

第 1 章 symfony 介绍

symfony 能做什么？使用 symfony 需要掌握哪些知识？这一章会解答这些问题。

symfony 简介

开发框架可以自动化地完成一些特定的开发模式来简化应用程序开发过程。开发框架还使程序代码结构更好，通过开发框架开发人员可以写出更好、更易读的、更容易维护的代码。总之，开发框架简化了编程，因为它把复杂的操作封装成了简单的语句。

symfony 是一个完整的 web 应用开发框架，它为加速开发提供了几个关键功能。首先，它把 web 应用的业务规则，服务逻辑还有表现页面分割开来。它为减少开发复杂 web 应用提供了大量的工具和类。另外，它把一些常用的任务变成了自动化的方式从而使开发人员能够完全专注于每个程序的独特的地方。这三个优点综合起来就意味着每次开发新的 web 应用的时候不用重新发明轮子了！

symfony 完全由 PHP 5 编写。它经过了广泛的测试并被用于很多实际的项目中，一些高访问量的电子商务网站正在使用 symfony。symfony 与包括 MySQL、PostgreSQL、Oracle 还有 Microsoft SQL Server 在内的大多数数据库系统兼容。symfony 能够在 *nix 与 Windows 平台上运行。我们现在来看一下 symfony 具体有哪些功能。

symfony 的功能

symfony 是为了满足下面的需求而开发的：

- 在大多数平台上都能够很容易安装（保证能运行在标准 *nix 和 Windows 平台）
- 不依赖某种特定的数据库
- 多数情况下容易使用，并且有足够的弹性来实现更复杂的功能
- 遵循惯例重于配置原则--开发人员只需要配置与惯例不同的部分
- 能够适应大多数 web 最佳实践与设计模式
- 可供企业使用--能够适应现有的 IT (information technology) 策略与体系，并且很稳定适合长期项目
- 非常易读的代码、带有 phpDocumentor 注释，维护轻松
- 易于扩展，允许与非 symfony 的库进行整合

自动化 Web 项目功能

symfony 能够自动化 web 开发的大部分要素：

- 内建的国际化层能够实现数据与界面翻译，还有内容本地化。
- 表现部分由于使用了模板和布局，不懂任何框架知识的网页设计师也可以掌握。辅助函数封装了大量的代码从而减少了表现部分的代码。
- 表单支持自动验证和重新提交，这确保了数据库里的数据的质量，用户体验也更好。
- 输出转义能够保护程序不受到利用错误数据的攻击。
- 缓存管理功能减少了网络带宽和服务器负载消耗。
- 使用验证与证书功能能很容易地实现受限制区域还有用户权限管理。
- 路由与漂亮的 URL 使网页的网址容易被搜索引擎接受。
- 内建的 e-mail 与 API 管理使 web 程序超越传统的浏览器交互。
- 方便的数据列表提供了自动的分页、排序还有筛选功能。
- Factories, plug-ins, 还有 mixins 提供了高层次的扩展性。
- 容易使用的 Ajax 交互，以一行代码(辅助函数 helper)去封装跨浏览器支持的 Javascript

开发环境与工具

symfony 可以完全的定制以满足有自己代码规范与项目管理规则的企业。 它自带了一些开发环境还有多种工具来实现自动化软件工程任务：

- 代码生成工具非常适合于原形设计与一键式后台管理。
- 内建的单元测试与功能测试框架为测试驱动的开发提供了完美的工具。
- 测试工具条能够把当前页面的所有开发者需要的信息显示出来从而加快调试速度。
- 能够实现两台服务器之间自动化部署的命令行工具。
- 能够有效地即时更新配置。
- 日志功能让管理员能够掌握程序的一举一动。

谁开发了 symfony？为什么要开发 symfony？

symfony 的第一个版本由项目发起者同时也是本书的合著者 Fabien Potencier 于 2005 年 10 月发布。Fabien 是 Sensio (<http://www.sensio.com/>) 的 CEO，Sensio 是一个法国对 web 开发有创新见解的知名网络机构。

早在 2003 年，Fabien 研究了当时的 PHP 开源开发工具，发现没有一个能够满足上面的这些需求。PHP 5 发布的时候，他发现现有的工具已经都很成熟，很难整合到一个全功能的框架里。随后他花了 1 年的时间开发了 symfony 的核心，这个核心基于 Mojavi MVC (Model-View-Controller) 框架，Propel ORM (object-relational mapping)，还有 Ruby on Rails 的模板辅助方法。

Fabien 最初是为了 Sensio 的项目开发了 symfony，因为一个高效的开发框架能大大加快开发效率。它使得开发更直观，开发的程序更健壮更容易维护。这个框架最初在一个女性内衣的零售商的电子商务网站的项目中应用并取得了成功，随后被用到了其他的项目中。

在成功地将 symfony 运用于几个项目之后，Fabi en 决定用开源的协议来发布 symfony。他之所以这么做是为了把自己的工作成果捐献给社区，并且收集用户的意见来改进框架，展示 Sensi o 的经验，另外这样做本身也是一种乐趣。

NOTE 为什么是"symfony"而不是"FooBarFramework"呢？因为 Fabi en 想要一个简单的名字，包含代表 Sensi o 的"s"还有代表 framework 的"f"，并且容易记忆，不跟其他的开发工具名字相似。另外，他不喜欢大写字母。尽管不是标准的英语，"symfony"成为了 Fabi en 的选择，并且 symfony 也是这个项目的名称。另外一个项目名称是"baguette"。

要成为一个成功的开源项目，symfony 需要大量的英文文档，这样能使更多的开发者使用 symfony。Fabi en 请 Sensi o 的职员 François Zani notto(本书的作者)深入源代码来写电子版的手册。写作花了不少时间，但是当项目公开的时候，它优秀的文档吸引了不少开发者。其余的就什么也不用说了。

symfony 社区

当 symfony 的网站(<http://www.symfony-project.com/>)推出后，全世界许许多多的开发者下载、安装、阅读了在线文档，随后去开发了他们的第一个 symfony 的程序，社区开始热闹起来。

当时 web 应用程序开发框架正在变得流行，开发者们对全功能的 PHP 开发框架的需求很高。symfony 由于代码质量还有文档的数量成为一个引人注目的解决方案，这也是它相对于其他框架的优势。很快就有志愿者参与进来，提出修改或增强的意见，校对文档，并参与到其他一些必要的工作中来。

我们欢迎所有愿意参与的人加入 symfony 开发，我们提供公开的源代码仓库还有 ticket 系统。目前 symfony 的代码主干主要还是 Fabi en 在维护，这保证了代码的质量。

目前，symfony 的论坛，邮件列表，还有 IRC 频道为整个社区提供了很优秀的支持，平均每个问题会有 4 个回复之多。每天都有人安装 symfony，wiki 与代码片段部分汇集了大量的用户提交的文档。每个星期平均会出现 5 个新的 symfony 项目，这个数字还在不断地增加。

symfony 社区是这个框架的第三个力量，我们希望读了本书后你能加入到这个社区中来。

symfony 适合我吗？

不论你是 PHP 5 专家或 Web 应用程序开发的新手，你都能用 symfony。决定是否使用 symfony 的主要因素是项目的大小。

如果你想开发一个只有 5 到 10 个网页的简单网站，少量的数据库访问，并且不需要保证速度或者提供文档，那么你只要用 PHP 就够了。这种情况下使用 web 开发框架并没有太大的价值，而且面向对象或者 MVC 模型会使开发速度变慢。另外，symfony 在只能以 CGI 模式下运行 PHP 的虚拟主机环境下并不能很有效率的运作。

另一方面，如果你要开发一个更复杂的有更多业务逻辑的 web 应用程序，那么单独使用 PHP 是不够的。如果你打算以后维护或扩展你的应用程序，那么你的代码需要是简洁，易读和高效的。如果你打算在用户界面中方便的使用最新的技术(例如 Ajax)，你不能只是写几百行 Javascript 代码。如果你想愉快而且快速的开发，那么单独使用 PHP 可能会令你失望。上面这些，就是 symfony 适合你的原因。

当然，如果你是一个专业的 web 开发人员，你已经知道了 web 应用框架的所有优点，并且你需要一个成熟的，具有详细文档和一个大的社区支持的 web 应用框架。那么不要再犹豫，symfony 就是你的解决方案。

TIP 如果你想看比较直观的演示，建议去 symfony 的网站上的 screencast。看了以后你会发现使用 symfony 开发是多么快速而愉快。

基本概念

在开始学习 symfony 之前，有几个概念需要理解。如果你已经知道 OOP、ORM、RAD、DRY、KISS、TDD、YAML 还有 PEAR 你可以跳过本节。

PHP 5

symfony 本身是由 PHP 5 (<http://www.php.net/>) 开发的，symfony 专注于利用 PHP 5 制作网络应用程序。所以，想要彻底地了解 symfony 框架必须要对 PHP 5 有十分深刻的理解。

建议熟悉 PHP 4 但刚刚接触或者不熟悉 PHP 5 的开发者专注于 PHP 5 语言的新的面向对象模型部分。

面向对象程序设计 (OOP)

我们在本章不去详细解释面向对象程序设计(OOP)，因为这个话题足够写一本书了。由于 symfony 大量运用了 PHP 5 中的面向对象机制，面向对象程序设计(OOP)是学习 symfony 的先决条件。

维基百科这样解释 OOP:

面向对象程序设计(OOP)可以被视作一种在程序中包含各种独立而又互相调用的单位和对象的思想,这与传统的思想刚好相反:传统的面向过程程序设计主张将程序看作一系列函数的集合,或者直接就是一系列对电脑下达的指令。

PHP 5 实现了面向对象中的类、对象、方法、继承等。如果你对这些概念不熟悉,建议阅读相关的 PHP 文档,网址如下

<http://www.php.net/manual/zh/language.oop5.basics.php>。

魔术方法 (Magic Methods)

PHP 对象的一个优势是可以使用魔术方法。这些方法可以不需要修改外部代码而重写一个类的默认行为。这使得 PHP 语法有更少的冗余性和更具有扩展性。这些方法很好识别,他们都是以双下划线(__)开始的。

例如, 当显示一个对象的时候, PHP 会暗中去寻找是否开发者定义过 `__toString()` 方法:

```
$myObject = new myClass();  
echo $myObject;  
// Will look for a magic method echo  
$myObject->__toString();
```

symfony 使用了魔术方法, 所以你必须完全了解这些概念。这些在 PHP 文档中有描述(<http://www.php.net/manual/zh/language.oop5.magic.php>)。

PHP 扩展与应用程序库(PEAR)

PEAR 是"一个 PHP 可重用代码的框架和发布系统"。PEAR 可以下载、安装、升级及删除 PHP 脚本。使用 PEAR 包的时候,不用为了脚本的位置或者怎么找到他们而担心,扩展命令行接口(CLI)也很容易使用。

PEAR 是一个由社区推动的 PHP 项目,官方发布的 PHP 中就包含了 PEAR。

TIP PEAR 网站, <http://pear.php.net/>, 有 PEAR 文档与分类的 PEAR 包下载。

PEAR 是最专业的安装 PHP 库的方法。建议使用 PEAR 来管理一个由多个项目共用的 symfony 框架。symfony 的插件(plugin)是一种有特殊设置的 PEAR 包。symfony 本身也可以通过 PEAR 安装。

使用 symfony 并不需要懂 PEAR 命令的语法。你只要知道 PEAR 的用途还有确定它已经安装好就行了。你可以通过在你的电脑的命令行(CLI)下输入下面的命令来检查 PEAR 是否安装:

```
> pear info pear
```

这个命令会返回安装在你的电脑上的 PEAR 的版本。

symfony 项目有自己的 PEAR 仓库(或频道)。仓库功能只有 PEAR 1.4.0 以上版本才支持，所以如果你的版本比较老就需要升级。升级 PEAR，只要在命令行输入下面的命令就可以了：

```
> pear upgrade PEAR
```

对象关系映射(ORM)

数据库是关系型数据库。PHP5 和 symfony 是面向对象的。为了用面向对象的方法访问数据库，必须用一个接口来表示对象之间的逻辑关系。这个接口就称作对象关系映射(object-relational mapping)或者 ORM。

ORM 是由能够访问数据和存放业务规则的对象组成。

对象/关系抽象层的其中一个优点是可以不用直接去访问数据库。它会使用经过优化的模型对象来访问当前的数据库。

这就意味着在项目中后期换一套数据库将是很简单的事情。想象一下当你必须为程序写一个原型的时候，客户并不能确定哪种数据库最适合他们。你可以先使用 SQLite 来开发程序，当客户决定使用 MySQL, PostgreSQL 或者 Oracle 的时候，我们只要在配置文件中稍作修改就可以正常工作了。

抽象层封装了数据逻辑。其他程序并不需要了解 SQL 的查询语句，却依旧能轻松的访问数据库。那些数据库开发专家也很清楚的知道该做些什么。

使用对象而非记录，用类而非表，还有其他益处：你能为你的表增加一些新的存取方法。例如，你有一个名叫 Client 的表，有两个字段，分别是 FirstName 和 LastName，你也许想直接获得一个完整的姓名。在面向对象中，为一个 Client 类添加一个访问方法，就像这样：

```
public function getName()
{
    return $this->getFirstName().' '.$this->getLastName();
}
```

所有重复数据访问功能和数据的业务逻辑都可以在对象中维护。例如，有一个用来生成对象的 ShoppingCart 类。我们在结帐时想获得一个总价，你可以加一个 getTotal() 方法，就像这样：

```
public function getTotal()
{
    $total = 0;
    foreach ($this->getItems() as $item)
```

```
{  
    $total += $item->getPrice() * $item->getQuantity();  
}  
return $total;  
}
```

就这好了，想象一下要写多长的 SQL 语句才能完成同样的事情！

Propel，另一个开源项目，是当前最好的基于 PHP5 的对象/关系抽象层。symfony 框架无缝集成了 Propel，所以本书大多数的数据处理描述都使用了 Propel 的语法。本书将描述如何使用 Propel 对象，但是更详细的资料可以参考 Propel 的网站(<http://propel.phpdb.org/trac/>)。

快速应用程序开发 (RAD)

开发网页程序是一件单调乏味的，慢速的事情。按照常规软件开发生存周期(类似于 Rational 统一过程里设想的)，要等到有完整的需求，绘制大量的 UML 图，产生了大量的正式文档之前准备阶段的文档才能开始开发。这是由于一般的开发速度，缺少通用性的程序语言（在能正式看到程序可用之前不知道要多少次的建立，编译，重运行），最主要的就是客户不会经常改变他们的主意。

今天，商业节奏更快了，客户总是倾向于在制作项目中经常改变他们的需求。当然，他们期望开发组能接受他们的需求并且快速更改应用程序的架构。幸运的是，使用脚本语言就像 Perl 和 PHP 会更容易的去实现这些，例如快速应用程序开发 (RAD) 和敏捷程序开发。

尽快去建立一个工作原型以便客户可以审阅并且提出问题是一个好方法。如此周而复始，在较短的开发周期发布新的功能。

有很多东西需要开发者考虑。开发者不需要去想在未来如何实现一个新功能。用最简单的方法去实现需要的功能。这就是 KISS 原则的一个很好体现：Keep It Simple, Stupid.

当需求变化或者功能增加的时候，有时需要重写代码。这就叫做重构，这经常发生在网页应用程序开发中。代码会根据需要改变存放位置。把重复的地方合并到一个地方，这就是 Don't Repeat Yourself (DRY) 原则。

当程序改变后要确保依旧能够运行，这需要一系列的自动测试来完成。如果写得好的话，单元测试将是在重构代码后检查的一个好的方法。一些开发方法学 (development methodologies) 甚至于规定在编写代码前先写测试--这就使称之为测试驱动开发(TDD)。

NOTE 关于敏捷开发还有其它一些原则和好习惯。其中一个最有效率的方法叫做极限编程 (Extreme Programming) (简称 XP)，极限编程的教材会教你如何去

快速而有效的去开发一个程序。推荐从 Kent Beck(Addison-Wesley)的极限编程系列开始学习。

symfony 非常适合进行 RAD。事实上，制作这个框架的网络公司就使用 RAD 原则进行他们的项目。这意味着学习使用 symfony 不是去学习新语言，而是去学习如何正确的反应和判断从而更有效率的开发应用程序。

symfony 项目的网站有一个步骤详细的教程，完整的介绍了如何快速进行开发。它叫做 askeet(<http://www.symfony-project.com/asket>)，推荐想了解敏捷开发的朋友阅读这个教程。

YAML

来自 YAML 官方网站 (<http://www.yaml.org/>) 的定义：YAML 是一种直观的能够被电脑识别的数据序列化格式，并且它容易被人类阅读，容易与脚本语言交互的。换种说法，YAML 是一种非常简单的类似于 XML 的数据描述语言，语法比 XML 简单很多。他在描述可以被转化成数组或者 hash 的数据是非常有用，例如：

```
$house = array(
    'family' => array(
        'name'      => 'Doe',
        'parents'   => array('John', 'Jane'),
        'children'  => array('Paul', 'Mark', 'Simone')
    ),
    'address' => array(
        'number'    => 34,
        'street'    => 'Main Street',
        'city'      => 'Nowheretown',
        'zipcode'   => '12345'
    )
);
```

解析这个 YAML 将会自动创建下面的 PHP 数组：

```
house:
  family:
    name: Doe
    parents:
      - John
      - Jane
    children:
      - Paul
      - Mark
      - Simone
  address:
    number: 34
```



```
street: Main Street
city: Nowheretown
zipcode: 12345
```

在 YAML 里面，结构通过缩进来表示，连续的项目通过减号 "-" 来表示，map 结构里面的键/值 (key/value) 对用冒号 ":" 来分隔。YAML 也有用来描述好几行相同结构的数据的缩写语法，数组用 '[]' 包括起来，hash 用 '{}' 来包括。因此，前面的这个 YAML 可以缩写成这样：

```
house:
  family: { name: Doe, parents: [John, Jane], children: [Paul, Mark, Simone] }
  address: { number: 34, street: Main Street, city: Nowheretown, zipcode: 12345 }
```

YAML 是 "Yet Another Markup Language (另一种标记语言)" 的缩写，读音 "yamel"，或者 "雅梅尔"。这种格式大约是 2001 年出现的，目前为止已经有多种语言的 YAML 解析器。

TIP YAML 格式的详细规格可以在 YAML 官方网站 <http://www.yaml.org/> 找到。

如你所见，写 YAML 要比 XML 快得多 (不需要关闭标签或者引号)，并且比 '.ini' 文件功能更强 (ini 文件不支持层次)。所以 symfony 选择 YAML 作为配置信息的首选格式。在本书你会看到很多 YAML 文件，不过它很直观你用不着更深入地研究 YAML。

总结

symfony 是一个 PHP 5 web 应用程序开发框架。他在 PHP 语言的基础上增加了一个新层，为加速开发复杂的 web 应用程序提供了工具。本书全面介绍 symfony 的使用，你只需要熟悉并且理解现代编程的基本概念 -- 面向对象程序设计 (OOP)、对象关系映射 (ORM) 还有快速程序开发 (RAD)。唯一需要的技术背景是 PHP 5 的知识。

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

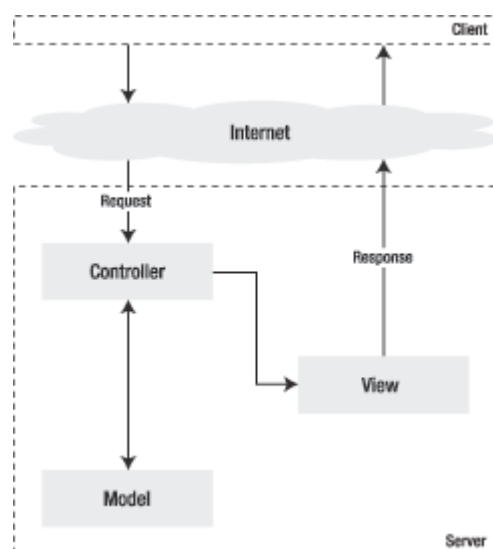
symfony 基于 MVC 架构这个经典的 Web 设计模式，MVC 架构包含三层：

- 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

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

// 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。所以我们将视图分成布局(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

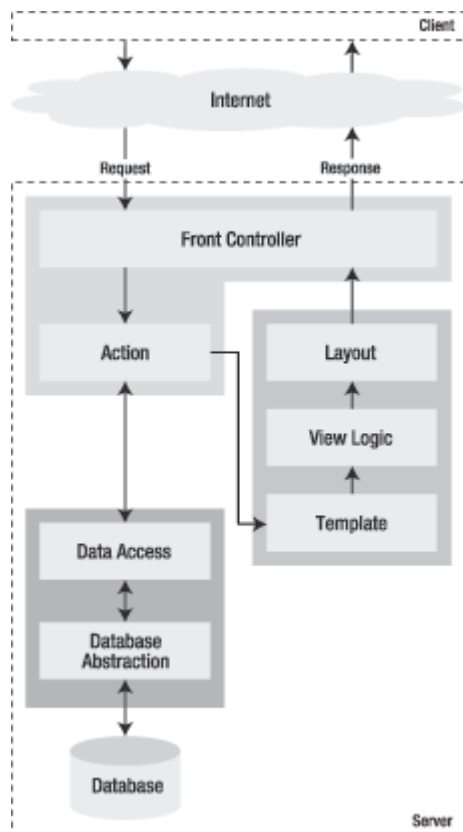
7 个脚本--每次修改一个页面需要打开这么多文件！可是，symfony 做了些简化。虽然使用最好的 MVC 架构，但是 symfony 的方式使得开发程序更加快速容易。

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(UpperCamel Case)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。

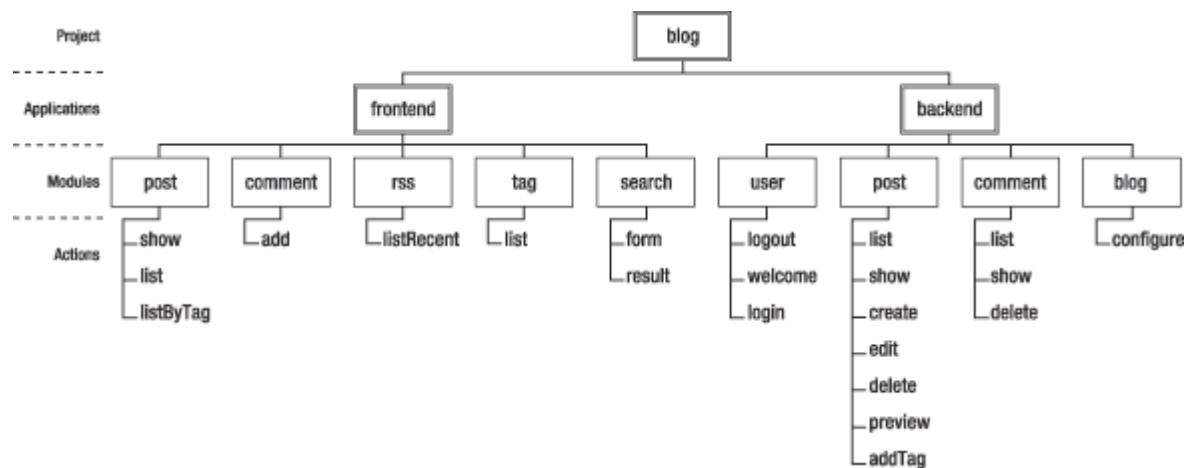
每个应用程序由一个或更多模块组成。模块就是功能相近的一个页面或者一组页面。例如，模块 `home` , `articles`, `help`, `shoppingCart`, `account` 等。

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, 这种树形结构与 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)。
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。
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。

目录	描述
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_F00', 'bar');
echo SF_F00;
// 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。

第 3 章 运行 symfony

如上章所述，symfony 是由许多 PHP 文件组成的框架。symfony 的项目需要使用这些文件，所以安装 symfony 其实就是让项目中可以使用这些文件。

symfony 是基于 PHP5 的框架。所以用以下命令确认你安装了正确的 PHP 版本：

```
> php -v
```

```
PHP 5.2.0 (cli) (built: Nov 2 2006 11:57:36)
Copyright (c) 1997-2006 The PHP Group
Zend Engine v2.2.0, Copyright (c) 1998-2006 Zend Technologies
```

如果版本号大于 5.0，你就可以开始安装了，安装过程将在此章节介绍。

安装沙盒(Sandbox)

如果你只是想要快速安装，试用一下 symfony，你应该使用沙盒。

沙盒里有一个空的 symfony 项目，这个项目包括基本的配置，一个默认的应用程序，还有 symfony 所需要的库(symfony、pake、lime、Creole、Propel 和 Phing)。它可以独立运行，不需要特别的服务器配置。

沙盒可以从 http://www.symfony-project.com/get/sf_sandbox.tgz 下载。解压缩到 web 服务器的根目录中（通常是 web/ 或者 www/）。为了统一性，本章将假设你把它解压到 sf_sandbox 目录下。

NOTE 把所有的文件放在 web 根目录下对于测试没有什么问题，不过在正式服务器上这么作是一个坏习惯。这样会把所有程序的内部文件暴露给最终用户。

执行 symfony 命令来测试安装是否成功。在 sf_sandbox/目录下，输入以下命令：Linux 系统下：

```
> ./symfony -V
```

Windows 系统下：

```
> symfony -V
```

你会看到沙盒的版本号：

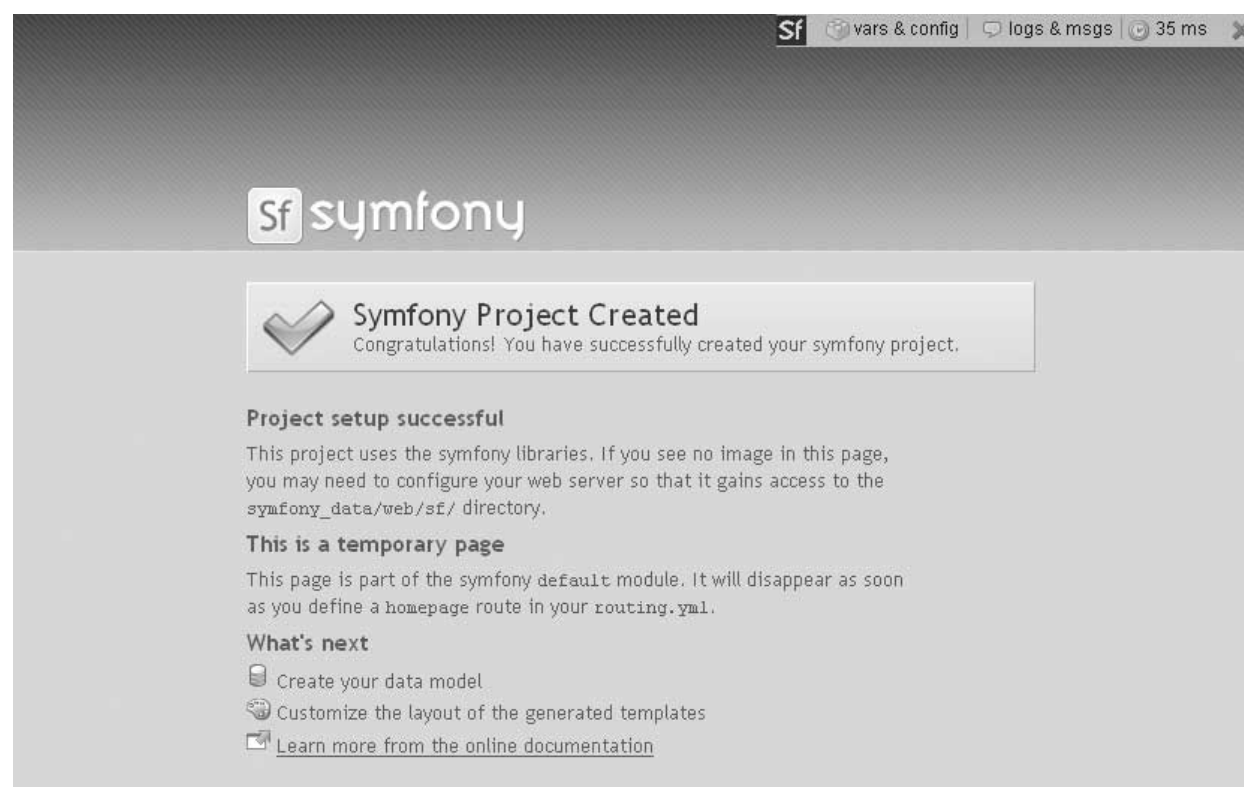
```
symfony version 1.0.0
```

现在请确认你的 web 服务器可以从下面的地址来访问沙盒：

http://localhost/sf_sandbox/web/frontend_dev.php/

如果你看到一个类似图 3-1 一样的成功页面，这就意味着安装已经完成。 如果没看到， 将会有有一个错误信息告诉你如何去修改配置文件。 你也可以参考下面的"安装问题"章节。

图 3-1 - 沙盒的祝贺页面



沙盒是用来给你在自己的电脑上面练习的，并不适合开发复杂的应用程序。不过，沙盒里的 symfony 的功能是完整的，与通过 PEAR 安装的没有差别。

要卸载沙盒，只要把 web/ 目录下的 sf_sandbox/ 删除即可。

安装 symfony 库

开发程序的时候，你也许会安装 symfony 两次：一次是你的开发环境，另外一次是在服务器上(除非服务器上已经安装过 symfony)。对于每台服务器而言，为了避免重复你也许会把所有的 symfony 文件放在一个地方，不管你开发几个程序。

因为 symfony 框架更新的很快，一个新的稳定版本可能在你安装后的几天内就发布了。 所以你需要认真考虑 symfony 框架更新的问题，这也是另外一个所有的项目应该共用同一个 symfony 的理由。

当要在真正程序开发中安装库的时候, 你有 2 个选择:

- 对大多数人而言推荐用 PEAR 安装方式。 他很容易共享和升级, 安装过程直接了当。
- Subversion (SVN) 安装模式通常是高级 PHP 程序开发者使用的, 可以获得最新的补丁, 增加自己开发的功能, 发布 symfony 的项目。

symfony 集成了一些其他的包:

- pake 是一个命令行工具。
- lime 是单元测试工具。
- Creole 是数据库抽象引擎。类似于 PHP 数据对象 (PDO), 他提供了程序代码与 SQL 数据库代码之间的一个接口, 以便切换到其他数据库。
- Propel 是 ORM 工具。它提供持续对象与查询服务。
- Phing 是 Propel 的命令行接口。

Pake 和 lime 是 symfony 小组开发的。Creole、Propel 和 Phing 是由其他小组开发并置于 GNU Lesser Public General License (LGPL) 协议下。所有这些包都绑定在 symfony 中。

Pear 方式安装 symfony

symfony 的 PEAR 包包含了 symfony 库。它也包含一个将 symfony 命令加入你的命令行的脚本。

安装第一步是把 symfony 频道加入 PEAR, 执行以下命令:

```
> pear channel-discover pear.symfony-project.com
```

用以下命令查看这个频道中的可用库列表:

```
> pear remote-list -c symfony
```

现在可以安装稳定版本的 symfony 了。 执行以下命令:

```
> pear install symfony/symfony
```

```
downloading symfony-1.0.0.tgz ...
```

```
Starting to download symfony-1.0.0.tgz (1,283,270 bytes)
```

```
.....
```

```
.....
```

```
.....done: 1,283,270 bytes
```

```
install ok: channel://pear.symfony-project.com/symfony-1.0.0
```

symfony 文件和命令行工具已经安装好了。在命令行执行 symfony 来确认安装是否成功, 查看版本号:

```
> symfony -V
```

```
symfony version 1.0.0
```

TIP 如果要安装最新的 beta 版本， 用命令 `pear install symfony/symfony-beta` 来安装。 Beta 版本通常是不稳定的， 不推荐在生产环境中用。

symfony 库安装在以下的目录中：

- `$php_dir/symfony/` 存放主要库文件。
- `$data_dir/symfony/` 存放 symfony 程序的结构； 默认模块； 配置文件， i18n 数据， 和其他。
- `$doc_dir/symfony/` 存放文档。
- `$test_dir/symfony/` 存放单元测试。

`_dir` 结尾的变量是 PEAR 配置的一部分， 可以用以下命令查看它的值：

```
> pear config-show
```

从 SVN 库中获得

对于生产服务器， 或者不能从 PEAR 安装的时候， 你可以直接从 symfony 的 subversion 库里通过检出最新版的 symfony：

```
> mkdir /path/to/symfony
> cd /path/to/symfony
> svn checkout http://svn.symfony-project.com/tags/RELEASE\_1\_0\_0/ .
```

命令 symfony， 只有在用 PEAR 安装时候才有效， 实际上是调用了 `/path/to/symfony/data/bin/symfony` 脚本。 所以 SVN 安装模式需就用下面的命令查看版本：

```
> php /path/to/symfony/data/bin/symfony -V
```

```
symfony version 1.0.0
```

如果选择用 SVN 安装方式， 你也许已经有了一个 symfony 项目。 你需要在你的项目中更改两个变量使其可以找到 symfony 文件：

```
<?php
```

```
$sf_symfony_lib_dir = '/path/to/symfony/lib/';
$sf_symfony_data_dir = '/path/to/symfony/data/';
```

第 19 章提供了用另外一种把项目和 symfony 安装关联起来(包括了符号连接和相对路径)的方法。

TIP 或者，你也可以下载 PEAR 包 (<http://pear.symfony-project.com/get/symfony-1.0.0.tgz>) 然后把它解压. 这与从版本库中取得是一样的。

配置一个程序

就如在第 2 章所学，symfony 把相关的应用程序放在项目中。项目中所有的应用程序都共享同一个数据库。为了设置一个应用程序，你必须先建立一个项目。

建立一个项目

每个 symfony 项目都有一个预定义的目录结构。symfony 命令行会自动建立一个新的项目的框架，合适的树状结构和访问权限。所以要建立一个新项目，只需简单的建立一个目录，然后告诉 symfony 在此建立一个项目即可。

PEAR 安装方式用以下命令：

```
> mkdir ~/myproject
> cd ~/myproject
> symfony init-project myproject
```

SVN 安装方式，用以下命令建立项目：

```
> mkdir ~/myproject
> cd ~/myproject
> php /path/to/symfony/data/bin/symfony init-project myproject
```

symfony 命令只能在项目的根目录下使用(前面提到的例子 myproject/)，因为用这个命令所作的事情都是基于项目的。

symfony 会建立一个下面这样的目录结构：

```
apps/
batch/
cache/
config/
data/
doc/
lib/
log/
plugins/
```

```
test/  
web/
```

TIP `init-project` 在项目根目录中增加一个 `symfony` 脚本。这个 PHP 脚本和用 PEAR 安装的 `symfony` 命令做的一样，所以没有原生命令行(SVN 安装方式)支持的话你可以用 `php symfony` 来替代 `symfony`。

建立一个应用程序

项目现在还没法用，因为他还至少需要一个应用程序。用 `symfony init-app` 命令传送一个应用程序的名字作为一个参数去初始化它：

```
> symfony init-app myapp
```

这将在项目根的 `apps/` 目录下建立一个叫 `myapp/` 的目录，它包含了一个默认的应用程序配置和一系列的子目录：

```
apps/  
  myapp/  
    config/  
    i18n/  
    lib/  
    modules/  
    templates/
```

在项目 `web` 目录里还会建立这个应用程序的两个默认环境对应的前端控制器的 PHP 文件：

```
web/  
  index.php  
  myapp_dev.php
```

`index.php` 是新建应用程序的生产环境前端控制器。因为当你在项目中创建了第一个应用程序的时候，`symfony` 建立一个叫 `index.php` 的文件来代替 `myapp.php`（如果你现在增加一个新的应用程序 `mynewapp`，新的生产环境前台会是 `mynewapp.php`）。要在开发环境中呼叫前端控制器 `myapp_dev.php` 来运行你的应用程序，你会在第 5 章了解更多关于环境的知识。

配置 Web 服务器

`web/` 目录下的这些脚本是应用程序的入口。Web 服务器要先配置才能在 Internet 中被访问。在开发服务器中，与专业主机服务解决方案一样，也需要访问 Apache 配置文件去设置一个虚拟主机。在一个共享的服务器中，你也许只能修改 `htaccess` 文件。

设置虚拟主机

例 3-1 是一个 Apache 配置的示例，可以了解如何在 httpd.conf 中添加新的虚拟主机。

例 3-1 - Apache 配置示例，apache/conf/httpd.conf

```
<VirtualHost *:80>
    ServerName myapp.example.com
    DocumentRoot "/home/steve/myproject/web"
    DirectoryIndex index.php
    Alias /sf /$sf_symfony_data_dir/web/sf
    <Directory "/$sf_symfony_data_dir/web/sf">
        AllowOverride All
        Allow from All
    </Directory>
    <Directory "/home/steve/myproject/web">
        AllowOverride All
        Allow from All
    </Directory>
</VirtualHost>
```

在例 3-1 配置中，需要用真实的路径替换掉/path/to/symfony/data。例如，在*nix 中用 PEAR 安装的话，你要输入类似这样的配置行：

```
Alias /sf /usr/local/lib/php/data/symfony/web/sf
```

NOTE 给 web/sf/设置别名并非是强制的。这可以让 Apache 找到网页 debug 工具条所用的图片，样式表和 JavaScript 文件，管理界面生成器，symfony 的默认页面还有 Ajax 支持。另外一个设置别名的方法是建立一个符号连接(sym link)或者把 /path/to/symfony/data/web/sf/复制到 myproject/web/sf/中。

重启 Apache。你新建立的应用程序现在能用以下的 URL 来访问：

http://localhost/myapp_dev.php/

你可以看到一个类似先前图 3-1 的成功页面。

SIDEBAR URL 重写

symfony 用 URL 重写来显示“漂亮的 URL”--有意义的地址对搜索引擎更友好并且对使用者隐藏了所有的数据。你会在第 9 章学习更多这方面的知识--路由(routing)。

如果你的 Apache 编译的时候没有选择 `mod_rewrite` 模块，检查 `httpd.conf` 中是否包含了 `mod_rewrite` 动态共享对象(DSO)。

```
AddModule mod_rewrite.c LoadModule rewrite_module
modules/mod_rewrite.so
```

对于 Internet 信息服务(IIS)，你需要安装和运行 `isapi/rewrite`。 `symfony` 在线手册有关于 IIS 安装的详细指导。

配置一个共享服务器

在共享服务器上配置一个应用程序有一些麻烦，因为服务器通常有你无法改变的特殊目录结构。

CAUTION 直接在共享服务器上测试和开发并不是一个好的做法。一个原因是因为这会让应用程序在未完成的时候可以被访问到，泄露其内部信息将会带来很大的安全隐患。另一个原因是，共享主机的性能往往不能满足开启调试工具时快速浏览应用程序的需要。因此你不应在共享服务器上开始你的开发，而应该在本地建立你的应用程序，完成后再移植到共享服务其上。第 16 章将告诉你移植技术和工具。

我们假设你的共享服务器需要把网页目录命名为 `www/` 来取代 `web/`，并且不允许你修改 `httpd.conf` 而只允许你修改在网页目录中的 `.htaccess` 文件。

在一个 `symfony` 项目中，每一个目录路径都是可配置的。第 19 章将带给你更多的信息，但是与此同时，你可以把 `web` 目录改为 `www` 目录，并且修改应用程序的配置，就如例 3-2 所示。这些将在应用程序的 `config.php` 文件底部加上。

例 3-2 - 在 `apps/myapp/config/config.php` 修改默认目录结构设定

```
$sf_root_dir = sfConfig::get('sf_root_dir');
sfConfig::add(array(
    'sf_web_dir_name' => $sf_web_dir_name = 'www',
    'sf_web_dir'      =>
        $sf_root_dir.DIRECTORY_SEPARATOR.$sf_web_dir_name,
    'sf_upload_dir'   =>
        $sf_root_dir.DIRECTORY_SEPARATOR.$sf_web_dir_name.DIRECTORY_SEPARATOR
        . sfConfig::get('sf_upload_dir_name'),
));
```

项目的网页根目录默认包含了一个 `.htaccess` 文件。就如例 3-3 所示。适当的修改它以配合你的共享服务器的需求。

例 3-3 - 默认的 .htaccess 配置, 当前在 myproject/www/.htaccess

```
Options +FollowSymLinks +ExecCGI

<IfModule mod_rewrite.c>
    RewriteEngine On

    # we skip all files with .something
    RewriteCond %{REQUEST_URI} \.+$
    RewriteCond %{REQUEST_URI} !\.html$
    RewriteRule .* - [L]

    # we check if the .html version is here (caching)
    RewriteRule ^$ index.html [QSA]
    RewriteRule ^([^.]+)$ $1.html [QSA]
    RewriteCond %{REQUEST_FILENAME} !-f

    # no, so we redirect to our front web controller
    RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>

# big crash from our front web controller
ErrorDocument 500 "<h2>Application error</h2>symfony
applicationfailed to start properly"
```

现在可以访问你的应用程序了。你可以通过这个 URL 访问 symfony 成功页面:

http://www.example.com/myapp_dev.php/

SIDEBAR 其他服务器配置

symfony 和其他服务器配置兼容。例如, 你可以用 alias 代替虚拟主机来访问 symfony 程序。你也能在 IIS 上运行 symfony 程序。关于配置有很多技巧, 本书并不准备解释所有的技巧。

想要找到具体的服务器配置指南, 可以参考 symfony 的 wiki (<http://www.symfony-project.com/trac/wiki>), 这里有详尽的指导。

安装问题

这些通常会有错误说明, 甚至会提供网络上针对此问题的资源连接。如果在安装中遇到问题, 尽量把错误或例外显示在 shell 或者浏览器上。

常见问题

如果你还是无法运行 symfony，检查以下几点：

- 一些 PHP 环境同时包含了 PHP4 和 PHP5 的命令。因此，在命令行用 php5 替代 php，也就是说试着用 php5 symfony 代替 symfony。你也许在 .htaccess 配置中需要增加 SetEnv PHP_VER 5 参数，或者把 web/ 目录中的 .php 换成 .php5。在 PHP4 命令行下试着访问 symfony 就会有类似下面的提示：

```
Parse error, unexpected ',', expecting '(' in .../symfony.php on line 19.
```

- 在 php.ini 中的内存限制，至少需要设置为 16M。通常的症状就是通过 PEAR 方式安装 symfony 的时候出现的错误信息。

```
Allowed memory size of 8388608 bytes exhausted
```

- 必须在 php.ini 中把 zend.ze1_compatibility_mode 参数设置为 off。否则通过浏览器去访问脚本的话会出现 "implicit cloning" 错误：

```
Strict Standards: Implicit cloning object of class 'sfTimer' because of 'zend.ze1_compatibility_mode'
```

- 在你的项目中位于 Web 服务器上的 log/ 和 cache/ 目录必须是可写的。如果在没有正确设定的时候访问 symfony 程序会出现以下提示：

```
sfCacheException [message] Unable to write cache file"/usr/myproject/cache/frontend/prod/config/config_handlers.yml.php"
```

- 系统的路径需要包含 php 命令的路径，你的 php.ini 的包含路径必须包括 PEAR 的路径（如果你使用 PEAR）。
- 有时，服务器上会有多个 php.ini 文件（例如，如果你使用 WAMP 包）。可以用 phpinfo() 函数去了解程序所用的 php.ini 文件所在的确切位置。

NOTE 虽然不是强制性的，但是这里强烈推荐，为了运行得更顺畅，在 php.ini 中设置 magic_quotes_gpc 和 register_globals 参数为 off。

symfony 资源

你可以在这些地方找到一些已经发现的问题的答案：

- symfony 安装论坛 (<http://www.symfony-project.com/forum/>) 这里有各种平台，环境配置，主机上安装 symfony 的问题讨论。

- 用户邮件列表档案(<http://groups.google.fr/group/symfony-users>)也可以搜索。你也许会找到一些人遇到同样的问题。
- symfony wiki (<http://www.symfony-project.com/trac/wiki#Installingsymfony>) 有由 symfony 用户提供的详细安装教程。

如果没有找到答案，试着把问题放到 symfony 社区。你可以在论坛，邮件列表甚至在#symfony IRC 频道得到大家的回应。

源代码版本控制

设置程序完成后，推荐进行版本控制。版本控制能跟踪对代码的所有修改，可以回退到以前的版本，更容易地给程序打补丁和更有效地进行团队合作开发。symfony 生来就支持 CVS，虽然更推荐使用 Subversion (<http://subversion.tigris.org/>)。下面的例子展示了 Subversion 的命令，我们假设你已经有 Subversion 服务器并且希望在项目中建立一个新的版本库。Windows 使用者推荐用叫做 TortoiseSVN (<http://tortoisetsvn.tigris.org/>) 的 Subversion 客户端。在 Subversion 文档中可以找到关于版本控制命令的更多信息。

下面的例子假设系统环境参数中已经定义了 \$SVNREP_DIR。如果还没有定义，你要以实际存放位置代替 \$SVNREP_DIR。

让我们在 myproject 项目中建立一个新的版本库：

```
> svnadmin create $SVNREP_DIR/myproject
```

建立 trunk、tags 和 branches 作为版本库的基础结构(layout)用以下命令：

```
> svn mkdir -m "layout creation" file:/// $SVNREP_DIR/myproject/trunk  
file:/// $SVNREP_DIR/myproject/tags  
file:/// $SVNREP_DIR/myproject/branches
```

这会是你的第一个版本。现在你需要把项目中除 cache/ 和 log/ 目录之外所有的文件导入版本库：

```
> cd ~/myproject  
> rm -rf cache/*  
> rm -rf log/*  
> svn import -m "initial import" .  
file:/// $SVNREP_DIR/myproject/trunk
```

输入以下命令检查已经提交的文件：

```
> svn ls file:/// $SVNREP_DIR/myproject/trunk/
```

看上去没问题。现在 SVN 库包含了你所有的项目文件的参考版本（还有历史）。这意味着 `~/myproject/` 目录需要与 SVN 库关联。要实现关联，首先修改 `myproject/` 目录的名字（如果一切正常你很快就可以删了它了）然后在一个新目录里签出 SVN 库里的文件：

```
> cd ~
> mv myproject myproject.origin
> svn co file:///SVNREP_DIR/myproject/trunk myproject
> ls myproject
```

现在你可以改写 `~/myproject/` 下的文件并提交到版本库中去。
`myproject.origin/` 目录已经没用了，别忘了把它删除掉。

还有一件事情需要配置。如果你提交当前工作目录到版本库中，也许包含了一些无用的文件，例如项目中的 `cache` 和 `log` 目录。所以必须为这个项目设置一个 SVN 忽略列表。当然，你还需要重新设置 `cache/` 和 `log/` 目录的权限：

```
> cd ~/myproject
> chmod 777 cache
> chmod 777 log
> svn propedit svn:ignore log
> svn propedit svn:ignore cache
```

SVN 默认的文字编辑器会启动。如果没有，在 Subversion 中设置你想用的文字编辑器：

```
> export SVN_EDITOR=<name of editor>
> svn propedit svn:ignore log
> svn propedit svn:ignore cache
```

现在只要把 `myproject/` 子目录的所有文件都添加到 SVN 中，SVN 会在提交的时候忽略掉列表中的文件：

*

保存，退出。完成了。

总结

如果在本地服务器上想测试或者尝试一下 `symfony`，最好安装一个已经预配置好环境的沙盒。

如果是真正的开发或者在生产服务器上，最好使用 `PEAR` 安装或者用 SVN 检出。这样做会安装 `symfony` 的库，还需要初始化一个项目和应用程序。应用程序设

置的最后一步是服务器配置，这有很多种做法。symfony 能完美地在虚拟主机下运行，这也是推荐的解决方案。

如果在安装中遇到问题，你能从 symfony 网站上找到许多教程、回答、FAQ。如果需要，你可以把问题提交到 symfony 社区，这样会很快得到答案。

当项目初始化好后，开始版本控制流程是一个好习惯。

现在你已经准备好使用 symfony 了，是时候去建立一个基础的网页程序了。

第 4 章 建立页面的基础知识

很奇怪，每次学习新语言或者框架的第一个例子都是在屏幕上显示"Hello, world!". 目前为止所有利用人工智能来实现交谈的尝试的结果都很差，所以电脑能问候整个世界这种想法实在有些古怪。但是 `symfony` 并不比其他程序笨，证据是，你可以用 `symfony` 创建一个说"Hello, <你的名字>"的页面。

本章会告诉你如何创建一个模块，也就是一组页面的集合体。你还将了解到如何建立一个页面，由于 MVC，页面由一个动作和一个模板构成。链接和表单是 web 交互的基础，你将在这一章了解如何在模板里增加他们，如何用动作处理他们。

建立模块框架

在第二章中我们介绍过，`symfony` 把页面组织成模块。建立页面之前，你必须先建立一个模块，也就是一个 `symfony` 能识别的目录结构的一个空壳。

`symfony` 命令行工具能自动建立模块。你只需要用 `symfony` 命令行工具执行 `init-module` 任务并传应用程序名与模块名这两个参数给它就可以了。在前一章里，你建立了 `myapp` 应用程序。如果要在这个应用程序里增加一个 `mymodule` 模块，只需要在命令行下输入下面的命令：

```
> cd ~/myproject
> symfony init-module myapp mymodule

>> dir+      ~/myproject/apps/myapp/modules/mymodule
>> dir+      ~/myproject/apps/myapp/modules/mymodule/actions
>> file+
~/myproject/apps/myapp/modules/mymodule/actions/actions.class.php
>> dir+      ~/myproject/apps/myapp/modules/mymodule/config
>> dir+      ~/myproject/apps/myapp/modules/mymodule/lib
>> dir+      ~/myproject/apps/myapp/modules/mymodule/templates
>> file+
~/myproject/apps/myapp/modules/mymodule/templates/indexSuccess.php
>> dir+      ~/myproject/apps/myapp/modules/mymodule/validate
>> file+
~/myproject/test/functional/myapp/mymoduleActionsTest.php
>> tokens
~/myproject/test/functional/myapp/mymoduleActionsTest.php
>> tokens
~/myproject/apps/myapp/modules/mymodule/actions/actions.class.php
>> tokens
~/myproject/apps/myapp/modules/mymodule/templates/indexSuccess.php
```

除了 actions/, config/, lib/, templates/, 与 validate/目录, 这条命令只建立了三个文件。test/目录里的文件与单元测试有关, 在第 15 章之前你都不用管它。actions.class.php(见例 4-1)做了一个到默认模块的成功页面的跳转。templates/indexSuccess.php 文件是空的。

例 4-1 - 默认自动生成的动作 actions/actions.class.php

```
<?php

class mymoduleActions extends sfActions
{
    public function executeIndex()
    {
        $this->forward('default', 'module');
    }
}
```

NOTE 如果你看一下实际的 actions.class.php 文件, 你会注意到除了上面的这几行之外还有其他的内容, 包括一些注释。这是因为 symfony 推荐使用 PHP 注释来为你的项目生成文档, 所以每个类文件都与 phpDocumentor 工具 (<http://www.phpdoc.org/>) 兼容。

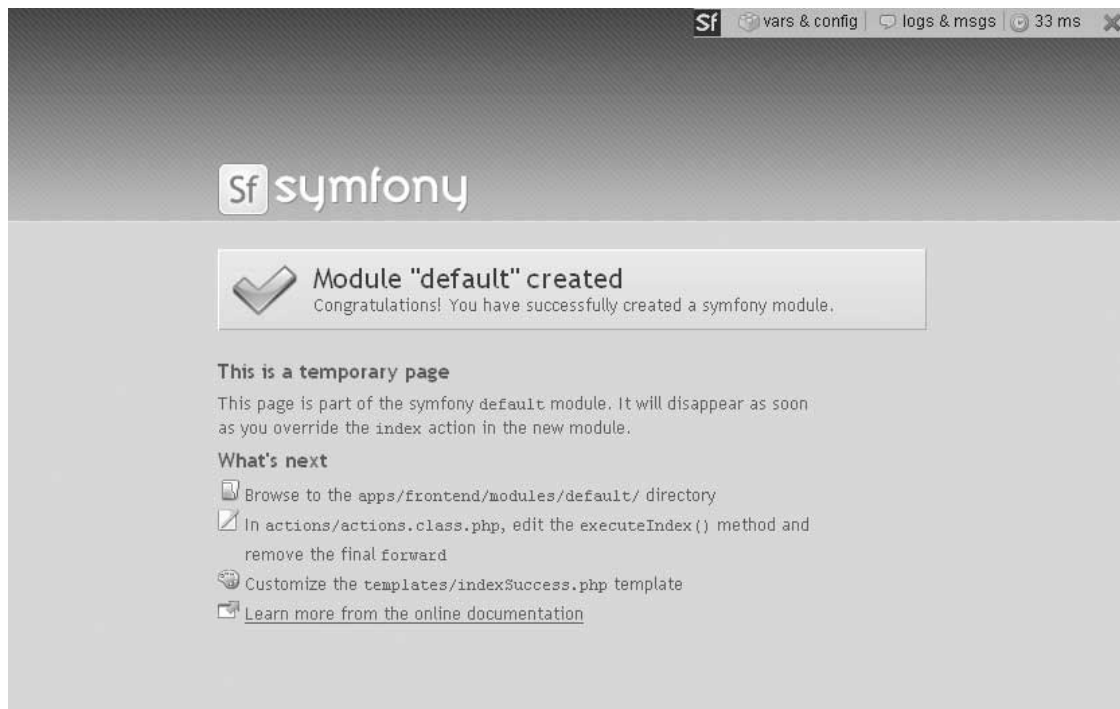
symfony 为每一个新模块建立一个 index 动作。它是由一个 executeIndex 的方法与一个叫 indexSuccess.php 的模板组成的。execute 前缀与 Success 后缀的含义会在第 6 章与第 7 章中分别解释。现在你可以认为这是一种命名习惯。在浏览器中输入下面的网址就可以看到这个页面(图 4-1):

http://localhost/myapp_dev.php/mymodule/index

本章不会用到这个默认 index 动作, 所以你可以把 executeIndex() 方法从 actions.class.php 文件中去掉, 并把 indexSuccess.php 文件从 templates/ 目录中删除。

NOTE 除了命令行, symfony 还提供了其他的建立模块的方法。其中之一是你自己来建立这些文件与目录。很多时候, 模块中的动作和模板用来处理一个表里面的数据。由于建立、获取、更新与删除所需的代码往往是一样的, symfony 提供一种称之为脚手架(scaffolding)的机制来自动生成一个模块。这种技术详见第 14 章。

图 4-1 - 自动生成的默认 index 页



增加一个页面

symfony 里面，页面背后的逻辑放在动作里面，表现放在模板里。不需要逻辑的页面也需要一个空的动作。

增加一个动作

我们需要一个通过 myAction 动作来访问 "Hello, world!" 的页面。要建立这个页面，只要在 myModuleActions 类里面增加一个 executeMyAction 方法，如例 4-2。

例 4-2 - 增加一个动作就是给动作类增加一个执行方法

```
<?php

class myModuleActions extends sfActions
{
    public function executeMyAction()
    {
    }
}
```

动作方法的名字永远是 execute`XXX`()，方法名字的第二部分的第一个字母总是大写。

现在，如果你访问下面的网址：

http://localhost/myapp_dev.php/mymodule/myAction

symfony 会抱怨缺少`myActionSuccess.php' 模板。这很正常；在 symfony 里，一个页面永远是由一个动作与一个模板组成。

NOTE URL(不是域名)是区分大小写的，symfony 也区分大小写(虽然在 PHP 里方法名不区分大小写)。这就是说，如果你增加一个 `executemyaction()` 方法，或者 `executeMyaction()`，然后你在浏览器里访问 `myAction`，symfony 会返回 404 错误信息。

SIDEBAR URL 是响应的一部分

symfony 包含一个路由系统，这个系统可以把真正的动作名与 URL 的形式分开来。这样就可以实现特殊 URL 格式。你可以不受文件结构或者请求参数的限制；动作的 URL 可以是你想要的样子。例如，请求一个 `article` 模块的 `index` 动作的 URL 常常是这样的：

http://localhost/myapp_dev.php/article/index?id=123

这个 URL 从数据库里面取出指定的文章。在这个例子里，这篇文章(`id=123`)是欧洲(`europe`)栏目里的一篇关于法国金融(`finance in France`)的文章。但是通过修改 `routing.yml` 配置文件，这个 URL 可以完全改成另外一种直观的形式：

<http://localhost/articles/europe/france/finance.html>

这个 URL 不仅对搜索引擎更友好，也对用户更有意义，用户可以像使用命令行一样通过在地址栏执行特定的查找，例如：

<http://localhost/articles/tagged/finance+france+euro>

symfony 知道如何为用户解析与生成漂亮的 URL。路由系统自动地从一个漂亮的 URL 中剥离出参数然后传给动作。它也能格式化回应的超链接使他们看起来更"漂亮"。这个功能详见第 9 章。

总之，这意味着应用程序的动作的命名可以和他们的 URL 不一致，但是动作方法的命名必须与动作名相统一。动作名说明动作要做的事情，它通常是一个不定式动词(例如 `show`, `list`, `edit` 等)。动作名可以隐藏起来不让用户知道，所以请放心的使用动作的名字(例如 `listByName` 或者 `showWithComments`)。这样可以有效地节省注释，另外代码的可读性也大大增强了。

增加一个模板

动作需要一个模板来表现自己。模板是模块的 `templates/` 目录里的一个文件，模板名字由动作名与动作终止组成。默认的动作终止是 `"success"` 也就是成功，所以 `myAction` 动作的模板名是 `myActionSuccess.php`。

理想的模板只包含显示代码，所以 PHP 代码越少越好。显示"Hello, world!"的页面的模板可以如同例 4-3 中的那么简单。

例 4-3 - mymodule/templates/myActionSuccess.php 模板

```
<p>Hello, world!</p>
```

如果需要在模板里执行一些 PHP 代码，你应该避免使用通常的 PHP 语法(如例 4-4)。相反，你应该在模板里面使用特殊的 PHP 语法，如例 4-5 所示，这样不是 PHP 程序员的人也能理解。这样不仅最终生成的代码的缩进格式正确，而且可以让你把复杂的代码放在动作里面，因为只有控制语句(if, foreach, while 等)有特殊语法。

例 4-4 - 通常的 PHP 语法，对于动作没问题，对于模板就很糟糕

```
<p>Hello, world!</p>
<?php

if ($test)
{
    echo "<p>".time()."</p>";
}

?>
```

例 4-5 - 另类 PHP 语法，适合于模板

```
<p>Hello, world!</p>
<?php if ($test): ?>
<p><?php echo time(); ?></p>
<?php endif; ?>
```

TIP 一般来说模板语法的可读性是否够强是看这个文件是否不包含 PHP 的 echo 语句或者"{}"。大多数时候，开始的<?php 应该与结束的?>在同一行。

从动作传递信息给模板

动作要做的事情是所有的复杂计算，取出数据，测试，为模板设定显示或者测试用的变量。symfony 让动作类的属性(动作里的可以通过\$this->variableName 访问)能够直接在模板里面的全局命名空间里面访问得到(通过\$variableName)。例 4-6 与 4-7 演示如何从动作传递信息给模板。

例 4-6 - 设定动作的一个属性，把它传给模板

```
<?php

class mymoduleActions extends sfActions
{
    public function executeMyAction()
    {
        $today = getdate();
        $this->hour = $today['hours'];
    }
}
```

例 4-7 - 模板能直接访问动作的属性

```
<p>Hello, world!</p>
<?php if ($hour >= 18): ?>
<p>Or should I say good evening? It's already <?php echo $hour ?>. </p>
<?php endif; ?>
```

NOTE 有几个数据可以直接在模板中访问而不需要在动作里面设置。每个模板都可以执行 `$sf_context`, `$sf_request`, `$sf_params` 还有 `$sf_user` 对象的方法。它们包含当前上下文、请求、请求参数还有 session 的信息。不久你就能学会怎么有效的利用它们。

从用户表单取得数据

表单是从用户取得信息的好方法。用 HTML 写表单的元素有时会很麻烦，特别是你想要 XHTML 兼容时。你可以按照平常的方式在 symfony 模板里面使用表单元素，如例 4-8 所示，不过 symfony 提供了一些辅助函数来简化这个任务。

例 4-8 - 模板可以包含普通的 HTML 代码

```
<p>Hello, world!</p>
<?php if ($hour >= 18): ?>
<p>Or should I say good evening? It's already <?php echo $hour ?>. </p>
<?php endif; ?>
<form method="post" target="/myapp_dev.php/mymodule/anotherAction">
    <label for="name">What is your name?</label>
    <input type="text" name="name" id="name" value="" />
    <input type="submit" value="Ok" />
</form>
```

辅助函数是 symfony 定义的用在模板里的函数。它输出 HTML 代码从而节省你写 HTML 代码的时间。使用 symfony 辅助函数，你可以用例 4-9 的代码达到与例 4-8 同样的结果。

例 4-9 - 用辅助函数比写 HTML 标签更快更容易

```
<p>Hello, world!</p>
<?php if ($hour >= 18): ?>
<p>Or should I say good evening? It's already <?php echo $hour ?>.</p>
<?php endif; ?>
<?php echo form_tag('myModule/anotherAction') ?>
    <?php echo label_for('name', 'What is your name?') ?>
    <?php echo input_tag('name') ?>
    <?php echo submit_tag('Ok') ?>
</form>
```

SIDEBAR 辅助函数是来帮助你的。

如果，你认为在例 4-9 的例子中，辅助函数的版本没有写 HTML 快，看看这个例子：

```
$card_list = array(
>     'VISA' => 'Visa',
>     'MAST' => 'MasterCard',
>     'AMEX' => 'American Express',
>     'DISC' => 'Discover');
>     echo select_tag('cc_type', options_for_select($card_list,
'AMEX'));
>     ?>
```

上面的代码的 HTML 输出如下：

```
<select name="cc_type" id="cc_type">
    <option value="VISA">Visa</option>
    <option value="MAST">MasterCard</option>
    <option value="AMEX" selected="selected">American Express</option>
    <option value="DISC">Discover</option>
</select>
```

在模板里使用辅助函数使编写代码的速度提高，代码更清晰，更简洁。唯一的代价是需要花时间学习他们，学习过程将一直持续到本书完结，到你在你习惯的编辑器中用快捷键写的时候。所以如果不会用 symfony 的辅助函数，你仍然可以继续使用 HTML 标签，不过这很浪费也很枯燥。

注意我们不推荐专业 web 开发者使用短开始标签(<?=, 等效于 <?php echo)，因为你的正式服务器可能能够支持多种脚本语言而把短标签与别的脚本语言混淆。另外，短开始标签不是 PHP 的默认设置，需要打开才能使用。最后，如果你需要处理 XML 与验证，短标签会有问题因为<?在 XML 里有特殊含义。

由于 symfony 提供了很多辅助函数简化表单，表单处理需要一整章来讲解。表单处理详见第 10 章。

链接到另一个动作

我们已经讲到动作名与访问这个动作的 URL 之间需要有一个转换过程。所以如果你建立一个到 `anotherAction` 的链接，如例 4-10 所示，它只适用于默认的路由设置。如果以后你决定修改 URL 格式，那你还要修改所有包含这个链接的模板。

例 4-10 - 传统的超链接

```
<a href="/myapp_dev.php/mymodule/anotherAction?name=anonymous">
  I never say my name
</a>
```

为了避免这样的麻烦，请使用 `link_to()` 辅助函数来建立所有的链接到应用程序内部的动作的超链接。例 4-11 演示了如何使用超链接辅助函数。

例 4-11 - `link_to()` 辅助函数

```
<p>Hello, world!</p>
<?php if ($hour >= 18): ?>
<p>Or should I say good evening? It's already <?php echo $hour ?>.</p>
<?php endif; ?>
<?php echo form_tag('mymodule/anotherAction') ?>
  <?php echo label_for('name', 'What is your name?') ?>
  <?php echo input_tag('name') ?>
  <?php echo submit_tag('Ok') ?>
  <?php echo link_to('I never say my
name', 'mymodule/anotherAction?name=anonymous') ?>
</form>
```

上面的代码生成的 HTML 与前一个例子完全一样，但是如果修改路由规则，所有的模板会根据规则重新格式 URL。

`link_to()` 辅助函数，与很多辅助函数类似，接受另一个特殊的参数，这个参数用来传递 HTML 标签属性。例 4-12 是一个 `option` 属性的例子还有生成的 HTML。`option` 参数可以是一个数组或者一个简单的由几个 `key=value` 与空格组成的字符串。

例 4-12 - 大多数辅助函数有 `Option` 参数

```
// 用数组作 option 参数
```

```

<?php echo link_to('I never say my name',
'mymodule/anotherAction?name=anonymous',
    array(
        'class'      => 'special_link',
        'confirm'    => 'Are you sure?',
        'absolute'   => true
    )) ?>

// 用字符串作 option 参数
<?php echo link_to('I never say my name',
'mymodule/anotherAction?name=anonymous',
    'class=special_link confirm=Are you sure? absolute=true') ?>

// 结果一样
=> <a class="special_link" onclick="return confirm('Are you sure?'); "
href="http://localhost/myapp_dev.php/mymodule/anotherAction/name/anonymous">
    I never say my name</a>

```

任何使用 `symfony` 辅助函数输出 HTML 标签的时候，都可以在 `option` 参数中加入额外的属性(例如例 4-12 中的 `class` 属性)。你甚至可以用 HTML 4.0 的“快速而肮脏(quick-and-dirty)”的方式(不写双引号)，`symfony` 会用漂亮的 XHTML 方式输出。这是用辅助函数比写 HTML 快的又一个原因。

NOTE 由于需要额外的解析与转换，字符串形式比数组要慢。

与其它辅助函数类似，链接辅助函数有好几种形式与参数。第 9 章将向你详细介绍这些内容。

从请求中取得信息

无论用户通过表单(通常是 POST 请求)还是通过 URL(GET 请求)取得信息，你都可以在动作中通过 `sfActions` 对象的 `getRequestParameter()` 方法取得相关的数据。例 4-13 演示了如何在 `actionAction` 中取得 `name` 参数的值。

例 4-13 - 在动作中取得请求参数的值

```

<?php

class mymoduleActions extends sfActions
{
    ...

```

```

public function executeAnotherAction()
{
    $this->name = $this->getRequestParameter('name');
}
}

```

如果数据操作很简单，你甚至不必用动作来取得参数值。模板可以直接通过 `$sf_params` 的 `get()` 方法来取得参数的值，类似于动作中的 `getRequestParameter()` 方法。

如果 `executeAnotherAction()` 方法是空的，例 4-14 中的这种方法也可以从 `anotherActionSuccess.php` 模板中取到 `name` 参数的值。

例 4-14 - 直接从模板中取得参数的值

```
<p>Hello, <?php echo $sf_params->get('name') ?>!</p>
```

NOTE 为什么不直接使用 `$_POST`, `$_GET`, 或 `$_REQUEST` 变量呢？因为如果你的 URL 的格式会变化(例如 <http://localhost/articles/europe/france/finance.html>，没有?或者=)，这样这些 PHP 变量就不管用了，只有路由系统能够取得请求参数。还有你可能需要输入过滤器来防止恶意代码注入，只有保持所有的参数使用一个干净的参数存储器的时候才能实现。

`$sf_params` 对象的作用仅仅是数组的替代品。例如，如果你想判断一个请求参数是否存在，你可以只用 `$sf_params->has()` 方法而不必用 `get()` 方法取得实际的值，如例 4-15。

例 4-15 - 在模板中判断一个参数是否存在

```

<?php if ($sf_params->has('name')): ?>
    <p>Hello, <?php echo $sf_params->get('name') ?>!</p>
<?php else: ?>
    <p>Hello, John Doe!</p>
<?php endif; ?>

```

你可能已经猜到这用一行代码就可以完成。与 `symfony` 里面的大多数 `getter` 方法一样，动作里的 `getRequestParameter()` 还有模板里的 `$sf_params->get()` 方法(实际上两者调用的是同一个对象的同一个方法)可以有第二个参数：默认值，在参数不存在的时候起作用。

```
<p>Hello, <?php echo $sf_params->get('name', 'John Doe') ?>!</p>
```

总结

在 symfony 里面，页面由一个动作(actions/actions.class.php 文件里的一个方法，以 execute 开头)还有一个模板(templates/目录里的一个文件，通常以 Success.php 结尾)组成。功能有关联的页面组成模块。写模板有辅助函数帮忙，辅助函数是 symfony 提供的返回 HTML 代码的函数。并且你需要把 URL 考虑成回应的一部分，URL 也可以根据需要重新安排格式，所以你需要避免绕过超链接辅助方法直接写动作的 URL。

一旦了解了这些基本原理，你就可以开始用 symfony 写一个完整的 web 应用程序了。但是这需要花很长时间，因为几乎所有的功能都可以通过 symfony 的某种功能来简化开发……，所以这本书还没结束。

第 5 章 配置 symfony

为了达到简单易用的目的，symfony 定义了一些惯例，这些惯例能够满足大多数情况的需求。另一方面，使用一系列简单而强大的配置文件，我们可以定制这个框架及应用程序的几乎所有的地方。使用这些配置文件可以为程序增加一些特殊的参数。

这一章介绍配置系统如何工作：

- symfony 配置信息保存在 YAML 格式的文件里，当然也可以换成其它格式。
- 配置文件分成项目、应用程序、模块这几个等级，分别存放在项目目录中对应子目录里。
- 你可以定义几套不同的配置文件，symfony 里面的一套配置文件称之为环境。
- 配置文件里面定义的值可以在 PHP 代码里面取得。
- 另外，symfony 可以识别 YAML 里面的 PHP 代码等，这使得配置系统更灵活。

配置系统

不论什么用途，大多数 web 应用程序都有一些共同的特征。例如，有些区域只允许一部分用户访问；很多页面共用一个布局；表单填写验证失败后自动把用户输入的内容放进表单。框架定义了一些实现这些特性的结构，开发者通过配置系统进一步的调整它们。这种策略可以节省大量开发时间，因为很多改变并不用改写代码，尽管实现这些改变需要很多代码。这种策略也更有效率，因为这些信息可以存放在容易识别的位置。

但是，这样的做法有两个严重缺点：

- 开发者最后整天不停的写复杂的 XML 文件。
- 使用 PHP 处理每个请求花费的时间更多了。

考虑到这些缺点，同时 symfony 又要充分利用配置文件的优点。事实上，symfony 的配置系统的目标是：

- 强大：所有的可以配置的东西都可以用配置文件配置
- 简单：很多配置信息不会出现在普通的应用程序里，因为它们很少需要改变
- 容易：开发者可以很容易的阅读，建立，修改配置文件
- 可定制：默认的配置语言是 YAML，但也可以换成 INI、XML，或者是别的格式
- 快速：应用程序本身不用处理配置文件，由配置系统处理配置文件，把配置文件编译成能够快速执行的 PHP 代码

YAML 语法与 symfony 惯例

symfony 默认使用 YAML 格式存放配置信息，而不用传统的 INI 或者 XML 格式。YAML 通过缩进表示结构而且写起来很快。它的特点与基本规则在第一章里面已经讲到了。不过，在写 YAML 的时候，你还需要记住几条规则。本节将介绍最常用的几个规则。想完整的了解 YAML，请访问 YAML 网站(<http://www.yaml.org/>)。

首先，绝不要在 YAML 文件里使用制表符(tab)，应该使用空格。YAML 解析器不能解析制表符，所以请使用空格来缩进(在 symfony 里面使用两个空格缩进)，如例 5-1。

例 5-1 - YAML 文件禁用制表符(tab)

```
# 绝不用制表符
all:
-> mail:
-> -> webmaster:  webmaster@example.com

# 应该使用空格
all:
  mail:
    webmaster: webmaster@example.com
```

如果你的参数是以空格开始或者结束的字符串，应使用单引号把它包起来。如果一个字符串参数包含特殊字符，也要用单引号包起来，如例 5-2。

例 5-2 - 非标准的字符串要用单引号包起来

```
错误 1: This field is compulsory
错误 2: ' This field is compulsory '
错误 3: 'Don't leave this field blank'    # 必须用两个单引号来表示字符串中的'
```

利用特殊字符头(> 或 |)与一个缩进，长的字符串可以跨行表示。如例 5-3。

例 5-3 - 定义长的多行字符串

```
accomplishment: >          # 由>开头的折叠式
  Mark set a major league  # 每一个换行被折叠成一个空格
  home run record in 1998. # 使得 YAML 可读性更强
stats: |                   # 由|开头的原始式
  65 Home Runs             # 所有的换行都被保留
  0.278 Batting Average    # 缩进不会在结果里出现
```

如果要定义数组，需要用方括号把元素括起来或者使用展开的减号语法，如例 5-4。

例 5-4 - YAML 的数组语法

数组语法的简写

```
players: [ Mark McGwire, Sammy Sosa, Ken Griffey ]
```

数组的展开语法

```
players:  
  - Mark McGwire  
  - Sammy Sosa  
  - Ken Griffey
```

如果要定义关联数组或者说哈希表，要用大括号把元素括起来，键与值 key: value 中间保留一个空格。也可以用展开语法，每一个新的键增加一个缩进与换行，如例 5-5。

例 5-5 - YAML 关联数组语法

错误的语法，冒号后缺少空格

```
mail: {webmaster:webmaster@example.com,contact:contact@example.com}
```

关联数组的正确简写

```
mail: { webmaster: webmaster@example.com, contact: contact@example.com }
```

关联数组的展开语法

```
mail:  
  webmaster: webmaster@example.com  
  contact:   contact@example.com
```

定义一个布尔值时，on、1 或者 true 代表肯定值，off、0、或者 false 代表否定值。如例 5-6。

例 5-6 - YAML 布尔值语法

```
true_values:  [ on, 1, true ]  
false_values: [ off, 0, false ]
```

请不要吝啬使用注释(以井号#开头)还有空格，这会使注释文件更易读，如例 5-7。

例 5-7 - YAML 注释语法与值对齐

这是一个注释

```
mail:
  webmaster: webmaster@example.com
  contact:   contact@example.com
  admin:     admin@example.com    # 多的空格可以帮助对齐
```

在某些 symfony 配置文件里面，你会发现一些行以#开头(YAML 解析器会忽略这些行),但这些行看上去像普通的设置行。这是一个 symfony 的惯例：默认设置，从 symfony 内核里其他的 YAML 文件里面继承的设置，这些会在你的应用程序配置文件里面出现并用#注释起来，给你参考。如果你想改变这些参数，只要把注释去掉就可以了。如例 5-8。

例 5-8 - 注释里的默认配置

```
# 缓存的默认值是关闭
settings:
# cache: off

# 如果你想修改这个值，去掉注释
settings:
  cache: on
```

symfony 有时会把一些参数定义分类。一个分类里面的所有设置都放在分类头下面。把长的 key: value 列表分组能增强可读性。分类头以点(.)开头。如例 5-9。

例 5-9 - 分类头与键类似，但是以. 开头

```
all:
  .general:
    tax:          19.6

  mail:
    webmaster:    webmaster@example.com
```

在这个例子里，mail 是一个键，general 只是一个分类头。分类头可以当作不存在，如例 5-10。tax 参数实际上是 all 键的直接子元素。

例 5-10 - 分类头只用于增强可读性，实际上可以忽略

```
all:
  tax:          19.6

  mail:
    webmaster:    webmaster@example.com
```

SIDEBAR 如果你不喜欢 YAML

YAML 只是一个给 PHP 代码定义设置的界面，所以 YAML 里面定义的配置信息都会被转换成 PHP 代码。浏览一个应用程序，查看他的缓存的配置信息(例，在 `cache/myapp/dev/config/`)。你会发现 YAML 配置对应的 PHP 文件。这一章后面我们会详细介绍配置缓存。

好消息是如果你不喜欢 YAML 文件，你可以自己动手，使用 PHP 代码或者其他格式(XML, INI 等)。在本书中，你会遇到其他的不使用 YAML 定义配置的方法，在第 19 章你会了解如何替换 `symfony` 的配置文件处理器。如果你用好它们，你可以利用这些技巧绕开配置文件或者定义你自己的格式。

救命，YAML 文件把我的程序搞死了

YAML 文件会被解析成 PHP 哈希与数组，然后这些值会在程序的不同地方改变视图，控制器或者模型的行为。很多时候，配置文件里面的问题直到用到的时候才被察觉。而且，显示的错误信息通常不是明显与 YAML 配置文件相关的。

如果改变了配置文件之后程序突然停止运行了，应该检查一下你是否犯了下面的常见的 YAML 错误：

- 键和值中间缺少空格：

```
key1:value1      # 冒号:后缺少空格
```

- 数组里面的键应该按照同样的方式缩进：

```
all:
  key1: value1
  key2: value2 # 缩进与其他的数组元素不同
  key3: value3
```

- 键或值里有 YAML 保留字符而且没有用单引号：

```
message: tell him: go way # :, [, ], { and } 是 YAML 保留字符
message: 'tell him: go way' # 正确的语法
```

- 修改一个被注释了的行：

```
# key: value      # 由于前面的#, 这行永远不会生效
```

- 同一级别相同键的值设置了两次：

```
key1: value1
key2: value2
key1: value3      # key1 定义了两次，取最后一次定义的值
```

- 你认为设置值应该是一个特定的类型，实际上如果你不转换它，设置值永远是字符串：

```
income: 12,345 # 除非你转换它，否则它永远是字符串
```

配置文件概述

配置信息按照目的存放在不同的文件里。这些文件包含参数定义或者设置。一些参数可以在不同的级别覆盖(项目，应用程序，模块)；一些只针对特定级别。本节将介绍这些配置文件，第 19 章将更深入的介绍配置文件。

项目配置

symfony 项目有一些默认配置文件。下面是 myproject/config/目录下面的配置文件：

- config.php：这是所有页面或者命令行方式 PHP 脚本执行的第一个文件。它包含 symfony 框架的路径，你可以把这个路径指定到另外一个 symfony 框架。如果你在这里增加一些 define 语句，这样定义的常量在项目的每个应用程序都可以访问得到。第 19 章将会深入介绍这个文件的使用。
- databases.yml：这个文件用来存放数据库连接设置(主机，登录名，密码等)。第 8 章有更详细的介绍。这个文件的配置信息可以在应用程序这一级别覆盖。
- properties.ini：这个文件存放命令行工具需要的参数，包括项目名称，远程服务器的连接设置。第 16 章将详细介绍此文件的功能。
- rsync_exclude.txt：这个文件定义同步服务器的时候哪些文件不需要同步。详见第 16 章。
- schema.yml 与 propel.ini：这两个文件是 Propel (symfony 的 ORM 层) 的数据访问配置文件。它们用来使 Propel 与 symfony 的类还有项目的数据协同工作。propel.ini 是自动生成的，所以你不用修改它。如果你不用 Propel，就不需要这些文件。这两个文件的使用详见第 8 章。

这些文件多数时候是被外部组件或者命令行使用，或者 symfony 在 YAML 解析程序载入之前就要用到它们。所以有些文件不是 YAML 格式。

应用程序配置

配置的主要部分是应用程序配置。前端控制器(在 web/目录)里定义主要的常量，YAML 文件在应用程序目录的 config/目录里，i18n/目录里是国际化需要的文件，还有一些在框架文件里隐藏着的但是很有用的其他项目配置信息。

前端控制器配置

前端控制器里存放了应用程序最开始的配置信息；它是一个请求最开始执行的脚本。请看例 5-11 中默认的 web/index.php。

例 5-11 - 默认的生产环境前端控制器

```
<?php
```

```
define('SF_ROOT_DIR',    dirname(__FILE__).'/..');
define('SF_APP',          'myapp');
define('SF_ENVIRONMENT', 'prod');
define('SF_DEBUG',        true);
```

```
require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.
SF_APP.DIRECTORY_SEPARATOR.'config'.DIRECTORY_SEPARATOR.'config.php');
```

```
$sfContext->getInstance()->getController()->dispatch();
```

定义了应用程序名(myapp)与环境(prod)之后，先载入通用配置文件，然后继续分派请求。这里定义了一些有用的常量：

- SF_ROOT_DIR：项目根目录（一般情况请保留默认值，除非想变更目录结构）。
- SF_APP：项目中的应用程序名。需要它来生成文件路径。
- SF_ENVIRONMENT：环境名（prod, dev, 或者其他你定义的本项目的环境）。用来决定使用哪一套配置信息。本章稍后会解释环境的概念。
- SF_DEBUG：是否启用调试模式（详见第 16 章）。

如果你要改变这些值，那么你可能需要另一个前端控制器。下一章将介绍前端控制器以及如何新建一个前端控制器。

SIDEBAR 根目录可以在任何地方

只有 web 根目录(symfony 项目的 web/目录)里的脚本对外界公开。前端控制器脚本，图片，样式表，还有 JavaScript 文件是公开的。其他文件必须放在服务器 web 根目录之外--也就是说其他任何地方。

前端控制器通过 SF_ROOT_DIR 这个路径访问项目的非公开的文件。一般来说，项目根目录就是 web/目录的上一层目录。但是你可以选择一个完全不同的文件结构。假设你的主目录结构由两个目录组成，一个公开另外一个私有：

```
symfony/    # 私有区域
  apps/
  batch/
  cache/
```



```
...
www/          # 公开区域
  images/
  css/
  js/
  index.php
```

在这种情况下，项目根目录是 `symfony/` 目录。所以 `index.php` 前端控制器只要这样定义整个应用程序就可以工作了：

```
define('SF_ROOT_DIR', dirname(__FILE__).'/../symfony');
```

第 19 章将会告诉你更多如何修改 `symfony` 来实现特殊的目录结构的信息。

主应用程序配置

主应用程序配置存放在 `myproject/apps/myapp/config/` 目录下的文件里：

- `app.yml`：这个文件存放应用程序相关的配置信息，包括定义业务或者程序逻辑的全局变量，这些都不需要存放在数据库里。税率，运费，e-mail 地址等经常存放在这个文件。这个文件默认是空的。
- `config.php`：这个文件引导应用程序，它会做所有的基础初始化来启动应用程序。这里你可以定制目录结构或者增加应用程序相关的常量(详见第 19 章)。他首先会包含项目的 `config.php` 文件。
- `factories.yml`：`symfony` 在这里定义处理视图、请求、回应、会话(session)等的类。如果你想用你自己的类取代 `symfony` 的类，你可以在这里定义它们。详见第 19 章。
- `filters.yml`：过滤器是在每个请求都被执行的一小段代码。这个文件用来定义哪些过滤器需要被执行，每个模块都可以改写过滤器配置。详见第 6 章。
- `logging.yml`：这个文件定义哪些情况需要被记录到日志里，从而管理与调试应用程序。详见第 16 章。
- `routing.yml`：路由规则，能把难以理解的不好记忆的 URL 变成“漂亮”的直观形式。此配置文件用来存放这些信息。每个新应用程序都会有一些默认路由规则。详见第 9 章。
- `settings.yml`：这个文件存放 `symfony` 应用程序的主要配置信息。你的应用程序是否使用国际化功能，它的默认语言，请求 timeout 时间，是否开启缓存功能等都在这个文件里面定义。只要改变这个文件里的一行代码，你就可以关闭网站来执行维护升级。这些设置还有它们的用法详见第 19 章。
- `view.yml`：这个文件里定义默认视图的结构(布局的名称，标题，还有 meta tag；默认载入的样式表及 Javascript；默认的 content-type 等)。还有默认的 meta 与标题标签。详见第 7 章。这些配置信息可以在模块里改写。

国际化配置

国际化的应用程序可以显示多种语言。这需要特殊的配置。国际化配置信息存放在如下两个地方：

- 应用程序 `config/` 目录里的 `i18n.yml`：这个文件定义一般的翻译设置，例如原文的语言、在数据库还是文件里面存放翻译信息还有翻译信息的格式等。
- 应用程序 `i18n/` 目录里的翻译文件：这些文件基本上是字典，里面包含程序模板里面出现的所有文字的翻译，这样切换语言的地方就会显示对应的翻译。

注意开启 `i18n`(国际化)功能需要在 `setting.yml` 文件里面设置。详见第 13 章。

其它应用程序配置

还有一部分配置文件在 `symfony` 的安装目录里(在 `$sf_symfony_data_dir/config/`)，这些文件在所有的项目配置目录里都找不到。这是一些很少需要修改或者对于全部项目共用的配置信息。不过，如果你需要改动它们，只要在你的 `myproject/apps/myapp/config/` 里建立一个相同名字的空文件，然后在里面改写你要修改的配置信息就可以了。应用程序里的配置信息总是优先于框架的配置信息。下面是 `symfony` 安装目录的 `config/` 目录里的文件：

- `autoload.yml`：此文件包含了自动载入功能的配置信息。这个功能帮你从特定的目录自动载入你写的类。详见第 19 章。
- `constants.php`：此文件包含了默认的应用程序文件结构。请使用应用程序的 `config.php` 来覆盖这些配置信息。详见第 19 章。
- `core_compile.yml` 和 `bootstrap_compile.yml`：这两个文件记录启动应用程序(在 `bootstrap_compile.yml` 里)和处理请求(在 `core_compile.yml` 里)需要载入哪些类。这些类被压缩在一个没有注释的 PHP 文件里，这样可以最小化文件处理从而加快执行速度(每个请求只载入 1 个文件而不是 40 多个文件)。这在没有安装 PHP 加速器的时候特别有效。优化技术详见第 18 章。
- `config_handlers.yml`：在这个文件里你可以增加或者修改处理配置文件的工具。详见第 19 章。
- `php.yml`：这个文件用来检查 `php.ini` 里的配置信息是否满足程序需要，并且可以帮你覆盖 `php.ini` 的配置。详见第 19 章。

模块配置

默认情况下，模块没有特别的配置信息。不过，如果你需要，你可以为某个模块覆盖应用程序级的配置。例如，你需要修改一个模块里所有动作的 HTML 描述信息，或是载入一个特定的 Javascript 文件。你可以选择针对某个特定的模块增加新参数来实现保护性封装。

可能你已经猜到，模块配置文件必须放在 `myproject/apps/myapp/modules/mymodule/config/` 目录里，模块配置信息有下面这几个文件：

- `generator.yml`：根据数据库表自动生成的模块(脚手架与管理后台)会用这个文件，它用来定义界面怎么显示行和列，用户可以执行哪些操作(过滤器，排序，按钮等)。详见第 14 章。
- `module.yml`：这个文件包含模块的特殊参数(相当于 `app.yml`，但这是模块级的)，还有动作的配置信息。详见第 6 章。
- `security.yml`：这个文件用来给动作设置访问限制。你可以在这里设置哪个页面只能给注册用户看或是一部分有特殊权限的注册用户看。详见第 6 章。
- `view.yml`：这个文件包含模块的一个或者所有动作的视图配置信息。它会覆盖应用程序级的 `view.yml`，详见第 7 章。
- 数据验证文件：虽然这些用来验证表单输入数据的 YAML 数据验证文件存放在 `validate/` 目录而不是 `config/` 目录里，它们仍然属于模块配置文件。详见第 10 章。

大多数模块配置文件能让你为一个模块的所有视图或者动作定义参数，也可以只为模块里的一部份视图或者动作定义参数。

SIDEBAR 文件太多了？

可能应用程序里的配置文件太多了，使你受到了打击。不过请注意：

大多数时候你都不用修改配置，因为默认的配置就能够满足大部分的需求。每个配置文件与一个特定的功能相关，以后的章节会一个一个详细介绍它们的用法。当你关注某一个配置文件的时候，你就能清楚的了解它的用途还有它的组织形式。对于专业 web 开发，默认的配置不会经常被完全重写。配置文件使你不用修改一行代码轻易地改变 `symfony` 的功能。想想看要达到同样的目的需要多少 PHP 代码吧。如果所有的配置文件都存放在同一个文件里，那么这个文件不仅仅会变得完全无法阅读，同样你也无法在不同的级别重新定义配置信息(请看本章后面的“配置层叠”小节)。配置系统是 `symfony` 重大优点之一，它使 `symfony` 适合于几乎所有类型的 web 应用程序，不仅限于这个框架原本的设计目的。

环境

在开发应用程序的过程中，你可能同时需要好几套配置信息。例如，开发过程中用来测试的数据库配置信息，还有生产环境中正式的数据库配置信息。`symfony` 为满足同时使用不同配置信息的需求，提供了不同的环境。

什么是环境？

一个应用程序可以在不同的环境中运行。不同的环境共享相同的 PHP 代码(前端控制器除外)，但是配置信息可能完全不同。symfony 为每个应用程序提供三种默认的环境：生产(prod)、测试(test)和开发(dev)。只要你愿意你可以无限制的增加环境的数量。

所以基本上，环境与配置是同义词。例如，测试环境会记录警告与错误，生产(prod)环境只记录错误。缓存加速功能在开发(dev)环境下通常是关闭的，但是在测试(test)与生产(prod)环境下开启。开发与测试环境所需要的测试数据，存放在与生产环境不同的数据库里。所以两种环境的数据库配置会有所不同。所有的环境可以在一台机器上共存，不过通常一台正式的服务器只包含生产(prod)环境。

在开发环境下，日志与调试功能都是开启的，因为排除问题比性能更重要。相反，生产环境的配置默认是为性能优化的，所以生产环境会关闭一些功能。建议呆在开发环境直到你对开发的功能满意为止，然后切换到生产环境测试速度。

测试环境又与开发与生产环境有所不同。你只能通过命令行来访问这个环境，通常是做功能测试和执行批处理脚本。因此，测试环境与生产环境接近，但是不能通过浏览器访问。它能模拟 cookie 与其它 HTTP 相关的组件。

改变你正在访问的应用程序的环境，只要改变前端控制器就可以了。到目前为止，你只见过开发环境，因为例子里的 URL 都是开发环境前端控制器的：

http://localhost/myapp_dev.php/mymodule/index

不过，如果你想看看应用程序在生产环境中的样子，可以执行生产环境的前端控制器：

<http://localhost/index.php/mymodule/index>

如果你的 web 服务器支持 mod_rewrite，你甚至可以在 web/.htaccess 里自定义 symfony 重写规则。这些规则把生产环境的前端控制器作为默认的执行脚本能让 URL 看上去像这样：

<http://localhost/mymodule/index>

SIDEBAR 环境与服务器

不要把环境与服务器的概念混淆了。在 symfony 里，不同的环境是指不同的配置信息，对应不同的前端控制器(执行请求的脚本)。不同的服务器对应 URL 里的不同域名。

http://localhost/myapp_dev.php/mymodule/index

服务器	环境
-----	----

通常，开发人员在一台开发服务器上工作，这台服务器不与互联网相连，所有的服务与 PHP 配置文件都可以自由修改。到了发布应用程序的时候，程序文件会传到生产服务器上然后最终用户才能访问得到。

这意味着同一台服务器上可以有很多个环境。例如，你甚至可以在你的开发服务器上运行生产环境。不过，大多数时候，生产服务器上只能访问生产环境，这样可以避免服务器的配置信息泄露以减少安全方面的风险。

定义一个新环境，不需要建立目录或者使用 `symfony` 命令行工具。只要建立一个新的前端控制器，修改这个前端控制器里面定义的环境名就可以了。这个环境会继承所有默认配置信息和所有环境的共同配置信息。下一章会有详细介绍。

配置层叠

同样的配置信息可以在不同的地方定义多次。例如，你想把你的程序的所有页面的 `mime-type` 定义成 `text/html`，只有一个 `rss` 模块例外，这个模块需要 `text/xml` 的 `mime-type`。`symfony` 可以让你在 `myapp/config/view.yml` 里面写第一个定义，然后在 `myapp/modules/rss/config/view.yml` 里面写第二个定义。配置系统了解模块级别的定义一定要覆盖应用程序级的定义。

实际上，`symfony` 具有如下的配置级别：

- 粒度级别：
 - 框架里的默认配置
 - 整个项目的全局配置（在 `myproject/config/` 里）
 - 项目中应用程序的配置（在 `myproject/apps/myapp/config/` 里）
 - 模块的配置（在 `myproject/apps/myapp/modules/mymodule/config/` 里）
- 环境级别：
 - 针对某一个环境
 - 所有环境

所有可以自定义的属性里，有一些是与环境有关的。因此，很多 YAML 配置文件是按照环境分成了好几段，最后一段针对所有环境。因此一个典型的 `symfony` 配置文件类似于例 5-12。

例 5-12 - `symfony` 配置文件的结构

```
# 生产环境设置
prod:
    ...

# 开发环境设置
dev:
```

```

...

# 测试环境设置
test:
  ...

# 自定义环境设置
myenv:
  ...

# 所有环境的设置
all:
  ...

```

另外，symfony 框架本身定义的默认值并不在项目的目录里，它们在你的 symfony 的 `$sf_symfony_data_dir/config/` 目录里。默认的配置信息在例 5-13 的文件里设置。所有的应用程序都会继承到这些设置。

例 5-13 - 默认配置信息，在 `$sf_symfony_data_dir/config/settings.yml` 里

```

# Default settings:
default:
  default_module:      default
  default_action:      index
  ...

```

这些配置会在项目，应用程序，模块的配置信息里面以注释的形式反复出现，如例 5-14 所示，这样你就可以知道这些默认值并且可以修改它们。

例 5-14 - 默认配置信息，在 `myapp/config/settings.yml` 中出现以供参考

```

#all:
#  default_module:      default
#  default_action:      index
  ...

```

这意味着一个属性可以多次定义，最后的取值取决于层叠的结构。任何一个特定环境里定义的参数优先于所有环境里定义的参数，所有环境里的参数优先于默认配置。模块配置里的参数优先于应用程序级里定义的同样参数，应用程序里的参数优先于项目级。这可以通过下面的优先级列表来表示：

1. 模块
2. 应用程序
3. 项目
4. 特定的环境

- 5. 所有环境
- 6. 默认

配置缓存

执行时解析 YAML 处理配置文件的层叠结构会增加每次请求的负担。symfony 内建了配置文件缓存机制来提高请求速度。

不管什么格式的配置文件都需要一些特别的类来处理，又叫处理器，这些配置文件被转换成快速执行的 PHP 代码。开发环境里，处理着每次请求都会去检查配置文件的变化，这样提高交互性。它们解析改变的配置文件使你能马上看到 YAML 改变的效果。但是在生产环境，这样的处理只在第一次请求时进行，处理得到的 PHP 代码被保存下来给后面的请求使用。这样能提高性能，因为生产环境里的每次请求只需要执行一些优化过的 PHP 代码。

例如，如果 app.yml 文件内容如下：

```
all:                # 所有环境的设置
  mail:
    webmaster:       webmaster@example.com
```

那么 cache/目录下的 config_app.yml.php 文件, 将包括下面的内容：

```
<?php

sfConfig::add(array(
  'app_mail_webmaster' => 'webmaster@example.com',
));
```

这样，大多数时候，symfony 不需要去解析 YAML 文件，只需要执行 cache 里的配置信息就可以了。不过在开发环境, symfony 会自动比较 YAML 文件还有配置缓存的修改时间，只重新处理上次请求后修改过的配置文件。

这是 symfony 与其他很多 PHP 框架相比的一个主要优势，这些 PHP 框架里的每次请求都会去处理配置文件，即使是生产环境。与 Java 不同，PHP 不会在请求之间共享执行状态。其他依赖 XML 配置文件的框架在每次请求时处理 XML 性能损失很大。symfony 不存在这个问题，配置文件带来的速度影响很小。

这样的机制带来一个重要的问题，如果你改变了生产环境的配置信息，你需要强制重新解析所有你修改过的配置文件，这样改变才能生效。你只需要清除缓存就可以了，可以直接清空 cache/目录的内容，或是执行 clear-cache 这个 symfony 任务：

```
> symfony clear-cache
```

从代码里访问配置信息

所有的配置文件最终都被转换成 PHP 代码，框架会自动使用很多设置。不过，有时你需要在你的代码中(动作，模板，自定义类等)访问配置文件里定义的设置。settings.yml、apps.yml、module.yml、logging.yml 还有 i18n.yml 里的配置信息可以通过一个特殊的 sfConfig 类来访问。

sfConfig 类

你可以在程序代码里通过 sfConfig 类访问配置信息。它是一个配置信息的登记处，它有一些简单的存取方法，这些存取方法可以在程序的任何地方使用。

```
// 取得一个设置
parameter = sfConfig::get('param_name', $default_value);
```

注意你也可以在 PHP 代码里定义或者覆盖一个设置：

```
// 定义一个设置
sfConfig::set('param_name', $value);
```

参数的名字由几部分组成，中间用下划线分割，顺序如下：

- 与配置文件名有关的前缀 (sf_ 代表 settings.yml, app_ 代表 app.yml, mod_ 代表 module.yml, sf_i18n_ 代表 i18n.yml, sf_logging_ 代表 logging.yml)
- 父键名 (如果有)，小写形式
- 键名，小写形式

参数名字不包括环境名称，因为 PHP 代码只能访问到执行时所在的环境里定义的参数。

例如，如果你需要访问 app.yml 里定义的值，见例 5-15，你需要例 5-16 中的代码。

例 5-15 - app.yml 配置文件样本

```
all:
  version:      1.5
  .general:
    tax:        19.6
  default_user:
    name:       John Doe
  mail:
```



```
webmaster:  webmaster@example.com
contact:    contact@example.com
dev:
  mail:
    webmaster:  dummy@example.com
    contact:    dummy@example.com
```

例 5-16 - 在 dev 环境从 PHP 代码里访问配置信息

```
echo sfConfig::get('app_version');
=> '1.5'
echo sfConfig::get('app_tax');    // 请注意分类的头会被忽略掉
=> '19.6'
echo sfConfig::get('app_default_user_name');
=> 'John Doe'
echo sfConfig::get('app_mail_webmaster');
=> 'dummy@example.com'
echo sfConfig::get('app_mail_contact');
=> 'dummy@example.com'
```

所以 symfony 的配置信息有所有 PHP 常量的优点,但是没有 PHP 常量的缺点,因为 symfony 配置的值可以改变。

所以,用来给应用程序设置框架设置的 settings.yml 文件,相当于一系列的 sfConfig::set()调用。例 5-17 会被解释为 例 5-18。

例 5-17 - 不完整的 settings.yml

```
all:
  .settings:
    available:          on
    path_info_array:     SERVER
    path_info_key:       PATH_INFO
    url_format:          PATH
```

例 5-18 - symfony 处理 settings.yml 文件的结果

```
sfConfig::add(array(
  'sf_available' => true,
  'sf_path_info_array' => 'SERVER',
  'sf_path_info_key' => 'PATH_INFO',
  'sf_url_format' => 'PATH',
));
```

settings.yml 文件里面设置的含义请参考第 19 章。

自定义应用程序配置与 app.yml

大部分与程序功能有关的设置存放在 `app.yml` 文件里，这个文件在 `myproject/apps/myapp/config/` 目录。`app.yml` 与环境有关，默认是空的。把所有你需要很容易修改的设置放在这个文件里，在代码里用 `SfConfig` 类访问它们。如例 5-19。

例 5-19 - 这个 `app.yml` 给指定的网站定义接受的信用卡类型

```
all:
  creditcards:
    fake:          off
    visa:          on
    americanexpress: on

dev:
  creditcards:
    fake:          on
```

想要知道当前环境是否接受 `fake` 信用卡，需要这么写代码：

```
$sfConfig->get('app_creditcards_fake');
```

TIP 当你要定义一个常量或者一个设置的时候，考虑一下把它放在 `app.yml` 里面会不会更好。在这里存放应用程序配置很方便。

如果你的自定义参数用 `app.yml` 的语法难以处理，你可以考虑自己定义一套语法。这样你可以把配置信息存在一个新的文件里，用新的处理器解析配置文件。配置文件处理器的资料详见第 19 章。

更多使用配置文件的技巧

在开始写你自己的 YAML 文件之前，有一些最后的技巧需要掌握。这些技巧可以避免配置信息的重复及其如何处理你自己的 YAML 格式。

在 YAML 文件里使用常量

一些配置设置的值取决于其他的设置。为了避免重复设置同样的值，`symfony` 支持在 YAML 文件里使用常量。如果遇到%包起来的大写形式的设置名(可以通过 `$sfConfig->get()` 取得值)，配置文件处理器会用这个设置的当前值来替换这个常量。见例 5-20。

例 5-20 - 在 YAML 文件里使用常量，以 `autoload.yml` 为例

```
autoload:
  symfony:
    name:          symfony
    path:          %SF_SYMFONY_LIB_DIR%
    recursive:     on
    exclude:       [vendor]
```

`path` 参数的值会是执行 `sfConfig::get('sf_symfony_lib_dir')` 的结果。如果一个配置文件依赖于另一个配置文件，被依赖的配置文件必须先被解析(请查看 `symfony` 的源代码来了解配置文件载入的顺序)。`app.yml` 是最后被解析的文件之一，所以你可以在这个文件里使用其它文件里定义的设置。

在配置文件里使用脚本

有可能你的配置信息与外部参数有关(例如数据库或者其他配置文件)。为了解决这种问题，`symfony` 在把配置文件传给 YAML 处理器之前先用 PHP 来解析配置文件。这意味着你可以在 YAML 文件里使用 PHP 代码，如例 5-21。

例 5-21 - YAML 文件可以包含 PHP

```
all:
  translation:
    format: <?php echo sfConfig::get('sf_i18n') == true ? 'xliff' :
'none' ?>
```

但是请注意配置文件在请求周期早期就被处理了，所以你不能使用 `symfony` 内建的方法或者函数。

CAUTION 在生产环境中，配置文件会被缓存，所以配置文件只会在清除缓存后被处理(并执行)一次。

浏览你的 YAML 文件

如果你想直接读取 YAML 文件，你可以使用 `sfYaml` 类。它是一个可以把 YAML 文件转化成 PHP 数组的 YAML 解析器。例 5-22 是一个 YAML 文件的例子，例 5-23 是前面 YAML 文件的解析结果。

例 5-22 - `test.yml`

```
house:
  family:
```

```

    name:      Doe
    parents:   [John, Jane]
    children:  [Paul, Mark, Simone]
address:
    number:    34
    street:    Main Street
    city:      Nowheretown
    zipcode:   12345

```

例 5-23 - 使用 sfYaml 类把 YAML 转换成一个数组

```

$test = sfYaml::load('/path/to/test.yml');
print_r($test);

```

```

Array(
    [house] => Array(
        [family] => Array(
            [name] => Doe
            [parents] => Array(
                [0] => John
                [1] => Jane
            )
            [children] => Array(
                [0] => Paul
                [1] => Mark
                [2] => Simone
            )
        )
        [address] => Array(
            [number] => 34
            [street] => Main Street
            [city] => Nowheretown
            [zipcode] => 12345
        )
    )
)

```

总结

symfony 配置系统使用的 YAML 语言简单并且可读性强。多种环境与层叠结构的参数定义为开发者提供了多种选择。一些配置文件可以从代码里通过 sfConfig 类访问，特别是 app.yml 里的应用程序配置。

的确, `symfony` 有很多配置文件, 不过这使 `symfony` 适用性更强。注意除非你的应用程序需要高级别的定制, 否则你根本不需要去修改它们。

第 6 章 深入了解控制器层

在 symfony 中，控制器是连接商业逻辑和视图层的程序。根据不同的功能，它又被细分为几个小部分：

- 前端控制器是应用程序的唯一入口。它载入配置文件并且指定将被执行的动作。
- 动作包含应用逻辑。它们首先确定请求的完整性，然后为表现层准备好数据。
- 通过请求，应答，session 对象，我们可以获取请求参数，应答 HTTP 头，和持久性的用户数据。这些数据经常被用在控制器层。
- 每一个请求都要执行过滤器程序，这个程序将在动作的前后执行。例如，安全和确认过滤器是网络编程经常要用到的。你可以延伸架构去创作自己的过滤器。

这一章将介绍这些组件。不要担心。一个简单的页面，你只需要在动作的类里面写几行代码而已。其他的控制器层组件仅仅被用于一些特定的情况。

前端控制器

前端控制器接收并且处理所有的请求。所以前端控制器是整个应用程序的唯一入口。

当前端控制器接收到一个请求，路由选择系统将根据用户所用的 URL 连接相应的动作和模块。例如，下列的 URL 调用 index.php 脚本(这是前端控制器)，它可以被理解为调用 mymodule 模块里的 myAction 动作：

<http://localhost/index.php/mymodule/myAction>

如果你对 symfony 的内部结构没有兴趣，以上对前端控制器的认识已经足够了。它是 symfony MVC 架构中不可缺少的组件，但是你很少去更改它。你可以跳过下一节，除非你想更深入地了解前端控制器的结构。

前端控制器的工作细节

前端控制器处理请求的时候不仅仅是分配将需要执行的动作。其实，它还执行所有动作的公共代码，它们包括：

1. 定义核心常量。
2. 定位 symfony 程序库。
3. 调入并初始化核心架构的类。
4. 调入配置文件。

5. 处理请求的 URL 并且指定将被执行的动作和请求参数。
6. 如果动作不存在， 转发给专门接收 404 错误信息的动作。
7. 启动过滤器 （例如， 如果请求需要被认证）。
8. 执行过滤器， 第一回。
9. 执行所需要的动作并将结果提交给视图。
10. 执行过滤器， 第二回。
11. 输出应答。

默认的前端控制器

`index.php` 是默认的前端控制器。它是一个非常简单的 PHP 文件，在 `web/` 文件夹里。请看例 6-1。

例 6-1 - 默认的前端控制器

```
<?php

define('SF_ROOT_DIR',    realpath(dirname(__FILE__).' /..'));
define('SF_APP',          'myapp');
define('SF_ENVIRONMENT', 'prod');
define('SF_DEBUG',        false);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.
SF_APP.DIRECTORY_SEPARATOR.'config'.DIRECTORY_SEPARATOR.'config.php');

sfContext::getInstance()->getController()->dispatch();
```

首先是上一节讲的第一步，定义核心常量。然后前端控制器调入 `config.php`，也就是上面的第二步到第四步。执行 `sfController` 对象（它是 `symfony MVC` 架构里的核心控制器对象）中的 `dispatch()` 函数处理请求，也就是上面的第五步到第七步。最后一步是执行过滤器。这部分稍后再讲解。

调用其他的前端控制器来切换环境

一个环境只能有一个前端控制器。事实上，我们应该说一个前端控制器确定了一个环境。`SF_ENVIRONMENT` 常量是环境的定义。

如果你想在项目中切换另一个环境，只需要选择另一个前端控制器。当你用 `symfony init-app` 创建一个新项目时，在生产环境中默认的前端控制器是 `index.php`。而在开发环境中默认的前端控制器是 `myapp_dev.php`（如果你的项目叫 `myapp`）。如果 URL 没有指明前端控制器文件名，`mod_rewrite` 将用 `index.php` 作为默认值。所以下面这两个 url 在生产环境中是相同的（`mymodule/index`）。

<http://localhost/index.php/mymodule/index>

<http://localhost/mymodule/index>

与此页面相同的开发环境的 URL 为：

http://localhost/myapp_dev.php/mymodule/index

建立一个新的环境非常容易，只需要增添一个新的前端控制器。比如你想让客户检验你的项目，你可以建立一个展示环境。你只需要拷贝一份

web/myapp_dev.php，命名为 web/myapp_staging.php。然后把常量 SF_ENVIRONMENT 改为 staging。最后，在所有的配置文件里添加 staging: 部分并赋值，请参看例 6-2。

例 6-2 - 样本 app.yml，展示（staging）环境的设置

```
staging:
  mail:
    webmaster:    dummy@mysite.com
    contact:      dummy@mysite.com
all:
  mail:
    webmaster:    webmaster@mysite.com
    contact:      contact@mysite.com
```

如果你想看一下新环境带来的变化，调用相应的前端控制器：

http://localhost/myapp_staging.php/mymodule/index

批处理文件

如果你想在命令行或 crontab 上执行一个脚本并能享用 symfony 的类和其他功能，比如发出大批电子邮件或通过大量计算而定时地更新你的模型。对于这种脚本，你需要包括前端控制器里的前几行代码。例 6-3 显示批处理脚本文件的开头部分。

例 6-3 - 批处理脚本文件样本

```
<?php

define(' SF_ROOT_DIR',    realpath(dirname(__FILE__).'/.'));
define(' SF_APP',         'myapp');
define(' SF_ENVIRONMENT', 'prod');
define(' SF_DEBUG',       false);
```



```
require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.SF_APP.DIRECTORY_SEPARATOR.'config'.DIRECTORY_SEPARATOR.'config.php');
```

```
// add code here
```

你会发现只有 `sfController` 对象中的 `dispatch()` 方法被去掉了。这个方法只能用在互联网的服务器上。它不支持批处理脚本文件。 建立一个项目和环境后你可以进入一个相应的配置文件。包括项目的 `config.php` 设定相对的关系和自动装载。

TIP `symfony` 命令行提供了一个 `init-batch` 任务，这个任务可以自动在 `batch/` 目录里建立与例 6-3 中类似的批处理脚本的框架，只要传递应用程序名，环境名，还有批处理名三个参数就可以了。

动作 (Actions)

动作是整个项目的核心，因为它们包含所有的应用逻辑。它们用模型提供的数据为视图层的变量赋值。 当你在 `symfony` 的项目中执行一个请求时，URL 就确定了动作和请求参数。

动作类

`moduleNameActions` 是 `sfActions` 的子类(通过继承)。动作是 `moduleNameActions` 里命名为 `executeActionName` 的方法，它们组成不同的模块。 模块里的动作类被存在 `actions.class.php` 文件里。这个文件在模块的文件夹 `actions/` 里。

例 6-4 是一个 `actions.class.php` 文件的例子。这个模块只有一个动作 `index`。

例 6-4 - 动作类样本,

`apps/myapp/modules/mymodule/actions/actions.class.php`

```
class mymoduleActions extends sfActions
{
    public function executeIndex()
    {

    }
}
```

CAUTION PHP 里方法名是不区分大小写的，不过在 `symfony` 里需要区分。所以别忘了动作方法必须以小写的 `execute` 开头，后面是首字母大写的动作名。

每个请求必须标明动作和模块名，所以你必须附上模块名和动作名作为参数。通常，你需要附上 `module_name/action_name`。这就是说例 6-4 里的动作可以被这个 URL 调用。

<http://localhost/index.php/mymodule/index>

添加更多的动作就意味着在 `SfActions` 的对象中添加更多 `execute` 的方法。请参看例 6-5。

例 6-5 - 包含两个动作的动作类，
`myapp/modules/mymodule/actions/actions.class.php`。

```
class mymoduleActions extends SfActions
{
    public function executeIndex()
    {
        ...
    }

    public function executeList()
    {
        ...
    }
}
```

如果动作类增长得太大了，你或许应该重构你的程序把一些代码放入模型里。动作应该保持很短（不超过几行），所有的商业逻辑应该放入模型里。

另外，如果模块里的动作太多，可以考虑把它分成两个模块。

SIDEBAR `symfony` 代码编写规范

你可能会发现在本书给出的代码例子里，开始和结束的大括号（{和}）都独占一行。这个规范使代码更容易阅读。

`symfony` 框架的其他代码规范包括，缩进使用两个空格，而不实用 `tab`。这是由于不同的编辑器的 `tab` 的宽度不一样，还有混合了空格和 `tab` 缩进的程序很难阅读。

`symfony` 的核心和生成的 PHP 文件都不包括 `<?>` 关闭标签。因为关闭标签不是必须的，而且如果在关闭标签之后包含空格会造成问题。

另外，如果你特别注意，你会发现 `symfony` 程序的行从来不会以空格结尾。原因很简单：因为在 Fabien 的编辑器里空格结尾的行看起来很丑。

另一种动作类语法

还有一个方法是把动作分开，一个文件只有一个动作。在这种情况下，每一个动作类扩展 `SfAction`（而不是 `SfActions`）并且命名为 `actionNameAction`。动作被命名为 `execute`。文件名和类名相同。所以例 6-5 可以分成两个文件，例 6-6 和 6-7。

例 6-6 - 单个动作文件，
`myapp/modules/mymodule/actions/indexAction.class.php`

```
class indexAction extends SfAction
{
    public function execute()
    {
        ...
    }
}
```

例 6-7 - 单个动作文件，
`myapp/modules/mymodule/actions/listAction.class.php`

```
class listAction extends SfAction
{
    public function execute()
    {
        ...
    }
}
```

在动作里获取信息

在动作类里你可以获取控制器相关的信息和 `symfony` 的核心对象。例 6-8 展示它的用法

例 6-8 - `SfActions` 常用方法

```
class mymoduleActions extends SfActions
{
    public function executeIndex()
    {
```

```

// Retrieving request parameters
$password = $this->getRequestParameter('password');

// Retrieving controller information
$moduleName = $this->getModuleName();
$actionName = $this->getActionName();

// Retrieving framework core objects
$request = $this->getRequest();
$userSession = $this->getUser();
$response = $this->getResponse();
$controller = $this->getController();
$context = $this->getContext();

// Setting action variables to pass information to the template
$this->setVar('foo', 'bar');
$this->foo = 'bar'; // Shorter version

}
}

```

SIDEBAR 环境单例 (context singleton)

你已经看到了，在前端控制器里，有一个 `sfContext::getInstance()` 调用。在动作里，`getContext()` 方法也会返回同样的单例。这个很有用的对象保存了与某个请求有关的所有 symfony 核心对象，并且为它们提供了读取方法：

`sfController`: 控制器对象 (`->getController()`) `sfRequest`: 请求对象 (`->getRequest()`) `sfResponse`: 应答对象 (`->getResponse()`) `sfUser`: 用户 session 对象 (`->getUser()`) `sfDatabaseConnection`: 数据库链接对象 (`->getDatabaseConnection()`) `sfLogger`: 日志对象 (`->getLogger()`) `sfI18N`: 国际化对象 (`->getI18N()`)

可以在代码的任何地方调用 `sfContext::getInstance()`。

动作结束

在动作结束前有几种状况。动作返回的数据将决定如何显示视图。在 `sfView` 类里的常量决定哪一个模板被用于展示动作的结果。

如果有一个默认的视图（最普遍的情况），动作的结尾应该是这样的。

```
return sfView::SUCCESS;
```

symfony 将寻找 `actionNameSuccess.php` 模板。这是动作的默认方式，所以即使你省略了 `return` 这一行，symfony 一样会寻找并使用 `actionNameSuccess.php` 模板。即便动作是空的也一样。例 6-9 是动作结束的例子。

例 6-9 - 动作返回数据给 `indexSuccess.php` 和 `listSuccess.php` 模板

```
public function executeIndex()
{
    return sfView::SUCCESS;
}
```

```
public function executeList()
{
}
```

如果有错误，动作应该这样结尾：

```
return sfView::ERROR;
```

symfony 就会去寻找 `actionNameError.php` 模板。

如果你想用一个特别的视图，你可以这样结尾：

```
return 'MyResult';
```

symfony 就会去寻找 `actionNameMyResult.php` 模板。

如果根本就没有或不需要视图--例如，批处理文件的执行--应该这样结尾：

```
return sfView::NONE;
```

在这种情况下，视图层就不会被执行了。这就意味着你完全可以越过视图层直接从动作输出 HTML 代码。请参看 6-10，symfony 提供一个 `renderText()` 方法。这个方法在响应速度要求很高的动作中会非常有用，比如和 Ajax 的互动。我们将在第 11 章讨论。

例 6-10 - 用 `sfView::NONE` 越过视图层直接输出回应

```
public function executeIndex()
{
    echo "<html><body>Hello, World!</body></html>";

    return sfView::NONE;
}
```

// Is equivalent to

```
public function executeIndex()
{
    return $this->renderText("<html><body>Hello, World!</body></html>");
}
```

在一些情况下，你需要回复一个空的应答但要有 HTTP 头（特别是 X-JSONHTTP 头）。通过 sfResponse 对象定义 HTTP 头将在下一章讨论。返回 HTTP 头 sfView::HEADER_ONLY 常量，请参看例 6-11。

例 6-11 - 避开视图层，但应答有 HTTP 头

```
public function executeRefresh()
{
    $output = '<"title", "My basic letter", ["name", "Mr Brown">';
    $this->getResponse()->setHTTPHeader("X-JSON", '('.$output.')');

    return sfView::HEADER_ONLY;
}
```

如果动作需要一个特殊的模板，去掉 return 声明，用 setTemplate() 方法。

```
$this->setTemplate('myCustomTemplate');
```

跳到另一个动作

在一些情况下，一个动作结束时需要执行另一个动作。例如，一个处理表单提交的动作在更新数据库后通常被跳转到另一个动作上。第二个例子是动作别名：动作 index 经常展示一个表，其实它转给了动作 list。

动作类里提供了两个方法可以执行另一个动作：

- 如果动作转发给另一个动作：

```
$this->forward('otherModule', 'index');
```

- 如果跳转到另一个动作：

```
$this->redirect('otherModule/index');
$this->redirect('http://www.google.com/');
```

NOTE forward 与 redirect 之后的动作代码不会被执行。这一点上与 return 语句是一样的。它们会抛出一个 sfStopException 异常来停止动作的执行；这个异常稍后会被 symfony 截获然后忽略。

选择转发或跳转有时并不容易。为了做出最好的选择，请记住转发在应用程序内部进行，所以对于用户来说比较直接易懂。在用户的眼里，浏览器显示的 URL 和用户请求的 URL 是一样的。相反地，跳转是一条发给用户浏览器的消息，包括一个新的请求并改变了最终的 URL。

如果一个表单通过 `method="post"` 调用动作，你应该使用跳转。最大的优点是，如果用户刷新页面，表单不会被重新提交；另外，如果用户点击后退键，浏览器会再显示表单，而不是询问用户是否要重新提交表单。

`forward404()` 是一个特殊并很常用的 `forward` 方法。它 `forward` 给 "page not found" 动作。当动作所需的请求参数不全时（比如查出一个错误的 URL），这个方法就会被用到。例 6-12 展示的例子是一个 `show` 动作需要一个 `id` 参数。

例 6-12 - 使用 `forward404()` 方法

```
public function executeShow()
{
    $article =
ArticlePeer::retrieveByPK($this->getRequestParameter('id'));
    if (!$article)
    {
        $this->forward404();
    }
}
```

TIP 如果要找错误 404 的动作和模板，可以在 `$sf_symfony_data_dir/modules/default/` 目录里找到它们。可以通过在你的应用程序里增加一个新的 `default` 模块来定制这个页面，覆盖框架里的内容，定义一个 `error404` 动作和 `error404Success` 模板就可以了。另外，还可以修改 `settings.yml` 文件里的 `error_404_module` 和 `error_404_action` 常量来使用已有的动作作为 404 页面。

经验告诉我们，在多数情况下，动作需要在做出一个判断后再转发或跳转。例如 6-12。所以 `sfActions` 类有几个方法叫 `forwardIf()`, `forwardUnless()`, `forward404If()`, `forward404Unless()`, `redirectIf()`, `redirectUnless()`。这些方法接受一个参数并且对它进行判断，如果判断结果是 `true`, `xxxIf()` 方法将被执行；如果判断结果是 `false`, `xxxUnless()` 方法将被执行。请参看例 6-13。

例 6-13 - 使用 `forward404If()` 方法

```
// 这个动作于例 6-12 中的作用相同
public function executeShow()
{
```

```

    $article =
ArticlePeer::retrieveByPK($this->getRequestParameter('id'));
    $this->forward404If(!$article);
}

// 这一个也是
public function executeShow()
{
    $article =
ArticlePeer::retrieveByPK($this->getRequestParameter('id'));
    $this->forward404Unless($article);
}

```

这些方法不但使你的程序简短而且清晰易懂。

TIP 当动作执行 `forward404()` 或者类似的方法的时候，`symfony` 会抛出 `sfError404Exception` 这个管理 404 回应的异常。这意味着如果在一个你不想访问控制器的地方显示 404 错误信息，你只要抛出一个类似的异常就可以了。

几个动作共享的代码

`symfony` 给动作命名为 `executeActionName()`（如果使用 `sfActions` 类）或 `execute()`（如果使用 `sfActions` 类）。这样 `symfony` 就能保证找到动作。你也可以添加自己的方法，只要你不以 `execute` 开头，`symfony` 就不会把这些方法当做动作。

如果你需要在执行每个动作前都要执行一段代码，你可以把这段代码放入动作类里的 `preExecute()` 方法里。你可能猜到如果你想在执行每个动作后执行一段代码，那你可以把这段代码放入 `postExecute()` 方法里。例 6-14 介绍了使用这些方法的规则。

例 6-14 - 使用 `preExecute`, `postExecute`, 和自己定义的方法

```

class mymoduleActions extends sfActions
{
    public function preExecute()
    {
        // 这里的代码会在每个动作之前执行
        ...
    }

    public function executeIndex()
    {
        ...
    }
}

```



```

    }

    public function executeList()
    {
        ...
        $this->myCustomMethod(); // 可以使用动作类里定义的方法
    }

    public function postExecute()
    {
        // 这里的代码会在每次执行完动作之后执行
        ...
    }

    protected function myCustomMethod()
    {
        // 还可以添加自己的方法，只要不以"execute"开头
        // 最好把这样的方法声明成 protected 或者 private
        ...
    }
}

```

访问请求

你或许熟悉了 `getRequestParameter('myparam')` 方法：它被用于取得请求参数值。事实上，这个方法只是调入 `getRequest()->getParameter('myparam')` 的代理方法。在动作类里可以通过 `getRequest()` 方法访问请求对象，在 symfony 里叫 `sfWebRequest`，和它所有的方法。表格 6-1 展示一些常用的 `sfWebRequest` 方法。

表格 6-1 - `sfWebRequest` 对象里的方法

名称	功能	输出例子
请求信息		
<code>getMethod()</code>	请求的方法	返回 <code>sfRequest::GET</code> 或者 <code>sfRequest::POST</code> 常量
<code>getMethodName()</code>	请求方法名	'POST'
<code>getHttpHeader('Server')</code>	某个指定的	'Apache/2.0.59 (Unix) DAV/2 PHP/5.1.6'

名称	功能	输出例子
	HTTP 头的值	
getCookie('foo')	某个 cookie 的值	'bar'
isXmlHttpRequest()	是否是 Ajax 请求?	true
isSecure()	是否是 SSL 请求?	true
请求参数		
hasParameter('foo')	请求里是否包含某个参数?	true
getParameter('foo')	某个参数的值	'bar'
getParameterHolder()->getAll()	包含所有参数的数组	
URI 相关的信息		
getUri()	完整的 URI	'http://localhost/myapp_dev.php/mymodule/myaction'
getPathInfo()	路径信息	'/module/myaction'
getReferer()	来源	'http://localhost/myapp_dev.php/'
getHost()	主机	'localhost'

名称	功能	输出例子
getScriptName()	前端控制器的路径和名字	'myapp_dev.php'
客户端浏览器信息		
getLanguages()	所有可接受的语言组成的数组	Array([0] => fr [1] => fr_FR [2] => en_US [3] => en)
getCharsets()	所有可接受的字符集组成的数组	Array([0] => ISO-8859-1 [1] => UTF-8 [2] => *)
getAcceptableContentType()	所有可接受的内容类型组成的数组	Array([0] => text/xml [1] => text/html)

*只能用于 *Prototype Javascript* 库

** 有时会被代理服务器阻拦

sfActions 类里有几个代理方法可以更简捷地访问请求方法，请参看例 6-15。

例 6-15 - 从动作类里访问 sfRequest 对象方法

```
class mymoduleActions extends sfActions
{
    public function executeIndex()
```

```

{
    $hasFoo = $this->getRequest()->hasParameter('foo');
    $hasFoo = $this->hasRequestParameter('foo'); // Shorter version
    $foo     = $this->getRequest()->getParameter('foo');
    $foo     = $this->getRequestParameter('foo'); // Shorter version
}
}

```

如果请求有附件，sfWebRequest 对象可以访问或移动这些文件，请参看例 6-16。

例 6-16 - sfWebRequest 对象知道如何处理附件

```

class mymoduleActions extends sfActions
{
    public function executeUpload()
    {
        if ($this->getRequest()->hasFiles())
        {
            foreach ($this->getRequest()->getFileNames() as $fileName)
            {
                $fileSize  = $this->getRequest()->getFileSize($fileName);
                $fileType  = $this->getRequest()->getFileType($fileName);
                $fileError  = $this->getRequest()->hasFileError($fileName);
                $uploadDir  = sfConfig::get('sf_upload_dir');
                $this->getRequest()->moveFile('file',
$uploadDir.'/'.$fileName);
            }
        }
    }
}

```

你不需要考虑服务器是否支持\$_SERVER 或\$_ENV 变量，默认值或服务器兼容问题 --sfWebRequest 的方法会帮你解决这些问题。另外，这些方法的名称简单易懂，你不再需要查看复杂的 PHP 文档，寻找有关请求方面的信息了。

用户会话

symfony 会自动地处理用户会话并保持用户请求的连续性。symfony 利用 PHP 内置的会话管理功能并增强它灵活性，使它更容易使用。

访问用户会话

在动作里，你可以通过 `getUser()` 方法访问从 `sfUser` 类生成的用户会话对象。这个类里有一个参数存储方法，可用于存储用户属性。这些存储的用户属性在用户会话结束前是有效的。请参看例 6-17。用户属性可以是任何数据结构（字符串，数组，关联数组）。每个用户都有这个功能，即使是没注册的用户。

例 6-17 - `sfUser` 对象可以存储用户属性

```
class mymoduleActions extends sfActions
{
    public function executeFirstPage()
    {
        $nickname = $this->getRequestParameter('nickname');

        // 将数据保存到用户会话
        $this->getUser()->setAttribute('nickname', $nickname);
    }

    public function executeSecondPage()
    {
        // 从用户会话中取得数据，如果取不到值则使用默认值
        $nickname = $this->getUser()->getAttribute('nickname', 'Anonymous Coward');
    }
}
```

CAUTION 你可以在用户会话中保存对象，但是请不要这么作。这是因为会话对象在不同的请求之间是通过序列化的方式保存在文件里的。从序列化的数据里重建会话的时候，对象的类必须是已经载入的，不过有的时候他们没有被载入。另外，如果在会话里保存 Propel 对象，可能会有“卡住”的对象。

就像 `symfony` 里其他的获取方法一样，`getAttribute()` 方法接受另一个参数，这个参数指定一个默认值（如果要存储的用户属性是空的）。`hasAttribute()` 方法可以用来检查用户属性是否已经被定义了。`getAttributeHolder()` 方法可以用来访问参数存储器。这也使清除用户属性变得更简单。请参看例 6-18。

例 6-18 - 删除用户会话数据

```
class mymoduleActions extends sfActions
{
    public function executeRemoveNickname()
    {
        $this->getUser()->getAttributeHolder()->remove('nickname');
    }

    public function executeCleanup()
```

```

    {
        $this->getUser()->getAttributeHolder()->clear();
    }
}

```

在模板里，你可以通过储存在`$sf_user` 变量里的 `sfUser` 对象访问用户会话属性，请参看例 6-19。

例 6-19 - 在模板里也可以直接访问用户会话属性

```

<p>
    Hello, <?php echo $sf_user->getAttribute('nickname') ?>
</p>

```

NOTE 如果只需要在当前请求里存储信息（例如，在一连串动作调用里）更适合用 `sfRequest` 类，它有 `getAttribute()` 和 `setAttribute` 方法。只有 `sfUser` 对象的属性能在不同的请求中持续存在。

短暂的属性

删除用户会话属性（如果不再需要这个属性了）是一个常见的问题。例如，在用户提交表单后，你想显示一个确认信息。当处理表单的动作需要转发到另一个动作时，把信息从一个动作传到另一个动作唯一的办法就是把信息存在用户会话的属性里。在显示确认信息之后，你需要删除这个属性。否则，这个属性就会被存入会话里，一直到会话到期。

你只需要定义短暂的属性，而不需要去删除。因为短暂的属性在接受到下一个请求后会自动删除，使用户会话更清洁。在动作里，你可以这样来定义短暂的属性：

```
$this->setFlash('attrib', $value);
```

用户看到这一页后，发出一个新的请求并触发了下一个动作。在这第二个动作里，你可以这样取得属性的值：

```
$value = $this->getFlash('attrib');
```

在显示出下一页后，短暂属性 `attrib` 就被删除了。即使在下一页里你没有调用这个属性，它也一样会被从会话里删除。

如果你在模板里调用这个属性，用 `$sf_flash` 对象：

```

<?php if ($sf_flash->has('attrib')): ?>
    <?php echo $sf_flash->get('attrib') ?>
<?php endif; ?>

```

或:

```
<?php echo $sf_flash->get('attrib') ?>
```

短暂的属性是传递信息给下一个请求很好的方法。

会话管理

对开发者来说，symfony 的会话功能完全掩饰了对会话 ID 的存储方式。但是，你仍然可以改变会话管理的默认机制。这是为高级用户设计的。

symfony 把会话 ID 存在客户端的 cookies 上。symfony 的会话 cookies 就叫 symfony，但是你可以在 factories.yml 里改变会话的名称。请参看例 6-20。

例 6-20 - 在 apps/myapp/config/factories.yml 里，改变会话的 Cookie 名称

```
all:
  storage:
    class: sfSessionStorage
  param:
    session_name: my_cookie_name
```

TIP 会话只有在 factories.yml 里的 auto_start 参数设置成 true 时(这是默认设置)才会开始开启(通过 PHP 的 session_start() 函数)。如果想手动开始用户会话，关闭会话存储机制里的这个设置就可以了。

symfony 的会话是基于 PHP 会话功能的。这就意味着如果你想用 URL 参数来代替 cookies 的话，你只需要在 php.ini 里修改 use_trans_sid 的设置。我们不主张使用这种方法。

```
session.use_trans_sid = 1
```

在服务器方面，symfony 把用户会话存在文件系统里面。如果你想把它们存在数据库里，你需要修改 factories.yml 里的 class 参数，请参看例 6-21。

例 6-21 - 修改服务器会话的存储方式，在 apps/myapp/config/factories.yml 里

```
all:
  storage:
    class: sfMySQLSessionStorage
  param:
    db_table: SESSION_TABLE_NAME      # 存放会话的表的名字
    database: DATABASE_CONNECTION    # 使用的数据库连接的名字
```

现有的会话存储类有 `sfMySQLSessionStorage`, `sfPostgreSQLSessionStorage`, 和 `sfPDOSessionStorage`, 建议用最后这个。 `database` 不是必需的配置, 它确定数据库的连接方式; `symfony` 会用 `databases.yml` (见第 8 章) 里的配置(主机, 数据库名, 用户名, 密码)去连接数据库。

在 `sf_timeout` 秒后, 会话将自动期满。这个常量的默认值是 30 分钟。当然你可以在 `settings.yml` 里修改这个常量。请参看例 6-22。

例 6-22 - 修改会话届期, 在 `apps/myapp/config/settings.yml` 里

```
default:
  .settings:
    timeout:      1800          # 会话存活的秒数
```

动作安全

可能会只有某些拥有特定权限的用户能够执行某个动作。`symfony` 提供的工具可以让我们建立有安全设置的应用程序, 用户必须认证后才能访问应用程序的某些功能或者部分。权利限制包括两个步骤: 首先要声明每一个动作的安全条件, 然后给登录的用户对应的权限。

访问限制 Access Restriction

在执行每一个动作之前, 动作都需要经过一个特殊的过滤器。这个过滤器将检查当前的用户是否有权利执行该动作。在 `symfony` 中, 权限包括两个部分:

- 用户必须被认证后才能执行有安全限制的动作。
- 证书是具名权限, 可以通过它来按组管理组织权限。

动作的权限可以在模块的 `config/` 目录的 YAML 配置文件 `security.yml` 里添加或修改。在这个文件里, 你可以设定每一个动作或所有动作的限制条件。例 6-23 是 `security.yml` 的一个示例。

例 6-23 - 设置访问限制, 在 `apps/myapp/modules/mymodule/config/security.yml` 里

```
read:
  is_secure:  off          # 所有的用户都可以执行 read 动作

update:
  is_secure:  on           # update 动作只有认证的用户可以执行

delete:
```



```
is_secure: on          # 只有认证用户
credentials: admin     # 并且有 admin 证书可以执行
```

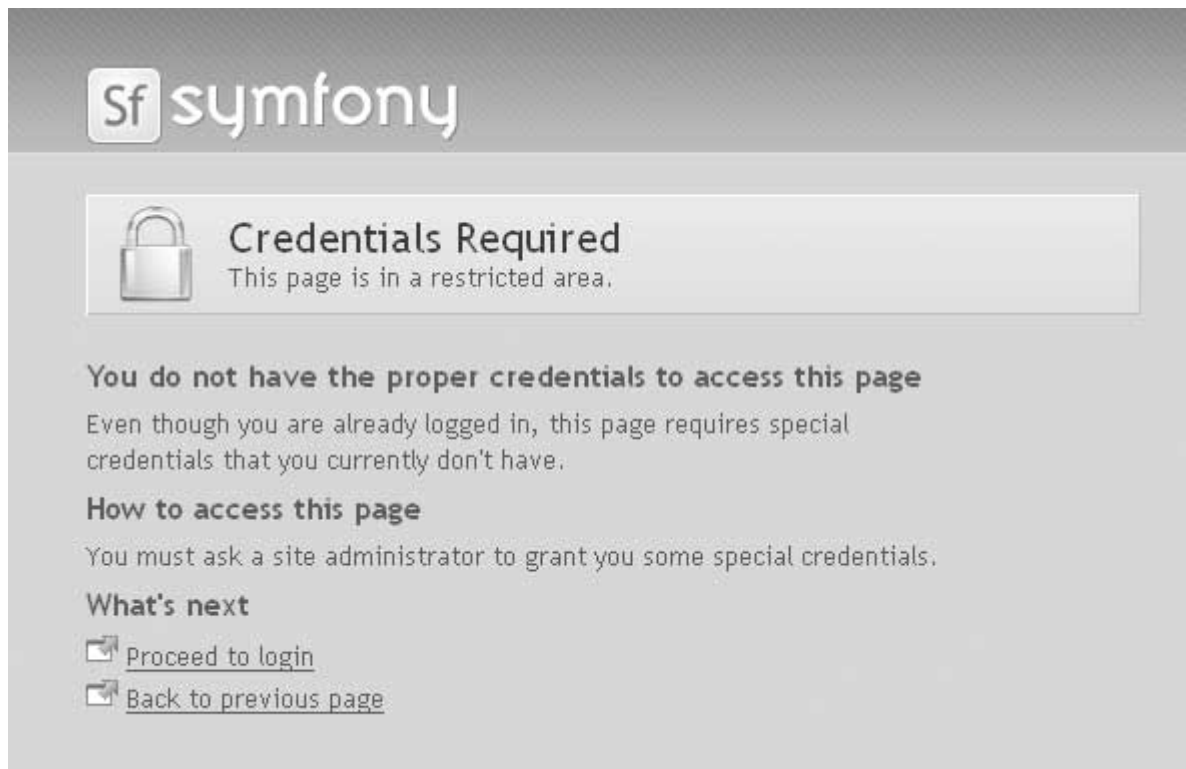
```
all:
  is_secure: off       # off 是默认值
```

动作的权利限制不是默认的。所以如果没有 `security.yml` 文件,或 `security.yml` 里面没有对动作的权利限制,任何人都可以执行所有的动作。如果已设定了 `security.yml`, `symfony` 将检查该请求是否符合被访问的动作权利限制。 当一个用户访问一个设有权利限制的动作时, 结果由用户的证书决定:

- 如果用户已被认证并有相应的证书,动作将被执行。
- 如果用户没有被认证,他将被跳转到登录的动作上。
- 如果用户已被认证但没有相应的证书,他将被跳转到一个默认的安全动作。 请参看图 6-1。

默认的登录和安全页面很简单,你可能要重新设计。如果用户没有相应的权利,你可以在 `settings.yml` 里指定被调用的动作。请参看例 6-24。

图 6-1 - 默认的安全动作



例 6-24 - 默认的安全动作在 `apps/myapp/config/settings.yml` 里设置

```
all:
  .actions:
```

login_module:	default
login_action:	login
secure_module:	default
secure_action:	secure

访问授权

如果要调用设有权限的动作，用户必须被认证后并有相应的证书。你可以用 `sfUser` 对象扩展用户的权限。`setAuthenticated()` 方法可以改变用户的认证状态。例 6-25 是一个用户认证的简单例子。

例 6-25 - 设置用户的认证状态

```
class myAccountActions extends sfActions
{
    public function executeLogin()
    {
        if ($this->getRequestParameter('login') == 'foobar')
        {
            $this->getUser()->setAuthenticated(true);
        }
    }

    public function executeLogout()
    {
        $this->getUser()->setAuthenticated(false);
    }
}
```

认证稍微有一点复杂，你可以检查，添加，删除认证。例 6-26 描述了 `sfUser` 类里的认证方法。

例 6-26 - 在动作里处理用户的认证

```
class myAccountActions extends sfActions
{
    public function executeDoThingsWithCredentials()
    {
        $user = $this->getUser();

        // 增加一个或者两个证书
        $user->addCredential('foo');
        $user->addCredentials('foo', 'bar');
```

```

// 检查用户是否有某个证书
echo $user->hasCredential('foo');           =>    true

// 检查用户是否拥有这些证书中的一个
echo $user->hasCredential(array('foo', 'bar'));           =>    true

// 检查用户是否同时拥有两个证书
echo $user->hasCredential(array('foo', 'bar'), true); =>    true

// 删除一个证书
$user->removeCredential('foo');
echo $user->hasCredential('foo');           =>    false

// 删除所有的证书(在登出是特别有用)
$user->clearCredentials();
echo $user->hasCredential('bar');           =>    false
}
}

```

如果一个用户有'foo'的证书,这个用户可以访问在 security.yml 里设有该证书的动作。认证也可以在模板里用来显示被授权的内容。请参看例 6-27。

例 6-27 - 在模板里处理用户的证书

```

<ul>
  <li><?php echo link_to('section1', 'content/section1') ?></li>
  <li><?php echo link_to('section2', 'content/section2') ?></li>
  <?php if ($sf_user->hasCredential('section3')): ?>
  <li><?php echo link_to('section3', 'content/section3') ?></li>
  <?php endif; ?>
</ul>

```

至于认证状态,证书通常在用户登录时授予用户。这就是为什么 sfUser 对象常常扩展登录和注销的方法的原因,这样就可以把用户的认证状态放在一个中心位置。

TIP symfony 的 plugin 里, sfGuardPlugin 扩展了会话类使登录和注销更容易。详情请参考第 17 章。

复合证书

你可以利用 YAML 的语法(在 security.yml 文件里)和 AND 类型或 OR 类型关联,去认证有组合证书的用户。有效地利用组合证书,你可以建立一个复杂的工作流

程和用户权限管理系统。例如，一个内容管理系统（CMS）的后台管理系统只允许有 admin 特权的用户使用，编辑文章需要有 editor 特权，发布需要有 publisher 特权等等。请参看例 6-28。

例 6-28 - 认证组合语法

```
editArticle:
    credentials: [ admin, editor ]           # admin AND editor

publishArticle:
    credentials: [ admin, publisher ]       # admin AND publisher

userManagement:
    credentials: [[ admin, superuser ]]     # admin OR superuser
```

每次添加一层方括号，逻辑将转变到另一方（AND 和 OR）。你可以建立非常复杂的证书组合，比如：

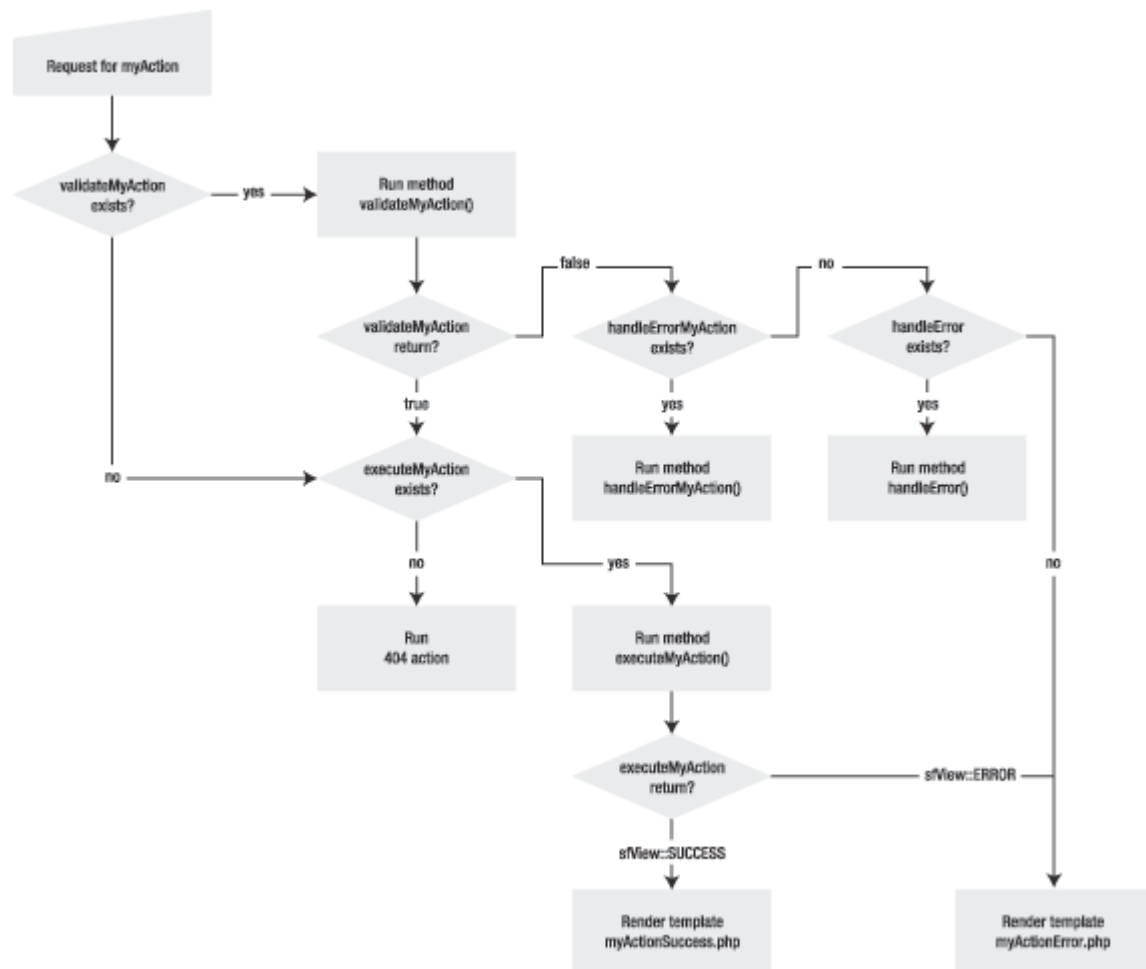
```
credentials: [[root, [supplier, [owner, quasiowner]], accounts]]
               # root OR (supplier AND (owner OR quasiowner)) OR accounts
```

检验和处理错误的方法

检验动作的输入（大部份是请求参数）是一件经常重复而乏味的工作。symfony 用动作类里的方法提供一个内置的请求检验系统。

先举一个例子。当动作 myAction 接收到一个用户的请求时，symfony 首先要寻找 validateMyAction() 方法。如果找到了，symfony 就会执行它。检验方法的返回值决定了下一个被执行的方法：如果返回值是 true，那么 executeMyAction() 将被执行；否则，handleErrorMyAction() 将被执行。如果是第二种情况，而且 handleErrorMyAction() 不存在，symfony 将自动寻找常规的 handleError() 方法。如果这个方法也不存在的话，它就返回 sfView::ERROR 并显示 myActionError.php 模板。图 6-2 描述了这个过程。

图 6-2 - 检验过程



检验的关键是掌握动作方法命名的约定：

- `validateActionName` 是检验方法，它返回 `true` 或 `false`。当动作 `ActionName` 接收到一个请求时，它会先被执行。如果这个检验方法不存在，动作将被直接执行。
- 如果上面的检验方法返回 `false`，`handleErrorActionName` 方法将被执行。如果这个验检方法不存在，`symfony` 将显示错误信息。
- `executeActionName` 是动作方法。任何动作都必须有这个方法。

例 6-29 是一个动作类和它的检验方法。不论检验的结果是 `true` 或 `false`，`myActionSuccess.php` 模板（和不同的参数）将被执行。

例 6-29 - 检验方法的示例

```

class mymoduleActions extends sfActions
{
    public function validateMyAction()
    {
        return ($this->getRequestParameter('id') > 0);
    }
}
  
```

```

public function handleErrorMyAction()
{
    $this->message = "Invalid parameters";

    return sfView::SUCCESS;
}

public function executeMyAction()
{
    $this->message = "The parameters are correct";
}
}

```

你可以在 `validate()` 方法里填入相关的程序。但返回值必须是 `true` 或 `false`。作为 `sfActions` 类里的方法，它同样可以使用 `sfRequest` 和 `sfUser` 对象，这对输入的参数和上下文的检验非常有帮助。

你可以利用这个结构实施表单检验（检验用户在表单里输入的数据之后，再处理这些数据）。当然，对于这些经常重复的工作，`symfony` 提供了一些自动工具。我们将在第 10 章讲解。

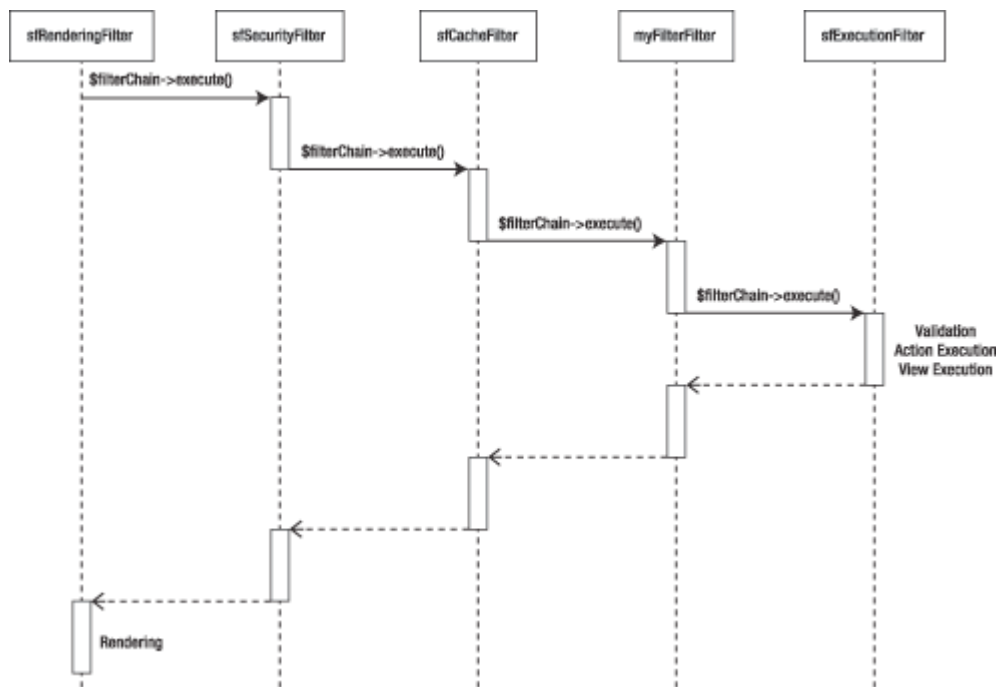
过滤器

安全检验过程可以被理解为：在执行动作前所有的请求都要通过一个过滤器。根据过滤器的测试情况，请求的结果将有所变动--比如执行其它的动作（执行默认的安全检验，而不是被请求的动作）。`symfony` 把这个思想扩展到过滤器的类里。在执行动作或在执行答复之前，你可以指定先被执行的过滤器。过滤器可以被视为程序包，就象 `preExecute()` 和 `postExecute()`，但级别较高（针对整个应用程序，而不是一个模块）。

过滤器链

`symfony` 把请求处理过程看作是一条过滤器链。当架构收到一个请求时，第一个过滤器（通常是 `sfRenderingFilter`）将被执行。之后，每一个在链上的过滤器将依次被执行。当最后一个过滤器（通常是 `sfExecutionFilter`）被执行时，上一个过滤器就结束了，然后返回到第一个过滤器上。图 6-3 是执行一个假设的过滤器链的示意图（真实的链有更多的过滤器）。

图 6-3 - 过滤器链示例



这个过程确定了过滤器类的结构。每个过滤器都是 `sfFilter` 的子类，都有一个 `execute()` 方法，并接收一个 `$filterChain` 对象作为参数。在这些方法里，每个过滤器都用 `$filterChain->execute()` 传递到下一个过滤器上。请参看例 6-30。所以，过滤器可以分成两部分：

- 在 `$filterChain->execute()` 之前的代码在动作之前执行。
- 在 `$filterChain->execute()` 之后的代码在动作之后，并在视图前执行。

例 6-30 - 过滤器类的结构

```

class myFilter extends sfFilter
{
    public function execute ($filterChain)
    {
        // 动作执行前的代码
        ...

        // 执行过滤器链中的下一个过滤器
        $filterChain->execute();

        // 动作执行完以后，显示之前需要执行的代码
        ...
    }
}
  
```

默认的过滤器链是在配置文件 `filters.yml` 中定义的，请参看例 6-31。所有的请求都必须通过这个文件里的过滤器。

例 6-31 - 默认的过滤器链，在 `myapp/config/filters.yml` 里

```
rendering: ~
web_debug: ~
security: ~
```

一般来说，你可以在这里加入你的过滤器

```
cache: ~
common: ~
flash: ~
execution: ~
```

上面的配置文件没有任何参数（~，在 YAML 里的意思是"null"），因为这些参数是从 `symfony` 核心继承下来的。在 `symfony` 核心里，`symfony` 设置每一个过滤器的 `class` 和 `param`。例如，例 6-32 是默认的 `rendering` 过滤器。

例 6-32 - `rendering` 过滤器里默认的参数，在 `$sf_symfony_data_dir/config/filters.yml` 里

```
rendering:
  class: sfRenderingFilter # Filter class
  param:                  # Filter parameters
    type: rendering
```

在 `filters.yml` 保留空值（~）的意思是：过滤器将用 `symfony` 核心里的配置文件。

你可以定制不同的过滤器链：

- 如果要关闭一些过滤器，添加 `enabled: off` 参数。例如，要关闭网页调试过滤器：

```
web_debug:
  enabled: off
```

- 关闭过滤器时，禁止在 `filters.yml` 删除任何条目；否则 `symfony` 将显示错误信息。
- 你可以添加你自己的过滤器（通常在 `security` 过滤器之后，将在下一节讲解）。注意 `rendering` 必须是第一个过滤器，而 `execution` 必须是最后一个过滤器。
- 覆盖超类和默认的过滤器设置（特别是改变安全系统，使用你自己的安全过滤器）。

TIP enabled: off 参数可以用来禁用你自己的过滤器，也可以通过修改 settings.yml 文件的 web_debug、use_security、cache 和 use_flash 参数来禁用默认的过滤器。这是因为每个默认过滤器都有一个检测这些值的`条件`参数。

建立自己的过滤器

建立一个过滤器很简单。首先像例 6-30 一样创建一个类，然后把它放在项目的 lib/里，还可以利用自动加载的功能。

动作可以转发或跳转到另一个动作上，这样就会重新执行过滤器链。你或许只想让第一个动作通过你的过滤器。 sfFilter 类里的 isFirstCall() 方法（返回布尔值）就是为此设计的。当然这个方法必须在动作前执行。

这些概念将在例子里加以说明。例 6-33 是一个用户自动登录的过滤器。过滤器里的 MyWebSite cookie 应该由登录动作设立。这是做登录表单里中"记住我"的基础。

例 6-33 - 过滤器类文件，在 apps/myapp/lib/rememberFilter.class.php 里

```
class rememberFilter extends sfFilter
{
    public function execute($filterChain)
    {
        // Execute this filter only once
        if ($this->isFirstCall())
        {
            // Filters don't have direct access to the request and user objects.
            // You will need to use the context object to get them
            $request = $this->getContext()->getRequest();
            $user     = $this->getContext()->getUser();

            if ($request->getCookie('MyWebSite'))
            {
                // sign in
                $user->setAuthenticated(true);
            }
        }

        // Execute next filter
        $filterChain->execute();
    }
}
```

有时候，在执行一个过滤器后，你需要直接转发给一个动作，而不是继续执行过滤器链。sfFilter 没有 forward()（转发）方法，但 sfController 有。所以你可以像下面这样转发给动作：

```
return $this->getController()->forward('mymodule', 'myAction');
```

NOTE sfFilter 类有一个 initialize() 方法，它会在过滤器对象建立的时候执行。如果你需要用自己的方式处理过滤器参数（在 filters.yml 里定义，稍后将介绍），你可以重写这个方法。

过滤器激活和参数

建立一个过滤器后还需要激活它。你需要把你的过滤器加在过滤器链上，也就是说，你必须在 filters.yml 声明你的过滤器类。filters.yml 在应用程序或模块的 config/ 里。请参看例 6-34。

例 6-34 - 过滤器激活文件样本，在 apps/myapp/config/filters.yml 里

```
rendering: ~
web_debug: ~
security: ~

remember:          # 过滤器需要一个唯一的名字
  class: rememberFilter
  param:
    cookie_name: MyWebSite
    condition:   %APP_ENABLE_REMEMBER_ME%

cache:             ~
common:            ~
flash:             ~
execution:         ~
```

过滤器被激活后，所有的请求都通过这个过滤器。过滤器的配置文件可以在 param 下面定义一个或多个参数。过滤器类可以通过 getParameter() 方法获取这些参数。例 6-35 展示如何获取一个参数值。

例 6-35 - 获取一个参数值，在 apps/myapp/lib/rememberFilter.class.php 里

```
class rememberFilter extends sfFilter
{
  public function execute ($filterChain)
  {
```

```

    ...
    if ($request->getCookie($this->getParameter('cookie_name'))))
    ...
}
}

```

过滤器链首先测试条件参数，并决定是否必须执行该过滤器。所以过滤器的声明可以依赖应用程序的配置，就像例 6-34 一样。如果你的应用程序 `app.yml` 和下面设置相同，“记住我”过滤器将被执行。

```

all:
  enable_remember_me: on

```

过滤器实例

每一个动作都需要通过过滤器，这个特性还有其它的用途。例如，如果你使用一个远程的分析报告系统，你需要在每页加入一块远程跟踪代码。你可以把这块代码放在共用的版面，但这样一来，整个应用程序的活动都会被分析报告系统记录下来。一个更好的方法，就是你可以把它放在过滤器里，就像例 6-36 一样。这样可以跟踪记录每一个模块的活动。

例 6-36 - Google 分析报告过滤器

```

class sfGoogleAnalyticsFilter extends sfFilter
{
    public function execute($filterChain)
    {
        // 执行动作前什么也不必作
        $filterChain->execute();

        // 在回应中加入跟踪代码
        $googleCode = '
<script src="http://www.google-analytics.com/urchin.js"
type="text/javascript">
</script>
<script type="text/javascript">
    _uacct="UA-' . $this->getParameter('google_id') . '";urchinTracker();
</script>';
        $response = $this->getContext()->getResponse();
        $response->setContent(str_replace('</body>',
$googleCode.' </body>', $response->getContent()));
    }
}

```

请注意这个过滤器是不完美的，它不应该把跟踪系统放在不是 HTML 的答复里。

另一个例子是，过滤器可以把一般的请求转换到 SSL 上，以确保安全交流。请参看例 6-37。

例 6-37 - 安全交流过滤器

```
class sfSecureFilter extends sfFilter
{
    public function execute($filterChain)
    {
        $context = $this->getContext();
        $request = $context->getRequest();
        if (!$request->isSecure())
        {
            $secure_url = str_replace('http', 'https', $request->getUri());
            return $context->getController()->redirect($secure_url);
            // 不继续过滤器链
        }
        else
        {
            // 请求是安全的，所以继续
            $filterChain->execute();
        }
    }
}
```

过滤器广泛地在插件里使用，因为这样可以使这些功能在整个应用程序里共用。第 17 章讲解插件。另外，请参考网上 wiki (<http://www.symfony-project.com/trac/wiki>)，那里有更多有关过滤器的例子。

模块配置

模块特性依赖于配置文件。如果你想修改这些特性，你必须在模块的 config/ 建立一个 module.yml 文件，并为每个环境建立一个设（或在 all: 下增加所有环境共用的设置）。例 6-38 显示一个模块 mymodule 的配置文件 module.yml。

例 6-38 - 模块配置，在 apps/myapp/modules/mymodule/config/module.yml 里

```
all:                                # 所有的环境
    enabled:      true
    is_internal:  false
```

`view_name: sfPhpView`

"enabled"参数可以用来关闭模块里所有的动作。这些动作被跳转到 `module_disabled_module/module_disabled_action` 动作上（在 `settings.yml` 设置）。

"is_internal"参数可以限制所有的动作内部访问。例如，邮件动作可以由另一个内部的动作访问，但不可以从外部访问。

"view_name"参数限定视图类。它必须继承 `sfView`。覆盖这个参数你就可以使用其它的视图系统和模板引擎，比如 `smarty`。

总结

在 `symfony` 里，控制器层由两部分组成：前端控制器是在一个环境下整个应用程序的唯一入口；动作里包含应用逻辑。动作返回一个 `sfView` 常数，从而决定如何展示视图。在动作里，你可以操纵不同的组成元素，包括请求对象(`sfRequest`)和用户会话对象(`sfUser`)。

结合会话对象，动作对象，安全配置，你可以建立一个有访问限制及认证的安全系统。在动作里专有的 `validate()`和 `handleError()`方法用于检验请求。如果说 `preExecute()`和 `postExecute()`方法是为了代码重用而设计的（在一个模块里），那么过滤器对整个应用程序有相同的代码重用功能，并检验每一个请求。

第 7 章 深入视图层

视图(view)的作用是显示特定动作(action)的输出。在 symfony 里,视图由几部分组成,这些部分都很容易修改。

- Web 设计师通常会与模板(当前动作的数据的表现形式)和布局(包含所有页面都会用到的代码)打交道。这些模板由 HTML 加上 PHP 代码片段(主要是辅助函数调用)组成。
- 为了重用,开发者往往会把模板代码的片段放在局部模板(Partials)或者组件(Components)里。开发者使用槽(Slots)与组件(Components)来影响布局的多个区域。web 设计师也可以修改这些模板片段。
- 开发者专注于 YAML 视图配置文件(用来设置回应与其他界面元素的属性)还有回应对象(response object)。处理模板里的变量的时候,跨站脚本(cross-site scripting)的风险不可忽略,这就需要在记录用户数据的时候很好的理解输出转义(output escaping)技术。

不论你是哪一个角色,你都可以发现能加快输出动作结果这件乏味的工作的工具。这一章将会介绍这些工具。

模板

例 7-1 是一个典型的 symfony 模板。它包含一些 HTML 代码和一些基本的 PHP 代码,通常是显示动作(action)里定义的(通过\$this->name = 'foo';)变量还有辅助函数。

例 7-1 - indexSuccess.php 模板样本

```
<h1>欢迎</h1>
<p>欢迎回来 <?php echo $name ?>!</p>
<ul>您要做什么?
  <li><?php echo link_to(' 阅读最新的文章', 'article/read') ?></li>
  <li><?php echo link_to(' 写一篇新文章', 'article/write') ?></li>
</ul>
```

在第 4 章里介绍过,这种另类的 PHP 语法对非 PHP 开发者来说也很容易理解因此很适合于用在模板里。请注意在模板里面尽量减少 PHP 代码量,由于这些文件用来设计程序的界面,这些模板有些时候是由其他的团队维护的,例如表现团队而不是应用程序逻辑团队。把逻辑放在动作(action)里还可以使一个动作对应多个模板更容易,减少代码重复。

辅助函数 (Helpers)

辅助函数是返回模板里使用的 HTML 代码的 PHP 函数。在 例 7-1 里, `link_to()` 函数就是一个辅助函数。有时, 辅助函数只是用来节约时间, 把模板里常用的代码封装起来。例如, 你很容易想得到下面这个辅助函数的定义:

```
<?php echo input_tag('nickname') ?>
=> <input type="text" name="nickname" id="nickname" value="" />
```

它应该与 例 7-2 中的差不多。

例 7-2 - 辅助函数定义的例子

```
function input_tag($name, $value = null)
{
    return '<input type="text" name="'. $name. '"
id="'. $name. '" value="'. $value. '" />';
}
```

事实上, `symfony` 内建的 `input_tag()` 函数比这个要复杂一点, 它有第三个参数, 这个参数用来指定 `<input>` 标签的属性。你可以去在线 API 文档查看这个函数详细的语法与参数。(<http://www.symfony-project.com/api/symfony.html>)。

大多数时候, 辅助函数更聪明并且节省大量写代码的时间:

```
<?php echo auto_link_text('请访问我们的网站 www.example.com') ?>
=> 请访问我们的网站 <a
href="http://www.example.com">www.example.com</a>
```

辅助函数能加快写模板的速度, 同时辅助函数生成的 HTML 兼具性能与可访问性。当然, 你还是可以写普通 HTML 代码, 不过辅助函数写起来总是要快一些。

TIP 你可能会问为什么辅助函数的命名用下划线而不是 `symfony` 里随处可见的大小写字母规则。这是因为辅助函数是函数, 所有的 PHP 核心函数都用下划线命名规则。

声明辅助函数

包含辅助函数定义的 `symfony` 文件不能被自动载入(因为它们是函数而不是类)。辅助函数按照目的分组。例如, 所有处理文字的辅助函数都在一个名叫 `TextHelper.php` 的文件里定义, 称作 Text 辅助函数组。所以如果你要在模板里使用一个辅助函数, 你必须在使用之前通过 `user_helper()` 函数声明载入这个辅助函数相关的辅助函数组。例 7-3 里的这个模板使用了 `auto_link_text()` 辅助函数, 它属于 Text 辅助函数组。

例 7-3 - 声名使用一个辅助函数

```
// 在这个模板里使用一个特定的辅助函数
```

```
<?php echo use_helper('Text') ?>
...
<h1>描述</h1>
<p><?php echo auto_link_text($description) ?></p>
```

TIP 如果你要声明多个辅助函数组，给 `use_helper()` 函数传多个参数就可以了。例如，要在一个模板里载入 `Text` 和 `Javascript` 辅助函数组，可以使用 `<?php echo use_helper('Text', 'Javascript') ?>` 来声明。

有一些辅助函数在所有的模板里都可以使用，不需要事先声明。它们是以下的辅助函数组：

- `Helper`：用来载入辅助函数(`use_helper()` 函数本身就是一个辅助函数)
- `Tag`：基本的标签辅助函数，几乎所有的辅助函数都用到它
- `Url`：链接与 URL 管理辅助函数
- `Asset`：用来生成 HTML<head>部分的内容，还包括简化使用外部资源(图片，Javascript，样式表)的函数
- `Partial`：用来调用局部模板的辅助函数
- `Cache`：管理代码片段的缓存
- `Form`：表单辅助函数

这里列出的标准辅助函数，在每个模板中都会被自动载入，可以在 `settings.yml` 文件里面设置。所以如果你确定你不会用到 `Cache` 辅助函数组的辅助函数，或者你每次都需要用到 `Text` 组，你可以修改 `standard_helper` 这个设置。这会稍稍加快你的程序。但是你不能删除这个列表里的前四个辅助函数组 (`Helper`、`Tag`、`Url` 和 `Asset`)，因为模板引擎需要它们才能正常工作。所以在标准辅助函数设置(`standard_helper`)里找不到这四个辅助函数组。

TIP 如果你需要在模板之外使用辅助函数，你也可以通过 `sfLoader::loadHelper($helpers)` 来载入一个辅助函数组，`$helpers` 可以是辅助函数组的名字或几个辅助函数组名字组成的数组。例如，如果你想在一个动作(`action`)里使用 `auto_link_text()`，你需要首先执行 `sfLoader::loadHelper('Text')`。

常用辅助函数

在本节里你会了解一些后面要用到的辅助函数的详情。例 7-4 给出了一个常用辅助函数列表，还有它们输出的 HTML 代码。

例 7-4 - 常用的默认辅助函数

```
// Helper 组
<?php echo use_helper('HelperName') ?>
<?php echo use_helper('HelperName1', 'HelperName2', 'HelperName3') ?>
```



```
// Tag 组
<?php echo tag('input', array('name' => 'foo', 'type' => 'text')) ?>
<?php echo tag('input', 'name=foo type=text') ?> // 另一种选项格式
=> <input name="foo" type="text" />
<?php echo content_tag('textarea', 'dummy content', 'name=foo') ?>
=> <textarea name="foo">dummy content</textarea>

// Url 组
<?php echo link_to('点我', 'mymodule/myaction') ?>
=> <a href="/route/to/myaction">点我</a> // 取决于路由(routing)设置

// Asset 组
<?php echo image_tag('myimage', 'alt=foo size=200x100') ?>
=> 
<?php echo javascript_include_tag('myscript') ?>
=> <script language="JavaScript" type="text/javascript"
src="/js/myscript.js"></script>
<?php echo stylesheet_tag('style') ?>
=> <link href="/stylesheets/style.css" media="screen"
rel="stylesheet" type="text/css" />
```

symfony 里还有很多其他的辅助函数，如果要讲完它们需要一整本书。辅助函数的最佳参考是在线 API 文档(<http://www.symfony-project.com/api/symfony.html>)，所有的辅助函数都有详细的介绍，包括语法，参数，还有例子。

自己写辅助函数

symfony 本身包含了大量的各类辅助函数，不过如果你在 API 文档里找不到你需要的辅助函数，你可能会想自己写新的辅助函数。这很简单。

一组辅助函数(返回 HTML 代码的标准 PHP 函数)被存放在名叫 FooBarHelper.php 的文件里，FooBar 是这个辅助函数组的名字。这个文件在 app/myapp/lib/help/目录下(或者任意一个 lib/目录下的 help/目录)，这样做的目的是为了让 use_helper('FooBar')辅助函数能自动载入这组辅助函数。

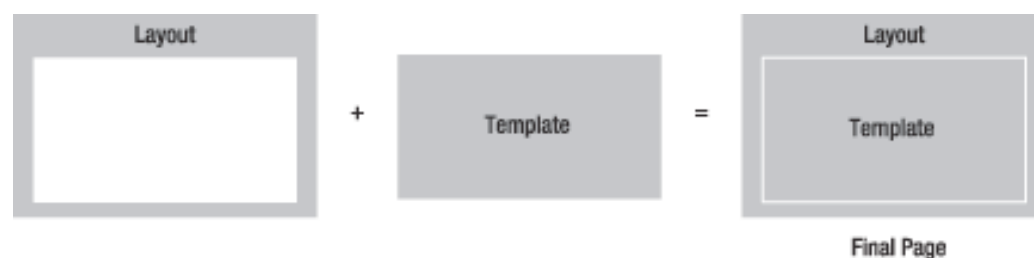
TIP 系统允许你覆盖 symfony 自己的辅助函数。例如，如果你想重新定义 Text 辅助函数的所有内容，只要在 apps/myapp/lib/helper 目录下建立 TextHelper.php 文件。当使用 use_helper('Text')的时候 symfony 会使用你定义的辅助函数而不是系统默认的。但是请注意：由于原始文件没有载入，你需要重新定义这组辅助函数里所有的函数，否则某些系统的辅助函数就用不了了。

页面布局

例 7-1 里的模板不是一个有效的 XHTML 文档。它缺少 DOCTYPE 定义还有<html>与<body>标签。这是因为它们存放在程序的其他地方，即 layout.php 这个文件里，它包含页面布局。这个文件也被称为全局模板，存放所有页面都会使用的 HTML 代码，这样避免在每个页面里面重复。模板的内容被包括在布局里，或者说，布局“装饰”模板。图 7-1 是这种装饰模式的程序。

TIP 想要详细了解装饰模式与其他设计模式，请参考《*Patterns of Enterprise Application Architecture*(企业应用架构模式)》这本书，作者是 Martin Fowler (Addison-Wesley, ISBN: 0-32112-742-0，中文版由机械工业出版社 书号 7-111-14305-1)。

图 7-1 - 用布局装饰模板



例 7-5 是一个默认的页面布局，在应用程序的 templates/目录下。

例 7-5 - 默认布局，myproject/apps/myapp/templates/layout.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-transitional
1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <?php echo include_http_metas() ?>
  <?php echo include_metas() ?>
  <?php echo include_title() ?>
  <link rel="shortcut icon" href="/favicon.ico" />
</head>
<body>

<?php echo $sf_data->getRaw('sf_content') ?>

</body>
</html>
```

<head>部分的辅助函数用来取得视图配置文件里面的信息。<body>标签里的内容输出模板的执行结果。这个布局加上例 7-1 里的模板还有默认的视图配置文件，会得到例 7-6 的输出。

例 7-6 - 布局，视图配置，还有模板加起来的结果

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/2000/REC-xhtml1-20000126/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <meta name="title" content="symfony project" />
  <meta name="robots" content="index, follow" />
  <meta name="description" content="symfony project" />
  <meta name="keywords" content="symfony, project" />
  <title>symfony project</title>
  <link rel="stylesheet" type="text/css" href="/css/main.css" />
  <link rel="shortcut icon" href="/favicon.ico">
</head>
<body>

<h1>欢迎</h1>
<p>欢迎回来, <?php echo $name ?>!</p>
<ul>你要做什么?
  <li><?php echo link_to(' 阅读最新的文章', 'article/read') ?></li>
  <li><?php echo link_to(' 写一篇新文章', 'article/write') ?></li>
</ul>

</body>
</html>
```

每个应用程序的全局模板都可以彻底的修改，添加必要的 HTML 代码。布局常常用来放置网站导航，标志等。你甚至可以有多个布局，不同的动作(action)可以用不同的布局。不需要担心 JavaScript 还有样式表的包含问题，在本章的“视图配置”这一节里会介绍如何处理这个问题。

模板快捷变量

在模板里，有一些 symfony 变量可以直接使用。通过这些快捷变量可以从 symfony 的对象里取得一些最常用的模板信息：

- \$sf_context：完整的环境对象(context object) (sfContext 类的实例)
- \$sf_request：请求对象(sfRequest 类的实例)
- \$sf_params：请求的参数
- \$sf_user：当前的用户 session 对象 (sfUser 类的实例)

在上一章里介绍了 `$sfRequest` 还有 `$sfUser` 对象的常用方法，这些方法可以在模板里通过 `$sf_request` 和 `$sf_user` 变量调用。例如，如果请求里包含 `total` 参数，它的值可以在模板里通过下面的方法取得：

```
// 长一点的版本
<?php echo $sf_request->getParameter('total'); ?>

// 短版本
<?php echo $sf_params->get('total'); ?>

// 相当于在动作(action)里面执行下面的代码
echo $this->getRequestParameter('total');
```

代码片段 (Code Fragments)

你可能常常会在好几个页面包含一些 HTML 或者 PHP 代码。为了避免重复，PHP 的 `include()` 语句大多数时候就足够了。

例如，如果你的程序的很多模板都需要同一段代码，把这断代码存在全局模板目录里 (`myproject/apps/myapp/templates/`) 命名为 `myFragment.php`，然后在模板里这样去包含它：

```
<?php
include(sfConfig::get('sf_app_template_dir').'/myFragment.php') ?>
```

但是这样封装一段代码并不是一个很好的做法，因为你需要很多变量名来在这段代码与不同的模板之间传递信息。另外，`symfony` 的缓存系统(将在第 12 章介绍)不能够检测到这种包含，所以这段代码没法被单独缓存起来。`symfony` 提供了三种不同的聪明的代码片段来取代 `include`：

- 如果逻辑部分代码量很小，只需要包含一个能访问一些你传递的数据的模板。这样，你需要用局部模板(`partial`)。
- 如果逻辑的代码量比较大（例如，你需要访问数据模型，并且根据 `session` 修改数据），你可能回想把逻辑与表现分开。这种情况，你需要用组件(`component`)。
- 如果这个片段用来替换布局里的特定部分，这个部分有一个默认的内容。你需要用槽(`slot`)。

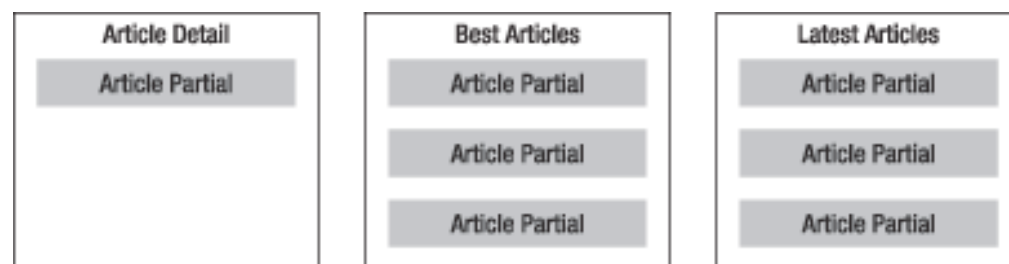
Note 还有一种代码片段，组件槽(`component slot`)，它用于代码片段与环境有关的情况(例如，对一个模块内的不同动作，这段代码需要有所不同)。组件槽(`component slot`)将在稍后介绍。

这些片段由局部模板(`Partial`)辅助函数组完成。这些辅助函数不要用声明就可以在 `symfony` 模板中使用。

局部模板（Partials）

局部模板是可重用的一段模板。例如，在一个发布程序里，文章详情页面用来显示文章的代码，也可以用在最佳文章列表和最新文章页面。这代码就很适合作为局部模板，如图 7-2 所示。

图 7-2 - 重用局部模板



与模板类似，局部模板也位于 `templates/` 目录，也是由 HTML 代码与嵌入式 PHP 代码组成。局部模板文件名以下划线(`_`)打头，这样可以把它们与同在 `templates/` 目录的模板区分开来。

模板中可以包含同一个模块内或者其他模块的局部模板，也可以是在全局的 `templates/` 目录中的局部模板。使用 `include_partial()` 辅助函数包含局部模板，参数是模块与局部模板的名字（但是请省略开头的下划线与结尾的 `.php`），如例 7-7。

例 7-7 - 在 `mymodule` 模块的模板中包含一个局部模板

```
// 包含 myapp/modules/mymodule/templates/_mypartial1.php 局部模板
// 由于模板与这个局部模板在同一个模块里
// 可以省略模块名
<?php include_partial('mypartial1') ?>

// 包含 myapp/modules/foobar/templates/_mypartial2.php 局部模板
// 必须些模块名
<?php include_partial('foobar/mypartial2') ?>

// 包含 myapp/templates/_mypartial3.php 局部模板
// 这是 'global' 模块的局部模板
<?php include_partial('global/mypartial3') ?>
```

局部模板中可以使用标准 `symfony` 辅助函数和模板快捷变量。但是由于局部模板可以在任何地方使用，它们不能直接访问使用它们的模板对应的动作定义的变量，除非作为参数传递给它们。例如，如果你希望局部模板能够访问 `$total` 变

量，必须由动作(action)先传递给模板，然后模板通过 `include_partial()` 辅助函数的第二个参数传递给局部模板，如例 7-8，7-9，7-10 所示。

例 7-8 - 在动作里定义一个变量 `mymodule/actions/actions.class.php`

```
class mymoduleActions extends sfActions
{
    public function executeIndex()
    {
        $this->total = 100;
    }
}
```

例 7-9 - 模板把变量传递给局部模板 `mymodule/templates/indexSuccess.php`

```
<p>Hello, world!</p>
<?php include_partial('mypartial',
array('mytotal' => $total)
) ?>
```

例 7-10 - 局部模板现在可以使用这个变量了
`mymodule/templates/_mypartial.php`

```
<p>Total: <?php echo $mytotal ?></p>
```

TIP 到目前为止所有的辅助函数都是通过 `<?php echo functionName() ?>` 这样来调用的。局部模板辅助函数，直接通过调用就可以了，不需要 `echo`，这有点类似 PHP 的 `include()` 语句。如果你需要一个能返回局部模板内容而不显示的函数，你可以用 `get_partial()`。所有本章介绍的 `include_` 辅助函数都有一个对应的 `get_` 辅助函数，这个 `get_` 辅助函数与 `echo` 语句配合使用的功能与 `include_` 函数相同。

组件 (Components)

第 2 章的第 1 个例子按照逻辑于表现分成了两部分。与 MVC 模式的动作(action)与模板类似，你可能会需要把局部模板分成逻辑部分于表现部分。遇到这种情况，你需要使用组件。

组件类似于动作(action)，不过他要快很多。组件的逻辑存放在 `sfComponents` 类的子类里，位于 `action/components.class.php` 里。它的表现部分存放在局部模板里。`sfComponents` 类的方法由 `execute`(执行)这个词开头，类似于动作(action)，它们传递变量给表现层的方式也与动作(action)一样。组件的局部模板根据组件的方法命名（去掉 `execute`，前面加下划线）。表 7-1 比较了动作与组件的命名方式。

Table 7-1 - 动作与组件命名方式比较

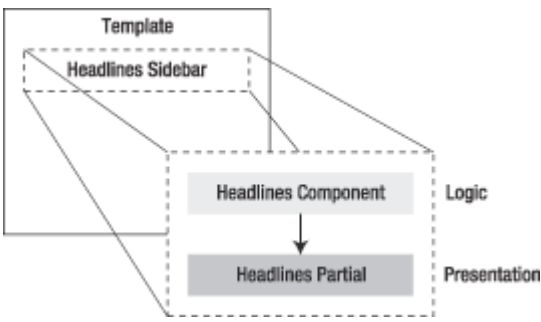
命名方式	动作	shitu lingjian
逻辑文件	actions.class.php	components.class.php
继承的类	sfActions	sfComponents
方法命名	executeMyAction()	executeMyComponent()
表现文件命名	myActionSuccess.php	_myComponent.php

TIP 与动作类似，可以把组件文件分成几个文件，sfComponents 类也有一个对应的 sfComponent，单独的组件文件可以使用同样的语法。

例如，假设你有一个根据用户信息显示特定主题的最新新闻的侧边栏，好几个页面都需要用到。取得新闻的查询比较复杂，把它们放在局部模板里面有点困难，所以你需要把它们移动到一个与单独的动作文件类似的组件文件里，如图 7-3 所示。

这个例子里，如例 7-11 与 7-12 所示，组件放在自己的模块里（news），不过如果从功能上来说更合理，你也可以把组件与动作放在一个模块里面。

图 7-3 - 在模板里使用组件



例 7-11 - 组件类， modules/news/actions/components.class.php

```
<?php

class newsComponents extends sfComponents
{
    public function executeHeadlines()
    {
        $c = new Criteria();
        $c->addDescendingOrderByColumn(NewsPeer::PUBLISHED_AT);
        $c->setLimit(5);
        $this->news = NewsPeer::doSelect($c);
    }
}
```

例 7-12 - 局部模板, modules/news/templates/_headlines.php

```
<div>
  <h1>最新消息</h1>
  <ul>
    <?php foreach($news as $headline): ?>
      <li>
        <?php echo $headline->getPublishedAt() ?>
        <?php echo
Link_to($headline->getTitle(), 'news/show?id=' . $headline->getId()) ?>
      </li>
    <?php endforeach ?>
  </ul>
</div>
```

现在, 要在模板里使用组件的时候, 只要执行下面的代码:

```
<?php include_component('news', 'headlines') ?>
```

与局部模板类似, 组件也接受数组形式的参数。这些参数在局部模板里可以通过名字访问, 在组件里通过\$this 对象访问。如例 7-13。

例 7-13 - 传递参数给组件和组件的模板

```
// 载入组件
<?php include_component('news', 'headlines', array('foo' => 'bar')) ?>

// 在组件里
echo $this->foo;
=> 'bar'

// 在_headlines.php 局部模板里
echo $foo;
=> 'bar'
```

除了在一般模板里, 也可以在组件里或者全局模板里包含组件。与动作类似, 组件的 execute 方法可以传递变量给对应的局部模板, 组件的局部模板里也可以访问模板快捷变量。不过相似性仅限与此。组件不能处理安全性和验证, 不能从网络直接调用 (只能从程序内部), 不能有多种返回形式。所以组件要比动作快。

槽 (Slots)

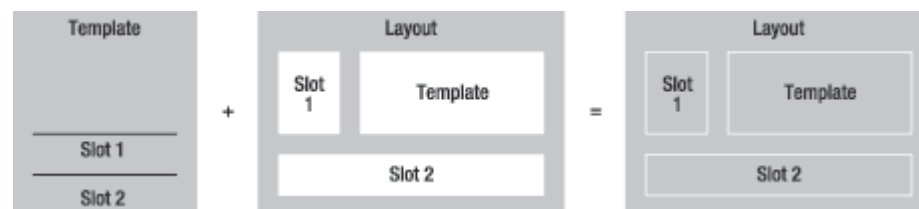
局部模板与组件很容易重用。但是很多时候, 布局里有多个需要用代码片段去填充的动态区域。例如, 假设你想在<head>里增加一些与动作(action)相关的标签,

或者，布局包含一个主要动态区域，这个区域的内容由动作(action)产生，另外还有很多小的区域，这些区域有默认的内容同时它们能够在模板里面改写。

这些情况下，需要用到槽(slot)。简单来讲，槽是可以放在任意视图元素（布局 layout, 模板或局部模板）的占位符。填充这个占位符类似于给变量赋值。填充的代码在回应(response)的全局空间里，所以可以在任何地方定义它（布局，模板或者局部模板）。只要注意在使用槽之前定义它，另外请记住布局是在模板之后执行的（这是装饰过程），局部模板是它们在模板里被调用的时候执行的。这些听起来很抽象吗？来看例子。

假设这个布局有一个模板区域和两个槽：一个是侧边栏，另一个是页尾。模板里定义了两个槽的值。在装饰过程中，布局代码把模板代码包含进来，然后用之前定义的值填充槽，如图 7-4 所示。侧边栏与页尾可以跟主动作(action)相关联。这就好像一个有多个“洞”的布局。

图 7-4 - 模板里定义的布局槽



为了更进一步理解，我们来看点代码。include_slot()辅助函数用来调用槽。对于定义过的槽 has_slot()辅助函数会返回真，这样的代码是程序不容易出错。例如，在布局里定义一个名叫 sidebar 的槽以及它的默认值，如例 7-14 所示。

例 7-14 - 在布局里定义 sidebar 槽

```
<div id="sidebar">
<?php if (has_slot('sidebar')): ?>
    <?php include_slot('sidebar') ?>
<?php else: ?>
    <!-- 默认的 sidebar 代码 -->
    <h1>与环境有关的区域</h1>
    <p>这个区域的内容与主区域的内容有关。</p>
<?php endif; ?>
</div>
```

槽可以在任何一个模板里定义（事实上，局部模板里也可以）。由于槽用来包含 HTML 代码，symfony 提供了一种方便的定义方式：直接在 slot()与 end_slot()辅助函数之间书写槽代码就可以了，如例 7-15。

例 7-15 - 在模板里重新定义 sidebar 槽的内容

```

...
<?php slot('sidebar') ?>
    <!-- 当前模板的侧边栏代码-->
    <h1>用户详情</h1>
    <p>姓名: <?php echo $user->getName() ?></p>
    <p>电子邮件: <?php echo $user->getEmail() ?></p>
<?php end_slot() ?>

```

槽辅助函数之间的代码是在模板的环境里执行的，所以它可以访问所有在动作里定义的变量。`symfony` 会自动把代码的结果放在回应（response）对象里。它不会在模板里显示出来，但是将来可以通过 `include_slot()` 来显示，如例 7-14。

槽在定义与环境有关的内容的时候非常有用。它们也可以用来在布局里增加某些特定动作的 HTML 代码。例如，在显示最新新闻的模板里会想在布局的 `<head>` 里增加 RSS 种子的连接。只要在布局里增加一个 `feed` 槽然后在这个模板里重新定义它就可以了。

SIDEBAR 在哪寻找模板片段

与模板打交道的通常是 web 设计师，他们可能对 `symfony` 不是很了解，而且由于模板分散在整个程序里，寻找模板对他们来说可能有点困难。下面这些可以帮助他们熟悉 `symfony` 的模板系统。

首先，虽然 `symfony` 项目包含很多目录，所有的布局，模板，还有局部模板都放在名叫 `templates/` 的目录里。所以对于设计师来讲，项目结构可以简化成这样：

```

myproject/
  apps/
    application1/
      templates/      # application 1 项目的布局
      modules/
        module1/
          templates/   # module1 模块的模板和局部模板
        module2/
          templates/   # module2 模块的模板和局部模板
        module3/
          templates/   # module3 模块的模板和局部模板

```

所有的目录都可以忽略。

遇到 `include_partial()` 的时候，web 设计师只需要明白只有第一个参数是最重要的。这个参数的内容通常类似于 `module_name/partial_name`，这就是说局部模板的代码位于 `modules/module_name/templates/_partial_name.php`。

`include_component()` 辅助函数，前两个参数是模块名与局部模板名。另外，只要基本了解辅助函数的概念还有基本的辅助函数，设计师就可以开始设计 symfony 程序的模板了。

视图配置

在 symfony 里，视图由两个不同的部分组成：

- 动作(action)结果的 HTML 表现（存放在模板，布局，还有局部模板里）
- 其他部分，包括如下内容：
 - Meta 声明：关键字，描述，缓存时间
 - 页面标题：不仅可以帮你在多个浏览器窗口中找到你要的网页，对搜索引擎同样重要。
 - 文件包含：JavaScript 与 CSS 文件。
 - 布局(layout)：有些动作(action)需要特殊的布局（比如弹出窗口等），或者不需要布局（比如 Ajax 动作）。

在视图中，所有非 HTML 代码的部分都是在视图配置里定义的。symfony 提供了两种修改配置的方法。常用的方法是通过 `view.yml` 配置文件。这种方法适用于与环境无关或者不需要数据库查询的情况。如果需要设置动态值，可以通过在动作里直接修改 `sfResponse` 对象的属性来实现。

NOTE 如果同时在 `view.yml` 配置文件和 `sfResponse` 里定义一个视图配置，以 `sfResponse` 里的定义为准。

view.yml 文件

每个模块都可以有一个 `view.yml` 来定义模块的视图。在这个文件里可以对整个模块的视图作设置也可以对某个视图作特别设置。`view.yml` 文件中第一级别的键名是视图的名字。例 7-16 是一个视图配置文件。

例 7-16 - 模块级 `view.yml`

```
editSuccess:
  metas:
    title: 修改个人资料

editError:
  metas:
    title: 个人资料修改错误

all:
  stylesheets: [my_style]
```

```
metas:
  title: 我的网站
```

CAUTION 注意 `view.yml` 文件的主键是视图的名字，而不是动作名。我们之前有提到过视图的名字由动作名与动作终止组成。例如，如果 `edit` 动作返回 `sfView::SUCCESS`（或者什么也不返回，因为是默认的动作终止），那么视图名应该是 `editSuccess`。

模块的默认设置是在模块的 `view.yml` 文件的 `all` 键下定义的。应用程序的默认设置则是在应用程序的 `view.yml` 文件里定义的。这里你会发现配置文件层叠再次起用：

- `apps/myapp/modules/mymodule/config/view.yml`，视图级的定义仅仅针对一个视图并且会覆盖模块级的定义。
- `apps/myapp/modules/mymodule/config/view.yml` 里的 `all`：定义对模块的所有动作的视图生效并且会覆盖应用程序级别的定义。
- `apps/myapp/config/view.yml` 里的 `default`：定义会对应用程序的所有模块里的所有动作起作用。

TIP 模块级的 `view.yml` 文件默认情况是不存在的。所以第一次修改一个模块的视图配置的时候，需要在模块的 `config/` 目录里建立一个空的 `view.yml` 文件。

在看了例 7-5 例的默认模版和例 7-6 里给出的最终回应的示例以后，你也许会问这些头部 meta 值是哪来的。事实上，它们是项目的 `view.yml` 里的默认视图设置，见例 7-17。

例 7-17 - 默认的应用程序级的视图配置文件，`apps/myapp/config/view.yml`

```
default:
  http_metas:
    content-type: text/html

  metas:
    title:      symfony project
    robots:    index, follow
    description: symfony project
    keywords:   symfony, project
    language:   en

  stylesheets: [main]

  javascripts: [ ]

  has_layout:  on
  layout:      layout
```

这些配置条目会在“视图配置文件”这一节里介绍。

回应对象

尽管回应对象是视图层的一部分，它常常被动作修改。动作可以通过 `getResponse()` 方法来访问 symfony 的回应对象 `sfResponse`。例 7-18 列出一些在动作里常常会用到的 `sfResponse` 对象的方法。

例 7-18 - 在动作里访问 `sfResponse` 对象的方法

```
class mymoduleActions extends sfActions
{
    public function executeIndex()
    {
        $response = $this->getResponse();

        // HTTP 头
        $response->setContentType('text/xml');
        $response->setHttpHeader('Content-Language', 'en');
        $response->setStatusCode(403);
        $response->addVaryHttpHeader('Accept-Language');
        $response->addCacheControlHttpHeader('no-cache');

        // Cookies
        $response->setCookie($name, $content, $expire, $path, $domain);

        // Metas 与 page 头
        $response->addMeta('robots', 'NONE');
        $response->addMeta('keywords', 'foo bar');
        $response->setTitle('My FooBar Page');
        $response->addStyleSheet('custom_style');
        $response->addJavaScript('custom_behavior');
    }
}
```

除了这里看到的 setter 方法外，`sfResponse` 类还有用来返回 response 对象属性的 getter 方法。

头 setters 在 symfony 里功能很强。在 `sfRenderingFilter` 里，头排在比较靠后发送，所以可以随时修改。symfony 提供了一些比较有用的快速的方法。例如，如果你不想在 `setContentType()` 的时候置顶 charset，symfony 会自动加入 `settings.yml` 里面的 charset。

```
$response->setContentType('text/xml');
```

```
echo $response->getContentType();  
=> 'text/xml; charset=utf-8'
```

回应的状态码遵循 HTTP 规范。错误返回 500 状态，页面没找到返回 404 状态，正常情况返回 200 状态，页面没有修改只返回一个 304 状态的头（详见第 12 章）。不过你也可以覆盖这些默认设置在动作里通过 `setStatusCode()` 回应方法使用你自己的状态码。你可以指定一个自定义状态码与状态消息或者一个简单的自定义状态码（这种情况下，`symfony` 会自动添加一个此状态码最常见的状态消息）。

```
$response->setStatusCode(404, '页面已经不存在了');
```

TIP 在发送头之前，`symfony` 会规范它们的名字。这样你就不必担心在 `setHTTPHeader()` 的时候把 `Content-Language` 写成了 `content-language`，因为 `symfony` 会帮你自动的转成正确的方式。

视图配置

你也许会注意到有两种类型的视图配置设定：

- 有唯一值的（在 `view.yml` 文件里值是字符串的，`response` 对象用 `set` 方法来定义）
- 有多个值的（在 `view.yml` 文件里的值是数组，`response` 对象用 `add` 方法来定义）

请注意配置层叠会清除有唯一值的设定，多个值的设定会在后面增加值。读完本章，你能够理解这点了。

Meta 标签配置

回应中 `<meta>` 标签里的信息虽然不会显示在浏览器里，但是对 robots 和搜索引擎很有用。它还能控制每个页面的缓存设定。例 7-19 是通过 `view.yml` 文件里的 `http_metas:` 和 `metas:` 定义的例子，例 7-20 是通过在动作里调用回应（`response`）的 `addHttpMeta()` 和 `addMeta()` 来定义。

例 7-19 - 在 `view.yml` 里 Meta 的键: 值对定义

```
http_metas:  
  cache-control: public  
  
metas:  
  description: Finance in France  
  keywords:    finance, France
```

例 7-20 - 在动作里通过 `response` 对象来定义

```
$this->getResponse()->addHttpMeta('cache-control', 'public');
$this->getResponse()->addMeta('description', 'Finance in France');
$this->getResponse()->addMeta('keywords', 'finance, France');
```

增加一个已经存在的键会覆盖它的当前值。对于 HTTP meta 标签，可以指定第三个参数为 `false` 让 `addHttpMeta()` 方法(也可以是 `setHTTPHeader()`) 在已经存在的键后追加值，而不是替换。

```
$this->getResponse()->addHttpMeta('accept-language', 'en');
$this->getResponse()->addHttpMeta('accept-language', 'fr', false);
echo $this->getResponse()->getHTTPHeader('accept-language');
=> 'en, fr'
```

要使这些标签出现在最后的 HTML 文档里，需要在 `<head>` 标签中使用 `include_http_metas()` 和 `include_metas()` 辅助函数(这是默认情况，见例 7-5)。symfony 会自动把所有 `view.yml` 文件里的设置(包括例 7-11 里的默认设置)还有回应(response)对象的属性聚集起来输出成正确的 `<meta>` 标签。例 7-21 展示了例 7-19 的输出结果。

例 7-21 - Meta 标签在页面里的输出结果

```
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<meta http-equiv="cache-control" content="public" />
<meta name="robots" content="index, follow" />
<meta name="description" content="Finance in France" />
<meta name="keywords" content="finance, France" />
```

另外，即使布局里没有 `include_http_metas()` 或者没有布局，回应的 HTTP 头也会受到 `http-metas:` 定义的影响。例如，如果你想把一个页面显示成纯文本，可以定义如下的 `view.yml` 文件：

```
http_metas:
  content-type: text/plain
```

```
has_layout: false
```

标题配置

页面标题是搜索引擎索引的关键部分。在使用新一代的支持标签页浏览的浏览器的时候页面标题也很有用。在 HTML 里，标题是一个标签同时也是页面的 meta 信息，所以在 `view.yml` 文件里我们会发现 `title:` 键是 `metas:` 键的一个子键。例 7-22 展示了在 `view.yml` 里面定义标题，例 7-23 展示了在动作里定义标题。

例 7-22 - `view.yml` 文件里定义标题

```
indexSuccess:
  metas:
    title: Three little piggies
```

例 7-23 - 在动作里定义标题--可以实现动态标题

```
$this->getResponse()->setTitle(sprintf('%d little piggies', $number));
```

在最终输出文档的<head>部分，如果 include_metas() 辅助函数存在，标题定义会设置<meta name="title">标签，如果 include_title() 辅助函数存在的话，会设置<title>标签。如果两者同时存在（如例 7-5 的默认布局），标题会在页面源代码里出现两次（见例 7-6），当然这并没有坏处。

文件包含配置

载入一个样式表或者 Javascript 很容易，如例 7-24 与 例 7-25 所示。

例 7-24 - view.yml 里的文件包含

```
indexSuccess:
  stylesheets: [mystyle1, mystyle2]
  javascripts: [myscript]
```

例 7-25 - 在动作(action)里的文件包含

```
$this->getResponse()->addStyleSheet('mystyle1');
$this->getResponse()->addStyleSheet('mystyle2');
$this->getResponse()->addJavaScript('myscript');
```

在上面的例子中，参数是文件名。如果文件的扩展名合理（.css 对应样式表，.js 对应 JavaScript 文件），你可以省略。如果文件的位置恰当（样式表在/css/目录下，JavaScript 在/js/目录下），你也可以省略这个位置。symfony 可以聪明的自己加上扩展名，找到位置。

与 meta 和标题定义不同，文件包含的定义不需要在模板或者布局里使用任何辅助方法。这就是说，不论模板和布局的内容怎么变化，前面例子中的设置都会输出例 7-26 所示的 HTML 代码。

例 7-26 - 输出结果中的文件载入--不需要在布局里使用辅助函数

```
<head>
...
<link rel="stylesheet" type="text/css" media="screen"
href="/css/mystyle1.css" />
<link rel="stylesheet" type="text/css" media="screen"
href="/css/mystyle2.css" />
```



```
<script language="javascript" type="text/javascript"
src="/js/myscript.js">
</script>
</head>
```

NOTE 回应里的样式表与 JavaScript 文件的包含由 `sfCommonFilter` 的过滤器来完成。它会在回应里寻找 `<head>` 标签，然后把 `<link>` 与 `<script>` 标签添加到 `</head>` 标签之前。也就是说，如果你的布局或者模板里没有 `<head>` 标签，就不会进行载入。

请记住配置层叠在这里同样起作用，所以在应用程序的 `view.yml` 里面定义的文件载入会在应用程序的每个页面里面出现，如例 7-27，例 7-28 和例 7-29 所演示的。

例 7-27 - 应用程序的 `view.yml`

```
default:
  stylesheets: [main]
```

例 7-28 - 模块的 `view.yml`

```
indexSuccess:
  stylesheets: [special]

all:
  stylesheets: [additional]
```

例 7-29 - `indexSuccess` 视图的输出结果

```
<link rel="stylesheet" type="text/css" media="screen"
href="/css/main.css" />
<link rel="stylesheet" type="text/css" media="screen"
href="/css/additional.css" />
<link rel="stylesheet" type="text/css" media="screen"
href="/css/special.css" />
```

如果想去掉高级别的配置文件里定义的文件，只要在低级别的配置文件的这个文件的名字前加一个减号（-）就可以了，如例 7-30。

例 7-30 - 某模块的 `view.yml`，这个配置文件里去掉了应用程序级里定义的一个文件

```
indexSuccess:
  stylesheets: [-main, special]
```

```
all:
  stylesheets: [additional]
```

如果要去掉所有的样式表与 JavaScript，可以使用下面的语法：

```
indexSuccess:
  stylesheets: [-*]
  javascripts: [-*]
```

如果要更准确，可以通过一个附加参数指定一个文件载入的位置（第一或者是最后）：

```
// view.yml 里
indexSuccess:
  stylesheets: [special: { position: first }]

// 动作(action)里
$this->getResponse()->addStyleSheet('special', 'first');
```

如果要指定样式表的媒体（media），可以修改默认的样式表标签选项，如例 7-31, 7-32, 7-33 所示。

例 7-31 - view.yml 里包含媒体选项的样式表载入

```
indexSuccess:
  stylesheets: [main, paper: { media: print }]
```

例 7-32 - 动作（action）里包含媒体选项的样式表载入

```
$this->getResponse()->addStyleSheet('paper', '', array('media' =>
'print'));
```

例 7-33 - 结果

```
<link rel="stylesheet" type="text/css" media="print"
href="/css/paper.css" />
```

布局配置

根据网站的实际图，你可能会需要好几种布局。经典的网站至少有两种布局：默认布局与弹出布局。

我们已经知道默认布局是在 myproject/apps/myapp/templates/layout.php。另外的布局也应该在全局的 templates/目录里。如果你想使用 myapp/templates/my_layout.php 这个布局文件，在 view.yml 里需要例 7-34 那样的语法，在动作里需要使用 7-35 所示的方法。

例 7-34 - view.yml 里的布局定义

```
indexSuccess:
  layout: my_layout
```

例 7-35 - 动作(action)里的布局定义

```
$this->setLayout('my_layout');
```

有些视图不需要任何布局（例如，纯文本或者 RSS 种子）。这种情况下，要把 has_layout 设置成 false，如例 7-36 和 7-37 所示。

例 7-36 - view.yml 里去掉布局

```
indexSuccess:
  has_layout: false
```

例 7-37 - 在动作(action)里去掉布局

```
$this->setLayout(false);
```

NOTE Ajax 动作(action)默认就没有布局。

组件槽（Component Slots）

结合视图组件与视图配置的力量为视图开发带来了新的方法：组件槽系统。它是一个特殊的专注于重用性和层分离的槽(slot)。所以组件槽比槽的架构更好，但是速度稍稍慢一点。

与槽(slot)一样，组件槽也是视图元素里定义的有名字的占位符。不同点是填充代码的确定方式。槽的填充代码是在另外的视图元素里定义的；对组件槽来说，填充代码是一个组件的运行结果，这个组件的名字通过视图配置确定。看了例子你就能够更好的理解组件槽。

可以用 include_component_slot() 来设定一个组件槽占位符。这个函数需要一个标签作为参数。例如，假设应用程序的 layout.php 包含一个跟上下文有关的侧边栏。例 7-38 向我们展示了如何载入组件槽。

例 7-38 - 载入一个名叫 sidebar 的组件槽

```
...
<div id="sidebar">
  <?php include_component_slot('sidebar') ?>
</div>
```

然后在视图配置里定义贴上？？？ sidebar 标签的组件槽对应哪个组件。例如，在应用程序 view.yml 的 components 里定义 sidebar 默认的组件。键名是组件槽的标签；它的值必须是一个包含模块名还有组件名的数组。如例 7-39。

例 7-39 - 在 myapp/config/view.yml 里定义 sidebar 组件槽的默认组件

```
default:
  components:
    sidebar: [bar, default]
```

这样执行布局的时候，sidebar 组件槽会被 barComponents 类的 executeDefault() 方法的结果填充，这个方法会显示 modules/bar/templates/目录下的 _default.php 局部模板文件。

配置层叠是你能够在某个模块里重新定义组件槽的组件。例如，在 user 模块里，你可能会想要一个跟上下文有关的组件显示用户的名字和用户发表的文章的数量。这样，需要例 7-40 所示的，在这个模块的 view.yml 里给 sidebar 组件槽指定与默认值不同的值。

例 7-40 - 在 myapp/modules/user/config/view.yml 里给 sidebar 组件槽指定不同于默认值的值。

```
all:
  components:
    sidebar: [bar, user]
```

这里定义的组件如例 7-41 所示。

例 7-41 - sidebar 槽的组件，modules/bar/actions/components.class.php

```
class barComponents extends sfComponents
{
    public function executeDefault()
    {
    }

    public function executeUser()
    {
        $current_user = $this->getUser()->getCurrentUser();
        $c = new Criteria();
        $c->add(ArticlePeer::AUTHOR_ID, $current_user->getId());
        $this->nb_articles = ArticlePeer::doCount($c);
        $this->current_user = $current_user;
    }
}
```

例 7-42 是两个组件的结果

例 7-42 - sidebar 组件槽的局部模板, modules/bar/templates/

```
// _default.php
<p>This zone contains contextual information.</p>

// _user.php
<p>User name: <?php echo $current_user->getName() ?></p>
<p><?php echo $nb_articles ?> articles published</p>
```

组件槽可以用在面包屑型的导航连接, 上下文相关的导航, 还有各种动态插入。作为组件, 它们可以用在全局模板, 或者普通模板, 甚至在其他的组件里。组件的配置设定总是从最后一个执行的动作中取得。

如果你在某个模块里想暂停使用一个组件, 只要再声明一个空的模块/组件定义就可以了, 如例 7-43 所示。

例 7-43 - 在 view.yml 里取消一个组件槽

```
all:
  components:
    sidebar: []
```

输出转义 (Output Escaping)

当你在模板里加入动态数据的时候, 必须确保数据的完整性。例如, 如果数据来自匿名用户填写的表单, 就有可能包含恶意的用来发起 cross-site scripting(XSS)攻击的脚本。所以必须转义替换输出的数据, 使得里面包含的 HTML 标签变得无害。

举例来说, 假设一个用户用下面的内容填写表单:

```
<script>alert(document.cookie)</script>
```

如果把这个值直接输出, 这段 JavaScript 就会在浏览器里执行, 如果用户输入危险的攻击脚本就不止是显示一个提示框那么简单了。所以在显示用户输入的数据之前必须先进行转义替换, 这样输入的值会变成下面这样:

```
&l t; scrip t&gt; alert(document.cookie)&l t; /scrip t&gt;
```

你可以手动的给每个不确定的值加上 `html entities()` 来进行转义替换, 不过这很麻烦而且容易出错。symfony 提供了一种特殊的方式, 叫做输出转义, 它会自动的对模板里的每个输出的变量进行转义替换。可以通过修改应用程序的 settings.yml 里的一个参数来开启这个功能。

开启输出转义

输出转义是在应用程序的 `settings.yml` 文件里设置，对整个应用程序生效。输出转义有两个配置参数：`strategy` 控制变量怎么传给视图，`method` 决定默认转义函数。

下一节将会详细介绍这些设置，不过基本上你只要将 `escaping_strategy` 参数从默认的 `bc` 设置成 `both` 就可以开启输出转义了，如例 7-44。

例 7-44 - 在 `myapp/config/settings.yml` 里开启输出转义

```
all:
  . settings:
    escaping_strategy: both
    escaping_method:   ESC_ENTITIES
```

上面的配置会自动在所有的输出变量上实施 `htmlentities()`。例如，假设你在动作(action)里定义了一个 `test` 变量：

```
$this->test = '<script>alert(document.cookie)</script>';
```

输出转义开启后，在模板里输出这个变量会显示下面的数据：

```
echo $test;
=> &lt;&lt;script&gt;alert(document.cookie)&lt;/script&gt;
```

开启输出转义会在模板里面增加一个 `$sf_data` 变量。这是一个包含了所有被转义替换了的变量的对象。所以你也可以这样输出 `test` 变量：

```
echo $sf_data->get('test');
=> &lt;&lt;script&gt;alert(document.cookie)&lt;/script&gt;
```

TIP `$sf_data` 对象实现了数组接口，所以除了使用 `$sf_data->get('myvariable')` 之外，还可以使用 `$sf_data['myvariable']` 来获取被转义替换的值。但是它并不是一个真正的数组，所以 `print_r()` 这类函数用在 `$sf_data` 上不会如你想像的那样工作。

通过这个对象也可以访问到未被转义的数据。这在把一个存放 HTML 代码的变量显示出来的时候很有用，这需要你信任这个变量。需要输出原始数据的时候可以执行 `getRaw()` 方法。

```
echo $sf_data->getRaw('test');
=> <script>alert(document.cookie)</script>
```

当你要变更包含 HTML 代码的变量输出成 HTML 的时候，你就需要访问原始数据。这就是为什么默认的布局里会用`$sf_data->getRaw('sf_content')`来包含模板，而不是直接使用`$sf_content`，因为直接使用`$sf_content` 在输出转义开启的时候会使模板乱掉。

转义策略

转义策略（`escaping_strategy`）的设置决定变量输出的方式。下面是它的可能取值：

- **bc** (向后兼容模式)：变量不会自动转义，但是可以通过`$sf_data` 容器访问转义替换后的版本。所以数据默认就是原始的，除非你使用`$sf_data` 里的转义之后的值。这是默认值，在这种情况下你的应用程序有被 XSS 攻击的危险。
- **both**：所有的变量会自动被转义替换。转义替换后的值也可以通过`$sf_data` 容器访问。推荐这种策略，因为只有在输出原始数据的时候才有被攻击的危险。某些时候，需要用到未转义的数据，例如，如果你需要输出包含 HTML 的代码显示在浏览器里。所以请注意，如果你的程序开发到一半改成这个策略，有些功能会坏掉。最好是一开始就使用这个设定。
- **on**：变量的值只能通过`$sf_data` 容器来访问。这是处理转义最安全快速的方法，因为每次输出变量你都要决定使用`get()`取得转义版本还是使用`getRaw()`取得原始版本。这样你就总是能够了解数据破坏的可能性。
- **off**：关闭输出转义。模板里不能使用`$sf_data` 容器。如果你确定不会用到转义数据，你可以使用这个策略而不是 **bc** 来加快程序的速度。

转义辅助函数

转义辅助函数是用来返回输入变量的转义版本的函数。可以在 `settings.yml` 中的 `escaping_method` 里面定义或者对视图里某个特定值直接使用一个转义方法。转义辅助函数包括：

- **ESC_RAW**：不转义。
- **ESC_ENTITIES**：使用 PHP 函数 `htmlentities()` 的 `ENT_QUOTES` 转义方式进行转义。
- **ESC_JS**：将包含 HTML 代码的值转义使它能够在 JavaScript 的字符串里。用 JavaScript 动态改变 HTML 的时候很有用。
- **ESC_JS_NO_ENTITIES**：转义一个值从而用于 JavaScript 字符串，但不转义 HTML 实体。这在把这个值的内容用对话框显示出来的时候很有用（例如，`myString` 变量用于 `javascript:alert(myString);`）。

转义数组与对象

输出转义不仅可以用于字符串，也可以用于数组和对象任意对象或数组会将他们的转义状态传递到它们下一级。假设你的转义策略是 both，例 7-45 演示了转义层叠。

例 7-45 - 转义也作用于数组和对象

```
// 类定义
class myClass
{
    public function testSpecialChars($value = '')
    {
        return '<'.$value.'>';
    }
}

// 在动作里
$this->test_array = array('&', '<', '>');
$this->test_array_of_arrays = array(array('&'));
$this->test_object = new myClass();

// 在模板里
<?php foreach($test_array as $value): ?>
    <?php echo $value ?>
<?php endforeach; ?>
=> & < >
<?php echo $test_array_of_arrays[0][0] ?>
=> &
<?php echo $test_object->testSpecialChars('&') ?>
=> & & >
```

事实上，模板里的变量可能不是你期望的那样。输出转义系统会“装饰”它们，把它们转换成特殊的对象：

```
<?php echo get_class($test_array) ?>
=> sfOutputEscaperArrayDecorator
<?php echo get_class($test_object) ?>
=> sfOutputEscaperObjectDecorator
```

这就解释了为什么有些 PHP 函数（像 `array_shift()`, `print_r()` 等）对转义过的数组不起作用。但是它们仍然可通过 `[]` 来访问，可以用 `foreach` 来遍历，`count()` 的结果也正确（`count()` 只在 PHP5.2 及以后版本工作正常）。在模板里，数据应该是只读的，所以大多数访问应该在模型或者动作的方法里完成。

通过 `$sf_data` 对象我们仍然可以访问原始数据。另外，转义过的对象会被修改从而接受一个额外的参数：转义方法。所以在显示模板里的变量的时候你可以选

择其他的转义方法，或者使用 ESC_RAW 辅助函数来取消转义。让我们来看 例 7-46：

例 7-46 - 转义过的对象的方法接受额外的参数

```
<?php echo $test_object->testSpecialChars('&') ?>
=> &lt;&gt;
// 下面三行返回同样的结果
<?php echo $test_object->testSpecialChars('&', ESC_RAW) ?>
<?php echo $sf_data->getRaw('test_object')->testSpecialChars('&') ?>
<?php echo $sf_data->get('test_object',
ESC_RAW)->testSpecialChars('&') ?>
=> <&>
```

如果要在模板里处理对象，你可能会大量的利用这个额外参数的技巧，这是从方法调用中取得原始数据的最快的方法了。

CAUTION 输出转义开启后，symfony 变量也会被转义。所以当心虽然 \$sf_user, \$sf_request, \$sf_param 和 \$sf_context 还能用，不过它们的方法会返回转义后的数据，除非你将 ESC_RAW 作为最后一个参数传给方法。

总结

表现层可以使用各种工具。辅助方法可以加快写模板的速度。布局，局部模板，组件和组件槽能使表现层模块化，可重用。由于 YAML 的书写速度快，所以通过视图配置我们可以很快的修改几乎所有页面的 header 部分。配置层叠使你不必在每个视图里定义所有的设置。如果表现层的改变需要动态数据，需要从动作里访问 sfResponse 对象。由于有输出转义系统，可以免除 XSS 攻击的危险。

第 8 章 - 深入模型层

到目前为止本书大多数的讨论都专注于建立页面，处理请求与回应。但是网页应用程序的业务逻辑大多依赖于它的数据模型。symfony 的默认模型组件是基于一个对象/关系映射层，也就是我们所知的 Propel 项目 (<http://propel.phpdb.org/>)。在 symfony 应用程序中，你不用关注数据库的实际位置，而是通过对象来访问储存在数据库中的数据。这保持了高度的抽象和可移植性。

本章解释了如何建立一个数据对象模型和如何通过 Propel 来访问和修改数据。同时也展示了 symfony 是如何整合 Propel 的。

为什么使用 ORM 和抽象层？

数据库是关系型的。PHP 5 和 symfony 都是面向对象的。为了在面向对象环境中最有效的访问数据库，需要一个接口用来把对象逻辑转换为关系逻辑。如第一章所述，这个接口就叫做对象-关系映射(ORM)，它是由可以访问数据和保持业务规则的对象组成的。

ORM 最大的优势就是可重用性，它允许数据对象的方法可以被应用程序的其他部分调用，甚至可以从不同的应用程序中调用。ORM 层也可以封装数据逻辑，例如，基于用户的贡献度和如何作出的贡献来计算论坛用户的评分。当一个页面需要显示例如一个用户的评分，不需要担心如何去计算，只要很简单的调用数据模型的方法即可。如果以后计算方法有所变化，你只需要修改模型的评分方法即可，应用程序的其他部分不需要改变。

使用对象来代替记录，用类来代替表，还有其他好处：他们允许你在对象中增加一个新的读取方法而不需要对应到表的一个列。例如，如果你有一个 client 表，它拥有两个字段分别叫做 first_name 和 last_name，你可能只想要获得一个 Name。在面向对象的世界里，Client 类中增加一个存取方法是非常简单的，如例 8-1 所示。从应用程序的角度来看，Client 类的 FirstName, LastName 和 Name 属性没有什么区别。只有类本身才能决定属性所对应的数据库的列。

例 8-1 - 在模型类中的存取方法掩盖了实际表结构

```
public function getName()
{
    return $this->getFirstName().' '.$this->getLastName();
}
```

所有重复的数据访问函数和数据自身的业务逻辑可以存在对象中。假设你有一个 ShoppingCart 类，里面有一个 Item（是个对象）。只要写一个自定义的方法来封装实际的计算过程，就可以在结账时得到购物车的总价。如例 8-2 所示。

例 8-2 - 存取方法掩盖了数据逻辑

```
public function getTotal()  
{  
    $total = 0;  
    foreach ($this->getItems() as $item)  
    {  
        $total += $item->getPrice() * $item->getQuantity();  
    }  
  
    return $total;  
}
```

在建立数据访问过程的时候还要考虑另外一个要点：数据库厂商所使用的不同的 SQL 语法变种。换用另外一个数据库管理系统（DBMS）会让你不得不重写一部分为以前设计的 SQL 查询。如果用数据库独立语法来建立一个查询，并把实际 SQL 翻译为第三方语言，换数据库系统就不会麻烦了。这就是数据抽象层存在的目的。它强制让你使用特定的语法规则来写查询，同时把它转到到相应的 DBMS 并优化 SQL 语句。

抽象层的主要优势是可移植性，因为他让换用另一种数据库成为可能，甚至可以在项目中后期换用。假设你需要为应用程序写一个快速原型，但客户还没有确定哪个数据库最适合他。你能先用 SQLite 建立你的应用程序，然后在客户有了决定后切换到 MySQL，PostgreSQL 或者 Oracle。这只要改变配置文件中一行代码即可。

symfony 使用 Propel 来实现 ORM，Propel 使用 Creole 做数据库抽象。这两个第三方组件，都是由 Propel 小组开发的，并且都无缝集合到了 symfony 中，你可以把他们作为框架的一部分。在本章描述的 Propel 和 Creole 的约定和语法规则都被改写过，因此 symfony 的语法与原始语法会有一些不同。

NOTE 在 symfony 项目中，所有的应用程序共享同一个模型。这就是项目层的全局观：依靠通用商业规则重组应用程序。这就是让模型独立于应用程序之外并且模型文件存在项目根目录的 lib/model/目录下的原因。

symfony 的数据库设计(schema)

为了创建 symfony 使用的数据对象模型，需要把数据库关系模型翻译为对象数据模型。ORM 需要关系模型的描述来做映射，这就叫做设计。在设计中，你定义表，表之间的关系，和表中列的特性。

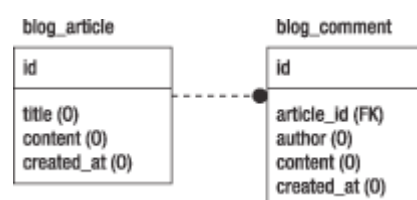
symfony 中设计的语法使用了 YAML 格式。schema.yml 文件必须放在 myproject/config/目录下。

NOTE symfony 也接受 Propel 原生的 XML 设计格式，在本章稍后的超越 schema.yml：schema.xml 小节会做描述。

设计示例

如何把数据库结构转换为设计呢？看例子是最好的理解方法。试想一下你有一个 blog 数据库，包含两个表 blog_article 和 blog_comment，结构如图 8-1 所示。

图 8-1 - 一个 blog 数据库表结构



对应的 schema.yml 文件应该看上去如例 8-3 所示。

例 8-3 - schema.yml 示例

```
propel:
  blog_article:
    _attributes: { phpName: Article }
  id:
  title:      varchar(255)
  content:    longvarchar
  created_at:
  blog_comment:
    _attributes: { phpName: Comment }
  id: Mutator
  article_id:
  author:     varchar(255)
  content:    longvarchar
  created_at:
```

注意数据库名字本身（blog）并没有出现在 schema.yml 文件中。反而在连接名下有数据库描述（本例中是 propel）。这是因为实际的连接设置可以依照应用程序运行的环境来定。例如，当你在开发环境中运行应用程序时，你会访问

一个开发数据库（也许是 `blog_dev`），但是生产数据库也用的是同一个设计文件。在 `database.yml` 文件中有连接，本章稍后的“数据库连接”会有介绍。设计不包含任何连接设置的细节，只有连接名用来保持数据库抽象。

基本设计语法

在 `schema.yml` 文件中，第一个关键字表示的是连接名。它可以包含多个表，每个表都有一些列。根据 YAML 语法，关键字用冒号作结束标记，结构用缩进来表示（一个或多个空格，但不是制表符）。

表可以有特殊的属性，包括 `phpName`（会生成同名的类）。如果没有设置表的 `phpName`，`symfony` 会以表名的驼峰命名法来创建类。

TIP 驼峰命名法把单词之间的下划线去掉了，并把每个单词的首字母大写。`blog_article` 和 `blog_comment` 的默认驼峰命名法版本是 `BlogArticle` 和 `BlogComment`。这种转换方法约定字的首字母大写，就像骆驼的驼峰一样。

一个表包含了许多列。列的值可以用三种方式来定义：

- 如果你没有给出定义，`symfony` 会根据列名和一些约定来猜测最适合的属性，这些在下面的“空列”段落会有描述。例如，在例 8-3 中的 `id` 列不需要定义。`symfony` 会定义它为自增长的数值类型，表的主键。`blog_comment` 表的 `article_id` 会理解为 `blog_article` 表的外键（结尾是 `_id` 的列被理解为外键，根据列前面部分的名字来确认是和哪张表有关联）。`create_at` 列会自动设置为 `timestamp` 类型。对于这种类型的列，你不需要特别指定他们的类型。这就是为什么说 `schema.yml` 是非常容易写的原因。
- 如果你只定义了一个属性，那这就是列的类型。`symfony` 支持一些常用的列类型：`boolean`，`integer`，`float`，`date`，`varchar(size)`，`longvarchar`（转换过的，例如，在 MySQL 下就会转换为 `text`）和其他。对于 `text` 内容超过 256 个字符的，需要使用 `longvarchar` 类型，这是没有大小限制的（MySQL 中不能超过 65KB）。注意 `date` 和 `timestamp` 类型有通常的 Unix 日期限制并且不能设置早于 1970-01-01。如果需要设置一个更早的日期（例如，生日），“unix 日期之前”的日期类型可以使用 `bu_date` 和 `bu_timestamp`。
- 如果需要定义其他的列属性（如默认值，必填属性或者其他），你需要把列属性写为一组 `key: value`。这种扩充设计语法会在本章稍后介绍。

列可以有一个 `phpName` 属性，它是首字母大写的（`Id`，`Title`，`Content`，等）并且在大多数情况下不能覆盖。

表也可以包含详细的外键和索引，以及少量的数据库结构定义。参考本章节后面的“扩展设计语法”部分。

模型类

设计是用来建立 ORM 层的模型类的。为了省时，这些类是通过命令行调用 `propel -build-model` 来生成的。

```
>symfony propel -build-model
```

输入这个命令后会先分析模型接着在项目的 `lib/model/om` 目录下生成基础数据模型类：

- `BaseArticle.php`
- `BaseArticlePeer.php`
- `BaseComment.php`
- `BaseCommentPeer.php`

还有，实际的数据模型类会建立在 `lib/model` 下：

- `Article.php`
- `ArticlePeer.php`
- `Comment.php`
- `CommentPeer.php`

你只定义了两个表，但会生成八个文件。这并无不妥， 但应该解释一下。

基础类和自定义类

为什么我们在两个不同的目录保留了两个版本的数据对象模型？

你也许会需要在模型对象中增加自定义方法和属性（试想例 8-1 中的 `getName()` 方法）。但是由于项目开发需要，你将会需要增加表或者列。当你修改了 `schema.yml` 文件时，需要重新调用 `propel -build-model` 来生成对象模型类。如果你的自定义方法写在自动生成的类中，他们会在每一次重新生成的时候被覆盖。

由设计直接生成的 Base 类放在 `lib/model/om/` 目录中。你永远不需要去修改他们，因为每一次新建模型都会完全删除这些文件。

另一方面，自定义对象类会放在 `lib/model/` 目录下，实际上是继承自 Base 类。 当对已有的模型调用 `propel -build-model` 任务时，这些类不会被修改。因此这就是你可以增加自定义方法的地方。

例 8-4 展示了第一次调用 `propel -build-model` 任务建立的自定义模型类的一个示例。

例 8-4 - `lib/model/Article.php` 中的模型类文件示例

```
<?php

class Article extends BaseArticle
{
}
```

它继承了 `BaseArticle` 类所有的方法，但是修改设计不会影响到这个文件。

用自定义类来扩展基础类的机制可以让你在不知道最终数据库中模型之间的关系的时候开始编程。相关的文件结构会让模型既可以自定义又可以进化。

对象和 Peer 类

`Article` 和 `Comment` 是用来显示数据库中记录的对象类。他们赋予了记录的列和相关记录的访问权限。这就是说你可以调用 `Article` 对象的方法来获取文章的标题，如例 8-5 所示。

例 8-5 - 在对象类中获得记录列

```
$article = new Article();
...
$title = $article->getTitle();
```

`ArticlePeer` 和 `CommentPeer` 都是 `peer` 类；因此，类包含了静态方法来操作表。他们提供了从表中获得记录的方法。他们的方法通常返回了一个对象或是相关对象类的对象的集合，如例 8-6 所示。

例 8-6 - `Peer` 类可以用静态方法来获得记录

```
$articles = ArticlePeer::retrieveByPks(array(123, 124, 125));
// $articles 是一个 Article 类的对象数组
```

NOTE 从数据模型的角度来看，不可能有 `peer` 对象。这就是为什么调用 `peer` 类的方法会使用 `::`（调用静态方法）而不是通常的 `->`（调用实例方法）。

所以把对象类和 `peer` 类的基础类和自定义类加起来，数据库设计里的一个表会自动生成四个类。事实上，有第五种类生成在 `lib/model/map/` 目录下，它们包含了关于表运行时所需要的 `metadata` 信息。但是也许你永远不需要修改这个类，你完全可以忘了它。

访问数据

在 symfony 中，是通过对象来访问数据的。如果你习惯使用关系模型和使用 SQL 来获取、修改你的数据的话，对象模型方法会让你觉得有些复杂。但是当你尝试过用面向对象方法来访问数据的话，就会喜欢上它的。

但是首先，让我们确信我们说的是同一个词汇。 关系型和对象数据模型有一些相似点，但是他们都有自己的术语：

关系的	面向对象的
表	类
行，记录	对象
字段，列	属性

获得列值

当 symfony 建立模型时，它为每一个在 schema.yml 中存在的表都建立一个基础对象类。每一个类都有一个基于列定义的默认的构造器， 读取方法和设置方法：new, getXXX() 和 setXXX()方法帮助创建对象并给予访问对象属性的权限，如例 8-7 所示。

例 8-7 - 生成对象类方法

```
$article = new Article();
$article->setTitle('My first article');
$article->setContent('This is my very first article.\n Hope you enjoy it!');

$title = $article->getTitle();
$content = $article->getContent();
```

NOTE 生成的对象类名为 Article，这是由 blog_article 表的 phpName 定义的。如果在设计中没有定义 phpName，这个类就会取名为 BlogArticle。读取方法和设置方法使用驼峰命名法的变异来定义列名，所以 getTitle()方法会获得 title 列的值。

可以使用 fromArray()方法一次定义多个字段，在生成的每个类对象中都有此方法，如例 8-8 所示。

例 8-8 - fromArray()方法是一个多重设置方法

```
$article->fromArray(array(
    'title' => 'My first article',
    'content' => 'This is my very first article.\n Hope you enjoy it!',
```



```
));
```

获得相关联的数据

在 `blog_comment` 表中 `article_id` 列实际上定义了 `blog_article` 表的一个外键。每一个 `comment` 都与一篇文章相对应，同时一篇文章可以有多个 `comment`。生成的类包含五个方法来把这些对应关系转换成面向对象的方法，如下：

- `$comment->getArticle()`: 获得相关联的 `Article` 对象
- `$comment->getArticleId()`: 获得相关联的 `Article` 对象的 ID
- `$comment->setArticle($article)`: 定义相关联的 `Article` 对象
- `$comment->setArticleId($id)`: 通过 ID 定义相关联的 `Article` 对象
- `$article->getComments()`: 获得相关联的 `Comment` 对象

`getArticleId()` 和 `setArticleId()` 方法说明了你可以把 `article_id` 列作为一个普通列并手动设置对应关系，但这么做并不好。面向对象方法的优点让其他三个方法更容易理解。例 8-9 显示了如何使用生成的设置方法。

例 8-9 - 外键转换为特别的设置方法

```
$comment = new Comment();
$comment->setAuthor('Steve');
$comment->setContent('Gee, dude, you rock: best article ever!');

// 把此 comment 和 $article 对象关联
$comment->setArticle($article);

// 另一种语法
// 仅当对象已经存在于数据库中才有意义
$comment->setArticleId($article->getId());
```

例 8-10 展示了生成的获取方法是如何使用的。同时也演示了如何在模型对象中调用关联方法。

例 8-10 - 外键转为特别的 getters

```
// 多对一关系
echo $comment->getArticle()->getTitle();
=> My first article
echo $comment->getArticle()->getContent();
=> This is my very first article.
    Hope you enjoy it!
```

```
// 一对多关系
$comments = $article->getComments();
```

`getArticle()` 方法返回了一个 `Article` 类的对象，从而可以使用 `getTitle()` 获取方法。这比直接使用 `join` 要好得多，而仅仅只会多几行代码（从调用 `$comment->GetArticleId()` 开始）。

例 8-10 中的 `$comments` 变量包含了 `Comment` 类的一个对象数组。你能用 `$comments[0]` 来显示第一个对象或是用 `foreach($comments as $comment)` 来遍历这个对象数组。

NOTE 你现在知道为什么模型对象是以单数命名的了。在 `Comment` 对象名字后面增加 `s`，会在 `blog_comment` 表中制造一个外键并产生建立 `getComments()` 方法的动作。如果你给模型对象一个复数名字，生成时候会产生一个叫做 `getCommentss()` 的无意义的方法。

保存和删除数据

创建一个新的对象可以通过调用 `new` 构造器，但修改此对象不会对数据库有任何影响，也就是说这并不对应于 `blog_article` 表中存在的实际记录。但你可以调用对象的 `save()` 方法把数据保存到数据库中。

```
$article->save();
```

ORM 可以查明对象之间的关系，因此保存 `$article` 对象同时也就保存了相关的 `$comment` 对象。也就是说它知道保存了的对象在数据库中有关联的数据，当调用 `save()` 的时候，有时候会转换为 `INSERT` 语句，有时候会使 `UPDATE` 语句。`save()` 方法会自动设置主键，所以在保存后，你能用 `$article->getId()` 得到一个新的主键。

TIP 你能通过调用 `isNew()` 来检查对象是否是新建的。如果你想知道对象是否被修改过是否该保存，可以调用它的 `isModified()` 方法。

如果你读了文章的 `comment`，也许会后悔把他们发布到互联网上。如果你觉得一些回复者的回复不合适的话，可以很方便的使用 `delete()` 方法来删除评论，如例 8-11 所示。

例 8-11 - 用 `delete()` 方法从数据库删除记录的相关对象

```
foreach ($article->getComments() as $comment)
```

```
{  
    $comment->delete();  
}
```

TIP 在调用 delete() 方法后，请求结束之前对象依旧可以访问。
要确认是否在数据库中已经把对象删除的话就需要调用
isDeleted() 方法了。

通过主键来获得记录

如果你知道特定记录的主键值，可以使用 peer 类的 retrieveByPk() 方法来获得相关对象。

```
$article = ArticlePeer::retrieveByPk(7);
```

schema.yml 文件中定义了 id 字段作为 blog_article 表的主键，因此这个语句会返回 id 为 7 的文章。由于你使用了主键，所以只会返回一条记录；\$article 变量包含了类 Article 的对象。

有时候，也许包含了多个主键（复合主键）。在这些情况中，retrieveByPK() 方法会接受多个参数，每一个对应一个主键。

你也能用 retrieveByPKs() 方法，输入一组主键组成的数组作为参数来获得多个对象。

通过 Criteria 获得数据

当你想获得多个记录时，你需要调用 peer 类的 doSelect() 方法来获得你想要的对象。例如，调用 ArticlePeer::doSelect() 来获得 Article 类的对象。

doSelect() 方法的第一个参数是 Criteria 类的一个对象，Criteria 类是一个简单查询定义类，它为了用数据库抽象而没有使用 SQL。

一个空的 Criteria 返回了类的所有对象。例如，例 8-12 的代码就返回了所有的 article。

例 8-12 - 通过 Criteria 的 doSelect() 来获得数据--空的 Criteria

```
$c = new Criteria();  
$articles = ArticlePeer::doSelect($c);
```

```
// 和下面 SQL 查询结果是一样的  
SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT,
```

```
        blog_article.CREATED_AT
FROM    blog_article;
```

SIDEBAR 化合 (hydrating)

调用 `::doSelect()` 比使用简单的 SQL 查询强大的多。首先，SQL 会针对使用的 DBMS 而优化。其次，任何传递给 `Criteria` 的值都会在整合入 SQL 代码之前被转义，这能防止 SQL 注入的风险。第三点，此方法返回了一个对象数组而不是一个结果集。ORM 基于数据库结果集自动创建并丢出对象。这个过程叫做化合 (hydrating)。

如果遇到一个更复杂的对象选择时，你需要用到 `WHERE`，`ORDER BY`，`GROUP BY` 和其他 SQL 语句。`Criteria` 对象有针对所有这些情况的方法和参数。例如，在例 8-13 中我们建立了一个 `Criteria` 来取得 Steve 写的所有的 comments，按照日期排序。

例 8-13 - 通过 `Criteria` 的 `doSelect()` 来获得记录--有条件的 `Criteria`

```
$c = new Criteria();
$c->add(CommentPeer::AUTHOR, 'Steve');
$c->addAscendingOrderByColumn(CommentPeer::CREATED_AT);
$comments = CommentPeer::doSelect($c);
```

```
// 等同于下面 SQL 语句执行的结果
SELECT blog_comment.ARTICLE_ID, blog_comment.AUTHOR,
       blog_comment.CONTENT,
       blog_comment.CREATED_AT
FROM    blog_comment
WHERE   blog_comment.author = 'Steve'
ORDER BY blog_comment.CREATED_AT ASC;
```

把类常量作为参数传递给 `add()` 方法，参考属性名字。他们的名字都是列名的大写字符版本。例如，要在 `blog_article` 表中增加 `content` 列，用 `ArticlePeer::CONTENT` 类常量。

NOTE 为什么使用 `CommentPeer::AUTHOR` 来代替 `blog_comment.AUTHOR`，这是因为他会输出 SQL 查询语句？假设你需要在数据库中更改 `author` 字段为 `contributor`。如果你是用 `blog_comment.AUTHOR`，你就需要在每个调用模型的地方都进行修改。而另外一个，使用 `CommentPeer::AUTHOR`，只要简单的在 `schema.yml` 文件中修改字段名，设置 `phpName` 为 `AUTHOR` 然后重新编译模型即可。

表 8-1 对比了 SQL 语法和 Criteria 对象语法。

表 8-1 - SQL 和 Criteria 对象语法

SQL	Criteria
WHERE column = value	->add(column, value);
WHERE column <> value	->add(column, value, Criteria::NOT_EQUAL);
其他比较操作符	
> , <	Criteria::GREATER_THAN, Criteria::LESS_THAN
>=, <=	Criteria::GREATER_EQUAL, Criteria::LESS_EQUAL
IS NULL, IS NOT NULL	Criteria::ISNULL, Criteria::ISNOTNULL
LIKE, ILIKE	Criteria::LIKE, Criteria::ILIKE
IN, NOT IN	Criteria::IN, Criteria::NOT_IN
其他 SQL 关键字	
ORDER BY column ASC	->addAscendingOrderByColumn(column);
ORDER BY column DESC	->addDescendingOrderByColumn(column);
LIMIT limit	->setLimit(limit)
OFFSET offset	->setOffset(offset)
FROM table1, table2 WHERE table1.col1 = table2.col2	->addJoin(col1, col2)
FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2	->addJoin(col1, col2, Criteria::LEFT_JOIN)
FROM table1 RIGHT JOIN table2 ON table1.col1 = table2.col2	->addJoin(col1, col2, Criteria::RIGHT_JOIN)

TIP 最好的理解生成类的方法的途径就是查看位于 lib/model/om/目录下的 Base 文件。他们的方法名是很直接的，但如果你需要了解更多的信息，可以在 config/propel.ini 文件中设置 propel.builder.addComments 参数为 true，然后重建模型。

例 8-14 是多个条件的 Criteria 的另一个实例。它搜索所有 Steve 的包含 "enjoy"这个词的评论，并按照日期排列。

例 8-14 - 通过 Criteria 用 doSelect() 获得记录的另一个示例--有条件的 Criteria

```
$c = new Criteria();
```

```

$c->add(CommentPeer::AUTHOR, 'Steve');
$c->addJoin(CommentPeer::ARTICLE_ID, ArticlePeer::ID);
$c->add(ArticlePeer::CONTENT, '%enjoy%', Criteria::LIKE);
$c->addAscendingOrderByColumn(CommentPeer::CREATED_AT);
$comments = CommentPeer::doSelect($c);

// 等同于下面 SQL 语句执行的结果
SELECT blog_comment.ID, blog_comment.ARTICLE_ID, blog_comment.AUTHOR,
       blog_comment.CONTENT, blog_comment.CREATED_AT
FROM   blog_comment, blog_article
WHERE  blog_comment.AUTHOR = 'Steve'
       AND blog_article.CONTENT LIKE '%enjoy%'
       AND blog_comment.ARTICLE_ID = blog_article.ID
ORDER BY blog_comment.CREATED_AT ASC

```

就如同 SQL 是一个很简单的语言却可让你建立非常复杂的查询一样，Criteria 对象也可以处理任意复杂的问题。但是很多开发者会在转换为面向对象逻辑之前先考虑 SQL，Criteria 对象开始也许会有些难以掌握。最好的理解方法是从例子和简单的应用程序中学习。例如 symfony 项目网站，到处都有 Criteria 建立的演示会从各种方面来启发你。

除了 doSelect() 方法外，每一个 peer 类都有一个 doCount() 方法用来简单的统计记录的数量，用作给 criteria 传递的参数并返回数字作为结果。因为没有返回对象，所以例子中没有化合（hydrating）过程，因此 doCount() 方法比 doSelect() 方法快。

Peer 类也提供了 doDelete(), doInsert() 和 doUpdate() 方法，用来作为 Criteria 的参数。这些方法允许你对数据库执行 DELETE, INSERT 和 UPDATE 查询语句。可以在你的模型中查看生成的 peer 类来获得更多的关于 propel 方法的细节。

最后，如果你只想返回第一个对象，调用 doSelectOne() 替换掉 doSelect()。这会让 Criteria 只返回一个结果，好处就是这个方法返回的是一个对象而不是一个对象数组。

TIP 当一个 doSelect() 查询返回大量结果的时候，你也许只会看到他们中的部分结果。Symfony 提供了一个叫做 sfPropelPager 的翻页类，可以自动的处理结果的翻页。看 API 手册 <http://www.symfony-project.com/api/symfony.html> 来获得更多的信息和使用示例。

直接使用 SQL 查询语句

有时候，你不想得到对象而只是想得到由数据库执行得到的结果。例如，要获得所有文章最新的建立时间，取得到所有的文章然后遍历数组是没有意义的。

你会倾向于让数据库来处理并只返回结果，因为这会跳过对象化合（hydrating）过程。

另一方面，你不想为了管理数据库直接调用 PHP 命令，因为会失去使用数据库抽象层的优势。这意味着需要绕过 ORM(Propel)而不是数据库抽象层(Creole)。

你需要做如下步骤通过 Creole 查询数据库：

1. 获得数据库连接。
2. 建立查询字符串。
3. 建立一个声明。
4. 从声明执行结果中循环结果集。

如果这对你没帮助，例 8-15 中的代码也许会让你更清晰一些。

例 8-15 - 用 Creole 来自定义查询

```
$connection = Propel::getConnection();
$query = 'SELECT MAX(%s) AS max FROM %s';
$query = sprintf($query, ArticlePeer::CREATED_AT,
ArticlePeer::TABLE_NAME);
$statement = $connection->prepareStatement($query);
$resultset = $statement->executeQuery();
$resultset->next();
$max = $resultset->getInt('max');
```

就像 Propel 的选择功能，初次使用 Creole 查询的时候会觉得这需要一些技巧。再说一次，从已有应用程序的示例和指南会展示给你正确的使用方法。

CAUTION 如果你倾向绕过这个过程并直接访问数据库，你会面临失去安全性和 Creole 提供的抽象层。Creole 做了对数据库所有可用的转义和安全性的处理。尽管用 Creole 方法会花更长的时间，但它会强迫你使用好的方法，这就保证了应用程序的性能，可移植性和安全性。这对于包含从不信任的来源获得参数（比如 Internet 使用者）的查询来说特别有用。而直接访问数据库会给你带来 SQL 注入攻击的危险性。

使用特殊日期列

通常，当一个表有一个列叫做 created_at 时，它通常储存的是记录创建时的日期的时间戳格式。updated_at 列也一样，因此当每次更新记录时，它本身也会被更新为当前时间。

有个好消息是 `symfony` 会知道这些列的名字并会自动为你处理。 你不需要手动设置 `created_at` 和 `updated_at` 列；它会自动为你更新，如例 8-16 所示。 对于 `created_on` 和 `updated_on` 也一样。

例 8-16 - `created_at` 和 `updated_at` 列的数据会被自动处理

```
$comment = new Comment();
$comment->setAuthor('Steve');
$comment->save();

// 显示创建日期
echo $comment->getCreatedAt();
=> [date of the database INSERT operation]
```

此外，日期列的 `getter` 允许日期格式作为参数：

```
echo $comment->getCreatedAt('Y-m-d');
```

SIDEBAR 数据层重构

当开发一个 `symfony` 项目时，通常开始在动作中写逻辑代码。但在控制层中并没有数据库查询和模型处理。因此所有的数据逻辑都应该转移到模型层。当你需要在动作中的多个地方执行同一个请求时，想一下如何把相关代码转移到模型中。这会对你保持动作简洁和可读性有很好的帮助。

例如，假设需要一段代码在 `blog` 中获得指定标签的十个最流行的文章（传递请求参数）。这个代码在动作中，但在模型中。事实上，如果你需要在模板中列出来，这个动作应该看上去像这样：

```
public function executeShowPopularArticlesForTag()
{
    $tag = TagPeer::retrieveByName($this->getRequestParameter('tag'));
    $this->forward404Unless($tag);
    $this->articles = $tag->getPopularArticles(10);
}
```

这个动作使用 `request` 参数创建了一个 `Tag` 类的对象。因此所有查询数据库所需的代码都位于这个类的 `getPopularArticles()` 方法中。这使行为更易读，并且让模型代码更易在其他行为中重用。

重构的其中一个方法就是把代码移到一个更合适的位置。如果你经常这么做，你的代码会易于维护并被其他开发者理解。有一个不成文的规则是：动作的代码应该尽量低于 10 行 PHP 代码。

数据库连接

数据模型是和使用的数据库分离的，但是你还是要使用数据库。要发送请求给项目数据库的话至少要让 `symfony` 知道数据库名字，访问的代码和数据库的类型。这些连接设置应该放在位于 `config/` 目录下的 `databases.yml` 文件中。例 8-17 中的示例。

例 8-17 - 在 `myproject/config/databases.yml` 的数据库连接配置示例

```
prod:
propel:
param:
host:          mydataserver
username:      myusername
password:      xxxxxxxxxxxx

all:
propel:
class:         sfPropel Database
param:
    phptype:    mysql      # 数据库类型
hostspec:      localhost
database:      blog
username:      login
password:      passwd
port:          80
    encoding:   utf-8      # 创建表默认的 charset
    persistent: true      # 是否使用持久连接
```

连接设置是基于环境的。你可以给 `prod`、`dev` 和 `test` 环境或是任何应用程序中的环境截然不同的设置。这个配置也可以由每个应用程序通过设置应用程序相关的文件中不同的值而覆盖，例如在 `apps/myapp/config/databases.yml` 中。举个例子，你可以使用此方法来给前台和后台应用程序定义不同的安全策略，并定义几个数据库用户拥有不同的数据库权限来实施控制。

对于每一个环境，你能定义很多连接，每一个连接对应了一个同名的设计。在例 8-17 中，`propel` 连接对应了例 8-3 中的 `propel` 设计。

Creole 支持的数据库系统，也就是 `phptype` 参数允许的值是：

- `mysql`

- sql server
- pgsql
- sqlite
- oracle

hosts spec, database, username 和 password 是通常数据库连接设置需要的。他们能写成更短的数据库源名(DSN)。例 8-18 等同于例 8-17 中的 all:。

例 8-18 - 简单的数据库连接方法设置

```
all:
propel:
class:          sfPropel Database
param:
dsn:            mysql://login:passwd@localhost/blog
```

如果使用 SQLite 数据库， 必须在数据库文件中设置 hosts spec 参数。例如，如果你把 blog 数据库放在 data/blog.db 中， databases.yml 文件将看上去就像例 8-19 一样。

例 8-19 - SQLite 使用了文件路径作为 HOST 的数据库连接设置

```
all:
propel:
class:          sfPropel Database
param:
phptype:        sqlite
database:        %SF_DATA_DIR%/blog.db
```

扩展模型

symfony 生成的模型方法是很棒的但通常并不够用。当你实现你自己的业务逻辑时，你需要去扩展它，不论是增加新的方法或是覆盖现有方法。

增加新的方法

你可以在 lib/model/ 目录下生成的空模型类中增加新的方法。使用 \$this 来调用当前对象的方法， 并使用 self:: 来调用当前类的静态方法。 记住，自定义类继承 lib/model/om/ 目录下 Base 类的方法。

例如，对于在例 8-3 中生成的 Article 对象， 你能增加一个神奇的 __toString() 方法让 Article 类显示他自己的标题， 如例 8-20 所示。

例 8-20 - 在 lib/model/Article.php 中自定义模型

```
<?php
```

```
class Article extends BaseArticle
{
    public function __toString()
    {
        return $this->getTitle(); // getTitle()继承自 BaseArticle
    }
}
```

你也可以扩展 peer 类，例如，增加一个新方法来获得按照创建日期排序的所有文章，如例 8-21 所示。

例 8-21 - 在 lib/model/ArticlePeer.php 中自定义模型

```
<?php
```

```
class ArticlePeer extends BaseArticlePeer
{
    public static function getAllOrderedByDate()
    {
        $c = new Criteria();
        $c->addAscendingOrderByColumn(self::CREATED_AT);
        return self::doSelect($c);
    }
}
```

新方法和系统生成的方法使用方法一样， 如例 8-22 所示。

例 8-22 - 使用自定义的模型方法就如使用生成的方法一样

```
foreach (ArticlePeer::getAllOrderedByDate() as $article)
{
    echo $article; // 会调用 __toString()这个魔术方法
}
```

覆盖现有方法

如果在 Base 类中一些生成的方法和你的需求并不一致， 可以在自定义类中覆盖它们。只要确认你使用相同的方法签名（就是说相同数量的参数）。

例如，`$article->getComments()`方法返回一个没有排序的 `Comment` 对象的数组。如果你想让结果按照创建日期排序过并有最新的 `comment` 在前面，只要覆盖 `getComments()`方法，如例 8-23 所示。小心原始 `getComments()`方法（在 `lib/model/om/BaseArticle.php`）需要一个 `criteria` 值和一个连接值作为参数，所以你的函数也必须包含它们。

例 8-23 - 在 `lib/model/Article.php` 覆盖现有的模型方法

```
public function getComments($criteria = null, $con = null )
{
    // 在 PHP5 下，对象是引用传递的， 所以要避免修改原始的，
    // 你必须克隆它
    $criteria = clone $criteria;
    $criteria->addDescendingOrderByColumn(ArticlePeer::CREATED_AT);

    return parent::getComments($criteria, $con);
}
```

自定义方法最终调用一个基础父类， 这很好。 不过，你也可以完全绕过父类并返回你希望的值。

使用模型行为

一些模型修改是通用的可以重复使用的。例如，用来让模型对象排序的方法、优化锁定来防止并发对象之间的冲突，都是通用的扩展，可以在许多类中使用。

`symfony` 把这些扩展打包为行为。行为是外部的给模型类提供了额外的方法的类。模型类已经包含了钩子，`symfony` 知道如何用 `sfMixer`（参考 17 章获得更多的细节）扩展他们。

要在你的模型类中激活行为，你必须修改 `config/propel.ini` 文件中的一个设置：

```
propel.builder.AddBehaviors = true      // 默认值是 false
```

`symfony` 默认是没有绑定行为的，但是可以通过插件来安装它。当行为插件安装好之后，你能用一行代码来设置行为给一个类。例如，如果在你的应用程序中安装了 `sfPropelParanoidBehaviorPlugin`，你能通过这个行为增加下面这段在文章尾部来扩展文章类：

```
sfPropelBehavior::add('Article', array(
    'paranoid' => array('column' => 'deleted_at')
```

));

重建模型后，删除 `Article` 对象时，对象依旧会保留在数据库中，仅仅对 ORM 的查询是不可见的，除非你用 `sfPropelParanoidBehavior::disable()` 把行为禁用了。

在 `wiki` 查看 `symfony` 插件列表来搜索 behaviors(<http://www.symfony-project.com/trac/wiki/SymfonyPlugins#Propelbehaviorplugins>)。每个行为都有自己的文档和安装方法。

扩展设计语法

`schema.yml` 文件可以是简洁的，如例 8-3 所示。但是相关联的模型通常是复杂的。这就是为什么设计有一个扩展语法来处理几乎每一种情况。

属性

连接和表可以有特殊的属性，如例 8-24 所示。它们的定义在 `_attributes` 关键字下。

例 8-24 - 连接和表的属性

```
propel:
  _attributes: { noXsd: false, defaultIdMethod: none, package:
lib.model }
  blog_article:
    _attributes: { phpName: Article }
```

你也许想在代码生成前验证你的设计。要这样做的话，在连接中需要设置 `noXSD` 属性为 `false`。连接也支持 `defaultIdMethod` 属性。如果没有提供，会使用数据库的原生方法生成的 ID，例如，MySQL 的 `autoincrement` 或者 PostgreSQL 的 `sequences`。另外一个可能的值是 `none`。

`package` 属性有点像命名空间；他决定了生成的类的储存路径。默认是 `lib/model/`，但是你能通过在 `subpackage` 组织你的模型来改变它。例如，如果你不想在同一个目录下放置核心的业务类与数据库定义的类，可以用 `lib.model.business` 和 `lib.model.stats` 包来定义两个设计。

你已经看到过表属性 `phpName`，它用来设置生成的类名并映射到相应的表。

包含本地化内容的表（就是不同版本的内容在一张相关联的表中做国际化处理）也使用两个额外的属性（13 章有详细介绍），如例 8-25 所示。

例 8-25 - `i18n` 表属性

```
propel :  
  blog_article:  
    _attributes: { is118N: true, i18nTable: db_group_i18n }
```

SIDEBAR 处理多个设计

你可以在每一个应用程序中拥有多个设计。 `symfony` 会查看 `config/` 目录下文件名结尾是 `schema.yml` 或者 `schema.xml` 的文件。 如果你的应用程序有多个表，或者如果一些表不想分享同一个连接，你会觉得这个方法非常有用。

试想一下这两个设计：

```
// 在 config/business-schema.yml  
propel :  
  blog_article:  
    _attributes: { phpName: Article }  
id:  
title: varchar(50)
```

```
// 在 config/stats-schema.yml  
propel :  
  stats_hit:  
    _attributes: { phpName: Hit }  
id:  
resource: varchar(100)  
created_at:
```

这两个 schemas 共享了同一个连接(propel)， `Article` 和 `Hit` 类会生成在同一个 `lib/model/` 目录下。 每件事情处理的都如同写在同一个设计中一样。

你也能让不同的设计使用不同的连接（例如，定义在 `databases.yml` 中的 `propel` 和 `propel_bis`）并在子目录中组织生成的类：

```
// 在 config/business-schema.yml  
propel :  
  blog_article:  
    _attributes: { phpName: Article, package:  
lib.model.business }  
id:  
title: varchar(50)
```

```
// 在 config/stats-schema.yml
```

```

propel_bis:
  stats_hit:
    _attributes: { phpName: Hit, package.lib.model.stat }
id:
resource: varchar(100)
created_at:

```

许多应用程序使用多个设计。特别是，一些插件为了防止和你自己的类有冲突打包了他自己的设计（第 17 章有详细介绍）。

列详细资料

基础设计语法提供了两种选择：让 `symfony` 根据列名推算出列的特征（给一个空值）或者用于类型关键字定义一个类型。例 8-26 演示了这些选择。

例 8-26 - 基础列属性

```

propel:
  blog_article:
    id:                # 让 symfony 自己来处理
    title: varchar(50) # 自定义一个类型

```

但是你能对列定义更多。如果这么做了，你需要用一个数组来定义列属性，如例 8-27 所示。

例 8-27 - 复杂的列属性

```

propel:
  blog_article:
    id: { type: integer, required: true, primaryKey: true,
    autoIncrement: true }
    name: { type: varchar(50), default: foobar, index: true }
    group_id: { type: integer, foreignTable: db_group, foreignReference:
    id, onDelete: cascade }

```

列参数有以下几种：

- `type`: 列类型分 `boolean`, `tinyint`, `smallint`, `integer`, `bigint`, `double`, `float`, `real`, `decimal`, `char`, `varchar(size)`, `longvarchar`, `date`, `time`, `timestamp`, `bu_date`, `bu_timestamp`, `blob`, and `clob`.
- `required`: 布尔值。如果这个列是必须的话就设置为 `true`。
- `default`: 默认值。
- `primaryKey`: 布尔值。如果是主键就设为 `true`。
- `autoIncrement`: 布尔值。如果是 `integer` 并需要自动增长的话就设为 `true`。

- **sequence:** 数据库的序列名用来给需要 `autoIncrement` 列使用（例如，PostgreSQL 和 Oracle）。
- **index:** 布尔值。 如果想使用简单的索引时设置为 `true` 或是如果想要为这个列创建 `unique` 索引。
- **foreignTable:** 一个表名， 用来创建对其他表用的外键。
- **foreignReference:** 相关列的名字，如果是通过 `foreignTable` 定义的外键
- **onDelete:** 当相关表的字段被删除时候决定作什么操作。 当设为 `setNull`，则外键列会设置为 `null`。当设为 `cascade`，记录会被删除。如果数据库引擎不支持 `set` 行为，ORM 会模拟。 只有在有 `foreignTable` 和 `foreignReference` 时有意义。
- **isCulture:** 布尔值。 如果在本地化的内容表中有 `culture` 列，则设置为 `true`。

外键

就如可选择的 `foreignTable` 和 `foreignReference` 列属性， 你可以在表的 `_foreignKeys:` 关键字下增加外键。 例 8-28 的设计会在 `user_id` 列创建一个外键， 来匹配 `blog_user` 表的 `id` 字段。

例 8-28 - 外键的另一种语法

```
propel:
  blog_article:
    id:
    title:  varchar(50)
    user_id: { type: integer }
    _foreignKeys:
      -
    foreignTable: blog_user
    onDelete:    cascade
    references:
      - { local: user_id, foreign: id }
```

在有多重引用外键时另一种语法非常有用，他给外键一个名字，如例 8-29 所示。

例 8-29 - 多重引用外键时候的另一种语法

```
_foreignKeys:
  my_foreign_key:
foreignTable: db_user
onDelete:    cascade
references:
  - { local: user_id, foreign: id }
```



```
- { local: post_id, foreign: id }
```

索引

作为 `index` 列属性的另一个选择，你能在表的 `_indexes:` 下增加索引。 如果你想要定义唯一索引，你必须用 `_unique:` 替代。 例 8-30 展示了索引的另一个语法。

例 8-30 - 索引和唯一索引的另一种语法

```
propel:
  blog_article:
    id:
    title:          varchar(50)
    created_at:
    _indexes:
      my_index:      [title, user_id]
    _unique:
      my_other_index: [created_at]
```

当为多个列建立索引的时候另一种语法是非常有用的。

空列

当遇到一个空值列的时候，`symfony` 会猜测并增加一个值。 例 8-31 有关于增加一个空列的详细信息。

例 8-31 - 从列名推演出列详细资料

```
// 叫做 id 的空列被当作主键
id: { type: integer, required: true, primaryKey: true,
autoIncrement: true }

// 叫做 XXX_id 的空列被当作外键
foobar_id: { type: integer, foreignTable: db_foobar,
foreignReference: id }

// 叫做 created_at, updated_at, created_on 和 updated_on
// 被看作日期并自动设置为 timestamp 类型
created_at: { type: timestamp }
updated_at: { type: timestamp }
```

对于外键，`symfony` 会查找和列名开头相同 `phpName` 的表名， 如果找到了，他就会认为这就是 `foreignTable`。

I18n 表

symfony 支持相关的表的内容国际化。这就意味着当你要把标题国际化时，他会储存在两个分开的表中：一个放在常规的列中，另一个放在国际化的列中。

在 schema.yml 文件中，当你把表命名为 foobar_i18n 就意味着是 I18n 表了。例如，在例 8-32 示例中的数据库设计会根据列和表属性自动完成让国际化内容机制工作。就内部机制来说，symfony 会把它当作例 8-33 所写的一样。第 13 章会告诉你更多的关于 i18n 的信息。

例 8-32 - 隐含的 i18n 机制

```
propel :
  db_group:
    id:
      created_at:

    db_group_i18n:
      name:          varchar(50)
```

例 8-33 - 详述的 i18n 机制

```
propel :
  db_group:
    _attributes: { isI18N: true, i18nTable: db_group_i18n }
    id:
      created_at:

    db_group_i18n:
      id:          { type: integer, required: true, primaryKey:
true, foreignTable: db_group, foreignReference: id, onDelete: cascade }
      culture:    { isCulture: true, type: varchar(7), required:
true, primaryKey: true }
      name:       varchar(50)
```

超越 schema.yml : schema.xml

事实上，schema.yml 格式是 symfony 内部格式。当你调用一个 propel -命令行时，symfony 实际上会把这个文件转换为 generated-schema.xml 文件，这是 Propel 实际处理的模型的文件类型。

schema.xml 文件包含了和它 YAML 版本相同的信息。例如，例 8-34 就是例 8-3 转换后的 XML 文件。

例 8-34 - 对应例 8-3 的 schema.xml 示例

```
<?xml version="1.0" encoding="UTF-8"?>
<database name="propel" defaultIdMethod="native" noXsd="true"
package="lib.model">
<table name="blog_article" phpName="Article">
<column name="id" type="integer" required="true"
primaryKey="true"autoIncrement="true" />
<column name="title" type="varchar" size="255" />
<column name="content" type="longvarchar" />
<column name="created_at" type="timestamp" />
</table>
<table name="blog_comment" phpName="Comment">
<column name="id" type="integer" required="true"
primaryKey="true"autoIncrement="true" />
<column name="article_id" type="integer" />
<foreign-key foreignTable="blog_article">
<reference local="article_id" foreign="id"/>
</foreign-key>
<column name="author" type="varchar" size="255" />
<column name="content" type="longvarchar" />
<column name="created_at" type="timestamp" />
</table>
</database>
```

schema.xml 格式描述可以在文档中找到，也能在 Propel 项目网站的"Getting Started"章节

(http://propel.phpdb.org/docs/user_guide/chapters/appendices/Appendix B-SchemaReference.html) 找到。

YAML 格式设计用来让设计的读和写保持简洁，但是代价是复杂的设计不能在 schema.yml 文件中描述出来。另一方面，XML 格式允许所有的 schema 描述，不论它多复杂，不论他是否包含数据库提供商特别的设置，表继承关系等等。

symfony 实际上用的是 XML 格式的 schema。所以如果你的 schema 对于 YAML 语法来说太复杂，如果已有一个 XML 的设计文件，或者你已经对 Propel XML 语法很熟悉了，你可以不使用 symfony YAML 语法。用你在项目的 config/目录下的 schema.xml，来建立模型，开始所有的事情。

SIDEBAR symfony 中的 Propel

本章所述所有的细节都不是针对 symfony 的，而是针对 Propel 的。Propel 是 symfony 所推荐的对象/关系抽象层，但是你可以选择其他的。无论如何，symfony 和 Propel 结合的更紧密，因为如下原因：

所有的对象数据模型类和 `Criteria` 类都是自动载入的类。当你使用它们的时候，`symfony` 会包含相应的文件并且你不需要手动的去增加相关的文件声明。在 `symfony` 中，`Propel` 不需要执行或者初始化。当一个对象使用 `Propel` 的时候，框架会自己初始化它。一些 `symfony` 辅助函数使用 `Propel` 对象作为他的参数来完成高级任务（例如翻页或者过滤）。使用 `Propel` 对象可以快速建模和生成应用程序后台（第 14 章有详细介绍）。通过 `schema.yml` 文件可以让设计写起来更快。

并且，`Propel` 与使用的数据库无关，`symfony` 也是如此。

不要重复建立模型

使用 ORM 的代价是你必须定义数据结构两次：一次为了数据库，一次为了对象模型。幸运的是，`symfony` 提供了从其中一个生成另一个的命令行工具，这样就可以避免重复工作。

基于已存在的设计建立一个 SQL 数据库结构

如果从写 `schema.yml` 文件开始写应用程序的话，`symfony` 能生成一个 SQL 查询并直接从 YAML 数据模型生成数据表。要使用这些查询语句，在项目根目录，键入：

```
>symfony propel-build-sql
```

`myproject/data/sql/`目录下会创建一个 `lib.model.schema.sql` 文件。注意生成的 SQL 代码会针对在 `propel.ini` 文件中定义的 `phptype` 参数来对数据库系统进行优化。

你可以使用 `schema.sql` 文件直接建立表。例如，在 MySQL 中，输入：

```
>mysqladmin -u root -p create blog
>mysql -u root -p blog < data/sql/lib.model.schema.sql
```

生成的 SQL 也对在其他环境中建立数据库有用，或者改成其他的 DBMS。如果连接设置在 `propel.ini` 中定义正确，你甚至能使用 `symfony propel-insert-sql` 命令去自动完成。

TIP 命令行也提供了一个基于文本文件来填充数据库和数据的机制。看第 16 章来获得更多的关于 `propel-load-data` 和 YAML 格式文件的信息。

从已有数据库建立 YAML 数据模型

symfony 能使用 Creole 数据库访问层来访问数据库从而生成 schema.yml 文件，这要归功于内省 (introspection) 机制 (获取数据库表结构的功能)。当你做反向工程时候特别有用，或者如果你期待在有对象模型前先做基于数据库的工作。

为了这个目的，你需要确定项目的 propel.ini 文件指向了正确的数据库和包含了所有的连接设置，然后调用 propel-build-schema 命令：

```
>symfony propel-build-schema
```

从数据库结构生成的全新的 schema.yml 文件会放在 config/目录下。你能基于这个设计建立模型。

生成设计命令行十分强大，能在你的 schema 中增加一系列的数据库相关的信息。如 YAML 格式不能处理所有的数据库信息，你能生成一个 XML 设计来代替。要生成 XML 文件只要在 build-schema 上加一个 xml 参数就行了：

```
>symfony propel-build-schema xml
```

你会生成一个 schema.xml 文件用来替代生成 schema.yml 文件，这和 Propel 完全兼容，包含了所有的 vendor 信息。但是注意，生成的 XML 设计会有些冗长和难以阅读。

SIDEBAR propel.ini 配置

propel-build-sql 和 propel-build-schema 任务不使用定义于 databases.yml 文件中的连接设置。替代的是，这些任务使用了其他文件中的连接设置，调用存于项目 config/目录下的 propel.ini：

```
propel.database.createUrl =  
mysql://login:passwd@localhost  
propel.database.url      =  
mysql://login:passwd@localhost/blog
```

这个文件包含了其他的设置用来配置 Propel 生成器来生成和 symfony 兼容的模型类。多数设置是内部的，除了少数一些外，其他大多用户都不需要去关心：

```
// 在 symfony 中，基类是自动载入的  
// 设置这个为 true 使用 include_once  
// （对性能有一些负面的影响）  
propel.builder.addIncludes = false  
  
// 生成的类默认是没有注释的
```

```
// 设置这个为 true 可以给基类增加注释
// （对性能有一些负面的影响）
propel.builder.addComments = false

// 默认不处理行为（Behaviors）
// 设置为 true 可以处理他们
propel.builder.AddBehaviors = false
```

修改 `propel.ini` 设置后，别忘了重建模型让所有的修改生效。

总结

symfony 使用 Propel 作为 ORM，Creole 作为数据库抽象层。这意味着你需要首先在生成对象模型类前用数据库设计的 YAML 语法来描述数据的关系。然后，在运行时，使用类的方法和 peer 类获得关于记录或者一组记录的信息。你能覆盖他们，用自定义类中增加方法来非常容易的扩展模型。连接设置定义在 `databases.yml` 文件中，用来支持多于一个连接。命令行包含了特别的任务来避免重复数据定义。

模型层是 symfony 框架中最复杂的。为什么这么说的一个理由就是数据处理是一个复杂的问题。对于网站相关的安全性是至关重要的并不能被忽略的。另一个理由就是 symfony 更适合企业级的中等到大型的数据库应用。在这些应用程序中，symfony 模型层提供的自动机制确实省了很多时间，值得去研究和学习。

所以，不要犹豫了，花一些时间来测试这些模型对象和方法让自己完全了解他们。你的应用程序的可靠性和可扩展性能让你觉得学习的花费很值得。

第 9 章 - 链接和路由系统

链接和 URL 在 web 应用程序框架中需要特别关注。因为这是应用程序的唯一进入点（前端控制器）；在模板中使用辅助函数可以让 URL 工作方式和他们的外观完全分离。这就叫做路由。路由可以使建立的应用程序对用户更友好更安全。本章会讲述在 symfony 应用程序里处理 URL 所需要了解的内容：

- 什么是路由系统及它是如何工作的
- 如何在模板中使用链接辅助函数让路由处理外部 URL
- 如何配置路由规则来改变 URL 的组成

你还会找到一些控制路由的性能和增加 最后的修饰小技巧。

什么是路由？

路由是一种为了显示更友好的 URL 而重写 URL 的机制。但要去了解他为何如此重要的之前，先需要花几分钟来理解什么是 URL。

URL 是服务器指令

URL 包含了从浏览器到服务器用来请求执行一个用户需求行为的信息。例如，一个传统的 URL 包含了脚本文件路径和一些请求所必要的参数，就像下面这个例子：

http://www.example.com/web/controller/article.php?id=123456&format_code=6532

这个 URL 显示了关于应用程序结构和数据库的信息。开发者通常在界面里隐藏了应用程序的架构（例如，他们设置页标题“个人资料页”而不是“QZ7.65”）。应用程序内部重要信息的泄露是违背 URL 本意的，并会带来严重弊端：

- 技术数据出现在 URL 中会造成潜在的安全隐患。在前面的例子中，如果一个不怀好意的用户更改了 id 参数的值会发生什么呢？这是否意味着应用程序可以提供一个直接访问数据库的方法？或者要是用户为了好玩而尝试其他脚本名字，例如 admin.php？总而言之，原始的 URL 会让黑客轻易的破坏你的应用程序，这样无法进行安全控管。
- 不管在哪里出现费解的 URL 都会让人不安，他们会弱化内容的效果。如今，URL 不止出现在地址栏里。当用户把鼠标悬停在链接上，或者在搜索结果中也会出现。当用户要查找信息时，他们希望得到的反馈是易于理解的，而不是如图 9-1 所示的混乱的 URL。

图 9-1 - URL 在很多地方显示，例如在搜索结果中

Microsoft Office Clip Art and Media Home Page

Microsoft Office Clip Art and Media - Over 140000 clip art graphics, animations, photos, and sounds for use in **Microsoft** Office products.

office.microsoft.com/clipart/default.aspx?lc=en-us - 56k - 30 Aug 2006 -

Cached - Similar pages

- 如果一条 URL 必须被改变（例如，如果脚本名或者它的一个参数需要改变），每一个链接到它的 URL 都要相应的更改。这就意味着修改控制结构是非常累人的和昂贵的，这不符合理想的敏捷开发。

如果 symfony 不使用前端控制器概念也许会更糟，更确切地说，如果应用程序包含了许多可以从 Internet 访问的脚本，在许多目录中，就如这些：

<http://www.example.com/web/gallery/album.php?name=my%20holidays>

<http://www.example.com/web/weblog/public/post/list.php>

<http://www.example.com/web/general/content/page.php?name=about%20us>

在这种情况下，开发者需要用文件结构来匹配 URL 结构，结果就会导致当结构改变的时候，维护将会非常困难。

URL 是界面的一部分

路由概念还有一层含义就是把 URL 看作界面的一部分。应用程序可以通过格式化的 URL 带给用户信息，用户也可以使用 URL 来访问应用程序的资源。

symfony 应用程序中这是可能实现的，因为展示给最终用户的 URL 与执行请求需要的服务器指令完全无关。相反，它涉及到资源的需求，并且他们可以自由的格式化。例如，symfony 可以理解下面的 URL 并使其显示和本章第一个 URL 同样的页面：

<http://www.example.com/articles/finance/2006/activity-breakdown.html>

这样有很大的优势：

- URL 实际上没有意义，但他们可以帮助用户来决定连接后的页面是否有他们期待的内容。一个连接能包含关于它所返回资源的额外细节。这对搜索引擎结果特别有用。此外，URL 有时候不会随着页面标题出现（试想在 email 里复制 URL），在这种情况下，你必须让它有一些含义。图 9-2 就是一个有意义的 URL 的例子。

图 9-2 URL 能传达关于页面的额外信息，如发布日期

symfony PHP5 framework » **AJAX pagination** made simple

The **AJAX pagination** demo uses two very simple actions, both passing a pager to their template: `class pagerActions extends sfActions { public function ...`

www.symfony-project.com/weblog/2006/07/17/ajax-pagination-made-simple.html - 12k -

[Cached](#) - [Similar pages](#)

- 在纸上的 URL 会更容易输入和记住。如果你的公司网站在你的名片上写了 <http://www.example.com/controller/web/index.jsp?id=ERD4>，这也许不会带来很多的访问者。
- URL 本身可以成为一个命令行工具去执行命令或者获得信息。应用程序给高级用户提供了这种可以更快捷的可能性。

// 结果列表：新增一个新的 tag 来限制结果列表

<http://del.icio.us/tag/symfony+ajax>

// 用户信息页：改变用户名来获得其他用户的信息

<http://www.asket.com/user/francois>

- 只要用一个简单的修改就能改变 URL 格式和动作名/参数。这就意味着你可以先开发程序，然后再确定 URL 格式，而不会把应用程序搞得一团糟。
- 甚至在你重组应用程序内部时，对外的 URL 依旧保持原样。这就可以使动态页面 URL 可以被收藏了。
- 搜索引擎在索引网站时倾向于忽略动态页面（结尾是 .php, .asp 之类）。所以格式化后的 URL 可以让搜索引擎认为是静态的内容，尽管实际上是动态的页面，因此就能更好的索引你的应用程序页面。
- 这会更安全。用户不能通过测试 URL 来浏览网页根文件结构，任何不能识别的 URL 会转向到一个开发者指定的页面，请求呼叫实际的脚本和它的参数都是隐藏的。

用户访问的 URL 和实际的脚本名和请求的参数都是由路由系统根据在配置文件中定义的规则来转义的。

NOTE 网页资源文件怎么办？幸运的是，网页资源文件的 URL（图片、样式表和 JavaScript）在浏览中不会出现很多，所以不需要对它们设置路由。在 symfony 中，所有的网页资源文件都位于 web/ 目录下，他们的 URL 对应了他们在文件系统中的位置。然后，你能使用网页资源文件辅助函数来生成 URL 去控制动态网页资源（通过动作）。例如，要显示一个动态生成的图片，使用 `image_tag(url_for('captcha/image?key=' . $key))`。

它是如何工作的

symfony 把外部 URL 和内部 URL 分离了。他们之间的联系是由路由系统定义的。为了简化处理，symfony 为内部 URL 使用了和通用 URL 相似的语法。例 9-1 是一个示例：

例 9-1 - 外部 URL 和内部 URL

// 内部 URL 语法

```
<module>/<action>[?param1=value1][&param2=value2][&param3=value3]...
```

// 内部 URL 示例，永远不会让最终用户看到

```
article/permalink?year=2006&subject=finance&title=activity-breakdown
```

// 外部 URL 示例，显示给最终用户看的

```
http://www.example.com/articles/finance/2006/activity-breakdown.html
```

路由系统使用了一个特殊的配置文件 `routing.yml` 来定义路由规则。看一下例 9-2 所示的规则，它定义了用来把显示内容转换为 `articles/*/*/*` 的模式。

例 9-2 一个路由规则示例

```
article_by_title:
```

```
  url:    articles/:subject/:year/:title.html
```

```
  param:  { module: article, action: permalink }
```

每一个发送给 `symfony` 应用程序的请求首先会被路由系统分析（这很方便，因为每一个请求都由一个前端控制器来处理）。路由系统在请求 URL 和路由系统定义的规则模式中寻找匹配规则。如果匹配，通配符的名字变成请求的参数，并被合并为 `param:` 定义的关键字。例 9-3 显示了它是如何工作的。

例 9-3 - 路由系统解释输入的请求 URL

// 用户输入（或点击）这个外部 URL

```
http://www.example.com/articles/finance/2006/activity-breakdown.html
```

// 前端控制器确认它和 `article_by_title` 规则匹配

// 路由系统建立了以下请求参数

```
'module' => 'article'
```

```
'action' => 'permalink'
```

```
'subject' => 'finance'
```

```
'year'    => '2006'
```

```
'title'   => 'activity-breakdown'
```

TIP 外部 URL 的 `.html` 扩展名只是一个简单的装饰，会被路由系统忽略掉。他只是让动态页面看上去和静态页面一样。你会在本章稍后的“路由配置”部分看到如何激活这个扩展。

接下来请求会传递给 `article` 模块的 `permalink` 动作，它用所有请求参数中的请求信息来决定显示哪个文章。

但是这个机制也能反向工作。为了让应用程序在自己的链接中显示外部 URL，你必须提供给路由系统足够的数据让它确定使用哪个规则。你可以通过一个特殊的辅助函数而不能直接用标签来写超链接--这会完全绕过路由系统，如例 9-4 所示。

例 9-4 路由系统格式化模板中对外的 URL

```
// 辅助函数 url_for() 把内部 URL 转换为外部 URL
<a href="<?php echo
url_for('article/permalink?subject=finance&year=2006&title=activity-
breakdown') ?>">click here</a>

// 辅助函数确认 URL 匹配 article_by_title 规则
// 路由系统由此建立了一个外部 URL
=> <a href="http://www.example.com/articles/finance/2006/activity-
breakdown.html">click here</a>

// 辅助函数 link_to() 直接输出一个超链接
// 并避免 PHP 和 HTML 的混淆
<?php echo link_to(
    'click here',
    'article/permalink?subject=finance&year=2006&title=activity-
breakdown'
) ?>

// 内部处理的时候，link_to() 会调用 url_for()，所以结果是一样的
=> <a href="http://www.example.com/articles/finance/2006/activity-
breakdown.html">click here</a>
```

因此，路由是一个双向机制，他只会在你使用 link_to 辅助函数来格式化你的链接时工作。

URL 重写

有一个问题必须在更深入了解路由系统之前澄清一下。在前面部分的示例，并没有在内部 URL 中提到前端控制器（index.php 或 myapp_dev.php）。前端控制器决定环境，而不是应用程序的元素。所以，所有的链接必须是独立于环境之外的，同时前端控制器名永远不应该出现在内部 URL 中。

在生成的 URL 示例中也没有脚本名出现。这是因为生成的 URL 在默认生产环境中不包含任何脚本名。在 settings.yml 文件中的 no_script_name 参数精确的控制了前台控制器名是否显示在生成的 URL 中。如例 9-5 设置它为 off 状态，则每一个通过链接辅助函数输出的 URL 都不会有前端控制器名字。

例 9-5 - 是否在 URL 中显示前端控制器名字，设置在 apps/myapp/settings.yml

```
prod:
  .settings
    no_script_name: off
```

现在，生成的 URL 看上去像这样：

<http://www.example.com/index.php/articles/finance/2006/activity-breakdown.html>

除了生产环境外的其他所有环境中，no_script_name 参数默认是设置为 off 的。所以当你在开发环境中浏览应用程序的时候，URL 中总是会有前端控制器名字。

http://www.example.com/myapp_dev.php/articles/finance/2006/activity-breakdown.html

在生产环境中，no_script_name 设置为 on，所以 URL 只显示了路由信息，这样看上去更友好。并且也没有技术信息显示。

<http://www.example.com/articles/finance/2006/activity-breakdown.html>

但是应用程序事如何知道去调用哪个前端控制器？这就是 URL 重写要做的事。网页服务器可以配置为如果 URL 中什么都没有的时候去调用一个指定的脚本。

在 Apache 中，当 mod_rewrite 扩展激活的时候是可以这么做的。每一个 symfony 项目都有一个 .htaccess 文件用来配置服务器 web 目录的 mod_rewrite 设置。默认的文件内容如例 9-6 所示。

例 9-6 在 myproject/web/.htaccess 中的 Apache 默认重写规则

```
<IfModule mod_rewrite.c>
  RewriteEngine On

  # we skip all files with .something
  RewriteCond %{REQUEST_URI} \\.+$
  RewriteCond %{REQUEST_URI} !\.html$
  RewriteRule .* - [L]

  # we check if the .html version is here (caching)
  RewriteRule ^$ index.html [QSA]
  RewriteRule ^([^.]+)$ $1.html [QSA]
  RewriteCond %{REQUEST_FILENAME} !-f
```

```
# no, so we redirect to our front web controller
RewriteRule ^(.*)$ index.php [QSA,L]
</IfModule>
```

网页服务器检查收到的 URL 形态。如果 URL 不包含后缀或者如果没有此页面的缓存（第 12 章讲述了缓存），就交由 `index.php` 处理。

无论如何，`symfony` 项目的 `web/` 目录是让项目中所有应用程序共享的。这就意味在 `web` 目录中通常有不止一个前端控制器。例如，项目有前台和后台应用程序，`dev` 和 `prod` 环境，这样在 `web` 目录下就有四个前端控制器：

```
index.php           // 生产环境前台
frontend_dev.php    // 开发环境前台
backend.php          // 生产环境后台
backend_dev.php      // 开发环境后台
```

`mod_rewrite` 设置只能指定一个默认的脚本名。如果在全部应用程序和环境中设置 `no_script_name` 为 `on`，所有的 URL 都会解析给在 `prod` 环境中的 `frontend` 应用程序。这就是为什么在一个项目中 URL 重写只能针对一个环境中的一个应用程序。

TIP 还有一种方法可以允许多个应用程序没有脚本名。那就是在网页根目录下建立子目录，并把前端控制器放在里面。然后相应的更改 `SF_ROOT_DIR` 常量，并为每个需要的应用程序建立 `.htaccess` URL 重写配置。

链接辅助函数

因为路由系统的原因，所以在你的模板中要使用链接辅助函数替换掉 `<a>` 标签。别为这个烦恼，这是一个让你的应用程序保持干净，并易于维护的机会。除此之外，链接辅助函数提供了一些你不应该忽视的非常有用的快捷方式。

超链接，按钮和表单

你已经知道 `link_to()` 辅助函数。它输出一个 XHTML 兼容的超链接，它有 2 个参数：可以点击的元素和所指向的资源内部 URL。如果你想用一个按钮来替代超链接，应该使用 `button_to()` 辅助函数。表单也有一个辅助函数来管理 `action` 属性的值。你会在下一段学习表单。例 9-7 展示了一些链接辅助函数的例子。

例 9-7 - `<a>`，`<input>` 和 `<form>` 标签的链接辅助函数

```
// 字符串的超链接
```

```
<?php echo link_to('my article',  
'article/read?title=Finance_in_France') ?>  
=> <a href="/routed/url/to/Finance_in_France">my article</a>
```

// 图片的超链接

```
<?php echo link_to(image_tag('read.gif'),  
'article/read?title=Finance_in_France') ?>  
=> <a href="/routed/url/to/Finance_in_France"></a>
```

// 按钮标签

```
<?php echo button_to('my article',  
'article/read?title=Finance_in_France') ?>  
=> <input value="my article"  
type="button"onclick="document.location.href='/routed/url/to/Finance_  
in_France';" />
```

// 表单标签

```
<?php echo form_tag('article/read?title=Finance_in_France') ?>  
=> <form method="post" action="/routed/url/to/Finance_in_France" />
```

链接辅助函数能接受内部 URL 和绝对 URL(http://开始, 被路由系统忽略)还有定位符。注意在真正的应用程序中, 内部 URL 是由动态参数建立的。例 9-8 是这些情况的例子。

例 9-8 - 链接辅助函数接受的 URL

// 内部 URL

```
<?php echo link_to('my article',  
'article/read?title=Finance_in_France') ?>  
=> <a href="/routed/url/to/Finance_in_France">my article</a>
```

// 带有动态参数的内部 URL

```
<?php echo link_to('my article', 'article/read?title='.$article-  
>getTitle()) ?>
```

// 带有定位符的内部 URL

```
<?php echo link_to('my article',  
'article/read?title=Finance_in_France#foo') ?>  
=> <a href="/routed/url/to/Finance_in_France#foo">my article</a>
```

// 绝对 URL

```
<?php echo link_to('my article',  
'http://www.example.com/foobar.html') ?>
```

```
=> <a href="http://www.example.com/foobar.html">my article</a>
```

链接辅助函数选项

如第 7 章所述，辅助函数能接受额外的选项参数，甚至可以是一个数组或者一个字符串。这对链接辅助函数也一样，如例 9-9 所示。

例 9-9 - 链接辅助函数接受额外的选项

```
// 数组作为额外选项
<?php echo link_to('my article',
' article/read?title=Finance_in_France', array(
    'class' => 'foobar',
    'target' => '_blank'
)) ?>

// 字符串作为额外选项（同样结果）
<?php echo link_to('my article',
' article/read?title=Finance_in_France', 'class=foobar
target=_blank') ?>
=> <a href="/routed/url/to/Finance_in_France" class="foobar"
target="_blank">my article</a>
```

也可以给链接辅助函数增加一个 symfony 特有的选项：confirm 和 popup。第一个会使链接在被点击时弹出一个 JavaScript 确认对话框，第二个会让连接在新窗口中打开，如例 9-10 所示。

例 9-10 - 链接辅助函数的 'confirm' 和 'popup' 选项

```
<?php echo link_to('delete item', 'item/delete?id=123', 'confirm=Are
you sure?') ?>
=> <a onclick="return confirm('Are you sure?');"
href="/routed/url/to/delete/123.html">add to cart</a>

<?php echo link_to('add to cart', 'shoppingCart/add?id=100',
'popup=true') ?>
=> <a onclick="window.open(this.href);return false;"
href="/fo_dev.php/shoppingCart/add/id/100.html">add to
cart</a>

<?php echo link_to('add to cart', 'shoppingCart/add?id=100', array(
    'popup' => array('Window title',
'width=310,height=400,left=320,top=0'))
```

```

)) ?>
=> <a onclick="window.open(this.href, 'Window
title', 'width=310,height=400,left=320,top=0');return false;"
href="/fo_dev.php/shoppingCart/add/id/100.html">add to
cart</a>

```

可以一起使用这些选项。

伪装的 GET 和 POST 选项

有时候网页开发者使用一个 GET 请求来执行一个 POST 操作。例如，看一下下面的 URL：

http://www.example.com/index.php/shopping_cart/add/id/100

这个请求会改变应用程序中的数据，他会在购物车对象中增加一个项目并保存在用户会话或者数据库中。这个 URL 可以被收藏，缓存，或者被搜索引擎索引。想一下使用这个技术会对数据库产生作用的负面影响。实际上这个请求应被当作 POST 来处理，因为搜索引擎机器人不会在索引的时候执行一个 POST 的请求。

symfony 提供了一个方法用来转换 `link_to()` 或者 `button_to()` 辅助函数的调用为一个实际的 POST。只要增加 `post=true` 选项，如例 9-11 所示。

例 9-11 - 让一个链接调用一个 POST 请求

```

<?php echo link_to('go to shopping cart', 'shoppingCart/add?id=100',
'post=true') ?>
=> <a onclick="f = document.createElement('form');
document.body.appendChild(f);
f.method = 'POST'; f.action = this.href;
f.submit();return false;"
href="/shoppingCart/add/id/100.html">go to shopping cart</a>

```

这个 `<a>` 标签有一个 `href` 属性同时浏览器没有 Javascript 支持，如搜索引擎机器人，会根据这个链接执行一个默认的 GET 动作。因此你必须通过增加一些限制让你的动作只会对 POST 方法有响应，如接下来所示：

```
$this->forward404If($request->getMethod() != sfRequest::POST);
```

请确定不要在表单里使用这个选项，这是因为它会生成自己的 `<form>` 标签。

把会修改数据的链接标记成 POST 是一个好习惯。

强制设置请求参数为 GET 变量

根据你的路由规则，变量通过 `link_to()` 转换为参数，同时被模式化。如果在 `routing.yml` 文件中无法找到能匹配内部 URL 的规则时，默认转换规则会把 `module/action?key=value` 转换为 `/module/action/key/value`，如例 9-12 所示。

例 9-12 - 默认的路由规则

```
<?php echo link_to('my article',  
'article/read?title=Finance_in_France') ?>  
=> <a href="/article/read/title/Finance_in_France">my article</a>
```

如果你确实需要保留 GET 语法（通过 `?key=val` 方式传递参数的话）你应该把参数放在 URL 参数外，也就是 `query_string` 选项中。所有的链接辅助函数都支持这个选项，就如例 9-13 所示。

例 9-13 - 用 `query_string` 选项强制指定 GET 变量

```
<?php echo link_to('my article',  
'article/read?title=Finance_in_France', array(  
    'query_string' => 'title=Finance_in_France'  
)) ?>  
=> <a href="/article/read?title=Finance_in_France">my article</a>
```

一个包含表现为 GET 变量的请求参数的 URL 可以被客户端的脚本解释，在服务端可以通过 `$_GET` 和 `$_REQUEST` 变量访问。

SIDEBAR 网页资源文件辅助函数 (Asset helpers)

第 7 章介绍了网页资源文件辅助函数 `image_tag()`，`stylesheet_tag()`，和 `javascript_include_tag()`，分别用来让你在回应中包含图片，样式表，或者 JavaScript 文件。这些网页资源文件路径并不包含在路由系统中，因为他们链接的资源实际上都在公共网页目录中。

你不用去介意网页资源文件的文件扩展名。`symfony` 会在调用图片，JavaScript，或者样式表辅助函数时自动增加 `.png`，`.js`，或 `.css` 后缀。同样的，`symfony` 会自动在 `web/images/`，`web/js/`，和 `web/css/` 目录中自动搜索网页资源文件。当然，如果你想要包含一个特殊的文件格式或者位于一个特殊位置的文件，只要使用文件全名或者文件完全路径作为变量即可。如果你的媒体文件有一个清楚名字的话，不要去管 `alt` 属性，因为 `symfony` 会为你处理的。

```
<?php echo image_tag('test') ?>
```

```
<?php echo image_tag('test.gif') ?>
<?php echo image_tag('/my_images/test.gif') ?>
```

```
=> <img href="/images/test.png" alt="Test" />
    <img href="/images/test.gif" alt="Test" />
    <img href="/my_images/test.gif" alt="Test" />
```

用 `size` 属性来固定图像的大小。他需要用像素来定义长和宽，用 `x` 分割。

```
<?php echo image_tag('test', 'size=100x20') ?>
```

```
=> <img href="/images/test.png" alt="Test" width="100" height="20"/>
```

如果你想让网页资源文件包含在`<head>`部分中（JavaScript 文件和样式表），你在模板中应该使用 `use_stylesheet()` 和 `use_javascript()` 辅助函数，而不是 `_tag()`。他们会为回应增加网页资源文件，这些网页资源文件会在`</head>`之间被包含，并发送给浏览器。

使用绝对路径

链接和网页资源文件辅助函数默认生成的是相对路径。要强制输出绝对路径的话，需要设置 `absolute` 选项为 `true`，如例 9-14 所示。在链接一个 email 信息，RSS 种子或者一个 API 的时候这个技巧非常有用。

例 9-14 - 获得绝对 URL 来替代相对 URL

```
<?php echo url_for('article/read?title=Finance_in_France') ?>
=> '/routed/url/to/Finance_in_France'
<?php echo url_for('article/read?title=Finance_in_France', true) ?>
=> 'http://www.example.com/routed/url/to/Finance_in_France'
```

```
<?php echo link_to('finance',
'article/read?title=Finance_in_France') ?>
=> <a href="/routed/url/to/Finance_in_France">finance</a>
<?php echo link_to('finance',
'article/read?title=Finance_in_France', 'absolute=true') ?>
=> <a href="
http://www.example.com/routed/url/to/Finance_in_France">finance</a>
```

// 同样的情况也适合用 `asset` 辅助函数处理

```
<?php echo image_tag('test', 'absolute=true') ?>
<?php echo javascript_include_tag('myscript', 'absolute=true') ?>
```

SIDEBAR 邮件辅助函数

如今，网络中到处都是 email 搜集机器人，所以你不能直接把 email 地址显示在网页上，那会让你很快就变为垃圾邮件的受害者。这就是为什么 symfony 提供了 mail_to() 辅助函数。

mail_to() 辅助函数使用了 2 个参数：实际的 email 地址和显示的字符串。附加选项有 encode 参数，用来输出一个无法在 HTML 源码中阅读的，机器人不认识的，但是可以被浏览器阅读的字符串。

```
<?php echo mail_to('myaddress@mydomain.com', 'contact') ?>
=> <a href="mailto:myaddress@mydomain.com">contact</a>
<?php echo mail_to('myaddress@mydomain.com', 'contact',
'encode=true') ?>
=> <a href="mailto:myaddress@mydomain.com" >contact</a>
```

经过编码的 email 信息是由随机十进制和十六进制转换过的字符组成的。这个技巧让大多数如今的地址收集器无法工作，但是要注意的是搜集技术更新的非常快。

路由配置

路由系统做两件事：

- 他分析输入请求中的外部 URL 并转换到内部 URL 格式来决定使用哪个模块/动作和请求参数。
- 他把内部 URL 格式的链接格式化成为外部 URL（通过使用链接辅助函数）

转换基于一系列的路由规则。这些规则放在应用程序的 config/ 目录中的 routing.yml 配置文件中。例 9-15 展示了每一个 symfony 项目中都有的默认路由规则。

例 9-15 - 在 myapp/config/routing.yml 中的默认的路由规则

```
# 默认规则
homepage:
  url:    /
  param: { module: default, action: index }

default_symfony:
  url:    /symfony/:action/*
  param: { module: default }

default_index:
  url:    /:module
```

```
param: { action: index }
```

```
default:
```

```
url:    /:module/:action/*
```

规则和模式

路由规则是在外部 URL 和内部 URL 之间双向工作的。基本的规则有以下几点组成：

- 一个唯一的，易于识别的，快速的，可被链接辅助函数使用的标记
- 一个可以被匹配的模式(url 键)
- 一个请求参数数组(param 键)

模式能包含通配符（用*来表示）模式名也能为通配符（由冒号: 开始）。一条和模式名通配符匹配的记录会转换为请求参数。例如，在例 9-15 中定义的 default 规则会匹配任何包含 /foo/bar 的 URL，并设置 module 参数为 foo，action 参数为 bar。在 default_symfony 规则中，symfony 是关键字，action 是具名通配符参数。

路由系统由顶至底来分析 routing.yml 文件，遇到匹配的就停止。这就是为什么要把自定的规则放在默认规则前面。例如，URL /foo/123 与例 9-16 中的两个定义都匹配，但是 symfony 会用 my_rule:，因为他甚至都没有测试到 default: 规则。这个请求由 mymodule/myaction 动作处理，设置 bar 为 123（而不是 foo/123 动作）。

例 9-16 - 规则由顶至底解析

```
my_rule:
```

```
url:    /foo/:bar
```

```
param: { module: mymodule, action: myaction }
```

```
# 默认规则
```

```
default:
```

```
url:    /:module/:action/*
```

NOTE 当一个新的动作建立的时候，这并不意味着你必须为它建立一个路由规则。如果默认的模式/动作模式适合你的话，就不需要考虑 routing.yml 文件了。不管怎样，如果你想自定义动作的外部 URL，就在默认的规则上面增加一条新的规则。

例 9-17 显示了一个为 article/read 动作更改外部 URL 格式的过程。

例 9-17 - 为 acticle/read 动作更改外部 URL 格式

```
<?php echo url_for('my article', 'article/read?id=123') ?>
=> /article/read/id/123          // 默认格式
```

```
// 要改为/article/123 的话
// 在你的 routing.yml 文件开头增加一个新的规则
article_by_id:
  url:    /article/:id
  param: { module: article, action: read }
```

问题是例 9-17 中的 `article_by_id` 规则打破了所有 `article` 模块的默认路由规则。事实上，一个类似 `article/delete` 的 URL 会匹配这个规则而不是 `default`，并会调用 `read` 动作并把 `id` 设置为 `delete` 而不是调用 `delete` 动作。要处理这个问题的话，你必须增加一个模式来限制 `article_by_id` 规则，让其只在 URL 的 `id` 是一个数字的时候才会去匹配。

模式限制

当一个 URL 可以匹配多于一个规则的时候，你必须为模式增加限制或者需求来重定义规则。一个需求是一组正则表达式组成的，他必须能用通配符来匹配规则。

例如，要修改 `article_by_id` 规则，让它只匹配 URL 的 `id` 参数是数字的情况，则需要在规则中增加一项，如例 9-18 所示。

例 9-18 - 路由规则中增加一个需求

```
article_by_id:
  url:    /article/:id
  param: { module: article, action: read }
  requirements: { id: \d+ }
```

现在，`article/delete` URL 不会再匹配 `article_by_id` 规则了，因为 `'delete'` 字符串与需求不匹配。因此，路由系统会继续寻找其他的规则，最终会找到 `default` 规则。

SIDEBAR 永久链接 (Permalinks)

一个好的路由安全原则是隐藏主键并把他们替换为尽可能有意义的字符串。要是你想用文章名字而不是他们的 ID 来显示文章内容会怎么样？这会让外部 URL 看上去像这样：

[http://www.example.com/article/Finance in France](http://www.example.com/article/Finance_in_France)

为了这个扩展，你必须建立一个新的 `permalink` 动作，它使用了一个 `slug` 参数替代了 `id` 参数，并为此增加一个新的规则：

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read }
  requirements: { id: \d+ }

article_by_slug:
  url:          /article/:slug
  param:        { module: article, action: permalink }
```

`permalink` 动作需要由文章主题来确定请求，因此你的模型必须提供相应的方法。

```
public function executePermalink()
{
    $article = ArticlePeer::retrieveBySlug($this->getRequestParameter('slug'));
    $this->forward404Unless($article); // 如果没有文章匹配 slug 则显示 404 错误
    $this->article = $article;        // 把对象传递给模板
}
```

你同时需要在模板中把 `read` 动作的链接替换为 `permalink` 的链接，并激活相应的内部 URL 格式。

```
// 替换
<?php echo link_to('my article', 'article/read?id='.$article->getId()) ?>

// 为
<?php echo link_to('my article', 'article/permalink?slug='.$article->getSlug()) ?>
```

由于有了 `requirements` 行，尽管 `article_by_id` 规则在前面，一个外部的 URL 类似 `/article/Finance_in_France` 还是会匹配 `article_by_slug` 规则。

注意 `slug` 将获得文章，因此你必须在 `Article` 模型描述中增加一个索引给 `slug` 列来优化数据库性能。

设置默认的值

你可以给具名通配符一个默认值让规则工作，甚至这个参数是没有定义的。在 `param` 数组中设置默认的值。

例如，如果没有设置 `id` 参数则不会匹配 `article_by_id` 规则。你能强制设置它，如例 9-19 所示。

例 9-19 - 为通配符设置一个默认值

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read,
id: 1
  }
```

在模式中默认参数不需要找到通配符。在例 9-20 中，`display` 参数设置为 `true`，尽管他不会出现在 URL 中。

例 9-20 - 为请求参数设置默认值

```
article_by_id:
  url:          /article/:id
  param:        { module: article, action: read, id: 1, display:
true }
```

如果仔细观察，你能看到 `article` 和 `read` 是 `module` 和 `action` 的默认值，虽然他们没有在模式中出现。

TIP 你可以在 `sf_routing_default` 配置参数中为所有路由规则定义一个默认的参数。例如，如果你想让所有的规则都有一个 `theme` 参数并被默认设置为 `default` 的话，在你的应用程序的 `config.php` 中增加一行 `sfConfig::set('sf_routing_defaults', array('theme' => 'default'));`。

使用规则名字来加快路由速度

如果规则记号带有标记（@）的话，链接辅助函数接受一个规则记号来替换掉模块/动作，如例 9-21 所示。

例 9-21 - 用规则标签来代替模块/动作

```
<?php echo link_to('my article', 'article/read?id='.$article-
>getId()) ?>
```

// 也能这样写

```
<?php echo link_to('my article', '@article_by_id?id='.$article->getId()) ?>
```

这个技巧既有优点也有缺点。这样写有以下好处：

- 内部 URL 格式完成的更快，因为 symfony 不必去浏览所有的规则来匹配到这个链接。在一个有大量路由格式的超链接的页面，如果使用规则标签来替代模块/动作的话就会快很多。
- 使用规则标签有助于抽象动作后的逻辑。如果决定改变动作名但要保持 URL 不变，只要稍微修改一下 routing.yml 文件即可。所有的 link_to() 调用依旧可以工作而不用做其他修改。
- 用规则名会让调用的逻辑更显而易见。尽管你的模块和动作有一个详细的名字，调用 @display_article_by_slug 还是比 article/display 要好。

另一方面，一个不好的地方是新增超链接不是那么显而易见，因为你总是需要参考 routing.yml 文件来找到动作需要使用哪个标签。

最优的选择取决于项目。最终，这取决于你。

TIP 在测试过程中（在 dev 环境中），如果你想在浏览器中检查哪个规则匹配一个请求（request），可以在网页调试工具条中的 logs and msgs 查找类似 matched route XXX 的日志信息。你会在第 16 章找到关于网页调试模式更多的信息。

增加.html 扩展名

比较这两个 URL：

http://myapp.example.com/article/Finance_in_France

http://myapp.example.com/article/Finance_in_France.html

尽管这是同一个页面，用户和（机器人）也许会因为他们的 URL 不同而认为他们是不同的页面。第二个 URL 是一个深层次的组织完好的静态页面网页目录，可以让搜索引擎知道如何去索引的一个结构。

用路由系统给每一个外部 URL 生成时增加一个后缀，要在应用程序的 settings.yml 中设置 suffix 值，如例 9-22 所示。

例 9-22 - 在 myapp/config/settings.yml 为所有 URL 设置一个后缀

```
prod:
  . settings
    suffix: .html
```


默认的后缀设置是一个点(.)，这意味着路由系统除非特别指定一般不需要增加后缀。

有时候需要为一个特定的路由规则指定一个后缀。在这种情况下，在 `routing.yml` 文件中相对 `url:` 行中直接设置一个后缀，如例 9-23 所示。这样全局后缀设置就会被忽略。

例 9-23 - 在 `myapp/config/routing.yml` 为一个 URL 设置后缀

```
article_list:
  url:          /latest_articles
  param:        { module: article, action: list }

article_list_feed:
  url:          /latest_articles.rss
  param:        { module: article, action: list, type: feed }
```

不使用 `routing.yml` 创建规则

与大多数配置文件一样，`routing.yml` 是一个定义路由规则的解决方案，但不是唯一的方案。你可以在 PHP 中定义规则，也可以在应用程序的 `config.php` 文件中定义，或者在前端控制器脚本中定义，但必须在调用 `dispatch()` 函数前定义，因为这个方法根据现行的路由规则来确定执行的动作。在 PHP 中允许建立动态规则，这取决于你的配置和其他参数。

处理路由规则的对象是 `sfRouting` 单例（**singleton**）。只要加载 `sfRouting::getInstance()`，那么每一段代码中都会有路由工作。它的 `prependRoute()` 方法在 `routing.yml` 文件的已存定义上面增加了一条规则。它需要 4 个参数，就是用来定义一条规则需要的参数：一个路由标签，模式，数组的默认值和所需的其他数组。例如，例 9-18 中的 `routing.yml` 规则定义和例 9-24 中的 PHP 代码是一样的。

例 9-24 - 在 PHP 中定义一个规则

```
sfRouting::getInstance()->prependRoute(
    'article_by_id',           // 路由名
    '/article/:id',           // 路由模式
    array('module' => 'article', 'action' => 'read'), // 默认值
    array('id' => '\d+'),      // 需求
);
```

`sfRouting` 单例（`singleton`）还有其他有用的方法来手动处理路由：`clearRoutes()`，`hasRoutes()`，`getRoutesByName()` 和其他的。要了解更多信息

料，可以参考 API 文档(<http://www.symfony-project.com/api/symfony.html>)。

TIP 当你开始完全理解本书中所展示的理念时，你能通过浏览在线 API 文档或者其他更好的 symfony 源文件来增加对框架的理解。本书中没有描述所有 symfony 的 tweaks 和参数。在线文档却有更详尽的叙述。

在动作中处理路由

如果你需要获得关于当前路由的信息--例如，准备一个功能"返回到 xxx 页"链接--你需要使用 sfRouting 对象的方法。getCurrentInternalUri () 方法返回的 URL 可以被 link_to() 辅助函数调用，就如在例 9-25 中显示的。

例 9-25 - 使用 sfRouting 来获得关于当前路由的信息

```
// 如果这是你想要的一个 URL
http://myapp.example.com/article/21

// 在 article/read 动作中使用下面的语句
$url = sfRouting::getInstance()->getCurrentInternalUri ();
=> article/read?id=21

$url = sfRouting::getInstance()->getCurrentInternalUri (true);
=> @article_by_id?id=21

$route = sfRouting::getInstance()->getCurrentRouteName();
=> article_by_id

// 如果只需要当前的模块/动作名，
// 记住他们只是真实 request 参数
$module = $this->getRequestParameter('module');
$action = $this->getRequestParameter('action');
```

如果需要在动作中转换一个内部 URL 为外部 URL（就如 url_for() 在模板中做的一样）在 sfController 对象中用 genUrl () 方法，就如在例 9-26 中显示的。

例 9-26 - 使用 sfController 来转换一个内部 URI

```
$url = 'article/read?id=21';

$url = $this->getController()->genUrl ($url);
=> /article/21
```

```
$url = $this->getController()->genUrl($uri, true);  
=> http://myapp.example.com/article/21
```

总结

路由是一种双向的机制,目的是为了格式化外部 URL 使他们更友好。URL 重写允许在每个项目中的一个应用程序的 URL 中省略前端控制器名字。如果想要在路由系统双向工作的话,你必须在每次模板中需要输出一个 URL 的时候都使用链接辅助函数。在 `routing.yml` 文件中使用按照流程顺序和规则需求的方式来配置路由系统的规则。`settings.yml` 文件中包含了关于前端控制器名字和在外部 URL 中可能的后缀的附加配置。

第 10 章 表单

可以说，表单占据了开发人员编写模板的大部分时间，而且表单一般都设计得相当糟糕。由于涉及默认值，数据格式，验证，重填，表单处理等许多内容，开发者常常忽略了表单中的一些重要细节。而 `symfony` 恰恰对这个问题给予了特别的关注。本章介绍了为加速表单开发而设计的可以自动完成多种要求的开发工具：

- 表单辅助函数提供了一种比较快地在模板中编写表单控件的方法，特别是在编写诸如日期，下拉列表和富文本之类复杂的元素时。
- 如果要用一个表单去编辑一个对象的属性时，利用对象表单辅助函数可以进一步加速模板的编写。
- YAML 验证文件可以方便表单验证和重填。
- 验证器集成了用于验证输入数据的代码，`symfony` 绑定了满足最常用需求的验证器，开发人员也很容易定制自己的验证器。

表单辅助函数

在模板中，表单元素的 HTML 标签常常和 PHP 代码混杂在一起。`symfony` 中的表单辅助函数就是为了减少这种情形的发生并且避免在 `<input>` 标签中不断重复 `<?php echo` 标签。

主要的表单标签

根据前面章节的介绍，你必须用 `form_tag()` 辅助函数创建表单，因为它可以将用参数表示的动作转换为经路由过的 URL。第二个参数还可以支持额外的选项。例如，可以改变默认的 `method`，可以改变默认的 `enctype` 或指定其他的属性，参见 例 10-1。

例 10-1 `form_tag()` 辅助函数

```
<?php echo form_tag('test/save') ?>
=> <form method="post" action="/path/to/save">

<?php echo form_tag('test/save', 'method=get multipart=true
class=simpleForm') ?>
=> <form method="get" enctype="multipart/form-data"
class="simpleForm" action="/path/to/save">
```

因为没有必要提供表单结束辅助函数，所以尽管看起来不怎么美观，你仍旧需要加上 HTML 的 `</form>` 标签。

标准的表单元素

有了表单辅助函数，表单中的每个元素都会默认以元素名作为其 `id` 属性。这个约定很有用。例 10-2 给出了所有标准表单辅助函数及相关的选项。

例 10-2 标准表单辅助函数语法

```
// 输入框(text field)
<?php echo input_tag('name', 'default value') ?>
=> <input type="text" name="name" id="name" value="default value" />
```

```
// 所有表单辅助函数都接受一个额外的选项参数
// 它允许你为生成的标签加上定制的属性
<?php echo input_tag('name', 'default value', 'maxlength=20') ?>
=> <input type="text" name="name" id="name" value="default value"
maxlength="20" />
```

```
// 文本框 (textarea)
<?php echo textarea_tag('name', 'default content', 'size=10x20') ?>
=> <textarea name="name" id="name" cols="10" rows="20">
    default content
</textarea>
```

```
// 复选框 (checkbox)
<?php echo checkbox_tag('single', 1, true) ?>
<?php echo checkbox_tag('driverslicense', 'B', false) ?>
=> <input type="checkbox" name="single" id="single" value="1"
checked="checked" />
      <input type="checkbox" name="driverslicense" id="driverslicense"
value="B" />
```

```
// 单选按钮 (Radio button)
<?php echo radiobutton_tag('status[]', 'value1', true) ?>
<?php echo radiobutton_tag('status[]', 'value2', false) ?>
=> <input type="radio" name="status[]" id="status_value1"
value="value1" checked="checked" />
    <input type="radio" name="status[]" id="status_value2"
value="value2" />
```

```
// 下拉列表 (Dropdown list/select)
<?php echo select_tag('payment',
    ' <option selected="selected">Vi sa</option>
    <option>Eurocard</option>
    <option>Mastercard</option>' )
```

```

?>
=> <select name="payment" id="payment">
    <option selected="selected">Vi sa</option>
    <option>Eurocard</option>
    <option>Mastercard</option>
</select>

// 可选项列表
<?php echo options_for_select(array('Vi sa', 'Eurocard', 'Mastercard'),
0) ?>
=> <option value="0" selected="selected">Vi sa</option>
    <option value="1">Eurocard</option>
    <option value="2">Mastercard</option>

// 混合了可选项的下拉列表辅助函数
<?php echo select_tag('payment', options_for_select(array(
'Vi sa',
'Eurocard',
'Mastercard'
), 0)) ?>
=> <select name="payment" id="payment">
    <option value="0" selected="selected">Vi sa</option>
    <option value="1">Eurocard</option>
    <option value="2">Mastercard</option>
</select>

// 用关联数组指明选项名称
<?php echo select_tag('name', options_for_select(array(
'Steve' => 'Steve',
'Bob'    => 'Bob',
'Al bert' => 'Al bert',
'I an'    => 'I an',
'Buck'    => 'Buck'
), 'I an')) ?>
=> <select name="name" id="name">
    <option value="Steve">Steve</option>
    <option value="Bob">Bob</option>
    <option value="Al bert">Al bert</option>
    <option value="I an" selected="selected">I an</option>
    <option value="Buck">Buck</option>
</select>

// 可复选的下拉列表(选中值可以是一个数组)
<?php echo select_tag('payment', options_for_select(

```

```

        array('Vi sa' => 'Vi sa', 'Eurocard' => 'Eurocard', 'Mastercard' =>
'Mastercard'),
        array('Vi sa', 'Mastecard'),
    ), array('multiple' => true))) ?>

```

```

=> <select name="payment[]" id="payment" multiple="multiple">
    <option value="Vi sa" selected="selected">Vi sa</option>
    <option value="Eurocard">Eurocard</option>
    <option value="Mastercard">Mastercard</option>
</select>
// 可复选的下拉列表(选中值可以是一个数组)
<?php echo select_tag('payment', options_for_select(
    array('Vi sa' => 'Vi sa', 'Eurocard' => 'Eurocard', 'Mastercard' =>
'Mastercard'),
    array('Vi sa', 'Mastecard')
), 'multiple=multiple') ?>
=> <select name="payment" id="payment" multiple="multiple">
    <option value="Vi sa" selected="selected">
    <option value="Eurocard">Eurocard</option>
    <option value="Mastercard"
selected="selected">Mastercard</option>
</select>

```

```

// 上传文件域 (Upload file field)
<?php echo input_file_tag('name') ?>
=> <input type="file" name="name" id="name" value="" />

```

```

// 密码输入框 (Password field)
<?php echo input_password_tag('name', 'value') ?>
=> <input type="password" name="name" id="name" value="value" />

```

```

// 隐藏域 (Password field)
<?php echo input_hidden_tag('name', 'value') ?>
=> <input type="hidden" name="name" id="name" value="value" />

```

```

// 提交按钮(文本格式) (Submit button (as text))
<?php echo submit_tag('Save') ?>
=> <input type="submit" name="submit" value="Save" />

```

```

// 提交按钮(图片格式) (Submit button (as image))
<?php echo submit_image_tag('submit_img') ?>
=> <input type="image" name="submit" src="/images/submit_img.png" />

```

`submit_image_tag()` 辅助函数使用的语法和 `image_tag()` 相同，也具有相同的优点。

NOTE 对于单选按钮，`id` 属性没有默认地被设定为 `name` 属性的值，而是将 `name` 属性值和选项值混合后作为 `id` 属性的默认值。之所以这样做，是为了实现“选中一个就自动去除另一个”的目的，你需要有多个同名的单选按钮标签，而根据前面 `id=name` 的约定，将导致在页面中出现多个含有同样 `id` 属性的 HTML 标签，这是被严格禁止的。

SIDEBAR 处理表单提交

如何取得用户通过表单提交的数据呢？这些数据存放在请求参数中，所以动作只要调用 `$this->getRequestParameter($elementName)` 就可以取得数据。

在同一个动作中既显示表单又处理表单是一种比较好的方法。对应不同的请求方法(GET 或 POST)，要么调用表单模板，要么处理表单并将请求重定向到另一个动作去。

```
// mymodule/actions/actions.class.php
public function executeEditAuthor()
{
    if ($this->getRequest()->getMethod() != sfRequest::POST)
    {
        // 显示表单
        return sfView::SUCCESS;
    }
    else
    {
        // 对提交的表单加以处理
        $name = $this->getRequestParameter('name');
        ...
        $this->redirect('mymodule/anotheraction');
    }
}
```

这段代码要能正常工作，表单处理和表单显示必须在同一个动作中。

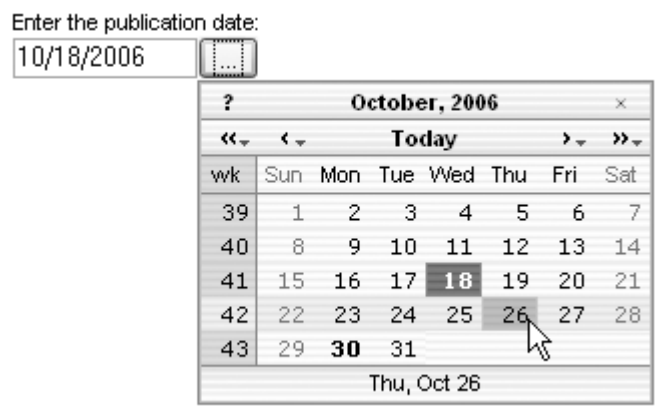
```
// mymodule/templates/editAuthorSuccess.php
...
```


symfony 还专门设计了为后台处理异步请求的表单辅助函数。下一章有关 AJAX 的介绍将会提供更详细的信息。

日期输入控件

表单常用于输入日期，而日期格式错误常常是表单提交失败的主要原因。如果你将 rich 选项设定为 true，则 input_date_tag() 辅助函数可以用一个交互式的 JavaScript 日历来帮助用户输入日期，见图 10-1 所示。

图 10-1 富日期输入标签



如果未设置 rich 选项，则辅助函数将会输出三个 select 标签，可取值为年、月、日的正常取值范围。你也可以通过调用三个辅助函数 select_day_tag(), select_month_tag() 和 select_year_tag(), 来分别显示下拉列表。这些元素的默认值是当前的年、月、日。例 10-3 演示了日期输入辅助函数的应用。

例 10-3 日期输入辅助函数

```
<?php echo input_date_tag('dateofbirth', '2005-05-03',  
'rich=true') ?>  
=> 一个文本输入域加一个日期输入控件  
  
// 以下辅助函数需要包括日期辅助函数组  
<?php use_helper('Date') ?>  
<?php echo select_day_tag('day', 1, 'include_custom=Choose a day') ?>  
=> <select name="day" id="day">  
    <option value="">Choose a day</option>  
    <option value="1" selected="selected">01</option>  
    <option value="2">02</option>  
    ...  
    <option value="31">31</option>  
</select>
```

```
<?php echo select_month_tag('month', 1, 'include_custom=Choose a month use_short_month=true') ?>
```

```
=> <select name="month" id="month">
    <option value="">Choose a month</option>
    <option value="1" selected="selected">Jan</option>
    <option value="2">Feb</option>
    ...
    <option value="12">Dec</option>
</select>
```

```
<?php echo select_year_tag('year', 2007, 'include_custom=Choose a year year_end=2010') ?>
```

```
=> <select name="year" id="year">
    <option value="">Choose a year</option>
    <option value="2006">2006</option>
    <option value="2007" selected="selected">2007</option>
    ...
</select>
```

`input_date_tag()` 辅助函数可接受的日期值是 PHP 函数 `strtotime()` 可以识别的值。例 10—4 列出的格式是可以使用的，而例 10—5 中的格式是严格禁止使用的。

例 10—4 日期辅助函数可以接受的日期格式

// 运行正常

```
<?php echo input_date_tag('test', '2006-04-01', 'rich=true') ?>
<?php echo input_date_tag('test', 1143884373, 'rich=true') ?>
<?php echo input_date_tag('test', 'now', 'rich=true') ?>
<?php echo input_date_tag('test', '23 October 2005', 'rich=true') ?>
<?php echo input_date_tag('test', 'next tuesday', 'rich=true') ?>
<?php echo input_date_tag('test', '1 week 2 days 4 hours 2 seconds',
'rich=true') ?>
```

// 返回空值

```
<?php echo input_date_tag('test', null, 'rich=true') ?>
<?php echo input_date_tag('test', '', 'rich=true') ?>
```

例 10—5 日期辅助函数中的错误日期格式

// 日期 0 = 01/01/1970

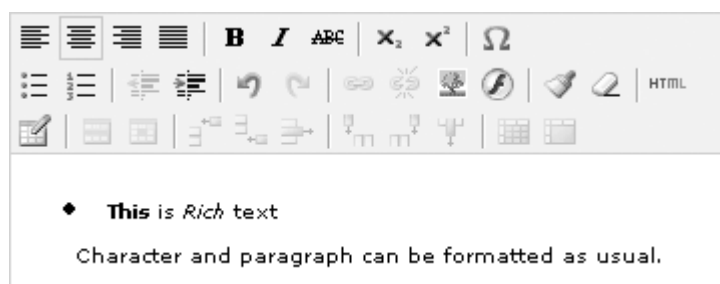
```
<?php echo input_date_tag('test', 0, 'rich=true') ?>
```

```
// 非英语日期格式不能正常运行
<?php echo input_date_tag('test', '01/04/2006', 'rich=true') ?>
```

编辑富文本 (rich text)

因为集成了 TinyMCE 和 FCKEditor 插件，因而可以对<textarea>标签的文本进行富文本编辑，也就是说可以用类似于文字处理器界面的按钮将文本格式化为粗体，斜体或其它样式，见图 10—2。

图 10—2 编辑富文本



这两种插件都需要手工安装。因为安装方法相同，这里只介绍 TinyMCE 的安装方法。你可以从该项目的网站(<http://tinymce.moxiecode.com/>)下载它并将它解压到一个临时目录中，然后将 tinymce/jscripts/tiny_mce/ 目录复制到你的项目的 web/js/ 目录中，并在 settings.yml 中定义指向这个库的路径，如例 10—6 所示。

例 10—6 设置 TinyMCE 库路径

```
all:
  .settings:
    rich_text_js_dir: js/tiny_mce
```

设置好后，再加入 rich=true 选项，就可以在文本框内进行富文本编辑了。你也可以用 tinymce_options 选项为 Javascript 编辑器设置定制的选项。参见例 10—7

例 10—7 富文本框

```
<?php echo textarea_tag('name', 'default content', 'rich=true
size=10x20') ?>
=> 一个具有 TinyMCE 功能的富文本编辑区
<?php echo textarea_tag('name', 'default content', 'rich=true
size=10x20tinymce_options=language:"fr", theme_advanced_buttons2:"sepa
rator"') ?>
```

=>一个具有定制过的 TinyMCE 功能的富文本编辑区

选择国家和语言

你可能会需要一个可以选择国家的域。因为在不同的语言中国家名也不相同，所以国家名下拉列表要根据用户的 culture 值来调整（第 13 章有 culture 的详细介绍）。如例 10—8 所示，select_country_tag() 辅助函数可以完成所有工作，它会按照不同的语言显示国家名，并用 ISO 标准的国家代码作为选项的值。

例 10-8 选取国家名称辅助函数

```
<?php echo select_country_tag('country', 'AL') ?>
=> <select name="country" id="country">
    <option value="AF">Afghanistan</option>
    <option value="AL" selected="selected">Albania</option>
    <option value="DZ">Algeria</option>
    <option value="AS">American Samoa</option>
    ...
```

类似于 select_country_tag() 辅助函数，select_language_tag() 辅助函数可以显示一个语言名称列表，见例 10—9

例 10—9 选取语言名称辅助函数

```
<?php echo select_language_tag('language', 'en') ?>
=> <select name="language" id="language">
    ...
    <option value="elx">Elamite</option>
    <option value="en" selected="selected">English</option>
    <option value="enm">English, Middle (1100-1500)</option>
    <option value="ang">English, Old (ca. 450-1100)</option>
    <option value="myv">Erzya</option>
    <option value="eo">Esperanto</option>
    ...
```

对象表单辅助函数

如果利用表单去编辑对象的属性，那么用标准的链接辅助函数去写代码非常费时。比如，要编辑 Customer 对象的 telephone 属性，应该写成：

```
<?php echo input_tag('telephone', $customer->getTelephone()) ?>
```

```
=> <input type="text" name="telephone" id="telephone"
value="0123456789" />
```

为了避免重复地写属性名，symfony 为每个表单辅助函数另外提供了一个对象表单辅助函数。对象表单辅助函数将从对象和方法名来获得表单元素的名字和默认值。前面用过的 `input_tag()` 可以变换为如下形式：

```
<?php echo object_input_tag($customer, 'getTelephone') ?>
=> <input type="text" name="telephone" id="telephone"
value="0123456789" />
```

虽然 `object_input_tag()` 看上去并不省事，但是，每个标准的表单辅助函数都有一个对应的对象表单辅助函数，并且它们的语法都一样，这样要生成表单就非常简单。这也是为什么在 symfony 生成的框架和后台管理都广泛使用了对象表单辅助函数的原因(详见第 14 章)。例 10-10 展示了对象表单辅助函数的用法。

例 10-10 对象表单辅助函数语法

```
<?php echo object_input_tag($object, $method, $options) ?>
<?php echo object_input_date_tag($object, $method, $options) ?>
<?php echo object_input_hidden_tag($object, $method, $options) ?>
<?php echo object_textarea_tag($object, $method, $options) ?>
<?php echo object_checkbox_tag($object, $method, $options) ?>
<?php echo object_select_tag($object, $method, $options) ?>
<?php echo object_select_country_tag($object, $method, $options) ?>
<?php echo object_select_language_tag($object, $method, $options) ?>
```

因为不宜替一个密码元素预先设定默认值，所以没有提供 `object_password_tag()` 辅助函数。

CAUTION 与一般的表单辅助函数不同，只有你在模板中用 `use_helper('Object')` 明确声明你将使用对象辅助函数组后，才能使用对象表单辅助函数。

对象表单辅助函数中最有趣的就是 `objects_for_select()` 和 `object_select_tag()` 两个，它们都与下拉列表有关。

生成对象的下拉列表

用前面介绍过的 `options_for_select()` 辅助函数及其他标准辅助函数，可以将一个 PHP 关联数组转化为一个选项列表。见例 10-11 所示。

例 10-11 用 `options_for_select()` 产生一个基于数组的选项列表。

```
<?php echo options_for_select(array(
    '1' => 'Steve',
    '2' => 'Bob',
    '3' => 'Albert',
    '4' => 'Ian',
    '5' => 'Buck'
), 4) ?>
=> <option value="1">Steve</option>
    <option value="2">Bob</option>
    <option value="3">Albert</option>
    <option value="4" selected="selected">Ian</option>
    <option value="5">Buck</option>
```

假设你已经有了一个从 Propel 查询得到的类 `Author` 的对象数组，如果你想建立一个基于这个数组的选项列表，你需要用一个循环遍历每个对象的 `id` 和 `name` 项，如例 10-12 所示。

例 10-12 用 `options_for_select()` 产生一个基于对象数组的选项列表

```
// 动作
$options = array();
foreach ($authors as $author)
{
    $options[$author->getId()] = $author->getName();
}
$this->options = $options;

// 模板
<?php echo options_for_select($options, 4) ?>
```

因为经常需要进行这类处理，所以 `symfony` 提供了一个 `objects_for_select()` 辅助函数来直接从一个对象数组创建一个选项列表。这个辅助函数需要两个另外的参数：用于遍历值的方法名和要生成的标签的文本内容。例 10-12 可以简化为如下形式：

```
<?php echo objects_for_select($authors, 'getId', 'getName', 4) ?>
```

这样已经既快又好，但是当你要处理外键列时，`symfony` 还提供了更多的方便。

创建一个基于外键列的下拉列表

外键列可取的值是外键所在表的记录的主键值。例如，如果 `article` 表有一个 `author_id` 列，它是 `author` 表的外键，那么这个列的可能值就是 `author` 表的所有 `id` 列的值。通常，用于编辑一篇文章的作者的下拉列表看上去如例 10—13 所示。

例 10—13 用 `objects_for_select()` 创建一个基于外键的选项列表

```
<?php echo select_tag('author_id', objects_for_select(
    AuthorPeer::doSelect(new Criteria()),
    'getId',
    '__toString()',
    $article->getAuthorId()
)) ?>
=> <select name="author_id" id="author_id">
    <option value="1">Steve</option>
    <option value="2">Bob</option>
    <option value="3">Albert</option>
    <option value="4" selected="selected">Ian</option>
    <option value="5">Buck</option>
</select>
```

实际上，仅仅用 `object_select_tag()` 就可以完成所有的工作。它显示一个下拉列表，其选项为外键所在表的所有可能记录值。这个辅助函数可以根据数据库模式文件猜出外键列和外键所在的表，因此它的语法非常简洁。以下代码和例 10—13 完成同样的功能：

```
<?php echo object_select_tag($article, 'getAuthorId') ?>
```

`object_select_tag()` 辅助函数根据作为参数传递的方法名就可以得到相关的 `peer` 类名（在本例中就是 `AuthorPeer`）。但是，你可以在第 3 个参数 `relate_class` 中设定你自己的类名。`<option>` 标签的文本内容是根据类的 `_toString()` 方法得到的记录名（如果 `$author->_toString()` 方法没有定义，就用主键代替）。另外，选项列表是从带有空条件的 `doSelect()` 方法运行得出的结果中取得的，它根据记录的创建时间排序。如果你想按指定的顺序显示记录的一个子集，你可以在 `peer` 类中创建一个方法，该方法返回满足条件的对象数组，然后你就可以设定 `peer_method` 选项。最后，你可以通过设定 `include_blank` 或 `include_custom` 选项，在下拉列表的顶部加一个空的选项或某个定制的选项。例 10—14 显示了 `object_select_tag()` 辅助函数的各种选项的用法。

例 10—14 `object_select_tag()` 辅助函数的选项

```
// 基本语法
<?php echo object_select_tag($article, 'getAuthorId') ?>
// 用 AuthorPeer::doSelect(new Criteria()) 生成列表

// 修改用来获取可选值的 peer 类
<?php echo object_select_tag($article, 'getAuthorId',
'related_class=Foobar') ?>
// 用 FoobarPeer::doSelect(new Criteria()) 生成列表

// 修改获取可选值的 peer 方法
<?php echo object_select_tag($article,
'getAuthorId', 'peer_method=getMostFamousAuthors') ?>
// 用 AuthorPeer::getMostFamousAuthors(new Criteria()) 生成列表

// 在列表的最上面增加<option value="">&nbsp;  </option>
<?php echo object_select_tag($article, 'getAuthorId',
'include_blank=true') ?>

// 在列表最上面增加<option value="">Choose an author</option>
<?php echo object_select_tag($article, 'getAuthorId',
'include_custom=Choose an author') ?>
```

更新对象

如果要编写专用于编辑对象属性的表单，那么利用对象辅助函数就比在动作中编写代码容易。例如，如果你有一个类 Author 包含 name, age, address 等属性，你就可以如 10—15 那样编写表单：

例 10—15 仅用对象辅助函数的表单

```
<?php echo form_tag('author/update') ?>
  <?php echo object_input_hidden_tag($author, 'getId') ?>
  Name: <?php echo object_input_tag($author, 'getName') ?><br />
  Age:  <?php echo object_input_tag($author, 'getAge') ?><br />
  Address: <br />
           <?php echo object_textarea_tag($author, 'getAddress') ?>
</form>
```

提交表单时，将调用 author 模块的 update 动作，该动作只要调用由 Propel 生成的 fromArray() 方法就可以更新对象，参见例 10—16。

例 10—16 基于对象表单辅助函数的表单提交处理函数。


```

public function executeUpdate ()
{
    $author = AuthorPeer::retrieveByPk($this->getRequestParameter('id'));
    $this->forward404Unless($author);

    $author->fromArray($this->getRequest()->getParameterHolder()->getAll(), AuthorPeer::TYPE_FIELDNAME);
    $author->save();

    return $this->redirect('/author/show?id=' . $author->getId());
}

```

表单验证

第 6 章介绍过如何在动作类中用 `validateXXX()` 方法验证请求参数。但是，如果你用这种方法去验证表单提交的话，你就会没完没了地写同样的代码。`symfony` 提供了一种技术，不用动作类中的 PHP 代码，而仅仅是用一个 YAML 文件就可以验证表单。

为了说明表单验证，让我们先看一下例 10-17 中的表单示例。这是一个典型的联系人表单，包括 `name`、`email`、`age` 和 `message` 域。

例 10-17 联系人表单示例。文件路径为 `modules/contact/templates/indexSuccess.php`

```

<?php echo form_tag('contact/send') ?>
    Name:    <?php echo input_tag('name') ?><br />
    Email:   <?php echo input_tag('email') ?><br />
    Age:     <?php echo input_tag('age') ?><br />
    Message: <?php echo textarea_tag('message') ?><br />
    <?php echo submit_tag() ?>
</form>

```

表单验证的要求就是：如果用户提交的表单中包含了无效数据，浏览器将在页面上显示出错误信息。我们先对上述表单定义什么样的数据才是有效的。

- l `name` 域是必需的，且必须是 2 到 100 个字符的文本。
- l `email` 域是必需的，且必须是 2 到 100 个字符的文本，并是一个有效的电子邮件地址。
- l `age` 域是必需的，且必须是 0-100 之间的整数。
- l `message` 域是必需的。

你当然可以为联系人表单定义更复杂的验证规则，不过在本例中以上规则就足以说明问题了。

NOTE 表单验证可以在服务器端进行，也可以在客户端进行。为了防止错误数据破坏数据库，服务器端的验证是必需的。尽管客户端的验证可以极大地提高用户体验，但客户端的验证却不是必需的。客户端验证通常用 JavaScript 定制。

验证器

你可以看到例子中的 name 和 email 域使用同样的验证规则。有些验证规则在 web 表单中频繁出现，为此，symfony 将这些规则的 PHP 代码打包成验证器。一个验证器是一个提供了 execute() 方法的简单类。这个方法以域的值为参数，如果值有效则返回 true，否则返回 false。

symfony 包含多个验证器（本章后面的“标准 symfony 验证器”一节将详细介绍），这里我们先重点说明一下 sfStringValidator。该验证器检测输入值是否为一个字符串，且字符数在两个指定的值之间（通过调用 initialize() 方法定义）。这正是我们验证 name 域时所需要的。例 10-18 显示了如何在一个验证方法中使用这个验证器。

例 10-18 用可复用的验证器验证请求参数，文件路径为 modules/contact/action/actions.class.php

```
public function validateSend()
{
    $name = $this->getRequestParameter('name');

    // name 域必需有值
    if (!$name)
    {
        $this->getRequest()->setError('name', 'The name field cannot be left blank');

        return false;
    }

    // name 域的值必需是长度在 2-100 之间的字符串
    $myValidator = new sfStringValidator();
    $myValidator->initialize($this->getContext(), array(
        'min'          => 2,
        'min_error'    => 'This name is too short (2 characters minimum)',
        'max'          => 100,
        'max_error'    => 'This name is too long. (100 characters maximum)',
    ));
```

```

    if (!$myValidator->execute($name))
    {
        return false;
    }

    return true;
}

```

如果用户在例 10-17 中的表单的 name 域里输入值 a，则 sfStringValidator 的 execute() 方法将返回 false（因为字符串长度小于 2），因而 validateSend() 方法也返回 false。这样，executeSend() 方法将不执行，而是执行 handleErrorSend() 方法。

TIP sfRequest 对象的 setError() 方法将要显示的出错信息提供给模板(本章后面的“显示表单的出错信息”一节将会详细说明)。验证器在内部设定了错误信息，所以你可以为不能通过验证的不同情形定义不同的错误信息。这正是 sfStringValidator 中的初始化参数 min_error 和 max_error 的作用。

本例中定义的所有规则都可以用验证器替代：

- name: sfStringValidator (min=2, max=100)
- email: sfStringValidator (min=2, max=100) 和 sfEmailValidator
- age: sfNumberValidator (min=0, max=120)

不过“域是必需的”规则却不是由验证器处理的。

验证文件

虽然你可以在 validateSend() 方法中用验证器轻易地实现你的联系人表单验证，但这也意味着你将编写大量重复的代码。symfony 用另一种方法来定义表单的验证规则，这就是用 YAML 文件。例 10-19 给出了 name 域的验证规则的替换表达方法，其验证结果与例 10-18 得到的相同。

例 10-19 验证文件，路径为 modules/contact/validate/send.yml

```

fields:
  name:
    required:
      msg:      The name field cannot be left blank
    sfStringValidator:
      min:      2
      min_error: This name is too short (2 characters minimum)
      max:      100
      max_error: This name is too long. (100 characters maximum)

```

在一个验证文件里，`fields` 头列出了所有需要验证的域，并且如果有值时，验证器必须对该值进行检验。每个验证器的参数和你手工初始化使用的参数相同。只要需要，一个域可以被多个验证器验证。

NOTE 一个验证器验证失败并不会导致整个验证过程结束。`symfony` 会检验所有的验证器，只有至少一个验证失败时，才会宣布整个验证失败。并且即使验证文件中的某些验证规则失败，`symfony` 仍旧会找一个 `validateXXX()` 方法去执行。所以两种验证技术是互补的。好处就是对于有多个错误的表单，可以揭示出所有的错误信息。

验证文件在模块的 `validate` 目录下，并用要验证的动作名称来命名。例如，例 10-19 的文件路径就是 `validate/send.yml`。

重新显示表单

只要验证失败，`symfony` 自动在动作类中找 `handleErrorSend()` 方法，如果没有这个方法，就去显示 `sendError.php` 模板。

不过，告诉用户未能通过验证的更好方法却是再显示一遍包含出错信息的表单。为此，你需要重载 `handleErrorSend()` 方法，并且重定向到显示表单的动作去（上例中，应该是 `module/index`），参见例 10-20。

例 10-20 再次显示表单，文件路径为
`modules/contact/actions/actions.class.php`

```
class ContactActions extends sfActions
{
    public function executeIndex()
    {
        // 显示表单
    }

    public function handleErrorSend()
    {
        $this->forward('contact', 'index');
    }

    public function executeSend()
    {
        // 处理表单提交
    }
}
```

如果你用同一个动作显示表单和处理表单提交，只要让 `handleErrorSend()` 方法简单地返回 `sfView::SUCCESS`，就可以从 `sendSuccess.php` 重新显示表单，参见例 10-21。

例 10-21 一个动作同时显示和处理表单，文件路径为 `modules/contact/actions/actions.class.php`

```
class ContactActions extends sfActions
{
    public function executeSend()
    {
        if ($this->getRequest()->getMethod() != sfRequest::POST)
        {
            // 为模板准备数据

            // 显示表单
            return sfView::SUCCESS;
        }
        else
        {
            // 处理表单提交

            ...
            $this->redirect('myModule/anotherAction');
        }
    }
    public function handleErrorSend()
    {
        // 为模板准备数据

        // 显示表单

        return sfView::SUCCESS;
    }
}
```

准备数据的代码可以分离到动作类一个的保护方法中，以免在 `executeSend()` 和 `handleErrorSend()` 方法中重复该代码。

经过这样修改后，如果用户输入一个无效的名字，表单就会重新显示，但是输入的数据没有了，而且解释错误原因的出错信息也没有显示。为解决这个问题，你必须修改显示表单的模板，以便将出错信息显示在校验出错域的旁边。

在表单中显示出错信息

当一个域验证失败时，出错信息会作为一个验证器参数传送给请求（就像在例 10-18 中你用 `setError()` 手工添加错误一样）。`sfRequest` 对象提供两个有用的方法用于查看错误信息：`hasError()` 和 `getError()`，它们都以域名为参数。另外，你可以借助 `hasErrors()` 在表单的顶部显示一个警告信息，以引起用户注意有一个或多个域输入无效。例 10-22 和 10-23 给出了如何使用这些方法的例子。

例 10-22 在表单顶部显示错误信息，文件路径 `templates/indexSuccess.php`

```
<?php if ($sf_request->hasErrors()): ?>
    <p>The data you entered seems to be incorrect.
    Please correct the following errors and resubmit:</p>
    <ul>
        <?php foreach($sf_request->getErrors() as $name => $error): ?>
            <li><?php echo $name ?>: <?php echo $error ?></li>
        <?php endforeach; ?>
    </ul>
<?php endif; ?>
```

例 10-23 在表单内显示错误信息，文件路径 `templates/indexSuccess.php`

```
<?php echo form_tag('contact/send') ?>
    <?php if ($sf_request->hasError('name')): ?>
        <?php echo $sf_request->getError('name') ?> <br />
    <?php endif; ?>
    Name:    <?php echo input_tag('name') ?><br />
    ...
    <?php echo submit_tag() ?>
</form>
```

例 10-23 中包含 `getError()` 的条件句写起来有点长，所以假如你预先声明了 `Validation` 辅助函数组，`symfony` 提供了一个 `form_error()` 辅助函数来替代这些长的代码。例 10-24 用这个辅助函数替换了例 10-23 的代码。

例 10-24 用缩写方法在表单中显示出错信息

```
<?php use_helper('Validation') ?>
<?php echo form_tag('contact/send') ?>

        <?php echo form_error('name') ?><br />
    Name:    <?php echo input_tag('name') ?><br />
    ...
```

```
<?php echo submit_tag() ?>
</form>
```

`form_error()` 辅助函数在每个出错信息的前后分别加了一个特殊字符，以使出错信息更醒目。这个字符的默认值是向下的箭头（对应于 `&darr`），你可以在 `settings.yml` 文件里改变这个默认值：

```
all:
  .settings:
    validation_error_prefix: ' &darr; &nbsp; '
    validation_error_suffix: ' &nbsp; &darr; '
```

现在，当验证失败时，表单可以正确地显示出错信息了，但是用户之前输入的数据都没有了。你应该将这些数据重新放入表单中，以便提供更好用户体验。

重新填充表单数据

如例 10—20 所示，当一个错误通过 `forward()` 方法处理后，原来的请求数据仍是可以读取的，而且用户输入的数据保存在请求参数中。所以你可以将默认值加入到每个域中去为表单填充数据，见例 10—25。

例 10—25 验证失败后，用默认值为表单重新填充数据，文件路径为 `templates/indexSuccess.php`

```
<?php use_helper('Validation') ?>
<?php echo form_tag('contact/send') ?>
    <?php echo form_error('name') ?><br />
    Name: <?php echo input_tag('name', $sf_params-
>get('name')) ?><br />
    <?php echo form_error('email') ?><br />
    Email: <?php echo input_tag('email', $sf_params-
>get('email')) ?><br />
    <?php echo form_error('age') ?><br />
    Age: <?php echo input_tag('age', $sf_params->get('age')) ?><br
/>
    <?php echo form_error('message') ?><br />
    Message: <?php echo textarea_tag('message', $sf_params-
>get('message')) ?><br />
    <?php echo submit_tag() ?>
</form>
```

当然写这些代码又是件很麻烦的事，所以 `symfony` 再次提供了一种方法让你不用改变元素的默认值，就可以重新为表单的所有域填充数据，这个方法就是直接在 `YAML` 验证文件中设置表单的 `fillin` 属性，参见例 10—26。

例 10—26 激活 `fillin` 属性，以便在验证失败时为表单重新填充数据，文件路径为 `validate/send.yml`

```
fillin:
  enabled: true # 允许重填表单
  param:
    name: test # 表单名, 如果页面中只有一个表单, 可以忽略
    skip_fields: [email] # 不重填这些域
    exclude_types: [hidden, password] # 不重填这些类型的域
    check_types: [text, checkbox, radio, password, hidden] # 重填这些类型的域
```

自动重新填充适用于输入框、复选框、单选按钮、文本框和下拉列表（包括简单列表和多重列表），但不适用于密码和隐藏域，也不适用于文件标签。

NOTE `fillin` 属性是在发送响应内容给用户之前，通过编译用 XML 表示的响应内容来工作的，所以如果响应内容不是一个有效的 XHTML 文档，`fillin` 就不能工作。

你有可能想在写回表单域之前对用户输入的值进行转换。只要你在 `converters` 属性下定义了可以用函数调用的转换，包括转义，URL 重写，特殊字符转换等，就可以将这些转换作用到表单的域上去。

例 10—27 在 `fillin` 之前转换输入，文件路径是 `validate/send.yml`

```
fillin:
  enabled: true
  param:
    name: test
    converters: # 要实施的转换
      htmlentities: [first_name, comments]
      htmlspecialchars: [comments]
```

标准 symfony 验证器

symfony 为你的表单提供了下列标准的验证器

- `SfStringValidator`
- `SfNumberValidator`
- `SfEmailValidator`
- `SfUrlValidator`
- `SfRegexValidator`
- `SfCompareValidator`
- `SfPropelUniqueValidator`
- `SfFileValidator`

- `sfCallbackValidator`

每个标准验证器都有一组默认的参数和错误信息，这些值和信息可以轻松通过 `initialize()` 方法或 YAML 文件来重新设置。 下面几节内容将描述这些验证器并给出示例。

字符串验证器

`sfStringValidator` 对参数进行与字符串有关的验证。

`sfStringValidator`:

```
values:      [foo, bar]
values_error: The only accepted values are foo and bar
insensitive: false # If true, comparison with values is case
insensitive
min:         2
min_error:   Please enter at least 2 characters
max:         100
max_error:   Please enter less than 100 characters
```

数字验证器

如果参数是一个数值， 你可以用 `sfNumberValidator` 来验证数的大小。

`sfNumberValidator`:

```
nan_error:   Please enter an integer
min:         0
min_error:   The value must be more than zero
max:         100
max_error:   The value must be less than 100
```

E-Mail 验证器

`sfEmailValidator` 用于验证参数值是否符合电子邮件地址的格式标准。

`sfEmailValidator`:

```
strict:      true
email_error: This email address is invalid
```

虽然 RFC822 定义了电子邮件的地址格式，但通常认可的电子邮件地址格式要比这个标准更加严格。比如，RFC 认为 `me@localhost` 是有效的电子邮件地址，而你却可能不接受。如果你将 `strict` 参数设定为 `true`（这是默认值），那么只有符合 `name@domain.extension` 格式的电子邮件地址才能通过验证；如果将该参数设定为 `false`，则采用 RFC 的规则。

URL 验证器

`sfUrlValidator` 用于验证一个域的值是否是一个有效的 URL。

```
sfUrlValidator:
  url_error:    This URL is invalid
```

正则表达式验证器

`sfRegexValidator` 验证一个值是否与一个 Perl 兼容正则表达式模式相匹配。

```
sfRegexValidator:
  match:        No
  match_error:  Posts containing more than one URL are considered as
spam
  pattern:      /http.*http/si
```

其中的 `match` 参数为 Yes 时，表明请求参数与模式匹配为有效；如果为 No，则表明请求参数与模式匹配为无效。

比较验证器

`sfCompareValidator` 用于检测两个不同的请求参数是否相同。这对于密码检测非常有用。

```
fields:
  password1:
    required:
      msg:    Please enter a password
  password2:
    required:
      msg:    Please retype the password
  sfCompareValidator:
    check:    password1
    compare_error: The two passwords do not match
```

`check` 参数是一个域的名字，当前域必须与该域匹配才有效。

Propel 唯一性验证器

`sfPropelUniqueValidator` 检测一个请求参数的值是否在数据库中已经存在。这对于唯一索引非常有用。

```
fields:
  nickname:
    sfPropelUniqueValidator:
      class:    User
      column:   login
```

unique_error: This login already exists. Please choose another one.

本例中，验证器将在数据库中查找类 User 的记录，以确定是否有和域值相同的 login 列。

文件验证器

sfFileValidator 用于检测文件上传域的文件格式（一组 mime 类型）和大小。

fields:

image:

required:

msg: Please upload an image file

file: True

sfFileValidator:

mime_types:

- 'image/jpeg'
- 'image/png'
- 'image/x-png'
- 'image/pjpeg'

mime_types_error: Only PNG and JPEG images are allowed

max_size: 512000

max_size_error: Max size is 512Kb

注意 file 属性必须设置为 True，并且模式中的表单也必须是 multipart 的。

回调验证器

sfCallbackValidator 可以利用第三方的可调用方法或函数进行验证，该可调用方法或函数必须能返回 true 或 false 值。

fields:

account_number:

sfCallbackValidator:

callback: is_integer

invalid_error: Please enter a number.

credit_card_number:

sfCallbackValidator:

callback: [myTools, validateCreditCard]

invalid_error: Please enter a valid credit card number.

回调方法或函数的第一个参数是需要验证的值。当你想要重用现成的方法或函数时非常有用，可以避免重建一个完整的验证器类。

TIP 你也可以编写你自己的验证器，本章后面的“创建定制的验证器”将具体描述。

具名验证器

如果你预计会重复使用一个验证器类以及有关设置，你可以将它们打包成一个具名验证器。在联系人表单的例子中，email 域要用到和 name 域相同的 `sfStringValidator` 参数，这样你就可以创建一个 `myStringValidator` 具名验证器，以免将所有设置重复一遍。你可以在 `validators` 头下增加一个 `myStringValidator` 标签，并将 `class` 和 `param` 属性设置为你需要的参数值。然后你就可以在 `fields` 中象使用标准验证器一样使用该具名验证器。参见例 10—28。

例 10—28 在验证文件中重用具名验证器， 文件路径为 `validate/send.yml`

```
validators:
  myStringValidator:
    class: sfStringValidator
    param:
      min:          2
      min_error:    This field is too short (2 characters minimum)
      max:          100
      max_error:    This field is too long (100 characters maximum)

fields:
  name:
    required:
      msg:          The name field cannot be left blank
    myStringValidator:
  email:
    required:
      msg:          The email field cannot be left blank
    myStringValidator:
    sfEmailValidator:
      email_error:  This email address is invalid
```

重新指定验证方法

默认情况下， 只要一个动作伴随着 POST 方式被调用， 则验证文件中设定的验证器都会执行。为了对不同的方式（POST 或 GET）设定不同的验证，你可以为 `methods` 属性指定其他的值，既可以是全局范围的也可以是针对每个域重新设置调用验证的方式（POST 或 GET）。 参见例 10—29 所示。

例 10—29 在 `validate/send.yml` 中定义何时测试一个域

```

methods:      [post]      # 这是默认值

fields:
  name:
    required:
      msg:      The name field cannot be left blank
    myStringValidator:
  email:
    methods:    [post, get] # 重新指定全局方式
    required:
      msg:      The email field cannot be left blank
    myStringValidator:
    sfEmailValidator:
      email_error: This email address is invalid

```

验证文件全貌

到现在为止，你只是见到了验证文件的若干片段。等你将这些全部合到一起，你就知道如何用 YAML 清晰地描述验证规则了。例 10—30 是联系人表单的完整验证文件，它综合了前面定义的所有规则。

例 10—30 完整的验证文件示例

```

fillin:
  enabled:      true

validators:
  myStringValidator:
    class: sfStringValidator
    param:
      min:      2
      min_error: This field is too short (2 characters minimum)
      max:      100
      max_error: This field is too long (100 characters maximum)

fields:
  name:
    required:
      msg:      The name field cannot be left blank
    myStringValidator:
  email:
    required:
      msg:      The email field cannot be left blank
    myStringValidator:
    sfEmailValidator:

```

```
email_error: This email address is invalid
age:
  sfNumberValidator
    nan_error: Please enter an integer
    min: 0
    min_error: "You're not even born. How do you want to send a
message?"
    max: 120
    max_error: "Hey, grandma, aren't you too old to surf on the
Internet?"
  message:
    required:
      msg: The message field cannot be left blank
```

复杂的验证

利用验证文件已经能够满足大多数验证方面的需求，但是当验证非常复杂时，仅仅使用验证文件就不能满足需要了。这时，你既可以自己编写动作的 `validateXXX()` 方法，也可以从下面叙述的方法中找到解决方案。

创建一个定制的验证器

每个验证器都是一个继承了 `SfValidator` 类的子类。如果 `symfony` 自带的验证器类不符合你的需要，你可以在任何一个可以自动加载的 `lib/` 目录中创建一个新的验证器类。语法相当简单：执行验证器就是执行类的 `execute()` 方法。你还可以在 `initialize()` 方法中定义默认设置。

`execute()` 方法对第一个参数的值进行验证，然后输出第二个参数的值作为出错信息。这两个参数都以引用方式传送，所以你可以在方法内部修改出错信息的内容。

`initialize()` 方法以上下文（`context singleton`）和 `YAML` 文件中的参数数组为参数。定义 `initialize()` 时，必须先调用父类 `SfValidator` 的 `initialize()` 方法，然后才能设置默认值。

每个验证器都有一个参数存取器，它可以用 `$this->getParameterHolder()` 存取。

例如，如果你想创建一个 `SfSpamValidator` 检测一个字符串是否是一个垃圾信息串，可以在 `SfSpamValidator.class.php` 中加入如例 10—31 那样的代码，它可以检测出 `$values` 中出现字符串 `http` 的次数是否超过由 `max_url` 属性定义的次数。

例 10—31 创建一个定制的验证器 `lib/SfSpamValidator.class.php`

```

class sfSpamValidator extends sfValidator
{
    public function execute (&$value, &$error)
    {
        // 当 max_url=2 时, regexp 是 /http.*http/is
        $re = '/'.implode('.', array_fill(0, $this->getParameter('max_url') + 1, 'http')).'/is';

        if (preg_match($re, $value))
        {
            $error = $this->getParameter('spam_error');

            return false;
        }

        return true;
    }

    public function initialize ($context, $parameters = null)
    {
        // 初始化父类
        parent::initialize($context);

        // 设定参数默认值
        $this->setParameter('max_url', 2);
        $this->setParameter('spam_error', 'This is spam');

        // 设置参数
        $this->getParameterHolder()->add($parameters);

        return true;
    }
}

```

一旦将验证器加入可自动加载的目录中（需清除缓存），你就可以在你的验证文件中使用它，见例 10—32 所示。

例 10—32 在 validate/send.yml 中使用定制的验证器

```

fields:
  message:
    required:
      msg:          The message field cannot be left blank
    sfSpamValidator:

```

```
max_url:      3
spam_error:   Leave this site immediately, you filthy spammer!
```

用数组表示表单域

在 PHP 中，你可以将数组用于表单域。当你编写你自己的表单或用由 Propel 后台管理模块生成的表单（参见第 14 章）时，你可以利用例 10—33 中所示的代码。

例 10—33 使用数组的表单

```
<label for="story[title]">Title:</label>
<input type="text" name="story[title]" id="story[title]"
value="default value"
size="45" />
```

在验证文件中使用一个带有方括号的输入名会导致编译错误。解决的方法是在验证文件的 `fields` 段中用花括号 `{}` 替代方括号 `[]`，`symfony` 会自行转换名字后再传送给验证器。参见例 10—34。

例 10—34 使用数组的表单的验证文件

```
fields:
  story{title}:
    required:      Yes
```

验证空域

一个域不一定有值，但是你却可能需要验证这个域是否有一个空值。比如说，在一个表单中有一个密码域，用户可以不改变密码；也可以重设密码，这时，用户还必须输入一个确认密码。参见例 10—35 所示。

例 10—35 具有两个密码域的表单的验证文件

```
fields:
  password1:
  password2:
    sfCompareValidator:
      check:      password1
      compare_error: The two passwords do not match
```

该验证过程按如下方式处理：

- 如果 `password1` 和 `password2` 都为空值：

- 值存在测试通过。
 - 不运行验证器。
 - 表单有效。
- 如果 password2 为空， 而 password1 不为空：
 - 值存在测试通过。
 - 不运行验证器。
 - 表单有效。

你可能希望在 password1 非空时能运行你的 password2 验证器。利用 group 参数， symfony 验证器可以处理这种情形。当一个域在一个组中时，如果这个域不为空且同一个组中的任一个域不为空，这个域的验证器就会执行。

所以，如果你像例 10—36 那样改变你的配置，验证过程就能正确执行。

例 10—36 带有两个密码域和一个组的表单的验证文件

```
fields: c
  password1:
    group:          password_group
  password2:
    group:          password_group
  sfCompareValidator:
    check:          password1
    compare_error:  The two passwords do not match
```

现在验证器按下述方式执行：

- 如果 password1 和 password2 都为空：
 - 值存在测试通过。
 - 验证器未执行。
 - 表单有效。
- 如果 password1 为空而 password2 为 foo：
 - 值存在测试通过。
 - 因为 password2 不为空， 所以将运行验证器， 验证失败。
 - password2 的验证将抛出一个出错信息。
- 如果 password1 为 foo 而 password2 为空：
 - 值存在测试通过。
 - 同理，因为 password1 不为空，所以同一个组中的 password2 的验证器将运行，且验证失败。
 - password2 的验证将抛出一个出错信息。
- 如果 password1 和 password2 都为 foo：
 - 值存在测试通过。
 - 因为 password2 不为空，所以将执行验证器，验证成功。
 - 表单有效。

总结

有了标准表单辅助函数及其灵活的选项，编写表单将非常方便。如果你要设计一个可以编辑对象属性的表单时，对象表单辅助函数将会提供更大的帮助。借助于验证文件，验证辅助函数和重填特性，在表单域上编写一个强壮且用户友好的服务器控件所需的工作量将大大减少。而要满足最复杂的验证需求，只需写一个定制的验证器或在动作类中创建一个 `validateXXX()` 方法。

第 11 章 集成 Ajax

在 Web2.0 的应用中，经常碰到诸如在客户端交互作用、复杂视觉效果及异步通讯之类的问题，这些功能都要用 javascript 去实现。但是，用 javascript 编程是很繁琐且难以测试的。为了自动完成许多要用 javascript 编写的常用功能，symfony 在模板中提供了一组完整的辅助函数。许多客户端功能甚至连一行 javascript 语句都不用写就可以开发出来，开发人员只需要考虑他们想要达到的效果，而复杂的语法和兼容性问题则交给 symfony 去处理。

本章介绍以下内容，这些内容将会帮助你了解如何用 symfony 提供的工具来编写客户端代码：

- 在 symfony 模板中，基本的 javascript 辅助函数输出与标准兼容的 `<script>` 标记，用于更新 DOM（文档对象模型）元素或用链接触发一段代码。
- symfony 中集成了一个 javascript 库 Prototype，它为 javascript 核心增加了新的函数和方法，以便加速客户端代码的开发。
- ajax 辅助函数让用户可以通过点击一个链接，提交一个表单或修改一个表单元素来更新一个页面的某些部分。
- 辅助函数的选项可以提供更大的灵活性和能力，回调函数的运用就是最典型的例子。
- symfony 中还集成了另一个 javascript 库 Script.aculo.us，利用它可以增强动态视觉效果，最终提高用户体验。
- JSON（Javascript 对象标识）是一种用于在客户端和服务端相互通信的标准。
- 结合了前述各种技术的复杂的客户端交互，都可以在 symfony 的应用中加以实现。只要写一行 PHP 语句去调用一个 symfony 辅助函数，就可以实现诸如自动完成、拖拽、可排序列表及可编辑文本等功能。

基本的 javascript 辅助函数

过去，由于 javascript 存在浏览器兼容性问题，所以在专业的 web 应用中很少用到它。但是现在，兼容性问题已经基本解决，利用许多质量稳定的 javascript 库，无需写大量代码和进行耗时的测试，就可以开发出复杂的 javascript 交互应用。ajax 就是进展最快和最普及的 javascript 应用，本章后面的“ajax 辅助函数”将会介绍它。

你可能会奇怪为什么在这一章里只有极少的 javascript 代码。这正是 symfony 的独特之处，因为 symfony 已经将 javascript 行为封装和抽象进辅助函数中了，所以在你的模板中可以完全不用编写 javascript 代码。开发人员只需要用一行 PHP 语句就可以为页面中的一个元素增加一种行为，但是被调用的辅助函数会输出 javascript 代码，只要分析一下生成的响应就可以揭示出封装的复杂

性，因为辅助函数处理了包括浏览器一致性、复杂的限制条件和扩展性等问题，所以 javascript 代码的总量非常重要。本章将告诉你怎样不使用 javascript 语句达到用 javascript 才能产生的效果。

这里介绍的所有辅助函数都用于模板，只要你事先声明使用 Javascript 辅助函数即可。

```
<?php use_helper('Javascript') ?>
```

你很快会看到，一些辅助函数只输出 HTML 代码，而另一些会输出 javascript 代码。

模板中的 javascript

在 XHTML 中，javascript 代码块必须包含在 CDATA 声明中。但是如果在页面中需要写多个 javascript 代码块时，这样做就变得非常繁琐。所以 symfony 提供了 javascript_tag() 辅助函数，用于将一个字符串转化为一个符合 XHTML 规范的 <script> 标记。例 11-1 是这个辅助函数的例子：

例 11-1 用 javascript_tag() 辅助函数插入 javascript 代码

```
<?php echo javascript_tag("
    function foobar()
    {
        ...
    }
") ?>
=> <script type="text/javascript">
    //
        function foobar()
        {
            ...
        }
    //]]&gt;
&lt;/script&gt;</pre></div><div data-bbox="144 772 850 809" data-label="Text"><p>但是 javascript 的最主要应用并不是编写代码块，而是用超链接去触发某个具体脚本。例 11-2 显示了 link_to_function() 辅助函数的使用。</p></div><div data-bbox="144 825 851 862" data-label="Text"><p>例 11-2 利用 link_to_function() 辅助函数，一个链接可以触发 javascript 代码。</p></div>
```

```
<?php echo link_to_function('Click me!', "alert('foobar')") ?>
=> <a href="#" onClick="alert('foobar'); return none;">Click me!</a>
```

和 link_to() 辅助函数一样，第三个参数可以为标记加入选项。

NOTE 与 link_to() 相似的还有 button_to()，你可以调用 button_to_function() 辅助函数通过一个按钮(<input type="button">) 触发 javascript 代码。如果你还想要一个可点击图片，只要调用 link_to_function(image_tag("myimage"), "alert('foobar')") 即可。

更新一个 DOM 元素

更新页面中的一个元素是动态界面经常要解决的问题。例 11-3 是为此而经常编写的代码。

例 11-3 用 javascript 更新一个元素

```
<div id="indicator">Data processing beginning</div>
<?php echo javascript_tag(
    document.getElementById("indicator").innerHTML =
        "<strong>Data processing complete</strong>";
    ')?>
```

symfony 为此提供了一个名为 update_element_function() 的辅助函数，用于生成 javascript 代码而不是 html 代码。例 11-4 是一个示例。

例 11-4 在 javascript 代码块中用 update_element_function() 辅助函数更新一个元素

```
<div id="indicator">Data processing beginning</div>
<?php echo javascript_tag(
    update_element_function("indicator", array(
        "content" => "<strong>Data processing complete</strong>",
    ))
)?>
```

你也许在想：这条辅助函数语句和真正的 javascript 代码一样长，那使用它有什么特别的好处呢？好处在于它的可读性。例如，如果你想根据某种条件在一个元素之前或之后插入内容，或者不是更新元素内容而是删除一个元素，或者不做任何处理时，javascript 代码将会变得相当混乱，但是利用 update_element_function()，却可以像例 11-5 那样保持清晰易读。

例 11-5 update_element_function() 辅助函数的选项

```
// 在 indicator 元素之后插入内容
update_element_function('indicator', array(
    'position' => 'after',
    'content'  => "<strong>Data processing complete</strong>",
));

// 如果$condition 成立，则删除 indicator 之前的元素
update_element_function('indicator', array(
    'action'    => $condition ? 'remove' : 'empty',
    'position' => 'before',
));
```

可以看出，这个辅助函数让你的模板比任何 javascript 代码都要容易理解，而且你只要使用一种语法就可以处理相似的行为。这也是为什么这个辅助函数名字这样长的原因：无需额外的说明，它就可以充分地解释自身的用途。

轻松地降级 (Graceful Degradation)

用<noscript> 指明的 html 代码，仅在不支持 javascript 的浏览器中显示。symfony 用一个辅助函数从另一方面进行补充这个功能，也就是利用它可以让代码仅在支持 javascript 的浏览器中执行。例 11-6 中，显示了用 if_javascript() 和 end_if_javascript() 实现这种降级的用法：

例 11-6 利用 if_javascript() 辅助函数轻松地降级

```
<?php if_javascript(); ?>
    <p>You have JavaScript enabled.</p>
<?php end_if_javascript(); ?>

<noscript>
    <p>You don't have JavaScript enabled.</p>
</noscript>
```

NOTE 调用 if_javascript() 和 end_if_javascript() 时，不需要用 echo.

Prototype

Prototype 是一个优秀的 javascript 库，它扩展了客户端脚本的能力，增加了开发者想要的功能，并且提供了新的机制去操作 DOM，该项目的网站是 <http://prototypejs.org/>。

symfony 框架中绑定了 Prototype 文件，在每个项目的 web/sf/prototype 目录中可以找到它的文件。这样只要在你的 action 中增加下列代码就可以使用 Prototype:

```
$prototypeDir = sfConfig::get('sf_prototype_web_dir');  
$this->getResponse()->addJavascript($prototypeDir.'/js/prototype');
```

或者在 view.yml 文件中加入:

all:

```
javascripts: [%SF_PROTOTYPE_WEB_DIR%/js/prototype]
```

NOTE 因为 symfony Ajax 辅助函数(下一节介绍)需要用到 Prototype, 所以只要你用到 Prototype 库, 它就会自动被包含进来。也就是说, 如果你的模板调用一个_remote 辅助函数, 你无需在你的响应里手工添加 Prototype Javascript。

一旦你载入了 Prototype 库, 就可以利用它为 javascript 核心增加的新函数。本书的主要目的不是讲述这些函数, 你可以很容易在互联网上找到所需的文档, 以下是几个有关的网站:

- Particletree: <http://particletree.com/features/quick-guide-to-prototype/>
- Sergio Pereira: <http://www.sergiopereira.com/articles/prototype.js.html>
- Script.aculo.us: <http://wiki.script.aculo.us/scriptaculous/show/Prototype>

Prototype 新增的 Javascript 函数之一是\$()函数。简单地说, 这个函数可以看成是 document.getElementById()函数的缩写, 但它有更强的功能。例 11-7 给出了一个应用的例子。

例 11-7 利用\$()函数根据 DOM 的元素 ID 取得元素值。

```
node = $('elementID');
```

```
// 相当于
```

```
node = document.getElementById('elementID');
```

```
// 也可以一次检索多个元素
```

```
// 在本例中返回值是一个由 DOM 元素组成的数组。
```

```
nodes = $('firstDiv', 'secondDiv');
```

Prototype 还提供了 javascript 核心真正缺乏的一个方法，这个方法返回所有 CSS className 属性等于它接收的参数的 DOM 元素组成的数组：

```
nodes = document.getElementsByClassName('myclass');
```

当然你不太会用到这个函数，因为 Prototype 提供了一个更强大的 \$\$() 函数。这个函数根据 CSS 选择器返回一个由 DOM 元素组成的数组。所以前面的调用可以写成如下形式：

```
nodes = $$('.myclass');
```

凭借 CSS 选择器的作用，你可以根据 class、ID、父子关系和前后关系去解析 DOM，这比通过 XPath 表达式去解析更为简单。你甚至可以用一个混合了所有这些选择器的复杂选择器去访问对应的元素。

```
nodes = $$('body div#main ul li:last img > span.legend');
```

Prototype 增强 Javascript 语法功能的最后一个例子是数组迭代。它为 Javascript 定义匿名函数和闭包（closure）（译者注：如果在一个 javascript 函数体中又出现一个函数定义时，称此函数为闭包（closure））功能提供了和 PHP 同样的简化形式。如果你编写 javascript 代码，可能会大量用到这个功能。

```
var vegetables = ['Carrots', 'Lettuce', 'Garlic'];  
vegetables.each(function(food) { alert('I love ' + food); });
```

因为用 Prototype 编写 Javascript 比纯手工编写更为有趣，并且因为 Prototype 也是 symfony 的一个组成部分，所以你应该花些时间去研究它的文档。

Ajax 辅助函数

如果你想在服务器端用 PHP 脚本去更新页面中的元素内容，而不想用 javascript 去更新（如例 11-5 所示），这样你可以根据一个服务器的响应来改变页面的某个部分，那该怎么做呢？remote_function() 辅助函数就可以完成这个任务，如例 11-8 所示：

例 11-8 应用 remote_function() 辅助函数

```
<div id="myzone"></div>  
<?php echo javascript_tag(
```



```
remote_function(array(
    'update' => 'myzone',
    'url'     => 'mymodule/myaction',
))
)?>
```

NOTE url 参数既可以包含一个内部 URI (module/action?key1=value1&...), 也可以包含一个路由规则名, 就像在一个标准的 url_for() 中那样。

当你调用这个函数时, 这段脚本就会根据 mymodule/myaction 动作的请求或响应, 去更新 id 为 myzone 的元素。这种交互就是 Ajax, 也正是高度可交互的 web 应用的核心。Wikipedia (<http://en.wikipedia.org/wiki/AJAX>) 描述了 Ajax 的特点:

Ajax 让页面只和服务器交换很少量的数据, 因而每当用户改变页面时, 不需要重新导入整个网页, 而使得页面的响应更快。也就是说增强了网页的交互性, 速度和可用性。

Ajax 依赖于 javascript 对象 XMLHttpRequest, 该对象的行为如同一个隐藏的帧, 你可以从一个服务器请求更新它并且重用它去操纵页面的剩余部分。这是一个相当底层的对象, 不同的浏览器用不同的方法去处理它。所幸的是, Prototype 封装了所有 Ajax 需要的代码并提供了一个更为简化的 Ajax 对象, symfony 就借助于这个对象。这也是为什么当你在一个模板中使用 Ajax 辅助函数时, Prototype 就会自动装入的原因。

CAUTION 如果远程动作的 URL 不属于当前页所在的域, Ajax 辅助函数将不工作。这既是出于安全考虑, 也受到浏览器禁止远程动作通过的限制。

一个 Ajax 交互由三个部分组成: 一个调用者 (链接、按钮、表单、时钟或其它用户可以操纵以启动动作的任何控件), 一个服务器动作和页面中的一个显示动作响应的区域。如果远程动作返回的数据还要被客户端的 Javascript 函数处理, 你可以创建更复杂的交互。symfony 提供了多个名字中包含 remote 的辅助函数, 以便你在模板中插入 Ajax 交互。它们使用共同的语法, 可以将所有的 Ajax 参数放进一个关联数组中。注意, Ajax 辅助函数输出的是 HTML, 而不是 Javascript。

SIDEBAR Ajax 动作如何工作?

远程函数被调用的动作就是一个通常的动作。与其它动作一样, 它们可以被路由, 可以确定用哪个视图提交它们返回的响应, 也可以向模板传递参数以及改变模型等。

但是, 当通过 Ajax 调用动作时, 动作将返回 true 给下面的调用:

```
$i sAjax = $this->isXmlHttpRequest();
```

symfony 知道动作处于 Ajax 环境中，因而能够对响应做相应的处理。因此，在默认情况下，开发环境中的 Ajax 动作不包含 web 调试工具栏，而且也会跳过装饰处理（默认情况下，模板不会被包含在布局中）。如果你想装饰 Ajax 的视图，你需要在模块的 `view.yml` 文件中，为这个视图明确设置 `has_layout` 的值为 `true`。

还有一点：因为响应性在 Ajax 的交互中至关重要，所以如果响应不是非常复杂，最好不要创建视图，而是直接从动作返回响应。这样你可以在动作中用 `renderText()` 方法，直接跳过模板而加速 Ajax 请求。

Ajax 链接

在 Web 2.0 应用中，Ajax 链接是 Ajax 交互应用的主要内容。显而易见，`link_to_remote()` 辅助函数就可以输出一个调用远程函数的链接。除了第二个参数是一个由 Ajax 选项组成的关联数组以外，其他语法类似于 `link_to()`。例 11-9 是一个示例。

例 11-9 利用 `link_to_remote()` 辅助函数得到 Ajax 链接

```
<div id="feedback"></div>
<?php echo link_to_remote('Delete this post', array(
    'update' => 'feedback',
    'url'     => 'post/delete?id=' . $post->getId(),
)) ?>
```

在这个例子中，点击 `Delete this post` 链接就会在后台发出一个对 `post/delete` 的调用。从服务器返回的响应将出现在 `id` 为 `feedback` 的元素中。图 11-1 显示了运行的过程。

图 11-1 用链接触发一个远程更新



对于链接，你可以用图片代替字符串，用规则名代替内部的模块/动作 URL，还可以将选项加进标记的第三个参数中。如例 11-10 所示。

例 11-10 `link_to_remote()` 辅助函数的选项

```

<div id="emails"></div>
<?php echo link_to_remote(image_tag('refresh'), array(
    'update' => 'emails',
    'url'     => '@list_emails',
), array(
    'class' => 'ajax_link',
)) ?>

```

Ajax 驱动的表单

Web 表单一般要调用另一个动作，但这会导致整个页面被刷新。对表单来说，类似于 `link_to_function()` 可以在表单提交后，仅用服务器的响应去更新页面中的一个元素，`form_remote_tag()` 辅助函数就是完成这个任务的，例 11-11 展示了它的语法：

例 11-11 利用 `form_remote_tag()` 辅助函数得到 Ajax 表单

```

<div id="item_list"></div>
<?php echo form_remote_tag(array(
    'update' => 'item_list',
    'url'     => 'item/add',
)) ?>
<label for="item">Item:</label>
<?php echo input_tag('item') ?>
<?php echo submit_tag('Add') ?>
</form>

```

就像 `form_tag()` 辅助函数一样，`form_remote_tag()` 也打开一个 `<form>`。提交这个表单会在后台向 `item/add` 动作发出一个 POST 请求，请求参数就是 `item` 字段的内容。响应会替换 `item_list` 元素的内容（如图 11-2 所示）。最后用通常的 `</form>` 标记关闭 Ajax 表单。

图 11-2 利用表单触发远程更新。



CAUTION 由于 `XMLHttpRequest` 对象的限制，Ajax 表单不可以分为多个部分。这意味着你不能通过 Ajax 表单上传文件。不过你可以用其他方法解决这个问题。

题—比如，用隐式的 `iframe` 代替 `XMLHttpRequest`（参看 <http://www.air4web.com/files/upload/> 给出的一个实现）。

如果你想让一个表单同时工作在页面模式和 Ajax 模式，最好的方法是定义一个通常的表单，然后除了提供通用的提交按钮，再增加一个按钮（`<input type="button"/>`）用于以 Ajax 方式提交表单。Symfony 用 `submit_to_remote()` 调用这个按钮。这可以帮助你建立一个可以轻松降级的 Ajax 交互表单，即可以与不支持 Javascript 的浏览器兼容。见例 11-12 所示。

例 11-12 具有普通提交方式和 Ajax 提交方式的表单

```
<div id="item_list"></div>
<?php echo form_tag('@item_add_regular') ?>
  <label for="item">Item: </label>
  <?php echo input_tag('item') ?>
  <?php if_javascript(); ?>
    <?php echo submit_to_remote('ajax_submit', 'Add in Ajax', array(
      'update' => 'item_list',
      'url'     => '@item_add',
    )) ?>
  <?php end_if_javascript(); ?>
  <noscript>
    <?php echo submit_tag('Add') ?>
  </noscript>
</form>
```

另一个混合了普通提交和 Ajax 提交标记的例子是编辑文章的表单。它可以提供一个实现了 Ajax 的预览按钮和一个用普通提交实现的发布按钮。

NOTE 当用户按下回车键时，表单会用定义在主 `<form>` 标记中的动作去提交，在本例中，就是普通提交动作。

现代表单不仅仅在提交时作出回应，在用户改变某个域的值时也会有回应。在 Symfony 中，你可以用 `observe_field()` 辅助函数来实现这个功能。例 11-13 应用这个辅助函数建立一个具有建议特性的页面，也就是在 `item` 字段中每输入一个字符，都会触发一次 Ajax 调用去刷新页面中的 `item_suggestion` 元素。

例 11-13 当字段值变化时，`observe_field()` 调用一个远程函数

```
<?php echo form_tag('@item_add_regular') ?>
  <label for="item">Item: </label>
  <?php echo input_tag('item') ?>
  <div id="item_suggestion"></div>
  <?php echo observe_field('item', array(
```

```

        'update'    => 'item_suggestion',
        'url'       => '@item_being_typed',
    )) ?>
    <?php echo submit_tag('Add') ?>
</form>,

```

每当用户改变字段 `item`（称为“发现域”）的值时，即使没有提交表单，也会调用记录在 `@item_being_typed` 规则中的模块/动作。这个动作将从 `value` 请求参数中获得当前的 `item` 值。如果你想传递非发现域的值，你可以在 `with` 选项 中用 `javascript` 表达式来指明。例如，如果你想让动作得到 `param` 参数，可以将 `observe_field()` 辅助函数写成例 11-14 中的形式：

例 11-14 用 `with` 选项将你自己的参数传递给远程动作

```

<?php echo observe_field('item', array(
    'update'    => 'item_suggestion',
    'url'       => '@item_being_typed',
    'with'      => "'param=' + value",
)) ?>

```

注意，这个辅助函数并不输出一个 HTML 元素，而是输出作为参数传递的元素的 行为。在本章中你会看到更多用 `javascript` 辅助函数指定行为的例子。

如果你想发现一个表单中的所有域，你可以使用 `observe_form()` 辅助函数， 每当表单中的任意一个域发生变化时，它都会调用一个远程动作。

周期性调用远程函数

`symfony` 还有一个 `periodically_call_remote()` 辅助函数用于每隔几秒钟触发 一次 `Ajax` 交互。它并不与某个 HTML 控件结合，而是作为整个页面的一个行为 在后台透明地运行。在实现追踪鼠标位置、自动保存大块文本输入区内容等功 能时，这个函数非常有用。例 11-15 是一个示例。

例 11-15 用 `periodically_call_remote()` 周期性调用一个远程函数

```

<div id="notification"></div>
<?php echo periodically_call_remote(array(
    'frequency' => 60,
    'update'    => 'notification',
    'url'       => '@watch',
    'with'      => "'param=' + $('mycontent').value",
)) ?>

```

如果你不想指明两次调用远程函数之间的间隔秒数(frequency)，则默认值是 10 秒。注意，with 参数是 javascript 计算出来的，所以你可以在其中用 Prototype 函数，比如\$()。

远程调用参数

除了参数 update 和 url 以外，前面介绍的所有 Ajax 辅助函数还可以带另外一些参数。由 Ajax 参数组成的关联数组可以调整 and 改变远程调用的行为和对它们的响应的处理。

根据响应状态更新确定的元素

如果远程动作失败了，远程辅助函数可以选择更新另一个元素，而不是更新由成功的响应更新的元素，为了达到这个目的，只要将 update 参数的值分开存放在一个关联数组中，并且为 success 和 failure 两种情况设置要更新的不同的元素即可。如果一个页面有许多 Ajax 交互和一个错误反馈区，就可以利用这个功能。例 11-16 展示了这种有条件更新的用法。

例 11-16 处理一个有条件更新

```
<div id="error"></div>
<div id="feedback"></div>
<p>Hello, World!</p>
<?php echo link_to_remote('Delete this post', array(
    'update' => array('success' => 'feedback', 'failure' => 'error')
    'url' => 'post/delete?id=' . $post->getId(),
)) ?>
```

TIP 只有 HTTP 错误代码(500, 400 及所有不在 2XX 范围内的代码)才会触发失败更新，返回 sfView::ERROR 的动作并不会触发失败更新。所以如果你想写一个返回 Ajax 失败的动作，你必须调用类似\$this->getResponse()->setStatusCode(404)的函数。

根据元素位置更新元素

通过加入 position 参数，你可以更新相对于某个具体元素位置的元素，就像 update_element_function() 辅助函数一样。例 11-17 是一个示例。

例 11-17 用位置参数改变响应位置

```
<div id="feedback"></div>
<p>Hello, World!</p>
<?php echo link_to_remote('Delete this post', array(
```

```

        'update'    => 'feedback',
        'url'       => 'post/delete?id=' . $post->getId(),
        'position'  => 'after',
    )) ?>

```

这个例子将在 `feedback` 元素之后插入 Ajax 调用的响应，也就是在 `<div>` 和 `<p>` 之间。有了这个方法，你可以调用多个 Ajax 调用，并在 `update` 参数指定的元素之后聚集所有的响应。

`position` 参数可以取以下值：

- `before`：在元素之前
- `after`：在元素之后
- `top`：在元素的内容顶部
- `bottom`：在元素的内容底部

根据条件更新元素

远程调用还可以设置 `confirm` 参数，以便在真正发出 `XMLHttpRequest` 之前用户可以确认。如例 11-18 所示：

例 11-18 在远程函数中加入确认参数以便在调用之前取得确认

```

<div id="feedback"></div>
<?php echo link_to_remote('Delete this post', array(
    'update'    => 'feedback',
    'url'       => 'post/delete?id=' . $post->getId(),
    'confirm'   => 'Are you sure?',
)) ?>

```

这样，当用户点击了链接后，就会弹出一个显示了“Are you sure?”的 javascript 对话框，只有当用户点击 OK 表示确认后，`post/delete` 动作才会被调用。

如果你还设置了 `condition` 参数，远程调用将根据在浏览器端执行的条件测试来确定是否被执行，例 11-19 给出一个例子：

例 11-19 根据客户端的测试来确定是否调用远程函数

```

<div id="feedback"></div>
<?php echo link_to_remote('Delete this post', array(
    'update'    => 'feedback',
    'url'       => 'post/delete?id=' . $post->getId(),

```

```
        'condition' => "$('elementID') == true",
    )) ?>
```

确定 Ajax 请求方法

Ajax 请求默认采用 POST 方法。如果你调用一个并不改变数据的 Ajax 函数，或者，你想显示一个含有内置验证方法的表单作为 Ajax 调用的结果，你可能想将 Ajax 请求方法改为 GET。例 11-20 显示了用 method 选项改变 Ajax 请求方法的例子。

例 11-20 改变 Ajax 的请求方法

```
<div id="feedback"></div>
<?php echo link_to_remote('Delete this post', array(
    'update'      => 'feedback',
    'url'         => 'post/delete?id=' . $post->getId(),
    'method'      => 'get',
)) ?>
```

授权脚本运行

如果一个 Ajax 函数的响应代码（即插入在 update 元素中的由服务器发出的代码）中含有 Javascript，在缺省情况下这些代码并不会执行。这样做的目的是为了减少远程攻击的风险，并且只有在开发者确认了响应中的代码后才执行脚本。

这样为了能执行远程响应中的脚本，你需要设置 script 参数来明确声明可以执行。例 11-21 的 Ajax 调用指明了可以执行从远程响应中得到的 Javascript。

例 11-21 授权执行 Ajax 响应中的脚本

```
<div id="feedback"></div>
// 如果 post/delete 动作的响应中含有 JavaScript，
// 允许其在浏览器中执行。
<?php echo link_to_remote('Delete this post', array(
    'update' => 'feedback',
    'url'    => 'post/delete?id=' . $post->getId(),
    'script' => true,
)) ?>
```

如果远程模板包含像 remote_function() 之类的 Ajax 辅助函数，要记住这些 PHP 函数会生成 Javascript 代码，如果你不加上 'script' => true 选项，这些 Javascript 代码就不会被执行。

NOTE 对于远程响应，虽然你允许脚本执行，但如果你想用某种工具去查看生成的代码，你是看不到远程代码中的脚本的。这些脚本只会执行却不会出现在代码中。虽然看起来这有些奇怪，但这种处理却是很正常的。

创建回调函数

Aj ax 交互的一个严重缺点就是在要更新的区域被更新之前，用户感受不到这种交互。也就是说，如果网络很慢或服务器运行失败，用户以为动作已经被处理了，而实际上，动作根本就没有被执行。所以，将 Aj ax 交互中发生的事件传达给用户是非常重要的。

每个远程请求都被默认为一个异步过程，在这个过程中，可以触发各种 javascript 回调函数，诸如进度条之类。所有的回调函数都可以访问 request 对象，该对象暗含着 XMLHttpRequest。回调函数对应于 Aj ax 交互中发生的以下事件：

- before: 在初始化请求之前
- after: 在初始化请求之后并在导入之前
- loading: 当远程响应正被浏览器导入时
- loaded: 当远程响应被浏览器导入完成时
- interactive: 当用户可以和远程响应交互时，即使该响应还没有完全导入
- success: 当 XMLHttpRequest 已完成，而且 HTTP 状态码在 2XX 的范围内
- failure: 当 XMLHttpRequest 已完成，而且 HTTP 状态码不在 2XX 的范围内
- 404: 当请求返回 404 状态时
- complete: 当 XMLHttpRequest 完成时（如果存在的话，将会在 success 或 failure 之后触发）

例如，我们常会在初始化远程调用时显示一个导入进度条，而在收到响应后隐去进度条。要达到这个目的，只要在 Aj ax 调用中简单地加入 loading 和 complete 参数即可，如例 11-22 所示。

例 11-22 用 Aj ax 回调函数显示和隐藏一个活动进度条

```
<div id="feedback"></div>
<div id="indicator">Loading...</div>
<?php echo link_to_remote('Delete this post', array(
    'update' => 'feedback',
    'url' => 'post/delete?id='.$post->getId(),
    'loading' => "Element.show('indicator')",
    'complete' => "Element.hide('indicator')",
)) ?>
```

show 和 hide 方法以及 Javascript Element 对象都是 Prototype 中有用的工具。

创建视觉效果

symfony 中集成了 script.aculo.us 库的视觉效果，这个库可以帮助你完成不仅仅是在页面中显示和隐藏<div> 元素的功能。在 <http://script.aculo.us/> 中可以了解有关文档。简单地讲，这个库提供了许多为了获得复杂视觉效果而操纵 DOM 的 Javascript 对象和功能。例 11-23 中列出了一些示例。因为结果是 web 页中某个区域的视觉模拟，所以建议你测试一下它们的效果，以便理解它们到底是如何运行的。Script.aculo.us 网站提供了动态效果的汇总，你可以从中选择。

例 11-23 在 Javascript 中用 script.aculo.us 实现视觉效果

```
// 加亮 my_field 元素
Effect.Highlight('my_field', { startcolor: '#ff99ff',
endcolor: '#999999' })
```

```
// 为元素添加下拉百叶窗效果
Effect.BlindDown('id_of_element');
```

```
// 为元素添加渐隐效果
Effect.Fade('id_of_element', { transition:
Effect.Transitions.wobble })
```

symfony 用 visual_effect() 辅助函数封装了 Javascript 的 Effect 对象，这个辅助函数是 Javascript 辅助函数组的成员，它将输出通常的链接会用到的 Javascript 代码，如例 11-24 所示。

例 11-24 用 visual_effect() 辅助函数在模板中实现视觉效果

```
<div id="secret_div" style="display:none">I was here all along!</div>
<?php echo link_to_function(
    'Show the secret div',
    visual_effect('appear', 'secret_div')
) ?>
// 将调用 Effect.Appear('secret_div')
```

visual_effects() 辅助函数还可以用在 Ajax 回调函数中，例 11-25 实现了一个类似例 11-22 的活动进度条，但是视觉感受更舒适。当启动 Ajax 调用时，indicator 元素渐渐显现出来，而当响应到达时，indicator 元素又渐渐隐去。

另外，为了吸引用户注意 feedback 元素，当远程调用更新它以后，它会被加亮显示。

例 11-25 Ajax 回调函数的视觉效果

```
<div id="feedback"></div>
<div id="indicator" style="display: none">Loading...</div>
<?php echo link_to_remote('Delete this post', array(
    'update' => 'feedback',
    'url'     => 'post/delete?id=' . $post->getId(),
    'loading' => visual_effect('appear', 'indicator'),
    'complete' => visual_effect('fade', 'indicator').
                    visual_effect('highlight', 'feedback'),
)) ?>
```

注意观察是如何在回调函数中通过链接方式将各种视觉效果混合起来的。

JSON

Javascript 对象表示模型（JavaScript Object Notation, JSON）是一个轻量级的数据交换格式。简单地讲，它就是用于传送对象信息的 Javascript 数组，如例 11-26 中的例子所示。它对 Ajax 交互来说有两个主要的好处，一个是 Javascript 易于读取它，二是它可以减少 web 响应的数据量。

例 11-26 Javascript 中的 JSON 对象

```
var myJsonData = {"menu": {
    "id": "file",
    "value": "File",
    "popup": {
        "menuItem": [
            {"value": "New", "onclick": "CreateNewDoc()"},
            {"value": "Open", "onclick": "OpenDoc()"},
            {"value": "Close", "onclick": "CloseDoc()"}
        ]
    }
}}
```

如果一个 Ajax 动作需要返回结构化的数据给调用页面，以便 javascript 进一步处理的话，JSON 是一种比较适合作为响应的格式。例如，当一个 Ajax 调用想更新调用页面的多个元素时，JSON 就非常有用。

设想有一个像例 11-27 那样的调用页面，其中含有两个需要更新的元素。一个远程辅助函数只能更新页面中的一个元素（要么是 title，要么是 name），而不能同时更新两个。

例 11-27 用于多个 Ajax 更新的模板示例

```
<h1 id="title">Basic letter</h1>
<p>Dear <span id="name">name_here</span>, </p>
<p>Your e-mail was received and will be answered shortly.</p>
<p>Sincerely, </p>
```

要更新两者，把 Ajax 响应想象成一个包括以下数组的 JSON 头：

```
[["title", "My basic letter"], ["name", "Mr Brown"]]
```

远程调用就能轻易地解析这个响应，并且稍稍利用 Javascript 就可以更新一行中的几个域。为达到这个效果，将例 11-28 中的代码加进例 11-27 的模板即可。

例 11-28 从一个远程响应更新多个元素

```
<?php echo link_to_remote('Refresh the letter', array(
    'url'          => 'publishing/refresh',
    'complete' => 'updateJSON(request, json)'
)) ?>
```

```
<?php echo javascript_tag("
function updateJSON(request, json)
{
    var nbElementsInResponse = json.length;
    for (var i = 0; i < nbElementsInResponse; i++)
    {
        Element.update(json[i][0], json[i][1]);
    }
}
") ?>
```

complete 回调函数可以访问响应的 json 头并把它传递给第三方函数。这个 updateJSON() 函数对 JSON 头进行迭代，对数组的每个成员，用第二个参数设置的元素名称去更新第一个参数设置的元素名称。例 11-29 显示了 publishing/refresh 动作如何返回一个 JSON 响应。

例 11-29 返回 JSON 头的动作示例

```

class publishingActions extends sfActions
{
    public function executeRefresh()
    {
        $output = '["title", "My basic letter"], ["name", "Mr Brown"]';
        $this->getResponse()->setHTTPHeader("X-JSON", '('.$output.')');

        return sfView::HEADER_ONLY;
    }
}

```

HTTP 协议允许在响应头中存放 JSON。因为响应不包含任何内容，动作只是立即将它作为一个头发送出去。这样它就完全跳过了视图层，不仅和 `->renderText()` 一样快，而且更小。

CAUTION 例 11-29 中的方法有一个服务器的限制，即 HTTP 头的最大容量。虽然没有正式的规定，但浏览器有时不能很好地转换和解释大的 HTTP 头。也就是说，如果你的 JSON 数组比较大，那么远程动作应该将 JSON 数组放在一个标准的响应中，而不是将 JSON 数组放在 HTTP 头中。

JSON 已经成为 web 应用的一个标准。Web 服务经常建议用 JSON 而不是 XML 去响应，以便将服务集成到客户端(mashup)而不是服务器端。所以如果你要选择用哪种格式在服务器和 javascript 函数间进行通信，JSON 也许是最好的选择。

TIP 从 PHP5.2 开始，PHP 提供了 `json_encode()` 和 `json_decode()` 两个函数，以便你在 PHP 语法和 JSON 语法之间进行数组转换。这有助于 JSON 数组和 Ajax 的集成，请参考(<http://www.php.net/manual/en/ref.json.php>) 的说明。

用 Ajax 完成复杂的交互

在 symfony 的 Ajax 辅助函数里，有些工具只需一个简单的调用，你就可以完成复杂的交互功能。有了类似桌面应用的交互功能（拖拽，自动完成和实时编辑），你无需编写复杂的 javascript 代码就可以提高用户体验。下面几节将描述这些用于复杂交互的辅助函数，并给出简单的例子。其他的参数和技巧请参考 `script.aculo.us` 的文档。

CAUTION 如果你要提供复杂的交互功能，界面显示将花费更多的时间才能达到自然的效果。只有你确信这样有助于提高用户体验时，才使用复杂的交互功能。否则应该避免使用。

自动完成

用户在一个文本录入控件中输入字符时，如果能列出与用户输入匹配的一些词汇，这就称为自动完成。如果远程动作以例 11-30 那样的 HTML 列表形式返回

响应时，利用 `input_auto_complete_tag()` 辅助函数，你就可以达到这个效果。

例 11-30 能够使用自动完成标记的响应示例

```
<ul>
  <li>suggestion1</li>
  <li>suggestion2</li>
  ...
</ul>
```

你可以仿照例 11-31 中的例子，在进行常规文本输入时，将辅助函数插入模板。

例 11-31 在模板中使用自动完成标记辅助函数

```
<?php echo form_tag('mymodule/myaction') ?>
Find an author by name:
<?php echo input_auto_complete_tag('author', 'default name',
  'author/autocomplete',
  array('autocomplete' => 'off'),
  array('use_style' => true)
) ?>
<?php echo submit_tag('Find') ?>
</form>
```

当用户每次在 `author` 域中输入一个字符，它就会调用 `author/autocomplete` 动作。你可以设计动作，以便根据 `author` 请求参数确定一个可能的匹配列表，并且以类似例 11-30 的格式返回。然后辅助函数就会在 `author` 标记下显示这个列表，当用户点击其中的一个提示或用键盘选择一个提示时，输入就可以完成。图 11-3 是它的图示。

图 11-3 一个自动完成的例子



`input_auto_complete_tag()` 辅助函数的第三个参数可以取以下参数：

- `use_style`: 自动控制响应列表的形式。
- `frequency`: 周期性调用的频率（默认值为 0.4 秒）。

- **tokens**: 开通标记化的递增自动完成功能。例如，如果你设置了这个参数为，而用户输入了 `jane,george`，则动作只接受 `'george'` 值。

NOTE 和其它辅助函数一样，`input_auto_complete_tag()` 辅助函数也接受本章前面描述的常用的远程辅助函数选项。特别是设置 `loading` 和 `complete` 视觉效果，可以得到更好的用户体验。

拖放

在桌面应用中用鼠标将一个元素拖放到某个地方是很常见的，但在浏览器中却极少见。这是因为用 Javascript 编写这些行为是非常复杂的。但是在 symfony 中却可以方便地只用一行代码就能实现。

symfony 框架提供了两个辅助函数：`draggable_element()` 和 `drop_receiving_element()`，它们可以被看成是行为修改器(modifier)，它们为相应的元素加上发现器(observer)和拖动能力，可以声明一个元素是可拖动的或是可拖动元素的释放接收元素。一个可拖动元素就是能用鼠标点击抓取的元素，只要鼠标按钮未松开，这个元素就可以在整个窗口中移动。当这个可拖动元素被释放，一个接收元素就会去调用一个远程函数。例 11-32 显示了用一个购物车接收元素进行交互的例子。

例 11-32 购物车中的可拖放元素和释放接收元素

```
<ul id="items">
  <li id="item_1" class="food">Carrot</li>
  <?php echo draggable_element('item_1', array('revert' => true)) ?>
  <li id="item_2" class="food">Apple</li>
  <?php echo draggable_element('item_2', array('revert' => true)) ?>
  <li id="item_3" class="food">Orange</li>
  <?php echo draggable_element('item_3', array('revert' => true)) ?>
</li>
<div id="cart">
  <p>Your cart is empty</p>
  <p>Drag items here to add them to your cart</p>
</div>
<?php echo drop_receiving_element('cart', array(
  'url'          => 'cart/add',
  'accept'       => 'food',
  'update'       => 'cart',
)) ?>
```

无序列表中的每一项都可以被鼠标拖动并在整个窗口中移动。鼠标松开时，它们将返回开始的位置。如果在 `cart` 元素上松开鼠标，它就触发一个调用 `cart/add` 动作的远程函数。这个动作根据 `id` 请求参数确定该将哪个项目放进

cart 元素。例 11-32 模拟了一次真正的购物会话：你可以抓取所需项目放进购物车，然后结帐。

TIP 在例 11-32 中，辅助函数正好写在要修改的元素的后边，但你也可以不这样安排，你可以在模板的最后将 `draggable_element()` 和 `drop_receiving_element()` 分组列出。重要的是辅助函数调用的第一个参数，它指明要接受行为的元素标识符。

`draggable_element()` 辅助函数接受以下参数：

- **revert**：如果为真，当释放鼠标时，元素将返回原来位置。它也可以是任意的函数指针，用于拖动结束时调用。
- **ghosting**：克隆一个元素并且拖动这个复制品，放下这个复制品之前，原先的元素留在原地不动。
- **snap**：如果为假，则不会出现快照。否则，可拖动元素只能被拖拽到由 `xy` 或 `[x,y]` 或 `function(x,y){ return [x,y] }` 指定的 `xy` 交叉点。

`drop_receiving_element()` 辅助函数接受以下参数：

- **accept**：描述 CSS 类的一个字符串或一个字符串数组。只有当可拖放元素有一个或多个 CSS 类时，才能被接受。
- **hoverclass**：当用户在一个元素上拖动一个被接受的可拖动元素，则将 CSS 类加到这个元素上。

可排序列表

可拖动元素还能用鼠标移动列表的选项，来对一个列表进行排序。

`sortable_element()` 辅助函数将可排序行为加给选项，例 11-33 是实现这个特性的一个例子。

例 11-33 可排序列表

```
<p>What do you like most?</p>
<ul id="order">
  <li id="item_1" class="sortable">Carrots</li>
  <li id="item_2" class="sortable">Apples</li>
  <li id="item_3" class="sortable">Oranges</li>
  // Nobody likes Brussel sprouts anyway
  <li id="item_4">Brussel sprouts</li>
</ul>
<div id="feedback"></div>
<?php echo sortable_element('order', array(
  'url'      => 'item/sort',
  'update'   => 'feedback',
```



```
'only' => 'sortable'
)); ?>
```

借助于 `sortable_element()` 辅助函数的神奇能力，`` 元素成为可排序的，也就是说，它的子元素可以通过拖放被重新排序。每当用户通过拖动并释放一个选项来重新排序时，就会产生一个带有以下参数的 Ajax 请求：

```
POST /sf_sandbox/web/frontend_dev.php/item/sort HTTP/1.1
order[]=1&order[]=3&order[]=2&_ =
```

整个有序列表作为一个数组传送，格式是 `order[$rank]=$id`，`$order` 从 0 开始，`$id` 则是列表元素中 `id` 属性中下划线（“_”）后的值。

`sortable_element()` 辅助函数接受以下参数：

- `only`：描述 CSS 类的一个字符串或一个字符串数组。只有具有这个类的可排序元素的子元素才可以被移动。
- `hoverclass`：当鼠标在元素上浮动时，将 CSS 类加到元素上。
- `overlap`：如果多个选项显示在同一行，则设为 `horizontal`；如果每行显示一个选项（如例中所示），则设为 `vertical`。
- `tag`：如果要排序的列表不是一组 `` 元素，则你必须指定可排序元素中的哪些子元素是可以被拖放的（例如 `div` 或 `dl`）。

就地编辑

越来越多的互联网应用程序允许用户在页面中直接编辑页面内容，而无需在表单中重新显示内容。这种交互的原理很简单。当用户将鼠标浮动到一块文本上时，这块文本将被加亮。如果用户在这个块中点击，普通文本就转换为填写了文本的文本区，并且出现一个保存按钮。用户就可以在文本区中编辑，保存之后，文本区消失，普通文本重新出现。在 `symfony` 中，用 `input_in_place_editor_tag()` 辅助函数就可以将这个可编辑行为加给一个元素。例 11-34 显示了这个辅助函数的使用。

例 11-34 可编辑的文本

```
<div id="edit_me">You can edit this text</div>
<?php echo input_in_place_editor_tag('edit_me', 'mymodule/myaction',
array(
    'cols'      => 40,
    'rows'      => 10,
)); ?>
```

当用户点击可编辑文本时，它就被替换为一个填写了文本的文本输入区，这个输入区是可编辑的。当提交表单时，Ajax 将编辑过的值作为 `value` 参数调用

`mymodule/myaction` 动作，动作完成的结果就是更新了可编辑元素。写这样的代码非常快而且很有用。

`input_in_place_editor_tag()` 辅助函数接受以下参数：

- `cols` 和 `rows`：指明用于编辑的文本输入区的大小（如果 `rows` 大于 1，就成为 `<textarea>`）。
- `loadTextURL`：指明动作的 URI，调用该动作可以显示要编辑的文本。如果可编辑元素的内容使用特殊的格式，并且你想让用户不用格式化就可以编辑该文本时，这个参数非常有用。
- `save_text` 和 `cancel_text`：保存链接（默认值是 `ok`）和取消链接（默认值是 `cancel`）上的文本。

总结

如果你不喜欢在模板中用 Javascript 来实现客户端的行为，你可以用 Javascript 辅助函数。它们不仅能自动实现基本的链接和元素更新，还提供一种快速开发 Ajax 交互的方法。Prototype 有力地增强了 Javascript 语法，而 `script.aculo.us` 则提供了强大的视觉效果，有了它们的帮助，你只需要几行代码就可以写出复杂的交互程序。

因为用 `symfony` 生成高度交互性的应用就像写静态页面那样容易，所以你可以在 web 应用中实现大多数桌面应用程序的交互功能。

第 12 章 缓存

有一种提高应用程序速度的方法—存储生成的 HTML 代码片段，或者整个页面，以后直接使用存储的内容。这种技术叫做缓存，它可以在服务器端和客户端实现。

symfony 提供了一个灵活的服务器端缓存系统。通过很直观的 YAML 文件设置，它可以保存整个回应、一个动作、一个局部模板或者一个模板片段的的结果到一个文件里。当对应的数据变化时，你可以很方便的使用命令行或者特殊的动作方法有选择的清除缓存。symfony 还提供了一个通过 HTTP 1.1 头信息控制客户端缓存的简单方法。本章将详细介绍这些内容，并且还有一些监控缓存对程序性能影响的技巧。

缓存回应

HTML 缓存的原理很简单：重用之前类似请求的部分或者全部 HTML 代码。这些 HTML 代码存储在一个特定的地方（symfony 项目的 cache/目录），前端控制器在执行动作之前会先检查这个目录。如果找到缓存内容，就把它发送到客户端而不执行动作，因此大大加快了执行速度。如果没有缓存内容，就执行动作，然后把动作的结果（视图）保存在 cache/目录供以后使用。

由于所有的页面都可能包含动态内容，所以 HTML 缓存默认是关闭的。网站管理员可以开启它来提高性能。

symfony 能够处理三种不同类型的 HTML 缓存：

- 动作的缓存（包含或者不包含布局）
- 局部模板，组件或者组件槽的缓存
- 模板片段的缓存

前两种类型可以通过 YAML 配置文件来控制。模板片段的缓存是通过模板里的辅助函数来管理的。

全局缓存设置

项目的每个应用程序的不同环境，HTML 缓存机制都可以在 settings.yml 文件里的 cache 部分设置成开启或者关闭（默认）。例 12-1 是一个开启缓存的例子。

例 12-1 - 开启缓存， myapp/config/settings.yml

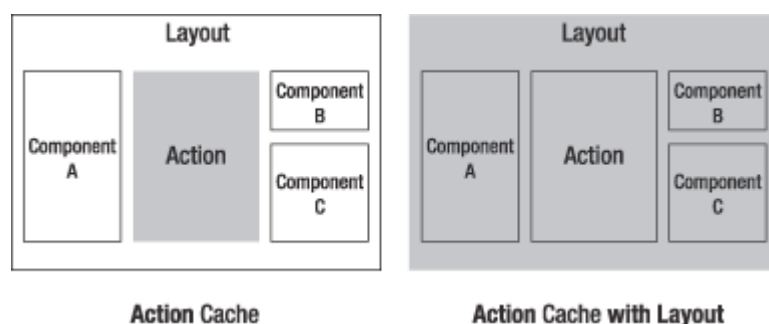
dev:

```
. settings:
  cache: on
```

缓存一个动作

显示静态内容的动作（不依赖数据库或者与 session 无关的数据）或者从数据读取信息的动作（比如，GET 请求），这类的动作通常比较适合作缓存。图 12-1 显示了不同情况下页面的哪些部分被缓存：动作结果（它的模板）或者动作结果与布局一起。

图 12-1 - 缓存动作



例如，有一个 `user/list` 动作，它返回网站所有用户的列表。除非有用户被修改、增加或者删除（这种情况会在“从缓存里移除内容”这一小节里讨论），这个动作都会显示同样的内容，所以它非常适合作缓存。

可以在 `config/` 目录的 `cache.yml` 文件里设置各个动作的开启关闭。请看例 12-2 里的例子。

例 12-2 - 为一个动作开启缓存，`myapp/modules/user/config/cache.yml`

```
list:
  enabled: on
  with_layout: false # 默认值
  lifetime: 86400 # 默认值
```

这个配置里开启了 `list` 动作的缓存，布局不会与动作一起缓存（一起缓存是默认设置）。这就是说，即使这个动作已经被缓存了，布局（还有其中的局部模板与组件）还是会被执行。如果 `with_layout` 设置成 `true`，那么布局就会与动作一起被缓存而不会再次执行。

测试缓存的设置，在你的浏览器里执行测试环境的这个动作。

http://myapp.example.com/myapp_dev.php/user/list

你会注意到页面的动作区域的边框，第一次，这个区域有一个蓝色的头部，这说明它不是缓存的内容，刷新页面，动作区域有一个黄色的头部，这说明这是

缓存的内容（并且速度提升明显）。在本章你会了解更多测试与检测缓存的方法。

NOTE 槽是模板的一部分，缓存动作的同时也会保存这个动作的模板里定义的槽的值。所以槽可以被缓存。

缓存系统也可以对有参数的页面起作用。假设 `user` 模块有一个 `show` 动作，它根据参数 `id` 显示一个用户的资料。修改 `cache.yml` 文件这个动作的缓存也打开，如例 12-3 所示。

为了更好地组织 `cache.yml` 文件，可以把一个模块的所有动作的设置放在 `all:` 键下，如例 12-3 里所示。

例 12-3 - 完整的 `cache.yml` 文件示例，
`myapp/modules/user/config/cache.yml`

```
list:
  enabled:    on
show:
  enabled:    on

all:
  with_layout: false    # 默认值
  lifetime:    86400    # 默认值
```

现在，每次用不同的 `id` 参数执行 `user/show` 动作会在缓存里新增一条记录。所以下面这个的缓存：

<http://myapp.example.com/user/show/id/12>

与下面的缓存不一样：

<http://myapp.example.com/user/show/id/25>

CAUTION 通过 `POST` 方法调用的动作或者通过直接的 `GET` 参数（不通过 `symfony` 的路由系统转义，直接指定 `GET` 参数）调用的动作不会被缓存。

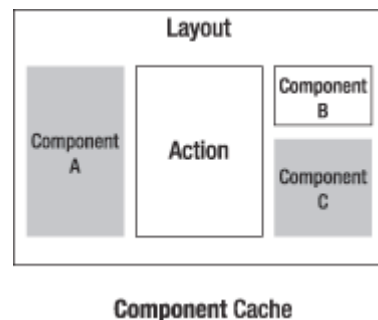
`with_layout` 这个设置还需要多作一点说明。这个参数决定了缓存里存放哪种数据。不缓存布局的情况下，只有模板的执行结果与动作变量存放在缓存里。缓存布局的时候，整个回应对象都会缓存。这就是说缓存布局要比不缓存布局要快很多。

如果功能上允许（也就是说，布局不依赖于与 `session` 有关的数据），你应该选择缓存布局。不幸的是，布局经常会包含动态元素（例如，登录的用户的名字），所以缓存动作的时候不包括布局是最常见的配置。不过，RSS 种子，弹出窗口，还有不依赖于 `cookies` 的页面可以连同布局一起缓存。

缓存一个局部模板，组件或者组件槽

第 7 章介绍了如何在多个模板里使用 `include_partial()` 辅助函数重用代码片段。局部模板与动作一样非常容易缓存，局部模板缓存开启的规则也一样，如图 12-2 所示。

图 12-2 - 缓存一个局部模板，组件或者组件槽



例如，例 12-4 展示了如何通过编辑 `cache.yml` 文件来开启 `user` 模块的 `_my_partial.php` 这个局部模板的缓存。注意 `with_layout` 设置在这里是没有意义的。

例 12-4 - 缓存一个局部模板，`myapp/modules/user/config/cache.yml`

```
_my_partial:
  enabled:    on
list:
  enabled:    on
...
```

现在所有使用到这个局部模板的模板都不会真正执行这个局部模板的 PHP 代码，而是直接使用缓存里的内容。

```
<?php include_partial('user/my_partial') ?>
```

与动作类似，局部模板的结果也会跟参数有关。缓存系统把所有不同参数的局部模板的结果保存下来。

```
<?php include_partial('user/my_other_partial', array('foo' =>
'bar')) ?>
```

TIP 动作缓存要比局部模板缓存功能强大，因为当动作被缓存时，模板是不会执行的；如果模板包含局部模板的调用，这些调用也不会执行。所以，局部模板缓存只在不使用动作缓存或者布局里的局部模板时有用。

让我们回顾一下第 7 章：组件是局部模板之上的轻量级的动作，组件槽是一个动作会随着调用动作而变化的组件。这两种包含类型很类似于局部模板，缓存的方法是一样。例如，如果你的全局布局用

`include_component('general/day')` 包含 `day` 这个组件，用来显示当前日期，设置 `general` 模块的 `cache.yml` 文件为如下内容可以开启这个组件的缓存：

```
_day:
  enabled: on
```

缓存组件或者局部模板的时候，你要决定是为所有调用的模板保存单一的版本还是为每个调用的模板保存一个版本。默认情况，缓存的组件与调用它的模板是无关的。不过与环境有关的组件，例如一个在不同的动作里显示不同的侧边栏的组件，应该为每一个调用它的模板保存一个缓存版本。缓存系统可以处理这种情况，只要把 `contextual` 参数设置成 `true`，如下：

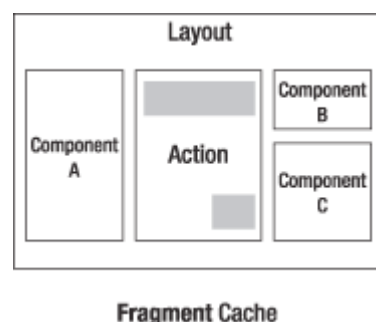
```
_day:
  contextual: true
  enabled: on
```

NOTE 全局组件（位于应用程序的 `template/` 目录）也可以缓存，这需要在应用程序的 `cache.yml` 里声明它的缓存设置。

缓存模板片段

动作缓存只适合一部分动作。但是对于其他的动作（用来更新数据或者在模板里显示与 `session` 有关的信息的动作）仍然有进行缓存提升性能的可能，不过要使用不同的方法。`symfony` 还提供了第三种缓存，专门用来缓存模板片段并且可以直接在模板里使用。在这种模式下，动作总是会执行，模板被分成执行片段和缓存片段，如图 12-3 所示。

图 12-3 - 缓存模板片段



例如，一个用户列表里有一个到最近访问用户的链接，这个内容是动态的。`cache()` 辅助函数定义模板的那些部分的内容要放到缓存里。语法详情见例 12-5。

例 12-5 - 使用 `cache()` 辅助函数，
`myapp/modules/user/templates/listSuccess.php`

```
<!-- 每次都执行的代码 -->
<?php echo link_to('last accessed user',
'user/show?id='.$last_accessed_user_id) ?>

<!-- 缓存代码 -->
<?php if (!cache('users')): ?>
    <?php foreach ($users as $user): ?>
        <?php echo $user->getName() ?>
    <?php endforeach; ?>
    <?php cache_save() ?>
<?php endif; ?>
```

它是这样工作的：

- 如果找到了名为 `users` 的缓存片段，就用它代替 `<?php if (!cache($unique_fragment_name)): ?>` 与 `<?php endif; ?>` 之间的代码。
- 如果没有，则执行这两行之间的代码然后保存到缓存，命名为 `users`。

不在这两行之间的代码总会执行而不会被缓存。

CAUTION 这个动作（例子中的 `list`）的缓存必须是关闭的，因为开启缓存以后会忽略整个模板还有模板片段的声明。

使用模板片段缓存对速度的提升没有动作缓存的明显，因为动作还是会执行，模板也要处理一部分，布局也还是会用来装饰。

你可以在同一个模板里声明多个模板片段缓存；不过，你必须给它们起不同的名字好让 `symfony` 缓存系统找到它们。

与动作和组件缓存一样，模板片段缓存的 `cache()` 辅助函数也可以接受持续时间的秒数作为第二个参数。

```
<?php if (!cache('users', 43200)): ?>
```

调用这个辅助函数时如果不指定，则使用默认缓存持续时间（86400 秒或 1 天）。

TIP 还有一个办法可以使这种的动作变得可以缓存，那就是在动作的路由模式（路径）中加入变量。例如，如果首页显示登录的用户名，那么只要使这个页

面的 URL 包含用户名这个动作就可以缓存了。另一个例子是对于国际化的项目：如果你想缓存一个有多种语言的页面，语言代码必须包含在 URL 的某处。这个方法增加了缓存里存放的页面数，不过它可以大大加快很繁忙的交互应用程序的速度。

动态配置缓存

cache.yml 是一种定义缓存设置的方法，但是要修改 cache.yml 的设置比较麻烦。不过，与 symfony 其他的地方一样，你可以用 PHP 代替 YAML，这样就可以动态设置缓存了。

为什么需要动态设置缓存呢？有个例子可以很好的说明这个问题，有一个页面登录用户和未登录用户看到的这个页面是不一样的，但是 URL 相同。假设 article/show 页面包含一个文章评分系统。未登录用户不能使用评分系统。这些未登录用户看到的是一个登录表单而不是评分系统。这个版本的页面可以被缓存。另外一方面，登录用户点击一个评分链接会发出一个 POST 请求并做出评分。这次，这个页面不需要缓存而应该是动态建立。

动态缓存应该在 sfCacheFilter 执行之前的过滤器(filter)里设置。事实上，缓存是 symfony 里的一个过滤器，这与网页调试工具条和安全功能一样。如果要设置 article/show 的缓存只针对未登录用户开启，需要在应用程序的 lib/ 目录里加一个 conditionalCacheFilter，内容见例 12-6。

例 12-6 - 通过 PHP 配置缓存，
myapp/lib/conditionalCacheFilter.class.php

```
class conditionalCacheFilter extends sfFilter
{
    public function execute($filterChain)
    {
        $context = $this->getContext();
        if (!$context->getUser()->isAuthenticated())
        {
            foreach ($this->getParameter('pages') as $page)
            {
                $context->getViewCacheManager()->addCache($page['module'],
                $page['action'], array('lifetime' => 86400));
            }
        }

        // Execute next filter
        $filterChain->execute();
    }
}
```

你还要在 `filter.yml` 文件里 `sfCacheFilter` 的地方注册这个过滤器，如例 12-7。

例 12-7 - 注册自己的过滤器，`myapp/config/filters.yml`

```
...
security: ~

conditionalCache:
  class: conditionalCacheFilter
  param:
    pages:
      - { module: article, action: show }

cache: ~
...
```

清除缓存（为了自动载入新的过滤器类），条件缓存就可以用了。它会使 `pages` 参数指定的页面的缓存在用户未登录时生效。

`sfViewCacheManager` 对象的 `addCache()` 方法需要一个模块名，动作名，还有一个与 `cache.yml` 文件里指定的参数内容一样的关联数组作为参数。例如，如果你想定义 `article/show` 动作必须与布局一起缓存，时间限制是 3600 秒，那么这样写代码：

```
$context->getViewCacheManager()->addCache('article', 'show', array(
  'withLayout' => true,
  'lifetime'    => 3600,
));
```

SIDEBAR 其他的缓存存储方式

默认情况，`symfony` 缓存系统将数据保存在服务器的硬盘上。你可能会想把缓存放在内存里（例如，通过 `memcache`）或者数据库里（特别是你想在几台服务器之间共享缓存数据或者加快缓存删除速度）。你可以轻松的修改 `symfony` 默认的缓存存储系统，因为 `symfony` 视图缓存管理器使用的类定义在 `factories.yml` 文件里。

默认的视图缓存存储 `factory` 是 `sfFileCache` 类：

```
view_cache:
  class: sfFileCache
  param:
    automaticCleaningFactor: 0
```

cacheDir: %SF_TEMPLATE_CACHE_DIR%

你可以用自己的存储类代替这个类或者使用 `symfony` 提供的其他的类（例如 `sfSQLiteCache`）。`param` 键下定义参数会作为关联数组传给这个类的 `initialize()` 方法。所有的视图缓存存储类都必须定义抽象类 `sfCache` 里定义的方法。详情请参考 API 文档（<http://www.symfony-project.com/api/symfony.html>）。

使用极速缓存

即使是缓存的页面也需要执行一些 PHP 代码。缓存的页面，`symfony` 还是要载入配置文件，建立回应等。如果你非常确定一个页面在一段时间内不会改变，你可以完全绕过 `symfony`，直接把生成的 HTML 代码放在 `web/` 目录下。这需要 Apache 的 `mod_rewrite`，还有你的路由规则需要指定没有后缀或者以 `.html` 为后缀。

你可以手动的，一页一页的，使用这个简单的命令作这件事：

```
> curl http://myapp.example.com/user/list.html > web/user/list.html
```

设置好以后，每次请求 `user/list` 动作的时候，Apache 会找到对应的 `list.html` 文件然后完全忽略 `symfony`。这样做的代价是你不能用 `symfony` 控制页面缓存了（生存时间，自动删除等），但是速度提高非常明显。

另外，你可以使用 `sfSuperCache` 这个 `symfony` 插件，它能自动生成极速缓存并且支持存活时间和清除缓存。插件的使用请参考第 17 章。

SIDEBAR 其他的加速策略

除了 HTML 缓存之外，`symfony` 还有两种其他的缓存机制，这两种缓存机制是完全自动并且对开发者透明的。在生产环境，配置和模板翻译的缓存会自动放在 `myproject/cache/config/` 和 `myproject/cache/i18n/` 目录里。

PHP 加速器（`eAccelerator`，`APC`，`XCache` 等），也被称作 `opcode` 缓存模块，可以通过缓存 PHP 代码编译后的状态来提高 PHP 代码的执行速度，所以代码解析还有编译的负担可以完全被消除。这对包含大量代码的 `Propel` 类特别有用。这些加速器与 `symfony` 兼容并且可以轻易的将程序的速度提高三倍。所以推荐流量大的 `symfony` 应用程序在生产环境使用这些加速器。

使用 PHP 加速器，你可以使用 `sfProcessCache` 类在内存里保存持久数据，这样可以避免每次请求作同样的处理。另外如果你想保存一个很消耗 CPU 的函数的结果到一个文件，你可以使用 `sfFunctionCache` 对象。关于这些机制的详细信息，请参考第 18 章。

从缓存里删除项目

如果程序的脚本或者数据发生了改变，那么缓存的数据会过期。为了避免这样的情况同样也为了避免 bug，你可以根据需要用不同的方法删除缓存的部分内容。

删除整个缓存

symfony 命令行的 clear-cache 任务可以删除缓存（HTML，配置文件还有 i18n 的缓存）。你可以指定参数让它只删除一部分缓存，如例 12-8 所示。请注意只能在 symfony 项目的根目录执行这个命令。

例 12-8 - 清除缓存

```
// 清除整个缓存
> symfony clear-cache

// 简写
> symfony cc

// 只清除 myapp 应用程序的缓存
> symfony clear-cache myapp

// 只清除 myapp 应用程序的 HTML 缓存
> symfony clear-cache myapp template

// 只清除 myapp 应用程序的配置缓存
> symfony clear-cache myapp config
```

清除指定的缓存

数据库更新以后，与修改的数据有关动作的缓存必须清除。你可以清除整个缓存，不过这就把其他不相关的缓存内容浪费了。这里就需要 sfViewCacheManager 对象的 remove() 方法。它需要一个内部 URL 作为参数（与传给 link_to() 的一样），它会删除相关的动作缓存。

例如，假设 user 模块的 update 动作修改 User 对象的字段。那么 list 和 show 动作的缓存就需要清除，否则包含错误数据的旧版本就会显示出来。可以使用 remove() 方法来处理，如例 12-9 所示。

例 12-9 - 清除指定动作的缓存， modules/user/actions/actions.class.php

```
public function executeUpdate()
{
    // 更新用户
    $user_id = $this->getRequestParameter('id');
```

```

$user = UserPeer::retrieveByPk($user_id);
$this->forward404Unless($user);
$user->setName($this->getRequestParameter('name'));
...
$user->save();

// 清除与这个用户有关的动作的缓存
$cacheManager = $this->getContext()->getViewCacheManager();
$cacheManager->remove('user/list');
$cacheManager->remove('user/show?id=' . $user_id);
...
}

```

删除缓存的局部模板，组件或者组件槽有点复杂。你可以传给它们任何种类的参数（包括对象），这之后你几乎无法识别缓存的版本。这里我们重点介绍局部模板，这与其他模板缓存是相同的。symfony 用一个特殊的前缀（sf_cache_partial）标识一个缓存的局部模板，然后再加上模块的名字、局部模板的名字还有所有参数的哈希值，如下所示：

```

// 调用一个局部模板
<?php include_partial('user/my_partial', array('user' => $user) ?>

// 这个局部模板在缓存中的标识
/sf_cache_partial/user/_my_partial/sf_cache_key/bf41dd9c84d59f3574a5da244626dcc8

```

理论上，如果你知道识别参数的哈希值你可以用 remove() 方法删除一个缓存的局部模板，但这是很不现实的。幸运的是，如果你在 include_partial() 辅助函数调用的之后指定一个 sf_cache_key 参数，你就可以用你知道的东西来标识这个缓存。例 12-10 里，清除一个缓存的局部模板，清除修改过的 User 相关的局部模板，变得很容易。

例 12-10 - 清除缓存里的局部模板

```

<?php include_partial('user/my_partial', array(
    'user' => $user,
    'sf_cache_key' => $user->getId()
) ?>

// 这个模板在缓存里的标识
/sf_cache_partial/user/_my_partial/sf_cache_key/12

```

```
// 清除某个特定的 user 的_my_partial，使用 $cacheManager-
>remove('@sf_cache_partial?module=user&action=_my_partial&sf_cache_key=' . $user->getId());
```

使用这个方法并不能清除一个局部模板的所有缓存。这些你会在本章后面的“手动清除缓存”里了解到。

要清除模板片段的缓存，也可以用同样的 `remove()` 方法。缓存里模板片段的标识符由跟刚才一样的 `sf_cache_partial` 前缀，模块名，动作名还有 `sf_cache_key`（使用 `cache()` 辅助函数使指定的不重复的名字）组成。见例 12-11。

例 12-11 - 清除模板片段的缓存

```
<!-- 缓存的代码 -->
<?php if (!cache('users')): ?>
    ... // Whatever
    <?php cache_save() ?>
<?php endif; ?>

// 这个缓存的标识符使用了缓存页面的网页调试工具条
/sf_cache_partial/user/list/sf_cache_key/users

// 这样清除它
$cacheManager-
>remove('@sf_cache_partial?module=user&action=list&sf_cache_key=users
');
```

SIDEBAR 选择性清除缓存比较伤脑筋

缓存清除工作中最麻烦的事情是判断那些动作受到数据更新的影响。

例如，假设当前应用程序有一个 `publication` 模块，这个模块显示出版物列表（`list` 动作）和连同作者（`User` 类的实例）详细信息的出版物描述（`show` 动作）。修改一条作者记录会影响这个作者所有的出版物列表及其详细描述。这意味着你需要在 `user` 模块的 `update` 动作里增加一些这样的东西：

```
$c = new Criteria();
$c->add(PublicationPeer::AUTHOR_ID, $this->getRequestParameter('id'));
$publications = PublicationPeer::doSelect($c);

$cacheManager = sfContext::getInstance()->getViewCacheManager();
foreach ($publications as $publication)
```

```
{
    $cacheManager->remove('publication/show?id=' . $publication->getId());
}
$cacheManager->remove('publication/list');
```

当你开始使用 HTML 缓存的时候，你要清楚地了解模型和动作之间的相关性，这样才不会出现因为错误的关系造成的新问题。注意如果应用程序里使用了 HTML 缓存所有的修改模型的动作都应该包含一些 `remove()` 方法的调用。

不过，如果你不想伤脑筋来去作复杂分析，完全可以每次更新数据的时候清除整个缓存。

缓存目录结构

应用程序的 cache/ 目录包含下面的结果：

```
cache/                # sf_root_cache_dir
  [APP_NAME]/          # sf_base_cache_dir
    [ENV_NAME]/        # sf_cache_dir
      confi g/         # sf_confi g_cache_dir
      i 18n/           # sf_i 18n_cache_dir
      modul es/        # sf_modul e_cache_dir
      templ ate/       # sf_templ ate_cache_dir
        [HOST_NAME]/
          all /
```

缓存的模板存放在 `[HOST_NAME]`（主机名）目录下（为了文件系统的兼容性，点被替换成了下划线），然后按照 URL 来组织下面的目录结构。例如，下面一个页面的模板缓存：

<http://www.myapp.com/user/show/id/12>

存放在：

`cache/myapp/prod/template/www_myapp_com/all/user/show/id/12.cache`

在代码里不要直接使用文件路径，应该使用文件路径常量来代替。例如，获取当前应用程序的当前环境下的 `template/` 的绝对路径可以使用 `SfConfig::get('sf_template_cache_dir')`。

了解这个目录结构有助于手工清除模板。

手工清除缓存

跨应用程序清除缓存是个麻烦。例如，如果一个管理员通过 **backend** 应用程序修改了 **user** 表里的一条记录，所有 **frontend** 应用程序的与这条数据有关的动作的缓存都需要清除。`remove()` 方法需要的参数是一个内部 URL，但是应用程序不知道其他应用程序的路由规则（应用程序间是独立的），所以不能使用 `remove()` 方法来清除其他应用程序的缓存。

解决方法是根据路径手工清除 **cache/**目录下的文件。例如，如果 **backend** 应用程序需要清除 **frontend** 应用程序的 **user/show** 动作的 **id** 参数为 **12** 的缓存，可以使用下面的代码：

```
$sf_root_cache_dir = sfConfig::get('sf_root_cache_dir');
$cache_dir =
$sf_root_cache_dir.'/frontend/prod/template/www_myapp_com/all';
unlink($cache_dir.'/user/show/id/12.cache');
```

但这并不能让人满意。这个命令只能清除当前环境的缓存，而且还需要在文件路径里写明环境名和当前的主机名。使用 `sfToolkit::clearGlob()` 方法可以避免这个问题。它可以接受一个包含通配符的文件路径作为参数。例如，清除上个例子中的缓存文件，不必指定主机名与环境，可以使用下面的代码：

```
$cache_dir = $sf_root_cache_dir.'/frontend/*/template/*/all';
sfToolkit::clearGlob($cache_dir.'/user/show/id/12.cache');
```

这个方法在清除与特定参数无关的动作缓存时也很有用。例如，如果你的应用程序处理多种语言，你会在所有的 URL 里加上语言代码。所以到用户档案页面的 URL 可能会类似这样：

<http://www.myapp.com/en/user/show/id/12>

要清除缓存的所有语言的 **id** 为 **12** 的用户档案，可以简单的这样做：

```
sfToolkit::clearGlob($cache_dir.'/*/user/show/id/12.cache');
```

缓存测试与监测

HTML 缓存如果处理的不好，可能会造成显示的数据混乱。所以每当你禁用一个元素的缓存的时候，你都应该完整的测试它并且监测执行速度并进行调整。

建立一个临时工作环境

由于在开发模式下缓存默认是关闭的，所有缓存系统可能造成生产环境中无法察觉的新问题。如果你开启某些动作的 HTML 缓存，你应该增加一个新环境，在

本章称作临时工作环境（staging），设置与 prod 环境相同（例如，开启缓存）不过 web_debug 设置成 on。

修改应用程序的 settings.yml，增加例 12-12 里的这几行内容到这个文件的最前面。

例 12-12 - 设置一个临时工作环境 staging，myapp/config/settings.yml

```
staging:
  .settings:
    web_debug:  on
    cache:      on
```

另外，复制生产环境的前端控制器（比如 myproject/web/index.php）来创建一个新的前端控制器 myapp_staging.php。修改 SF_ENVIRONMENT 和 SF_DEBUG 的值，如下：

```
define('SF_ENVIRONMENT', 'staging');
define('SF_DEBUG',      true);
```

好了，你建立了一个新的环境。在域名后加这个前端控制器的名字来调用：

http://myapp.example.com/myapp_staging.php/user/list

TIP 除了复制旧的前端控制器，还可以通过 symfony 命令行建立一个新的前端控制器。例如，建立 myapp 应用程序的 staging 环境，文件名为 myapp_staging.php，SF_DEBUG 值为 true，只要使用 symfony init-controller myapp staging true 命令即可。

监测性能

第 16 章会详细介绍网页调试工具条的内容。不过，由于工具条包含了有关缓存元素的有用的信息，下面先简单介绍一下工具条的缓存功能。

浏览包含可缓存元素（动作，局部模板，模板片段等）的页面的时候，网页工具条（在窗口的右上角）上会有一个忽略缓存按钮（绿色的，环行箭头），如图 12-4。这个按钮会重新载入页面并且强制处理缓存的元素。注意它并不会清除缓存。

调试工具条最右边的数字是请求执行的时间。如果对某个页面开启了缓存，第二次打开这个页面的时候这个数字会减少，因为 symfony 使用缓存的数据而不是重新处理脚本。可以很方便的通过这个指示器监视缓存对性能的提升。

图 12-4 - 使用了缓存的页面的网页调试工具条



调试工具条还会显示当前请求执行的数据库查询数，还会按照分类显示本次查询消耗的时间（点总时间查看详情）。通过监视这些数据及处理总时间，可以帮你衡量缓存带来的性能提高。

基准化分析 Benchmarking

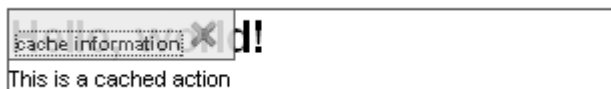
由于要在日志和网页工具条上显示调试信息，调试模式很影响应用程序的速度。所以在临时环境 `staging` 的处理时间并不能代表生产环境里调试模式关闭状态下的处理时间。

要更好的了解每次请求的时间，需要使用基准化分析工具，例如 `Apache Bench` 或者 `JMeter`。这些工具可以进行负载测试并且提供两个重要的信息：某个页面的平均载入时间和服务器的最大处理能力。平均载入时间这个数据在监测缓存带来的性能提升时特别有用。

识别缓存的部分

开启了网页调试工具条以后，页面里缓存的部分会用一个红色的框标识出来，左上角会显示缓存的信息框，如图 12-5 所示。如果这个元素被执行，框会是蓝色背景，如果是黄色背景那么就是缓存的数据。点击缓存信息链接会显示缓存的标识符，它的生存时间和距离上次改变的时间。这在处理没有环境元素的时候比较有用，它可以显示这个元素建立的时间还有模板的哪些部分可以缓存。

图 12-5 - 识别页面里缓存的元素



HTTP 1.1 与客户端缓存

HTTP 1.1 协议定义了一些控制浏览器缓存系统的头信息，这对进一步提高应用程序的速度有很大的帮助。

万维网联盟定义的 HTTP 1.1 规范（W3C, <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>）详细介绍了这些头信息。如果一个动作开启了缓存，并且使用了 `with_layout` 选项，那么还可以通过下面的一个或者多个方法进一步优化。

即使有些网站访问者的浏览器不支持 HTTP 1.1，使用 HTTP 1.1 的缓存功能也没有任何害处。浏览器接受到不能识别的头信息以后会直接简单的把这样的头信息忽略掉，所以建议使用 HTTP 1.1 缓存。

另外，代理服务器和缓存服务器也能识别 HTTP 1.1 的头信息。即使用户的浏览器不支持 HTTP 1.1，还是有可能通过代理服务器来获得 HTTP 1.1 缓存带来的好处。

增加 ETag 头信息来避免发送重复的内容

ETag 功能开启以后，web 服务器会在 HTTP 头部增加回应信息的签名。

ETag: 1A2Z3E4R5T6Y7U

用户的浏览器会把这个签名保存起来，然后下次请求同一个页面的时候把这个签名一起发出去。如果新的签名比对后发现页面没有改变，那么服务器不会发送回应，而只是发一个 304: Not modified 的头信息。这样节省了服务器的 CPU 时间（例如 GZIP 开启）和带宽（页面传输），还有客户端的时间（页面传输）。这样加起来，有 ETag 缓存的页面的载入时间比没有 ETag 缓存的要短。

在 symfony 里，可以修改 settings.yml 来为整个应用程序开启 ETag 功能。下面是默认的 ETag 设置：

```
all:
  .settings:
    etag: on
```

缓存了布局的动作，回应是从 cache/目录里取出的，所以处理速度更快。

增加 Last-Modified 头信息避免发送仍然有效的内容

服务器向浏览器发送回应的时候，它会增加一个特殊的头部信息来说明数据包含的页面最后修改的时间：

Last-Modified: Sat, 23 Nov 2006 13:27:31 GMT

浏览器能理解这个头信息，当再次请求这个页面的时候，会对应的加上一个 If-Modified 头信息：

If-Modified-Since: Sat, 23 Nov 2006 13:27:31 GMT

服务器可以比较客户端和应用程序返回的这个值。如果匹配，服务器返回 304: Not modified 头信息，与 ETags 很像，这样可以节约带宽和 CPU 时间。

在 symfony 里，你可以像其他的头信息一样设定 Last-Modified。例如，可以在动作里这么使用：

```
$this->getResponse()->setHttpHeader('Last-Modified', $this->getResponse()->getDate($timestamp));
```

这个日期可以从数据库或者文件系统取得数据更新的真实时间。`sfResponse` 对象的 `getDate()` 将时间戳转化成 `Last-Modified` 需要的日期时间格式（RFC1123）。

通过增加 Vary 头信息来保存一个页面的多个缓存版本

另一个 HTTP 1.1 头信息是 `Vary`。它可以定义页面取决于哪个参数，可以被浏览器和代理服务器识别。例如，如果页面的内容取决于 `cookies`，可以把 `Vary` 设置成这样：

`Vary: Cookie`

多数时候，由于页面内容会随着 `cookie`、用户语言或者其他因素的影响改变，所以很难开启对动作的缓存。如果你不介意增加缓存的大小，就应该正确地设置回应的 `Vary` 头信息。可以通过 `cache.yml` 配置文件或者 `sfResponse` 相关的方法对整个对象或者其中的某些动作设置 `Vary` 头信息：

```
$this->getResponse()->addVaryHTTPHeader('Cookie');  
$this->getResponse()->addVaryHTTPHeader('User-Agent');  
$this->getResponse()->addVaryHTTPHeader('Accept-Language');
```

`symfony` 会对这些参数的每个值保存一个不同的缓存版本。这会增加缓存的尺寸，不过服务器收到与这些匹配的头信息的时候，回应就直接从缓存里面取而不用处理。这对只取决于请求头信息的页面来说可以大大的提高性能。

通过增加 Cache-Control 头信息来允许客户端缓存

到目前为止，即使增加了缓存有关的 HTTP 头或是存在缓存的页面，浏览器还是会从服务器请求数据。可以通过增加 `Cache-Control` 和 `Expires` 头信息来避免重复请求。这些头信息在 PHP 是默认关闭的，不过 `symfony` 改写了这个设置来避免不必要的请求。

与之前一样，可以通过调用 `sfResponse` 对象的方法来开启这个功能。在动作里，定义页面最长的缓存时间（以秒为单位）：

```
$this->getResponse()->addCacheControlHTTPHeader('max_age=60');
```

也可以指定页面缓存的条件，防止服务器的缓存里保存私有数据（例如银行帐号）：

```
$this->getResponse()->addCacheControlHTTPHeader('private=True');
```

使用 Cache-Control HTTP 指令，你可以很好的调整服务器和浏览器之间的缓存机制。这些指令的详细信息，请看 W3C Cache-Control 的规范说明。

最后一个可以通过 symfony 设定的头信息是 Expires 头信息：

```
$this->getResponse()->setHTTPHeader('Expires', $this->getResponse()->getDate($timestamp));
```

CAUTION 使用了 Cache-Control 之后导致了最主要的一个问题是服务器的日志不会记录所有的请求，它只记录真正收到的请求。如果性能提高了，网站统计信息上的数字反而会减少。

总结

根据不同的缓存类型，缓存系统提供了多种提升性能的方法。效果从最好到最差，缓存分成下面几种类型：

- 极速缓存
- 包含布局的动作缓存
- 不包括布局的动作缓存
- 模板里的片段缓存

另外，局部模板和组件也可以被缓存。

如果改变了模型或者 session 里的数据，需要清除缓存来保持一致性，可以通过微调来优化性能——只清除改变了的东西，保留其他的。

请注意测试所有开启缓存页面的时候要格外小心，如果缓存了错误的数据或者更新数据时忘记更新缓存可能造成新问题。临时工作环境 staging 就是专门为这个准备的。

最后，尽量利用好 HTTP 1.1 协议还有 symfony 的先进缓存调整功能，客户端的缓存可以使性能更进一步提高。

第 13 章 国际化（I18n）与本地化（L10n）

如果你曾经开发过国际化的应用，你一定知道要处理好文本翻译，地方标准和本地化内容是极为困难的事。但是，symfony 能够自动处理与国际化有关的各种问题。

因为“国际化”这个词的英文名字太长（internationalization），所以开发者们通常将它简写为 i18n，你只要数一下这个英文词的字母个数，你就会知道为什么会有这么一个奇怪的缩写了。同样，“本地化”（localization）被简写为 l10n。这两个概念涵盖了多语种互联网应用的两个不同的方面。

一个国际化应用程序通常包括多个版本，它们的内容相同而语言或格式不同。例如，一个电子邮件系统就可以用多种语言提供同一种服务，仅仅是界面不同而已。

而一个本地化应用则会根据用户浏览的地区不同而显示不同的信息。如果你浏览一个新闻网站，那么在美国访问这个网站，页面将显示与美国有关的最新话题，而在法国访问这个网站，页面就应该显示在法国发生的最新事件。所以说，l10n 不仅要翻译页面内容，而且要根据不同的本地化版本提供不同的内容。

总之，i18n 和 l10n 将会处理以下问题：

- l 文本翻译（包括界面，资源和内容）
- l 标准和格式（包括日期，数量，数字等等）
- l 本地化内容（根据不同地区，给出某个内容的不同版本）

本章将介绍 symfony 处理这些问题的方法以及如何运用这些方法来开发国际化和本地化的应用系统。

用户的国家和语言（User Culture）

一个称为 culture 的用户会话参数决定了 symfony 中内置的 i18n 特性。这个 culture 参数由用户所在的国家 and 语言组成，它确定如何显示文本以及与 culture 有关的信息。因为 culture 参数被序列化保存在用户会话中，所以它在页面之间是持久有效的。

设定默认的国家 and 语言

默认情况下，新用户的 `cul ture` 是 `default_cul ture`。可以在 `i18n.yml` 配置文件中改变它的设定，参见例 13—1 所示。

例 13—1 在 `myapp/config/i18n.yml` 中设定默认 `cul ture`。

```
all:
  default_cul ture:      fr_FR
```

NOTE 你也许会奇怪为什么虽然在开发过程中改变了 `i18n.yml` 中的值，浏览器中显示的内容却没有相应变化。这是因为浏览器从前面的页面中保留了先前的 `cul ture` 值。如果你想看到新 `cul ture` 值带来的变化，你需要清除 `cookie` 或重启你的浏览器。

因为来自法国、比利时或加拿大的用户使用不同的法语翻译；而西班牙或墨西哥的用户也使用不同的西班牙语内容，所以 `cul ture` 参数是由语言和国家两个参数值组成的。语言参数根据 ISO 639-1 标准，用两个小写字母表示，例如，英语用 `en` 表示。而国家参数根据 ISO 3166-1 标准，用两个大写字母表示，例如，英国用 `GB` 表示。

改变用户的国家和语言

在与浏览器的会话过程中，用户的 `cul ture` 值是可以被改变的。例如，当用户决定从英文版切换到法文版，或当用户登录时要使用自己预设的语言的时候。这就是为什么 `SfUser` 类提供了存取用户 `cul ture` 值的获取方法和设置方法的原因。例 13—2 显示了如何在一个动作中运用这些方法。

例 13—2 在动作中设定和获取 `cul ture` 值

```
// 设定 cul ture
$this->getUser()->setCul ture(' en_US');

// 获取 cul ture
$cul ture = $this->getUser()->getCul ture();
=> en_US
```

SIDEBAR URL 里的 `cul ture`

当你运用 `symfony` 的国际化和本地化特性时，一个 URL 的页面就可以有多个不同的版本——这取决于用户的会话，而这样会使你不能在搜索引擎中缓存或索引你的页面。

解决这个问题的一种方法是在每个 URL 中加上 `cul ture` 值，这样，被翻译过的页面就会被看成是不同的 URL。为此，你可以在应用的 `routing.yml` 中为每个规则加上 `sf_cul ture` 标记：

```
page:
  url: /:sf_culture/:page
  requirements: { sf_culture: (?:(fr|en|de)) }
  params: ...
```

```
article:
  url: /:sf_culture/:year/:month/:day/:slug
  requirements: { sf_culture: (?:(fr|en|de)) }
  params: ...
```

为了避免在每个 `link_to()` 中手工加 `sf_culture` 参数，`symfony` 会将用户 `culture` 自动加到默认的路由参数中。反之，如果在 URL 中出现 `sf_culture` 参数，`symfony` 也会自动改变用户 `culture` 的值。

自动确定用户的国家和语言

在许多应用中，用户的国家和语言是根据浏览器的设定在第一次发出请求时确定的。用户可以在浏览器中定义一组可接受的语言，每当浏览器向服务器发出一个请求，这组语言数据就会被放在 HTTP 头的 `Accept-Language` 参数中。你可以通过 `symfony` 的 `sfRequest` 对象来获取它。例如，要获得一个动作中用户的预设语言列表，可以用以下方法：

```
$languages = $this->getRequest()->getLanguages();
```

HTTP 头是一个字符串，但 `symfony` 自动将它转换为一个数组。所以可以用 `$languages[0]` 取得上面例子中的用户预设语言。

在网站首页或在每个页面的过滤器中，自动将浏览器语言设定为用户的国家和语言是比较有用的方法。

CAUTION HTTP 头的 `Accept-Language` 值并不是很可靠的，因为很少有用户知道如何改变浏览器中的语言值。大多数时候，预设的浏览器语言是界面的语言，而浏览器并不是在所有语言下都是可用的。如果你根据浏览器预设语言自动设定 `culture` 值，最好可以提供一个让用户选择语言的方法。

标准与格式

Web 应用内部是不考虑国家和语言因素的。比如说数据库，它使用的日期、数量等数据都是按国际标准存放的。但是，当用户浏览数据时，就需要进行格式转换。用户不可能理解时间戳的概念，而且对于法国人来说，他们更愿意将自己的母语称为 `Français` 而不是 `French`。所以你需要根据用户 `culture` 值，对格式进行自动转换。

根据用户 `culture` 值输出数据

一旦定义了 culture 值，辅助函数就会据此自动进行适当的输出。例如，format_number() 辅助函数就会根据用户的 culture 值，自动将数字以用户熟悉的格式输出。参见例 13—3。

例 13—3 根据用户 culture 值显示数字

```
<?php use_helper('Number') ?>

<?php $sf_user->setCulture('en_US') ?>
<?php echo format_number(12000.10) ?>
=> '12,000.10'

<?php $sf_user->setCulture('fr_FR') ?>
<?php echo format_number(12000.10) ?>
=> '12 000,10'
```

你无需明确地将 culture 值传递给辅助函数。辅助函数会在当前的会话对象中自动找到 culture 值。例 13—4 中列出的辅助函数在输出数据时都会考虑用户的 culture 值。

例 13—4 与用户 culture 有关的辅助函数。

```
<?php use_helper('Date') ?>

<?php echo format_date(time()) ?>
=> '9/14/06'

<?php echo format_datetime(time()) ?>
=> 'September 14, 2006 6:11:07 PM CEST'

<?php use_helper('Number') ?>

<?php echo format_number(12000.10) ?>
=> '12,000.10'

<?php echo format_currency(1350, 'USD') ?>
=> '$1,350.00'

<?php use_helper('I18N') ?>

<?php echo format_country('US') ?>
=> 'United States'
```

```

<?php format_language('en') ?>
=> 'English'

<?php use_helper('Form') ?>

<?php echo input_date_tag('birth_date', mktime(0, 0, 0, 9, 14,
2006)) ?>
=> input type="text" name="birth_date" id="birth_date"
value="9/14/06" size="11" />

<?php echo select_country_tag('country', 'US') ?>
=> <select name="country" id="country"><option
value="AF">Afghanistan</option>
...
<option value="GB">United Kingdom</option>
<option value="US" selected="selected">United States</option>
<option value="UM">United States Minor Outlying
Islands</option>
<option value="UY">Uruguay</option>
...
</select>

```

如果给日期辅助函数加一个额外的格式参数，它可以不受 `culture` 值的影响进行输出。不过如果你的应用是国际化的，就不要使用这个额外的格式参数。

从本地化输入获取数据

如果说在获取数据时要根据用户的 `culture` 值来显示数据，那么你也可以让用户输入已经国际化了的数据。这可以帮你在转换不同格式和不确定的本地特性的数据方面节省时间。例如，谁有可能在输入框中输入了一个以逗号分割的货币值呢？

你可以通过隐藏真正的数据（就象在 `select_country_tag()` 中一样），或者将复杂数据的不同部分放入多个简单的输入框去的办法来标准化用户的输入格式。

但是对于日期来说，这通常很难做到。用户习惯于用他们自己的格式输入日期，而你需要将这样的日期转换成内部的（也是国际化的）格式。这正是 `sfi18N` 类要做的事。例 13—5 显示了如何应用这个类。

例 13—5 在动作中从一个本地化格式得到日期

```

$date= $this->getRequestParameter('birth_date');
$user_culture = $this->getUser()->getCulture();

```

```
// 取得时间戳
$timestamp = sf118N::getTimestampForCulture($date, $user_culture);

// 取得结构化了日期
list($d, $m, $y) = sf118N::getDateForCulture($date, $user_culture);
```

数据库中的文本信息

本地化的应用会根据用户 `culture` 来提供不同的内容。例如，一个在线商店在全球范围销售某产品，价格统一，但各个国家的产品说明却各不相同。这意味着数据库需要为数据存储不同的版本，为此，你需要用某种特别的方法来设计你的数据库模式，并在你要操作本地化模型对象时，使用 `culture` 值。

创建本地化数据库设计（schema）

对于包括本地化数据的表，你需要将它分成两个表：一个表不包含任何 `i18n` 列，另一个表则只包含 `i18n` 列。这两个表通过 1 对多关系连接起来。这种方法可以让你不用改变模型就可以增加任意多种语言。让我们看一下 `Product` 表。

首先，在 `schema.yml` 文件中创建表，如例 13—6 所示。

例 13—6 为 `i18n` 数据建立的数据库设计，文件路径为 `config/schema.yml`

```
my_connection:
  my_product:
    _attributes: { phpName: Product, isI18N: true, i18nTable:
my_product_i18n }
    id: { type: integer, required: true, primaryKey: true,
autoincrement: true }
    price: { type: float }

  my_product_i18n:
    _attributes: { phpName: ProductI18n }
    id: { type: integer, required: true, primaryKey: true,
foreignTable: my_product, foreignReference: id }
    culture: { isCulture: true, type: varchar, size: 7, required:
true, primaryKey: true }
    name: { type: varchar, size: 50 }
```

注意第一个表中的 `isI18N` 和 `i18nTable` 属性，及第二个中的特殊的 `culture` 列。这些都是 Propel 针对 `symfony` 而增强的特性。

`symfony` 的自动化工具可以快速完成这一过程。如果包含国际化数据的表名是主表的名字加上 `_i18n` 后缀，而且两个表之间通过表中的 `id` 列相互关联，那么

你就可以省略主表中的 `i18n` 属性，同时可以省略 `_i18n` 表的 `id` 和 `culture` 列，`symfony` 会自动处理这些表。也就是说，在 `symfony` 看来，例 13—7 的设计和例 13—6 的设计是完全一样的。

例 13—7 在 `config/schema.yml` 用简化格式写的 `i18n` 数据库设计

```
my_connection:
  my_product:
    _attributes: { phpName: Product }
    id:
    price: float
  my_product_i18n:
    _attributes: { phpName: ProductI18n }
    name: varchar(50)
```

运用生成的 `i18n` 对象

一旦建立了相应的对象模型（每次改变 `schema.yml` 后，别忘了调用 `symfony propel -build-model`，并用 `symfony cc` 清空缓存），你就可以使用支持 `i18n` 的 `Product` 类，就像只有一个表一样。示例请参看例 13—8。

例 13—8 处理 `i18n` 对象

```
$product = ProductPeer::retrieveByPk(1);
$product->setCulture('fr');
$product->setName('Nom du produit');
$product->save();

$product->setCulture('en');
$product->setName('Product name');
$product->save();

echo $product->getName();
=> 'Product name'

$product->setCulture('fr');
echo $product->getName();
=> 'Nom du produit'
```

如果你不想在每次使用 `i18n` 对象时都要设定 `culture` 值，你也可以在对象类中改变 `hydrate()` 方法。见例 13—9 的示例。

例 13—9 在 `myproject/lib/model/Product.php` 中重载 `hydrate()` 方法以设置 `culture` 值

```
public function hydrate(ResultSet $rs, $startcol = 1)
{
    parent::hydrate($rs, $startcol);
    $this->setCulture(sfContext::getInstance()->getUser()-
>getCulture());
}
```

查询 peer 对象时，如果不用通常的 doSelect 方法，而用 doSelectWithI18n 方法，就可以根据当前的 culture 值得到翻译的结果。如例 13—10 所示。另外，这个方法还同时建立与 i18n 有关的对象，因而可以以较少的查询得到全部内容（参看第 18 章的内容，该方法有助于提高性能）。

例 13—10 用 i18n 规则获取对象

```
$c = new Criteria();
$c->add(ProductPeer::PRICE, 100, Criteria::LESS_THAN);
$products = ProductPeer::doSelectWithI18n($c, $culture);
// $culture 参数是可选的，
// 如果没有指明 culture，则使用当前的用户 culture。
```

总的来说，你不要直接操作 i18n 对象，而应该在每次用规则的对象查询时将 culture 值传递给模型。

界面翻译

I18n 应用系统中的用户界面应该加以适当的处理，虽然模板中的标签、信息和导航栏使用多种语言，但显示时应该使用一致的表达方式。symfony 建议你用默认语言建立模板，然后在一个字典文件中，为模板中要用的词汇提供一种对应的翻译。这样，当你要修改、增加或删除一个翻译时，你就不必修改你的模板了。

翻译的配置

模板在默认情况下是不会被翻译的，你需要在执行其他命令之前在 setting.yml 文件中激活模板翻译特性才能自动翻译模板。见例 13—11 所示。

例 13—11 在 myapp/config/settings.yml 中激活界面翻译

```
all:
  .settings:
    i18n: on
```

运用翻译辅助函数

我们假设你要创建一个包含英文和法文的网站，而英文是网站的默认语言。在翻译网站前，你也许写了一个象例 12—12 所示的模板。

例 13—12 单一语言的模板

```
Welcome to our website. Today's date is <?php echo  
format_date(date()) ?>
```

要在 symfony 中翻译模板中的词汇，它们必须先被识别为文本，然后才能翻译。这个可以通过 I18N 辅助函数组的 `__()`（双下划线）辅助函数做到。因此，你的所有模板在这样的函数调用中都要包含要翻译的词汇。例如，例 13—12 可以改成例 13—13 中的样子（在你看了本章后面的“处理复杂的翻译需要”一节后，将有更好的方法调用本例中的翻译辅助函数）。

例 13—13 一个可用于多语言环境的模板

```
<?php use_helper('I18N') ?>  
  
<?php echo __('Welcome to our website.') ?>  
<?php echo __('Today's date is ') ?>  
<?php echo format_date(date()) ?>
```

TIP 如果你的应用程序中每页都要使用 I18N 辅助函数组，那么在 `settings.yml` 文件中设定 `standard_helpers` 参数应该是一种更好的方法，这样你可以避免在每个模板中都重复地写 `use_helper('I18N')`。

运用字典文件

每次调用 `__()` 函数时，symfony 就根据用户当前的 `culture`，在相应的字典中对变量进行翻译。如果找到合适的翻译结果，就会将结果返回并显示出来。所以说，用户界面翻译依赖于字典文件。

字典文件是用 XLIFF 格式写成的（XLIFF 是 XML Localization Interchange File Format 的缩写，中文译作“XML 本地化交换文件格式”），文件名则根据 `messages.[语言代码].xml` 文件中定义的模式来命名，文件路径在应用程序的 `i18n/` 目录下。

XLIFF 是一种基于 XML 的标准格式。因为它广为人知，所以你可以用第三方翻译工具去处理和翻译你站点中的所有文本。翻译公司知道如何添加一个 XLIFF 翻译文件去处理这些文件和翻译整个站点。

TIP 除了 XLIFF 标准，symfony 还支持 `gettext`、MySQL、SQLite 和 Creole 等的字典翻译。更多的信息和相应的配置方法请参考 API 文档。

例 13—14 中的 messages.fr.xml 展示了 XLIFF 语法的使用，利用它可以将例 13—13 翻译成法语。

例 13—14 一个 XLIFF 字典，文件名为 myapp/i18n/messages.fr.xml

```
[xml]
<?xml version="1.0" ?>
<xliff version="1.0">
  <file original="global" source-language="en_US"
datatype="plaintext">
    <body>
      <trans-unit id="1">
        <source>Welcome to our website.</source>
        <target>Bienvenue sur notre site web.</target>
      </trans-unit>
      <trans-unit id="2">
        <source>Today's date is </source>
        <target>La date d'aujourd'hui est </target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

source-language 属性必须总是包含默认 culture 值的完整 ISO 代码。每个翻译都用一个包含了唯一 id 属性的 trans-unit 标记来定义。

对于默认的用户 culture 值（设定为 en_US），词汇不会被翻译，而且显示的是__()调用的原始参数值。这样例 13—13 的结果就和例 13—12 的类似了。但是如果 culture 值变成了 fr_FR 或 fr_BE，显示的就是 messages.fr.xml 中翻译后的值了，结果如例 13—15 所示。

例 13—15 一个翻译后的模板

```
Bienvenue sur notre site web. La date d'aujourd'hui est
<?php echo format_date(date()) ?>
```

如果还要增加其他的语言版本，只需将新的翻译文件 messages.XX.xml（XX 是语言代码）放入同一个目录中就可以了。

管理字典

如果你的 messages.XX.xml 文件长得无法阅读，你可以按主题将这个文件分成多个字典文件。例如，你可以在应用程序的 i18n/目录中将 messages.fr.xml 文件分成以下三个：

- navigation.fr.xml
- terms_of_service.fr.xml
- search.fr.xml

注意，如果在默认的消息文件 `messages.XX.xml` 中找不到翻译，那么你每次要调用 `__()` 时，都必须用它的第三个参数指明使用那个字典。例如，如果要显示一个用 `navigation.fr.xml` 字典翻译的字符串，你应该用以下语法：

```
<?php echo __('Welcome to our website', null, 'navigation') ?>
```

另一种管理字典文件的方法是按模块分割。你可以在每个模块的 `modules/[模块名]/i18n/` 目录中放一个 `messages.XX.xml` 翻译文件，而不是在整个应用程序中用一个统一的翻译文件。这可以让模块独立于整个应用程序，如果你想重用模块（例如第 17 章介绍的 `plug-in`），这样做就很有必要。

处理需要翻译的其他元素

需要翻译的其他元素还有如下一些：

- 根据用户 `culture` 的不同，图片，文本文档或其他类型的资源也会变化。最好的例子是一个使用特殊排版的文本，它实际上是一幅图片。为此，我们可以创建一个以用户 `culture` 值命名的子目录：

```
getCulture().'/myText.gif') ?>
```

- 来自验证文件的错误信息会由一个 `__()` 方法自动输出，所以你需要将对应的翻译加入到字典中以便输出对应的翻译。
- 默认的 `symfony` 页面（`page not found`, `internal server error`, `restricted access` 等）都是用英语写的，你可以在 `i18n` 应用程序中重写这些页面。你应该在应用程序中创建了自己的 `default` 模块，并且在模板中使用了 `__()` 方法。你可以在第 19 章中找到如何定制这些页面的方法。

处理复杂的翻译需求

只有当 `__()` 的参数是一个完整的句子时，翻译才有意义。但是，当你的格式或变量名中混合了其它词汇时，你就会想把句子分割成多个小段，这样辅助函数就遇到了无意义的词汇。幸亏，`__()` 辅助函数提供了一个基于标记的替换功能，可以帮助你构造一个易被其它翻译器处理的有意义的词典。对于 HTML 格式，只需把它留给辅助函数处理即可。例 13-16 给出了示例。例 13-16 翻译包含代码的语句。

```
// 原文
```



```
Welcome to all the <b>new</b> users.<br />
There are <?php echo count_logged() ?> persons logged.
```

// 拆分文本翻译的较差方法

```
<?php echo __('Welcome to all the') ?>
<b><?php echo __('new') ?></b>
<?php echo __('users') ?>.<br />
<?php echo __('There are') ?>
<?php echo count_logged() ?>
<?php echo __('persons logged') ?>
```

// 拆分文本翻译的较好方法

```
<?php echo __('Welcome to all the <b>new</b> users') ?> <br />
<?php echo __('There are %1% persons logged', array('%1%' =>
count_logged())) ?>
```

本例中，标记是%1%，当然也可以是其它任何标记，因为翻译辅助函数使用的替换函数是 `strtr()`。

翻译中的一个常见问题是复数的使用。结果数值的不同会引起文本的变化，但在不同的语言里，变化的方式却不一定相同。例如，例 13—16 中，如果 `count_logged()` 返回 0 或 1 时，最后句子就不正确了。你可以根据函数的返回值进行测试以选择哪个句子能正确使用，但那意味着要增加大量的代码。另外，不同的语言有不同的语法规则，复数的词尾变化规则非常复杂。因为这个问题非常普遍，`symfony` 提供了一个辅助函数 `format_number_choice()` 来处理它，例 13—17 是使用这个辅助函数的示例。

例 13—17 根据参数值翻译语句

```
<?php echo format_number_choice(
    '[0]Nobody is logged|[1]There is 1 person logged|(1,+Inf]There
are%1% persons logged', array('%1%' => count_logged()),
count_logged()) ?>
```

第一个参数给出了文本的多种可能值。第二个参数是替换模式（就象 `__()` 辅助函数一样），该参数是可选的。第三个参数是一个返回数值的函数，通过这个函数的返回结果确定采用哪个文本。

信息/字符串选项用管道符号(|)分割，后跟一个可接受值的数列，该数列的语法如下：

- [1,2]: 可接受值在 1 和 2 之间，包含 1 和 2。
- (1,2): 可接受值在 1 和 2 之间，不包含 1 和 2。
- {1,2,3,4}: 只接受定义在集合中的值。

- [-Inf, 0): 接受小于 0 且大于等于负无穷大的值。

用方括号或圆括号括起的任何非空组合都是可接受的。为了能正确地翻译，这个信息必须精确地在 XLIFF 文件中定义。例 13—18 给出了示例。

例 13—18 包含 `format_number_choice()` 参数的 XLIFF 字典

```
...
<trans-unit id="3">
  <source>[0]Nobody is logged|[1]There is 1 person
logged|(1,+Inf]There are%% persons logged</source>
  <target>[0]Personne n'est connecté|[1]Une personne est
connectée|(1,+Inf]Il y a %% personnes en ligne</target>
</trans-unit>
...
```

SIDEBAR 关于字符集

在模板中处理国际化内容常常会出现字符集问题。如果你用本地化字符集，那么每当用户改变 `culture` 时，你就要改变字符集。另外，用一种字符集编写的模板通常不能很好地在另一种字符集环境下显示。

所以一旦你开始处理多国家和语言的应用时，你的所有模板都应保存为 UTF-8 字符格式，所有的界面也应声明为使用 UTF-8 字符集。如果你一直使用 UTF-8，将会给你解决许多麻烦。

在 `settings.yml` 文件中定义了整个 `symfony` 应用程序要使用的字符集。修改这个参数会修改所有回应的 `content-type` 头信息的值。

```
all:
  .settings:
    charset: utf-8
```

在模版外调用翻译辅助函数

页面中显示的文本不一定都来自于模版。这也是你为什么要经常在应用中的动作、过滤器、模型类等部分调用 `__()` 辅助函数的原因。例 13—19 显示了如何通过获取上下文环境中 `l18n` 对象的当前实例，在动作中调用辅助函数。

例 13—19 在动作中调用 `__()` 辅助函数。

```
$this->getContext()->getl18n()->__($text, $args, 'messages');
```

总结

如果你掌握了如何运用用户 culture 值，那么你就能在 web 应用中轻松地处理国际化和本地化问题了。辅助函数将自动输出经过正确格式化的数据，而数据库中的本地化数据也会被看作一个简单表的一部分。至于界面翻译，辅助函数 `__()` 和 XLIFF 字典可以确保你用最少的工作量获得最大的灵活性。

第 14 章 - 生成器

很多程序基于存储在数据库里的数据并提供访问这些数据的界面。symfony 能够自动完成根据 Propel 对象生成数据处理模块这样的重复任务。如果模型定义的好，symfony 甚至可以自动生成整个网站后台。本章将会介绍 symfony 的两种生成器：脚手架生成器和管理生成器。其中后者依赖于一个特别的语法复杂的配置文件，所以这一章的大部分篇幅会用来介绍管理生成器的各种用法。

基于模型生成代码

在 web 应用程序里，数据访问操作可以归结为以下几类：

- 新增 (Creation) 一条记录
- 取得 (Retrieval) 记录
- 更新 (Update) 一条记录 (并且修改它的字段)
- 删除 (Deletion) 一条记录

这些操作很常见，它们有一个专门的缩写：CRUD。很多页面都可以简化成其中之一。例如，在论坛程序里，最新帖子列表就是一个取得记录的过程，回帖子是一个新增过程。

针对一个表的 CRUD 操作制作基本的动作 (action) 和模板在 web 程序里会经常出现。在 symfony 里，模型层包含的信息足够生成 CRUD 操作代码的需要，这样可以加快早期的后台界面开发。

所有的基于模型的代码生成任务都会建立整个模块，只要通过类似下面的一行 symfony 命令就可以完成：

```
> symfony <任务名> <应用程序名> <模块名> <类名>
```

代码生成任务包括 `propel -init-crud`、`propel -generate-crud` 和 `propel -init-admin`。

脚手架与管理界面

开发应用程序的过程中，代码生成有两种不同的用途：

- 脚手架是给定表 CRUD 操作所需的基本结构（动作与模板）。它的代码是最小化的，因为它需要成为后续开发的指导。它是起步的基础，经过修改后才能满足你的逻辑与表现的需求。脚手架大多用在开发阶段，用来提供数据库的 web 访问界面，建立一个原型，或者以此为基础制作一个与某个表相关的模块。

- 管理界面是专门用于数据处理的界面，多用于后台管理。管理界面与脚手架的不同点是它的代码不是用来手动修改的。它们可以被定制，扩展或者通过配置或继承进行装配。它们的外观很重要，它需要有排序，分页，还有过滤功能。管理界面可以作为软件的成品交给客户。

symfony 命令行用 crud 代表脚手架，用 admin 代表管理界面。

初始化或生成代码

symfony 有两种生成代码的方式：通过继承(init)或者代码生成(generate)。

你可以初始化一个模块，也就是建立空的继承自框架的类。这样可以避免动作(action)和模板的 PHP 代码被修改。如果你的数据结构还没最终确定或者你只需要一个快速的数据库接口来操作数据，这个功能很有用。运行时执行的代码不在你的应用程序里，而是在缓存里。这类生成任务的命令行任务名以 propel -init- 开头。

初始化的动作(action)代码是空的。例如，一个初始化的 article 模块的代码可能会是这样：

```
class articleActions extends autoarticleActions
{
}
```

另一方面，你也可以生成动作(action)和模板的代码，这样可以修改它们。这样生成的模块不依赖于框架，并且不能被配置文件修改。这种生成任务的命令行任务名以 propel -generate- 开头。

由于脚手架是后续开发的基础，所以最好生成一个脚手架。另外，管理界面需要能够通过配置文件方便的修改，模型变化的情况下也要能够使用。所以管理界面只能够初始化。

数据模型的例子

本章的所有演示 symfony 生成器功能的例子都基于这个简单的例子，这个例子可能会让你回想起第 8 章。这就是那个有名的博客应用程序，包含 Article 和 Comment 两个类。例 14-1 是它的设计(schema)。

例 14-1 - 博客应用程序的 schema.yml 文件

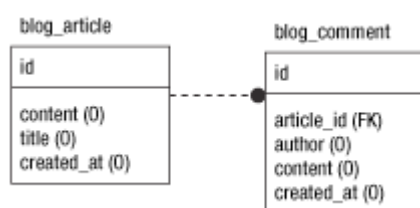
```
propel:
  blog_article:
    _attributes: { phpName: Article }
    id:
```

```

    title:          varchar(255)
    content:         longvarchar
    created_at:
blog_comment:
  _attributes: { phpName: Comment }
  id:
  article_id:
  author:          varchar(255)
  content:         longvarchar
  created_at:

```

图 14-1 - 例子的数据模型



代码生成并不会对设计(schema)的建立有什么特别的要求。`symfony` 会使用已有的设计(schema)，解释它的属性并生成脚手架或管理界面。

TIP 学习本章要达到最好的效果，你需要跟着这些例子去做。如果你按照这些例子中的每一个步骤去做，你会更好的理解 `symfony` 生成的代码以及它们的作用。所以建议你跟我们一起从刚才的这个例子做起，在一个数据库里面建立 `blog_article` 和 `blog_comment` 两个表，输入一些测试用的数据。

脚手架

脚手架在开发初期很有用。只要一条简单的命令，`symfony` 就能根据某个表的信息建立整个模块。

生成脚手架

根据 `Article` 模型类生成 `article` 模块，输入下面的命令：

```
> symfony propel-generate-crud myapp article Article
```

`symfony` 会读取 `schema.yml` 里 `Article` 类的定义并根据这些定义在 `myapp/modules/article/` 目录建立一些模板和动作(action)。

生成的模块包括三个视图。`list` 视图，它是默认的视图，在浏览 http://localhost/myapp_dev.php/article_dev.php/article 的时候会显示 `blog_article` 表的记录，如图 14-2 所示。

图 14-2 - article 模块的 list 视图

article

Id	Title	Content	Created at
1	Welcome to the symfony weblog!	This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.	2006-11-12 20:20:25
2	Life is beautiful	The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours?	2006-11-12 20:20:25

create

点击文章 id 会显示 show 视图。这个页面显示的是这个记录的详细情况，如图 14-3。

图 14-3 - article 模块的 show 视图

Id:

1

Title:

Welcome to the symfony weblog!

Content:

This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.

Created at:

2006-11-12 20:20:25

edit list

点击 edit 链接可以修改这篇文章，或者在 list 视图点击 create 链接新增一篇文章，会显示 edit 视图，如图 14-4 所示。

用这个模块，你可以新增文章，也可以修改或者删除已有的文章。生成的代码是未来开发的良好基础。例 14-2 列出了生成的新模块的动作(action)和模板代码

图 14-4 - article 模块的 edit 视图

Title:

Content:

This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as

↑

≡

↓

save

delete

cancel

例 14-2 - 生成的 CRUD 元素，在 myapp/modules/article/目录下

```
// 在 actions/actions.class.php 文件里
index // 转到下面的 list 动作
```

```
list           // 显示表里面的所有记录
show          // 显示一个记录的所有字段
edit          // 显示一个修改一条记录的表单
update        // 被 edit 动作调用的动作
delete        // 删除一条记录
create        // 新增一条记录
```

```
// 在 templates/ 目录下
editSuccess.php // 记录修改表单(edit 视图)
listSuccess.php // 显示所有的记录 (list 视图)
showSuccess.php // 记录详情 (show 视图)
```

这些动作和模板的逻辑很简单明白，把它们列出来就能说明一切。例 14-3 里是一部分生成的动作类的代码。

例 14-3 - 生成的动作类，位于
myapp/modules/article/actions/actions.class.php

```
class articleActions extends sfActions
{
    public function executeIndex()
    {
        return $this->forward('article', 'list');
    }

    public function executeList()
    {
        $this->articles = ArticlePeer::doSelect(new Criteria());
    }

    public function executeShow()
    {
        $this->article = ArticlePeer::retrieveByPk($this->getRequestParameter('id'));
        $this->forward404Unless($this->article);
    }
    ...
}
```

按照你的需求修改生成的代码，重复对所有需要交互的表进行 CRUD 生成，这样你就有了一个可以工作的基本的应用程序了。生成脚手架大大加快了开发速度，让 symfony 来为你干脏活，你只要专注与界面还有细节。

初始化脚手架

初始化一个脚手架在你需要检查是否能够访问数据库里的数据的时候很有用。它建立起来很快，一旦你确定一切工作正常，删除它也很快。

初始化一个 Propel 脚手架，它将建立一个负责处理 Article 模型类的数据的名叫 article 的模块，输入下面的命令：

```
> symfony propel-init-crud myapp article Article
```

你可以通过默认的动作(action)访问 list 视图：

http://localhost/myapp_dev.php/article

结果页面和生成的脚手架完全一样。你可以把它们作为数据库的简单 web 界面。

如果你去看新建立的 article 模块的 action.class.php 文件，你会发现他是空的：所有的东西都是继承自自动生成的类。模板也一样，templates/ 目录里面没有模板文件。初始化的动作和模板后面的代码与生成的脚手架的代码与模板一样，不过它们是放在应用程序缓存里 (myproject/cache/myapp/prod/module/autoArticle/)。

在应用程序的开发过程中，开发者初始化脚手架来处理数据，而不去管界面。初始化的代码不是用来定制的；初始化的脚手架可以看作 PHPmyadmin 的简单替代品来管理数据。

管理界面

根据由 schema.yml 文件生成的模型类，symfony 可以生成更加先进，更适合你的应用程序的后台模块。完全可以仅仅使用生成的管理界面制作整个网站后台。本节例子将在 backend 应用程序中增加管理模块。如果你的项目没有 backend 应用程序，请用 init-app 任务建立 backend 应用程序的框架：

```
> symfony init-app backend
```

管理界面模块通过一个叫 generator.yml 的特殊配置文件的设置来表现模型，通过这个文件可以完全控制生成的组件还有外观。之前介绍的模块机制（布局，验证，路由，自定义配置，自动载入，等）也可以在这些生成的管理模块中使用。你还可以重写生成的动作或者模板来给生成的管理界面增加你自己的功能，不过修改 generator.yml 就能够达到大多数的需求，只有需求很特殊的时候才使用 PHP 代码。

初始化管理界面模块

在 symfony 里建立管理界面是以模块为单位的。使用 propel-init-admin 任务生成基于 Propel 对象的模块，这与初始化脚手架的语法类似：

> symfony propel-init-admin backend article Article

这条命令会在 backend 应用程序里根据 Article 类的定义新建一个 article 模块，可以通过下面的网址访问：

<http://localhost/backend.php/article>

生成的模块的界面如图 14-5、14-6 所示，这样专业的界面，直接用在商业应用程序里也没什么不妥。

图 14-5 - backend 应用程序的 article 模块的 list 视图

article list

Id	Title	Content	Created at
1	Welcome to the symfony weblog!	This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.	December 1, 2006 1:17 PM
2	Life is beautiful	The purpose of a weblog is usually to talk about one's mood. Mine is great today. How is yours?	December 1, 2006 1:17 PM
2 results			

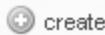


图 14-6 - backend 应用程序的 article 模块的 edit 视图

edit article

Title:


Welcome to the symfony \


Content:


This is the first post of this weblog. Honestly, it is just a test to check if it works fine. Please comment it as much as you like.


Created at:


12/1/06



 list

 save

 save and add

 delete

现在脚手架与管理界面的外观看上去并没有什么太大的差别，不过管理界面的可配置性可以使你不用写一行 PHP 代码就能够给这个基本布局增加很多功能。

NOTE 管理界面模块只能初始化（不能生成）。

浏览生成的代码

article 管理界面模块的代码，位于 `apps/backend/modules/article/` 目录，由于刚刚初始化，还是空的。最好的查看生成代码的方法是在浏览器里访问它，然后去看 `cache/` 目录的内容。例 14-4 列出了缓存中生成的动作与模板。

例 14-4 - 生成的管理界面元素，位于 `cache/backend/ENV/modules/article/`

```
// 位于 actions/actions.class.php
create          // 转到 edit
delete          // 删除一条记录
edit            // 显示修改记录的表单
                // 并且处理提交的表单
index           // 转到 list
list            // 显示表里的所有记录
save            // 转到 edit
```

```
// 位于 templates/
_edit_actions.php
_edit_footer.php
_edit_form.php
_edit_header.php
_edit_messages.php
_filters.php
_list.php
_list_actions.php
_list_footer.php
_list_header.php
_list_messages.php
_list_td_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php
editSuccess.php
listSuccess.php
```

从这里可以看出生成的管理界面模块主要由 `edit` 和 `list` 两个视图组成。如果你去看它们的代码，你会发现它们非常模块化，可读性强，并且容易扩展。

generator.yml 配置文件介绍

脚手架与管理界面的主要不同（除了管理界面不包括 `show` 动作）是管理界面依赖于 `generator.yml` YAML 配置文件里的参数。要看新建的管理界面模块的默认

配置文件，打开位于 backend/modules/article/config/ 的 generator.yml 文件，文件内容如例 14-5 所示。

例 14-5 - 默认的生成器配置，位于 backend/modules/article/config/generator.yml

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:   Article
    theme:          default
```

这个配置足够生成一个基本的管理界面。所有的定制参数都要加在 param 键下的 theme 这一行之后（也就是说所有在 generator.yml 文件里增加的内容至少以 4 个空格开头）。例 14-6 是一个典型的定制后的 generator.yml 文件。

例 14-6 - 典型的完全定制的生成器配置

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:   Article
    theme:          default

  fields:
    author_id:     { name: Article author }

  list:
    title:         List of all articles
    display:       [title, author_id, category_id]
    fields:
      published_on: { params: date_format=' dd/MM/yy' }
    layout:        stacked
    params:        |
      %%is_published%%<strong>%%=title%%</strong><br /><em>by
%%author%%
      in %%category%%
    (%%published_on%%)</em><p>%%content_summary%%</p>
    filters:       [title, category_id, author_id, is_published]
    max_per_page:  2

  edit:
    title:         Editing article "%%title%%"
    display:
      "Post":      [title, category_id, content]
```

```

    "Workflow": [author_id, is_published, created_on]
fields:
    category_id: { params: disabled=true }
    is_published: { type: plain }
    created_on: { type: plain, params: date_format='dd/MM/yy' }
    author_id: { params: size=5 include_custom=>> Choose an
author << }
    published_on: { credentials: }
    content: { params: rich=true tinymce_options=height:150 }

```

接下来将会详细解释这个配置文件里用到的所有参数。

生成器配置

生成器配置文件功能很强，可以用多种方式改变生成的管理界面。但是强大的功能需要付出代价：它的语法描述很长，难于阅读和学习，所以本章是整本书最长的章节之一。symfony 网站上提供了一个可以帮助你学习生成器配置的好东西：管理界面简要参考，如图 14-7 所示。下载网址(<http://www.symfony-project.com/uploads/assets/sfAdminGeneratorRefCard.pdf>)，建议在阅读本章的时候把它打印出来放在手边。

本节的例子会调整 article 管理界面模块还有基于 Comment 类定义的 comment 管理界面模块。通过 `propel-init-admin` 建立后者：

```
> symfony propel-init-admin backend comment Comment
```

图 14-7 - 管理界面简要参考

[illegible]

```

    fields:
      title:          { name: Article Title, type: textarea_tag,
params: class=foo }
      content:        { name: Body, help: Fill in the article body }

```

除了这个针对所有视图的默认字段定义，你还可以在指定的视图里（list 与 edit）重写字段设置，如例 14-8 所示。

例 14-8 - 为每个视图重写字段设置

```

generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Article
    theme:        default

    fields:
      title:      { name: Article Title }
      content:    { name: Body }

  list:
    fields:
      title:      { name: Title }

  edit:
    fields:
      content:    { name: Body of the article }

```

基本原则是：所有 fields 键下的模块全局设置可以在特定的视图里（list 与 edit）重写。

增加显示的字段

在 fields 部分定义的字段可以在各视图中显示、隐藏、排列或者按照不同的方式分组。display 键就用于这个目的。例如，使用例 14-9 的代码安排 comment 模块的字段。

例 14-9 - 选择显示的字段，位于 modules/comment/config/generator.yml

```

generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default

```

```

fields:
  article_id:      { name: Article }
  created_at:      { name: Published on }
  content:          { name: Body }

list:
  display:          [id, article_id, content]

edit:
  display:
    NONE:            [article_id]
    Editable:        [author, content, created_at]

```

list 视图会显示 3 个字段，如图 14-8 所示，edit 表单将会显示分成两组的 4 个字段，如图 14-9 所示。

图 14-8 - comment 模块的 list 视图的自定义字段设置

comment list

Id Article Body		
1	1	Well, if this comment displays, it means that your weblog does work...
2	1	Thank you for your feedback. It really helps to see people understanding you.
3	2	Mine is not bad either. I'd like to see more deers, though.
4	2	How can you be so positive? There are so many subjects to worry about out there!
5	2	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results		








 create

图 14-9 - 分组显示 comment 模块的 edit 视图里的字段

edit comment

Article:	1 
Editable	
Author:	<input type="text" value="Anonymous"/>
Body:	<div>Well, if this comment displays, it means that your weblog does work...</div>
Published on:	<input type="text" value="12/1/06"/> 

 list |  save |  save and add |  delete

有两种方式设置 display:

- 选择要显示的字段，按照出现的顺序排列，放在简单的数组里——如前一个 list 视图。
- 分组显示，使用一个关联数组，组的名字作为键名，或者使用 NONE 代表没有名字的组。值仍然是有序的字段名字的数组。

TIP 默认情况，主键字段不会在任何一个视图里出现。

自定义字段

事实上，generator.yml 里定义的字段甚至可以不必对应 schema 里定义的字段。只要相关的类提供一个自定的 getter，它就可以在 list 视图作为一个字段显示；如果同时有 getter 和 setter，它也可以用在 edit 视图里作为一个字段显示。例如，你可以用 getNbComments() 扩展 Article 模型，如例 14-10。

例 14-10 - 在模块中添加自定义的获取方法，位于 lib/model/Article.class.php

```
public function getNbComments()
{
    return $this->countComments();
}
```

那么 nb_comments 就可以作为一个生成的模块的字段（注意 getter 使用的是驼峰命名法 camel Case 版本的字段名作为方法名），如例 14-11。

例 14-11 - 自定义获取方法为管理界面模块提供额外的字段， 位于 `backend/modules/article/config/generator.yml`

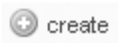
```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:   Article
    theme:         default

    list:
      display:     [id, title, nb_comments, created_at]
```

article 模块的 list 视图的结果如图 14-10 所示。

图 14-10 - article 模块的 list 视图的自定义字段

article list

Id Title		Nb comments Created at	
1	Welcome to the symfony weblog!	2	December 1, 2006 1:17 PM
2	Life is beautiful	3	December 1, 2006 1:17 PM
2 results			
			

自定义字段除了返回原始数据之外还可以返回 HTML 代码。例如，你可以像例 14-12 那样用 `getArticleLink()` 方法扩展 `Comment` 类。

例 14-12 - 添加自定义的获取方法返回 HTML 代码，位于 `lib/model/Comment.class.php`

```
public function getArticleLink()
{
    return link_to($this->getArticle()->getTitle(),
        'article/edit?id='.$this->getArticleId());
}
```

你可以在 `comment/list` 视图里使用与例 14-11 里相同的语法把这个新 `getter` 作为自定义字段。请看例 14-13，还有图 14-11 里的结果，`getter` 返回的 HTML 代码（到文章的超链接）出现在第 2 个字段而不是原来的主键。

例 14-13 - 自定义的返回 HTML 代码的 `getter` 也可以作为自定义字段，位于 `modules/comment/config/generator.yml`

```

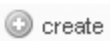
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default

  list:
    display:      [id, article_link, content]

```

图 14-11 - comment 模块的 list 视图的自定义字段

comment list

Id	Article link	Body
1	Welcome to the symfony weblog!	Well, if this comment displays, it means that your weblog does work...
2	Welcome to the symfony weblog!	Thank you for your feedback. It really helps to see people understanding you.
3	Life is beautiful	Mine is not bad either. I'd like to see more deers, though.
4	Life is beautiful	How can you be so positive? There are so many subjects to worry about out there!
5	Life is beautiful	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results		
		

局部模板字段

模型中的代码必须与表现无关。之前的 `getArticleLink()` 这个方法没有遵循这个层分离原则，因为它的代码里包含了视图代码。要用正确的方法达到相同的目的，必须要把 HTML 输出的代码放在一个自定义的局部模板里。幸运的是，只要在字段名前加上一个下划线，管理界面生成器就会认为这是一个局部模板字段。这样，例 14-13 中的 `generator.yml` 需要被改成例 14-14 中列出来的内容。

例 14-14 - 局部模板能用于附加列 — 使用 `_` 前缀

```

list:
  display:      [id, _article_link, created_at]

```

为了让这个配置能够正常工作，还需要在 `modules/comment/templates/` 目录里面增加一个名为 `_article_link.php` 的局部模板，内容如例 14-15 所示。

例 14-15 - 为 article 模块的 edit 视图自定义局部模板的例子，位于 `modules/comment/templates/_article_link.php`

```
<?php echo link_to($comment->getArticle()->getTitle(),  
'article/edit?id=' . $comment->getArticleId()) ?>
```

请注意局部模板字段的局部模板中可以通过以类命名的变量（此例中是 `$comment`）访问到当前对象。例如，一个为名为 `UserGroup` 的类建立的模块，在它的局部模板里可以通过 `$user_group` 变量访问到当前对象。

上面例子的结果与图 14-11 中的一样，不过这样作符合层分离原则。如果你习惯遵循这一原则，你的应用程序会更加容易维护。

给一个局部模板字段自定义参数的方法与普通字段完全一样——在 `field` 键下作出定义。只是要去掉前面的下划线（`_`）——请看例 14-16。

例 14-16 - 局部模板的属性可以在 `field` 键下定义

```
fields:  
  article_link: { name: Article }
```

如果局部模板包含复杂的逻辑，你可以用组件取代它。把 `_` 前缀改成 `~`，你就定义了一个组件字段，如例 14-17 所示。

例 14-17 - 使用 `~` 前缀可以定义组件字段

```
...  
list:  
  display: [id, ~article_link, created_at]
```

在生成的模板里，这个例子的结果会是一个到当前模块的 `articleLink` 组件的调用。

NOTE 自定义字段与局部模板字段可以用在 `list` 视图与 `edit` 视图，还有过滤器里。如果在几个视图里使用同一个局部模板，可以从 `$type` 变量里取得当前环境（`list`，`edit` 或者 `filter`）。

视图定制

如果要修改 `edit` 与 `list` 视图的外观，你可能会尝试修改模板。但是因为他们自动生成的，这并不是一个好主意。你应该使用 `generator.yml` 配置文件，因为它几乎可以完成每件事而不需要太多修改。

修改视图标题

除了自定义字段，`list` 与 `edit` 页面还可以自定义页面标题。例如，如果想自定义 `article` 模块视图的标题，可以用例 14-18 里的配置。`edit` 视图的结果如图 14-12 所示。

例 14-18 - 为每个视图设置自定义标题，位于
backend/modules/article/config/generator.yml

```
list:
  title:      List of Articles
  ...

edit:
  title:      Body of article %%title%%
  display:    [content]
```

图 14-12 - 自定义 article 模块的 edit 视图的标题

Body of article Welcome to the symfony weblog!

Content:

This is the first post of this weblog.
Honestly, it is just a test to check if it
works fine. Please comment it as much
as you like.

list

save

save and add

delete

默认情况会用类名作为标题，很多时候这么作就够了——不过这需要你的类命名比较清楚明白。

TIP generator.yml 里的字符串值里，字段的值可以通过用 %%字段名%% 的方式取得。

增加提示

在 list 与 edit 视图里，可以增加用来描述字段的提示。例如，comment 模块的 edit 视图的 article_id 字段，在 fields 定义下面增加一个 help 属性，如例 14-19 所示。结果见图 14-13。

例 14-19 - 给 edit 视图设置提示，位于
modules/comment/config/generator.yml

```
edit:
  fields:
    ...
    article_id: { help: The current comment relates to this
article }
```

图 14-13 - comment 模块的 edit 视图里的提示

edit comment

Article:

1

▼

The current comment relates to this article

在 list 视图里，提示会显示在表头里，在 edit 视图里，它们会在输入框下面出现。

修改日期格式

可以用 `date_format` 参数指定日期的显示格式，如例 14-20 所示。

例 14-20 - 在 list 视图里设置日期显示格式

```
list:
  fields:
    created_at: { name: Published, params:
date_format='dd/MM' }
```

它的参数与之前介绍过的 `format_date()` 辅助方法一样。

SIDEBAR 管理界面模板是 I18N(国际化)的

所有生成的模板里的文字都经过了自动的国际化处理（例如，在外边包了一个 `__()` 辅助函数调用）。这也就是说，要翻译生成的管理界面很容易，只要在 `apps/myapp/i18n/` 目录的 XLIFF 文件里增加词语的翻译就可以了，详见之前章节的介绍。

list 视图相关的定制

list 视图可以以表格的形式显示记录的细节，或者把所有的细节放在一行显示。它还可以包含过滤器，翻页，还有排序功能。这些功能可以通过配置文件修改，下面将会介绍这些功能。

修改布局

默认情况，list 视图到 edit 视图的链接放在主键字段。如果你回去看图 14-11，你会发现留言列表的 `id` 字段不仅会显示每个留言的主键值，而且会有一个可以让用户访问 edit 视图的超链接。

如果你想要把修改记录的超链接放到其他字段，只要在 `display` 键的这个字段的名字前加一个等于号(=)。例 14-21 告诉我们如何把 `id` 从留言 list 里去

掉，然后把它上面的超链接放到 content 字段上。图 14-14 是这个配置的运行结果的截图。

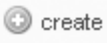
例 14-21 - 改变 list 视图里到 edit 视图超链接的位置，位于 modules/comment/config/generator.yml

```
list:
  display: [article_link, =content]
```

图 14-14 - 移动 comment 模块的 list 视图里的到 edit 的超链接到其他字段

comment list

Article	Body
Welcome to the symfony weblog!	Well, if this comment displays, it means that your weblog does work...
Welcome to the symfony weblog!	Thank you for your feedback. It really helps to see people understanding you.
Life is beautiful	Mine is not bad either. I'd like to see more deers, though.
Life is beautiful	How can you be so positive? There are so many subjects to worry about out there!
Life is beautiful	Why is it always like that, people quarrelling as soon as they have room to express themselves?
5 results	



默认情况，list 视图使用 tabular 布局，这种布局里字段显示成表格的列，前面的图里面就是这种布局。不过你也可以使用 stacked 布局把字段集中到一个字符串里，整个表格只有一列。如果你选择 stacked 布局，你需要设置 params 键的值来定义每一行要怎么显示。例如， 例 14-22 给 comment 模块的 list 视图定义了一个 stacked 布局，结果如图 14-15。

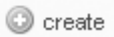
例 14-22 - 在 list 视图里使用 stacked 布局，位于 modules/comment/config/generator.yml

```
list:
  layout: stacked
  params: |
    %=content% <br />
    (sent by %%author%% on %%created_at%% about %%article_link%%)
  display: [created_at, author, content]
```

图 14-15 - comment 模块的 list 视图里使用 stacked 布局

comment list

Published on	Author	Body
Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)		
How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)		
Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)		
5 results		



请注意 `tabular` 布局需要一个由字段名组成的数组作为 `display` 键的值，但是 `stacked` 布局使用 `params` 键为每个记录生成 HTML。不过，在 `stacked` 布局中也需要使用 `display` 数组，决定需要哪些列作为排序的表头。

过滤结果

在 `list` 视图里可以加一些过滤器。使用过滤器，可以减少显示的结果并且更快的找到需要的记录。过滤器在 `filters` 键下设置，它的值是字段名组成的数组。例如，在 `comment` 模块的 `list` 视图增加 `author_id`, `author` 和 `created_at` 字段的过滤器，如例 14-23，会显示一个类似图 14-16 所示的过滤框。要让这段配置正常工作需要在 `Article` 类增加一个 `__toString()` 方法（例如，返回文章标题）。

例 14-23 - 在 `list` 视图里设置过滤器，位于 `modules/comment/config/generator.yml`

```
list:
  filters: [article_id, author, created_at]
  layout: stacked
  params: |
    %=content% <br />
    (sent by %author% on %created_at% about %article_link%)
  display: [created_at, author, content]
```

图 14-16 - `comment` 模块的 `list` 视图里的过滤器

comment list

Published on	Author	Body
Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)		
How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)		
Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)		
5 results		

filters

Article:

Author:

Published on:

symfony 显示的过滤器取决于字段类型：

- 对于文字字段（比如 comment 模块的 author 字段），过滤器会是一个可以使用通配符（*）的文字输入框。
- 对于外键字段（比如 comment 模块里的 article_id 字段），过滤器会是一个显示相关表记录的下拉列表。至于普通的 object_select_tag()，下拉列表的选项是由相关类的 __toString() 方法的结果组成的。
- 对于日期字段（比如 comment 模块的 created_at 字段），过滤器会是一对可以让人选择时间范围的日期控件。
- 对于布尔字段，过滤器会是一个包含 true、false 和 true or false 选项的下拉列表——最后一个值会重设过滤器。

正如你可以在列表中使用局部模板，你也可以在过滤器中使用局部模板来实现一个 symfony 处理不了的过滤器。例如，假设 state 字段只可能有两个取值（open 与 closed），但是由于某些原因你直接在字段里存放这些值而不是使用表关联。symfony 会自动给这个字段(string 类型) 一个文字搜索，不过你想要的可能是下拉列表。这用局部模板很容易实现。例 14-24 是实现的方法。

例 14-24 - 使用局部模板过滤器

```
// 定义局部模板，位于 templates/_state.php
<?php echo select_tag('filters[state]', options_for_select(array(
    '' => '',
    'open' => 'open',
    'closed' => 'closed',
), isset($filters['state']) ? $filters['state'] : '')) ?>
```

```
// 在过滤器列表里增加局部模板过滤器，位于 config/generator.yml
list:
  filters:      [date, _state]
```

注意这里用到的局部模板可以访问到`$filters` 这个变量，这对取得过滤器当前值很有用。

还有一个选项对搜索空值很有用。假设你想找所有没有作者的留言，问题是如果你不填写作者过滤器的输入框，`symfony` 会忽略它。解决办法是把字段的 `filter_is_empty` 属性设置成 `true`，如例 14-25 所示，这样会多显示一个复选框让你可以寻找空值如图 14-17。

例 14-25 - 给 `list` 视图的 `author` 过滤器增加空值选项

```
list:
  fields:
    author:  { filter_is_empty: true }
  filters:  [article_id, author, created_at]
```

图 14-17 - 允许过滤空的 `author` 的值



The screenshot shows a 'filters' section with three filter groups. The first group, 'Article:', has a dropdown menu. The second group, 'Author:', has a text input field and a checkbox labeled 'is empty'. The third group, 'Published on:', has two date range pickers. At the bottom, there are 'reset' and 'filter' buttons.

列表排序

在 `list` 视图里，表头是可以改变列表排列顺序的超链接，如图 14-18 所示。这些表头在 `tabular` 和 `stacked` 布局中都会显示。点击这些链接会用 `sort` 参数重新载入页面从而改变列表显示顺序。

图 14-18 - `list` 视图的表头可以控制排列顺序

List of Articles

Id	Title	nb comments	Created at
1	Welcome to the symfony weblog!	2	December 1, 2006 1:17 PM
2	Life is beautiful	3	December 1, 2006 1:17 PM
2 results			



可以重用这里的参数排列来指向一个直接按照某个字段排列的列表：

```
<?php echo link_to('Comment list by date',  
'comment/list?sort=created_at&type=desc' ) ?>
```

也可以在 `generator.yml` 里为 `list` 视图设置一个默认的 `sort` 排列顺序。格式如例 14-26 所示。

例 14-26 - 给 `list` 视图一个默认的排列顺序

```
list:  
  sort:  created_at  
  # 另一种格式，指定一个特定的排列方向  
  sort:  [created_at, desc]
```

NOTE 只有实际的字段可以转化成排序链接——自定义字段或者局部模板字段不可以。

自定义分页

自动生成的管理界面可以有效的处理大规模的表，因为 `list` 自动进行分页。当表的记录数大于每页最大记录数的时候，翻页控件会出现在列表的底部。例如，图 14-19 显示的是一个有 6 条记录的留言表，但是每页只显示 5 条记录。分页能提高显示速度，因为只有需要显示的记录才会从数据库里读取，它还提高了易用性，因为即使是有上百万条记录的表都可以用生成的管理界面进行管理。

图 14-19 - 分页控件会显示在长的列表里

comment list

Published on	Author	Body
Well, if this comment displays, it means that your weblog does work... (sent by Anonymous on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Thank you for your feedback. It really helps to see people understanding you. (sent by Myself on December 1, 2006 1:17 PM about Welcome to the symfony weblog!)		
Mine is not bad either. I'd like to see more deers, though. (sent by John Doe on December 1, 2006 1:17 PM about Life is beautiful)		
How can you be so positive? There are so many subjects to worry about out there! (sent by Anonymous on December 1, 2006 1:17 PM about Life is beautiful)		
Why is it always like that, people quarrelling as soon as they have room to express themselves? (sent by Myself on December 1, 2006 1:17 PM about Life is beautiful)		
6 results		
create		

可以通过指定 `max_per_page` 参数来自定义每页显示的记录条数:

```
list:
    max_per_page: 5
```

使用 join 加快页面速度

默认情况, 管理界面生成器使用简单的 `doSelect()` 方法来获取一个列表。不过, 如果在列表里面使用了关联对象, 数据库查询次数就会飞速增长。例如, 如果你想在留言列表里显示文章的名字, 每个留言需要多做一次查询来取得相关的 `Article` 对象。所以需要强制分页系统使用 `doSelectJoinXXX()` 方法来优化查询次数。这可以通过 `peer_method` 来设置。

```
list:
    peer_method: doSelectJoinArticle
```

第 18 章详细解释了 `join` 的概念。

edit 视图相关定制

在 `edit` 视图里, 用户可以修改指定记录的每一个字段的值。`symfony` 根据字段的数据类型确定表单控件的类型, 然后生成一个 `object_*_tag()` 辅助函数, 并将当前对象与控件属性传给这个辅助函数。例如, 如果文章 `edit` 视图配置里设置了用户可以编辑 `title` 字段:

```
edit:
    display: [title, ...]
```

那么由于 `schema` 里字段的类型是 `varchar`, `edit` 页面会显示一个用来修改 `title` 的普通的文字输入框。

```
<?php echo object_input_tag($article, 'getTitle') ?>
```

修改表单控件类型

默认的数据类型到表单控件的转换规则如下：

- integer, float, char, varchar(size)类型的字段会在 edit 视图里以 object_input_tag() 的形式出现。
- longvarchar 类型的字段会以 object_textarea_tag() 的形式出现。
- 外键字段会以 object_select_tag() 的形式出现。
- boolean 类型的字段会以 object_checkbox_tag() 的形式出现。
- timestamp 与 date 类型的字段会以 object_input_date_tag() 的形式出现。

你可能会想重写这些规则为某个字段指定一个特殊的表单控件类型。要做到这点，需要设置 fields 定义下的字段 type 参数，指定一个表单辅助函数的名字。生成的 object_*_tag() 的属性，可以通过 params 参数来修改。如例 14-27 所示。

例 14-27 - 在 edit 视图里设置一个自定的表单控件类型与参数

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default

  edit:
    fields:
      id:          { type: plain }
                  ## 表单控件不可编辑
      author:      { params: disabled=true }
                  ## 文本编辑框(object_textarea_tag)
      content:     { type: textarea_tag, params: rich=true
css=user.css tinymce_options=width: 330 }
                  ## 下拉列表(object_select_tag)
      article_id:  { params: include_custom=Choose an article }
      ...
```

params 参数会作为选项传给 object_*_tag()。例如，之前的 article_id 的 params 定义会产生如下的模板：

```
<?php echo object_select_tag($comment, 'getArticleId',  
'related_class=Article', 'include_custom=Choose an article') ?>
```

这就是说 edit 视图里的表单辅助函数的所有参数都可以通过配置文件定制。

处理局部模板字段

局部模板不仅可以在 list 视图里使用，也可以在 edit 视图里使用。不同之处是在 edit 视图里你需要亲自处理局部模板，自己写动作(action)里的根据局部模板传来的数据更新字段的部分代码。symfony 知道如何处理普通的字段（实际的字段），它也会自己猜测的出如何处理你的局部模板里传来的数据。

例如，假设一个 User 类的管理界面模块有 id, nickname 和 password 三个字段。网站管理员需要能根据请求修改用户的密码，但是处于安全考虑 edit 视图不能显示密码的值。表单里应该显示一个空的密码输入框。例 14-28 里的配置可以实现上面的需求。

例 14-28 - 在 edit 视图里包含一个局部模板字段

```
edit:  
  display:      [id, nickname, _newpassword]  
  fields:  
    newpassword: { name: Password, help: Enter a password to  
change it, leave the field blank to keep the current one }
```

templates/_newpassword.php 局部模板文件的内容如下：

```
<?php echo input_password_tag('newpassword', '') ?>
```

注意这个局部模板使用的是简单的表单辅助方法，而不是对象表单辅助方法，因为这里不需要从当前 User 独享取得密码的值来生成这个表单控件——这可能泄漏用户的密码。

现在，要在动作(action)里使用表单的值来更新对象，你还需要扩展动作里的 updateUserFromRequest() 方法。这需要在动作文件里增加一个包含处理这个模板字段的自定义行为的同名方法，如例 14-29。

例 14-29 - 在动作里处理局部模板， 位于
modules/user/actions/actions.class.php

```
class userActions extends sfActions  
{  
  protected function updateUserFromRequest()  
  {
```

```

// 处理局部模板字段的输入
$password = $this->getRequestParameter('newpassword');

if ($password)
{
    $this->user->setPassword($password);
}

// 让 symfony 处理其他的字段

parent::updateUserFromRequest();

}

}

```

NOTE 在实际的应用中，`user/edit` 视图通常应该包含两个密码字段，第二个用来跟第一个作比对从而避免输入错误。你可以在第 10 章找到，这可以通过 `validator` 来实现。自动生成的管理界面模块与普通模块一样也可以从这种机制中获益。

处理外键

如果你的 `schema` 里定义了表间关系，那么自动生成的管理界面模块会利用定义的表间关系并能自动处理它们，这极大的简化了关系管理。

一对多关系

管理界面生成器可以很好的处理 1-n(一对多)的表关系。在图 14-1 里，表 `blog_comment` 与 `blog_article` 表通过 `article_id` 字段相关联。如果你用管理界面生成器初始化 `Comment` 类的模块，`comment/edit` 动作会自动用下拉列表来显示 `blog_article` 表里的记录作为 `article_id` 字段的表单控件（请看图 14-9）。

另外，如果给 `Article` 对象定义一个 `__toString()` 方法，下拉列表的选项文字会从原来的主键变成这个方法的返回值。

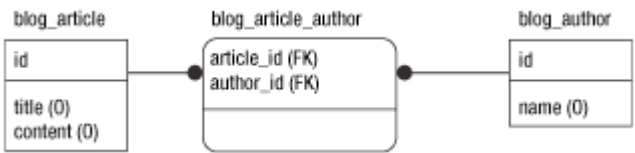
如果你要在 `article` 模块里显示一篇文章的留言列表（n-1 关系），你需要用一个局部模板字段来对这个模块进行定制。

多对多关系

`symfony` 也可以处理多对多关系，但是在 `schema` 里没有办法定义它们，需要在 `generator.yml` 里面通过一些额外的参数来实现。

多对多关系的实现需要一个中间表。例如，如果 `blog_article` 与 `blog_author` 之间存在 `n-n` 关系（一篇文章可以有多个作者，很明显，一个作者可以写多篇文章），你的数据库肯定会有一个 `blog_article_author` 或者类似的表，如图 14-20。

图 14-20 - 使用 “`through_class` 中间类” 实现多对多关系



这个模型有一个叫 `ArticleAuthor` 的类，管理界面生成器只需要它就可以了，不过你还要把字段的 `through_class` 参数设置成这个类。

例如，在基于 `Article` 类生成的模块里，可以通过例 14-30 里的 `generator.yml` 文件新增一个代表 `Author` 类的 `n-n` 关联的字段。

例 14-30 - 通过 `through_class` 参数处理多对多关系

```
edit:
  fields:
    article_author: { type: admin_double_list, params:
through_class=ArticleAuthor }
```

这个字段会处理存在的对象之间的关联，所以下拉列表就显得不够了。需要给它设置一个特殊的表单控件。`symfony` 提供了三个表单控件来处理多对多关系（如图 14-21）：

- `admin_double_list` 是两个展开的选择列表组成的选择控件，它还包含在第一个列表（可选元素）和第二个列表（选中元素）切换元素的按钮。
- `admin_select_list` 是一个可以用来选择多个元素的展开选择控件。
- `admin_check_list` 是由一组复选框组成的。

图 14-21 - 可用于多对多关系的控件



增加交互

管理界面模块允许用户进行常见的 CRUD 操作，不过你也可以增加你自己的交互或者对某个视图的交互做限制。例如，例 14-31 里的交互定义包括了所有 `article` 模块的默认 CRUD 动作。

例 14-31 - 给每个视图定义交互，位于 `backend/modules/article/config/generator.yml`






```
list:
  title:          List of Articles
  object_actions:
    _edit:         ~
    _delete:       ~
  actions:
    _create:       ~

edit:
  title:          Body of article %%title%%
  actions:
    _list:         ~
    _save:         ~
    _save_and_add: ~
    _delete:       ~
```

在 `list` 视图里有两组动作设置：针对每个对象的动作和针对整个页面的动作。例 14-31 里定义的交互结果如图 14-22 所示。每行都有一个编辑记录的按钮还有一个删除按钮。在列表的底部，有一个增加记录的按钮。

图 14-22 - `list` 视图的交互

List of Articles

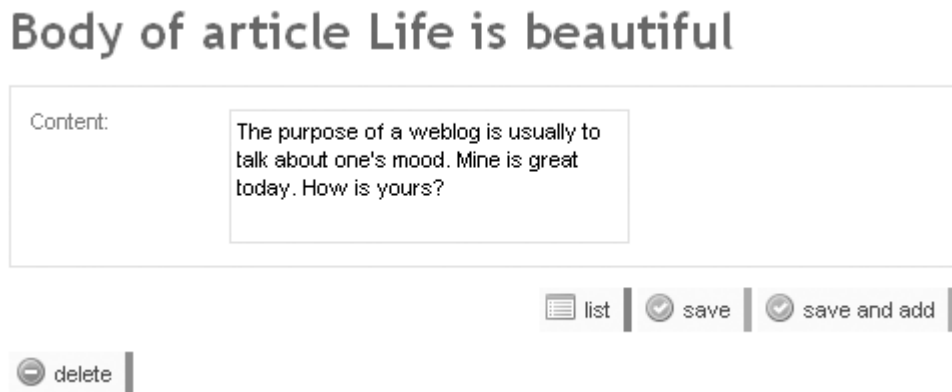
Id	Title	lib comments	Created at	Actions
1	Welcome to the symfony weblog!	3	December 1, 2006 1:17 PM	 
2	Life is beautiful	3	December 1, 2006 1:17 PM	 
2 results				
				 create

在 `edit` 视图里，由于一次只修改一条记录，所以只要定义一组动作。例 14-31 里定义的 `edit` 视图的交互结果如图 14-23 所示。`save` 还有 `save_and_add` 动作都会把当前对象写回数据库，不同的是 `save` 动作保存了记录之后会回到当前记录的 `edit` 视图，而 `save_and_add` 动作会显示一个空的 `edit` 视图来新增一条记

录。save_and_add 在一次增加多条记录的时候很有用。delete 按钮的位置与其他的按钮分开，这样用户不容易点错。

交互的名字以下划线(_)开头告诉 symfony 使用这个动作默认的图标与动作。管理界面生成器可以理解的交互有_edit、_delete、_create、_list、_save、_save_and_add 和_create。

图 14-23 - edit 视图的交互



不过你也可以指定一个自定义的交互，这样需要指定一个不以下划线开头的名字，如例 14-32。

例 14-32 - 定义一个自定义交互

```
list:
  title:          List of Articles
  object_actions:
    _edit:        -
    _delete:      -
    addcomment:   { name: Add a comment, action: addComment, icon:
backend/addcomment.png }
```

如图 14-24，列表里面的每篇文章现在有一个显示 addcomment.png 的按钮。点击这个按钮触发当前模块的 addComment 动作。当前对象的主键会自动附加到请求参数里。

图 14-24 - list 视图的自定义交互

List of Articles

Id	Title	lib comments	Created at	Actions
1	Welcome to the symfony weblog!	3	December 1, 2006 1:17 PM	  
2	Life is beautiful	3	December 1, 2006 1:17 PM	  
2 results				

 create

Add a comment

addComment 的代码如例 14-33 所示。

例 14-33 - 自定义交互的动作代码，actions/actions.class.php

```
public function executeAddComment()
{
    $comment = new Comment();
    $comment->setArticleId($this->getRequestParameter('id'));
    $comment->save();

    $this->redirect('comment/edit?id=' . $comment->getId());
}
```

关于交互最后说一下：如果想要完全抑制某种交互，请像例 14-34 里那样使用空列表。

例 14-34 - 去掉 list 视图里的所有动作

```
list:
  title:      List of Articles
  actions:    {}
```

表单验证

如果你看一下项目 cache/目录里生成的 edit_form.php 模板，你会发现表单的字段使用了一个特殊的命名规则。在生成的 edit 视图里，控件的名字由模块名与方括号括起来的字段名组成。

例如，article 模块的 edit 视图有一个 title 字段，对应的模板可能会像例 14-35 中的那样，模板里的控件名将会是 article[title]。

例 14-35 - 生成的控件名格式

```
// generator.yml
generator:
```

```

class:                sfPropelAdminGenerator
param:
  model_class:        Article
  theme:              default
  edit:
    display: [title]

// 生成的_edit_form.php 模板
<?php echo object_input_tag($article, 'getTitle',
array('control_name' => 'article[title]')) ?>

// 返回的 HTML 代码
<input type="text" name="article[title]" id="article[title]"
value="My Title" />

```

这对内部表单处理过程有很多好处。不过，根据第 10 章里介绍的内容，这会使表单验证有点麻烦，你要把 `fields` 定义里的方括号 `[]` 改成大括号 `{}`。并且，如果要在 `validator` 的参数里使用字段名，需要用它在 HTML 里面的代码（也就是方括号，不过要用引号括起来）。生成表单的特殊的 `validator` 语法的详情请参考例 14-36。

例 14-36 - 管理界面生成表单的 `validator` 文件语法

```

## 用大括号替换方括号
fields:
  article{title}:
    required:
      msg: You must provide a title
    ## validator 的参数里，使用原始的字段名并用引号引用
    sfCompareValidator:
      check: "user[newpassword]"
      compare_error: The password confirmation does not match the
password.

```

使用证书限制用户动作

对于某个管理界面模块，可用字段和交互可以根据当前登录用户的证书而变化（详情见第 6 章 `symfony` 的安全功能部分）。

生成器里的字段可以使用 `credentials` 来限定只有拥有特定证书的用户使用。这在 `list` 视图和 `edit` 视图里都会起作用。另外，生成器可以根据证书隐藏交互。例 14-37 演示了这些功能。

例 14-37 - 在 `generator.yml` 里使用证书

```
## 只有拥有 admin 证书的用户可以看到 id 字段
list:
  title:          List of Articles
  layout:         tabular
  display:        [id, =title, content, nb_comments]
  fields:
    id:           { credentials: [admin] }
```

```
## 只有拥有 admin 证书的用户可以使用 addcomment 交互
list:
  title:          List of Articles
  object_actions:
    _edit:        -
    _delete:      -
    addcomment:   { credentials: [admin], name: Add a comment,
action: addComment, icon: backend/addcomment.png }
```

修改生成模块的外观

你可以修改生成模块的外观来配合已经存在的设计，不仅可以使用你自己的样式表，还可以重写默认的模板。

使用自定义样式表

由于生成的 HTML 内容是结构化的，所以你可以尽情的修改它的外观。

你可以为管理模块定义自己的 CSS 取代默认的 CSS，只要在 `generator.yml` 里加一个 `css` 参数就可以了，如例 14-38。

例 14-38 - 使用自定义样式表取代默认样式表

```
generator:
  class:          sfPropelAdminGenerator
  param:
    model_class:  Comment
    theme:        default
    css:          mystylesheet
```

另外，也可以使用 `view.yml` 为每个视图重写样式设置。

增加自定义头部与尾部

`list` 视图和 `edit` 视图会自动载入一个头部和尾部局部模板。默认情况管理界面模块的 `templates/` 目录里没有这个局部模板，不过只要按照下面的名字自己建立这些模板，它们就会被自动载入：

```
_list_header.php
_list_footer.php
_edit_header.php
_edit_footer.php
```

例如，如果你想在 `article/edit` 视图增加一个自定义头部，按照例 14-39 的内容建立一个名叫 `_edit_header.php` 的文件。不需要其他配置它就会起作用。

例 14-39 - `edit` 头部局部模板，位于
`modules/articles/template/_edit_header.php`

```
<?php if ($article->getNbComments() > 0): ?>
    <h2>This article has <?php echo $article->getNbComments() ?>
comments.</h2>
<?php endif; ?>
```

注意 `edit` 视图里的局部模板总可以通过与模块名相同的变量访问到当前对象，另外 `list` 视图里的局部模板总可以通过 `$pager` 变量访问到当前页的信息。

SIDEBAR 使用自定参数调用管理界面的动作

管理界面模块可以通过 `link_to()` 辅助函数里的 `query_string` 参数来接受自定参数。例如，给前面的 `_edit_header` 局部模板增加一个给文章留言的链接，要这样写：

```
getNbComments() > 0): ?>

This article has getNbComments(). ' comments', 'comment/list',
array('query_string' =>
'filter=filter&filters%5Barticle_id%5D='.$article->getId())) ?>
```

这个 `query_string` 是编码过的版本，这样更可靠

```
'filter=filter&filters[article_id]='.$article->getId()
```

它只过滤显示跟 `$article` 有关的留言。使用 `query_string` 参数，你可以指定排列顺序和过滤器来显示一个特殊的列表视图。这对自定义交互也很有用。

自定义主题

还有一些继承自框架的局部模板可以在 `templates/` 目录里被重写来满足你的特殊需求。

生成的模板被分成小块，它们可以单独重写，动作也可以单独修改。

不过，如果你想用同样的方式在多个模块里多次重写，你应该作一个可重用的主题。主题是一系列用于管理界面生成器的完整的模板与动作，要使用一个主题需要在 `generator.yml` 文件的开头指定这个主题。默认的主题的文件放在 `$sf_symfony_data_dir/generator/sfPropelAdmin/default/template/` 目录。

这些主题文件需要放在项目目录的

`data/generator/sfPropelAdmin/[theme_name]/template/` 目录下，可以通过复制默认主题的文件(在

`$sf_symfony_data_dir/generator/sfPropelAdmin/default/template/` 目录)快速地建立一个新主题。用这种方法，你要确定你的自定义主题里面包含下面这些必须的文件：

```
// 局部模板，[theme_name]/template/templates/
_edit_actions.php
_edit_footer.php
_edit_form.php
_edit_header.php
_edit_messages.php
_filters.php
_list.php
_list_actions.php
_list_footer.php
_list_header.php
_list_messages.php
_list_td_actions.php
_list_td_stacked.php
_list_td_tabular.php
_list_th_stacked.php
_list_th_tabular.php

// 动作，[theme_name]/template/actions/actions.class.php
processFilters() // 处理请求的过滤器
addFiltersCriteria() // 把过滤器增加到条件对象
processSort()
addSortCriteria()
```

注意这些模板文件实际上是模板的模板，也就是说，这个 PHP 文件会被用一个特殊的工具解析并且根据生成器设置生成模板（这被称作编译阶段）。生成的模板也许要包含实际浏览的时候执行的 PHP 代码，所以模板的模板使用了特殊的语法来使 PHP 代码在第一阶段不被执行。例 14-40 是一个默认模板的模板的实际内容。

例 14-40 - 模板的模板的语法

```
<?php foreach ($this->getPrimaryKey() as $pk): ?>
[?php echo object_input_hidden_tag($<?php echo $this->
>getSingularName() ?>, 'get<?php echo $pk->getPhpName() ?>') ?]
<?php endforeach; ?>
```

在这个例子里，<?里的 PHP 代码会立即执行（编译时），[?里的只在执行时执行，但模板引擎最终会把[?标签转换成<?标签，所以最后的模板看上去是这样的：

```
<?php echo object_input_hidden_tag($article, 'getId') ?>
```

处理模板的模板需要特别小心，所以要作一个新的主题最好还是从默认模板开始，一步一步修改他，多作测试。

TIP 你也可以把生成器的模板打包做成 plug-in，这会使它更容易在多个项目中重用。详情请参考第 17 章。

-

SIDEBAR 建立你自己的生成器

脚手架与管理界面生成器都使用了一系列 symfony 内部的自动地在缓存里生成代码的组件，还有主题和模板的模板。

这意味着 symfony 提供了所有的建立你自己的生成器所需要的工具，你自己的生成器可能会与现有的生成器类似或者完全不同。模块的生成是由 sfGeneratorManager 类的 generate() 方法来管理的。例如，要生成一个管理界面，symfony 内部会调用下面的代码：

```
$generator_manager = new sfGeneratorManager();
$data = $generator_manager->generate('sfPropelAdminGenerator',
$parameters);
```

如果你想作自己的生成器，你需要看 sfGeneratorManager 和 sfGenerator 的 API 文档，还要参考 sfAdminGenerator 和 sfCRUDGenerator 类。

总结

要快速建立模块或者自动生成后台，基础是定义良好的数据库设计（schema）和对对象模型。你可以修改脚手架的 PHP 代码，但是管理界面模块的大部分修改是通过配置文件进行的。

`generator.yml` 文件是后台编程的中心。它可以完全定制 `list` 视图和 `edit` 视图的内容、功能以及外观。字段的标签、提示、过滤器、排序还有页面记录数、控件类型、外键关系、自定义交互及其证书都可以通过直接修改 `YAML` 文件来实现，而不用写 `PHP` 代码。

如果管理界面生成器不支持你需要的功能，局部模板字段还有重写动作为你提供了完整的扩展性。另外你可以使用管理界面生成器的主题机制来重用你的修改。

第 15 章 单元测试和功能测试

自从有了面向对象的程序设计方法，自动化测试也迅速发展。特别是在开发 web 应用程序时，即使应用程序非常复杂，测试自动化也能保证程序的质量。symfony 提供了多种工具来实现测试自动化，本章将介绍这些方法。

测试自动化

任何有经验的 web 应用程序开发者都很在意要花多少时间来进行完备的测试。准备并运行测试案例然后分析测试结果是非常费时的工作。而且，web 应用的需求经常变动，因而一个应用会有一连串的版本，代码重构成为常事，错误因此不断地产生。

因而，虽然一个开发环境并不一定要有测试自动化，但是成功的开发环境还是应该包括测试自动化的。一组测试用例可以确保应用程序按照预计的去执行。尽管经常会重新设计程序内部结构，但是测试自动化可以防止重复犯错。而且，测试自动化会要求开发者用测试框架能理解的方式，去书写标准化了的、固定格式的测试用例。

因为测试自动化能清晰地描述一个应用程序能做什么，因而有时候它可以替代开发文档。一个好的测试包可以揭示出在一组测试输入下应该得到哪些输出结果，因而可以很好地解释一个方法的运行目标。

symfony 框架本身就采用了测试自动化方法。框架内部由测试自动化进行验证，这些单元测试和功能测试并不随 symfony 一起发布，但你可以从 SVN 库导出或从网站 <http://www.symfony-project.com/trac/browser/trunk/test> 得到。

单元测试和功能测试

单元测试检测一个单元的代码，以确定对于一个给定的输入，它能否得到正确的输出结果。对每个具体的测试用例，它可以验证函数或方法是否能正确执行。一次单元测试处理一个用例，所以如果一个方法在不同的情况下运行，就需要进行多次单元测试。

功能测试不是简单地测试输入输出间的转化是否正确，而是针对一个完整的功能特性。例如，一个缓存系统只能用一个功能测试去验证，因为它包括多个运行步骤。第一次请求一个页面时，产生缓存数据，第二次再请求页面时，页面就直接从缓存中取出。所以，功能测试可以验证一个运行过程，同时也需要一个运行环境，你可以用 symfony 为你的每一个动作写出功能测试代码。

对于最复杂的交互，只有这两种测试是不够的。比如说 ajax 交互，就需要一个 web 浏览器来执行 javascript，所以要自动测试还需要一个特殊的第三方工具。更进一步的例子是视觉效果测试，它只能由人亲自去完成。

如果你有多种测试自动化的方法，你可能组合使用这些方法。有一个原则，你应该尽量让测试简单而且易于理解。

NOTE 测试自动化将测试结果和预期输出进行比较，也就是说，它会计算一个象 \$a==2 这样的断言 (assertion) 的值。断言的结果要么是真要么是假，对应着测试通过或失败。在测试自动化技术中，经常会用到断言这个词。

测试驱动的开发方法

在测试驱动的开发 (TDD) 方法中，编码之前就写好了测试用例。先写测试用例可以帮助你真正开发一个功能之前明确一个功能会完成什么任务。象极限编程 (XP) 之类的开发方法也建议这样做。而且一个不可否认的事实是：如果你不事先写好单元测试用例，你将再也写不出来。

例如，你如果想写一个文本删节的功能，目的是将字符串开头和末尾的空格清除，用下划线替代非字母字符，并将所有大写字母转换成小写字母。在测试驱动的开发方法中，你要考虑到所有可能出现的情况，并对每种情况提供一个输入和预期输出的实例，如表 15-1 所示。

表 15-1 一个文本删节功能的测试用例表

输入	预期输出
" foo "	"foo"
"foo bar"	"foo_bar"
"-)foo:..=bar?"	"__foo____bar_"
"FooBar"	"foobar"
"Don't foo-bar me!"	"don_t_foo_bar_me_"

你要写出单元测试用例，运行这些测试用例，然后看结果是否正确。你要在代码中添加一些内容来处理第一个测试用例，运行看是否通过，再处理第二个测试用例，一直到所有测试用例都通过了，表明功能正确了。

采用驱动测试的开发方法设计的应用程序，测试代码几乎和实际代码一样多。如果你不想在调试测试用例方面花费太多时间，你应该尽力保持测试用例简单。

NOTE 重构一个方法会产生之前没有的错误。所以在将一个新版应用程序部署到生产环境去之前，应该运行所有的自动化测试代码——这个过程叫做回归测试。

Lime 测试框架

PHP 开发领域有许多单元测试框架，最著名的应该是 PHPUnit 和 SimpleTest。symfony 采用自己的测试框架，叫做 lime，它基于 Test::More Perl 库，并且遵守 TAP 规则，也就是说，测试结果按照 Test Anything Protocol（译者注：一种测试输出的格式标准，见本章的示例）规定的格式显示，这样有助于理解测试的输出结果。

Lime 用于单元测试，它是比较轻量级的 php 测试框架，并有以下一些特点：

- 它在一个受限的环境中运行测试文件，这样可以避免每次测试之间产生奇怪的副作用。不是所有的测试框架都能保证为每次测试提供一个干净的测试环境的。
- lime 测试用例和测试结果都非常容易理解，在某些系统中，lime 还可以用彩色输出这种直观的方法来区分重要信息。
- symfony 就是利用 lime 进行回归测试的，所以在 symfony 源代码中可以看到许多单元测试和功能测试的例子。
- lime 核心就被单元测试验证过。
- lime 是用 PHP 写的，速度快且经过良好地编码。它只需一个 lime.php 文件，而不依赖于其它任何文件。

以下将使用 lime 语法描述各种不同的测试，在 symfony 安装版本中不包括这些功能。

NOTE 在生产环境中不应该运行单元测试和功能测试。这些是开发工具，应该运行在用于开发的机器上，而不应该运行在实际的服务器上。

单元测试

symfony 单元测试是以 Test.php 结尾的一些 PHP 文件，存放在你的应用程序的 test/unit/目录下。它们采用一种简单易读的语法。

单元测试概述

例 15-1 给出了一组典型的对 strtolower()函数进行单元测试的代码。从实例化 lime_test 对象开始（现在你无需考虑如何设置参数），接下来，每条单元测试就是对 lime_test 实例的方法的一次调用。这些方法的最后一个参数都是可选的用作输出的字符串。

例 15-1 单元测试文件示例，位于 test/unit/strtolowerTest.php

```
<?php
```

```
include(dirname(__FILE__).'../bootstrap/unit.php');
```

```

require_once(dirname(__FILE__).'../../lib/strtolower.php');

$t = new lime_test(7, new lime_output_color());

// strtolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower(' Foo'), 'string',
    'strtolower() returns a string');
$t->is(strtolower(' F00'), 'foo',
    'strtolower() transforms the input to lowercase');
$t->is(strtolower(' foo'), 'foo',
    'strtolower() leaves lowercase characters unchanged');
$t->is(strtolower(' 12#?@~'), '12#?@~',
    'strtolower() leaves non alphabetical characters unchanged');
$t->is(strtolower(' F00 BAR'), 'foo bar',
    'strtolower() leaves blanks alone');
$t->is(strtolower(' Fo0 bAr'), 'foo bar',
    'strtolower() deals with mixed case input');
$t->is(strtolower(''), 'foo',
    'strtolower() transforms empty strings into foo');

```

在命令行执行 `test-unit` 即可启动测试。命令行输出非常简明，可以帮助你确定哪些测试失败，哪些通过。例 15-2 列出了测试的结果。

例 15-2 从命令行启动单个单元测试

```
> symfony test-unit strtolower
```

```

1..7
# strtolower()
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase
ok 3 - strtolower() leaves lowercase characters unchanged
ok 4 - strtolower() leaves non alphabetical characters unchanged
ok 5 - strtolower() leaves blanks alone
ok 6 - strtolower() deals with mixed case input
not ok 7 - strtolower() transforms empty strings into foo
#     Failed test (.batch\test.php at line 21)
#         got: ''
#         expected: 'foo'
# Looks like you failed 1 tests of 7.

```

TIP 例 15-1 中开头的 `include` 命令是可选的，但是有了这条 `include` 命令，这个测试文件就变成了一个独立的 PHP 脚本，无需 `symfony` 命令行就可以执行。

单元测试方法

Lime_test 对象包含了大量测试方法，如表 15-2 所示。

表 15-2 Lime_test 对象中用于单元测试的方法

方法	描述
diag(\$msg)	仅输出备注信息而不进行测试
ok(\$test, \$msg)	测试一个条件，如果条件为真，则通过
is(\$value1, \$value2, \$msg)	比较两个数值，如果全等(==)则通过
isnt(\$value1, \$value2, \$msg)	比较两个数值，如果不等则通过
like(\$string, \$regexp, \$msg)	测试字符串是否匹配正则表达式
unlike(\$string, \$regexp, \$msg)	测试字符串是否不匹配正则表达式
cmp_ok(\$value1, \$operator, \$value2, \$msg)	比较两个参数的值是否与某个运算符匹配
isa_ok(\$variable, \$type, \$msg)	测试变量的类型
isa_ok(\$object, \$class, \$msg)	测试对象所属的类
can_ok(\$object, \$method, \$msg)	测试一个方法是否适用于某个对象或某个类
is_deeply(\$array1, \$array2, \$msg)	测试两个数组是否有相同的值
include_ok(\$file, \$msg)	验证某个文件是否存在并且已经被包含
fail()	永远失败——用于测试异常
pass()	永远通过——用于测试异常
skip(\$msg, \$nb_tests)	跳过\$nb_tests 条后续的测试——用于条件测试
todo()	作为一条测试参加测试计数——为将要写但还未写的测试预留位置

语法很直观，并且你会看出多数方法用一个 message(信息)作为最后一个参数，当测试通过时，就会在屏幕上输出这个参数值。实际上，掌握这些方法的最好办法就是去测试它们，例 15-3 列出了调用这些方法的代码。

例 15-3 Lime_test 对象的测试方法，位于 test/unit/exampleTest.php

```
<?php

include(dirname(__FILE__).'../bootstrap/unit.php');

// 用于测试的桩对象和函数
```

```

class myObject
{
    public function myMethod()
    {
    }
}

function throw_an_exception()
{
    throw new Exception('exception thrown');
}

// 初始化测试对象
$t = new lime_test(16, new lime_output_color());

$t->diag('hello world');
$t->ok(1 == '1', 'the equal operator ignores type');
$t->is(1, '1', 'a string is converted to a number for comparison');
$t->isnt(0, 1, 'zero and one are not equal');
$t->like('test01', '/test\d+/', 'test01 follows the test numbering
pattern');
$t->unlike('tests01', '/test\d+/', 'tests01 does not follow the
pattern');
$t->cmp_ok(1, '<', 2, 'one is inferior to two');
$t->cmp_ok(1, '!=', true, 'one and true are not identical');
$t->isa_ok('foobar', 'string', '\foobar\' is a string');
$t->isa_ok(new myObject(), 'myObject', 'new creates object of the
right class');
$t->can_ok(new myObject(), 'myMethod', 'objects of class myObject do
have amyMethod method');
$array1 = array(1, 2, array(1 => 'foo', 'a' => '4'));
$t->is_deeply($array1, array(1, 2, array(1 => 'foo', 'a' => '4')),
    'the first and the second array are the same');
$t->include_ok('./fooBar.php', 'the fooBar.php file was properly
included');

try
{
    throw_an_exception();
    $t->fail('no code should be executed after throwing an exception');
}
catch (Exception $e)
{
    $t->pass('exception catched successfully');
}

```

```

}

if (!isset($foobar))
{
    $t->skip('skipping one test to keep the test count exact in the
condition', 1);
}
else
{
    $t->ok($foobar, 'foobar');
}

$t->todo('one test left to do');

```

在 symfony 单元测试中你会看到更多使用这些方法的例子。

TIP 你会奇怪这里为什么用 `is()` 而不是 `ok()`。原因是 `is()` 的输出信息比 `ok()` 更精确，`ok()` 仅指出条件失败，而 `is()` 还同时输出测试的成员。

测试参数

初始化 `lime_test` 对象时，它的第一个参数代表将要执行测试的项数。如果最终执行测试的项数与该参数值不同，`lime` 会输出相关的警告信息。比如，例 15-3 的测试会输出例 15-4 的内容。对象初始化时要求进行 16 项测试，而最终只测试了 15 项，输出结果中给出了相关信息。

例 15-4 测试计数有助于进行测试规划

```
> symfony test-unit example
```

```

1..16
# hello world
ok 1 - the equal operator ignores type
ok 2 - a string is converted to a number for comparison
ok 3 - zero and one are not equal
ok 4 - test01 follows the test numbering pattern
ok 5 - tests01 does not follow the pattern
ok 6 - one is inferior to two
ok 7 - one and true are not identical
ok 8 - 'foobar' is a string
ok 9 - new creates object of the right class
ok 10 - objects of class myObject do have a myMethod method
ok 11 - the first and the second array are the same
not ok 12 - the fooBar.php file was properly included
#      Failed test (.\\test\\unit\\testTest.php at line 27)

```



```
#      Tried to include './fooBar.php'
ok 13 - exception caught successfully
ok 14 # SKIP skipping one test to keep the test count exact in the
condition
ok 15 # TODO one test left to do
# Looks like you planned 16 tests but only ran 15.
# Looks like you failed 1 tests of 16.
```

`diag()`方法只用于显示注释信息，而不作为一条测试被计数，利用它，你的测试输出可以保持条理分明。另一方面，`todo()`和`skip()`方法则按照正常的测试被计数。在`try/catch`块中的`pass()/fail()`组合会被当作一项测试被计数。

一个经过仔细规划的测试策略应该包括这个预计的测试条数。你会发现用它来验证你自己的测试文件非常有用，有些测试运行在判断条件或处理异常的复杂情况下，这时候测试条数就更为有用。

构造函数的第二个参数是一个继承了`lime_output`类的输出对象。因为大多数时候，测试是指在命令行方式下进行，输出一个`lime_output_color`对象，就可以利用`bash`的彩色去显示。

测试单元任务

从命令行启动的单元测试称为测试单元任务，它以一组测试名或一个包含通配符的文件模式作为参数。如例 15-5 所示。

例 15-5 启动单元测试

```
// 测试目录结构
test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
  foo/
    barTest.php

> symfony test-unit myFunction          ## 运行
myFunctionTest.php
> symfony test-unit myFunction mySecondFunction ##运行两种测试
> symfony test-unit 'foo/*'             ##运行 barTest.php
> symfony test-unit '*'                  ##递归运行所有的测试
```

测试桩 (Stubs)，测试资源和自动加载

默认情况下，单元测试不支持自动加载特性。所以，测试中的每个类要么在一个测试文件中定义，要么提供外部支持文件。所以许多测试文件都象例 15—6 所示的那样以一组 `include` 行开头。

例 15-6 在单元测试中包含类

```
<?php

include(dirname(__FILE__).'../bootstrap/unit.php');
include(dirname(__FILE__).'../../config/config.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');

$t = new Lime_test(7, new Lime_output_color());

// isPathAbsolute()
$t->diag('isPathAbsolute()');
$t->is(sfToolkit::isPathAbsolute('/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('C:\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('d:/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('../test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('../\\test'), false,
    'isPathAbsolute() returns false if path is relative');
```

在单元测试中，你不仅要实例化要测试的对象，而且要实例化它们所依赖的对象。因为单元测试必须保持单一性，所以依赖于其它类，将可能导致因为一个类错误而引起多个测试失败。另外，从代码行数和执行时间的角度来看，设置真正的对象也是非常费时的。因为开发人员很快就不能忍受缓慢的测试，所以要牢记单元测试的速度非常重要。

无论何时你开始进行一个包含多个脚本的单元测试，你都需要一个简单的自动加载系统。为此，`sfCore` 类(需要手工包含进去)提供了一个 `initSimpleAutoload()` 方法，它用一条绝对路径作为参数，在这条路径下的所有类都会被自动加载。例如，如果想自动加载路径 `$sf_symfony_lib_dir/util/` 下的所有类，就在你的测试脚本中以下列代码开始：

```
require_once($sf_symfony_lib_dir.'/util/sfCore.class.php');
sfCore::initSimpleAutoload($sf_symfony_lib_dir.'/util');
```

TIP 因为生成的 Propel 对象依赖于一连串类，所以只要你测试 Propel 对象，就应该自动加载。注意，如果你要让 Propel 工作，你还需要包含 vendor/propel 目录下的文件，这样，调用 sfCore 就变成了

```
sfCore::initSimpleAutoload(array (SF_ROOT_DIR.'/lib/model',
$sf_symfony_lib_dir.'/vendor/propel')); 同时还要用 set_include_path($
sf_symfony_lib_dir.'/vendor'.PATH_SEPARATOR.SF_ROOT_DIR.PATH_SEPARATOR.
R.get_include_path ())将 Propel 核心加入到包含路径中去。
```

替代自动加载的另一种方法是使用测试桩。测试桩用另一种方法实现一个类，也就是用一些简单的封闭好的数据替代真实的类方法，以模拟真实类的行为，同时又不需要象真实的类那样复杂。测试桩的一个典型的例子就是一个数据库连接或一个 web service 接口。在例 15-7 中，要对一个映射 API 进行单元测试需要 WebService 类，但在测试时却不调用真正的 web service 类的 fetch() 方法，而是用一个测试桩去返回测试数据。

例 15-7 在单元测试中运用测试桩

```
require_once(dirname(__FILE__).'../../lib/WebService.class.php');
require_once(dirname(__FILE__).'../../lib/MapAPI.class.php');

class testWebService extends WebService
{
    public static function fetch()
    {
        return
file_get_contents(dirname(__FILE__).'../../fixtures/data/fake_web_service.
xml');
    }
}

$myMap = new MapAPI();

$t = new lime_test(1, new lime_output_color());

$t->is($myMap->getMapSize(testWebService::fetch(), 100));
```

测试数据有时比一个字符串或一个方法调用更复杂，复杂的测试数据常被称为测试资源。为了清晰地编写代码，最好是将测试资源放在独立的文件中，特别是这些测试数据会被多个单元测试文件用到的时候。另外，不要忘记 symfony

可以用 `sfYAML::load()` 方便地将 YAML 文件转换为数组。这样就可以将测试数据放在一个 YAML 文件中，而不用写很长的 PHP 数组，如例 15-8 所示。

例 15-8 在单元测试中运用测试资源文件

```
// 在 fixtures.yml :
-
  input:  '/test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  '\\test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  'C:\\test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  'd:/test'
  output: true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:  'test'
  output: false
  comment: isPathAbsolute() returns false if path is relative
-
  input:  '../test'
  output: false
  comment: isPathAbsolute() returns false if path is relative
-
  input:  '..\\test'
  output: false
  comment: isPathAbsolute() returns false if path is relative

// 在 testTest.php
<?php

include(dirname(__FILE__).'../bootstrap/unit.php');
include(dirname(__FILE__).'../..../config/config.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');
require_once($sf_symfony_lib_dir.'/util/sfYaml.class.php');

$testCases = sfYaml::load(dirname(__FILE__).'fixtures.yml');
```

```

$t = new lime_test(count($testCases), new lime_output_color());

// isPathAbsolute()
$t->diag('isPathAbsolute()');
foreach ($testCases as $case)
{
    $t->is(sfToolkit::isPathAbsolute($case['input']),
    $case['output'], $case['comment']);
}

```

功能测试

功能测试的目的是验证应用系统的各个部分。它们会模拟浏览器会话、发出请求、再检测响应中的元素值，就象你手工验证一个动作是否按预计的去执行一样。在功能测试中，你将运行一个与用例一致的场景。

功能测试概要

你可以用一个文本浏览器和一大堆正则表达式断言来进行功能测试，但是这样做是很费时间的。symfony 提供了一个称为 sfBrowser 的特别的对象，它连接到 symfony 应用程序，象一个浏览器一样工作，而不需要一个真实的服务器，同时不会降低 HTTP 传输速度。有了它，你可以存取每个请求的核心对象，包括 request、session、context 和 response 对象。symfony 还专门为功能测试提供了该类的一个扩展类，称为 sfTestBrowser，它提供了 sfBrowser 的所有功能并添加了一些灵活的断言方法。

一般来说，功能测试从初始化一个测试浏览器对象开始，这个对象发出一个动作请求然后检验响应中的元素值。

例如，每当你用 init-module 或 propel-init-crud 生成一个模块框架时，symfony 就为这个模块创建一个简单的功能测试。这个测试可以对默认的动作发出请求，然后检测响应的状态代码、路由得到的模块和动作、以及响应内容中的某个句子。例 15-9 是为 foobar 模块生成的 foobarActionsTest.php 文件。

例 15-9 一个新模块的默认功能测试文件，位于
tests/functional/frontend/foobarActionsTest.php

```

<?php

include(dirname(__FILE__).'../../bootstrap/functional.php');

// 创建一个新的测试浏览器

```

```

$browser = new sfTestBrowser();
$browser->initialize();

$browser->
    get('/foobar/index')->
        isStatusCode(200)->
            isRequestParameter('module', 'foobar')->
                isRequestParameter('action', 'index')->
                    checkResponseElement('body', '!/This is a temporary page/')
;

```

TIP 浏览器方法返回一个 `sfTestBrowser` 对象，为了让你的测试文件更具可读性，你可以将方法调用串接起来。因为这一串方法调用可以不停顿地执行，所以称之为对象的流动接口。

功能测试可以包含多个请求和更复杂的断言，在接下来的章节中你会看到所有的可能情况。

运用 `symfony` 的命令行语句 `test-functional`，可以执行一个功能测试，参见例 15-10。这个命令以一个应用程序名和一个测试名为参数，测试名省略了 `Test.php` 后缀。

例 15-10 从命令行启动一个功能测试

```
> symfony test-functional frontend foobarActions
```

```

# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
not ok 4 - response selector body does not match regex /This is a
temporary page/
# Looks like you failed 1 tests of 4.
1..4

```

默认情况下，为一个新模块而生成的功能测试是不会通过的。这是因为在一个新创建的模块里，`index` 动作会指向 `symfony` 的 `default` 模块的祝贺页面，其中包含 “This is a temporary page” 的句子。只要你没有修改 `index` 动作，测试该模块就总是失败，这样可以保证未完成的模块不能通过所有测试。

NOTE 功能测试启动了自动加载特性，所以你不需要用 `include` 包括所需文件。

用 `sfTestBrowser` 对象浏览

测试浏览器可以发出 GET 和 POST 请求，并且都是用一个真实的 URI 作为参数。
例 15-11 显示了如何调用 sfTestBrowser 对象的方法来模拟请求。

例 15-11 模拟 sfTestBrowser 对象的请求

```
include(dirname(__FILE__).'../../bootstrap/functional.php');

// 创建一个新的浏览器
$b = new sfTestBrowser();
$b->initialize();

$b->get('/foobar/show/id/1');           // GET 请求
$b->post('/foobar/show', array('id' => 1)); // POST 请求

// get() 和 post()方法是 call()方法的简写形式

$b->call('/foobar/show/id/1', 'get');
$b->call('/foobar/show', 'post', array('id' => 1));

// call() 可以模拟任何请求方式
$b->call('/foobar/show/id/1', 'head');
$b->call('/foobar/add/id/1', 'put');
$b->call('/foobar/delete/id/1', 'delete');
```

一个典型的浏览器会话不仅包括对具体动作的请求，还包括在超链接和浏览器按钮上点击。sfTestBrowser 对象也能模拟这些请求，如例 15-12 所示。

例 15-12 模拟 sfTestBrowser 对象的浏览

```
$b->get('/');           // 浏览主页
$b->get('/foobar/show/id/1');
$b->back();              // 倒退一页
$b->forward();           // 前进一页
$b->reload();            // 刷新当前页
$b->click('go');         // 找到名为`go`的链接或按钮并单击
```

测试浏览器能够处理调用栈，所以 back()和 forward()方法就象在一个真正的浏览器里一样工作。

TIP 测试浏览器有自己的管理 session(sfTestStorage)和 cookie 的机制。

在最需要测试的交互中，与表单有关的交互可能要排在首位。为了模拟表单的输入和提交，你有三种选择：一种是用你希望提交的参数发出一个 POST 请

求，第二种是调用 `click()` 并将表单参数作为一个数组来传递，最后一种是在每个表单字段中一个又一个地填入值并点击提交按钮。三种方法最后都得到同样的 POST 请求。例 15-13 是一个示例。

例 15-13 用 `sfTestBrowser` 对象模拟表单输入

```
// 示例模板，位于 modules/foobar/templates/editSuccess.php
<?php echo form_tag('foobar/update') ?>
    <?php echo input_hidden_tag('id', $sf_params->get('id')) ?>
    <?php echo input_tag('name', 'foo') ?>
    <?php echo submit_tag('go') ?>
    <?php echo textarea('text1', 'foo') ?>
    <?php echo textarea('text2', 'bar') ?>
</form>
```

```
// 为该表单设计的功能测试示例 $b = new sfTestBrowser(); $b-
->initialize(); $b->get('/foobar/edit/id/1');
```

```
// 方法 1: POST 请求
```

```
$b->post('/foobar/update', array('id' => 1, 'name' => 'dummy',
'commit' => 'go'));
```

```
// 方法 2: 用数组作参数并点击提交按钮
```

```
$b->click('go', array('name' => 'dummy'));
```

```
// 方法 3: 根据表单字段名输入值后点击提交按钮.
```

```
$b->setField('name', 'dummy')->click('go');
```

NOTE 注意。后两种方法在提交表单时，会自动包括默认的表单值，并且无需指明表单目标(处理表单的方法)。

当一个方法以 `redirect()` 结束时，测试浏览器并不自动重定向，你必须用 `followRedirect()` 手工重定向，参见例 15-14。

例 15-14 测试浏览器不能自动重定向

```
// 动作示例文件位于 modules/foobar/actions/actions.class.php
public function executeUpdate()
{
    ...
    $this->redirect('foobar/show?id=' . $this->getRequestParameter('id'));
}
```



```
// 该动作的功能测试示例
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    click('go', array('name' => 'dummy'))->
    isRedirected()->      // 检测请求是否被重定向
    followRedirect();     // 手工重定向
```

对浏览有用的最后一个方法是 `restart()` 方法，它可以重新初始化浏览器的 `history`、`session` 和 `cookie`——就象你重新启动了浏览器一样。

一旦有了第一个请求，`sfTestBrowser` 对象就可以存取 `request`、`context` 和 `response` 对象。也就是说，从文本内容到响应头以及请求参数和配置等，你都可以进行检测。

```
$request = $b->getRequest();
$context = $b->getContext();
$response = $b->getResponse();
```

SIDEBAR `sfBrowser` 对象

除了用于测试，在例 15-10 到例 15-13 中的所有浏览方法，还可以用于别的领域，只要通过 `sfBrowser` 对象来调用就可以，你可以用如下所示的方法去调用：

```
// 创建一个新的浏览器
$b = new sfBrowser();
$b->initialize();
$b->get('/foobar/show/id/1')->
    setField('name', 'dummy')->
    click('go');
$content = $b->getResponse()->getContent();
...
```

`sfBrowser` 对象对于批处理脚本是非常有用的，比如说，当你想浏览一组页面，以便为每个页面生成缓存版本的时候（请参考第 18 章的具体示例）。

运用断言

因为 `sfTestBrowser` 对象可以存取响应和请求的组成内容，所以你可以对这些组成内容进行测试。虽然为了测试你可以创建一个 `lime_test` 对象，但是 `sfTestBrowser` 对象的 `test()` 方法就能返回一个 `lime_test` 对象，有了这个对

象，你就可以调用前面介绍的单元断言方法。例 15-15 显示了如何通过 `sfTestBrowser` 来操作断言。

例 15-15 测试浏览器利用 `test()` 方法来提供测试手段

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$request = $b->getRequest();
$context = $b->getContext();
$response = $b->getResponse();

// 通过 test() 方调用 lime_test 的方法
$b->test()->is($request->getParameter('id'), 1);
$b->test()->is($response->getStatusCode(), 200);
$b->test()->is($response->getHttpHeader('content-type'),
' text/html; charset=utf-8');
$b->test()->like($response->getContent(), '/edit/');
```

NOTE `getResponse()`、`getContext()`、`getRequest()` 和 `test()` 等方法并不返回一个 `sfTestBrowser` 对象，所以你不能在这些方法之后串接其他 `sfTestBrowser` 的方法。

如例 15-16 所示，你可以通过请求和响应对象方便地检测输入输出 cookie 值。

例 15-16 用 `sfTestBrowser` 检测 cookie 值

```
$b->test()->is($request->getCookie('foo'), 'bar'); // 输入 cookie
$cookies = $response->getCookies();
$b->test()->is($cookies['foo'], 'foo=bar'); // 输出 cookie
```

利用 `test` 方法测试请求元素会导致代码行很长。为此，`sfTestBrowser` 提供了一组代理方法，以便让你的功能测试变得简明易读，并且返回一个 `sfTestBrowser` 对象。比如，在例 15-17 中，你可以用更快的方法写出与例 15-15 等价的代码。

例 15-17 直接用 `sfTestBrowser` 测试

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1')->
    isRequestParameter('id', 1)->
```

```
isStatutsCode()->
isResponseHeader('content-type', 'text/html; charset=utf-8')->
responseContains('edit');
```

isStatusCode()的状态参数默认值为 200，所以你可以用不加参数的调用来测试响应是否成功。

代理方法的另一个优点是，你不需要象调用 `lime_test` 方法那样去指明输出文本，代理方法会自动生成输出信息，所以测试结果简明易懂。

```
# get /foobar/edit/id/1
ok 1 - request parameter "id" is "1"
ok 2 - status code is "200"
ok 3 - response header "content-type" is "text/html"
ok 4 - response contains "edit"
1..4
```

例 15-17 的代理方法涵盖了大多数常规测试，所以你很少会再用 `sfTestBrowser` 对象的 `test()` 方法去测试。

例 15-14 中指出了 `sfTestBrowser` 不会自动重定向。这有一个好处，就是你可以测试一个重定向。例 15-18 显示了如何测试例 15-14 中的响应。

例 15-18 用 `sfTestBrowser` 测试重定向

```
$b = new sfTestBrowser();
$b->initialize();
$b->
    get('/foobar/edit/id/1')->
    click('go', array('name' => 'dummy'))->
    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'update')->

    isRedirected()->      // 检测响应是否是一个重定向
    followRedirect()->    //手工重定向。

    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'show');
```

运用 CSS 选择器

许多功能测试通过检测文本是否出现在页面中来验证页面是否正确。有了 `responseContains()` 方法中的正则表达式的帮助，你可以测试显示出来的文本、标记属性和值是否正确。但是，一旦你想测试深藏在响应的 DOM 中的内容的话，正则表达式就不太好用了。

为此，`SfTestBrowser` 对象提供了一个返回 `LibXML2DOM` 对象的 `getResponseDom()` 方法，编译和测试该 `LibXML2DOM` 对象比编译和测试一个普通文本要容易。例 15-19 给出一个使用这个方法例子。

例 15-19 用测试浏览器存取以 DOM 对象表示的响应内容

```
$b = new SfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$dom = $b->getResponseDom();
$b->test()->is($dom->getElementsByTagName('input')->item(1)-
>getAttribute('type'),'text');
```

但是，用 PHP DOM 方法去编译一个 HTML 文档仍旧不够快捷。如果你熟悉 CSS 选择器，你知道要从 HTML 文档中检索元素值，CSS 选择器才是更有效的工具。`symfony` 提供一个 `SfDomCssSelector` 工具类，它以 DOM 文档作为构造函数的参数。其中的 `get texts()` 方法根据一个 CSS 选择器返回一个字符串数组，而 `getElements()` 方法则返回一个 DOM 元素数组。例 15-20 给出一个示例。

例 15-20 测试浏览器存取以一个 `SfDomCssSelector` 对象表示的响应内容

```
$b = new SfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$c = new SfDomCssSelector($b->getResponseDom())
$b->test()->is($c->get texts('form input[type="hidden"][value="1"]'),
array(''));
$b->test()->is($c->get texts('form textarea[name="text1"]'),
array('foo'));
$b->test()->is($c->get texts('form input[type="submit"]'), array(''));
```

为了更加简明，`symfony` 提供了一种缩写方法，即代理方法 `checkResponseElement()`。例 15-20 利用这个方法可以得到和例 15-20 同样的结果。

例 15-21 测试浏览器通过 CSS 选择器存取响应元素

```

$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1')->
    checkResponseElement('form input[type="hidden"][value="1"]',
true)->
    checkResponseElement('form textarea[name="text1"]', 'foo')->
    checkResponseElement('form input[type="submit"]', 1);

```

如何执行 `checkResponseElement()` 方法，取决于它接收到的第二个参数的类型：

- 如果是逻辑型，则测试与 CSS 选择器匹配的元素是否存在。
- 如果是整型，则测试 CSS 选择器是否返回该整数个数的结果。
- 如果是正则表达式，则测试由 CSS 选择器发现的第一个元素是否与之匹配。
- 如果是由 `!` 开头的正则表达式，则测试第一个元素是否不匹配该正则表达式所表示的模式。
- 其他情形，则将由 CSS 选择器发现的第一个元素与第二个字符串参数进行比较。

该方法还接受以关联数组形式出现的第三个参数。它将测试某个指定位置的另一个元素，而不是测试由 CSS 选择器返回的第一个元素，参见例 15-22。

例 15-22 用位置选项匹配某个指定位置的元素

```

$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    checkResponseElement('form textarea', 'foo')->
    checkResponseElement('form textarea', 'bar', array('position' =>
1));

```

这个数组选项还可以用来同时进行两个测试。你可以测试是否有一个元素匹配 CSS 选择器以及有几个匹配元素，参见例 15-23。

例 15-23 运用 `count` 选项计算匹配的个数

```

$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    checkResponseElement('form input', true, array('count' => 3));

```

选择器工具功能很强，它可以接受 CSS2.1 的大多数选择器，你也可以用它测试类似例 15-24 中那样的复杂查询。

例 15-24 `checkResponseElement()` 接受的复杂 CSS 选择器

```
$b->checkResponseElement('ul#list li a[href]', 'click me');
$b->checkResponseElement('ul > li', 'click me');
$b->checkResponseElement('ul + li', 'click me');
$b->checkResponseElement('h1, h2', 'click me');
$b->checkResponseElement('a[class$="foo"][href*="bar.html"]', 'my link');
```

在测试环境中工作。

`sfTestBrowser` 对象可以使用一个特别的前端控制器来设置一个测试环境，。例 15-25 中是这种环境的默认配置。

例 15-25 `myapp/config/settings.php` 中的默认测试环境配置

```
test:
  . settings:
    # E_ALL | E_STRICT & ~E_NOTICE = 2047
    error_reporting:      2047
    cache:                off
    web_debug:            off
    no_script_name:       off
    etag:                 off
```

在这个环境中，`cache` 和 `web debug toolbar` 设为 `off`，但是，代码执行仍旧会在日志文件中留下执行痕迹，与 `dev` 和 `prod` 日志文件不同，`test` 日志存放在 `myproject/log/myapp_test.log` 中，这样，你可以分开查询。在这个环境里，异常不会阻止脚本的执行---所以即使某个测试出错，也可以完成整个测试。你还可以指明数据库连接设置，比如，可以使用含有测试数据的另一个数据库。

在运用 `sfTestBrowser` 对象之前，你要先对它进行初始化。如果有必要，你可以为一个应用程序指定一个主机名，为客户端指定一个 IP 地址---如果你的应用程序通过主机名和 IP 地址进行控制的话，例 15-26 是一个示例。

例 15-26 用主机名和 IP 设置测试浏览器

```
$b = new sfTestBrowser();
$b->initialize('myapp.example.com', '123.456.789.123');
```

功能测试任务

功能测试任务根据接收到的参数的个数可以执行一个或多个功能测试，其规则看上去与单元测试任务很接近，只是功能测试任务总是要以一个应用程序作为第一个参数。参见例 15-27。

例 15-27 功能测试任务语法

```
// 测试目录结构
test/
  functional /
    frontend/
      myModuleActionsTest.php
      myScenarioTest.php
    backend/
      myOtherScenarioTest.php

## 对一个应用程序的所有功能进行递归测试
> symfony test-functional frontend

## 运行一个给定的功能测试
> symfony test-functional frontend myScenario

## 按照一个模式运行多个测试
> symfony test-functional frontend my*
```

为测试命名

本节给出一些实用方法，以便于组织和维护你的测试。

对于文件结构，你应该用需要测试的类来命名单元测试文件，用需要测试的模块或场景来命名功能测试文件。例 15-28 是一个例子。你的 test 目录很快就会包含许多文件，如果你不遵守这些原则的话，在长期的运行中将很难找到你要的测试文件。

例 15-28 文件命名实例

```
test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
  foo/
    barTest.php
  functional /
    frontend/
```

```
myModuleActionsTest.php
myScenarioTest.php
backend/
myOtherScenarioTest.php
```

对单元测试来说，根据函数或方法分组，并且用一个 `diag()` 调用来启动每个测试组是一个比较好的方法。每个单元测试的信息应该包括要测试的功能或方法的名字，后接动作动词或属性名，这样测试的输出结果就象一个描述对象属性的句子。例 15-29 是个例子。

例 15-29 单元测试命名实例

```
// strtolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower(' Foo'), 'string', 'strtolower() returns a
string');
$t->is(strtolower(' F00'), 'foo', 'strtolower() transforms the input
to lowercase');
```

strtolower()

ok 1 - strtolower() returns a string

ok 2 - strtolower() transforms the input to lowercase

功能测试应该根据页面分组并且用一个请求来启动。例 15-30 是个例子。

例 15-30 功能测试命名实例

```
$browser->
  get('/foobar/index')->
  isStatusCode(200)->
  isRequestParameter('module', 'foobar')->
  isRequestParameter('action', 'index')->
  checkResponseElement('body', '/foobar/');
;
```

get /comment/index

ok 1 - status code is 200

ok 2 - request parameter module is foobar

ok 3 - request parameter action is index

ok 4 - response selector body matches regex /foobar/

如果你遵循这个习惯，你的测试输出结果将会非常清晰，它可以用作你的项目的开发文档，甚至有时候，这个文档可以完全取代实际的开发文档。

特别的测试需求

symfony 提供的单元和功能测试工具可以满足大多数需要。下面列出的一些技术用于解决测试自动化中会遇到的一些常见问题，包括：在孤立环境中启动测试，在测试中访问数据库，测试缓存及测试客户端的交互等。

在测试框架（Test Harness）中进行测试

单元测试任务和功能测试任务可以启动一个或一组测试。但是如果你不指明任何参数就调用这些任务，它们会启动 test 目录下的所有单元和功能测试。为了避免各个测试间互相影响，一种专门的机制用于将每个测试文件孤立在一个独立的环境中。进一步说，因为象保留单个测试的输出结果那样保留所有测试的输出结果（那种情况下，输出可能达到几千行）没有意义，因此测试结果将被压缩进一个综合的视图中。这就是为什么要用一个测试框架（也就是一个具备特殊能力的自动测试框架）来运行大量的测试文件的原因。测试框架来自于 lime 框架的 lime_harness 组件，它可以一个文件一个文件地显示测试状态，并且在测试完大量文件后给出一个统计结果。如例 15-31 所示。

例 15-31 在测试框架中启动所有的测试

```
> symfony test-unit t
```

```
unit/myFunctionTest.php.....ok
unit/mySecondFunctionTest.php.....ok
unit/foo/barTest.php.....not ok
```

Failed Test	Stat	Total	Fail	List of Failed
unit/foo/barTest.php	0	2	2	62 63

Failed 1/3 test scripts, 66.66% okay. 2/53 subtests failed, 96.22% okay.

测试的运行和一个一个分别运行一样，只是输出结果变得简短，特别是最后有关失败测试的数据可以帮助你找到原因，非常有用。

你也可以用一条 test-all 任务来启动所有的测试，该任务也使用测试框架，参见例 15-32。这是在每次传送到生产环境之前你必须要做的工作，目的是确保最后的版本不存在回归错误。

例 15-32 启动一个项目的所有测试

```
> symfony test-all
```

存取数据库

单元测试经常需要访问数据库。要初始化数据库连接，可以调用 Propel 类的 `getConnection()` 方法，如例 15-33 所示。

例 15-33 在测试中初始化数据库

```
$databaseManager = new sfDatabaseManager();  
  
$databaseManager->initialize();
```

// 可以取得当前数据库链接，这不是必须的

```
$con = Propel::getConnection();
```

在测试之前，你应该用 `sfPropelData` 对象将测试资源放入你的数据库中。该对象可以象 `propel-load-data` 任务一样从一个文件导入数据，或者从一个数组导入数据，参见例 15-34。

例 15-34 从一个测试文件导入数据库

```
$data = new sfPropelData();  
  
// 从文件导入数据  
$data->loadData(sfConfig::get('sf_data_dir').'/fixtures/test_data.yml');  
  
// 从数组导入数据  
$fixtures = array(  
    'Article' => array(  
        'article_1' => array(  
            'title'      => 'foo title',  
            'body'       => 'bar body',  
            'created_at' => time(),  
        ),  
        'article_2' => array(  
            'title'      => 'foo foo title',  
            'body'       => 'bar bar body',  
            'created_at' => time(),  
        ),  
    ),  
);  
$data->loadDataFromArray($fixtures);
```

然后你可以根据需要，象在通常的应用中那样使用 **Propel** 对象了。记住要在单元测试中包括它们的文件（在测试桩、测试资源和自动加载的 TIPS 中已经介绍过用 `sfCore::sfSimpleAutoloading()` 方法自动加载）。在功能测试中，**Propel** 对象是自动加载的。

测试缓存

当你在一个应用程序中使用缓存时，功能测试需要检测被缓存的动作是否按照期望的动作执行。

首先要做的是在测试环境中(`setting.yml`)允许缓存功能运行。然后，你就能用 **sfTestBrowser** 对象提供的 `isCached()` 测试方法来测试一个页面是生成的还是来自于缓存的。例 15-35 示例了这个方法的使用。

例 15-35 用 `isCached()` 方法测试缓存

```
<?php

include(dirname(__FILE__).'/../..../bootstrap/functional.php');

// 创建一个新的测试浏览器
$b = new sfTestBrowser();
$b->initialize();

$b->get('/mymodule');
$b->isCached(true);           // 测试响应是否来自缓存
$b->isCached(true, true);    // 测试被缓存的响应是否带有布局
$b->isCached(false);         // 测试响应是否不是来自缓存
```

NOTE 在功能测试开始时不需要清除缓存，启动脚本会替你执行。

测试客户端交互

前面介绍的技术的主要缺陷是它们不能模拟 **Javascript**。对于象 **Ajax** 交互那样的非常复杂的交互，你需要准确地复制由用户操作的鼠标点击和键盘输入并在客户端运行脚本。通常这些测试是手工完成的，但这样非常费时而且非常容易出错。

有一个完全用 **Javascript** 写成的测试框架 **Selenium** 可以作为一种解决方法加以考虑(<http://www.openga.org/selenium/>)，它可以利用当前的浏览器窗口，象用户操作的那样在一个页面上执行一组操作。与 **sfBrowser** 对象相比 **Selenium** 的好处是它能够在页面内执行 **Javascript**，因此你可以用它来测试 **Ajax** 交互。

Selenium 没有被自动封装在 symfony 中。如果要安装它，你可以在 web 目录下建一个新的 selenium 目录，在这个目录里解压 Selenium 文件 (<http://www.openga.org/selenium-core/download.action>)。这是因为 Selenium 依赖于 Javascript，而大多数浏览器在标准的安全设置中都禁止运行 Javascript，除非在和你的应用程序相同的主机和端口上可以运行。

CAUTION 不要将 selenium 目录传送到你的生产服务器中，因为任何能通过浏览器访问你的 web 文档根目录的人都可以操作它。

Selenium 测试用 HTML 编写，并存放在 web/selenium/tests 目录中。例 15-36 给出了一个功能测试，它导入主页，点击 click me 链接，然后在响应中能查到“Hello, World”。记住，为了能在测试环境中访问应用程序，你必须指定 myapp_test.php 前端控制器。

例 15-36 一个 Selenium 测试实例，位于 web/selenium/test/testIndex.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type">
  <title>Index tests</title>
</head>
<body>
<table cellspacing="0">
<tbody>
  <tr><td colspan="3">First step</td></tr>
  <tr><td>open</td>          <td>/myapp_test.php</td>
<td>&nbsp;</td></tr>
  <tr><td>clickAndWait</td>    <td>link=click me</td>
<td>&nbsp;</td></tr>
  <tr><td>assertTextPresent</td> <td>Hello, World!</td>
<td>&nbsp;</td></tr>
</tbody>
</table>
</body>
</html>
```

这个测试用例是用 HTML 写的，中间的一个表包含了三列，分别是命令、目标和值，不是所有命令都有对应的值。在本例中，要么让列空着，要么用 来让表更好看一些。要了解完整的 Selenium 命令，请参考 Selenium 网站。

在同一个目录中，你可以通过在 TestSuite.html 中的表格里插入一个新行，将这个测试添加到总的测试包中。例 15-37 是相关的示例。

例 15-37 在测试包中添加一个测试文件，位于
web/selenium/test/TestSuite.html

```
...  
<tr><td><a href='./testIndex.html'>My First Test</a></td></tr>  
...
```

要运行这个测试，只需在浏览器地址栏中输入：

<http://myapp.example.com/selenium/index.html>

选择 Main Test Suite，单击按钮运行所有的测试，观察你的浏览器是否模拟你想要执行的步骤。

NOTE 因为 Selenium 测试在真实的浏览器中运行，所以它可以测试浏览器的不一致性。你可以在某种浏览器中编制一个测试，然后在所有将支持你的应用程序的浏览器上检验。

由于 Selenium 测试是用 HTML 编写的，所以要编写一个 Selenium 测试是非常困难的。但是现在有了 Firefox 的 Selenium 扩展 ([<http://seleniumrecorder.mozdev.org/>] (<http://seleniumrecorder.mozdev.org/>))，要创建一个测试，只需在一个被记录的会话中执行一次测试即可。当你浏览记录的会话时，只需右击浏览器窗口并在弹出菜单中的 Append Selenium Command 项下选中适当的选项，就可以增加 assert-type 测试。

你可以将测试保存在一个 HTML 文件中，以便为你的应用程序建立一个测试包。Firefox 扩展还允许你运行你已经记录过的 Selenium 测试。

NOTE 在启动 Selenium 测试之前，不要忘了重新初始化测试数据。

总结

测试自动化包括单元测试和功能测试两部分：前者用于验证方法或函数，后者验证功能特性。symfony 的单元测试基于 lime 测试框架，并且为功能测试专门提供了 sfTestBrowser 类。两者都提供许多断言方法，从最基本的到最高级的，例如 CSS 选择器。使用 symfony 命令行启动测试，可以用 test-unit 和 test-functional 任务一个接一个地测试，也可以用 test-all 任务在一个测试框架中测试。当处理数据时，自动化测试要利用测试资源和测试桩模块，而这些在 symfony 单元测试中是很容易取得的。

如果，你确实已经为应用程序的大部分内容写了足够多的单元测试(也许你用的就是测试驱动的开发方法)，那么，当你重构内部结构或增加新的功能特性时，你就会感到更加放心，同时这也缩短了编写文档的时间。

第 16 章 应用程序管理工具

在开发与部署阶段，开发者需要稳定持续的调试信息来源来确定应用程序是否工作正常。这些信息通常是通过日志和调试工具来获取的。由于 `symfony` 这类框架的主要任务是加快应用程序开发，所以这些功能必须紧密的整合在框架里，以保证开发与日常操作的效率。

应用程序在生产服务器上的生命周期里，应用程序的管理者要做包括循环日志到升级在内的大量的重复任务。作为框架必须提供尽可能多的自动完成这些任务的工具。

本章详细介绍 `symfony` 应用程序如何完成这些任务。

日志

发现请求执行时出现问题的唯一手段是检查执行过程。在本章的学习里，你会了解到 `PHP` 与 `symfony` 都会提供大量的这类数据。

PHP 日志

`PHP` 的 `php.ini` 里有一个 `error_reporting` 参数，这个参数指定是否记录 `PHP` 事件。`symfony` 可以让你通过 `settings.yml` 文件在不同的应用程序里重写这个设置的值，如例 16-1 所示。

例 16-1 - 设置错误报告级别，`myapp/config/settings.yml`

```
prod:
  .settings:
    error_reporting: 257

dev:
  .settings:
    error_reporting: 4095
```

这些数字是错误级别的简写形式（详情请参考 `PHP` 文档的错误报告部分）。在这里，4095 是 `E_ALL|E_STRICT` 的简写，257 代表 `E_ERROR|E_USER_ERROR`（每个新环境的默认值）。

为了避免生产环境的速度问题，在生产环境里日志只记录严重 `PHP` 错误。不过，在开发环境里，所有类型的事件都会被记录，这样开发者能够有足够的信息来跟踪错误。

PHP 日志文件的位置取决与你的 `php.ini` 设置。如果你没修改过这个设置，PHP 很可能会使用 web 服务器的日志功能（例如 Apache 的错误日志）。在这种情况下，你会在 web 服务器的 `log` 目录里找到 PHP 日志。

symfony 日志

除了标准的 PHP 日志外，symfony 可以记录大量的自定义事件。symfony 的日志位于 `myproject/log/` 目录下。每个应用程序的每个环境都有一个对应的日志文件。例如，myapp 应用程序的开发环境的日志文件的名称是 `myapp_dev.log`，生产环境的是 `myapp_prod.log`。

如果你有一个运行中的 symfony 应用程序，请看一下它的日志文件，它的语法结构很简单。应用程序日志的每行记录一个事件。每行都包括事件发生的事件，事件的类型，正在处理的对象，还有一些额外的相关细节。例 16-2 是一个 symfony 日志文件的例子。

例 16-2 - symfony 日志文件例子， `log/myapp_dev.php`

```
Nov 15 16:30:25 symfony [info ] {sfAction} call "barActions->executemessages()"
Nov 15 16:30:25 symfony [debug] SELECT bd_message.ID,
bd_message.SENDER_ID, bd_...
Nov 15 16:30:25 symfony [info ] {sfCreole} executeQuery(): SELECT
bd_message.ID...
Nov 15 16:30:25 symfony [info ] {sfView} set slot "leftbar"
(bar/index)
Nov 15 16:30:25 symfony [info ] {sfView} set slot "messageblock"
(bar/mes...
Nov 15 16:30:25 symfony [info ] {sfView} execute view for template
"messa...
Nov 15 16:30:25 symfony [info ] {sfView} render
"/home/production/myproject/...
Nov 15 16:30:25 symfony [info ] {sfView} render to client
```

你可以在这些文件发现很多细节，包括数据库执行的 SQL 查询，调用的模板，对象间的方法调用等。

symfony 日志级别设置

symfony 日志消息共有八种级别：emerg, alert, crit, err, warning, notice, info, 还有 debug，与 PEAR::Log 包 (<http://pear.php.net/package/Log/>) 里的级别相同。可以在每个引用程序的 `logging.yml` 里定义各个环境要记录的事件的最大级别，如例 16-3 所示。

例 16-3 - 默认的日志配置，`myapp/config/logging.yml`

```

prod:
    enabled: off
    level:   err
    rotate:  on
    purge:   off

dev:

test:

#all:
#  enabled:  on
#  level:    debug
#  rotate:   off
#  period:   7
#  history:  10
#  purge:    on

```

默认情况下，在除了生产环境外的所有环境里，所有的消息都会被记录（设置值为最不重要的级别，debug 级）。在生产环境里，默认情况日志功能是关闭的；如果把 enabled 改成 on，只有最重要的消息（从 crit 到 emerg）会出现在日志里。

你可以在 logging.yml 文件里修改各个环境的日志级别来限制记录的消息类型。rotate, period, history 还有 purge 设置将在接下来的“清除与循环日志文件”这一节里介绍。

TIP 日志参数的值可以通过 sfConfig 对象和 sf_logging_前缀来访问。例如，要知道日志是否开启，可以调用 sfConfig::get('sf_logging_enabled')。

新增一条日志消息

你可以通过例 16-4 中的一种方法来手动给 symfony 日志文件里增加一条消息。

例 16-4 - 增加一条自定的消息

```

// 在动作里
$this->logMessage($message, $level);

// 在模板里
<?php use_helper('Debug') ?>
<?php log_message($message, $level) ?>

```

\$level 的取值可以与刚才的日志文件消息里的相同。

另外，直接使用 `sfLogger` 的方法可以在应用程序的任何地方往日志里增加一条消息，如例 16-5 所示。这个对象的方法名与日志的级别对应。

例 16-5 - 在任意的地方增加一个自定义日志消息

```
if (sfConfig::get('sf_logging_enabled'))
{
    sfContext::getInstance()->getLogger()->info($message);
}
```

SIDEBAR 自定义日志

symfony 的日志系统很简单，也很容易定制。你可以通过调用 `sfLogger::getInstance()->registerLogger()` 来指定你自己的日志对象。例如，如果你想使用 `PEAR::Log` 对象，只要在你的应用程序的 `config.php` 里增加下面的代码就可以了：

```
require_once('Log.php');
$log = Log::singleton('error_log', PEAR_LOG_TYPE_SYSTEM, 'symfony');
sfLogger::getInstance()->registerLogger($log);
```

如果你要注册自己写的日志类，唯一的要求是这个类必须定义一个 `log()` 方法。symfony 调用这个方法的时候使用两个参数：`$message`（需要记录的消息）还有 `$priority`（级别）。

清除与循环日志文件

不要忘记定期清除 `log/` 目录的内容，因为这些文件有一个大小每天增长好几 MB 的怪习惯，当然，这取决于你的流量。symfony 为此准备了 `log-purge` 任务，你可以手动执行这个任务或者把它放进 `crontab` (Linux 的定时计划表)。例如，下面的命令会清除在 `logging.yml` 里设置了 `purge: on`（这是默认值）的应用程序与环境的日志：

```
> symfony log-purge
```

为了性能与安全性的考虑，可以把 symfony 的日志文件存在多个小文件里而不是一个大文件。日志文件的理想存储策略是定期备份并清空主日志文件，只保留一定数量的备份。你可以在 `logging.yml` 里面开启并指定日志循环的设置。在例 16-6 中，定义的周期 `period` 为 7 天，历史记录 `history` (备份的数量) 为 10，在这样的设置下，有一个主日志文件和最多 10 个包含 7 天日志的备份文

件。当 7 天的周期结束以后，当前的日志文件会变成备份，最早的那个备份会被删除。

例 16-6 - 设置日志循环， `myapp/config/logging.yml`

```
prod:
  rotate: on
  period: 7      ## 日志文件默认的循环周期是 7 天
  history: 10    ## 最多保持 10 个历史记录
```

要执行日志循环，需要定期执行 `log-roate` 任务。这个任务只在 `roate` 为 `on` 的时候清除日志文件。你也可以在执行这个任务的时候指定一个应用程序和环境：

```
> symfony log-rotate myapp prod
```

日志备份存在 `logs/history/` 目录，它们以存储日期为后缀。

调试

不管你是个多么熟练的程序员，都会犯错，即使你使用 `symfony`。检查与了解错误是快速开发的一个关键。幸运的是，`symfony` 为开发者提供了好几种调试工具。

symfony 调试模式

`symfony` 有一个易于应用程序开发与调试的调试模式。当它开启时，会有下面的变化：

- 每次请求时都会检查配置文件，这样任何配置文件的改变都会立即生效，而不用每次清空配置文件缓存。
- 错误信息页面会用清晰有用的方式显示完整的跟踪信息，使你能够更快的发现出错之处。
- 有更多的调试工具可以使用(例如数据库查询的详情)。
- `Propel` 的调试模式也会开启，`Propel` 对象调用时候的错误也会通过 `Propel` 的机制显示在跟踪信息里。

另一方面，当调试模式关闭的时候，是这样处理的：

- `YAML` 配置文件只解析一次，转化为 `PHP` 文件存在 `cache/config/` 目录中。以后每次请求都会忽略 `YAML` 文件而直接使用缓存里的配置文件。这样，处理请求的速度大大提高了。
- 如果要重新处理配置文件，必须手动清除配置缓存。
- 处理请求时如果遇到错误会返回代码 500（内部服务器错误 `Internal Server Error`），并不会显示问题的详细情况。

调试模式通过各个应用程序的前端控制器开启。通过 SF_DEBUG 常量的值来定义，如例 16-7 所示。

例 16-7 在前端控制器里开启调试模式的例子，web/myapp_dev.php

```
<?php
```

```
define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/.'));
define('SF_APP',         'myapp');
define('SF_ENVIRONMENT', 'dev');
define('SF_DEBUG',       true);
```

```
require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.
SF_APP.DIRECTORY_SEPARATOR.'config'.DIRECTORY_SEPARATOR.'config.php');
```

```
sfContext::getInstance()->getController()->dispatch();
```

CAUTION 在生产服务器中，不要开启调试模式，也不要保留任何调试模式开启的前端控制器。调试模式不仅仅会影响速度，它也可能会暴露程序的内部信息。即使调试工具绝不会暴露数据库连接信息，出错页面的跟踪信息里面也包含了大量的对于不怀好意的访问者有用的危险信息。

symfony 异常

在调试模式下发生异常时，symfony 会显示包含了所有你需要用来寻找问题原因的异常提示信息。

异常提示信息写的很清楚，它指出了最有可能引起问题的东西。这些提示信息还经常能够提供解决问题的办法，并且对于最常见的问题，甚至会提供一个包含这个异常详细情况的 symfony 网页链接。异常页面会显示发生错误的 PHP 代码（包含语法高亮），还有完整的方法调用跟踪，如图 16-1 所示。你可以找到引起问题的第一个调用，以及调用这个方法的参数。

NOTE symfony 实际上靠 PHP 异常报告错误，这比 PHP 4 应用程序的报错方式好多了。例如，404 错误可以通过 sfError404Exception 来触发。

图 16-1 - symfony 应用程序异常信息的例子

[sfException]

The date is not in the expected UNIX timestamp format

stack trace

1. at ()
in SF_ROOT_DIR\apps\frontend\modules\test\templates\indexSuccess.php line 2 ...
 1. <?php use_helper('Date', 'Number', 'I18N') ?>
 2. <?php throw new sfException("The date is not in the expected UNIX timestamp format") ?>
 3. <?php \$now = time() ?>
 4. <?php echo format_datetime(\$now) ?>

 5. <?php \$sf_user->setCulture('en_US') ?>
2. at require()
in SF_ROOT_DIR\lib\symfony\view\sfPHPView.class.php line 109 ...
 106. // render to variable
 107. ob_start();
 108. ob_implicit_flush(0);
 109. require(\$sfFile);
 110. \$retval = ob_get_clean();
 - 111.
 112. return \$retval;
3. at sfPHPView->renderFile('C:\wamp\www\sf_sandbox\apps\frontend\modules\test\templates\indexSuccess.php')
in SF_ROOT_DIR\lib\symfony\view\sfPHPView.class.php line 240 ...
4. at sfPHPView->render()
in SF_ROOT_DIR\lib\symfony\filter\sfExecutionFilter.class.php line 170 ...
5. at sfExecutionFilter->execute(object('sfFilterChain'))
in SF_ROOT_DIR\lib\symfony\filter\sfFilterChain.class.php line 76 ...
6. at sfFilterChain->execute()
in SF_ROOT_DIR\lib\symfony\filter\sfFlashFilter.class.php line 50 ...

在开发阶段，symfony 异常对调试应用程序有巨大的帮助。

Xdebug 扩展

Xdebug PHP 扩展(<http://xdebug.org/>)可以增加 web 服务器记录的信息量。symfony 在自己的调试回馈里整合了 Xdebug 消息，所以建议在调试应用程序的时候使用这个扩展。这个扩展的安装在不同的平台下会有所不同，详细安装指南请参考 Xdebug 的网站。Xdebug 安装好后，你需要在 php.ini 里手工启用这个扩展。在 *nix 平台，可以通过增加下面这行代码实现：

```
zend_extension="/usr/local/lib/php/extensions/no-debug-non-zts-20041030/xdebug.so"
```

在 Windows 平台，Xdebug 扩展可以通过下面这行代码开启：

```
extension=php_xdebug.dll
```

例 16-8 是一个 Xdebug 配置的例子，这部分配置也要加到 php.ini 文件里

例 16-8 - Xdebug 配置的例子

```
; xdebug.profiler_enable=1
```

```
;xdebug.profiler_output_dir="/tmp/xdebug"
xdebug.auto_trace=1           ; 开启跟踪
xdebug.trace_format=0
;xdebug.show_mem_delta=0      ; 内存差异
;xdebug.show_local_vars=1
;xdebug.max_nesting_level=100
```

要启用 Xdebug 模式还需要重新启动你的 web 服务器。

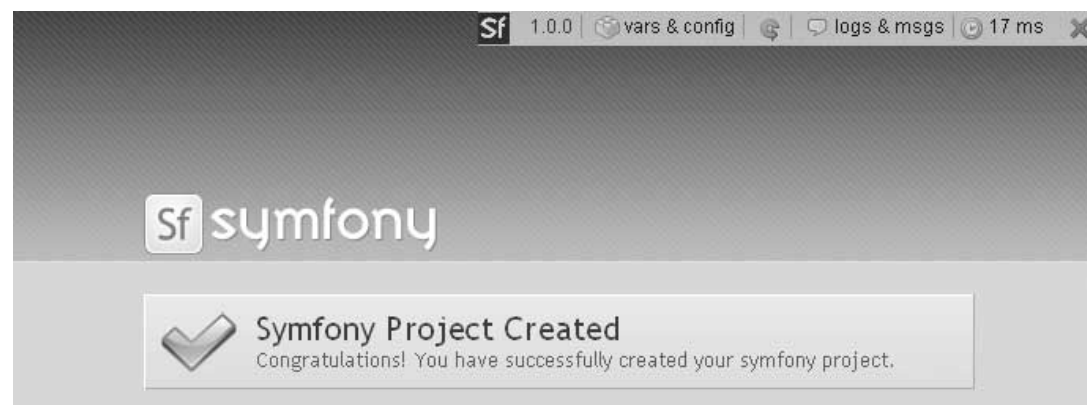
CAUTION 别忘记在生产平台下关闭 Xdebug 模式。否则会大大降低每个页面的执行速度。

网页调试工具条

日志文件包含了有趣的信息，不过它们不太容易阅读。一个最基本的任务是寻找某个特定请求的日志信息，如果有多个用户同时使用这个应用程序并且日志记录量比较大时，要完成这个任务会比较麻烦。这时你需要网页调试工具条。

这个工具条在浏览器里是一个在普通内容上的半透明的长方形，位于网页的右上角，如图 16-2 所示。通过他可以看到 symfony 日志事件，当前的配置信息，当前请求与回应对象的属性，当前请求引起的数据库查询的详情，还有本次请求相关的处理时间的图表。

图 16-2 - 网页调试工具显示在网页的右上角



调试工具条的背景颜色取决于发出请求时设置的日志信息最高级别的高低。如果最高级别是 debug，工具条将会是灰色背景。如果有 err 级别的消息，工具条的背景会是红色。

NOTE 不要把调试模式与网页调试工具条搞混。调试模式关闭的时候也可以显示调试工具条，不过这种情况下，它显示的信息要少的多。

要为某个应用程序开启网页调试工具条，打开 settings.yml 文件，寻找 web_debug 键。在 prod 与 test 环境下，web_debug 的默认值是 off，如果你需要你可以手动打开他。在 dev 环境，默认值是 on，如例 16-9 所示。

例 16-9 - 开启网页调试工具条, myapp/config/settings.yml

dev:

```
.settings:
  web_debug:          on
```

网页调试工具条会显示大量的信息/交互:

- 点击 **symfony** 图标切换工具条的大小。缩小的工具条不会遮住页面右上角的内容。
- 点击 **vars & config** 会显示请求, 回应, 设置, 全局标量还有 PHP 设置, 如图 16-3 所示。最顶上一行汇总了重要的设置, 例如调试模式, 缓存还有是否安装了 PHP 加速器 (如果没有开启显示红色, 如果开启显示绿色)。

图 16-3 - vars & config 显示请求的所有变量与常量



- 当缓存开启的时候, 工具条上会出现一个绿色的箭头。点击这个箭头会忽略缓存的内容重新生成当前页 (但是并不清空缓存)。
- 点击 **logs & msgs** 会显示当前请求有关的日志消息, 如图 16-4 所示。根据事件的重要性, 它们会显示成灰色、黄色或红色。还可以通过列表顶部的链接按照分类过滤日志消息列表。

图 16-4 - logs & msg 部分是当前请求的日志消息

Log and debug messages

Sf 1.0.0 vars & config logs & msgs 1 31 ms

[all] [none] sfAction sfContext sfController sfCreole sfFilter sfRequest sfRouting sfView

#	type	message
1	Context	initialization
2	Controller	initialization
3	Routing	match route [default] "/module/action/"
4	Request	request parameters array ('module' => 'article', 'action' => 'list')
5	Controller	dispatch request
6	Filter	executing filter "sfRenderingFilter"
7	Filter	executing filter "sfWebDebugFilter"
8	Filter	executing filter "sfCommonFilter"
9	Filter	executing filter "sfFlashFilter"
10	Filter	executing filter "sfExecutionFilter"
11	Action	call "articleActions->executeList()"
12	Creole	connect(): DSN: array ('database' => '*****', 'encoding' => '*****', 'hostspec' => '*****', 'password' => '*****', 'persistent' => '*****', 'phptype' => '*****', 'port' => '*****', 'username' => '*****',), FLAGS: 0
13	Creole	prepareStatement(): SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
14	Creole	executeQuery(): [8.66 ms] SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
15	View	initialize view for "article/list"
16	View	render "sf_app_dir/modules/article/templates/listSuccess.php"
17	View	decorate content with "sf_app_dir/templates/layout.php"
18	View	render "sf_app_dir/templates/layout.php"

NOTE 如果当前动作是跳转过来的，那么只有最后一次请求的日志会显示在 logs & msg 面板，所以日志文件仍然是调试所不可或缺的。

- 如果请求涉及数据库查询，工具条上会有一个数据库图标。点击这个图标可以看到数据库查询的详情，如图 16-5 所示。
- 工具条的右边有一个时钟的图标，它后面是处理当前请求所耗费的总时间。请注意网页调试工具条还有调试模式会降低请求的执行速度，所以请不要考虑请求执行的时间本身，而是只考虑两个页面之间的执行时间的差异。点击时钟图标来查看各个不同分类的细节所消耗的处理时间，如图 16-6 所示。symfony 显示消耗在请求处理的特定部分的时间。只有与当前请求相关的时间对优化有帮助，所以 symfony 内核消耗的时间不会显示。所以这些时间加起来比总时间少。
- 点击最右边的红色的叉会隐藏整个工具条。

图 16-5 - 数据库查询部分显示了当前请求的数据库查询执行的时间

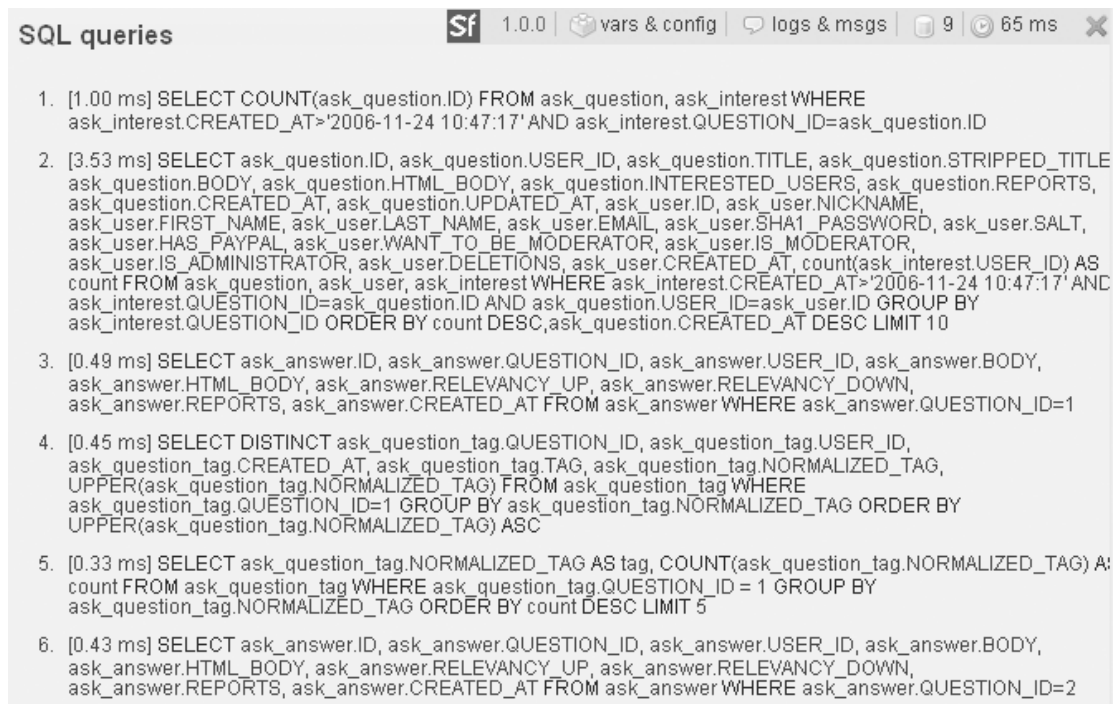


图 16-6 - 时钟图标按照类别显示执行时间

Timers		
type	calls	time (ms)
Configuration	14	60.42
Action "question/frontpage"	1	132.43
Database	9	7.56
View "Success" for "question/frontpage"	1	243.24
Partial "question/_question_list"	1	151.49
Partial "question/_question_block"	2	119.74
Partial "question/_interested_user"	2	12.99
Partial "moderator/_question_options"	2	2.80
Component "sidebar/default"	1	0.02
Partial "sidebar/_default"	1	25.62
Partial "tag/_tag_cloud"	1	2.09
Partial "question/_search"	1	0.97
Partial "sidebar/_rss_links"	1	3.08
Partial "sidebar/_moderation"	1	1.32
Partial "sidebar/_administration"	1	1.85

SIDEBAR 增加自己的计时器

symfony 使用 `sfTimer` 类来计算消耗在配置、模型、动作和视图的时间。使用这个对象，你可以计算一个特定的过程消耗的时间并把统计结果与其他计时器的结果一起显示在网页调试工具条里。这在做优化的时候很有用。

对某段代码初始化计时器，可以执行 `getTimer()` 方法。它会返回一个 `sfTimer` 对象并且开始计时。对这个对象执行 `addTime()` 方法可以停止计时。消耗的时间可以通过 `getElapsedTime()` 方法取得，并且显示在网页调试工具条里。


```
// 初始化计时器并开始计时
$timer = sfTimerManager::getTimer('myTimer');

// 作事情
...

// 停止计时器，增加消耗的时间
$timer->addTime();

// 取得结果（如果计时器没有停止，停止计时器）
$elapsedTime = $timer->getElapsedTime();
```

给每个计时器命名的好处是你可以多次调用它来计算时间的总和。例如，如果某个工具方法里的 `myTimer` 计时器在每次请求都会执行两次，第二次执行 `getTimer('myTimer')` 方法会从 `addTime()` 上次执行的那个时间点开始继续计时，所以会加上上次的时间。对计时器对象执行 `getCalls()` 方法会告诉你这个计时器使用的次数，这个数据也会在网页调试工具条里显示。

```
// 取得计时器执行的次数
$nbCalls = $timer->getCalls();
```

在 Xdebug 模式下，日志信息会更丰富。所有执行的 PHP 脚本文件和函数都会被记录，`symfony` 知道如何从内部日志链接到这些信息。日志信息表里的每行都有一个双箭头按钮，你可以点击这个按钮看到更多的相关信息。如果出了问题，Xdebug 会给你尽可能多的信息让你查找原因。

NOTE 网页调试工具条默认情况不会在 Ajax 回应和不是 HTML 类型的文档的回应里出现。在其他页面里，可以通过简单的调用 `sfConfig::set('sf_web_debug', false)` 用手动禁用网页调试工具条。

手动调试

能够获取框架的调试信息很不错，不过能够记录你自己的消息就更好了。`symfony` 提供了能够直接从动作与模板使用的捷径，来帮助你记录请求执行中的事件或者值。

与其他普通事件一样，你自定义的日志消息会出现在日志文件与网页调试工具条里。（例 16-4 给出了一个自定义日志消息的语法例子）自定义日志消息可以用来检查模板里的某个变量的值，例如：例 16-10 展示了如何使用网页调试工具条从模板获取开发者需要的回馈（也可以在动作里使用 `$this->logMessage()`）。

例 16-10 - 增加一条调试用的日志信息

```

<?php use_helper('Debug') ?>
...
<?php if ($problem): ?>
    <?php log_message(' {sfAction} been there', 'err') ?>
    ...
<?php endif ?>

```

使用 err 级别确保了这个问题肯定会在消息列表里出现，如图 16-7 所示。

图 16-7 - 网页调试工具条中 logs & msgs 部分的一个自定义调试消息

#	type	message
1	Context	initialization
2	Controller	initialization
3	Routing	match route [default] "/module/action/"
4	Request	request parameters array ('module' => 'article', 'action' => 'list',)
5	Controller	dispatch request
6	Filter	executing filter "sfRenderingFilter"
7	Filter	executing filter "sfWebDebugFilter"
8	Filter	executing filter "sfCommonFilter"
9	Filter	executing filter "sfFlashFilter"
10	Filter	executing filter "sfExecutionFilter"
11	Action	call "articleActions->executeList()"
12	Creole	connect(): DSN: array ('database' => '*****', 'encoding' => '*****', 'hostspec' => '*****', 'password' => '*****', 'persistent' => '*****', 'phptype' => '*****', 'port' => '*****', 'username' => '*****',), FLAGS: 0
13	Creole	prepareStatement(): SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
14	Creole	executeQuery(): [8.35 ms] SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT, blog_article.CREATED_AT FROM blog_article
15	View	initialize view for "article/list"
16	View	render "sf_app_dir/modules/article/templates/listSuccess.php"
17	Action	been there
18	View	decorate content with "sf_app_dir/templates/layout.php"
19	View	render "sf_app_dir/templates/layout.php"

如果你不想在日志里增加一行记录，但是想显示一小段消息或者一个值，应该使用 debug_message 而不是 log_message。这个动作方法（也有同名的辅助函数）在网页调试工具条中 logs & msgs 部分的顶部显示一条消息。例 16-11 是一个使用 debugMessage 的例子。

例 16-11 - 在调试工具条中增加一条消息

```

// 动作里
$this->debugMessage($message);

```

```
// 模板里
<?php use_helper('Debug') ?>
<?php debug_message($message) ?>
```

填充数据库

在应用程序开发过程中，开发者经常面对数据库填充的问题。有一些特殊地针对一些数据库的解决方案，不过它们都不能用于对象关系模型 ORM。由于有 YAML 和 sfPropelData 对象，symfony 可以自动将数据从文本转到数据库。虽然为了数据而建立文本文件看上去似乎比通过 CRUD 界面要消耗更多时间，但从长远来看这样会更节省时间。这个功能对自动存储和填充应用程序的测试数据很有用。

fixture 文件格式

symfony 可以从 data/fixtures/ 目录中简单格式的 YAML 文件里读取数据。fixture 文件是按照类组织的，每个类以类名为开头。类里面的记录都会有一个唯一的字符串作为标记，记录的值以一系列 fieldname: value 的形式表示。例 16-12 是一个填充数据库用的数据文件的例子。

例 16-12 - fixture 文件示例， data/fixtures/import_data.yml

```
Article:                                     ## 在 blog_article 表里增加记录
  first_post:                               ## 第一条记录的标签
    title:      My first memories
    content: |
      For a long time I used to go to bed early. Sometimes, when I
had put
      out my candle, my eyes would close so quickly that I had not
even time
      to say "I'm going to sleep."

  second_post:                               ## 第二条记录的标签
    title:      Things got worse
    content: |
      Sometimes he hoped that she would die, painlessly, in some
accident,
      she who was out of doors in the streets, crossing busy
thoroughfares,
      from morning to night.
```

symfony 用驼峰命名法把字段名转换成 setter 方法（setTitle(), setContent()）。这意味着你可以定义一个 password 键即使实际上表里没有

password 字段--只要在 User 对象定义一个 setPassword() 方法，然后你可以根据 password 的值生成其他字段的值（例如，加密过的 password）。

主键字段不需要定义，因为它是自动增加的字段，数据库知道如何处理。

created_at 字段也不用定义，因为 symfony 知道这个字段必须设置成记录创建时的系统时间。

导入数据

propel -load-data 任务可以从 YAML 文件读取数据导入到数据库。导入的数据库连接设置来自 databases.yml 文件，另外导入任务还需要一个引用程序名作为参数，还可以指定一个可选参数——环境名（默认值 dev）。

```
> symfony propel -load-data frontend
```

这条命令从 data/fixtures/ 目录里读取所有的 YAML fixtures 文件然后把里面的数据增加到数据库。默认情况，会替换数据库里的内容，但是如果最后一个参数是 append，这条命令就不会删除当前的数据。

```
> symfony propel -load-data frontend append
```

你也可以在这条命令里指定另一个 fixture 文件或者目录。这种情况下，需要增加一个到项目 data/ 目录的相对路径。

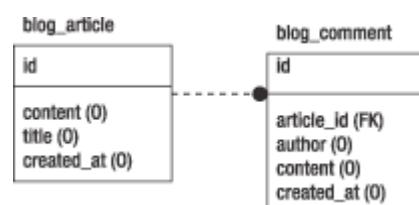
```
> symfony propel -load-data frontend myfixtures/myfile.yml
```

使用关联表

你现在知道如何在一个表里增加记录了，但是如何增加与其他表关联的记录呢？由于 fixture 数据里不包括主键，需要其他的方法进行关联。

让我们回到第 8 章的例子，blog_article 表与 blog_comment 表关联，如图 16-8 所示。

图 16-8 - 数据库关系模型示例



现在记录的标签会变得很有用。要在 Comment 增加一个到 first_post 这篇文章的字段，只要简单的在 import_data.yml 数据文件里增加如例 16-13 所示的几行就可以了。

例 16-13 - 增加一个与其他表关联的记录, `data/fixtures/import_data.yml`

Comment:

```
first_comment:
  article_id:  first_post
  author:      Anonymous
  content:     Your prose is too verbose. Write shorter sentences.
```

`propel-load-data` 任务会识别出你之前在 `import_data.yml` 里定义的标签, 然后获取对应的 `Article` 记录的主键去设置 `article_id` 字段。你不必看到记录的 ID, 只需要用它们的标签连接它们——这样更简单。

关联记录的唯一限制是相关的表里的记录必须在之前定义过, 也就是说你要按照顺序定义它们。数据文件按照从头到尾的顺序进行解析, 记录书写的顺序很重要。

一个数据文件可以包含多个类的定义。不过如果你需要在多个不同的表里增加很多数据, 你的 `fixture` 文件可能会变得很长而且不好处理。

`propel-load-data` 任务会解析 `fixtures/` 目录下的所有文件, 所以你可以把一个 `YAML fixture` 文件拆分成多个小文件。不过要注意外键会影响表的处理顺序, 要确定它们会按照正确的顺序解析, 可以在这些文件的文件名之前加上数字前缀。

```
100_article_import_data.yml
200_comment_import_data.yml
300_rating_import_data.yml
```

部署应用程序

`symfony` 提供了用来同步一个网站的两个版本的简单命令。这些命令主要用于把一个网站从开发服务器部署到最终的连接到因特网的服务器。

为 FTP 传输冻结项目

部署项目到生产环境最普通的方法是通过 `FTP`(或者 `SFTP`) 传输所有的文件。不过, 由于 `symfony` 项目使用 `symfony` 库, 除非你使用 `sandbox` 开发(不推荐), 或者 `symfony` 的 `lib/` 还有 `data/` 目录使用 `svn:externals` 连到你的项目, 这些库文件并不在项目目录里。不论你使用 `PEAR` 安装或者符号链接, 把复制 `symfony` 库文件复制到生产环境都是一个费时间的麻烦事。

所以 `symfony` 提供了一个“冻结”项目用的工具——它会把所有需要的 `symfony` 库复制到项目的 `data/`, `lib/` 还有 `web/` 目录。这样项目就变的跟 `sandbox` 类似的独立的应用程序了。

> symfony freeze

项目被冻结之后，你就可以把项目目录传到正式服务器，项目不需要 PEAR，符号链接等其他的东西就可以运行。

TIP 很多冻结的项目可以与在相同服务器上的不同版本的 symfony 上完美运行。

要把项目恢复到初始状态，使用 unfreeze 任务。这个任务会删除 data/symfony/、lib/symfony/和 web/sf/目录。

> symfony unfreeze

注意如果在冻结前已经有到 symfony 的符号链接，symfony 会按照记录的内容自动的重新建立这些符号链接。

使用 rsync 进行增量文件传输

第一次使用 FTP 传输项目目录的时候没什么问题，不过如果你需要上传应用程序的更新，只修改了一部分文件，这种时候 FTP 就不合适了。你需要重新上传整个项目，这很浪费时间与带宽；或者浏览到你所知到的修改过的文件，只上传修改过的文件。这也是个浪费时间的事情，而且容易出错。另外，传输的时候网站可能会不能访问或者出现问题。

symfony 提供的解决方案是使用 rsync 通过 SSH 同步文件。rsync (<http://samba.anu.edu.au/rsync/>) 是一个快速增量传输文件的命令行工具，并且是开源的。使用增量传输，只有修改过的数据才会被发送。如果文件没有改变，它就不会被发送到服务器。如果文件只有一部分改变了，那么仅仅是改变的部分会被发送。rsync 同步传输的优势是它只传输很小数量的数据并且很快。

symfony 使用基于 SSH 的 rsync 来确保数据传输的安全。越来越多的主机提供商支持通过 SSH 上传来确保服务器的安全，并且这是一个避免安全问题的好办法。

symfony 调用的 SSH 客户端会使用 config/properties.ini 文件里的连接信息。例 16-14 是一个生产服务器的连接信息的配置信息的例子。请在同步之前把你的正式服务器的设置写在这个文件里。你还可以指定一个 parameters 设置作为 rsync 命令行参数。

例 16-14 - 服务器同步的连接配置信息的例子，
myproject/config/properties.ini

```
[symfony]
name=myproject
```

```
[production]
host=myapp.example.com
port=22
user=myuser
dir=/home/myaccount/myproject/
```

NOTE 不要把这里的生产服务器（项目 `properties.ini` 文件里定义的服务器主机）与生产环境（用于生产的前端控制器与配置，在项目的配置文件里）搞混。

进行基于 SSH 的 `rsync` 同步需要几个命令，并且同步在应用程序的整个生命周期里经常发生。幸运的是，`symfony` 为我们提供了一条命令来自动化进行这些操作：

```
> symfony sync production
```

这条命令在 `dry` 模式下运行 `rsync` 命令；它会显示哪些文件需要同步但并不真正的同步它们。如果你想要同步它们，需要在这个命令后加一个 `go`。

```
> symfony sync production go
```

不要忘记同步后要在生产服务器上清空缓存。

TIP 有时生产环境会出现开发环境不存在的 bug。90%的情况是不同的版本（PHP，web 服务器或数据库）或配置造成的。为了避免这样的麻烦，你应该在应用程序的 `php.yml` 里定义需要的 PHP 配置，这样可以确保开发环境使用同样的设置。有关这个配置文件的详细内容请参考第 19 章。

-

SIDEBAR 你的应用程序完成了吗？

在把你的应用程序传到生产服务器之前，你需要确定它已经可以公开了。在你真正决定部署应用程序之前请现检查下面的事项：

错误页面的外观应该根据你的应用程序进行定制。关于如何定制 500 错误，404 错误和安全页面，以及网站不能访问时候的页面，请参考第 19 章还有本章的“管理投入使用的应用程序”部分。

`default` 模块应该从 `settings.yml` 文件的 `enabled_modules` 数组里去掉，这样就不会有 `symfony` 的默认页面出现。

`session` 处理机制使用一个客户端的 `cookie`，这个 `cookie` 的默认名字是 `symfony`。在部署应用程序之前，你也许应该把它改成别的名字从而避免暴露出

你的应用程序使用了 `symfony`。关于如何在 `factories.yml` 文件里定制这个 `cookie` 的名字请参考第 6 章。

项目 `web/` 目录下的 `robots.txt` 文件，默认是空的。你应该修改它的内容来告诉网页爬虫和其他网页机器人网站的哪些部分它们可以看哪些不能看。大多数时候，这个文件用来避免特定的 URL 被索引——例如，很消耗资源的页面，不需要被索引的页面（例如 `bug` 档案），或者那些网页机器人可能处理不了的 URL。

现代浏览器在用户第一次浏览你的应用程序的时候会请求 `favicon.ico` 文件，用来在地址栏和收藏夹里代表你的应用程序。提供这个文件不仅可以使你的应用程序看起来更好，而且可以避免服务器日志里出现大量的 404 错误。

忽略无关文件

如果你要同步你的 `symfony` 项目到生产服务器，有些文件和目录不需要传：

- 所有的版本控制目录（`.svn/`，`CVS/`等）还有它们的内容只在开发与整合时有用。
- 开发环境的前端控制器不需要被最终用户访问到。使用这个前端控制器时，调试还有日志工具会降低应用程序的速度而且会显示动作的核心变量。这些不需要被公众知道。
- 每次同步的时候，生产服务器上的项目目录下的 `cache/`和 `log/`目录不能被删除。这些目录也必须被忽略。如果你有 `stats/`目录，它也应该被忽略。
- 用户上传的文件不需要传输。`symfony` 项目的一个不错的做法是把所有的上传文件放在 `web/uploads/`目录。这使你能只通过一个目录就排除掉这些文件。

要使这些文件排除不被 `rsync` 同步，打开并修改 `myproject/config/`目录下的 `rsync_exclude.txt` 文件，每行包括一个文件、目录、或者匹配方式。`symfony` 文件组织的很有逻辑，这样的设计使手动排除的工作量达到最小化。见例 16-15。

例 16-15 - `rsync` 排除设置的例子， `myproject/config/rsync_exclude.txt`

```
.svn
/cache/*
/log/*
/stats/*
/web/uploads/*
/web/myapp_dev.php
```

NOTE `cache/`和 `log/`目录不应该与开发服务器同步，不过它们至少要在生产服务器中存在。如果生产服务器的项目目录中不包含它们，请手动建立它们。

管理投入使用的应用程序

生产服务器上使用最多的命令是 `clear-cache`。每次升级项目或者是修改配置的时候都要运行它（例如，执行完 `sync` 任务后）。

```
> symfony clear-cache
```

TIP 如果你的生产服务器没有提供命令行界面，你还是可以通过手动删除 `cache/` 目录的内容来清除缓存。

当你需要升级一个库或者大量数据的时候，你可以临时禁用你的应用程序。

```
> symfony disable APPLICATION_NAME ENVIRONMENT_NAME
```

默认情况，被禁用的应用程序会显示 `default/unavailable` 动作（位于框架里），不过你可以在 `settings.yml` 文件里自定义这个模块和方法。例 16-16 给出了一个例子。

例 16-16 - 设置被禁用的应用程序执行的动作， `myapp/config/settings.yml`

```
all:
  .settings:
    unavailable_module: mymodule
    unavailable_action: maintenance
```

`enable` 任务重新启用应用程序并清空缓存。

```
> symfony enable APPLICATION_NAME ENVIRONMENT_NAME
```

SIDEBAR 清除缓存时显示不可用页面

如果你把 `settings.yml` 里的 `check_lock` 参数设置成 `on`，在清除缓存的时候 `symfony` 会把应用程序锁起来，缓存清除完之前的所有请求都会被转到一个显示应用程序暂时无法使用的页面。如果缓存很大，清除需要的时间会大于几毫秒，如果你的网站流量很高，推荐使用这个设置。

不可用页面与你使用 `symfony disable` 命令后显示的页面相同（因为清除缓存的时候，`symfony` 不能够正常工作）。它位于 `$sf_symfony_data_dir/web/errors/` 目录，但是如果在你的项目的 `web/errors/` 目录创建自己的 `unavailable.php` 文件，`symfony` 会使用这个文件代替默认的。`check_lock` 默认设置是关闭的由于它会影响性能。

`clear-controllers` 任务可以清除 `web/` 目录里的前端控制器，只保留生产环境所需的控制器。如果你没有在 `rsync_exclude.txt` 里包含开发环境的前端控制器，这条命令可以确保你的应用程序的信息不会被生产环境的前端控制器所泄露。

```
> symfony clear-controllers
```

如果你从 SVN 库签出项目，文件和目录的权限会有错误。`fix-perms` 任务可以修复目录权限，例如，把 `log/` 和 `cache/` 目录的权限设置成 `0777`（这些目录必须可写 `symfony` 才能正常工作）。

```
> symfony fix-perms
```

SIDEBAR 在生产服务器上执行 `symfony` 命令

如果你的生产服务器有 PEAR 安装的 `symfony`，那么 `symfony` 命令行可以在所有的目录使用并且和开发环境下完全一样。如果是冻结过的项目，你需要在 `symfony` 之前加上 `php` 来启动任务：

```
// 通过 PEAR 安装的 symfony  
> symfony [options] <TASK> [parameters]
```

```
// 冻结的项目或者通过符号连接  
> php symfony [options] <TASK> [parameters]
```

总结

结合使用 PHP 日志和 `symfony` 日志，你可以方便的监视和调试你的应用程序。在开发过程中，调试模式，异常，还有网页工具条可以帮助你找到问题。你甚至可以在日志文件或者工具条里增加自定义消息来简化调试。

在开发与生产阶段，命令行界面提供了大量的帮助管理应用程序的工具。其中，数据库填充，冻结还有同步任务可以帮我们节省大量的时间。

第 17 章 扩展 symfony

无论你需要修改核心类的行为或者增加自制的功能，都不可避免地需要调整 symfony 的行为，没有一个框架可以预测到用户的所有特殊需求。事实上，这种情况很常见，因此 symfony 提供一个可以扩展已有类的机制，叫做 `mixins`。你甚至可以用 `factories` 设置替换掉 symfony 核心类。当建立好一个扩展，你可以非常方便地把它打包为一个插件，这时候就可以在你的其他应用程序中重复使用，或者被其他 symfony 用户使用了。

Mixins

目前 PHP 的限制中，最恼人的一个问题是一个类无法继承一个以上的类。另一个是你不能对一个已有类增加新的方法或者覆写现存方法。为了解决这两个限制并使框架真正的可以扩展，symfony 使用了一个叫做 `sfMixer` 的类。这绝不是一个什么烹饪工具，而是面向对象编程中的一个 `mixin` 概念。一个 `mixin` 是一组可被混入一个类的方法或者功能。

理解多重继承

多重继承就是指一个类同时继承多个类并继承了这些类的属性和方法。考察例 17-1：想像一下 `Story` 和 `Book` 这两个类，每一个类都有自己的属性和方法。

例 17-1 - 两个示例类

```
class Story
{
    protected $title = '';
    protected $topic = '';
    protected $characters = array();

    public function __construct($title = '', $topic = '', $characters = array())
    {
        $this->title = $title;
        $this->topic = $topic;
        $this->characters = $characters;
    }

    public function getSummary()
    {
        return $this->title.', a story about '.$this->topic;
```

```

    }
}

class Book
{
    protected $isbn = 0;

    function setISBN($isbn = 0)
    {
        $this->isbn = $isbn;
    }

    public function getISBN()
    {
        return $this->isbn;
    }
}

```

ShortStory 类继承自 Story 类，ComputerBook 类继承自 Book 类，逻辑上说，Novel 应该同时继承自 Story 和 Book 这两个类、使用他们的方法。不幸的是，这在 PHP 中无法实现。你不能像例 17-2 一样写 Novel 的声明。

例 17-2 - PHP 无法使用多重继承

```

class Novel extends Story, Book
{
}

```

```

$myNovel = new Novel ();
$myNovel ->getISBN();

```

一个可行的解决方法就是让 Novel 实现两个接口来代替继承两个类，但这会让你无法使用父类已有的方法。

Mixing 类

sfMixer 类用另一种方法来解决这个问题，假如类包含了合适的钩子，就能用它来扩展一个已有类。这个过程包含两个步骤：

- 声明类是可扩展的
- 在声明类后注册一个扩展（或者 mixins）

例 17-3 展示了如何用 sfMixer 实现 Novel 类。

例 17-3 - 通过 sfMixer 可以实现多重继承

```
class Novel extends Story
{
    public function __call($method, $arguments)
    {
        return sfMixer::callMixins();
    }
}

sfMixer::register('Novel', array('Book', 'getISBN'));
$myNovel = new Novel();
$myNovel->getISBN();
```

其中一个类（Story）被选择作为父类，这符合了 PHP 中只能继承一个类的特性。用__call()方法声明了 Novel 类是可扩展的。另一个类（Book）的方法随后也通过调用 sfMixer::register()加入到 Novel 类中。下面将详述这个过程。

调用 Novel 类中的 getISBN()方法时，可以实现例 17-2 同样的效果，不同之处在于这是由__call()方法的魔术和 sfMixer 的静态方法模拟的。getISBN()方法就是这么混入 Novel 类的。

SIDEBAR 什么时候使用 mixin

symfony 的 mixin 机制在很多情况下都是有用的。如前所述的模拟多重继承，就是其中之一。

你可以在声明后使用 mixins 来改变一个方法。例如，当建立一个图库时，你也许会实现一个 Line 对象——来显示一条线。它将会有四个属性（两端的坐标）和 draw()方法来渲染自身。ColoredLine 也应有相同的属性和方法，但多了一个额外的属性——color，来表示它的颜色。并且，ColoredLine 类的 draw()方法也和 Line 类中的有些不同，他会使用对象的颜色。你可以在 ColoredElement 类中包装一个图形单元的功能来处理颜色。这可以让你在其他图形单元（Dot, Polygon 及其他）重复使用 color 方法。在这个例子中，是用 Line 类的扩展，也就是把 ColoredElement 类混入的方式来实现 ColoredLine 类。最终的 draw()方法是由 Line 中原有的 draw 方法和 ColoredElement 中的 draw 方法混合而成的。

Mixins 也能被用来在已有类中新增一个方法。例如，symfony 框架中定义的一个叫做 sfAction 的行为类，你也许只想在你的应用程序中为 sfAction 增加一个自定义的方法——例如，把一个请求指向一个特别的网页服务。对此，PHP 无法完成，因为 PHP 的一个限制是在初始声明后无法修改 sfAction 的定义。但是 mixin 机制提供了一个完美的解决方案。

声明一个类是可扩展的

你必须在代码中插入一个或多个"钩子(hooks)"来声明此类是可扩展的，这样才能让 `sfMixer` 类可以识别。这些钩子其实是 `sfMixer::callMixins()` 方法调用。许多 `symfony` 类已经含有这些钩子，包括 `sfRequest`, `sfResponse`, `sfController`, `sfUser`, `sfAction` 和其他。

钩子可以放在类中的不同地方，根据扩展需要的程度而定：

- 要在类中增加新的方法，你必须在 `__call()` 方法中插入钩子并返回它的结果，如例 17-4 所示。

例 17-4 - 让一个类可以增加新方法

```
class SomeClass
{
    public function __call($method, $arguments)
    {
        return sfMixer::callMixins();
    }
}
```

- 要更改已有的方法，你必须在方法中插入钩子，如例 17-5 所示。由 `mixin` 类增加的代码会在放置钩子的地方被执行

例 17-5 - 让一个方法可被修改

```
class SomeOtherClass
{
    public function doThings()
    {
        echo "I'm working...";
        sfMixer::callMixins();
    }
}
```

你也许想要在一个方法中放入多个钩子。因此，你需要给钩子命名，这样你能在以后定义哪个钩子要被扩展，如例 17-6 所示。可以把钩子名字作为参数，让 `callMixins()` 方法来调用一个钩子。这个名字会在注册 `mixin` 的时候告诉方法需要去执行哪段 `mixin` 代码。

例 17-6 - 一个方法可以包含多个钩子，但是必须命名它们

```

class AgainAnotherClass
{
    public function doMoreThings()
    {
        echo "I'm ready.";
        sfMixer::callMixins('beginning');
        echo "I'm working...";
        sfMixer::callMixins('end');
        echo "I'm done.";
    }
}

```

当然，你可以配置一个新的、可扩展的方法这些技巧来建立新的类，如例 17-7 所示。

例 17-7 - 可用多种方法来扩展类

```

class BicycleRider
{
    protected $name = 'John';

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

    public function sprint($distance)
    {
        echo $this->name." sprints ".$distance." meters\n";
        sfMixer::callMixins(); // sprint()方法是可扩展的
    }

    public function climb()
    {
        echo $this->name.' climbs';
        sfMixer::callMixins('slope'); // sprint()方法可在此扩展
        echo $this->name.' gets to the top';
        sfMixer::callMixins('top'); // 也可以在这里
    }

    public function __call($method, $arguments)
    {
        return sfMixer::callMixins(); // BicycleRider 类是可扩展的
    }
}

```

```
}  
}
```

CAUTION 只有声明为可扩展的类才能用 `sfMixer` 来扩展。也就是说你无法利用此机制来扩展一个并没有订阅此服务的类。

注册扩展 (Extensions)

用 `sfMixer::register()` 方法在现有钩子上注册扩展。它的第一个参数是需要扩展的元素名，第二个参数是一个代表 `mixin` 的 PHP 调用名。

第一个参数的格式取决于你想扩展什么内容：

- 如果要扩展一个类，就用类名。
- 如果要用未命名的钩子扩展一个方法，用 `class:method` 模式。
- 要用已命名的钩子来扩展一个方法，用 `class:method:hook` 模式。

例 17-8 通过举例扩展例 17-7 定义的类来说明这个原理。可扩展的对象自动传递第一个参数给 `mixin` 方法（当然，除非可扩展对象是静态的）。`Mixin` 方法也获得了访问原方法参数的权限。

例 17-8 - 注册扩展

```
class Steroids  
{  
    protected $brand = 'foobar';  
  
    public function partyAllNight($bicycleRider)  
    {  
        echo $bicycleRider->getName(). " spends the night dancing.\n";  
        echo "Thanks ". $brand. "!\n";  
    }  
  
    public function breakRecord($bicycleRider, $distance)  
    {  
        echo "Nobody ever made ". $distance. " meters that fast before!\n";  
    }  
  
    static function pass()  
    {  
        echo " and passes half the peloton.\n";  
    }  
}
```



```

sfMixer::register('Bi cycleRider', array('Steroids', 'partyAllNight'));
sfMixer::register('Bi cycleRider:sprint', array('Steroids',
'breakRecord'));
sfMixer::register('Bi cycleRider:climb:slope', array('Steroids',
'pass'));
sfMixer::register('Bi cycleRider:climb:top', array('Steroids',
'pass'));

$superRider = new Bi cycleRider();
$superRider->climb();
=> John climbs and passes half the peloton
=> John gets to the top and passes half the peloton
$superRider->sprint(2000);
=> John sprints 2000 meters
=> Nobody ever made 2000 meters that fast before!
$superRider->partyAllNight();
=> John spends the night dancing.
=> Thanks foobar!

```

扩展机制不只是用来新增方法。`partyAllNight()`方法使用了 `Steroids` 类的一个属性。这就意味着当用 `Steroids` 类的方法来扩展 `Bi cycleRider` 类时，你实际上在 `Bi cycleRider` 对象里建立了一个新的 `Steroids` 实例。

CAUTION 你不能在现有类中增加两个同名方法。这是因为在 `__call()` 方法中 `callMixins()` 调用时使用的 `mixin` 方法名是关键字。同样，你不能在类中增加一个和类中方法同名的方法，因为 `mixin` 机制依靠 `__call()` 方法，所以在这种情况下，将会无法被调用到。

`register()`调用的第二个参数是 PHP 调用名，所以这可以是一个 `class::method` 数组，或是一个 `object->method` 数组，甚至是一个函数名，见例 17-9 的示例。

例 17-9 - 任何调用名都可以注册为 `Mixer` 扩展

```

// 用一个类方法作为调用名
sfMixer::register('Bi cycleRider', array('Steroids', 'partyAllNight'));

// 用一个对象方法作为调用名
$mySteroids = new Steroids();
sfMixer::register('Bi cycleRider', array($mySteroids,
'partyAllNight'));

// 用一个函数作为调用名
sfMixer::register('Bi cycleRider', 'die');

```

扩展机制是动态的，这就意味着尽管你已经实例化了一个对象，它还是能在类中利用进一步的扩展。见例 17-10 的示例。

例 17-10 - 扩展机制是动态的，甚至可以在实例化后发生

```
$simpleRider = new BicycleRider();
$simpleRider->sprint(500);
=> John sprints 500 meters
sfMixer::register('BicycleRider:sprint', array('Steroids',
'breakRecord'));
$simpleRider->sprint(500);
=> John sprints 500 meters
=> Nobody ever made 500 meters that fast before!
```

更精确的扩展

`sfMixer::callMixins()` 指令实际上是一个复杂处理过程的快捷方式。它自动在已注册的 `mixin` 上循环，逐个调用它们，传递给它当前对象和当前方法参数。简单来说，一个 `sfMixer::callMixins()` 调用行为或多或少就像例 17-11 所示。

例 17-11 - `callMixin()` 在已注册 `Mixin` 上循环并执行它们

```
foreach (sfMixer::getCallables($class.'.'.$method.'.'.$hookName) as
$callable)
{
    call_user_func_array($callable, $parameters);
}
```

如果想对返回值传递其他参数或者做其他设置，可以写一个详尽的 `foreach` 循环替换掉快捷方法。例 17-12 展示了一个在类中更完整的 `mixin`。

例 17-12 - 用定制的循环替换 `callMixin()`

```
class Income
{
    protected $amount = 0;

    public function calculateTaxes($rate = 0)
    {
        $taxes = $this->amount * $rate;
```

```

        foreach (sfMixer::getCallables('Income:calculateTaxes') as
$callable)
        {
            $taxes += call_user_func($callable, $this->amount, $rate);
        }

        return $taxes;
    }
}

```

```

class FixedTax
{
    protected $minIncome = 10000;
    protected $taxAmount = 500;

    public function calculateTaxes($amount)
    {
        return ($amount > $this->minIncome) ? $this->taxAmount : 0;
    }
}

```

```

sfMixer::register('Income:calculateTaxes', array('FixedTax',
'calculateTaxes'));

```

SIDEBAR Propel 行为

我们在前面第八章讨论过的 Propel 行为 (behavior) 是一个特殊的 mixin 类型：它们扩展了 Propel 生成的对象。让我们看一个例子。

Propel 对象对应数据库的表，他们都有一个 `delete()` 方法用来在数据库中删除相关的记录。但是因为不能删除一个 `Invoice` 类的记录，因此你想要把 `delete()` 方法改为让记录保留在数据库并设置 `is_deleted` 属性为 `true`。通常的对象检索方法 (`doSelect()`, `retrieveByPk()`) 只会考虑记录的 `is_deleted` 是否是 `false`。你也可以增加一个 `forceDelete()` 方法使你彻底删除记录。事实上，所有的这些修改可以包装为一个新的 `ParanoidBehavior` 类。最终 `Invoice` 类扩展自 `propel BaseInvoice` 类并把 `ParanoidBehavior` 的方法混入。

因此行为是 Propel 对象的 mixin。实际上，symfony 术语“行为”还包含了另一个意思：mixin 被包装成了插件。刚刚提到的 `ParanoidBehavior` 类对应于一个叫做 `sfPropelParanoidBehaviorPlugin` 的 symfony 插件。可以参照 symfony wiki (<http://www.symfony-project.com/trac/wiki/sfPropelParanoidBehaviorPlugin>) 来获得此插件更详细的安装使用说明。

最后一件关于行为的事情：如果想要使用行为，生成的 Propel 对象必须包含一些钩子。如果你不使用行为的话这样会降低一些执行效率及性能。所以钩子不是默认就激活的。要打开行为支持，需要在 `propel.ini` 文件中设置 `propel.builder.addBehaviors` 属性为 `true`，然后重建模块。

Factories

`factory` 是对某任务所用类的定义。`symfony` 依靠 `factories` 作为它的核心功能，就像控制器和用户会话（`session`）。例如，当框架需要建立一个新的请求对象的时候，它会在 `factory` 的定义里搜索用来创建这个对象的类名。请求对象默认的 `factory` 定义是 `sfWebRequest`，所以 `symfony` 建立这个类的对象来处理请求。使用 `factory` 定义最大的好处就是十分容易修改框架的核心功能：只要修改他的 `factory` 定义，然后 `symfony` 将使用自定义的请求类替代原有定义。

`factory` 定义存放在 `factories.yml` 配置文件中。例 17-13 展示了默认的 `factory` 定义文件。每一个定义是由自动载入类的名字和（可选）一系列的参数组成。例如，用户会话储存 `factory`（在 `storage:` 关键字下设置）使用了一个 `session_name` 参数来命名在客户电脑上的 `cookie`，由此来建立一个永久的用户会话。

例 17-13 - 默认的 `Factories` 文件在 `myapp/config/factories.yml`

```
cli:
  controller:
    class: sfConsoleController
  request:
    class: sfConsoleRequest

test:
  storage:
    class: sfSessionTestStorage

#all:
#  controller:
#    class: sfFrontWebController
#
#  request:
#    class: sfWebRequest
#
#  response:
#    class: sfWebResponse
#
#  user:
```

```

#   class: myUser
#
# storage:
#   class: sfSessionStorage
#   param:
#     session_name: symfony
#
# view_cache:
#   class: sfFileCache
#   param:
#     automaticCleaningFactor: 0
#     cacheDir: %SF_TEMPLATE_CACHE_DIR%

```

改变 factory 最好的办法就是建立一个新的继承自默认 factory 的类并加入新的方法。例如，用户会话 factory 设置为 myUser 类（在 myapp/lib/）并继承自 sfUser。利用已存在的 factory 去使用相同的机制。

例 17-14 - 重写 factories

```

// 在一个可以自动载入目录中建立一个 myRequest.class.php,
// 例如在 myapp/lib/
<?php

class myRequest extends sfRequest
{
// 你的代码放这里
}

// 在 factories.yml 中把此类作为请求`request` factory 声明
all:
  request:
    class: myRequest

```

桥接其他框架组件

如果你需要使用第三方类，但并不准备把这个类复制到 symfony 的 lib/ 目录中，你也许将会在其他目录中安装并让 symfony 使用这个类。因此，除非你使用 symfony 的自动加载机制，否则使用这个类也就意味着需要在代码中出现 require。

symfony（目前）尚未提供所有的工具。如果你想要一个 PDF 生成器，Google 地图的 API 或者是 Lucene 搜索引擎的 PHP 实现，你也许需要 Zend 框架中的一些类。如果你想在 PHP 中直接处理图片，连上一个 POP3 帐号读取 e-mail，或是

设计一个终端界面，你可以使用 eZcomponents 的一些类库。幸运的是，如果你正确的定义了设置，这些类库在 symfony 中会运行的很好。

首先，你需要在应用程序的 settings.yml 中声明（除非你通过 PEAR 安装了第三方的库）库根目录的路径：

```
.settings:
  zend_lib_dir:  /usr/local/zend/library/
  ez_lib_dir:    /usr/local/ezcomponents/
```

然后通过指定当 symfony 自动载入失败要用哪个库来扩展自动载入路由机制：

```
.settings:
  autoloading_functions:
    - [sfZendFrameworkBridge, autoloader]
    - [sfEzComponentsBridge, autoloader]
```

这个设置和在 autoloader.yml 定义的规则不同（在 19 章有关于此文件的更多信息）。autoloading_functions 设置了 bridge 类，autoloader.yml 中设置了路径和搜索规则。下面的描述说明了当新建一个未载入类的对象会产生状况：

1. symfony 自动载入功能(sfCore::splAutoload())首先查找在 autoloader.yml 中声明的路径中的类。
2. 如果没有找到，会逐个的调用在 sf_autoloading_functions 中声明的 callback 方法，直到找到为止。
3. sfZendFrameworkBridge::autoloader()
4. sfEzComponentsBridge::autoloader()
5. 如果以上都没有找到，而你使用的是 PHP 5.0.X，symfony 会抛出一个 exception 说明类不存在。从 PHP 5.1 开始，PHP 会自生成错误讯息。

这就是说，用自动载入机制可以使用其他的框架组件，甚至比它们自身的环境下更加方便。例如，如果你想在 PHP 中用 Zend 框架的 Zend_Search 组件来实现 Lucene 搜索引擎，你需要这样写：

```
require_once 'Zend/Search/Lucene.php';
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
...
```

用 symfony 桥接 Zend 框架的话，就简单了。写法如下：

```
$doc = new Zend_Search_Lucene_Document(); // 此类可以自动载入
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
```

...

在\$sf_symfony_lib_dir/addon/bridge/目录中有可用的 bridge。

插件

你也许会需要重用自行开发的 symfony 应用程序中的一些代码。如果可以把这些代码包装为单一的类：把这个类放在其他应用程序的 lib/目录下，自动载入机制会处理其他的事情。但是如果这些代码是分散的，不单只是一个文件，就像是一个完整的管理界面生成器用的主题或者是一组自动完成你所喜欢的特殊效果的 Javascript 文件和辅助函数，单纯复制这些文件不是最好的解决方案。

插件提供了一个把分散在一系列文件中的代码包装起来并能在几个项目中重新使用的方法。在插件里，你能包装类、过滤器、mixins、辅助函数、配置文件、任务、模块、设计（schema）和模型扩展、fixtures、网页资源等。插件易于安装、升级、卸载。他们可以发布为 .tgz 压缩包，PEAR 包，或者是直接从版本库里检出。打包为 PEAR 的插件有利于管理关联关系，易于升级和自动维护。symfony 载入机制会把插件考虑在内，项目中可以像使用 symfony 的功能一样使用插件提供的功能。

因此，通常来说，插件就是 symfony 打包的扩展。有了插件，不仅可以在不同的应用程序中使用你的代码，而且可以使用其他的第三方的扩展程序。

查找 symfony 插件

symfony 官方网站上有一个用来发布 symfony 插件的网页。它在网站的 wiki 部分，可以通过以下网址访问：

<http://www.symfony-project.com/trac/wiki/symfonyPlugins>

每一个插件都有自己独立的页面，包含了详细的安装指南和文档。

其中的一些插件是社区贡献的，一些是 symfony 核心开发组发布的。symfony 核心开发组发布的插件包括下面这些：

- sfFeedPlugin: 自动处理 RSS 和 Atom feeds
- sfThumbnailPlugin: 建立图片的缩略图
- sfMediaLibraryPlugin: 允许上传和管理媒体文件，包括一个富文本编辑器的扩展（可以实现在富文本编辑器里选择图片）
- sfShoppingCartPlugin: 购物车管理
- sfPageNavigationPlugin: 基于 sfPager 对象提供了传统和 ajax 的页面控制
- sfGuardPlugin: 在 symfony 标准安全功能上提供验证，认证和其他用户管理功能

- `sfPrototypePlugin`: 提供了 Prototype 和 `script.aculo.us` 这两个 Javascript 框架的 Javascript 文件，这是一个独立库（区别于 symfony 自带的 prototype 和 `script.aculo.us` 文件）
- `sfSuperCachePlugin`: 将页面写入网页根目录下的 `cache` 目录，使服务器直接输出这些内容，从而最大限度的提高速度
- `sfOptimizerPlugin`: 优化应用程序的代码让其在生产环境中执行的更快（详情看下一章）
- `sfErrorLoggerPlugin`: 把 404 和 505 错误记录到数据库中，提供一个管理模块来浏览这些错误
- `sfSslRequirementPlugin`: 为动作提供了 SSL 加密支持

Wiki 上也有一些扩展 Propel 对象的插件，我们称它做行为（behaviors）。在他们中，你可以找到这些功能：

- `sfPropelParanoidBehaviorPlugin`: 禁止直接删除功能，改为更新 `deleted_at` 列
- `sfPropelOptimisticLockBehaviorPlugin`: 对 Propel 对象实现优化锁定

你应该经常去看一下 symfony wiki，那里会随时增加新的插件，他们会给你的 web 应用程序开发带来很多便捷。

除了 symfony wiki 之外，还有另外一种方式来发布插件，提供压缩文件的插件下载，或是在 PEAR 频道中，或者它们放在公开的版本控制库中。

安装插件

插件安装过程会因为不同的打包方式而有所不同。可以在插件下载页找到说明文件或者安装指南。同时，在安装了插件后都需要更新一下 symfony 缓存。

插件是在项目级别安装的应用程序。接下来章节中讨论的所有的方法都是把插件安装到 `myproject/plugins/pluginName/` 目录下。

PEAR 插件

在 symfony wiki 列出的插件都是 PEAR 包形式的。可以使用 `plugin-install` 加上完整的 URL 地址来安装插件，如例 17-15 所示。

例 17-15 - 从 symfony wiki 安装插件

```
> cd myproject
> php symfony plugin-install http://plugins.symfony-
project.com/pluginName
> php symfony cc
```


另一种方法是，你可以把插件下载到硬盘中，然后从硬盘上安装。这种模式下把频道名字替换为插件包的完整路径名，如例 17-16 所示。

例 17-16 - 安装一个已下载的 PEAR 包

```
> cd myproject
> php symfony plugin-install /home/path/to/downloads/pluginName.tgz
> php symfony cc
```

一些插件存在 PEAR 频道中。需要用 `plugin-install` 来安装，别忘了设置频道名字，如例 17-17 所示。

例 17-17 - 从 PEAR 频道安装插件

```
> cd myproject
> php symfony plugin-install channelName/pluginName
> php symfony cc
```

所以无论是从 `symfony wiki`，PEAR 频道或者是下载的 PEAR 包安装，这三种方式都用了 PEAR 包，都可用术语“PEAR 插件”来表示。

压缩包形式的插件

一些插件发布的是一个压缩包。只要解压缩到项目的 `plugins/` 目录就能完成安装。如果插件包含 `web/` 子目录，复制一份或者做一个符号链接到项目的 `web/` 目录，如例 17-18 所示。最后，别忘了清空缓存。

例 17-18 - 从压缩包安装插件

```
> cd plugins
> tar -zxpf myPlugin.tgz
> cd ..
> ln -sf plugins/myPlugin/web web/myPlugin
> php symfony cc
```

从版本库安装插件

插件有时候放在他们的版本库中。只要检出到 `plugins/` 目录即可，但如果你的项目本身就在版本控制下就会有一些问题。

另一种选择，你可以声明插件是依赖于外部库，这样，每次更新你的项目源代码的时候也会更新插件的源代码。例如，`Subversion` 在 `svn:externals` 中配置外部依赖。因此你能修改这个属性然后更新源代码来增加插件，如例 17-19 所示。

例 17-19 - 从版本库安装插件

```
> cd myproject
> svn propedit svn:externals plugins
    pluginName  http://svn.example.com/pluginName/trunk
> svn up
> php symfony cc
```

NOTE 如果插件包含 web/ 目录，与压缩文件的插件一样必须为它建立一个符号链接。

激活插件模块

一些插件包含完整的模块。模块插件和标准插件的区别就是模块插件不会在 myproject/apps/myapp/modules/ 目录中（易于更新）。他们还需要在 settings.yml 中激活，如例 17-20 所示。

例 17-20 - 在 myapp/config/settings.yml 中激活一个插件模块

```
all:
  .settings:
    enabled_modules: [default, sfMyPluginModule]
```

这是为了避免当插件模块在应用程序不需要的时候出现，这会导致安全违例。试想一个插件提供了前台和后台模块。你会让前台模块在前台应用程序中工作，后台只是在后台应用程序中。这就是为什么插件模块默认是不激活的。

TIP 只有默认模块是默认被激活的。因为它属于框架，所以这不是一个真正的插件模块，它位于 \$sf_symfony_data_dir/modules/default/ 中。这个模块提供了配置文件和默认的 404 错误文件和需要证书错误提示。如果不想使用 symfony 默认页面，只要从 enabled_modules 设置中移除这个模块即可。

列出已安装的插件

看一下项目的 plugins/ 目录就能知道哪些插件已经安装好了，plugin-list 任务告诉你一些更多的信息：每个已安装插件的版本号，频道名字（例 17-21）。

例 17-21 - 列出已安装的插件

```
> cd myproject
> php symfony plugin-list
```

```
Installed plugins:
sfPrototypePlugin          1.0.0-stable # pear.symfony-
project.com (symfony)
```

<code>sfSuperCachePlugin</code>	<code>1.0.0-stable # pear.symfony-</code>
<code>project.com (symfony)</code>	
<code>sfThumbnail</code>	<code>1.1.0-stable # pear.symfony-</code>
<code>project.com (symfony)</code>	

升级和卸载插件

要去卸载一个 PEAR 插件，只需要在项目根目录执行 `plugin-uninstall`，如例 17-22 所示。需要在插件名前加入安装的频道名（使用 `plugin-list` 来确认频道名）。

例 17-22 - 卸载插件

```
> cd myproject
> php symfony plugin-uninstall pear.symfony-
project.com/sfPrototypePlugin
> php symfony cc
```

TIP 一些频道有别名。例如，`pear.symfony-project.com` 频道也可以叫做 `symfony`，这就是说你可以卸载执行 `php symfony plugin-uninstall symfony/sfPrototypePlugin` 来卸载 `sfPrototypePlugin`，和例 17-22 效果一样。

要卸载一个压缩包形式的插件或者一个 SVN 插件，需要手工从项目的 `plugins/` 和 `web/` 目录把文件移除，然后清空缓存。

要升级一个插件，可以使用 `plugin-upgrade`（用 PEAR 安装的插件）或做一个 `svn update`（如果是从版本库中获得的插件）。压缩包形式的插件升级不是很方便。

解读插件

插件是用 PHP 语言写的。如果你能了解应用程序是如何组织的，你就能了解插件的结构了。

插件文件结构

插件目录组织有点类似一个项目目录。插件文件必须放在正确的目录，这样才能在 `symfony` 需要的时候自动载入。

例 17-23 - 插件的文件结构

```
pluginName/
  config/
    *schema.yml      // 数据模型
    *schema.xml
```

```

    config.php          // 插件配置
data/
  generator/
    sfPropelAdmin
      */              // 管理界面生成器主题
      templates/
      skeleton/
  fixtures/
    *.yaml            // Fixtures 文件
  tasks/
    *.php             // Pake 任务
lib/
  *.php              // 类
  helper/
    *.php            // 辅助函数
  model/
    *.php            // 模型类
modules/
  */                // 模块
  actions/
    actions.class.php
  config/
    module.yaml
    view.yaml
    security.yaml
  templates/
    *.php
  validate/
    *.yaml
web/
  *                  // 网页资源文件

```

插件能做的事

插件能包含很多东西。他们的内容可以在通过命令行调用任务的时候和程序运行的时候处理。但要让插件正常工作的话，你必须注意以下几个方面：

- 数据模型是由 `propel` -任务检查的。当在项目中调用 `propel -build-model` 时候，你重建了项目模型和所有的插件模型。注意插件模型必须总是在 `plugins.plugins.lib.model` 中有一个包（package）属性，如例 17-24 所示。

例 17-24 - 在 `myPlugin/config/schema.yaml` 中的插件模式声明

`propel:`

```

_attributes:    { package: plugins.myPlugin.lib.model }
my_plugin_foobar:
  _attributes:  { phpName: myPluginFoobar }
  id:
  name:        { type: varchar, size: 255, index: unique }
  ...

```

- 插件配置文件必须包含在插件的引导脚本中（`config.php`）。这个文件会在应用程序和项目配置程序之后执行，所以 `symfony` 会在那时引导。例如你可以使用这个文件来增加 PHP 包含路径或通过 `mixin` 来扩展已有类。
- `Fixtures` 文件位于插件的 `data/fixtures/` 目录中，由 `propel-load-data` 任务处理。
- 插件中的任务在插件安装好后就能立即在 `symfony` 命令行中使用了。在任务前面加个前缀会比较有意义——例如，加上插件名字。输入 `symfony` 可以看到所有的任务，包含通过插件安装的任务。
- 自定义类会自动载入，就如你放在 `lib/` 目录下一样。
- 辅助函数会在模板调用 `use_helper()` 的时候自动找到。他们必须在其中一个插件的 `lib/` 目录的 `helper/` 子目录下。
- 模型类在 `myplugin/lib/model/` 意味着模型类是由 `Propel` 生成器生成的（在 `myplugin/lib/model/om/` 和 `myplugin/lib/model/map/`）。这就是说，他们会自动被载入。小心你可以在自己的项目目录中覆写生成的插件模型类。
- 模块提供了新的行为来访问外部，也就是在你应用程序的 `enabled_modules` 设置中声明的那些。
- 网页资源（图片，脚本，样式表及其他）需要给服务器使用。当你通过命令行安装了一个插件的时候，如果服务器允许的话 `symfony` 会建立一个项目 `web/` 目录的符号连接，或者 `copy` 一份模块 `web/` 目录下的内容到项目中。如果是通过压缩包或者版本控制安装的插件，你必须手动复制插件的 `web/` 目录（插件的 `README` 应有提示）。

手工设置插件

`plugin-install` 无法处理一些元素，所以需要在安装过程中手动设置：

- 插件代码中可以使用自定义的应用程序配置（例如，使用 `sfConfig::get('app_myplugin_foo')`），但是你不能把默认值放在插件 `config/` 目录下的 `app.yml` 文件中。要处理默认值的话，用 `sfConfig::get()` 方法的第二个参数。这些设置可以在应用层被覆盖（参考例 17-25 的示例）。
- 自定义的路由规则必须在应用程序的 `routing.yml` 中手动增加
- 自定义的过滤器必须在应用程序的 `filters.yml` 中手动增加
- 自定义的 `factories` 必须在应用程序的 `factories.yml` 中手动增加

通常来说，所有的配置都可以归结于一个需要手动配置的应用程序配置文件。需要这样手工配置的插件应该会包含一个 README 文件详细介绍安装过程。

为应用程序定制插件

想要定制插件的时候，绝对不要去修改在 `plugins/` 目录下的代码。如果修改了，在更新插件的时候，所有的修改都会丢失的。为了可以自定义，插件提供了自定义的设置，这些设置是可以覆盖的。

设计优秀的插件可以在应用程序的 `app.yml` 中修改设置。如例 17-25 所示。

例 17-25 - 用应用程序的配置文件自定义插件

```
// 插件代码示例
$foo = sfConfig::get('app_my_plugin_foo', 'bar');

// 在应用程序的 app.yml 中设置'foo'的默认值('bar')
all:
  my_plugin:
    foo:      barbar
```

通常模块设置和他们的默认值都会在插件的 README 文件中有说明。

可以在你的应用程序中创建同名模块来替换插件模块的默认内容。这不是真正的覆写，因为使用应用程序中的元素来替换掉插件中的同名元素。如果创建和插件同名模板和配置文件将会工作的很好。

另一方面，如果插件要提供一个用来覆写它本身的行为的模块的话，插件模块中的 `actions.class.php` 必须为空并且是从自动载入类中继承的，因此类的方法也可以通过应用程序模块的 `actions.class.php` 继承。

例 17-26 - 自定义插件行为

```
// 在 myPlugin/modules/mymodule/lib/myPluginmymoduleActions.class.php
class myPluginmymoduleActions extends sfActions
{
    public function executeIndex()
    {
        // 一些代码
    }
}

// 在 myPlugin/modules/mymodule/actions/actions.class.php
```

```

class mymoduleActions extends myPluginmymoduleActions
{
    // 空
}

// 在 myapp/modules/mymodule/actions/actions.class.php
class mymoduleActions extends myPluginmymoduleActions
{
    public function executeIndex()
    {
        // 覆写插件代码
    }
}

```

如何写一个插件

plugin-install 只能安装那些包装为 PEAR 包的插件。记住这些插件可以发布在 symfony wiki 中，PEAR 频道中，或是下载的文件。因此，如果你想要制作一个插件，最好把它包装为 PEAR 压缩文件包。另外，PEAR 包的插件易于升级，可以声明关联，并自动将资源文件部署在 web/ 目录中。

文件组织

假设你开发了一个新的功能并想把它打包为一个插件。第一步是有逻辑的组织文件，让 symfony 载入机制可以在需要的时候找到它们。为了这个目的，你必须用例 17-23 给出的结构。例 17-27 展示了插件 sfSamplePlugin 文件的结构示例。

例 17-27 - 打包的插件的文件列表示例

```

sfSamplePlugin/
  README
  LICENSE
  config/
    schema.yml
  data/
    fixtures/
      fixtures.yml
    tasks/
      sfSampleTask.php
  lib/
    model/
      sfSampleFooBar.php
      sfSampleFooBarPeer.php
    validator/

```

```

    sfSampleValidator.class.php
modules/
  sfSampleModule/
    actions/
      actions.class.php
    config/
      security.yml
    lib/
      BaseSFSampleModuleActions.class.php
    templates/
      indexSuccess.php
web/
  css/
    sfSampleStyle.css
  images/
    sfSampleImage.png

```

对于开发来说，插件目录的位置(例 17-27 中的 `sfSamplePlugin/`)并不重要。它可以在硬盘的任何地方。

TIP 用一个已有插件的例子，作为初次开发插件的模板，试着去重写它们的名字和文件结构。

建立 package.xml 文件

插件制作下一步就是在插件根目录增加一个 `package.xml` 文件。`package.xml` 遵循 PEAR 语法。看一下例 17-28 的标准 symfony 插件 `package.xml`。

例 17-28 - symfony 插件的 `package.xml` 示例

```

[xml]
<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.6" version="2.0"
xmlns="http://pear.php.net/dtd/package-2.0"
xmlns:tasks="http://pear.php.net/dtd/tasks-1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pear.php.net/dtd/tasks-1.0
http://pear.php.net/dtd/tasks-1.0.xsd
http://pear.php.net/dtd/package-2.0 http://pear.php.net/dtd/package-
2.0.xsd">
  <name>sfSamplePlugin</name>
  <channel>pear.symfony-project.com</channel>
  <summary>symfony sample plugin</summary>
  <description>Just a sample plugin to illustrate PEAR
packaging</description>

```



```
<lead>
  <name>Fabien POTENCIER</name>
  <user>fabpot</user>
  <email>fabien.potencier@symfony-project.com</email>
  <active>yes</active>
</lead>
<date>2006-01-18</date>
<time>15:54:35</time>
<version>
  <release>1.0.0</release>
  <api>1.0.0</api>
</version>
<stability>
  <release>stable</release>
  <api>stable</api>
</stability>
<license uri="http://www.symfony-project.com/license">MIT
license</license>
<notes>-</notes>
<contents>
  <dir name="/">
    <file role="data" name="README" />
    <file role="data" name="LICENSE" />
    <dir name="config">
      <!-- model -->
      <file role="data" name="schema.yml" />
    </dir>
    <dir name="data">
      <dir name="fixtures">
        <!-- fixtures -->
        <file role="data" name="fixtures.yml" />
      </dir>
      <dir name="tasks">
        <!-- tasks -->
        <file role="data" name="sfSampleTask.php" />
      </dir>
    </dir>
    <dir name="lib">
      <dir name="model">
        <!-- model classes -->
        <file role="data" name="sfSampleFooBar.php" />
        <file role="data" name="sfSampleFooBarPeer.php" />
      </dir>
      <dir name="validator">
```

```

    <!-- validators -->
    <file role="data" name="sfSampleValidator.class.php" />
  </dir>
</dir>
<dir name="modules">
  <dir name="sfSampleModule">
    <file role="data" name="actions/actions.class.php" />
    <file role="data" name="config/security.yml" />
    <file role="data" name="lib/BasesfSampleModuleActions.class.php"
  />
    <file role="data" name="templates/indexSuccess.php" />
  </dir>
</dir>
<dir name="web">
  <dir name="css">
    <!-- stylesheets -->
    <file role="data" name="sfSampleStyle.css" />
  </dir>
  <dir name="images">
    <!-- images -->
    <file role="data" name="sfSampleImage.png" />
  </dir>
</dir>
</dir>
</contents>
<dependencies>
  <required>
    <php>
      <min>5.0.0</min>
    </php>
    <pearinstaller>
      <min>1.4.1</min>
    </pearinstaller>
    <package>
      <name>symfony</name>
      <channel>pear.symfony-project.com</channel>
      <min>1.0.0</min>
      <max>1.1.0</max>
      <exclude>1.1.0</exclude>
    </package>
  </required>
</dependencies>
<phprelease />
<changelog />

```

</package>

这里有趣的部分是<contents>和<dependencies>标签，我们会在下面介绍。其余的标签，对 symfony 没有什么特别的意义，关于 package.xml 格式详细信息你可以参考 PEAR 在线手册(<http://pear.php.net/manual/en/>)。

内容

<contents>标签是用来描述插件的文件结构的。这会让 PEAR 知道哪些文件要复制到什么地方。文件结构用<dir>和<file>标签。所有的<file>标签必须有一个 role="data"属性。例 17-28 的<contents>部分描述了与例 17-27 完全一致的目录结构。

NOTE 并不是强制使用<dir>标签，因为你可以在<file>标签中定义 name 来使用相对路径。无论如何，我们建议 package.xml 文件保持可读状态。

插件依赖性

插件是设计用来和一些版本的 PHP、PEAR、symfony、PEAR 包或者其他插件一起工作的。在<dependencies>标签声明这些依赖性告诉了 PEAR 在安装时要检查这些包是否已经安装了，如果没有，会给出异常信息。

你需要声明 PHP，PEAR 和 symfony 的依赖性，最小的需求，要保证至少其中之一对应了你的安装。如果你不知道该设置什么，建立设置 PHP5.0、PEAR 1.4 和 symfony 1.0。

另外，建议在每个插件里设置一个最高允许的 symfony 版本号。这会导致在更高版本的框架下试着使用插件时会产生一个错误讯息，同时这会帮助插件作者在重新发布插件前确认插件在此版本下工作是否正常。在下载或者升级的时候得到一个错误信息总好过于插件遇到错误时不报错。

打包插件

PEAR 组件有一个命令(pear package)用来创建.tgz 压缩文件包，例 17-29 是从一个包含 package.xml 目录中使用命令的例子。

例 17-29 - 把插件打包为 PEAR 包

```
> cd sfSamplePlugin
> pear package
```

```
Package sfSamplePlugin-1.0.0.tgz done
```

当你的插件打包好之后，自行安装测试一下，如例 17-30 所示。

例 17-30 - 安装插件

```
> cp sfSamplePlugin-1.0.0.tgz /home/production/myproject/
> cd /home/production/myproject/
> php symfony plugin-install sfSamplePlugin-1.0.0.tgz
```

就如<contents>标签描述的，包里的文件最终会放在你的项目的不同的目录下，例 17-31 展示了 sfSamplePlugin 的文件安装后的位置。

例 17-31 - 插件的文件安装在 plugins/和 web/目录下

```
plugins/
  sfSamplePlugin/
    README
    LICENSE
    config/
      schema.yml
    data/
      fixtures/
        fixtures.yml
      tasks/
        sfSampleTask.php
    lib/
      model/
        sfSampleFooBar.php
        sfSampleFooBarPeer.php
      validator/
        sfSampleValidator.class.php
    modules/
      sfSampleModule/
        actions/
          actions.class.php
        config/
          security.yml
        lib/
          BaseSfSampleModuleActions.class.php
        templates/
          indexSuccess.php
web/
  sfSamplePlugin/                                ## 复制或者做链接，这取决于系统
  css/
    sfSampleStyle.css
  images/
    sfSampleImage.png
```

在应用程序中测试插件是否正常。如果工作正常，你可以准备在其他项目中使用它或者发布到 symfony 社区去。

在 symfony 项目主页发布你的插件

通过 `symfony-project.com` 网站发布的 `symfony` 插件拥有广大的用户。只要遵循这些步骤，你自己的插件也能通过这种方式发布：

1. 确保 `README` 文件描述了如何去安装和使用你的插件，`LICENSE` 文件说明了详细的许可信息。用 `Wiki` 语法格式(<http://www.symfony-project.com/trac/wiki/WikiFormatting>)来组织 `README` 文件。
2. 通过 `pear package` 命令为你的插件建立一个 `PEAR` 包，测试它。`PEAR` 包必须命名成 `sfSamplePlugin-1.0.0.tgz` (`1.0.0` 是插件版本号)格式。
3. 在 `symfony wiki` 建立一个叫做 `sfSamplePlugin` (`Plugin` 是必须的后缀)的新页面。在这个页面里，描述了插件的使用方法，许可信息，依赖信息和安装过程。你可以重复使用插件的 `README` 文件。请参考检查现存插件的 `wiki` 页。
4. 把你的 `PEAR` 包放到 `wiki` 页里(`sfSamplePlugin-1.0.0.tgz`)。
5. 在可用插件列表页面——也是一个 `wiki` 页面(<http://www.symfony-project.com/trac/wiki/SymfonyPlugins>)增加一个新的插件页 (`[wiki: sfSamplePlugin]`)。

如果遵循这个流程，用户只要在项目目录中输入以下命令就能安装你的插件了：

```
> php symfony plugin-install http://plugins.symfony-project.com/sfSamplePlugin
```

命名约定

保持 `plugins/` 目录干净，确保所有的插件名字用的都是驼峰命名方法并且都用 `Plugin` 作为结尾（例如，`shoppingCartPlugin`，`feedPlugin`）。在命名你的插件前，检查一下是否已经有插件叫这个名字了。

NOTE 如果插件要用到 `Propel`，名字应该包含 `Propel`。例如，一个验证插件使用了 `Propel` 数据访问对象就应该叫做 `sfPropelAuth`。

插件应该有一个 `LICENSE` 文件来描述使用的条件并选定使用范围。我们也建议增加一个 `README` 文件来描述版本更改，插件的目标，它的作用，安装和配置指引，等等。

总结

`symfony` 类包含了 `sfMixer` 钩子，让它们可以在应用层被更改。`mixin` 机制允许多重继承和 `PHP` 禁止的动态覆盖。因此尽管你不许要修改核心类——`factories` 的配置文件，但是还是能十分方便的扩展 `symfony` 的功能。

很多扩展已经有了；他们被打包为插件，可以通过 `symfony` 命令行方便的安装，升级，卸载。建立一个插件就和建立一个 PEAR 包一样容易，这就让它可以在多个应用程序中重复使用。

`symfony` wiki 包含了许多插件，你甚至可以加入你自己的。所以现在你知道了如何去做，我们希望你能通过很多有用的扩展来加强 `symfony` 内核！

第 18 章 - 性能

如果你希望你的网站能够吸引很多人，优化网站性能将是在开发阶段的一个主要要素。令人安心的是 symfony 核心开发者总会非常关注性能问题。

通过加速开发带来好处的同时也会带来一些多余的开销，symfony 核心开发者总是会认识到性能的需求。因此，每一个类每一个方法都会仔细的分析并优化到尽可能的快速。基本的开销，可以通过比较使用和不使用 symfony 来显示 "hello, world" 的时间来测量，这个开销很小。因此，这个框架是可扩展的并能在压力测试下表现的很好。最好的证据是，一些高访问量的网站（有百万活跃用户的并有大量消耗服务器资源的 Ajax 交互的）使用 symfony 并且非常满意它的性能。在 wiki 上可以看一下这用 symfony 开发的网站列表 (<http://www.symfony-project.com/trac/wiki/ApplicationsDevelopedWithSymfony>)。

不过，很显然高访问量的站点通常会扩展服务器数量并升级到他们想要的硬件。如果你没有足够的资源做到这一点，或者如果你想确保框架的全部力量都在你的掌握中，你可以使用几个调整来进一步加快你的 symfony 应用程序。本章列出了一些在框架所有层次中和更多高级用户的推荐优化性能方法。它们中的一些在以前的章节中已经提过，但是你会觉得把它们都集中在一起会对你十分有帮助。

调整服务器

一个精心优化的应用程序应该放在一个优化良好的服务器上。你应该了解服务器性能的基础知识，以确保 symfony 运行没有瓶颈。这里有几样东西需要查核，以确保你的服务器不会过于缓慢。

在 php.ini 中设置 magic_quotes_gpc 为 on 会降低应用程序效率，因为这会让 PHP 把请求参数中的所有引用都转义，但 symfony 会在后来系统化的过程中还原它们，这样唯一的后果就是时间上的损失--并会在一些平台上带来引用-转义问题。因此，如果能修改 PHP 配置的话，设置这个参数为 off。

PHP 版本越新越好。PHP5.2 比 PHP5.1 快，PHP5.1 比 PHP5.0 快。所以请升级你的 PHP 来获得最新的性能提升。

在生产服务器上使用 PHP 加速器（例如 APC，XCache 或者 eAccelerator）几乎是必须的，因为它能让 PHP 跑的比平均快 50%。安装其中一个加速器扩展来感受一下 PHP 的真实速度。

此外，在生产服务器上确认关闭了 debug 程序，例如 Xdebug 或者 APD 扩展。

NOTE 你也许会担心 `mod_rewrite` 扩展的开销：这其实是可以忽略的。确实，通过重写规则来读取一张图片比不通过重写规则来读取慢，但是放慢的量级低于执行任何的 `php` 语句。

一些 `symfony` 开发者喜欢使用 `syck`，这是一个 `YAML` 分析器，`PHP` 的一个扩展，它可以替代 `symfony` 的内部分析器。这确实比较快，但 `symfony` 的缓存系统已经让 `YAML` 分析的开销最小化了，所以使用 `syck` 不会给已有生产环境带来什么益处。你小心 `syck` 不是很成熟，使用的时候也许会发生错误。不管怎么说，如果你感兴趣，安装这个扩展(<http://whytheluckystiff.net/syck/>)，`symfony` 会自动使用它的。

TIP 当一台服务器不够用的时候，你可以增加其他服务器来负载均衡。只要 `uploads/` 目录是共享的并且使用了数据库存储用户会话，`symfony` 项目会无缝的嵌入负载均衡架构。

调整模型

在 `symfony` 中，模型层是公认最慢的部分。如果通过基准程序测试发现需要优化模型层，这里有一些可能的改进方法。

优化 Propel 整合

初始化模型层（核心 `Propel` 类）会花一些时间，因为它需要去载入一些类并构造多个对象。无论如何，因为 `symfony` 整合了 `Propel`，所以这些初始化任务只会在动作确实需要模型的时候才会发生，并且会尽量晚发生。`Propel` 类只会在当生成的模型对象自动载入的时候才会被初始化。这就意味不使用模型的页面不会被模型层所累。

如果你的应用程序完全不需要使用模型层，你也能在 `settings.yml` 中设置关闭所有的层并保存在 `sfDatabaseManager` 的初始化值中：

```
all:
  settings:
    use_database: off
```

生成的模型类（在 `lib/model/om/`）已经被优化过了——他们不包含注释，并且他们从自动载入机制中获益。依靠自动载入替代手动包含文件意味着类会在确实需要的时候才会被载入。因此在不需要模型类的情况下，有自动载入机制会节省执行时间，若使用 `include` 语法来实现则不会节省时间。对于注释，他们注解了生成的方法，但是会使模型文件变大——结果会导致轻微的磁盘读取开销。因为生成的方法名是非常清楚的，所以默认注解是关闭的。

这两个加强是针对 `symfony` 的，但你能通过修改 `propel.ini` 文件恢复默认值，如下：


```
propel.builder.addIncludes = true    # 在生成的类中增加 include
                                     # 来替代自动载入机制
propel.builder.addComments = true    # 在生成的类中增加注释
```

限制化合（Hydrate）对象数量

当用 `peer` 类的方法来获得对象的时候，查询通过化合（hydrating）处理（基于查询结果的行来创建和填充对象）。例如，通常可以使用下面语句通过 Propel 获得 `article` 表的所有行：

```
$articles = ArticlePeer::doSelect(new Criteria());
```

`$articles` 变量得到的值是 `Article` 类的对象数组。每一个对象都被创建并初始化了，这需要一些时间。从而得到一个结论：相对于直接访问数据库的查询语句，Propel 查询语句的速度直接取决于它返回结果的数量。这就是说你的模型方法应该经过优化过只返回指定数量的结果。当你不需要从 `Criteria` 获得所有结果的时候，你应该使用 `setLimit()` 和 `setOffset()` 方法来限制返回结果数量。例如，如果你只需要获得第 10-20 行结果，可以如例 18-1 中一样改进一下 `Criteria`。

例 18-1 - 限制 Criteria 返回的结果数量

```
$c = new Criteria();
$c->setOffset(10); // 第一个返回记录的偏移量
$c->setLimit(10);  // 返回记录数量
$articles = ArticlePeer::doSelect($c);
```

这可以通过使用翻页来自动完成。`SfPropelPager` 对象通过自动处理 `offset` 和 Propel 查询语句的 `limit` 来获得对象的特定页的数据。更多这个类的信息可以参考 API 文档。

用 Join 让结果数量最小化

在应用程序开发过程中，你应该关注每个请求会产生多少个数据库查询语句。网页调试工具条显示了每页有多少条查询语句，点击数据库图标会显示出这些 SQL 查询语句。如果看到查询语句的数量增加有异常，就要考虑一下使用 `join` 了。

在解释 `join` 方法之前，让我们回顾一下循环一个对象数组并用 Propel 获得相关类的资料时会发生什么，如例 18-2 所示。这个例子假设你的设计（schema）描述了一个带有 `author` 表外键的 `article` 表。

例 18-2 - 在循环中获得相关类的详细信息

```
// 在动作中
$this->articles = ArticlePeer::doSelect(new Criteria());

// 通过 doSelect()发出的数据库查询
SELECT article.id, article.title, article.author_id, ...
FROM   article

// 在模板中
<ul>
<?php foreach ($articles as $article): ?>
    <li><?php echo $article->getTitle() ?>,
        written by <?php echo $article->getAuthor()->getName() ?></li>
<?php endforeach; ?>
</ul>
```

如果\$articles 数组包含了十个对象，那么当类 Author 的对象调用化合（hydrate）的时候会依次执行十次 getAuthor()方法，如例 18-3 所示。

例 18-3 - 外键获取方法发出了一个数据库查询

```
// 在模板中
$article->getAuthor()

// getAuthor()发出的数据库查询
SELECT author.id, author.name, ...
FROM   author
WHERE  author.id = ?           // ? 是 article.author_id
```

所以例 18-2 中的翻页总共需执行 11 条查询语句： 其中一个当然是用来建立 Article 对象的，其余的 10 条查询语句是用来逐次建立 Author 对象。仅仅显示文章和他们的作者列表却需用多条查询语句来完成。

如果只是使用简单的 SQL 语句，你应该知道如何减少查询语句，只用一条语句来获得 article 表和相关 author 表的内容。这就是 ArticlePeer 类的 doSelectJoinAuthor()方法做的事情。它提供了比单纯 doSelect()调用更复杂的查询语句，但在结果集中增加了列，设置允许 Propel 来融合 Article 对象和相关的 Author 对象。例 18-4 中的代码展示了和例 18-2 同样的效果，但是只需要一条数据库查询语句而不是以前的 11 条语句来处理，这会处理的更快。

例 18-4 - 在一条语句中获得文章详细资料和他们的作者

```
// 在动作中
```

```
$this->articles = ArticlePeer::doSelectJoinAuthor(new Criteria());
```

```
// doSelectJoinAuthor()发出的数据库查询
```

```
SELECT article.id, article.title, article.author_id, ...
       author.id, author.name, ...
FROM   article, author
WHERE  article.author_id = author.id
```

```
// 在模板中(没有改变)
```

```
<ul>
```

```
<?php foreach ($articles as $article): ?>
```

```
  <li><?php echo $article->getTitle() ?>,
```

```
    written by <?php echo $article->getAuthor()->getName() ?></li>
```

```
<?php endforeach; ?>
```

```
</ul>
```

调用 `doSelect()` 或 `doSelectJoinXXX()` 方法对返回的结果来说没有区别；他们都返回了同样的对象数组（如例中的 `Article` 类）。在这之后使用这些对象的外键获取方法才会看出不同。在用 `doSelect()` 的情况下，他发出了查询，一个对象会产生一个结果；而在用 `doSelectJoinXXX()` 的情况下，外部对象已经存在了，不需要用查询语句了，所以处理过程会快些。因此，如果你知道需要用到相关的对象的话，调用 `doSelectJoinXXX()` 方法会减少数据库查询语句的数量——并提高了分页的效率。

`doSelectJoinAuthor()` 方法是根据 `article` 和 `author` 表的关系在调用 `propel-build-model` 时自动产生的。如果在 `article` 表结构中有其他的外键（例如，对于分类表）生成的 `BaseArticlePeer` 类就会有其他的 `Join` 方法，如例 18-5 所示。

例 18-5 - `ArticlePeer` 类可用的 `doSelect` 方法示例

```
// 获得 Article 对象
```

```
doSelect()
```

```
// 获得 Article 对象和 hydrate 相关作者对象
```

```
doSelectJoinAuthor()
```

```
// 获得 Article 对象和 hydrate 相关目录对象
```

```
doSelectJoinCategory()
```

```
// 获得 Article 对象和 hydrate 相关作者和目录对象
```

```
doSelectJoinAuthorAndCategory()
```

```
// 等价于
```

`doSelectJoinAll()`

`Peer` 类也包含了 `doCount()` 的 `Join` 方法。有 `i18n` 关联的类（见第 13 章），提供了 `doSelectWithI18n()` 方法，这个方法和 `Join` 方法很像不过它作用于 `i18n` 对象。要在模型类中发现可用的 `Join` 方法，你应该检查 `lib/model/om/` 中生成的 `peer` 类。如果你没找到查询所需要的 `Join` 方法的话（例如，没有自动生成多对多关系的 `Join` 方法），可以自行建立并扩展你的模型。

TIP 当然，调用 `doSelectJoinXXX()` 会比调用 `doSelect()` 慢些，所以只有之后需要使用化合后（`hydrated`）的外键对象的时候才会提高整体性能。

避免使用临时数组

当时用 `Propel` 时，对象已经被化合（`hydrated`），所以不需要在模板中准备临时数组了。开发者不习惯使用 ORM 通常导致：尽管模板可以直接依靠现有对象的数组来实现，他们还是想要准备一个字符串或者数字数组。例如，想象一下一个模板来显示从数据库中获得的所有文章主题列表的情况。一个不使用 OOP 的开发者通常会写成如例 18-6 这样。

例 18-6 - 已经有数组了，动作中再准备一个数组是没有用处的

```
// 在动作中
$articles = ArticlePeer::doSelect(new Criteria());
$titles = array();
foreach ($articles as $article)
{
    $titles[] = $article->getTitle();
}
$this->titles = $titles;
```

```
// 在模板中
<ul>
<?php foreach ($titles as $title): ?>
    <li><?php echo $title ?></li>
<?php endforeach; ?>
</ul>
```

这段代码的问题是 `hydrating` 已经在调用 `doSelect()` 时完成了（需要花些时间），建立 `$titles` 数组纯属多余，因为你能改写为例 18-7 所示的代码。因此用来建立 `$titles` 数组的时间可以节省下来用来提高应用程序的效率。

例 18-7 - 使用对象数组可以让你不用建立临时数组

```
// 在动作中
$this->articles = ArticlePeer::doSelect(new Criteria());
```

```
// 在模板中
<ul>
<?php foreach ($articles as $article): ?>
    <li><?php echo $article->getTitle() ?></li>
<?php endforeach; ?>
</ul>
```

如果因为一些对象的处理过程中确实需要用临时数组，正确的方法是在你的模型类中建立一个新的方法直接返回这些数组。例如，如果需要一个文章主题数组和每个文章的评论数量的话，动作和模板应如例 18-8 这样。

例 18-8 - 使用自定义的方法替代临时数组

```
// 在动作中
$this->articles = ArticlePeer::getArticleTitlesWithNbComments();
```

```
// 在模板中
<ul>
<?php foreach ($articles as $article): ?>
    <li><?php echo $article[0] ?> (<?php echo $article[1] ?>
comments)</li>
<?php endforeach; ?>
</ul>
```

是否在模型中建立一个快速的处理过程 `getArticleTitlesWithNbComments()` 方法取决于你——例如，通过绕过整个对象关系映射和数据库抽象层来完成。

绕过 ORM

当你确实不需要对象而只需要从一些表中获得一些字段的时候，如同以前的示例中，你能在模型中建立特殊的方法，直接通过 `Creole` 调用数据库完全绕过 ORM 层。例如，返回一个自定义的数组。例 18-9 说明了这个构想。

例 18-9 - 在 `lib/model/ArticlePeer.php` 中直接 `Creole` 访问来优化模型方法

```
class ArticlePeer extends BaseArticlePeer
{
    public static function getArticleTitlesWithNbComments()
    {
```

```

        $connection = Propel::getConnection();
        $query = 'SELECT %s as title, COUNT(%s) AS nb FROM %s LEFT JOIN
%s ON %s = %sGROUP BY %s';
        $query = sprintf($query,
            ArticlePeer::TITLE, CommentPeer::ID,
            ArticlePeer::TABLE_NAME, CommentPeer::TABLE_NAME,
            ArticlePeer::ID, CommentPeer::ARTICLE_ID,
            ArticlePeer::ID
        );
        $statement = $connection->prepareStatement($query);
        $resultset = $statement->executeQuery();
        $results = array();
        while ($resultset->next())
        {
            $results[] = array($resultset->getString('title'), $resultset->getInt('nb'));
        }

        return $results;
    }
}

```

当你开始建立这些方法的时候，你最后可能会为每个动作写一个自定义方法，这会失去层分离带来的好处，而且还失去了数据库独立性。

TIP 如果 Propel 作为模型层不适合你，在手写查询语句前考虑一下使用其他的 ORM。例如，如果想用 **PhpDoctrine ORM** 的话，可以看一下 **sfDoctrine** 插件。还有，你能用其他的数据库抽象层来代替 **Creole**，从而直接访问数据库。在 **PHP 5.1** 里，**PDO** 绑定在 **PHP** 中，而且比 **Creole** 快。

数据库加速

有许多针对数据库的优化技巧可以在使用 **symfony** 的时候用到。本节简单地列出最常用的数据库优化策略，但是良好的理解数据库引擎和管理数据库对于使用模型层会有很好的帮助。

TIP 记住网页调试工具条显示了每个页面执行查询语句的数量，应该监测每一个微调来确认是否增强了性能。

全表查询通常会发生在没有主键的列。要加速这些查询语句，你应该在数据库设计（**schema**）中定义索引。要增加一列索引，给列定义增加 **index: true** 属性，如例 18-10 所示。

例 18-10 - 在 **config/schema.yml** 增加一个单列索引

```
propel :  
  article:  
    id:  
    author_id:  
    title: { type: varchar(100), index: true }
```

你也可以使用另一个选择：用 `index: unique` 语法定义一个唯一索引替代标准的索引。你也可以在 `schema.yml` 中定义多列索引（关于索引语法可以参考第 8 章）。强烈建议考虑这些方法，因为这通常会对一个复杂的查询有很大的帮助。

在 `schema` 中增加索引后，你还需要对数据库作同样的操作，可以在数据库中直接使用 `ADD INDEX` 语句或是调用 `propel -build-all` 命令行（这不只是重建表结构，也会清空所有已存在的数据）。

TIP 索引会加速 `SELECT` 语句的查询效率，但会让 `INSERT`，`UPDATE` 和 `DELETE` 语句变慢。数据库引擎在每个查询语句只使用一个索引并且基于内部启发式的方法来推断使用哪个索引。增加索引有时会对效率带来不利的影响，所以确认你在监测效率是否有所提高。

除另有规定外，在 `symfony` 中每一个请求使用一个数据库连接，每一个连接在请求完成后会被关闭。在 `databases.yml` 文件中设置 `persistent: true`，可以开启持久数据库连接，这样在不同的查询之间数据库连接池会一直保持开启，如例 18-11 所示。

例 18-11 - 在 `config/databases.yml` 中激活永久数据库连接支持

```
prod:  
  propel :  
    class:          sfPropel Database  
    param:  
      persistent:    true  
      dsn:           mysql://login:passwd@localhost/blog
```

这可能会增强数据库总体性能也可能不会，取决于很多因素。在因特网上关于这个主题的文档很多。请确定在修改选项前你测试过应用程序的性能来验证它的结果。

SIDEBAR 针对 MySQL 的技巧

MySQL 配置文件中的许多可以改变数据库性能的设置都放在 `my.cnf` 文件中。确认读过关于此主题的在线文档

(<http://dev.mysql.com/doc/refman/5.0/en/option-files.html>)。

MySQL 提供了一个工具，慢查询记录（slow queries log）。所有 SQL 执行时间超过 `long_query_time` 设置（此设置可以在 `my.cnf` 中更改）的都会被记录在一个文件中，这很难手动统计，但是用一个 `mysql dumpslow` 命令可以方便地列出总结。这是一个很棒的用来查找需要优化的查询语句的工具。

调整视图

按照不同的方法设计和实现视图层，可能会有一些小的速度减少或者提升。这小节讲述的是替代品和它们的优缺点。

使用最快的代码片段

如果没有使用缓存系统，你要注意 `include_component()` 比 `include_partial()` 要慢一些，`include_partial()` 比 PHP 的 `include` 也要慢一些。这是因为 `symfony` 初始化了一个视图来包含一个局部模板和一个 `SfComponent` 类的对象来包含一个组件，包含这些文件会对总体性能带来一些小的影响。

不过，除非在模板中引用了许多局部模板或者组件，否则这对系统开销不是很大。这也许会发生在每次在 `foreach` 中调用 `include_partial()` 辅助函数来做列表或者表格的时候。当你注意到有大量的局部模板或者组件包含非常影响性能时，应该考虑使用缓存（见第 12 章），如果不想用缓存，那只能用简单的 `include` 替代了。

槽（slot）和组件槽（component slot）之间的性能的差别是可以感觉得到的。设置并包含一个槽（slot）的处理时间是可以忽略的——这等于初始化一个变量。但是组件槽（component slot）依靠一个视图配置，他们需要初始化一些对象才能工作。不过，组件槽（component slot）可以在调用模板时被单独缓存，与之相反槽（slot）总是在包含它的模板里被缓存的。

加速路由过程

正如第 9 章解释过的，在模板中每一次调用链接辅助函数都会请求路由系统来把内部 URL 转换为外部 URL。这是通过在 `routing.yml` 文件中查找匹配 URI 和模式来完成的。`symfony` 做起来很简单：它尝试用给予的 URL 去匹配第一个规则，如果不匹配，就接着尝试下一个，然后继续此步骤。由于每次测试都涉及正则表达式，这会相当耗费时间。

有一个简单的方法：使用规则名称代替模块/动作。这会告诉 `symfony` 使用哪一个规则并且路由系统不会花时间去尝试匹配所有前面的规则。

在具体的条件中，假设定义在 `routing.yml` 文件中的路由规则如下：

```
article_by_id:
  url:           /article/:id
  param:         { module: article, action: read }
```


然后用这个方法把输出的超连接替换掉：

```
<?php echo link_to('my article', 'article/read?id='.$article->getId()) ?>
```

你应该用最快实现的方法：

```
<?php echo link_to('my article', '@article_by_id?id='.$article->getId()) ?>
```

注意只有在包含了很多路由链接的页面里，这种差别才会比较明显。

略过模板

通常，一个回应是由一组头信息和内容组成的。但是有些回应不需要内容。例如，一些 Ajax 交互只需要从数据库获得一些数据并提供给 JavaScript 程序用来更新页面的不同部分。对于这些短回应，一套单独的头信息会更适合传递。如第 11 章讨论的，一个动作只能返回一个 JSON 头。例 18-12 重现了第 11 章的一个例子。

例 18-12 - 动作返回一个 JSON 头信息的示例

```
public function executeRefresh()
{
    $output = '<"title", "My basic letter"', ["name", "Mr Brown">';
    $this->getResponse()->setHttpHeader("X-JSON", '('.$output.')');

    return sfView::HEADER_ONLY;
}
```

这跳过了模板和布局，可以立即发出回应。由于它仅包含头，这会更轻巧，并会用较少时间传递给用户。

第 6 章解释了另一个跳过模板的方法，就是直接从动作返回内容文字。这就打破了 MVC 的规则，但这能显著提高动作的响应速度。看例 18-13 的示例。

例 18-13 - 动作直接返回内容的示例

```
public function executeFastAction()
{
    return $this->renderText("<html><body>Hello, World!</body></html>");
}
```

```
}
```

限制默认的辅助函数

标准的辅助函数组(局部模板 `Partial`，缓存 `Cache` 和表单 `Form`)在每次请求的时候都会载入。如果你确认你不使用它们中的一些，从标准列表中移除一个辅助函数组会节省解析辅助函数文件的时间。特别是表单 `Form` 辅助函数组，尽管默认包含了，但是因为他的大小，还是会减慢没有表单的页面的时间。所以在 `settings.yml` 文件的 `standard_helpers` 设置中去掉它也许是个好办法：

```
all:
  .settings:
    standard_helpers: [Partial, Cache]    # Form 被移除
```

相对的，但是你必须在每一个使用 `Form` 辅助函数组的模板中使用 `use_helper('Form')`。

压缩回应

`symfony` 在发送给用户回应前压缩了相应内容。这个功能基于 `PHP` 的 `zlib` 模块。你可以在 `settings.yml` 文件中关闭这个选项来获得一些 `CPU` 时间：

```
all:
  .settings:
    compressed: off
```

要注意获得 `CPU` 时间会损失带宽，所以并不是在所有的配置中改变这个设置都会增加性能。

TIP 如果关闭了 `PHP` 的 `zip` 压缩功能，你可以在服务器层激活它。`Apache` 有他自己的压缩扩展。

调整缓存

第 12 章已经说过如何缓存部分或者全部回应。回应缓存会带来很大的性能提升，这应该是最优先考虑的优化。如果你想要最大化地利用缓存系统，进一步阅读本章节来了解一些你未曾想过的技巧。

选择性的清除部分缓存

在应用程序开发过程中，你必须在一些环境中清除缓存：

- 当建立一个新类：在自动载入目录中增加一个类（在项目的 `lib/` 目录下）是无法让 `symfony` 自动找到它的。你必须清除自动载入配置缓存才

能让 `symfony` 再次浏览所有 `autoload.yml` 文件中定义的目录并引用可自动载入类的位置——这才能包含新建的类。

- 当你在生产环境中修改了配置文件的时候：配置文件只在生产环境的第一次请求的时候会被解析。其余的请求使用的是缓存了的配置文件。所以在生产环境中修改的配置文件（或者任何 `SF_DEBUG` 是 `off` 状态下的环境）只会在清除缓存后才会生效。
- 当你在模板缓存已经激活的环境中修改了模板的时候：在生产环境中，有效的缓存模板总是会替代已经存在的模板而优先得到，所以只有模板缓存被清空后模板的修改才会生效。
- 当用 `sync` 命令行去更新应用程序的时候：这通常包括了前面三个修改。

清除所有缓存会带来一个问题，因为需要生成配置缓存，所以下一个请求会需要花较长的时间来处理。除此之外，未修改过的模板缓存也会被清除掉，这样就会失去之前的请求中缓存带来的速度提升。

这就是说最好的方法是只清除需要重新生成的那部分缓存。使用 `clear-cache` 去定义哪些缓存需要清除，如例 18-14 这样。

例 18-14 - 有选择的清除部分缓存

```
// 只清除 myapp 应用程序缓存
> symfony clear-cache myapp

// 只清除 myapp 应用程序 HTML 缓存
> symfony clear-cache myapp template

// 只清除 myapp 应用程序配置缓存
> symfony clear-cache myapp config
```

你也可以手动删除 `cache/` 目录下的文件，或者有选择的通过 `$cacheManager->remove()` 方法来清除模板缓存，如第 12 章所述。

上面列出的所有这些技巧会使性能的负面影响最小化。

TIP 当升级了 `symfony`，缓存会自动清除（如果在 `settings.yml` 中设置了 `check_symfony_version` 参数为 `true`）。

生成缓存页

当你部署一个新的应用程序到生产服务器上的时候，模板缓存是空的。你必须等待用户访问页面一次让页面生成缓存。在一些关键的部署中，生成页面的系统开销是无法接受的，在第一次请求之前必须生成缓存。

解决办法是在临时工作环境（staging）中（配置文件和生产服务器上很相似）自动浏览应用程序的页面从而生成模板缓存，然后把应用程序和缓存一起放到生产服务器上。

要去自动浏览页面，一个办法是建立一个 shell 脚本调用浏览器（例如 curl）依次访问外部连接。但是有一个更好更快的解决方案：使用 sfBrowser 对象的一个 symfony 批处理，这在第 15 章已经讨论过。这是一个 PHP 写的内部浏览器（使用 sfTestBrowser 来做功能测试）。它访问外部 URL 并返回一个结果，但有趣的是这会像用正常浏览器访问一样生成模板缓存。因为他只是初始化一次 symfony 而且并不通过 HTTP 传送层传递，这个方法会更快一些。

例 18-15 展示了一个批处理脚本的示例，用来在临时工作环境（staging）中生成模板缓存文件。用 php batch/generate_cache.php 开始这个缓存过程。

例 18-15 - 在 batch/generate_cache.php 生成模板缓存

```
<?php

define('SF_ROOT_DIR',    realpath(dirname(__FILE__).'/.'));
define('SF_APP',          'myapp');
define('SF_ENVIRONMENT', 'staging');
define('SF_DEBUG',        false);

require_once(SF_ROOT_DIR.DIRECTORY_SEPARATOR.'apps'.DIRECTORY_SEPARATOR.
SF_APP.DIRECTORY_SEPARATOR.'config'.DIRECTORY_SEPARATOR.'config.php');

// 需要去浏览的 URL 数组
$uris = array(
    '/foo/index',
    '/foo/bar/id/1',
    '/foo/bar/id/2',
    ...
);

$b = new sfBrowser();
foreach ($uris as $uri)
{
    $b->get($uri);
}
```

使用数据库存储作为缓存

symfony 默认使用文件系统作为模板缓存的：HTML 块或者回应对象序列化之后储存在项目的 cache/目录下。symfony 建议另一个方法来储存缓存：一个 SQLite 数据库。这是一个 PHP 原生的，可以有效实现查询的简单文件数据库。

要让 symfony 使用 SQLite 储存代替文件系统储存模板缓存的话，打开 factories.yml 文件并编辑 view_cache：

```
view_cache:
  class: sfSQLiteCache
  param:
    database: %SF_TEMPLATE_CACHE_DIR%/cache.db
```

使用 SQLite 储存作为模板缓存的好处是当缓存元素数量很关键的时候可以更快的做读写操作。如果你的应用程序的缓存压力非常大，模板缓存文件最终会分散在很深的文件结构中；在这时候，用 SQLite 存储会更快。另外，在文件系统存储中清除缓存会有一个从磁盘删除很多文件的动作；这个操作会持续好几秒，这时应用程序是无法访问的。使用 SQLite 存储系统的话，清除缓存过程将只是一个简单的文件删除操作：删除 SQLite 数据库文件。无论缓存元素数据有多大，操作瞬间就会完成。

绕过 symfony

也许最好的加速 symfony 的方法是完全绕过它……这不是一个玩笑。在每次请求中会有一些页面由于没有更改过所以不需要重新由框架来处理。模板缓存已经在加速传递页面了，但是这依旧是依靠 symfony 来处理的。

在第 12 章说过的一些小技巧允许一些页面完全绕过 symfony。第一个是对页面缓存本身请求代理服务器和客户端浏览器做缓存，包含了使用 HTTP 1.1 头文件，所以当这个页面需要的时候他们不需要再次请求了。第二个就是极速缓存（由 sfSuperCachePlugin 插件自动完成），这包含了在 web/目录中储存一份回应的副本和修改重写规则，这样 Apache 会在把请求指向 symfony 前先看缓存版本。

尽管他们只是针对静态页面但上述两种方法都非常有效，它们将为 symfony 分担这些页面的处理，这也会让服务器能全力处理其他请求。

缓存函数调用的结果

如果一个函数不是环境敏感的值也不是随机调用的话，用相同的参数调用它两次应该返回同一个结果。这就是说当第二次调用的时候如果第一次结果已经储存下来的话就可以避免再次调用它。这就是 sfFunctionCache 类做的事情。这个类有一个 call() 方法，可以通过输入一些参数来调用。当被调用的时候，这个方法用所有他的参数建立一个 md5 哈希值作为名字并在缓存目录下找此名字的文件。如果文件找到了，此方法就会返回存在文件中的结果。如果没有，

`sfFunctionCache` 就执行这个函数，并把结果储存在缓存中，并返回值。所以第二次执行例 18-16 会比第一次执行更快。

例 18-16 - 缓存函数结果

```
$function_cache_dir = sfConfig::get('sf_cache_dir').'/function';
$fc = new sfFunctionCache($function_cache_dir);
$result1 = $fc->call('cos', M_PI);
$result2 = $fc->call('preg_replace', '/\s\s+/', ' ', $input);
```

`sfFunctionCache` 的构造函数需要一个绝对路径作为参数（该目录必须在对象初始化之前就存在）。`call()` 方法的第一个参数必须是 PHP 调用名，所以它可以是一个函数名，一个类名字的数组，静态方法名字，对象名字的数组或者公共方法名。你能用任意多的其他参数作为 `call()` 的参数——它们都会被作为调用的参数。

这个对象对很消耗 CPU 的函数特别有用，因为文件 I/O 的开销超过处理一个简单函数的时间。它依赖于 `sfFileCache` 类，这也是模板缓存引擎的一个组件。详情请查阅 API 文档。

CAUTION `clear-cache` 任务只是删除 `cache/` 下文件。如果函数缓存储存在其他地方，通过命令行执行这个命令的时候不会被自动清除。

在服务器上缓存数据

PHP 加速器提供了一些特别的函数在内存中储存数据，因此你可以再次通过它处理请求。问题是他们都用了一些不同的语法，每一个都用自己的方法来处理这个任务。`symfony` 提供了一个叫做 `sfProcessCache` 的类用来抽象化所有的这些不同的工作而不管你用的是什么加速器。参见例 18-17 的语法。

例 18-17 - `sfProcessCache` 方法的语法

```
// 在 Process 缓存中存储数据
sfProcessCache::set($name, $value, $lifetime);

// 获得数据
$value = sfProcessCache::get($name);

// 检查 process 缓存中是否有此数据
$value_exists = sfProcessCache::has($name);

// 清除 process 缓存
sfProcessCache::clear();
```

如果缓存不工作的话 `set()` 方法会返回 `false`。缓存的值可以是任意的（一个字符串，一个数组，一个对象）；`sfProcessCache` 类会处理序列化的过程。如果缓存中没有需求的值，`get` 方法会返回 `null`。

甚至在没有安装加速器的情况下 `sfProcessCache` 类的方法依旧会工作。因此，只要你提供一个返回值，尝试从 `process` 缓存中获得数据总是安全的。例如，例 18-18 显示了如何从 `process` 缓存中获得参数设置的过程。

例 18-18 - 安全的使用 `Process` 缓存

```
if (sfProcessCache::has('myapp_parameters'))
{
    $params = sfProcessCache::get('myapp_parameters');
}
else
{
    $params = retrieve_parameters();
}
```

TIP 如果你想更进一步了解内存缓存，仔细的阅读一下 PHP 的 `memcache` 扩展。它能帮助在负载均衡的应用程序中减少数据库负载，并且 PHP5 提供了它的 OO 接口 (<http://www.php.net/memcache/>)。

屏蔽未使用过的功能

默认的 `symfony` 配置激活了大多数网页应用程序常用的功能。然而，如果你不想要所有的这些，你可以屏蔽他们从而在每个请求中节省初始化的时间。

例如，如果你的应用程序不使用用户会话机制，或者你想手动处理用户会话，你应该将 `factories.yml` 文件的 `storage` 键值 `auto_start` 设置为 `false`，如例 18-19 所示。

例 18-19 - 在 `myapp/config/factories.yml` 中把用户会话关闭

```
all:
  storage:
    class: sfSessionStorage
  param:
    auto_start: false
```

同样的对于数据库（如先前讨论的“调整模型”）和转义输出功能（见第 7 章）。如果应用程序不需要使用他们，屏蔽他们会让系统效率有些许提升，他们的设置在 `settings.yml` 文件中（见例 18-20）。

例 18-20 - 在 `myapp/config/settings.yml` 中屏蔽功能

```
all:
  .settings:
    use_database:      off    # 数据库和模型功能
    escaping_strategy: off    # 输出转义功能
```

关于安全和短暂属性功能（见第 6 章），你可以在 `filters.yml` 文件中屏蔽他们，如例 18-21 所示。

例 18-21 - 在 `myapp/config/filters.yml` 中屏蔽功能

```
rendering: ~
web_debug: ~
security:
  enabled: off

# generally, you will want to insert your own filters here

cache:      ~
common:     ~
flash:
  enabled:  off

execution: ~
```

一些功能只是在开发过程中有用处，所有开发过程中最影响性能的就是 `SF_DEBUG` 模式了。所以你应该在生产环境中屏蔽他们。默认就是这么做的，因为 `symfony` 的生产环境已经优化过性能了。还有就是 `symfony` 日志，这个功能已经在生产环境中默认关闭了。

你也许会想在日志关闭的时候如何在生产环境中得到错误信息，并认为这个问题并不只出现在开发过程中。幸运的是，`symfony` 可以使用 `SfErrorLoggerPlugin` 插件，用来在生产环境后台中记录 404 和 500 错误到数据库中。这比写入文件日志功能更快，因为插件方法只在请求失败时候被调用，当日志机制打开后，不论在什么层次都增加了一个不可忽视的开销。这个插件的安装指南和操作手册网址是 <http://www.symfony-project.com/trac/wiki/SfErrorLoggerPlugin>。

TIP 要确保经常检查服务器错误记录——他们也许有关于 404 和 500 错误的非常有用的信息。

优化你的代码

优化代码本身也可以加速你的应用程序。本节提供一些改进的好意见。

编译核心

载入 10 个文件需要比载入一个大文件花费更多的 I/O 操作，特别是在低速磁盘中。载入一个非常大的文件需要比载入一些小文件占用更多的资源——特别是文件的很大一部分不需要使用 PHP 解析器的时候，例如注释。

因此合并大量的文件并且把它们的注释删除是一个很好的增强性能的方法。symfony 已经做了优化；这就是所谓的核心编译。在第一个请求开始的时候（或者在缓存清空后），一个 symfony 应用程序合并所有的核心框架类（sfActions, sfRequest, sfView 和其他）到一个文件中，删除了注释和多余的空格来优化文件大小，并把它存入缓存中，取名为 config_core_compile.yml.php。每一个接下去的请求只是读取了这个优化过的文件。

如果你的应用程序有类需要加载，尤其是有一个庞大的包含了很多注释的类，把他们加入核心编译文件会很有好处。要这么做的话，只要在应用程序的 config/目录下增加一个 core_compile.yml 文件，列出所有需要增加的类，就如例 18-22 一样。

例 18-22 - 在核心编译文件 myapp/config/core_compile.yml 中增加你的类

```
- %SF_ROOT_DIR%/lib/myClass.class.php
- %SF_ROOT_DIR%/apps/myapp/lib/myToolkit.class.php
- %SF_ROOT_DIR%/plugins/myPlugin/lib/myPluginCore.class.php
...
```

sfOptimizer 插件

symfony 也提供了其他的优化工具，叫做 sfOptimizer。它把许多优化策略应用到了 symfony 和应用程序代码中，用来加速执行效率。

symfony 代码依赖于很多依靠配置文件参数的测试——你的应用程序也是这么做的。例如，如果你看一下 symfony 类，你会经常看到在调用 sfLogger 对象前会有一个带有 sf_logging_enabled 的测试值：

```
if (sfConfig::get('sf_logging_enabled'))
{
    $this->getContext()->getLogger()->info('Been there');
}
```

尽管 sfConfig 注册表已经很好的优化过了，但在每一次处理请求调用它的 get() 方法的次数还是很重要的——这会影响最终的性能。sfOptimizer 的一个优化技巧是用配置常量的值替换它们本身，只要这些常量在运行时不变。例

如，用 `sf_logging_enable` 参数； 当它定义为 `false` 的时候，`sfOptimizer` 会把它转换为：

```
if (0)
{
    $this->getContext()->getLogger()->info('Been there');
}
```

另外，之前的这个例子里，配置值如果是空字符串也会有这样的优化结果。

要用到这个优化，你必须先安装插件 <http://www.symfony-project.com/trac/wiki/sfOptimizerPlugin> 然后调用 `optimize` 任务，制定一个应用程序和环境：

```
> symfony optimize myapp prod
```

如果你想用到其他的优化策略，`sfOptimizer` 插件应该是一个好的开始。

总结

`symfony` 是一个已经优化得非常好的框架了，能用来处理高访问量网站。但如果你确实需要优化你的应用程序性能，调整配置文件（无论是服务器配置，PHP 配置或者是应用程序设置）会带来一些小的加速。你也应该遵循好的策略来写有效的模型方法；因为数据库通常是网页应用程序的瓶颈，这点应该引起你的注意。模板总能用一些小技巧来优化，但最好的加速方法是用缓存。最后，不要犹豫，去看看已经有的插件，这些插件会提供一些创新的技巧来加速你的网页的（如：`sfSuperCache`，`sfOptimizer`）。

第 19 章 管理 symfony 配置文件

现在你对 symfony 应该已经有了相当的认识，但是，你可以进一步研究它的代码，以便了解它的核心设计并挖掘它的潜能。在扩展 symfony 类以符合你的需要之前，你应该仔细研究一下配置文件。因为 symfony 内置了许多特性，只需要更改配置就可以激活这些特性，也就是说，你无需重载 symfony 的类就可以调整 symfony 的核心行为。本章就带你深入研究这些配置文件以了解这些配置的强大威力。

symfony 配置参数

对于 myapp 应用程序来说，它的主要 symfony 配置都在 myapp/config/settings.yml 文件中。在前面章节中你已经看到许多配置参数的作用，这里我们再看一下。

在第 5 章中我们已经解释过，这个文件是与环境有关的，也就是说每个参数可以为每个环境取一个不同的值。记住，通过 sfConfig 类，用 PHP 代码可以访问这个文件中定义的每一个参数。这些参数都以 sf_ 开头。例如，如果你想得到 cache 参数的值，你只需调用 sfConfig::get(sf_cache) 即可。

默认模块和动作

当一个程序规则没有定义模块或动作的参数时，settings.yml 文件中的值就会用默认值代替：

- default_module: 默认 module 请求参数，是 default 模块的默认值。
- default_action: 默认 action 请求参数，是 index 动作的默认值。

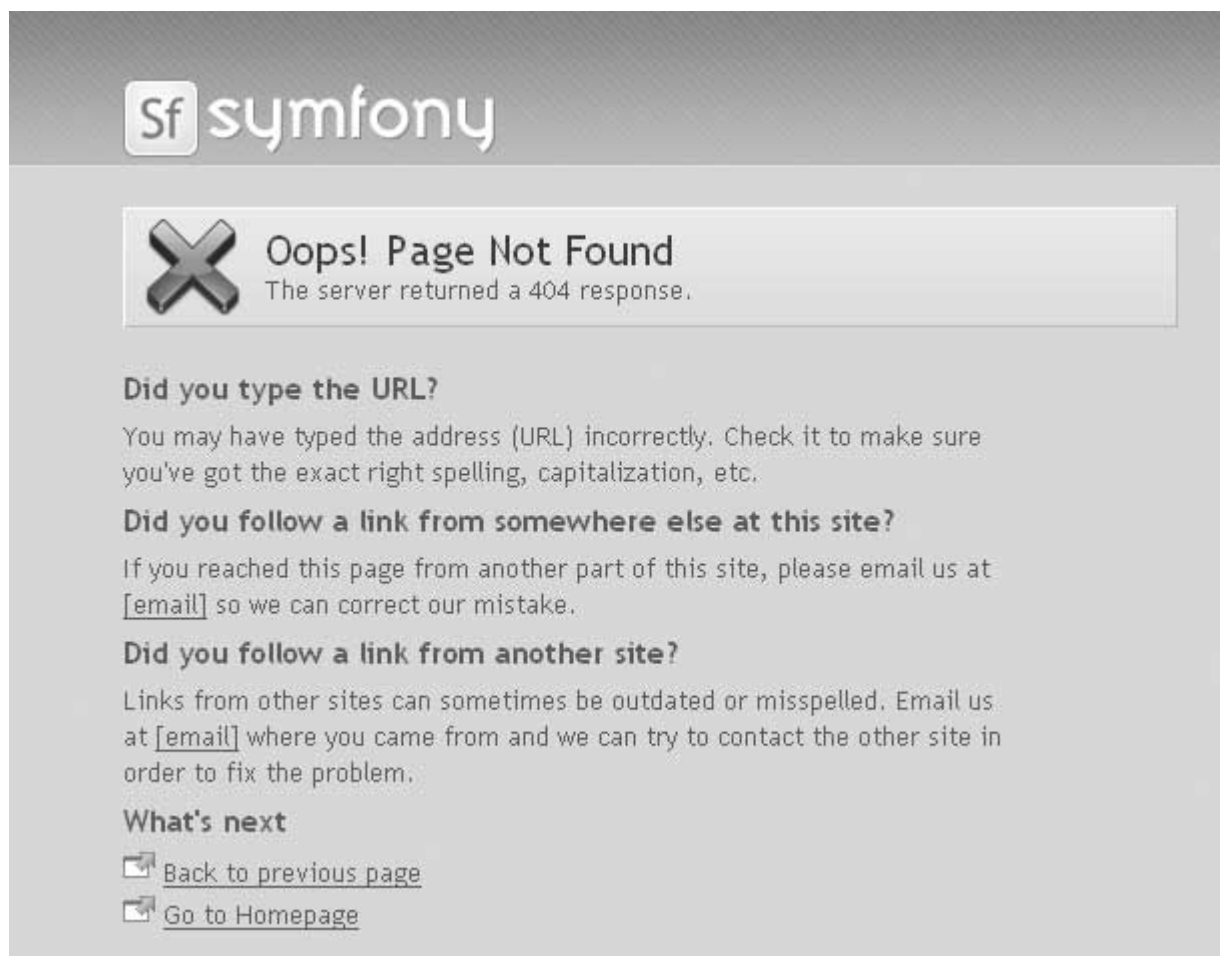
symfony 为一些特殊情形提供了默认页。在 \$sf_symfony_data_dir/modules/default/ 目录下存放着 default 模块，在程序出错的情况下，symfony 就执行 default 模块的一个动作。settings.yml 文件则根据错误的不同，定义了可以执行的动作：

- error_404_module 和 error_404_action: 当用户输入的 URL 和任何路由都不匹配时或当一个 sfError404Exception 产生时，就调用这个动作。默认值是 default/error404。
- login_module 和 login_action: 当一个未经授权的用户试图访问由 security.yml 中定义为 secure 的页面时(请参考第 6 章的解释)，这个动作将被调用。默认值是 default/login。
- secure_module 和 secure_action: 当一个用户不具备某个动作所需的信任证书时，这个动作将被调用。默认值是 default/secure。

- `module_disabled_module` 和 `module_disabled_action`: 当一个用户请求一个被 `module.yml` 定义为 `disabled` 的模块时, 这个动作将被调用。默认值是 `default/disabled`。
- `unavailable_module` 和 `unavailable_action`: 当一个用户从一个被禁用的应用程序中请求一个页面时, 这个动作将被调用。默认值是 `default/unavailable`。要禁用一个应用程序, 在 `settings.yml` 中设置 `available` 参数为 `off` 即可。

在将一个应用程序部署到生产环境之前, 你需要定制这些动作, 因为 `default` 模块的模板页面中都含有 `symfony` 的标识。

图 19-1 是页面之一的 404 错误页面的截屏



你可以通过两种方法重载这些默认页:

- 对于 `settings.yml` 文件中定义的所有动作(`index`, `error404`, `login`, `secure`, `disabled` 和 `unavailable`)和所有相应的模板(`indexSuccess.php`, `error404Success.php`, `loginSuccess.php`, `secureSuccess.php`, `disabledSuccess.php` 和 `unavailableSuccess.php`), 你可以在应用程序的 `modules/` 目录中创建自己的 `default` 模块。

- 你可以将 `settings.yml` 文件中的默认模块和动作参数设置为你的应用程序的页面。

另外，还有两个页面也包括 `symfony` 标识，所以在实际部署之前也需要对它们重新定制。这两个页面不在默认模块中，因为只有当 `symfony` 不能正常运行时，才会被调用。你可以在 `$sf_symfony_data_dir/web/errors` 目录中看到这些页面：

- `error500.php`: 当生产环境中出现内部服务器错误时，该页面被调用。在 `SF_DEBUG` 设置为 `true` 并且出现一个错误时，`symfony` 会显示所有的执行栈和精确的错误信息(请参看第 16 章的介绍)。
- `unavailable.php`: 当用户请求一个缓存已被清除的页面时(也就是在调用 `symfony` 的 `clear-cache` 任务和该任务结束之间请求时)，该页面被调用。对于一个有着很大缓存的系统，清除缓存可能需要几秒。`symfony` 在部分清除缓存的情况下不能正常工作，所以在清除缓存完成之前接收到的请求都会被重定向到这个页面。如果用 `symfony` 的 `disable` 命令禁用了一个应用程序，也会用到 `unavailable.php`。

要定制这些页面，只需在应用程序的 `web/errors` 目录中创建 `error500.php` 和 `unavailable.php` 即可。`symfony` 将用这些页面代替默认页面。

NOTE 如果要将请求转向 `unavailable.php` 页面，你应该将 `settings.yml` 中的 `check_lock` 设置为 `on`。该参数的默认值是 `off`，因为它会为每一个请求增加一些极为轻微的负担。

激活可选特性

设置或取消 `settings.yml` 文件中的一些参数就可以设置或取消某些框架特性。取消不会用到的特性可以提高系统性能，所以在部署应用程序之前，你应该确保已经检查了例 19-1 中列出的参数值。

例 19-1 - 可以通过 `settings.yml` 设置的可选特性

参数	描述	默认值
<code>use_database</code>	激活数据库管理。如果设置为 <code>off</code> ，你将不能使用数据。	<code>on</code>
<code>use_security</code>	激活安全特性（包括安全的动作和信任证书，请参看第 6 章）。只有在这个特性被激活的条件下，才能使用默认的安全过滤器（ <code>SfBasicSecurityFilter</code> ）。	<code>on</code>
<code>use_flash</code>	激活 <code>flash</code> 特性（请参看第 6 章）。如果你在动作中从来都不用 <code>flash()</code> ，就将这个参数设为 <code>off</code> 。只有在这个特性被激活的条件下，才能使用 <code>flash</code> 过滤器（ <code>SfBasicFlashFilter</code> ）。	<code>on</code>

参数	描述	默认值
i18n	激活接口翻译特性（请参看第 6 章）。如果你的应用程序是多种语言的，将它设为 on。	off
logging_enabled	允许 symfony 事件日志。如果你想忽略 logging.yml 的配置并且将 symfony 日志完全关闭，就将它设为 off。	on
escaping_strategy	激活输出转义特性并定义输出转义的策略（请参看第 7 章）。如果在你的模板中不会用到 \$sf_data 容器的话，就将它设为 off。	bc
cache	激活模板缓存（请参看第 12 章）。如果你的模块中包括 cache.yml 文件，就将它设为 on。只有在 on 的状态下，才能用缓存过滤器（sfCacheFilter）。	开发时为 off，实际生产环境为 on
web_debug	允许使用 web 调试工具栏，以便于调试(请参看第 16 章)。如果你想在每一页中都显示工具栏， 就将它设为 on。只有在 on 的状态下，才能用 web 调试过滤器（sfWebDebugFilter）。	开发时为 on，实际生产环境为 off
check_symfony_version	允许每次请求都检查 symfony 的版本。如果为 on，框架升级后能自动清除缓存；如果为 off，升级后总是需要你清除缓存。	off
check_lock	允许 clear-cache 和 disable 任务触发应用程序去锁定系统（请参看前面部分的内容）。如果相让被禁用的应用程序的所有请求都被转向至 \$sf_symfony_data_dir/web/errors/unavailable.php 页面的话，就将它设为 on。	off
compressed	激活 PHP 应答压缩特性。如果你想通过 PHP 压缩处理器压缩出现的 HTML 页时，就将它设为 on。	off
use_process_cache	激活基于 PHP 加速器的 symfony 优化特性。如果你安装了诸如 APC，XCache 或 eAccelerator 之类的加速器，symfony 可以利用这些加速器的能力在请求之间将对象和配置保存在内存中。开发过程中或者你不想使用 PHP 加速器优化时，将它设为 off。即使你没有安装任何加速器，设为 on 也不会影响系统性能。	on

功能特性配置

设置 `setting.yml` 中的某些参数可以改变内置特性的行为，比如表单验证、缓存和第三方模块等。

设置输出转义参数

设置输出转义参数可以控制模板中的变量可存取的方法（请参看第 7 章）。`setting.yml` 中包括两个与这个特性有关的参数：

- `escaping_strategy` 参数可取 `bc`, `both`, `on`, 或 `off` 等值。
- `escaping_method` 参数可取 `ESC_RAW`, `ESC_ENTITIES`, `ESC_JS`, 或 `ESC_JS_NO_ENTITIES`。

设置路由参数

`settings.yml` 中有两个路由参数：

- `suffi x` 参数为生成的 URL 设置默认的后缀。`suffi x` 的默认值是句号 (`.`)，对应于没有后缀。如果设置为 `.html`，则所有生成的 URL 就都成了静态页面。
- `no_script_name` 参数可以让前端控制器的名字出现在生成的 URL 中。除非你将前端控制器存放在不同的目录中并且修改了默认的 URL 重写规则。在项目的主应用程序的生产环境中一般设置为 `on`，其他情况下则设置为 `off`。

设置表单验证参数

表单验证参数控制由 `Validation` 辅助函数产生的错误信息的输出方式(请参看第 10 章)。这些错误包括在 `<div>` 标记中，它们将 `validation_error_class` 参数作为 `classs` 属性，将 `validation_error_id_prefix` 参数作为 `id` 属性，默认值是 `form_error` 和 `error_for_`，因此，对于一个名为 `foobar` 的输入来说，调用 `form_error()` 而输出的属性将会是 `class=form_error`，`id=error_for_foobar`。

`validation_error_prefix` 和 `validation_error_suffi x` 参数分别确定了每条错误信息的前缀字符和后缀字符。你可以根据需要定制你自己的错误信息。

设置缓存参数

缓存参数大多数在 `cache.yml` 中定义，只有 `cache` 和 `etag` 在 `settings.yml` 中定义，前者打开模板缓存机制，后者允许 ETAG 在服务器端操作(请参看第 15 章)。

设置日志参数

`settings.yml` 中包括以下两个日志参数(请参看第 16 章)：

- `error_reporting` 指明在 php 日志中记录哪些事件。通常在生产环境中设为 341，也即记录 `E_PARSE`，`E_COMPILE_ERROR`，`E_ERROR`，`E_CORE_ERROR` 和 `E_USER_ERROR` 产生的事件，而在开发环境中设为 4095，即只记录 `E_ALL` 和 `E_STRICT` 产生的事件。
- `web_debug` 激活 web 调试工具条。仅在开发和测试环境中才需要设置这个参数。

设置指向资源 (Assets) 的路径

`settings.yml` 中还包括指向资源的路径的参数。如果你想用与 `symfony` 设置的路径不同的路径值，你可以改变以下路径参数的设置：

- `rich_text_js_dir`：富文本编辑器 Javascript 文件的存放路径，默认值为 `js/tiny_mce`
- `prototype_web_dir`：Prototype 库的路径，默认值为 `/sf/prototype`
- `admin_web_dir`：管理生成器所需文件的存放路径
- `web_debug_web_dir`：网页调试工具条所需文件的存放路径
- `calendar_web_dir`：javascript 日历所需文件的存放路径

配置默认辅助函数参数

`standard_helpers` 参数指明每个模板都会导入的默认辅助函数(请参看第 7 章)，默认值是 `Partial`，`Cache` 和 `Form` 辅助函数。如果你要在某个应用程序的所有模板中都导入一个辅助函数组，就将这个组的名字加入到 `standard_helpers` 中，这样就可以避免在每个模板中都用 `use_helper()` 去声明。

配置被激活模块参数

`enabled_modules` 参数指明从插件或 `symfony` 核心激活的模块。即使某个插件绑定了一个模块，如果没有预先在 `enabled_modules` 中声明，用户也不能请求这个模块。默认情况下，提供默认的 `symfony` 页(成功页，未发现页等)的 `default` 模块是唯一被激活的模块。

设置字符集

字符集是应用程序的一项常用设置，因为框架的许多部分都会用到字符集，如模板，输出转义和辅助函数等。`Charset` 参数用于定义字符集，默认值是 `utf-8`。

其他配置

`settings.yml` 文件中还包括一些用于控制 `symfony` 核心行为的参数。例 19-1 列出了这些参数。

例 19-1 `myapp/config/settings.yml` 中的其他配置参数


```
# 去掉 core_compile.yml 里定义的框架核心类的注释
strip_comments: on
# 当类被调用但还没载入时调用的函数
# 参数值为可调用的函数组成的数组。由框架桥调用。
autoloading_functions: ~
# 会话超时参数，单位是秒
timeout: 1800
# 在动作抛出异常前可转发的最大次数
max_forwards: 5
# 全局常量
path_info_array: SERVER
path_info_key: PATH_INFO
url_format: PATH
```

SIDEBAR 加入你的应用程序的配置参数

settings.yml 定义了一个应用程序的 symfony 参数配置。在第五章中我们说过，如果你想加入新的参数，最合适的地方是 myapp/config/app.yml。这个文件也是与环境有关的，其中定义的参数可以通过带 app_前缀的 sfConfig 类访问。

```
all:
  creditcards:
    fake: off # app_creditcards_fake
    visa: on # app_creditcards_visa
    americanexpress: on # app_creditcards_americanexpress
```

你还可以在项目的配置目录中写一个 app.yml 文件，用于定制项目的配置参数。这个文件也受配置级联的影响，所以在应用程序的 app.yml 定义的参数会覆盖项目级的 app.yml 中定义的参数。

扩展自动载入特性

在第二章中我们介绍了自动载入特性，如果类在某些特定的目录中，该特性可以让你无需在你的代码中用 require 语句。也就是说，框架会在你需要的适当时候替你完成这些工作。

autoload.yml 文件中列出了所有可以自动载入类的路径。这个配置文件第一次被处理时，symfony 就会编译该文件中引用的所有路径。一旦在这些目录中发现有.php 文件，那么这个文件的路径和类名就会被加进一个存放自动载入类的内部列表。这个列表放在缓存的 config/config_autoload.yml.php 文件中。系统运行的时候，如果需要用到一个类，symfony 就会在这个列表中自动查找类的路径，并将找到的.php 文件包括进去。

所有包含类和/或接口的.php 文件都是自动载入的对象。

默认情况下，在项目的以下目录中的类都能被自动加载：

- myproject/lib/
- myproject/lib/model
- myproject/apps/myapp/lib/
- myproject/apps/myapp/modules/mymodule/lib

在默认的应用程序配置目录中，并没有 `autoload.yml` 文件。如果你想修改框架参数——比如，你希望你的文件结构中某些目录中的类也能被自动载入——你可以创建一个空的 `autoload.yml` 文件，并重载 `$sf_symfony_data_dir/config/autoload.yml` 或增加自己的定义。

`autoload.yml` 文件必须以 `autoload` 关键字开头，然后列出 `symfony` 要找的所有类的位置。每个位置需要一个标签，这个标签可以让你重载 `symfony` 的定义。每个位置还需要提供一个名字(它以注释形式出现在 `config/autoload.yml.php` 中)和一个绝对路径。然后还需要用 `recursive` 参数确定是否需要递归，也就是说 `symfony` 是否需要在所有的子目录中查找 `.php` 文件。最后用 `exclude` 参数排除无需查找的子目录。19-2 列出了默认的位置和定义方法。

例 19-2 `$sf_symfony_data_dir/config/autoload.yml` 中定义的默认的自动载入配置

`autoload:`

```
# symfony core
symfony:
  name:          symfony
  path:          %SF_SYMFONY_LIB_DIR%
  recursive:    on
  exclude:      [vendor]

propel:
  name:          propel
  path:          %SF_SYMFONY_LIB_DIR%/vendor/propel
  recursive:    on

creole:
  name:          creole
  path:          %SF_SYMFONY_LIB_DIR%/vendor/creole
  recursive:    on

propel_addon:
  name:          propel_addon
  files:
```

```
Propel :
%SF_SYMFONY_LIB_DIR%/addon/propel/sfPropelAutoload.php
```

```
# plugins
plugins_lib:
  name:          plugins lib
  path:          %SF_PLUGINS_DIR%/*/*lib
  recursive:     on

plugins_module_lib:
  name:          plugins module lib
  path:          %SF_PLUGINS_DIR%/*/*modules/*/*lib
  prefix:        2
  recursive:     on

# project
project:
  name:          project
  path:          %SF_LIB_DIR%
  recursive:     on
  exclude:       [model, symfony]

project_model:
  name:          project model
  path:          %SF_MODEL_LIB_DIR%
  recursive:     on

# application
application:
  name:          application
  path:          %SF_APP_LIB_DIR%
  recursive:     on

modules:
  name:          module
  path:          %SF_APP_DIR%/*/*modules/*/*lib
  prefix:        1
  recursive:     on
```

路径中可以包括通配符，还可以使用 `constants.php` 文件中的路径参数（见下节）。如果在配置文件中用到这些参数，它们必须用大写字母并以%开头和结尾。

编辑你自己的 `autoload.yml` 就可以增加新的位置，但你可能还想扩展这个机制，并把你自己的自动载入处理器添加到 `symfony` 的处理器中。这可以通过在

settings.yml 文件中的 autoloading_functions 参数来实现。这个配置需要一个以可调用方法为元素的数组作为参数，示例如下：

```
.settings:
  autoloading_functions:
    - [myToolkit, autoload]
```

当 symfony 遇到一个新的类，它首先用自己的自动载入机制（和 autoload.yml 中定义的位置）。如果没有找到类的定义，它会用 settings.yml 中的其他自动载入函数继续寻找，一直到类被找到为止。所以你可以任意添加你想要的自动载入功能，例如在第 17 章中介绍过的桥接其他框架组件。

定制文件结构

每当框架利用一条路径搜索核心类、模板、插件或配置文件等的时候，它用的都是路径变量而不是实际路径。通过改变这些变量，你可以遵照客户的需要完全改变 symfony 项目的目录结构。

CAUTION 你可以定制 symfony 项目的目录结构，但这并不是很有必要。symfony 框架的一个优点就是任何一个开发者只要看到默认的目录结构，就会根据习惯知道项目的结构。在你改变项目结构之前应该考虑这个因素。

基本的文件结构

在应用程序被启动时，\$sf_symfony_data_dir/config/constants.php 文件中定义的路径变量就被载入。这些变量存放在 sfConfig 对象中，所以很容易被重载。19-3 列出了路径变量和对应的路径。

例 19-3 \$sf_symfony_data_dir/config/constants.php 中定义的文件结构变量的默认值

```
sf_root_dir          # myproject/
                      # apps/
sf_app_dir           # myapp/
sf_app_config_dir    # config/
sf_app_i18n_dir      # i18n/
sf_app_lib_dir       # lib/
sf_app_module_dir    # modules/
sf_app_template_dir  # templates/
sf_bin_dir           # batch/
                      # cache/
sf_base_cache_dir    # myapp/
sf_cache_dir         # prod/
sf_template_cache_dir # templates/
sf_i18n_cache_dir    # i18n/
```

```

sf_config_cache_dir # config/
sf_test_cache_dir # test/
sf_module_cache_dir # modules/
sf_config_dir # config/
sf_data_dir # data/
sf_doc_dir # doc/
sf_lib_dir # lib/
sf_model_lib_dir # model/
sf_log_dir # log/
sf_test_dir # test/
sf_plugins_dir # plugins/
sf_web_dir # web/
sf_upload_dir # uploads/

```

以`_dir`结尾的参数定义了这些关键目录的路径。为了以后可以改变路径变量值，用路径变量而不要用真实（无论是绝对或相对）文件路径。例如，如果想将一个文件移动到项目的`uploads/`目录中，你应该用`SfConfig::get('sf_upload_dir')`来取得路径，而不应该使用`SF_ROOT_DIR/web/uploads/`。

当运行系统确定了模块名（`$module_name`）时，模块的目录结构在运行时被定义。这是根据`constants.php`文件中定义的路径名自动建立的，例 19-4 列出了`constants.php`的内容。

例 19-4 默认模块文件结构变量

```

sf_app_module_dir # modules/
module_name # mymodule/
sf_app_module_action_dir_name # actions/
sf_app_module_template_dir_name # templates/
sf_app_module_lib_dir_name # lib/
sf_app_module_view_dir_name # views/
sf_app_module_validate_dir_name # validate/
sf_app_module_config_dir_name # config/
sf_app_module_i18n_dir_name # i18n/

```

根据这个文件，当前模块的`validate/`目录的路径在运行时自动生成成为：

```

SfConfig::get('sf_app_module_dir' . '/' . module_name . '/' . SfConfig::get('sf_app_module_validate_dir_name'))

```

定制文件结构

如果用户的应用程序已经有了一个目录结构或者不想采用 symfony 的目录结构，你可以修改默认的项目文件结构。只要用 sfConfig 重载 sf_XXX_dir 和 sf_XXX_dir_name 变量，就可以获得一个与默认结构完全不同的文件结构。最适合修改的地方是应用程序的 config.php 文件。

CAUTION 要用应用程序的 config.php 而不是项目的 config.php 去重载 sf_XXX_dir 和 sf_XXX_dir_name。因为项目的 config/config.php 文件很早就被载入，而此时 sfConfig 类尚不存在，而且 constants.php 文件也还未载入。

例如，如果想让所有的应用程序能共享模板布局的共用目录，将下面这行代码加入到 myapp/config/config.php 中以重载 sf_app_template_dir 参数：

```
sfConfig::set('sf_app_template_dir',  
sfConfig::get('sf_root_dir').DIRECTORY_SEPARATOR.'templates');
```

注意，应用程序的 config.php 文件是非空的，所以要将文件结构定义加在文件的最后。

修改项目的 Web 根目录

在前端控制器中用常量 SF_ROOT_DIR 定义了项目根目录，而 constants.php 中的所有路径都和这个根目录有关。通常根目录是 web/ 目录的上一级目录，但是你可以用不同的目录结构。假如你的主要目录结构由例 19-5 的两个目录组成，就象在一个共享主机上部署项目一样，包括一个公用目录和一个私有目录。

例 19-5 共享主机的定制目录结构示例

```
symfony/    # 私有区域  
  apps/  
  batch/  
  cache/  
  ...  
www/        # 公用区域  
  images/  
  css/  
  js/  
  index.php
```

在这个例子中，symfony/ 是根目录，所以在前端控制器 index.php 中要如下定义 SF_ROOT_DIR：

```
define('SF_ROOT_DIR', dirname(__FILE__).'../symfony');
```

另外，因为公共目录在 `www/` 下，而不是在通常的 `web/` 下，所以要在 `config.php` 中重新定义两个文件路径：

```
sfConfig::add(array(
    'sf_web_dir'      => SF_ROOT_DIR.DIRECTORY_SEPARATOR.'www' ,
    'sf_upload_dir'   =>
SF_ROOT_DIR.DIRECTORY_SEPARATOR.'www'.DIRECTORY_SEPARATOR.sfConfig::get('sf_upload_dir_name' ),
));
```

连接 symfony 库

在 `config.php` 文件中定义了框架文件路径，如例 19-6 所示。

例 19-6 框架文件路径，`myproject/config/config.php`

```
<?php

// symfony 目录
$sf_symfony_lib_dir = '/path/to/symfony/lib';
$sf_symfony_data_dir = '/path/to/symfony/data';
```

当你从命令行调用 `symfony init-project` 时，`symfony` 初始化这些路径，并指向用于创建项目的 `symfony` 安装目录。命令行和 MVC 架构都会用到这些路径。

这也就是说如果你改变指向框架文件的路径，你就可以切换到另一个 `symfony` 的安装目录去。

这些路径应该是绝对路径，但是利用 `dirname(FILE)`，你可以指向项目结构内部的文件并且为项目安装保留选定的目录的独立性。例如，许多项目选取 `symfony` 的 `lib/` 目录作为项目的 `lib/symfony/` 目录的符号链接，对 `symfony` 的 `data/` 目录也同样如此：

```
myproject/
  lib/
    symfony/ => /path/to/symfony/lib
  data/
    symfony/ => /path/to/symfony/data
```

在这种情况下，只需在 `config.php` 中如下定义 `symfony` 目录即可：

```
$sf_symfony_lib_dir = dirname(__FILE__).'../lib/symfony';  
$sf_symfony_data_dir = dirname(__FILE__).'../data/symfony';
```

如果你在项目的 lib/vendor 目录中决定包含 symfony 文件作为一个 svn:externals，也可以采用同样的方法：

```
myproject/  
  lib/  
    vendor/  
      svn:externals symfony http://svn.symfony-project.com/trunk/
```

这样，config.php 应该如下所示：

```
$sf_symfony_lib_dir = dirname(__FILE__).'../lib/vendor/symfony/lib';  
$sf_symfony_data_dir =  
dirname(__FILE__).'../lib/vendor/symfony/data';
```

TIP 有时，在运行同一个应用程序的不同服务器中，symfony 库有不同的路径。要能有效工作，一种方法是将 config.php 文件从同步定义文件 (rsync_exclude.txt) 中删除掉，另一种方法是在开发环境和生产环境中保留同样的 config.php 路径，但是根据不同的服务器改变路径的符号链接。

理解配置处理器

每个配置文件都有一个处理器。配置处理器的任务就是管理配置的级联，并且在配置文件和在运行时优化的 PHP 可执行代码之间进行转换。

默认的配置处理器

\$sf_symfony_data_dir/config/config_handlers.yml 中存放默认的配置处理器。该文件根据文件路径来连接配置文件的处理器。例 19-7 显示了这个文件的摘要。

例 19-7 \$sf_symfony_data_dir/config/config_handler.yml 文件摘要

```
config/settings.yml :  
  class: sfDefineEnvironmentConfigHandler  
  param:  
    prefix: sf_
```

```
config/app.yml :  
  class: sfDefineEnvironmentConfigHandler  
  param:
```



```
    prefix: app_

config/filters.yml:
    class:      sfFilterConfigHandler

modules/*/config/module.yml:
    class:      sfDefineEnvironmentConfigHandler
    param:
        prefix: mod_
        module: yes
```

handlers.yml 用一个带通配符的文件路径来标识每个配置文件，在 class 关键字下指明处理器类。

在程序中，由 sfDefineEnvironmentConfigHandler 处理的配置文件参数，可以直接由 sfConfig 类和包含一个 prefix 值的 param 键来直接访问。

你可以增加和修改用于处理每一个配置文件的处理器，例如，可以用 INI 或 XML 文件，而不用 YML 文件。

NOTE config_handlers.yml 文件的配置处理器是 sfRootConfigHandler，而且它是不可更改的。

如果你想改变编译配置的方法，可以在 config/目录下创建一个空 config_handlers.yml 文件，然后将你自己写的类替换到 class 行里即可。

加入你自己的处理器

用一个处理器来处理一个配置文件有以下两个重要的好处：

- 配置文件转化为可执行的 PHP 代码后存放在缓存里。也就是说，在生产环境中配置仅被编译一次，因而性能可以达到最优。
- 配置文件可以在项目或应用程序级别上定义，并且最终将从不同级别的定义中级联后得到参数。因此，你可以在项目一级定义参数，然后在应用程序的级别上重新设置这些参数。

如果你想写你自己的配置处理器，在 \$sf_symfony_lib_dir/config 目录中有一个用于框架的结构示例可供参考。

我们假设你的应用程序中包含一个 myMapAPI 类，用于为第三方的发送地图 web 服务提供接口。这个类需要用一个 URL 和一个用户名去初始化，如例 19-8 所示。

例 19-8 myMapAPI 类的初始化示例

```
$mapApi = new myMapAPI();  
$mapApi->setUrl($url);  
$mapApi->setUser($user);
```

在应用程序的 `config` 目录下有一个定制的配置文件 `map.yml`，你也许想将这两个参数存放在该文件中，那么文件将包含以下内容：

```
api :  
  url: map.api.example.com  
  user: foobar
```

为了将这些参数设置转化为等同于例 19-8 的代码，你必须构造一个配置处理器。每个配置处理器必须继承 `SfConfigHandler` 并且实现一个 `execute()` 方法，这个方法的参数是一个元素为配置文件路径的数组，并返回写入缓存文件的数据。YAML 文件的处理器必须继承 `SfYamlConfigHandler` 类，该类提供了编译 YAML 的附加的功能。对于 `map.yml` 文件，典型的配置处理器应该如下例 19-9 所示：

例 19-9 定制配置处理器，文件路径是
`myapp/lib/myMapConfigHandler.class.php`

```
<?php  
  
class myMapConfigHandler extends SfYamlConfigHandler  
{  
    public function execute($configFiles)  
    {  
        $this->initialize();  
  
        // 编译 yaml  
        $config = $this->parseYamls($configFiles);  
  
        $data = "<?php\n";  
        $data. = "\$mapApi = new myMapAPI();\n";  
  
        if (isset($config['api']['url'])  
        {  
            $data. = sprintf("\$mapApi->setUrl('%s');\n",  
$config['api']['url']);  
        }  
  
        if (isset($config['api']['user'])  
        {
```

```

        $data. = sprintf("\$mapApi->setUser('%s');\n",
$config['api']['user']);
    }

    return $data;
}
}

```

symfony 传递\$configFiles 数组给 execute()方法，该数组包含一个指向 config/目录中的所有 map.yml 文件的路径。parseYaml s()方法则处理配置级联。

为了让这个新的处理器能与 map.yml 协同工作，你必须创建一个包括如下内容的 config_handlers.yml 配置文件：

```

config/map.yml :
class: myMapConfigHandler

```

NOTE 这个类要么被自动载入（如本例所示），要么在一个文件中定义，该文件的路径记录在 param 关键字的 file 参数里。

当你需要在应用程序中基于 map.yml 文件并由 myMapConfigHandler 处理器生成代码的时候。执行下面的代码：

```

include(sfConfigCache::getInstance()-
>checkConfig(sfConfig::get('sf_app_config_dir_name').'/map.yml'));

```

调用 checkConfig()方法时，如果缓存里没有 map.yml.php，或者如果 map.yml 比缓存中的更新，那么，symfony 在配置目录中寻找存在的 map.yml 文件，并用 config_handlers.yml 中指明的处理器来处理这些文件。

TIP 如果要在一个 YAML 配置文件中处理环境变量，你可以继承 sfDefineEnvironmentConfigHandler 类，而不是继承 sfYamlConfigHandler。在调用 parseYaml ()方法遍历了配置以后，你需要调用 mergeEnvironment()方法。你可以在一条代码行里完成所有任务：\$config = \$this->mergeEnvironment(\$this->parseYaml s (\$configFiles));。

-

SIDEBAR 使用现成的配置处理器

如果你仅想让用户通过 sfConfig 从代码中遍历值，你可以使用 sfDefineEnvironmentConfigHandler 配置处理器。例如，要实现

`sfConfig::get('map_url')`和`sfConfig::get('map_user')`，你可以定义如下处理器：

```
config/map.yml :
  class: sfDefineEnvironmentConfigHandler
  param:
    prefix: map_
```

注意不要使用已经被别的处理器使用过的前缀，现有的前缀包括 `sf_`，`app_`和`mod_`。

控制 PHP 参数

为了让 PHP 环境和敏捷开发的原则及实践相配合，`symfony` 检测并修改了 `php.ini` 中的有些参数。`php.yml` 文件就是为这个目的而出现的。

例 19-10 是默认的 `php.yml` 文件的内容，位于`$sf_symfony_data_dir/config`

set:

```
magic_quotes_runtime:    off
log_errors:              on
arg_separator.output:    |
&#x2D;
```

check:

```
zend.ze1_compatibility_mode: off
```

warn:

```
magic_quotes_gpc:        off
register_globals:         off
session.auto_start:      off
```

这个文件的主要目的就是检测 `php` 配置是否和你的应用程序兼容，它也可用于检测你的开发服务器配置是否和生产服务器足够接近。在项目开始时，检查生产服务器的配置，将配置放进 `php.yml` 文件中，这样你在开发和测试时就有信心确保将项目部署到生产环境时不会遇到兼容性错误。

不管服务器的 `php.ini` 文件如何定义，定义在 `set` 头的变量已被修改。而 `warn` 部分的变量则不会被立即修改，即使这些参数设置得不对，`symfony` 仍旧能正常运行。如果某次误将 `warn` 中的参数都设置为 `off`，则 `symfony` 将记录一个警告日志。`check` 部分的参数也无需修改，但是为了 `symfony` 能运行，必须为它们设置一个值。如果 `php.ini` 设置错误，将会抛出一个异常。

默认的 `php.yml` 文件将 `log_errors` 设置为 `on`，因而你可以在 `symfony` 项目中追踪错误。建议将 `register_globals` 设置为 `off`，以免留下安全漏洞。

如果你不希望 `symfony` 采用这些设置，或者你想将 `magic_quotes` 和 `register_globals` 设置为 `on` 而不发出警告，那就在你的应用程序的 `config` 目录下创建一个 `php.yml` 文件，并且用你所要的值来设置那些参数。

另外，如果你的项目需要一个 PHP 扩展，你可以在 `extension` 类别中使用如下的数组来指明：

```
extensions: [gd, mysql, mbstring]
```

总结

配置文件对框架的运行有重要的影响。因为 `symfony` 的核心特性和文件导入都依赖于配置，所以它能适应许多不同的环境，而不仅仅是标准环境。`symfony` 的重要特点就是它的高可配置性。尽管这么多配置文件和一大堆规定会吓坏初学者，但是它确实可以让 `symfony` 应用程序适用于大量的平台和环境。一旦你掌握了 `symfony` 配置，你的应用程序将可运行在任意服务器上。