2017-07-14，田宇伟

Web表单处理

目录

Web表单处理

请求对象包含客户端发出的所有请求信息。其中， `request.form` 能获取**POST**请求中提交的表单数据。

有些Web表单处理任务比较单调，比如生成表单的HTML代码和验证提交的表单数据。

Flask-WTF扩展可以提供非常方便的Web表单处理体验。这个扩展对独立的WTForms包进行了包装，方便集成到 Flask 程序中。

使用 pip 安装 Flask-WTF 及其依赖：

```
(flask_env3) $ pip install flask-wtf
```

跨站请求伪造保护

默认情况下，Flask-WTF 能保护所有表单免受跨站请求伪造（Cross-Site Request Forgery， CSRF）的攻击。恶意网站把请求发送到被攻击者已登录的其他网站时就会引发 CSRF 攻击。

为了实现 CSRF 保护，Flask-WTF 需要程序设置一个密钥。Flask-WTF 使用这个密钥生成加密令牌，再用令牌验证请求中表单数据的真伪。

设置密钥的方法如下：

```
# hello.py

app = Flask(__name__)
app.config['SECRET_KEY'] = 'hard to guess string'
```

`app.config` 字典可用来存储框架、扩展和程序本身的配置变量。使用标准的字典句法就能把配置值添加到 `app.config` 对象中。这个对象还提供了一些方法，可以从文件或环境中导入配置值。

`SECRET_KEY` 配置变量是通用密钥，可在 Flask 和多个第三方扩展中使用。如其名所示，加密的强度取决于变量值的机密程度。不同的程序要使用不同的密钥，而且要保证其他人不知道你所用的字符串。

注意：为了增强安全性，密钥不应该直接写入代码，而要保存在环境变量中。[1]

表单类

使用 Flask-WTF 时，每个 Web 表单都由一个继承自 FlaskForm 的类表示。这个类定义表单中的一组字段，每个字段都用对象表示。字段对象可附属一个或多个验证函数。验证函数用来验证用户提交的输入值是否符合要求。

```
from flask_wtf import FlaskForm
from wtforms import StringField, SubmitField
from wtforms.validators import Required


class NameForm(FlaskForm):
    name = StringField('What is your name?', validators=[Required()])
    submit = SubmitField('Submit')
```

这个表单中的字段都定义为类变量，类变量的值是相应字段类型的对象。其中， NameForm 表单中有一个名为 name 的文本字段和一个名为 submit 的提交按钮。 StringField 类表示属性为 type="text" 的 <input> 元素。 SubmitField 类表示属性为 type="submit" 的 <input> 元素。字段构造函数的第一个参数是把表单渲染成 HTML 时使用的标号。

StringField 构造函数中的可选参数 validators 指定一个由验证函数组成的列表，在接受用户提交的数据之前验证数据。验证函数 Required() 确保提交的字段不为空。

WTForms 支持的HTML标准字段如下表所示：

表1 WTForms支持的HTML标准字段

字段类型	说明
StringField	文本字段
TextAreaField	多行文本字段
PasswordField	密码文本字段
HiddenField	隐藏文本字段
DateField	文本字段，值为 datetime.date 格式
DateTimeField	文本字段，值为 datetime.datetime 格式
IntegerField	文本字段，值为整数
DecimalField	文本字段，值为 decimal.Decimal
FloatField	文本字段，值为浮点数
BooleanField	复选框，值为 True 和 False
RadioField	一组单选框
SelectField	下拉列表
SelectMultipleField	下拉列表，可选择多个值
FileField	文件上传字段
SubmitField	表单提交按钮
FormField	把表单作为字段嵌入另一个表单
FieldList	一组指定类型的字段

WTForms 内建的验证函数如下标表所示：

表2 WTForms验证函数

验证函数	说明
Email	验证电子邮件地址
EqualTo	比较两个字段的值;常用于要求输入两次密码进行确认的情况
IPAddress	验证 IPv4 网络地址
Length	验证输入字符串的长度

验证函数	说明
NumberRange	验证输入的值在数字范围内
Optional	无输入值时跳过其他验证函数
Required	确保字段中有数据
Regexp	使用正则表达式验证输入值
URL	验证 URL
AnyOf	确保输入值在可选值列表中
NoneOf	确保输入值不在可选值列表中

把表单渲染为HTML

表单字段是可调用的，在模板中调用后会渲染成 HTML。假设视图函数把一个 NameForm 实例通过参数 form 传入模板，在模板中可以生成一个简单的表单。

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name() }}
    {{ form.submit() }}
</form>
```

如果想改进表单的外观，可以把参数传入渲染字段的函数，传入的参数会被转换成字段的 HTML 属性。例如，可以为字段指定 id 或 class 属性，然后定义 CSS 样式：

```
<form method="POST">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(id='my-text-field') }}
    {{ form.submit() }}
</form>
```

使用 Bootstrap 中的表单样式

Flask-Bootstrap提供了一个非常高端的辅助函数，可以使用 Bootstrap 中预先定义好的表单样式渲染整个表单。

```
{% import "bootstrap/wtf.html" as wtf %}
{{ wtf.quick_form(form) }}
```

导入的 bootstrap/wtf.html 文件中定义了一个使用 Bootstrap 渲染 Flask-WTF 表单对象的辅助函数。 wtf.quick_form() 函数的参数为 Flask-WTF 表单对象，使用 Bootstrap 的默认样式渲染传入的表单。

```
{# templates/index.html #}

{% extends "base.html" %}
{% import "bootstrap/wtf.html" as wtf %}

{% block page_content %}
    <div class="page-header">
        <h1>Hello, {% if name %}{{ name }}{% else %}Stranger{% endif %}!</h1>
    </div>
    {{ wtf.quick_form(form) }}
{% endblock %}
```

在视图函数中处理表单

在新的 hello.py 中，视图函数 index() 不仅要渲染表单，还要接收表单中的数据。

```
@app.route('/', methods=['GET', 'POST'])
def index():
    name = None
    form = NameForm()
    if form.validate_on_submit():
        name = form.name.data
        form.name.data = ''
    return render_template('index.html', form=form, name=name)
```

app.route 装饰器中添加的 methods 参数告诉 Flask 在 URL 映射中把这个视图函数注册为 GET 和 POST 请求的处理程序。如果没指定 methods 参数，就只把视图函数注册为 GET 请求的处理程序。

把 POST 加入方法列表很有必要，因为将提交表单作为 POST 请求进行处理更加便利。表单也可作为 GET 请求提交，不过 GET 请求没有主体，提交的数据以查询字符串的形式附加到 URL 中，可在浏览器的地址栏中看到。基于这个以及其他多个原因，提交表单大都作为 POST 请求进行处理。

局部变量 name 用来存放表单中输入的有效名字，如果没有输入，其值为 None 。如上述代码所示，在视图函数中创建一个 NameForm 类实例用于表示表单。提交表单后，如果数据能被所有验证函数接受，那么 validate_on_submit() 方法的返回值为 True ，否则返回 False 。这个函数的返回值决定是重新渲染表单还是处理表单提交的数据。

用户第一次访问程序时，服务器会收到一个没有表单数据的 GET 请求，所以 validate_on_ submit() 将返回 False 。 if 语句的内容将被跳过，通过渲染模板处理请求，并传入表单对象和值为 None 的 name 变量作为参数。用户会看到浏览器中显示了一个表单。

用户提交表单后，服务器收到一个包含数据的 POST 请求。 validate_on_submit() 会调用 name 字段上附属的 Required() 验证函数。如果名字不为空，就能通过验证， validate_on_submit() 返回 True 。现在，用户输入的名字可通过字段的 data 属性获取。在 if 语句中，把名字赋值给局部变量 name ，然后再把 data 属性设为空字符串，从而清空表单字段。最后一行调用 render_template() 函数渲染模板，但这一次参数 name 的值为表单中输入的名字，因此会显示一个针对该用户的欢迎消息。

如果用户提交表单之前没有输入名字， Required() 验证函数会捕获这个错误。

重定向和用户会话

刷新页面时浏览器会重新发送之前已经发送过的最后一个请求。如果这个请求是一个包含表单数据的 POST 请求，刷新页面后会再次提交表单。大多数情况下，这并不是理想的处理方式。

基于这个原因，**最好使用重定向作为 POST 请求的响应，而不是使用常规响应。**

重定向是一种特殊的响应，响应内容是 URL，而不是包含 HTML 代码的字符串。浏览器收到这种响应时，会向重定向的 URL 发起 GET 请求，显示页面的内容。这个页面的加载可能要多花几微秒，因为要先把第二个请求发给服务器。除此之外，用户不会察觉到有什么不同。现在，最后一个请求是 GET 请求，所以刷新命令就能像预期的那样正常使用。这个技巧称为**Post/重定向/Get**模式。

另外一个问题，程序处理 POST 请求时，使用 form.name.data 获取用户输入的名字，可是一旦这个请求结束，数据也就丢失了。因为这个 POST 请求使用重定向处理，所以程序需要保存输入的名字，这样重定向后的请求才能获得并使用这个名字，从而构建真正的响应。

程序可以把数据存储在用户会话中，在请求之间"记住"数据。 **用户会话**是一种私有存储，存在于每个连接到服务器的客户端中。 [2]

```
from flask import Flask, render_template, session, redirect, url_for

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'))
```

在程序的前一个版本中，局部变量 name 被用于存储用户在表单中输入的名字。这个变量现在保存在用户会话中，即 session['name'] ，所以在两次请求之间也能记住输入的值。

现在，包含合法表单数据的请求最后会调用 redirect() 函数。 redirect() 是个辅助函数，用来生成 HTTP 重定向响应，其参数是重定向的 URL，这里使用的重定向 URL 是程序的根地址，因此重定向响应本可以写得更简单一些，写成 redirect('/')，但却会使用 Flask 提供的 URL 生成函数 url_for() 。推荐使用 url_for() 生成 URL，因为这个函数使用 URL 映射生成 URL，从而保证 URL 和定义的路由兼容，而且修改路由名字后依然可用。

url_for() 函数的第一个且唯一必须指定的参数是端点名，即路由的内部名字。默认情况下，路由的端点是相应视图函数的名字。在这个示例中，处理根地址的视图函数是 index()，因此传给 url_for() 函数的名字是 index 。

最后一处改动位于 render_function() 函数中，使用 session.get('name') 直接从会话中读取 name 参数的值。和普通的字典一样，这里使用 get() 获取字典中键对应的值以避免未找到键的异常情况，因为对于不存在的键， get() 会返回默认值 None 。

Flash消息

请求完成后，有时需要让用户知道状态发生了变化。这里可以使用确认消息、警告或者错误提醒。一个典型例子是，用户提交了有一项错误的登录表单后，服务器发回的响应重新渲染了登录表单，并在表单上面显示一个消息，提示用户用户名或密码错误。

flash() 函数可实现这种效果：

```
from flask import Flask, render_template, session, redirect, url_for, flash

@app.route('/', methods=['GET', 'POST'])
def index():
    form = NameForm()
    if form.validate_on_submit():
        old_name = session.get('name')
        if old_name is not None and old_name != form.name.data:
            flash('Looks like you have changed your name!')
        session['name'] = form.name.data
        return redirect(url_for('index'))
    return render_template('index.html', form=form, name=session.get('name'))
```

每次提交的名字都会和存储在用户会话中的名字进行比较，而会话中存储的名字是前一次在这个表单中提交的数据。如果两个名字不一样，就会调用 flash() 函数，在发给客户端的下一个响应中显示一个消息。

仅调用 flash() 函数并不能把消息显示出来，程序使用的模板要渲染这些消息。最好在基模板中渲染 Flash 消息，因为这样所有页面都能使用这些消息。 Flask 把 get_flashed_messages() 函数开放给模板，用来获取并渲染消息。

```
{% block content %}
    <div class="container">
        {% for message in get_flashed_messages() %}
            <div class="alert alert-warning">
                <button type="button" class="close" data-dismiss="alert">&times;</button>
                {{ message }}
            </div>
        {% endfor %}
        {% block page_content %}{% endblock %}
    </div>
{% endblock %}
```

在模板中使用循环是因为在之前的请求循环中每次调用 flash() 函数时都会生成一个消息，所以可能有多个消息在排队等待显示。 get_flashed_messages() 函数获取的消息在下次调用时不会再次返回，因此 Flash 消息只显示一次，然后就消失了。

脚注:

[1] 后面章节会学习。

[2] 默认情况下， 用户会话保存在客户端 cookie 中，使用设置的 SECRET_KEY 进行加密签名。 如果篡改了 cookie 中的内容，签名就会失效，会话也会随之失效。