2017-07-14，田宇伟

模板

目录

模板

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。为了提升程序的可维护性，通常把表现逻辑移到**模板**中。

模板是一个包含响应文本的文件，其中包含用占位变量表示的动态部分，其具体值只在请求的上下文中才能知道。使用真实值替换变量，再返回最终得到的响应字符串，这一过程称为**渲染**。为了渲染模板，Flask 使用了一个名为**Jinja2**的强大模板引擎。

Jinja2模板引擎

基本使用

Jinja2 通过Template类创建并渲染模板：

```
from jinja2 import Template

template = Template('Hello {{ name }}!')
print(template.render(name='Xiao Ming'))
```

其背后通过 Environment 实例来存储配置和全局对象，从文件系统或其他位置加载模板：

```
{# templates/hello.html #}
Hello {{ name }}!
```

```
from jinja2 import Environment, PackageLoader


env = Environment(loader=PackageLoader('app', 'templates'))
```

```
template = env.get_template('hello.html')
template.render(name='Xiao Ming')
```

通过 Environment 创建了一个模板环境，模板加载器（loader）会在 templates 文件夹中寻找模板。

Flask中使用模板

默认情况下，Flask 在程序文件夹中的 templates 子文件夹中寻找模板。

形式最简单的 Jinja2 模板就是一个包含响应文本的文件。

首先创建 templates 子文件夹，然后在其中创建 index.html 和 user.html 文件。

```
{# templates/index.html #}
<h1>Hello World!</h1>
```

视图函数 user() 返回的响应中包含一个使用变量表示的动态部分。下例使用模板实现这个响应：

```
{# templates/user.html #}
<h1>Hello, {{ name }}!</h1>
```

渲染模板

接下来为 hello.py 增加模板渲染。

修改程序中的视图函数，以便渲染上面创建的2个模板。

```
# hello.py
from flask import Flask, render_template

# ...

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


@app.route('/user/<name>')
def user(name):
    return render_template('user.html', name=name)
```

Flask 提供的 render_template 函数把 Jinja2 模板引擎集成到了程序中。 render_template 函数的第一个参数是模板的文件名。随后的参数都是键值对，表示模板中变量对应的真实值。

变量和过滤器

在模板中使用的 {{ name }} 结构表示一个变量，它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中获取。

Jinja2 能识别所有类型的变量，甚至是一些复杂的类型，例如列表、字典和对象。

使用变量的一些示例：

```
<p>A value from a dictionary: {{ mydict['key'] }}.</p>
<p>A value from a list: {{ mylist[3] }}.</p>
<p>A value from a list, with a variable index: {{ mylist[myintvar] }}.</p>
<p>A value from an object's method: {{ myobj.somemethod() }}.</p>
```

可以使用**过滤器**修改变量，过滤器名添加在变量名之后，中间使用竖线（ | ）分隔。

```
{# 以首字母大写形式显示变量 name 的值 #}
Hello, {{ name|capitalize }}
```

下标列举了 Jinja2 提供的部分常用过滤器 [1]：

过滤器名	说明
safe	渲染值时不转义
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都转换成大写
trim	把值的首尾空格去掉
striptags	渲染之前把值中所有的HTML标签都删掉

默认情况下，出于安全考虑，Jinja2 会转义所有变量。例如，如果一个变量的值为 `'<h1>Hello</h1>'`，Jinja2 会将其渲染成 `'<h1>Hello</h1>'`，浏览器能显示这个 h1 元素，但不会进行解释。很多情况下需要显示变量中存储的 HTML 代码，这时就可使用 safe 过滤器。

注意：千万别在不可信的值上使用 safe 过滤器，例如用户在表单中输入的文本。可能会引发跨站脚本攻击（XSS）。

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 user %}
    Hello, {{ user }}!
{% else %}
    Hello, Stranger!
{% endif %}
```

for 循环

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

宏

宏类似于 Python 代码中的函数。

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

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

为了重复使用宏，可以将其保存在单独的文件中，然后在需要使用的模板中导入：

```
{% import 'macros.html' as macros %}

<ul>
    {% for comment in comments %}
        {{ macros.render_comment(comment) }}
```

```
    {% endfor %}
</ul>
```

导入使用于Python中类似的 `import` 语句，可以直接把整个模板导入到一个变量（ `import xxx as yyy` ），像上面那样。或者从其中导入特定的宏（ `from xxx import yyy` ）。

```
{% from 'macros.html' import render_comment as r_comment %}
```

include 包含

为了避免重复，需要在多处重复使用的模板代码片段可以写入单独的文件，再包含在所有模板中。

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

渲染时会在 `include` 语句的对应位置添加被包含的模板内容：

```
{% include "header.html" %}
    Body
{% include "footer.html" %}
```

`include` 可以使用 `ignore missing` 标记，如果模板不存在，会直接忽略：

```
{% include "sidebar.html" ignore missing %}
```

模板继承

类似于 Python 代码中的类继承。合理使用模板继承，让模板能重用，能提高工作效率和代码质量。

首先，创建一个名为 `base.html` 的基模板：

```
<html>
    <head>
        {% block head %}
            <title>{% block title %}{% endblock %} - My Application</title>
        {% endblock %}
    </head>
    <body>
        {% block body %}
        {% endblock %}
    </body>
</html>
```

`block` 标签定义的元素可在衍生模板（子模板）中重载，如果子模板没有重载，就用基模板的定义显示默认内容。上面例子中，定义了名为 `head` 、 `title` 和 `body` 的块。 `title` 包含在 `head` 中。

下面是基模板的衍生模板：

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

{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
    <style>
    </style>
{% endblock %}
{% block body %}
    <h1>Hello, World!</h1>
{% endblock %}
```

`extends` 指令声明这个模板衍生自 `base.html` 。在 `extends` 指令之后，基模板中的 3 个块被重新定义，模板引擎会将其插入适当的位置。 **注意：** 新定义的 `head` 块，在基模板中其内容不是空的，所以使用 `super()` 获取原来的内容。

赋值

在代码块中使用 set 标签为变量赋值，并且可以为多个变量赋值：

```
from jinja2 import Template

print(Template("""
{% set a = 1 %}
{% set b, c = range(2) %}
<p>{{ a }} {{ b }} {{ c }}</p>
""").render())
```

使用 Flask-Bootstrap

Bootstrap是非常流行的前端开发框架。

要在程序中集成Bootstrap，需要对模板进行修改，加入 Bootstrap 层叠样式表（CSS）和 JavaScript 文件的引用。但是，更简单的办法是直接使用Flask扩展Flask-Bootstrap。

安装 Flask-Bootstrap

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

Flask 扩展一般都在创建程序实例时初始化。

代码1　初始化 Flask-Bootstrap

```
# hello.py
from flask_bootstrap import Bootstrap

# ...
bootstrap = Bootstrap(app)
```

导入 Bootstrap ，然后把程序实例传入构造方法进行初始化。

初始化 Flask-Bootstrap 之后，就可以在程序中使用一个包含所有 Bootstrap 文件的基模板。这个模板利用 Jinja2 的模板继承机制，让程序扩展一个具有基本页面结构的基模板，其中就有用来引入 Bootstrap 的元素。

代码2　使用 Flask-Bootstrap 的模板

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

{% extends "bootstrap/base.html" %}

{% block title %}Flaskr{% endblock %}

{% block navbar %}
    <div class="navbar navbar-inverse" role="navigation">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle"
                        data-toggle="collapse" data-target=".navbar-collapse">
                        <span class="sr-only">Toggle navigation</span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="/">Flaskr</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a href="/">Home</a></li>
                </ul>
            </div>
        </div>
    </div>
{% endblock %}
```

```
{% block content %}
    <div class="container">
        {% block page_content %}{% endblock %}
    </div>
{% endblock %}
```

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

{% extends "base.html" %}

{% block page_content %}
    <div class="page-header">
        <h1>Hello, world!</h1>
    </div>
{% endblock %}
```

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

{% extends "index.html" %}

{% block page_content %}
    <div class="page-header">
        <h1>Hello, {{ name }}!</h1>
    </div>
{% endblock %}
```

Jinja2 中 的 `extends` 指令从 Flask-Bootstrap 中导入 `bootstrap/base.html` ，从而实现模板继承。Flask-Bootstrap 中的基模板提供了一个网页框架，引入了 Bootstrap 中的所有 CSS 和 JavaScript 文件。

基模板中定义的Block

Flask-Bootstrap 的 `base.html` 模板还定义了很多其他块，都可在衍生模板中使用。

表1 Flask-Bootstrap 基模板中定义的block

block名	外层block	说明
doc		整个HTML文档
html	**doc**	<html>标签中的内容
html_attribs	**doc**	<html>标签的属性
head	**doc**	<head>标签中的内容
title	**head**	<title>标签中的内容
metas	**head**	一组<meta>标签
styles	**head**	层叠样式表定义
body	**doc**	<body>标签中的内容
body_attribs	**body**	<body>标签的属性
navbar	**body**	用户定义的导航条
content	**body**	用户定义的页面内容
scripts	**body**	文档底部的 JavaScript 声明

下面是一些例子：

- 添加自定义CSS文件

```
{% block styles %}
    {{super()}}
    <link rel="stylesheet"
```

```
          href="{{ url_for('.static', filename='mystyle.css') }}">
{% endblock %}
```

- 在Bootstrap JavaScript文件之前自定义加载的JavaScript文件

```
{% block scripts %}
    <script src="{{ url_for('.static', filename='myscripts.js') }}"></script>
    {{ super() }}
{% endblock %}
```

- 给 <html> 标签添加 lang 属性

```
{% block html_attribs %} lang="zh-CN"{% endblock %}
```

自定义错误页面

Flask允许程序使用基于模板的自定义错误页面。最常见的错误代码有两个：

404

客户端请求未知页面或路由时显示

500

有未处理的异常时显示

hello.py 自定义错误页面：

```
@app.errorhandler(404)
def page_not_found(e):
    return render_template('404.html'), 404


@app.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

编写错误处理程序中引用的模板：

```
{# templates/404.html #}

{% extends 'base.html' %}

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

{% block page_content %}
    <div class="page-header">
        <h1>Not Found</h1>
    </div>
{% endblock %}
```

```
{# templates/500.html #}

{% extends 'base.html' %}

{% block title %}
    Flaskr - Internal Server Error
{% endblock %}

{% block page_conent %}
    <div class="page-header">
        <h1>Internal Server Error</h1>
    </div>
{% endblock %}
```

链接

任何具有多个路由的程序都需要可以连接不同页面的链接。

在模板中直接编写URL会对代码中定义的路由产生不必要的依赖关系。如果重新定义路由，模板中的链接可能会失效。

Flask 提供了 `url_for()` 辅助函数，它可以使用程序URL映射中保存的信息生成URL。

`url_for()` 函数最简单的用法是以视图函数名（或者 `app.add_url_route()` 定义路由时使用的端点名）作为参数，返回对应的URL。

例如，在当前版本的 `hello.py` 程序中调用 `url_ for('index')` 得到的结果是 `/` 。调用 `url_for('index', _external=True)` 返回的则是绝对地址，在这个示例中是 `http://localhost:5000/` 。[2]

使用 `url_for()` 生成动态地址时，将动态部分作为关键字参数传入。例如， `url_for('user', name='john', _external=True)` 的返回结果是 `http://localhost:5000/user/john` 。

传入 `url_for()` 的关键字参数不仅限于动态路由中的参数。函数能将任何额外参数添加到查询字符串中。例如， `url_for('index', page=2)` 的返回结果是 `/?page=2` 。

静态文件

默认设置下，Flask在程序根目录中名为 `static` 的子目录中寻找静态文件。如果需要，可在 `static` 文件夹中使用子文件夹存放文件。

URL 映射中有一个 `static` 路由。对静态文件的引用被当成一个特殊的路由，即 `/static/<filename>` 。调用 `url_for('static', filename='css/styles.css', _external=True)` 得到的结果是 `http://localhost:5000/static/css/styles.css` 。

下面的例子展示了如何在程序的基模板中放置 `favicon.ico` 图标。这个图标会显示在浏览器的地址栏中。

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

图标的声明会插入 `head` 块的末尾。注意如何使用 `super()` 保留基模板中定义的块的原始内容。

脚注:

[1] 完整的过滤器列表可在 Jinja2 文档（ http://jinja.pocoo.org/docs/templates/#builtin-filters ）中查看。

[2] 生成连接程序内不同路由的链接时，使用相对地址就足够了。如果要生成在浏览器之外使用的链接，则必须使用绝对地址，例如在电子邮件中发送的链接。