制器负责获取数据并渲染一个模板，而该模板的渲染内容将嵌入当前调用模板中。

我们来看一个实例，是用来显示与一本书相关的tag的。

首先是当前模板（`detail.html.twig`）中对控制器的嵌入：

```
<div class="text">
    <h3>TAG</h3>
    <small>{% render ('AppBundle:Book:tagsbyid', {"id":book.id}) %}</small>
    <a class="btn btn-info btn-sm" data-toggle="modal" href="#addtag" >增加更多TAG </a><br>
</div>
```

然后是显示某本书的tag的控制器：

```
public function tagsbyidAction($id)
{
    $em = $this->getDoctrine()->getManager();
    $tags = $em->getRepository('AppBundle:BookBook')->getTagsByBookId($id);

    return $this->render('AppBundle:book:tags.html.twig', array('tags' => $tags));
}
```

这里通过调用仓库的方法（`getTagsByBookId`），获得与这本书关联的tag数据，然后再通过 `tags.html.twig` 模板显示：

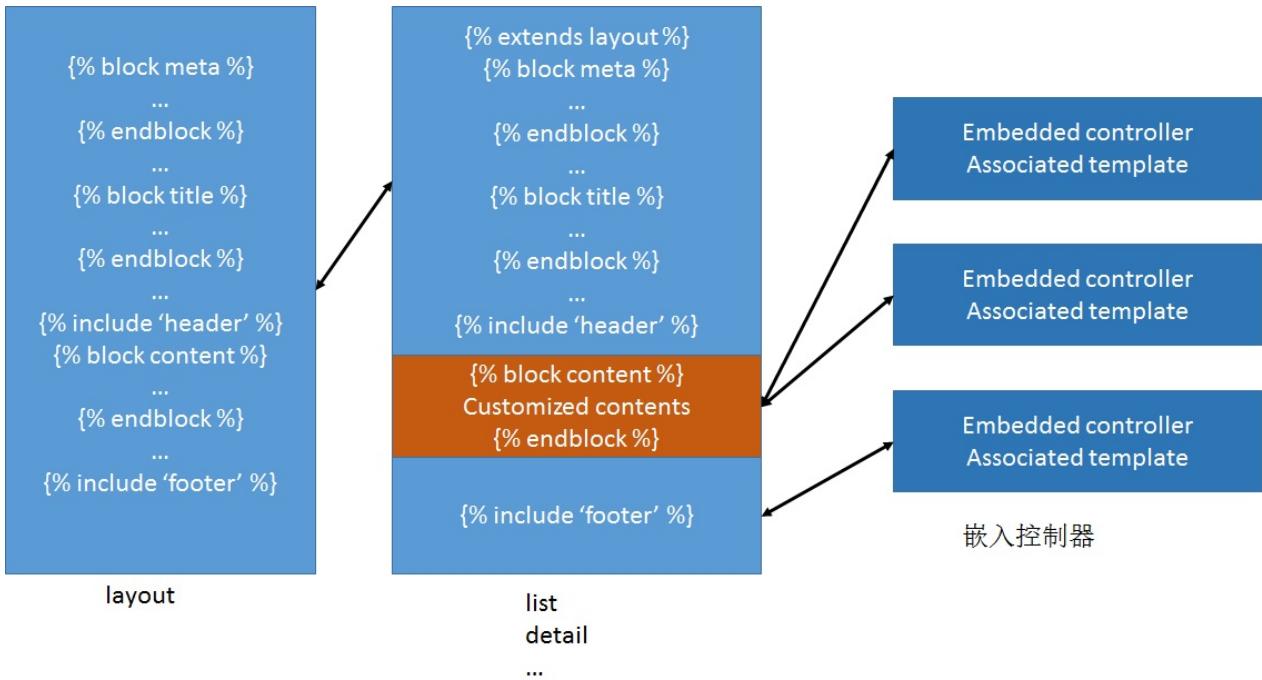
```
{% for tag in tags %}
<a href="{{path('book_list', {'type':'tag', 'page':1, 'key':tag.tag})}}>{{tag.tag}}</a>
{% endfor %}
```

这样的安排是非常去耦合化的，便于程序的开发和维护。

小结

由于Twig的语法相对简单，所以我们没有在之前的[Twig介绍](#)和本节的介绍中过多着眼其语法的介绍——这部分会在后面的开发过程中结合实践逐一介绍——而更多地介绍Twig模板的设计过程和一些重要的概念。

我们以上讨论的内容可以用如下这张的图来说明：



03.08 测试

只要你编程，就一定有错误。

作为程序开发的一个基本要求，我们必须对程序是否能正确运行进行测试。在PHP的世界中，我们可以使用[PHPUnit](#)，它和[Symfony](#)的配合是非常好的。

测试分为两种，一种是单元测试（Unit Test），一种是功能测试（Functional Test）。
PHPUnit可以配合SF3完成这两种测试。

具体的测试用例，我们会在后面编程的时候加以详细讨论。这里就不再展开。

另外，测试往往要用到很多测试数据，这就牵涉到样本数据的导入。我们也会在具体编程时加以讨论。

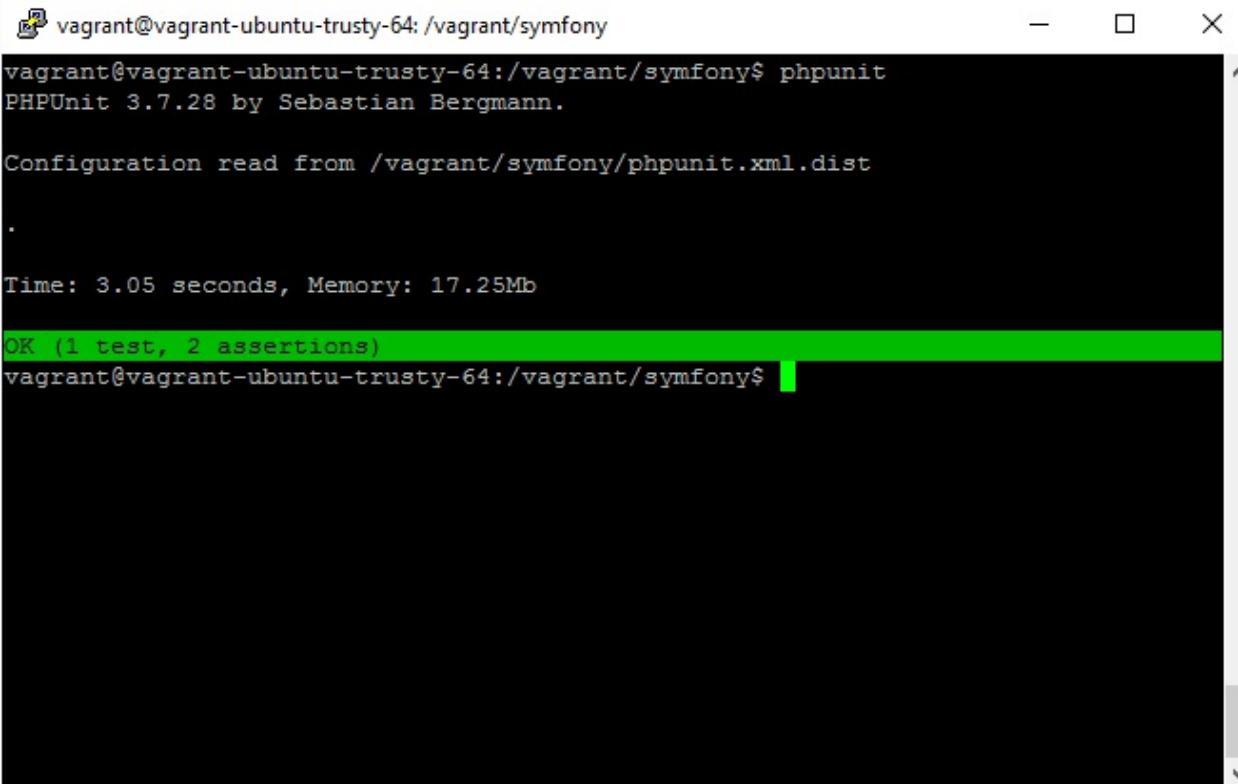
要在SF3的应用中使用PHPUnit，只要在根目录下运行：

```
phpunit
```

就可以了。

注意：我们假定PHPUnit的安装方式是全局的，同时 `phpunit.xml` 文件保存在项目根目录下（也就是和 `composer.json` 同一个目录）。

如果一切正常，那么会有一个类似如下的提示：



vagrant@vagrant-ubuntu-trusty-64: /vagrant/symfony

```
vagrant@vagrant-ubuntu-trusty-64:/vagrant/symfony$ phpunit
PHPUnit 3.7.28 by Sebastian Bergmann.

Configuration read from /vagrant/symfony/phpunit.xml.dist

.

Time: 3.05 seconds, Memory: 17.25Mb

OK (1 test, 2 assertions)
vagrant@vagrant-ubuntu-trusty-64:/vagrant/symfony$
```

04 藏书管理程序的结构

我们要开发的是一个主要供个人使用的藏书管理程序，该程序的实际运行版本见“[任氏有无轩](#)”主页。

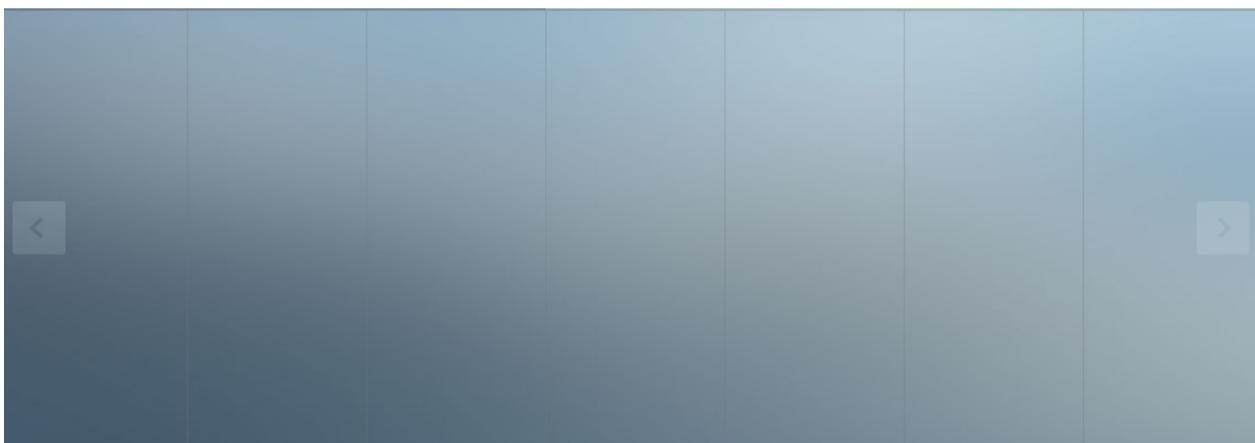
这个应用主要提供了这样一些功能：

- 书籍列表：显示藏书的一个列表，提供关键字搜索、页数跳转、分页显示等功能；
- 书籍详细：显示某一本书籍的详细信息，并从豆瓣处抓取信息作为补充显示，提供添加个人TAG的功能（以便访客日后搜索）；
- 读书心得：显示读书后写的心得（以博客文章形式呈现，牵涉到博客的整合，见以后的章节）；
- 博客：使用的是[WordPress](#)作为博客平台，并对[WordPress](#)数据库进行查询并获得信息。
- 维客：使用的是[DokuWiki](#)作为维客平台，只是整合，不做进一步编程。
- 资源：比如我喜欢的湖人队的赛程。
- 联系：列出一些和站点主人的联系方法，包括Google Map的调用等。
- 首页：提供站点的接口。
- 后台：提供一个后台入口，以比较直观的方式来管理数据。

在首页中，对书籍信息需要进行汇总并显示，有一个Dart的部件将显示“每日引言”和“今日天气”信息，列出最近的博客文章等。这是所有页面中编程量最大的页面（没有之一），而且又是站点的入口，因此将首先加以深入研究。我们先给出一张我的站点的首页截屏，作为效果展示：



首页 藏书 读书 博客 维客 联络



藏书

据不完全统计，截至2015年01月22日，任氏有无轩藏书1,803本，共计470,265字，704,060页。最近（2014年01月14日）收藏/整理的书籍是宝树的《时间之墟》。

读节

截至2015年01月22日，任氏有无轩主人撰写了79篇评论。最近（2015年01月15日）评论的书籍是《时间之墟》，题为“刹那成永劫”。

博客

本博客自2003年开始设立。写得少不是因为我不思考。我思考的越多，写下来的就越少。最近的文章发布于2015年01月15日，题为《刹那成永劫》。

杂客

这是我整理的一些资源，主要是电子书（911调查委员会报告，上帝创造了整数等）和我最喜爱的湖人队的赛程。还可以和我取得联系。

最近的博客

2015/1/1：以悲剧开始

发布于2015年01月01日

2015年的第一天就是以悲剧开始的。上海外滩发生踩踏事件，35人死亡，43人受伤。（新闻链接：<http://news.sina.com.cn/c/2015-01-01/043131350553.shtml>）这真的是一个人间惨剧。“起...

[浏览全文](#)

杂谈：二：Gmail的被封掉

发布于2014年12月29日

也许是我之前一篇杂谈文章的“回应”，这几天在广大互联网用户中说的最多的一件事情，是Gmail的被封掉。这次的封闭Gmail，据我了解，封闭了IMAP/POP/SMTP所有的端口，采用了路由封锁。所以，要想使用Gmail，只有一个方法：...

[浏览全文](#)

杂谈：一

发布于2014年12月27日

有这么一个问题，一直让我百撕不得骑姐百思不得其解：一个极度封闭的社会，有可能推广、发展、使用一个极度开放的技术吗？我觉得是不可能的。我的理由如下：首先，一个封闭的社会必然是一个集权垄断的社会。社会利益的分配优先考虑的是特权...

[浏览全文](#)

版权说明

任氏有无轩，V5.0，1989 - 2015，基于Apache 2 + MySQL 5 + PHP 5.5开发
其它技术：Symfony 2.4, Bootstrap, Dart

社交网络



05 创建应用

我们已经进行了大量的准备工作和前导阅读，目的当然就是为了我们接下来要进行的应用开发。

创建应用需要如下几个步骤。

规划项目位置

我们之前看到了关于[Vagrant虚拟机的安装和如何进入虚拟机](#)。

让我们先登录到虚拟机，并转到 `/vagrant` 目录。我们之前讲过，这个目录实际上就是我们 Windows宿主机中的某个目录。

这个目录将成为我们所有项目的存放位置。在该目录下，各个项目有各自的项目目录。我们将这个项目目录称为某个项目的根目录。以后的教程中，如果再次提到这个“根目录”，那就一定是指这个目录。

获得Symfony的 `installer`

```
sudo curl -LsS http://symfony.com/installer -o /usr/local/bin/symfony  
sudo chmod a+x /usr/local/bin/symfony
```

注意：这么做是在 `/usr/local/bin` 下创建了一个可执行的 `symfony` 命令。我比较喜欢将这个命令放置在 `/vagrant` 目录下。

创建应用

假定我们在 `/vagrant` 目录，现在用如下的命令创建我们的应用：

```
$ symfony new the_new_project_name
```

请根据实际情况将 `the_new_project_name` 替换为更有意义的名字，在我们这本书中，我们用的项目名称 `symfony`。

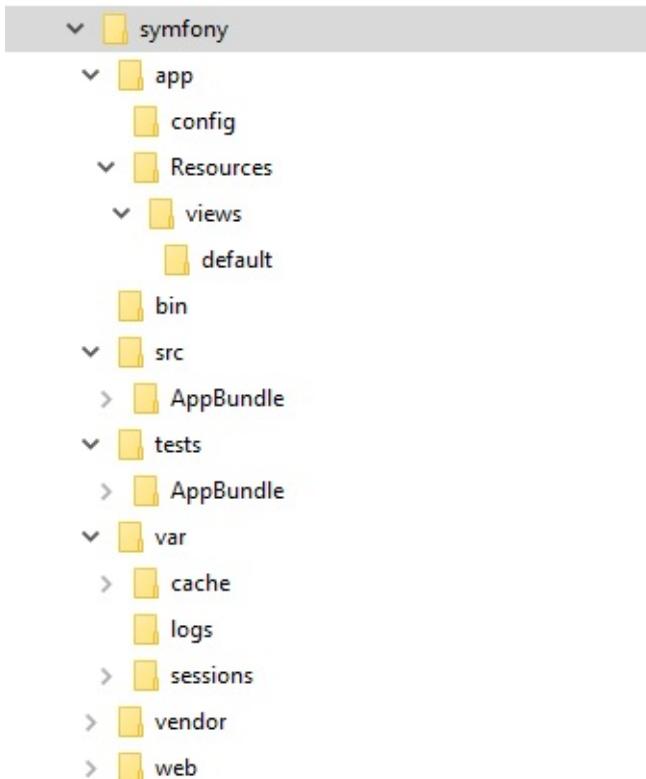
注意：在我们虚拟机中，由于 `/vagrant` 这个目录并不是实实在在的Linux系统中的目录，而是一个Windows系统下映射过去的目录，上面的命令有可能会执行出错（取决于你使用的Vagrant版本）。此时，我们可以在对应的Windows目录中进行如上操作。这样生成的应用框架是可以在Linux中使用的。或者，可以参考SF官方文档中的[方法](#)，用Composer来创建项目。

这个命令将在 `vagrant` 目录下创建一个 `symfony` 目录（也就是我们的项目名字），并在该目录下进行必要的设置，创建一个全新的Symfony应用框架。

现在我们进入 `symfony` 目录，也就是“根目录”中，按照之前“[Composer](#)”一章中的介绍下载 `composer.phar`，并执行一次更新。

目录结构说明

一个空空如也的Symfony 3框架约莫有32M，这也是SF3被称为重量级框架的原因。它的目录结构如下：



app 目录

这个目录是整个框架的运行核心。一些重要的核心文件，如 `autoload.php`，`AppKernel` 等文件都在该目录中。

它又包括几个子目录，也非常重要。

- `config`：这里存放应用所有的配置。在日后的讨论中，我们会慢慢接触这些文件。

- `Resources`：这里可以存放应用级别的资源，如模板文件（在 `views` 子目录下）。

bin 目录

- `console`：该文件是SF命令行界面，我们稍后在开发过程中会经常用到这个命令。

src 目录

用户编写的所有内容都在该目录下，严格的说，是在 `AppBundle` 目录下。根据代码的用途，`AppBundle` 目录下又可以分为：

- `Controller`：控制器，即MVC中的C。
- `Entity`：实体，即MVC中的M。
- `Repository`：仓库，存放实体操作的代码。
- `Resources\config`：存放当前应用包的配置，如路由，数据库实体等。
- `Resources\views`：存放模板，即MVC中的V。
- `Tests`：存放单元测试和功能测试代码。

在项目刚创建完成时，这些目录（除了 `Controller`）都不存在。我们在日后开发过程中，可以选择生成。

tests 目录

此处存放所有的测试文件，包括单元测试和功能测试。

var 目录

该目录中有三个子目录。

- `cache`：存放SF编译用户代码和系统代码后的缓存。根据实际使用情况，又可能会有 `prod`，`dev` 和 `test` 子目录，分别对应生产、开发、测试环境。
- `logs`：存放日志文件，如 `dev.log` 对应的是开发环境下的日志文件。
- `sessions`：存放PHP和SF运行时创建的对话信息。

vendor 目录

所有第三方的包和代码存放在此处。一般情况下我们在此处进行操作。

web 目录

这个目录是SF3应用开放给Web服务器的入口，也就是我们常规情况下访问 `http://www.somewhere.com` 时，Web服务器所访问的根目录。请不要和我们之前说的“项目根目录”混淆。

在这个目录中，有SF3应用的入口文件：`app.php`（生产模式）和`app_dev.php`（开发模式）。在实际应用中，我们访问的是`app.php`——当然，因为有重写规则的存在和该目录下`.htaccess`文件的配合，我们访问一个SF应用时，不需要指明`app.php`，而可以直接用类似`http://www.somewhere.com/path/to/resource`这样的方式。在开发时，我们更多的是使用`app_dev.php`，此时我们访问的URI形如：`http://www.somewhere.com/app_dev.php/path/to/resource`。

远端调试

如果我们现在在Windows宿主机中访问`http://symfony/app_dev.php`，那么我们会看到一个错误信息：

You are not allowed to access this file. Check `app_dev.php` for more information.

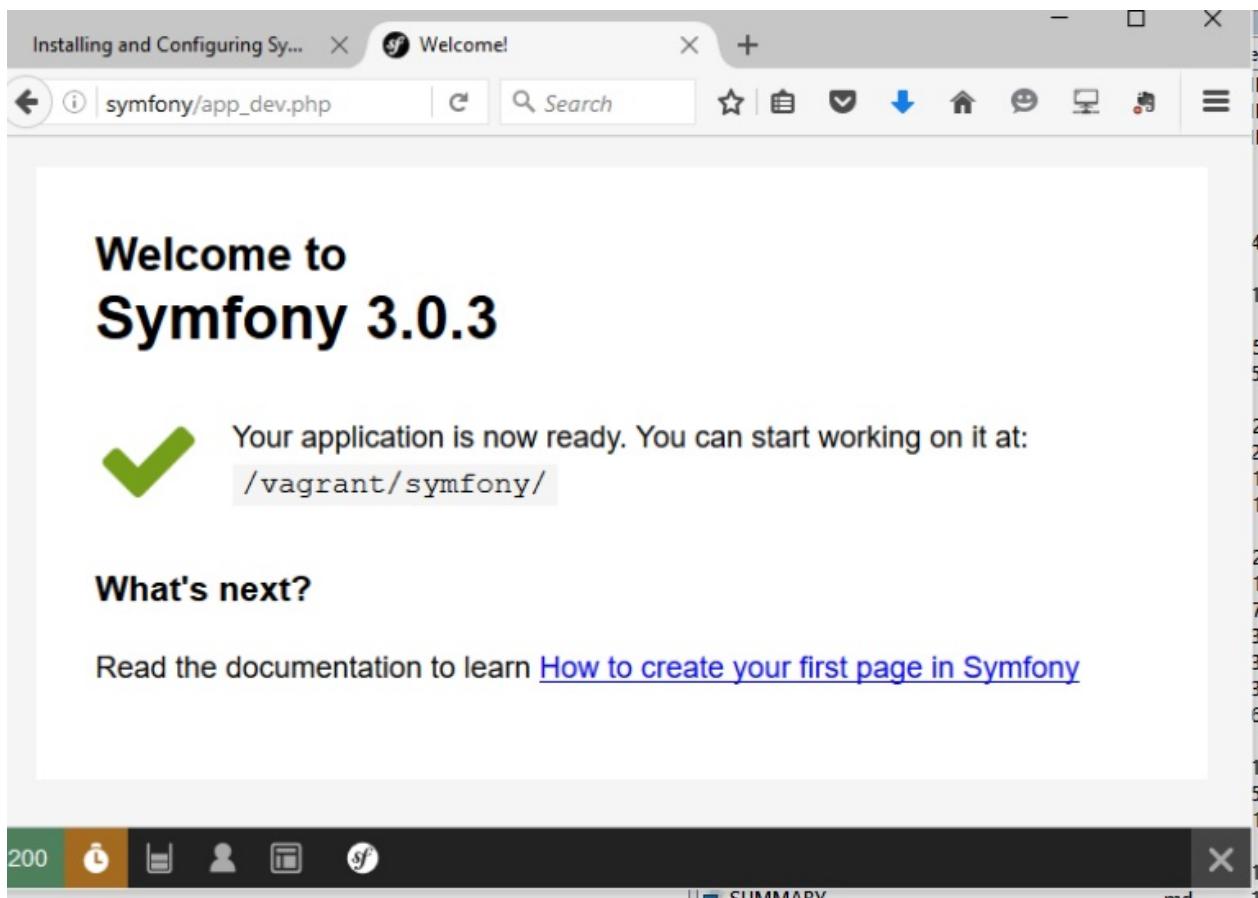
这是因为`app_dev.php`是个用在开发模式下的文件，缺省时开发环境是不对远程主机开放的。

既然我们采用目前的Windows+Vagrant的开发方式，我们显然必须进行远程开发，所以需要修改一下`app.php`文件：

原来的文件中找到这一段：

```
// This check prevents access to debug front controllers that are deployed by accident to
// Feel free to remove this, extend it, or make something more sophisticated.
if (isset($_SERVER['HTTP_CLIENT_IP']))
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !(in_array(@$_SERVER['REMOTE_ADDR'], array('127.0.0.1', 'fe80::1', '::1')) || php_
) {
    header('HTTP/1.0 403 Forbidden');
    exit('You are not allowed to access this file. Check '.basename(__FILE__).' for more
}'
```

将整个`if`段全部注释掉。然后再次访问，我们会看到如下的欢迎界面：



看到这个界面，那么我们的SF应用创建就大功告成。接下来就是真正的开发过程了。

注意：经过修改的这个 `app_dev.php` 程序可千万不要放到生产环境中，这会带来巨大的安全风险。因此，我一般会将这个文件放入版本控制中的被忽略文件列表中（见下一节[建立版本控制](#)）。

不过在进入开发之前，我们先要进行代码仓库的管理。

建立版本管理

为了便于管理代码，我们最好将我们的应用置于版本管理之下。

我们可以选择[GitHub](#)或者[BitBucket](#)或者别的什么代码管理仓库，哪怕是自己搭建的都可以。

将代码置于某个仓库下进行版本管理不是很复杂。但是针对SF的话，我们需要生成自己的`.gitignore`或者`.hgignore`文件，从而避免将一大堆第三方代码和不必要的文件置于版本控制之下。

经过我的实践，我建议用如下的`.gitignore`文件——如果你使用[Hg](#)，可以加以参考。该文件放置在项目根目录下。

```
/web/bundles/
/app/bootstrap.php.cache
/app/cache/*
/app/config/parameters.yml
/app/logs/*
!app/cache/.gitkeep
!app/logs/.gitkeep
/app/phpunit.xml
/build/
/vendor/
/bin/
/composer.phar
/nbproject/private/
*.php~
/web/app_dev.php
```

根据你的实际情况，还可以加入更多的忽视清单。

现在你可以`commit`，`push`到远程代码仓库去了！

如果你对命令行的操作感到厌烦，可以考虑使用[SourceTree](#)这样的GUI界面。

建立数据库

数据库是任何现代应用的核心。我们要开发的藏书管理程序也不例外。

虽然当今编程有向着TDD，DDD导向的趋势，但是在我们这个程序中，我们还是遵循最传统的从数据模型出发的流程。

如果我们去浏览Symfony的官方文档，会发现SF3使用的是从实体（Entity）到数据库（Database）的流程。但是我们这个教程遵照的是一个完全不同的方向：所有数据的来源都是通过RESTful API提供的。换句话说，所有牵涉到数据库的操作都在另外一个应用中实现。关于这个RESTful API的实现，请参见我的另外一个教程《用Silex开发一个RESTful API》中的讲解。

尽管如此，数据库还是整个应用的核心——即使在我们这个应用中不直接对其进行操作。

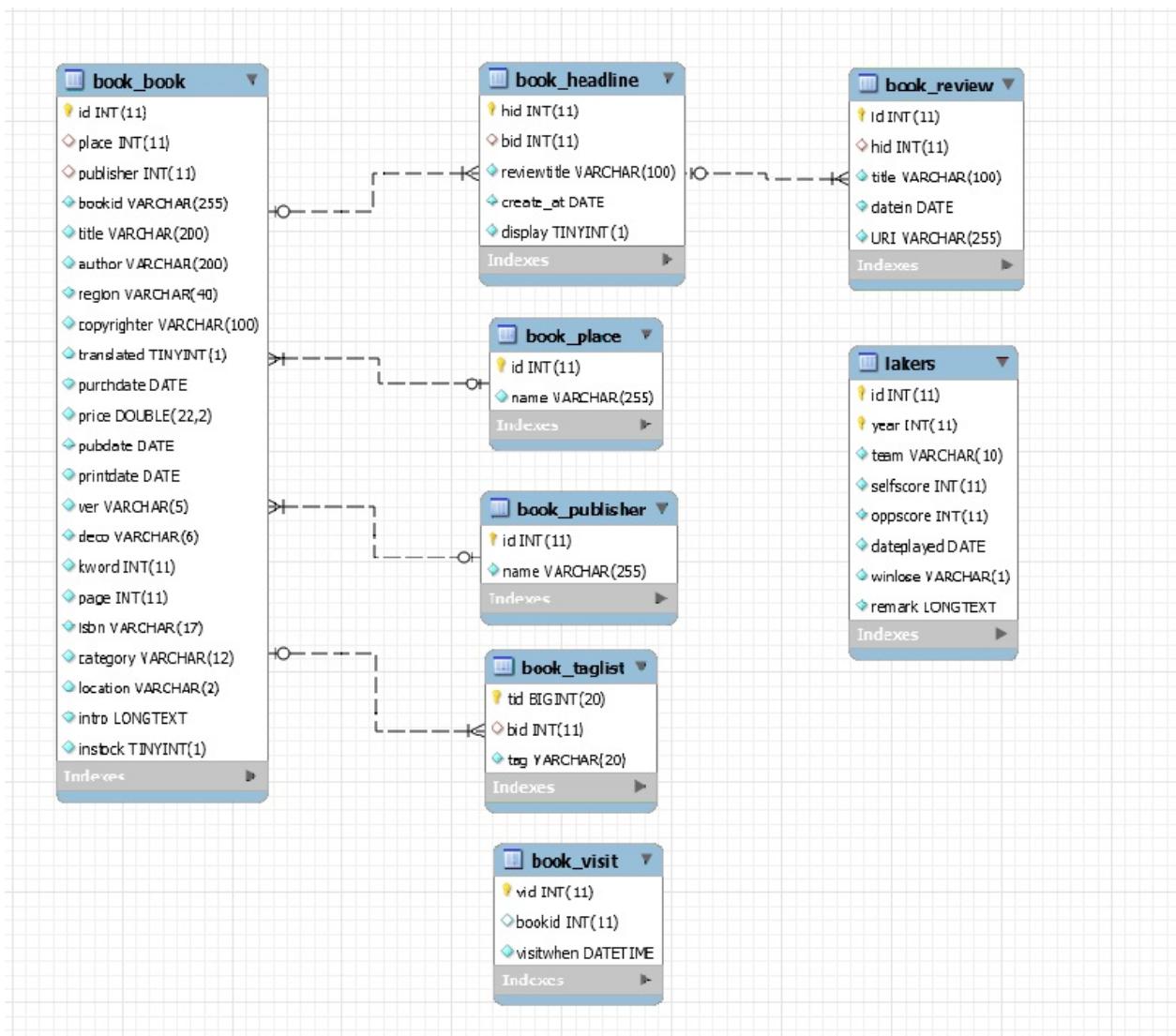
笔者已经将本应用使用到的数据库的结构SQL语句上传到了本书对应的Github仓库，请下载该文件，并在你的开发环境中创建该数据库。

注意：在本文中的写作中，笔者使用的开发机已经有了一个名为rsywx的数据库——这是我生产环境使用的数据库的一个本地备份。所以，用于在笔者的开发环境中真正使用的数据库会是rsywxTutorial。不过这不影响本教程的正常使用。读者可以使用rsywx也可以用自己喜欢的名字来命名这个数据库。

数据库结构

rsywx数据库包括了若干表格。从功能来看，有收录书籍信息（以及书籍出版社、购买地点、Tag）的表格，收录书籍评论的表格以及书籍访问记录的表格，还有一个记录我最喜欢的NBA球队湖人队赛程的表格。

该数据库结构以及表之间的相互关系如下图所示：



我不去一一解释各个字段、各个表之间的关联，只是简单地说几句。

这是一个符合3NF的数据库。以 `book_book` 为核心，其他表格（除 `lakers`）之外，都直接或间接地和该表有关联。

`book_visit` 用来记录书籍详情页面被访问的情况。目前我只是简单地记录了书籍、访问时间。这些数据会在后台管理中用作各类统计。

这肯定不是一个完美的数据库设计。读者可以根据自己的需求加以改进和修订。

数据库用户

一般来说，用 `root` 来操作数据库总不是一个很好的做法。在开发过程中也许可以出于简单的考虑，我们可以先用 `root`，但是在生产环境这样做是不推荐的——除非你对 `root` 的密码有充分信心。

我们可以借助相应的工具来创建一个新的用户，只给他相应的CRUD权限或者其它必要的权限。有关数据库用户创建和权限分配的操作，可以参见相应的文档。

在本文中，我们使用的用户和密码是：`tr/trtrtr`。这不是一个很好的用户/密码组合，不过只是作为开发和演示而已。

修改数据库配置

SF应用中数据库配置保存在`app\config\parameters.yml`中。我们在安装SF、创建项目时可以制定数据库的连接，也可以在稍后手工修改。

```
# This file is auto-generated during the composer install
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: rsywx_tutorial
    database_user: tr
    database_password: trtrtr
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    secret: ThisTokenIsNotSoSecretChangeIt
```

我这里已经将使用到的数据库、用户、密码进行了更新。

这个文件也可以被当成“配置”文件，存放一些供整个应用使用的“全局”变量。我们会在后面的章节看到更详细的介绍。

应用结构

我们之前已经讲过，SF是一个非常严格的MVC框架。所以，我们的应用也严格遵循MVC分离的原则。

但是，由于本应用已经开发到了6.0版本，笔者对应用结构也有了全新的布局，所以在该版本的应用中，M模块其实已经不再存在，而改用RESTful API调用的方式。因此本应用的结构也相对比较扁平。

简单来说，我们创建了一系列的 controller，其中的 action 与 routing 关联，负责接收来自入口文件（app.php）的调派。

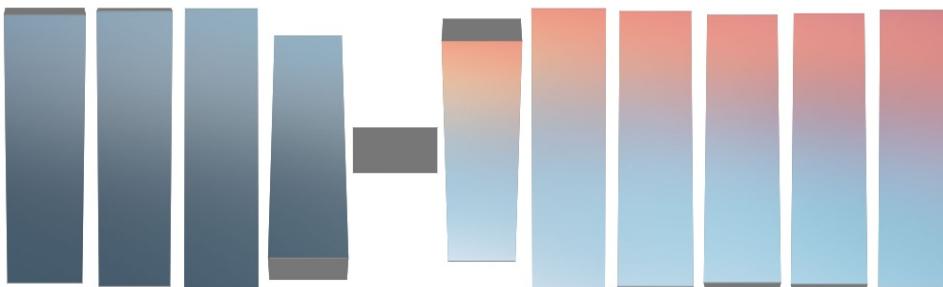
在某个具体的 action 中，一般的流程是：

- 获取传入的参数；
- 构建要调用的API URI；
- 获得返回数据并解析¹；
- 将构造好的数据传递给一个模板并显示；

根据应用要提供的功能，我们可以创建相应的 controller。我们会在稍后的章节中说明各个 controller 的创建。

这个应用有前台，也有后台。

前台是各个公共页面，如首页、书籍列表、书籍详情、书评列表、其它页面等。下图是首页的效果²。



藏书

截至2016年03月19日，任氏有无轩藏书1,843本。约478,745千字，718,896页。
最近（2016年02月22日）收藏/整理的书籍是《科学新闻》杂志社的《人类与社会》。

读书

截至2016年03月19日，任氏有无轩主人撰写了91篇评论。
最近（2016年01月31日）评论的书籍是《The Death of Caesar》，题为“对独裁的抗争”。

博客

本博客自2003年开始设立。写得少不是因为我不思考。我思考的越多，写下来的就越少。
最近的文章发布于2016年03月12日21时18分，题为“字母狗击败李世石”。

维客

这里是找整理的一些资源，主要是电子书和我最喜爱的湖人队的赛季。
还可以和我取得联系。

历史上的今天

历史上的03月19日，任氏有无轩主人购买和/或登录了2本书。它们是：

北京兰德（1995） 完美应用Ubuntu（2009）

最近的博客文章



老彼得初中的最后一个学期

发表日期：2016年02月22日

今天老彼得学校开学了。初三下学期，初中的最后一个学期了。昨天晚上和他聊天——这也是我们每个学期开始前的谈话，这个习惯也保持了很多年了，我记得是从他小学一年级开始的。谈话的内容是问他高中的打算：国内高中中国内大学；国内高中国外大学；国际高...

[浏览全文 »](#)

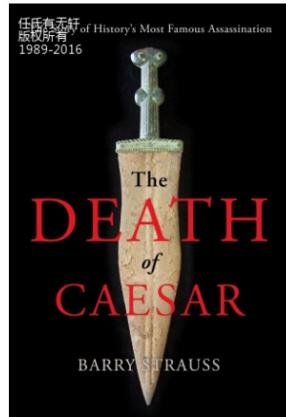


微信订阅号的开发

发表日期：2016年02月01日

最近看了一下微信订阅号的开发。我要做一个个人订阅号，用户订阅之后，可以通过输入命令获得交互，得到英文单词的解释，大概就是这样。微信开发权限获得就不讲了，需要帮助的同学可以去访问相应的站点。首先，我开了一个域名 (<http://...>)

[浏览全文 »](#)



对独裁的抗争

发表日期：2016年01月31日

《The Death of Caesar》是一本讲述历史上最著名的一次刺杀的书。————— 我们所学的历史基本只讲谁是凯撒，谁又在何时何地刺杀了他，然后安东尼和屋大维为凯撒复仇，最

任氏有无轩
版权所有
1989-2016

任氏有无轩
版权所有
1989-2016

任氏有无轩
版权所有
1989-2016

澳大利亚寻宝记

保险概论

各国概况 (上)

版权说明

任氏有无轩，V6.0，1989 - 2016，基于Apache 2 + MySQL 5 + PHP 5.5开发
其它技术：Symfony 3.0, Bootstrap



[CC BY-NC] 除非另有显式声明，本作品均采用知识共享署名-禁止演绎 3.0 Unported 许可协议进行许可。

推荐使用Chrome, Firefox等现代浏览器浏览，分辨率1680*1050以上。

社交网络



安全站点

本站点为安全站点，采用HTTPS证书保护。



后台需要登陆，显示相关的统计数据，如下图所示。

rsywx.net (104.224.160.168) 主机位于US, California。

截至2016年03月19日08时21分57秒，本站点藏书已经被浏览503,567次。

最近访问的20本书籍

写给孩子的哲学启蒙书（一）	2016-03-19 08:15:46 (已经访问了: 420次)
七种武器（一）	2016-03-19 08:12:46 (已经访问了: 207次)
哥德尔、艾舍尔、巴赫——集异壁之大成	2016-03-19 08:12:39 (已经访问了: 1201次)
项狄传	2016-03-19 08:10:56 (已经访问了: 518次)
婴幼儿饮食指南	2016-03-19 08:05:07 (已经访问了: 232次)
Principles of Marketing	2016-03-19 07:56:36 (已经访问了: 235次)
崇祯长编	2016-03-19 07:53:48 (已经访问了: 253次)
中国历代政治得失	2016-03-19 07:52:17 (已经访问了: 1160次)
增订晚明史籍考	2016-03-19 07:46:47 (已经访问了: 416次)
纽约客	2016-03-19 07:43:00 (已经访问了: 311次)
生命中不能承受的轻	2016-03-19 07:40:23 (已经访问了: 835次)
项狄传	2016-03-19 08:10:56 (已经访问了: 518次)
婴幼儿饮食指南	2016-03-19 08:05:07 (已经访问了: 232次)
Principles of Marketing	2016-03-19 07:56:36 (已经访问了: 235次)
崇祯长编	2016-03-19 07:53:48 (已经访问了: 253次)
中国历代政治得失	2016-03-19 07:52:17 (已经访问了: 1160次)
增订晚明史籍考	2016-03-19 07:46:47 (已经访问了: 416次)
纽约客	2016-03-19 07:43:00 (已经访问了: 311次)
生命中不能承受的轻	2016-03-19 07:40:23 (已经访问了: 835次)
理想国	2016-03-19 07:34:57 (已经访问了: 424次)
英语高级听力（教师用书）	2016-03-19 07:29:17 (已经访问了: 224次)
上学就看（植物园）	2016-03-19 07:27:16 (已经访问了: 277次)
经济基础理论与相关知识	2016-03-19 07:20:24 (已经访问了: 209次)
希腊的神话和传说（上）	2016-03-19 07:07:34 (已经访问了: 553次)
海上花列传	2016-03-19 07:06:59 (已经访问了: 155次)
Chicken Soup for the Soul 3	2016-03-19 07:05:35 (已经访问了: 185次)
勇敢的人——哈默传	2016-03-19 06:32:30 (已经访问了: 1166次)
史纲评要（上）	2016-03-19 06:32:28 (已经访问了: 395次)

访问最多的20本书籍

访问最少的20本书籍

最近14天访问量

最近7天都没有被访问的20本书籍

版权说明

任氏无纤，V6.0，1989 - 2016，基于Apache 2 + MySQL 5 + PHP 5.5开发
其它技术：Symfony 3.0, Bootstrap



(cc) BY-NC 除非另有显式声明，本作品均采用知识共享署名-禁止演绎 3.0 Unported 许可协议进行许可。

推荐使用Chrome, FireFox等现代浏览器浏览，分辨率1680*1050以上。

社交网络



安全站点

本站点为安全站点，采用HTTPS证书保护。



1. RESTful API接口返回的都是JSON格式的数据，所以必须将其转换到一个对象或者数组以便PHP进一步使用。 ↵
2. 这是我运行中的站点的首页，比本教程要创建的首页更复杂。 ↵

建立数据库实体

ORM的本质在于将一个数据库（更确切的说是其中的表格）“转换”到一个PHP对象。于是我们不用类似“`select * from ...`”或者“`insert into ...`”这样的SQL语句来操作表格中的数据，而是改用更直观、也更容易出错的方式。比如下面这段代码的最终运行效果是在`rsywx`数据库的`book_place`中插入了一个新的纪录。

```
$place = new BookPlace();
$place->setName('Common');

$manager->persist($place);

$manager->flush();
```

这样的过程也许比`mysqli_query($connection, 'insert into ...')`多了几行代码的输入，但是出错机会少，而且也更安全。

导入MySQL数据库

我们的应用开发里程中，数据库已经建立完成（见[建立数据库一节](#)）。所以，我们需要将数据库转换到ORM中可以操作的类。

在严格的MVC框架下，这样形成的类是M(odel)层。但由于在本应用中，我们将提供数据这一任务全部放置到API中完成，所以我们导入MySQL数据库形成M层并不是必须的。我们在本教程中还是这么做是为了后面一节[样本数据](#)的需要。

进入虚拟机中项目的根目录（`/vagrant/symfony`），输入如下命令：

```
php bin/console doctrine:mapping:import AppBundle yml
```

其中：

- `bin/console` 是SF的命令行接口，对SF框架应用的操作都要通过这个接口。
- `doctrine:mapping:import` 表明我们要导入一个数据库。
- `AppBundle` 表明导入的数据库要为`AppBundle`这个包所用。
- `yml` 表明我们要用[YAML](#)格式保存导出的数据库信息。

命令顺利执行后，在`symfony\src\AppBundle\Resources\config\doctrine\`目录下会多出几个文件，这几个文件一一与数据库中的表格对应。比如`BookPlace.orm.yml`对应的就 是`book_place`数据库。而它的内容也是如此：

```

AppBundle\Entity\BookPlace:
    type: entity
    table: book_place
    id:
        id:
            type: integer
            nullable: false
            options:
                unsigned: false
            id: true
            generator:
                strategy: IDENTITY
    fields:
        name:
            type: string
            nullable: false
            length: 255
            options:
                fixed: false
    lifecycleCallbacks: { }

```

如果我们回忆一下 `book_place` 的结构，会看到这个yml文件对表格的描述是与该表格的定义完全一致的。

我们暂时不会深入讨论各个字段定义中各个选项的意义。而且在一般情况下，我更喜欢从数据库到类的映射方式。

生成实体类

导入了数据库后，我们执行如下命令来生成实体类：

```
php bin/console doctrine:generate:entities AppBundle
```

该命令会在 `AppBundle` 目录下生成一个新目录 `Entity`，其中会有若干个PHP文件。这些文件一一与上一步生成的YML文件对应，也因此一一与数据库中的表格对应。比如 `BookPlace.php` 文件对应的是 `BookPlace.orm.yml` 文件，并进而对应的是 `book_place` 这个表格。它的内容如下：

```

<?php

namespace AppBundle\Entity;

/**
 * BookPlace
 */
class BookPlace
{

```

```

/**
 * @var integer
 */
private $id;

/**
 * @var string
 */
private $name;

/**
 * Get id
 *
 * @return integer
 */
public function getId()
{
    return $this->id;
}

/**
 * Set name
 *
 * @param string $name
 *
 * @return BookPlace
 */
public function setName($name)
{
    $this->name = $name;

    return $this;
}

/**
 * Get name
 *
 * @return string
 */
public function getName()
{
    return $this->name;
}

```

这个文件中包含一个命名空间（`AppBundle\Entity`）的声明和一个类（`BookPlace`）的声明。而在类的声明中，包括了两部分：

- 第一部分是成员声明。在 `BookPlace` 类中，只有两个成员：一个是 `$id`，一个是 `$name`。它们分别于 `book_place` 表格中的两个字段 `id` 和 `name` 对应。

- 第二部分是方法声明。一般情况下，对于每个成员，都有两个方法，一个是setter，一个是getter。对于只读字段或者应该由数据库引擎自动生成的字段（如本类中表明记录唯一的 id）就只有一个getter。

从数据库（表格）到ORM映射，再到PHP类声明，我们完成了将数据库加以对象化的步骤。

注意：上面产生的PHP文件是自动生成的。我们对这个文件和其中的类声明不应该做任何的改动。

我们只能修改YML文件然后用 doctrine:generate:entities 来生成PHP文件，用 doctrine:schema:update 命令更新数据库；或者直接操作数据库，并在此通过上面讲到的两个步骤来更新ORM和Entity。

ORM表述和Entity类中对表间关系的描述

在结束本小节之前，我们有必要看看ORM表述和Entity类中是怎样描述表之间的关系的。

我们的 book_book 表格与若干表格有“1对多”的关系。比如一本书的出版社和 book_publisher ，它的购买地点和 book_place 都有1对多的关系。

在 BookBook.orm.yml 中，我们可以找到这样一段：

```
AppBundle\Entity\BookBook:
...
manyToOne:
    place:
        targetEntity: BookPlace
        cascade: { }
        fetch: LAZY
        mappedBy: null
        inverseBy: null
        joinColumns:
            place:
                referencedColumnName: id
                orphanRemoval: false
    publisher:
        targetEntity: BookPublisher
        cascade: { }
        fetch: LAZY
        mappedBy: null
        inverseBy: null
        joinColumns:
            publisher:
                referencedColumnName: id
                orphanRemoval: false
    lifecycleCallbacks: { }
```

在这里我们可以清楚看到， BookBook 是多端，它与两个1端对应。

而在 BookBook.php 中，我们可以找到这样的代码：

```
<?php

namespace AppBundle\Entity;

/**
 * BookBook
 */
class BookBook
{
    ...
    /**
     * @var \AppBundle\Entity\BookPlace
     */
    private $place;

    /**
     * @var \AppBundle\Entity\BookPublisher
     */
    private $publisher;

    ...
    /**
     * Set place
     *
     * @param \AppBundle\Entity\BookPlace $place
     *
     * @return BookBook
     */
    public function setPlace(\AppBundle\Entity\BookPlace $place = null)
    {
        $this->place = $place;

        return $this;
    }

    /**
     * Get place
     *
     * @return \AppBundle\Entity\BookPlace
     */
    public function getPlace()
    {
        return $this->place;
    }

    /**
     * Set publisher
     *
     * @param \AppBundle\Entity\BookPublisher $publisher
     *
     * @return BookBook
     */
    public function setPublisher(\AppBundle\Entity\BookPublisher $publisher)
    {
        $this->publisher = $publisher;

        return $this;
    }

    /**
     * Get publisher
     *
     * @return \AppBundle\Entity\BookPublisher
     */
    public function getPublisher()
    {
        return $this->publisher;
    }
}
```

```
/*
 * @param \AppBundle\Entity\BookPublisher $publisher
 *
 * @return BookBook
 */
public function setPublisher(\AppBundle\Entity\BookPublisher $publisher = null)
{
    $this->publisher = $publisher;

    return $this;
}

/**
 * Get publisher
 *
 * @return \AppBundle\Entity\BookPublisher
 */
public function getPublisher()
{
    return $this->publisher;
}
```

针对 `place` 和 `publisher` 这两个字段，在 `book_book` 中存放的只是一个ID（一个整数），但是在根据ORM生成的PHP类中，它们以各自PHP类出现

（`\AppBundle\Entity\BookPlace` 和 `\AppBundle\Entity\BookPublisher`）。对应的，它们的 `setter` 和 `getter` 也是对这两个类的操作，而不是对 `id` 这个字段本身的操作。

在本教程中，我们不再对此进行进一步的展开。

样本数据

应用开发，其实有很大一部分是在样本数据的创建上。我们为什么要用样本数据？

首先，有了样本数据，我们的页面显示不会那么“空白”，有了比较实在的内容。

其次，样本数据一般都是非常有规则的，因此便于我们进行单元测试和功能测试。

第三，有一些应用上的逻辑问题，需要有足够的样本数据才会体现。比如我们在测试分页数据的时候。

如果我们只能人工或者半自动地往数据库中添加数据，那么效率就太低了。SF考虑到了这个问题，因此提供了一个专门的部件帮助我们来完成这个工作。这个部件就是 `DoctrineFixturesBundle`。

安装 `DoctrineFixturesBundle`

按照SF官方文档的[说明](#)，该部件的分装有两步。

第一步，下载部件包。而这是通过运行如下命令完成的：

```
composer require --dev doctrine/doctrine-fixtures-bundle
```

下载安装完毕后，项目根目录下 `composer.json` 文件中会增加一行：

```
"require-dev": {  
    ...  
    "doctrine/doctrine-fixtures-bundle": "^2.3" //这是增加的一行  
},
```

第二步，注册并激活该部件包。

找到 `app/AppKernel.php` 文件，并作如下修改：

```
// app/AppKernel.php
// ...

class AppKernel extends Kernel
{
    public function registerBundles()
    {
        // ...
        if (in_array($this->getEnvironment(), array('dev', 'test'))) {
            $bundles[] = new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle();
        }

        return $bundles
    }

    // ...
}
```

从这段代码我们也可以看出，这个部件包只能使用在开发和测试环境中。这当然是很明显的。

编写样本数据文件

首先我们要确定这些样本数据文件所在的文件夹。一般而言，如果我们用Doctrine的话，样本文件应该存放在 `src/AppBundle/DataFixtures/ORM` 之下。每个需要样本数据填充的表格都要有一个对应的样本文件。

我们先看一个比较简单的样本文件。该文件用来为 `book_publisher` 表格填充样本数据：

```

<?php

namespace AppBundle\DataFixtures\ORM;

use \Doctrine\Common\DataFixtures\AbstractFixture;
use \Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use \Doctrine\Common\Persistence\ObjectManager;
use AppBundle\Entity\BookPublisher as BookPublisher;

class LoadPublisherData extends AbstractFixture implements OrderedFixtureInterface
{
    /**
     *
     * {@inheritDoc}
     */
    public function load(ObjectManager $manager)
    {
        //Create a common publisher
        $pub1=new BookPublisher();
        $pub1->setName('Common');
        $this->addReference('commonPub', $pub1);

        //Create a special publisher
        $pub2=new BookPublisher();
        $pub2->setName('Special');
        $this->addReference('specialPub', $pub2);

        $manager->persist($pub1);
        $manager->persist($pub2);

        $manager->flush();
    }

    /**
     *
     * {@inheritDoc}
     */
    public function getOrder()
    {
        return 1;
    }
}

```

我们首先约定本文件的命名空间。其次是四个 `use` 语句。这四个语句中，前三个是标准的，也是所有样本文件需要用到的。第四个（`use AppBundle\Entity\BookPublisher as BookPublisher;`）用到的命名空间是本样本文件需要操作的表格所对应的实体类。我们不对该实体类加以引用也是可以的，只是这样做的话，后面的代码中将要用到这个类的FQN（Fully Qualified Name）。

所有样本数据类都派生自 `AbstractFixture` 并实现了 `OrderedFixtureInterface` 接口。从这点我们可以看出，样本数据文件是有顺序的。这是因为，在数据库中，由于存在表格之间的依赖关系和引用一致性检查，有些表格的数据必须在另外一些表格的数据能得以填充之前先得到填充。

比如，`book_book` 表格中的 `publisher` 字段是 `book_publisher` 的外键。基于引用一致性的要求，我们不能为 `book_book.publisher` 赋一个并不存在于 `book_publisher.id` 中的值。所以，我们必须先填充 `book_publisher` 表格才能再填充 `book_book` 表格中的数据。

类的实现中，至少必须有两个函数：一个是 `load()`，一个是 `getOrder()`。

`getOrder()` 用来制定本样本类中的数据需要在第几位被填充。从上面的代码中可以看到，该样本数据是在第一位被填充，也就是我们首先在 `book_publisher` 中填充数据。

`load()` 完成真正的数据填充。在上述代码中，我们创建了两个出版商的信息：一个被我们命名为"Common"，一个被命名为"Special"。

另外，由于我们已经知道这两个出版社一定会被 `book_book` 引用，所以我们用" `addReference()`"方法添加了各自的引用，以便我们在后续的样本文件中引用这两个对象。

最后我们持久化这两个新创建的出版社对象并保存到物理数据库中。

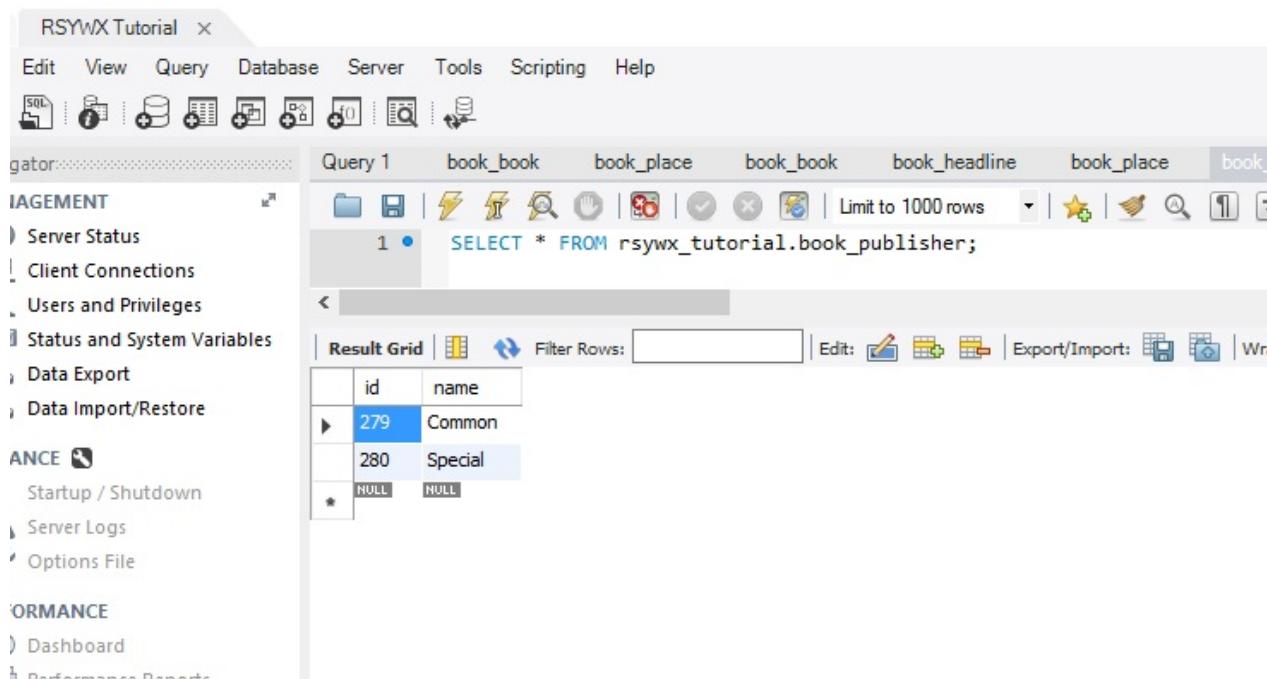
数据填充

有了这个样本文件，我们就已经可以开始数据填充了。我们输入如下的命令：

```
php bin/console doctrine:fixtures:load
```

注意：每一次我们进行样本数据填充的工作，原来在数据库表格中的数据都会被清除。这是为了保证在开发和测试的时候所有原始的数据都是一致的。

数据填充完毕后，我们使用的数据库中就有了对应的数据。即以 `book_publisher` 为例，此时应该有两个数据：



引用其它对象的样本文档

如前所述，`book_book` 这个表格依赖其它表格数据，所以它不能第一个被填充，而且在填充的时候需要引用其它表格中的数据。

我们已经在上面看到，`book_publisher` 表格的填充文件中创建了两个对象引用。类似的创建在 `book_place` 中也一样存在。

一旦我们完成了其它表格的填充和相应依赖对象的创建，我们就可以在 `book_book` 对应的样本文档中加以必要的引用而进一步创建 `book_book` 中的记录。

```
<?php

namespace AppBundle\DataFixtures\ORM;
use \Doctrine\Common\DataFixtures\AbstractFixture;
use \Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use \Doctrine\Common\Persistence\ObjectManager;
use AppBundle\Entity\BookBook as BookBook;

class LoadBookData extends AbstractFixture implements OrderedFixtureInterface
{
    /**
     *
     * {@inheritDoc}
     */
    public function load(ObjectManager $manager)
    {
        //Now we create a 100 general book
        for ($i = 10000; $i <= 10099; $i++)
        {
            $n = new BookBook();
            $n->setTitle("Book " . $i);
            $n->setDescription("Description " . $i);
            $n->setPrice(10 + $i);
            $n->setAuthor("Author " . $i);
            $n->setCategory("Category " . $i);
            $n->setImage("image" . $i);
            $n->setIsbn("ISBN " . $i);
            $n->setPageCount($i);
            $n->setPublishedDate(new \DateTime("2015-01-01"));
            $n->setPublisher("Publisher " . $i);
            $n->setRating(4.5 + $i / 100);
            $n->setStockCount($i);
            $n->setThumbnail("Thumbnail " . $i);
            $n->setTitleCase("Title Case " . $i);
            $n->setYearPublished(2015 + $i / 100);
            $manager->persist($n);
        }
    }
}
```

```

$p->setAuthor('Normal');
$p->setCategory('Normal');
$p->setCopyrighter('');
$p->setDeco('Normal');
$p->setInstock(1);
$p->setTitle('Normal Book Title');
$p->setRegion('Normal');
$p->setTranslated(0);
$p->setPurchdate(new \DateTime());
$p->setPubdate(new \DateTime());
$p->setPrintdate(new \DateTime());
$p->setVer('1.1');
$p->setKword($i);
$p->setPage($i);
$p->setIsbn("$i");
$p->setPrice("$i.99");
$p->setLocation('a1');
$p->setIntro('This is a normal book.');
$p->setPublisher($this->getReference('commonPub'));
$p->setPlace($this->getReference('commonPlace'));
$p->setBookid("$i");
$manager->persist($p);
}

//Create a special book
$s = new BookBook();
$s->setAuthor('Special');
$s->setCategory('Special');
$s->setCopyrighter('');
$s->setDeco('Special');
$s->setInstock(1);
$s->setTitle('Special Book Title');
$s->setRegion('Somewhere On Earth');
$s->setTranslated(0);
$s->setPurchdate(new \DateTime('1970-1-1'));
$s->setPubdate(new \DateTime('1970-1-1'));
$s->setPrintdate(new \DateTime('1970-1-1'));
$s->setVer('1.1');
$s->setKword(999);
$s->setPage(999);
$s->setIsbn('123456789');
$s->setPrice('9999.99');
$s->setLocation('x1');
$s->setIntro('This is a very special book.\nIt is special because it is purchased');
$s->setPublisher($this->getReference('specialPub'));
$s->setPlace($this->getReference('specialPlace'));
$s->setBookid('99999');
$this->addReference('aBook', $s);

$manager->persist($s);
$manager->flush();
}
*/

```

```

/*
 * {@inheritDoc}
 */
public function getOrder()
{
    return 3;
}

```

在该样本文件中，我们一共创建了101本书。其中有100本是常规的，有1本是特殊的。

对于常规的图书，我们设置其相应的日期都是样本数据创建当日：

```

$p->setPurchdate(new \DateTime());
$p->setPubdate(new \DateTime());
$p->setPrintdate(new \DateTime());

```

我们在设置其购买地点和出版商时，用到了之前样本文件中创建的对象引用：

```

$p->setPublisher($this->getReference('commonPub'));
$p->setPlace($this->getReference('commonPlace'));

```

对于那本特殊的书，我们设置其相应的日期是一个很特别的日期，1970年1月1日，也就是计算机元年。其购买地点和出版商的设置也用到了之前样本文件中创建的对象引用：

```

$s->setPublisher($this->getReference('specialPub'));
$s->setPlace($this->getReference('specialPlace'));

```

最后，我们为这本特殊的书创建了一个对象引用。因为这本书的对象在后续的样本文件中还会用到。

小结

样本数据填充部件是功能很强大的一个部件。它能帮助我们系统、高效地创建大量有组织、有规律的数据，方便今后的开发和调试。

样本填充文件采用PHP写成，所以不必麻烦开发人员再去熟悉一种新的方式或者语法。

笔者个人认为，开发初期使用样本数据填充绝对是事半功倍的。

路由

SF以及所有现代PHP框架都采用“单一入口”的方式。

所谓“单一入口”说的是，一个Web应用，不管要访问哪个资源和URI，都统一由一个单一的入口文件进行调派。在SF中，这个文件就是 `web/app.php` （生产环境）或者 `web/app_dev.php` （开发环境）。

在单一入口模式下，用户在浏览器中键入类似“`mysite/book/list`”这样的地址的时候，这样的请求会被入口文件处理，从中分离出不同的部分。在SF中，这样的部分可能包括：控制器（一个类）、动作（类方法）、参数等。

怎样来进行这个分离的动作呢？SF采用的是路由（router）的方法。

在SF中，定义路由有几种方式。比如注释方式（annotation）、YML、XML、PHP等。我个人比较喜欢的是用YML的方式。

定义入口路径

不管我们如何设计WEB应用，总是需要定义一个“入口”。

修改或者创建该文件 `src/AppBundle/Resources/config/routing.yml`，使之包含如下内容：

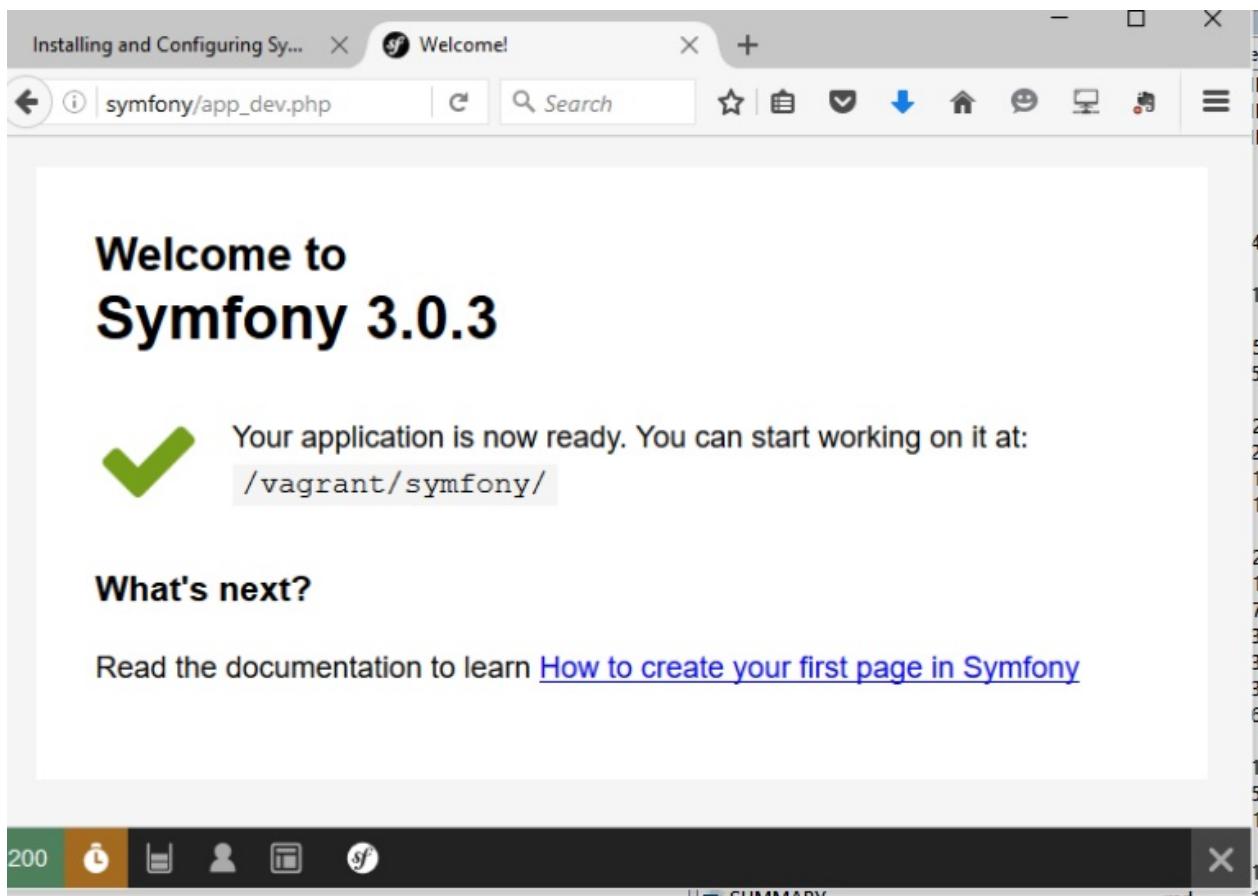
```
home:
  path: /
  defaults: { _controller: AppBundle:Default:index }
```

同时修改 `app/config/routing.yml`，使之只有如下内容：

```
rsywx:
  resource: "@AppBundle/Resources/config/routing.yml"
```

修改 `app/config/routing.yml` 的目的是向SF应用表明，我们的路由配置将来自 `src/AppBundle/Resources/config/routing.yml` 文件。这个文件是一个YML格式的文件，定义了我们应用中所要提供的所有资源的路径配置。

修改完毕后我们再次访问应用，浏览器将会显示我们之前看到的SF欢迎页面：



路径配置

路径配置的核心包括三个部分：

1. 路径名。如 `home` 这样的一个名称。该名称必须在某个路径配置文件中唯一。
2. 路径。如 `path: /`。该路径定义了应用能提供的URI。在本例中，我们定义的是入口，也就是通常所说的“首页”、“主页”。所以它的路径是 `/`。我们在WEB中用 `http(s)://sitename/` 对该资源进行访问。
3. 动作。如 `defaults: { _controller: AppBundle:Default:index }`。该动作表明，该路由将调用控制器的某个动作。该控制器位于 `src/AppBundle/Controller/DefaultController.php` 中，而调用的具体动作是 `indexAction` 方法。

由此，我们得到此类路径动作的一个重要约定。SF在寻找动作的时候，会在指定的 Bundle (本例中的 `AppBundle` 目录，即 `src/AppBundle` 的控制器目录 (即 `src/AppBundle/Controller`) 下寻找一个名为“类名+Controller.php”的文件 (即 `DefaultController.php`)，并在其中寻找一个名为“类名+Controller”的类 (即 `class DefaultController`)，再在其中找到一个“动作名+Action”的公共方法 (即 `public function indexAction`) 并加以调用。

我们略微看一些这个控制器文件：

```
<?php

namespace AppBundle\Controller;

class DefaultController extends Controller
{
    public function indexAction(Request $request)
    {
        // replace this example code with whatever you need
        return $this->render('default/index.html.twig', [
            'base_dir' => realpath($this->getParameter('kernel.root_dir'). '/../'),
        ]);
    }
}
```

我们在以后还会详细解释控制器的编写。这里只是简单地提一句：一般情况下，一个控制器中的动作都会返回一个模板的渲染，于是浏览器就有内容加以显示。

两个重要的命令

在深入讨论更多路由配置之前，我们先看两个SF提供的和路由密切相关的命令。

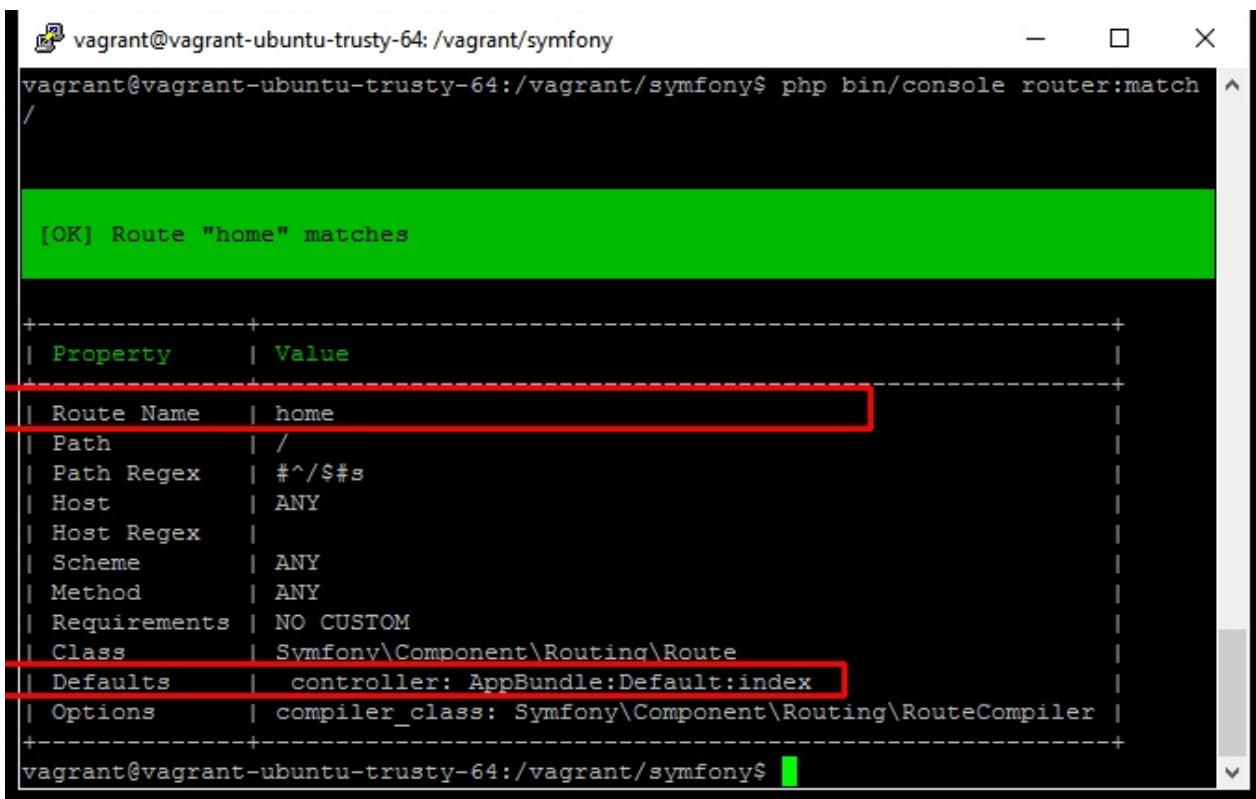
路由匹配

总有一天，我们的路由配置会越来越复杂，于是我们会产生疑惑（应用也可能产生bug）：某个URI到底匹配哪个路由？其匹配的路由到底是不是我们原先设计中想要的呢？

我们可以使用 `php bin/console router:match` 命令来对一个URI匹配哪个路由进行调试。比如对 / 路由的调试命令为：

```
php bin/console router:match /
```

该命令会产生如下输出：



```
vagrant@vagrant-ubuntu-trusty-64:/vagrant/symfony$ php bin/console router:match /
[OK] Route "home" matches

+-----+-----+
| Property | Value
+-----+-----+
| Route Name | home
| Path | /
| Path Regex | #^/$#s
| Host | ANY
| Host Regex |
| Scheme | ANY
| Method | ANY
| Requirements | NO CUSTOM
| Class | Symfony\Component\Routing\Route
| Defaults | controller: AppBundle:Default:index
| Options | compiler_class: Symfony\Component\Routing\RouteCompiler
+-----+-----+
vagrant@vagrant-ubuntu-trusty-64:/vagrant/symfony$
```

可见，如我们的设计，`/` 匹配了我们定义的 `home` 路由，它所调用的正是我们规定的 `AppBundle:Default:index` 动作。

路由调试

有时，我们需要知道在应用中到底定义了多少路由，这时我们可以用如下的命令：

```
php bin/console debug:router
```

该命令将列出所有的路径名、调用方法（是 `POST` 、 `GET` 或者其它还是无所谓）、协议（比如是不是必须要求 `https` ） 、主机（可以由哪些主机对此访问）和路径。

更多的路由配置

我们再来看几个路由，以了解更多的路由配置。

在这个藏书管理程序中，有一个功能是书籍列表（分页）。该路由定义如下：

```
book_list:  
    path: /books/list/{type}/{key}/{page}  
    defaults:  
        page: 1  
        type: title  
        key: all  
    _controller: AppBundle:Book:list
```

SF采用`{...}`来标记路径中的参数。在上例的路由中，其路径有三个参数：

- `type`：确定书籍列表的类型。一种是列书名，一种是列tag（更多的说明见后续章节）；
- `key`：如果`type`是列书名，这里就是书名的开始部分；如果`type`是列tag，这里就是一个tag；
- `page`：确定要显示第几页。

因此用这样一个单一的路径，我们可以可以显示三种不同的书籍列表：

1. 不带任何参数，或者参数为缺省值，那么列出所有藏书（按照`id`降序，亦即最新登录的书籍最先展示）的第一页。
2. 按照书名开头进行搜索，显示匹配书名开头部分的那些书籍。
3. 按照tag进行搜索，显示匹配tag的那些书籍。

在我的网站中，这些页面的效果如下所示¹：

任氏有无轩 | 藏书列表 | 首页

<https://rsywx.net/books/list>

直接去第几页

直接去

编号	书名	作者	购买/整理日期	位置
01846	资本主义与二十一世纪	【美国】黄仁宇	2016年03月23日	n1
01845	菊与刀	【美国】本尼迪克特	2016年03月21日	n1
01844	人类与社会	【美国】《科学新闻》杂志社	2016年02月22日	n3
01843	小心轻放的光阴	【中国】陆苏	2016年02月14日	n1
01842	这些人，那些事	【台湾】吴念真	2016年02月14日	n1
01841	莎士比亚	【英国】伯吉斯	2016年01月23日	n3
01840	全本红楼梦	【清朝】曹雪芹	2016年01月11日	s2
01839	我们需要什么样的文化繁荣	【大陆】王京生	2016年01月11日	n3
01838	岛上书店	【美国】泽文	2016年01月11日	n3
01837	切尔诺贝利的悲鸣	【白俄罗斯】阿列克谢耶维奇	2016年01月11日	n3
01836	用一间书房抵抗全世界	【大陆】魏小河	2016年01月11日	n3
01835	当上她的明眸	【大陆】刘慈欣	2016年01月11日	n2

任氏有无轩 | 藏书列表 | 首页

<https://rsywx.net/books/list/title/红>

直接去第几页

直接去

编号	书名	作者	购买/整理日期	位置
01778	红楼梦（连环画）	【清朝】曹雪芹	2014年07月05日	f2
01222	红字	【美国】霍桑	1982年01月28日	i6
00976	红与黑	【法国】司汤达	1991年03月05日	h4
00484	红楼复梦	【清朝】小和山樵	1995年12月08日	d5
00483	红楼梦诗词鉴赏辞典	【中国大陆】贺新辉	1991年05月17日	d5
00482	红楼梦诗词最新全译本	【港澳台】陈龙安	2006年11月11日	d5
00481	红楼梦鉴赏辞典	【中国大陆】上海市红楼梦学会	1989年12月15日	d5
00480	红楼梦艺术管探	【中国大陆】杜景华	2006年11月11日	d5
00478	红楼梦之谜	【中国大陆】上海市红楼梦学会	1994年06月10日	d5
00477	红楼梦人物论	【中国大陆】王昆仑	2006年11月11日	d5
00476	红楼梦新续	【中国大陆】周玉清	1991年06月27日	d5
00475	红楼梦学习材料	【中国大陆】甘春田杭州师大系宣传组	1974年06月04日	d5

任氏有无轩 | 藏书列表 | 首页

<https://rsywx.net/books/list/tag/美国>

直接去第几页

直接去



美国	直接去第几页			
搜索	直接去			
<hr/>				
编号	书名	作者	购买/整理日期	位置
01845	菊与刀	【美国】本尼迪克特	2016年03月21日	n1
01838	岛上书店	【美国】泽文	2016年01月11日	n3
01832	禅与摩托车维修技术	【美国】波西格	2015年12月30日	n3
01826	Becoming Steve Jobs	【美国】Schlender	2015年11月18日	s5
01784	Star Wars: Trilogy	【美国】Lucas	2014年06月23日	z1
01754	Killing Lincoln	【美国】O'Reilly	2013年11月27日	z1
01750	Killing Kennedy	【美国】O'Reilly	2013年04月17日	z1
01736	论中国	【美国】基辛格	2012年09月22日	s5
01734	银河帝国——第二基地	【美国】阿西莫夫	2012年08月09日	n3
01733	银河帝国——基地与帝国	【美国】阿西莫夫	2012年08月09日	n3
01732	银河帝国——基地	【美国】阿西莫夫	2012年08月09日	n3
01687	Dinner	【美国】Gardner	2011年01月26日	z1

我们需要注意的是浏览器地址栏显示的地址。还有就是，虽然这是三个不同的动作，但是它们使用的显示模板是一样的。

在该路由的配置中，其 `defaults` 段和之前的不同。除了按照常规要制定一个控制器和动作外，我们对该路由的路径中出现的三个参数设置了一个缺省值。所以我们在访问 `books/list` 的时候，实际上就是访问了 `/books/list/title/all/1`。

只能进行 POST 访问的路径

该应用中还有一些路径是用来处理表单输入的。对于这样的路径，我们不希望用户在浏览器中直接输入URI而进行误操作，所以需要对该路径可以通过怎样的方法进行访问加以限制。

比如下面这个为一本书增加tag的路径：

```
tags_add:
  path: /books/addtag
  defaults: {_controller: AppBundle:Book:tagsAdd}
  requirements:
    _method: POST
```

这里我们设置了路由的一些额外要求。其中的 `_method: POST` 规定该路由只能通过 `POST` 方式访问。

对参数的限制

我们有一个书籍详情的页面，列出书籍的详细信息。该路径定义如下：

```
book_detail:
    path: /books/{id}.html
    defaults: { _controller: AppBundle:Book:detail }
```

于是我们就可以用类似 `/books/00005.html` 这样的方式来访问一本书籍。但是这么做有一个小问题。

在我们的数据库中，一本书的 `bookid` 有5位，按照约定，它应该都是数字并有前导0，比如 `00666`，`01234` 等。类似 `1234`（位数不够），`abcd8`（混杂了字母）这样的参数是不合理的。如果用上述的这个路径定义，我们访问 `/books/1234.html` 的时候，也还会匹配到上面的那个路径。这样做不会有什么致命的后果，只是数据库中无法找到这本书，显示一个“该书籍找不到”的页面而已²。但是这样不是很好的方法，如果我们能对路径中参数加以限制，使得那些不符合要求的参数（和URI）根本不访问该路由，我们至少解决了部分问题。

于是我们要对该路由中参数 `id` 加以限制。我们修改上述路由为：

```
book_detail:
    path: /books/{id}.html
    defaults: { _controller: AppBundle:Book:detail }
    requirements:
        id: \d{5}
```

通过一个简单的正则表达式，我们约定 `id` 这个参数必须是5位数字，因此类似 `1234`，`abcd8` 这样的参数将不会触发这个路径。访问这样的URI只会出现一个Apache自身的404页面。

我个人认为，我会比较喜欢这种处理方式。这样做的一个好处是减少了后台控制器中的判断。

路由定义的陷阱

随着我们应用的开发，路由的定义肯定会越来越多。我们有必要强调一些在路由定义时可能会犯的错误。

用YML定义的路由，遵循“最先匹配”的原则。某个URI只要符合某个特定的路径模式就会触发相应的动作。这么一来就可能会有问题。

假定在我们的路由文件中，有这样两个路由：

```
display_by_tag:  
    path: /tag/{tag}  
  
add_tag:  
    path: /tag/add  
    requirements:  
        _method: POST
```

如果我们在一个表单中增加了一些tag，然后提交，我们的本意当然是要让 `add_tag` 这个路由中指定的动作去执行为一本书增加tag的动作。但是，在这样的路由配置情形下，首先被匹配的是 `display_by_tag` 这个路径，因此我们试图添加的tag不会真正地保存。

当然，要解决上面提到的问题也有很多方法。我们可以重新规划路径，调整路由定义的顺序等。

一般而言，路由的设计需要考虑到两点：

1. 简单、直观
2. 越是特殊的路由就要越早定义。

路由是SF中非常核心的一个部件。它可以由其它应用独立引用。

对于路由的解说，本文只能给出最基本的讲解。[SF的官方文档中对于路由的说明](#) 才是最权威的指南。

本应用完整的[路由文件](#)已经上传。

路由定义完毕后，我们需要开始模板的编写。

1. 我们现在的应用因为只有样本数据，所以是无法显示出这样的结果的。但是我们在后面会看到，即便如此，我们还是可以显示一个示范的效果。 ↵
2. 该页面不是Apache自己的404页面，而是我们定制的一个页面。 ↵

模板

模板属于MVC结构中的V。它的出现是为了更好地分离程序代码（对用户不可见）HTML呈现（用户可见）。

对模板开发者的要求是能掌握一些基本的模板语法，写出基本框架，加入对应的控制和输出，最终形成一个可以呈现给用户的页面。

SF模板引擎：Twig

SF的创始人Fabien Potencier研制了一个全新的模板引擎：[Twig](#)。我很喜欢这个引擎。我个人认为它有这么几个优点：

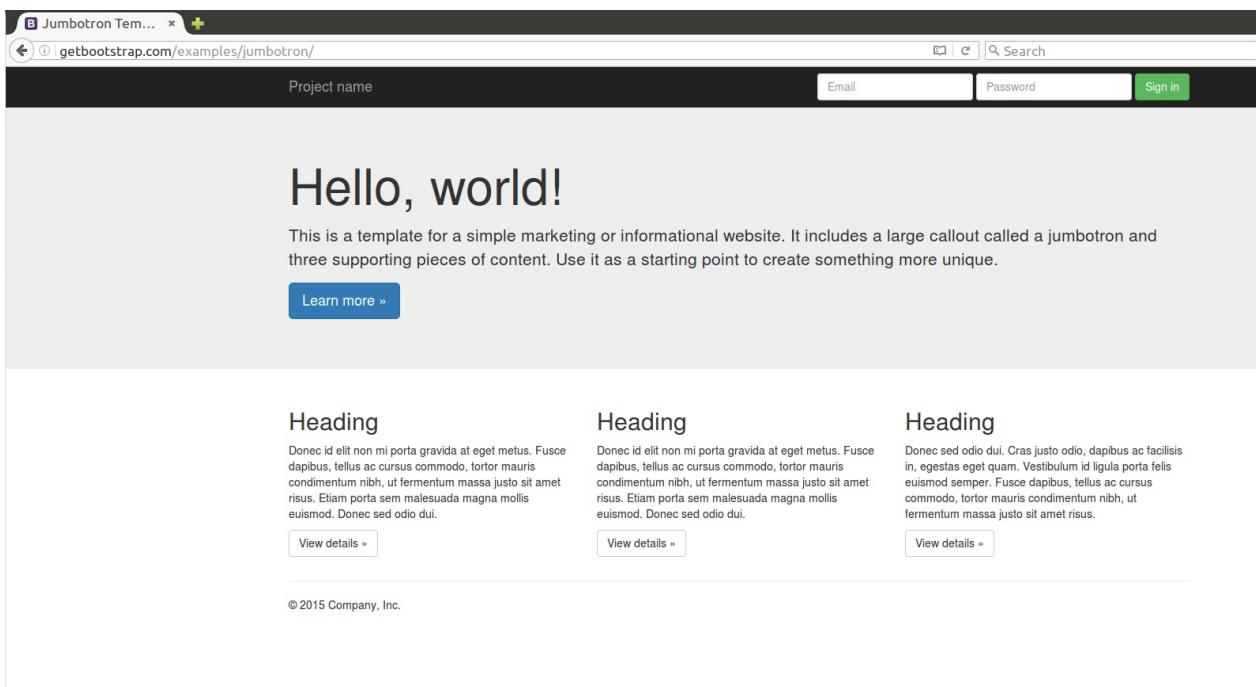
1. 语法简单，上手快；
2. 支持嵌套、继承等模板高级特性；
3. 扩展性好，可以开发Twig插件扩充功能；
4. 编译速度快，效率高。

Twig是SF框架中的一个部件，也可以独立被其它框架使用。在SF应用创建后，已经支持Twig。

页面结构的设计

我们的应用不会只有一个页面，但是众多页面却可能有着相同的布局。比如，都采用“顶部导航栏+中间内容+底部导航栏”的上中下布局，或者采用“左导航栏+中间内容+底部导航栏”的布局，等等等等。

我一般会采用基于[Bootstrap](#)的模板加以定制。我现在[任氏有无轩](#)使用的是从[WrapBootstrap](#)上购买的一个商业模板。在本应用开发中，我们会用一个免费的Bootstrap模板，[Jumbotron](#)：



这个页面布局很合适我们的应用。

1. 顶部的黑色导航区可以放应用名称，导航链接，右边的用户登录部分可以用搜索框来代替；
2. 下面的Jumbotron部分可以用在某些页面中作为强调部分；
3. 主内容区三个并排的列可以在首页中显示不同栏目的内容，也很方便变成一个列用在别的页面中；
4. 最下面的一块地方正好可以用来显示一些版权信息之类的。

为了在我们的应用中使用这个布局，我们需要做一些额外的工作。

保存Bootstrap模板

我们首先要将Jumbotron这个页面保存下来。在浏览器中选择保存页面。如果有选择保存方式的话，选择“全部”。这样我们保存的就不仅是该页面的HTML文件，而且包括该HTML页面中引用到的CSS，JS和图片文件等。

在我们SF项目的 web 目录下创建几个目录，如 `css`，`js`，`img`，将保存下来的页面引用到的CSS，JS和图片文件分别拷贝到对应的目录中去。

接着，我们将刚才保存的那个HTML文件移到 `src/AppBundle/Resources/views/default` 并改名为 `layout.html.twig`。

修改首页显示该模板

下一步，我们修改当前首页所渲染的模板，使其显示我们刚才创建的模板，这样方便我们进一步修改该模板并看到效果。

让我们修改 `src/AppBundle/Controller/DefaultController.php` 中的 `indexAction` 为：

```
public function indexAction(Request $request)
{
    // replace this example code with whatever you need
    return $this->render('AppBundle:default:layout.html.twig');
}
```

刷新我们的应用页面，可以看到Bootstrap的页面已经替换了原来的SF欢迎页面。但是由于该页面中引用的CSS、JS文件的相对路径已经发生变化，所以需要进行调整。具体怎样调整这里就不再赘述，原则上只是一些文件路径和文件名的调整罢了。

如果一切顺利，我们应该看到和之前在Bootstrap上一样的页面效果。

模板的继承和包含

Twig模板引擎的一个强大之处在于它支持模板的继承和包含，而且还可以嵌入控制器¹。这为我们定义灵活的页面布局并同时保持一致性提供了方便。

包含

包含是`include`，它并不是Twig的重点，所以我们只是提一下。它用来在一个模板中导入另一个模板。通常情形下，那些相对固定且在多个页面中都会重复出现的内容比较适合被剥离到一个单独的模板文件中。在需要这些内容的地方加以导入即可。

考虑我们使用的Bootstrap样板，我们会很快地发现，顶部导航栏和底部的导航栏都具有这样的性质：相对固定、作为页眉和页脚也会在多个页面重复出现。所以，这两部分可以被提取出去作为一个独立的模板供其它模板使用。

继承

Twig的强大之处在于模板的继承。

学习过OOP的一定知道类继承是怎样的机制。简单地说，子类继承了父类中所有公共特性和公共方法，同时也按照子类的特殊要求加入新的特性和方法。

Twig模板的继承与此也有类似之处。不过由于我们讨论的是页面布局，所以这里的继承也是对页面布局的继承。

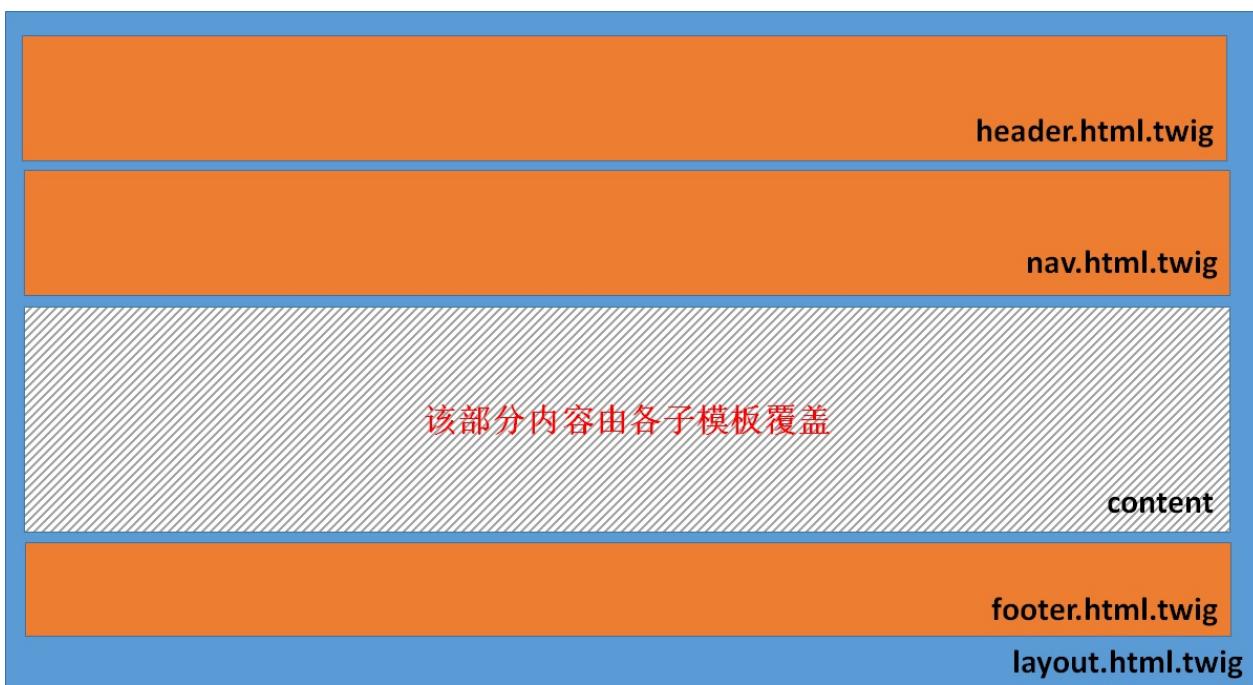
根据上面对Bootstrap样式的分析，我们可以看到，以页面布局来看，我们的页面基本上有三个部分：

1. 顶部导航栏，我们不妨称之为" nav ²"。
2. 中间的内容部分，我们不妨称之为" content "。
3. 底部的信息栏，我们不妨称之为" footer "。

这三部分中，1/3我们已经讨论过了，我们需要将它们独立出去，成为两个独立的模板供包含。

²这部分是非常动态的，各个不同页面要显示的主要内容肯定不同。所以，它也必须独立出去，由各个页面各自完成。

于是我们可以对主模板进行如下图所示的调整：



简要说明一下：

1. 原本的 `layout` 模板被拆分为四块：`header`，`nav`，`index`，`footer`。
2. `index` 模板成为我们首页最终使用的模板，它将继承 `layout` 模板，并用属于该页面的内容覆盖 `content` 页面组块。
3. 其它三个模板分别在 `layout` 模板中被引用。

随着应用的开发，我们会创建更多的页面。但是，其创建方式和创建 `index` 页面的方式类似。

Twig模板中的 block

可以这么说，block（块）是Twig模板构造页面的基础。

在 `layout` 模板中，我们会注意到这么一段：

```
<html lang="zh">
    {% include "AppBundle:default:header.html.twig" %}

    <body>

        {% include "AppBundle:default:nav.html.twig" %}

        {% block content %}{% endblock %}

        {% include "AppBundle:default:footer.html.twig" %}
    </body>
</html>
```

`{% block content %}{% endblock %}` 在 `layout` 模板中定义了一个名为 `content` 的块。我们注意到，`layout` 中这个块中没有任何内容。它的内容是由各个具体的页面填充的，比如在 `index` 模板中：

```
{% extends "AppBundle:default:layout.html.twig" %}

{% block content %}
    <div class="jumbotron">
        ...
    </div> <!-- /container -->
{% endblock %}
```

首先，`index` 模板声明它会扩展（也就是继承自）`layout.html.twig` 模板。因此，所有在 `layout` 模板中出现的布局因素（包括 `layout` 需要引用的其它模板中的布局因素）也都会在 `index` 中出现。

但是，`index` 模板重写了 `content` 块，替换成自己的内容。而 `index` 模板中未覆盖的那些块，会仍然采用 `layout` 中的相应内容。

这样的一个过程，正是继承的过程。

一般而言，每当我们要创建一个新的页面，我们会采用类似的方式：

1. 从主模板（本例中的 `layout`）继承；
2. 重写相应的块。一般而言，也是一个诸如名为 `content` 的块；
3. 根据需要，重写另外的一些块，比如 `title`，`extralink`，`extrajs` 等以达到高度定制的目的。

Twig模板的存放位置

缺省情况下，所有的Twig模板文件都应该存放在 `src/AppBundle/Resources/views` 之下。为了更方便的组织模板文件，我们可以在这个目录之下再创建一些子目录。

引用一个模板的时候，无论是在模板之中还是在控制器之中，方法都是类似“`AppBundle:目录:模板`”的格式。比如，`AppBundle:default:nav.html.twig` 对应的模板文件是 `src/AppBundle/Resources/views/default/nav.html.twig` 文件。

更深层的路径也是支持的，比如 `AppBundle:Theme1/default:index.html.twig` 对应的模板文件是 `src/AppBundle/Resources/views/Theme1/default/index.html.twig`。这里的关键是，“`目录`”部分可以被替换成更完整的路径层次，但是我们不能修改 `AppBundle` 和最终调用的模板文件的名字。

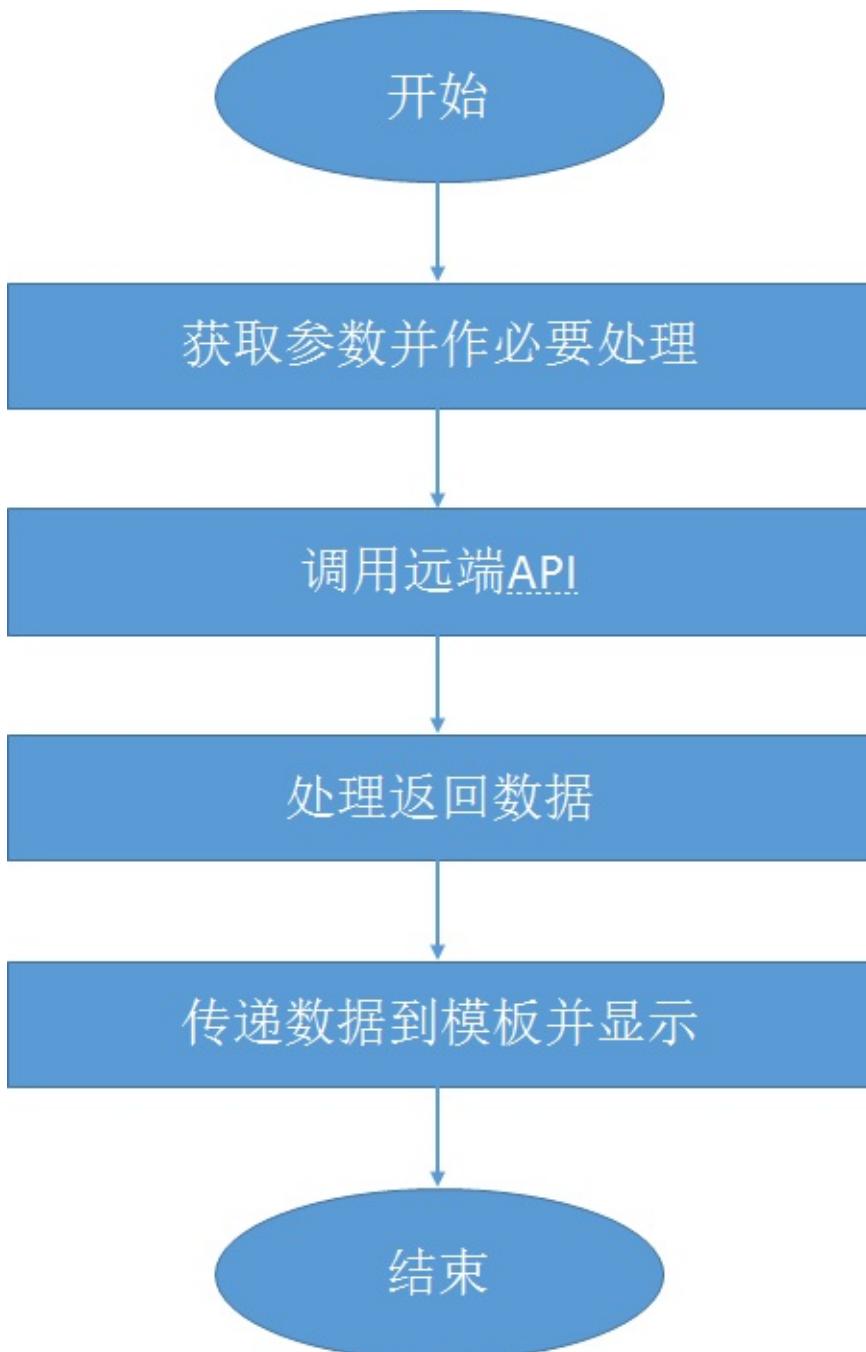
1. 在模板中嵌入控制器是高级用法，但是其语法却非常简明。我们在后续章节会看到详细讲解。 ↪
2. 我们没有把它叫做 `header` 是因为我们还会有一个 `header` 的模板，用来统一管理 HTML 页面中的头信息（如 `meta`，CSS/JS 的引用等）。 ↪

首页的编写

由于本站点采用了新的架构（即不再应用中包含数据的CRUD操作，而改为调用对应的RESTful API的方式），因此站点编写工作量大大降低。

SF基于MVC架构，所以编程部分在C部分，也就是控制器（Controller）部分。

一个常规控制器的流程如下：



我们之所以首先用首页来进行控制器编写的说明，主要是因为它是一个站点的入口，具有特别重要的地位，而且它往往独一无二与别的页面不同（别的页面如书籍详情等会重复）。另外，首页中也用到控制器嵌入等重要技术，值得首先加以描述。

顶端项目栏和搜索框

创建一个新的路由

首页的顶端我们要设计成项目名称和搜索框。修改的是 `AppBundle:default:nav.html.twig` 文件，并最终呈现在 `AppBundle:default:index.html.twig` 的渲染效果中。

既然是搜索栏，我们要为搜索这个动作设定一个路由。出于简化编程和示范的目的，我们约定搜索只搜索书籍。我们修改 `src/AppBundle/Resources/config/routing.yml` 并增加一个新的路由如下：

```
books_search:  
    path: /books/search  
    defaults: {_controller: AppBundle:Book:search}  
    requirements:  
        _method: POST
```

创建一个新的控制器

从路由的设置我们看到，我们同时要创建一个新的控制器（`BookController`）并在其中创建一个搜索的方法（`searchAction`）来处理这个工作。

创建新的控制器很简单，我们可以简单地从 `DefaultController.php` 出发，将其拷贝黏贴为 `BookController.php`，并作相应修改：

```
<?php

namespace AppBundle\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;

class BookController extends Controller
{
    /**
     * Test changes in Linux
     */
    public function searchAction(Request $request)
    {
        // replace this example code with whatever you need
        dump($request);
        die();
    }
}
```

简单说明如下：

1. 这个控制器的代码并未完成，我们只是将通过表单提交的数据简单地加以显示而已。该方法的具体实现需要借助于别的尚未完成的模板，所以我们这里进行了简单处理。

修改 **nav.html.twig**

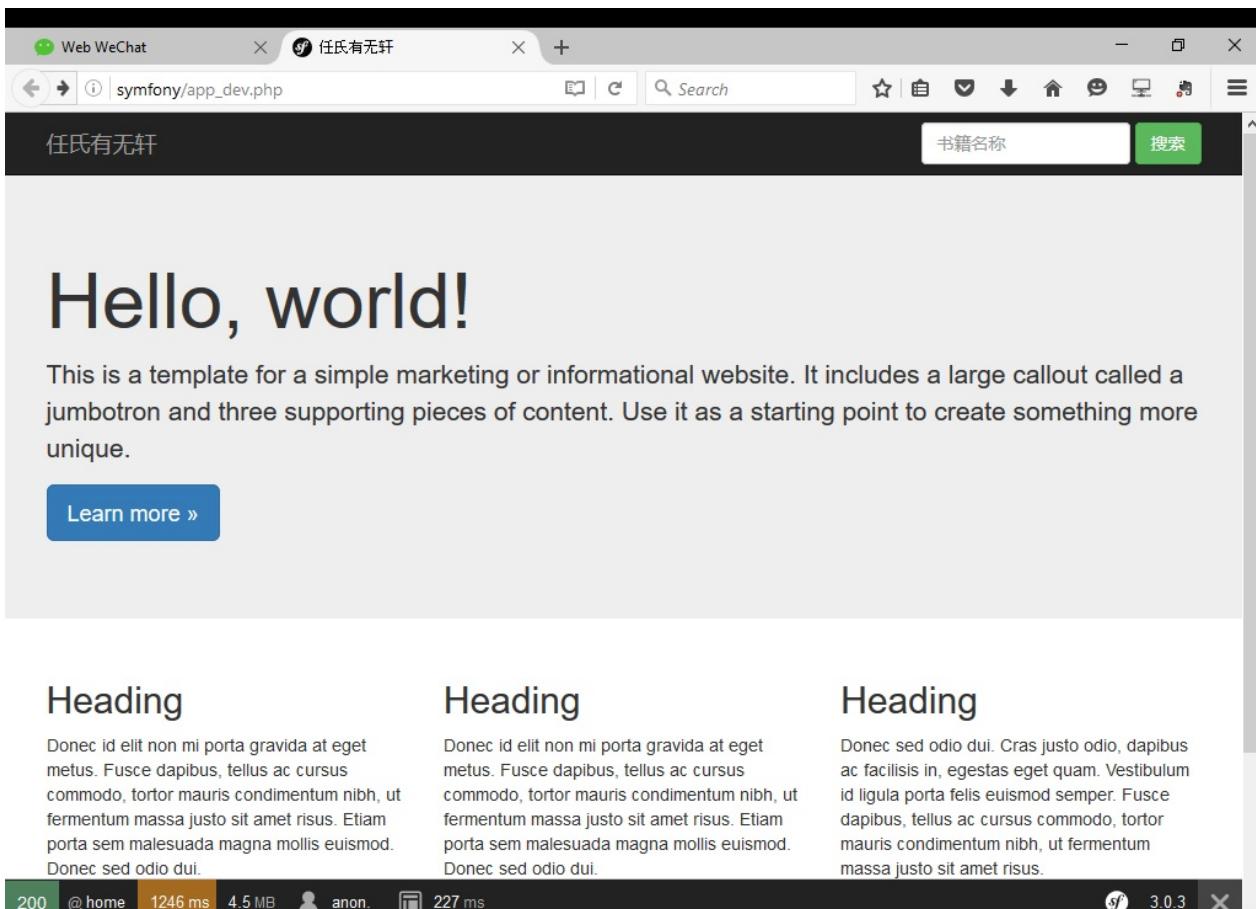
至此，我们可以修改 `nav.html.twig` 文件如下：

```
<nav class="navbar navbar-inverse navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="{{path('home')}}">任氏有无轩</a>
        </div>
        <div id="navbar" class="navbar-collapse collapse">
            <form class="navbar-form navbar-right" action="{{path('books_search')}}" method="get">
                <div class="form-group">
                    <input type="text" placeholder="书籍名称" class="form-control" name="key">
                </div>
                <button type="submit" class="btn btn-success">搜索</button>
            </form>
        </div><!-- .navbar-collapse -->
    </div>
</nav>
```

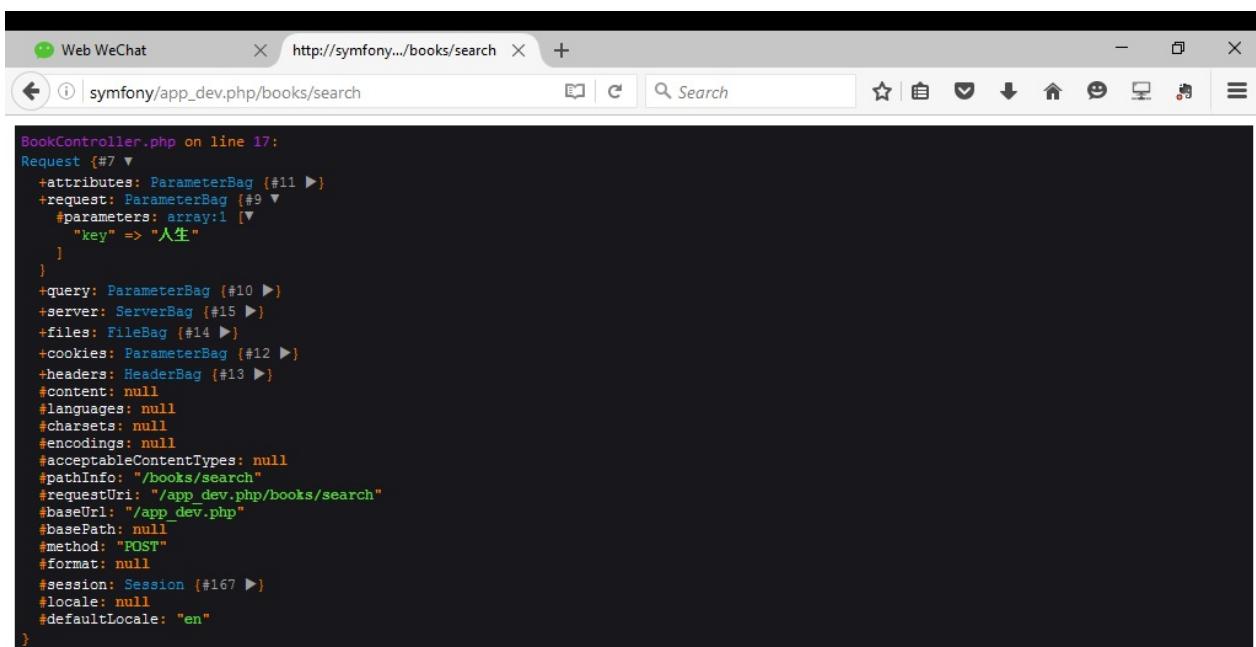
简单说明如下：

1. 我们修改了项目名称为“任氏有无轩”。
2. 我们为表单提供了 `action` 和 `method` 参数。
3. 我们使用了Twig专用的辅助函数 `path` 来为制定表单指定动作。它指向我们在上一步定义的 `books_search` 路由。
4. `method` 只是简单的定义为POST。
5. 我们为一个文本输入框定义了一个名字为 `key`。

我们刷新一下页面，会看到如下效果：



而如果我们在“书籍名称”框中输入一些文字并点击“搜索”后会出现如下界面（部分分支已展开）：



可见，VarDumper包中提供的dump函数能比PHP内置的var_dump函数更有组织地、更灵活地显示一个变量的值。从图中也能看到，我们通过表单提交的变量(key)获得了正确的赋值。

Jumbotron的编写

顶端项目栏和搜索栏之下是所谓的Jumbotron部分。这是首页特有的内容。在本应用中，我们用来显示最近购买的一本书。

要显示最近购买的一本书，有两种做法。一种是在所谓的 `Default:indexAction` 中直接获取最新的一本书，并将代表这本书的对象直接传递给模板；另外一种是在模板中嵌入一个控制器，让控制器获取相应的最新图书并显示。在Jumbotron的编写中我们采用第二种方式。

我们首先编写控制器如下：

```
public function latestAction()
{
    $b= json_decode(file_get_contents('http://api/book/latestBook'));

    return $this->render("AppBundle:book:latest.html.twig", ['book' => $b->out[0]]);
}
```

由于我们采用了API调用，这个控制器可以写得非常简单。

latest.html.twig 模板

我们可以简单地将当前 `index.html.twig` 模板中Jumbotron的 `<div>...</div>` 部分提取出来，成为 `latest.html.twig` 模板的基础，并进行一些修改。此时，我们从控制器的编写中可以看到，该模板会使用到一个 `book` 变量。

```
<div class="jumbotron">
<div class="container">
    <div class="row">
        <div class="col-md-8">
            <h1>{{book.title}}</h1>
            <p>作者：{{book.author}} ({{book.region}}) </p>
            <p>收录时间：{{book.purchdate|date('Y年m月d日')}}</p>
            <p>版次：{{book.ver}}</p>
            <p><a class="btn btn-primary btn-lg" href="{{path('book_detail', {'id':book.id})}}>查看详情</a></p>
        </div>
        <div class="col-md-4">
            
        </div>
    </div>
</div>
```

注意如下几点：

1. 我们这里用Bootstrap布局来安排我们的书籍信息部分和书籍封面部分，左右分列。
2. 所有的书籍信息的显示都来自从控制器传递过来的 book 变量。
3. 我们再次用到 path 函数来构造浏览书籍详情的连接。这里我们还为该路径提供了一个参数，请读者必须注意参数传递的方式：{'id':book.bookid}。
4. 暂时我们用一个缺省的书籍封面作为所有书籍的封面。在后续章节中，我们还会进一步编写一个方法来显示书籍封面。

我们简化了Jumbotron的编写。在我自己开发的[rsywx.net](#)站点中，

index.html.twig 的修改

最后，我们修改 `index.html.twig`，用嵌入控制器的方式替代原来的Jumbotron的 `<div>...</div>` 部分：

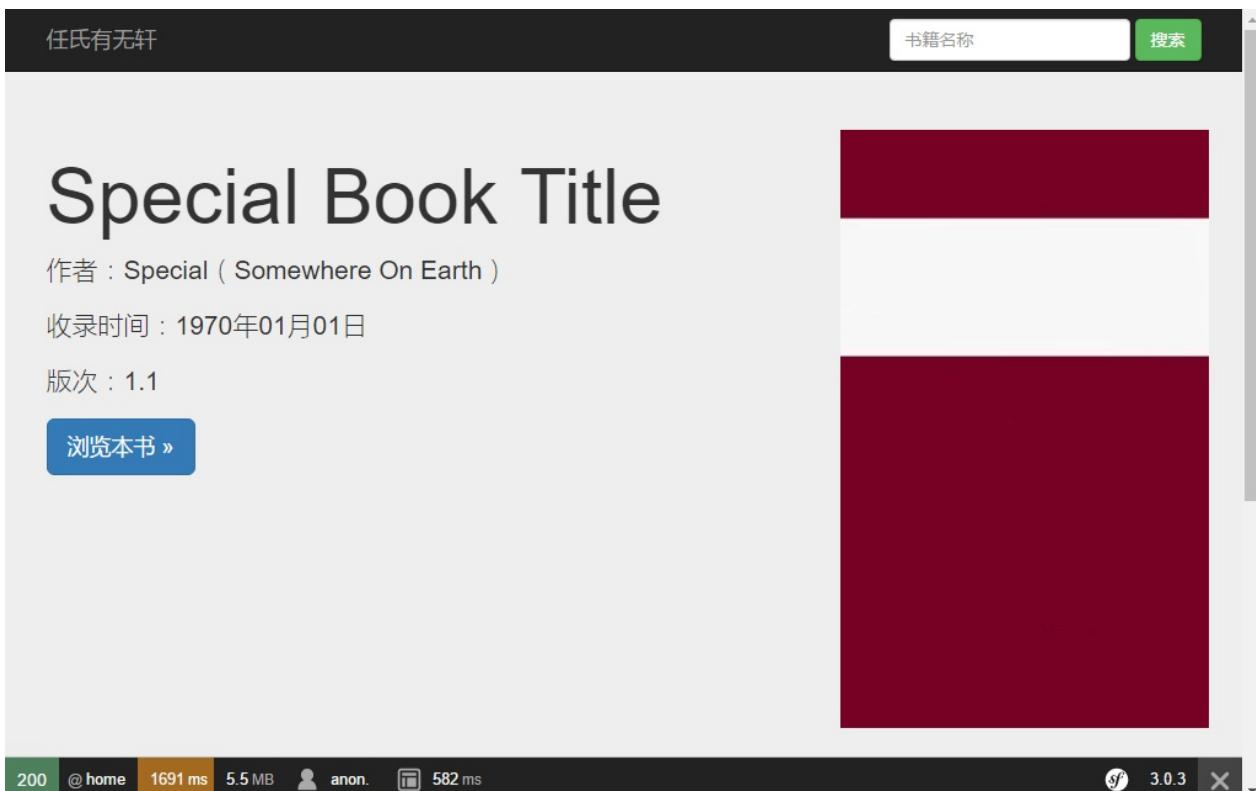
```
{% block content %}
    {{ render(controller('AppBundle:Book:latest')) }}

    <div class="container">
        ...
    </div>
```

此处我们用到Twig的一个高级语法 `render`，它用来嵌入一个从控制器的返回。

效果

让我们保存所有修改，然后重新刷新首页页面：



由于我们对“最新登录书籍”的定义是按照其ID，所以我们会看到之前在样本数据填充中最后生成的书籍记录会被选出。

Headline的编写

Jumbotron之下是所谓的Headline。原布局中由3个 `col-md-4` 构成，在我们的应用中，会改成4个，它们分别是：

1. 藏书统计信息；
2. 书评统计信息；
3. 博客最新文章¹；
4. 维客的链接和其它链接²；

这四个小板块的渲染会采用嵌入控制器和变量直接传递的方式。

`index.html.twig` 的进一步改写

我们进一步改写 `index.html.twig` 文件，将原先三个 `<div>` 的部分抽取出来成为 `AppBundle:book:summary.html.twig` 和 `AppBundle:reading:summary.html.twig` 两个模板。

```
<!-- book:summary.html.twig -->
<p>截止{{'today'|date('Y年m月d日')}}，任氏有无轩藏书{{summary.summary.0.bc|number_format(0,'.')}}本，最近({{summary.last.0.purchdate|date('Y年m月d日')}})收藏/整理的书籍是<strong>{{summary.last.0.title}}</strong>，购买价格是{{summary.last.0.purchprice}}元，购买日期是{{summary.last.0.purchdate|date('Y年m月d日')}}。</p>
```

```
<!-- reading:summary.html.twig -->
<p>截至{{"now"|date('Y年m月d日')}}，任氏有无轩主人撰写了{{rs.summary}}篇评论。</p>
<p>最近({{rs.last.datein|date('Y年m月d日')}})评论的书籍是<strong><a href="{{path("book_detail", {id: rs.last.id})}}>{{rs.last.title}}</a></strong>，评论人是{{rs.last.username}}，评论内容是{{rs.last.content}}。</p>
```

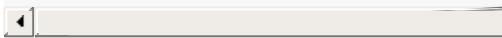
对 `index.html.twig` 改写如下：

```

{{ render(controller('AppBundle:Book:latest')) }}

<div class="container">
    <!-- Example row of columns -->
    <div class="row">
        <div class="col-md-3 feature">
            <h3><a href="{{path("book_list")}}><i class="glyphicon book"></i>藏书</a></h3>
            {{ render(controller('AppBundle:Book:summary')) }}
        </div>
        <div class="col-md-3 feature">
            <h3><a href="{{path('reading_list')}}><i class="glyphicon tags"></i>读书</a></h3>
            {{ render(controller('AppBundle:Reading:summary')) }}
        </div>
        <div class="col-md-3 feature">
            <h3><a href="http://www.rsywx.net/wordpress"><i class="glyphicon pen"></i></a></h3>
            <p>本博客自2003年开始设立。写的少不是因为我不思考。我思考的越多，写下来的就越少。</p>
            <p>最近的文章发布于{{wp.post_date|date('Y年m月d日H时i分')}}，题为“<strong><a href={{path('post', {id: wp.id})}}>{{wp.title}}湖边小屋</a></strong>。<br>还可以和我<strong><a href="{{path('contact')}}>取得联系</a></strong>。</p>
        </div>
    </div> <!-- /row -->
</div> <!-- /container -->

```



编写对应的控制器

我们需要在 `BookController` 和 `ReadingController` 中增加对应的控制器方法分别响应上述模板中对控制器的调用：

```

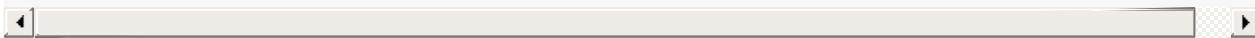
// BookController.php
public function summaryAction()
{
    $summary = json_decode(file_get_contents('http://api/book/summary'))->out;

    return $this->render("AppBundle:book:summary.html.twig", ['summary' => $summary]);
}

// ReadingController.php
public function summaryAction()
{
    $summary = json_decode(file_get_contents('http://api/reading/summary'));

    return $this->render("AppBundle:reading:summary.html.twig", ['rs' => $summary->out]);
}

```



效果

我们可以看效果了。再次刷新首页³得到效果如下：

藏书

截至2016年04月26日，任氏有无轩藏书101本。约6,049千字，6,049页。

最近(2016年03月19日)评论的书籍是《Special Book Title》，题为“Review 2.”。

读书

截至2016年04月26日，任氏有无轩主人撰写了2篇评论。

博客

本博客自2003年开始设立。写的少不是因为我不思考。我思考的越多，写下来的就越少。

最近的文章发布于2016年02月22日10时37分，题为“老彼得初中的最后一个学期”。

维客

这是我整理的一些资源，主要是电子书和我最喜爱的[湖人队的赛程](#)。

还可以和我[取得联系](#)。

2016, RSYWX

200 @ home 2593 ms 6.5 MB anon. 720 ms sf 3.0.3

小结

至此，我们已经基本完成首页的编写（还有一些内容我们会在后续章节讲述）。

1. 我们通过**Twig**模板的继承和包含，对布局模板进行了重构，分成了几个互相独立又互相依赖的子模板。这样做的好处是，每个模板的**HTML**代码总量都不大，方便调整和调试。
2. 通过内嵌控制器，我们进一步重构了模板，并就此编写了对应的控制器动作和模板。
3. 模板的编写基本基于当今流行的**BootStrap**框架，大大缩短了时间，提升了效率。

我们还将看两个页面的编写。重点是分页（在“书籍列表页面”中讲述）和图片处理、**jQuery**的集成（在“书籍详情页面”中讲述）。

不过在此之前，我们先来熟悉一下**SF**调试环境下的状态栏。



该状态栏只有在调试环境下出现。从左到右，该状态栏图标的含义为：

1. HTTP返回状态。200表示正常。

2. 当前调用路由名。
3. 峰值内存占用。
4. 当前身份。
5. 渲染耗时。
6. SF当前版本。

将鼠标停在各图标上还有更详细的信息。该状态栏对调试还是很有一定的用处的。

1. 在本教程中，为了方便起见，博客文章是“虚拟”的。在实际应用中，用到了对后台 WordPress数据库的调用。 ↵
2. 这部分代码非常简单，只有静态代码。所以本教程中不再展开描述。 ↵
3. 该页面效果所使用的博客数据库是真实的，而不是虚拟的。 ↵

书籍详情页面

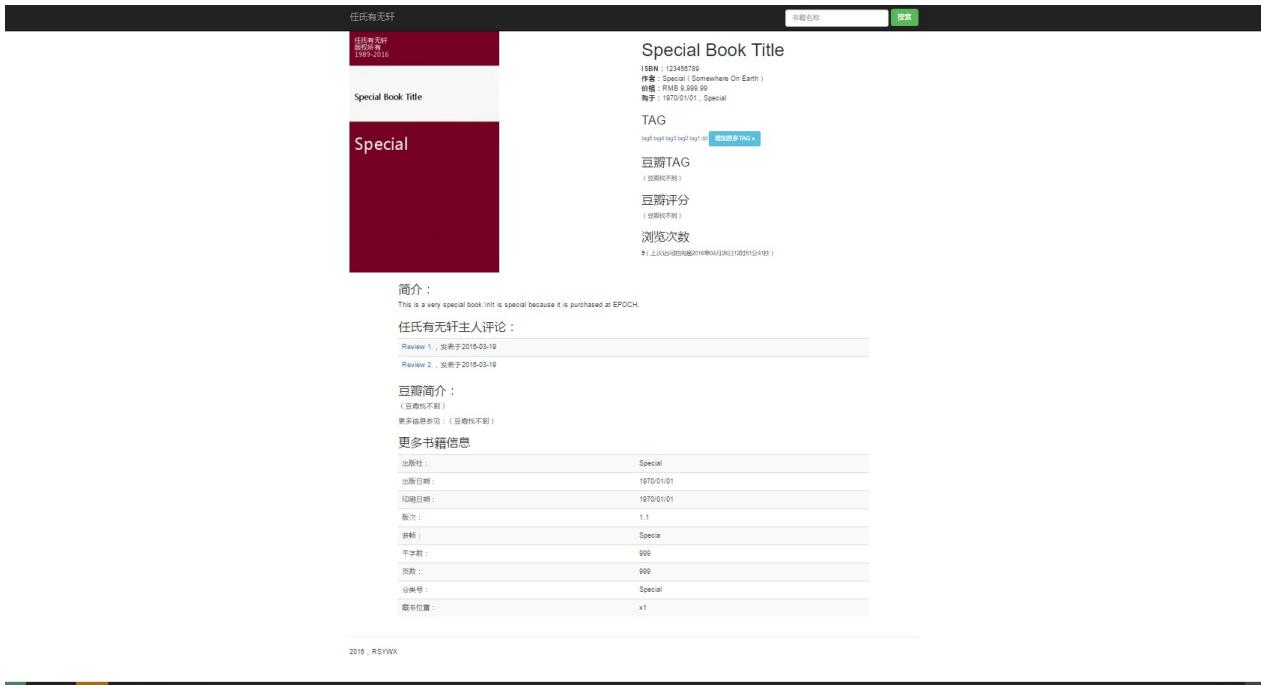
第二个我们要详细解释的页面编程是书籍详情。

在该页面中我们要显示很多东西：书籍的所有登陆信息，包括TAG，一个jQuery的表单方便访客增加自己的TAG，如果这本书有相关的评论，那么还要显示评论的连接等等等。该页面编程量非常大，请做好准备。

创建 detail.html.twig 页面

这个页面很长，我这里就不贴出全部的代码了。同时我也不会写出所有的步骤：一般而言，SF给出的错误提示是非常精确的。在过程中参考错误提示，不断地修订——增加路由，控制器方法，模板等，最终一定能调试通过。

我们贴一张最终渲染的效果图：



我们分别来看几个重要的版块。

显示书籍封面图片

我们这里要显示的书籍封面图片，以JPG图片格式保存在 `web/covers` 目录之下，每本书都对应一个图片，其格式是 `{{bookid}}.jpg`，例如途中所示的书其 `bookid` 为 `99999`，那么它对应的封面图片应该是 `99999.jpg`。如果该文件名的图片不存在，我们将显示一个 `default.jpg`。

但是我们不是简单地显示图片，而是要加一个水印。同时，对于缺省图片，我们需要根据必要的书籍信息（书名、作者）在缺省图片上显示。因此，虽然只有一个缺省图片，其最终效果是动态的。

我们先看其路由定义：

```
cover:
path: /books/cover/{id}_{title}_{author}_{width}.png
defaults: {_controller: AppBundle:Default:cover, width: 300}
requirements:
title: .+
```

从路由定义我们可以看到，我们的图片是动态生成的。

再看对应的动作，我们将其放置在 `DefaultController` 中的 `cover` 动作中：

```
public function coverAction($id, $title, $author, $width)
{
    // Construct image file name based on
    $path = 'covers/';
    $ext = '.jpg';
    $filename = $path . $id . $ext;

    $default = false;

    // Check if the file exists
    if (!file_exists($filename))
    {
        $filename = $path . 'default' . $ext;
        $default = true;
    }

    list($w, $h) = getimagesize($filename);

    $nw = 300;
    $nh = $nw / $w * $h; // Propotionally change the width/height
    // Resize image
    $oimg = imagecreatefromjpeg($filename);
    $nimg = imagecreatetruecolor($nw, $nh);

    imagecopyresampled($nimg, $oimg, 0, 0, 0, 0, $nw, $nh, $w, $h);

    // Print copyright texts
    $copytext1 = "任氏有无轩";
    $copytext2 = "版权所有";
    $copytext3 = "1989-" . date("Y");

    $color = imagecolorallocate($nimg, 255, 255, 255);
    $color2 = imagecolorallocate($nimg, 0, 0, 0);
    $font = $path . 'msyh.ttf';
```

```

imagettftext($nimg, 10, 0, 10, 26, $color, $font, $copytext1);
imagettftext($nimg, 10, 0, 10, 40, $color, $font, $copytext2);
imagettftext($nimg, 10, 0, 10, 54, $color, $font, $copytext3);

if ($default)
{
    //Print title
    imagettftext($nimg, 12, 0, 10, 140, $color2, $font, $title);
    // Print author
    imagettftext($nimg, 24, 0, 10, 240, $color, $font, $author);
}

//Resize the image to fit into reading list
if ($width <> 300) //300 is the image width for book detail
{
    $height = $width / $nw * $nh;

    $timg = imagecreatetruecolor($width, $height);
    imagecopyresampled($timg, $nimg, 0, 0, 0, 0, $width, $height, $nw, $nh);
}
// Output the image
header('Content-type: image/png');
if ($width == 300)
{
    imagepng($nimg, null, 9);
}
else
{
    imagepng($timg, null, 9);
}
imagedestroy($nimg);
imagedestroy($oimg);
imagedestroy($timg);
}

```

这段代码比较长，用到了PHP的GD扩展对图片进行操作。但是流程还是比较直观，请读者自行分析。

注意：图片处理过程中要用到字体文件。由于字体文件比较大，作者没有将其放入代码仓库中，请读者自行拷贝。

显示TAG并允许用户自行添加TAG

显示TAG的过程比较简单，直接贴代码：

```

public function tagsbyidAction($id)
{
    $tags = json_decode(file_get_contents("http://api/book/tagsByBookId/$id"))->out;

    return $this->render("AppBundle:book:tags.html.twig", array('tags' => $tags));
}

```

我们用的是嵌入控制器的方式，所以对应的模板中代码如下：

```
<small>{{ render(controller('AppBundle:Book:tagsbyid', {"id":book.id})) }}</small>
```

显示完书籍的TAG后，我们显示一个按钮，让用户能添加自己的TAG：

```

<a class="btn btn-info btn-sm" data-toggle="modal" href="#addtag" >增加更多TAG </a><br/>
...
<!-- The modal dialog to add more tags -->
<div>
    <div class="modal fade" id="addtag">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header">
                    <button type="button" class="close" data-dismiss="modal" aria-hidden="true" >X</button>
                    <h4 class="modal-title">增加自己的TAG</h4>
                </div>
                <div class="modal-body">
                    <form role="form" method="post" action="{{path('tags_add')}}" id="addtagForm">
                        <div class="row">
                            <div class="col-sm-4 form-group">
                                <label class="control-label" for="newtags">新增TAG</label>
                                <input type="text" class="input-xlarge" id="newtags" name="newtags" />
                                <p class="help-block">(用空格分隔)</p>
                                <input type="hidden" value="{{book.id}}" id="id" name="id" />
                            </div>
                        </div>
                    </form>
                </div>
            </div>
        </div>
    </div>
</div>
<!-- Example row of columns -->
</div>

```

这是标准jQuery的模态对话框。我们不进行特别的分析。只是提请大家注意表单的编写方式。

获取豆瓣信息

我们希望从第三方（豆瓣）获取书籍的一些信息（TAG、介绍、评分等）。这里我们要用到已经用了很多次的 `file_get_contents` 函数：

```
public function detailAction($id, Request $request)
{
    $logger = $this->createLogger();

    $books = json_decode(file_get_contents("http://api/book/bookByBookId/$id"))->out;

    if (count($books) == 0) // Book not found
    {
        return $this->render("AppBundle:book:BookNotFound.html.twig", array('bookid' => $id));
    }
    $book = $books[0];
    $isbn = $book->isbn;
    $douban = json_decode(file_get_contents("http://api/douban/douban/$isbn"))->out;

    if ($douban->summary == ' (豆瓣找不到)')
    {
        $logger->addError('豆瓣找不到ISBN ' . $book->isbn . ' 的书：' . $book->title);
    }
    else
    {
        $logger->addInfo('豆瓣找到ISBN ' . $book->isbn . ' 的书：' . $book->title);
    }

    $lvt = json_decode(file_get_contents("http://api/book/lastvisit/" . $book->bookid));
    $session = $request->getSession();
    $count = $session->get('addvc', 1);
    $session->remove('addvc');

    $vc = json_decode(file_get_contents("http://api/book/visitCount/" . $book->id));
    return $this->render("AppBundle:book:detail.html.twig", array('book' => $book, 'douban' => $douban, 'lvt' => $lvt, 'count' => $count));
}
```

注意到我们此时用的是传递变量到模板的方式。但是方法本身全部采用RESTful API调用的方式。

获得相关评论

基于我们的样本数据，这本书应该有2个评论文章。所以我们还要获取相应的评论并显示。该段代码比较简单，请读者自行完成。

小结

至此，我们已经基本完成书籍详情页面的编写。

请读者认真学习本段，因为本段牵涉到的代码量是非常大的。

书籍列表页面

在这个页面，我们会分页显示书籍列表页面。同时我们也会看到，SF中一个模板是可以被多个控制器复用的。当然，我们还要讨论如何编写分页。

分页模块

首先我们看看分页模块的编写。

在 `src` 目录下创建一个 `lib` 目录，并创建一个 `Paginator.php`：

```
namespace lib;

class Paginator
{
    private $totalPages;
    private $page;
    private $rpp;

    public function __construct($page, $totalcount, $rpp)
    {
        $this->rpp=$rpp;
        $this->page=$page;

        $this->totalPages=$this->setTotalPages($totalcount, $rpp);
    }

    /*
     * var recCount: the total count of records
     * var $rpp: the record per page
     */

    private function setTotalPages($totalcount, $rpp)
    {
        if ($rpp == 0)
        {
            $rpp = 20; //This is forced in this. Need to get parameter from configuration
        }

        $this->totalPages=ceil($totalcount / $rpp);
        return $this->totalPages;
    }

    public function getTotalPages()
    {
        return $this->totalPages;
```

```

    }

    public function getPagesList()
    {
        $pageCount = 5;
        if ($this->totalPages <= $pageCount) //Less than total 5 pages
        {
            return [1, 2, 3, 4, 5];
        }
        if($this->page <=3)
        {
            return [1,2,3,4,5];
        }

        $i = $pageCount;
        $r=array();
        $half = floor($pageCount / 2);
        if ($this->page + $half > $this->totalPages) // Close to end
        {
            while ($i >= 1)
            {
                $r[] = $this->totalPages - $i + 1;
                $i--;
            }
            return $r;
        } else
        {
            while ($i >= 1)
            {
                $r[] = $this->page - $i + $half + 1;
                $i--;
            }
            return $r;
        }
    }
}

```

这个模块只负责一件事情：根据记录总数和每页的记录数算出总页面，并根据当前页数返回一个（合理的）包含前后各2个页数及当前页数（共5个）的数组，使得调用端可以显示去往不同页面的链接。这个类的编写并不复杂，这里不再做进一步的解释。

书籍列表的路由定义

书籍列表的路由定义比较长，这是因为在设计这个路由（和对应的动作）时，我们考虑要将该路由（和对应的动作）复用。它不仅只是简单地按照登录顺序的逆序分页显示若干书籍，而且还能按照搜索方式的不同只显示符合搜索条件的若干书籍¹。

该路由定义如下：

```
book_list:
    path: /books/list/{type}/{key}/{page}
    defaults:
        page: 1
        type: title
        key: all
    _controller: AppBundle:Book:list
```

这个路由定义的路径可以解读为：按照某个类型（`type`）下的关键字（`key`）搜索并返回搜索结果中的某一页（`page`）。

书籍列表动作的编写

由于我们采用API的方式获得远程数据，所以控制器中的动作编写相对简单：

```
public function listAction($page, $key, $type)
{
    $uri="http://api/books/list/$type/$key/$page";
    $out= json_decode(file_get_contents($uri))->out;

    $res=$out->books;
    $totalcount=$out->count->bc;

    $rpp=$this->container->getParameter('books_per_page');

    $paginator = new \lib\Paginator($page, $totalcount, $rpp);
    $pagelist = $paginator->getPagesList();

    return $this->render("AppBundle:book:list.html.twig", array('res' => $res, 'paginator' => $paginator));
}
```

简单说来，我们通过API调用获得适当的数据（符合搜索条件的书籍和书籍总数），从SF全局配置文件中获得 `books_per_page` 这个参数并调用上文提到的 `Paginator` 类构造一个分页列表。最后将相应的参数（共6个）传递到模板中显示。

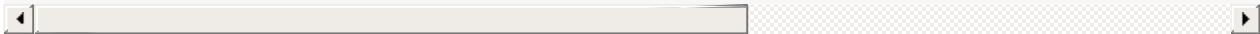
注意：`books_per_page` 参数应该在 `/app/config/parameters.yml` 中得到定义。方法是在该文件中加入一行：

```
books_per_page: 20
```

书籍列表模板

书籍列表模板比较长，这里不再列出。我们只是重点分析一下分页导航部分的代码。

```
<section id="pagination" class="col-md-12">
    <a href="{{path('book_list', {'page':1, 'key':key, 'type':type})}}" title="首页"><i class="glyphicon glyphicon-home" style="font-size: 18px;"></i></a>
    {% if cur==1 %}
        <a class="disabled" title="上一页"><i class="glyphicon glyphicon-backward" style="font-size: 18px;"></i></a>
    {% else %}
        <a href="{{path('book_list', {'page':cur-1, 'key':key, 'type':type})}}" title="上一页" style="border: 1px solid #ccc; border-radius: 5px; padding: 2px 10px; margin-right: 5px;"><i class="glyphicon glyphicon-backward" style="font-size: 18px;"></i></a>
    {% endif %}
    {% if cur==total %}
        <a class="disabled" title="下一页"><i class="glyphicon glyphicon-forward" style="font-size: 18px;"></i></a>
    {% else %}
        <a href="{{path('book_list', {'page':cur+1, 'key':key, 'type':type})}}" title="下一页" style="border: 1px solid #ccc; border-radius: 5px; padding: 2px 10px; margin-right: 5px;"><i class="glyphicon glyphicon-forward" style="font-size: 18px;"></i></a>
    {% endif %}
    <a href="{{path('book_list', {'page':total, 'key':key, 'type':type})}}" title="末页"><i class="glyphicon glyphicon-home" style="font-size: 18px;"></i></a>
</section>
```



虽然说从 `Paginator` 获得的是一个页面导航列表，但是我们选择用“前后页”（加上首页、末页）的方式显示导航按钮。也因此，我们分离了分页本身、导航、显示这三部分。这样做能提供最大程度的灵活性。

效果

至此页面效果如下：

编号	书名	作者	购买整理日期	位置
99999	Special Book Title	【Somewhere On Earth】 Special	1970年01月01日	x1
10099	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10098	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10097	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10096	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10095	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10094	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10093	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10092	Normal Book Title	【Normal】 Normal	2016年05月07日	a1
10091	Normal Book Title	【Normal】 Normal	2016年05月07日	a1

书名: 搜索:

直接去第几页:

书名: 搜索:

2016 , RSYWX

200 @ book_list 1355 ms 3.8 MB anon. 47 ms

当然，这个页面还有一些功能没有完成。比如搜索以及直接跳转页面。搜索会在后续章节讲述。直接页面跳转比较简单，请自行完成。

1. 目前只支持按照书籍标题起始字符搜索和单一TAG搜索。 ↵

书籍搜索

基于我们API的设置，该Web应用可以提供两种类型的搜索：

- 根据书名的开始部分搜索；
- 根据单一TAG进行搜索；

上一节中我们提到，书籍列表页面有两个功能没有完成，其中一个就是搜索书名。

对应的控制器方法如下：

```
public function searchAction(Request $req)
{
    $q = $req->request->all();

    $page = 1;
    $key = $q['key'];

    $uri = $this->get('router')->generate('book_list', array('page' => $page, 'key' =
    return $this->redirect($uri);
}
```

这里我们需要注意的是，在搜索栏里输入的文字缺省被认为是书名的开始部分，也就是说，我们缺省认为我们按照书名搜索。

另外，搜索的过程其实并不搜索！我们只是根据当前情景构造了一个URI而已！这就是之前我们提到的 book_list 这个路由灵活性带来的好处了。而且，在此情形下，显示书籍列表的模板也可以被复用，而且在书名搜索模式（和TAG搜索）模式下，关键字在分页时不会被丢失。这是因为在构造所有相关的URI的时候，搜索类型和关键字都是被传递的。

至此，所有重要的前端页面都已经基本描述完毕。

笔者在此鼓励读者自行完成其它页面的构建。

下一小节我们开始讲述用户和后台的编写。

用户及后台

我们在之前的章节已经看到如何开发一个WEB应用。严格的说，那只是前台的应用。

一般而言，我们总还有一个后台的应用，而后台的应用一般也会要求用户登录。

所以在本章，我们详细讲述用户和后台的开发。

用户

SF中的用户概念和常规应用中的没有什么不同。对用户的验证也有多种方法。详情可以参考[SF官方文档中的相关章节](#)。

在本应用中，我们只采用最简单的所谓Plain Authorization，而且用户及密码都以明文保存在配置文件中。

security.yml

SF所有的安全配置都在 `/app/config/security.yml` 中，我们将该文件修改为：

```

security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    providers:
        in_memory:
            memory:
                users:
                    admin: { password: 123456, roles: [ 'ROLE_ADMIN' ] }

    firewalls:
        dev:
            pattern:  ^/(_(profiler|wdt)|css|images|js)/
            security: false

        login:
            pattern:  ^/demo/secured/login$
            security: false

        secured_area:
            pattern:      ^/admin
            anonymous: ~
            http_basic:
                realm: "Secured RSYWX Admin Area"

        access_control:
            - { path: ^/admin, roles: ROLE_ADMIN }

```

我们略作解释。

- encoders 一段中，说明我们的用户验证机制是 plaintext，也就是明文的方式。
- providers 一段中，说明我们的用户是 in memory 方式，也就是说用户信息存放在内存中。
- firewalls 中，主要看 secured_area 中 pattern 的说明。^/admin 表示类似 '/admin' 这样的路径是属于受控路径，需要验证。
- access_control 中，我们规定类似 /admin 这样的目录只能由具有 ROLE_ADMIN 权限的用户访问。

这里我们略微讲一下验证和授权的区别。

验证是对一个用户是否合法的判定。常规情形下，一个用户用正确的密码登录系统后，就认为该用户已经获得验证。

授权是对一个验证用户的进一步判定。在应用配置中，该用户能做什么不能做什么是由授权来完成的。

后台

本应用的后台只是一些统计信息的显示。具体编程不再赘述，显示效果如下：

The screenshot shows a web browser window with the following details:

- Tab title: 任氏有无轩
- Address bar: Web WeChat | symfony/app_dev.php/admin
- Search bar: Search
- Toolbar icons: Back, Forward, Stop, Refresh, Home, etc.
- Content area:
 - Header: 任氏有无轩
 - Search bar: 书籍名称 (Books Name) with a green '搜索' (Search) button.
 - Text message: rsywx.net (104.224.160.168) 主机位于US, California.
截止2016年05月07日15时45分18秒，本站点藏书已经被浏览3次。
 - Section: 最近访问的20本书籍 (Recent 20 Books Accessed)
 - Special Book Title (2016-05-07 15:40:02) 已经访问了：2次
 - Normal Book Title (2016-05-07 15:11:59) 已经访问了：1次
 - Section: 访问最多的20本书籍 (Most Visited 20 Books)
 - Section: 访问最少的20本书籍 (Least Visited 20 Books)
 - Section: 最近14天访问量 (Recent 14 Days Access Volume)
 - Section: 最近7天都没有被访问的20本书籍 (Last 7 Days Unvisited Books)

2016 , RSYWX



如果配置正确的话，在访问该页面之前，浏览器会弹出一个对话框，要求输入用户/密码（我们的配置是admin/123456）。

结语

我不是一个专业的编程人员，但是很喜欢编程，还喜欢写东西。

将我自己从零开始用SF编写我的[任氏有无轩](#)应用的过程全盘写出，一是给自己的工作做个总结；二是分享出来，推广SF这个优秀的PHP框架。

限于个人水平，书中错误在所难免，恳请不吝指正。

如果有任何问题，可以与我联络。我的邮箱是taylor.ren@gmail.com。

祝编程愉快！

TR@SOE

2016.5.7