WSGI

2024年8月11日　　21:51

WSGI

Web Server Gateway Interface　Web服务器的网关接口协议

是一个协议

作用：

uWSGI 是服务器，webpy是框架（是一个半成品）

如果没有框架，你客户端与服务器进行通信，你就需要自己去拆包，封包，所有的事情你都需要考虑到

而框架，是别人写好的代码，你拿来直接可以使用的，也可以进行再次加工，只需要调用接口，知道怎么用这个接口就行。而不需要知道框架内部具体是的逻辑是什么。只需要关系你自己的实际业务逻辑即可

这次项目使用的是webpy框架，他内部其实就是实现了一个WSGI的协议

python常见的web框架有很多 Django、Flask、tornado/sanic/fastapi等

而webpy自己也是能够部署服务器的功能的。

那我们为什么还要使用uWSGI来部署服务器呢？
原因是webpy是轻量级的服务器，并发量比较低。无法解决高并发的问题

这样就产生了一个问题，不同的框架之间，与uWSGI进行通信，如果使用不同的协议的话，那么服务器就需要实现兼容不同框架的协议。这对服务器来说是非常影响效率的。

那么PEP规范就制定了一个协议，不同的框架要想和 uWSGI服务器进行通信，就必须用WSGI协议

WSGI的三大组件

Application、Server、Middleware
满足三大组件的要求，才可以应用这个协议

（1）Application(应用框架)

uWSGI服务器必须提供两个参数，webpy框架必须接收两个参数

application()函数就是符合WSGI标准的一个HTTP处理函数，它接收两个参数

是一个可重复调用的对象（函数或类）

如果是**类**的话，想要进行调用，就必须实现 __call__ 方法

而且WSGI规定，你的框架，无论是函数，还是类，都必须接收**两个参数：** environ 和 start_response

environ
web服务器解析的http协议的一些信息，类型为dict类型，比如请求方法、请求url等信息构成的一个dict对象

存储一些环境变量，例如客户端发送一个HTTP的post请求，由服务器解析之后，传给这个参数

那么我的框架里就会存这个HTTP的一些信息（IP，环境变量之类的内容）

satrt_response:一个发送HTTP响应的函数。start_response()函数接收两个参数，一个是HTTP响应码，一个是一组list表示的HTTP Header，每个Header用一个包含两个str的tuple表示。

这个函数是由服务器给框架提供的，当框架处理完之后，会调用服务器的函数，给服务器传入一内容

```
#-*- encoding:utf-8 -*-

from wsgiref.simple_server import make_server
HELLO_WORLD = b"Hello World!\n"
# 基本的WSGI协议的实现
# 函数
def simple_app(environ, start_response):
    # 打印请求中的参数，得到请求的url，这样就可以根据信息返回内容;
    print(environ.get('PATH_INFO'))
    status = '200 OK'
    response_headers = [('Content-type','text/plain')]
    # 环境变量中，一些参数的使用
    if environ.get('PATH_INFO') == '/index':
        start_response(status, response_headers)
        return [b"this is index page"]
    elif environ.get('PATH_INFO') == '/home':
        start_response('200 OK',[('Content-type','text/plain')])
        return [b"this is home page"]
    elif environ.get('PATH_INFO') == '/login':
        start_response('200 OK',[('Content-type','text/plain')])
        return [b"this is login page"]
    # 什么都不写，默认有个斜杠
    elif environ.get('PATH_INFO') == '/':
        start_response('200 OK',[('Content-type','text/plain')])
        return [b'{name: weihong,age: 23}']
    else:
        start_response('404 not found',[('Content-type','text/plain')])
        return [b'404 Not Found']

# 类
class AppClass:
    def __call__ (self,environ, start_response):
        status = '200 OK'
        response_headers = [('Content-type','text/plain')]
        start_response(status, response_headers)
        return [HELLO_WORLD]

server = make_server('192.168.0.186',8080,app=simple_app)
#启动服务器
server.serve_forever();
```

使用url调用函数

```
#-*- encoding:utf-8 -*-
from wsgiref.simple_server import make_server
HELLO_WORLD = b"Hello World!\n"
# 使用url 调用函数
def login():
    return 'login'
def home():
    return 'home'
def index():
    return 'index'
def regist():
    return 'regist'
all_url = {
    '/':home,
    '/login':login,
    '/regist':regist,
    '/index':index,
}
# 基本的WSGI协议的实现
```

```python
# 函数
def simple_app(environ, start_response):
    # 打印请求中的参数，得到请求的url，这样就可以根据信息返回内容；
    print(environ.get('PATH_INFO'))
    url = environ.get('PATH_INFO')
    if url is None or url not in all_url.keys():
        start_response('404 not found',[('Content-type','text/plain')])
        return [b'404 Not Found']
    res = all_url.get(url)
    if res is None:
        start_response('404 not found',[('Content-type','text/plain')])
        return [b'404 Not Found']
    start_response('200 OK',[('Content-type','text/plain')])
    return [res().encode()]

server = make_server('192.168.0.186',8080,app=simple_app)
#启动服务器
server.serve_forever();
```

（2）Server

Web服务器，主要是实现相应信息的转换，将网络请求中的数据，按照HTTP协议将内容拿出，并按照WSGI协议的格式组装成新的数据，同时将提供的start_response传递给Application。最后接收Application返回的内容，并按照WSGI协议进行解析，最终按照HTTP协议组织好内容返回就完成了一次请求

其实就是从网络中，接收HTTP的请求包，按照HTTP协议解析数据包，并将一些内容按照WSGI协议封装好，发给Application（同时还发送了一个函数start_response，目的是为了接收Application返回的内容）。
Application将请求处理完成后，调用start_response函数，将结果返回给Server，Server按照WSGI协议，进行解析，然后再按照HTTP协议进行封装，将结果返回给请求方

Server操作的步骤如下

（1）客户端发来一个HTTP请求，拆包，根据HTTP协议内容构建environ
（2）提供一个start_response函数，用于接收HTTP STATU(HTTP 状态码) 和 HTTP HEADER（HTTP 响应头）
（3）接收Application 返回的结果
（4）按照HTTP协议，顺序写入HTTP响应头（start_response接收），HTTP响应体（Application的返回结果）
（5）把封装好的HTTP数据包返回客户端

这里我们调用库，来实现了一个服务器的功能

```python
from wsgiref.simple_server import make_server

server = make_server('192.168.0.186',8080,app=simple_app)
#启动服务器
server.serve_forever();
```

```python
def make_server(
    host, port, app, server_class=WSGIServer, handler_class=WSGIRequestHandler
):
    """Create a new WSGI server listening on `host` and `port` for `app`"""
    server = server_class((host, port), handler_class)
    server.set_app(app)
    return server
```

1、使用 make_server 初始化服务器

（1）使用(ip,port) 和 WSGIRequestHandler 来初始化一个 WSGIServer的类
WSGIServer 继承 HTTPServer 继承 TCPServer 继承 BaseServer
下面是BaseServer类的构造函数

```python
def __init__(self, server_address, RequestHandlerClass):
    """Constructor.  May be extended, do not override."""
    self.server_address = server_address
    self.RequestHandlerClass = RequestHandlerClass
    self.__is_shut_down = threading.Event()
    self.__shutdown_request = False
```

（2）使用set_app 给server对象的成员变量 application赋值

```
def set_app(self,application):
    self.application = application
```

（3）返回一个WSGIServer 类的 server对象

2、通过server对象，调用server_forever()函数，启动服务器，初始化请求消息的类

函数内部 time_out 默认 0.5，用来设置 select的超时时间，然后调用一个非阻塞处理请求的函数

所以，从 wsgiref.simple_server 库导入的这个 make_server 函数，只是我们用来测试的一个东西，它内部处理网络消息是用select来实现的，性能很低

（1）server_forever()函数 内部，如果select受到网络消息，则开始调用非阻塞处理请求的函数 _handle_request_noblock()函数

```
def serve_forever(self, poll_interval=0.5):
    self.__is_shut_down.clear()
    try:
        with _ServerSelector() as selector:
            selector.register(self, selectors.EVENT_READ)
            while not self.__shutdown_request:
                ready = selector.select(poll_interval)
                # bpo-35017: shutdown() called during select(), exit immediately.
                if self.__shutdown_request:
                    break
                if ready:
                    self._handle_request_noblock()
                self.service_actions()
    finally:
        self.__shutdown_request = False
        self.__is_shut_down.set()
```

（2）_handle_request_noblock()函数内部 尝试调用 进程请求函数 process_request()，传入的是请求内容、客户端的地址元组（ip、port）

```
def _handle_request_noblock(self):
    I assume that selector.select() has returned that the socket is
    readable before this function was called, so there should be no ri
    blocking in get_request().
    """
    try:
        request, client_address = self.get_request()
    except OSError:
        return
    if self.verify_request(request, client_address):
        try:
            self.process_request(request, client_address)
        except Exception:
            self.handle_error(request, client_address)
            self.shutdown_request(request)
        except:
            self.shutdown_request(request)
            raise
    else:
```

（3） process_request() 内部调用 finish_request()函数

```
def process_request(self, request, client_address):
    """Call finish_request.

    Overridden by ForkingMixIn and ThreadingMixIn.

    """
    self.finish_request(request, client_address)
    self.shutdown_request(request)
```

（4）finish_request() 内部完成对 RequestHandlerClass的初始化

```
def finish_request(self, request, client_address):
    """Finish one request by instantiating RequestHandlerClass."""
    self.RequestHandlerClass(request, client_address, self)
```

RequestHandlerClass是WSGIRequestHandler类型的成员变量 在make_server函数 WSGIServer类的 基类BaseServer中

WSGIRequestHandler的 基类BaseRequestHandler类中，存在构造函数，接收RequestHandlerClass的参数 这里传入的self就是server对象。

```python
def finish_request(self, request, client_address):
    """Finish one request by instantiating RequestHandlerClass."""
    self.RequestHandlerClass(request, client_address, self)
```

```python
class BaseRequestHandler:
    def __init__(self, request, client_address, server):
        self.request = request
        self.client_address = client_address
        self.server = server
        self.setup()
        try:
            self.handle()
        finally:
            self.finish()
```

（5）基类BaseRequestHandler的构造函数，尝试调用handle()函数，基类中，没实现这个函数，子类WSGIRequestHandler中实现了这个函数

```python
class WSGIRequestHandler(BaseHTTPRequestHandler):
    def handle(self):
        """Handle a single HTTP request"""
        self.raw_requestline = self.rfile.readline(65537)
        if len(self.raw_requestline) > 65536:
            self.requestline = ''
            self.request_version = ''
            self.command = ''
            self.send_error(414)
            return
        if not self.parse_request(): # An error code has been sent, just exit
            return
        handler = ServerHandler(
            self.rfile, self.wfile, self.get_stderr(), self.get_environ(),
            multithread=False,
        )
        handler.request_handler = self      # backpointer for logging
        handler.run(self.server.get_app())
```

（6）在handle()函数的最后，调用了get_app函数，而在make_server中，我们调用set_app，给成员变量application进行了赋值（我们传入的simple_app()函数），使用get_app运行了这个函数

```python
def get_app(self):
    return self.application

def set_app(self,application):
    self.application = application
```

（7）run() 函数内部执行3个函数

```python
def run(self, application):
    try:
        self.setup_environ()
        self.result = application(self.environ, self.start_response)
        self.finish_response()
    except (ConnectionAbortedError, BrokenPipeError, ConnectionResetError):
        # We expect the client to close the connection abruptly from time
        # to time.
        return
    except:
        try:
            self.handle_error()
```

（8）首先调用setup_environ()函数，设置环境变量的参数

```python
def setup_environ(self):
    """Set up the environment for one request"""
    env = self.environ = self.os_environ.copy()
    self.add_cgi_vars()
    env['wsgi.input']        = self.get_stdin()
    env['wsgi.errors']       = self.get_stderr()
    env['wsgi.version']      = self.wsgi_version
    env['wsgi.run_once']     = self.wsgi_run_once
    env['wsgi.url_scheme']   = self.get_scheme()
    env['wsgi.multithread']  = self.wsgi_multithread
    env['wsgi.multiprocess'] = self.wsgi_multiprocess

    if self.wsgi_file_wrapper is not None:
        env['wsgi.file_wrapper'] = self.wsgi_file_wrapper

    if self.origin_server and self.server_software:
        env.setdefault('SERVER_SOFTWARE',self.server_software)
```

（9）第二步，调用你传入的函数simple_app()，给这个函数传入刚刚设置的环境变量，和自己的一个函数（用于获取请求的一些状态之类的），得到返回值

```
def run(self, application):
    try:
        self.setup_environ()
        self.result = application(self.environ, self.start_response)
        self.finish_response()
    except (ConnectionAbortedError, BrokenPipeError, ConnectionResetError):
        # We expect the client to close the connection abruptly from time
        # to time.
        return
    except:
        try:
            self.handle_error()
```

（10）第三步 调用 finish_response()函数，将第二步的返回值 挨个取出来，写回到客户端

```
def finish_response(self):
    try:
        if not self.result_is_file() or not self.sendfile():
            for data in self.result:
                self.write(data)
            self.finish_content()
    except:
        # Call close() on the iterable returned by the WSGI application
        # in case of an exception.
        if hasattr(self.result, 'close'):
            self.result.close()
        raise
    else:
```

实际上的流程就是

1、初始化server对象，启动服务器，监听客户端请求

（1）调用make_server()函数，传入ip地址，端口号port，application组件中的simplp_app函数，来初始化一个server对象

（2）通过server对象，调用server_forever()函数，来通过select 循环监听 客户端的请求（timeout设为0.5，非阻塞，一直监听）

（3）一旦有客户端发来请求，就会调用非阻塞处理请求的函数 _handle_request_noblock()来处理请求

2、客户端发来请求，初始化请求处理器

（1）在非阻塞处理请求的函数 _handle_request_noblock()中，首先先通过 process_request() 内部调用finish_request()函数来初始化请求处理器（通过WSGIRequestHandler的基类BaseRequestHandler的构造函数）

（2）完成请求处理器的初始化之后，调用handle函数对请求进行处理

3、调用处理函数，完成客户端请求的处理

（1）handle函数 调用setup_environ()函数，设置环境变量的参数 environ（用来给Application组件中的simplp_app函数传值）

（2）调用Application组件中的simplp_app函数（传入的就是第一步中的environ，和一个server内部的回调函数start_response，目的是得到处理后的状态和请求的响应头），得到函数的返回值

（3）调用 finish_response()函数，将第二步的返回值 挨个取出来，写回到客户端

（3）Middleware

中间件，可以理解称对应用程序的一组装饰器，对两边都起作用的元素。

- 重写 environ，然后基于 URL，将请求对象路由给不同的应用对象
- 支持多个应用或者框架顺序地运行于同一个进程中
- 通过转发请求和相应，支持负载均衡和远程处理
- 支持对内容做后处理

在应用程序(Application)看来，它可以提供一个类start response 函数，可以像start response 函数一样接收 HTTP STATU和Headers和environ.

在服务端看来，它可以接收 2个参数，并且可以返回一个类Application 对象。

在应用程序(Application)看来，中间件就是服务器
在服务端看来，中间件就是应用程序

就是在服务器和应用程序中间添加的一些组件，可以加一些其他的功能（甚至可以有多个中间件）

例子：记录每次请求耗时的中间件

```
#-*- encoding:utf-8 -*-
from wsgiref.simple_server import make_server
import time
# 修改解释器的编码格式
import sys
reload(sys)
sys.setdefaultencoding('utf-8')

# 记录请求耗时的中间件
class ResponseTimingMiddleware(object):
    """记录请求耗时"""
    def __init__(self,app):
        self.app = app
    def __call__(self, environ, start_response):
        # 记录开始时间
        start_time = time.time()
        # 调用请求处理函数，得到返回值
        response = self.app(environ, start_response)
        # 记录请求耗时
        response_time = (time.time() - start_time)*1000
        # 设置日志
        timing_text = "\n\n\n记录请求耗时中间件输出\n\n本次请求耗时: {:.10f}ms \n\n\n".format(response_time)
        # 添加到返回值
        response.append(timing_text.encode('utf-8'))
        # 返回请求处理结果
        return response
# 使用url 调用函数
def login(req):
    print(req)
    return 'login'
def home(req):
    print(req)
    return 'home'
def index(req):
    print(req)
    return 'index'
def regist(req):
    print(req)
    return 'regist'
all_url = {
    '/':home,
    '/login':login,
    '/regist':regist,
    '/index':index,
}
# 基本的WSGI协议的实现
# 函数
def simple_app(environ, start_response):
    # 打印请求中的参数，得到请求的url，这样就可以根据信息返回内容;
    print(environ.get('PATH_INFO'))
    url = environ.get('PATH_INFO')   # 附加的路径信息，由浏览器发出
    params = environ.get('QUERY_STRING') # 请求URL的 “?” 后面的部分
    if url is None or url not in all_url.keys():
        start_response('404 not found',[('Content-type','text/plain;charset=utf-8')])
        return [b'404 Not Found']
    res = all_url.get(url)
    if res is None:
        start_response('404 not found',[('Content-type','text/plain;charset=utf-8')])
        return [b'404 Not Found']

    return_body = []
    return_body.append(res(params))
    # 将环境变量也返回
    for k,v in environ.items():
        return_body.append("{} : {}".format(k,v))
    start_response('200 OK',[('Content-type','text/plain;charset=utf-8')])
    return ["\n".join(return_body).encode('utf-8')]
```

```
application = ResponseTimingMiddleware(simple_app)
server = make_server('192.168.0.186',8080,app=application)
#启动服务器
server.serve_forever();
```

协议内容

这里重点看environ 中有那些内容，这里面包含的其实就是浏览器每次请求时的一些信息

environ是一个字典，environ中需要包含CGI定义的变量，比如请求方法，POST/GEY，请求URL等，另外就是WSGI协议自己定义的变量，比如请求boby中要读取的信息等

CGI相关变量

变量	说明
REQUEST_METHOD	POST,GET等,HTTP请求的动词标识
SERVER_PROTOCOL	服务器运行的HTTP协议. 这里当是HTTP/1.0.
PATH_INFO	附加的路径信息, 由浏览器发出.
QUERY_STRING	请求URL的"?"后面的部分
CONTENT_TYPE	HTTP请求中任何Content-Type字段的内容
CONTENT_LENGTH	标准输入口的字节数.
HTTP_[变量]	其他一些变量，例如HTTP_ACCEPT，HTTP_REFERER等

WSGI相关变量

变量	说明
wsgi.version	WSGI版本,要求是元组(1,0),标识WSGI 1.0协议
wsgi.url_scheme	表示调用应用程序的URL的协议，http或https
wsgi.input	类文件对象，读取HTTP请求体字节的输入流
wsgi.errors	类文件对象，写入错误输出的输出流
wsgi.multithread	如果是多线程，则设置为True，否则为False。
wsgi.multiprocess	如果是多进程，则设置为True，否则为False。
wsgi.run_once	如果只需要运行一次，设置为True

总结：其实就是Server端（uWSGI）按照协议的内容生成这些environ字段，然后将请求信息交给Application，Application根据这些信息确认请求要处理的内容，然后返回相应的消息。