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）

动作是整个项目的心脏，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。原因就很简单：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~~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~~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~~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

synfony 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， 只要你不用 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		
getMethod()	请求的方法	返回 sfRequest::GET 或者 sfRequest::POST 常量

名称	功能	输出例子
getMethodName()	请求方法名	'POST'
getHttpHeader('Server')	某个指定的 HTTP 头的值	'Apache/2.0.59 (Unix) DAV/2 PHP/5.1.6'
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	'/mymodule/myaction'

名称	功能	输出例子
	信息	
getReferer()**	来源	'http://localhost/myapp_dev.php/'
getHost()	主机名	'localhost'
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~~symfony 会自动地处理用户会话并保持用户请求的连续性。
~~Symfony~~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~~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~~symfony 把会话 ID 存在客户端的 cookies 上。~~Symfony~~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~~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

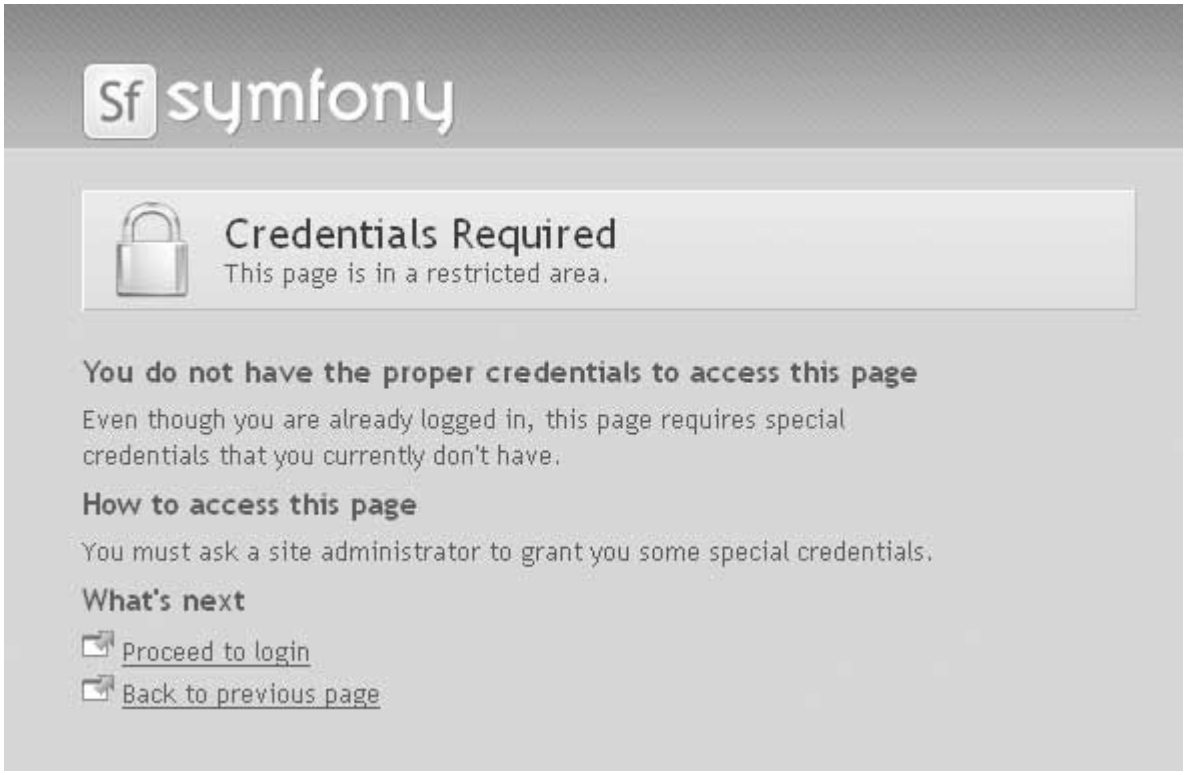
```
read:
  is_secure:   off        # 所有的用户都可以执行 read 动作

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]]
```

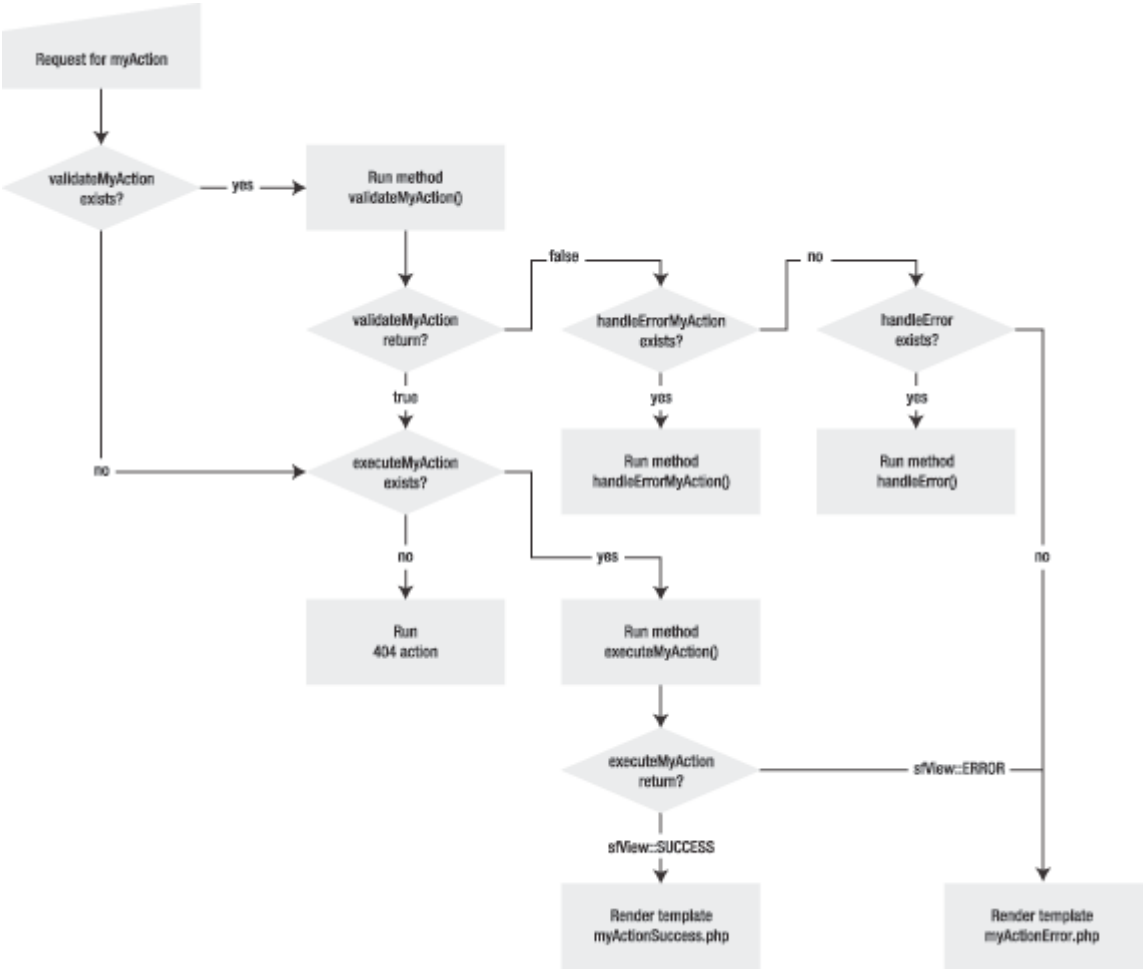
```
                # root OR (supplier AND (owner OR quasiowner)) OR
accounts
```

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~~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。

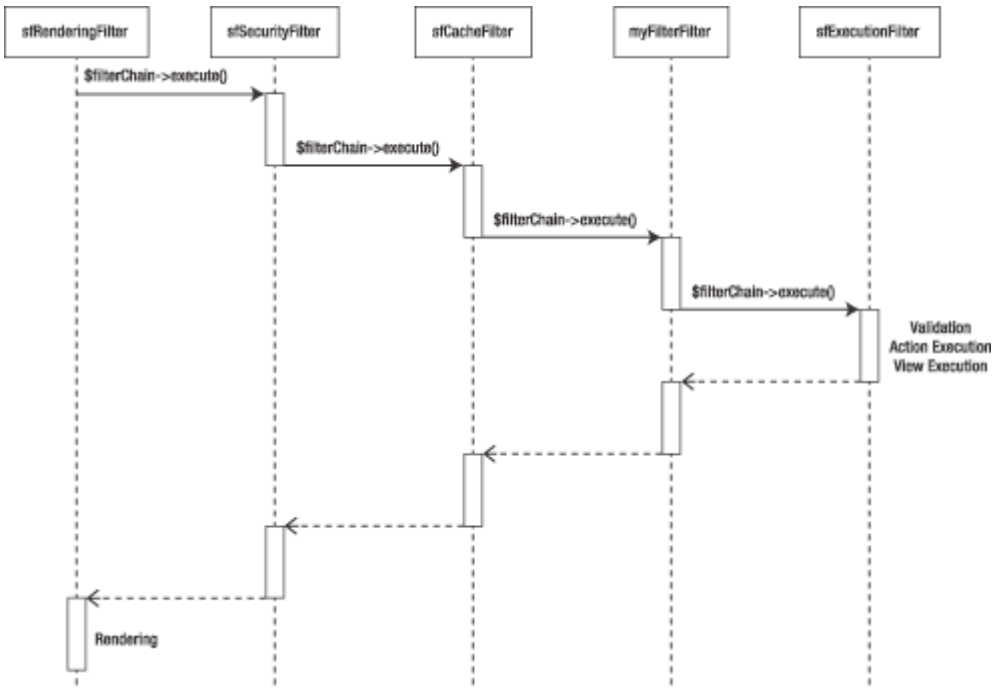
过滤器

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~~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~~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），你可以出重写这个方法。

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_ireplace('</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。