# Flask简介

Flask是一个轻量级的可定制框架，使用Python语言编写，较其他同类型框架更为灵活、轻便、安全且容易上手。

Flask 可以很好地结合MVC模式进行开发，开发人员分工合作，小型团队在短时间内就可以完成功能丰富的中小型网站或Web服务的实现。

Flask还有很强的定制性，用户可以根据自己的需求来添加相应的功能，在保持核心功能简单的同时实现功能的丰富与扩展，其强大的插件库可以让用户实现个性化的网站定制，开发出功能强大的网站。

## Flask主要特征

Flask是目前十分流行的web框架。

Flask 被称为微框架(microframework)，“微”并不是意味着把整个Web应用放入到一个Python文件，微框架中的“微”是指Flask旨在保持代码简洁且易于扩展。

Flask框架的主要特征是核心构成比较简单，但具有很强的扩展性和兼容性，程序员可以使用Python语言快速实现一个网站或Web服务。

## Flask两个核心函数库

Flask主要包括Werkzeug和Jinja2两个核心函数库，它们分别负责业务处理和安全方面的功能，这些基础函数为web项目开发过程提供了丰富的基础组件。

### Werkzeug

Werkzeug库十分强大，功能比较完善，支持URL路由请求集成，一次可以响应多个用户的访问请求；

支持Cookie和会话管理，通过身份缓存数据建立长久连接关系，并提高用户访问速度；支持交互式Javascript调试，提高用户体验；

可以处理HTTP基本事务，快速响应客户端推送过来的访问请求。

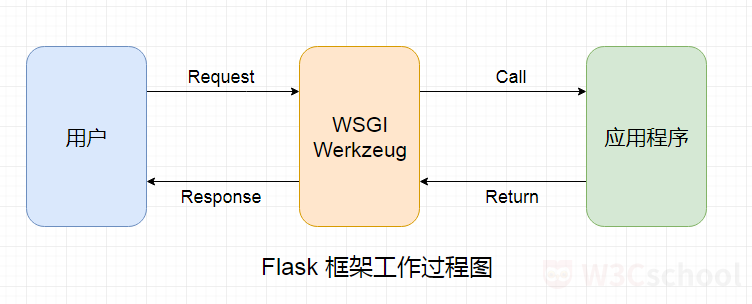
### Jinja2

Jinja2库支持自动HTML转移功能，能够很好控制外部黑客的脚本攻击。系统运行速度很快，页面加载过程会将源码进行编译形成python字节码，从而实现模板的高效运行；

模板继承机制可以对模板内容进行修改和维护，为不同需求的用户提供相应的模板。

## Flask 基本模式

Flask的基本模式是在程序里将一个视图函数分配给一个URL，每当用户访问这个URL时，系统就会执行给该URL分配好的视图函数，获取函数的返回值并将其显示到浏览器上，工作过程见图。



## Flask 目录结构

flask-demo/

├ run.py # 应用启动程序

├ config.py # 环境配置

├ requirements.txt # 列出应用程序依赖的所有Python包

├ tests/ # 测试代码包

│ ├ \_\_init\_\_.py

│ └ test\_\*.py # 测试用例

└ myapp/

├ admin/ # 蓝图目录

├ static/

│ ├ css/ # css文件目录

│ ├ img/ # 图片文件目录

│ └ js/ # js文件目录

├ templates/ # 模板文件目录

├ \_\_init\_\_.py

├ forms.py # 存放所有表单，如果多，将其变为一个包

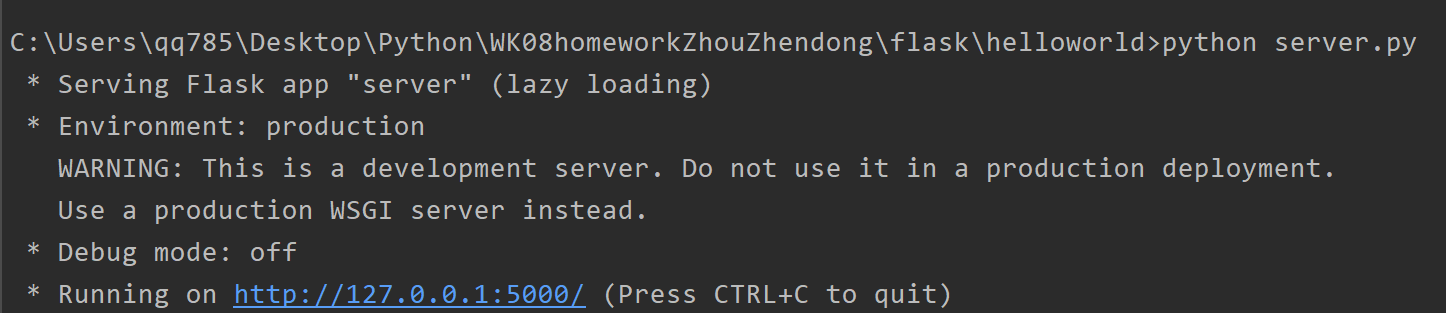
├ models.py # 存放所有数据模型，如果多，将其变为一个包

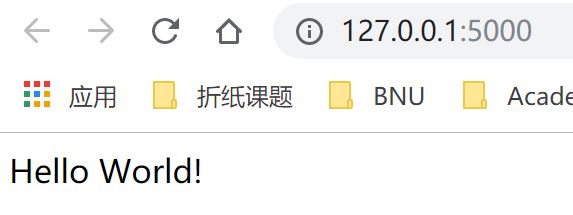
└ views.py # 存放所有视图函数，如果多，将其变为一个包

# Hello World 开始

## 实例

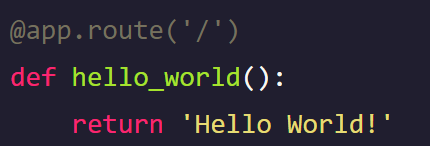






## 代码解析

变量app是一个Flask实例，通过下面的方式



当客户端访问/时，将响应hello\_world()函数返回的内容。注意，这不是返回Hello World!这么简单，Hello World!只是HTTP响应报文的实体部分，状态码等信息既可以由Flask自动处理，当然也可以通过编程来制定。

# 修改Flask的配置

app = Flask(\_\_name\_\_)

上面的代码中，python内置变量\_\_name\_\_的值是字符串\_\_main\_\_ 。Flask类将这个参数作为程序名称。当然这个是可以自定义的，比如:

app = Flask("myapp")

Flask默认使用static目录存放静态资源，templates目录存放模板，也可以通过设置参数更改：

app = Flask("myapp", static\_folder="path1", template\_folder="path2")

# 调试模式

上面的server.py中以app.run()方式运行，这种方式下，如果服务器端出现错误是不会在客户端显示的。但是在开发环境中，显示错误信息是很有必要的，要显示错误信息，应该以下面的方式运行Flask：

app.run(debug=True)

将debug设置为True的另一个好处是，程序启动后，会自动检测源码是否发生变化，若有变化则自动重启程序。这可以帮我们省下很多时间。

还可以使用这种途径开启调试模式:

app.debug = True

app.run()

# 绑定IP和端口

默认情况下，Flask绑定IP为127.0.0.1，端口为5000。也可以通过下面的方式自定义：

app.run(host='0.0.0.0', port=8080, debug=True)

0.0.0.0代表电脑所有的IP。以上我们绑定了8080端口, 启动服务后我们访问的网址将是: [http://127.0.0.1:8080/](http://127.0.0.1:5000/)

# 列出所有的url参数

在server.py中添加以下内容：

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

return request.args.\_\_str\_\_()

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

在浏览器中访问http://127.0.0.1:5000/?name=Loen&age&app=ios&app=android，将显示：

ImmutableMultiDict([('name', 'Loen'), ('age', ''), ('app', 'ios'), ('app', 'android')])

较新的浏览器也支持直接在url中输入中文（最新的火狐浏览器内部会帮忙将中文转换成符合URL规范的数据），在浏览器中访问http://127.0.0.1:5000/?info=我爱你，将显示：

ImmutableMultiDict([('info', '我爱你')])

# request.full\_path和request.path

可以通过request.full\_path和request.path 查看浏览器传给我们的Flask服务的数据

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

print(request.path) # 这里的 print 会在控制台中输出

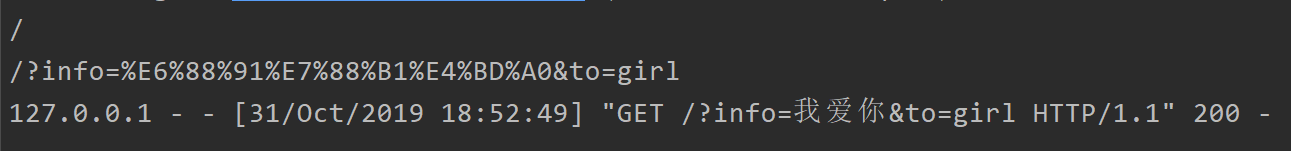
print(request.full\_path)

return request.args.\_\_str\_\_()

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

浏览器访问http://127.0.0.1:5000/?info=我爱你&to=girl，运行server.py的终端会输出：



# 获取指定的参数值

要获取键info对应的值，如下修改server.py：

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/')

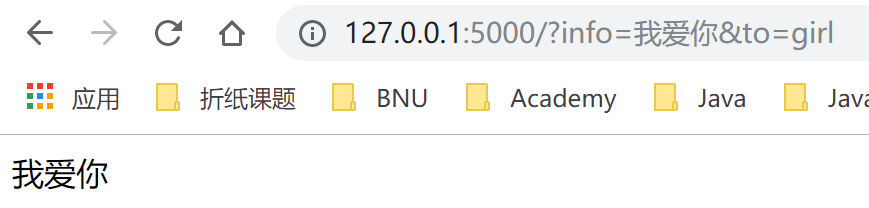
def hello\_world():

return request.args.get('info')

if \_\_name\_\_ == '\_\_main\_\_':

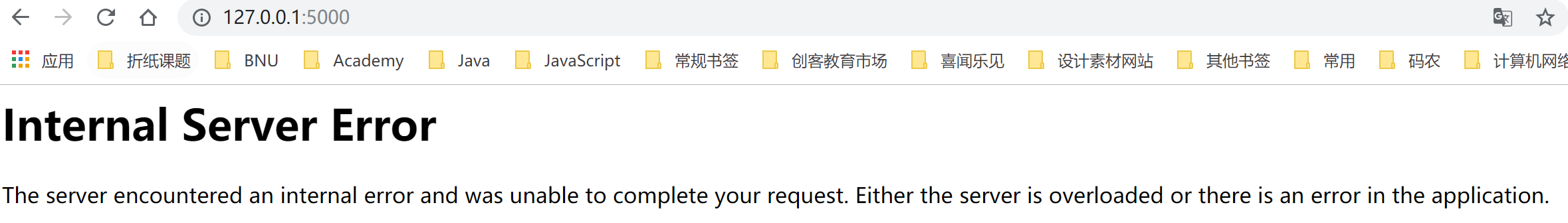
app.run(port=5000)

运行server.py，在浏览器中访问http://127.0.0.1:5000/?info=我爱你&to=girl，浏览器将显示

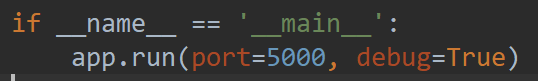


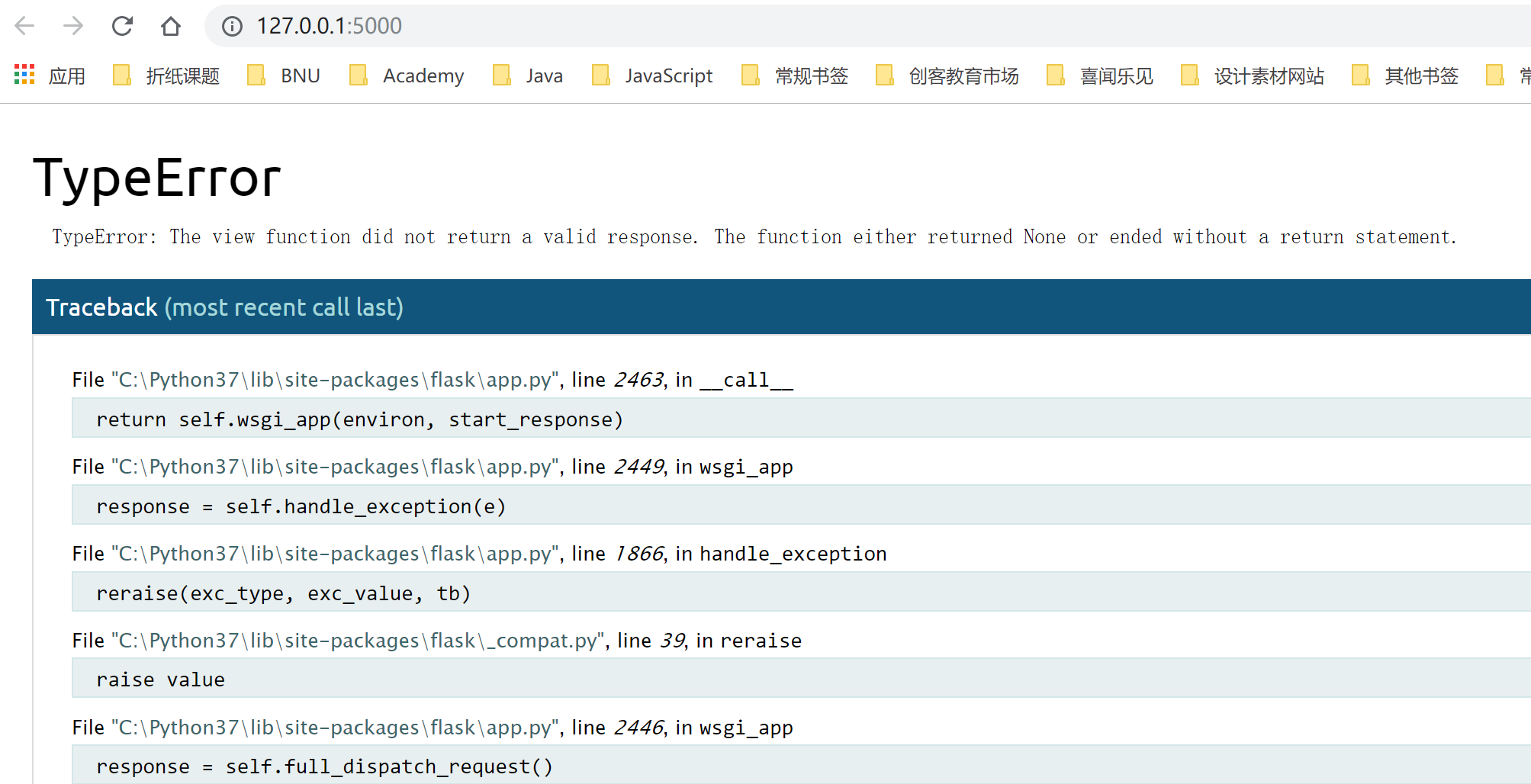
# 开启debug

刚刚的程序直接访问127.0.0.1:5000会报错



开启debug模式之后，会显示错误信息





因为Flask不允许返回None, 而没有在URL参数中找到info。所以request.args.get('info') 返回Python内置的None。

可以设置取不到数据时用的值：

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

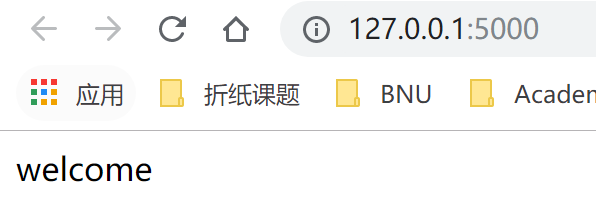
r = request.args.get('info', 'welcome')

return r

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

函数request.args.get的第二个参数用来设置默认值。此时在浏览器访问http://127.0.0.1:5000/，将显示：



# getlist 处理多值

如果我们请求 http://127.0.0.1:5000/?name=Loen&age&app=ios&app=android，app有两个值。

如果我们的代码是：

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

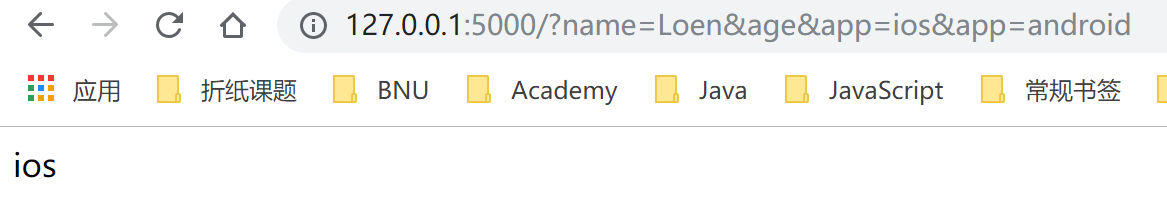
r = request.args.get('app')

return r

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

在浏览器中请求时，我们只会看到 ios。



我们可以使用getlist获取所有的app的值。

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

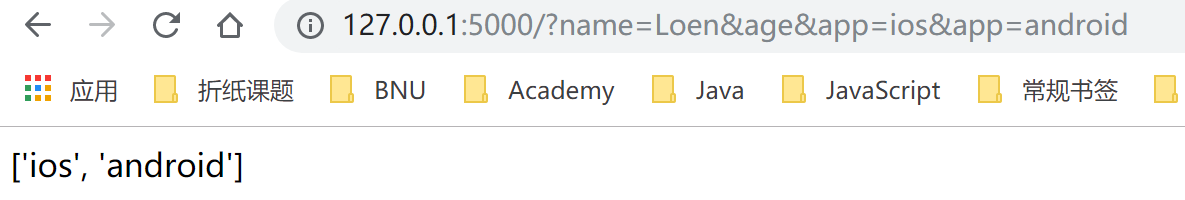
r = request.args.getlist('app') # 返回一个list

return str(r)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

浏览器输入 http://127.0.0.1:5000/?name=Loen&age&app=ios&app=android，我们会看到['ios', 'android']。



# 获取POST方法传送的数据

## 看POST数据内容

以用户注册为例子，我们需要向服务器/register传送用户名name和密码password。如下编写server.py。

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/register', methods=['POST'])

def register():

print(request.headers)

print(request.stream.read())

return 'welcome'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

[@app.route](mailto:%60@app.route)('/register', methods=['POST'])是指url/register只接受POST方法。可以根据需要修改methods参数，例如如果想要让它同时支持GET和POST，这样写：

@app.route('/register', methods=['GET', 'POST'])

浏览器模拟工具client.py内容如下：

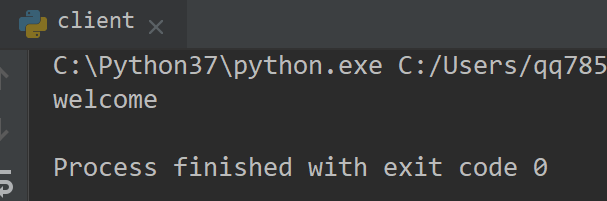
import requests

user\_info = {'name': 'Loen', 'password': 'loveyou'}

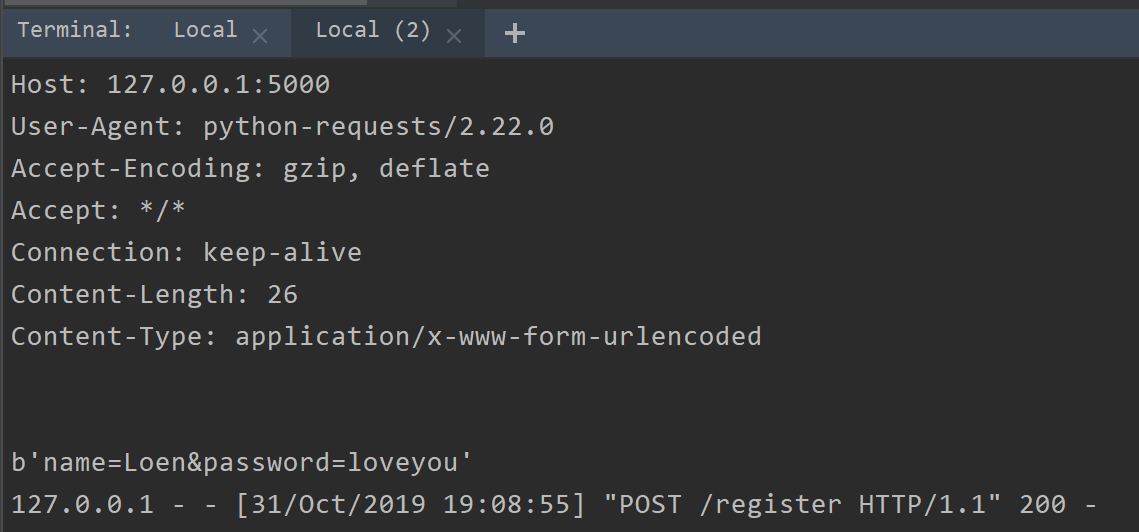
r = requests.post("http://127.0.0.1:5000/register", data=user\_info)

print(r.text)

运行server.py，然后运行client.py。client.py将输出：



而server.py在终端中输出以下调试信息（通过print输出）：



前6行是client.py生成的HTTP请求头，由print(request.headers)输出。

请求体的数据，我们通过print(request.stream.read())输出，结果是：

b'name=Loen&password=loveyou'

## 解析POST数据

将服务器代码改成：

from flask import Flask, request

app = Flask(\_\_name\_\_)

@app.route('/register', methods=['POST'])

def register():

print(request.headers)

# print(request.stream.read()) # 不要用，否则下面的form取不到数据

print(request.form)

print(request.form['name'])

print(request.form.get('name'))

print(request.form.getlist('name'))

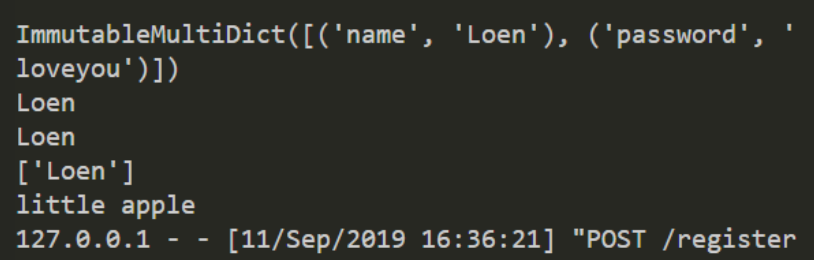
print(request.form.get('nickname', default='little apple'))

return 'welcome'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

执行client.py请求数据，服务器代码会在终端输出：



request.form会自动解析数据。

request.form['name']和request.form.get('name')都可以获取name对应的值。对于request.form.get()可以为参数default指定值以作为默认值。所以：

print(request.form.get('nickname', default='little apple'))

输出的是默认值

little apple

## 获取POST中的列表数据

如果name有多个值，可以使用request.form.getlist('name')，该方法将返回一个列表。将client.py改一下：

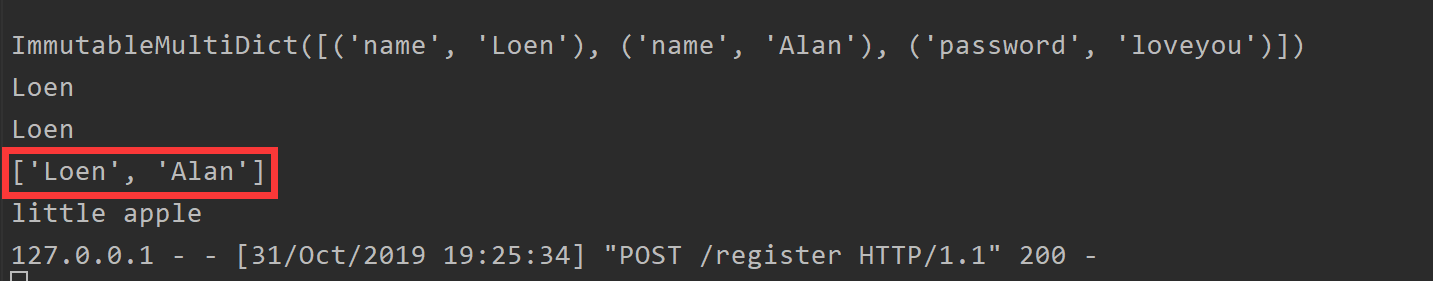
import requests

user\_info = {'name': ['Loen', 'Alan'], 'password': 'loveyou'}

r = requests.post("http://127.0.0.1:5000/register", data=user\_info)

print(r.text)

此时运行client.py，print(request.form.getlist('name'))将输出：



# 处理和响应JSON数据

使用 HTTP POST 方法传到网站服务器的数据格式可以有很多种，比如「获取POST方法传送的数据」中的name=Loen&password=loveyou这种用过&符号分割的key-value键值对格式。我们也可以用JSON格式、XML格式。相比XML的重量、规范繁琐，JSON显得非常小巧和易用。

如果POST的数据是JSON格式，request.json会自动将json数据转换成Python类型（字典或者列表）。

编写server.py:

from flask import Flask, request

app = Flask("myapp")

@app.route('/add', methods=['POST'])

def add():

print(request.headers)

print(type(request.json))

print(request.json)

result = request.json['n1'] + request.json['n2']

return str(result)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='127.0.0.1', port=5000, debug=True)

编写client.py模拟浏览器请求：

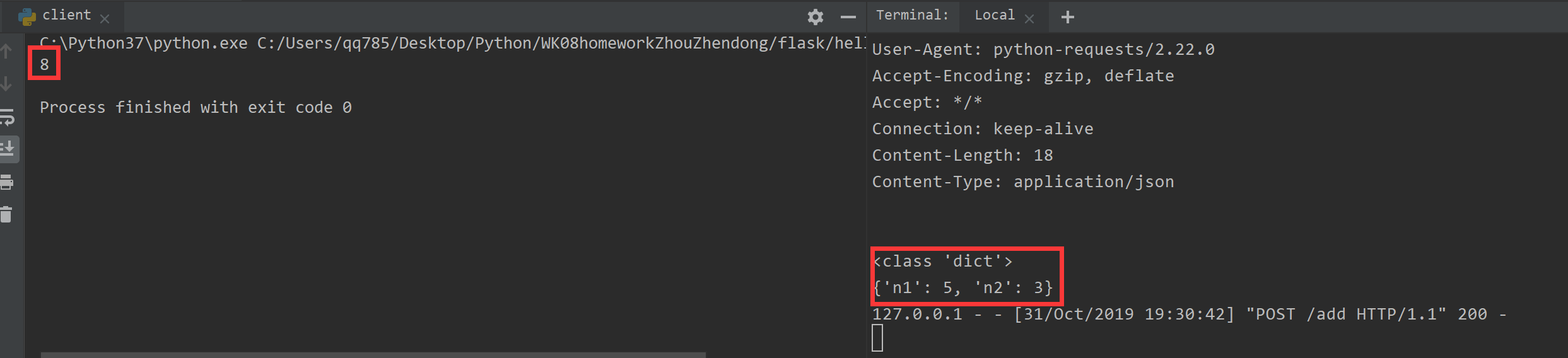
import requests

json\_data = {'n1': 5, 'n2': 3}

r = requests.post("http://127.0.0.1:5000/add", json=json\_data)

print(r.text)

运行server.py，然后运行client.py，client.py 会在终端输出：



注意，请求头中Content-Type的值是application/json。

## 响应JSON

响应JSON时，除了要把响应体改成JSON格式，响应头的Content-Type也要设置为application/json。

编写server.py：

from flask import Flask, request, Response

import json

app = Flask("myapp")

@app.route('/add', methods=['POST'])

def add():

result = {'sum': request.json['n1'] + request.json['n2']}

return Response(json.dumps(result), mimetype='application/json')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='127.0.0.1', port=5000, debug=True)

修改后运行。

编写client.py：

import requests

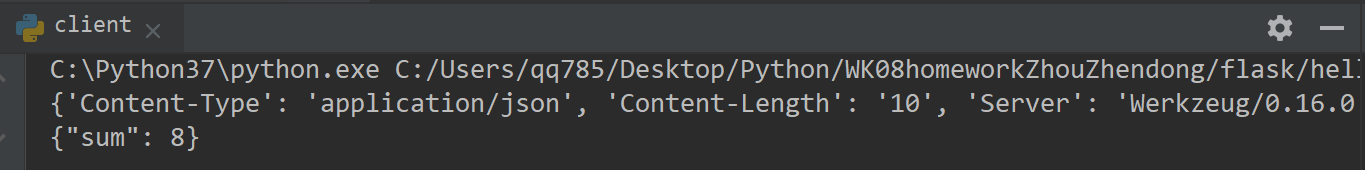
json\_data = {'n1': 5, 'n2': 3}

r = requests.post("http://127.0.0.1:5000/add", json=json\_data)

print(r.headers)

print(r.text)

运行client.py，将显示：



client终端返回的第一段内容是服务器的响应头，第二段内容是响应体，也就是服务器返回的JSON格式数据。

另外，如果需要服务器的HTTP响应头具有更好的可定制性，比如自定义Server，可以如下修改add()函数：

@app.route('/add', methods=['POST'])

def add():

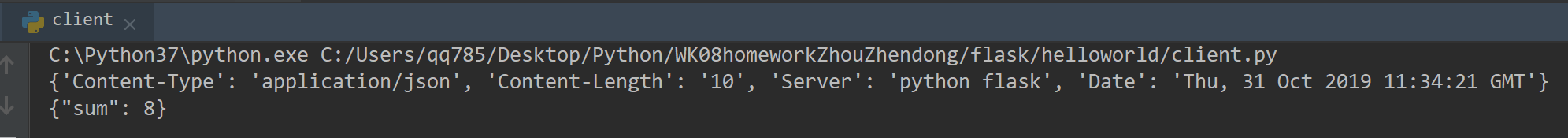
result = {'sum': request.json['n1'] + request.json['n2']}

resp = Response(json.dumps(result), mimetype='application/json')

resp.headers.add('Server', 'python flask')

return resp

client.py运行后会输出：



## 响应JSON

使用 jsonify 工具函数。

from flask import Flask, request, jsonify

app = Flask("myapp")

@app.route('/add', methods=['POST'])

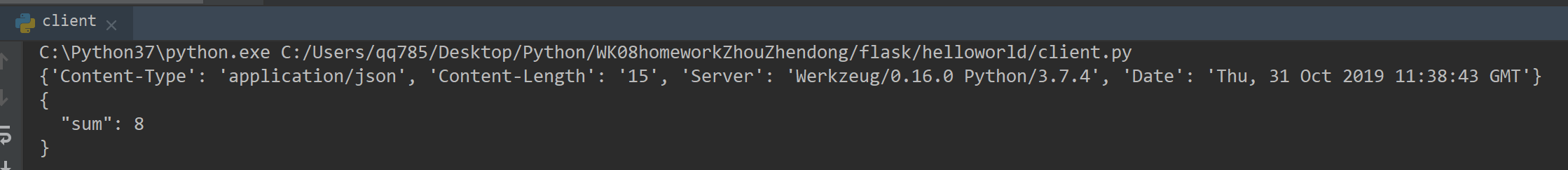
def add():

result = {'sum': request.json['n1'] + request.json['n2']}

return jsonify(result)

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='127.0.0.1', port=5000, debug=True)



# 上传文件

已上传的文件存储在内存或是文件系统中一个临时的位置。你可以通过请求对象的 files 属性访问它们。每个上传的文件都会存储在这个字典里。它表现近乎为一个标准的 Python file 对象，但它还有一个 save() 方法，这个方法允许你把文件保存到服务器的文件系统上。

我们以上传图片为例：假设将上传的图片只允许'png'、'jpg'、'jpeg'、'gif' 这四种格式，通过url/upload使用POST上传，上传的图片存放在服务器端的static/uploads目录下。

首先在项目HelloWorld中创建目录static/uploads, 这时候我们的目录结构为：

HelloWorld/

├ static/

│ ├ uploads/

├ templates/

├ server.py

├ client.py

werkzeug库可以判断文件名是否安全

server.py代码：

from flask import Flask, request

from werkzeug.utils import secure\_filename

import os

app = Flask(\_\_name\_\_)

# 文件上传目录

app.config['UPLOAD\_FOLDER'] = 'static/uploads/'

# 支持的文件格式

app.config['ALLOWED\_EXTENSIONS'] = {'png', 'jpg', 'jpeg', 'gif'} # 集合类型

# 判断文件名是否是我们支持的格式

def allowed\_file(filename):

return '.' in filename and \

filename.rsplit('.', 1)[1] in app.config['ALLOWED\_EXTENSIONS']

@app.route('/upload', methods=['POST'])

def upload():

upload\_file = request.files['image']

if upload\_file and allowed\_file(upload\_file.filename): # 上传前文件在客户端的文件名

filename = secure\_filename(upload\_file.filename)

# 将文件保存到 static/uploads 目录，文件名同上传时使用的文件名

upload\_file.save(os.path.join(app.root\_path, app.config['UPLOAD\_FOLDER'], filename))

return 'info is '+request.form.get('info', '')+'. success'

else:

return 'failed'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

app.config中的config是字典的子类，可以用来设置自有的配置信息，也可以设置自己的配置信息。

函数allowed\_file(filename)用来判断filename是否有后缀以及后缀是否在app.config['ALLOWED\_EXTENSIONS']中。

upload\_file是上传文件对应的对象。

app.root\_path获取server.py所在目录在文件系统中的绝对路径。

upload\_file.save(path)用来将upload\_file保存在服务器的文件系统中，参数最好是绝对路径。函数os.path.join()用来将使用合适的路径分隔符将路径组合起来。

客户端client.py：

import requests

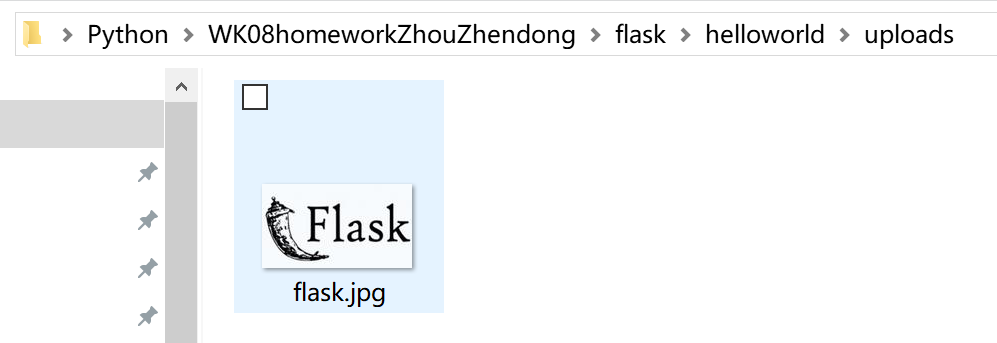
file\_data = {'image': open('flask.jpg', 'rb')}

user\_info = {'info': 'flask'}

r = requests.post("http://127.0.0.1:5000/upload", data=user\_info, files=file\_data)

print(r.text)

上传成功



要控制上产文件的大小，可以设置请求实体的大小，例如：

app.config['MAX\_CONTENT\_LENGTH'] = 16 \* 1024 \* 1024 #16MB

不过，在处理上传文件时候，需要使用try:...except:...。

如果要获取上传文件的内容可以：

file\_content = request.files['image'].stream.read()

# Restful URL - 变量规则

简单来说，Restful URL可以看做是对 URL 参数的替代。

编辑server.py：

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/user/<username>')

def user(username):

print(username)

print(type(username))

return 'hello ' + username

@app.route('/user/<username>/friends')

def user\_friends(username):

print(username)

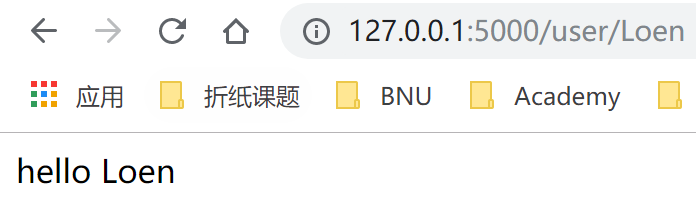
print(type(username))

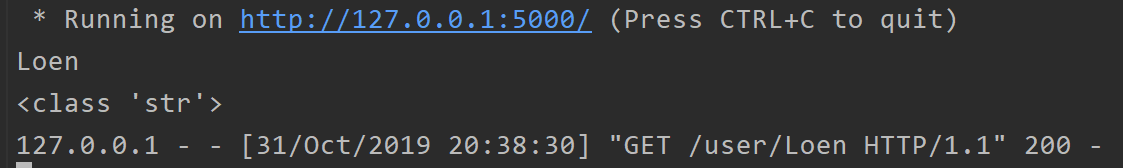
return 'In user\_friends get username: ' + username

if \_\_name\_\_ == '\_\_main\_\_':

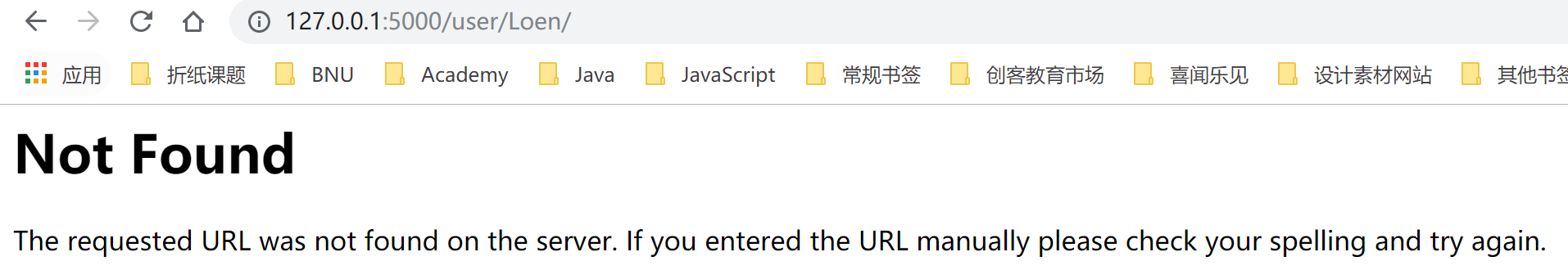
app.run(port=5000, debug=True)

运行server.py。使用浏览器访问http://127.0.0.1:5000/user/Loen，server.py将输出：

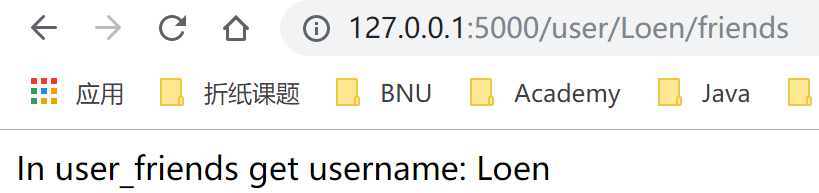




而访问<http://127.0.0.1:5000/user/Loen/>



浏览器访问http://127.0.0.1:5000/user/Loen/friends，可以看到：



# 转换类型

由上面的示例可以看出，使用 Restful URL 得到的变量默认为str对象。我们可以用flask内置的转换机制，即在route中指定转换类型。

编写server.py：

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/page/<int:num>')

def page(num):

print(num)

print(type(num))

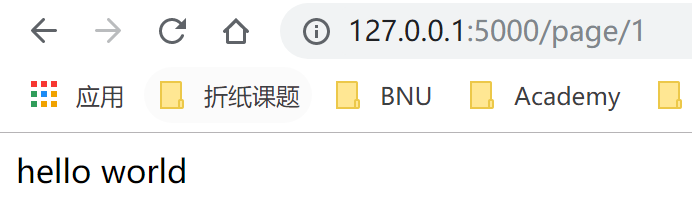
return 'hello world'

if \_\_name\_\_ == '\_\_main\_\_':

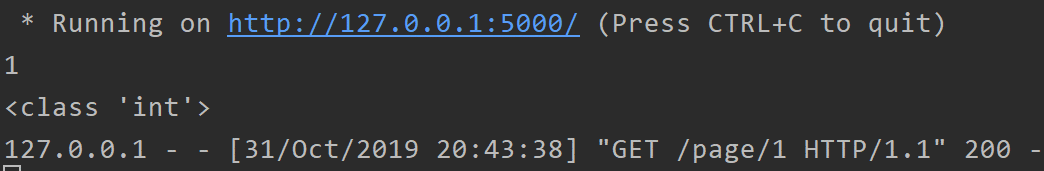
app.run(port=5000, debug=True)

[@app.route](mailto:%60@app.route)('/page/int:num')会将num变量自动转换成int类型。

运行上面的程序，在浏览器中访问 <http://127.0.0.1:5000/page/1>，



server.py控制台将输出如下内容：



如果访问的是http://127.0.0.1:5000/page/loen，我们会得到404响应。

有3个默认的转换器：

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

# 编写转换器

自定义的转换器是一个继承werkzeug.routing.BaseConverter的类，修改to\_python和to\_url方法即可。

to\_python方法用于将url中的变量转换后供被[@app.route](mailto:%60@app.route)包装的函数使用，to\_url方法用于flask.url\_for中的参数转换。

下面是一个示例，将server.py修改如下：

from flask import Flask, url\_for

from werkzeug.routing import BaseConverter

class MyIntConverter(BaseConverter):

def \_\_init\_\_(self, url\_map):

super(MyIntConverter, self).\_\_init\_\_(url\_map)

def to\_python(self, value):

return int(value)

def to\_url(self, value):

return value \* 2

app = Flask(\_\_name\_\_)

app.url\_map.converters['my\_int'] = MyIntConverter

@app.route('/page/<my\_int:num>')

def page(num):

print(num)

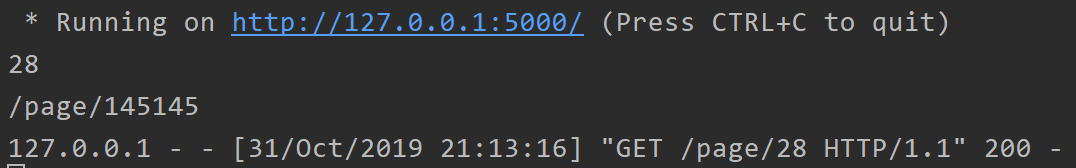
print(url\_for('page', num='145')) # page 对应的是 page函数 ，num 对应对应`/page/<my\_int:num>`中的num，必须是str

return 'hello world'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

浏览器访问http://127.0.0.1:5000/page/28后，server.py的输出信息是：



# 使用url\_for生成链接

工具函数url\_for可以以软编码的形式生成url，提供开发效率。

编辑 server.py：

from flask import Flask, url\_for

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

    pass

@app.route('/user/<name>')

def user(name):

    pass

@app.route('/page/<int:num>')

def page(num):

    pass

@app.route('/test')

def test():

print(url\_for('hello\_test'))

print(url\_for('user', name='loen'))

print(url\_for('page', num=1, q='welcome to w3c 15%2'))

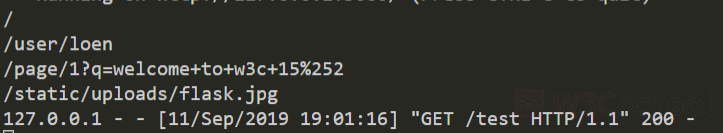
print(url\_for('static', filename='uploads/flask.jpg'))

return 'Hello'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

运行server.py。然后在浏览器中访问http://127.0.0.1:5000/test，server.py控制台将输出以下信息：



反向构建通常比硬编码的描述性更好。更重要的是，它允许一次性修改 URL， 而不是到处边找边改。

URL 构建会转义特殊字符和 Unicode 数据，免去很多麻烦。

如果应用不位于 URL 的根路径（比如，在 /myapplication 下，而不是 / ）， url\_for() 会妥善处理这个问题。

# 使用redirect重定向网址

运行server.py，在浏览器中访问http://127.0.0.1:5000/old，浏览器的url会变成http://127.0.0.1:5000/new，并显示：redirect函数用于重定向，实现机制很简单，就是向客户端（浏览器）发送一个重定向的HTTP报文，浏览器会去访问报文中指定的url。

使用redirect时，给它一个字符串类型的参数就行了。

编辑 server.py：

from flask import Flask, url\_for, redirect

app = Flask(\_\_name\_\_)

@app.route('/old')

def old():

print('this is old')

return redirect(url\_for('new'))

@app.route('/new')

def new():

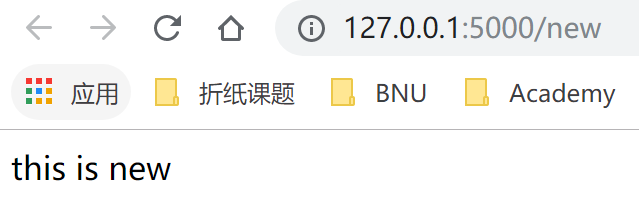
print('this is new')

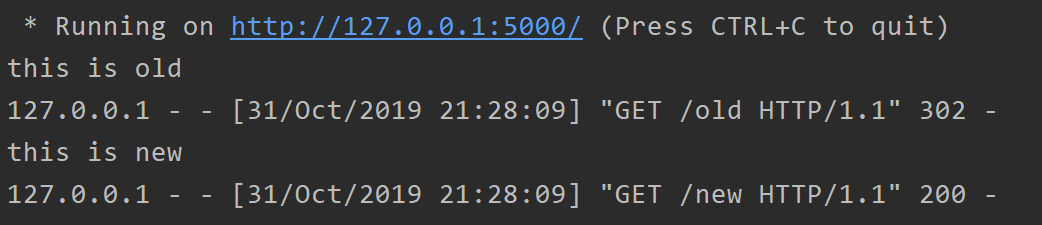
return 'this is new'

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

运行server.py，在浏览器中访问http://127.0.0.1:5000/old，浏览器的url会变成http://127.0.0.1:5000/new，并显示：





# jinja2

jinja2是Flask作者开发的一个模板系统，起初是仿django模板的一个模板引擎，为Flask提供模板支持，由于其灵活，快速和安全等优点被广泛使用。

## jinja2的优点

jinja2之所以被广泛使用是因为它具有以下优点：

相对于Template，jinja2更加灵活，它提供了控制结构，表达式和继承等。

相对于Mako，jinja2仅有控制结构，不允许在模板中编写太多的业务逻辑。

相对于Django模板，jinja2性能更好。

Jinja2模板的可读性很棒。

## jinja2语法

作为一个模板系统，它还提供了特殊的语法，按照它支持的语法进行编写之后，就能使用jinja2模块进行渲染。基本语法在jinja2中，存在三种语法：

1.控制结构 {% %}

2.变量取值 {{ }}

3.注释 {# #}

下面是一个简单的jinja2例子:

{# This is jinja code #}

{% for file in filenames %}

...

{% endfor %}

可以看到，for循环的使用方式和Python比较类似，但是没有了句尾的冒号，另外需要使用endfor最为结尾，

其实在jinja2中，if也是一样的，结尾需要使用endif。

## jinja2变量

jinja2模板中使用 {{ }} 语法表示一个变量，它是一种特殊的占位符。

当利用jinja2进行渲染的时候，它会把这些特殊的占位符进行填充/替换，jinja2支持python中所有的Python数据类型比如列表、元组、对象等。

<p>this is a dicectory:{{ mydict['key'] }} </p>

<p>this is a list:{{ mylist[3] }} </p>

<p>this is a object:{{ myobject.something() }} </p>

## jinja2中的过滤器

变量可以通过“过滤器”进行修改，过滤器可以理解为是jinja2里面的内置函数和字符串处理函数。常用的过滤器有：

| **过滤器名称** | **说明** |
| --- | --- |
| safe | 渲染时值不转义 |
| capitialize | 把值的首字母转换成大写 |
| 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 |
| join | 拼接多个值为字符串 |
| replace | 替换字符串的值 |
| round | 默认对数字进行四舍五入，也可以用参数进行控制 |
| int | 把值转换成整型 |

只需要在变量后面使用管道 |  分割，多个过滤器可以链式调用，前一个过滤器的输出会作为后一个过滤器的输入。

{{ 'abc' | captialize }}

# Abc

{{ 'abc' | upper }}

# ABC

{{ 'hello world' | title }}

# Hello World

{{ "hello world" | replace('world','Loen') | upper }}

# HELLO LOEN

{{ 16.18 | round | int }}

# 16

## jinja2的for循环

jinja2中的for循环用于迭代Python的数据类型，包括列表，元组和字典。在jinja2中不存在while循环。迭代列表

<ul>

{% for user in users %}

<li>{{ user.username|title }}</li>

{% endfor %}

</ul>

迭代字典

<dl>

{% for key, value in my\_dict.iteritems() %}

<dt>{{ key }}</dt>

<dd>{{ value}}</dd>

{% endfor %}

</dl>

当然也可以加入else语句，在循环正确执行完毕后执行

## jinja2的宏

宏类似于Python中的函数，我们在宏中定义行为，还可以进行传递参数，就像Python中的函数一样。在宏中定义一个宏的关键字是macro，后面跟宏的名称和参数等

{% macro input(name,age=18) %} # 参数age的默认值为18

<input type='text' name="{{ name }}" value="{{ age }}" >

{% endmacro %}

调用方法也和Python的类似

<p>{{ input('Loen') }} </p>

<p>{{ input('Loen',age=28) }} </p>

## jinja2的继承和Super函数

jinja2中最强大的部分就是模板继承。

模板继承允许我们创建一个基本(骨架)文件，其他文件从该骨架文件继承，然后针对自己需要的地方进行修改。

jinja2的骨架文件中，利用block关键字表示其包涵的内容可以进行修改。

以下面的骨架文件base.html为例：

<!DOCTYPE html>

<html lang="en">

<head>

{% block head %}

<link rel="stylesheet" href="style.css"/>

<title>{% block title %}{% endblock %} - My Webpage</title>

{% endblock %}

</head>

<body>

<div id="content">{% block content %}{% endblock %}</div>

<div id="footer">

{% block footer %}

<script>This is javascript code </script>

{% endblock %}

</div>

</body>

</html>

这里定义了四处 block，即：head，title，content，footer。

那怎么进行继承和变量替换呢？

注意看下面的文件

{% extend "base.html" %} # 继承base.html文件

{% block title %} Dachenzi {% endblock %} # 定制title部分的内容

{% block head %}

{{ super() }} # 用于获取原有的信息

<style type='text/css'>

.important { color: #FFFFFF }

</style>

{% endblock %}

# 其他不修改的原封不同的继承

注意: super()函数 表示获取block块中定义的原来的内容。

## 使用Jinja2模板引擎

在模板里，可以访问 request 、 session 和 g [1] 对象， 以及 get\_flashed\_messages() 函数。Flask与模板相关的函数有：

flask.render\_template(template\_name\_or\_list, \*\*context)

flask.render\_template\_string(source, \*\*context)

flask.get\_template\_attribute(template\_name, attribute)

这个实例中使用了模板继承、if判断、for循环。

创建并编辑 templates/default.html

<html>

<head>

    <title>

        {% if page\_title %}

            {{ page\_title }}

        {% endif %}

    </title>

</head>

<!--

在``标签中使用了if判断，如果给模板传递了`page\_title`变量，显示之，否则，不显示。

-->

<body>

{% block body %}

    {% for key in user\_info %}

        <p>{{ key }}: {{ user\_info[key] }} </p>

    {% endfor %}

{% endblock %}

变量user\_info应该是一个字典，for循环用来循环输出键值对。

编辑 server.py

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route('/user')

def user():

user\_info = {

'name': 'loen',

'email': '425389019@qq.com',

'age':0,

'github': 'https://github.com/lucoo01'

}

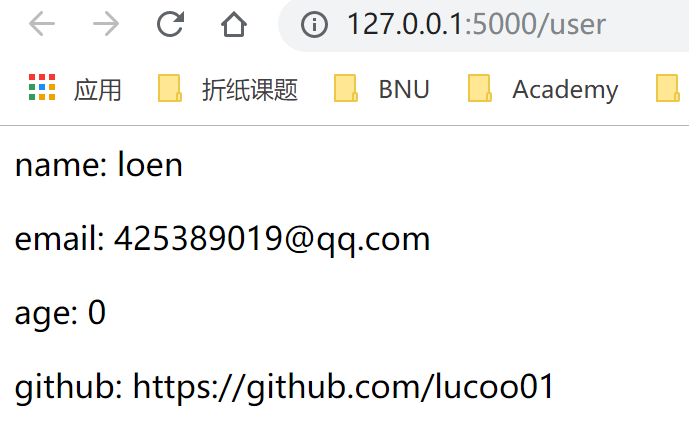
return render\_template('default.html', page\_title='loen\'s info', user\_info=user\_info)

if \_\_name\_\_ == '\_\_main\_\_':

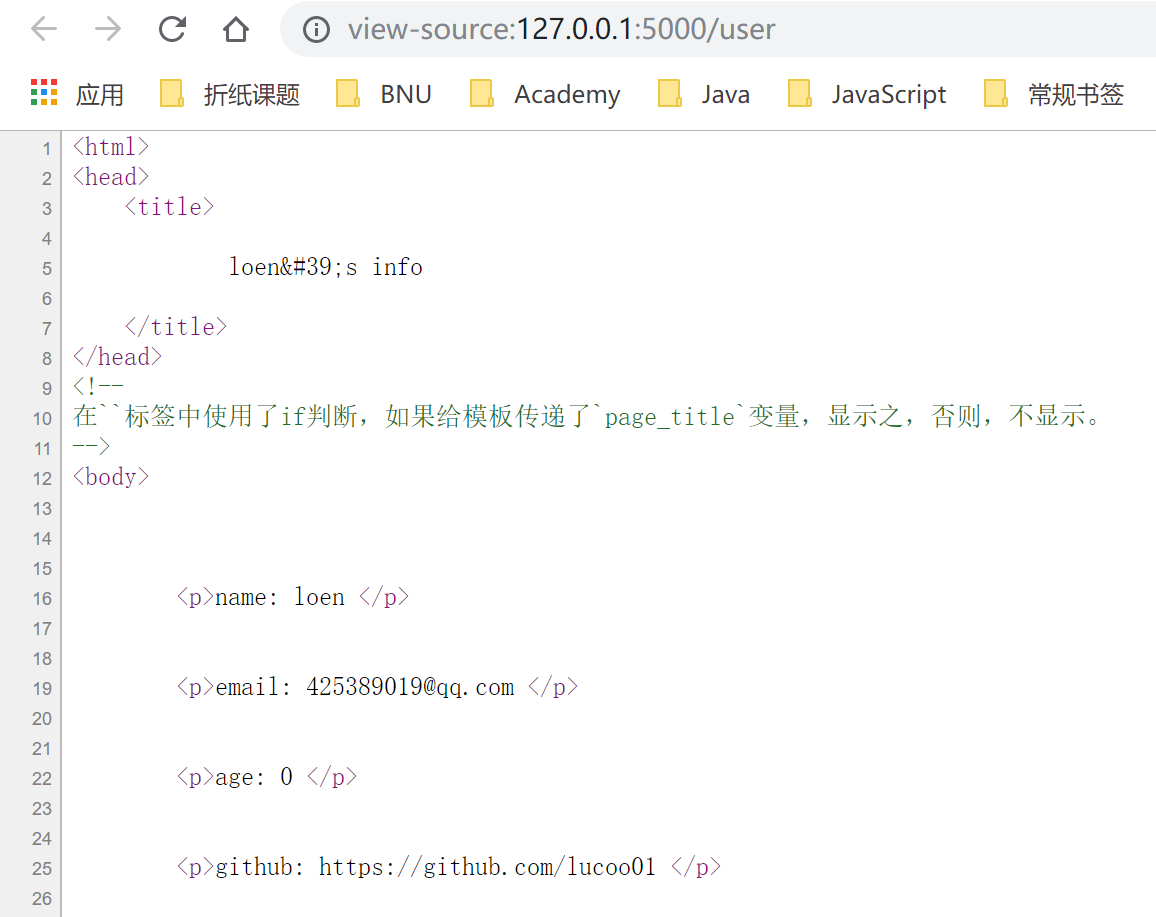
app.run(port=5000, debug=True)

render\_template()函数的第一个参数指定模板文件，后面的参数是要传递的数据。

运行 server.py 在浏览器中访问http://127.0.0.1:5000/user，效果图如下：



网页源代码：



各个位置的数据已经修改。

在模板里，也可以访问 request 、 session 和 g 对象， 以及 get\_flashed\_messages() 函数。

# 自定义404等错误的响应

要处理HTTP错误，可以使用flask.abort函数。

编辑server.py：

from flask import Flask, render\_template\_string, abort

app = Flask(\_\_name\_\_)

@app.route('/user')

def user():

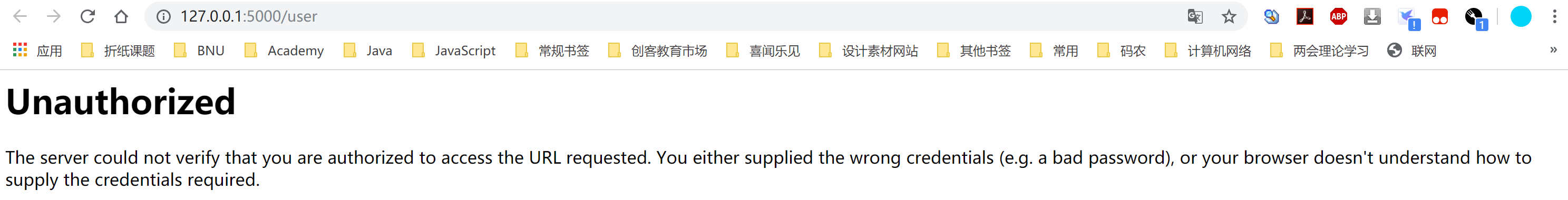
abort(401) # Unauthorized 未授权

print('Unauthorized, 请先登录')

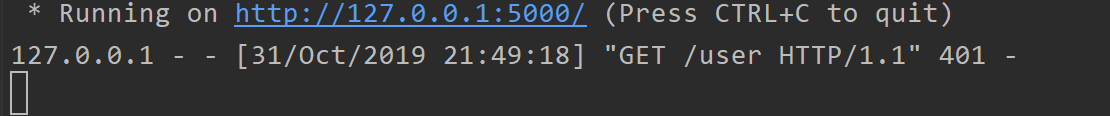
if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

运行 server.py，浏览器访问http://127.0.0.1:5000/user，效果如下：



要注意的是，server.py中abort(401)后的print并没有执行。



## 自定义错误页面

编辑server.py：

from flask import Flask, render\_template\_string, abort

app = Flask(\_\_name\_\_)

@app.route('/user')

def user():

abort(401) # Unauthorized

@app.errorhandler(401)

def page\_unauthorized(error):

return render\_template\_string('<h1> Unauthorized </h1><h2>{{ error\_info }}</h2>', error\_info=error), 401

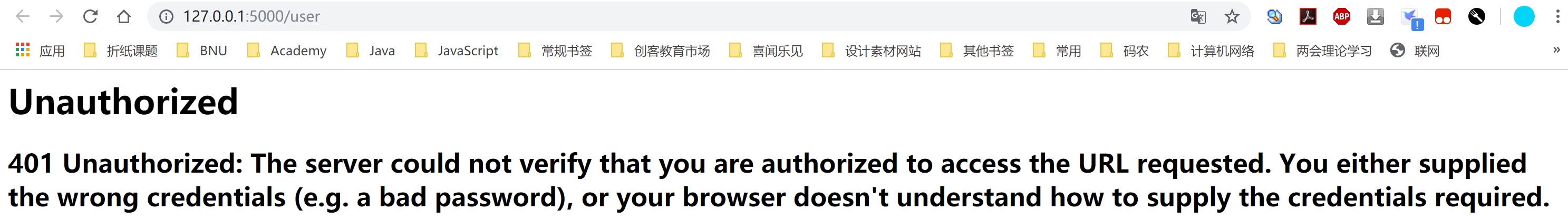
if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

page\_unauthorized 函数返回的是一个元组，401 代表HTTP 响应状态码。

如果省略401，则响应状态码会变成默认的 200。

运行 server.py，浏览器访问 http://127.0.0.1:5000/user，效果如下：



# 用户会话

session 用来记录用户的登录状态，一般基于cookie实现。

编辑 server.py

from flask import Flask, render\_template\_string, \

session, request, redirect, url\_for

app = Flask(\_\_name\_\_)

app.secret\_key = 'LoenDSdtj\9bX#%@!!\*(0&^%)'

@app.route('/login')

def login():

page = '''

<form action="{{ url\_for('do\_login') }}" method="post">

<p>name: <input type="text" name="user\_name" /></p>

<input type="submit" value="Submit" />

</form>

'''

return render\_template\_string(page)

@app.route('/do\_login', methods=['POST'])

def do\_login():

name = request.form.get('user\_name')

session['user\_name'] = name

return 'success'

@app.route('/show')

def show():

return session['user\_name']

@app.route('/logout')

def logout():

session.pop('user\_name', None)

return redirect(url\_for('login'))

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

**代码的含义**

app.secret\_key用于给session加密。

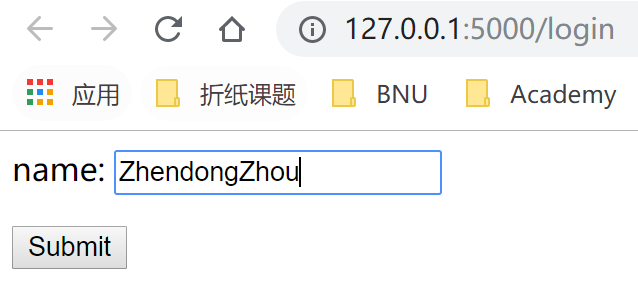
在/login中将向用户展示一个表单，要求输入一个名字，submit后将数据以post的方式传递给/do\_login，/do\_login将名字存放在session中。

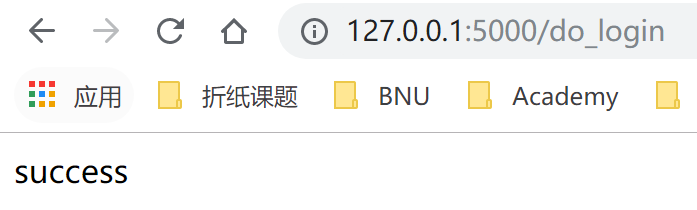
如果用户成功登录，访问/show时会显示用户的名字。此时，打开调试工具，选择session面板，会看到有一个cookie的名称为session。

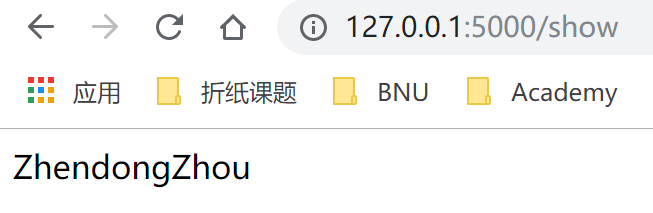
/logout用于登出，通过将session中的user\_name字段pop即可。Flask中的session基于字典类型实现，调用pop方法时会返回pop的键对应的值；如果要pop的键并不存在，那么返回值是pop()的第二个参数。

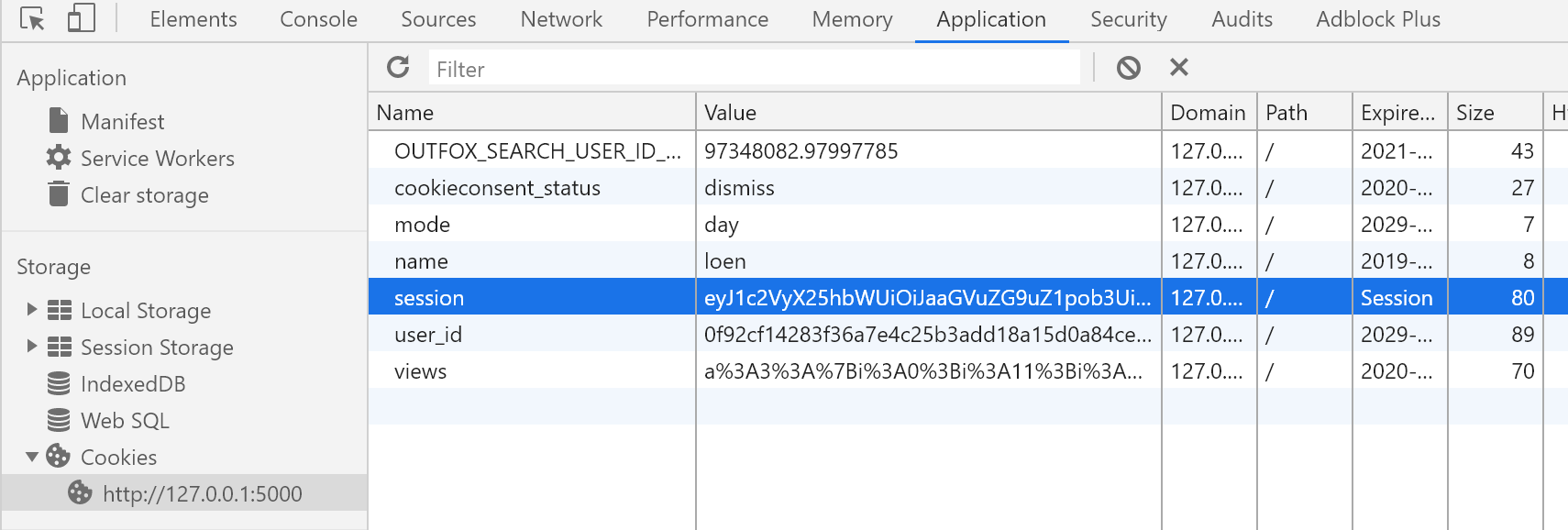
另外，使用redirect()重定向时，一定要在前面加上return。

进入http://127.0.0.1:5000/login，输入name，点击submit：









## 设置session的有效时间

设置session的有效时间设置为5分钟。

from datetime import timedelta

from flask import session, app

session.permanent = True

app.permanent\_session\_lifetime = timedelta(minutes=5)

# 使用Cookie

Cookie是存储在客户端的记录访问者状态的数据。

常用的用于记录用户登录状态的session大多是基于cookie实现的。

cookie可以借助flask.Response来实现。

修改server.py：

from flask import Flask, request, Response, make\_response

import time

app = Flask(\_\_name\_\_)

@app.route('/add')

def login():

res = Response('add cookies')

res.set\_cookie(key='name', value='loen', expires=time.time()+6\*60)

return res

@app.route('/show')

def show():

return request.cookies.\_\_str\_\_()

@app.route('/del')

def del\_cookie():

res = Response('delete cookies')

res.set\_cookie('name', '', expires=0)

return res

if \_\_name\_\_ == '\_\_main\_\_':

app.run(port=5000, debug=True)

使用Response.set\_cookie添加和删除cookie。

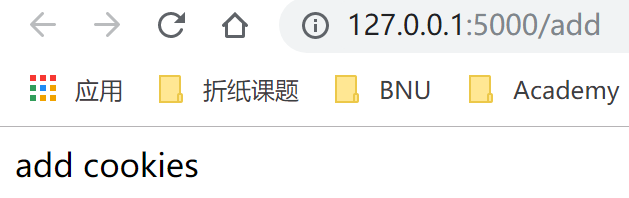
expires参数用来设置cookie有效时间，值可以是datetime对象或者unix时间戳。

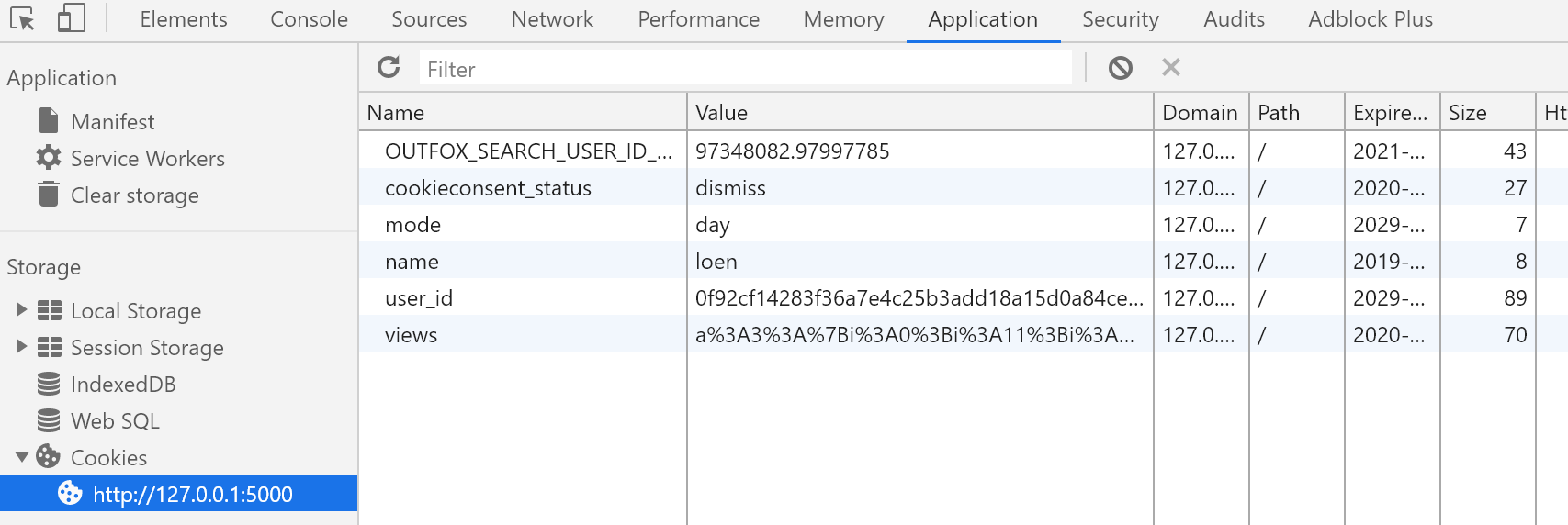
res.set\_cookie(key='name', value='loen', expires=time.time()+6\*60)

上面的expire参数的值表示cookie在从现在开始的6分钟内都是有效的。

要删除cookie，将expire参数的值设为0即可：

res.set\_cookie('name', '', expires=0)





# 闪存系统 flashing system

Flask 的闪存系统（flashing system）用于向用户提供反馈信息，这些反馈信息一般是对用户上一次操作的反馈。

反馈信息是存储在服务器端的，当服务器向客户端返回反馈信息后，**这些反馈信息会被服务器端删除。**

编写 server.py

from flask import Flask, flash, get\_flashed\_messages

import time

app = Flask(\_\_name\_\_)

app.secret\_key = 'some\_secret'

@app.route('/')

def index():

return 'Hello index'

@app.route('/gen')

def gen():

info = 'access at '+ time.time().\_\_str\_\_()

flash(info)

return info

@app.route('/show1')

def show1():

return get\_flashed\_messages().\_\_str\_\_()

@app.route('/show2')

def show2():

return get\_flashed\_messages().\_\_str\_\_()

if \_\_name\_\_ == "\_\_main\_\_":

app.run(port=5000, debug=True)

运行 server.py：

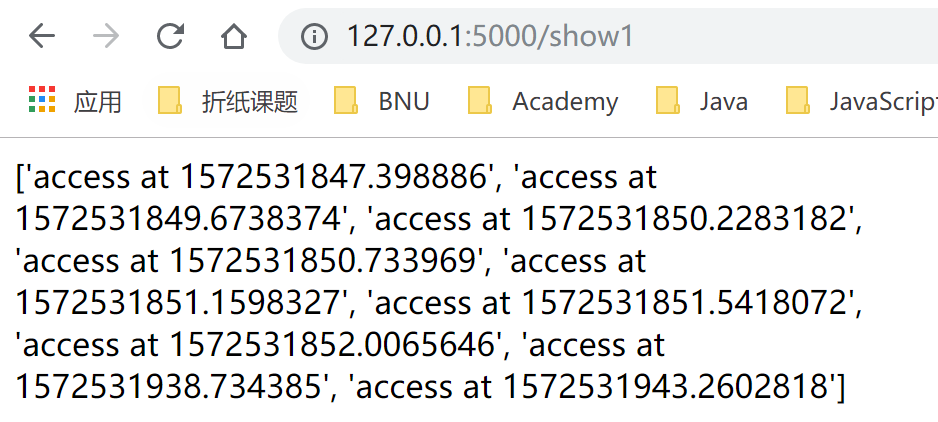
打开浏览器，访问http://127.0.0.1:5000/gen，浏览器界面显示（注意，时间戳是动态生成的，每次都会不一样，除非并行访问）：

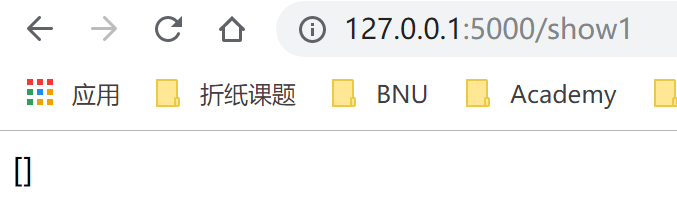


查看浏览器的cookie，可以看到session，随着gen页面的生成不断变化：

访问show1 得到的列表内容 是之前访问http://127.0.0.1:5000/gen得到的内容。此时，cookie中已经没有session了。

如果使用浏览器访问http://127.0.0.1:5000/show1或者http://127.0.0.1:5000/show2，只会得到空





## 高级用法

flash系统也支持对flash的内容进行分类。修改server.py内容：

from flask import Flask, flash, get\_flashed\_messages

import time

app = Flask(\_\_name\_\_)

app.secret\_key = 'some\_secret'

@app.route('/')

def index():

return 'Hello Index'

@app.route('/gen')

def gen():

info = 'access at '+ time.time().\_\_str\_\_()

flash('show1 '+info, category='show1')

flash('show2 '+info, category='show2')

return info

@app.route('/show1')

def show1():

return get\_flashed\_messages(category\_filter='show1').\_\_str\_\_()

@app.route('/show2')

def show2():

return get\_flashed\_messages(category\_filter='show2').\_\_str\_\_()

if \_\_name\_\_ == "\_\_main\_\_":

app.run(port=5000, debug=True)

不过，由上面的代码可以知道，此时生成了两个flash信息，但分类(category)不同。

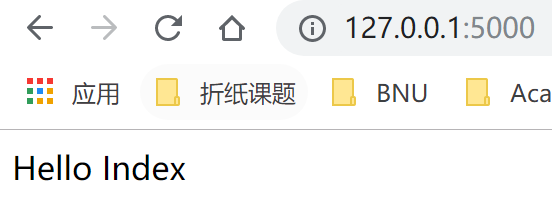
使用浏览器访问http://127.0.0.1:5000/show1，得到如下内容：



而继续访问http://127.0.0.1:5000/show2，得到的内容为空：



运行结果:



## 在模板文件中获取flash的内容

在Flask中，get\_flashed\_messages()默认已经集成到Jinja2模板引擎中，易用性很强。下面是来自官方的一个示例：

{% with messages = get\_flashed\_messages(with\_categories=true) %}

{% if messages %}

<ul>

{% for category, message in messages %}

<li class="{{ category }}">{{ message }}</li>

{% endfor %}

</ul>

{% endif %}

{% endwith %}