# 编写web框架总结

**app.py 里的init()部分**

async def init(loop):

await orm.create\_pool(loop=loop, host='127.0.0.1', port=3306, user='www-data', password='password', db='awesome')

app = web.Application(middlewares=[

logger\_factory, response\_factory

])

# 出错：DeprecationWarning: loop argument is deprecated

# 改：把web.Application(loop=loop,里的loop=loop删掉

init\_jinja2(app, filters=dict(datetime=datetime\_filter))

**add\_routes(app, 'handlers')**

add\_static(app)

runner = web.AppRunner(app)

await runner.setup()

site = web.TCPSite(runner, '127.0.0.1', 9000)

logging.info('server started at http://127.0.0.1:9000...')

await site.start()

# 以前的写法

# 出错:

# DeprecationWarning: Application.make\_handler(...) is deprecated, use AppRunner API instead

# srv = await loop.create\_server(app.make\_handler(), '127.0.0.1', 9000)

# logging.info('server started at http://127.0.0.1:9000...')

# return srv

loop = asyncio.get\_event\_loop()

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

loop.run\_forever()

**对比aiohttp的应用例子：**

import asyncio

from aiohttp import web

async def index(request):

await asyncio.sleep(0.5)

return web.Response(body=b'<h1>Index</h1>')

async def hello(request):

await asyncio.sleep(0.5)

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

return web.Response(body=text.encode('utf-8'))

async def init(loop):

app = web.Application(loop=loop)

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

# 浏览器输入<http://127.0.0.1:8000/> 执行index()函数

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

# 浏览器输入<http://127.0.0.1:8000/hello/name> 则执行hello()函数

srv = await loop.create\_server(app.make\_handler(), '127.0.0.1', 8000)

print('Server started at http://...')

return srv

loop = asyncio.get\_event\_loop()

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

loop.run\_forever()

**看下框架的调用代码：**

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

init\_jinja2(app, filters=dict(datetime=datetime\_filter))

add\_routes(app, 'handlers')

add\_static(app)

**1、使用web.Application类创建aiohttp server——app，其中loop为Eventloop用来处理HTTP请求，middlewares为中间件，在这里用来记录日志并处理handler返回的数据为web.response对象，这里看下response\_factory的代码：**

async def response\_factory(app, handler):

async def response(request):

logging.info('Response handler...')

#获取handler的返回值，根据返回值的不同类型进行处理

r = await handler(request)

print(type(r))

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):

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

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

# default:

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

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

return resp

return response

**2、使用jinjia2模板来构建前端页面，这里我们暂时没有用到**

**3、注册处理url的handler，aiohttp中的add\_route函数进行注册，我们这里使用add\_routes对'handlers'模块的handler进行批量注册**

def add\_route(app, fn):

method = getattr(fn, '\_\_method\_\_', None)

path = getattr(fn, '\_\_route\_\_', None)

if path is None or method is None:

raise ValueError('@get or @post not defined in %s.' % str(fn))

if not asyncio.iscoroutinefunction(fn) and not inspect.isgeneratorfunction(fn):

fn = asyncio.coroutine(fn)

logging.info('add route %s %s => %s(%s)' % (method, path, fn.\_\_name\_\_, ', '.join(inspect.signature(fn).parameters.keys())))

**app.router.add\_route(method, path,** **RequestHandler(app, fn))**

**这里出现了一个RequestHandler类，它具有call魔术方法，所以可以像调用函数一样调用其实例，这里****RequestHandler类主要是对handler进行封装，获取request中传入的参数并传入handler中。**

**RequestHandler中**

**try:**

**r = await self.\_func(\*\*kw)**

**return r**

**\*\*kw为request中传入的参数，其中****handles.index等为****RequestHandler(app, fn)中的fn, RequestHandler(app, fn)返回经过处理kw后执行的函数handles.index等**

**最后一步，把很多次add\_route()注册的调用：**

**add\_route(app,** **handles.index)**

**add\_route(app, handles.blog)**

**add\_route(app, handles.create\_comment)**

**...**

**变成自动扫描：**

**# 自动把handler模块的所有符合条件的函数注册了:**

**add\_routes(app, 'handlers')**

def add\_routes(app, module\_name):

#找到'.'则返回其所在位置，否则返回-1

n = module\_name.rfind('.')

if n == (-1):

#mod为包含module\_name模块中全部属性和方法的list

mod = \_\_import\_\_(module\_name, globals(), locals())

else:

name = module\_name[n+1:]

mod = getattr(\_\_import\_\_(module\_name[:n], globals(), locals(), [name]), name)

for attr in dir(mod):

#检查handler是否被@get或@post装饰

if attr.startswith('\_'):

continue

fn = getattr(mod, attr)

if callable(fn):

method = getattr(fn, '\_\_method\_\_', None)

path = getattr(fn, '\_\_route\_\_', None)

if method and path:

add\_route(app, fn)

**装饰器**

def get(path):

'''

Define decorator @get('/path')

'''

def decorator(func):

@functools.wraps(func)

def wrapper(\*args, \*\*kw):

return func(\*args, \*\*kw)

wrapper.\_\_method\_\_ = 'GET'

wrapper.\_\_route\_\_ = path

return wrapper

return decorator

@get('/blog')

async def handler\_url\_blog(request):

body='<h1>Awesome: /blog</h1>'

return body

**这里handler\_url\_blog(request)相当于handler\_url\_blog(request)=** **get('****/blog')(****handler\_url\_blog(request))。首先执行get('/blog')，返回的是decorator函数，wrapper.\_\_route\_\_ =/blog，再调用返回的函数，参数是handler\_url\_blog函数，返回值最终是wrapper函数。**