第 13 章 国际化（I18n）与本地化（L10n）

如果你曾经开发过国际化的应用，你一定知道要处理好文本翻译，地方标准和本地化内容是极为困难的事。但是，symfony 能够自动处理与国际化有关的各种问题。

因为“国际化”这个词的英文名字太长（internationalization），所以开发者们通常将它简写为 i18n，你只要数一下这个英文词的字母个数，你就会知道为什么会有这么一个奇怪的缩写了。同样，“本地化”（localization）被简写为 l10n。这两个概念涵盖了多语种互联网应用的两个不同的方面。

一个国际化应用程序通常包括多个版本，它们的内容相同而语言或格式不同。例如，一个电子邮件系统就可以用多种语言提供同一种服务，仅仅是界面不同而已。

而一个本地化应用则会根据用户浏览的地区不同而显示不同的信息。如果你浏览一个新闻网站，那么在美国访问这个网站，页面将显示与美国有关的最新话题，而在法国访问这个网站，页面就应该显示在法国发生的最新事件。所以说，l10n 不仅要翻译页面内容，而且要根据不同的本地化版本提供不同的内容。

总之，i18n 和 l10n 将会处理以下问题：

- 文本翻译（包括界面，资源和内容）
- 标准和格式（包括日期，数量，数字等等）
- 本地化内容（根据不同地区，给出某个内容的不同版本）

本章将介绍 symfony 处理这些问题的方法以及如何运用这些方法来开发国际化和本地化的应用系统。

用户的国家和语言（Uesr Culture）

一个称为 culture 的用户会话参数决定了 symfony 中内置的 i18n 特性。这个 culture 参数由用户所在的国家和语言组成，它确定如何显示文本以及与 culture 有关的信息。 因为 culture 参数被序列化保存在用户会话中，所以它在页面之间是持久有效的。

设定默认的国家和语言

默认情况下，新用户的 culture 是 default_culture。可以在 i18n.yml 配置文件中改变它的设定，参见例 13－1 所示。

例 13－1 在 myapp/config/i18n.yml 中设定默认 culture。

```
all:
  default_culture:     fr_FR
```

NOTE 你也许会奇怪为什么虽然在开发过程中改变了 i18n.yml 中的值，浏览器中显示的内容却没有相应变化。这是因为浏览器从前面的页面中保留了先前的 culture 值。如果你想看到新 culture 值带来的变化，你需要清除 cookie 或重启你的浏览器。

因为来自法国、比利时或加拿大的用户使用不同的法语翻译；而西班牙或墨西哥的用户也使用不同的西班牙语内容，所以 culture 参数是由语言和国家两个参数值组成的。语言参数根据 ISO 639-1 标准，用两个小写字母表示，例如，英语用 en 表示。而国家参数根据 ISO 3166-1 标准，用两个大写字母表示，例如，英国用 GB 表示。

改变用户的国家和语言

在与浏览器的会话过程中，用户的 culture 值是可以被改变的。例如，当用户决定从英文版切换到法文版，或当用户登录时要使用自己预设的语言的时候。这就是为什么 sfUser 类提供了存取用户 culture 值的获取方法和设置方法的原因。例 13－2 显示了如何在一个动作中运用这些方法。

例 13－2 在动作中设定和获取 culture 值

```
// 设定 culture
$this->getUser()->setCulture('en_US');

// 获取 culture
$culture = $this->getUser()->getCulture();
 => en_US
```

SIDEBAR URL 里的 culture

当你运用 symfony 的国际化和本地化特性时，一个 URL 的页面就可以有多个不同的版本——这取决于用户的会话，而这样会使你不能在搜索引擎中缓存或索引你的页面。

解决这个问题的一种方法是在每个 URL 中加上 culture 值，这样，被翻译过的页面就会被看成是不同的 URL。为此，你可以在应用的 routing.yml 中为每个规则加上 sf_culture 标记：

```
page:
  url: /:sf_culture/:page
```

```
  requirements: { sf_culture: (?:fr|en|de) }
  params: ...

article:
  url: /:sf_culture/:year/:month/:day/:slug
  requirements: { sf_culture: (?:fr|en|de) }
  params: ...
```

为了避免在每个 link_to()中手工加 sf_culture 参数，symfony 会将用户 culture 自动加到默认的路由参数中。反之，如果在 URL 中出现 sf_culture 参数，symfony 也会自动改变用户 culture 的值。

自动确定用户的国家和语言

在许多应用中，用户的国家和语言是根据浏览器的设定在第一次发出请求时确定的。用户可以在浏览器中定义一组可接受的语言，每当浏览器向服务器发出一个请求，这组语言数据就会被放在 HTTP 头的 Accept-Language 参数中。你可以通过 symfony 的 sfRequest 对象来获取它。例如，要获得一个动作中用户的预设语言列表，可以用以下方法：

```
$languages = $this->getRequest()->getLanguages();
```

HTTP 头是一个字符串，但 symfony 自动将它转换为一个数组。所以可以用 $languages[0]取得上面例子中的用户预设语言。

在网站首页或在每个页面的过滤器中，自动将浏览器语言设定为用户的国家和语言是比较有用的方法。

CAUTION HTTP 头的 Accept-Language 值并不是很可靠的，因为很少有用户知道如何改变浏览器中的语言值。大多数时候，预设的浏览器语言是界面的语言，而浏览器并不是在所有语言下都是可用的。如果你根据浏览器预设语言自动设定 culture 值，最好可以提供一个让用户选择语言的方法。

标准与格式

Web 应用内部是不考虑国家和语言因素的。比如说数据库，它使用的日期、数量等数据都是按国际标准存放的。但是，当用户浏览数据时，就需要进行格式转换。用户不可能理解时间戳的概念，而且对于法国人来说，他们更愿意将自己的母语称为 Français 而不是 French。所以你需要根据用户 culture 值，对格式进行自动转换。

根据用户 culture 值输出数据

一旦定义了 culture 值，辅助函数就会据此自动进行适当的输出。例如，

format_number()辅助函数就会根据用户的culture值，自动将数字以用户熟悉的格式输出。参见例13－3。

例13－3 根据用户culture值显示数字

```
<?php use_helper('Number') ?>

<?php $sf_user->setCulture('en_US') ?>
<?php echo format_number(12000.10) ?>
 => '12,000.10'

<?php $sf_user->setCulture('fr_FR') ?>
<?php echo format_number(12000.10) ?>
 => '12 000,10'
```

你无需明确地将culture值传递给辅助函数。辅助函数会在当前的会话对象中自动找到culture值。例13－4中列出的辅助函数在输出数据时都会考虑用户的culture值。

例13－4 与用户culture有关的辅助函数。

```
<?php use_helper('Date') ?>

<?php echo format_date(time()) ?>
 => '9/14/06'

<?php echo format_datetime(time()) ?>
 => 'September 14, 2006 6:11:07 PM CEST'

<?php use_helper('Number') ?>

<?php echo format_number(12000.10) ?>
 => '12,000.10'

<?php echo format_currency(1350, 'USD') ?>
 => '$1,350.00'

<?php use_helper('I18N') ?>

<?php echo format_country('US') ?>
 => 'United States'

<?php format_language('en') ?>
```

```
 => 'English'

<?php use_helper('Form') ?>

<?php echo input_date_tag('birth_date', mktime(0, 0, 0, 9, 14, 2006))
?>
 => input type="text" name="birth_date" id="birth_date"
value="9/14/06" size="11" />

<?php echo select_country_tag('country', 'US') ?>
 => <select name="country" id="country"><option
value="AF">Afghanistan</option>
      ...
      <option value="GB">United Kingdom</option>
      <option value="US" selected="selected">United States</option>
      <option value="UM">United States Minor Outlying
Islands</option>
      <option value="UY">Uruguay</option>
      ...
    </select>
```

如果给日期辅助函数加一个额外的格式参数，它可以不受 culture 值的影响进行输出。不过如果你的应用是国际化的，就不要使用这个额外的格式参数。

从本地化输入获取数据

如果说在获取数据时要根据用户的 culture 值来显示数据，那么你也可以让用户输入已经国际化了的数据。这可以帮你在转换不同格式和不确定的本地特性的数据方面节省时间。例如，谁有可能在输入框中输入了一个以逗号分割的货币值呢？

你可以通过隐藏真正的数据（就象在 select_country_tag()中一样），或者将复杂数据的不同部分放入多个简单的输入框去的办法来标准化用户的输入格式。

但是对于日期来说，这通常很难做到。用户习惯于用他们自己的格式输入日期，而你需要将这样的日期转换成内部的（也是国际化的）格式。这正是 sfI18N 类要做的事。例 13－5 显示了如何应用这个类。

例 13－5 在动作中从一个本地化格式得到日期

```
$date= $this->getRequestParameter('birth_date');
$user_culture = $this->getUser()->getCulture();

// 取得时间戳
```

```
$timestamp = sfI18N::getTimestampForCulture($date, $user_culture);

// 取得结构化了的日期
list($d, $m, $y) = sfI18N::getDateForCulture($date, $user_culture);
```

数据库中的文本信息

本地化的应用会根据用户 culture 来提供不同的内容。例如，一个在线商店在全球范围销售某产品，价格统一，但各个国家的产品说明却各不相同。这意味着数据库需要为数据存储不同的版本，为此，你需要用某种特别的方法来设计你的数据库模式，并在你要操作本地化模型对象时，使用 culture 值。

创建本地化数据库设计（schema）

对于包括本地化数据的表，你需要将它分成两个表：一个表不包含任何 i18n 列，另一个表则只包含 i18n 列。这两个表通过 1 对多关系连接起来。这种方法可以让你不用改变模型就可以增加任意多种语言。让我们看一下 Product 表。

首先，在 schema.yml 文件中创建表，如例 13－6 所示。

例 13－6 为 i18n 数据建立的数据库设计，文件路径为 config/schema.yml

```
my_connection:
  my_product:
    _attributes: { phpName: Product, isI18N: true, i18nTable:
my_product_i18n }
    id:          { type: integer, required: true, primaryKey: true,
autoincrement: true }
    price:       { type: float }

  my_product_i18n:
    _attributes: { phpName: ProductI18n }
    id:          { type: integer, required: true, primaryKey: true,
foreignTable: my_product, foreignReference: id }
    culture:     { isCulture: true, type: varchar, size: 7, required:
true, primaryKey: true }
    name:        { type: varchar, size: 50 }
```

注意第一个表中的 isI18N 和 i18nTable 属性，及第二个中的特殊的 culture 列。这些都是 Propel 针对 symfony 而增强的特性。

~~Symfony~~symfony 的自动化工具可以快速完成这一过程。如果包含国际化数据的表名是主表的名字加上_i18n 后缀，而且两个表之间通过表中的 id 列相互关联，那么你就可以省略主表中的 i18n 属性，同时可以省略_i18n 表的 id 和 culture 列，symfony 会自动处理这些表。也就是说，在 symfony 看来，例 13－7 的设计

和例 13－6 的设计是完全一样的。

例 13－7 在 config/schema.yml 用简化格式写的 i18n 数据库设计

```
my_connection:
  my_product:
    _attributes: { phpName: Product }
    id:
    price:       float
  my_product_i18n:
    _attributes: { phpName: ProductI18n }
    name:        varchar(50)
```

运用生成的 i18n 对象

一旦建立了相应的对象模型（每次改变 schema.yml 后，别忘了调用 symfony propel-build-model，并用 symfony cc 清空缓存），你就可以使用支持 i18n 的 Product 类，就像只有一个表一样。示例请参看例 13－8.

例 13－8 处理 i18n 对象

```
$product = ProductPeer::retrieveByPk(1);
$product->setCulture('fr');
$product->setName('Nom du produit');
$product->save();

$product->setCulture('en');
$product->setName('Product name');
$product->save();

echo $product->getName();
 => 'Product name'

$product->setCulture('fr');
echo $product->getName();
 => 'Nom du produit'
```

如果你不想在每次使用 i18n 对象时都要设定 culture 值，你也可以在对象类中改变 hydrate()方法。见例 13－9 的示例。

例 13－9 在 myproject/lib/model/Product.php 中重载 hydrate()方法以设置 culture 值

```
public function hydrate(ResultSet $rs, $startcol = 1)
{
  parent::hydrate($rs, $startcol);
  $this->setCulture(sfContext::getInstance()->getUser()-
>getCulture());
}
```

查询 peer 对象时，如果不用通常的 doSelect 方法，而用 doSelectWithI18n 方法，就可以根据当前的 culture 值得到翻译的结果。如例 13－10 所示。另外，这个方法还同时建立与 i18n 有关的对象，因而可以以较少的查询得到全部内容(参看第 18 章的内容，该方法有助于提高性能)。

例 13－10 用 i18n 规则获取对象

```
$c = new Criteria();
$c->add(ProductPeer::PRICE, 100, Criteria::LESS_THAN);
$products = ProductPeer::doSelectWithI18n($c, $culture);
//  $culture 参数是可选的，
//  如果没有指明 culture，则使用当前的用户 culture。
```

总的来说，你不要直接操作 i18n 对象，而应该在每次用规则的对象查询时将 culture 值传递给模型。

界面翻译

I18n 应用系统中的用户界面应该加以适当的处理，虽然模板中的标签、信息和导航栏使用多种语言，但显示时应该使用一致的表达方式。symfony 建议你用默认语言建立模板，然后在一个字典文件中，为模板中要用的词汇提供一种对应的翻译。这样，当你要修改、增加或去除一个翻译时，你就不必修改你的模板了。

翻译的配置

模板在默认情况下是不会被翻译的，你需要在执行其他命令之前在 setting.yml 文件中激活模板翻译特性才能自动翻译模板。见例 13－11 所示。

例 13－11 在 myapp/config/settings.yml 中激活界面翻译

```
all:
  .settings:
    i18n: on
```

运用翻译辅助函数

我们假设你要创建一个包含英文和法文的网站，而英文是网站的默认语言。在翻译网站前，你也许写了一个象例 12－12 所示的模板。

例 13－12 单一语言的模板

```
Welcome to our website. Today's date is <?php echo
format_date(date()) ?>
```

要在 symfony 中翻译模板中的词汇，它们必须先被识别为文本，然后才能翻译。这个可以通过 I18N 辅助函数组的__()（双下划线）辅助函数做到。因此，你的所有模板在这样的函数调用中都要包含要翻译的词汇。例如，例 13－12 可以改成例 13－13 中的样子（在你看了本章后面的“处理复杂的翻译需要”一节后，将有更好的方法调用本例中的翻译辅助函数）。

例 13－13 一个可用于多语言环境的模板

```
<?php use_helper('I18N') ?>

<?php echo __('Welcome to our website.') ?>
<?php echo __("Today's date is ") ?>
<?php echo format_date(date()) ?>
```

TIP 如果你的应用程序中每页都要使用 I18N 辅助函数组，那么在 settings.yml 文件中设定 standard_helpers 参数应该是一种更好的方法，这样你可以避免在每个模板中都重复地写 use_helper('I18N')。

运用字典文件

每次调用__()函数时，symfony 就根据用户当前的 culture，在相应的字典中对变量进行翻译。如果找到合适的翻译结果，就会将结果返回并显示出来。所以说，用户界面翻译依赖于字典文件。

字典文件是用 XLIFF 格式写成的（XLIFF 是 XML Localization Interchange File Format 的缩写，中文译作“XML 本地化交换文件格式”），文件名则根据 messages.[语言代码].xml 文件中定义的模式来命名，文件路径在应用程序的 i18n/目录下。

XLIFF 是一种基于 XML 的标准格式。因为它广为人知，所以你可以用第三方翻译工具去处理和翻译你站点中的所有文本。翻译公司知道如何添加一个 XLIFF 翻译文件去处理这些文件和翻译整个站点。

TIP 除了 XLIFF 标准，symfony 还支持 gettext、MySQL、SQLite 和 Creole 等的字典翻译。更多的信息和相应的配置方法请参考 API 文档。

例 13－14 中的 messages.fr.xml 展示了 XLIFF 语法的使用，利用它可以将例 13－13 翻译成法语。

例 13－14 一个 XLIFF 字典，文件名为 myapp/i18n/messages.fr.xml

```
[xml]
<?xml version="1.0" ?>
<xliff version="1.0">
  <file orginal="global" source-language="en_US"
datatype="plaintext">
    <body>
      <trans-unit id="1">
        <source>Welcome to our website.</source>
        <target>Bienvenue sur notre site web.</target>
      </trans-unit>
      <trans-unit id="2">
        <source>Today's date is </source>
        <target>La date d'aujourd'hui est </target>
      </trans-unit>
    </body>
  </file>
</xliff>
```

source-language 属性必须总是包含默认 culture 值的完整 ISO 代码。每个翻译都用一个包含了唯一 id 属性的 trans-unit 标记来定义。

对于默认的用户 culture 值（设定为 en_US），词汇不会被翻译，而且显示的是 __() 调用的原始参数值。这样例 13－13 的结果就和例 13－12 的类似了。但是如果 culture 值变成了 fr_FR 或 fr_BE，显示的就是 messages.fr.xml 中翻译后的值了，结果如例 13－15 所示。

例 13－15 一个翻译后的模板

```
Bienvenue sur notre site web. La date d'aujourd'hui est
<?php echo format_date(date()) ?>
```

如果还要增加其他的语言版本，只需将新的翻译文件 messages.XX.xml（XX 是语言代码）放入同一个目录中就可以了。

管理字典

如果你的 messages.XX.xml 文件长得无法阅读，你可以按主题将这个文件分成多个字典文件。例如，你可以在应用程序的 i18n/ 目录中将 messages.fr.xml 文件分成以下三个：

- navigation.fr.xml
- terms_of_service.fr.xml
- search.fr.xml

注意，如果在默认的 messages.XX.xml 文件中找不到翻译，那么你每次要调用__()时，都必须用它的第三个参数指明使用那个字典。例如，如果要显示一个用 navigation.fr.xml 字典翻译的字符串，你应该用以下语法：

<?php echo __('Welcome to our website', null, 'navigation') ?>

另一种管理字典文件的方法是按模块分割。你可以在每个模块的 modules/[模块名]/i18n/目录中放一个 messages.XX.xml 翻译文件，而不是在整个应用程序中用一个统一的翻译文件。这可以让模块独立于整个应用程序，如果你想重用模块（例如第 17 章介绍的 plug-in），这样做就很有必要。

处理需要翻译的其他元素

需要翻译的其他元素还有如下一些：

- 根据用户 culture 的不同，图片，文本文档或其他类型的资源也会变化。最好的例子是一个使用特殊排版的文本，它实际上是一幅图片。为此，我们可以创建一个以用户 culture 值命名的子目录：

  getCulture().'/myText.gif') ?>

- 来自验证文件的错误信息会由一个__()方法自动输出，所以你需要将对应的翻译加入到字典中以便输出对应的翻译。
- 默认的 symfony 页面（page not found，internal server error，restricted access 等）都是用英语写的，你可以在 i18n 应用程序中重写这些页面。你应该在应用程序中创建了自己的 default 模块，并且在模板中使用了__()方法。你可以在第 19 章中找到如何定制这些页面的方法。

处理复杂的翻译需求

只有当__()的参数是一个完整的句子时，翻译才有意义。但是，当你的格式或变量名中混合了其它词汇时，你就会想把句子分割成多个小段，这样辅助函数就遇到了无意义的词汇。幸亏，__()辅助函数提供了一个基于标记的替换功能，可以帮助你构造一个易被其它翻译器处理的有意义的词典。对于 HTML 格式，只需把它留给辅助函数处理即可。例 13-16 给出了示例。 例 13－16 翻译包含代码的语句。

// 原文

```
Welcome to all the <b>new</b> users.<br />
There are <?php echo count_logged() ?> persons logged.

// 拆分文本翻译的较差方法
<?php echo __('Welcome to all the') ?>
<b><?php echo __('new') ?></b>
<?php echo __('users') ?>.<br />
<?php echo __('There are') ?>
<?php echo count_logged() ?>
<?php echo __('persons logged') ?>

// 拆分文本翻译的较好方法
<?php echo __('Welcome to all the <b>new</b> users') ?> <br />
<?php echo __('There are %1% persons logged', array('%1%' =>
count_logged())) ?>
```

本例中，标记是%1%，当然也可以是其它任何标记，因为翻译辅助函数使用的替换函数是strtr()。

翻译中的一个常见问题是复数的使用。结果数值的不同会引起文本的变化，但在不同的语言里，变化的方式却不一定相同。例如，例13－16中，如果count_logged()返回0或1时，最后句子就不正确了。你可以根据函数的返回值进行测试以选择哪个句子能正确使用，但那意味着要增加大量的代码。另外，不同的语言有不同的语法规则，复数的词尾变化规则非常复杂。因为这个问题非常普遍，symfony提供了一个辅助函数format_number_choice()来处理它，例13－17是使用这个辅助函数的示例。

例13－17 根据参数值翻译语句

```
<?php echo format_number_choice(
  '[0]Nobody is logged|[1]There is 1 person logged|(1,+Inf]There are
%1% persons logged', array('%1%' => count_logged()),
count_logged()) ?>
```

第一个参数给出了文本的多种可能值。第二个参数是替换模式（就象__()辅助函数一样），该参数是可选的。第三个参数是一个返回数值的函数，通过这个函数的返回结果确定采用哪个文本。

信息/字符串选项用管道符号(|)分割，后跟一个可接受值的数列，该数列的语法如下：

- [1,2]: 可接受值在1和2 之间，包含1和2。
- (1,2): 可接受值在1和2 之间，不包含1和2。
- {1,2,3,4}: 只接受定义在集合中的值。

- [-Inf,0): 接受小于0且大于等于负无穷大的值。

用方括号或圆括号括起的任何非空组合都是可接受的。 为了能正确地翻译，这个信息必须精确地在XLIFF文件中定义。例13－18给出了示例。

例13－18 包含format_number_choice()参数的XLIFF字典

```
...
<trans-unit id="3">
  <source>[0]Nobody is logged|[1]There is 1 person logged|
(1,+Inf]There are%1% persons logged</source>
  <target>[0]Personne n'est connecté|[1]Une personne est connectée|
(1,+Inf]Ily a %1% personnes en ligne</target>
</trans-unit>
...
```

SIDEBAR 关于字符集

在模板中处理国际化内容常常会出现字符集问题。如果你用本地化字符集，那么每当用户改变culture时，你就要改变字符集。另外，用一种字符集编写的模板通常不能很好地在另一种字符集环境下显示。

所以一旦你开始处理多国家和语言的应用时，你的所有模板都应保存为UTF-8字符格式，所有的界面也应声明为使用UTF-8字符集。如果你一直使用UTF-8，将会给你解决许多麻烦。

在settings.yml文件中定义了整个symfony应用程序要使用的字符集。修改这个参数会修改所有回应的content-type头信息的值。

```
all:
  .settings:
    charset: utf-8
```

在模版外调用翻译辅助函数

页面中显示的文本不一定都来自于模版。这也是你为什么要经常在应用中的动作、过滤器、模型类等部分调用__() 辅助函数的原因。例13－19显示了如何通过获取上下文环境中I18N对象的当前实例，在动作中调用辅助函数。

例13－19 在动作中调用__() 辅助函数。

```
$this->getContext()->getI18N()->__($text, $args, 'messages');
```

总结

如果你掌握了如何运用用户 culture 值，那么你就能在 web 应用中轻松地处理国际化和本地化问题了。辅助函数将自动输出经过正确格式化的数据，而数据库中的本地化数据也会被看作一个简单表的一部分。至于界面翻译，辅助函数__() 和 XLIFF 字典可以确保你用最少的工作量获得最大的灵活性。