

模板引擎

一、蓝图使用

- 说明：

当大量视图函数放在一个文件中，很明显是不合适的。最好的时解决方案是根据需要将相关的视图函数放在单独的文件中。蓝图就是为了解决这个问题而出现的。

- 安装：pip install blueprint
- 使用：

```
# 导入蓝图
from flask import Blueprint

# 创建对象，可以统一指定路由前缀
user = Blueprint('user', __name__, url_prefix='/user')

# 添加视图函数
@user.route('/login/')
def login():
    return '欢迎登录'

@user.route('/register/')
def register():
    return '欢迎注册'

# 注册蓝图(蓝图不注册不能使用)，也可以指定路由前缀(优先级更高)
from user import user
app.register_blueprint(user, url_prefix='/user')
```

二、模板引擎

模板文件就是按照特定规则书写的负责展示效果的HTML文件；模板引擎就是提供特定替换和解析的工具。

- Jinja2：在flask中使用的是Jinja2模板引擎，它是由flask核心开发人员开发的。
- 目录结构

```
project/
  manage.py      # 启动控制代码
  templates/     # 模板文件目录
```

- 设置模板自动加载

```
pp.config['TEMPLATES_AUTO_RELOAD'] = True
```

1.模板渲染

- 导入from flask import render_template,render_template_string
- 在 templates 目录下创建一个模板文件 index.html
- 在视图函数中渲染模板 render_templete('index.html')
- 渲染模板字符串 render_template_string('<h1>Hello World!</h1>')

```
#index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{{title}}</title>
</head>
<body>
<h3> 我是{{name}}</h3>
</body>
</html>

#view.py
@bbs.route('/')
def index():
    # return render_template('index.html',title="首页",name='jinja2')
    return render_template('index.html',**{'title':"首页",'name':'jinja2'})
```

2.变量

变量在模板中的表示为：{{ 变量名 }}，变量名就是render_template中提供的参数。变量可以基本类型中的数值、字符串、布尔，也可以是字典、对象、列表等。jinja2提供了点号或[]来访问复杂数据结构。

- 在渲染模板时需要将解析的变量分配过去，特殊的除外(g)
- 注释写在 {# #} 中
- 渲染模板字符串的方式与渲染模板文件相同
- 注意：
 - 变量就是视图函数传递给模板的数据
 - 变量要遵循标识符的命名规则
 - 如果模板中的变量不存在 则插入空字符串（什么都没有）

```
@bbs.route('/')
```

```
def index():
    dog = Dog()
    print(dog)
    # return render_template('index.html',title="首页",name='jinja2')
    return render_template('index.html',**{'title':"首页",'name':'jinja2',
                                           'a':5,'l1':[1,2,3,4,5],'b':
{'name':'tom',
                                           'age':20},'dog':dog})

#index.html
<!--列表-->
{{ l1.0 }}-- {{ l1.1}}---{{l1[2]}}
<p>
    <!--字典-->
    {{ b.name}} -- {{ b['name']}}
</p>
<p>
    <!--对象-->
    {{dog.name}} -- {{dog['age']}}
</p>
```

3 过滤器和测试

- 过滤器：变量可以通过 **过滤器** 修改。过滤器与变量用管道符号（`|`）分割，并且也可以用圆括号传递可选参数。多个过滤器可以链式调用，前一个过滤器的输出会被作为后一个过滤器的输入。
 - 使用：`{{ name | upper }}`，转换为全大写输出
 - 常用过滤器：

| 过滤器 | 说明 |
|------------|-------------------|
| upper | 全大写 |
| lower | 全小写 |
| title | 每个单词首字母大写 |
| capitalize | 首字母大写 |
| trim | 去掉两边的空白 |
| striptags | 过滤掉HTML标签 |
| safe | 渲染时不转义(默认会转义所有内容) |

- 在模板文件中，动态开启关闭转义

```
{% autoescape False %}
<div>Hello {{ name }}</div>
{% endautoescape %}
```

- 测试

测试可以用于对照普通表达式测试一个变量。要测试一个变量或表达式，你要在变量后加上一个 `is` 以及测试的名称。例如，要得出一个值是否定义过，你可以用 `name is defined`，这会根据 `name` 是否定义返回 `true` 或 `false`。

| 测试 | 说明 | 测试 | 说明 |
|---------------------------------|-----------------------|--------------------------|----------|
| <code>defined</code> | 变量是否定义过 | <code>iterable</code> | 是否可迭代 |
| <code>callable</code> | 对象是否可调用 | <code>upper/lower</code> | 是否大写/小写 |
| <code>escaped</code> | 变量是否转义 | <code>even/odd</code> | 是否是偶数/奇数 |
| <code>none/string/number</code> | 是否是none/string/number | <code>divisibleby</code> | 变量能不能用除法 |

```
{% if num is defined %}
{{ num }}
{% endif %}
```

4.表达式

Jinja 中到处都允许使用基本表达式。表达式中可以使用的类型：字符串、数值、布尔值、列表、字典、元组。

可以进行运算：

- 算数运算符：`+`、`-`、`*`、`/`、`//`、`**`
- 关系运算：`==`、`!=`、`>`、`>=`、`<`、`<=`
- 逻辑运算：`and`、`or`、`not`
- 其它运算：`in`、`is`、`if else`表达式

5.流程控制

- `if`

```
{% if value %}
    . . . .
{% endif %}

{% if value %}
    . . .
{% else %}
    . . . .
{% endif %}
```

```
{% if value %}
    . . . .
{% elif value %}
    . . .
{% else %}
    . . . .
{% endif %}
```

- for

```
{% for v in value %}
    ...
{% else %} #可选
    ...
{% endfor %}
```

loop对象说明

| 属性 | 说明 |
|---------------|-------------------|
| loop.index | 当前循环迭代的次数（从1开始计数） |
| loop.index0 | 从0开始计数的循环迭代次数 |
| loop.revindex | 到循环结束为止需要迭代的次数 |
| loop.first | 是否是第一次迭代 |
| loop.last | 是否是最后一次迭代 |
| loop.length | 序列中项目的数量 |
| loop.cycle | 在一串序列间周期取值的辅助函数 |

- range函数

返回一个等差数列，包头不包尾

```
{% for i in range(10) %}
    <li>{{ i }}</li>
{% endfor %}
```

6.文件包含

- 说明：可以避免大量的重复书写，包含相当于将被包含的内容粘贴过来。
- 使用： `{% include 'include2.html' %}`

7.宏的使用

- 定义宏：`{% macro 宏名(参数) %}宏内容{% endmacro %}`
- 调用宏：`{{ 宏名(参数) }}`
- 导入宏：`{% from '宏所在文件' import 宏名 %}`
- 说明：宏采用了类似于python中的函数进行定义和调用，可以减少代码的重复书写。

8.模板继承

- 说明：一个网站的多个页面，具有相同的结构，只有少许的差别，可以通过继承减少重复书写。
- 使用：
 - 需要先定义一个基础模板，专门用来被其他模板继承的，将其中可能需要修改的地方使用block起名
 - 子模板继承基础模板需要使用：`{% extends '基础模板' %}`
 - 在子模板中可以根据block名字对父级模板中的内容进行修改、删除等操作
 - 若想保留父模板中的内容，可以通过：`{{ super() }}`

9.flask-bootstrap

- 安装：`pip install flask-bootstrap`
- 使用

```
from flask_bootstrap import Bootstrap
app = Flask(__name__)
bootstrap = Bootstrap(app) #实例化Bootstrap
```

- bootstrap中base.html模块说明:

| 块名 | 说明 |
|-------------|--------------------|
| doc | 整个html文档 |
| html_attrbs | html标签属性 |
| html | html标签中的内容 |
| head | head标签中的内容 |
| title | title标签中的内容 |
| metas | 一组meta标签 |
| styles | 层叠样式表定义 |
| body_attrbs | body标签的属性 |
| body | body标签中的内容 |
| navbar | 用户定义的导航条 |
| content | 用户定义的页面内容 |
| scripts | 文档底部的JavaScript 声明 |

- 定制base.html

```
{% extends 'bootstrap/base.html' %}
{% block navbar %}
<nav class="navbar navbar-inverse" style="border-radius: 0;">
  <div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-
expanded="false">
        <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="#">Brand</a>
    </div>

    <!-- Collect the nav links, forms, and other content for toggling -->
    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-
1">
      <ul class="nav navbar-nav">
```

```

        <li class="active"><a href="#">首页 <span class="sr-only">(current)
</span></a></li>
        <li><a href="#">发表博客</a></li>
    </ul>

    <ul class="nav navbar-nav navbar-right">
        <form class="navbar-form navbar-left">
            <div class="form-group">
                <input type="text" class="form-control" placeholder="Search">
            </div>
            <button type="submit" class="btn btn-default">搜索</button>
        </form>
        <li><a href="#">登录</a></li>
        <li><a href="#">注册</a></li>
        <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-haspopup="true" aria-expanded="false">个人中心 <span
class="caret"></span></a>
            <ul class="dropdown-menu">
                <li><a href="#">修改用户名</a></li>
                <li><a href="#">修改密码</a></li>
                <li><a href="#">上传头像</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="#">退出登录</a></li>
            </ul>
        </li>
    </ul>
</div><!-- /.navbar-collapse -->
</div><!-- /.container-fluid -->
</nav>
{% endblock %}
{% block content %}
<div class="container">
    {% block pagecontent %}
        <h2>这是我们自己定义的base 以后所有的子模板都继承我就好</h2>
    {% endblock %}
</div>
{% endblock %}

```

- 定制错误模板


```
# 视图函数
@app.errorhandler(404)
def page_not_found(err):
    return render_template('error.html',title='404
PAGE_NOT_FOUND',info=err,con='页面被外星人抓走了')

@app.errorhandler(500)
def server_error(err):
    return render_template('error.html',title='500
SERVER_ERROR',info=err,con='您的访问太热情了 请稍候在尝试访问')
```

```
#error.html
{% extends 'common/boot_base.html' %}
{% block title %}
    {{ title }}
{% endblock %}
{% block pagecontent %}
<div class="alert alert-danger alert-dismissible" role="alert">
    <button type="button" class="close" data-dismiss="alert" aria-
label="Close"><span aria-hidden="true">&times;</span></button>
    <strong>{{ con }}</strong>{{ info }}
</div>
{% endblock %}
{# 根据404和500对应显示出友好的图片的展示 #}
```

10. 静态资源

目录结构

```
project/
  static/
    img/
      abc.jpeg
  templates/
```

- 引用静态资源

```

#_external生成绝对url路径
```

11 全局对象

- request
- session

- g
- url_for
- 自定义变量

```
from flask import current_app

@app.context_processor
def appinfo():
    return dict(appname=current_app.name)
```

- 自定义函数

```
import time

@app.context_processor
def get_current_time():
    def get_time(timeFormat="%b %d, %Y - %H:%M:%S"):
        return time.strftime(timeFormat)
    return dict(current_time=get_time)
```

```
<!doctype html>
<title>Hello Sample</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<h1>Hello World!</h1>
<p>Request URL: {{ request.url }}</p>
<p>User: {{ session.user }}</p>
<p>DB: {{ g.db }}</p>
<p>Current App is: {{ appname }}</p>
<p>Current Time is: {{ current_time() }}</p>
<p>Current Day is: {{ current_time("%Y-%m-%d") }}</p>
```

三、会话控制 COOKIE

在网站中，http请求是无状态的。也就是说即使第一次和服务器连接后并且登录成功后，第二次请求服务器依然不能知道当前请求是哪个用户。cookie的出现就是为了解决这个问题，第一次登录后服务器返回一些数据（cookie）给浏览器，然后浏览器保存在本地，当该用户发送第二次请求的时候，就会把上次请求存储的cookie数据自动的携带给服务器，服务器通过浏览器携带的数据就能判断当前是哪个用户了。cookie存储的数据量有限，不同的浏览器有不同的存储大小，但一般不超过4kb。因此使用cookie只能存储一些小量的数据。

(1) 设置cookie

格式

response.set_cookie(key,value,max_age,expires,path='/')

| 参数 | 说明 |
|---------|-----------------------------------|
| key | 键 |
| value | 值 |
| max_age | 设置过期时间，单位秒，如果没有设置过期时间，默认为浏览器关闭后过期 |
| expires | 设置过期时间，以时间戳的形式设置 |
| path | 主域名，默认是'/'，表示在当前站点可用 |

```
@app.route('/set_cookie/')
def set_cookie():
    res = make_response('设置cookie')
    res.set_cookie('name','zhangsang')
    return res
```

注意：默认存活时间为浏览会话结束（关闭浏览器）

(2) 设置cookie并设置过期时间

```
import time
#设置cookie并设置过期时间
@app.route('/set_cookie_lefttime/')
def set_cookie_lefttime():
    res = make_response('设置cookie并设置过期时间')
    # res.set_cookie('name','zhangsang',max_age=40)
    timeout = time.time()+40
    res.set_cookie('name','zhangsang',expires=timeout)
    return res
```

(3) 获取cookie

```
#获取cookie
@app.route('/get_cookie/')
def get_cookie():
    print(request.cookies)
    return 'cookie的值为{}'.format(request.cookies.get('name'))
```

(4) 删除cookie

```
@app.route('/delete_cookie/')
def delete_cookie():
    res = make_response('删除cookie')
    res.delete_cookie('name')
    # res.set_cookie('name','zhansgan',expires=0) #重新设置cookie 刚开始就死了
    return res
```

四、会话控制 session

session和cookie的作用有点类似，都是为了存储用户相关的信息。不同的是，cookie是存储在本地浏览器，而session存储在服务器。存储在服务器的数据会更加安全，不容易被窃取。但存储在服务器也有一定的弊端，就是会占用服务器的资源，但现在服务器已经发展至今，存储一些session信息还是绰绰有余的。

服务器要区分哪一个用户的请求 会根据cookie携带着的唯一sessionid来进行区分 session将数据存储在服务器端 安全性高于cookie

注意：

sessionid需要根据secret_key 来进行生成（如果没有 则报错）

导入：from flask import session

(1) 设置session

```
@app.route('/set_session/')
def set_session():
    session['name'] = 'zhansgan'
    return '设置session'
```

(2) 设置session并设置过期时间

```
@app.route('/session_lefttime/')
def session_lefttime():
    session.permanent = True #开启session存储持久化
    app.permanent_session_lifetime = timedelta(minutes=1) #存活一分钟
    session['name'] = 'zhansgan'
    return '设置session并设置过期时间'
```

(3) 获取session

```
@app.route('/get_session/')
def get_session():
    return '值为{}'.format(session.get('name'))
```

(4) 删除session

```
@app.route('/delete_session/')
def delete_session():
    # session.pop(key)
    # session.pop('name')
    session.clear()
    return '删除session'
```

五、flask-session

- 说明：是将session数据保存到redis服务器的解决方案。
- 安装：`pip install flask-session`
- 使用：将session存储到redis中

```
from flask import Flask, session
from flask_script import Manager
from flask_session import Session
from redis import Redis

app = Flask(name)
# 配置
# 是否拥有生效，设置为True之后，有效期由下面选项决定
app.config['SESSION_PERMANENT'] = True
# 默认31天，可以是timedelta或int的数据
app.config['PERMANENT_SESSION_LIFETIME'] = 10
# 指定session的保存方式
app.config['SESSION_TYPE'] = 'redis'
# 指定redis连接实例，默认127.0.0.1，6379端口，0数据库
app.config['SESSION_REDIS'] = Redis()

manager = Manager(app)
# 实例化Session对象
Session(app)

@app.route('/')
def index():
    return 'flask-session'

@app.route('/get/')
def get_session():
    return session.get('name', 'who are you?')

@app.route('/set/')
def set_session():
```

```
    session['name'] = 'ergou'  
    return 'session已设置'  
  
if __name__ == '__main__':  
    manager.run()
```