

## 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：

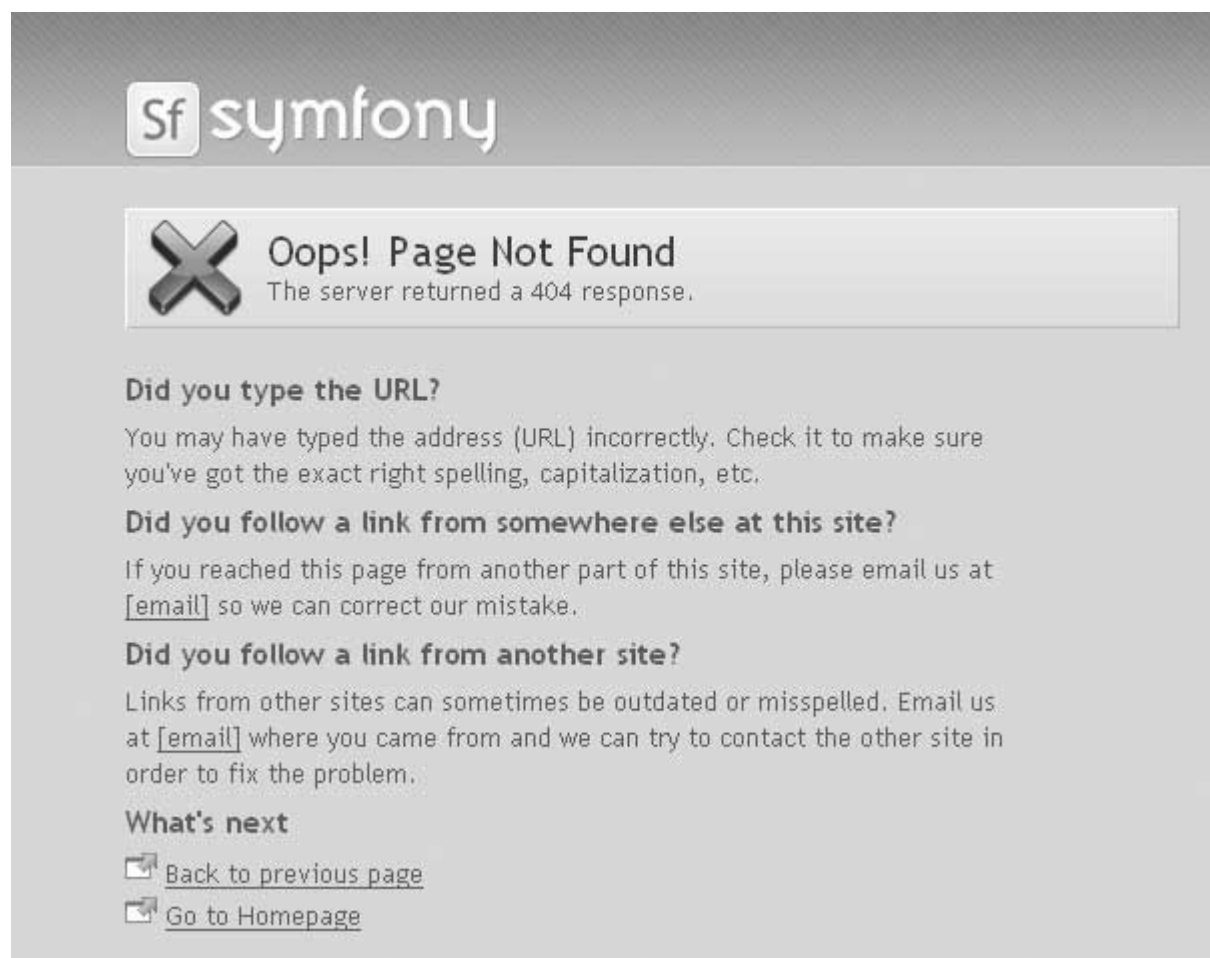
- error\_404\_module 和 error\_404\_action: 当用户输入的 URL 和任何路由都不匹配时或当一个 sfError404Exception 产生时，就调用这个动作。默认值是 default/error404。
- login\_module 和 login\_action: 当一个未经授权的用户试图访问由 security.yml 中定义为 secure 的页面时(请参考第 6 章的解释)，这个动作将被调用。默认值是 default/login。
- secure\_module 和 secure\_action: 当一个用户不具备某个动作所需的信任证书时，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

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> ，你将不能使用数据库。	<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>
<code>il18n</code>	激活接口翻译特性（请参看第 6 章）。如果你的应用	<code>off</code>

参数	描述	默认值
logging_enabled	程序是多种语言的，将它设为 on。 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：

- suffix 参数为生成的 URL 设置默认的后缀。suffix 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 class 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\_suffix 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使用 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_il8n_dir      # il8n/
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_il8n_cache_dir    # il8n/
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::g
et('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-projec` 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。parseYamls() 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 中指定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->parseYamls ($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:      |
    &

check:

  zend.zel_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。