第 17 章 扩展 symfony

无论你需要修改核心类的行为或者增加自制的功能，都不可避免地需要调整 ~~Symfony~~symfony 的行为，没有一个框架可以预测到用户的所有特殊需求。事实上，这种情况很常见，因此 symfony 提供一个可以扩展已有类的机制，叫做 mixin。你甚至可以用 factories 设置替换掉 symfony 核心类。当建立好一个扩展，你可以非常方便地把它打包为一个插件，这时候就可以在你的其他应用程序中重复使用，或者被其他 symfony 用户使用了。

Mixins

目前 PHP 的限制中，最恼人的一个问题是一个类无法继承一个以上的类。另一个是你不能对一个已有类增加新的方法或者覆写现存方法。为了解决这两个限制并使框架真正的可以扩展，symfony 使用了一个叫做 sfMixer 的类。这绝不是一个什么烹饪工具，而是面向对象编程中的一个 mixin 概念。一个 mixin 是一组可被混入一个类的方法或者功能。

理解多重继承

多重继承就是指一个类同时继承多个类并继承了这些类的属性和方法。考察例 17-1： 想像一下 Story 和 Book 这两个类，每一个类都有自己的属性和方法。

例 17-1 - 两个示例类

```
class Story
{
  protected $title = '';
  protected $topic = '';
  protected $characters = array();

  public function __construct($title = '', $topic = '', $characters =
array())
  {
    $this->title = $title;
    $this->topic = $topic;
    $this->characters = $characters;
  }

  public function getSummary()
  {
```

```
    return $this->title.', a story about '.$this->topic;
  }
}

class Book
{
  protected $isbn = 0;

  function setISBN($isbn = 0)
  {
    $this->isbn = $isbn;
  }

  public function getISBN()
  {
    return $this->isbn;
  }
}
```

ShortStory 类继承自 Story 类，ComputerBook 类继承自 Book 类，逻辑上说，Novel 应该同时继承自 Story 和 Book 这两个类、使用他们的方法。不幸的是，这在 PHP 中无法实现。你不能像例 17-2 一样写 Novel 的声明。

例 17-2 - PHP 无法使用多重继承

```
class Novel extends Story, Book
{
}

$myNovel = new Novel();
$myNovel->getISBN();
```

一个可行的解决方法就是让 Novel 实现两个接口来代替继承两个类，但这会让你无法使用父类已有的方法。

Mixing 类

sfMixer 类用另一种方法来解决了这个问题，假如类包含了合适的钩子，就能用它来扩展一个已有类。这个过程包含两个步骤：

- 声明类是可扩展的
- 在声明类后注册一个扩展 (或者 mixins)

例 17-3 展示了如何用 sfMixer 实现 Novel 类。

例 17-3 - 通过 sfMixer 可以实现多重继承

```
class Novel extends Story
{
  public function __call($method, $arguments)
  {
    return sfMixer::callMixins();
  }
}

sfMixer::register('Novel', array('Book', 'getISBN'));
$myNovel = new Novel();
$myNovel->getISBN();
```

其中一个类（Story）被选择作为父类，这符合了 PHP 中只能继承一个类的特性。用__call()方法声明了 Novel 类是可扩展的。另一个类（Book）的方法随后也通过调用 sfMixer::register()加入到 Novel 类中。下面将详述这个过程。

调用 Novel 类中的 getISBN()方法时，可以实现例 17-2 同样的效果，不同之处在于这是由_call()方法的魔术和 sfMixer 的静态方法模拟的。getISBN()方法就是这么混入 Novel 类的。

SIDEBAR 什么时候使用 mixin

symfony 的 mixin 机制在很多情况下都是有用的。如前所述的模拟多重继承，就是其中之一。

你可以在声明后使用 mixins 来改变一个方法。例如，当建立一个图库时，你也许会实现一个 Line 对象——来显示一条线。它将会有四个属性（两端的坐标）和 draw()方法来渲染自身。ColoredLine 也应有相同的属性和方法，但多了一个额外的属性——color，来表示它的颜色。并且，ColoredLine 类的 draw()方法也和 Line 类中的有些不同，他会使用对象的颜色。你可以在 ColoredElement 类中包装一个图形单元的功能来处理颜色。这可以让你在其他图形单元（Dot, Polygon 及其他）重复使用 color 方法。在这个例子中，是用 Line 类的扩展，也就是把 ColoredElement 类混入的方式来实现 ColoredLine 类。最终的 draw()方法是由 Line 中原有的 draw 方法和 ColoredElement 中的 draw 方法混合而成的。

Mixins 也能被用来在已有类中新增一个方法。例如，symfony 框架中定义的一个叫做 sfAction 的行为类，你也许只想在你的应用程序中为 sfAction 增加一个自定义的方法——例如，把一个请求指向一个特别的网页服务。对此，PHP 无法完成，因为 PHP 的一个限制是在初始声明后无法修改 sfAction 的定义。但是 mixin 机制提供了一个完美的解决方案。

声明一个类是可扩展的

你必须在代码中插入一个或多个"钩子（hooks）"来声明此类是可扩展的，这样才能让 sfMixer 类可以识别。这些钩子其实是 sfMixer::callMixins()方法调用。许多 symfony 类已经含有这些钩子，包括 sfRequest, sfResponse, sfController, sfUser, sfAction 和其他。

钩子可以放在类中的不同地方，根据扩展需要的程度而定：

- 要在类中增加新的方法，你必须在__call()方法中插入钩子并返回它的结果，如例 17-4 所示。

例 17-4 - 让一个类可以增加新方法

```
class SomeClass
{
  public function __call($method, $arguments)
  {
    return sfMixer::callMixins();
  }
}
```

- 要更改已有的方法，你必须在方法中插入钩子，如例 17-5 所示。由 mixin 类增加的代码会在放置钩子的地方被执行

例 17-5 - 让一个方法可被修改

```
class SomeOtherClass
{
  public function doThings()
  {
    echo "I'm working...";
    sfMixer::callMixins();
  }
}
```

你也许想要在一个方法中放入多个钩子。因此，你需要给钩子命名，这样你能在以后定义哪个钩子要被扩展，如例 17-6 所示。可以把钩子名字作为参数，让 callMinxins()方法来调用一个钩子。这个名字会在注册 mixin 的时候告诉方法需要去执行哪段 mixin 代码。

例 17-6 - 一个方法可以包含多个钩子，但是必须命名它们

```
class AgainAnotherClass
```

```
{
  public function doMoreThings()
  {
    echo "I'm ready.";
    sfMixer::callMixins('beginning');
    echo "I'm working...";
    sfMixer::callMixins('end');
    echo "I'm done.";
  }
}
```

当然，你可以配置一个新的、可扩展的方法这些技巧来建立新的类，如例 17-7 所示。

例 17-7 - 可用多种方法来扩展类

```
class BicycleRider
{
  protected $name = 'John';

  public function getName()
  {
    return $this->name;
  }

  public function sprint($distance)
  {
    echo $this->name." sprints ".$distance." meters\n";
    sfMixer::callMixins(); // sprint()方法是可扩展的
  }

  public function climb()
  {
    echo $this->name.' climbs';
    sfMixer::callMixins('slope'); // sprint()方法可在此扩展
    echo $this->name.' gets to the top';
    sfMixer::callMixins('top'); // 也可以在这里
  }

  public function __call($method, $arguments)
  {
    return sfMixer::callMixins(); // BicyleRider类是可扩展的
  }
}
```

CAUTION 只有声明为可扩展的类才能用 sfMixer 来扩展。也就是说你无法利用此机制来扩展一个并没有订阅此服务的类。

注册扩展（Extensions）

用 sfMixer::register()方法在现有钩子上注册扩展。它的第一个参数是需要扩展的元素名，第二个参数是一个代表 mixin 的 PHP 调用名。

第一个参数的格式取决于你想扩展什么内容：

- 如果要扩展一个类，就用类名。
- 如果要用未命名的钩子扩展一个方法，用 class:method 模式。
- 要用已命名的钩子来扩展一个方法，用 class:method:hook 模式。

例 17-8 通过举例扩展例 17-7 定义的类来说明这个原理。可扩展的对象自动传递第一个参数给 mixin 方法（当然，除非可扩展对象是静态的）。Mixin 方法也获得了访问原方法参数的权限。

例 17-8 - 注册扩展

```
class Steroids
{
  protected $brand = 'foobar';

  public function partyAllNight($bicycleRider)
  {
    echo $bicycleRider->getName()." spends the night dancing.\n";
    echo "Thanks ".$brand."!\n";
  }

  public function breakRecord($bicycleRider, $distance)
  {
    echo "Nobody ever made ".$distance." meters that fast before!\n";
  }

  static function pass()
  {
    echo " and passes half the peloton.\n";
  }
}

sfMixer::register('BicycleRider', array('Steroids',
'partyAllNight'));
sfMixer::register('BicycleRider:sprint', array('Steroids',
```

```
'breakRecord'));
sfMixer::register('BicycleRider:climb:slope', array('Steroids',
'pass'));
sfMixer::register('BicycleRider:climb:top', array('Steroids',
'pass'));

$superRider = new BicycleRider();
$superRider->climb();
=> John climbs and passes half the peloton
=> John gets to the top and passes half the peloton
$superRider->sprint(2000);
=> John sprints 2000 meters
=> Nobody ever made 2000 meters that fast before!
$superRider->partyAllNight();
=> John spends the night dancing.
=> Thanks foobar!
```

扩展机制不只是用来新增方法。partyAllNight()方法使用了 Steroids 类的一个属性。这就意味着当用 Steroids 类的方法来扩展 BicycleRider 类时，你实际上在 BicycleRider 对象里建立了一个新的 Steroids 实例。

CAUTION 你不能在现有类中增加两个同名方法。这是因为在__call()方法中 callMixins()调用时使用的 mixin 方法名是关键字。同样，你不能在类中增加一个和类中方法同名的方法，因为 mixin 机制依靠__call()方法，所以在这种情况下，将会无法被调用到。

register()调用的第二个参数是 PHP 调用名，所以这可以是一个 class::method 数组，或是一个 object->method 数组，甚至是一个函数名，见例 17-9 的示例。

例 17-9 - 任何调用名都可以注册为 Mixer 扩展

```
// 用一个类方法作为调用名
sfMixer::register('BicycleRider', array('Steroids',
'partyAllNight'));

// 用一个对象方法作为调用名
$mySteroids = new Steroids();
sfMixer::register('BicycleRider', array($mySteroids,
'partyAllNight'));

// 用一个函数作为调用名
sfMixer::register('BicycleRider', 'die');
```

扩展机制是动态的，这就意味着尽管你已经实例化了一个对象，它还是能在类

中利用进一步的扩展。见例 17-10 的示例。

例 17-10 - 扩展机制是动态的，甚至可以在实例化后发生

```
$simpleRider = new BicycleRider();
$simpleRider->sprint(500);
=> John sprints 500 meters
sfMixer::register('BicycleRider:sprint', array('Steroids',
'breakRecord'));
$simpleRider->sprint(500);
=> John sprints 500 meters
=> Nobody ever made 500 meters that fast before!
```

更精确的扩展

sfMixer::callMixins()指令实际上是一个复杂处理过程的快捷方式。它自动在已注册的 mixin 上循环，逐个调用它们，传递给它当前对象和当前方法参数。简单来说，一个 sfMixer::callMixins()调用行为或多或少就像例 17-11 所示。

例 17-11 - callMixin()在已注册 Mixin 上循环并执行它们

```
foreach (sfMixer::getCallables($class.':'.$method.':'.$hookName) as
$callable)
{
  call_user_func_array($callable, $parameters);
}
```

如果想对返回值传递其他参数或者做其他设置，可以写一个详尽的 foreach 循环替换掉快捷方法。例 17-12 展示了一个在类中更完整的 mixin。

例 17-12 - 用定制的循环替换 callMixin()

```
class Income
{
  protected $amout = 0;

  public function calculateTaxes($rate = 0)
  {
    $taxes = $this->amount * $rate;
    foreach (sfMixer::getCallables('Income:calculateTaxes') as
$callable)
    {
```

```
      $taxes += call_user_func($callable, $this->amount, $rate);
    }

    return $taxes;
  }
}

class FixedTax
{
  protected $minIncome = 10000;
  protected $taxAmount = 500;

  public function calculateTaxes($amount)
  {
    return ($amount > $this->minIncome) ? $this->taxAmount : 0;
  }
}

sfMixer::register('Income:calculateTaxes', array('FixedTax',
'calculateTaxes'));
```

SIDEBAR Propel 行为

我们在前面第八章讨论过的 Propel 行为（behavior)是一个特殊的 mixin 类型：它们扩展了 Propel 生成的对象。让我们看一个例子。

Propel 对象对应数据库的表，他们都有一个 delete()方法用来在数据库中删除相关的记录。但是因为不能删除一个 Invoice 类的记录，因此你想要把 delete()方法改为让记录保留在数据库并设置 is_deleted 属性为 true。通常的对象检索方法(doSelect(), retrieveByPk())只会考虑记录的 is_deleted 是否是 false。你也可以增加一个 forceDelete()方法使你彻底删除记录。事实上，所有的这些修改可以包装为一个新的 ParanoidBehavior 类。 最终 Invoice 类扩展自 propel BaseInvoice 类并把 ParanoidBehavior 的方法混入。

因此行为是 Propel 对象的 mixin。实际上，symfony 术语“行为”还包含了另一个意思：mixin 被包装成了插件。刚刚提到的 ParanoidBehavior 类对应于一个叫做 sfPropelParanoidBehaviorPlugin 的 symfony 插件。可以参照 symfony wiki(http://www.symfony-project.com/trac/wiki/sfPropelParanoidBehaviorPlugin)来获得此插件更详细的安装使用说明。

最后一件关于行为的事情：如果想要使用行为，生成的 Propel 对象必须包含一些钩子。如果你不使用行为的话这样会降低一些执行效率及性能。所以钩子不是默认就激活的。要打开行为支持，需要在 propel.ini 文件中设置 propel.builder.addBehaviors 属性为 true，然后重建模块。

Factories

factory 是对某任务所用类的定义。symfony 依靠 factories 作为它的核心功能，就像控制器和用户会话（session）。例如，当框架需要建立一个新的请求对象的时候，它会在 factory 的定义里搜索用来创建这个对象的类名 。请求对象默认的 factory 定义是 sfWebRequest，所以 symfony 建立这个类的对象来处理请求。使用 factory 定义最大的好处就是十分容易修改框架的核心功能：只要修改他的 factory 定义，然后 symfony 将使用自定义的请求类替代原有定义。

factory 定义存放在 factories.yml 配置文件中。例 17-13 展示了默认的 factory 定义文件。每一个定义是由自动载入类的名字和（可选）一系列的参数组成。例如，用户会话储存 factory（在 storage:关键字下设置）使用了一个 session_name 参数来命名在客户电脑上的 cookie，由此来建立一个永久的用户会话。

例 17-13 - 默认的 Factories 文件在 myapp/config/factories.yml

```
cli:
  controller:
    class: sfConsoleController
  request:
    class: sfConsoleRequest

test:
  storage:
    class: sfSessionTestStorage

#all:
#  controller:
#    class: sfFrontWebController
#
#  request:
#    class: sfWebRequest
#
#  response:
#    class: sfWebResponse
#
#  user:
#    class: myUser
#
#  storage:
#    class: sfSessionStorage
#    param:
#      session_name: symfony
```

```
#
#  view_cache:
#    class: sfFileCache
#    param:
#      automaticCleaningFactor: 0
#      cacheDir:                %SF_TEMPLATE_CACHE_DIR%
```

改变 factory 最好的办法就是建立一个新的继承自默认 factory 的类并加入新的方法。例如，用户会话 facotry 设置为 myUser 类（在 myapp/lib/）并继承自 sfUser。利用已存在的 factory 去使用相同的机制。

例 17-14 - 重写 factories

```
// 在一个可以自动载入目录中建立一个 myRequest.class.php，
// 例如在 myapp/lib/
<?php

class myRequest extends sfRequest
{
// 你的代码放这里
}

// 在 factories.yml 中把此类作为请求`request` factory 声明
all:
  request:
    class: myRequest
```

桥接其他框架组件

如果你需要使用第三方类，但并不准备把这个类复制到 symfony 的 lib/目录中，你也许将会在其他目录中安装并让 symfony 使用这个类。因此，除非你使用 symfony 的自动加载机制，否则使用这个类也就意味着需要在代码中出现 require。

symfony（目前）尚未提供所有的工具。如果你想要一个 PDF 生成器，Google 地图的 API 或者是 Lucene 搜索引擎的 PHP 实现，你也许需要 Zend 框架中的一些类。如果你想在 PHP 中直接处理图片，连上一个 POP3 帐号读取 e-mail,或是设计一个终端界面，你可以使用 eZcomponents 的一些类库。幸运的是，如果你正确的定义了设置，这些类库在 symfony 中会运行的很好。

首先，你需要在应用程序的 settings.yml 中声明（除非你通过 PEAR 安装了第三方的库）库根目录的路径：

```
.settings:
```

```
  zend_lib_dir:    /usr/local/zend/library/
  ez_lib_dir:      /usr/local/ezcomponents/
```

然后通过指定当 symfony 自动载入失败要用哪个库来扩展自动载入路由机制：

```
.settings:
  autoloading_functions:
    - [sfZendFrameworkBridge, autoload]
    - [sfEzComponentsBridge,  autoload]
```

这个设置和在 auutoload.yml 定义的规则不同（在 19 章有关于此文件的更多信息）。autoloading_functions 设置了 bridge 类，autoload.yml 中设置了路径和搜索规则。下面的描述说明了当新建一个未载入类的对象会产生的状况：

1. ~~Symfony~~symfony 自动载入功能(sfCore::splAutoload())首先查找在 autoload.yml 中声明的路径中的类。
2. 如果没有找到，会逐个的调用在 sf_autoloading_functions 中声明的 callback 方法，直到找到为止。
3. sfZendFrameworkBridge::autoload()
4. sfEzComponentsBridge::autoload()
5. 如果以上都没有找到，而你使用的是 PHP 5.0.X, symfony 会抛出一个 exception 说明类不存在。从 PHP 5.1 开始，PHP 会自生成错误讯息。

这就是说，用自动载入机制可以使用其他的框架组件，甚至比它们自身的环境下更加方便。例如，如果你想在 PHP 中用 Zend 框架的 Zend_Search 组件来实现 Lucene 搜索引擎，你需要这样写：

```
require_once 'Zend/Search/Lucene.php';
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
...
```

用 symfony 桥接 Zend 框架的话，就简单了。写法如下：

```
$doc = new Zend_Search_Lucene_Document(); // 此类可以自动载入
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
...
```

在$sf_symfony_lib_dir/addon/bridge/目录中有可用的 bridge。

插件

你也许会需要重用自行开发的 symfony 应用程序中的一些代码。如果可以把这

些代码包装为单一的类：把这个类放在其他应用程序的 lib/目录下，自动载入机制会处理其他的事情。但是如果这些代码是分散的，不单只是一个文件，就像是一个完整的管理界面生成器用的主题或者是一组自动完成你所喜欢的特殊效果的 Javascript 文件和辅助函数，单纯复制这些文件不是最好的解决方案。

插件提供了一个把分散在一系列文件中的代码包装起来并能在几个项目中重新使用的方法。在插件里，你能包装类、过滤器、mixins、辅助函数、 配置文件、任务、模块、设计（schema）和模型扩展、fixtures、网页资源等。插件易于安装、升级、卸载。他们可以发布为.tgz 压缩包，PEAR 包，或者是直接从版本库里检出。打包为 PEAR 的插件有利于管理关联关系，易于升级和自动维护。symfony 载入机制会把插件考虑在内，项目中可以像使用 symfony 的功能一样使用插件提供的功能。

因此，通常来说，插件就是 symfony 打包的扩展。有了插件，不仅可以在不同的应用程序中使用你的代码，而且可以使用其他的第三方的扩展程序。

查找 symfony 插件

symfony 官方网站上有一个用来发布 symfony 插件的网页。它在网站的 wiki 部分，可以通过以下网址访问：

> http://www.symfony-project.com/trac/wiki/~~Symfony~~symfonyPlugins

每一个插件都有自己独立的页面，包含了详细的安装指南和文档。

其中的一些插件是社区贡献的，一些是 symfony 核心开发组发布的。symfony 核心开发组发布的插件包括下面这些：

- sfFeedPlugin: 自动处理 RSS 和 Atom feeds
- sfThumbnailPlugin: 建立图片的缩略图
- sfMediaLibraryPlugin: 允许上传和管理媒体文件，包括一个富文本编辑器的扩展（可以实现在富文本编辑器里选择图片）
- sfShoppingCartPlugin: 购物车管理
- sfPagerNavigationPlugin: 基于 sfPager 对象提供了传统和 ajax 的页面控制
- sfGuardPlugin: 在 symfony 标准安全功能上提供验证，认证和其他用户管理功能
- sfPrototypePlugin: 提供了 Prototype 和 script.aculo.us 这两个 Javascript 框架的 Javascript 文件，这是一个独立库（区别于 symfony 自带的 prototype 和 script.aculo.us 文件
- sfSuperCachePlugin: 将页面写入网页根目录下的 cache 目录，使服务器直接输出这些内容，从而最大限度的提高速度
- sfOptimizerPlugin: 优化应用程序的代码让其在生产环境中执行的更快（详情看下一章）
- sfErrorLoggerPlugin: 把 404 和 505 错误记录到数据库中，提供一个管

理模块来浏览这些错误
- sfSslRequirementPlugin：为动作提供了 SSL 加密支持

Wiki 上也有一些扩展 Propel 对象的插件，我们称它做行为（behaviors）。在他们中，你可以找到这些功能：

- sfPropelParanoidBehaviorPlugin：禁止直接删除功能，改为更新 deleted_at 列
- sfPropelOptimisticLockBehaviorPlugin：对 Propel 对象实现优化锁定

你应该经常去看一下 symfony wiki，那里会随时增加新的插件，他们会给你的 web 应用程序开发带来很多便捷。

除了 symfony wiki 之外，还有另外一种方式来发布插件，提供压缩文件的插件下载，或是在 PEAR 频道中，或者它们放在公开的版本控制库中。

安装插件

插件安装过程会因为不同的打包方式而有所不同。可以在插件下载页找到说明文件或者安装指南。同时，在安装了插件后都需要更新一下 symfony 缓存。

插件是在项目级别安装的应用程序。接下来章节中讨论的所有的方法都是把插件安装到 myproject/plugins/pluginName/目录下。

PEAR 插件

在 symfony wiki 列出的插件都是 PEAR 包形式的。可以使用 plugin-install 加上完整的 URL 地址来安装插件，如例 17-15 所示。

例 17-15 - 从 ~~Symfony~~symfony wiki 安装插件

```
> cd myproject
> php symfony plugin-install http://plugins.symfony-
project.com/pluginName
> php symfony cc
```

另一种方法是，你可以把插件下载到硬盘中，然后从硬盘上安装。这种模式下把频道名字替换为插件包的完整路径名，如例 17-16 所示。

例 17-16 - 安装一个已下载的 PEAR 包

```
> cd myproject
> php symfony plugin-install /home/path/to/downloads/pluginName.tgz
> php symfony cc
```

一些插件存在 PEAR 频道中。需要用 plugin-install 来安装，别忘了设置频道

名字，如例 17-17 所示。

例 17-17 - 从 PEAR 频道安装插件

```
> cd myproject
> php symfony plugin-install channelName/pluginName
> php symfony cc
```

所以无论是从 symfony wiki, PEAR 频道或者是下载的 PEAR 包安装, 这三种方式都用了 PEAR 包，都可用术语“PEAR 插件”来表示。

压缩包形式的插件

一些插件发布的是一个压缩包。只要解压缩到项目的 plugins/目录就能完成安装。如果插件包含 web/子目录，复制一份或者做一个符号链接到项目的 web/目录，如例 17-18 所示。最后，别忘了清空缓存。

例 17-18 - 从压缩包安装插件

```
> cd plugins
> tar -zxpf myPlugin.tgz
> cd ..
> ln -sf plugins/myPlugin/web web/myPlugin
> php symfony cc
```

从版本库安装插件

插件有时候放在他们的版本库中。只要检出到 plugins/目录即可，但如果你的项目本身就在版本控制下就会有一些问题。

另一种选择，你可以声明插件是依赖于外部库，这样，每次更新你的项目源代码的时候也会更新插件的源代码。例如，Subversion 在 svn:externals 中配置外部依赖。因此你能修改这个属性然后更新源代码来增加插件，如例 17-19 所示。

例 17-19 - 从版本库安装插件

```
> cd myproject
> svn propedit svn:externals plugins
  pluginName    http://svn.example.com/pluginName/trunk
> svn up
> php symfony cc
```

NOTE 如果插件包含 web/目录，与压缩文件的插件一样必须为它建立一个符号链接。

激活插件模块

一些插件包含完整的模块。模块插件和标准插件的区别就是模块插件不会在myproject/apps/myapp/modules/目录中（易于更新）。他们还需要在setting.yml 中激活，如例 17-20 所示。

例 17-20 - 在 myapp/config/settings.yml 中激活一个插件模块

```
all:
  .settings:
    enabled_modules:  [default, sfMyPluginModule]
```

这是为了避免当插件模块在应用程序不需要的时候出现，这会导致安全违例。试想一个插件提供了前台和后台模块。你会让前台模块在前台应用程序中工作，后台只是在后台应用程序中。这就是为什么插件模块默认是不激活的。

TIP 只有默认模块是默认被激活的。因为它属于框架，所以这不是一个真正的插件模块，它位于$sf_symfony_data_dir/modules/default/中。这个模块提供了配置文件和默认的 404 错误文件和需要证书错误提示。如果不想使用 symfony 默认页面，只要从 enabled_modules 设置中移除这个模块即可。

列出已安装的插件

看一下项目的 plugins/目录就能知道哪些插件已经安装好了，plugin-list 任务告诉你一些更多的信息： 每个已安装插件的版本号，频道名字（例 17-21）。

例 17-21 - 列出已安装的插件

```
> cd myproject
> php symfony plugin-list

Installed plugins:
sfPrototypePlugin               1.0.0-stable # pear.symfony-
project.com (symfony)
sfSuperCachePlugin              1.0.0-stable # pear.symfony-
project.com (symfony)
sfThumbnail                     1.1.0-stable # pear.symfony-
project.com (symfony)
```

升级和卸载插件

要去卸载一个 PEAR 插件，只需要在项目根目录执行 plugin-uninstall，如例 17-22 所示。需要在插件名前加入安装的频道名（使用 plugin-list 来确认频道名）。

例 17-22 - 卸载插件

```
> cd myproject
> php symfony plugin-uninstall pear.symfony-
project.com/sfPrototypePlugin
> php symfony cc
```

TIP 一些频道有别名。例如，pear.symfony-project.com 频道也可以叫做 symfony，这就是说你可以卸载执行 php symfony plugin-uninstall symfony/sfPrototypePlugin 来卸载 sfPrototypePlugin，和例 17-22 效果一样。

要卸载一个压缩包形式的插件或者一个 SVN 插件，需要手工从项目的 plugins/ 和 web/目录把文件移除，然后清空缓存。

要升级一个插件，可以使用 plugin-upgrade（用 PEAR 安装的插件）或做一个 svn update（如果是从版本库中获得的插件）。压缩包形式的插件升级不是很方便。

解读插件

插件是用 PHP 语言写的。如果你能了解应用程序是如何组织的，你就能了解插件的结构了。

插件文件结构

插件目录组织有点类似一个项目目录。插件文件必须放在正确的目录，这样才能在 symfony 需要的时候自动载入。

例 17-23 - 插件的文件结构

```
pluginName/
  config/
    *schema.yml         // 数据模型
    *schema.xml
    config.php          // 插件配置
  data/
    generator/
      sfPropelAdmin
        */              // 管理界面生成器主题
          templates/
          skeleton/
    fixtures/
      *.yml             // Fixtures 文件
    tasks/
      *.php             // Pake 任务
```

```
  lib/
    *.php               // 类
    helper/
      *.php             // 辅助函数
    model/
      *.php             // 模型类
  modules/
    */                  // 模块
      actions/
        actions.class.php
      config/
        module.yml
        view.yml
        security.yml
      templates/
        *.php
      validate/
        *.yml
  web/
    *                   // 网页资源文件
```

插件能做的事

插件能包含很多东西。他们的内容可以在通过命令行调用任务的时候和程序运行的时候处理。但要让插件正常工作的话，你必须注意以下几个方面：

- 数据模型是由 propel-任务检查的。当在项目中调用 propel-build-model 时候，你重建了项目模型和所有的插件模型。注意插件模型必须总是在 plugins.plinName.lib.model 中有一个包（package）属性，如例 17-24 所示。

例 17-24 - 在 myPlugin/config/schema.yml 中的插件模式声明

```
propel:
  _attributes:    { package: plugins.myPlugin.lib.model }
  my_plugin_foobar:
    _attributes:    { phpName: myPluginFoobar }
      id:
      name:           { type: varchar, size: 255, index: unique }
      ...
```

- 插件配置文件必须包含在插件的引导脚本中（config.php）。这个文件会在应用程序和项目配置程序之后执行，所以 symfony 会在那时引导。例如你可以使用这个文件来增加 PHP 包含路径或通过 mixin 来扩展已有类。

- Fixtures 文件位于插件的 data/fixtures/目录中，由 propel-load-data 任务处理。
- 插件中的任务在插件安装好后就能立即在 symfony 命令行中使用了。在任务前面加个前缀会比较有意义——例如，加上插件名字。输入 symfony 可以看到所有的任务，包含通过插件安装的任务。
- 自定义类会自动载入，就如你放在 lib/目录下一样。
- 辅助函数会在模板调用 use_helper()的时候自动找到。他们必须在其中一个插件的 lib/目录的 helper/子目录下。
- 模型类在 myplugin/lib/model/意味着模型类是由 Propel 生成器生成的（在 myplugin/lib/model/om/和 myplugin/lib/model/map/）。这就是说，他们会自动被载入。小心你可以在自己的项目目录中覆写生成的插件模型类。
- 模块提供了新的行为来访问外部，也就是在你应用程序的 enabled_modules 设置中声明的那些。
- 网页资源（图片，脚本，样式表及其他）需要给服务器使用。当你通过命令行安装了一个插件的时候，如果服务器允许的话 symfony 会建立一个项目 web/目录的符号连接，或者 copy 一份模块 web/目录下的内容到项目中。如果是通过压缩包或者版本控制安装的插件，你必须手动复制插件的 web/目录（插件的 README 应有提示）。

手工设置插件

plugin-install 无法处理一些元素，所以需要在安装过程中手动设置：

- 插件代码中可以使用自定义的应用程序配置（例如，使用 sfConfig::get('app_myplugin_foo')），但是你不能把默认值放在插件 config/目录下的 app.yml 文件中。要处理默认值的话，用 sfConfig::get()方法的第二个参数。这些设置可以在应用层被覆盖（参考例 17-25 的示例）。
- 自定义的路由规则必须在应用程序的 routing.yml 中手动增加
- 自定义的过滤器必须在应用程序的 filters.yml 中手动增加
- 自定义的 factories 必须在应用程序的 factories.yml 中手动增加

通常来说，所有的配置都可以归结于一个需要手动配置的应用程序配置文件。需要这样手工配置的插件应该会包含一个 README 文件详细介绍安装过程。

为应用程序定制插件

想要定制插件的时候，绝对不要去修改在 plugins/目录下的代码。如果修改了，在更新插件的时候，所有的修改都会丢失的。为了可以自定义，插件提供了自定义的设置，这些设置是可以覆盖的。

设计优秀的插件可以在应用程序的 app.yml 中修改设置。如例 17-25 所示。

例 17-25 - 用应用程序的配置文件自定义插件

```
// 插件代码示例
$foo = sfConfig::get('app_my_plugin_foo', 'bar');

// 在应用程序的app.yml中设置'foo'的默认值('bar')
all:
  my_plugin:
    foo:       barbar
```

通常模块设置和他们的默认值都会在插件的README文件中有说明。

可以在你的应用程序中创建同名模块来替换插件模块的默认内容。这不是真正的覆写，因为是使用应用程序中的元素来替换掉插件中的同名元素。如果创建和插件同名模板和配置文件将会工作的很好。

另一方面，如果插件要提供一个用来覆写它本身的行为的模块的话，插件模块中的actions.class.php必须为空并且是从自动载入类中继承的，因此类的方法也可以通过应用程序模块的actions.class.php继承。

例 17-26 - 自定义插件行为

```
// 在myPlugin/modules/mymodule/lib/myPluginmymoduleActions.class.php
class myPluginmymoduleActions extends sfActions
{
  public function executeIndex()
  {
    // 一些代码
  }
}

// 在myPlugin/modules/mymodule/actions/actions.class.php
class mymoduleActions extends myPluginmymoduleActions
{
  // 空
}

// 在myapp/modules/mymodule/actions/actions.class.php
class mymoduleActions extends myPluginmymoduleActions
{
  public function executeIndex()
  {
    // 覆写插件代码
  }
}
```

如何写一个插件

plugin-install 只能安装那些包装为 PEAR 包的插件。记住这些插件可以发布在 symfony wiki 中，PEAR 频道中，或是下载的文件。因此，如果你想要制作一个插件，最好把它包装为 PEAR 压缩文件包。另外，PEAR 包的插件易于升级，可以声明关联，并自动将资源文件部署在 web/目录中。

文件组织

假设你开发了一个新的功能并想把它打包为一个插件。第一步是有逻辑的组织文件，让 symfony 载入机制可以在需要的时候找到它们。为了这个目的，你必须用例 17-23 给出的结构。例 17-27 展示了插件 sfSamplePlugin 文件的结构示例。

例 17-27 - 打包的插件的文件列表示例

```
sfSamplePlugin/
  README
  LICENSE
  config/
    schema.yml
  data/
    fixtures/
      fixtures.yml
    tasks/
      sfSampleTask.php
  lib/
    model/
      sfSampleFooBar.php
      sfSampleFooBarPeer.php
    validator/
      sfSampleValidator.class.php
  modules/
    sfSampleModule/
      actions/
        actions.class.php
      config/
        security.yml
      lib/
        BasesfSampleModuleActions.class.php
      templates/
        indexSuccess.php
  web/
    css/
      sfSampleStyle.css
```

```
    images/
      sfSampleImage.png
```

对于开发来说，插件目录的位置(例 17-27 中的 sfSamplePlugin/)并不重要。它可以在硬盘的任何地方。

TIP 用一个已有插件的例子，作为初次开发插件的模板，试着去重写它们的名字和文件结构。

建立 package.xml 文件

插件制作下一步就是在插件根目录增加一个 package.xml 文件。package.xml 遵循 PEAR 语法。看一下例 17-28 的标准 symfony 插件 package.xml。

例 17-28 - symfony 插件的 package.xml 示例

```
[xml]
<?xml version="1.0" encoding="UTF-8"?>
<package packagerversion="1.4.6" version="2.0"
xmlns="http://pear.php.net/dtd/package-2.0"
xmlns:tasks="http://pear.php.net/dtd/tasks-1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://pear.php.net/dtd/tasks-1.0
http://pear.php.net/dtd/tasks-1.0.xsd
http://pear.php.net/dtd/package-2.0 http://pear.php.net/dtd/package-
2.0.xsd">
 <name>sfSamplePlugin</name>
 <channel>pear.symfony-project.com</channel>
 <summary>symfony sample plugin</summary>
 <description>Just a sample plugin to illustrate PEAR
packaging</description>
 <lead>
  <name>Fabien POTENCIER</name>
  <user>fabpot</user>
  <email>fabien.potencier@symfony-project.com</email>
  <active>yes</active>
 </lead>
 <date>2006-01-18</date>
 <time>15:54:35</time>
 <version>
  <release>1.0.0</release>
  <api>1.0.0</api>
 </version>
 <stability>
  <release>stable</release>
```

```
  <api>stable</api>
 </stability>
 <license uri="http://www.symfony-project.com/license">MIT
license</license>
 <notes>-</notes>
 <contents>
  <dir name="/">
   <file role="data" name="README" />
   <file role="data" name="LICENSE" />
   <dir name="config">
    <!-- model -->
    <file role="data" name="schema.yml" />
   </dir>
   <dir name="data">
    <dir name="fixtures">
     <!-- fixtures -->
     <file role="data" name="fixtures.yml" />
    </dir>
    <dir name="tasks">
     <!-- tasks -->
     <file role="data" name="sfSampleTask.php" />
    </dir>
   </dir>
   <dir name="lib">
    <dir name="model">
     <!-- model classes -->
     <file role="data" name="sfSampleFooBar.php" />
     <file role="data" name="sfSampleFooBarPeer.php" />
    </dir>
    <dir name="validator">
     <!-- validators ->
     <file role="data" name="sfSampleValidator.class.php" />
    </dir>
   </dir>
   <dir name="modules">
    <dir name="sfSampleModule">
     <file role="data" name="actions/actions.class.php" />
     <file role="data" name="config/security.yml" />
     <file role="data" name="lib/BasesfSampleModuleActions.class.php"
/>
     <file role="data" name="templates/indexSuccess.php" />
    </dir>
   </dir>
   <dir name="web">
```

```
    <dir name="css">
     <!-- stylesheets -->
     <file role="data" name="sfSampleStyle.css" />
    </dir>
    <dir name="images">
     <!-- images -->
     <file role="data" name="sfSampleImage.png" />
    </dir>
   </dir>
  </dir>
 </contents>
 <dependencies>
  <required>
   <php>
    <min>5.0.0</min>
   </php>
   <pearinstaller>
    <min>1.4.1</min>
   </pearinstaller>
   <package>
    <name>symfony</name>
    <channel>pear.symfony-project.com</channel>
    <min>1.0.0</min>
    <max>1.1.0</max>
    <exclude>1.1.0</exclude>
   </package>
  </required>
 </dependencies>
 <phprelease />
 <changelog />
</package>
```

这里有趣的部分是<contents>和<dependencies>标签，我们会在下面介绍。其余的标签，对 symfony 没有什么特别的意义，关于 package.xml 格式详细信息你可以参考 PEAR 在线手册(http://pear.php.net/manual/en/)。

内容

<contents>标签是用来描述插件的文件结构的。这会让 PEAR 知道哪些文件要复制到什么地方。文件结构用<dir>和<file>标签。所有的<file>标签必须有一个 role="data"属性。例 17-28 的<contents>部分描述了与例 17-27 完全一致的目录结构。

NOTE 并不是强制使用<dir>标签，因为你可以在<file>标签中定义 name 来使用相对路径。无论如何，我们建议 package.xml 文件保持可读状态。

插件依赖性

插件是设计用来和一些版本的PHP、PEAR、symfony、PEAR包或者其他插件一起工作的。在<dependencies>标签声明这些依赖性告诉了PEAR在安装时要检查这些包是否已经安装了，如果没有，会给出异常信息。

你需要声明PHP，PEAR和symfony的依赖性，最小的需求，要保证至少其中之一对应了你的安装。如果你不知道该设置什么，建立设置PHP5.0、PEAR 1.4和symfony 1.0。

另外，建议在每个插件里设置一个最高允许的symfony版本号。这会导致在更高本版的框架下试着使用插件时会产生一个错误讯息，同时这会帮助插件作者在重新发布插件前确认插件在此版本下工作是否正常。在下载或者升级的时候得到一个错误信息总好过于插件遇到错误时不报错。

打包插件

PEAR组件有一个命令(pear package)用来创建.tgz压缩文件包，例17-29是从一个包含package.xml目录中使用命令的例子。

例 17-29 - 把插件打包为PEAR包

```
> cd sfSamplePlugin
> pear package

Package sfSamplePlugin-1.0.0.tgz done
```

当你的插件打包好之后，自行安装测试一下，如例17-30所示。

例 17-30 - 安装插件

```
> cp sfSamplePlugin-1.0.0.tgz /home/production/myproject/
> cd /home/production/myproject/
> php symfony plugin-install sfSamplePlugin-1.0.0.tgz
```

就如<contents>标签描述的，包里的文件最终会放在你的项目的不同的目录下，例17-31展示了sfSamplePlugin的文件安装后的位置。

例 17-31 - 插件的文件安装在plugins/和web/目录下

```
plugins/
  sfSamplePlugin/
    README
    LICENSE
    config/
      schema.yml
```

```
    data/
      fixtures/
        fixtures.yml
      tasks/
        sfSampleTask.php
    lib/
      model/
        sfSampleFooBar.php
        sfSampleFooBarPeer.php
      validator/
        sfSampleValidator.class.php
    modules/
      sfSampleModule/
        actions/
          actions.class.php
        config/
          security.yml
        lib/
          BasesfSampleModuleActions.class.php
        templates/
          indexSuccess.php
web/
  sfSamplePlugin/                ## 复制或者做链接，这取决于系统
    css/
      sfSampleStyle.css
    images/
      sfSampleImage.png
```

在应用程序中测试插件是否正常。如果工作正常，你可以准备在其他项目中使用它或者发布到 symfony 社区去。

在 symfony 项目主页发布你的插件

通过 symfony-project.com 网站发布的 symfony 插件拥有广大的用户。只要遵循这些步骤，你自己的插件也能通过这种方式发布：

1. 确保 README 文件描述了如何去安装和使用你的插件，LICENSE 文件说明了详细的许可信息。用 Wiki 语法格式(http://www.symfony-project.com/trac/wiki/WikiFormatting)来组织 README 文件。
2. 通过 pear package 命令为你的插件建立一个 PEAR 包，测试它。PEAR 包必须命名成 sfSamplePlugin-1.0.0.tgz (1.0.0 是插件版本号)格式。
3. 在 symfony wiki 建立一个叫做 sfSamplePlugin (Plugin 是必须的后缀)的新页面。在这个页面里，描述了插件的使用方法，许可信息，依赖信息和安装过程。你可以重复使用插件的 README 文件。请参考检查现存插件的 wiki 页。

4. 把你的 PEAR 包放到 wiki 页里(sfSamplePlugin-1.0.0.tgz)。
5. 在可用插件列表页面——也是一个 wiki 页面(http://www.symfony-project.com/trac/wiki/SymfonyPlugins)增加一个新的插件页([wiki:sfSamplePlugin])。

如果遵循这个流程，用户只要在项目目录中输入以下命令就能安装你的插件了：

```
> php symfony plugin-install http://plugins.symfony-project.com/sfSamplePlugin
```

命名约定

保持 plugins/目录干净，确保所有的插件名字用的都是驼峰命名方法并且都用 Plugin 作为结尾 （例如，shoppingCartPlugin，feedPlugin）。在命名你的插件前，检查一下是否已经有插件叫这个名字了。

NOTE 如果插件要用到 Propel，名字应该包含 Propel。例如，一个验证插件使用了 Propel 数据访问对象就应该叫做 sfPropelAuth。

插件应该有一个 LICENSE 文件来描述使用的条件并选定使用范围。我们也建议增加一个 README 文件来描述版本更改，插件的目标，它的作用，安装和配置指引，等等。

总结

~~Symfony~~symfony 类包含了 sfMixer 钩子，让它们可以在应用层被更改。mixin 机制允许多重继承和 PHP 禁止的动态覆盖。因此尽管你不许要修改核心类——factories 的配置文件，但是还是能十分方便的扩展 symfony 的功能。

很多扩展已经有了；他们被打包为插件，可以通过 symfony 命令行方便的安装，升级，卸载。建立一个插件就和建立一个 PEAR 包一样容易，这就让它可以在在多个应用程序中重复使用。

symfony wiki 包含了许多插件，你甚至可以加入你自己的。所以现在你知道了如何去做，我们希望你能通过很多有用的扩展来加强 symfony 内核！