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~~模板片段~~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(corss-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~~模板片段~~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') ?>
 => <img src="/images/myimage.png" alt="foo" width="200"
height="100"/>
<?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 helper/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_helpler('Text')的时候 symfony 会使用你定

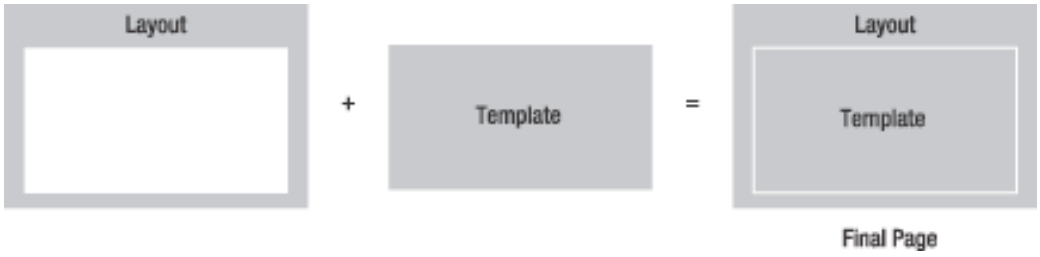
义的辅助函数而不是系统默认的。但是请注意：由于原始文件没有载入，你需要重新定义这组辅助函数里所有的函数，否则某些系统的辅助函数就用不了了。

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 出版 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.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。

模板片段局部模板 （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~~模板片段~~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

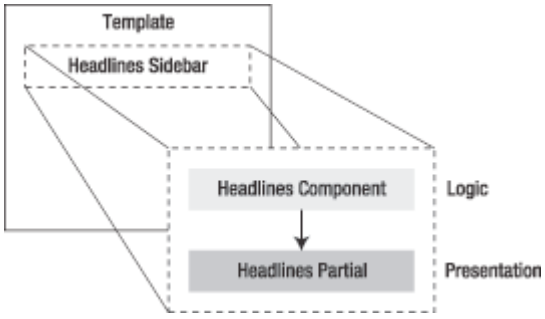
命名方式	动作	shitulingjian
逻辑文件	actions.class.php	components.class.php
继承的类	sfActions	sfComponents
方法命名	executeMyAction()	executeMyComponent()
表现文件命名	myActionSuccess.php	_myComponent.php

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。

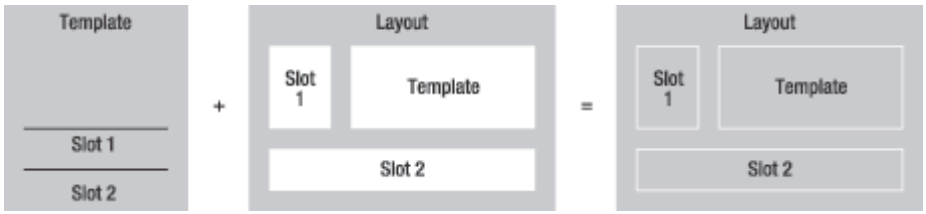
槽 （Slots）

~~模板片段~~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~~模板片段~~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~~模板片段~~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），~~模板片段~~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，~~模板片段~~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~~模板片段~~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~~模板片段~~局部模板
        module2/
          templates/   # module2 模块的模板和~~模板片段~~局部模板
        module3/
          templates/   # module3 模块的模板和~~模板片段~~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~~模板片段~~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~~模板片段~~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~~模板片段~~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的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，你就能够理解这点了。

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，前面例子里的设置都会输出例 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 ~~模板片段~~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~~模板片段~~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：

```
&lt;script&gt;alert(document.cookie)&lt;/script&gt;
```

你可以手动的给每个不确定的值加上 htmlentites()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 决定默认的转义函数。

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 htmlentites()。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;
 => &gt;&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');
=> &gt;&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 htmlentites()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放在 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 jvascript: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 enforeach; ?>
 => &amp; &lt; &gt;
<?php echo $test_array_of_arrays[0][0] ?>
 => &amp;
<?php echo $test_object->testSpecialChars('&') ?>
 => &lt;&amp;&gt;
```

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;&amp;&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，模板片段局部模板，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。