

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, 使你能够更快地发现出错之处。
- 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
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。

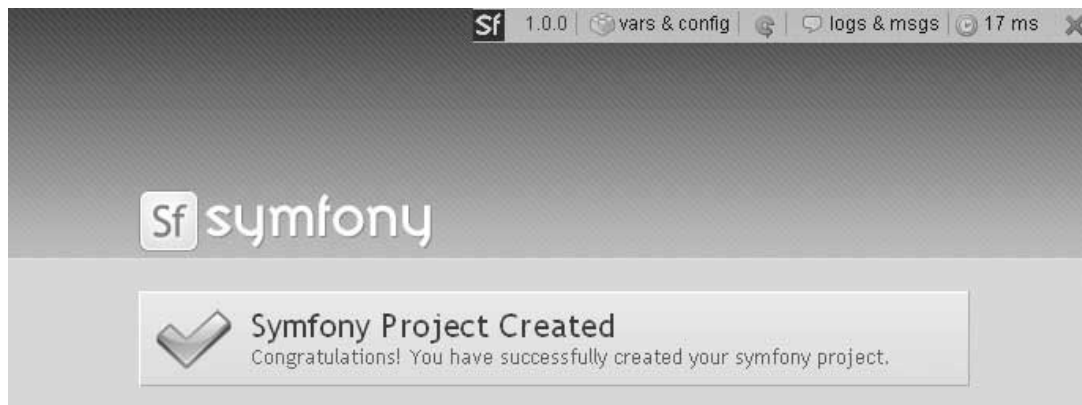
CAUTION 别忘记在生产平台下关闭 Xdebug 模式。否则会大大降低每个页面的执行速度。

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

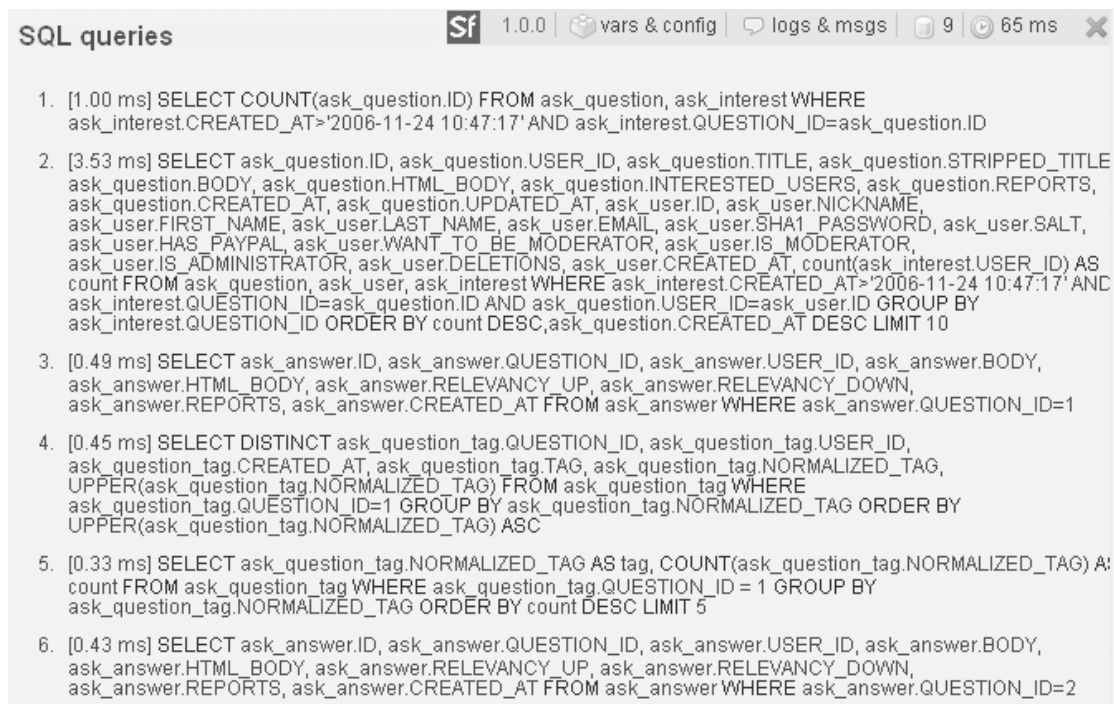


图 16-6 - 时钟图标按照类别显示执行时间

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 级别确保了这个事件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。

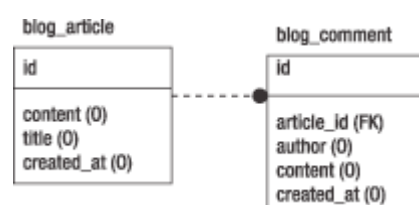
```
> symfony propel-load-data frontend myfixtures/myfile.yml
```

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速。

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。