1. what is web？

Web(world wide web)即全球广域网，也称万维网。它是一种基于超文本和HTTP的、全球性的、动态交互的、跨平台的分布式图形信息系统。是建立在Internet上的一种网络服务，为浏览者在Internet上查找和浏览信息提供了图形化的、易于访问的直观界面，其中的文档及超级链接将Internet上的信息节点组织成一个互为关联的网状结构。

1. how to make a web service？
2. 传统桌面软件，是C/S结构（client/server），web软件是B/S（Browser/server）。

C/S结构：软件由大的两部分构成，client端和server端，client顾名思义就是由软件的使用者操作的一部分，server也就是由软件的开发者控制的一部分。Client一般都是

B/S结构：将client放在browser上，并且是动态地放在browser上，当需要时才会将client放在用户的browser上。

1. Browser端：要将界面内容显示在browser上，并且是按需显示，这就说明界面文件不能事先放置在用户磁盘，只能是用户需要的时候，再将界面文件传给用户。那界面文件该是什么样的？桌面软件的界面文件可以是C语言文件、java语言文件、python语言文件等等，几乎所有的编程语言都可以实现GUI编程，但是，这些文件要在客户机上运行，先不说运行环境，首先文件得先下载到客户机上，这就不符合web的定义。那web的界面文件又该如何写？需求产生动力，html（HypeTextMarkupLanguage）问世！html是一种标记语言，所谓标记语言即通过标记符号来标记要显示以及如何显示内容。HypeText：超文本，即html文件中不止可以包含文本还可以包含图片、音乐、程序、链接等非文本元素。

接下来问题又来了，给你一个html文件，你知道<head><title><body><p><a>等等标记符号都代表什么。可是计算机不知道你的这些符号是什么？自然也就无法达到你想要的显示结果。怎么办？这时候browser就该起作用了，browser中会有html解析引擎，用来解析html文件，正确使用各个标记来显示html文件中的内容。这样显示的内容也就完成了。

现在，我们可以使用html做几个显示页面，然后在浏览器中访问。但是，此时你做的html网页只能通过浏览器打开，不能通过浏览器访问。这两者的区别就是：打开是以文件为主，就像打开一个word文档一样，html文件需要浏览器才能打开，打开之后再浏览器的地址栏显示的是html文件的绝对路径；访问是以浏览器为主，主动去获取html，浏览器地址栏上显示的是html文件所在计算机的ip或域名，以及html文件的相对地址。为了实现访问html文件，我们需要：

1. HTTP（HyperText Transfer Protocol）：超文本传输协议，即传送html文件的协议。这是你访问的html文件所在主机向你传输html文件的方法。http是一个客户端和服务器端请求和应答的标准（TCP）
2. Server：http只是一个传输html文件的方法，它将浏览器的访问/请求传递给html文件所在的主机。那主机如何去处理/应答这个请求？这就是服务器的事。首先一台拥有ip地址的主机上有很多服务，mail service、mysql service、ssh service、web service等等，这就出现了一个问题：这个浏览器请求如何正确地传给web service，而不是传给其他的服务，好让web service处理这个请求？这就出现了端口的概念，一个服务对应一个端口，开启服务之后，相应的端口也将被开启。这样在外部请求中只需加入一个端口号，请求就可以准确到达你想让它到达的服务。Web服务默认的是80端口，如果你在浏览器地址栏中不写端口号，它会默认将请求使用http协议传输到对应主机的80端口。
3. How to make web service？

有了端口的概念，现在我们需要在service实现响应browser请求的方法。

1>、使用web框架，使用基于python语言的aihttp异步框架。

A、安装aiohttp及它依赖的模块。

B、启动一个web服务：

import asyncio

from aiohttp import web

@asyncio.coroutine

def index(request):

yield from asyncio.sleep(0.5)

return web.Response(body = b"<h1>index</h1>",content\_type = "text/html")

@asyncio.coroutine

def hello(request):

yield from asyncio.sleep(0.5)

text = "<h1>hello,%s!</h1>" % request.match\_info["name"]

return web.Response(body = text.encode("utf-8"),content\_type = "text/html")

@asyncio.coroutine

def init(loop):

app = web.Application(loop = loop)

app.router.add\_route("GET","/",index)

app.router.add\_route("GET","/hello/{name}",hello)

srv = yield from loop.create\_server(app.make\_handler(),"127.0.0.1",8000)

print("server start at http://127.0.0.1:8000...")

return srv

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(init(loop))

loop.run\_forever()

asyncio.get\_event\_loop()：创建一个循环消息队列loop，将coroutine放到loop中即可实现异步IO

init()：是loop中放置的coroutine

loop.run\_forever()：让loop一直处于运行状态。

Index()和hello()：是URL处理函数，它们的url请求方式均是GET

现在在浏览器中输入：<http://127.0.0.1:8000> 即可得到<h1>index</h1>所表示的内容，这是一句html标记，会在浏览器中自动解析。

但是，这个html也页面过于简单，我们根本不会使用如此简单的界面，当然可以在body中继续添加html，但这样的代码不适合维护，并且前端和后端代码不分明。所以引入了html模板的概念。

Python中可以使用jinja2模板，而java中使用的html模板是jsp。

C、安装html模板jinja2，并使用jinja2模板。

问题1：模板就是一个规则，写html文件的规则，重点是aiohttp框架并没有兼容jinja2，换言之就是aiohttp框架不懂这个规则，无法识别jinja2模板写的html页面，为了让aiohttp可以理解jinja2页面，需要为app初始化jinja2，使之可以识别jinja2模板写的html页面。

def init\_jinja2(app,\*\*kw):

print("init jinja2...")

options = dict(

autoescape = kw.get("autoescape",True),

block\_start\_string = kw.get("block\_start\_string","{%"),

block\_end\_string = kw.get("block\_end\_string","%}"),

variable\_start\_string = kw.get("variable\_start\_string","{{"),

variable\_end\_string = kw.get("variable\_end\_string","}}"),

auto\_reload = kw.get("auto\_reload",True)

)

path = kw.get("path",None)

if path is None:

path = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)),"templates")

print("template path :%s" % path)

env = Environment(loader = FileSystemLoader(path),\*\*options)

filters = kw.get("filters",None)

app["\_\_templating\_\_"] = env

@asyncio.coroutine

def init(loop):

app = web.Application(loop = loop)

init\_jinja2(app)

app.router.add\_route("GET","/",index)

app.router.add\_route("GET","/hello/{name}",hello)

srv = yield from loop.create\_server(app.make\_handler(),"127.0.0.1",8000)

print("server start at http://127.0.0.1:8000...")

return srv

其中init\_jinja2是用来初始化jinja2的，path是template存放的路径，未指定时使用默认的路径：当前路径下的templates文件夹中。

但是，此时我们的URL处理函数该如何写，才能将jinja2模板写的html页面完整地回复给browser？

首先在当前目录下，新建文件夹：templates，写一个jinja2模板：index.html

<html>

<head>

<meta charset = "utf-8"/>

<title>Home</title>

</head>

<body>

<h1>index</h1>

<p>welcome,{{name}}</p>

</body>

</html>

修改URL处理函数，index():

@asyncio.coroutine

def index(request,app):

yield from asyncio.sleep(0.5)

r = {"\_\_template\_\_":"index.html","name":"fured"}

return web.Response(body = app["\_\_templating\_\_"].get\_template(r.get("\_\_template\_\_")).render(\*\*r).encode("utf-8"),content\_type = "text/html;charset = utf-8")

运行时出错：index函数没有app这个参数，所以运行出错。自然而然，想到为index函数增加参数app，可行不通，因为aiohttp框架add\_route传入的url处理函数只能有一个request参数，要想传入其它形式，可以重写aiohttp框架的RequestHandler类来实现，add\_route时传入ReauestHandler对象来进行url处理。但是，我们还是直接使用aiohttp框架，所以这里将引入一个新的概念，即aiohttp框架中的middlewares：拦截/过滤器，这个东西是在server和browser中间的，所有request和response都会经过middlewares，所以可以在middlewares中添加处理response的逻辑：response\_factory。

@asyncio.coroutine

def response\_factory(app,handler):

@asyncio.coroutine

def response(request)：

print("Response handler...")

r = yield from handler(request)

if isinstance(r,web.StreamResponse):

return r

if isinstance(r,bytes):

resp = web.Response(body = r)

resp.content\_type = "application/octet-stream"

return resp

if isinstance(r,str):

if r.startswith("redirect:"):

return web.HTTPFound(r[9:])

resp = web.Response(body = r.encode("utf-8"))

resp.content\_type = "text/html;charset=utf-8"

return resp

if isinstance(r,dict):

#pdb.set\_trace()

template = r.get("\_\_template\_\_")

if template is None:

resp = web.Response(body = json.dumps(r,ensure\_ascii = False,default = lambda o : o.\_\_dict\_\_).encode("utf-8"))

resp.content\_type = "application/json;charset=utf-8"

return resp

else:

resp = web.Response(body= app["\_\_templating\_\_"].get\_template(template).render(\*\*r).encode("utf-8"))

resp.content\_type = "text/html;charset=utf-8"

return resp

if isinstance(r,int) and r >=100 and r < 600:

return web.Response(r)

if isinstance(r,tuple) and len(r) == 2:

t,m = r

if isinstance(r,int) and t >=100 and t < 600:

return web.Response(t.str(m))

#default return

resp = web.Response(body=str(r).encode("utf-8"))

resp.content\_type = "text/plain;charset=utf-8"

return resp

return response

根据url处理函数返回的数据类型来决定以什么样的数据响应browser的请求。

在init(loop)中修改如下：

app = web.Application(loop = loop,middlewares = [response\_factory])

在index（）中修改如下：

return r

现在，我们就可以使用jinja2模板来编写html文件，并且可以传入变量的值，如index.html中就传入了“name”变量。

当然，jinja2模板支持的功能远远不止这些，如：jinja2支持“继承”，即子模版继承父模板的一些内容，我们可以将一些页面的头部、尾部等一些通用的元素放在父模板中，子模版只需继承即可，不需重写一遍html代码。

单纯的Html页面即使编写地再好，还是不够现在对美的需求。我们需要引入css和js去渲染我们的html页面，让我们的html页面更加完美。

D、css和js的使用

Browser发出request来请求html页，css、js作为html文件的附加资料，也会发给browser。

增加页面：blog.html，它继承自\_\_base\_\_.html

{% extends '\_\_base\_\_.html' %}

{% block title %}日志{% endblock %}

{% block beforehead %}

<script>

</script>

{% endblock %}

{% block content %}

<div class="uk-width-medium-3-4">

<article class="uk-article">

<h2><a href="/blog/{{ blog["id"] }}">{{ blog["name"] }}</a></h2>

<p class="uk-article-meta">发表于{{ blog["created\_at"] }}</p>

<p>{{ blog["summary"] }}</p>

<p><a href="/blog/{{ blog["id"] }}">继续阅读 <i class="uk-icon-angle-double-right"></i></a></p>

</article>

<hr class="uk-article-divider">

</div>

<div class="uk-width-medium-1-4">

<div class="uk-panel uk-panel-header">

<h3 class="uk-panel-title">友情链接</h3>

<ul class="uk-list uk-list-line">

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="http://www.liaoxuefeng.com/category/0013738748415562fee26e070fa4664ad926c8e30146c67000">编程</a></li>

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="http://www.liaoxuefeng.com/category/0013738748248885ddf38d8cd1b4803aa74bcda32f853fd000">读书</a></li>

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="http://www.liaoxuefeng.com/wiki/001374738125095c955c1e6d8bb493182103fac9270762a000">Python教程</a></li>

<li><i class="uk-icon-thumbs-o-up"></i> <a target="\_blank" href="http://www.liaoxuefeng.com/wiki/0013739516305929606dd18361248578c67b8067c8c017b000">Git教程</a></li>

</ul>

</div>

</div>

{% endblock %}

将所需的css和js文件放在static文件夹下，app会自动加载css和js文件，前提是我们把css和js文件的路径告知了app：

CssJsPath = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)),”static”)

app.router.add\_static(“/static/”,CssJsPath)

之后就可以在\_\_base\_\_.html中使用相对路径调用css和js文件。

增加/blog处理函数：blog（）

@asyncio.coroutine

def blog(request):

blog = {"id":1,"name":"test","summary":"you are so beautiful in white.boom,boom,boom.I don't know.what happen!","created\_at":time.time()}

return {"\_\_template\_\_":"blog.html","blog":blog}

到这里之后，就可以使用python、html、css、js搭建一个完整的静态网站，之所以说是静态，并不是说网页内容是静止的，而是说网页中的内容是不可变的，正如我们的blog页面的id、name、summary、created\_at都是我们直接写入的静态数据，要想改变它们，必须直接修改blog函数，这样的设计完全满足不了我们的需求，更别说维护了。所以我们需要动态网站，也就是有数据库的网站，将所有数据都放在数据库中，在server中调用数据库中的数据，然后显示在网页上。

E、使用数据库

数据库server 软件有很多，我们选择开源、成熟的mysql server作为我们的数据库。

a、安装mysql for Ubuntu：可参考百度网盘

创建测试库：testdb

为testdb创建关系表：users、blogs、users并添加测试用的数据:db-test-script.sql

b、使用python访问mysql数据库：对于同步io操作下访问mysql数据库，可以使用python模块：MySQLdb去实现访问与读写，但我们使用的web框架是aiohttp是一个异步io，python中一处异步，就要处处异步。所以我们要使用异步io下的python模块：aiomysql去访问数据库。

c、安装aiomysql模块并连接测试数据库:mysql-connect-aio.py

import asyncio

from aiomysql import create\_pool,DictCursor

loop = asyncio.get\_event\_loop()

@asyncio.coroutine

def go():

pool = yield from create\_pool(host='127.0.0.1', port=3306,

user='user-test', password='test-password',

db='testdb', loop=loop)

with (yield from pool) as conn:

cur = yield from conn.cursor(DictCursor)

yield from cur.execute("SELECT \* FROM users")

value = yield from cur.fetchall()

print(value)

print("\n")

loop.run\_until\_complete(go())

现在我们就可以正常访问testdb数据库了，通过cur.execute()执行sql语句来实现操作数据库。

但是，web需要和数据库交互的内容很多，每次交互都去执行sql语句，都要重写这段代码，这样的状况是不允许的，所以ORM就是来解决这一问题的。

d、ORM

ORM全称：Object Relationship Mapping即对象关系映射，python是面向对象的高级语言，python下一切皆对象。Mysql数据库是关系型数据库，没有对象一说。要是能像操作对象一样操作数据库的话，那就方便多了！所以我们通过orm将object和Relationship之间做一个映射，这样我们就可以像操作对象的属性和方法那样去操作一张表。

如何做映射？

关系型数据库的基础就是关系表即数据表，所以我们将一张表映射成一个类，将表的列/字段映射成类的属性，而每一条数据就是一个类的实例即类的对象。

字段数据类型也需要映射成类

如何通过对象操作数据表？

数据表最基本的操作就是：insert、delete、update、select，而这些方法都是在表的层面上进行的，所以将他们映射成类的方法而不是对象的方法。

创建连接池：create\_pool(),注：\_\_pool是一个全局变量。

@asyncio.coroutine

def create\_pool(loop, \*\*kw):

logging.info('create database connection pool...')

global \_\_pool

\_\_pool = yield from aiomysql.create\_pool(

host=kw.get('host', 'localhost'),

port=kw.get('port', 3306),

user=kw['user'],

password=kw['password'],

db=kw['db'],

charset=kw.get('charset', 'utf8'),

autocommit=kw.get('autocommit', True),

maxsize=kw.get('maxsize', 10),

minsize=kw.get('minsize', 1),

loop=loop

)

映射字段数据类型：

class Field(object):

def \_\_init\_\_(self, name, column\_type, primary\_key, default):

self.name = name

self.column\_type = column\_type

self.primary\_key = primary\_key

self.default = default

def \_\_str\_\_(self):

return '<%s, %s:%s>' % (self.\_\_class\_\_.\_\_name\_\_, self.column\_type, self.name)

class StringField(Field):

def \_\_init\_\_(self, name=None, primary\_key=False, default=None, ddl='varchar(100)'):

super().\_\_init\_\_(name, ddl, primary\_key, default)

class BooleanField(Field):

def \_\_init\_\_(self, name=None, default=False):

super().\_\_init\_\_(name, 'boolean', False, default)

class IntegerField(Field):

def \_\_init\_\_(self, name=None, primary\_key=False, default=0):

super().\_\_init\_\_(name, 'bigint', primary\_key, default)

class FloatField(Field):

def \_\_init\_\_(self, name=None, primary\_key=False, default=0.0):

super().\_\_init\_\_(name, 'real', primary\_key, default)

class TextField(Field):

def \_\_init\_\_(self, name=None, default=None):

super().\_\_init\_\_(name, 'text', False, default)

映射表：我们将表映射成类，不同的表是不同的类，但表的许多操作都是相同的，所以我们将这些操作写成一个父类，然后表映射的类去继承这个父类：Model。使用元类：ModelMetaclass去读取子类（表）的映射信息。

class ModelMetaclass(type):

def \_\_new\_\_(cls, name, bases, attrs):

if name=='Model':

return type.\_\_new\_\_(cls, name, bases, attrs)

tableName = attrs.get('\_\_table\_\_', None) or name

logging.info('found model: %s (table: %s)' % (name, tableName))

mappings = dict()

fields = []

primaryKey = None

for k, v in attrs.items():

if isinstance(v, Field):

logging.info(' found mapping: %s ==> %s' % (k, v))

mappings[k] = v

if v.primary\_key:

# 找到主键:

if primaryKey:

raise StandardError('Duplicate primary key for field: %s' % k)

primaryKey = k

else:

fields.append(k)

if not primaryKey:

raise StandardError('Primary key not found.')

for k in mappings.keys():

attrs.pop(k)

escaped\_fields = list(map(lambda f: '`%s`' % f, fields))

attrs['\_\_mappings\_\_'] = mappings #保存属性和列的映射关系

attrs['\_\_table\_\_'] = tableName

attrs['\_\_primary\_key\_\_'] = primaryKey #主键的属性名

attrs['\_\_fields\_\_'] = fields #除主键外的属性名

attrs['\_\_select\_\_'] = 'select `%s`, %s from `%s`' % (primaryKey, ', '.join(escaped\_fields), tableName)

attrs['\_\_insert\_\_'] = 'insert into `%s` (%s, `%s`) values (%s)' % (tableName, ', '.join(escaped\_fields), primaryKey, create\_args\_string(len(escaped\_fields) + 1))

attrs['\_\_update\_\_'] = 'update `%s` set %s where `%s`=?' % (tableName, ', '.join(map(lambda f: '`%s`=?' % (mappings.get(f).name or f), fields)), primaryKey)

attrs['\_\_delete\_\_'] = 'delete from `%s` where `%s`=?' % (tableName, primaryKey)

return type.\_\_new\_\_(cls, name, bases, attrs)

在Model中实现了三个类方法和三个对象方法

类方法：findAll()：根据WHERE条件查询表的数据

findNumber():根据WHERE条件查找，但返回的是整数，适用于select count(\*)类型的SQL

find()：根据主键查找

对象方法：save()：将对象（一行数据）写入表中

update()：修改对象的属性（即修改某一行的数据）

remove()：根据主键删除对象（即删除某一行）

至此，我们就可以使用数据库了。

F、browser向service提交数据

我们现在实现了，响应browser发出的html请求，并且可以在html中添加数据库的数据。有时我们的用户，需要向service提交一些东西，比如注册和登陆页。接下来就看如何实现注册和登陆。

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。URL全称是资源描述符，我们可以这样认为：一个URL地址，它用于描述一个网络上的资源，而HTTP中的GET，POST，PUT，DELETE就对应着对这个资源的查，改，增，删4个操作。到这里，大家应该有个大概的了解了，GET一般用于获取/查询资源信息，而POST一般用于更新资源信息。

GET:用于信息获取，是安全的（仅仅进行资源的获取，不去修改资源）、幂等的（对同一URL多次请求返回的内容相同）。当然非要使用GET去修改资源也是可以的，不过这样不够安全。因为GET是将参数直接附在URL后面发给service的。

POST：根据HTTP规范，POST表示可能修改变服务器上的资源的请求。

1.很多人贪方便，更新资源时用了GET，因为用POST必须要到FORM（表单），这样会麻烦一点。

2.对资源的增，删，改，查操作，其实都可以通过GET/POST完成，不需要用到PUT和DELETE。

3.另外一个是，早期的Web MVC框架设计者们并没有有意识地将URL当作抽象的资源来看待和设计，所以导致一个比较严重的问题是传统的Web MVC框架基本上都只支持GET和POST两种HTTP方法，而不支持PUT和DELETE方法。

GET和POST的区别：

1.GET请求的数据会附在URL之后（就是把数据放置在HTTP协议头中），以?分割URL和传输数据，参数之间以&相连，如：login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0%E5%A5%BD。如果数据是英文字母/数字，原样发送，如果是空格，转换为+，如果是中文/其他字符，则直接把字符串用BASE64加密，得出如：%E4%BD%A0%E5%A5%BD，其中％XX中的XX为该符号以16进制表示的ASCII。

　　POST把提交的数据则放置在是HTTP包的包体中。

2."GET方式提交的数据最多只能是1024字节，理论上POST没有限制，可传较大量的数据，IIS4中最大为80KB，IIS5中为100KB"？？！

a、编写注册页面：register.html

<form id="vm" v-on="submit:submit" class="uk-form uk-form-stacked">用来提交表单的，这里使用vue.js来提交表单，当点击“注册”按钮时，会触发v-on="submit:submit"，发出URL=/api/users的POST请求

<script>

function validateEmail(email) {

var re = /^[a-z0-9\.\-\\_]+\@[a-z0-9\-\\_]+(\.[a-z0-9\-\\_]+){1,4}$/;

return re.test(email.toLowerCase());

}

$(function () {

var vm = new Vue({

el: '#vm',

data: {

name: '',

email: '',

password1: '',

password2: ''

},

methods: {

submit: function (event) {

$form.postJSON('/api/users', {

name: this.name.trim(),

email: email,

passwd: CryptoJS.SHA1(email + ':' + this.password1).toString()

}, function (err, r) {

if (err) {

return $form.showFormError(err);

}

return location.assign('/');

});

}

}

});

$('#vm').show();

});</script>

b、添加注册页UPL处理函数：register(),URL:/register

@asyncio.coroutine

def register(request):

return {"\_\_template\_\_":"register.html"}

c、提交表单，当在register.html页中填好注册的表单后，需要将这个表单的内容发给service。向service发送数据，就是browser发出一个URL请求，并将数据信息包含在发出的请求包中，如果使用GET则是将数据放在URL后，我们使用POST方法，将数据放在http包的包体中。

当提交表单时，会发出一个/api/user的URL请求，http包将包含这个request和数据等信息一起发送给service。

添加/api/user 处理函数：register\_users()

问题1：注册时提交的数据是以http数据包的形式给server的，我们如何从包中取出数据？

1. aihttp将所有的http数据包中的数据封装在request对象中，并且request对象提供了方法来获取数据，request.text(),这时的数据类型是json，然后再用json模块的loads方法，将数据转化为dict（可以不用，因为json的dict数据类型和python的dict类型相同），并将数据写入数据库中的表users中.

在init(loop)中创建连接池，在table.py中创建User类

yield from oorm.create\_pool(loop = loop,user = "user-test",password = "test-password",db = "testdb")

register\_users():

@asyncio.coroutine

def register\_users(request):

body = yield from request.text()

kw = json.loads(body)

print(kw["name"],kw["email"],kw["passwd"])

users = yield from User.findAll("email = ?",[kw["email"]])

if len(users) >0:

return "register failed: email is already in use"

uid = next\_id()

sha1\_passwd = "%s:%s" % (uid,kw["passwd"])

user = User(id = uid,name = kw["name"].strip(),email = kw["email"],password = hashlib.sha1(sha1\_passwd.encode("utf-8")).hexdigest(),image = "/path/image/%s" % kw["name"].strip())

yield from user.save()

提交后，可在数据库的users表中看到增加了一行数据。

（2）使用middlewares拦截器来获取数据，提前获取处理请求中的数据。

|  |
| --- |
| @asyncio.coroutine |
|  | def data\_factory(app, handler): |
|  | @asyncio.coroutine |
|  | def parse\_data(request): |
|  | if request.method == 'POST': |
|  | if request.content\_type.startswith('application/json'): |
|  | request.\_\_data\_\_ = yield from request.json() |
|  | logging.info('request json: %s' % str(request.\_\_data\_\_)) |
|  | elif request.content\_type.startswith('application/x-www-form-urlencoded'): |
|  | request.\_\_data\_\_ = yield from request.post() |
|  | logging.info('request form: %s' % str(request.\_\_data\_\_)) |
|  | return (yield from handler(request)) |
|  | return parse\_data |

问题2：数据的格式又是什么样的？

数据是以json的格式发过来的，json数据格式是一种可在不同语言中使用的数据格式，python中json模块，可用于处理这种数据格式。

G、设置cookie

以上代码中，/api/users的URL处理函数中，没有返回值即没有响应browser，我们可以给browser返回cookie，web.Response对象有set\_cookie函数可用于设置cookie。

#make session cookie

r = web.Response()

r.set\_cookie(COOKIE\_NAME,usercookie(user,864000),max\_age = 86400,httponly=True)

user.passwd = "\*\*\*\*\*\*"

r.content\_type = "application/json"

r.body = json.dumps(user,ensure\_ascii=False).encode("utf-8")

return r

当register.html收到cookie后，会发出location.assign(“/”)的请求，browser会将cookie打包在request中一起发出去，server可以使用：request.cookies.get(COOKIE\_NAME)得到request中的cookie信息，然后响应这个请求，在response中添加cookie中的用户信息。