WSGI 概述

WSGI 不是服务器，也不是用于与程序交互的 API，更不是真实的代码，而只是定义的接口，其目标是在 WEB 服务器与 WEB 框架层之间提供一个通用的标准，减少之间的互操作性并形成统一的调用方式

那么这个接口到底是什么呢？

根据 WSGI 的定义，其应用 application 是可调用的对象，其参数固定为两个：

一个包含所有环境变量的 dict 对象 environ；

另一个也是可调用对象，该对象使用 HTTP 状态码和返回客户端的 HTTP 头来初始化响应，同时返回一个可迭代对象的用于组成响应负载。

我从一段具体的代码进行讲解

学过 python web 都知道用下面的这段代码就可以产生一个 web 应用

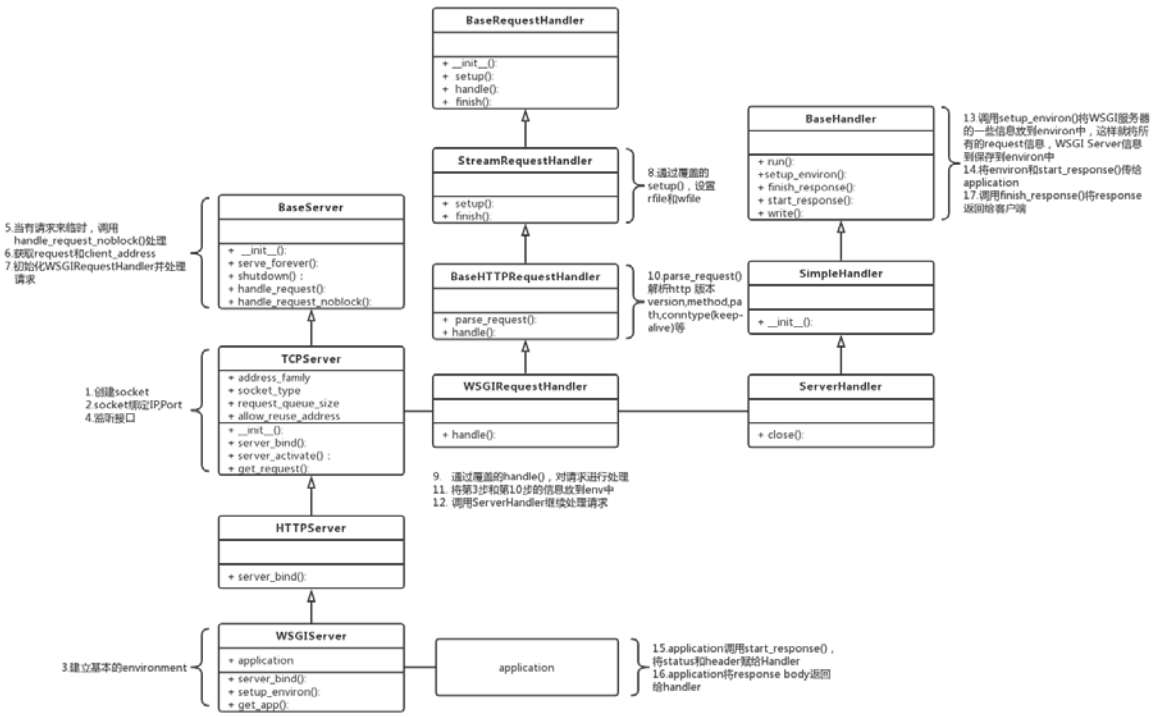
```
from wsgiref.simple_server import make_server

def application(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])
    return [b'Hello World',]

httpd = make_server('', 8000, application)
httpd.serve_forever()
print('Serving started at http://localhost:8000...')
```

当用浏览器访问 http://localhost:8000 时会在浏览器看到 Hello World 字样。

WSGI 相关类图详解




```
def make_server(host, port, app, server_class=WSGIServer,
handler_class=WSGIRequestHandler):
    server = server_class((host, port), handler_class)
    server.set_app(app)
    return server
```

wsgiref 模块是 Python 内置了一个 WSGI 服务器,其方法 make_server 返回一个包含 application 的 web server，make_server 通过传入的 server_class(这里默认是 WSGIServer) 创建一个 web server，并设置 web server 关联的 applicaion，咦，这里还有个默认参数 handler_class 是干嘛用的？别急，我们通过代码一步步分析。

Server 类

首先 make_server 会调用 WSGIServer 自 socketserver.TCPServer 继承的__init__创建实例,这过程会做这几步事情：

1.创建 socket：调用 socker.socket()方法

2.Socket 绑定 host,port： 调用 WSGIServer 类的 server_bind()

3.建立基本的 environment：调用 WSGIServer 类的 setup_environ()

4.Socket 监听端口：调用 TCPServer 类的 server_activate()，其实质也是调用 socket.listen()。

以上几步实质上是 socket 的网络编程，用来建立 TCP 连接。

注：BaseServer 里__init__(self, server_address, RequestHandlerClass)传入的 RequestHandlerClass 就代表着刚才 make_server 函数传入的参数 handler_class=WSGIRequestHandler。

```
class WSGIServer(HTTPServer):
    application = None
    def server_bind(self):
        HTTPServer.server_bind(self)
        self.setup_environ()                                    (3)

    def setup_environ(self):
    # Set up base environment
        env = self.base_environ = {}
        env['SERVER_NAME'] = self.server_name
        env['GATEWAY_INTERFACE'] = 'CGI/1.1'
        env['SERVER_PORT'] = str(self.server_port)
        env['REMOTE_HOST']=''
        env['CONTENT_LENGTH']=''
        env['SCRIPT_NAME'] = ''

class HTTPServer(socketserver.TCPServer):
    ......

class TCPServer(BaseServer):
    def __init__(self, server_address, RequestHandlerClass,
bind_and_activate=True):
        BaseServer.__init__(self, server_address, RequestHandlerClass)
        self.socket = socket.socket(self.address_family,
                                    self.socket_type)                (1)
        if bind_and_activate:
```

```
            ......
            self.server_bind()                                    (2)
            self.server_activate()                                (4)
            ......
    def server_activate(self):
        self.socket.listen(self.request_queue_size)

class BaseServer:
    def __init__(self, server_address, RequestHandlerClass):
        self.server_address = server_address
        self.RequestHandlerClass = RequestHandlerClass
```

5. 紧接着设置 WSGIServer 的 application，然后调用的 httpd.serve_forever()，它本质上是一个 select 的 I/O 模型，当有请求来临时调用_handle_request_noblock()函数来处理请求

```
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)
                if ready:
                    self._handle_request_noblock()                (5)

                self.service_actions()
    finally:
        self.__shutdown_request = False
        self.__is_shut_down.set()
```

6 _handle_request_noblock()调用 get_request()函数获得请求内容 request 和客户端 IP、Port 信息 client_address，紧接着会调用 process_request 处理请求

```
def _handle_request_noblock(self):
    try:
        request, client_address = self.get_request()              (6)
    except OSError:
        return
    if self.verify_request(request, client_address):
        try:
            self.process_request(request, client_address)
```

```
        except:
            self.handle_error(request, client_address)
            self.shutdown_request(request)
    else:
        self.shutdown_request(request)
```

7.process_request 会通过调用 finish_request()会初始化一个 RequestHandlerClass 的实例，实质是 make_server()函数传入的 handler_class=WSGIRequestHandler 的作用；这里是调用继承自 BaseRequestHandler 的__init__()方法。

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

def finish_request(self, request, client_address):
    self.RequestHandlerClass(request, client_address, self)            (7)
```

RequestHandler 类

```
class BaseRequestHandler:
    def __init__(self, request, client_address, server):
        self.request = request
        self.client_address = client_address
        self.server = server
        self.setup()                                                   (8)
        try:
            self.handle()                                              (9)
        finally:
            self.finish()
```

8.这个初始化 BaseRequestHandler 过程会调用 setup()，setup()函数中最后两行代码要注意，它们的作用是可以像读写文件那样读写 socket。

注：SocketServer.StreamRequestHandler 中对客户端发过来的数据是用 rfile 属性来处理的,rfile 是一个类 file 对象.有缓冲.可以按行分次读取;发往客户端的数据通过 wfile 属性来处理,wfile 不缓冲数据,对客户端发送的数据需一次性写入（引用自 http://a564941464.iteye.com/blog/1170464）

```
def setup(self):
    self.connection = self.request
    if self.timeout is not None:
        self.connection.settimeout(self.timeout)
    if self.disable_nagle_algorithm:
        self.connection.setsockopt(socket.IPPROTO_TCP,
                                   socket.TCP_NODELAY, True)
```

```
    self.rfile = self.connection.makefile('rb', self.rbufsize)
    self.wfile = self.connection.makefile('wb', self.wbufsize)
```

9.通过覆盖了的 handle()请求进行处理

```
def handle(self):
    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():                                    (10)
        return

    # Avoid passing the raw file object wfile, which can do partial
    # writes (Issue 24291)
    stdout = BufferedWriter(self.wfile)
    try:
        handler = ServerHandler(
            self.rfile, stdout, self.get_stderr(), self.get_environ()   (11)
        )
        handler.request_handler = self      # backpointer for logging
        handler.run(self.server.get_app())                          (12)
    finally:
        stdout.detach()
```

10.handler()会调用 parse_request()，parse_request()代码比较长，这里就不贴出来了，有兴趣的童鞋去 trace 下代码，它的大致作用是解析 http 版
本,method,path,Conntype(keep-alive）等，如果解析出错会返回 False。
如果成功，继续执行第 11 步

Handler 类

11.handler()会调用 handler = ServerHandler(
self.rfile, stdout, self.get_stderr(), self.get_environ()
),

最后一个参数是通过 get_environ()函数获取的，它会将调用 WSGIRequestHandler 的 get_environ()，将 3 步和第 10 步的信息放到 environ 中，也就是最初 application(environ, start_response)中传入的 environ。然后生成了 ServerHandler 的实例 handler

12.调用 handler.run(self.server.get_app())

self.server.get_app()是获取我们自己所写的 application。

```
def run(self, application):
    try:
        self.setup_environ()                                                    (13)
        self.result = application(self.environ, self.start_response)
        self.finish_response()
```

13.调用 setup_environ()将 WSGI 服务器的一些信息放到 environ 中，这样就将所有的 requset 信息、server 信息保存到 environ 中。

```
def setup_environ(self):
    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
```

14.将 environ 和 start_respnse()传给 application

15.application 调用 start_response()将 status，header 保存在 BaseHandler 实例中，并返回一个 write 函数

```
class BaseHandler:
    def start_response(self, status, headers,exc_info=None):
        ......
        self.status = status
        self.headers = self.headers_class(headers)
        ......
         return self.write
```

16.application 将 response body 返回赋给 result 变量

17.调用 finish_response()将 response 返回给客户端

```
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()
    finally:
        self.close()
```

我们看到 finish_response 函数体内是调用 write 函数，在发送 body 之前，要判断是否有 status 值，然后要判断 header 发出去的吗？如果没有则调用 self.send_headers()，最后再发送 response body。

```
def write(self, data):
    if not self.status:
        raise AssertionError("write() before start_response()")

    elif not self.headers_sent:
        # Before the first output, send the stored headers
        self.bytes_sent = len(data)    # make sure we know content-length
        self.send_headers()
    else:
        self.bytes_sent += len(data)
    self._write(data)
    self._flush()
```

小结

Server 类建立基本的 Socket 理解，然后将请求发送给 RequestHandle 处理
RequestHandler 类解析客户端的 Request 请求
Handler 类设置 server、request 信息，并交给 application
application 进行业务的处理，然后由 Handler 发送 response 给客户端。

WSGI application

application
必须是一个可调用对象，可调用对象可以是：

1. 函数
2. 类
3. 具有__call__方法的实例

同时返回一个可迭代对象

接受 environ 和 start_response 两个参数

符合上述三点的就是一个 WSGI 标准的 appication，因此这个 application 可以有以下几种形式

可调用函数

```
def application(environ, start_response):
    return [b'Hello World',]
```

可调用类

```
class Application:
    def__init__(self, environ, start_response):
        pass
    def__iter__(self):
        yield b'Hello World'
```

可调用实例

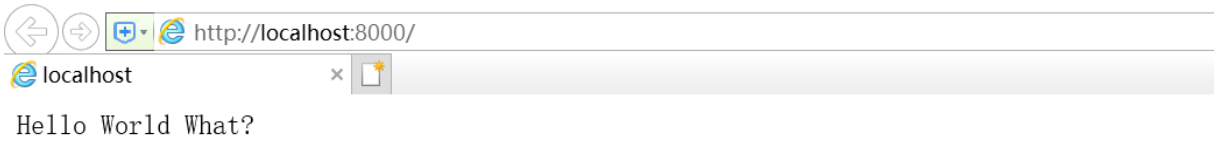
```
class ApplicationObj:
    def__call__(self, environ, start_response):
        return [b'Hello World',]
```

start_response 疑惑

WSGI 标准规定，start_respnsoe 必须返回一个可调用对象，第 15 步提到 start_response 返回了一个 wirte()函数，于是我们的 application 还可以这样写

```
def application(environ, start_response):
    write= start_response('200 OK', [('Content-Type', 'text/html')])
    write( 'Hello World!' )
  return 'What?'
```

通过浏览器访问得到下面的结果



其实 start_response 根本没有必要这样，它只要把设置 status、header 就好了，至于返回 response 的事情还是让 Server 去干吧

注：摘自 http://mp.weixin.qq.com/s/6uE_05D5FEGBycy5O28fXA