

第10章 Drupal 表单 API (form API) - 表单处理流程

表单 API

Drupal4.7 及更高版本提供了一个应用程序接口 (API) 用来生成、验证和处理 HTML 表单。表单 API 将表单抽象为一个关于属性和值的嵌套数组。在生成页面时，表单呈现引擎会在适当的时候将数组呈现出来。这种方式包含多层含义：

没有直接输出 HTML，我们而是创建了一个数组并让引擎生成 HTML

由于我们将表单的表示处理为结构化的数据，所以我们就可以对表单进行添加、删除、重新排序、和修改等操作。当你想用一种干净平和的方式对由其它模块创建的表单进行修改时，这会特别方便。

任意一个表元素可以映射到任意一个主题函数上

可以对任意表单添加额外的表单验证或处理函数

对表单操作进行了保护，从而防止表单注入攻击，特别是用户修改了表单并接着试图提交它时。

使用表单的学习曲线有点高

在本章，我们将迎难而上。我们将学习如何创建、验证、处理表单，以及当我们想要一个个性化的外观时如何编写主题函数：本章所讲的都是关于 Drupal5 版本所实现的表单 API。

理解表单处理流程

图10-1 展示了表单的构建、验证、和提交的流程

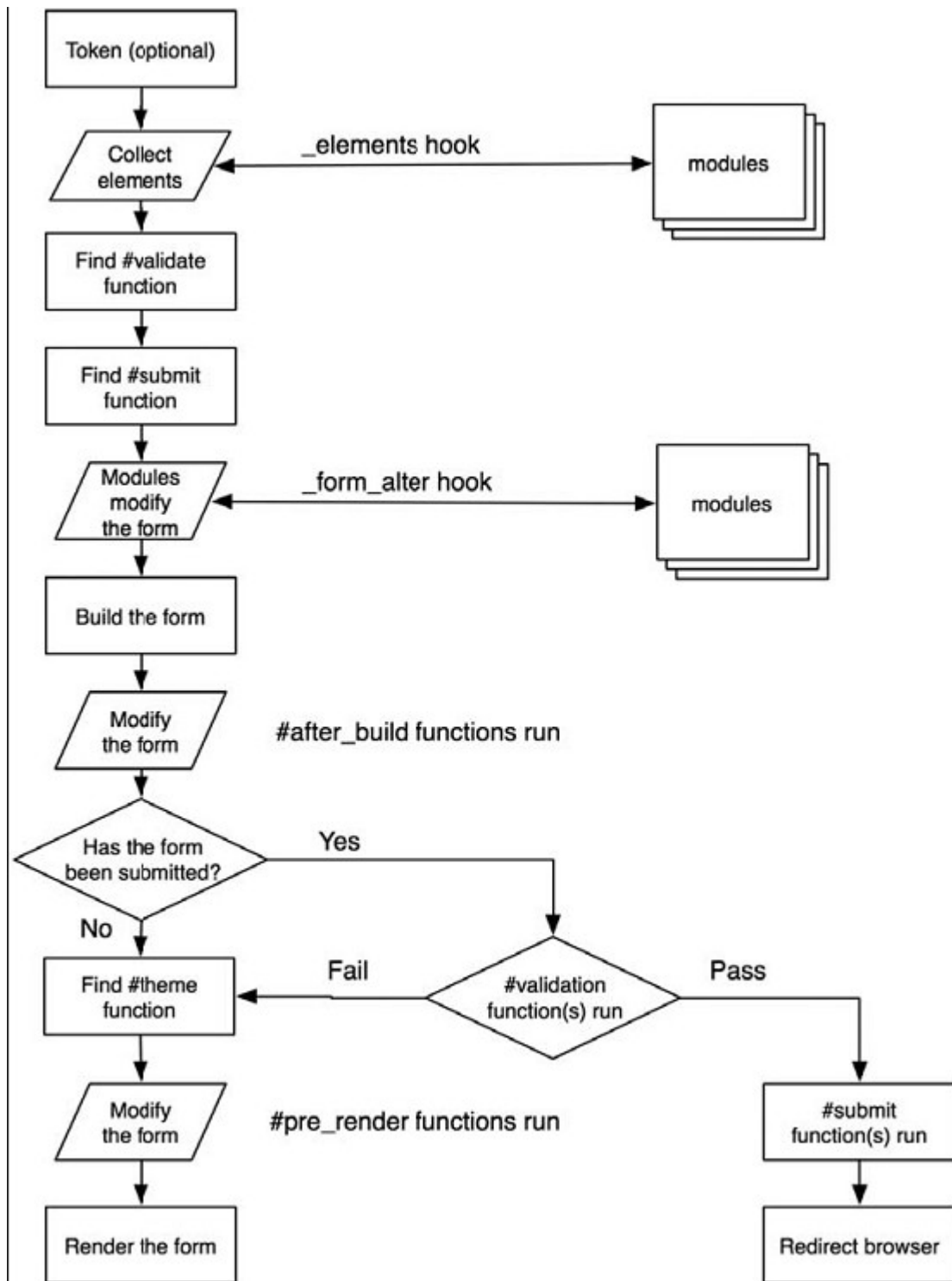


图10-1 Drupal 是如何处理表单的

为了更好的与表单 API 进行交互，理解 API 背后的引擎是如何工作的将会非常有用。在接下来的部分解释了调用 `drupal_get_form()` 时都发生了什么。

流程初始化

调用 `drupal_get_form()` 后，首先初始化，`$form_values`（用来保存提交数据的数组）置为空数组，`$form_submitted` 设为 `FALSE`。

设置一个令牌 (token)

表单系统的一个优点是，它尽力去保证被提交的表单是由 Drupal 真正创建的。这主要是为了安全性和防止垃圾信息的提交。为了达到这一点，Drupal 为每一个 Drupal 安装都设置了一个私钥，Drupal 将基于私钥生成一个随机的令牌作为隐藏于发送到表单中。当表单提交时，会对令牌进行测试。相关背景信息请参看 drupal.org/node/28420。令牌只在登录用户中适用，因为匿名用户的页面通常会被缓存起来，这样它们就没有一个唯一的令牌了。

设置一个ID

一个包含了当前表单 ID 的隐藏域作为表单的一部份被发送给浏览器。该 ID 一般对应于定义表单的函数，它也是方法 `drupal_get_form()` 的第一个参数。例如函数 `user_register()` 定义了用户注册表单，它的调用方式如下：

```
$output = drupal_get_form('user_register');
```

收集所有可能的表单元素定义

接着，调用 `element_info()`。它将触发所有实现了钩子函数 `hook_elements()` 的模块上的钩子函数。在 Drupal 核心内部，标准表单元素，比如单选按钮和复选框，都定义在 `system.module` 中的钩子函数 `hook_elements()` 中。当模块需要定义他们自己的元素类型时，就需要实现这个钩子。在以下几种情况中，你可能需要在你的模块中实现 `hook_elements()`：你想要一个特定的表单元素类型时，比如一个可在预览节点时展示图像缩略图的图像上传按钮；或者，你想通过定义更多的属性来扩展已有的表单元素时。

例如，模块 `views` 定义了它自己的元素类型：

```
/*  
  
 * Custom form element to do our nice images.  
 */  
function views_elements() {  
  $type['imagebutton'] = array(  
    '#input' => TRUE,  
    '#button_type' => 'submit',  
  );  
  return $type;  
}
```

模块 `TinyMCE` 通过禁用 `textarea` 的自适应大小能力修改了一个默认的已存在的表单元素类型（它还添加了一个属性 `#process`，这样当构建表单时，它将调用 `process_textarea()` 从而修改该表单元素）。

```
/**  
  
 * Implementation of hook_elements().  
 */  
function tinymce_elements() {  
  $type = array();  
  if (user_access('access tinymce')) {  
    // Set resizable to false to prevent drupal.js resizable function from  
    // taking control of the textarea.
```

```

$type['textarea'] = array(
  '#process' => array('tinymce_process_textarea' => array()),
  '#resizable' => FALSE
);
}
return $type;
}

```

钩子 `element_info()` 为所有的表元素收集所有的默认属性并将其放到一个本地缓存中。在进入下一步—为表单寻找一个验证器—以前，任何在表单定义中未曾出现的默认属性将被添加进来。

寻找一个验证函数

通过将表单的属性 `#validate` 设置为一个数组，其中函数名为键，一个数组作为值，从而为表单指定一个验证函数。在调用验证函数时，后面数组中的任何数据将被作为参数传递给验证函数。可以使用下面的方式定义多个验证器：

```

$form['#validate'] = array(
  'foo_validate' => array($extra_info),
  'bar_validate' => array()
);

```

如果表单中没有定义属性 `#validate`，那么接下来就要寻找名为“表单 ID”+“_validate”的函数。如果这个函数也不存在，Drupal 将寻找名为“表单 `#base` 属性的值”+“_validate”的函数。如果你有多个表单，它们仅有细微的差别，并且你想让它们共有一个验证器。你可以设置 `$form['#base']`，从而用它代替表单 ID 来构造验证函数的名称。例如，节点模块将 `$form['#base']` 设为“node_form”，这样就可以调用验证函数 `node_form_validate()` 了。

寻找一个提交函数

与前面的验证函数类似，通过将表单的属性 `#submit` 设置为一个数组，其中处理表单提交的函数的名称为键，一个向提交函数提供参数的数组作为值，从而为表单指定一个提交函数。（另外，`$form_id` 和 `$form_values` 恒为提交函数的第一第二参数）

```

// Call my_special_submit_function() on form submission.

$form['#submit'] = array(
  'my_special_submit_function' => array($extra_info)
);

```

如果表单中没有定义属性 `#submit`，那么接下来就要寻找名为“表单ID”+“_submit”的函数。如果这个函数也不存在，Drupal 将寻找名为“表单 `#base` 属性的值”+ “_submit”的函数。如果由任何一种情况成立，那么 Drupal 会将属性 `#submit` 设置为他第一个找到的表单处理器函数。（找到一个以后就不继续寻找了）

允许模块在构建表单以前修改表单

在构建表单以前，将调用钩子函数 `form_alter()`，任何实现了该钩子函数的模块都可以修改表单中的任何部分。这是修改、覆盖、混合由别人的模块所构造的表单的主要方式。

构建表单

现在表单被传递到了函数 `form_builder()` 中，该函数迭代的处理表单树，并添加必须的标准值。

在表单构建完成后允许函数修改表单

在函数 `form_builder()` 中,每次遇到 `$form` 树中的一个新的分支时（例如一个新的字段集或表单元素），它都寻找一个名为 `#after_build` 的可选属性。该属性是一个数组，一旦当前表单被构建完成以后，就会立即调用数组中的函数。当整个表单被构建完成以后，将会调用可选属性 `$form['#after_build']` 中定义的函数。所有的 `#after_build` 函数都接收 `$form_id` 和 `$form_values` 作为参数。

检查表单是否已被提交

我们将在稍后讨论这一点（参看本章后面的“验证表单”部分）。现在我们将假定表单是第一次出现在这里。

为表单寻找主题函数

如果已将 `form['#theme']` 设置为了一个已存在的函数，Drupal 只需要简单的使用该函数就可以了。如果没有设置，Drupal 将调用函数 `theme_get_function()` 来决定是否存在一个主题函数供本表单使用。我们知道表单是通过它的表单 ID 来标识的，Drupal 将根据优先级来寻找下面的主题函数。如果找到了 `bluebeach_foo()`，Drupal 将停止寻找。假定表单 ID 为 `foo`，并且我们使用了一个名为“bluebeach”的 PHPTemplate 主题：

- 1、`bluebeach_foo()`
- 2、`phptemplate_foo()`
- 3、`theme_foo()`

当找到了一个主题函数时，就会将其指定到 `form['#theme']` 上。如果一个也没有找到，将会使用可选的 `$form['#base']` 代替表单 ID 以继续寻找。

允许模块在表单呈现以前修改表单

剩下的最后一件事就是将表单从结构化数组转化为 HTML。但是在这以前，模块还有最后一个机会来修改表单。这对于跨页面向导表单或者其它需要在最后时刻修改表单的方式，将会非常有用。此时将会调用属性 `#pre_render` 中定义的任何函数，函数参数有 `$form_id` 和 `$form`。

显示表单

为了将表单树从一个嵌套数组转化为 HTML 代码，表单构建器调用函数 `drupal_render()`。这一递归函数将会遍历表单树的每个层次，对于每个层次，它将进行以下操作：

- 1、看是否定义了 `#children` 元素（与“为该元素已经生成内容”同义）；如果没有，这样显示表单树中该节点的孩子：

看是否为该元素定义了一个 `#theme` 函数。

如果定义了，临时将该元素的 `#type` 设为 `markup`（标识字体）。接着，接该元素传递到主题函数中，并将 `#type` 重置为原来的值。

如果没有内容生成（可能是由于没有为该元素定义 `#theme` 函数，或者由于调用的 `#theme` 函数没有返回值），那么逐一显示该元素的子元素（也就是，将子元素传递给 `drupal_render()`）。

相反，如果 `#theme` 函数生成了内容，那么将生成的内容存储在该元素的 `#children` 属性中。

2、如果该元素还没有被打印出来，调用该元素的 `#type` 所属类型的元素呈现器。如果没有为该元素设置 `#type` 属性，那么将其默认为 `markup`。

3、在内容前面追加 `#prefix`，在后面追加 `#suffix`，并将它从函数中返回。

该递归反复执行的结果就是为表单树的每一层次生成 HTML。例如，在一个表单包含一个字段集，而字段集又包含两个字段，那么该字段集的 `#children` 元素将包含两个字段的 HTML，而表单的 `#children` 元素将包含整个表单的 HTML（其中包括字段集的 HTML）。

生成的 HTML 将会返回给函数 `drupal_get_form()` 的调用者。这就是显示表单所要做的全部工作！

验证表单

现在让我们看一下需要检查表单是否被提交的情景。决定一个表单是否已被提交有以下几点：`$_POST` 是否为空，`$_POST['form_id']` 中的字符串是否匹配表单的 ID 或者 `$form['#base']` 的 ID。如果都匹配，那么 Drupal 将开始验证了。

验证中首先检查的是该表单是否使用了 Drupal 的令牌机制（参看本章前面的“设置一个令牌”部分）。所有使用令牌的 Drupal 表单将会有有一个唯一的令牌，它与表单一同发送给浏览器，并且将会和其字段值一同被提交。如果令牌不匹配或者不存在，验证失败（尽管剩下的验证会继续进行，以将其它验证错误标识出来）。

接着检查必填字段，看用户有没有漏填的。检查带有选项的元素（复选框、单选按钮、下拉选择框），看所选的值是否是位于构建表单时所生成的原始选项列表中。

如果为单个表单元素定义了一个 `#validate` 属性，将会调用该属性定义的函数。

最后，表单 ID 和表单值将被传递到为表单声明的验证函数中（一般函数名为“表单 ID”+“`_validate`”）。

提交表单

如果验证通过了，那么就该把表单和表单中的值传递到一个函数中，该函数将做一些实际逻辑处理作为表单提交的结果。事实上，可以有多个函数用来处理表单，这是由于 `#submit` 属性可以包含多个键值对，其中键为要调用的函数的名称，值为一个用来为函数传递参数的数组（另外，表单 ID 和表单值，分别为函数的第一第二参数）。

为用户重定向

处理表单的函数的返回值是一个为用户重定向的 Drupal 路径，比如 `node/1234`。如果 `#submit` 属性中有多个函数，那么将会使用最后一个函数的返回值。如果函数没有返回一个 Drupal 路径的话，那么用户将回到原来的页面（也就是，`$_GET['q']` 的值）。从最后一个函数返回 `FALSE` 将阻止重定向。

可以通过在表单中定义 `#redirect` 属性来覆盖函数的返回值，比如 `$form['#redirect'] = 'node/1'` 或 `$form['#redirect'] = array('node/1', $query_string, $named_anchor)`。实际的重定向由 `drupal_goto()` 负责，它为 Web 服务器返回一个定位（Location）头部信息。

第10章 Drupal 表单 API (form API) - 创建基本的表单

创建基本的表单

如果你拥有直接通过 HTML 创建自己表单的经验，那么，刚开始，你可能会觉得 Drupal 的方式让人感到困惑。本节通过例子使你能够快速的创建自己的表单。在开始，我们将创建一个简单的模块，用来让你输入自己的名字并将其打印出来。我们把它放在一个独立的模块里面，这样我们就不需要修改任何已存在的代码了。我们的表单仅包含两个元素：文本输入框和提交按钮。我们首先在 简历一个 .info 文件，输入以下内容：

```
; $Id$
```

```
name = Form example
```

```
description = Shows how to build a Drupal form.
```

```
version = "$Name$"
```

接下来，我们把真实的模块放在 sites/all/modules/custom/formexample/formexample.module:

```
<?php
```

```
// $Id$
```

```
/**
```

```
 * Implementation of hook_menu().
```

```
 */
```

```
function formexample_menu($may_cache) {
```

```
  $items = array();
```

```
  if ($may_cache) {
```

```
    $items[] = array(
```

```
      'path' => 'formexample',
```

```
      'title' => t('View the form'),
```

```
      'callback' => 'formexample_page',
```

```
      'access' => TRUE
```

```
    );
```

```
  }
```

```
  return $items;
```

```
}
```

```
/**
```

```
 * Called when user goes to example.com/?q=formexample
```

```
 */
```

```
function formexample_page() {
```

```
  $output = t('This page contains our example form.');
```

```
  // Return the HTML generated from the $form data structure.
```

```
  $output .= drupal_get_form('formexample_nameform');
```

```
  return $output;
```

```
}
```

```
/**
```

```
 * Defines a form.
```

```
 */
```

```
function formexample_nameform() {
```

```
  $form['user_name'] = array(
```

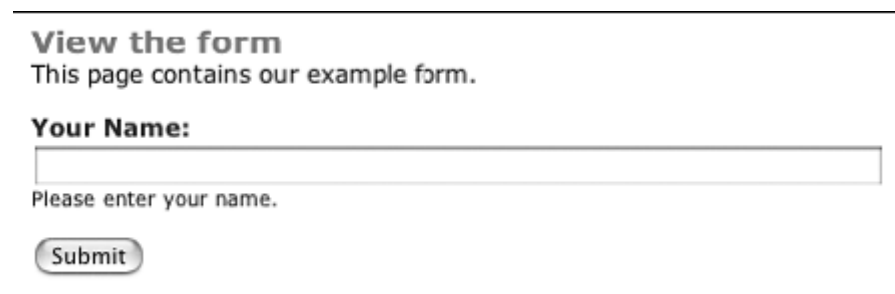
```
    '#title' => t('Your Name'),
```

```

'#type' => 'textfield',
'#description' => t('Please enter your name.'),
);
$form['submit'] = array(
  '#type' => 'submit',
  '#value' => t('Submit')
);
return $form;
}
/**
 * Validate the form.
 */
function formexample_nameform_validate($form_id, $form_values) {
  if ($form_values['user_name'] == 'King Kong') {
    // We notify the form API that this field has failed validation.
    form_set_error('user_name',
      t('King Kong is not allowed to use this form.'));
  }
}
/**
 * Handle post-validation form submission.
 */
function formexample_nameform_submit($form_id, $form_values) {
  $name = $form_values['user_name'];
  drupal_set_message(t('Thanks for filling out the form, %name',
    array('%name' => $name)));
}

```

我们在这里实现了处理表单的基本函数：一个函数用户定义表单，一个用于验证，一个用于提交。另外，我们实现了一个菜单钩子和它对应的函数，这样就将一个 URL 和我们的函数联系在了一起。如果你现在正在计算机上实践并且已安装了这一模块，那么就可以访问 <http://example.com/?q=formexample> 以查看表单了。我们的简单表单如图10-2所示：



View the form
This page contains our example form.

Your Name:

Please enter your name.

图10-2 一个包含了一个文本输入框和一个提交按钮的基本表单

工作的重点就是向表单的数据结构中填充数据，或句话说，就是向 Drupal 表述表单。这一信息包含在一个嵌套的数组中，该数组描述了表单的元素和属性，它一般包含在一个名为 `$form` 的变量中。

在前面的例子中，定义表单的重点在于方法 `formexample_nameform()`，在这里我们提供了 Drupal 展示表单所需要的最小信息。

注意：属性和元素有哪些区别呢？最基本的区别就是，属性没有属性，而元素可以有属性。一个元素的例子就是提交按钮，属性的例子就是提交按钮的属性 `#type`。你一眼便可以认出属性，因为属性拥有前缀 `#`。我们有时把属性称为键，因为它们拥有一个值，为了得到该值，你必须知道相应的键。一个初学者常遇到的错误是忘记了前缀 `#`，此时，无论是 Drupal 还是你自己，都会感到非常困惑。

表单属性

有些属性是通用的，而有些则特定于一个元素，比如一个按钮。对于属性的完整列表，参看本章的最后部分。下面这个表单是比上例中表单复杂一点的版本：

```
$form['#method'] = 'post';

$form['#action'] = 'http://example.com/?q=foo/bar';
$form['#attributes'] = array(
  'enctype' => 'multipart/form-data',
  'target' => 'name_of_target_frame'
);
$form['#prefix'] = '';
$form['#suffix'] = '';
```

`#method` 属性的默认值为 `post`，因此可以被忽略。表单 API 不支持 `get` 方法，它在 Drupal 中也不常用，这是因为通过 Drupal 的菜单分发机制可以很容易的解析路径中的参数。`#action` 属性在 `system_elements` 中定义，默认值为函数 `request_uri()` 的结果，一般与显示表单的 URL 相同。

表单 IDs

Drupal 需要有一些方式来唯一的标识表单，这样当一个页面有多个表单时，它就可以决定提交的是哪一个表单，并且可以将表单与处理该表单的函数联系在一起。为了唯一的标识表单，我们为每个表单分配了一个表单 ID。通过调用方法 `drupal_get_form()` 来定义 ID，如下所示：

```
drupal_get_form('mymodulename_identifier');
```

对于大多数表单，其 ID 的命名规则为模块名称加上一个表述该表单做什么的标识。例如，由用户模块创建的用户登录表单，它的 ID 为 `user_login`。

Drupal 使用表单 ID 来决定表单的验证、提交、主体函数的名称。另外，Drupal 使用表单 ID 作为基础来为该表单生成一个 CSS ID，这样在 Drupal 中所有的表单都有一个为一个的 CSS ID。通过设置 `#id` 属性，你可以覆盖 CSS ID。

```
$form['#id'] = 'my-special-css-identifier';
```

表单 ID 作为名为 `form_id` 的隐藏于也嵌套在表单之中。在我们的例子中，我们选择 `formexample_nameform` 作为我们的表单名，这一因为它很好的描述了我们的表单，从名称就可以看出来了，我们表单的目的是让用户输入他们的名称。我们也可以将它命名为 `formexample_form`，但是它的描述性不好，而且在接下来我们可能向再为我们的模块添加一个表单。

字段集

很多时候，你想将你的表单化分成不同的字段集 - 使用表单 API 可以很容易的做到这一点。每一个字段集都定义在表单的数据结构中，而它所包含的字段都定义为它的孩子。让我们向我们的例子中添加一个字段 - 喜欢的颜色：

```
function formexample_nameform() {  
  
    $form = array();  
    $form['name'] = array(  
        '#title' => t('Your Name'),  
        '#type' => 'fieldset',  
        '#description' => t('What people call you.')  
    );  
    $form['name']['user_name'] = array(  
        '#title' => t('Your Name'),  
        '#type' => 'textfield',  
        '#description' => t('Please enter your name.')  
    );  
    $form['color'] = array(  
        '#title' => t('Color'),  
        '#type' => 'fieldset',  
        '#collapsible' => TRUE,  
        '#collapsed' => FALSE  
    );  
    $form['color_options'] = array(  
        '#type' => 'value',  
        '#value' => array(t('red'), t('green'), t('blue'))  
    );  
    $form['color']['favorite_color'] = array(  
        '#title' => t('Favorite Color'),  
        '#type' => 'select',  
        '#description' => t('Please select your favorite color.'),  
        '#options' => $form['color_options']['#value']  
    );  
    $form['submit'] = array(  
        '#type' => 'submit',  
        '#value' => t('Submit')  
    );  
    return $form;  
}
```

表单的显示结果如图10-3所示：

View the form
This page contains our example form.

Your Name
What people call you.

Your Name:

Please enter your name.

Color

Favorite Color:

red

Please select your favorite color.

Submit

图10-3 带有字段集的简单表单

我们使用可选的属性 `#collapsible` 和 `#collapsed` 来告诉 Drupal 使用 Javascript，在点击第2个字段集标题使它可以伸缩。

这里有个问题值得思考：当 `$form_values` 传递到验证和提交函数是，字段 `color` 应该是 `$form_values['color']` 还是 `$form_values['favorite_color']`？换句话说，值需不需要嵌套在字段集里面。默认时，在表单处理器中，表单值不用嵌套，所以 下面的代码是正确的：

```
function formexample_nameform_submit($form_id, $form_values) {  
  $name = $form_values['user_name'];  
  $color_key = $form_values['favorite_color'];  
  $color = $form_values['color_options'][$color_key];  
  drupal_set_message(t('%name loves the color %color!',  
    array('%name' => $name, '%color' => $color)));  
}
```

在提交函数中设置的消息可以在图10-4中看到。

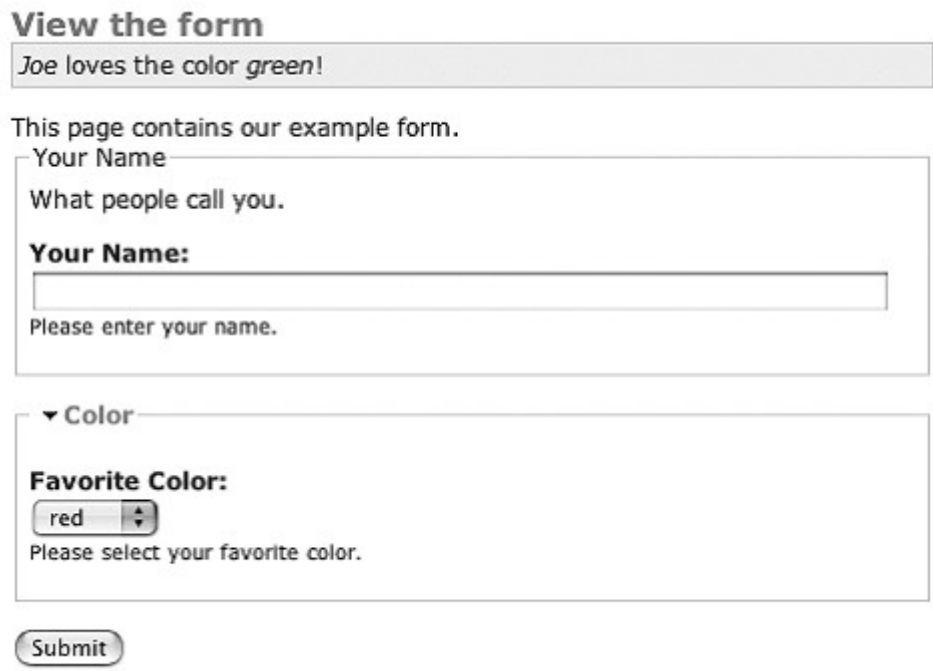
然而，如果将属性 `#tree` 设为 `TRUE`，那么表单值数组的结构就是嵌套的格式。所以，如果我们在定义表单时声明了：

```
$form['#tree'] = TRUE;
```

那么我们就可以使用下面的方式访问数据了：

```
function formexample_nameform_submit($form_id, $form_values) {  
  $name = $form_values['name']['user_name'];  
  $color_key = $form_values['color']['favorite_color'];  
  $color = $form_values['color_options'][$color_key];  
  drupal_set_message(t('%name loves the color %color!',  
    array('%name' => $name, '%color' => $color)));  
}
```

}



View the form

Joe loves the color *green*!

This page contains our example form.

Your Name

What people call you.

Your Name:

Please enter your name.

▼ Color

Favorite Color:

red

Please select your favorite color.

Submit

图10-4 表单提交函数中设置的消息

提示：将属性 `#tree` 设为 `TRUE`，你将得到一个嵌套的表单值数组。当将属性 `#tree` 设为 `FALSE`（默认情况），你将得到一个未嵌套的表单值数组。

第10章 Drupal 表单 API（form API）- 创建基本的表单

(2)

主题化表单

Drupal 拥有内置的函数，用以处理你定义的表单数据结构，并将其翻译或者说是呈现成 HTML 代码。然后，许多时候你可能需要修改 Drupal 生成的输出，或者你可能想其安全的控制整个流程。幸运的是，Drupal 很容易做到这一点。

使用 `#prefix`、`#suffix` 和 `#markup`

如果你的主体化需求非常简单，那么你就可以使用属性 `#prefix` 和 `#suffix` 来为表单元素前面和/或后面添加 HTML 代码，从而满足需求：

```
$form['color'] = array(
  '#prefix' => "
  '#title' => t('Color'),
  '#type' => 'fieldset',
  '#suffix' => "
  t('This information will be displayed publicly!') . "
);
```

该代码在颜色（color）字段集上方添加了一条水平线，在其下方添加了一条私有消息。你

甚至可以在你的表单中使用类型 `#markup` 来声明标示字体。任何一个没有属性 `#type` 的表单元素默认为类型 `#markup`。

```
$form['blinky'] = array(
  '#type' => 'markup',
  '#value' => 'Hello!'
);
```

注意：本方法向你的表单中引入了 HTML 标示字体，一般认为该方法与使用 标签效果差不多好。但是与编写一个主题函数相比，它不够干净利落，同时也增加了你网站设计人员的工作量。

使用主题函数

主题化表单的最灵活的方式是为表单或者表单元素单独构建一个主题函数。Drupal 默认的主题函数名称为“`theme_`”加上你的表单 ID 的名称。在我们的例子中，使用了 `theme_formexample_nameform()`。下面的主题函数将被调用并生成完全一样的输出

```
function theme_formexample_nameform($form) {
  $output = drupal_render($form);
  return $output;
}
```

拥有我们自己的主题函数的好处是，我们可以按照我们的意愿对变量 `$output` 进行解析、混合、添加等操作。我们可以快速的讲一个特定元素放到表单的最前面。例如在下面的例子中，我们把颜色字段集放在了最前面。

```
function theme_formexample_nameform($form) {
  // Always put the the color selection at the top.
  $output = drupal_render($form['color']);
  // Then add the rest of the form.
  $output .= drupal_render($form);
  return $output;
}
```

告诉 Drupal 使用哪一个主题函数

通过为一个表单声明属性 `#theme`，你可以命令 Drupal 使用给定的主题函数，无论它是否匹配“`theme_`”加表单 ID 的格式。

```
// Now our form will be themed by the function theme_formexample_special_theme().
```

```
$form['#theme'] = 'formexample_special_theme';
```

或者，你也可以让Drupal为一个表单元素使用一个特定的主题函数：

```
// Theme this fieldset element with theme_formexample_coloredfieldset().
```

```
$form['color'] = array(
  '#title' => t('Color'),
  '#type' => 'fieldset',
  '#theme' => 'formexample_coloredfieldset'
```

);

注意：Drupal 将在你设定的 `#theme` 属性的字符串前面添加前缀“`theme_`”，所以我们将 `#theme` 设置为 `formexample_coloredfieldset` 而不是 `theme_formexample_coloredfieldset`，尽管后者是函数名称。为什么这样呢？请参看第8章。

用钩子函数 `hook_forms()` 声明验证和提交函数

有时，你回遇到一种特殊的情况，你想让许多不同的表单共有同一个验证或者提交函数。这就是所谓的代码复用。在该情况下，这是一个不错的主意。例如，在节点模块中，所有的节点类型的表单都共有一个验证和提交函数。那么我们就需要一种方式，以将表单 ID 映射到验证和提交函数上。让我们进入到 `hook_forms()` 中。

在 Drupal 回显表单时，它首先查找基于表单 ID 定义表单的函数（正因为这样，在我们的代码中，我们输用函数 `formexample_nameform()`）。如果找不到该函数，它将触发 `hook_forms()`，该钩子函数在所有的模块中查找匹配的表单 ID 以进行回调。例如，在 `node.module` 中，使用下面的代码将不同类型的节点表单 ID 映射到同一个处理器上：

```
/**
 * Implementation of hook_forms(). All node forms share the same form handler.
 */
function node_forms() {
  foreach (array_keys(node_get_types()) as $type) {
    $forms[$type . '_node_form']['callback'] = 'node_form';
  }
  return $forms;
}
```

在我们的例子中，我们也可以使用钩子函数 `hook_forms()` 以将另一个表单 ID 映射到我们现有的代码上：

```
/**
 * Implementation of hook_forms().
 */
function formexample_forms() {
  $forms['formexample_alternate'] = array(
    'callback' => 'formexample_nameform');
  return $forms;
}
```

现在，如果我们调用 `drupal_get_form("formexample_alternate")`，Drupal 将调用 `formexample_nameform()` 来得到表单定义。然后试图分别调用 `formexample_alternate_validate()` 和 `formexample_alternate_submit()` 以进行验证和提交。

主题、验证、提交函数的调用次序

你已经看到，有多个地方可以为 Drupal 放置你的主题、验证、提交函数。拥有这么多的可选项会让人感到困惑的，到底要调用哪个函数呢？下面是一个 drupal 所查找位置的次序小结。这里给定一个主题函数，假定你使用基于 `PHPTemplate` 的名为 `bluebeach` 的主题，表单定义将可选的属性 `$form['#base']` 设为 `foo`，并且你调用函数

`drupal_get_form('formexample_nameform')`。Drupal 使用它找到的第一个主题函数。

- 1、`$form['#theme']` // A function defined in the element form definition.
- 2、`bluebeach_formexample_nameform()` // Theme function provided by theme.
- 3、`phptemplate_formexample_nameform()` // Theme function provided by theme engine.
- 4、`theme_formexample_nameform()` // 'theme_' plus the form ID.
- 5、`bluebeach_formexample_foo()` // Theme name plus `$form['#base']`
- 6、`phptemplate_formexample_foo()` // Theme engine name plus `$form['#base']`
- 7、`theme_formexample_foo()` // 'theme_' plus the `$form['#base']`

在验证期间，表单验证器的调用次序如下：

- 1、由 `$form['#validate']` 定义的函数
- 2、`formexample_nameform_validate()` // 表单 ID + 'validate'.
- 3、`formexample_foo_validate()` // `$form['#base']` + 'validate'.

当需要查找处理提交的函数时，查找位置的次序如下：

- 1、由 `$form['#submit']` 定义的函数
- 2、`formexample_nameform_submit()` // 表单 ID + 'submit'.
- 3、`formexample_foo_submit()` // `$form['#base']` + 'submit'.

什么时候需要设置 `$form['#base']` 呢？当你拥有多个表单，而它们共有同一个验证和提交函数时，设置它。

编写一个验证函数

Drupal 拥有一个内置的机制，能对未通过验证的表单元素进行高亮显示，并为用户展示一条错误消息。检查一下我们例子中的验证函数来看一下它是如何工作的：

```
/**
 * Validate the form.
 */
function formexample_nameform_validate($form_id, $form_values) {
  if ($form_values['user_name'] == 'King Kong') {
    // We notify the form API that this field has failed validation.
    form_set_error('user_name',
      t('King Kong is not allowed to use this form.));
  }
}
```

注意 `form_set_error()` 的使用。当 King Kong 访问我们的表单并用他的巨大的键盘键入他的姓名的时候，它将会在页面的顶部看到一条错误消息，而包含错误的字段被高亮显示，如图10-5所示：

图10-5 验证失败后为用户给出提示

当然，他也有可能仅仅输入他的名字（不包含姓）。我们在这里仅仅拿他作为一个例子，

来说明 `form_set_error()` 为我们的表单设置了一条错误消息，并使得验证失败。

验证函数应该专注于验证。作为一条一般规则，它不应该改变数据。然而，它们可以为 `$form_values` 数组添加信息，如我们接下来所讲的。

使用 `form_set_value()` 传递数据

第10章 Drupal 表单 API (form API) - 创建基本的表单

(3)

使用 `form_set_value()` 传递数据

如果你的验证函数做了大量的处理，而你又想把结果保存此阿莱以在提交函数中使用，那么你可以使用 `form_set_value()` 暗中将结果保存到表单数据中。首先要做的是，当你在你的表单定义函数中创建表单时，你将需要为验证函数中要保存的处理结果预留一个位置：

```
$form['my_placeholder'] = array(
  '#type' => 'value',
  '#value' => array()
);
```

接下来，在你的验证程序中，你把数据保存起来：

```
// Lots of work here to generate $my_data as part of validation.
```

```
...
```

```
// Now save our work.
```

```
form_set_value($form['my_placeholder'], $my_data);
```

然后在你的提交函数中你就可以访问数据了：

```
// Instead of repeating the work we did in the validation function,
```

```
// we can just use the data that we stored.
```

```
$my_data = $form_values['my_placeholder'];
```

或者假定你想将数据转化为标准形式。例如，你在数据库中有国家代码列表，你需要使用它们进行验证，但是你的不讲道理的老板坚持让用户可以在文本输入框中键入国家名称。你需要在你的表单中创建一个占位表单元素，通过使用一种聪明的方式对用户进行验证，这样你就可以同时将“The Netherlands”和“Nederland”都映射为 ISO 3166 国家代码“NL”了。

```
$form['country'] = array(
  '#title' => t('Country'),
  '#type' => 'textfield',
  '#description' => t('Enter your country.')
);
// Create a placeholder. Will be filled in during validation.
$form['country_code'] = array(
  '#type' => 'value',
  '#value' => ''
);
```



```
);
```

在验证函数内部，你将国家代码保存到占位表单元素中：

```
// Find out if we have a match.

$country_code = formexample_find_country_code($form_values['country']);
if ($country_code) {
// Found one. Save it so that the submit handler can see it.
form_set_value($form['country_code'], $country_code);
}
else {
form_set_error('country', t('Your country was not recognized. Please use a standard name or
country code.));
}
}
```

现在，在提交处理器（函数）中就可以使用 `$form_values['country_code']` 来访问国家代码了。

针对表单元素的验证

一般情况下，一个表单使用一个验证函数，但是也可以像为整个表单设置验证函数一样，为单个表单元素设置一个验证函数。为了实现这一点，我们需要为元素的属性 `#validate` 设置为一个数组，其中验证函数的名称作为键，任何你要传递的参数作为值。表单数据结构中该元素分支的一份完全拷贝，将被作为验证函数的第一个参数。下面是一个专门用来说明这一点的示例，在这里我们强制用户只能向文本输入框中输入香料（spicy）和糖果（sweet）：

```
$allowed_flavors = array(t('spicy'), t('sweet'));

$form['flavor'] = array(
  '#type' => 'textfield',
  '#title' => 'flavor',
  '#validate' => array('formexample_flavor_validate' => array($allowed_flavors))
);
```

那么你的表单元素验证函数应该如下所示：

```
function formexample_flavor_validate($element, $allowed_flavors) {
  if (!in_array($element['#value'], $allowed_flavors) {
    form_error($element, t('You must enter spicy or sweet.));
  }
}
```

在调用完所有的表单元素验证函数以后，仍需调用表单验证函数。

提示：在你的表单元素未通过验证，你希望为它给出一条错误消息时，如果你知道表单元素的名称，那么使用 `form_set_error`，如果你拥有表单元素本身，那么使用 `form_error()`。后者对前者做了简单封装。

编写提交函数

提交函数是表单在被验证以后负责实际表单处理的函数。只有当表单验证完全通过时它才执行。如果要返回的还是同一页面，那么在提交函数中不需要使用关键字“return”。

如果在表单提交以后，你想让用户转到另一个页面，那么你就需要返回一个你要用户接下来访问的Drupal路径：

```
function formexample_form_submit($form, $form_values) {  
  // Do some stuff.  
  ...  
  // Now send user to node number 3.  
  return 'node/3';  
}
```

如果你有多个函数用来处理表单提交（参看本章前面的“提交表单”部分），只有从最后一个函数返回的值才被认为有效（但是如果 Drupal 在第一个函数执行完以后进行了重定向（译者注：重定向应该使用 `drupal_goto()`），那么其它函数就不被执行了，这是因为用户已经被重定向到了别的页面了）。可以通过在表单中定义 `#redirect` 来覆盖提交函数中的重定向（参看本章前面的“为用户重定向”部分）。通常使用钩子函数 `form_alter()` 来做这项工作。

使用 `form_alter()` 来修改表单

使用 `form_alter()`，你可以修改任何表单。你只需要知道表单的 ID 就可以了。让我们修改一下位于用户登录区块和用户登录页面的登录表单。

```
function formexample_form_alter($form_id, &$amp;$form) {  
  // We'll get called for every form Drupal builds; use an if statement  
  // to respond only to the user login block and user login forms.  
  if ($form_id == 'user_login_block' || $form_id == 'user_login') {  
    // Add a dire warning to the top of the login form.  
    $form['warning'] = array(  
      '#value' => t('We log all login attempts!'),  
      '#weight' => -5  
    );  
    // Change 'Log in' to 'Sign in'.  
    $form['submit']['#value'] = t('Sign in');  
  }  
}
```

由于 `&$form` 是通过引用传递过来的，所以在这里我们对表单定义拥有完全的访问权并且可以做出任何我们想要的修改。在示例中，我们使用表单默认元素（参看本章后面的“标示字体（markup）”部分）添加了一些文本并接着修改了提交按钮的值。

使用 `drupal_execute()` 通过程序来提交表单

从 Drupal5 开始，可在网页浏览器中展示的任何表单，也都可以使用程序的方式填充表单。让我们通过程序填入我们的名称和喜欢的颜色。

```
$form_id = 'formexample_nameform';  
  
$field_values = array(  
  'user_name' => t('Marvin'),  
  'favorite_color' => t('green')
```

```
);  
// Submit the form using these values.  
drupal_execute($form_id, $field_values);
```

要做的就是这些！简单的提供表单的 ID 和表单的数值，然后调用 `drupal_execute()` 即可。

警告：大多数提交函数都假定制造请求的用户与提交表单的用户是同一个人。当通过程序来提交表单时，你需要对此格外小心，因为此时这两种用户不必相同。

第10章 Drupal 表单 API (form API) - 创建跨页面表单

跨页面表单

我们已经学习了简单的单页面表单。但是你可能需要让用户跨越多个页面来填充一个表单或者通过多步来输入数据。让我们创建一个简洁的模块来说明跨页面 表单技术，在该模块中通过3个独立的步骤来从用户收集3种成分 (ingredient)。我们的方式是使用表单隐藏域来传递数值。我们将模块命名为 `formwizard.module`。当然，我们首先需要有一个 `formwizard.info` 文件。

```
; $Id$  
  
name = Form Wizard Example  
description = An example of a multistep form.  
version = "$Name$"
```

接着，我们将编写真实的模块。该模块展示两个页面：一个用来输入数据（我们将重复使用这一页面），一个最终页面用来展示用户的输入以及对用户输入的致谢。

```
<?php  
  
// $Id$  
/**  
 * Implementation of hook_menu().  
 */  
  
function formwizard_menu($may_cache) {  
  $items = array();  
  if ($may_cache) {  
    $items[] = array(  
      'title' => t('Form Wizard'),  
      'path' => 'formwizard',  
      'callback' => 'drupal_get_form',  
      'callback arguments' => array('formwizard_multiform'),  
      'type' => MENU_CALLBACK,  
      'access' => user_access('access content'),  
    );  
    $items[] = array(  
      'title' => t('Thanks!'),  
      'path' => 'formwizard/thanks',  
      'callback' => 'formwizard_thanks',
```

```

'type' => MENU_CALLBACK,
'access' => user_access('access_content')
);
}
return $items;
}
/**
 * Build the form differently depending on which step we're on.
 */
function formwizard_multiform($form_values = NULL) {
    $form['#multistep'] = TRUE;
    // Find out which step we are on. If $form_values is NULL,
    // that means we are on step 1.
    $step = isset($form_values) ? (int) $form_values['step'] : 1;
    // Store next step in hidden field.
    $form['step'] = array(
        '#type' => 'hidden',
        '#value' => $step + 1
    );
    // Customize the fieldset title to indicate the current step to the user.
    $form['indicator'] = array(
        '#type' => 'fieldset',
        '#title' => t('Step @number', array('@number' => $step))
    );
    // The name of our ingredient form element is unique for
    // each step, e.g., ingredient_1, ingredient_2...
    $form['indicator']['ingredient_' . $step] = array(
        '#type' => 'textfield',
        '#title' => t('Ingredient'),
        '#description' => t('Enter ingredient @number of 3.', array('@number' => $step))
    );
    // The button will say Next until the last step, when it will say Submit.
    // Also, we turn off redirection until the last step.
    $button_name = t('Submit');
    if ($step < 3) {
        $form['#redirect'] = FALSE;
        $button_name = t('Next');
    }
    $form['submit'] = array(
        '#type' => 'submit',
        '#value' => $button_name
    );
    switch($step) {
        case 2:

```

```

$form['ingredient_1'] = array(
  '#type' => 'hidden',
  '#value' => $form_values['ingredient_1']
);
break;
case 3:
$form['ingredient_1'] = array(
  '#type' => 'hidden',
  '#value' => $form_values['ingredient_1']
);
$form['ingredient_2'] = array(
  '#type' => 'hidden',
  '#value' => $form_values['ingredient_2']
);
}
return $form;
}
/**
 * Validate handler for form ID 'formwizard_multiform'.
 */
function formwizard_multiform_validate($form_id, $form_values) {
  drupal_set_message(t('Validation called for step @step',
    array('@step' => $form_values['step'] - 1)));
}
/**
 * Submit handler for form ID 'formwizard_multiform'.
 */
function formwizard_multiform_submit($form_id, $form_values) {
  if ($form_values['step'] < 4) {
    return;
  }
  drupal_set_message(t('Your three ingredients were %ingredient_1, %ingredient_2,
    and %ingredient_3.', array(
    '%ingredient_1' => $form_values['ingredient_1'],
    '%ingredient_2' => $form_values['ingredient_2'],
    '%ingredient_3' => $form_values['ingredient_3']
  )
  )
  );
  return 'formwizard/thanks';
}
function formwizard_thanks() {
  return t('Thanks, and have a nice day.');
```

在这个简单模块中，我们需要注意几点。在我们的建造表单函数 `formwizard_multiform()` 中，我们将 `$form['#multistep']` 设置为 `TRUE`，这将向 Drupal 说明本表单是一个多步表单。在我们已经跳出第一个页面时，这将促使 Drupal 两次建造该表单。让我们看一下整个流程。如果我们访问 `http://example.com/?q=formwizard`，我们就会得到初始的表单，如图10-6所示：

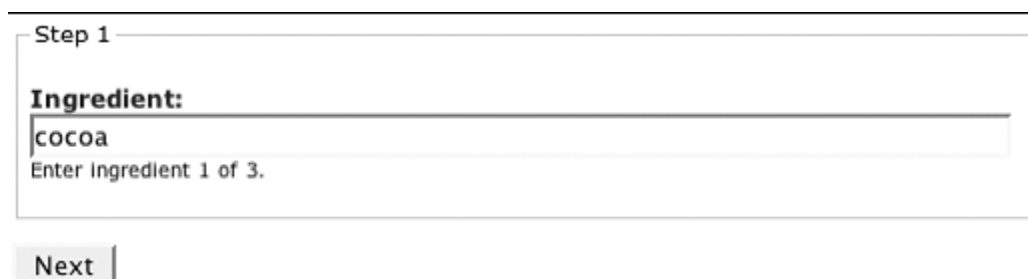


图10-6 多步表单的初始步骤

当我们点击按钮 `Next` 时，Drupal 将像处理其它表单一样处理本表单：构建表单，调用验证函数，调用提交函数。但是接着，由于它是一个多步表单，Drupal 将再次调用建造表单函数 `formwizard_multiform()`，这次带了一个 `$form_values` 拷贝。这允许在我们模块函数 `formwizard_multiform()` 中得到 `$form_values['step']` 从而决定我们所处的步骤并据此建造表单。我们在表单结束本步以后得到的结果如图10-7所示。

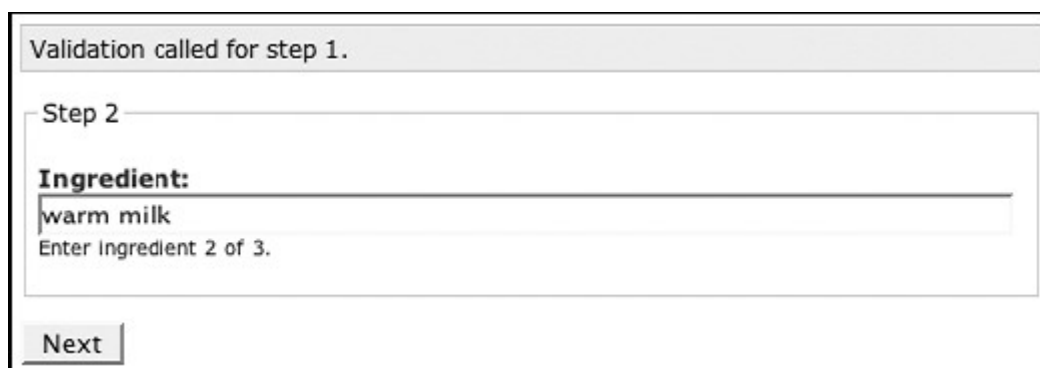


图10-7 多步表单的第2步

我们有证据证明我们的验证函数运行了，因为它通过调用 `drupal_set_message()` 在屏幕上展示了一条消息。而且我们的字段集标题和文本输入框的描述也被恰当的设置了，这意味着用户到达了第2步。让我们继续。在如图10-8所示的表单中输入最后一个成分。

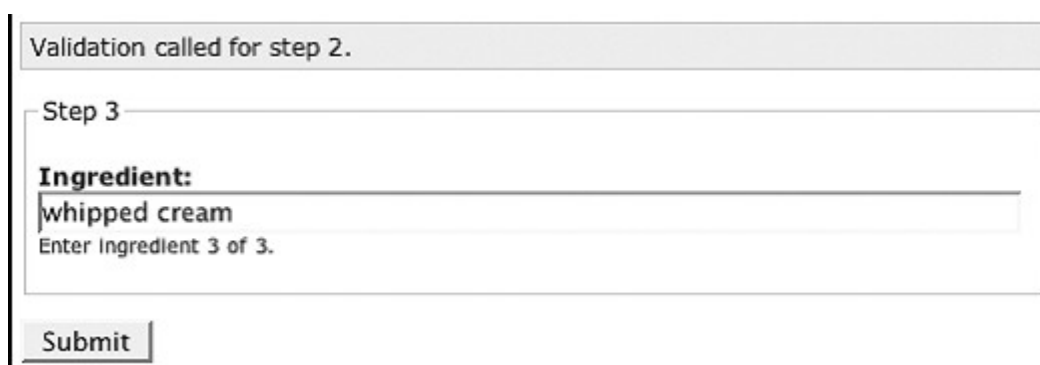


图10-8 多步表单的最后一步

注意，在第3步，我们将提交按钮的名称从 `Next` 修改为 `Submit`。还有，我们也没有将

`$form['#redirect']` 设置为 `FALSE`，而是没有对它进行设置，这样当处理完成以后提交处理器就可以将用户转到一个新的页面上。这样当我们点击提交按钮“Submit”时，我们的提交处理器将识别出本步是第四步，与前面几步直接跳到下一步不同，本步将对数据进行处理在本例中，我们仅仅调用了 `drupal_set_message()`，这将在 Drupal 提供的下一个页面中展示用户输入信息，接着将用户重定向到了 `formwizard/thankyou`。结果页面如图10-9所示。

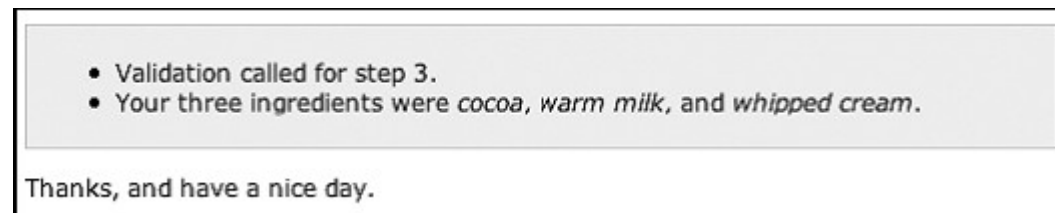


图10-9 多步表单的提交处理器已经运行。

前面的例子试图向你展示多步表单如何工作的基本轮廓。除了将数据保存到隐藏域中从而将其传到下一步，你也可以在你的提交处理器中将其保存到数据库中，或者使用表单 ID 作为键将其保存到全局变量 `$_SESSION`。需要理解的重点是构造表单函数将被继续调用，这是因为设置了 `$form['#multistep']` 和 `$form['#redirect']`，通过使用前面的方式增加 `$form_values['step']`，从而验证和提交函数能够智能的决定要做什么。

第10章 Drupal 表单 API (form API) - 表单 API 属性

表单 API 属性

当在你的表单构造函数中构建一个表单的定义时，数组中的键用来声明表单的信息。下面部分列出了常用的键。一些键将被表单构建器自动添加进来。

表单的属性

下面部分中的属性特定于表单。换句话说，你可以设置 `$form['#programmed'] = TRUE`，但是如果你设置 `$form['myfieldset']['mytextfield']['#programmed'] = TRUE` 那么表单构建器将不知道你要它做什么了。

#parameters

该属性是一个数组，包含了传递给 `drupal_get_form()` 的原始参数。通过 `drupal_retrieve_form()` 可添加该属性。

#programmed

这是一个布尔值属性，用来指示一个表单被以程序的方式比如通过 `drupal_execute()` 来提交的。如果在表单处理前设置了属性 `#post`，那么可以使用 `drupal_prepare_form()` 来设置该属性。

#build_id

该属性是一个字符串（用 MD5 哈希）。在多步表单中，`#build_id` 用来标识一个特定的表单实例。它作为一个隐藏域放在表单中，通过使用 `drupal_prepare_form()` 来设置该表单元素，如下所示：

```
$form['form_build_id'] = array(
```

```
  '#type' => 'hidden',
```

```
'#value' => $form['#build_id'],  
'#id' => $form['#build_id'],  
'#name' => 'form_build_id',  
);
```

#base

这是一个可选的字符串属性，当 Drupal 决定要调用哪个验证、提交、和主题函数时使用。将 `$form['#base']` 设置为你想让 Drupal 使用的前缀。例如，如果你将 `$form['#base']` 设置为 `foo`，并调用 `drupal_get_form('bar')`，在 `bar_validate()` 和 `bar_submit()` 不存在的情况下，Drupal 将使用 `foo_validate()` 和 `foo_submit()` 作为处理器。该属性也用于映射主题函数。参看 `drupal_render_form()` (http://api.drupal.org/api/5/function/drupal_render_form)。

#token

该字符串（用 MD5 哈希）是一个唯一的令牌，每个表单中都带有它，通过该令牌 Drupal 能够决定一个表单是一个真的 Drupal 表单而不是一个恶意用户修改后的。

#id

该属性是一个由 `form_clean_id($form_id)` 生成的字符串，并且它是一个 HTML ID 属性。`$form_id` 中的任何翻转的括号对“`]]`”，下划线“`_`”，或者空格”都将被连字符替换以生成一致的 CSS ID。

#action

该字符串属性是表单标签的动作属性。默认情况，它是 `request_uri()` 的返回值。

#method

该字符串属性指的是表单的提交方式 - 通常为 `post`。表单 API 是基于 `post` 方法构建的，它将不会处理使用 `GET` 方法提交的表单。参看 HTML 规范中的关于 `GET` 和 `POST` 的区分部分。如果你需要使用 `GET` 方法时，你可能需要使用 Drupal 的菜单 API，而不是表单 API。

#redirect

该属性可以是一个字符串或者一个数组。如果是字符串时，那么它是在表单提交以后用户想要重定向的 Drupal 路径。如果将其设为一个数组，该数组将被传递给 `drupal_goto()`，其中数组中的第一个元素应该是目标路径（这将允许向 `drupal_goto()` 传递额外的参数）。

#pre_render

该属性是一个数组，它包含了在表单呈现以前所要调用的函数。每个要调用的函数都包含参数 `$form_id` 和 `$form`。例如，设置 `#pre_render = array('foo', 'bar')` 将促使 Drupal 先调用函数 `foo($form_id, $form)`，然后接着调用 `bar($form_id, $form)`。当你在表单验证完以后，显示以前，使用钩子修改表单结构时非常有用。如果想在验证以前修改表单，使用 `hook_form_alter()`。

添加到所有表单元素上的属性

当表单构建器使用表单定义构建表单时，它需要保证每一个表单元素都要有一些默认设置。这些默认值在 `includes/form.inc` 的函数 `_element_info()` 中设置，但是可以被 `hook_elements()` 中的表单元素定义所覆盖。

#description

将为所有表单元素添加该字符串属性，它默认为 NULL。通过表单元素的主题函数来呈现它。例如，textfield 的描述出现在 textfield 输入框的下面，如图10-2所示。

`#required`

将为所有表单元素添加该布尔值属性，它默认为 FALSE。将它设为 TRUE，如果表单被提交以后而该字段为空时，Drupal 内置的表单验证将抛出一个错误消息。还有，如果将它设为 TRUE，那么就会为该表单元素设置一个 CSS 类（参看 includes/form.inc 中的 theme_form_element()）

`#tree`

将为所有表单元素添加该布尔值属性，它默认为 FALSE。如果将它设为 TRUE，表单提交后的 \$form_values 数组将会是嵌套的（而不是平坦的）。这将影响你访问提交数据的格式。（参看本章中的“字段集（Fieldsets）”部分）。

`#post`

该数组属性是原始 \$_POST 数据的一个拷贝，它将被表单构建器添加到所有的表单元素上。这样，在 #process 和 #after_build 中定义的函数就可以基于 #post 的内容做出智能的决定。

`#parents`

将为所有表单元素添加该数组属性，它默认为一个空数组。它在表单构建器的内部使用，以标识表单树中的父表单元素节点。更多信息，参看 <http://drupal.org/node/48643>。

`#attributes`

将为所有表单元素添加该数组属性，它默认为一个空数组，但是主题函数将逐步的填充该数组。该数组中的成员将被作为 HTML 属性。例如 \$form['#attributes'] = array('enctype' => 'multipart/form-data')。

通用表单元素属性

本部分解释的属性在所有表单元素中都可以使用。

`#type`

该字符串声明了一个表单元素的类型。例如，#type = 'textfield'。表单根部必须包含声明 #type = 'form'。

`#access`

该布尔值属性决定表单元素对于用户是否可见。如果表单元素有子表单元素的话，如果父表单元素的 #access 属性为 FALSE 的话，那么子表单元素将不被显示出来。例如，如果表单元素是一个字段集的话，如果它的 #access 为 FALSE 的话，那么字段集里面的所有的字段都不被显示。

`#process`

该属性是一个关联数组。在数组的每个项目中，一个函数名作为键，传递给函数的任何参数作为值。当构建表单元素时将调用这些函数，从而允许在构建表单元素时添加额外的操作。例如，在 system.module 定义了 checkboxes 类型，在构建表单期间，将调用设置的 includes/form.inc 里面的函数 expand_checkboxes()：

```
$type['checkboxes'] = array(
```

```
'#input' => TRUE,  
'#process' => array('expand_checkboxes' => array()),  
'#tree' => TRUE);
```

参看本章中“收集所有可能的表单元素定义”部分的另一个例子。当 `#process` 数组中的所有函数都被调用之后，将为每个表单元素添加一个 `#processed` 属性。

`#after_build`

该属性是一个数组。数组中保存了在表单元素构建完以后要立即调用的函数。每个要被调用的函数都有两个参数：`$form` 和 `$form_values`。例如，如果 `$form['#after_build'] = array('foo', 'bar')`，那么 Drupal 在表单元素构建完以后，分别调用 `foo($form, $form_values)` 和 `bar($form, $form_values)`。一旦这些函数都被调用之后，Drupal 在内部将为每个表单元素添加一个 `#after_build_done` 属性。

`#theme`

该可选属性定义了一个字符串，当 Drupal 为该表单元素寻找主题函数的时候使用它。例如，设置 `#theme = 'foo'`，将使得 Drupal 调用 `theme_get_function('foo', $element)`，该函数将按照顺序查找函数 `themenamename_foo()`、`themeengine_foo()` 和 `theme_foo()`。参看本章前面的“为表单寻找主题函数”部分。

`#prefix`

该属性是一个字符串，在表单元素呈现时它将被添加到表单元素的前面。

`#suffix`

该属性是一个字符串，在表单元素呈现时它将被添加到表单元素的后面。

`#title`

该字符串是表单元素的标题。

`#weight`

该属性可以是一个整数或者分数。当呈现表单元素时，将根据它们的重量进行排序。重量小的元素将被放到前面，重量大的元素将被放到页面的下面位置。

`#default_value`

该属性的类型根据表单元素的类型而定。对于输入表单元素，如果表单还没有被提交，那么它就是该字段所使用的值。不要将它于表单元素 `#value` 相混淆。表单元素 `#value` 定义了一个内部表单值，用户看不到该值，但是它却定义在表单中并出现在 `$form_values` 中。

第10章 Drupal 表单 API (form API) - 表单元素

表单元素

在本部分，我们将通过例子来展示内置的 Drupal 表单元素。

Textfield

元素 `textfield` 的例子如下：

```
$form['pet_name'] = array(
```

```

'#title' => t('Name'),
'#type' => 'textfield',
'#description' => t('Enter the name of your pet.'),
'#default_value' => $user->pet_name,
'#maxlength' => 32,
'#required' => TRUE,
'#size' => 15,
'#weight' => 5,
'#autocomplete_path' => 'pet/common_pet_names'
);
$form['pet_weight'] = array(
'#title' => t('Weight'),
'#type' => 'textfield',
'#description' => t('Enter the weight of your pet in kilograms.'),
'#after_field' => t('kilograms'),
'#default_value' => '0',
'#size' => 4,
'#weight' => 10
);

```

表单元素的结果如图10-10所示

图10-10 元素 textfield

`#field_prefix` 和 `#field_suffix` 属性为 textfield 专有属性，它们在 textfield 输入框前面或者后面紧接着放置一个字符串。

`#autocomplete` 属性定义了一个路径，Drupal 自动包含进来的 Javascript 将使用 jQuery 向该路径发送 HTTP 请求。在前面的例子中，它将请求 `http://example.com/pet/common_pet_names`。实际例子可以参看 `modules/user.module` 中的 `user_autocomplete()`。

元素 textfield 常用属性如下：`#attributes`、`#autocomplete_path`（默认为 FALSE）、`#default_value`、`#description`、`#field_prefix`、`#field_suffix`、`#maxlength`（默认为 128）、`#prefix`、`#required`、`#size`（the default is 60）、`#suffix`、`#title` 和 `#weight`。

Password

该元素创建一个 HTML 密码字段，在这里用户的输入不被显示（一般在屏幕上使用星号 * 代替）。`user_login_block()` 中的一个例子如下：

```

$form['pass'] = array('#type' => 'password',

```

```
'#title' => t('Password'),
'#maxlength' => 60,
'#size' => 15,
'#required' => TRUE,
);
```

元素 password 常用属性如下：
#attributes、#default_value、#description、#maxlength、#prefix、#required、#size（默认为60）、#suffix、#title 和 #weight。

Textarea<、strong>

元素 textarea 的示例如下：

```
$form['pet_habits'] = array(
'#title' => t('Habits'),
'#type' => 'textarea',
'#description' => t('Describe the habits of your pet.'),
'#default_value' => $user->pet_habits,
'#cols' => 40,
'#rows' => 3,
'#resizable' => FALSE,
'#weight' => 15
);
```

元素 textarea 常用属性如下：#attributes、#cols（默认为60）、#default_value、#description、#prefix、#required、#resizable、#suffix、#title、#rows（默认为5）和 #weight。

如果通过设置 #resizable 为 TRUE 将 textarea 输入框设置为动态调整大小，那么属性 #cols 的设置将不起作用。

Select

statistics.module 中的元素 textarea 示例如下：

```
$period = drupal_map_assoc(array(3600, 10800, 21600, 32400, 43200, 86400, 172800, 259200,
604800, 1209600, 2419200, 4838400, 9676800), 'format_interval');
```

/* 现在日期如下所示:

```
Array (
[3600] => 1 hour
[10800] => 3 hours
[21600] => 6 hours
[32400] => 9 hours
[43200] => 12 hours
[86400] => 1 day
[172800] => 2 days
[259200] => 3 days
```

```

[604800] => 1 week
[1209600] => 2 weeks
[2419200] => 4 weeks
[4838400] => 8 weeks
[9676800] => 16 weeks )
*/
$form['access']['statistics_flush_accesslog_timer'] = array(
  '#type' => 'select',
  '#title' => t('Discard access logs older than'),
  '#default_value' => variable_get('statistics_flush_accesslog_timer', 259200),
  '#options' => $period,
  '#description' => t('Older access log entries (including referrer statistics)
will be automatically discarded. Requires crontab.')
);

```

通过将属性 #options 定义为一个关于子菜单选项的关联数组，Drupal 支持对下拉选项的分组，如图10-11所示。

```

$options = array(
  array(
    t('Healthy') => array(t('wagging'), t('upright'), t('no tail'))),
    t('Unhealthy') => array(t('bleeding'), t('oozing'))
  );
$form['pet_tail'] = array(
  '#title' => t('Tail demeanor'),
  '#type' => 'select',
  '#description' => t('Pick the closest match that describes the tail of your pet.'),
  '#options' => $options,
  '#weight' => 20
);

```

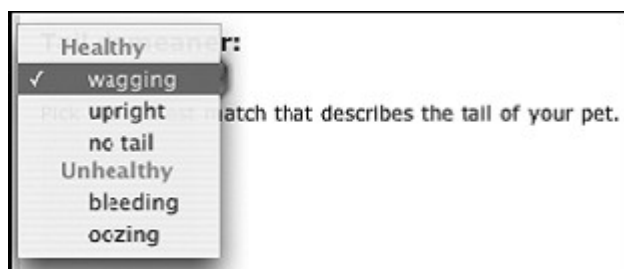


图10-11 使用分组的下拉选择框

元素 `textarea` 常用属性如下：
#attributes、#default_value、#description、#multiple、#options、#prefix、#required、#suffix、#title 和 #weight。

提示：对于拥有选项的元素（包括select、radios、checkboxes），如果用户为其提交的数值不在最初选项列表中的话，Drupal 将自动抛出一个验证错误。这是一个安全特性。然而，

在一些特定场合，你可能需要绕过这一点（比如一个下拉选择框带有一个名为“其它”的选项，当用户选择它时，弹出一段 Javascript 代码从而允许用户直接输入一个值）。在这些情况下，在你的表单元素中，将 #DANGEROUS_SKIP_CHECK 设置为 TRUE。单词“dangerous”太些的原因：对用户输入总要格外小心的。

Radio 按钮

Block.module 中的元素 radio 按钮的示例如下：

```
$form['user_vis_settings']['custom'] = array(
  '#type' => 'radios',
  '#title' => t('Custom visibility settings'),
  '#options' => array(
    t('Users cannot control whether or not they see this block.'),
    t('Show this block by default, but let individual users hide it.'),
    t('Hide this block by default but let individual users show it.')
  ),
  '#description' => t('Allow individual users to customize the visibility of this block in their account settings.'),
  '#default_value' => $edit['custom'],
);
```

元素 radio 常用属性如下：#attributes、#default_value、#description、#options、#prefix、#required、#suffix、#title 和 #weight。注意 #process 属性默认设为 expand_radios()（参看 includes/form.inc）。

Checkboxes

元素 checkboxes 按钮的示例如下。该元素的呈现版本如图10-12所示。

```
$options = array(
  'poison' => t('Sprays deadly poison'),
  'metal' => t('Can bite/claw through metal'),
  'deadly' => t('Killed previous owner') );
$form['danger'] = array(
  '#title' => t('Special conditions'),
  '#type' => 'checkboxes',
  '#description' => (t('Please note if any of these conditions apply to your pet.')),
  '#options' => $options,
  '#weight' => 25
);
```

Special conditions:

- ☐ Sprays deadly poison
- ☐ Can bite/claw through metal
- ☐ Killed previous owner

Please note if any of these conditions apply to your pet.

图10-12 元素 checkboxes 示例图

元素 checkboxes 常用属性如下：
#attributes、#default_value、#description、#options、#prefix、
#required、#suffix、#title、#tree（默认为 TRUE）和 #weight。注意 #process 属性默认设为
expand_checkboxes()（参看 includes/form.inc）。

Value

表单元素 value 用来在 drupal 内部将数值从 \$form 传递到 \$form_values，而不需要将其发送到浏览器端，示例如下：

```
$form['pid'] = array(  
  '#type' => 'value',  
  '#value' => 123  
);
```

不要混淆了 type = '#value' 和 #value = 123。前者声明了元素的类型，而后者声明了元素的值。在前面的例子中，表单提交后 \$form_values['pid'] 将等于123。

表单元素 value 只有属性 #type 和 #value 可用。

Hidden

该元素使用一个 HTML 隐藏域将一个隐藏值传递到一个表单中，示例如下：

```
$form['step'] = array(  
  '#type' => 'hidden',  
  '#value' => $step  
);
```

如果你想在表单中传递一个隐藏值的话，通常使用表单元素 value 会更好一些，只有当表单元素 value 不能满足需求时才使用表单元素 hidden。这是因为用户可以通过网页表单的 HTML 源代码来查看表单元素 hidden，而表单元素 value 位于 Drupal 内部而不包含在 HTML 中。

表单元素 hidden 只有属性#type、#value、#prefix 和 #suffix 可用。

Date

元素date，如图10-13所示，它是一个由3个下拉选择框符合而成的元素：

```
$form['deadline'] = array(  
  '#title' => t('Deadline'),  
  '#type' => 'date',  
  '#description' => t('Set the deadline.'),  
  '#default_value' => array(  
    'month' => format_date(time(), 'custom', 'n'),  
    'day' => format_date(time(), 'custom', 'j'),  
    'year' => format_date(time(), 'custom', 'Y')  
  )  
);
```



图10-13 表单元素 date

元素 date 常用属性如下：#attributes、#default_value、#description、#prefix、#required、#suffix、#title 和 #weight。属性 #process 默认设为 expand_date()，在该方法中年选择器被硬编码为从1900到2050。属性 #validate 默认设为 date_validate()（两个方法都位于 includes/form.inc 中）。当在你的表单中定义表单元素 date 时，你可以通过定义这些属性来使用你自己的代码以代替默认的。

Weight

表单元素 weight（不要与属性 #weight 混淆了）是一个用来声明重量的下拉选择框：

```
$form['weight'] = array('#type' => 'weight',
'#title' => t('Weight'),
'#default_value' => $edit['weight'],
'#delta' => 10,
'#description' => t('In listings, the heavier vocabularies will sink and the lighter vocabularies will
be positioned nearer the top.'),
);
```

前面代码的显示结果如图10-14所示。

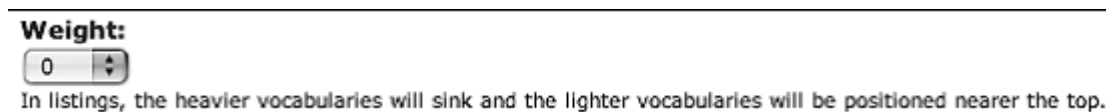


图10-14 表单元素weight

属性 #delta 决定了重量可供选择的范围，默认为10。例如，如果你将 #delta 设为50，那么重量的范围就应该为从-50到50。元素 weight 常用属性如下：#attributes、#delta（默认为10）、#default_value、#description、#prefix、#required、#suffix、#title 和 #weight。

File

表单元素 file 创建了一个上传文件的接口。下面是一个来自于 user.module 的示例：

```
$form['picture']['picture_upload'] = array(
'#type' => 'file',
'#title' => t('Upload picture'),
'#size' => 48,
'#description' => t('Your virtual face or picture.')
);
```

本元素的显示方式如图10-15所示。



图10-15 表单元素 file

注意，如果你使用了表单元素 file，那么你需要在你表单的根部设置属性 enctype：

```
$form['#attributes']['enctype'] = 'multipart/form-data';
```

元素 file 常用属性如下：
#attributes、#default_value、#description、#prefix、#required、#size（默认为60）、#suffix、#title 和 #weight。

Fieldset

表单元素 fieldset 是用来对其它表单元素进行归类分组的。可将其声明为可伸缩的，这样当用户查看表单时点击字段集（fieldset）的标题，由 Drupal 自动提供的 Javascript 将用来动态的打开和关闭 fieldset。注意，在本例中属性 #access 用来允许或者拒绝对 fieldset 内部的所有字段的访问：

```
// Node author information for administrators
```

```
$form['author'] = array(  
  '#type' => 'fieldset',  
  '#access' => user_access('administer nodes'),  
  '#title' => t('Authoring information'),  
  '#collapsible' => TRUE,  
  '#collapsed' => TRUE,  
  '#weight' => 20,  
);
```

元素 fieldset 常用属性如下：#attributes、#collapsed（默认为FALSE）、#collapsible（默认为FALSE）、#description、#prefix、#suffix、#title 和 #weight。

Submit<、strong>

表单元素 submit 是用来提交表单。按钮内部展示的单词默认为“Submit”，但是可以使用属性 #value 来修改它：

```
$form['submit'] = array(  
  '#type' => 'submit',  
  '#value' => t('Continue'),  
);
```

元素 submit 常用属性如下：#attributes、#button_type（默认为submit）、#executes_submit_callback（默认为TRUE）、#name（默认为op）、#prefix、#suffix、#value 和 #weight。

Button

表单元素 button 除了属性 #executes_submit_callback 默认为 FALSE 以外，其它与元素 submit 完全相同。属性 #executes_submit_callback 决定 Drupal 是不是要处理表单，为 TRUE 时处理表单，为 FALSE 时则简单的重新显示表单。

Markup

如果没有设置属性 #type 的话，表单元素 markup 就是默认的元素类型了。它用来在表单中

间引入一段文本或者一段 HTML 代码。

```
$form['disclaimer'] = array(
  '#prefix' => "
  '#value' => t('The information below is entirely optional.'),
  '#suffix' => "
);
```

元素 markup 常用属性如下：#attributes、#prefix（默认为空字符串"）、#suffix（默认为空字符串"）、#value 和 #weight。

警告：如果你在一个可伸缩的字段集内部输出文本的话，使用 标签对其包装，如同例子中所展示的，这样当字段集被压缩以后，你的文本也一同被隐藏了。

Item

表单元素 item 的格式与其它输入表单元素比如 textfield 或 select 的格式相同，但是它缺少输入框。

```
$form['removed'] = array(
  '#title' => t('Shoe size'),
  '#type' => 'item',
  '#description' => t('This question has been removed because the law prohibits us from asking your shoe size.')
);
```

上面元素呈现出来如图10-16所示。

Shoe size:
This question has been removed because the law prohibits us from asking your shoe size.

图10-16 表单元素 item

元素 item 常用属性如下：#attributes、#description、#prefix（默认为空字符串"）、#required、#suffix（默认为空字符串"）、#title、#value 和 #weight。

总结

读完本章后，你应该理解一下概念：

- 表单 IPI 如何工作的

- 创建简单的表单

- 使用主题函数修改表单外观

- 为表单或者独立的表单元素编写一个验证函数

- 编写一个提交函数并在表单处理完后进行重定向

- 修改已存在的表单

- 编写跨页面表单（多页面表单，或者多步表单）

- 你可以使用的表单定义属性和它们的含义

- Drupal中的表单元素（textfield、select、radio、checkboxes 等等）

关于表单的更多信息，包括各种提示和技巧，参看 Drupal 参考手册 <http://drupal.org/node/37775>。

第11章 Drupal 过滤器系统（filter）（2）创建一个定制的过滤器

创建一个定制的过滤器

当然，Drupal 过滤器可以添加超链接，格式化你的内容，将文本转化为空中的 pirate-speak，但是它能够聪明的帮我们写日志么，或者至少能够帮我们把我们创造性的火花碰撞出来么？当然，它也可以做到这一点！让我们创建一个带有过滤器的模块，用来向日志中插入随机的句子。我们将启用这一模块，这样当你在编写文章毫无灵感需要一些火花时，你可以简单的键入[juice!]，当你保存文章时，它将会被替换为一个随机生成的句子。如果你需要更多的智慧火花时，你可以在一个页面中多次插入[juice!]标签。

在 sites/all/modules/custom/ 下面创建一个名为 creativejuice 的文件夹。首先，向 creativejuice 文件夹下面添加 creativejuice.info 文件：

```
; $Id$

name = Creative Juice
description = Adds a random sentence filter to content.
version = $Name$
```

接着，创建 creativejuice.module 文件并也将其添加到 creativejuice 文件夹下：

```
<?php

// $Id$
/**
 * @file
 * A silly module to assist whizbang novelists who are in a rut by providing a
 * random sentence generator for their posts.
 */

hook_filter()
```

现在我们已经为创建模块做好了铺垫，让我们在 creativejuice.module 中添加 hook_filter() 的实现：

```
/**
 * Implementation of hook_filter().
 */
function creativejuice_filter($op, $delta = 0, $format = -1, $text = "") {
  switch ($op) {
    case 'list':
      return array(0 => t('Creative Juices filter'));
    case 'description':
      return t('Enables users to insert random sentences into their posts.');
```

case 'settings':

```
// No settings user interface for this filter.
break;
```

```

case 'no cache':
return FALSE;
case 'prepare':
return $text;
case 'process':
return preg_replace_callback("/\[juice!\]\|i", 'creativejuice_sentence', $text);
default:
return $text;
}
}

```

过滤器 API 将穿过多个阶段，从收集过滤器的名称，到缓存，再到进行真实操作的处理阶段。通过检查 `creativejuice_filter()`，让我们看一下这些阶段或者操作。下面是对这个钩子函数中的参数的分解：

\$op：要进行的操作。在接下来的部分我们将详细的讨论它。

\$delta：`hook_filter()` 中可实现多个过滤器。你可以使用它来追踪当前执行的过滤器的 ID。**\$delta** 是一个整数。

\$format：一个整数，表示使用的哪一个输入格式。

\$text：将被过滤的内容。

根据 **\$op** 参数的不同，可进行不同的操作。

列表操作 (The list \$op)

列表操作 (list) 返回的是一个关联数组，其中以数字为键以过滤器名称为值，这是由于单个 `hook_filter()` 钩子实例中可以声明多个过滤器的缘故。这些键可在接下来的操作中使用，我们使用 **\$delta** 参数来将它们传递给钩子。

```

case 'list':
return array(
0 => t('Creative Juices filter'),
1 => t('The name of my second filter'),
);

```

描述操作 (The description \$op)

该操作返回了一个关于过滤器能做什么的简单描述。只有具有管理过滤器权限的用户才可看到这些描述。

```

case 'description':
switch ($delta) {
case 0:
return t('Enables users to insert random sentences into their posts.');
```

```

case 1:
return t('If this module provided a second filter, the description for that second filter would go here.');
```

```

// Should never reach here as value of $delta never exceeds

```

```
// the last index of the 'list' array.
default:
return;
}
```

设置操作 (The settings \$op)

当过滤器需要一个用于配置的表单接口时，使用该操作。它返回用于控制的 HTML 表单，当表单提交时，将会使用 `variable_set()` 自动的将数据保存起来。这意味在取回数据时使用 `variable_get()`。使用该操作的实例，可参看 `modules/filter/filter.module` 中的 `filter_filter()`。

禁用缓存操作 (The no cache \$op)

过滤器系统应该绕过对已过滤文本进行缓存操作么？如果要禁用缓存，那么返回的代码应该为 `TRUE`。当你开发过滤器时，你需要禁用缓存，这样调试起来就会很容易。如果你修改禁用缓存操作所返回的布尔值，在生效以前你需要对使用到了你的过滤器的输入格式进行编辑。

警告：禁用单个过滤器的缓存，会把任何使用了该过滤器的输入格式的缓存清空。

准备操作 (The prepare \$op)

对内容的实际过滤流程包含两步。首先，允许过滤器准备用于处理的文本。该步的主要目的是将 HTML 转化为相应的实体。例如，有这样一个过滤器，它可让用户粘贴代码片段。在准备阶段会将代码转化为 HTML 实体，这样就可以阻止接下来的过滤器对这些进行操作了。例如，如果没有这一步的话，HTML 过滤器将会清除掉里面的所有 HTML 标签。下面是 `codefilter.module` 中使用准备阶段的例子，该模块用于处理 `<?php ?>` 标签，从而允许用户发布代码而不用担心过滤它内部的 HTML 实体：

```
case 'prepare':

// Note: we use the bytes 0xFE and 0xFF to replace < > during the
// filtering process.
// These bytes are not valid in UTF-8 data and thus least likely to
// cause problems.
$text = preg_replace('@(.+?)@se', "'\xFEcode\xFF'. codefilter_escape('\1') .'\xFE/code\xFF'",
$text);
$text = preg_replace('@[<](\?php|%)(.+?)(\?|%)>@se', "'\xFEphp\xFF'.
codefilter_escape('\2') .'\xFE/php\xFF'", $text);
return $text;
```

处理操作 (The process \$op)

从准备操作阶段里面返回的结果将被传递到 `hook_filter()` 中来。在这里进行实际的本文处理：为 URL 添加超链接，删除恶意数据，添加单词定义，等等。在准备操作和处理操作阶段返回的都应该为 `$text`。

默认操作 (The default \$op)

包含默认操作情况非常重要。如果你的模块没有实现一些操作时，将调用该操作，要保证在这里返回的是 `$text`（你的模块要过滤的文本）。

助手函数

当 \$op 为“process”时，每次遇到[juice!]标签你都要执行一个名为 creativejuice_sentence() 的助手函数。将该函数也添加到 creativejuice.module 中。

```
/**
 * Generate a random sentence.
 */
function creativejuice_sentence() {
  $phrase[0][] = t('A majority of us believe');
  $phrase[0][] = t('Generally speaking,');
  $phrase[0][] = t('As times carry on');
  $phrase[0][] = t('Barren in intellect,');
  $phrase[0][] = t('Deficient in insight,');
  $phrase[0][] = t('As blazing blue sky poured down torrents of light,');
  $phrase[0][] = t('Aloof from the motley throng,');
  $phrase[1][] = t('life flowed in its accustomed stream');
  $phrase[1][] = t('he ransacked the vocabulary');
  $phrase[1][] = t('the grimaces and caperings of buffoonery');
  $phrase[1][] = t('the mind freezes at the thought');
  $phrase[1][] = t('she reverted to another matter');
  $phrase[1][] = t('he lived as modestly as a hermit');
  $phrase[2][] = t('through the red tape of officialdom. ');
  $phrase[2][] = t('as it set anew in some fresh and appealing form. ');
  $phrase[2][] = t('supported by evidence. ');
  $phrase[2][] = t('as fatal as the fang of the most venomous snake. ');
  $phrase[2][] = t('as full of spirit as a gray squirrel. ');
  $phrase[2][] = t('as dumb as a fish. ');
  $phrase[2][] = t('like a damp-handed auctioneer. ');
  $phrase[2][] = t('like a bald ferret. ');
  foreach ($phrase as $key => $value) {
    $rand_key = array_rand($phrase[$key]);
    $sentence[] = $phrase[$key][$rand_key];
  }
  return implode(' ', $sentence);
}
```

hook_filter_tips()

你使用 creativejuice_filter_tips() 来向终端用户显示帮助信息。默认情况下，展示一个短的消息并带有一个指向 <http://example.com/?q=filter/tips> 的链接，链接页面包含了所有过滤器的详细说明。

```
/**
 * Implementation of hook_filter_tips().
 */
function creativejuice_filter_tips($delta, $format, $long = FALSE) {
  return t('Insert a random sentence into your post with the [juice!] tag.');
```

```
}
```

在前面的代码中，无论是简洁的帮助文本还是详细的帮助文本，都是用相同的文本信息，如果你想返回一个更详细的解释信息时，你需要使用 \$long 进行切换，如下所示：

```
/**
 * Implementation of hook_filter_tips().
 */
function creativejuice_filter_tips($delta, $format, $long = FALSE) {
if ($long) {
// Detailed explanation for example.com/?q=filter/tips page.
return t('The Creative Juices filter is for those times when your brain is incapable of being creative. These time comes for everyone, when even strong coffee and a barrel of jelly beans does not create the desired effect. When that happens, you can simply enter the [juice!] tag into your posts...');
}
else {
// Short explanation for underneath a post's textarea.
return t('Insert a random sentence into your post with the [juice!] tag.');
```

```
}
}
```

一旦在模块列表页面启用了这个模块，那么就可以使用 creativejuice 过滤器了，你可以将它用在一个已存在的输入格式中，也可以用到新建的输入格式中。你可以使用合适的输入格式创建一个日志，然后提交包含[juice!]标签的文本：

Today was a crazy day. [juice!] Even if that sounds a little odd,it still doesn't beat what I heard on the radio. [juice!]

提交的内容将会转化为如下所示的内容：

Today was a crazy day! Generally speaking, life flowed in its accustomed stream through the red tape of officialdom. Even if that sounds a little odd, it still doesn't beat what I heard on the radio. Barren in intellect, she reverted to another matter like a damp-handed auctioneer.

防止恶意数据

如果你想防止恶意 HTML，可以使用 Filtered HTML 过滤器对内容进行过滤，从而阻止 XSS 攻击。如果在一些情况下，你不能使用 Filtered HTML 过滤器时，你可以使用下面的方式手工的过滤 XSS：

```
function mymodule_filter($op, $delta = 0, $format = -1, $text = "") {
switch ($op) {
case 'process':
// Decide which tags are allowed.
$allowed_tags = '
';
```

```
return filter_xss($text, $allowed_tags);
default:
return $text;
break;
}
}
```

总结

读完本章后，你应该可以：

- 理解什么是过滤器，什么是输入格式，以及如何使用它们转化文本
- 理解为什么过滤器系统比使用其它钩子直接操纵文本更加有效
- 理解输入格式和过滤器的工作原理
- 创建一个定制的过滤器
- 理解过滤器中的各种操作函数

第11章 Drupal过滤器系统（filter）（1）

操纵用户输入：过滤器系统

如果你需要自己格式化输入信息的话，那么向网站添加内容将是一个繁琐的工作。相反，如果想让网站上输入文本有个好看的外观的话，那么你需要懂得 HTML – 但是大多数用户都不了解这一知识。对于我们这些了解 HTML 的人，如果在头脑风暴会议或者文章创作期间，不断的停下来并向要发布的文章中插入 HTML 标签的话，这也是件令人头痛的事情。段落标签、链接标签、换行标签……太烦了。幸好，Drupal 提供了预建的称为过滤器的程序，使得数据输入更加容易并且有效。过滤器进行文本操作，比如为 URL 添加链接，将换行符转化为

和

标签，甚至包括过滤有害的 HTML。钩子函数 `hook_filter()` 负责创建钩子和操纵用户提交的数据。大多数过滤器通常负责一个操作，比如“取消所有的超链接”、“为这篇文章添加一个随机图片”，或者甚至“将它翻译为 pirate-speak”（参看 `pirate.module` <http://drupal.org/project/pirate>）。

过滤器和输入格式

假定你已经理解了过滤器是做什么的，而且你知道自己想找的是已安装的过滤器列表，你还是不会很容易的在后台管理接口找到它。为了让过滤器履行它们的工作，你必须将其指定到如图11-1所示的 Drupal 输入格式上。输入格式将过滤器组合在一起，这样当内容提交时可以一起运行它们。这比为每一个提交的内容选择一些过滤器要容易很多。为了查看已安装的过滤器列表，你可以到 Administer > Site configuration > Input formats 里面，通过配置一个已存在的输入格式或者新建一个输入格式来查看。

提示：一个 Drupal 输入格式由一组过滤器构成。

Drupal 自带了3中输入格式（参看图11-2）：

Filtered HTML：由3个过滤器构成：HTML 过滤器，用于过滤 HTML 标签以阻止跨站脚本

攻击（通常简称为 XSS）；Line break converter 过滤器，用于将回车转换为它们的 HTML 对应标签；URL 过滤器，用于将 web 链接和 e-mail 地址转化为超链接。

Full HTML：它不对 HTML 进行限制，但是它使用了 Line break converter 过滤器。

PHP Code：有一个名为 PHP evaluator 的过滤器构成，它负责执行发布节点里面的 PHP 脚本。一般情况下，不要让用户能够选用使用 PHP evaluator 的输入格式。如果用户可以运行 PHP 的话，那么它们可以使用 PHP 做任何 PHP 能做到的事情，包括让你的站点当掉，或者更坏的是删除了你的所有数据。

警告：在你的站点上对任何用户启用 PHP Code 输入格式都会带来安全隐患。一个好的原则是尽量不要使用这一输入格式，能不用就不用，能少用就少用，而且只有超级用户才可以使用（用户 ID 为1的用户）。

Input formats

List**Add input format**

Every *filter* performs one particular change on the user input, for example stripping out malicious HTML or making URLs clickable. Choose which filters you want to apply to text in this input format.

If you notice some filters are causing conflicts in the output, you can **rearrange them**.

Name: *

Specify a unique name for this filter format.

Roles

Choose which roles may use this filter format. Note that roles with the "administer filters" permission can always use all the filter formats.

☐ anonymous user

☐ authenticated user

Filters

Choose the filters that will be used in this filter format.

☐ HTML filter

Allows you to restrict if users can pos: HTML and which tags to filter out.

☐ Line break converter

Converts line breaks into HTML (i.e.
 and <p> tags).

☐ PHP evaluator

Runs a piece of PHP code. The usage of this filter should be restricted to administrators only!

☐ URL filter

Turns web and e-mail addresses into clickable links.

Save configuration

图11-1. 位于“Add input format”（“添加输入格式”）表单中的已安装的过滤器列表

Default	Name	Roles	Operations
<input checked="" type="radio"/>	Filtered HTML	All roles may use default format	configure
<input type="radio"/>	PHP code	No roles may use this format	configure delete
<input type="radio"/>	Full HTML	No roles may use this format	configure delete

[Set default format](#)

图11-2 Drupal 自带了3种可配置的输入格式

因为输入格式是一组过滤器，所以可对它们进行扩展。你可以添加或者删除过滤器，如图11-3所示。你可以修改输入格式的名称，删除一个过滤器，或者 甚至可以重排输入格式中过滤器的执行顺序从而避免冲突。例如，你想在运行 HTML 过滤器之前运行 URL 过滤器，这样 HTML 过滤器就可以检查由 URL 过滤器生成的 标签了。

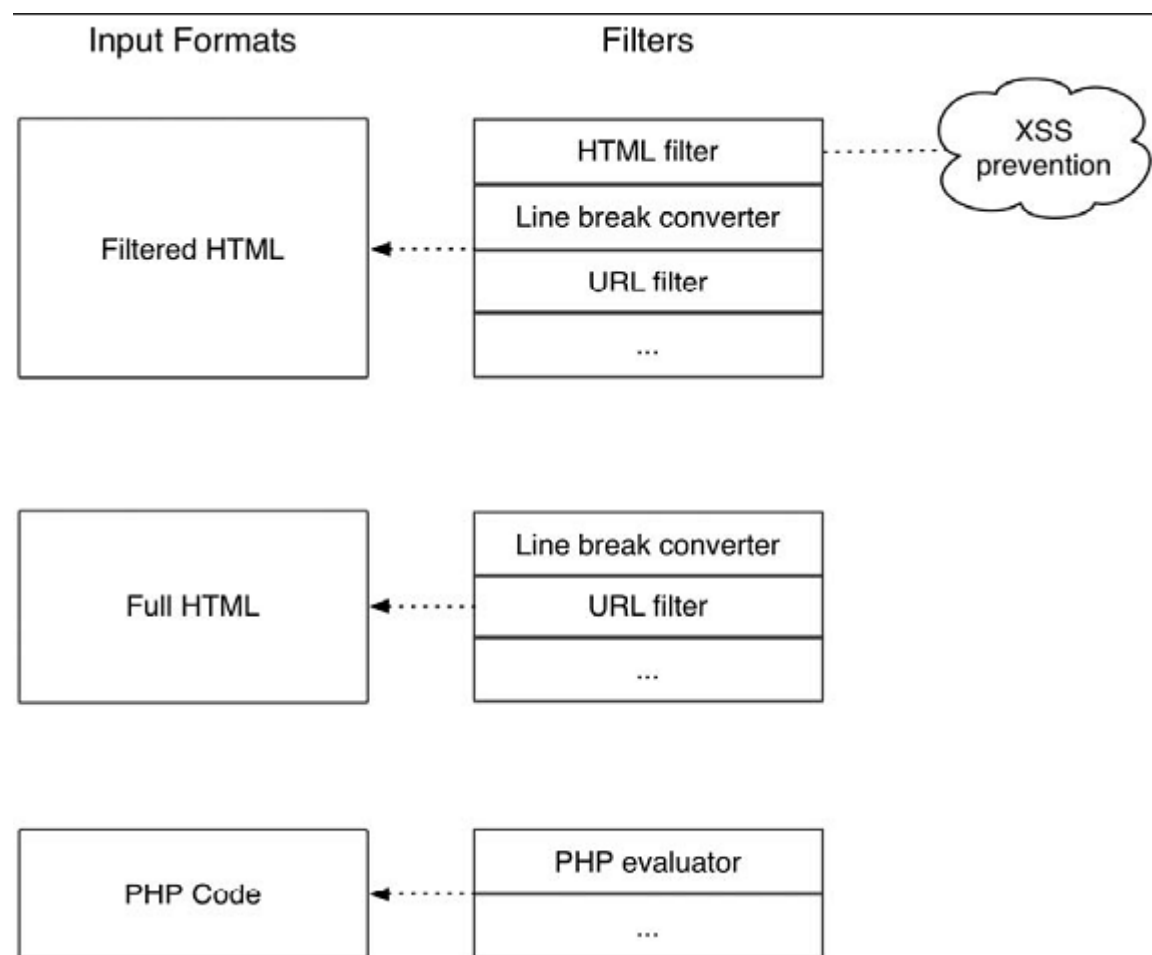


图11-3. 输入格式由一组过滤器构成。本土展示了 Drupal 默认输入格式的构成，可以扩展这些输入格式格式

注意：输入格式（过滤器组）可以通过管理接口层面进行控制。开发者在定义过滤器的时候不用考虑输入格式。该工作由 Drupal 站点管理员负责。

安装过滤器

安装过滤器与安装模块的流程是一样的，如图11-4所示，这是因为过滤器位于模块文件中对过滤器的启用或禁用，只需要在 Administer > Site building > Modules 下面启用或者禁用相应的模块就可以了。一旦安装了过滤器，就可以导航到 Administer > Site configuration > Input formats，以将新填的过滤器指定到你选定的输入格式中。

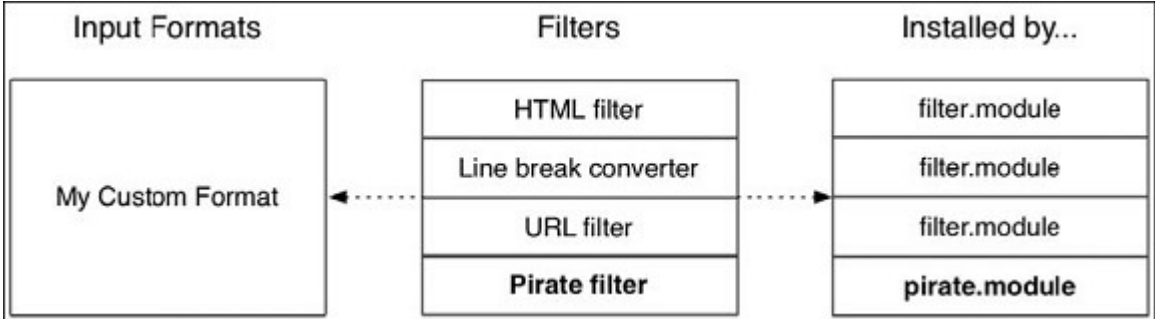


图11-4 在模块中创建的过滤器

知道什么时候使用过滤器

如果你可以使用其他钩子来操纵文本，那么你可能会想为什么这时候还需要过滤器呢。例如可以非常容易的使用 `hook_nodeapi()` 为 URL 添加超链接，这比 URL 过滤器还好用。但是考虑一下这种情况，你需要为节点的主体字段使用5个不同的过滤器。现在假定你查看默认页面 <http://example.com/?q=node>，它一次展示10个节点。那么算一下，现在为了展示这个页面你就需要运行50个过滤器，而对文本的过滤操作是很费资源的。这同时也意味着无论什么时候展示一个节点，都需要运行这些过滤操作，即便是文本没有被修改的情况。你在一次又一次的无谓的运行这一操作。

过滤器系统有一个缓存层，极大的提升了性能。一旦为给定的文本片段运行了所有的过滤器以后，该文本的过滤后的版本将被存储在 `cache_filter` 表中，在文本被再次修改以前缓存的内容不变（使用过滤文本的 MD5 哈希值来判断是否被修改）。现在让我们回到前面的例子中，当文本没有被修改时，我们就可以绕过过滤器直接从缓存表中加载10个节点的数据了---速度快多了！图11-5给出了过滤器系统处理流程的概貌。

提示：MD5 是一个用来计算文本字符串的哈希值的算法。Drupal 使用它来得到一个高效的数据库中索引列，用来查找节点过滤后的数据。

现在你应该有了比刚才更聪明的想法了，“好吧，如果在我们的 `nodeapi` 钩子函数中直接将过滤后的文本保存到 `node` 表中,那不更好么？它的运行结果和过滤器系统可是一样啊。”尽管这一方法也解决了性能问题，但是你破坏了 Drupal 架构的一个基本原则：永不修改用户的原始数据。假定你的一个初级用户回过来想编辑一个已发布的节点时，当他看到很多内容都被隐藏了的时候，我想他十有八九 会向你打电话寻求支持的。过滤器系统的目标是保持原始数据不变，而在 Drupal 框架中使用缓存了的数据过滤后的副本。你将会在其它的 Drupal API 中一次又一次的看到这一原则。

注意：即便是在 Drupal 中仅用了页面缓存，过滤器系统仍将对它的数据进行缓存。如果你看到的还是以前的内容时，而看不到新加的内容时，可以试着清空 `cache_filter` 表。

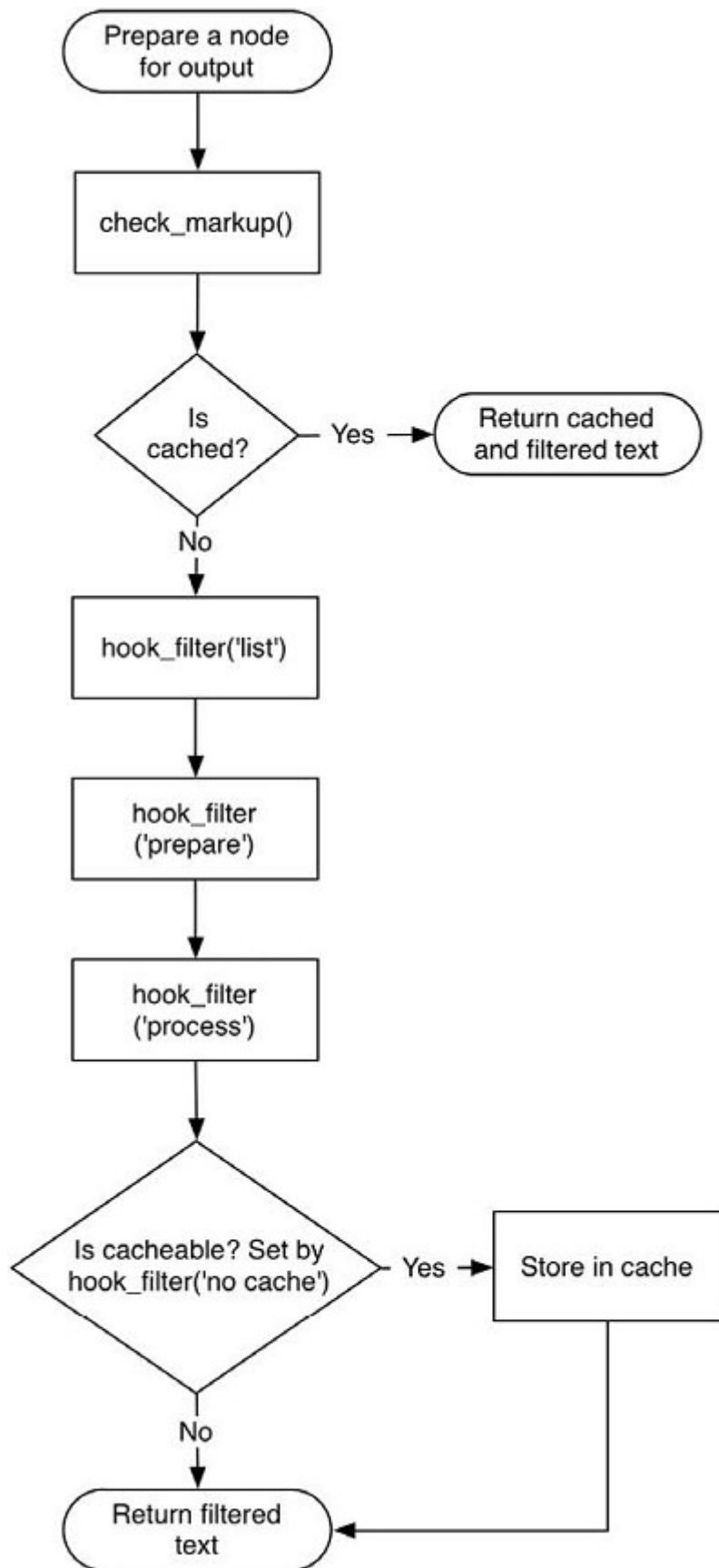


图11-5 文本过滤系统的生命周期

第12章 搜索和索引内容 (1)

搜索和索引内容

MySQL 和 PostgreSQL 都有内置的全文搜索能力。你可以很容易的使用这些数据库特定解决方案来建立一个搜索引擎，但你失去了对搜索机制的控制，同时也不能让你的搜索系统与你的应用完全匹配。而且有时候数据库认为优先级比较高的词语，而实际上在你的应用中则被认为是“噪音”。

由于数据库全文搜索不能很好的满足应用需求，Drupal 社区建立一个定制的搜索引擎，来实现针对 Drupal 的索引和页面等级算法。结果就建立了与 Drupal 其他框架部分相一致的搜索引擎，它具有标准的配置和用户接口 - 不用担心底层使用了什么数据库。

在本章我们讨论了如何使用模块编写搜索 API 的钩子函数和构建定制的搜索表单。我们还将学习一下 Drupal 是如何解析和索引内容的，还有就是如何编写索引器的钩子函数。

提示：Drupal 能够理解复杂的查询语句，比如包含布尔操作 and/or，精确短语，或者甚至可以排除每个词语。这些情况的一个实际例子如下所示：

Beatles OR John Lennon "Penny Lane" -insect

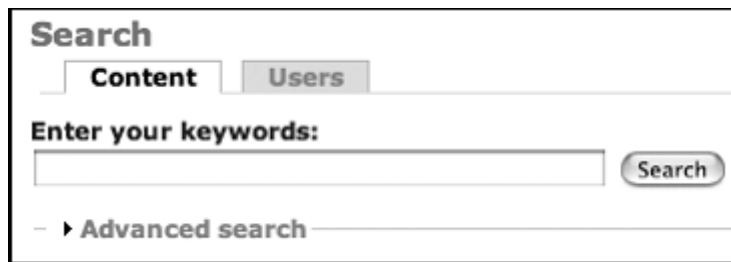
构建一个定制的搜索页面

Drupal 可对节点和用户名进行搜索。即使你开发了一个你自己定制节点类型，Drupal 也可以对其节点视图中显示的内容进行索引。例如，假定你有一个 recipe（处方）节点类型，它包含两个字段 ingredients（药物成分）和 instructions（用法说明）；你创建一个新的处方节点，其节点 ID 为 22。当有匿名用户访问 <http://example.com/?q=node/22> 时就可以看到这些节点字段，搜索模块将在它下次访问 <http://example.com/cron.php> 时（一般通过一个 cron 周期性运行它），将会对处方节点的内容和其它元数据进行索引。

Drupal 默认提供了节点搜索和用户搜索。开始你可能会觉得节点搜索和用户搜索的底层机制是一样的，事实上它们使用了两种独立的方式来实现搜索功能。对于节点搜索，每次所搜都没有直接对 node 表进行查询，它首先使用一个索引器把内容预处理为一种结构化的格式，在节点搜索时，将会对结构化的索引数据进行查询，这样就会产生更快更准确的结果。我们在下面的部分将会详细的介绍索引器。

用户搜索一点也不复杂，这是因为用户名只是数据库中的一个单独字段，搜索语句只需要对该字段进行检查就可以了。而且，用户名中不允许包含 HTML，所以你不需要使用 HTML 索引器。相反，你只需要几行代码直接对 user 表进行查询就可以了。

让我们看一个例子。假定我们的站点使用了 path.module，我们需要对数千个 URL 别名进行管理，这使得现有的 URL 别名管理页面显得非常笨拙。我们将编写一个搜索接口来快速的寻找我们想找的东西。幸运的是你可以使用搜索 API 提供的默认搜索表单（如图 12-1 所示）。如果这个接口能够满足你的需求，那么你只需要编写为搜索请求查找采集数（hits）的逻辑就可以了。这一搜索逻辑一般是一个数据库查询语句。



Search

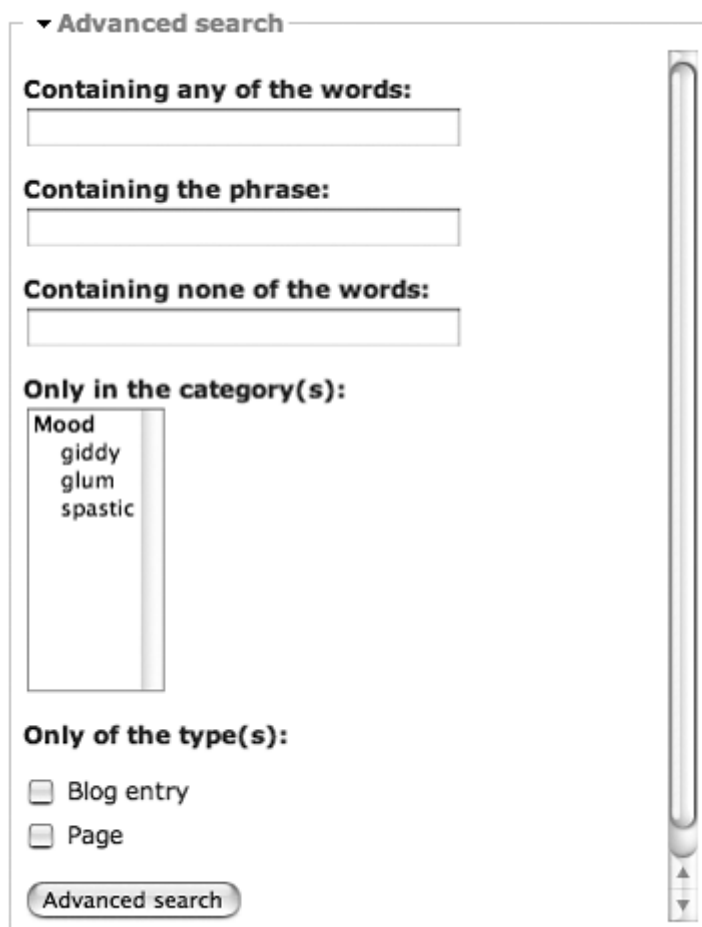
Enter your keywords:

[Advanced search](#)

图12-1 搜索 API 的搜索默认用户接口

它看起来很简单，默认内容搜索表单事实上很强大，它可以对你站点上的节点内容的所有可视的元素进行查询，这是由于索引器的缘故。也就是说，通过该接口，可以对节点的标题、主体、其它定制属性、评论、分类词语进行查询。高级搜索特性，如图12-2所示，是过滤搜索结果的另一种方式。

很有可能你想扩展默认搜索表单以添加额外的搜索字段，你已在第10章学到了如何去扩展明确的，你可以使用 `hook_form_alter()` 来添加和删除表单字段。由于在本章中我们的主题是搜索 API，我们将在这里使用默认的搜索表单。图 12-3展示了我们需要为我们的路径别名搜索模块实现的搜索 API 钩子函数的概貌。



Advanced search

Containing any of the words:

Containing the phrase:

Containing none of the words:

Only in the category(s):

Mood
giddy
glum
spastic

Only of the type(s):

☐ Blog entry
☐ Page

图12-2 有默认搜索表单提供的高级搜索选项

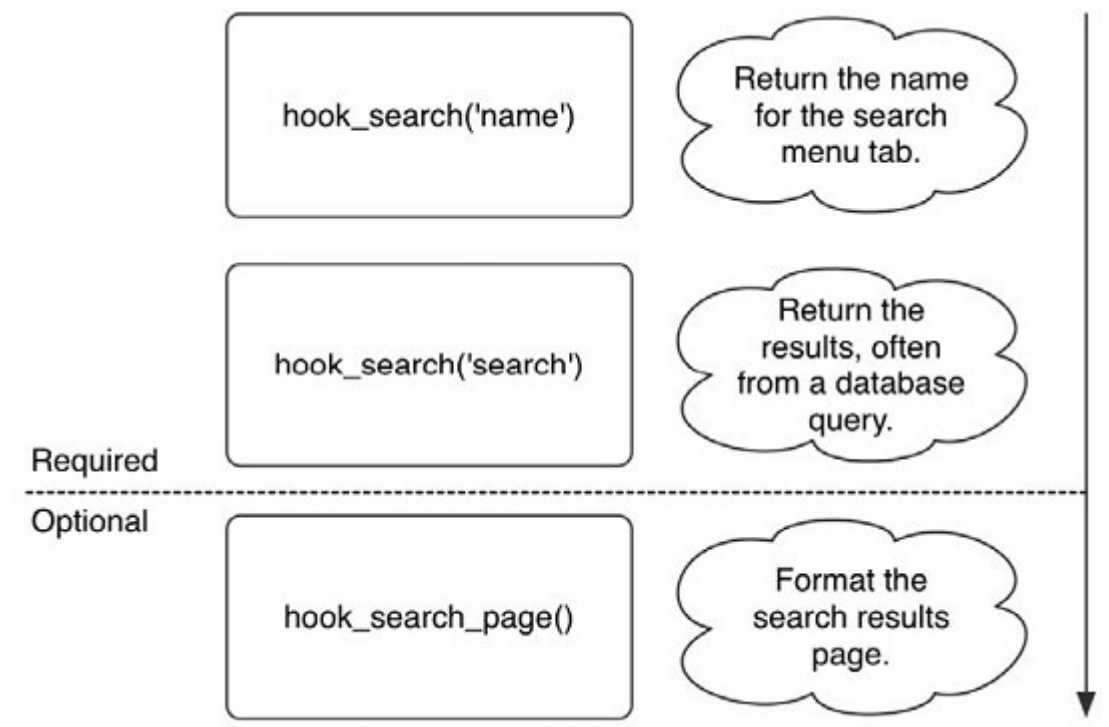


图12-3 为创建一个定制搜索页面，搜索 API 钩子函数的执行周期

注意：在测试这些例子以前，你需要重新构建你的搜索索引数据。你可以这样做：导航到 Administer > Site configuration > Search settings，点击“Re-index site”按钮，接着访问 <http://example.com/cron.php>。

在 `sites/all/modules/custom` 下面创建一个名为 `pathfinder` 的新文件夹，在新目录中创建列表 12-1 和 12-2 所示的文件。

列表12-1 `pathfinder.info`

```

; $Id$

name = Pathfinder
description = Gives administrators the ability to search URL aliases.
version = "$Name$"

```

列表12-2. `pathfinder.module`

```

<?php
// $Id$
/**
 * @file
 * Search interface for URL aliases.
 */

```

不要在你的文本编辑器中关闭 `pathfinder.module`，你将继续使用它。继续，在 Administer > Site building > Modules 中启用该模块。接下来要实现的函数是 `hook_search($op, $keys)`。该钩子函数根据操作（`$op`）参数的不同返回不同的信息。

```

/**

```

```

* Implementation of hook_search().
*/

function pathfinder_search($op = 'search', $keys = NULL) {
switch ($op) {
case 'name':
if (user_access('administer url aliases')) {
return t('URL aliases');
}
case 'search':
if (user_access('administer url aliases')) {
$found = array();
// Replace wildcards with MySQL/PostgreSQL wildcards.
$keys = preg_replace('!\*+!', '%', $keys);
$sql = "SELECT * FROM {url_alias} WHERE LOWER(dst) LIKE LOWER('%%%s%%')";
$result = pager_query($sql, 50, 0, NULL, $keys);
while ($path = db_fetch_object($result)) {
$found[] = array('title' => $path->dst,
'link' => url("admin/path/edit/$path->pid"));
}
return $found;
}
}
}
}

```

当搜索 API 调用 `hook_search('name')` 时，它将寻找显示在通用搜索页面中的菜单标签（tab）上的名字（参看图12-4）。在这里，我们返回的是“URL aliases”。通过返回菜单标签的名字，搜索 API 将为菜单标签的链接创建一个新的搜索表单。正如前面提到的，如果你需要扩展搜索接口的话，你可以使用 `hook_form_alter()`（高级搜索选项就是通过这种方式添加到节点搜索表单上的 - 参看 `node.module` 里面的 `node_form_alter()`）

Search

Content
URL aliases
Users

Enter your keywords:

图12-4 通过从 `hook_search()` 中返回菜单标签的名字，这样就可以访问搜索表单了

`hook_search('search')` 是 `hook_search()` 中干重活的部分。当提交搜索表单时，调用 `hook_search('search')`，它的任务是搜集并返回搜索结果。在其那面的代码中，我们使用表单中提交的搜索词语对 `url_alias` 表进行查询。我们接着将查询的结果搜集到一个数组中并将其返回。继续，测试一下你新的搜索引擎！一定要启用 `search.module` 和 `path.module`，创建一些 URL 别名，然后导航到 <http://example.com/?q=search/pathfinder> 并搜索一个已存在的别名。

注意：由于搜索 API 将提交搜索表单的 POST 请求转化为了 GET 请求，所以用户可以将搜

搜索结果页面添加到收藏夹中。例如，一个对节点的“surfing”字段进行搜索产生的搜索结果页面的 URL 为 `http://example.com/?q=search/node/surfing`，该 URL 可添加到收藏夹中。

让我们转移到搜索结果页面的外观上。如果默认的搜索结果页面与你期望的存在差距的话你可以覆盖它。在我们这里，我们不想把它仅仅展示为一列匹配的 别名，让我们使用一个可排序的表格，每行匹配别名后面带有一个独立的“edit”链接。通过对 `hook_search('search')` 的返回值进行一些修改并实现 `hook_search_page()`，从而完成这一工作。

```
/**
 * Implementation of hook_search().
 */
function pathfinder_search($op = 'search', $keys = NULL) {
  switch ($op) {
    case 'name':
      if (user_access('administer url aliases')) {
        return t('URL aliases');
      }
    case 'search':
      if (user_access('administer url aliases')) {
        $header = array(
          array('data' => t('Alias'), 'field' => 'dst'),
          t('Operations'),
        );
        // Return to this page after an 'edit' operation.
        $destination = drupal_get_destination();
        // Replace wildcards with MySQL/PostgreSQL wildcards.
        $keys = preg_replace('!\*+!', '%', $keys);
        $sql = "SELECT * FROM {url_alias} WHERE LOWER(dst) LIKE LOWER('%%%s%%')";
        tablesort_sql($header);
        $result = pager_query($sql, 50, 0, NULL, $keys);
        while ($path = db_fetch_object($result)) {
          $rows[] = array(l($path->dst, $path->dst), l(t('edit'),
            "admin/build/path/edit/$path->pid", array(), $destination));
        }
        if (!$rows) {
          $rows[] = array(array('data' => t('No URL aliases found.'),
            'colspan' => '2'));
        }
        return $rows;
      }
  }
}

/**
 * Implementation of hook_search_page().
 */
```

```
function pathfinder_search_page($rows) {
  $header = array(
    array('data' => t('Alias'), 'field' => 'dst'), ('Operations'));
  $output = theme('table', $header, $rows);
  $output .= theme('pager', NULL, 50, 0);
  return $output;
}
```

在前面的代码中，我们使用 `drupal_get_destination()` 来取回我们所在页面的当前位置，如果我们点击了链接并编辑一个别名，提交编辑表单以后我们将自动返回到这一搜索结果页面由于返回的路径信息作为编辑链接的一部分传递给了别名编辑表单，所以编辑表单知道要返回到哪个页面。你将在URL中看到一个名为 `destination` 的额外 GET 参数，该参数包含的就是保存表单后所要返回的URL。

为了对结果表格进行排序，我们将 `tablesort_sql()` 函数追加到搜索查询字符串上，从而保证向查询语句后追加正确的 SQL ORDER BY 语句。最后，`pathfinder_search_page()` 是钩子 `hook_search_page()` 的一个实现，它允许我们控制搜索结果页面的输出。图12-5展示了最终的搜索结果页面。

The screenshot shows a web interface for searching URL aliases. At the top, there are three tabs: 'Content', 'URL aliases' (which is selected), and 'Users'. Below the tabs is a search bar with the text 'Enter your keywords:' and a search button. The search bar contains the word 'about'. Below the search bar, the results are displayed under the heading 'Search results'. There is a table with two columns: 'Alias' and 'Operations'. The table contains three rows of results:

Alias	Operations
about/grandpa_ted	edit
about/granny_smith	edit
about/uncle_bob	edit

图12-5 URL 别名搜索的最终搜索页面

第12章 搜索和索引内容 (2) 索引器

使用搜索HTML索引器

索引器的目标是更有效的搜索大量的 HTML 内容。通过运行 `http://example.com/cron.php`（一般使用一个周期性的 cron 调用）处理内容来实现这一目标的。因此，从提交新内容以后，到新内容可被搜索到，基于 cron 运行周期的长短，会有一个时差。索引器解析数据并对文本进行切分成词语（成为分词），基于一组规则集为每个令牌（token）分配一个分数，可以使用搜索 API 扩展这一规则集。它接着将这些数据保存到数据库中，而当请求一个搜索时，它将使用这些索引过的表来代替直接使用节点表。

什么时候使用索引器

当搜索引擎评价大量的数据集时，而且当你的处理要求比标准的“最多词语匹配”查询更多时，一般需要使用索引器。当文件或者其它资源中的文本不是默认的格式时，索引器也用来从这些文本中提取和组织元数据。搜索相关度，指的是使用一组规则（通常很复杂）对

内容进行处理来决定一个索引的匹配等级值。

如果你需要对大量的 HTML 内容进行搜索时，你将需要利用索引器的能力。Drupal 中的最大优点之一就是，日志、论坛、页面等等都是节点。它们的基本数据结构是相同的，而这一联结也意味着它们的基本功能也一样。一个通用的特性就是当启用了搜索模块后将自动对所有的节点进行索引；而不需要额外的编程工作。即便是你创建了一个定制节点类型 Drupal 也会自动对该内容进行搜索。

索引器的工作原理

索引器有一个预处理模式，在该模式下将使用一组规则对文本进行过来从而指定分数。这些规则包括：处理缩略语、URLs 和数字数据。在于处理阶段，其它模块还有机会向该流程中添加逻辑，从而进行它们自己的数据操作。在针对特定语言调整期间，这非常方便，如下所示，这里我们使用了第3方模块 Porter-Stemmer：

```
resumé > resume (accent removal)
```

```
skipping > skip (stemming)
```

```
skips > skip (stemming)
```

其他的这样针对语言预处理的例子有是，为了保证文本被正确索引对汉语、日语、和韩语所进行的切词。

提示：Porter-Stemmer 模块 (<http://drupal.org/project/porterstemmer>) 是一个模块例子，它用来提供词根服务从而提高英语搜索的。同样，汉语切词模块 (<http://drupal.org/project/csplitter>) 是一个加强版的预处理器，用来汉语、日语、和韩语的搜索。在搜索模块中包含了一个简单的汉语切词器，可以在搜索设置页面启用它。

预处理阶段过后，索引器使用 HTML 标签来寻找更重要的字词（称为令牌（tokens）），基于默认的 HTML 标签分数和每个令牌出现的次数来分给它们适当的分数。下面是全部的默认 HTML 标签分数列表（在 `search_index()` 中定义的它们）：

```
= 25
= 18
= 15
= 12
= 10
= 9
= 6
= 3
= 3
= 3
= 3
= 3
```

让我们摘取一段 HTML，然后使用索引器对其处理，从而更好的理解索引器的工作原理。图12-6展示了 HTML 索引器的概貌：解析内容，为令牌分配分数，将信息存储在数据库中。

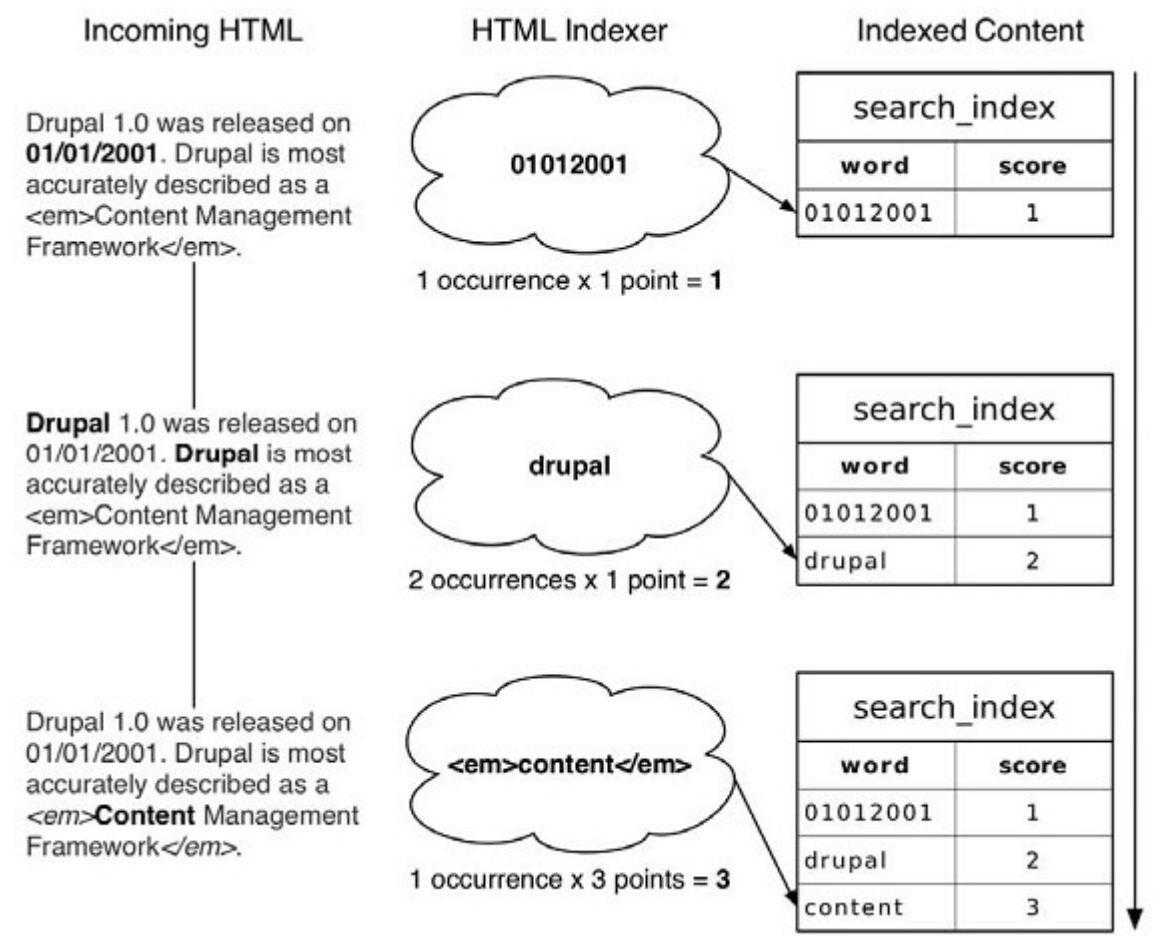


图12-6 对一段 HTML 进行索引并分配分数

当索引器遇到一个由标点符号分隔的数字数据时，它将删除标点符号并只对数字进行索引。这意味着数字元素比如日期、版本号、IP 地址等等将会更容易搜索到。图12-6中的中间步骤展示了如何对一个没有使用 HTML 标签的单词令牌（token）进行处理的。这些令牌的重量为1。最后一行展示了使用强调标签 的内容。决定一个令牌总分的公式如下：

匹配数量 * HTML 标签重量

还需要注意的是 Drupal 对节点过滤后的输出进行索引；例如，如果你有一个输入过滤器，它将 URL 自动转化为了超链接，或者其它的过滤器将换行转化为了 HTML 的
和

标签，索引器将看到这些带有标签的内容，并会根据这些标签分配分数。你可能知道 Drupal 内部有一个可选的 PHP 代码输入过滤器，在一个使用了该过滤器的节点中，可以看到对过滤后的内容进行索引所产生的效果会更加明显。索引动态内容应该非常麻烦，但是由于 Drupal 的索引器只看到 PHP 节点的输出，所以动态 PHP 节点将自动被索引。

当索引器遇到内部链接时，它也将以一种特定方式对它们进行处理。如果一个链接指向了另一个节点，那么连接中的词语将被添加到目标节点的内容中，这使得能够更容易的搜索到常见问题的答案和相关信息。有两种方式能够钩住索引器：

1、为了调整搜索的相关性，你可以向节点中添加在视图中不可见的信息。你可以在 Drupal 内核中看到这方面的实例，对于分类词语和评论，从技术上讲它们不是节点你对象的一部

份，但它们对搜索结果有影响。在索引阶段使用 `nodeapi('update index')` 钩子将这些项目添加到节点中。当仅仅处理节点时你可以再次调用 `hook_nodeapi()`。

2、通过 `hook_update_index()`，你可以使用索引器对不属于节点的 HTML 内容进行索引。Drupal 内核中 `hook_update_index()` 的实现，参看 `modules/node/node.module` 中的 `node_update_index()`。

在 cron 运行期间，为了索引新的数据，将会调用这两个钩子。图12-7展示了这些钩子的运行次序。

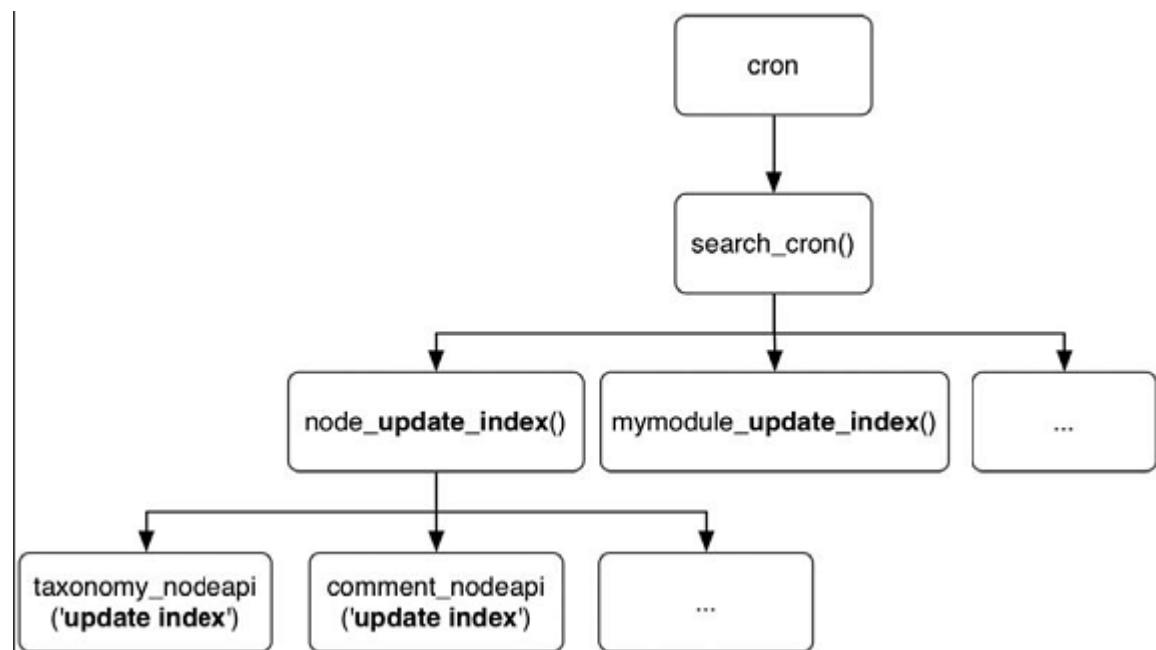


图12-7 HTML 索引钩子的概貌

我们将在接下来的部分更详细的讨论这些钩子

向节点添加元数据：`nodeapi('update index')`

当 Drupal 为了快速搜索对节点进行索引时，它首先使用 `node_view()` 来运行节点对象，从而生成与你在你的浏览器中看的完全一样的输出。这就意味着节点中任何可视的部分都将被索引。例如，假定我们有一个节点，它的 ID 为26。党察看URL <http://example.com/?q=node/26> 时可见得节点部分也都被索引器看到。

如果我们有一个定制节点类型，它包含隐藏的数据，而这些数据对搜索结果有影响，那该怎么办呢？在 `book.module` 中有个很好的现成的例子，我们看看它是如何处理这一点的。我们可以对章节标题和子页面一同索引来提高这些子页面的相关度。

注意：钩子 `nodeapi` 仅用来向节点追加元数据。为了对不是节点的元素进行索引，需要使用 `hook_update_index()`。

```
function book_boost_nodeapi($node, $op, $arg = 0) {  
  switch ($op) {  
    case 'update index':  
      // Book nodes have a parent attribute.  
      if ($node->parent) {
```

```

$parent = node_load($node->parent);
// Boost relevancy by using h2 tags.
return ". $parent->title .";
}
}
}

```

注意，我们在这里对标题使用了 HTML 标签进行封装，从而告诉索引器为这一文本分配一个相对较高的分数。

第12章 搜索和索引内容 (3) 对非节点的内容进行索引

对非节点的内容进行索引 hook_update_index()

在你需要封装搜索以对非 Drupal 节点的内容进行搜索时，你可以使用钩子函数正确的钩住索引器并向其中提供你需要的任何文本数据，从而使得 Drupal 能够搜索到它们。假定你的组织支持一个遗留应用系统，它用来输入和查看最近几年的关于产品的技术文件（note）。由于一些行政原因，你不能完全使用 Drupal 解决方案来代替这一遗留系统，但是你想在 Drupal 内部能够对这些技术文件进行搜索。当然可以。让我们假定遗留系统将它的数据放在了名为 technote 的数据库表中。我们将创建一个简短的模块，它使用 hook_update_index() 将这个数据库中的信息发送给 Drupal 的索引器，使用 hook_search() 将搜索结果展示出来。

注意：如果你想对非 Drupal 的数据库进行索引的话，更多关于连接多个数据库的信息，可参看第5章。

在 sites/all/modules/custom 下面创建一个名为 legacysearch 的文件夹。如果你需要一个用于测试的 legacy（遗留）数据库的话，创建一个名为 legacysearch.install 的文件，并添加以下内容：

```

<?php

// $Id$
/**
 * Implementation of hook_install().
 */
function legacysearch_install() {
  switch ($GLOBALS['db_type']) {
    case 'mysql':
    case 'mysqli':
      db_query("CREATE TABLE technote (
        id int NOT NULL,
        title varchar(255) NOT NULL,
        note text NOT NULL,
        last_modified int NOT NULL,
        PRIMARY KEY (id)
      ) /*!40100 DEFAULT CHARACTER SET UTF8 */");
      db_query("INSERT INTO technote VALUES (1, 'Web 1.0 Emulator', '

```

This handy product lets you emulate the blink tag but in hardware...a perfect gift.

```
', 1172542517)");
```

```
db_query("INSERT INTO technote VALUES (2, 'Squishy Debugger', '
```

Fully functional debugger inside a squishy gel case. The embedded ARM processor heats up...

```
', 1172502517)");
```

```
break;
```

```
case 'pgsql':
```

```
db_query("CREATE TABLE technote (
```

```
id int NOT NULL,
```

```
title varchar(255) NOT NULL,
```

```
note text NOT NULL,
```

```
last_modified int NOT NULL,
```

```
PRIMARY KEY (id)
```

```
) /*!40100 DEFAULT CHARACTER SET UTF8 */);
```

```
db_query("INSERT INTO technote VALUES (1, 'Web 1.0 Emulator', '
```

This handy product lets you emulate the blink tag but in hardware...a perfect gift.

```
', 1172542517)");
```

```
db_query("INSERT INTO technote VALUES (2, 'Squishy Debugger', '
```

Fully functional debugger inside a squishy gel case. The embedded ARM processor heats up...

```
', 1172502517)");
```

```
break;
```

```
}
```

```
}
```

```
/**
```

```
* Implementation of hook_uninstall().
```

```
*/
```

```
function legacysearch_uninstall() {
```

```
db_query('DROP TABLE {technote}');
```

```
}
```

这一模块一般不需要这个安装文件，这是由于遗留数据库已经存在了。我们在这里仅仅用它创建一个遗留数据库表并插入一些测试用的数据。你可以将模块中的查询语句连接到你的已存在的非 Drupal 表。下面的查询语句都假定了数据存放在非 Drupal 数据库中，它使用 settings.php 中 \$db_url['legacy'] 定义的数据库链接。

接着，向 legacysearch.info 添加以下内容：

```
; $Id$
```

```
name = Legacy Search
```

```
description = Enables searching of external content within Drupal.
```

```
version = "$Name$"
```

最后，在 legacysearch 目录下创建 legacysearch.module，并向其添加以下代码：

```
<?php
// $Id$
/**
 * @file
 * Enables searching of non-Drupal content.
 */
```

继续，不要关闭你的文本编辑器中的 legacysearch.module，我们将向它添加 hook_update_index()，从而将遗留数据提供给 HTML 索引器。在创建了这些文件后，现在你就可以安全的启用你的模块了。

```
/**
 * Implementation of hook_update_index().
 */
function legacysearch_update_index() {
// We define these variables as global so our shutdown function can
// access them.
global $last_change, $last_id;
// If PHP times out while indexing, run a function to save
// information about how far we got so we can continue at next cron run.
register_shutdown_function('legacysearch_update_shutdown');
$last_id = variable_get('legacysearch_cron_last_id', 0);
$last_change = variable_get('legacysearch_cron_last_change', 0);
// Switch database connection to legacy database.
db_set_active('technote');
$result = db_query("SELECT id, title, note, last_modified
FROM {technote}
WHERE (id > %d) OR (last_modified > %d)
ORDER BY last_modified ASC", $last_id, $last_change);
// Switch database connection back to Drupal database.
db_set_active('default');
// Feed the external information to the search indexer.
while ($data = db_fetch_object($result)) {
$last_change = $data->last_modified;
$last_id = $data->id;
$text = " . check_plain($data->title) . " . $data->note;
search_index($data->id, 'technote', $text);
}
}
```

每一内容片段被传递给 search_index()，一同传递的还有一个标识符（在这里标识符的值来自于遗留数据库表中的 ID 列），内容的类型（我们将类型设为 technote。对 Drupal 内容索引时类型一般为节点或者用户），还有要被索引的文本。

register_shutdown_function() 指定了一个函数，在为请求执行完 PHP 脚本后执行这一函数。因为在索引完所有内容以前 PHP 可能会终止运行，所以使用它来追踪最后索引项目的 ID。

```
/**
 * Shutdown function to make sure we remember the last element processed.
 */
function legacysearch_update_shutdown() {
    global $last_change, $last_id;
    if ($last_change && $last_id) {
        variable_set('legacysearch_cron_last', $last_change);
        variable_set('legacysearch_cron_last_id', $last_id);
    }
}
```

在这个模块中，我们需要实现的最后一个函数是钩子函数 hook_search()，它将让我们为为我们的遗留信息使用内置的用户接口。

```
/**
 * Implementation of hook_search().
 */
function legacysearch_search($op = 'search', $keys = NULL) {
    switch ($op) {
        case 'name':
            return t('Tech Notes'); // Used on search tab.
        case 'reset':
            variable_del('legacysearch_cron_last');
            variable_del('legacysearch_cron_last_id');
            return;
        case 'search':
            // Search the index for the keywords that were entered.
            $hits = do_search($keys, 'technote');
            $results = array();
            // Prepend URL of legacy system to each result. Assume a legacy URL
            // for a given tech note is http://technotes.example.com/note.pl?3
            $legacy_url = 'http://technotes.example.com/';
            // We now have the IDs of the results. Pull each result
            // from the legacy database.
            foreach ($hits as $item) {
                db_set_active('technote');
                $note = db_fetch_object(db_query("SELECT * FROM {technote} WHERE
                id = %d", $item->sid));
                db_set_active('default');
                $results[] = array(
                    'link' => url($legacy_url . 'note.pl?' . $item->sid, NULL, NULL, TRUE),

```

```

'type' => t('Note'),
'title' => $note->title,
'date' => $note->last_modified,
'score' => $item->score,
'snippet' => search_excerpt($keys, $note->note));
}
return $results;
}
}

```

在运行完 cron 并对信息索引以后，就可以搜索技术文件了，如图12-9所示：

The screenshot shows a web search interface. At the top, there's a 'Search' header with three tabs: 'Content', 'Tech Notes', and 'Users'. Below the tabs, a prompt 'Enter your keywords:' is followed by a text input field containing the word 'emulator' and a 'Search' button. Underneath, the 'Search results' section displays a result for 'Web 1.0 Emulator'. The result text reads: 'This handy product lets you emulate the blink tag but in hardware...a perfect gift. ...'. At the bottom of the result, it says 'Note - 02/27/2007 - 15:35'.

图12-8 搜索外部的遗留数据库

总结

读完本章后，你应该可以：

- 创建一个定制搜索表单
- 理解 HTML 索引器是如何工作的
- 使用索引器钩子索引各种内容类型