请求勾子

在客户端和服务器交互的过程中，有些准备工作或扫尾工作需要处理，比如：

- 在请求开始时，建立数据库连接；
- 在请求开始时，根据需求进行权限校验；
- 在请求结束时，指定数据的交互格式；

为了让每个视图函数避免编写重复功能的代码，Flask提供了通用设施的功能，即请求钩子。

请求钩子是通过装饰器的形式实现，Flask支持如下四种请求钩子：

- before_first_request
  - 在处理第一个请求前执行
- before_request
  - 在每次请求前执行
  - 如果在某修饰的函数中返回了一个响应，视图函数将不再被调用
- after_request
  - 如果没有抛出错误，在每次请求后执行
  - 接受一个参数：视图函数作出的响应
  - 在此函数中可以对响应值在返回之前做最后一步修改处理
  - 需要将参数中的响应在此参数中进行返回
- teardown_request：
  - 在每次请求后执行
  - 接受一个参数：错误信息，如果有相关错误抛出

代码

```
from flask import Flask
from settings.dev import DevConfig

app = Flask(__name__)
# 项目配置
app.config.from_object(DevConfig)

@app.before_first_request
def before_first_request():
    print("----before_first_request----")
    print("系统初始化的时候,执行这个钩子方法")
    print("会在接收到第一个客户端请求时,执行这里的代码")

@app.before_request
def before_request():
    print("----before_request----")
    print("每一次接收到客户端请求时,执行这个钩子方法")
    print("一般可以用来判断权限,或者转换路由参数或者预处理客户端请求的数据")
```

```python
@app.after_request
def after_request(response):
    print("----after_request----")
    print("在处理请求以后,执行这个钩子方法")
    print("一般可以用于记录会员/管理员的操作历史,浏览历史,清理收尾的工作")
    response.headers["Content-Type"] = "application/json"
    # 必须返回response参数
    return response


@app.teardown_request
def teardown_request(exc):
    print("----teardown_request----")
    print("在每一次请求以后,执行这个钩子方法,如果有异常错误,则会传递错误异常对象到当前方法的参数中")
    print(exc)

@app.route("/")
def index():
    print("----视图函数----")
    print("视图函数被运行了")
    return "视图函数被运行了<br>"

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=80)
```

- 在第1次请求时的打印:

```
----before_first_request----
系统初始化的时候,执行这个钩子方法
会在接收到第一个客户端请求时,执行这里的代码
----before_request----
每一次接收到客户端请求时,执行这个钩子方法
一般可以用来判断权限,或者转换路由参数或者预处理客户端请求的数据
----视图函数----
视图函数被运行了
----after_request----
在处理请求以后,执行这个钩子方法
一般可以用于记录会员/管理员的操作历史,浏览历史,清理收尾的工作
----teardown_request----
在每一次请求以后,执行这个钩子方法,如果有异常错误,则会传递错误异常对象到当前方法的参数中
None
```

- 在第2次请求时的打印:

```
----before_request----
127.0.0.1 - - [08/Apr/2019 09:23:53] "GET / HTTP/1.1" 200 -
每一次接收到客户端请求时,执行这个钩子方法
一般可以用来判断权限,或者转换路由参数或者预处理客户端请求的数据
----视图函数----
视图函数被运行了
----after_request----
在处理请求以后,执行这个钩子方法
一般可以用于记录会员/管理员的操作历史,浏览历史,清理收尾的工作
----teardown_request----
在每一次请求以后,执行这个钩子方法,如果有异常错误,则会传递错误异常对象到当前方法的参数中
None
```

异常捕获

主动抛出HTTP异常

- abort 方法
  - 抛出一个给定状态代码的 HTTPException 或者 指定响应，例如想要用一个页面未找到异常来终止请求，你可以调用 abort(404)。
- 参数:
  - code – HTTP的错误状态码

```
# abort(404)
abort(500)
```

> 抛出状态码的话，只能抛出 HTTP 协议的错误状态码

捕获错误

- errorhandler 装饰器
  - 注册一个错误处理程序，当程序抛出指定错误状态码的时候，就会调用该装饰器所装饰的方法
- 参数:
  - code_or_exception – HTTP的错误状态码或指定异常
- 例如统一处理状态码为500的错误给用户友好的提示:

```
@app.errorhandler(500)
def internal_server_error(e):
    return '服务器搬家了'
```

- 捕获指定异常

```
@app.errorhandler(ZeroDivisionError)
def zero_division_error(e):
    return '除数不能为0'
```

```
from flask import Flask,abort
from settings.dev import DevConfig

app = Flask(__name__)
# 项目配置
app.config.from_object(DevConfig)

# 异常处理
@app.errorhandler(500)
def internal_server_error(e):
    return '<h1>服务器搬家了</h1>'

@app.errorhandler(Exception)
def zero_division_error(e):
    return '除数不能为0'

@app.route("/")
def index():
    print("视图函数被运行了")
    raise Exception("异常")
    # abort(404)
    # abort(500)
    return "视图函数被运行了<br>"

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=80)
```

上下文

上下文：即语境，语意，在程序中可以理解为在代码执行到某一时刻时，根据之前代码所做的操作以及下文即将要执行的逻辑，可以决定在当前时刻下可以使用到的变量，或者可以完成的事情。

Flask中有两种上下文，请求上下文(request context)和应用上下文(application context)。

Flask中上下文对象：相当于一个容器，保存了 Flask 程序运行过程中的一些信息。

1. *application* 指的就是当你调用`app = Flask(__name__)`创建的这个对象`app`；
2. *request* 指的是每次`http`请求发生时，`WSGI server`(比如gunicorn)调用`Flask.__call__()`之后，在`Flask`对象内部创建的`Request`对象；
3. *application* 表示用于响应WSGI请求的应用本身，*request* 表示每次http请求；
4. *application*的生命周期大于*request*，一个*application*存活期间，可能发生多次http请求，所以，也就会有多个*request*

请求上下文(request context)

思考：在视图函数中，如何取到当前请求的相关数据？比如：请求地址，请求方式，cookie等等

在 flask 中，可以直接在视图函数中使用 **request** 这个对象进行获取相关数据，而 **request** 就是请求上下文的对象，保存了当前本次请求的相关数据，请求上下文对象有：request、session

- request
  - 封装了HTTP请求的内容，针对的是http请求。举例：user = request.args.get('user')，获取的是get请求的参数。
- session
  - 用来记录请求会话中的信息，针对的是用户信息。举例：session['name'] = user.id，可以记录用户信息。还可以通过session.get('name')获取用户信息。

应用上下文(application context)

它的字面意思是 应用上下文，但它不是一直存在的，它只是request context 中的一个对 app 的代理(人)，所谓 local proxy。它的作用主要是帮助 request 获取当前的应用，它是伴 request 而生，随 request 而灭的。

应用上下文对象有：current_app，g

current_app

应用程序上下文,用于存储应用程序中的变量，可以通过current_app.name打印当前app的名称，也可以在current_app中存储一些变量，例如：

- 应用的启动脚本是哪个文件，启动时指定了哪些参数
- 加载了哪些配置文件，导入了哪些配置
- 连接了哪个数据库
- 有哪些可以调用的工具类、常量
- 当前flask应用在哪个机器上，哪个IP上运行，内存多大

```
current_app.name
current_app.test_value='value'
```

g变量

g 作为 flask 程序全局的一个临时变量,充当者中间媒介的作用,我们可以通过它传递一些数据，g 保存的是当前请求的全局变量，不同的请求会有不同的全局变量，通过不同的thread id区别

```
g.name='abc'
```

> 注意：不同的请求，会有不同的全局变量

两者区别：

- 请求上下文：保存了客户端和服务器交互的数据
- 应用上下文：flask 应用程序运行过程中，保存的一些配置信息，比如程序名、数据库连接、应用信息等

Flask-Script 扩展

安装命令:

```
pip install flask-script
```

集成 Flask-Script到flask应用中

```
from flask import Flask
from flask_script import Manager

app = Flask(__name__)

# 把 Manager 类和应用程序实例进行关联
manager = Manager(app)

@app.route('/')
def index():
    return 'hello world'

if __name__ == "__main__":
    manager.run()
```

Flask-Script 还可以为当前应用程序添加脚本命令

```
class hello(Command):
    "prints hello world"
    def run(self):
        print("hello world")

manager.add_command('hello', hello())
```

Jinja2模板引擎

Flask内置的模板语言，它的设计思想来源于 Django 的模板引擎，并扩展了其语法和一系列强大的功能。

渲染模版函数

- Flask提供的 **render_template** 函数封装了该模板引擎
- **render_template** 函数的第一个参数是模板的文件名，后面的参数都是键值对，表示模板中变量对应的真实值。

模板基本使用

1. 在视图函数中设置渲染模板

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

2. 在项目下创建 `templates` 文件夹，用于存放所有的模板文件，并在目录下创建一个模板html文件 `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
我的模板html内容
</body>
</html>
```

{{}} 来表示变量名，这种 {{}} 语法叫做**变量代码块**

```
<h1>{{ post.title }}</h1>
```

Jinja2 模版中的变量代码块可以是任意 Python 类型或者对象，只要它能够被 Python 的 str() 方法转换为一个字符串就可以，比如，可以通过下面的方式显示一个字典或者列表中的某个元素:

```
{{your_dict['key']}}
{{your_list[0]}}
```

用 {**%%**} 定义的**控制代码块**，可以实现一些语言层次的功能，比如循环或者if语句

```
{% if user %}
    {{ user }}
{% else %}
    hello!
<ul>
    {% for index in indexs %}
    <li> {{ index }} </li>
    {% endfor %}
</ul>
```

使用 {# #} 进行注释，注释的内容不会在html中被渲染出来

```
{# {{ name }} #}
```

模板中特有的变量和函数

你可以在自己的模板中访问一些 Flask 默认内置的函数和对象

config

你可以从模板中直接访问Flask当前的config对象:

```
{{config.SQLALCHEMY_DATABASE_URI}}
sqlite:///database.db
```

request

就是flask中代表当前请求的request对象：

```
{{request.url}}
http://127.0.0.1
```

session

为Flask的session对象

```
{{session.new}}
True
```

g变量

在视图函数中设置g变量的 name 属性的值，然后在模板中直接可以取出

```
{{ g.name }}
```

url_for()

url_for会根据传入的路由器函数名,返回该路由对应的URL,在模板中始终使用url_for()就可以安全的修改路由绑定的URL,则不比担心模板中渲染出错的链接:

```
{{url_for('home')}}
```

如果我们定义的路由URL是带有参数的,则可以把它们作为关键字参数传入url_for(),Flask会把他们填充进最终生成的URL中:

```
{{ url_for('post', post_id=1)}}
/post/1
```

流程控制

主要包含两个:

```
- if/else if /else / endif
- for / endfor
```

if语句

Jinja2 语法中的if语句跟 Python 中的 if 语句相似,后面的布尔值或返回布尔值的表达式将决定代码中的哪个流程会被执行:

```
{%if user.is_logged_in() %}
    <a href='/logout'>Logout</a>
{% else %}
    <a href='/login'>Login</a>
{% endif %}
```

过滤器可以被用在 if 语句中:

```
{% if comments | length > 0 %}
    There are {{ comments | length }} comments
{% else %}
    There are no comments
{% endif %}
```

循环语句

- 我们可以在 Jinja2 中使用循环来迭代任何列表或者生成器函数

```
{% for post in posts %}
    <div>
        <h1>{{ post.title }}</h1>
        <p>{{ post.text | safe }}</p>
    </div>
{% endfor %}
```

- 循环和if语句可以组合使用，以模拟 Python 循环中的 continue 功能，下面这个循环将只会渲染post.text不为None的那些post:

```
{% for post in posts if post.text %}
    <div>
        <h1>{{ post.title }}</h1>
        <p>{{ post.text | safe }}</p>
    </div>
{% endfor %}
```

- 在一个 for 循环块中你可以访问这些特殊的变量:

变量	描述
loop.index	当前循环迭代的次数（从 1 开始）
loop.index0	当前循环迭代的次数（从 0 开始）
loop.revindex	到循环结束需要迭代的次数（从 1 开始）
loop.revindex0	到循环结束需要迭代的次数（从 0 开始）
loop.first	如果是第一次迭代，为 True 。
loop.last	如果是最后一次迭代，为 True 。
loop.length	序列中的项目数。
loop.cycle	在一串序列间期取值的辅助函数。见下面示例程序。

- 在循环内部,你可以使用一个叫做loop的特殊变量来获得关于for循环的一些信息
    - 比如：要是我们想知道当前被迭代的元素序号，并模拟Python中的enumerate函数做的事情，则可以使用loop变量的index属性,例如:

```
{% for post in posts%}
{{loop.index}}, {{post.title}}
{% endfor %}
```

- 会输出这样的结果

```
1, Post title
2, Second Post
```

- cycle函数会在每次循环的时候,返回其参数中的下一个元素,可以拿上面的例子来说明:

```
{% for post in posts%}
{{loop.cycle('odd','even')}} {{post.title}}
{% endfor %}
```

- 会输出这样的结果：

```
odd Post Title
even Second Post
```

过滤器

过滤器的本质就是函数。有时候我们不仅仅只是需要输出变量的值，我们还需要修改变量的显示，甚至格式化、运算等等，而在模板中是不能直接调用 Python 中的某些方法，那么这就用到了过滤器。

使用方式：

- 过滤器的使用方式为：变量名 | 过滤器。

```
{{variable | filter_name(*args)}}
```

- 如果没有任何参数传给过滤器,则可以把括号省略掉

```
{{variable | filter_name }}
```

- 如：``，这个过滤器的作用：把变量variable 的值的首字母转换为大写，其他字母转换为小写

在 jinja2 中，过滤器是可以支持链式调用的，示例如下：

```
{{ "hello world" | reverse | upper }}
```

常见的内建过滤器

字符串操作

- safe：禁用转义

```
<p>{{ '<em>hello</em>' | safe }}</p>
```

- capitalize：把变量值的首字母转成大写，其余字母转小写

```
<p>{{ 'hello' | capitalize }}</p>
```

- lower：把值转成小写

```
<p>{{ 'HELLO' | lower }}</p>
```

- upper：把值转成大写

```
<p>{{ 'hello' | upper }}</p>
```

- title：把值中的每个单词的首字母都转成大写

```
<p>{{ 'hello' | title }}</p>
```

- reverse：字符串反转

```
<p>{{ 'olleh' | reverse }}</p>
```

- format：格式化输出

```
<p>{{ '%s is %d' | format('name',17) }}</p>
```

- striptags：渲染之前把值中所有的HTML标签都删掉

```
<p>{{ '<em>hello</em>' | striptags }}</p>
```

- truncate: 字符串截断

```
<p>{{ 'hello every one' | truncate(9)}}</p>
```

列表操作

- first：取第一个元素

```
<p>{{ [1,2,3,4,5,6] | first }}</p>
```

- last：取最后一个元素

```
<p>{{ [1,2,3,4,5,6] | last }}</p>
```

- length：获取列表长度

```
<p>{{ [1,2,3,4,5,6] | length }}</p>
```

- sum：列表求和

```
<p>{{ [1,2,3,4,5,6] | sum }}</p>
```

- sort：列表排序

```
<p>{{ [6,2,3,1,5,4] | sort }}</p>
```

语句块过滤

```
{% filter upper %}
    #一大堆文字#
{% endfilter %}
```

自定义过滤器

过滤器的本质是函数。当模板内置的过滤器不能满足需求，可以自定义过滤器。自定义过滤器有两种实现方式：

- 一种是通过Flask应用对象的 **add_template_filter** 方法
- 通过装饰器来实现自定义过滤器

重要：自定义的过滤器名称如果和内置的过滤器重名，会覆盖内置的过滤器。

需求：添加列表反转的过滤器

方式一

通过调用应用程序实例的 add_template_filter 方法实现自定义过滤器。该方法第一个参数是函数名，第二个参数是自定义的过滤器名称：

```
def do_listreverse(li):
    # 通过原列表创建一个新列表
    temp_li = list(li)
    # 将新列表进行返转
    temp_li.reverse()
    return temp_li

app.add_template_filter(do_listreverse,'lireverse')
```

方式二

用装饰器来实现自定义过滤器。装饰器传入的参数是自定义的过滤器名称。

```
@app.template_filter('lireverse')
def do_listreverse(li):
    # 通过原列表创建一个新列表
    temp_li = list(li)
    # 将新列表进行返转
    temp_li.reverse()
    return temp_li
```

- 在 html 中使用该自定义过滤器

```
<br/> my_array 原内容: {{ my_array }}
<br/> my_array 反转: {{ my_array | lireverse }}
```

- 运行结果

```
my_array 原内容: [3, 4, 2, 1, 7, 9]
my_array 反转: [9, 7, 1, 2, 4, 3]
```

模板继承

在模板中，可能会遇到以下情况：

- 多个模板具有完全相同的顶部和底部内容
- 多个模板中具有相同的模板代码内容，但是内容中部分值不一样
- 多个模板中具有完全相同的 html 代码块内容

像遇到这种情况，可以使用 JinJa2 模板中的 **继承** 来进行实现

模板继承是为了重用模板中的公共内容。一般Web开发中，继承主要使用在网站的顶部菜单、底部。这些内容可以定义在父模板中，子模板直接继承，而不需要重复书写。

- 标签定义的内容

```
{% block top %} {% endblock %}
```

- 相当于在父模板中挖个坑，当子模板继承父模板时，可以进行填充。
- 子模板使用 extends 指令声明这个模板继承自哪个模板
- 父模板中定义的块在子模板中被重新定义，在子模板中调用父模板的内容可以使用super()

父模板代码：

base.html

```
{% block top %}
  顶部菜单
{% endblock top %}

{% block content %}
{% endblock content %}

{% block bottom %}
  底部
{% endblock bottom %}
```

子模板代码：

- extends指令声明这个模板继承自哪

```
{% extends 'base.html' %}
{% block content %}
 需要填充的内容
{% endblock content %}
```

模板继承使用时注意点：

1. 不支持多继承
2. 为了便于阅读，在子模板中使用extends时，尽量写在模板的第一行。
3. 不能在一个模板文件中定义多个相同名字的block标签。
4. 当在页面中使用多个block标签时，建议给结束标签起个名字，当多个block嵌套时，阅读性更好。

在 Flask 项目中解决 CSRF 攻击

在 Flask 中， Flask-wtf 扩展有一套完善的 csrf 防护体系，对于我们开发者来说，使用起来非常简单

1. 设置应用程序的 secret_key，用于加密生成的 csrf_token 的值

```
# session加密的时候已经配置过了.如果没有在配置项中设置,则如下:
app.secret_key = "#此处可以写随机字符串#"
```

1. 导入 flask_wtf.csrf 中的 CSRFProtect 类，进行初始化，并在初始化的时候关联 app

```
from flask.ext.wtf import CSRFProtect
CSRFProtect(app)
```

1. 在表单中使用 CSRF 令牌:

```
<form method="post" action="/">
    <input type="hidden" name="csrf_token" value="{{ csrf_token() }}" />
</form>
```