Flask视图：视图函数，类视图，蓝图使用方法整理


xiaogp 关注 IP属地: 江苏

0.095 2021.01.06 15:30:53 字数 1,370 阅读 3,203

摘要： Flask ， 视图 ， 视图函数 ， 类视图 ， 方法视图 ， 装饰器 ， 蓝图

视图函数

在Flask中 路由 是指用户请求的 URL 与 视图函数 之间的 映射 ，处理URL和函数之间关系的程序称为路由。Flask根据HTTP请求的URL在路由表中匹配预定义的URL找到对应的视图函数。将视图函数的执行结果返回给服务器。

app.route的使用

Flask中默认使用 @app.route 装饰器将视图函数和URL绑定，装饰器是一种接受函数的函数，返回新的函数。

```
from flask import Flask

app = Flask(__name__)


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


if __name__ == '__main__':
    print(app.url_map)
    app.run(host="0.0.0.0", port=5000)
```

使用装饰器将视图函数page和url '/'关系绑定带 app.url_map 属性上，打印app.url_map的结果如下，有两条url规则，分别是根目录下的URL规则和static目录下的URL规则

```
Map([<Rule '/' (HEAD, GET, OPTIONS) -> page>,
 <Rule '/static/<filename>' (HEAD, GET, OPTIONS) -> static>])
```

可以给装饰器增加 endpoint 参数给 url命名 ，一旦使用了endpoint参数 url_for 反转就不能使用视图函数名了而要使用定义的url名。

```
@app.route('/', endpoint='index')
def page():
    return "hello world"


@app.route('/page')
def page2():
    print(url_for('index'))
    print(type(url_for('index')))
    return redirect(url_for('index'))
```

url_for('index')的输出是字符串格式url的内容"/"

add_url_rule的使用

也可以不使用装饰器，使用 `add_url_rule` 将视图函数和url绑定，装饰器 `@app.route` 实际是调用的 `add_url_rule` 方法

```
def page3():
    return "hollow page3"


app.add_url_rule('/page3', endpoint='index3', view_func=page3)
```

类视图

视图函数也可以结合类来实现，类视图的好处是支持 继承 ，可以将共性的东西放到父类中，类视图需要使用 `app.add_url_rule()` 来进行注册，类视图分为 标准类视图 和 基于调度方法的类视图

标准类视图

标准类视图有标准的写法

- 父类继承 `flask.views.View`
- 子类调用 `dispatch_request` 进行返回，完成业务逻辑
- 子类需要使用 `app.add_url_rule` 进行注册，其中view_func参数不能直接传入子类的名称，需要使用 `as_view` 做类方法转换
- 如果同时也指定了 `endpoint` ，endpoint会覆盖as_view指定的视图名，先endpoint名后as_view名

使用类视图，在父类中定义一个属性，在子类中完成各自的业务逻辑，同时都继承父类中的这一个属性

```
from flask.views import View
from flask import Flask, render_template

app = Flask(__name__)


class Ads(View):
    def __init__(self):
        super().__init__()
        self.context = {
            'ads': '这是需要继承的内容'
        }


class Page1(Ads):
    def dispatch_request(self):
        return render_template('index1.html', **self.context)


class Page2(Ads):
    def dispatch_request(self):
        return render_template('index2.html', **self.context)


class Page3(Ads):
    def dispatch_request(self):
        return render_template('index3.html', **self.context)


app.add_url_rule('/page1', endpoint='page1', view_func=Page1.as_view('page1'))
app.add_url_rule('/page2', endpoint='page2', view_func=Page2.as_view('page2'))
app.add_url_rule('/page3', endpoint='page3', view_func=Page3.as_view('page3'))


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000)
```

分别定义三个子类的模板

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>测试</title>
</head>
<body>
    <p>page1</p>
    <p>{{ ads }}</p>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>测试</title>
</head>
<body>
    <p>page2</p>
    <p>{{ ads }}</p>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>测试</title>
</head>
<body>
    <p>page3</p>
    <p>{{ ads }}</p>
</body>
</html>
```

查看结果，三个url的返回除了三个模板各自的内容外都需要输出父类的ads属性

page1.png

page2.png

page3.png

基于方法判断的类视图

如果同一个视图函数需要根据 不同的请求方式 进行不一样的逻辑处理，需要在视图函数内部进行判断，可以使用 方法类视图 实现，使用类继承 `flask.views.MethodView` ，定义和请求方式 同名的小写方法 来完成了逻辑处理。
编辑一个页面直接访问是输出用户名密码页面，提交表单后是密码正确与否的提示。

```
from flask.views import View, MethodView
from flask import Flask, render_template, request

app = Flask(__name__)


class MyView(MethodView):
    def get(self):
        return render_template('index.html')

    def post(self):
        username = request.form.get('username')
        password = request.form.get('password')
        if username == "gp" and password == "mypassword":
            return '密码正确'
        else:
            return '密码错误'


app.add_url_rule('/', endpoint='login', view_func=MyView.as_view('login'))


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000)
```

在html中定义 `form` 标签action属性关联url名

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
{% macro input(name, type='text', value='') %}
    <input type="{{ type }}" name="{{ name }}" value="{{ value }}">
{% endmacro %}

<form action="/" method="post">
<p>用户名：{{ input('username') }}</p>
<p>密码：{{ input('password', type='password') }}</p>
{{ input('submit', type='submit', value='提交') }}
</form>
</body>
</html>
```

get请求页面.png

post请求页面.png

如果不用方法视图实现需要在普通视图内部调用 `request.method` 判断是否为 `GET` ， `POST` 进行判断

```
from flask import Flask, render_template, request

app = Flask(__name__)


@app.route('/', methods=['GET', 'POST'])
def login():
    if request.method == 'GET':
        return render_template('index.html')
    elif request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')
        if username == "gp" and password == "mypassword":
            return '密码正确'
        else:
            return '密码错误'


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000)
```

Flask装饰器

装饰器的本质是一个Python函数， 接受一个函数 ， 返回一个函数 ，目的是让一个函数获得 其他额外的功能 。

假设一个场景访问新闻详情页又一个函数实现，但是之前必须先登录，登录由另一个函数实现，此时需要将访问新闻函数传递给登录函数返回一个新的函数作为整体的逻辑实现，这个给登录函数增加新功能浏览网页的过程就是装饰器。

```
from flask import Flask, render_template, request

app = Flask(__name__)


def user_login(func):
    def inner():
        print("登录成功")
        func()
    return inner


def news():
    print("浏览网页内容")


new_func = user_login(news)
new_func()
print(new_func.__name__)


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000)
```

控制台输出，new_func()执行了新函数，基础函数user_login执行了新加入的功能，新函数真实的函数名还是inner

```
登录成功
浏览网页内容

```

```
    inner
```

如果使用装饰器魔法符号实现，此时直接调用被装饰的函数即可实现带有新功能的基础函数，函数作为参数传入的过程已经自动实现

```
from flask import Flask, render_template, request

app = Flask(__name__)


def user_login(func):
    def inner():
        print("登录成功")
        func()
    return inner


@user_login
def news():
    print("浏览网页内容")


news()
print(news.__name__)


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000)
```

带有参数额装饰器

在基础函数和要包装的函数上都支持传递参数

```
def user_login(rule: str):
    def wrapper(f):
        print("登录成功", "rule={}".format(rule))
        return f
    return wrapper


@user_login('my_rule')
def news(name: str):
    print("浏览网页内容" + "name={}".format(name))


news("c")
```

```
登录成功 rule=my_rule
浏览网页内容name=c
news
```

查看 `app.route()` 的源码内部也是将视图函数包装，在原函数执行之前调用 `add_url_rule` 绑定url，endpoint和视图函数的关系，再返回原函数实现业务逻辑

```
    def route(self, rule, **options):
        def decorator(f):
            endpoint = options.pop("endpoint", None)
            self.add_url_rule(rule, endpoint, f, **options)
            return f

        return decorator
```

Flask蓝图

蓝图的目的是实现**各个模块的视图函数写在不同的py文件中**，在主视图中导入分路由视图的模块，并注册蓝图对象，**降低各个功能模块的耦合度**，使用 `flask.Blueprint` 定义蓝图，`app.register_blueprint` 注册蓝图。

实现主页，详情页，对比页三个页面，在主页中导入两个其他功能页，先编写两个功能页的蓝图detail.py和compare.py

```
from flask import Blueprint

detail = Blueprint('detail', __name__)


@detail.route('/<string:name>.html')
def company_base_info(name: str):
    return '{}详情'.format(name)
```

```
from flask import Blueprint, request

compare = Blueprint('compare', __name__)


@compare.route('/compare', methods=['GET'])
def compare_info():
    data = request.args.to_dict()
    c1 = data['c1']
    c2 = data['c2']
    return "{} vs {}".format(c1, c2)
```

使用 `app = Blueprint('detail', __name__)` 定义蓝图对象， `detail` 是蓝图名，蓝图名不能重复。再编写主视图main.py，在主视图中注册之前的蓝图，其他视图函数的名字不能和蓝图名一致

```
from flask import Flask
from detail import detail
from compare import compare


app = Flask(__name__)


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


app.register_blueprint(detail)
app.register_blueprint(compare)

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5000)
```

查看效果

index.png

detail.png

compare.png

如果在蓝图的py脚本中调用了 `url_for` ，需要把蓝图的name（就是**name**之前的）也加入作为前缀，如下

```
from flask import Blueprint, render_template, request, redirect, url_for

index = Blueprint('index', __name__)

@index.route('/rank', methods=['GET'])
def rank():
    return render_template('index.html', **locals())


@index.route('/', methods=['GET'])
def home():
    return redirect(url_for('index.rank'))
```