第 10 章　表单

可以说，表单占据了开发人员编写模板的大部分时间，而且表单一般都设计得相当糟糕。由于涉及默认值，数据格式，验证，重填，表单处理等许多内容，开发者常常忽略了表单中的一些重要细节。而 symfony 恰恰对这个问题给予了特别的关注。本章介绍了为加速表单开发而设计的可以自动完成多种要求的开发工具：

- 表单辅助函数提供了一种比较快地在模板中编写表单控件的方法，特别是在编写诸如日期,下拉列表和富文本之类复杂的元素时。
- 如果要用一个表单去编辑一个对象的属性时，利用对象表单辅助函数可以进一步加速模板的编写。
- YAML 验证文件可以方便表单验证和重填。
- 验证器集成了用于验证输入数据的代码，symfony 绑定了满足最常用需求的验证器，开发人员也很容易定制自己的验证器。

表单辅助函数

在模板中，表单元素的 HTML 标签常常和 PHP 代码混杂在一起。symfony 中的表单辅助函数就是为了减少这种情形的发生并且避免在<input>标签中不断重复<?php echo 标签。

主要的表单标签

根据前面章节的介绍，你必须用 form_tag() 辅助函数创建表单，因为它可以将用参数表示的动作转换为经路由过的 URL。第二个参数还可以支持额外的选项。例如,可以改变默认的 method，可以改变默认的 enctype 或指定其他的属性，参见 例 10-1。

例 10-1 form_tag() 辅助函数

```
<?php echo form_tag('test/save') ?>
 => <form method="post" action="/path/to/save">

<?php echo form_tag('test/save', 'method=get multipart=true
class=simpleForm') ?>
 => <form method="get" enctype="multipart/form-data"
class="simpleForm"action="/path/to/save">
```

因为没有必要提供表单结束辅助函数，所以尽管看起来不怎么美观，你仍旧需要加上 HTML 的</form>标签。

标准的表单元素

有了表单辅助函数，表单中的每个元素都会默认以元素名作为其 id 属性。这个约定很有用。 例 10-2 给出了所有标准表单辅助函数及相关的选项。

例 10-2 标准表单辅助函数语法

```
// 输入框(text field)
<?php echo input_tag('name', 'default value') ?>
 => <input type="text" name="name" id="name" value="default value" />

// 所有表单辅助函数都接受一个额外的选项参数
// 它允许你为生成的标签加上定制的属性
<?php echo input_tag('name', 'default value', 'maxlength=20') ?>
 => <input type="text" name="name" id="name" value="default value"
maxlength="20" />

// 文本框（textarea）
<?php echo textarea_tag('name', 'default content', 'size=10x20') ?>
 => <textarea name="name" id="name" cols="10" rows="20">
      default content
    </textarea>

// 复选框（checkbox）
<?php echo checkbox_tag('single', 1, true) ?>
<?php echo checkbox_tag('driverslicense', 'B', false) ?>
 => <input type="checkbox" name="single" id="single" value="1"
checked="checked" />
    <input type="checkbox" name="driverslicense" id="driverslicense"
value="B" />

// 单选按钮（Radio button）
<?php echo radiobutton_tag('status[]', 'value1', true) ?>
<?php echo radiobutton_tag('status[]', 'value2', false) ?>
 => <input type="radio" name="status[]" id="status_value1"
value="value1"_checked="checked" />
    <input type="radio" name="status[]" id="status_value2"
value="value2" />

// 下拉列表（Dropdown list/select）
<?php echo select_tag('payment',
  '<option selected="selected">Visa</option>
   <option>Eurocard</option>
```

```
   <option>Mastercard</option>')
?>
 => <select name="payment" id="payment">
      <option selected="selected">Visa</option>
      <option>Eurocard</option>
      <option>Mastercard</option>
    </select>

// 可选项列表
<?php echo options_for_select(array('Visa', 'Eurocard',
'Mastercard'), 0) ?>
 => <option value="0" selected="selected">Visa</option>
    <option value="1">Eurocard</option>
    <option value="2">Mastercard</option>

// 混合了可选项的下拉列表辅助函数
<?php echo select_tag('payment', options_for_select(array(
  'Visa',
  'Eurocard',
  'Mastercard'
), 0)) ?>
 => <select name="payment" id="payment">
      <option value="0" selected="selected">Visa</option>
      <option value="1">Eurocard</option>
      <option value="2">Mastercard</option>
    </select>

// 用关联数组指明选项名称
<?php echo select_tag('name', options_for_select(array(
  'Steve'  => 'Steve',
  'Bob'    => 'Bob',
  'Albert' => 'Albert',
  'Ian'    => 'Ian',
  'Buck'   => 'Buck'
), 'Ian')) ?>
 => <select name="name" id="name">
      <option value="Steve">Steve</option>
      <option value="Bob">Bob</option>
      <option value="Albert">Albert</option>
      <option value="Ian" selected="selected">Ian</option>
      <option value="Buck">Buck</option>
    </select>

// 可复选的下拉列表(选中值可以是一个数组)
```

```
<?php echo select_tag('payment', options_for_select(
  array('Visa' => 'Visa', 'Eurocard' => 'Eurocard', 'Mastercard' =>
'Mastercard'),
  array('Visa', 'Mastecard'),
), array('multiple' => true))) ?>

 => <select name="payment[]" id="payment" multiple="multiple">
      <option value="Visa" selected="selected">Visa</option>
      <option value="Eurocard">Eurocard</option>
      <option value="Mastercard">Mastercard</option>
    </select>
// 可复选的下拉列表(选中值可以是一个数组)
<?php echo select_tag('payment', options_for_select(
  array('Visa' => 'Visa', 'Eurocard' => 'Eurocard', 'Mastercard' =>
'Mastercard'),
  array('Visa', 'Mastecard')
), 'multiple=multiple') ?>
 => <select name="payment" id="payment" multiple="multiple">
      <option value="Visa" selected="selected">
      <option value="Eurocard">Eurocard</option>
      <option value="Mastercard"
selected="selected">Mastercard</option>
    </select>

// 上传文件域 (Upload file field)
<?php echo input_file_tag('name') ?>
 => <input type="file" name="name" id="name" value="" />

// 密码输入框 (Password field)
<?php echo input_password_tag('name', 'value') ?>
 => <input type="password" name="name" id="name" value="value" />

// 隐藏域 (Password field)
<?php echo input_hidden_tag('name', 'value') ?>
 => <input type="hidden" name="name" id="name" value="value" />

// 提交按钮(文本格式) (Submit button (as text))
<?php echo submit_tag('Save') ?>
 => <input type="submit" name="submit" value="Save" />

// 提交按钮(图片格式) (Submit button (as image))
<?php echo submit_image_tag('submit_img') ?>
 => <input type="image" name="submit" src="/images/submit_img.png" />
```

submit_image_tag() 辅助函数使用的语法和 image_tag()相同，也具有相同的优点。

NOTE 对于单选按钮，id 属性没有默认地被设定为 name 属性的值，而是将 name 属性值和选项值混合后作为 id 属性的默认值。 之所以这样做，是为了实现“选中一个就自动去除另一个”的目的，你需要有多个同名的单选按钮标签，而根据前面 id=name 的约定，将导致在页面中出现多个含有同样 id 属性的 HTML 标签，这是被严格禁止的。

SIDEBAR 处理表单提交

如何取得用户通过表单提交的数据呢？这些数据存放在请求参数中，所以动作只要调用$this->getRequestParameter($elementName)就可以取得数据。

在同一个动作中既显示表单又处理表单是一种比较好的方法。对应不同的请求方法(GET 或 POST)，要么调用表单模板，要么处理表单并将请求重定向到另一个动作去。

```
// mymodule/actions/actions.class.php
public function executeEditAuthor()
{
  if ($this->getRequest()->getMethod() != sfRequest::POST)
  {
    // 显示表单
    return sfView::SUCCESS;
  }
  else
  {
    // 对提交的表单加以处理
    $name = $this->getRequestParameter('name');
    ...
    $this->redirect('mymodule/anotheraction');
  }
}
```

这段代码要能正常工作，表单处理和表单显示必须在同一个动作中。

```
// mymodule/templates/editAuthorSuccess.php

...
```

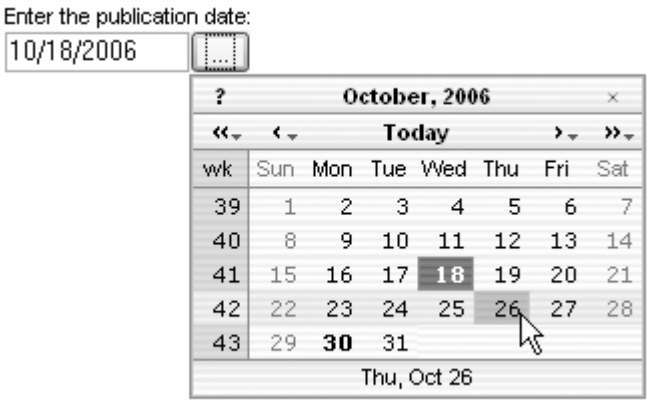
symfony 还专门设计了为后台处理异步请求的表单辅助函数。下一章有关 AJAX

的介绍将会提供更详细的信息。

日期输入控件

表单常用于输入日期，而日期格式错误常常是表单提交失败的主要原因。如果你将 rich 选项设定为 true，则 input_date_tag()辅助函数可以用一个交互式的 JavaScript 日历来帮助用户输入日期，见图 10-1 所示。

图 10-1 富日期输入标签



如果未设置 rich 选项，则辅助函数将会输出三个 select 标签，可取值为年、月、日的正常取值范围。你也可以通过调用三个辅助函数 select_day_tag()，select_month_tag()和 select_year_tag()，来分别显示下拉列表。这些元素的默认值是当前的年、月、日。例 10－3 演示了日期输入辅助函数的应用。

例 10－3 日期输入辅助函数

```
<?php echo input_date_tag('dateofbirth', '2005-05-03', 'rich=true') ?
>
 => 一个文本输入域加一个日期输入控件

// 以下辅助函数需要包括日期辅助函数组
<?php use_helper('Date') ?>
<?php echo select_day_tag('day', 1, 'include_custom=Choose a day') ?>
=> <select name="day" id="day">
      <option value="">Choose a day</option>
      <option value="1" selected="selected">01</option>
      <option value="2">02</option>
      ...
      <option value="31">31</option>
    </select>
```

```
<?php echo select_month_tag('month', 1, 'include_custom=Choose a
month use_short_month=true') ?>
=> <select name="month" id="month">
      <option value="">Choose a month</option>
      <option value="1" selected="selected">Jan</option>
      <option value="2">Feb</option>
      ...
      <option value="12">Dec</option>
    </select>

<?php echo select_year_tag('year', 2007, 'include_custom=Choose a
year year_end=2010') ?>
 => <select name="year" id="year">
      <option value="">Choose a year</option>
      <option value="2006">2006</option>
      <option value="2007" selected="selected">2007</option>
      ...
    </select>
```

input_date_tag() 辅助函数可接受的日期值是PHP 函数 strtotime()可以识别的值。 例 10－4 列出的格式是可以使用的， 而例 10－5 中的格式是严格禁止使用的。

例 10－4 日期辅助函数可以接受的日期格式

```
// 运行正常
<?php echo input_date_tag('test', '2006-04-01', 'rich=true') ?>
<?php echo input_date_tag('test', 1143884373, 'rich=true') ?>
<?php echo input_date_tag('test', 'now', 'rich=true') ?>
<?php echo input_date_tag('test', '23 October 2005', 'rich=true') ?>
<?php echo input_date_tag('test', 'next tuesday', 'rich=true') ?>
<?php echo input_date_tag('test', '1 week 2 days 4 hours 2 seconds',
'rich=true') ?>

// 返回空值
<?php echo input_date_tag('test', null, 'rich=true') ?>
<?php echo input_date_tag('test', '', 'rich=true') ?>
```

例 10－5 日期辅助函数中的错误日期格式

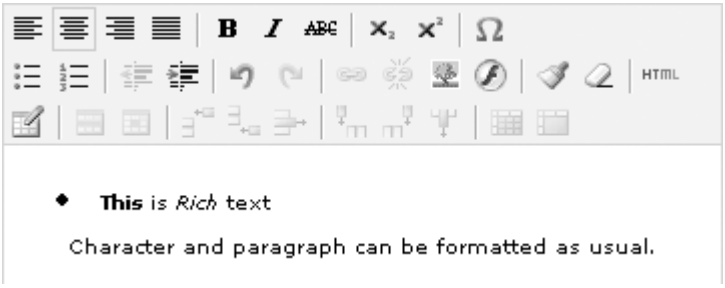
```
// 日期 0 = 01/01/1970
<?php echo input_date_tag('test', 0, 'rich=true') ?>
```

```
// 非英语日期格式不能正常运行
<?php echo input_date_tag('test', '01/04/2006', 'rich=true') ?>
```

编辑富文本（rich text）

因为集成了 TinyMCE 和 FCKEditor 插件，因而可以对<textarea>标签的文本进行富文本编辑，也就是说可以用类似于文字处理器界面的按钮将文本格式化为粗体，斜体或其它样式，见图 10－2。

图 10－2 编辑富文本



这两种插件都需要手工安装。因为安装方法相同，这里只介绍 TinyMCE 的安装方法。你可以从该项目的网站(http://tinymce.moxiecode.com/)下载它并将它解压到一个临时目录中，然后将 tinymce/jscripts/tiny_mce/目录复制到你的项目的 web/js/目录中，并在 settings.yml 中定义指向这个库的路径，如例 10－6 所示。

例 10－6 设置 TinyMCE 库路径

```
all:
  .settings:
    rich_text_js_dir:  js/tiny_mce
```

设置好后，再加入 rich=true 选项，就可以在文本框内进行富文本编辑了。你也可以用 tinymce_options 选项为 Javascript 编辑器设置定制的选项。参见例 10－7

例 10－7 富文本框

```
<?php echo textarea_tag('name', 'default content', 'rich=true
size=10x20')) ?>
 => 一个具有 TinyMCE 功能的富文本编辑区
<?php echo textarea_tag('name', 'default content', 'rich=true
size=10x20tinymce_options=language:"fr",theme_advanced_buttons2:"sepa
rator"')) ?>
```

=>一个具有定制过的 TinyMCE 功能的富文本编辑区

选择国家和语言

你可能会需要一个可以选择国家的域。因为在不同的语言中国家名也不相同，所以国家名下拉列表要根据用户的 culture 值来调整（第 13 章有 culture 的详细介绍）。如例 10－8 所示，select_country_tag()辅助函数可以完成所有工作，它会按照不同的语言显示国家名，并用 ISO 标准的国家代码作为选项的值。

例 10-8 选取国家名称辅助函数

```
<?php echo select_country_tag('country', 'AL') ?>
 => <select name="country" id="country">
      <option value="AF">Afghanistan</option>
      <option value="AL" selected="selected">Albania</option>
      <option value="DZ">Algeria</option>
      <option value="AS">American Samoa</option>
  ...
```

类似于 select_country_tag()辅助函数， select_language_tag()辅助函数可以显示一个语言名称列表，见例 10－9

例 10－9 选取语言名称辅助函数

```
<?php echo select_language_tag('language', 'en') ?>
 => <select name="language" id="language">
      ...
      <option value="elx">Elamite</option>
      <option value="en" selected="selected">English</option>
      <option value="enm">English, Middle (1100-1500)</option>
      <option value="ang">English, Old (ca.450-1100)</option>
      <option value="myv">Erzya</option>
      <option value="eo">Esperanto</option>
      ...
```

对象表单辅助函数

如果利用表单去编辑对象的属性，那么用标准的链接辅助函数去写代码非常费时。比如，要编辑 Customer 对象的 telephone 属性，应该写成：

```
<?php echo input_tag('telephone', $customer->getTelephone()) ?>
=> <input type="text" name="telephone" id="telephone"
```

value="0123456789" />

为了避免重复地写属性名，symfony 为每个表单辅助函数另外提供了一个对象表单辅助函数。对象表单辅助函数将从对象和方法名来获得表单元素的名字和默认值。前面用过的 input_tag()可以变换为如下形式：

```
<?php echo object_input_tag($customer, 'getTelephone') ?>
=> <input type="text" name="telephone" id="telephone"
value="0123456789" />
```

虽然 object_input_tag()看上去并不省事，但是，每个标准的表单辅助函数都有一个对应的对象表单辅助函数，并且它们的语法都一样，这样要生成表单就非常简单。这也是为什么在 symfony 生成的框架和后台管理都广泛使用了对象表单辅助函数的原因(详见第 14 章)。例 10-10 展示了对象表单辅助函数的用法。

例 10-10 对象表单辅助函数语法

```
<?php echo object_input_tag($object, $method, $options) ?>
<?php echo object_input_date_tag($object, $method, $options) ?>
<?php echo object_input_hidden_tag($object, $method, $options) ?>
<?php echo object_textarea_tag($object, $method, $options) ?>
<?php echo object_checkbox_tag($object, $method, $options) ?>
<?php echo object_select_tag($object, $method, $options) ?>
<?php echo object_select_country_tag($object, $method, $options) ?>
<?php echo object_select_language_tag($object, $method, $options) ?>
```

因为不宜替一个密码元素预先设定默认值，所以没有提供 object_password_tag() 辅助函数。

CAUTION 与一般的表单辅助函数不同，只有你在模板中用 use_helper('Object')明确声明你将使用对象辅助函数组后，才能使用对象表单辅助函数。

对象表单辅助函数中最有趣的就是 objects_for_select()和 object_select_tag()两个，它们都与下拉列表有关。

生成对象的下拉列表

用前面介绍过的 options_for_select() 辅助函数及其他标准辅助函数，可以将一个 PHP 关联数组转化为一个选项列表。见例 10-11 所示。

例 10-11 用 options_for_select()产生一个基于数组的选项列表。

```
<?php echo options_for_select(array(
  '1' => 'Steve',
  '2' => 'Bob',
  '3' => 'Albert',
  '4' => 'Ian',
  '5' => 'Buck'
), 4) ?>
 => <option value="1">Steve</option>
    <option value="2">Bob</option>
    <option value="3">Albert</option>
    <option value="4" selected="selected">Ian</option>
    <option value="5">Buck</option>
```

假设你已经有了一个从 Propel 查询得到的类 Author 的对象数组，如果你想建立一个基于这个数组的选项列表，你需要用一个循环遍历每个对象的 id 和 name 项，如例 10-12 所示。

例 10-12 用 options_for_select()产生一个基于对象数组的选项列表

```
// 动作
$options = array();
foreach ($authors as $author)
{
  $options[$author->getId()] = $author->getName();
}
$this->options = $options;

// 模板
<?php echo options_for_select($options, 4) ?>
```

因为经常需要进行这类处理，所以 symfony 提供了一个 objects_for_select()辅助函数来直接从一个对象数组创建一个选项列表。这个辅助函数需要两个另外的参数：用于遍历值的方法名和要生成的标签的文本内容。例 10－12 可以简化为如下形式：

```
<?php echo objects_for_select($authors, 'getId', 'getName', 4) ?>
```

这样已经既快又好，但是当你要处理外键列时，symfony 还提供了更多的方便。

创建一个基于外键列的下拉列表

外键列可取的值是外键所在表的记录的主键值。例如，如果 article 表有一个

author_id 列，它是 author 表的外键， 那么这个列的可能值就是 author 表的所有 id 列的值。通常，用于编辑一篇文章的作者的下拉列表看上去如例 10－13 所示。

例 10－13 用 objects_for_select()创建一个基于外键的选项列表

```
<?php echo select_tag('author_id', objects_for_select(
  AuthorPeer::doSelect(new Criteria()),
  'getId',
  '__toString()',
  $article->getAuthorId()
)) ?>
=> <select name="author_id" id="author_id">
      <option value="1">Steve</option>
      <option value="2">Bob</option>
      <option value="3">Albert</option>
      <option value="4" selected="selected">Ian</option>
      <option value="5">Buck</option>
    </select>
```

实际上，仅仅用 object_select_tag()就可以完成所有的工作。它显示一个下拉列表，其选项为外键所在表的所有可能记录值。这个辅助函数可以根据数据库模式文件猜出外键列和外键所在的表，因此它的语法非常简洁。以下代码和例 10－13 完成同样的功能：

```
<?php echo object_select_tag($article, 'getAuthorId') ?>
```

object_select_tag() 辅助函数根据作为参数传递的方法名就可以得到相关的 peer 类名（在本例中就是 AuthorPeer）。但是，你可以在第 3 个参数 relate_class 中设定你自己的类名。<option>标签的文本内容是根据类的 _toString()方法得到的记录名（如果$author->_toString()方法没有定义，就用主键代替）。另外，选项列表是从带有空条件的 doSelect()方法运行得出的结果中取得的，它根据记录的创建时间排序。如果你想按指定的顺序显示记录的一个子集，你可以在 peer 类中创建一个方法，该方法返回满足条件的对象数组，然后你就可以设定 peer_method 选项。最后，你可以通过设定 include_blank 或 include_custom 选项，在下拉列表的顶部加一个空的选项或某个定制的选项。例 10－14 显示了 object_select_tag()辅助函数的各种选项的用法。

例 10－14 object_select_tag() 辅助函数的选项

```
// 基本语法
```

```
<?php echo object_select_tag($article, 'getAuthorId') ?>
// 用 AuthorPeer::doSelect(new Criteria()) 生成列表

// 修改用来获取可选值的peer类
<?php echo object_select_tag($article, 'getAuthorId',
'related_class=Foobar') ?>
// 用_FoobarPeer::doSelect(new Criteria())_生成列表

// 修改获取可选值的peer方法
<?php echo object_select_tag($article,
'getAuthorId','peer_method=getMostFamousAuthors') ?>
// 用_AuthorPeer::getMostFamousAuthors(new Criteria())_生成列表

// 在列表的最上面增加<option value=""> </option>
<?php echo object_select_tag($article, 'getAuthorId',
'include_blank=true') ?>

// 在列表最上面增加<option value="">Choose an author</option>
<?php echo object_select_tag($article, 'getAuthorId',
  'include_custom=Choose an author') ?>
```

更新对象

如果要编写专用于编辑对象属性的表单，那么利用对象辅助函数就比在动作中编写代码容易。例如，如果你有一个类Author包含name，age，address等属性，你就可以如10－15那样编写表单：

例10－15 仅用对象辅助函数的表单

```
<?php echo form_tag('author/update') ?>
  <?php echo object_input_hidden_tag($author, 'getId') ?>
  Name: <?php echo object_input_tag($author, 'getName') ?><br />
  Age:  <?php echo object_input_tag($author, 'getAge') ?><br />
  Address: <br />
         <?php echo object_textarea_tag($author, 'getAddress') ?>
</form>
```

提交表单时，将调用author模块的update动作，该动作只要调用由Propel生成的fromArray()方法就可以更新对象，参见例10－16。

例10－16 基于对象表单辅助函数的表单提交处理函数。

```
public function executeUpdate ()
```

```
{
  $author = AuthorPeer::retrieveByPk($this->getRequestParameter('id'));
  $this->forward404Unless($author);

  $author->fromArray($this->getRequest()->getParameterHolder()->getAll(),AuthorPeer::TYPE_FIELDNAME);
  $author->save();

  return $this->redirect('/author/show?id='.$author->getId());
}
```

表单验证

第 6 章介绍过如何在动作类中用 validateXXX()方法验证请求参数。但是，如果你用这种方法去验证表单提交的话，你就会没完没了地写同样的代码。symfony 提供了一种技术，不用动作类中的 PHP 代码，而仅仅是用一个 YAML 文件就可以验证表单。

为了说明表单验证，让我们先看一下例 10-17 中的表单示例。这是一个典型的联系人表单，包括 name、email、age 和 message 域。

例 10-17 联系人表单示例。文件路径为 modules/contact/templates/indexSuccess.php

```
<?php echo form_tag('contact/send') ?>
  Name:    <?php echo input_tag('name') ?><br />
  Email:   <?php echo input_tag('email') ?><br />
  Age:     <?php echo input_tag('age') ?><br />
  Message: <?php echo textarea_tag('message') ?><br />
  <?php echo submit_tag() ?>
</form>
```

表单验证的要求就是：如果用户提交的表单中包含了无效数据，浏览器将在页面上显示出错信息。我们先对上述表单定义什么样的数据才是有效的。

- name 域是必需的，且必须是 2 到 100 个字符的文本。
- email 域是必需的，且必须是 2 到 100 个字符的文本，并是一个有效的电子邮件地址。
- age 域是必需的，且必须是 0-100 之间的整数。
- message 域是必需的。

你当然可以为联系人表单定义更复杂的验证规则，不过在本例中以上规则就足以说明问题了。

NOTE 表单验证可以在服务器端进行，也可以在客户端进行。为了防止错误数据破坏数据库，服务器端的验证是必需的。尽管客户端的验证可以极大地提高用户体验，但客户端的验证却不是必需的。客户端验证通常用 JavaScript 定制。

验证器

你可以看到例子中的 name 和 email 域使用同样的验证规则。有些验证规则在 web 表单中频繁出现，为此，symfony 将这些规则的 PHP 代码打包成验证器。一个验证器是一个提供了 execute()方法的简单类。这个方法以域的值为参数，如果值有效则返回 true，否则返回 false。

symfony 包含多个验证器（本章后面的“标准 symfony 验证器”一节将详细介绍），这里我们先重点说明一下 sfStringValidator。该验证器检测输入值是否为一个字符串，且字符数在两个指定的值之间（通过调用 initialize()方法定义）。 这正是我们验证 name 域时所需要的。例 10-18 显示了如何在一个验证方法中使用这个验证器。

例 10-18 用可复用的验证器验证请求参数，文件路径为 modules/contact/action/actions.class.php

```
public function validateSend()
{
  $name = $this->getRequestParameter('name');

  // name 域必需有值
  if (!$name)
  {
    $this->getRequest()->setError('name', 'The name field cannot be
left blank');

    return false;
  }

  // name 域的值必需是长度在 2-100 之间的字符串
  $myValidator = new sfStringValidator();
  $myValidator->initialize($this->getContext(), array(
    'min'       => 2,
    'min_error' => 'This name is too short (2 characters minimum)',
    'max'       => 100,
    'max_error' => 'This name is too long. (100 characters maximum)',
  ));
  if (!$myValidator->execute($name))
  {
    return false;
```

```
  }

  return true;
}
```

如果用户在例 10-17 中的表单的 name 域里输入值 a，则 sfStringValidator 的 execute()方法将返回 false （因为字符串长度小于 2），因而 validateSend()方法也返回 false。这样，executeSend()方法将不执行，而是执行 handleErrorSend()方法。

TIP sfRequest 对象的 setError()方法将要显示的出错信息提供给模板(本章后面的“显示表单的出错信息”一节将会详细说明)。验证器在内部设定了错误信息，所以你可以为不能通过验证的不同情形定义不同的错误信息。这正是 sfStringValidator 中的初始化参数 min_error 和 max_error 的作用。

本例中定义的所有规则都可以用验证器替代：

- name: sfStringValidator (min=2, max=100)
- email: sfStringValidator (min=2, max=100) 和 sfEmailValidator
- age: sfNumberValidator (min=0, max=120)

不过“域是必需的”规则却不是由验证器处理的。

验证文件

虽然你可以在 validateSend()方法中用验证器轻易地实现你的联系人表单验证，但这也意味着你将编写大量重复的代码。symfony 用另一种方法来定义表单的验证规则，这就是用 YAML 文件。例 10-19 给出了 name 域的验证规则的替换表达方法，其验证结果与例 10-18 得到的相同。

例 10-19 验证文件，路径为 modules/contact/validate/send.yml

```
fields:
  name:
    required:
      msg:       The name field cannot be left blank
    sfStringValidator:
      min:       2
      min_error: This name is too short (2 characters minimum)
      max:       100
      max_error: This name is too long. (100 characters maximum)
```

在一个验证文件里，fields 头列出了所有需要验证的域，并且如果有值时，验证器必须对该值进行检验。每个验证器的参数和你手工初始化使用的参数相同。只要需要，一个域可以被多个验证器验证。

NOTE 一个验证器验证失败并不会导致整个验证过程结束。symfony 会检验所有的验证器，只有至少一个验证失败时，才会宣布整个验证失败。并且即使验证文件中的某些验证规则失败，symfony 仍旧会找一个 validateXXX()方法去执行。所以两种验证技术是互补的。好处就是对于有多个错误的表单，可以揭示出所有的错误信息。

验证文件在模块的 validate 目录下，并用要验证的动作名称来命名。例如，例 10-19 的文件路径就是 validate/send.yml。

重新显示表单

只要验证失败，symfony 自动在动作类中找 handleErrorSend()方法，如果没有这个方法，就去显示 sendError.php 模板。

不过，告诉用户未能通过验证的更好方法却是再显示一遍包含出错信息的表单。为此，你需要重载 handleErrorSend()方法，并且重定向到显示表单的动作去（上例中，应该是 module/index），参见例 10-20。

例 10-20 再次显示表单，文件路径为 modules/contact/actions/actions.class.php

```
class ContactActions extends sfActions
{
  public function executeIndex()
  {
    // 显示表单
  }

  public function handleErrorSend()
  {
    $this->forward('contact', 'index');
  }

  public function executeSend()
  {
    // 处理表单提交
  }
}
```

如果你用同一个动作显示表单和处理表单提交，只要让 handlErrorSend()方法简单地返回 sfView::SUCCESS，就可以从 sendSuccess.php 重新显示表单，参见例 10-21。

例 10-21 一个动作同时显示和处理表单，文件路径为

modules/contact/actions/actions.class.php

```
class ContactActions extends sfActions
{
  public function executeSend()
  {
    if ($this->getRequest()->getMethod() != sfRequest::POST)
    {
      // 为模板准备数据

      // 显示表单
      return sfView::SUCCESS;
    }
    else
    {
      // 处理表单提交
      ...
      $this->redirect('mymodule/anotheraction');
    }
  }
  public function handleErrorSend()
  {
     // 为模板准备数据

      // 显示表单

    return sfView::SUCCESS;
  }
}
```

准备数据的代码可以分离到动作类一个的保护方法中，以免在 executeSend() 和 handleErrorSend()方法中重复该代码。

经过这样修改后，如果用户输入一个无效的名字，表单就会重新显示，但是输入的数据没有了,而且解释错误原因的出错信息也没有显示。为解决这个问题，你必须修改显示表单的模板，以便将出错信息显示在校验出错域的旁边。

在表单中显示出错信息

当一个域验证失败时，出错信息会作为一个验证器参数传送给请求（就像在例 10-18 中你用 setError()手工添加错误一样）。 sfRequest 对象提供两个有用的方法用于查看错误信息：hasError()和 getError()，它们都以域名为参数。另外，你可以借助 hasErrors()在表单的顶部显示一个警告信息，以引起用户注意有一个或多个域输入无效。例 10-22 和 10-23 给出了如何使用这些方法的

例子。

例 10-22 在表单顶部显示错误信息，文件路径 templates/indexSuccess.php

```
<?php if ($sf_request->hasErrors()): ?>
  <p>The data you entered seems to be incorrect.
  Please correct the following errors and resubmit:</p>
  <ul>
  <?php foreach($sf_request->getErrors() as $name => $error): ?>
    <li><?php echo $name ?>: <?php echo $error ?></li>
  <?php endforeach; ?>
  </ul>
<?php endif; ?>
```

例 10-23 在表单内显示错误信息，文件路径 templates/indexSuccess.php

```
<?php echo form_tag('contact/send') ?>
  <?php if ($sf_request->hasError('name')): ?>
    <?php echo $sf_request->getError('name') ?> <br />
  <?php endif; ?>
  Name:    <?php echo input_tag('name') ?><br />
  ...
  <?php echo submit_tag() ?>
</form>
```

例 10-23 中包含 getError()的条件句写起来有点长，所以假如你预先声明了 Validation 辅助函数组，symfony 提供了一个 form_error()辅助函数来替代这些长的代码。例 10-24 用这个辅助函数替换了例 10-23 的代码。

例 10-24 用缩写方法在表单中显示出错信息

```
<?php use_helper('Validation') ?>
<?php echo form_tag('contact/send') ?>

          <?php echo form_error('name') ?><br />
  Name:    <?php echo input_tag('name') ?><br />
  ...
  <?php echo submit_tag() ?>
</form>
```

form_error()辅助函数在每个出错信息的前后分别加了一个特殊字符，以使出错信息更醒目。这个字符的默认值是向下的箭头（对应于&darr），你可以在

settings.yml 文件里改变这个默认值：

```
all:
  .settings:
    validation_error_prefix:    ' &darr; '
    validation_error_suffix:    '  &darr;'
```

现在，当验证失败时，表单可以正确地显示出错信息了，但是用户之前输入的数据都没有了。你应该将这些数据重新放入表单中，以便提供更好用户体验。

重新填充表单数据

如例 10－20 所示，当一个错误通过 forward()方法处理后，原来的请求数据仍是可以读取的，而且用户输入的数据保存在请求参数中。所以你可以将默认值加入到每个域中去为表单填充数据，见例 10－25。

例 10－25 验证失败后， 用默认值为表单重新填充数据， 文件路径为 templates/indexSuccess.php

```
<?php use_helper('Validation') ?>
<?php echo form_tag('contact/send') ?>
          <?php echo form_error('name') ?><br />
  Name:   <?php echo input_tag('name', $sf_params->get('name')) ?
><br />
          <?php echo form_error('email') ?><br />
  Email:  <?php echo input_tag('email', $sf_params->get('email')) ?
><br />
          <?php echo form_error('age') ?><br />
  Age:    <?php echo input_tag('age', $sf_params->get('age')) ?
><br />
          <?php echo form_error('message') ?><br />
  Message: <?php echo textarea_tag('message', $sf_params-
>get('message')) ?><br />
  <?php echo submit_tag() ?>
</form>
```

当然写这些代码又是件很麻烦的事，所以 symfony 再次提供了一种方法让你不用改变元素的默认值， 就可以重新为表单的所有域填充数据，这个方法就是直接在 YAML 验证文件中设置表单的 fillin 属性，参见例 10－26。

例 10－26 激活 fillin 属性， 以便在验证失败时为表单重新填充数据，文件路径为 validate/send.yml

```
fillin:
```

```
  enabled: true  # 允许重填表单
  param:
    name: test  # 表单名,如果页面中只有一个表单,可以忽略
    skip_fields:  [email]  # 不重填这些域
    exclude_types: [hidden, password] # 不重填这些类型的域
    check_types:  [text, checkbox, radio, password, hidden] # 重填这
些类型的域
```

自动重新填充适用于输入框、复选框、单选按钮、文本框和下拉列表（包括简单列表和多重列表），但不适用于密码和隐藏域，也不适用于文件标签。

NOTE fillin 属性是在发送响应内容给用户之前，通过编译用 XML 表示的响应内容来工作的，所以如果响应内容不是一个有效的 XHTML 文档，fillin 就不能工作。

你有可能想在写回表单域之前对用户输入的值进行转换。只要你在 converters 属性下定义了可以用函数调用的转换，包括转义，URL 重写，特殊字符转换等，就可以将这些转换作用到表单的域上去。

例 10－27 在 fillin 之前转换输入， 文件路径是 validate/send.yml

```
fillin:
  enabled: true
  param:
    name: test
    converters:          # 要实施的转换
      htmlentities:     [first_name, comments]
      htmlspecialchars: [comments]
```

标准 symfony 验证器

symfony 为你的表单提供了下列标准的验证器

- sfStringValidator
- sfNumberValidator
- sfEmailValidator
- sfUrlValidator
- sfRegexValidator
- sfCompareValidator
- sfPropelUniqueValidator
- sfFileValidator
- sfCallbackValidator

每个标准验证器都有一组默认的参数和错误信息，这些值和信息可以轻松地通过 initialize()方法或 YAML 文件来重新设置。 下面几节内容将描述这些验证

器并给出示例。

字符串验证器

sfStringValidator 对参数进行与字符串有关的验证。

```
sfStringValidator:
  values:       [foo, bar]
  values_error: The only accepted values are foo and bar
  insensitive:  false  # If true, comparison with values is case
insensitive
  min:          2
  min_error:    Please enter at least 2 characters
  max:          100
  max_error:    Please enter less than 100 characters
```

数字验证器

如果参数是一个数值， 你可以用 sfNumberValidator 来验证数的大小。

```
sfNumberValidator:
  nan_error:    Please enter an integer
  min:          0
  min_error:    The value must be more than zero
  max:          100
  max_error:    The value must be less than 100
```

E-Mail 验证器

sfEmailValidator 用于验证参数值是否符合电子邮件地址的格式标准。

```
sfEmailValidator:
  strict:       true
  email_error:  This email address is invalid
```

虽然 RFC822 定义了电子邮件的地址格式，但通常认可的电子邮件地址格式要比这个标准更加严格。比如，RFC 认为 me@localhost 是有效的电子邮件地址，而你却可能不接受。如果你将 strict 参数设定为 true（这是默认值）， 那么只有符合 name@domain.extension 格式的电子邮件地址才能通过验证；如果将该参数设定为 false，则采用 RFC 的规则。

URL 验证器

sfUrlValidator 用于验证一个域的值是否是一个有效的 URL。

```
sfUrlValidator:
```

```
  url_error:    This URL is invalid
```

正则表达式验证器

sfRegexValidator 验证一个值是否与一个 Perl 兼容正则表达式模式相匹配。

```
sfRegexValidator:
  match:        No
  match_error:  Posts containing more than one URL are considered as
spam
  pattern:      /http.*http/si
```

其中的 match 参数为 Yes 时，表明请求参数与模式匹配为有效；如果为 No，则表明请求参数与模式匹配为无效。

比较验证器

sfCompareValidator 用于检测两个不同的请求参数是否相同。 这对于密码检测非常有用。

```
fields:
  password1:
    required:
      msg:      Please enter a password
  password2:
    required:
      msg:      Please retype the password
    sfCompareValidator:
      check:    password1
      compare_error: The two passwords do not match
```

check 参数是一个域的名字，当前域必须与该域匹配才有效。

Propel 唯一性验证器

sfPropelUniqueValidator 检测一个请求参数的值是否在数据库中已经存在。这对于唯一索引非常有用。

```
fields:
  nickname:
    sfPropelUniqueValidator:
      class:        User
      column:       login
      unique_error: This login already exists. Please choose another
one.
```

本例中，验证器将在数据库中查找类User的记录，以确定是否有和域值相同的login列。

文件验证器

sfFileValidator 用于检测文件上传域的文件格式（一组mime类型）和大小。

```
fields:
  image:
    required:
      msg:      Please upload an image file
    file:       True
    sfFileValidator:
      mime_types:
        - 'image/jpeg'
        - 'image/png'
        - 'image/x-png'
        - 'image/pjpeg'
      mime_types_error: Only PNG and JPEG images are allowed
      max_size:         512000
      max_size_error:   Max size is 512Kb
```

注意file属性必须设置为True，并且模式中的表单也必须是multipart的。

回调验证器

sfCallbackValidator 可以利用第三方的可调用方法或函数进行验证， 该可调用方法或函数必须能返回true或false值。

```
fields:
  account_number:
    sfCallbackValidator:
      callback:      is_integer
      invalid_error: Please enter a number.
  credit_card_number:
    sfCallbackValidator:
      callback:      [myTools, validateCreditCard]
      invalid_error: Please enter a valid credit card number.
```

回调方法或函数的第一个参数是需要验证的值。 当你想要重用现成的方法或函数时非常有用， 可以避免重建一个完整的验证器类。

TIP 你也可以编写你自己的验证器，本章后面的“创建定制的验证器”将具体描述。

具名验证器

如果你预计会重复使用一个验证器类以及有关设置，你可以将它们打包成一个具名验证器。在联系人表单的例子中，email 域要用到和 name 域相同的 sfStringValidator 参数，这样你就可以创建一个 myStringValidator 具名验证器，以免将所有设置重复一遍。你可以在 validators 头下增加一个 myStringValidator 标签，并将 class 和 param 属性设置为你需要的参数值。然后你就可以在 fields 中象使用标准验证器一样使用该具名验证器。参见例 10－28。

例 10－28 在验证文件中重用具名验证器， 文件路径为 validate/send.yml

```
validators:
  myStringValidator:
    class: sfStringValidator
    param:
      min:       2
      min_error: This field is too short (2 characters minimum)
      max:       100
      max_error: This field is too long (100 characters maximum)

fields:
  name:
    required:
      msg:       The name field cannot be left blank
    myStringValidator:
  email:
    required:
      msg:       The email field cannot be left blank
    myStringValidator:
    sfEmailValidator:
      email_error:  This email address is invalid
```

重新指定验证方法

默认情况下， 只要一个动作伴随着 POST 方式被调用， 则验证文件中设定的验证器都会执行。为了对不同的方式（POST 或 GET）设定不同的验证，你可以为 methods 属性指定其他的值，既可以是全局范围的也可以是针对每个域重新设置调用验证的方式（POST 或 GET）。 参见例 10－29 所示。

例 10－29 在 validate/send.yml 中定义何时测试一个域

```
methods:          [post]     # 这是默认值

fields:
```

```
  name:
    required:
      msg:       The name field cannot be left blank
    myStringValidator:
  email:
    methods:     [post, get] # 重新指定全局方式
    required:
      msg:       The email field cannot be left blank
    myStringValidator:
    sfEmailValidator:
      email_error:  This email address is invalid
```

验证文件全貌

到现在为止，你只是见到了验证文件的若干片段。等你将这些全部合到一起，你就知道如何用 YAML 清晰地描述验证规则了。 例 10－30 是联系人表单的完整验证文件，它综合了前面定义的所有规则。

例 10－30 完整的验证文件示例

```
fillin:
  enabled:      true

validators:
  myStringValidator:
    class: sfStringValidator
    param:
      min:       2
      min_error: This field is too short (2 characters minimum)
      max:       100
      max_error: This field is too long (100 characters maximum)

fields:
  name:
    required:
      msg:       The name field cannot be left blank
    myStringValidator:
  email:
    required:
      msg:       The email field cannot be left blank
    myStringValidator:
    sfEmailValidator:
      email_error:  This email address is invalid
  age:
    sfNumberValidator
```

```
      nan_error:    Please enter an integer
      min:          0
      min_error:    "You're not even born. How do you want to send a
message?"
      max:          120
      max_error:    "Hey, grandma, aren't you too old to surf on the
Internet?"
  message:
    required:
      msg:          The message field cannot be left blank
```

复杂的验证

利用验证文件已经能够满足大多数验证方面的需求，但是当验证非常复杂时，仅仅使用验证文件就不能满足需要了。这时，你既可以自己编写动作的validateXXX()方法，也可以从下面叙述的方法中找到解决方案。

创建一个定制的验证器

每个验证器都是一个继承了sfValidator类的子类。如果symfony自带的验证器类不符合你的需要，你可以在任何一个可以自动加载的lib/目录中创建一个新的验证器类。语法相当简单：执行验证器就是执行类的execute()方法。你还可以在initialize()方法中定义默认设置。

execute()方法对第一个参数的值进行验证，然后输出第二个参数的值作为出错信息。这两个参数都以引用方式传送，所以你可以在方法内部修改出错信息的内容。

initialize()方法以上下文（context singleton）和YAML文件中的参数数组为参数。定义initialize()时，必须先调用父类sfValidator的initialize()方法，然后才能设置默认值。

每个验证器都有一个参数存取器，它可以用$this->getParameterHolder()存取。

例如，如果你想创建一个sfSpamValidator检测一个字符串是否是一个垃圾信息串，可以在sfSpamValidator.class.php中加入如例10－31那样的代码，它可以检测出$values中出现字符串http的次数是否超过由max_url属性定义的次数。

例10－31 创建一个定制的验证器lib/sfSpamValidator.class.php

```
class sfSpamValidator extends sfValidator
{
  public function execute (&$value, &$error)
```

```
  {
    // 当max_url=2时, regexp 是 /http.*http/is
    $re = '/'.implode('.*', array_fill(0, $this-
>getParameter('max_url') + 1, 'http')).'/is';

    if (preg_match($re, $value))
    {
      $error = $this->getParameter('spam_error');

      return false;
    }

    return true;
  }

  public function initialize ($context, $parameters = null)
  {
    // 初始化父类
    parent::initialize($context);

    // 设定参数默认值
    $this->setParameter('max_url', 2);
    $this->setParameter('spam_error', 'This is spam');

    // 设置参数
    $this->getParameterHolder()->add($parameters);

    return true;
  }
}
```

一旦将验证器加入可自动加载的目录中（需清除缓存），你就可以在你的验证文件中使用它，见例10－32所示。

例10－32 在validate/send.yml中使用定制的验证器

```
fields:
  message:
    required:
      msg:          The message field cannot be left blank
    sfSpamValidator:
      max_url:      3
      spam_error:   Leave this site immediately, you filthy spammer!
```

用数组表示表单域

在 PHP 中，你可以将数组用于表单域。当你编写你自己的表单或用由 Propel 后台管理模块生成的表单（参见第 14 章）时，你可以利用例 10－33 中所示的代码。

例 10－33 使用数组的表单

```
<label for="story[title]">Title:</label>
<input type="text" name="story[title]" id="story[title]"
value="default value"
       size="45" />
```

在验证文件中使用一个带有方括号的输入名会导致编译错误。解决的方法是在验证文件的 fields 段中用花括号{}替代方括号[]，symfony 会自行转换名字后再传送给验证器。参见例 10－34。

例 10－34 使用数组的表单的验证文件

```
fields:
  story{title}:
    required:     Yes
```

验证空域

一个域不一定有值，但是你却可能需要验证这个域是否有一个空值。比如说，在一个表单中有一个密码域，用户可以不改变密码；也可以重设密码，这时，用户还必须输入一个确认密码。参见例 10－35 所示。

例 10－35 具有两个密码域的表单的验证文件

```
fields:
  password1:
  password2:
    sfCompareValidator:
      check:         password1
      compare_error: The two passwords do not match
```

该验证过程按如下方式处理：

- 如果 password1 和 password2 都为空值：
  - 值存在测试通过。
  - 不运行验证器。
  - 表单有效。
- 如果 password2 为空， 而 password1 不为空：
  - 值存在测试通过。

    - 不运行验证器。
    - 表单有效。

你可能希望在 password1 非空时能运行你的 password2 验证器。利用 group 参数， symfony 验证器可以处理这种情形。当一个域在一个组中时，如果这个域不为空且同一个组中的任一个域不为空，这个域的验证器就会执行。

所以，如果你像例 10－36 那样改变你的配置，验证过程就能正确执行。

例 10－36 带有两个密码域和一个组的表单的验证文件

```
fields:c
  password1:
    group:            password_group
  password2:
    group:            password_group
    sfCompareValidator:
      check:          password1
      compare_error: The two passwords do not match
```

现在验证器按下述方式执行：

- 如果 password1 和 password2 都为空：
    - 值存在测试通过。
    - 验证器未执行。
    - 表单有效。
- 如果 password1 为空而 password2 为 foo：
    - 值存在测试通过。
    - 因为 password2 不为空， 所以将运行验证器， 验证失败。
    - password2 的验证将抛出一个出错信息。
- 如果 password1 为 foo 而 password2 为空：
    - 值存在测试通过。
    - 同理，因为 password1 不为空，所以同一个组中的 password2 的验证器将运行，且验证失败。
    - password2 的验证将抛出一个出错信息。
- 如果 password1 和 password2 都为 foo：
    - 值存在测试通过。
    - 因为 password2 不为空，所以将执行验证器，验证成功。
    - 表单有效。

总结

有了标准表单辅助函数及其灵活的选项，编写表单将非常方便。如果你要设计一个可以编辑对象属性的表单时，对象表单辅助函数将会提供更大的帮助。借

助于验证文件，验证辅助函数和重填特性，在表单域上编写一个强壮且用户友好的服务器控件所需的工作量将大大减少。而要满足最复杂的验证需求，只需写一个定制的验证器或在动作类中创建一个 validateXXX()方法。