Python Web开发最难懂的WSGI协议，到底包含哪些内容？ WSGI服务器种类和性能对比

转载 hshl1214 2018-05-14 15:39:50 2039 收藏 2

分类专栏： uwsgi web server django python web测试 性能测试

http://python.jobbole.com/88653/

我想大部分Python开发者最先接触到的方向是WEB方向（因为总是有开发者希望马上给自己做个博客出来，例如我），既然是WEB，免不了接触到一些WEB框架，例如Django,Flask,Torando等等，在开发过程中，看过一些文档总会介绍生产环境和开发环境服务器的配置问题，服务器又设计web服务器和应用服务器，总而言之，我们碰到最多的，必定是这个词 — WSGI。

接下来的文章，会分为以下几个部分：

- 1.WSGI介绍
  - 1.1什么是WSGI
  - 1.2怎么实现WSGI
- 2.由Django框架分析WSGI
- 3.实际环境使用的wsgi服务器
- 4.WSGI服务器比较

开始

1 WSGI介绍

1.1 什么是WSGI

首先介绍几个关于WSGI相关的概念

WSGI：全称是**Web Server Gateway Interface**，**WSGI**不是服务器，python模块，框架，API或者任何软件，只是一种规范，描述**web server**如何与**web application**通信的规范。**server**和**application**的规范在PEP 3333中有具体描述。要实现WSGI协议，必须同时实现**web server**和**web application**，当前运行在WSGI协议之上的web框架有**Torando**,**Flask**,**Django**

uwsgi：与**WSGI**一样是一种通信协议，是**uWSGI**服务器的独占协议，用于定义传输信息的类型(type of information)，每一个uwsgi packet前4byte为传输信息类型的描述，

与**WSGI**协议是两种东西，据说该协议是**fcgi**协议的10倍快。

uWSGI：是一个**web**服务器，实现了**WSGI**协议、**uwsgi**协议、**http**协议等。

WSGI协议主要包括**server**和**application**两部分：

Python
WSGI server负责从客户端接收请求，将request转发给application，将application返回的response返回给客户端；
WSGI application接收由server转发的request，处理请求，并将处理结果返回给server。application中可以包括多个栈式的中间件(middlewares)，这些中间件需要同时实现server与application，因此可以在WSGI服务器与WSGI应用之间起调节作用：对服务器来说，中间件扮演应用程序，对应用程序来说，中间件扮演服务器。

```
1 WSGI server负责从客户端接收请求，将 request转发给 application，将 application返回的 response返回给客户端；
2 WSGI application接收由 server转发的 request，处理请求，并将处理结果返回给 server。 application中可以包括多个栈式的中间件 ( middlewares )，这些中间件需要同时实现 server与 application，因此可以在 WSGI服务器与WSGI应用之间起调节作用：对服务器来说，中间件扮演应用程序，对应用程序来说，中间件扮演服务器。
```

WSGI协议其实是定义了一种**server**与**application**解耦的规范，即可以有多个实现**WSGI server**的服务器，也可以有多个实现**WSGI application**的框架，那么就可以选择任意的server和application组合实现自己的web应用。例如**uWSGI**和**Gunicorn**都是实现了WSGI server协议的服务器，**Django**，**Flask**是实现了**WSGI application**协议的**web**框架，可以根据项目实际情况搭配使用。

以上介绍了相关的常识，接下来我们来看看如何简单实现WSGI协议。

1.2 怎么实现WSGI

上文说过，实现WSGI协议必须要有wsgi server和application，因此，我们就来实现这两个东西。

我们来看看官方WSGI使用WSGI的wsgiref模块实现的小demo

> 有关于wsgiref的快速入门可以看看这篇博客

Python
def demo_app(environ,start_response): from StringIO import StringIO stdout = StringIO() print >>stdout, "Hello world!" print >>stdout h = environ.items(); h.sort() for k,v in h: print >>stdout, k,'=', repr(v) start_response("200 OK", [('Content-Type','text/plain')]) return [stdout.getvalue()] httpd = make_server('localhost', 8002, demo_app) httpd.serve_forever() # 使用select

```
1  def demo_app ( environ , start_response ) :
2      from StringIO import StringIO
3      stdout = StringIO ( )
4      print >> stdout , "Hello world!"
5      print >> stdout
6      h = environ . items ( ) ; h . sort ( )
7      for k , v in h :
8          print >> stdout , k , '=' , repr ( v )
9      start_response ( "200 OK" , [ ( 'Content-Type' , 'text/plain' ) ] )
10     return [ stdout . getvalue ( ) ]
11
12 httpd = make_server ( 'localhost' , 8002 ,   demo_app )
```

```
13    httpd . serve_forever ( )   # 使用select
```

实现了一个application，来获取客户端的环境和回调函数两个参数，以及httpd服务端的实现，我们来看看make_server的源代码

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  def make_server (
2      host , port , app , server_class = WSGIServer , handler_class = WSGIRequestHandler
3  ) :
4    """Create a new WSGI server listening on `host` and `port` for `app`"""
5    server = server_class ( ( host , port ) , handler_class )
6    server . set_app ( app )
7    return server
```

接受一系列函数，返回一个server对象,实现还是比较简单，下面我们来看看在django中如何实现其自身的wsgi服务器的。

下面我们自己来实现一遍：

WSGI 规定每个 python 程序（Application）必须是一个可调用的对象（实现了__call__函数的方法或者类），接受两个参数 environ（WSGI 的环境信息） 和 start_response（开始响应请求的函数），并且返回 iterable。几点说明：

Python
environ 和 start_response 由 http server 提供并实现 environ 变量是包含了环境信息的字典 Application 内部在返回前调用 start_response start_response也是一个 callable，接受两个必须的参数，status（HTTP状态）和 response_headers（响应消息的头） 可调用对象要返回一个值，这个值是可迭代的。

```
1  environ 和 start _response 由 http server 提供并实现
2  environ 变量是包含了环境信息的字典
3  Application 内部在返回前调用 start_response
4  start _response也是一个 callable，接受两个必须的参数， status（ HTTP状态）和 response _headers（响应消息的头）
5  可调用对象要返回一个值，这个值是可迭代的。
```

Python
1. 可调用对象是一个函数 def application(environ, start_response): response_body = 'The request method was %s' % environ['REQUEST_METHOD'] # HTTP response code and message status = '200 OK' # 应答的头部是一个列表，每对键值都必须是一个 tuple。 response_headers = [('Content-Type', 'text/plain'), ('Content-Length', str(len(response_body)))] # 调用服务器程序提供的 start_response，填入两个参数 start_response(status, response_headers) # 返回必须是 iterable return [response_body] # 2. 可调用对象是一个类 class AppClass: """这里的可调用对象就是 AppClass 这个类，调用它就能生成可以迭代的结果。 使用方法类似于： for result in AppClass(env, start_response): do_somthing(result) """ def __init__(self, environ, start_response): self.environ = environ self.start = start_response def __iter__(self): status = '200 OK' response_headers = [('Content-type', 'text/plain')] self.start(status, response_headers) yield "Hello world!\n" # 3. 可调用对象是一个实例 class AppClass: """这里的可调用对象就是 AppClass 的实例，使用方法类似于： app = AppClass() for result in app(environ, start_response): do_somthing(result) """ def __init__(self): pass def __call__(self, environ, start_response): status = '200 OK' response_headers = [('Content-type', 'text/plain')] self.start(status, response_headers) yield "Hello world!\n"

```
1  # 1. 可调用对象是一个函数
2  def application ( environ , start_response ) :
3    response_body = 'The request method was %s' % environ [ 'REQUEST_METHOD' ]
```

```
    # HTTP response code and message
    status = '200 OK'
    # 应答的头部是一个列表，每对键值都必须是一个 tuple。
    response_headers = [ ( 'Content-Type' , 'text/plain' ) ,
                         ( 'Content-Length' , str ( len ( response_body ) ) ) ]
    # 调用服务器程序提供的 start_response，填入两个参数
    start_response ( status , response_headers )
    # 返回必须是 iterable
    return [ response_body ]

# 2. 可调用对象是一个类
class AppClass :
    """这里的可调用对象就是 AppClass 这个类，调用它就能生成可以迭代的结果。
       使用方法类似于:
       for result in AppClass(env, start_response):
           do_somthing(result)
    """
    def __init__ ( self , environ , start_response ) :
        self . environ = environ
        self . start = start_response
    def __iter__ ( self ) :
        status = '200 OK'
        response_headers = [ ( 'Content-type' , 'text/plain' ) ]
        self . start ( status , response_headers )
        yield "Hello world!\n"
# 3. 可调用对象是一个实例
class AppClass :
    """这里的可调用对象就是 AppClass 的实例，使用方法类似于:
       app = AppClass()
       for result in app(environ, start_response):
           do_somthing(result)
    """
    def __init__ ( self ) :
        pass
    def __call__ ( self , environ , start_response ) :
        status = '200 OK'
        response_headers = [ ( 'Content-type' , 'text/plain' ) ]
        self . start ( status , response_headers )
        yield "Hello world!\n"
```

服务器程序端

上面已经说过，标准要能够确切地实行，必须要求程序端和服务器端共同遵守。上面提到， envrion 和 start_response 都是服务器端提供的。下面就看看，服务器端要履行的义务。

Python
准备 environ 参数 定义 start_response 函数 调用程序端的可调用对象

```
准备 environ 参数
定义 start_response 函数
调用程序端的可调用对象
```

Python
import os, sys def run_with_cgi(application): # application 是程序端的可调用对象 # 准备 environ 参数，这是一个字典，里面的内容是一次 HTTP 请求的环境变量 environ = dict(os.environ.items()) environ['wsgi.input'] = sys.stdin environ['wsgi.errors'] = sys.stderr environ['wsgi.version'] = (1, 0) environ['wsgi.multithread'] = False environ['wsgi.multiprocess'] = True environ['wsgi.run_once'] = True environ['wsgi.url_scheme'] = 'http' headers_set = [] headers_sent = [] # 把应答的结果输出到终端 def write(data): sys.stdout.write(data) sys.stdout.flush() # 实现 start_response 函数，根据程序端传过来的 status 和 response_headers 参数， # 设置状态和头部 def start_response(status, response_headers, exc_info=None): headers_set[:] = [status, response_headers] return write # 调用客户端的可调用对象，把准备好的参数传递过去 result = application(environ, start_response) # 处理得到的结果，这里简单地把结果输出到标准输出。 try: for data in result: if data: # don't send headers until body appears write(data) finally: if hasattr(result, 'close'): result.close()

```
import os , sys
def run_with_cgi ( application ) :      # application 是程序端的可调用对象
    # 准备 environ 参数，这是一个字典，里面的内容是一次 HTTP 请求的环境变量
    environ = dict ( os . environ . items ( ) )
    environ [ 'wsgi.input' ]          = sys . stdin
    environ [ 'wsgi.errors' ]         = sys . stderr
    environ [ 'wsgi.version' ]        = ( 1 , 0 )
    environ [ 'wsgi.multithread' ]    = False
    environ [ 'wsgi.multiprocess' ] = True
    environ [ 'wsgi.run_once' ]       = True
    environ [ 'wsgi.url_scheme' ] = 'http'
    headers_set = [ ]
    headers_sent = [ ]
    # 把应答的结果输出到终端
    def write ( data ) :
        sys . stdout . write ( data )
        sys . stdout . flush ( )
    # 实现 start_response 函数，根据程序端传过来的 status 和 response_headers 参数，
    # 设置状态和头部
    def start_response ( status , response_headers , exc_info = None ) :
        headers_set [ : ] = [ status , response_headers ]
          return write
    # 调用客户端的可调用对象，把准备好的参数传递过去
    result = application ( environ , start_response )

    # 处理得到的结果，这里简单地把结果输出到标准输出。
    try :
        for data in result :
            if data :      # don't send headers until body appears
                write ( data )
    finally :
        if hasattr ( result , 'close' ) :
            result . close ( )
```

2 由Django框架分析WSGI

下面我们以**django**为例，分析一下**wsgi**的整个流程

django WSGI application

WSGI application应该实现为一个可调用**iter**对象，例如函数、方法、类(包含**__call__**方法)。需要接收两个参数：一个字典，该字典可以包含了客户端请求的信息以及其他信息，可以认为是请求上下文，一般叫做**environment**（编码中多简写为environ、env），一个用于发送HTTP响应状态（HTTP status）、响应头（HTTP headers）的回

调函数,也就是**start_response()**。通过回调函数将响应状态和响应头返回给server，同时返回响应正文(response body)，响应正文是可迭代的、并包含了多个字符串。

下面是Django中application的具体实现部分：

Python

```
class WSGIHandler(base.BaseHandler): initLock = Lock() request_class = WSGIRequest def __call__(self, environ, start_response): # 加载中间件 if self._request_middleware is None: with self.initLock: try: # Check that middleware is still uninitialized. if self._request_middleware is None: self.load_middleware() except: # Unload whatever middleware we got self._request_middleware = None raise set_script_prefix(get_script_name(environ)) # 请求处理之前发送信号 signals.request_started.send(sender=self.__class__, environ=environ) try: request = self.request_class(environ) except UnicodeDecodeError: logger.warning('Bad Request (UnicodeDecodeError)',exc_info=sys.exc_info(), extra={'status_code': 400,} response = http.HttpResponseBadRequest() else: response = self.get_response(request) response._handler_class = self.__class__ status = '%s %s' % (response.status_code, response.reason_phrase) response_headers = [(str(k), str(v)) for k, v in response.items()] for c in response.cookies.values(): response_headers.append((str('Set-Cookie'), str(c.output(header='')))) # server提供的回调方法，将响应的header和status返回给server start_response(force_str(status), response_headers) if getattr(response, 'file_to_stream', None) is not None and environ.get('wsgi.file_wrapper'): response = environ['wsgi.file_wrapper'](response.file_to_stream) return response
```

```
class WSGIHandler ( base . BaseHandler ) :
    initLock = Lock ( )
    request_class = WSGIRequest
    def __call__ ( self , environ , start_response ) :
    # 加载中间件
      if self . _request_middleware is None :
          with self . initLock :
              try : # Check that middleware is still uninitialized.
                  if self . _request_middleware is None :
                      self . load_middleware ( )
              except : # Unload whatever middleware we got
                      self . _request_middleware = None raise
      set_script_prefix ( get_script_name ( environ ) ) # 请求处理之前发送信号
      signals . request_started . send ( sender = self . __class__ , environ = environ )
      try :
          request = self . request_class ( environ )
      except UnicodeDecodeError :
          logger . warning ( 'Bad Request (UnicodeDecodeError)' , exc_info = sys . exc_info ( ) , extra = {'status
          response = http . HttpResponseBadRequest ( )
      else :
          response = self . get_response ( request )
      response . _handler_class = self . __class__ status = '%s %s' % ( response . status_code , response .reasor
      response_headers = [ ( str ( k ) , str ( v ) ) for k , v in response . items ( ) ] for c in response . cookies .valu
Cookie' ) , str ( c . output ( header = '' ) ) ) )
      # server提供的回调方法，将响应的header和status返回给server
      start_response ( force_str ( status ) , response_headers )
      if getattr ( response , 'file_to_stream' , None ) is not None and environ . get ( 'wsgi.file_wrapper' ) :
          response = environ [ 'wsgi.file_wrapper' ] ( response . file_to_stream )
      return response
```

可以看出application的流程包括:加载所有中间件，以及执行框架相关的操作，设置当前线程脚本前缀，发送请求开始信号；处理请求，调用get_response()方法处理当前请求，该方法的的主要逻辑是通过urlconf找到对应的view和callback，按顺序执行各种middleware和callback。调用由server传入的start_response()方法将响应header与status返回给server。返回响应正文

django WSGI Server

负责获取http请求，将请求传递给WSGI application，由application处理请求后返回response。以Django内建server为例看一下具体实现。通过runserver运行django项目，在启动时都会调用下面的run方法，创建一个WSGIServer的实例，之后再调用其serve_forever()方法启动服务。

Python
```
def run(addr, port, wsgi_handler, ipv6=False, threading=False): server_address = (addr, port) if threading: httpd_cls = type(str('WSGIServer'), (socketserver.ThreadingMixIn, WSGIServer), {}) else: httpd_cls = WSGIServer # 这里的wsgi_handler就是WSGIApplication httpd = httpd_cls(server_address, WSGIRequestHandler, ipv6=ipv6) if threading: httpd.daemon_threads = True httpd.set_app(wsgi_handler) httpd.serve_forever()
```

```
1   def run ( addr , port , wsgi_handler , ipv6 = False , threading = False ) :
2     server_address = ( addr , port )
3     if threading :
4         httpd_cls = type ( str ( 'WSGIServer' ) , ( socketserver . ThreadingMixIn , WSGIServer ) , { } )
5     else :
6         httpd_cls = WSGIServer # 这里的wsgi_handler就是WSGIApplication
7     httpd = httpd_cls ( server_address , WSGIRequestHandler , ipv6 = ipv6 )
8      if threading :
9         httpd . daemon_threads = True httpd . set_app ( wsgi_handler )
10     httpd . serve_forever ( )
```

下面表示WSGI server服务器处理流程中关键的类和方法。

WSGIServerrun()方法会创建**WSGIServer**实例，主要作用是接收客户端请求，将请求传递给**application**，然后将**application**返回的**response**返回给客户端。

创建实例时会指定**HTTP**请求的**handler：WSGIRequestHandler**类，通过**set_app**和**get_app**方法设置和获取**WSGIApplication**实例**wsgi_handler。**

处理http请求时，调用**handler_request**方法，会创建**WSGIRequestHandler，**实例处理http请求。WSGIServer中**get_request**方法通过**socket**接受请求数据。

WSGIRequestHandler由WSGIServer在调用handle_request时创建实例，传入**request**、**cient_address**、**WSGIServer**三个参数，__init__方法在实例化同时还会调用自身的**handle**方法handle方法会创建**ServerHandler**实例，然后调用其**run**方法处理请求

ServerHandlerWSGIRequestHandler在其handle方法中调用run方法，传入self.server.get_app()参数，获取**WSGIApplication**，然后调用实例(__call__)，获取**response**，其中会传入**start_response**回调，用来处理返回的**header**和**status**。通过application获取response以后，通过finish_response返回response

WSGIHandlerWSGI协议中的application，接收两个参数，environ字典包含了客户端请求的信息以及其他信息，可以认为是请求上下文，start_response用于发送返回status和header的回调函数

虽然上面一个WSGI server涉及到多个类实现以及相互引用，但其实原理还是调用WSGIHandler，传入请求参数以及回调方法start_response()，并将响应返回给客户端。

3 实际环境使用的wsgi服务器

因为每个web框架都不是专注于实现服务器方面的，因此，在生产环境部署的时候使用的服务器也不会简单的使用web框架自带的服务器，这里，我们来讨论一下用于生产环境的服务器有哪些?

> 1.**gunicorn**

Gunicorn（从Ruby下面的Unicorn得到的启发）应运而生：依赖Nginx的代理行为，同Nginx进行功能上的分离。由于不需要直接处理用户来的请求（都被Nginx先处理），Gunicorn不需要完成相关的功能，其内部逻辑非常简单：接受从Nginx来的动态请求，处理完之后返回给Nginx，由后者返回给用户。

由于功能定位很明确，Gunicorn得以用纯Python开发：大大缩短了开发时间的同时，性能上也不会很掉链子。同时，它也可以配合Nginx的代理之外的别的Proxy模块工作，其配置也相应比较简单。

配置上的简单，大概是它流行的最大的原因。

> 2.**uwsgi**

因为使用C语言开发，会和底层接触的更好，配置也是比较方便，目前和gunicorn两个算是部署时的唯二之选。

以下是通常的配置文件

Python
[uwsgi] http = $(HOSTNAME):9033 http-keepalive = 1 pythonpath = ../ module = service master = 1 processes = 8 daemonize = logs/uwsgi.log disable-logging = 1 buffer-size = 16384 harakiri = 5 pidfile = uwsgi.pid stats = $(HOSTNAME):1733 运行：uwsgi --ini conf.ini

```
1   [ uwsgi ]
2   http = $ ( HOSTNAME ) : 9033
3   http - keepalive = 1
4   pythonpath = . . /
5   module = service
6   master = 1
7   processes = 8
8   daemonize = logs / uwsgi . log
9   disable - logging = 1
10  buffer - size = 16384
11  harakiri = 5
12  pidfile = uwsgi . pid
13  stats = $ ( HOSTNAME ) : 1733
```

```
14
15
16          运行： uwsgi -- ini  conf . ini
```

> 3.**fcgi**

不多数，估计使用的人也是比较少，这里只是提一下

> 4.**bjoern**

Python WSGI界最牛逼性能的Server其中一个是bjoern，纯C，小于1000行代码，就是看不惯uWSGI的冗余自写的。

4 WSGI服务器比较

综合广大Python开发者的实际经历，我们可以得出，使用最广的当属uWSGI以及gunicorn，我们这里来比较比较两者与其他服务器的区别。
1.gunicorn本身是个多进程管理器，需要指定相关的不同类型的worker去工作，使用gevent作为worker时单机大概是3000RPS Hello World，胜过torando自带的服务器大概是2000左右，uWSGI则会更高一点。
2.相比于tornado对于现有代码需要大规模重构才能用上高级特性，Gevent只需要一个monkey，容易对代码进行快速加工。
3.gunicorn 可以做 pre hook and post hook.

下面来对比以下uWSGI和gunicorn的速度差比

可以看到，如果单纯追求性能，那uWSGI会更好一点，而gunicorn则会更易安装和结合gevent。

结合这篇文章,我们也可以得出相同结论，在阻塞响应较多的情况下，gunicorn的gevent模式无疑性能会更加强大。

功能实现方面，无疑uWSGI会更多一些，配置也会更加复杂一些，可以看看uWSGI的配置和gunicorn的配置。

至于怎么去选择，就看大家的项目结构怎么样了。

最后，宣传一下我们的开源组织，PSC开源组，希望以开源项目的方式让每个人都能更有融入性的去学习，公开化你的学习。