

```
text = Markup('<h1>Hello, Flask!</h1>')
return render_template('index.html', text=text)
```

这时在模板中可以直接使用 {{ text }}。



注意 绝对不要直接对用户输入的内容使用 safe 过滤器，否则容易被植入恶意代码，导致 XSS 攻击。

2. 自定义过滤器

如果内置的过滤器不能满足你的需要，还可以添加自定义过滤器。使用 app.template_filter() 装饰器可以注册自定义过滤器，代码清单 3-5 注册了一个 musical 过滤器。

代码清单3-5 template/app.py：注册自定义过滤器

```
from flask import Markup

@app.template_filter()
def musical(s):
    return s + Markup(' &#9835;')
```

和注册全局函数类似，你可以在 app.template_filter() 中使用 name 关键字设置过滤器的名称，默认会使用函数名称。过滤器函数需要接收被处理的值作为输入，返回处理后的值。过滤器函数接收 s 作为被过滤的变量值，返回处理后的值。我们创建的 musical 过滤器会在被过滤的变量字符后面添加一个音符（single bar note）图标，因为音符通过 HTML 实体 ♫ 表示，我们使用 Markup 类将它标记为安全字符。在使用时和其他过滤器用法相同：

```
{{ name|musical }}
```



附注 你可以直接使用 app.add_template_filter() 方法注册自定义过滤器，传入函数对象和可选的自定义名称（name），比如 app.add_template_filter(your_filter_function)。

3.2.4 测试器

在 Jinja2 中，测试器（Test）是一些用来测试变量或表达式，返回布尔值（True 或 False）的特殊函数。比如，number 测试器用来判断一个变量或表达式是否是数字，我们使用 is 连接变量和测试器：

```
{% if age is number %}
    {{ age * 365 }}
{% else %}
    无效的数字。
{% endif %}
```

1. 内置测试器

Jinja2 内置了许多测试器，常用的测试器及用法说明如表 3-6 所示。

表 3-6 常用的内置测试器

测 试 器	说 明
callable(object)	判断对象是否可被调用
defined(value)	判断变量是否已定义
undefined(value)	判断变量是否未定义
none(value)	判断变量是否为 None
number(value)	判断变量是否是数字
string(value)	判断变量是否是字符串
sequence(value)	判断变量是否是序列，比如字符串、列表、元组
iterable(value)	判断变量是否可迭代
mapping(value)	判断变量是否是匹配对象，比如字典
sameas(value, other)	判断变量与 other 是否指向相同的内存地址

 **附注** 这里只列出了一部分常用的测试器，完整的内置测试器列表请访问 <http://jinja.pocoo.org/docs/2.10/templates/#list-of-builtin-tests> 查看。

在使用测试器时，is 的左侧是测试器函数的第一个参数（value），其他参数可以添加括号传入，也可以在右侧使用空格连接，以 sameas 为例：

```
{% if foo is sameas(bar) %}...
```

等同于：

```
{% if foo is sameas bar %}...
```

2. 自定义测试器

和过滤器类似，我们可以使用 Flask 提供的 app.template_test() 装饰器来注册一个自定义测试器。在示例程序中，我们创建了一个没有意义的 baz 过滤器，仅用来验证被测值是否为 baz，如代码清单 3-6 所示。

代码清单3-6 template/app.py：注册自定义测试器

```
@app.template_test()
def baz(n):
    if n == 'baz':
        return True
    return False
```

测试器的名称默认为函数名称，你可以在 app.template_test() 中使用 name 关键字指定自定义名称。测试器函数需要接收被测试的值作为输入，返回布尔值。

 **附注** 你可以直接使用 app.add_template_test() 方法注册自定义测试器，传入函数对象和可选的自定义名称（name），比如 app.add_template_test(your_test_function)。

3.2.5 模板环境对象

在 Jinja2 中，渲染行为由 `jinja2.Environment` 类控制，所有的配置选项、上下文变量、全局函数、过滤器和测试器都存储在 `Environment` 实例上。当与 Flask 结合后，我们并不单独创建 `Environment` 对象，而是使用 Flask 创建的 `Environment` 对象，它存储在 `app.jinja_env` 属性上。

在程序中，我们可以使用 `app.jinja_env` 更改 Jinja2 设置。比如，你可以自定义所有的定界符。下面使用 `variable_start_string` 和 `variable_end_string` 分别自定义变量定界符的开始和结束符号：

```
app = Flask(__name__)
app.jinja_env.variable_start_string = '[['
app.jinja_env.variable_end_string = ']]'
```

 **注意** 在实际开发中，如果修改 Jinja2 的定界符，那么需要注意与扩展提供模板的兼容问题，一般不建议修改。

模板环境中的全局函数、过滤器和测试器分别存储在 `Environment` 对象的 `globals`、`filters` 和 `tests` 属性中，这三个属性都是字典对象。除了使用 Flask 提供的装饰器和方法注册自定义函数，我们也可以直接操作这三个字典来添加相应的函数或变量，这通过向对应的字典属性中添加一个键值对实现，传入模板的名称作为键，对应的函数对象或变量作为值。下面是几个简单的示例。

1. 添加自定义全局对象

和 `app.template_global()` 装饰器不同，直接操作 `globals` 字典允许我们传入任意 Python 对象，而不仅仅是函数，类似于上下文处理函数的作用。下面的代码使用 `app.jinja_env.globals` 分别向模板中添加全局函数 `bar` 和全局变量 `foo`：

```
def bar():
    return 'I am bar.'
foo = 'I am foo.'

app.jinja_env.globals['bar'] = bar
app.jinja_env.globals['foo'] = foo
```

2. 添加自定义过滤器

下面的代码使用 `app.jinja_env.filters` 向模板中添加自定义过滤器 `smiling`：

```
def smiling(s):
    return s + ' :)'
```

```
app.jinja_env.filters['smiling'] = smiling
```

3. 添加自定义测试器

下面的代码使用 `app.jinja_env.tests` 向模板中添加自定义测试器 `baz`：

```
def baz(n):
```

```

if n == 'baz':
    return True
return False

app.jinja_env.tests['baz'] = baz

```

 **附注** 访问 <http://jinja.pocoo.org/docs/latest/api/#jinja2.Environment> 查看 Environment 类的所有属性及用法说明。

3.3 模板结构组织

除了使用函数、过滤器等工具控制模板的输出外，Jinja2 还提供了一些工具来在宏观上组织模板内容。借助这些技术，我们可以更好地实践 DRY (Don't Repeat Yourself) 原则。

3.3.1 局部模板

在 Web 程序中，我们通常会为每一类页面编写一个独立的模板。比如主页模板、用户资料页模板、设置页模板等。这些模板可以直接在视图函数中渲染并作为 HTML 响应主体。除了这类模板，我们还会用到另一类非独立模板，这类模板通常被称为局部模板或次模板，因为它们仅包含部分代码，所以我们不会在视图函数中直接渲染它，而是插入到其他独立模板中。

 提示 当程序中的某个视图用来处理 AJAX 请求时，返回的数据不需要包含完整的 HTML 结构，这时就可以返回渲染后的局部模板。

当多个独立模板中都会使用同一块 HTML 代码时，我们可以把这部分代码抽离出来，存储到局部模板中。这样一方面可以避免重复，另一方面也可以方便统一管理。比如，多个页面中都要在页面顶部显示一个提示条，这个横幅可以定义在局部模板 _banner.html 中。

我们使用 include 标签来插入一个局部模板，这会把局部模板的全部内容插在使用 include 标签的位置。比如，在其他模板中，我们可以在任意位置使用下面的代码插入 _banner.html 的内容：

```
{% include '_banner.html' %}
```

 提示 为了和普通模板区分开，局部模板的命名通常以一个下划线开始。

3.3.2 宏

宏 (macro) 是 Jinja2 提供的一个非常有用的特性，它类似 Python 中的函数。使用宏可以把一部分模板代码封装到宏里，使用传递的参数来构建内容，最后返回构建后的内容。在功能上，它和局部模板类似，都是为了方便代码块的重用。

为了便于管理，我们可以把宏存储在单独的文件中，这个文件通常命名为 macros.html 或 macros.html。在创建宏时，我们使用 macro 和 endmacro 标签声明宏的开始和结束。在开始标签中定义宏的名称和接收的参数，下面是一个简单的示例：

```
{% macro qux(amount=1) %}
  {% if amount == 1 %}
    I am qux.
  {% elif amount > 1 %}
    We are quxs.
  {% endif %}
{% endmacro %}
```

使用时，需要像从 Python 模块中导入函数一样使用 import 语句导入它，然后作为函数调用，传入必要的参数，如下所示：

```
{% from 'macros.html' import qux %}
...
{{ qux(amount=5) }}
```

另外，在使用宏时我们需要注意上下文问题。在 Jinja2 中，出于性能的考虑，并且为了让这一切保持显式，默认情况下包含（include）一个局部模板会传递当前上下文到局部模板中，但导入（import）却不会。具体来说，当我们使用 render_template() 函数渲染一个 foo.html 模板时，这个 foo.html 的模板上下文中包含下列对象：

- Flask 使用内置的模板上下文处理函数提供的 g、session、config、request。
- 扩展使用内置的模板上下文处理函数提供的变量。
- 自定义模板上下文处理器传入的变量。
- 使用 render_template() 函数传入的变量。
- Jinja2 和 Flask 内置及自定义全局对象。
- Jinja2 内置及自定义过滤器。
- Jinja2 内置及自定义测试器。

使用 include 标签插入的局部模板（比如 _banner.html）同样可以使用上述上下文中的变量和函数。而导入另一个并非被直接渲染的模板（比如 macros.html）时，这个模板仅包含下列这些对象：

- Jinja2 和 Flask 内置的全局函数和自定义全局函数。
- Jinja2 内置及自定义过滤器。
- Jinja2 内置及自定义测试器。

因此，如果我们想在导入的宏中使用第一个列表中的 2、3、4 项，就需要在导入时显式地使用 with context 声明传入当前模板的上下文：

```
{% from "macros.html" import foo with context %}
```

 **注意** 虽然 Flask 使用内置的模板上下文处理函数传入 session、g、request 和 config，但它同时也使用 app.jinja_env.globals 字典将这几个变量设置为全局变量，所以我们仍然可以在不显示声明传入上下文的情况下，直接在导入的宏中使用它们。

 提示 关于宏的编写，更多的细节请访问 <http://jinja.pocoo.org/docs/latest/templates/#macros> 查看。

3.3.3 模板继承

Jinja2 的模板继承允许你定义一个基模板，把网页上的导航栏、页脚等通用内容放在基模板中，而每一个继承基模板的子模板在被渲染时都会自动包含这些部分。使用这种方式可以避免在多个模板中编写重复的代码。

1. 编写基模板

基模板存储了程序页面的固定部分，通常被命名为 base.html 或 layout.html。示例程序中的基模板 base.html 中包含了一个基本的 HTML 结构，我们还添加了一个简单的导航条和页脚，如代码清单 3-7 所示。

代码清单 3-7 template/templates/base.html：定义基模板

```
<!DOCTYPE html>
<html>
<head>
    {% block head %}
        <meta charset="utf-8">
        <title>{% block title %}Template - HelloFlask{% endblock %}</title>
        {% block styles %}{% endblock %}
    {% endblock %}
</head>
<body>
<nav>
    <ul><li><a href="{{ url_for('index') }}">Home</a></li></ul>
</nav>
<main>
    {% block content %}{% endblock %}
</main>
<footer>
    {% block footer %}...
    ...
    {% endblock %}
</footer>
    {% block scripts %}{% endblock %}
</body>
</html>
```

当子模板继承基模板后，子模板会自动包含基模板的内容和结构。为了能够让子模板方便地覆盖或插入内容到基模板中，我们需要在基模板中定义块（block），在子模板中可以通过定义同名的块来执行继承操作。

块的开始和结束分别使用 block 和 endblock 标签声明，而且块之间可以嵌套。在这个基模板中，我们创建了六个块：head、title、styles、content、footer 和 scripts，分别用来划分不同的代码。其中，head 块表示 <head> 标签的内容，title 表示 <title> 标签的内容，content 块表示页面主体

内容，`footer` 表示页脚部分，`styles` 块和 `scripts` 块，则分别用来包含 CSS 文件和 JavaScript 文件引用链接或页内的 CSS 和 JavaScript 代码。

 提示 这里的块名称可以随意指定，而且并不是必须的。你可以按照需要设置块，如果你只需要让子模板添加主体内容，那么仅定义一个 `content` 块就足够了。

以 `content` 块为例，模板继承示意图如图 3-2 所示。

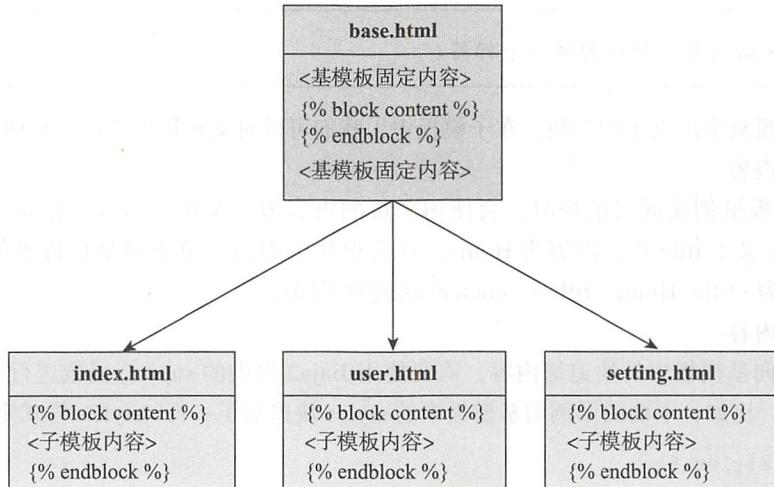


图 3-2 模板继承示意图

为了避免块的混乱，块的结束标签可以指明块名，同时要确保前后名称一致。比如：

```
{% block body %}
...
{% endblock body %}
```

2. 编写子模板

因为基模板中定义了 HTML 的基本结构，而且包含了页脚等固定信息，在子模板中我们不再需要定义这些内容，只需要对特定的块进行修改。这时我们可以修改前面创建的电影清单模板 `watchlist.html` 和主页模板 `index.html`，将这些子模板的通用部分合并到基模板中，并在子模板中定义块来组织内容，以便在渲染时将块中的内容插入到基模板的对应位置。以 `index.html` 为例，修改后的模板代码如代码清单 3-8 所示。

代码清单3-8 template/templates/index.html：子模板

```
{% extends 'base.html' %}
{% from 'macros.html' import qux %}

{% block content %}
{% set name='baz' %}
<h1>Template</h1>
<ul>
```

```
<li><a href="{{ url_for('watchlist') }}">Watchlist</a></li>
<li>Filter: {{ foo|musical }}</li>
<li>Global: {{ bar() }}</li>
<li>Test: {% if name is baz %}I am baz.{% endif %}</li>
<li>Macro: {{ qux(amount=5) }}</li>
</ul>
{% endblock %}
```

我们使用 `extends` 标签声明扩展基模板，它告诉模板引擎当前模板派生自 `base.html`。

 **注意** `extends` 必须是子模板的第一个标签。

我们在基模板中定义了四个块，在子模板中，我们可以对父模板中的块执行两种操作：

(1) 覆盖内容

当在子模板里创建同名的块时，会使用子块的内容覆盖父块的内容。比如我们在子模板 `index.html` 中定义了 `title` 块，内容为 `Home`，这会把块中的内容填充到基模板里的 `title` 块的位置，最终渲染为 `<title>Home</title>`，`content` 块的效果同理。

(2) 追加内容

如果想要向基模板中的块追加内容，需要使用 `Jinja2` 提供的 `super()` 函数进行声明，这会向父块添加内容。比如，下面的示例向基模板中的 `styles` 块追加了一行 `<style>` 样式定义：

```
{% block styles %}
{{ super() }}
<style>
    .foo {
        color: red;
    }
</style>
{% endblock %}
```

当子模板被渲染时，它会继承基模板的所有内容，然后根据我们定义的块进行覆盖或追加操作，渲染子模板 `index.html` 的结果如下所示：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Template - HelloFlask</title>
</head>
<body>
<nav>
    <ul><li><a href="/">Home</a></li></ul>
</nav>
<main>
    <h1>Template</h1>
    <ul>
        <li><a href="/watchlist">Watchlist</a></li>
        <li>Filter: I am foo. &#9835;</li>
        <li>Global: I am bar.</li>
    </ul>
</main>
<div>This is a template</div>
</body>
</html>
```

```
<li>Test: I am baz.</li>
<li>Macro: We are quxs.
</li>
</ul>
</main>
<footer>
    ...
</footer>
</body>
</html>
```

3.4 模板进阶实践

这一节我们会介绍模板在 Flask 程序中的常见应用，其中主要包括加载静态文件和自定义错误页面。

3.4.1 空白控制

在实际输出的 HTML 文件中，模板中的 Jinja2 语句、表达式和注释会保留移除后的空行，前面为了节省篇幅手动删掉了这些空行。以示例程序中的这段代码为例：

```
{% if user.bio %}
    <i>{{ user.bio }}</i>
{% else %}
    <i>This user has not provided a bio.</i>
{% endif %}
```

 提示 Jinja2 语句中的 HTML 代码缩进并不是必须的，只是为了增加可读性，在编写大量 Jinja2 代码时可读性尤其重要。

实际输出的 HTML 代码如下所示：

```
<i>{{ user.bio }}</i>
<i>This user has not provided a bio.</i>
```

如果你想在渲染时自动去掉这些空行，可以在定界符内侧添加减号。比如，`{%- endfor %}` 会移除该语句前的空白，同理，在右边的定界符内侧添加减号将移除该语句后的空白：

```
{% if user.bio -%
    <i>{{ user.bio }}</i>
{%- else -%
    <i>This user has not provided a bio.</i>
{%- endif %}}
```

现在输出的 HTML 代码如下所示：

```
<i>{{ user.bio }}</i>
<i>This user has not provided a bio.</i>
```



附注 你可以访问 <http://jinja.pocoo.org/docs/latest/templates/#whitespace-control> 查看更多细节。

除了在模板中使用减号来控制空白外，我们也可以使用模板环境对象提供的 `trim_blocks` 和 `lstrip_blocks` 属性设置，前者用来删除 Jinja2 语句后的第一个空行，后者则用来删除 Jinja2 语句所在行之前的空格和制表符（tabs）：

```
app.jinja_env.trim_blocks = True
app.jinja_env.lstrip_blocks = True
```



提示 `trim_blocks` 中的 block 指的是使用 `{% ... %}` 定界符的代码块，与我们前面介绍模板继承中的块无关。

需要注意的是，宏内的空白控制行为不受 `trim_blocks` 和 `lstrip_blocks` 属性控制，我们需要手动设置，比如：

```
{% macro qux(amount=1) %}
  {% if amount == 1 -%}
    I am qux.
  {% elif amount > 1 -%}
    We are quxs.
  {%- endif %}
{%- endmacro %}
```

事实上，我们没有必要严格控制 HTML 输出，因为多余的空白并不影响浏览器的解析。在部署时，我们甚至可以使用工具来去除 HTML 响应中所有的空白、空行和换行，这样可以减小文件体积，提高数据传输速度。所以，编写模板时应以可读性为先，在后面的示例程序中，我们将不再添加空白控制的代码，并且对 Jinja2 语句中的 HTML 代码进行必要的缩进来增加可读性。

3.4.2 加载静态文件

一个 Web 项目不仅需要 HTML 模板，还需要许多静态文件，比如 CSS、JavaScript 文件、图片以及音频等。在 Flask 程序中，默认我们需要将静态文件存储在与主脚本（包含程序实例的脚本）同级目录的 static 文件夹中。

为了在 HTML 文件中引用静态文件，我们需要使用 `url_for()` 函数获取静态文件的 URL。Flask 内置了用于获取静态文件的视图函数，端点值为 `static`，它的默认 URL 规则为 `/static/<path:filename>`，URL 变量 `filename` 是相对于 static 文件夹根目录的文件路径。



提示 如果你想使用其他文件夹来存储静态文件，可以在实例化 Flask 类时使用 `static_folder` 参数指定，静态文件的 URL 路径中的 `static` 也会自动跟随文件夹名称变化。在实例化 Flask 类时使用 `static_url_path` 参数则可以自定义静态文件的 URL 路径。

在示例程序的 static 目录下保存了一个头像图片 `avatar.jpg`，我们可以通过 `url_for('static', filename='avatar.jpg')` 获取这个文件的 URL，这个函数调用生成的 URL 为 `/static/avatar.jpg`，在

浏览器中输入 `http://localhost:5000/static/avatar.jpg` 即可访问这个图片。在模板 `watchlist2.html` 里，我们在用户名的左侧添加了这个图片，使用 `url_for()` 函数生成图片 `src` 属性所需的图片 URL，如下所示：

```

```

另外，我们还创建了一个存储 CSS 规则的 `styles.css` 文件，我们使用下面的方式在模板中加载这个文件：

```
<link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='styles.css') }}>
```

在浏览器中访问 `http://localhost:5000/watchlist2` 可以看到添加了头像图片并加载了 CSS 规则的电影清单页面，如图 3-3 所示。

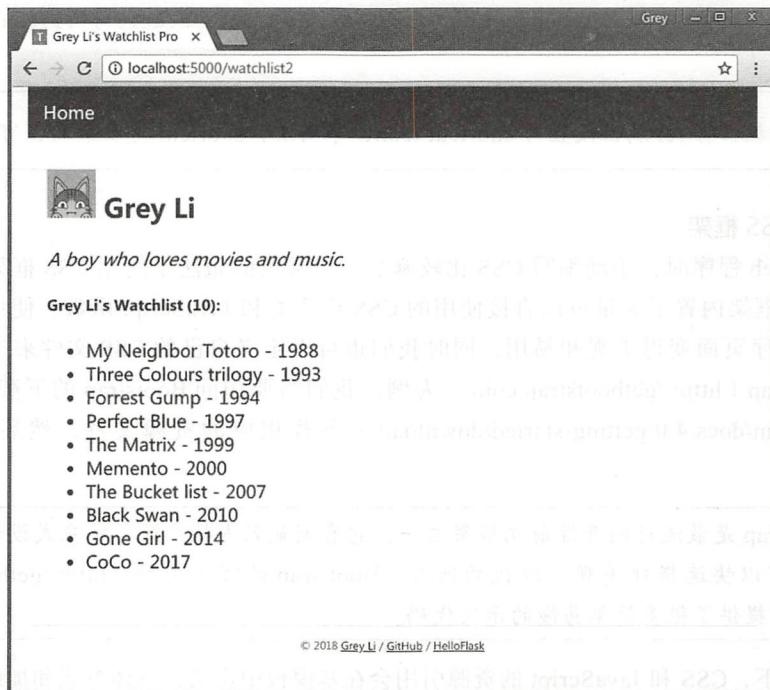


图 3-3 使用静态文件后的电影清单页面

1. 添加 Favicon

在运行前两章的示例程序时，我们经常在命令行看到一条 404 状态的 GET 请求记录，请求的 URL 为 `/favicon.ico`，如下所示：

```
127.0.0.1 - - [08/Feb/2018 18:31:12] "GET /favicon.ico HTTP/1.1" 404 -
```

这个 `favicon.ico` 文件指的是 Favicon (favorite icon，收藏夹头像 / 网站头像)，又称为 `shortcut icon`、`tab icon`、`website icon` 或是 `bookmark icon`。顾名思义，这是一个在浏览器标签页、

地址栏和书签收藏夹等处显示的小图标，作为网站的特殊标记。浏览器在发起请求时，会自动向根目录请求这个文件，在前面的示例程序中，我们没有提供这个文件，所以才会产生上面的404记录。

要想为Web项目添加Favicon，你要先有一个Favicon文件，并放置到static目录下。它通常是一个宽高相同的ICO格式文件，命名为favicon.ico。



附注 除了ICO格式，PNG和（无动画的）GIF格式也被所有主流浏览器支持。

Flask中静态文件的默认路径为/static/filename，为了正确返回Favicon，我们可以显式地在HTML页面中声明Favicon的路径。首先可以在<head>部分添加一个<link>元素，然后将rel属性设置为icon，如下所示：

```
<link rel="icon" type="image/x-icon" href="{{ url_for('static', filename='favicon.ico') }}>
```



附注 大部分教程将rel属性设置为shortcut icon，事实上，shortcut是多余的，可以省略掉。

2. 使用CSS框架

在编写Web程序时，手动编写CSS比较麻烦，更常见的做法是使用CSS框架来为程序添加样式。CSS框架内置了大量可以直接使用的CSS样式类和JavaScript函数，使用它们可以非常快速地让程序页面变得美观和易用，同时我们也可以定义自己的CSS文件来进行补充和调整。以Bootstrap (<http://getbootstrap.com/>) 为例，我们需要访问Bootstrap的下载页面 (<http://getbootstrap.com/docs/4.0/getting-started/download/>) 下载相应的资源文件，然后分类别放到static目录下。



附注 Bootstrap是最流行的开源前端框架之一，它有浏览器支持广泛、响应式设计等特点。使用它可以快速搭建美观、现代的网页。Bootstrap的官方文档 (<http://getbootstrap.com/docs/>) 提供了很多简单易懂的示例代码。

通常情况下，CSS和JavaScript的资源引用会在基模板中定义，具体方式和加载我们自定义的styles.css文件相同：

```
...
{% block styles %}
    <link rel="stylesheet" href="{{ url_for('static', filename='css/bootstrap.min.css') }}">
{% endblock %}
...
{% block scripts %}
    <script src="{{ url_for('static', filename='js/jquery.min.js') }}></script>
    <script src="{{ url_for('static', filename='js/popper.min.js') }}></script>
    <script src="{{ url_for('static', filename='js/bootstrap.min.js') }}></script>
{% endblock %}
...
```

 **注意** 如果不使用 Bootstrap 提供的 JavaScript 功能，那么也可以不加载。另外，Bootstrap 所依赖的 jQuery (<https://jquery.com/>) 和 Popper.js (<https://popper.js.org/>) 需要单独下载，这三个 JavaScript 文件在引入时要按照 jQuery→Popper.js→Bootstrap 的顺序引入。

虽然我建议在开发时统一管理静态资源，如果你想简化开发过程，那么从 CDN 加载是更方便的做法。从 CND 加载时，只需要将相应的 URL 替换为 CDN 提供的资源 URL，比如：

```
...
{%- block styles %}
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
        css/bootstrap.min.css">
{%- endblock %}
...
{%- block scripts %}
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.12.9/dist/umd/
        popper.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.
        min.js"></script>
{%- endblock %}
...

```

3. 使用宏加载静态资源

为了方便加载静态资源，我们可以创建一个专门用于加载静态资源的宏，如代码清单 3-9 所示。

代码清单 3-9 template/templates/macros.html：用于加载静态资源的宏

```
{% macro static_file(type, filename_or_url, local=True) %}
    {% if local %}
        {% set filename_or_url = url_for('static', filename=filename_or_url) %}
    {% endif %}
    {% if type == 'css' %}
        <link rel="stylesheet" href="{{ filename_or_url }}" type="text/css">
    {% elif type == 'js' %}
        <script type="text/javascript" src="{{ filename_or_url }}></script>
    {% elif type == 'icon' %}
        <link rel="icon" href="{{ filename_or_url }}>
    {% endif %}
{%- endmacro %}
```

在模板中导入宏后，只需在调用时传入静态资源的类别和文件路径就会获得完整的资源加载语句。使用它加载 CSS 文件的示例如下：

```
static_file('css', 'css/bootstrap.min.css')
```

使用它也可以从 CDN 加载资源，只需要将关键字参数 local 设为 False，然后传入资源的 URL 即可：

```
static_file('css', 'https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css', local=False)
```

3.4.3 消息闪现

Flask 提供了一个非常有用的 flash() 函数，它可以用来“闪现”需要显示给用户的消息，比如当用户登录成功后显示“欢迎回来！”。在视图函数调用 flash() 函数，传入消息内容即可“闪现”一条消息。当然，它并不是我们想象的，能够立刻在用户的浏览器弹出一条消息。实际上，使用功能 flash() 函数发送的消息会存储在 session 中，我们需要在模板中使用全局函数 get_flashed_messages() 获取消息并将其显示出来。



提示 通过 flash() 函数发送的消息会存储在 session 对象中，所以我们需要为程序设置密钥。可以通过 app.secret_key 属性或配置变量 SECRET_KEY 设置，具体可参考 2.3.4 节的相关内容。

你可以在任意视图函数中调用 flash() 函数发送消息。为了测试消息闪现，我们添加了一个 just_flash 视图，在函数中发送了一条消息，最后重定向到 index 视图，如代码清单 3-10 所示。

代码清单 3-10 app.py：使用 flash() 函数“闪现”消息

```
from flask import Flask, render_template, flash
app = Flask(__name__)
app.secret_key = 'secret string'

@app.route('/flash')
def just_flash():
    flash('I am flash, who is looking for me?')
    return redirect(url_for('index'))
```

Jinja2 内部使用 Unicode，所以你需要向模板传递 Unicode 对象或只包含 ASCII 字符的字符串。在 Python 2.x 中，如果字符串包含中文（或任何非 ASCII 字符），那么需要在字符串前添加 u 前缀，这会告诉 Python 把这个字符串编码成 Unicode 字符串，另外还需要在 Python 文件的首行添加编码声明，这会让 Python 使用 UTF-8 来解码字符串，后面不再提示。发送中文消息的示例如下所示：

```
# -*- coding: utf-8 -*-
...
@app.route('/flash')
def just_flash():
    flash(u'你好，我是闪电。')
    return redirect(url_for('index'))
```



提示 Flask、Jinja2 和 Werkzeug 等相关依赖均将文本的类型设为 Unicode，所以你在编写程序和它们交互时应该遵循同样的约定。比如，在 Python 脚本中添加编码声明；在 Python2 中为非 ASCII 字符添加 u 前缀；将编辑器的默认编码设为 UTF-8；在 HTML 文件的 head 标签中添加编码声明，即 <meta charset="utf-8">；当你需要读取文件传入模板时，手动使用 decode() 函数解码。

 提示 在 Python 3.x 中，字符串默认类型为 Unicode。如果你使用 Python3，那么包含中文的字符串前的 u 前缀可以省略掉，同时也不用在脚本开头添加编码声明。尽管如此，还是建议保留这个声明以便让某些编辑器自动切换设置的编码类型。

 附注 Unicode 又称为国际码，它对世界上大部分的文字系统进行了整理、编码，使电脑可以正常显示大部分文字。ASCII 和 UTF-8 是两种常见的编码系统，其中 ASCII 主要用来显示现代英语，而 UTF-8 是一种针对 Unicode 的编码系统。Python 2.x 默认使用 ASCII，Python 3.x 默认使用 UTF-8。你可以访问 <https://docs.python.org/3/howto/unicode.html> 来了解关于 Python 的 Unicode 支持。

Flask 提供了 `get_flashed_message()` 函数用来在模板里获取消息，因为程序的每一个页面都有可能需要显示消息，我们把获取并显示消息的代码放在基模板中 `content` 块的上面，这样就可以在页面主体内容的上面显示消息，如代码清单 3-11 所示。

代码清单 3-11 templates/base.html：渲染 flash 消息

```
<main>
    {%
        for message in get_flashed_messages() %}
        <div class="alert">{{ message }}</div>
    {% endfor %}
    {% block content %}{% endblock %}
</main>
```

因为同一个页面可能包含多条要显示的消息，所以这里使用 `for` 循环迭代 `get_flashed_message()` 返回的消息列表。另外，我们还为消息定义了一些 CSS 规则，你可以在示例程序中的 `static/styles.css` 文件中查看。现在访问 `http://localhost:5000` 打开示例程序的主页，如果你单击页面上的 Flash something 链接（指向 `/flash`），页面重载后就会显示一条消息，如图 3-4 所示。

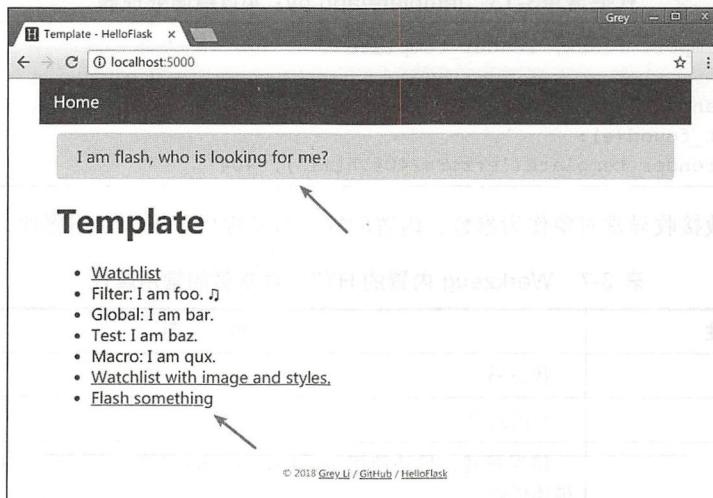


图 3-4 消息闪现示例

当 `get_flashed_message()` 函数被调用时，`session` 中存储的所有消息都会被移除。如果你这时刷新页面，会发现重载后的页面不再出现这条消息。

3.4.4 自定义错误页面

当程序返回错误响应时，会渲染一个默认的错误页面，我们在第 2 章和它们打过招呼。默认的错误页面太简单了，而且和其他页面的风格不符，导致用户看到这样的页面时往往不知所措。我们可以注册错误处理函数来自定义错误页面。

错误处理函数和视图函数很相似，返回值将会作为响应的主体，因此我们首先要创建错误页面的模板文件。为了和普通模板区分开来，我们在模板文件夹 `templates` 里为错误页面创建了一个 `errors` 子文件夹，并在其中为最常见的 404 和 500 错误创建了模板文件，表示 404 页面的 `404.html` 模板内容如代码清单 3-12 所示。

代码清单 3-12 `template/templates/errors/404.html`: 404 页面模板

```
{% extends 'base.html' %}

{% block title %}404 - Page Not Found{% endblock %}

{% block content %}
<h1>Page Not Found</h1>
<p>You are lost...</p>
{% endblock %}
```

错误处理函数需要附加 `app.errorhandler()` 装饰器，并传入错误状态码作为参数。错误处理函数本身则需要接收异常类作为参数，并在返回值中注明对应的 HTTP 状态码。当发生错误时，对应的错误处理函数会被调用，它的返回值会作为错误响应的主体。代码清单 3-13 是用来捕捉 404 错误的错误处理器。

代码清单 3-13 `template/app.py`: 404 错误处理器

```
from flask import Flask, render_template
...
@app.errorhandler(404)
def page_not_found(e):
    return render_template('errors/404.html'), 404
```

错误处理函数接收异常对象作为参数，内置的异常对象提供了下列常用属性，如表 3-7 所示。

表 3-7 Werkzeug 内置的 HTTP 异常类的常用属性

属 性	说 明
code	状态码
name	原因短语
description	错误描述，另外使用 <code>get_description()</code> 方法还可以获取 HTML 格式的错误描述代码

如果你不想手动编写错误页面的内容，可以将这些信息传入错误页面模板，在模板中用它们来构建错误页面。不过需要注意的是，传入 500 错误处理器的是真正的异常对象，通常不会提供这几个属性，你需要手动编写这些值。

提示 我们在第 2 章介绍过，Flask 通过抛出 Werkzeug 中定义的 HTTP 异常类来表示 HTTP 错误，错误处理函数接收的参数就是对应的异常类。基于这个原理，你也可以使用 `app.errorhandler()` 装饰器为其他异常注册处理函数，并返回自定义响应，只需要在 `app.errorhandler()` 装饰器中传入对应的异常类即可。比如，使用 `app.errorhandler(NameError)` 可以注册处理 `NameError` 异常的函数。

这时如果访问一个错误的 URL（即未在程序中定义的 URL），比如 `http://localhost:5000/nothing`，将会看到如图 3-5 所示的错误页面。



图 3-5 自定义 404 错误页面

除了 404 错误，我们还需要为另一个最常见的 500 错误编写错误处理器和模块，这些代码基本相同，具体可以到源码仓库中查看。

3.4.5 JavaScript 和 CSS 中的 Jinja2

当程序逐渐变大时，很多时候我们会需要在 JavaScript 和 CSS 代码中使用 Jinja2 提供的变量值，甚至是控制语句。比如，通过传入模板的 `theme_color` 变量来为页面设置主题色彩，或是根据用户是否登录来决定是否执行某个 JavaScript 函数。

首先要明白的是，只有使用 `render_template()` 传入的模板文件才会被渲染，如果你把 Jinja2 代码写在单独的 JavaScript 或是 CSS 文件中，尽管你在 HTML 中引入了它们，但它们包含的 Jinja2 代码永远也不会被执行。对于这类情况，下面有一些 Tips：

1. 行内 / 嵌入式 JavaScript/CSS

如果要在 JavaScript 和 CSS 文件中使用 Jinja2 代码，那么就在 HTML 中使用 `<style>` 和 `<script>` 标签定义这部分 CSS 和 JavaScript 代码。

在这部分 CSS 和 JavaScript 代码中加入 Jinja2 时，不用考虑编写时的语法错误，比如引号错误，因为 Jinja2 会在渲染后被替换掉，所以只需要确保渲染后的代码正确即可。

不过我并不推荐使用这种方式，尤其是行内 JavaScript/CSS 会让维护变得困难。避免把大量 JavaScript 代码留在 HTML 中的办法就是尽量将要使用的 Jinja2 变量值在 HTML 模板中定义为 JavaScript 变量。

2. 定义为 JavaScript/CSS 变量

对于想要在 JavaScript 中获取的数据，如果是元素特定的数据，比如某个文章条目对应的 id 值，可以通过 HTML 元素的 data-* 属性存储。你可以自定义横线后的名称，作为元素上的自定义数据变量，比如 data-id，data-username 等，比如：

```
<span data-id="{{ user.id }}" data-username="{{ user.username }}>{{ user.username }}</span>
```

在 JavaScript 中，我们可以使用 DOM 元素的 dataset 属性获取 data-* 属性值，比如 element.dataset.username，或是使用 getAttribute() 方法，比如 element.getAttribute('data-username')；使用 jQuery 时，可以直接对 jQuery 对象调用 data 方法获取，比如 \$element.data('username')。



提 示 在 HTML 中，“data-*”被称为自定义数据属性 (custom data attribute)，我们可以用它来存储自定义的数据供 JavaScript 获取。在后面的其他程序中，我们也会频繁使用这种方式来传递数据。

对于需要全局使用的数据，则可以在页面中使用嵌入式 JavaScript 定义变量，如果没法定义为 JavaScript 变量，那就考虑定义为函数，比如：

```
<script type="text/javascript">
    var foo = '{{ foo_variable }}';
</script>
```



提 示 当你在 JavaScript 中插入了太多 Jinja2 语法时，或许这时你该考虑将程序转变为 Web API，然后专心使用 JavaScript 来编写客户端，在本书的第二部分我们会介绍如何编写 Web API。

CSS 同理，有些时候你会需要将 Jinja2 变量值传入 CSS 文件，比如我们希望将用户设置的主题颜色设置到对应的 CSS 规则中，或是需要将 static 目录下某个图片的 URL 传入 CSS 来设置为背景图片，除了将这部分 CSS 定义直接写到 HTML 中外，我们可以将这些值定义为 CSS 变量，如下所示：

```
<style>
:root {
    --theme-color: {{ theme_color }};
    --background-url: {{ url_for('static', filename='background.jpg') }};
}
</style>
```

在 CSS 文件中，使用 var() 函数并传入变量名即可获取对应的变量值：

```
#foo {
    color: var(--theme-color);
}
#bar {
    background: var(--background-url);
}
```

3.5 本章小结

本章学习了 Jinja2 的基本用法和一些进阶技巧。如果你想了解更多的细节，或是其他进阶内容，可以阅读它的官方文档。下一章，我们会学习 Web 表单的使用，从而实现更丰富的用户交互。

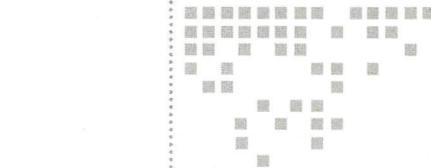


表 单

第4章

在 Web 程序中，表单是和用户交互最常见的方式之一。用户注册、登录、撰写文章、编辑设置，无一不用到表单。不过，表单的处理却并不简单。你不仅要创建表单，验证用户输入的内容，向用户显示错误提示，还要获取并保存数据。幸运的是，强大的 WTForms 可以帮我们解决这些问题。WTForms 是一个使用 Python 编写的表单库，它使得表单的定义、验证（服务器端）和处理变得非常轻松。这一章我们会介绍在 Web 程序中处理表单的方法和技巧。

本章新涉及的 Python 包如下所示：

- ❑ WTForms (2.2)
 - 主页: <https://github.com/wtforms/wtforms>
 - 文档: <https://wtforms.readthedocs.io/en/latest/>
- ❑ Flask-WTF (0.14.2)
 - 主页: <https://github.com/lepture/flask-wtf>
 - 文档: <https://flask-wtf.readthedocs.io/en/latest/>
- ❑ Flask-CKEditor (0.4.0)
 - 主页: <https://github.com/greyli/flask-ckeditor>
 - 文档: <https://flask-ckeditor.readthedocs.io/>

本章的示例程序在 `helloflask/demos/form` 目录下，确保当前目录在 `helloflask/demos/form` 下并激活了虚拟环境，然后执行 `flask run` 命令运行程序：

```
$ cd demos/form  
$ flask run
```

4.1 HTML 表单

在 HTML 中，表单通过 `<form>` 标签创建，表单中的字段使用 `<input>` 标签定义。下面是一

个非常简单的 HTML 表单：

```
<form method="post">
    <label for="username">Username</label><br>
    <input type="text" name="username" placeholder="Héctor Rivera"><br>
    <label for="password">Password</label><br>
    <input type="password" name="password" placeholder="19001130"><br>
    <input id="remember" name="remember" type="checkbox" checked>
    <label for="remember"><small>Remember me</small></label><br>
    <input type="submit" name="submit" value="Log in">
</form>
```

在 HTML 表单中，我们创建 `<input>` 标签表示各种输入字段，`<label>` 标签则用来定义字段的标签文字。我们可以在 `<form>` 和 `<input>` 标签中使用各种属性来对表单进行设置。上面的表单被浏览器解析后会生成两个输入框，一个勾选框和一个提交按钮。如果你运行了示例程序，访问 `http://localhost:5000/html` 可以看到渲染后的表单，如图 4-1 所示。

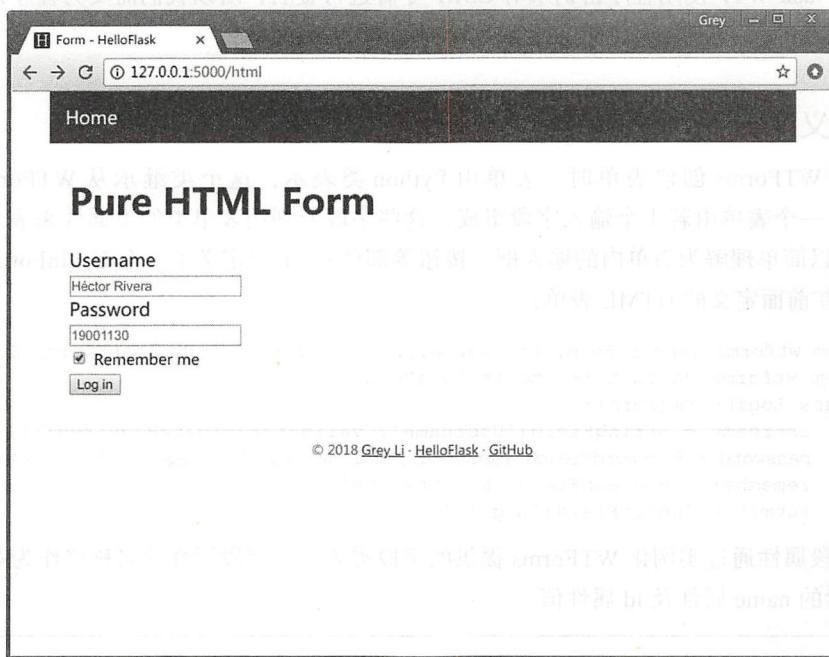


图 4-1 HTML 表单示例

 **附注** 关于 HTML 表单的具体定义和用法可以访问 <https://www.w3.org/TR/html401/interact/forms.html> 查看。

WTForms 支持在 Python 中使用类定义表单，然后直接通过类定义生成对应的 HTML 代码，这种方式更加方便，而且使表单更易于重用。因此，除非是非常简单的程序，或者是你想让表单的定义更加灵活，否则我们一般不会在模板中直接使用 HTML 编写表单，

4.2 使用 Flask-WTF 处理表单

扩展 Flask-WTF 集成了 WTForms，使用它可以在 Flask 中更方便地使用 WTForms。Flask-WTF 将表单数据解析、CSRF 保护、文件上传等功能与 Flask 集成，另外还附加了 reCAPTCHA 支持。



附注 reCAPTCHA (<https://www.google.com/recaptcha/>) 是 Google 开发的免费验证码服务，在国内目前无法直接使用。

首先，和其他扩展一样，我们先用 Pipenv 安装 Flask-WTF 及其依赖：

```
$ pipenv install flask-wtf
```

Flask-WTF 默认认为每个表单启用 CSRF 保护，它会为我们自动生成和验证 CSRF 令牌。默认情况下，Flask-WTF 使用程序密钥来对 CSRF 令牌进行签名，所以我们需要为程序设置密钥：

```
app.secret_key = 'secret string'
```

4.2.1 定义 WTForms 表单类

当使用 WTForms 创建表单时，表单由 Python 类表示，这个类继承从 WTForms 导入的 Form 基类。一个表单由若干个输入字段组成，这些字段分别用表单类的类属性来表示（字段即 Field，你可以简单理解为表单内的输入框、按钮等部件）。下面定义了一个 LoginForm 类，最终会生成我们在前面定义的 HTML 表单：

```
>>> from wtforms import Form, StringField, PasswordField, BooleanField, SubmitField
>>> from wtforms.validators import DataRequired, Length
>>> class LoginForm(Form):
...     username = StringField('Username', validators=[DataRequired()])
...     password = PasswordField('Password', validators=[DataRequired(), Length(8, 128)])
...     remember = BooleanField('Remember me')
...     submit = SubmitField('Log in')
```

每个字段属性通过实例化 WTForms 提供的字段类表示。字段属性的名称将作为对应 HTML <input> 元素的 name 属性及 id 属性值。



注意 字段属性名称大小写敏感，不能以下划线或 validate 开头。

这里的 LoginForm 表单类中定义了四个字段：文本字段 StringField、密码字段 PasswordField、勾选框字段 BooleanField 和提交按钮字段 SubmitField。字段类从 wtforms 包导入，常用的 WTForms 字段如表 4-1 所示。



提示 有些字段最终生成的 HTML 代码相同，不过 WTForms 会在表单提交后根据表单类中字段的类型对数据进行处理，转换成对应的 Python 类型，以便在 Python 脚本中对数据进行处理。

表 4-1 常用的 WTForms 字段

字段类	说 明	对应的 HTML 表示
BooleanField	复选框，值会被处理为 True 或 False	<input type="checkbox">
TextField	文本字段，值会被处理为 datetime.date 对象	<input type="text">
DateTimeField	文本字段，值会被处理为 datetime.datetime 对象	<input type="text">
FileField	文件上传字段	<input type="file">
FloatField	浮点数字段，值会被处理为浮点型	<input type="text">
IntegerField	整数字段，值会被处理为整型	<input type="text">
RadioField	一组单选按钮	<input type="radio">
SelectField	下拉列表	<select><option></option></select>
SelectMultipleField	多选下拉列表	<select multiple><option></option></select>
SubmitField	提交按钮	<input type="submit">
StringField	文本字段	<input type="text">
HiddenField	隐藏文本字段	<input type="hidden">
PasswordField	密码文本字段	<input type="password">
TextAreaField	多行文本字段	<textarea></textarea>

通过实例化字段类时传入的参数，我们可以对字段进行设置，字段类构造方法接收的常用参数如表 4-2 所示。

表 4-2 实例化字段类常用参数

参 数	说 明
label	字段标签 <label> 的值，也就是渲染后显示在输入字段前的文字
render_kw	一个字典，用来设置对应的 HTML <input> 标签的属性，比如传入 {'placeholder': 'Your Name'}，渲染后的 HTML 代码会将 <input> 标签的 placeholder 属性设为 Your Name
validators	一个列表，包含一系列验证器，会在表单提交后被逐一调用验证表单数据
default	字符串或可调用对象，用来为表单字段设置默认值

在 WTForms 中，验证器 (validator) 是一系列用于验证字段数据的类，我们在实例化字段类时使用 validators 关键字来指定附加的验证器列表。验证器从 wtforms.validators 模块中导入，常用的验证器如表 4-3 所示。

表 4-3 常用的 WTForms 验证器

验 证 器	说 明
DataRequired(message=None)	验证数据是否有效
Email(message=None)	验证 Email 地址
EqualTo(fieldname, message=None)	验证两个字段值是否相同

(续)

验证器	说 明
InputRequired(message=None)	验证是否有数据
Length(min=-1, max=-1, message=None)	验证输入值长度是否在给定范围内
NumberRange(min=None, max=None, message=None)	验证输入数字是否在给定范围内
Optional(strip_whitespace=True)	允许输入值为空，并跳过其他验证
Regexp(regex, flags=0, message=None)	使用正则表达式验证输入值
URL(require_tld=True, message=None)	验证 URL
AnyOf(values, message=None, values_formatter=None)	确保输入值在可选值列表中
NoneOf(values, message=None, values_formatter=None)	确保输入值不在可选值列表中

 在实例化验证类时，message 参数用来传入自定义错误消息，如果没有设置则使用内置的英文错误消息，后面我们会了解如何使用内置的中文错误消息。

 validators 参数接收一个传入可调用对象组成的列表。内置的验证器通过实现了 __call__() 方法的类表示，所以我们需要在验证器后添加括号。

在 name 和 password 字段里，我们都使用了 DataRequired 验证器，用来验证输入的数据是否有效。另外，password 字段里还添加了一个 Length 验证器，用来验证输入的数据长度是否在给定的范围内。验证器的第一个参数一般为错误提示消息，我们可以使用 message 关键字传递参数，通过传入自定义错误信息来覆盖内置消息，比如：

```
name = StringField('Your Name', validators=[DataRequired(message=u'名字不能为空! ')])
```

当使用 Flask-WTF 定义表单时，我们仍然使用 WTForms 提供的字段类和验证器，创建的方式也完全相同，只不过表单类要继承 Flask-WTF 提供的 FlaskForm 类。FlaskForm 类继承自 Form 类，进行了一些设置，并附加了一些辅助方法，以便与 Flask 集成。因为本章的示例程序中包含多个表单类，为了便于组织，我们创建了一个 forms.py 脚本，用来存储所有的表单类。代码清单 4-1 是继承 FlaskForm 类的 LoginForm 表单。

代码清单4-1 form/forms.py：定义表单类

```
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired, Length

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired(), Length(8, 128)])
    remember = BooleanField('Remember me')
    submit = SubmitField('Log in')
```

 提示 配置键 `WTF_CSRF_ENABLED` 用来设置是否开启 CSRF 保护，默认为 True。Flask-WTF 会自动在实例化表单类时添加一个包含 CSRF 令牌值的隐藏字段，字段名为 `csrf_token`。

4.2.2 输出 HTML 代码

以我们使用 WTForms 创建的 `LoginForm` 为例，实例化表单类，然后将实例属性转换成字符串或直接调用就可以获取表单字段对应的 HTML 代码：

```
>>> form = LoginForm()
>>> form.username()
u'<input id="username" name="username" type="text" value="">'
>>> form.submit()
u'<input id="submit" name="submit" type="submit" value="Submit">'
```

字段的 `<label>` 元素的 HTML 代码则可以通过 “`form.字段名.label`” 的形式获取：

```
>>> form.username.label()
u'<label for="username">Username</label>'
>>> form.submit.label()
u'<label for="submit">Submit</label>'
```

在创建 HTML 表单时，我们经常会需要使用 HTML `<input>` 元素的其他属性来对字段进行设置。比如，添加 `class` 属性设置对应的 CSS 类为字段添加样式；添加 `placeholder` 属性设置占位文本。默认情况下，WTForms 输出的字段 HTML 代码只会包含 `id` 和 `name` 属性，属性值均为表单类中对应的字段属性名称。如果要添加额外的属性，通常有两种方法。

1. 使用 `render_kw` 属性

比如下面为 `username` 字段使用 `render_kw` 设置了 `placeholder` HTML 属性：

```
username = StringField('Username', render_kw={'placeholder': 'Your Username'})
```

这个字段被调用后输出的 HTML 代码如下所示：

```
<input type="text" id="username" name="username" placeholder="Your Username">
```

2. 在调用字段时传入

在调用字段属性时，通过添加括号使用关键字参数的形式也可以传入字段额外的 HTML 属性：

```
>>> form.username(style='width: 200px;', class_='bar')
u'<input class="bar" id="username" name="username" style="width: 200px;" type="text">'
```

 附注 `class` 是 Python 的保留关键字，在这里我们使用 `class_` 来代替 `class`，渲染后的 `<input>` 会获得正确的 `class` 属性，在模板中调用时则可以直接使用 `class`。

 注意 通过上面的方法也可以修改 `id` 和 `name` 属性，但表单被提交后，WTForms 需要通过 `name` 属性来获取对应的数据，所以不能修改 `name` 属性值。

4.2.3 在模板中渲染表单

为了能够在模板中渲染表单，我们需要把表单类实例传入模板。首先在视图函数里实例化表单类 LoginForm，然后在 render_template() 函数中使用关键字参数 form 将表单实例传入模板，如代码清单 4-2 所示。

代码清单4-2 form/app.py：传入表单类实例

```
from forms import LoginForm

@app.route('/basic')
def basic():
    form = LoginForm()
    return render_template('login.html', form=form)
```

在模板中，只需要调用表单类的属性即可获取字段对应的 HTML 代码，如果需要传入参数，也可以添加括号，如代码清单 4-3 所示。

代码清单4-3 form/templates/basic.html：在模板中渲染表单

```
<form method="post">
    {{ form.csrf_token }} <!-- 渲染CSRF令牌隐藏字段 -->
    {{ form.username.label }}{{ form.username }}<br>
    {{ form.password.label }}{{ form.password }}<br>
    {{ form.remember }}{{ form.remember.label }}<br>
    {{ form.submit }}<br>
</form>
```

需要注意的是，在上面的代码中，除了渲染各个字段的标签和字段本身，我们还调用了 form.csrf_token 属性渲染 Flask-WTF 为表单类自动创建的 CSRF 令牌字段。form.csrf_token 字段包含了自动生成的 CSRF 令牌值，在提交表单后会自动被验证，为了确保表单通过验证，我们必须在表单中手动渲染这个字段。

 提示 Flask-WTF 为表单类实例提供了一个 form.hidden_tag() 方法，这个方法会依次渲染表单中所有的隐藏字段。因为 csrf_token 字段也是隐藏字段，所以当这个方法被调用时也会渲染 csrf_token 字段。

渲染后获得的实际 HTML 代码如下所示：

```
<form method="post">
    <input id="csrf_token" name="csrf_token" type="hidden" value="IjVmMDE1ZmFjM2
        VjYmZjY...i.DY1QSg.IWc1WEWxr3TvmAWCTHRMGjIcDOQ">
    <label for="username">Username</label><br>
    <input id="username" name="username" type="text" value=""><br>
    <label for="password">Password</label><br>
    <input id="password" name="password" type="password" value=""><br>
    <input id="remember" name="remember" type="checkbox" value="y"><label
        for="remember">Remember me</label><br>
    <input id="submit" name="submit" type="submit" value="Log in"><br>
</form>
```

如果你运行了示例程序，访问 `http://localhost:5000/basic` 可以看到渲染后的表单，页面中的表单和我们在上面使用 HTML 编写的表单完全相同。

在前面我们介绍过，使用 `render_kw` 字典或是在调用字段时传入参数来定义字段的额外 HTML 属性，通过这种方式添加 CSS 类，我们可以编写一个 Bootstrap 风格的表单，如代码清单 4-4 所示。

代码清单 4-4 `form/templates/bootstrap.html`: 渲染Bootstrap风格表单

```
...
<form method="post">
    {{ form.csrf_token }}
    <div class="form-group">
        {{ form.username.label }}
        {{ form.username(class='form-control') }}
    </div>
    <div class="form-group">
        {{ form.password.label }}
        {{ form.password(class='form-control') }}
    </div>
    <div class="form-check">
        {{ form.remember(class='form-check-input') }}
        {{ form.remember.label }}
    </div>
    {{ form.submit(class='btn btn-primary') }}
</form>
...
```

为了使用 Bootstrap，我们在模板中加载了 Bootstrap 资源。如果你运行了示例程序，可以访问 `http://localhost:5000/bootstrap` 查看渲染后的实际效果，如图 4-2 所示。

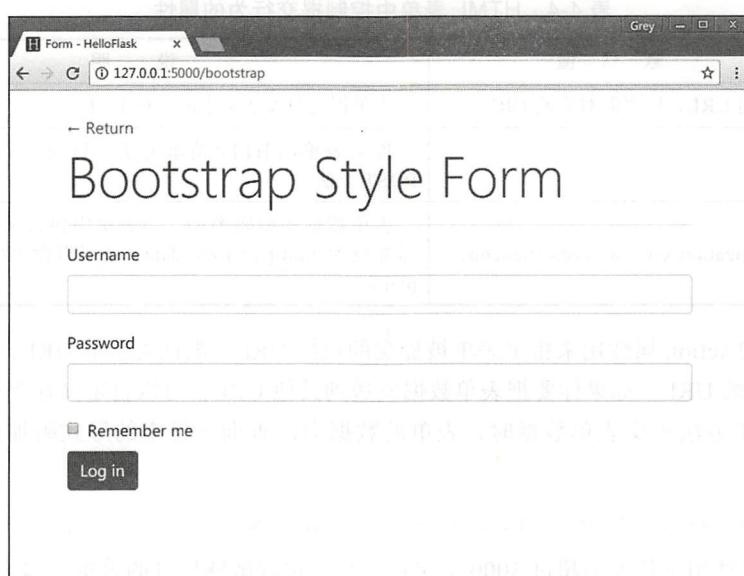


图 4-2 Bootstrap 风格表单



附注 如果你想手动编写 HTML 表单的代码，要注意表单字段的 name 属性值必须和表单类的字段名称相同，这样在提交表单时 WTForms 才能正确地获取数据并进行验证，具体会在后面介绍。

4.3 处理表单数据

表单数据的处理涉及很多内容，除去表单提交不说，从获取数据到保存数据大致会经历以下步骤：

- 1) 解析请求，获取表单数据。
- 2) 对数据进行必要的转换，比如将勾选框的值转换成 Python 的布尔值。
- 3) 验证数据是否符合要求，同时验证 CSRF 令牌。
- 4) 如果验证未通过则需要生成错误消息，并在模板中显示错误消息。
- 5) 如果通过验证，就把数据保存到数据库或做进一步处理。

除非是简单的程序，否则手动处理不太现实，使用 Flask-WTF 和 WTForms 可以极大地简化这些步骤。

4.3.1 提交表单

在 HTML 中，当 `<form>` 标签声明的表单中类型为 `submit` 的提交字段被单击时，就会创建一个提交表单的 HTTP 请求，请求中包含表单各个字段的数据。表单的提交行为主要由三个属性控制，如表 4-4 所示。

表 4-4 HTML 表单中控制提交行为的属性

属性	默 认 值	说 明
<code>action</code>	当前 URL，即页面对应的 URL	表单提交时发送请求的目标 URL
<code>method</code>	<code>get</code>	提交表单的 HTTP 请求方法，目前仅支持使用 GET 和 POST 方法
<code>enctype</code>	<code>application/x-www-form-urlencoded</code>	表单数据的编码类型，当表单中包含文件上传字段时，需要设为 <code>multipart/form-data</code> ，还可以设为纯文本类型 <code>text/plain</code>

`form` 标签的 `action` 属性用来指定表单被提交的目标 URL，默认为当前 URL，也就是渲染该模板的路由所在的 URL。如果你要把表单数据发送到其他 URL，可以自定义这个属性值。

当使用 GET 方法提交表单数据时，表单的数据会以查询字符串的形式附加在请求的 URL 里，比如：

```
http://localhost:5000/basic?username=greyli&password=12345
```

GET 方式仅适用于长度不超过 3000 个字符，且不包含敏感信息的表单。因为这种方式会直接将用户提交的表单数据暴露在 URL 中，容易被攻击者截获，示例中的情况明显是危险的。因

此，出于安全的考虑，我们一般使用 POST 方法提交表单。使用 POST 方法时，按照默认的编码类型，表单数据会被存储在请求主体中，比如：

```
POST /basic HTTP/1.0
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 30

username=greyli&password=12345
```

在第 2 章我们介绍过，Flask 为路由设置默认监听的 HTTP 方法为 GET。为了支持接收表单提交发送的 POST 请求，我们必须在 app.route() 装饰器里使用 methods 关键字为路由指定 HTTP 方法，如代码清单 4-5 所示。

代码清单4-5 form/app.py：设置监听POST方法

```
@app.route('/', methods=['GET', 'POST'])
def basic():
    form = LoginForm()
    return render_template('basic.html', form=form)
```

4.3.2 验证表单数据

表单数据的验证是 Web 表单中最重要的主题之一，这一节我们会学习如何使用 Flask-WTF 验证并获取表单数据。

1. 客户端验证和服务器端验证

表单的验证通常分为以下两种形式：

(1) 客户端验证

客户端验证（client side validation）是指在客户端（比如 Web 浏览器）对用户的输入值进行验证。比如，使用 HTML5 内置的验证属性即可实现基本的客户端验证（type、required、min、max、accept 等）。比如，下面的 username 字段添加了 required 标志：

```
<input type="text" name="username" required>
```

如果用户没有输入内容而按下提交按钮，会弹出浏览器内置的错误提示，如图 4-3 所示。

和其他附加 HTML 属性相同，我们可以在定义表单时通过 render_kw 传入这些属性，或是在渲染表单时传入。像 required 这类布尔值属性，值可以为空或是任意 ASCII 字符，比如：

```
{{ form.username(required='') }}
```

除了使用 HTML5 提供的属性实现基本的客户端验证，我们通常会使用 JavaScript 实现完善的验证机制。如果你不想手动编写 JavaScript 代码实现客户端验证，可以考虑使用各种 JavaScript 表单验证库，比如 jQuery Validation Plugin (<https://jqueryvalidation.org/>)、Parsley.js (<http://parsleyjs.org/>) 以及可与 Bootstrap 集成的 Bootstrap Validator (<http://1000hz.github.io/bootstrap-validator/>，目前仅支持 Bootstrap3 版本) 等。

客户端方式可以实时动态提示用户输入是否正确，只有用户输入正确后才会将表单数据发

送到服务器。客户端验证可以增强用户体验，降低服务器负载。

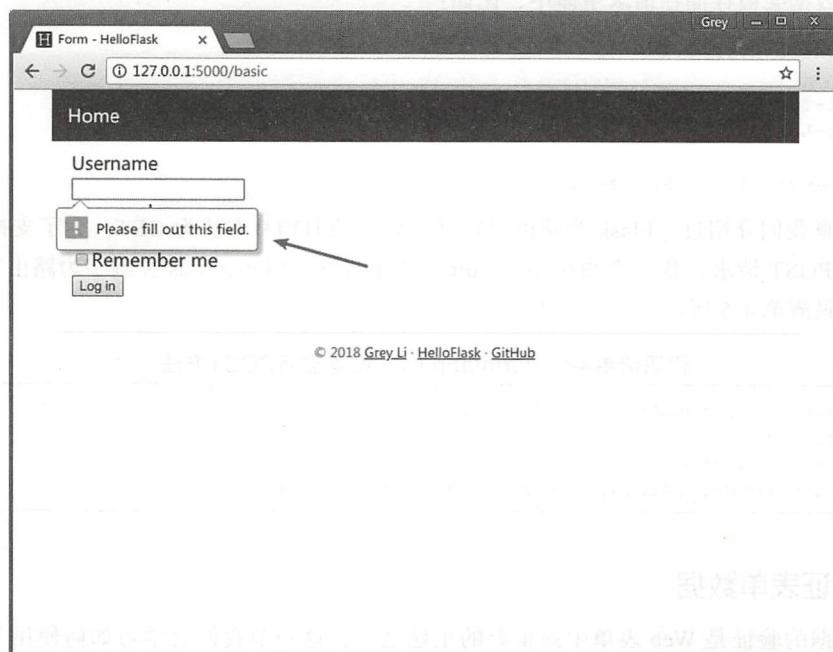


图 4-3 HTML5 表单验证

(2) 服务器端验证

服务器端验证 (server side validation) 是指用户把输入的数据提交到服务器端，在服务器端对数据进行验证。如果验证出错，就在返回的响应中加入错误信息。用户修改后再次提交表单，直到通过验证。我们在 Flask 程序中使用 WTForms 实现的就是服务器端验证。

在这里我们不必纠结使用何种形式，因为无论你是否使用客户端验证，服务器端验证都是必不可少的，因为用户可以通过各种方式绕过客户端验证，比如在客户端设置禁用 JavaScript。对于玩具程序来说，你可以不用考虑那么多，但对于真实项目来说，绝对不能忽视任何安全问题。因为客户端验证超出了本书介绍的范围，这里仅介绍服务器端验证的实现。

2. WTForms 验证机制

WTForms 验证表单字段的方式是在实例化表单类时传入表单数据，然后对表单实例调用 `validate()` 方法。这会逐个对字段调用字段实例化时定义的验证器，返回表示验证结果的布尔值。如果验证失败，就把错误消息存储到表单实例的 `errors` 属性对应的字典中，验证的过程如下所示：

```
>>> from wtforms import Form, StringField, PasswordField, BooleanField
>>> from wtforms.validators import DataRequired, Length
>>> class LoginForm(Form):
...     username = StringField('Username', validators=[DataRequired()])
...     password = PasswordField('Password', validators=[DataRequired()])
```

```

, Length(8, 128)])
>>> form = LoginForm(username='', password='123')
>>> form.data # 表单数据字典
{'username': '', 'password': '123'}
>>> form.validate()
False
>>> form.errors # 错误消息字典
{'username': [u'This field is required.'], 'password': [u'Field must be
at least 6 characters long.']}
>>> form2 = LoginForm(username='greyli', password='123456')
>>> form2.data
{'username': 'greyli', 'password': '123456'}
>>> form2.validate()
True
>>> form2.errors
{}

```

因为我们的表单使用 POST 方法提交，如果单纯使用 WTForms，我们在实例化表单类时需要首先把 request.form 传入表单类，而使用 Flask-WTF 时，表单类继承的 FlaskForm 基类默认会从 request.form 获取表单数据，所以不需要手动传入。



提示 使用 POST 方法提交的表单，其数据会被 Flask 解析为一个字典，可以通过请求对象的 form 属性获取 (request.form)；使用 GET 方法提交的表单的数据同样会被解析为字典，不过要通过请求对象的 args 属性获取 (request.args)。

3. 在视图函数中验证表单

因为现在的 basic_form 视图同时接收两种类型的请求：GET 请求和 POST 请求。所以我们要根据请求方法的不同执行不同的代码。具体来说：首先是实例化表单，如果是 GET 请求，那么就渲染模板；如果是 POST 请求，就调用 validate() 方法验证表单数据。

请求的 HTTP 方法可以通过 request.method 属性获取，我们可以使用下面的方式来组织视图函数：

```

from flask import request
...
@app.route('/basic', methods=['GET', 'POST'])
def basic():
    form = LoginForm() # GET + POST
    if request.method == 'POST' and form.validate():
        ... # 处理POST请求
    return render_template('forms/basic.html', form=form) # 处理GET请求

```

其中的 if 语句等价于：

```

if 用户提交表单 and 数据通过验证：
    获取表单数据并保存

```

当请求方法是 GET 时，会跳过这个 if 语句，渲染 basic.html 模板；当请求的方法是 POST 时（说明用户提交了表单），则验证表单数据。这会逐个字段（包括 CSRF 令牌字段）调用附加的验证器进行验证。

 **注意** 因为 WTForms 会自动对 CSRF 令牌字段进行验证，如果没有渲染该字段会导致验证出错，错误消息为“CSRF token is missing”。

Flask-WTF 提供的 `validate_on_submit()` 方法合并了这两个操作，因此上面的代码可以简化为：

```
@app.route('/basic', methods=['GET', 'POST'])
def basic():
    form = LoginForm()
    if form.validate_on_submit():
        ...
    return render_template('basic.html', form=form)
```

 **附注** 除了 POST 方法，如果请求的方法是 PUT、PATCH 和 DELETE 方法，`form.validate_on_submit()` 也会验证表单数据。

如果 `form.validate_on_submit()` 返回 True，则表示用户提交了表单，且表单通过验证，那么我们就可以在这个 if 语句内获取表单数据，如代码清单 4-6 所示。

代码清单 4-6 form/app.py：表单验证与获取数据

```
from flask import Flask, render_template, redirect, url_for, flash
...
@app.route('/basic', methods=['GET', 'POST'])
def basic():
    form = LoginForm()
    if form.validate_on_submit():
        username = form.username.data
        flash('Welcome home, %s!' % username)
        return redirect(url_for('index'))
    return render_template('basic.html', form=form)
```

表单类的 `data` 属性是一个匹配所有字段与对应数据的字典，我们一般直接通过“`form.字段属性名.data`”的形式来获取对应字段的数据。例如，`form.username.data` 返回 `username` 字段的值。在代码清单 4-6 中，当表单验证成功后，我们获取了 `username` 字段的数据，然后用来发送一条 `flash` 消息，最后将程序重定向到 `index` 视图。

 **提示** 表单的数据一般会存储到数据库中，这是我们下一章要学习的内容。这里仅仅将数据填充到 `flash()` 函数里。

在这个 if 语句内，如果不使用重定向的话，当 if 语句执行完毕后会继续执行最后的 `render_template()` 函数渲染模板，最后像往常一样返回一个常规的 200 响应，但这会造成一个问题：

在浏览器中，当单击 F5 刷新 / 重载时的默认行为是发送上一个请求。如果上一个请求是 POST 请求，那么就会弹出一个确认窗口，询问用户是否再次提交表单。为了避免出现这个容易让人产生困惑的提示，我们尽量不要让提交表单的 POST 请求作为最后一个请求。这就是为什么我们在处理表单后返回一个重定向响应，这会让浏览器重新发送一个新的 GET 请求到重定向

的目标 URL。最终，最后一个请求就变成了 GET 请求。这种用来防止重复提交表单的技术称为 PRG (Post/Redirect/Get) 模式，即通过对提交表单的 POST 请求返回重定向响应将最后一个请求转换为 GET 请求。

4.3.3 在模板中渲染错误消息

如果 form.validate_on_submit() 返回 False，那么说明验证没有通过。对于验证未通过的字段，WTForms 会把错误消息添加到表单类的 errors 属性中，这是一个匹配作为表单字段的类属性到对应的错误消息列表的字典。我们一般会直接通过字段名来获取对应字段的错误消息列表，即“form.字段名.errors”。比如，form.name.errors 返回 name 字段的错误消息列表。

像第 2 章渲染 flash() 消息一样，我们可以在模板里使用 for 循环迭代错误消息列表，如代码清单 4-7 所示。

代码清单4-7 form/templates/basic.html：渲染错误消息

```
<form method="post">
    {{ form.csrf_token }}
    {{ form.username.label }}<br>
    {{ form.username() }}<br>
    {% for message in form.username.errors %}
        <small class="error">{{ message }}</small><br>
    {% endfor %}
    {{ form.password.label }}<br>
    {{ form.password }}<br>
    {% for message in form.password.errors %}
        <small class="error">{{ message }}</small><br>
    {% endfor %}
    {{ form.remember }}{{ form.remember.label }}<br>
    {{ form.submit }}<br>
</form>
```



为了让错误消息更加醒目，我们为错误消息元素添加了 error 类，这个样式类在 style.css 文件中定义，它会将文字颜色设为红色。

如果你运行了示例程序，请访问 <http://localhost:5000/basic> 打开基本表单示例，如果你没有输入内容而按下提交按钮，会看到浏览器内置的错误提示。



在使用 DataRequired 和 InputRequired 验证器时，WTForms 会在字段输出的 HTML 代码中添加 required 属性，所以会弹出浏览器内置的错误提示。同时，WTForms 也会在表单字段的 flags 属性添加 required 标志（比如 form.username.flags.required），所以我们可以 在模板中通过这个标志值来判断是否在字段文本中添加一个 * 号或文字标注，以表示必填项。

如果你在用户名字段输入空格，在密码字段输入的数值长度小于 6，返回响应后会看到对应的错误消息显示在字段下方，如图 4-4 所示。

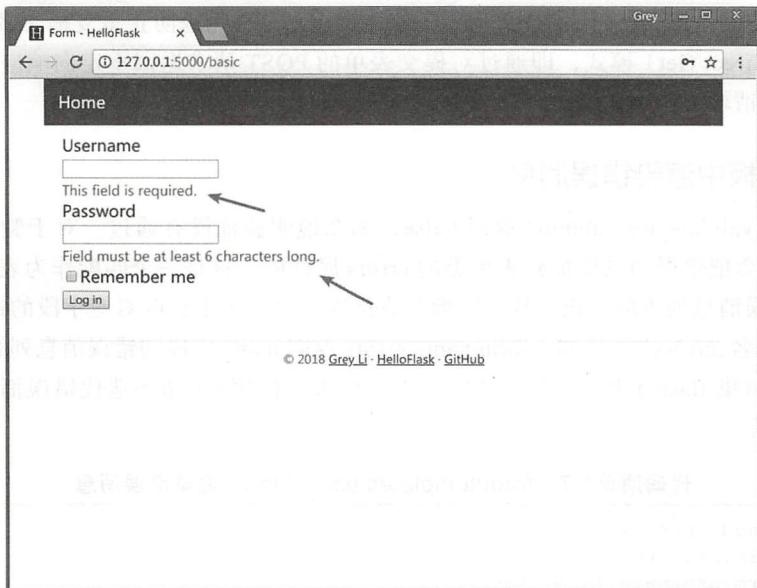


图 4-4 显示错误信息



InputRequired 验证器和 DataRequired 很相似，但 InputRequired 仅验证用户是否有输入，而不管输入的值是否有效。例如，由空格组成的数据也会通过验证。当使用 DataRequired 时，如果用户输入的数据不符合字段要求，比如在 IntegerField 输入非数字时会视为未输入，而不是类型错误。

至此，我们已经介绍了在 Python 中处理 HTML 表单的所有基本内容。完整的表单处理过程的流程图如图 4-5 所示。

4.4 表单进阶实践

这一节会介绍表单处理的相关技巧，这些技巧可以简化表单的处理过程。另外，我们还介绍了表单的一些非常规应用。

4.4.1 设置错误消息语言

WTForms 内置了多种语言的错误消息，如果你想改变内置错误消息的默认语言，可以通过自定义表单基类实现（Flask-WTF 版本 >0.14.2）。



实现这个功能需要确保 Flask-WTF 版本 >0.14.2 或单独使用 WTForms。在本书写作时，Flask-WTF 的最新版本为 0.14.2，所以这里介绍的方法暂时无法使用。

代码清单 4-8 中的示例程序创建了一个 MyBaseForm 基类，所有继承这个基类的表单类的

内置错误消息语言都会设为简体中文。

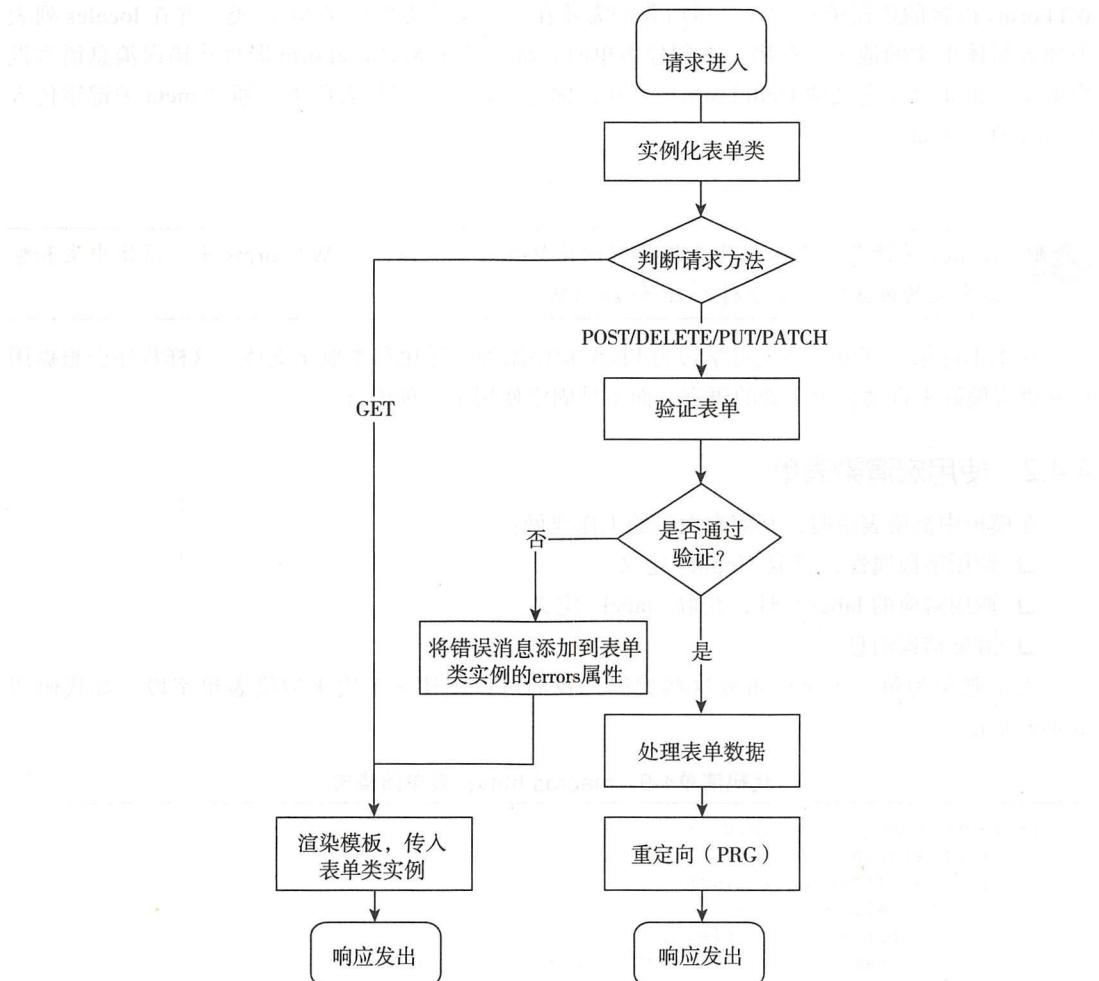


图 4-5 表单处理流程图

代码清单4-8 设置内置错误消息语言为中文

```

from flask_wtf import FlaskForm
app = Flask(__name__)
app.config['WTF_I18N_ENABLED'] = False

class MyBaseForm(FlaskForm):
    class Meta:
        locales = ['zh']

class HelloForm(MyBaseForm):
    name = StringField('Name', validators=[DataRequired()])
    submit = SubmitField()
  
```

