小白学Flask第三天| 今天把视图函数的路由给讲清楚！

原创JAP君 Python进击者
2019-08-17原文

点击蓝色字关注我们！

一个正在努力变强的公众号

本文主要内容：

1. 视图函数的路由规则设置说明
2. 路由提取参数与自定义路由转换器
3. 路由转换器的进阶使用

视图函数的路由规则设置说明

完整代码：

```
# -*- coding: utf-8 -*-

from flask import Flask


app = Flask(__name__)
```

```python
@app.route('/')
def index():
    return "hello flask"


if __name__ == '__main__':
    # 通过url_map可以查看整个flask中的路由信息
    print(app.url_map)
    # 启动flask程序
    app.run(debug=True)
```

首先我们来说说如何查看视图函数的路由：

```python
    # 通过url_map可以查看整个flask中的路由信息
    print(app.url_map)
```

我们可以通过url_map来查看所有的路由信息，我们来看下输出结果：

```
Map([<Rule '/' (OPTIONS, GET, HEAD) -> index>,
 <Rule '/static/<filename>' (OPTIONS, GET, HEAD) -> static>])
 * Serving Flask app "index" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
Map([<Rule '/' (HEAD, OPTIONS, GET) -> index>,
 <Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>])
 * Debugger is active!
 * Debugger PIN: 898-594-454
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

大家可以看到输出了一个Map映射的对象，里面有一个列表，列表里就有着路由的详细信息。这里我给大家详细讲解一下里面的内容

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

这里面有两个规则，首先看第一个，'/'是我们绑定的主路径，它所对应的视图函数是index，大家可以在上面的完整代码中查看。

那么中间那一部分到底是干啥的呢？中间括号中所代表的是当前视图函数所对应的路径的网络请求方法。HTTP里面的请求方式有很多，之前也跟大家谈到过。

在这个index视图函数中，我们看到默认有GET请求方式，那么如果我想要POST请求方式那该怎么做呢？

```
@app.route("/post_only", methods=["POST"])
def post_only():
    return "post"
```

我们再次定义一个函数，大家可以看到装饰器里面多了一个参数，那里我们可以添加我们所想要的网络请求方式。

我们再看一下输出结果：

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

大家可能会对HEAD、OPTIONS有一点疑问，但是大家不用去管它们，它们都是flask自动帮我们补充的。

大家最好把一些常见的请求方式都添加到参数当中，这样可以防止一些错误的产生。

不同的视图函数但是有着相同的装饰器

说完了url_map，接下来给大家看一段代码：

```
# -*- coding: utf-8 -*-

from flask import Flask


app = Flask(__name__)


@app.route("/hello")

def hello1():

    return "hello 1"


@app.route("/hello")

def hello2():

    return "hello 2"


if __name__ == '__main__':

    # 通过url_map可以查看整个flask中的路由信息

    print(app.url_map)

    # 启动flask程序

    app.run(debug=True)
```

大家可以看到我们两个不同的视图函数但是有着相同的装饰器，那这样还能正常打印出url_map吗？

```
 <Rule '/hello' (GET, OPTIONS, HEAD) -> hello2>,
 <Rule '/static/<filename>' (GET, OPTIONS, HEAD) -> static>])
 * Serving Flask app "index" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
Map([<Rule '/hello' (OPTIONS, HEAD, GET) -> hello1>,
 <Rule '/hello' (OPTIONS, HEAD, GET) -> hello2>,
 <Rule '/static/<filename>' (OPTIONS, HEAD, GET) -> static>])
 * Debugger is active!
 * Debugger PIN: 898-594-454
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

可以看到这样是OK的，那么真正运行的时候到底运行哪个呢？

我们访问这个装饰器，可以看到真正运行的是第一个hello

那么当我们将两个视图函数的网络请求方式改成不一样的，还会出现这种情况吗？

```
# -*- coding: utf-8 -*-

from flask import Flask


app = Flask(__name__)


@app.route("/hello", methods=["POST"])
```

```
def hello1():
    return "hello 1"


@app.route("/hello", methods=["GET"])
def hello2():
    return "hello 2"


if __name__ == '__main__':
    # 通过url_map可以查看整个flask中的路由信息
    print(app.url_map)
    # 启动flask程序
    app.run(debug=True)
```

可以看到当我们把请求方式修改时，运行的是hello2。这里大家可能就会知道，当装饰器和请求方式完全相同时，那么执行的是第一个，如果装饰器相同但是请求方式不相同，那么它们将是独立的个体。

装饰器不同，但是视图函数相同

除了上面的情况之外，有没有装饰器不同，但是视图函数相同的情况呢？当然是有滴，看代码：

```
@app.route("/hi1")
@app.route("/hi2")
def hi():
    return "hi nihao"
```

我们来执行一下：

可以看到无论是访问hi1还是hi2都是可以执行这个视图函数的

Flask中的重定向

```
# -*- coding: utf-8 -*-
from flask import Flask, redirect

app = Flask(__name__)
```

```
@app.route("/hello")
def hello1():
    return "hello 1"


@app.route("/login")
def login():
    url = '/hello'
    return redirect(url)


if __name__ == '__main__':
    # 启动flask程序
    app.run(debug=True)
```

上面代码意思就是通过访问login的视图函数跳转到/hello装饰器所对应的视图函数上。

我们可以发现上面的url是写死的，那如果某一天我把hello1视图函数的装饰器修改了，那我岂不是还要一个一个去修改？所以这里还有另外一种方法：

```
# -*- coding: utf-8 -*-
from flask import Flask, redirect, url_for

app = Flask(__name__)
```

```python
@app.route("/hello")
def hello1():
    return "hello 1"


@app.route("/login")
def login():
    url = url_for("hello1")
    return redirect(url)


if __name__ == '__main__':
    # 通过url_map可以查看整个flask中的路由信息
    print(app.url_map)
    # 启动flask程序
    app.run(debug=True)
```

可以看到我们又导入了一个叫url_for的方法，通过把视图函数的名称放进参数当中就可以找到视图函数所对应的url路径。

路由提取参数

转换器有下面几种：

int	接受整数
float	同 *int*，但是接受浮点数
path	和默认的相似，但也接受斜线

有关于路由提取参数，转换器是不得不说的。那么转换器怎么去使用？

```
# -*- coding: utf-8 -*-

from flask import Flask

app = Flask(__name__)


@app.route("/goods/<int:goods_id>")

def goods(goods_id):

     return "goods_id: %s" % goods_id


if __name__ == '__main__':


    # 启动flask程序

    app.run(debug=True)
```

大家可以看到装饰器中所写的<int:goods_id>就是所谓的转换器，我们运行看一看：

成功的截取到了我们路由上的参数123

除了使用自带的转换器外，我们还可以不使用转换器

```
@app.route("/goods/<goods_id>")
def goods(goods_id):

    return "goods_id: %s" % goods_id
```

此时抽取的类型是普通字符串类型（除了 / 字符）。

自定义转换器

虽然flask提供给我们一些转换器，但是在开发中这些往往不能满足我们，所以我们需要自定义一些转换器：

```
# -*- coding: utf-8 -*-
from flask import Flask
from werkzeug.routing import BaseConverter
app = Flask(__name__)


# 1.定义自己的转换器
```

```
class RegexConverte(BaseConverter):
    def __init__(self, url_map, regex):
        # 调用父类的初始化方法
        super(RegexConverte, self).__init__(url_map)
        # 将正则表达式的参数保存在对象的属性中，flask会去使用这个属性来进行路由的正则匹配
        self.regex = regex

# 2. 将自定义的转换器添加到flask的应用中
app.url_map.converters["re"] = RegexConverte

@app.route("/send/<re(r'1[345678]\d{9}'):moblie>")
def send_sms(moblie):

    return "send_sms: %s" % moblie

if __name__ == '__main__':

    # 启动flask程序
    app.run(debug=True)
```

上面代码是提取出路由中的电话号码。

连续打卡送书活动：

Flask系列文章大概会有15-
20篇，如果读者在每次文章发布后进行打卡，**该系列结束后会赠送一本或者多本书籍**。

打卡方式：参与“1元混脸熟”的赞赏小活动，简单点说就是每次文章发布在文末赞赏1元，记住只能是1元。

“1元混脸熟”活动我会把经常赞赏我的朋友拉进铁粉群，群内会有一系列送书活动，当然也可以聊任何东西(赚钱、推广、经验分享）。

该系列文章结束，我会送一直坚持连续打卡的读者朋友一本或者多本书，当然书的价值绝对比你打卡的金额多。

Flask系列文章：

精选留言

暂无...