Flask Web Development

发布 *1.0.0*

lin

2020 年 10 月 01 日

Contents

CHAPTER 1

前言

Flask 是一个微型 Web 开发框架，其目标是保持核心简单而又可扩展。Flask 提供了一个强健的核心，其中包含 Web 应用都需要的基本功能，而对于其他的个性化功能则交给生态系统中的众多第三方扩展。Flask 原生不支持数据库访问、Web 表单验证和用户身份验证等高级功能。这些功能以及其他大多数 Web 应用需要的核心服务都是以扩展形式实现，然后再和核心包集成。我们可以针对实际需求灵活选择扩展，比如使用何种类型的数据库、使用何种模板引擎等等。

本博客主要记录我自己学习使用 Flask 进行 Web 开发应用的整个历程。博客使用的是 Sphinx 来生成文档，使用 Github 托管文档，并使用 Read the Doc 发布文档。

CHAPTER 2

第一章：Flask 基本原理与核心知识

2.1 1.1 依赖

当安装 Flask 时，以下依赖包会被自动安装：

- Werkzeug：用于实现 Web 服务器网关接口 (WSGI, Web server gateway interface)，应用和服务之间的标准 Python 接口。
- Jinja2：是一个模板引擎，用于渲染页面。
- MarkupSafe：与 Jinja 共用，用来转义模板，在渲染页面时用于避免不可信的输入，防止注入攻击。
- ItsDangerous：对信息进行加密，用于保护 cookie 和 session。
- Click：是一个命令行应用的框架。用于提供 `flask` 命令，并允许添加自定义管理命令。

2.2 1.2 虚拟环境创建与 Flask 安装

安装 Flask 建议使用虚拟环境。为什么要使用虚拟环境？随着 Python 项目越来越多，可能会需要不同的版本的 Python 包，出现版本不兼容的情况。为每个项目单独创建虚拟环境，独立安装所需的 Python 库，这样就可以隔离不同项目之间的 Python 库，也可以隔离项目与系统预装的 Python 库。

创建虚拟环境的方式有多种，下面依次进行介绍。

2.2.1 1.2.1 在 Python3 中创建虚拟环境

Python3 内置了用于创建虚拟环境的 venv 模块。在 CentOS7 环境下，创建一个项目文件夹，然后创建一个虚拟环境。创建完成后项目文件夹中会有一个 venv 文件夹：

```
$ mkdir flasky
$ cd flasky
$ python3 -m venv venv
```

创建虚拟环境的命令格式：

`python3 -m venv virtual-environment-name`

`-m venv` 作用：以独立的脚本运行标准库中的 venv 包，`virtual-environment-name` 为虚拟环境名称。

2.2.2 1.2.2 在 Python2 中创建虚拟环境

Python2 没有内置的 venv 包，需要先安装第三方模块 `virtualenv`，然后再创建虚拟环境：

```
$ pip install virtualenv
$ python2 -m virtualenv venv
```

在开始工作前，先要激活相应的虚拟环境：

```
$ . venv/bin/activate  # 激活
$ deactivate # 退出激活环境
```

激活虚拟环境后，安装 Flask：

```
$ pip install flask
```

执行上面命令，会安装 Flask 及其所需依赖。可以使用 pip freeze 查看虚拟环境中安装了哪些包：

```
(venv) [root@centos7 flasky]$ pip freeze
click==7.1.2
Flask==1.1.2
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

2.2.3 1.2.3 使用官方推荐的 pipenv 创建虚拟环境

pipenv 是官方推荐的包管理工具，能够为项目创建和管理虚拟环境。

安装 pipenv：

```
$ pip install pipenv
```

在指定目录下创建虚拟环境并安装 Flask：

创建虚拟环境时使用本地默认的 python，也可使用指定 python 版本，如 `pipenv --python 3.8`。

```
$ cd flasky
$ pipenv install
$ pipenv shell    # 进入虚拟环境
$ pipenv install flask    # 安装 Flask 包
$ pipenv uninstall flask # 卸载包
$ pipenv graph  # 查看依赖关系
$ exit  # 退出虚拟环境
```

更多 pipenv 命令可参考 github 上 pipenv 项目。

CHAPTER 3

第二章：Flask 应用的基本结构

本章节将介绍 Flask 应用的基本结构，了解各部分的作用。

3.1 2.1 Flask Web 应用的不同组成部分

首先看一个最小的 Web 应用：

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello, World!</h1>'
```

上面这个脚本定义了一个**应用实例**、一个**路由**和一个**视图函数**，即是 Flask Web 应用的各组成部分。

3.1.1 2.1.1 应用实例

所有 Flask 应用都需创建一个**应用实例**，是 Flask 类的对象。Web 服务器使用 WGSI 协议，把接收自客户端的所有请求都转交给这个对象。

```
from flask import Flask
app = Flask(__name__)
```

Flask 类实例化时必须传入一个指定的参数，即应用主模块或者包的名称。大多数应用中，传入 `__name__` 作为参数。Flask 用这个参数确定应用的位置，进而找到其他文件的位置，如图像和模板。

3.1.2 2.1.2 路由

客户端（如浏览器）将请求发送给 Web 服务器，Web 服务器再把请求发送给 Flask 应用实例。应用实例需要知道对每个 URL 的请求要执行哪些代码，所以保存了 URL 到 Python 函数的映射关系。处理 URL 和函数值间关系的程序称为**路由**。

在 Flask 应用中定义路由的最简单方式是使用应用实例提供的 app.route 装饰器，把函数绑定到 URL:

```
@app.route('/')
def index():
    return '<h1>Hello, World!</h1>'
```

装饰器将函数注册为事件处理程序，在特定事件发生时调用。上述示例把 index() 函数注册为应用程序根地址的处理程序。

在日常应用中，URL 可能包含可变的部分，路由 URL 中放在尖括号里的内容就是动态部分，任何能匹配到静态部分的 URL 都会映射到这个路由上。

```
@app.route('/user/<username>')
def index():
    return '<h1>Hello, {}!</h1>'.format(username)
```

路由中的动态部分除了默认使用字符串，还支持其他类型。通过使用 `<converter:variable_name>` ，可以选择性的加上一个转换器，为变量 `variable_name` 指定规则。

转换器（converter）类型：

转换器类型	描述
string	接受任何不包含斜杠的文本
int	接受正整数
float	接受正浮点数
path	类似 string，但可以包含斜杠
uuid	接受 UUID 字符串

例如，路由 `/user/<int:id>` 只会匹配动态片段 id 为整数的 URL。

3.1.3 2.1.3 视图函数

处理入站请求的函数为**视图函数**。上面示例中的 `index()` 函数就是一个视图函数。如果应用部署在域名为 www.example.com 的服务器上，浏览器访问该域名后，会触发服务器执行视图函数 `index()`，函数的返回值为响应，即浏览器接收到的内容。

3.1.4 2.1.4 url_for

`url_for()` 函数用于构建指定函数的 URL，将视图函数名作为第一个参数。它可以接受任意个关键字参数，每个关键字参数对应 URL 中的变量。未知变量将添加到 URL 中作为查询参数。

```
from flask import Flask, url_for
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello, World!</h1>'

with app.test_request_context():
    print(url_for('index', _external=True))
```

返回一个绝对地址：http://localhost/

3.1.5 2.1.5 Web 开发服务器

Flask 应用自带 Web 开发服务器，通过 `flask run` 命令启动：

```
(venv) [root@centos7 flasky]$ export Flask_APP=hello.py
(venv) [root@centos7 flasky]$ flask run
* Serving Flask app "hello.py"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Flask Web 开发服务器也可以通过编程的方式启动，调用 `app.run()` 方法。若要启动应用，需要执行应用的主脚本，在脚本中包含下面代码：

```
if __name__ == '__main__':
    app.run()
```

3.1.6 2.1.6 调试模式

Flask 应用可以在调试模式下运行。调试模式默认禁用，若想启用，在执行 `flask run` 命令前设定 `FLASK_DEBUG=1` 环境变量：

```
(venv) [root@centos7 flasky]$ export Flask_APP=hello.py
(venv) [root@centos7 flasky]$ export FLASK_DEBUG=1
(venv) [root@centos7 flasky]$ flask run
* Serving Flask app "hello.py" (lazy loading)
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 277-103-738
```

若想以编程的方式启动调试模式，就使用 `app.run(debug=True)`。

3.2 2.2 请求与响应

3.2.1 2.2.1 应用和请求上下文

Flask 从客户端收到请求时，要让视图函数能访问一些对象，才能处理请求。要想让视图函数能访问请求对象，一种直接方式是将其作为参数传入视图函数，不过会导致每个视图函数都会多出一个参数。除了访问请求对象，如果还要访问其他对象，情况便的糟糕。为了避免这种情况，Flask 使用上下文机制临时把某些对象变为全局可访问。

在 Flask 中有两种上下文：**应用上下文**（application context）和**请求上下文**（request context）。

Flask 上下文全局变量：

变量名	上下文	说明
`current_app`	应用上下文	当前应用的应用实例
`g`	应用上下文	处理请求时用作临时存储的对象，每次请求都会重设
`request`	请求上下文	请求对象，封装了客户端发出的 HTTP 请求中的内容
`session`	请求上下文	用户会话，值为一个字典，存储请求之间需要“记住”的值

Flask 在分派请求之前激活（或推送）应用和请求上下文，请求处理完成后再将其删除。应用上下文被推送后，就可以在当前线程中使用 `current_app` 和 `g` 变量。请求上下文被推送后，就可以使用 `request` 和 `session` 变量。如果使用这些变量时没有激活应用和请求上下文，就会导致错误。

获取应用上下文的方法是在应用实例上调用 `app.app_context()`：

```
from flask import Flask, current_app

app = Flask(__name__)
with app.app_context():
    print(current_app.name)    # __main__
```

3.2.2 2.2.2 请求分派

应用收到客户端发送的请求时，要找到处理该请求的视图函数。Flask 在应用的 URL 映射中查找请求的 URL。URL 映射是 URL 和视图函数间的对应关系。Flask 使用 app.route 装饰器构建映射。

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello, World!</h1>'

print(app.url_map)
# 输出
Map([<Rule '/' (OPTIONS, GET, HEAD) -> index>,
 <Rule '/static/<filename>' (OPTIONS, GET, HEAD) -> static>])
```

输出中的 / 路由是在应用中使用 app.route 定义，/static/<filename> 路由是 Flask 添加的特殊路由，用于访问静态文件。URL 映射中的 (OPTIONS, GET, HEAD) 是请求方法。

3.2.3 2.2.3 请求对象

Flask 通过上下文变量 request 对外开放请求对象，包含客户端发送的 HTTP 请求的全部信息。使用时需要从 flask 模块中导入请求对象：

```
from flask import request
```

请求对象中常用的属性和方法：

属性或方法	说明
form	一个字典，存储请求提交的所有表单字段
args	一个字典，存储通过 URL 查询字符串传递的所有参数
values	一个字典，form 和 args 的合集
cookies	一个字典，存储请求的所有 cookie
headers	一个字典，存储请求的所有 HTTP 首部
files	一个字典，存储请求上传的所有文件
get_data()	返回请求主体缓存的数据
blueprint	处理请求的 Flask 蓝本的名称
endpoint	处理请求的 Flask 端点名称
method	HTTP 请求方法，如 GET 和 POST
scheme	URL 方案（http 或 https）
is_secure()	返回安全的连接（HTTPS）发送请求时返回 True
host	请求定义的主机名，如果客户端定义了端口号，还包括端口号
path	URL 的路径部分
url	客户端请求的完整 URL
base_url	同 url，但没有查询字符串部分
remote_addr	客户端的 IP 地址

- 示例 1：使用 `form` 属性处理表单数据

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
            error = 'Invalid username/password'
    return render_template('login.html', error=error)
```

- 示例 2：使用 `args` 属性操作 URL （如 `?key=value` ）中提交的参数

```
searchword = request.args.get('key', '')
```

3.2.4 2.2.4 请求钩子

有时在处理请求之前或之后有一些工作需要统一处理，比如创建数据库连接、验证用户身份等。为了避免在每个视图函数中都重复编写代码，Flask 提供了注册通用函数的功能，即请求钩子。注册的函数可在请求被反派到视图函数之前或之后调用。

请求钩子通过装饰器实现。Flask 支持以下 4 中钩子：

- before_request：注册一个函数，在每次请求之前运行。
- before_first_request：注册一个函数，只在处理第一个请求之前运行。可以通过这个钩子添加服务器初始化任务。
- after_request：注册一个函数，如果没有未处理的异常抛出，在每次请求之后运行。
- teardown_request：注册一个函数，即使有未处理的异常抛出，也在每次请求之后运行。

```
from flask import Flask

app = Flask(__name__)

@app.before_first_request
def before_first_request():
    print('before_first_request')

@app.before_request
def before_request():
    print('before_request')

@app.after_request
def after_request(response):
    print('after_request')
    return response

@app.teardown_request
def teardown_request(exc):
    print('teardown_request')

@app.route("/")
def index():
    return '<h1>Hello, World!</h1>'

if __name__ == '__main__':
    app.run(debug=True)
```

3.2.5 2.2.5 响应

Flask 调用视图函数后，会将其返回值作为响应的内容。视图函数的返回值会自动转换为一个响应对象。如果返回值是一个字符串，那么会被转换为一个包含作为响应体的字符串、一个出错代码和一个 `text/html` 类型的响应对象。如果返回值是一个字典，那么会调用 `jsonify()` 来产生一个响应。转换规则如下：

- 如果视图返回的是一个响应对象，则直接返回。
- 如果返回的是一个字符串，那么根据这个字符串和缺省参数生成一个用于返回的响应对象。
- 如果返回的是一个字典，那么调用 `jsonify` 创建一个响应对象。

响应对象常使用的属性和方法：

属性或方法	说明
status_code	HTTP 数字状态码
headers	一个类似字典的对象，包含随响应发送的所有首部
set_cookie()	为响应添加一个 cookie
delete_cookie()	删除一个 cookie
content_length	响应主体的长度
content_type	响应主体的媒体类型
set_data()	使用字符串或字节值设定响应
get_data()	获取响应主体

示例一：返回字符串：

如果返回的响应需要使用状态码，可以作为第二个返回值，添加在响应字符串之后。

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Bad Request!</h1>', 400

if __name__ == '__main__':
    app.run()
```

示例二：返回模板文件

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route("/")
```

(下页继续)

(续上页)

```
def error():
    return render_template("error.html"), 404

if __name__ == '__main__':
    app.run()
```

示例三：直接返回响应对象

使用 make_response() 创建一个响应对象。

```
from flask import Flask, make_response

app = Flask(__name__)

@app.route('/')
def index():
    response = make_response('<h1>Hello, World!</h1>')
    return response

if __name__ == '__main__':
    app.run()
```

示例四：重定向

重定向是响应的特殊类型，会告诉浏览器一个新的 URL，用以加载新页面。

```
from flask import Flask, redirect

app = Flask(__name__)

@app.route('/')
def index():
    return redirect('http://www.baidu.com')

if __name__ == '__main__':
    app.run()
```

示例五：jsonify 返回响应对象

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')
def index():
```

(下页继续)

(续上页)

```
    d = {'name': 'lin', 'age': '27'}
    return jsonify(d)

if __name__ == '__main__':
    app.run()
```

CHAPTER 4

第三章：模板

视图的作用很明确，即生成请求的响应。视图函数中往往会涉及两个过程，即业务逻辑和表现逻辑。在大型项目中，把业务逻辑和表现逻辑混在一起会导致代码难以理解和维护。把表现逻辑移到模板中可以提升应用的可维护性。模板是包含响应文本的文件，其中包含用占位符变量表示的动态部分。使用真实值替换变量，再返回最总得到的响应字符串，这一过程称为渲染。为了渲染模板，在 Flask 中，使用了 Jinja2 这个强大的模板引擎。

4.1 3.1 Jinja2 模板引擎

Jinja2 是一种面向 Python 的现代且易于设计的模板语言。在该部分介绍 Jinja2 的语法和语义结构。

4.1.1 3.1.1 变量

模板变量通过上下文字典定义并传递给模板。在模板中使用 `{{ variable }}` 结构表示一个变量，这是一种特殊的占位符，告诉模板引擎这个位置的值从渲染模板时使用的数据获取。

Jinja2 能识别所有类型的变量，如列表、字典和对象等。如果要访问变量中的属性，可以使用 `.` 或 `[]` 来访问。

```
{{ mydict['key'] }}
{{ mydict.key }}
{{ myobj.attr }}
{{mylist[2]}}
```

变量的值可以使用**过滤器**修改。过滤器添加在变量名之后，二者之间以竖线分隔，例如将变量的值变为首字母大写的形式：

```
{{ name|capitalize}}
```

Jinja2 提供的常用过滤器：

过滤器名	说明
safe	渲染时不转义
capitalize	把值的首字母转换成大写，其他字母转换成小写
lower	把值转换成小写形式
upper	把值转换成大写形式
title	把值中每个单词的首字母转换成大写
trim	把值的首尾空格删掉
striptags	渲染之前把值中所有的 HTML 标签删掉

4.1.2 3.1.2 注释

如果要把模板中的一部分内容注释掉，使用 `{# ... #}`：

```
{# note: disabled template because we no longer use this
    {% for user in users %}
        ...
    {% endfor %}
#}
```

4.1.3 3.1.3 控制结构

Jinja2 提供了多种控制结构，可用来改变模板的渲染流程。

- 条件控制（if-elif-else）

```
{% if condition1 %}
    ...
{% elif condition2 %}
    ...
{% else %}
    ...
{% endif %}
```

- for 循环

```
<ul>
{% for comment in comments%}
    <li>{{ comment }}</li>
{% endfor %}
</ul>
```

可以结合 `if` 进行一些条件过滤：

```
{% for comment in comments if condition %}
    ...
{% endfor %}
```

- 宏

宏类似 Python 中的函数。

```
{% macro render_comment(comment) %}
    <li>{{ comment }}</li>
{% endmacro %}
```

调用宏：

```
<ul>
    {% comment in comments %}
        {{ render_comment(comment) }}
    {% endfor %}
</ul>
```

为了重复使用宏，可以把宏保存在单独的文件中，然后再需要使用的模板中导入：

```
{% import 'macros.html' as macros %}
<ul>
    {% comment in comments %}
        {{ macros.render_comment(comment) }}
    {% endfor %}
</ul>
```

4.1.4 3.1.4 模板继承

模板继承是指将公用的一部分代码抽取出来放到一个基模板中，然后子模板继承这部分内容。模板继承包括基模板和子模板。基模板里包含了网站里基本元素的基本骨架，但里面有一些空的或不完善的块（block）需要用子模板来填充。

继承语法：

```
{% extends “模板名称” %}
```

Jinja2 使用 `block` 和 `endblock` 指定在基模板中定义内容区块。

示例：在 templates 目录中创建 base.html、index.html 文件。

base.html 模板的内容：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
    {% block head %}
    <title>{% block title %}{% endblock %} - 我的网站</title>
    {% endblock %}
</head>
<body>
    {% block body %}
        这是基模板（base.html）中的内容
    {% endblock %}
</body>
</html>
```

index.html 模板的内容：

```
{% extends "base.html" %}
{% block title %} 网站首页{% endblock %}
{% block body %}
    {{ super() }}
    <h4> 这是网站首页（index.html）的内容! </h4>
{% endblock %}
```

应用主程序 app.py：

```
from flask import Flask,render_template

app = Flask(__name__)
```

(下页继续)

(续上页)

```
@app.route('/')
def index():
    return render_template('index.html')

if __name__ == "__main__":
    app.run(debug=True)
```

执行主程序 app.py 后打开 http://127.0.0.1:5000/ 可以看到网站内容：

默认情况下，如果子模块实现了父模板定义的 block，那么子模板 block 中的代码就会覆盖父模板中的代码。例如，上述示例中在子模板 index.html 中 `{% block title %}` 定义内容会覆盖基模板 base.html 中的相应位置。如果想在子模版中仍然保持父模版代码，需要用 `super()` 函数调用。此外，如果想要在一个 block 中调用其他 block 中的代码，可以通过 `{{ self. 其他 block 名称 () }}` 实现。

4.2 3.2 静态文件

Web 应用中不仅由 Python 代码和模板组成。多数应用还会使用静态文件，例如模板中 HTML 代码引用的图像、Javascript 源码文件和 CSS。

默认设置下，Flask 在应用根目录中 static 这个子目录下寻找静态文件。如果需要，可以在 static 文件夹中使用子文件夹存放文件。

CHAPTER 5

第四章：Web 表单

本章介绍如何在 Flask 中使用 Flask-WTF 扩展来处理表单。这个扩展对独立的 WTForms 包进行了包装，方便集成到 Flask 应用中。

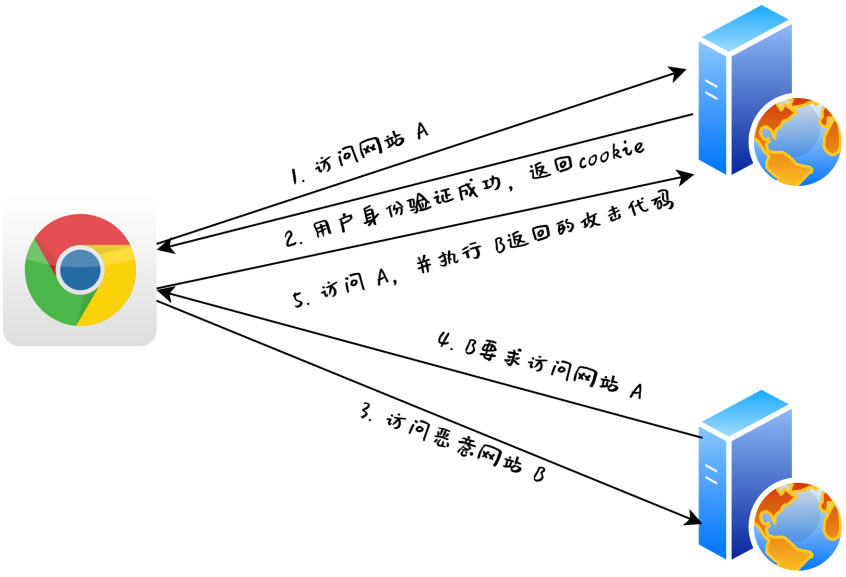
5.1 4.1 Web 表单

5.1.1 4.1.1 跨站请求伪造保护

跨站请求伪造（CSRF）：是一种挟制用户在当前已登录的 Web 应用程序上执行非本意的操作的攻击方法。

CSRF 攻击原理：

- 用户打开浏览器，访问受信任网站 A。登录成功后，网站 A 产生 Cookie 信息并返回给浏览器。
- 用户未退出网站 A 之前，在同一浏览器中，打开一个页面访问网站 B。
- 网站 B 接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访问第三方站点 A。
- 浏览器在接收到这些攻击性代码后，根据网站 B 的请求，在用户不知情的情况下携带 Cookie 信息，向网站 A 发出请求。网站 A 并不知道该请求其实是由 B 发起的，所以会根据用户的 Cookie 信息处理该请求，导致来自网站 B 的恶意代码被执行。



在 Flask 中，Flask-WTF 可以保护表单免受跨站请求伪造攻击。为了实现 CSRF 保护，Flask-WTF 需要应用程序配置一个加密密钥。Flask-WTF 使用这个加密密钥去生成安全令牌存储在用户会话中，用于验证请求表单数据的真实性。

示例：在 Flask 应用中配置密钥

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

app.config 字典可用来存储 Flask、扩展和程序自身的配置变量。

5.1.2 4.1.2 表单类

使用 Flask-WTF 时，在服务器端，每个 Web 表单都是由一个继承 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，用于验证用户提交的数据是否有效。

示例：定义一个简单的 Web 表单，包含一个文本字段和一个提交按钮。

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

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

WTForms 支持的 HTML 标准字段（部分）：

字段类型	说明
BooleanField	复选框，值为 True 和 False
FileField	文件上传字段
TextAreaField	多行文本字段
StringField	文本字段
SubmitField	表单提交按钮
SelectField	下拉列表
PsswordField	密码文本字段
HiddenField	隐藏的文本字段

WTForms 内建的验证函数（部分）：

验证函数	说明
DataRequired	确保转换类型后的字段中有数据
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
InputRequired	确保转换类型前字段中有数据
IPAddress	验证 IPV4 网络地址
Length	验证输入字符串的长度
Regexp	使用正则表达式验证输入值

5.1.3 4.1.3 把表单渲染成 HTML

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

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

除了上面这种简单的方式，还可以使用 Flask-Bootstrap 渲染表单：

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

5.1.4 4.1.4 在视图函数中处理表单

在视图函数中有两个任务：接收用户在表单中填写的数据、渲染表单。

示例：

```
@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)
```

CHAPTER 6

第四章：Web 表单

本章介绍使用包和模块组织大型应用的方式。

6.1 5.1 大型应用的结构

6.1.1 5.1.1 项目结构