2017-07-14，田宇伟

Flask框架简介

目录

Flask框架简介

Flask 是非常流行的Python Web框架。

官方网站：http://flask.pocoo.org

github：https://github.com/pallets/flask

Flask特性

- 非常齐全的官方文档
- 社区活跃度非常高
- 具备良好的扩展机制和第三方扩展环境
- 微框架的形式给开发者很大的选择空间
- Pocoo团队出品

Flask 主要依赖三个库：

Werkzeug
: 路由、调试和 Web 服务器网关接口（Web Server Gateway Interface，WSGI）。

Jinja2
: 默认的模板引擎。

Itsdangerous

基于Django签名模块的签名实现。

Flask 并不原生支持数据库访问、Web 表单验证和用户认证等高级功能。

其他特性

- 内置开发用服务器和debugger
- 集成单元测试（unit testing）
- RESTful request dispatching
- 支持secure cookies（client side sessions）
- 100% WSGI 1.0兼容
- Unicode based
- Google App Engine兼容

Flask 安装

1. 创建虚拟环境

   使用**virtualenvwrapper**创建一个虚拟环境（Python 3版本）：

   ```
   mkvirtualenv -p python3.6 flask_env3
   ```

   命令执行结束，自动激活虚拟环境flask_env3（只会影响当前的命令行会话）。命令行提示符会被修改，加入环境名：

   ```
   (flask_env3) $
   ```

   在命令行提示符下输入 `deactivate` 取消激活虚拟环境：

   ```
   (flask_env3) $ deactivate
   ```

2. 使用**pip**安装Python包

   执行下述命令在虚拟环境中安装 Flask：

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

   其会自动安装 Flask 及其依赖。安装完之后，可以验证 Flask 是否安装成功，启动 Python 解释器，尝试导入 Flask：

   ```
   (flask_env3) $ python
   >>> import flask
   >>>
   ```

   如果没有错误提示，则表示 Flask 安装成功。

一个完整的程序

```
# hello.py
from flask import Flask


app = Flask(__name__)


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

要想运行这个程序，请确保激活上面创建的虚拟环境，并在其中安装好了 Flask 。

设置环境变量：

```
(flask_env3) $ export FLASK_APP=hello.py
(flask_env3) $ export FLASK_DEBUG=1
```

在Windows中需将 `export` 替换为 `set` 。

使用下述命令启动程序：

```
(flask_env3) $ flask run
 * Serving Flask app "hello"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

打开Web浏览器，在地址栏中输入 `http://127.0.0.1:5000/` 。

调试模式

上面启动命令中通过 `export FLASK_DEBUG=1` 启用调试模式，服务器会在代码修改后自动重新载入。

如果 `FLASK_DEBUG` 设置为 `0` 则将关闭调试模式。

动态路由

```
# hello.py
from flask import Flask


app = Flask(__name__)


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


@app.route('/user/<name>')
def user(name):
    return '<h1>Hello, %s!</h1>' % name
```

访问 `http://localhost:5000/user/david` ，程序会显示一个使用 `name` 动态参数生成的欢迎消息。可尝试使用不同的名字，可看到试图函数总是使用指定的名字成成相应。

动态URL规则

URL规则可以添加变量部分，即将符合同种规则的URL抽象称一个URL模式，如 `/item/1/` 、 `/item/2/` 、 `/item/3/` ... 假如不抽象，就得这样写：

```
@app.route('/item/1/')
@app.route('/item/2/')
@app.route('/item/3/')
def item(id):
    return 'Item: {}'.format(id)
```

正确的用法是：

```
@app.route('/item/<id>/')
def item(id):
    return 'Item: {}'.format(id)
```

尖括号中的内容是动态的，凡是匹配到 `/item/` 前缀的URL都会被映射到这个路由上，在内部把 `id` 作为参数而获得。

它使用了特殊的字段标记**<variable_name>**，默认类型是字符串。如果需要指定参数类型则可标记成**<converter:variable_name>**这样的格式，**converter**有以下几种：

string
接受任何没有斜杠 “/” 的文本（默认）
int
接受整数
float
同 `int` ，但是接受浮点数
path
和默认的相似，但也接受斜杠
uuid
只接受uuid字符串
any

可以指定多种路径，但是需要传入参数

```
@app.route('/<any(a, b):page_name>/')
```

访问 `/a/` 和 访问 `/b/` 都符合这个规则， `/a/` 对应的 `page_name` 就是 `a` 。

如果不希望定制子路径，还可以通过**传递参数**的方式。比如 `/people/?name=a` ， `/people/?name=b` ，这样即可通过 `name = request.args.get('name')` 获得传入的 `name` 值。

如果使用**POST**方法，表单参数需要通过 `request.form.get('name')` 获得。

唯一URL

Flask的URL规则基于Werkzeug的路由模块，这个模块背后的思想是希望保证优雅且唯一的URL。

举个例子：

```
@app.route('/projects/')
def projects():
    return 'The project page'
```

访问一个结尾不带斜线的URL会被重定向到带斜线的规范的URL上去。

再看一个例子：

```
@app.route('/about')
def about():
    return 'The about page'
```

URL结尾不带斜线，很像文件的路径，当访问带斜线的URL（ `/about/` ）会产生一个404 “Not Found”错误。

请求—响应循环

接下来介绍 Flask 的工作方式，了解这个框架的一些设计理念。

程序和请求上下文

Flask 从客户端收到请求时，要让视图函数能访问一些对象来处理请求。 **请求对象**封装了客户端发送的 HTTP 请求。

要想让视图函数能够访问请求对象，一个方式是**将其作为参数传入视图函数**，不过这会导致程序中的每个视图函数都增加一个参数。如果视图函数在处理请求时还要访问其他对象，情况会变得更糟。

为了避免大量可有可无的参数把视图函数弄得一团糟，Flask 使用**上下文**临时把某些对象变为**全局可访问**。视图函数中使用上下文：

```
from flask import request
```

```
@app.route('/')
def index():
    user_agent = request.headers.get('User-Agent')
    return '<p>Your browser is %s</p>' % user_agent
```

上面这个视图函数把 request 当作全局变量使用。 Flask 使用上下文让特定的变量在一个线程中全局可访问，与此同时却不会干扰其他线程。 [1]

Flask 中有两种上下文：**程序上下文**和**请求上下文**。

变量名	上下文	说明
current_app	程序上下文	当前激活程序的程序实例
g	程序上下文	处理请求时用作临时存储的对象，每次请求都会重设这个变量
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，用于存储请求之间需要"记住"的值的词典

Flask 在分发请求之前激活（或推送）程序和请求上下文，请求处理完成后再将其删除。程序上下文被推送后，就可以在线程中使用 current_app 和 g 变量。类似地，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程序上下文或请求上下文，就会导致错误。

下面的Python shell会话演示了程序上下文的使用方法：

```
>>> from hello import app
>>> from flask import current_app
>>> current_app.name
Traceback (most recent call last):
...
RuntimeError: Working outside of application context.
...
>>> app_ctx = app.app_context()
>>> app_ctx.push()
>>> current_app.name
'hello'
>>> app_ctx.pop()
```

没激活程序上下文之前就调用 current_app.name 会导致错误，推送完上下文之后才可以调用。

注意：调用 app.app_context() 可获得一个程序上下文。

请求调度

程序收到客户端发来的请求时，要找到处理该请求的视图函数。

Flask 会在程序的 URL 映射中查找请求的 URL。 URL 映射是 URL 和视图函数之间的对应关系。 Flask 使用 app.route 装饰器或者非装饰器形式的 app.add_url_rule() 生成映射。

在 Python shell 中检查为 hello.py 生成的映射：

```
(flask_env3) $ python
>>> from hello import app
>>> app.url_map
Map([<Rule '/' (OPTIONS, GET, HEAD) -> index>,
 <Rule '/static/<filename>' (OPTIONS, GET, HEAD) -> static>,
 <Rule '/user/<name>' (OPTIONS, GET, HEAD) -> user>])
```

/ 和 /user/<name> 路由在程序中使用 app.route 装饰器定义。 /static/<filename> 是 Flask 添加的特殊路由，用于访问静态文件。

URL 映射中的**HEAD**、**Options**、**GET**是请求方法，由路由进行处理。 Flask 为每个路由都指定了请求方法，这样不同的请求方法发送到相同的 URL 上时，会使用不同的视图函数进行处理。 **HEAD**和**OPTIONS**方法由 Flask 自动处理，在这个程序中， URL 映射中的 3 个路由都使用**GET**方法。

请求钩子

在处理请求之前或之后执行代码有时会很有用。例如，在请求开始时，可能需要创建数据库连接或者认证发起请求的用户。为了避免在每个视图函数中都使用重复的代码， Flask 提供了注册通用函数的功能，注册的函数可在请求被分发到视图函数之前或之后调用。

请求钩子使用修饰器实现。Flask 支持以下 4 种钩子：

before_first_request
: 注册一个函数，在处理第一个请求之前运行。

before_request
: 注册一个函数，在每次请求之前运行。

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。

在请求钩子函数和视图函数之间共享数据一般使用上下文全局变量 `g` 。例如， `before_request` 处理程序可以从数据库中加载已登录用户，并将其保存到 `g.user` 中。随后调用视图函数时，视图函数再使用 `g.user` 获取用户。

响应

Flask 调用视图函数后，会将其返回值作为响应的内容。大多数情况下，响应就是一个简单的字符串，作为 HTML 页面回送给客户端。

HTTP 响应中一个很重要的部分是状态码， Flask 默认设为**200**，这个代码表明请求已经被成功处理。

如果视图函数返回的响应需要使用不同的状态码，那么可以把数字代码作为第二个返回值，添加到响应文本之后。

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

视图函数返回的响应还可接受第三个参数，这是一个由首部（header）组成的字典，可以添加到 HTTP 响应中。

除了元组以外， Flask 视图函数还可以返回 Response 对象。 `make_response` 函数接受1-3个参数（和视图函数的返回值一样），并返回一个 Response 对象。

下例中创建了一个响应对象，然后设置了cookie：

```
from flask import make_response

@app.route('/')
def index():
    response = make_response('<h1>This document carries a cookie!</h1>')
    response.set_cookie('answer', '42')
    return response
```

有一种名为**重定向**的特殊响应类型，经常在 Web 表单中使用。这种响应没有页面文档，只告诉浏览器一个新地址用以加载新页面。

重定向经常使用**302**状态码表示，指向的地址由 `Location` 首部提供。重定向响应可以使用 3 个值形式的返回值生成，也可在 `Response` 对象中设定。 Flask 还提供了 `redirect()` 辅助函数，用于生成这种响应：

```
from flask import redirect

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

还有一种特殊的响应由 `abort` 函数生成，用于处理错误。下例中，如果 URL 中动态参数 `id` 对应的用户不存在，就返回状态码**404**：

```
from flask import abort

@app.route('/user/<id>')
def get_user(id):
    user = load_user(id)
    if not user:
        abort(404)
    return '<h1>Hello, %s</h1>' % user.name
```

abort 不会把控制权交还给调用它的函数，而是抛出异常把控制权交给 Web 服务器。

脚注:

[1] 多线程 Web 服务器会创建一个线程池，再从线程池中选择一个线程用于处理接收到的请求。

日期：2017-07-14 作者：田宇伟