第 2 章　探索 symfony 代码

用 symfony 开发的程序乍看起来有些吓人。它包含很多目录和脚本，有 PHP 类，HTML 甚至两者的混合，程序里面有些类很难找到定义的地方，目录深达 6 层。不过一旦你了解了这些背后的原因，就会突然发现这其实是很自然的，symfony 程序的结构就应该是这样。读完本章你的这种害怕的想法就会消失。

MVC 模式

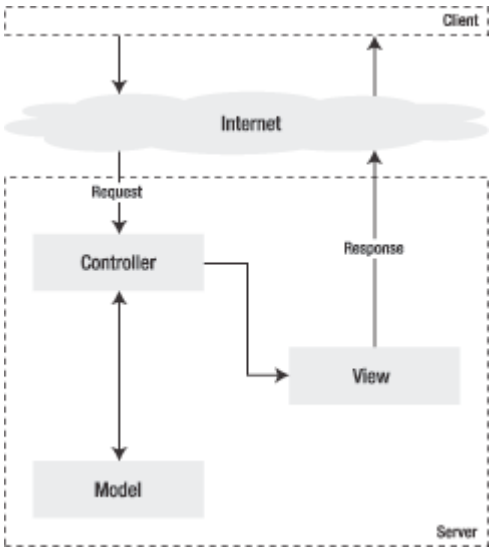
symfony 基于 MVC 架构这个经典的 Web 设计模式，MVC 架构包含三层：

- 模型(model)代表程序操作的信息--业务逻辑。
- 视图(view)将模型用网页的形式展现出来从而与用户进行交互。
- 控制器(controller)通过调用合适的模型或者视图来回应用户的动作。

图 2-1 解释了 MVC 模式。

MVC 架构把业务逻辑(模型)与展示(视图)分开，从而大大提高了可维护性。例如，如果你的程序需要能同时在标准 web 浏览器与手持设备上面运行，你只需要一个新的视图(view)，而无需改变原来的控制器(controller)与模型(model)。控制器(controller)把请求(request)的协议(HTTP，命令行模式，邮件等)与模型和视图分开来。模型抽象化逻辑与数据，从而独立于视图与动作(action)，例如，程序使用的数据库类型。

图 2-1 - MVC 模式



MVC层次

为了帮助你理解MVC的好处，让我们看看如何将一个基本的PHP程序转换成一个MVC架构的程序。这里我们用一个显示blog文章的程序作例子。

单一文件（平面的）编程

从数据库里面显示数据的一般写法跟例2-1类似

例 2-1 - **单一文件**脚本

```
<?php

// 连接，选择数据库
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

// 执行SQL查询
$result = mysql_query('SELECT date, title FROM post', $link);

?>

<html>
  <head>
    <title>List of Posts</title>
  </head>
  <body>
   <h1>List of Posts</h1>
   <table>
     <tr><th>Date</th><th>Title</th></tr>
<?php
// 用HTML显示结果
while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
echo "\t<tr>\n";
printf("\t\t<td> %s </td>\n", $row['date']);
printf("\t\t<td> %s </td>\n", $row['title']);
echo "\t</tr>\n";
}
?>
    </table>
  </body>
</html>
```

```
<?php

// 关闭连接
mysql_close($link);

?>
```

这样的代码写起来很快，执行也快，但是几乎没法维护。下面是这种代码的主要问题：

- 没有错误检查(如果数据库连接失败怎么办？)
- HTML 与 PHP 代码混合在一起，甚至是在 PHP 代码里输出 HTML 标签。
- 只能适用于 MySQL 数据库。

分离显示

例 2-1 中的'echo'与'printf'使代码难以阅读。如果要修改 HTML 代码来改进外观的话就需要改动 PHP 代码。因此代码应该分割成两部分。首先，把纯粹的包含了所有业务逻辑的 PHP 代码放在一个控制器(controller)脚本里，见例 2-2。

例 2-2 - index.php 控制器(controller)部分

```
<?php

// 连接，选择数据库
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);

// 执行 SQL 查询
$result = mysql_query('SELECT date, title FROM post', $link);

// 准备显示用得数组
$posts = array();
while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
{
   $posts[] = $row;
}

// 关闭连接
mysql_close($link);

// 载入显示部分
require('view.php');
```

```
    ?>
```

HTML 代码，包括一些类似模板的 PHP 语法，存放在一个显示脚本里，见例 2-3。

例 2-3 - view.php 显示部分

```
<html>
  <head>
    <title>List of Posts</title>
  </head>
  <body>
    <h1>List of Posts</h1>
    <table>
      <tr><th>Date</th><th>Title</th></tr>
    <?php foreach ($posts as $post): ?>
      <tr>
        <td><?php echo $post['date'] ?></td>
        <td><?php echo $post['title'] ?></td>
      </tr>
    <?php endforeach; ?>
    </table>
  </body>
</html>
```

按照经验来说视图是否足够干净取决于它是否仅包括最少的 PHP 代码，使得没有 PHP 知识 HTML 设计师能够理解。视图里最常用的语句是 echo, if/endif, foreach/endforeach。另外，不应用 PHP 代码输出 HTML 标签。

所有的逻辑都移到了控制器(controller)脚本，并且仅包含纯 PHP 代码，没有 HTML。事实上，你可以想象同样的控制器可以有完全不同的表现，例如 PDF 文件或者 XML 结构。

分离数据处理

大部分控制器(controller)脚本代码专注于数据处理。但是假如你需要另一个显示文章列表的控制器，例如输出 blog 文章的 RSS 种子的控制器呢？如果你想把所有的数据库查询放在一个地方，避免代码重复呢？如果你决定改变数据模型因为'post'表名改成了'weblog_post'呢？如果你想把数据库从 MySQL 换成 PostgreSQL 呢？为了让上面这些假设成为现实，你需要把数据处理代码从控制器里面去掉并把它们放在另外的脚本里面，我们称之为模型，如例 2-4 所示。

例 2-4 - model.php 模型部分

```php
<?php

function getAllPosts()
{
  // 连接数据库
  $link = mysql_connect('localhost', 'myuser', 'mypassword');
  mysql_select_db('blog_db', $link);

  // 执行 SQL 查询
  $result = mysql_query('SELECT date, title FROM post', $link);

  // 填充数组
  $posts = array();
  while ($row = mysql_fetch_array($result, MYSQL_ASSOC))
  {
     $posts[] = $row;
  }

  // 关闭连接
  mysql_close($link);

  return $posts;
}

?>
```

修改过的控制器如例 2-5 所示。

例 2-5 - index.php 修改过的控制器

```php
<?php

// Requiring the model
require_once('model.php');

// Retrieving the list of posts
$posts = getAllPosts();

// Requiring the view
require('view.php');

?>
```

这样控制器的可读性变强了。它唯一的任务是从模型中取得数据然后传给视图。在更复杂的程序里，控制器还要处理请求、用户 session、身份验证等。控制器中使用了直观地函数名使得我们不用注释就能读懂。

模型脚本将专注于数据访问的内容组织在一起。所有与数据层无关的参数(例如请求参数)必须由控制器提供而不能直接被模型访问到。模型函数可以方便的在另一个控制器里面重用。

MVC 以外的分离方式

MVC 架构的原理是把代码根据类型分成三层。数据逻辑代码放在模型里，表现代码放在视图里，应用逻辑代码放在控制器里。

还有其它的设计模式甚至可以使编写代码变得更加容易。模型，视图，控制器层还可以进一步细分。

数据库抽象

模型层可以分成数据访问层与数据库抽象层。这样，数据访问函数不使用与数据库有关的查询语句，由其它的函数执行。如果换数据库系统，只需要修改数据库抽象层。

例 2-6 是 MySQL 的数据库抽象层的例子，随后的例 2-7 是一个简单的数据访问层。

例 2-6 - 模型的数据库抽象层部分

```
<?php

function open_connection($host, $user, $password)
{
  return mysql_connect($host, $user, $password);
}

function close_connection($link)
{
  mysql_close($link);
}

function query_database($query, $database, $link)
{
  mysql_select_db($database, $link);
```

```
  return mysql_query($query, $link);
}

function fetch_results($result)
{
  return mysql_fetch_array($result, MYSQL_ASSOC);
}
```

例 2-7 - 模型的数据访问层

```
function getAllPosts()
{
  // 连接数据库
  $link = open_connection('localhost', 'myuser', 'mypassword');

  // 执行 SQL 查询
  $result = query_database('SELECT date, title FROM post', 'blog_db',
$link);

  // 填充数组
  $posts = array();
  while ($row = fetch_results($result))
  {
     $posts[] = $row;
  }

  // 关闭连接
  close_connection($link);

  return $posts;
}

?>
```

可以看到数据访问层的部分没有数据库引擎有关的函数，从而不依赖于特定的数据库。另外，建立数据库抽象层的函数可以在很多其它的模型函数中重用。

NOTE 例 2-6 与例 2-7 的例子并不十分让人满意，要完成一个完整的数据库抽象层还有很多事情要做(通过数据库无关的查询生成器抽象 SQL 代码，把所有的函数放到一个类，等等)。但是这本书的目的不是手把手教你怎么写一个数据库抽象层，在第 8 章里你会看到 symfony 本身是如何把这些抽象做好的。

视图元素

视图层也可以通过分离代码来优化。应用程序中的网页往往会包含一些固定的元素：页头，图形版面设计，页脚以及全局导航。只有网页的中间部分变化。所以我们把视图分成布局(layout)与模板。布局(layout)一般是整个程序通用的，或者一组页面公用。模板只负责把控制器的变量显示出来。我们需要一些逻辑使这些零件(components)和在一起能够起作用，这就是视图逻辑。根据这些原则，例 2-3 的视图部分可以分成 3 部分，如例 2-8,2-9,2-10 所示。

例 2-8 - mytemplate.php 视图的模板部分

```
<h1>List of Posts</h1>
<table>
<tr><th>Date</th><th>Title</th></tr>
<?php foreach ($posts as $post): ?>
  <tr>
    <td><?php echo $post['date'] ?></td>
    <td><?php echo $post['title'] ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

例 2-9 - 视图的视图逻辑部分

```
<?php

$title = 'List of Posts';
$content = include('mytemplate.php');

?>
```

例 2-10 - 视图的布局部分

```
<html>
  <head>
    <title><?php echo $title ?></title>
  </head>
  <body>
    <?php echo $content ?>
  </body>
</html>
```

动作与前端控制器

在上一个例子里，控制器(controller)并没有作太多事情，但是在真正的 web 应用程序里面，控制器要做很多事情。这些事情中的一些重要部分对于所有的控

制器都要做。这些事情包括处理请求、安全处理、载入应用程序配置信息，以及一些杂事。所以控制器经常被分成整个应用程序唯一的前端控制器和只负责某个特定页面的动作。

前端控制器的一个很大的好处就是整个应用程序唯一的入口。如果你决定关闭应用程序，你只要修改前端控制器脚本。如果一个应用程序没有前端控制器，那就要单独的关掉每一个控制器。

面向对象

所有前面的例子都是面向过程的。现代编程语言的面向对象特性能简化编程，因为对象可以封装逻辑，继承，以及提供干净的命名规则。

在非面向对象的语言里面实现 MVC 架构会引起命名空间及代码重复的问题，代码会比较难以阅读。

开发者通过面向对象的方式可以通过视图对象，控制器对象，模型对象把之前例子里面的函数转换成方法。这是 MVC 架构必须的。

TIP 如果你想更详细的了解面向对象环境中的 web 应用程序设计模式，请阅读《Patterns of Enterprise Application Architecture》(作者:Martin Fowler, 出版 Addison-Wesley, ISBN: 0-32112-742-0)。这本书里面的代码用 Java 或者 C#写的，PHP 开发者也可以读一读。

symfony 的 MVC 实现方式

暂停一下，先来看看一个显示 blog 文章列表的页面，有多少部分组成？如图 2-2 所示，由下面的部分组成：

- 模型层
    - 数据库抽象
    - 数据访问
- 视图层
    - 视图
    - 模板
    - 布局
- 控制器层

- 前端控制器
- 动作

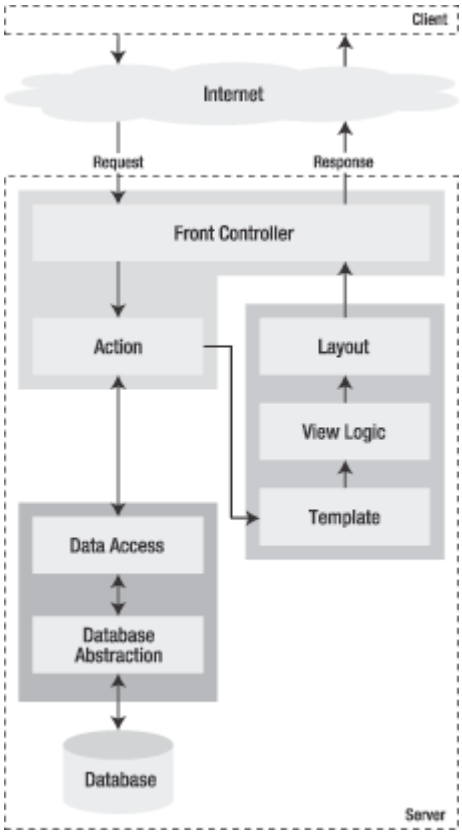
7 个脚本--每次修改一个页面需要打开这么多文件！可是，symfony 做了些简化。虽然使用最好的 MVC 架构，但是 symfony 的方式使得开发程序更加快速容易。

首先，前端控制器是应用程序里所有动作共用的。可以有多个控制器与多个布局，但是只需要一个前端控制器。前端控制器是纯 MVC 逻辑组件，你不必自己写一个，因为 symfony 会为你生成一个。

另外一个好消息是模型层的类也是根据数据结构自动生成的。这是由 Propel 库完成的，它有类的构架与代码生成功能。如果 Propel 找到外键或者日期字段，它会生成特殊的存取方法，这使得数据处理非常容易。另外，数据库抽象也是完全看不见的，它是由另外一个 Creole 组件处理的。所以如果你决定更换数据库引擎，你不必重写代码。你只要修改配置参数就可以了。

最后一件事情，视图逻辑可以很容易的转换成一个配置文件，不需要编写程序。

图 2-2 - symfony 工作流程



这就是说在 symfony 里面显示文章的例子需要 3 个文件，如例 2-11，2-12，2-13 所示。

例 2-11 - list 动作，
myproject/apps/myapp/modules/weblog/actions/actions.class.php

```
<?php
class weblogActions extends sfActions
{
  public function executeList()
  {
    $this->posts = PostPeer::doSelect(new Criteria());
  }
}

?>
```

例 2-12 - list 模板，
myproject/apps/myapp/modules/weblog/templates/listSuccess.php

```
<h1>List of Posts</h1>
<table>
<tr><th>Date</th><th>Title</th></tr>
<?php foreach ($posts as $post): ?>
  <tr>
    <td><?php echo $post->getDate() ?></td>
    <td><?php echo $post->getTitle() ?></td>
  </tr>
<?php endforeach; ?>
</table>
```

例 2-13 - list 视图，
myproject/apps/myapp/modules/weblog/config/view.yml

```
listSuccess:
  metas: { title: List of Posts }
```

另外，你需要定义一个布局，如例 2-14，但是它可以多次重用。

例 2-14 - 布局 myproject/apps/myapp/templates/layout.php

```
<html>
  <head>
    <?php echo include_title() ?>
```

```
  </head>
  <body>
    <?php echo $sf_data->getRaw('sf_content') ?>
  </body>
</html>
```

这些就是全部的了。你只需要这些代码来显示与例 2-1 完全一样的页面。余下的事情(使所有的组成部分共同工作)由 symfony 来处理。如果你计算行数，会发现用 MVC 架构的 symfony 来实现显示文章列表花的时间和编写的代码不比写一个普通脚本要多。不过，这样做的巨大好处是，代码组织得十分清楚，可重用，灵活性还有更多的乐趣。作为奖励，你会得到 XHTML 兼容性，调试能力，简单的配置，数据库抽象，智能 URL 定向，多种环境，还有很多开发工具。

symfony 核心类

在本书里你会经常碰到 symfony 的 MVC 核心的几个类：

- sfController 控制器类。它解析请求并交给动作处理。
- sfRequest 保存所有的请求元素(参数，cookie，请求的头 等)。
- sfResponse 包含回应的头和内容。它的内容最终会转化为 HTML 传给用户。

- context singleton (由 sfContext::getInstance()取得) 保存所有核心对象还有当前的配置的引用，它可以从任何地方访问到。

在第 6 章你会了解到更多这些对象的信息。

如你所见，所有的 symfon 类都有一个 sf 前缀，很多 symfony 模板中的核心变量也是这样。这样可以避免与你的类名与变量名重复，并使框架核心类更像是一家人，更好辨认。

NOTE 在 symfony 的编码规范中，开头字母大写的驼峰字(UpperCamelCase)是变量名与类名的标准。只有两个例外：核心 symfony 类以小写的 sf 开头,模板里面的变量使用小写下划线的方式。

代码组织

现在你了解了 symfony 应用程序的各个组成部分，你可能会想知道它们是怎么组织的。symfon 按照项目组织代码，项目文件放在标准的树结构里。

项目结构：应用程序、模块与动作

一个 symfony 项目由一个域名下的服务与操作组成，它们共享同样的对象模型。

在一个项目里，操作按照逻辑划分成不同的应用程序。同一个项目里面的不同应用程序相互独立。大多数情况，一个项目会包含两个应用程序:一个是前台，一个后台，它们共享同一个数据库。不过一个项目也可以包含很多小网站，每一个站点是一个不同的应用程序。注意应用程序间的链接必须用绝对形式。

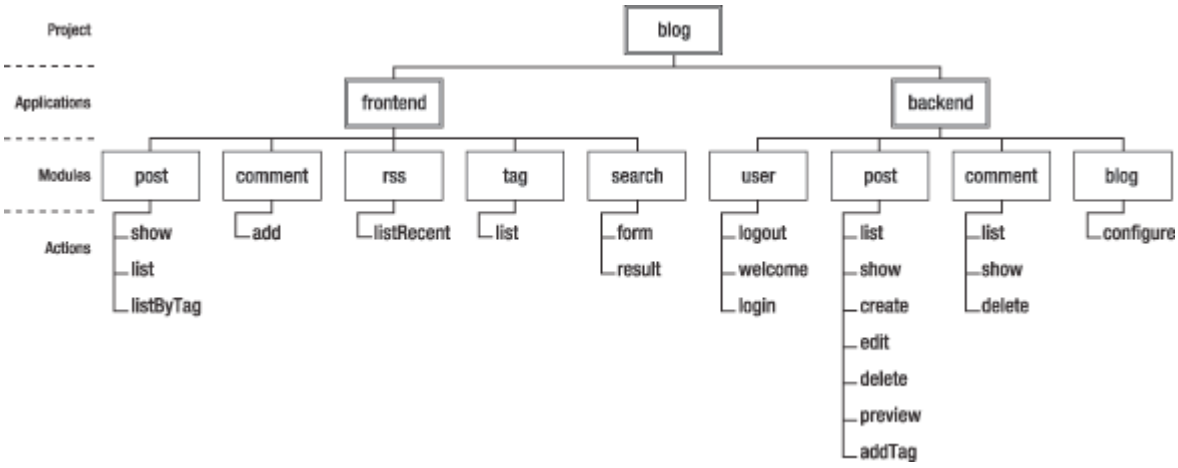
每个应用程序由一个或更多模块组成。模块就是功能相近的一个页面或者一组页面。例如，模块 home ,articles, help, shoppingCart, account 等。

模块包含动作，也就是说一个模块可以包含多个动作。例如，shoppingCart 模块也许会有 add,show 与 update 等动作。一般来说，动作的名字是动词。动作就好像一般的 web 应用程序的页面一样，尽管两个动作可能显示同样的页面(例如，在给文章留言后还会把文章显示出来)。

TIP 如果你认为这么做对于一个刚开始的项目来说层次太多了，你可以很方便的把所有的动作集中到一个模块里，这样文件结构就简单了。当应用程序越来越复杂，你就需要把这些动作分开放到不同的模块。本书第 1 章提到，通过重写代码来改善结构与可读性(同样保留功能)被称为重构，当你应用 RAD 原则的时候经常需要这么做。

图 2-3 是一个 blog 项目的代码组织结构图，按照项目/应用程序/模块/动作来划分。 但注意项目的实际文件结构可能会与图里面的不一样。

图 2-3 - 代码组织结构例子



目录结构

所有的 web 项目都有这些内容:

- 一个数据库，例如 MySQL 或者 PostgreSQL
- 静态文件(HTML, 图片， JavaScript 文件，样式表等)
- 网站管理员与用户上传的文件

- PHP 类与函数库
- 外部库(第三方脚本)
- 批处理文件 (用于命令行或者 cron 的脚本)
- 日志文件 (应用程序或者服务器的留下的脚印)
- 配置文件

symfony 用一种合理的目录结构组织所有这些内容，这种树形结与 symfony 的架构(MVC 模式与应用程序/项目/模块分组)相符合。这个目录结构是在项目，应用程序，模块初始化的时候自动生成的。当然，为了满足客户的需求你可以完全自定义这个结构。

根目录结构

下面是一个 symfony 项目根目录下的文件：

```
apps/
  frontend/
  backend/
batch/
cache/
config/
data/
  sql/
doc/
lib/
  model/
log/
plugins/
test/
  unit/
  functional/
web/
  css/
  images/
  js/
  uploads/
```

表 2-1 介绍了这些目录的内容。

表 2-1 - 根目录

目录	描述
apps/	包含此项目内所有应用程序(一般情况, frontend 与 backend 分别代表前台与后台)。

目录	描述
batch/	包含命令行下运行的PHP脚本或者定期执行的脚本。
cache/	包含了配置文件的缓存，如果你开了动作和模板，还有这两个部分的缓存。缓存机制(详见第12章)把这些信息存在文件里面加快响应web请求的速度。每个应用程序都会有一个子目录，包含了预处理的PHP与HTML文件。
config/	存放项目的配置信息。
data/	这里可以存放项目的数据文件，例如数据库schema，包含了建立数据表的SQL文件，或者一个SQLite数据库文件。
doc/	存放项目文档，包括你自己的文档和PHPdoc生成的文档。
lib/	主要用来存放外部类或者库。这里的内容整个项目都能访问到。model/子目录存放项目的对象模型(详见第8章)。
log/	存放symfony生成的应用程序的日志文件。也可以放web服务器的日志文件，数据库日志文件，或者项目的任何地方的日志文件。symfony自动为项目的每一个应用程序的每一个环境生成一个日志文件(日志文件详见第16章)。
plugins/	存放安装在项目里的插件(插件详见第17章)。
test/	包含PHP写的与symfony测试框架兼容的单元与功能测试(详见第15章)。 项目初始化的时候，symfony会自动建立一些基本的测试。
web/	web服务器的根目录。所有从因特网能够直接访问的文件都在这个目录里。

应用程序目录结构

所有应用程序的目录结构都是一样的：

```
apps/
  [应用程序名]/
    config/
    i18n/
    lib/
    modules/
    templates/
      layout.php
      error.php
      error.txt
```

表 2-2 介绍了应用程序的子目录。

表 2-2 - 应用程序的子目录

目录	描述
config/	包含一些YAML格式的配置文件。大部分应用程序的配置信息都在这

目录	描述
	里，symfony 框架自己的默认配置除外。 注意需要的话默认值可以修改。详见第 5 章。
i18n/	包含应用程序的国际化文件--大部分的界面翻译文件(详见第 13 章)。如果你用数据库存放翻译信息可以忽略这个目录。
lib/	包含应用程序用到的类与库。
modules/	存放应用程序的所有功能模块。
templates/	包含应用程序的全局模板--所有模块公用的模板。默认情况，这个目录会有一个 layout.php 文件，这是模块默认的主布局模板。

NOTE 新应用程序的 i18n/, lib/, 与 modules/ 目录是空的。

一个应用程序的类的方法或属性不能被同一个项目的其他应用程序访问到。另外，同一项目的两个应用程序之间的超链接必须用绝对形式。开始把项目分成不同的应用程序的时候，这个限制就存在了。

模块目录结构

每个应用程序包括一个或更多的模块。在 modules 目录中每个模块都有它自己的子目录，这个目录的名字是模块初始化的时候确定的。

下面是一个典型的模块目录结构：

```
apps/
  [应用程序名]/
    modules/
      [模块名]/
          actions/
            actions.class.php
          config/
          lib/
          templates/
            indexSuccess.php
          validate/
```

表 2-3 介绍了模块子目录结构。

表 2-3 - 模块子目录

目录	描述
actions/	一般只有一个文件 actions.class.php，这个文件里面包含了模块的所有动作。模块的不同动作也可以分开写在不同的文件里。
config/	可以存放模块的配置信息。

目录	描述
lib/	存放模块的类与库。
templates/	存放模块里所有动作的模板。模块初始化的时候，会建立一个默认模板 indexSuccess.php。
validate/	用户存放表单验证配置信息(详见第 10 章)。

NOTE 新模块的 config/, lib/, 与 validate/ 目录是空的。

web 目录结构

web 目录的限制很少，这里存放的是互联网可以访问得到的文件。模板的默认行为还有 helper 里包含了几个基本的命名规则。下面是一个 web 目录的结构的例子：

```
web/
  css/
  images/
  js/
  uploads/
```

表 2-4 介绍了 web 目录的内容。

表 2-4 - 典型的 web 目录的子目录

目录	描述
css/	存放.css 结尾的样式表文件。
images/	存放.jpg、.png 与.gif 扩展名的图片文件。
js/	存放.js 扩展名的 JavaScript 文件。
uploads/	只能存放用户上传的文件。虽然这个目录通常会存放图片我们还是把这个目录与图片目录分开，这样同步开发服务器与正式服务器的时候不会影响上传的文件。

NOTE 虽然强烈建议维持默认的目录结构，但是你还是可以进行修改，例如一个项目要运行在不同的目录结构与命名规则的服务器上。修改目录结构详见第 19 章。

常用工具

有些技巧在 symfony 里面很常用，在项目中你会经常碰到他们。这包括参数存储器，常量，还有类自动加载。

参数存储器

很多 symfony 类都包含一个参数存储器。参数存储器用简便的方式封装了获取方法与设置方法。例如，sfResponse 类包含了一个可以通过执行 getParameterHolder()方法获得参数存储器。每一个参数存储器都用同样的方式存取数据，如例 2-15 所示。

例 2-15 - 使用 sfResponse 参数存储器

```
$response->getParameterHolder()->set('foo', 'bar');
echo $response->getParameterHolder()->get('foo');
 => 'bar'
```

大部分类通过使用参数存储器的 proxy 方法来减少 get/set 操作的代码量。下面是 sfResponse 对象的例子，例 2-16 可以达到例 2-15 同样效果。

例 2-16 - 使用参数存储器的 proxy 方法

```
$response->setParameter('foo', 'bar');
echo $response->getParameter('foo');
 => 'bar'
```

参数存储器的 getter 方法可以有第二个参数作为默认值。这样在取值失败的时候比较简洁。见例 2-17。

例 2-17 - 使用参数存储器的 get 方法的默认值

```
// 'foobar' 参数没有定义, 所以 getter 返回空值
echo $response->getParameter('foobar');
 => null

// 利用条件判断给一个默认值
if ($response->hasParameter('foobar'))
{
  echo $response->getParameter('foobar');
}
else
{
  echo 'default';
}
 => default

// 但是使用第二个默认值参数要快的多
echo $response->getParameter('foobar', 'default');
 => default
```

参数存储器还支持命名空间。如果你给设置方法或者获取方法指定第三个参数，这个参数代表命名空间，那么这个参数就只会在这个命名空间里定义或者取值。见例 2-18

例 2-18 - sfResponse 参数存储器的命名空间

```
$response->setParameter('foo', 'bar1');
$response->setParameter('foo', 'bar2', 'my/name/space');
echo $response->getParameter('foo');
 => 'bar1'
echo $response->getParameter('foo', null, 'my/name/space');
 => 'bar2'
```

当然，你还可以给你自己的类增加参数存储器来获得这些好处。例 2-19 告诉我们如何定义一个有参数存储器的类。

例 2-19 - 给类增加参数存储器

```
class MyClass
{
  protected $parameter_holder = null;

  public function initialize ($parameters = array())
  {
    $this->parameter_holder = new sfParameterHolder();
    $this->parameter_holder->add($parameters);
  }

  public function getParameterHolder()
  {
    return $this->parameter_holder;
  }
}
```

常量

symfony 里的常量少得出奇。这是因为 PHP 的一大缺点：常量定义后就不能改变了。所以 symfony 使用自己的配置对象，称作 sfConfig，用来取代常量。它提供了在任何地方存取参数的静态方法。例 2-20 演示了 sfConfig 类的方法。

例 2-20 - 使用 sfConfig 类方法取代常量

```
// PHP 常量
```

```
define('SF_FOO', 'bar');
echo SF_FOO;
// symfony 使用 sfConfig 对象
sfConfig::set('sf_foo', 'bar');
echo sfConfig::get('sf_foo');
```

sfConfig 方法支持默认值，并且 sfConfig::set()方法可以多次调用来设置同一个参数的值。第 5 章详细讨论了 sfConfig 方法。

类自动载入

一般来说，当你在 PHP 中要用一个类来创建一个对象的时候，你需要首先包含这个类的定义。

```
include 'classes/MyClass.php';
$myObject = new MyClass();
```

但是大的项目包含了很深的目录结构，包含所有这些文件还有路径很浪费时间。由于有__autoload()函数(或者 spl_autoload_register()函数)，symfony 使得我们不需要写包含语句，你可以直接这么写：

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

symfony 会在项目的 lib 目录里的所有 php 文件里寻找 MyClass 的定义。如果找到，就自动包含它。

所以你可以把所有的类放在 lib 目录，你再也不必包含他们。所以 symfony 项目通常没有 include 或者 require 语句。

NOTE 为了提高效率，第一次 symfony 自动在一个目录列表(在配置文件里面定义)里寻找。然后 symfony 把这些目录里的所有类和文件的关联存放在一个 PHP 数组里。这样，以后的自动载入就不需要扫描整个目录了。所以你每次在项目里面增加一个类都需要通过 symfony clear-cache 命令清空 symfony 缓存。缓存详见第 12 章，自动载入配置文件详见第 19 章。

总结

使用 MVC 框架迫使你按照框架的规定把代码分开。显示的代码归到视图里，数据处理的代码归到模型，请求处理逻辑归到控制器。这对 MVC 模式的应用程序很有用，也是一个约束。

symfony 是一个 PHP5 写的 MVC 框架。它的结构充分发挥了 MVC 模式的好处，但也非常容易使用。这要感谢他的全面性与可配置性。

现在你已经了解了 symfony 背后的原理，差不多该是开发你的第一个应用程序的时候了。但是在这之前，你需要在你的开发服务器上安装一套 symfony 并运行起来。