花了两个星期，我终于把 WSGI 整明白了

在 三百六十行，行行转 IT 的现状下，很多来自各行各业的同学，都选择 Python 这门胶水语言做为踏入互联网大门的第
人里，又有相当大比例的同学选择了 Web 开发这个方向（包括我）。而从事 web 开发，绕不过一个知识点，就是 WS

不管你是否是这些如上同学中的一员，都应该好好地学习一下这个知识点。

由于我本人不从事专业的 python web 开发，所以在写这篇文章的时候，借鉴了许多优秀的网络博客，并花了很多的精
OpenStack 代码。

为了写这篇文章，零零散散地花了大概两个星期。本来可以拆成多篇文章，写成一个系列的，经过一番思虑，还是准备
文章这么长的原因。

另外，一篇文章是不能吃透一个知识点的，本篇涉及的背景知识也比较多的，若我有讲得不到位的，还请你多多查阅其他人的网络博客进一步学习。

在你往下看之前，我先问你几个问题，你带着这些问题往下看，可能更有目的性，学习可能更有效果。

问1：一个 HTTP 请求到达对应的 application处理函数要经过怎样的过程?

问2：如何不通过流行的 web 框架来写一个简单的web服务?

一个HTTP请求的过程可以分为两个阶段，第一阶段是从客户端到WSGI Server，第二阶段是从WSGI Server 到WSGI Application

今天主要是讲第二阶段，主要内容有以下几点：

1. WSGI 是什么，因何而生?
2. HTTP请求是如何到应用程序的?

01. WSGI 是什么，因何而生？

WSGI是 Web Server Gateway Interface 的缩写。

它是 Python应用程序（application）或框架（如 Django）和 Web服务器之间的一种接口，已经被广泛接受。

它是一种协议，一种规范，其是在 PEP 333提出的，并在 PEP 3333 进行补充（主要是为了支持 Python3.x）。这个协
框架和web server软件的兼容问题。有了WSGI，你不用再因为你使用的web 框架而去选择特定的 web server软件。

常见的web应用框架有：Django，Flask等

常用的web服务器软件有：uWSGI，Gunicorn等

那这个 WSGI 协议内容是什么呢？知乎上有人将 PEP 3333 翻译成中文，写得非常好，我将这段协议的内容搬运过来。

> WSGI 接口有服务端和应用端两部分，服务端也可以叫网关端，应用端也叫框架端。服务端调用一个由应用端提供的可调用对象。如何提供这个对象，由服务端决定。例如某些服务器或者网关需要应用的部署者写一段脚本，以创建服务器或者网关的实例，并且为这个实例提供一个应用实例。另一些服务器或者网关则可能使用配置文件或其他方法以指定应用实例应该从哪里导入或获取。

WSGI 对于 application 对象有如下三点要求

1. 必须是一个可调用的对象
2. 接收两个必选参数environ、start_response。
3. 返回值必须是可迭代对象，用来表示http body。

02. HTTP请求是如何到应用程序的？

当客户端发出一个 HTTP 请求后，是如何转到我们的应用程序处理并返回的呢？

关于这个过程，细节的点这里没法细讲，只能讲个大概。

我根据其架构组成的不同将这个过程的实现分为两种：

1、两级结构

在这种结构里，uWSGI作为服务器，它用到了HTTP协议以及wsgi协议，flask应用作为application，实现了wsgi协议。
求，uWSGI接受请求，调用flask app得到相应，之后相应给客户端。
这里说一点，通常来说，Flask等web框架会自己附带一个wsgi服务器(这就是flask应用可以直接启动的原因)，但是这只
的，在生产环境是不够用的，所以用到了uwsgi这个性能高的wsgi服务器。

2、三级结构

这种结构里，uWSGI作为中间件，它用到了uwsgi协议(与nginx通信)，wsgi协议(调用Flask app)。当有客户端发来请求
态资源是nginx的强项)，无法处理的请求(uWSGI)，最后的相应也是nginx回复给客户端的。
多了一层反向代理有什么好处?

提高web server性能(uWSGI处理静态资源不如nginx；nginx会在收到一个完整的http请求后再转发给wWSGI)

nginx可以做负载均衡(前提是有多个服务器)，保护了实际的web服务器(客户端是和nginx交互而不是uWSGI)

03. 实现一个简单的 WSGI Server

在上面的架构图里，不知道你发现没有，有个库叫做 `wsgiref` ，它是 Python 自带的一个 wsgi 服务器模块。

从其名字上就看出，它是用纯Python编写的WSGI服务器的参考实现。所谓“参考实现”是指该实现完全符合WSGI标准，但是不考虑任何运行效率，仅供开发和测试使用。

有了 wsgiref 这个模块，你就可以很快速的启动一个wsgi server。

```
# 这里的 appclass 暂且不说，后面会讲到
app = appclass()
server = make_server('', 64570, app)
server.serve_forever()
```


当你运行这段代码后，就会开启一个 wsgi server，监听 `0.0.0.0:64570` ，并接收请求。

使用 lsof 命令可以查到确实开启了这个端口

以上使用 wsgiref 写了一个demo，让你对wsgi有个初步的了解。其由于只适合在学习测试使用，在生产环境中应该另

04. 实现“高并发”的 WSGI Server

上面我们说不能在生产中使用 wsgiref ，那在生产中应该使用什么呢？选择有挺多的，比如优秀的 uWSGI，Gunicore等。但是今天我并不准备讲这些，一是因为我不怎么熟悉，二是因为我本人从事 OpenStack 的二次开发，对它比较熟悉。

所以下面，是我花了几天时间阅读 OpenStack 中的 Nova 组件代码的实现，刚好可以拿过来学习记录一下，若有理解偏差，还望你批评指出。

在 nova 组件里有不少服务，比如 nova-api，nova-compute，nova-conductor，nova-scheduler 等等。

其中，只有 nova-api 有对外开启 http 接口。

要了解这个http 接口是如何实现的，从服务启动入口开始看代码，肯定能找到一些线索。

从 Service 文件可以得知 nova-api 的入口是 `nova.cmd.api:main()`

打开 `nova.cmd.api:main()` ，一起看看是 OpenStack Nova 的代码。

在如下的黄框里，可以看到在这里使用了service.WSGIService 启动了一个 server，就是我们所说的的 wsgi server

那这里的 WSGI Server 是依靠什么实现的呢？让我们继续深入源代码。

wsgi.py 可以看到这里使用了 eventlet 这个网络并发框架，它先开启了一个绿色线程池，从配置里可以看到这个服务器一次接收的请求并发量是 1000 。

可是我们还没有看到 WSGI Server 的身影，上面使用eventlet 开启了线程池，那线程池里的每个线程应该都是一个服务请求的?

再继续往下，可以发现，每个线程都是使用 eventlet.wsgi.server 开启的 WSGI Server，还是使用的 eventlet。

由于源代码比较多，我提取了主要的代码，精简如下

```
# 创建绿色线程池
self._pool = eventlet.GreenPool(self.pool_size)

# 创建 socket: 监听的ip，端口
bind_addr = (host, port)
self._socket = eventlet.listen(bind_addr, family, backlog=backlog)
dup_socket = self._socket.dup()

# 整理孵化协程所需的各项参数
wsgi_kwargs = {
    'func': eventlet.wsgi.server,
    'sock': dup_socket,
    'site': self.app, # 这个就是 wsgi 的 application 函数
    'protocol': self._protocol,
    'custom_pool': self._pool,
    'log': self._logger,
```


目录

```
    'socket_timeout': self.client_socket_timeout
}

# 孵化协程
self._server = utils.spawn(**wsgi_kwargs)
```


就这样，nova 开启了一个可以接受1000个绿色协程并发的 WSGI Server。

05. 第一次路由：PasteDeploy

上面我们提到 WSGI Server 的创建要传入一个 Application，用来处理接收到的请求，对于一个有多个 app 的项目。

比如，你有一个个人网站提供了如下几个模块

Copy

```
/blog  # 博客 app
/wiki  # wiki app
```

如何根据 请求的url 地址，将请求转发到对应的application上呢?

答案是，使用 PasteDeploy 这个库（在 OpenStack 中各组件被广泛使用）。

PasteDeploy 到底是做什么的呢?

根据 官方文档 的说明，翻译如下

> PasteDeploy 是用来寻找和配置WSGI应用和服务的系统。PasteDeploy给开发者提供了一个简单的函数loadapp。通过这个函数，可以从一个配置文件或者Python egg中加载一个WSGI应用。

我会先讲下在 Nova 中，是如何借助 PasteDeploy 实现对url的路由转发。

还记得在上面创建WSGI Server的时候，传入了一个 self.app 参数，这个app并不是一个固定的app，而是使用 PasteD
loadapp 函数从 paste.ini 配置文件中加载application。

具体可以，看下nova的实现。

通过打印的 DEBUG 内容得知 config_url 和 app name 的值

```
app: osapi_compute
config_url: /etc/nova/api-paste.inia
```

通过查看 `/etc/nova/api-paste.ini` ，在 composite 段里找到了 `osapi_compute` 这个app（这里的app和wsgi app 是两个概念，需要注意区分），可以看出 nova 目前有两个版本的api，一个是 v2，一个是v2.1，目前我们在用的是 v2.1，从配置文件中，可以得到其指定的 application 的路径是 `nova.api.openstack.compute` 这个模块下的 APIRouterV21 类 的factory方法，这是一个工厂函数，返回 APIRouterV21 实例。

```
[composite:osapi_compute]
use = call:nova.api.openstack.urlmap:urlmap_factory
/: oscomputeversions
/v2: openstack_compute_api_v21_legacy_v2_compatible
/v2.1: openstack_compute_api_v21

[app:osapi_compute_app_v21]
paste.app_factory = nova.api.openstack.compute:APIRouterV21.factory
```

这是 OpenStack 使用 PasteDeploy 实现的第一层的路由，如果你不感兴趣，可以直接略过本节，进入下一节，下一节是 介绍 PasteDeploy 的一个简易的web server demo。推荐一定要看。

06. PasteDeploy 使用说明

到上一步，我已经得到了 application 的有用的线索。考虑到很多人是第一次接触 PasteDeploy，所以这里结合网上博客做了下总结，对你入门会有帮助。

掌握 PasteDeploy ，你只要按照以下三个步骤逐个完成即可。

1、配置 PasteDeploy使用的ini文件；

2、定义WSGI应用；

3、通过loadapp函数加载WSGI应用；

第一步：写 paste.ini 文件

在写之前，咱得知道 ini 文件的格式吧。

首先，像下面这样一个段叫做 `section` 。

```
[type:name]
key = value
...
```

其上的type，主要有如下几种

1. `composite` （组合）：多个app的路由分发；

```
[composite:main]
use = egg:Paste#urlmap
/ = home
/blog = blog
/wiki = wiki
```

2. app（应用）：指明 WSGI 应用的路径；

```
paste.app_factory = example:Home.factory
```

3. pipeline（管道）：给一个 app 绑定多个过滤器。将多个filter和最后一个WSGI应用串联起来。

```
[pipeline:main]
pipeline = filter1 filter2 filter3 myapp

[filter:filter1]
...

[filter:filter2]
...

[app:myapp]
...
```

4. filter（过滤器）：以 app 做为唯一参数的函数，并返回一个“过滤”后的app。通过键值next可以指定需要将请求
   定的可以是一个普通的WSGI应用，也可以是另一个过滤器。虽然名称上是过滤器，但是功能上不局限于过滤功能
   例如日志功能，即将认为重要的请求数据记录下来。

```
[app-filter:filter_name]
use = egg:...
next = next_app

[app:next_app]
...
```


对 ini 文件有了一定的了解后，就可以看懂下面这个 ini 配置文件了

```
[composite:main]
use = egg:Paste#urlmap
/blog = blog
/wiki = wiki

[app:blog]
paste.app_factory = example:Blog.factory

[app:wiki]
```

第二步是定义一个符合 WSGI 规范的 applicaiton 对象。

符合 WSGI 规范的 application 对象，可以有多种形式，函数，方法，类，实例对象。这里仅以实例对象为例（需要实现 __call__ 方法），做一个演示。


```
import os
from paste import deploy
from wsgiref.simple_server import make_server

class Blog(object):
    def __init__(self):
        print("Init Blog.")

    def __call__(self, environ, start_response):
        status_code = "200 OK"
        response_headers = [("Content-Type", "text/plain")]
        response_body = "This is Blog's response body.".encode('utf-8')

        start_response(status_code, response_headers)
        return [response_body]

    @classmethod
    def factory(cls, global_conf, **kwargs):
        print("Blog factory.")
        return Blog()
```

loadapp 函数可以接收两个实参：

- URI："config:<配置文件的全路径>"
- name：WSGI应用的名称

```
conf_path = os.path.abspath('paste.ini')

# 加载 app
applications = deploy.loadapp("config:{}".format(conf_path) , "main")

# 启动 server, 监听 localhost:22800
server = make_server("localhost", "22800", applications)
server.serve_forever()
```

applications 是URLMap 对象。

完善并整合第二步和第三步的内容，写成一个 Python 文件(wsgi_server.py)。内容如下

```
import os
from paste import deploy
from wsgiref.simple_server import make_server

class Blog(object):
    def __init__(self):
        print("Init Blog.")
```

```
        response_headers = [("Content-Type", "text/plain")]
        response_body = "This is Blog's response body.".encode('utf-8')

        start_response(status_code, response_headers)
        return [response_body]

    @classmethod
    def factory(cls, global_conf, **kwargs):
        print("Blog factory.")
        return Blog()


class Wiki(object):
    def __init__(self):
        print("Init Wiki.")

    def __call__(self, environ, start_response):
        status_code = "200 OK"
        response_headers = [("Content-Type", "text/plain")]
        response_body = "This is Wiki's response body.".encode('utf-8')

        start_response(status_code, response_headers)
        return [response_body]

    @classmethod
    def factory(cls, global_conf, **kwargs):
        print("Wiki factory.")
        return Wiki()
```

```
if __name__ == "__main__":
    app = "main"
    port = 22800
    conf_path = os.path.abspath('paste.ini')

    # 加载 app
    applications = deploy.loadapp("config:{}".format(conf_path) , app)
    server = make_server("localhost", port, applications)

    print('Started web server at port {}'.format(port))
    server.serve_forever()
```


一切都准备好后，在终端执行 `python wsgi_server.py` 来启动 web server

如果像上图一样一切正常，那么打开浏览器

- 访问http://127.0.0.1:8000/blog，应该显示：This is Blog's response body.
- 访问http://127.0.0.1:8000/wiki，应该显示：This is Wiki's response body.。

注意：urlmap对url的大小写是敏感的，例如如果访问http://127.0.0.1:8000/BLOG，在url映射中未能找到大写的BLOG。

到此，你学会了使用 PasteDeploy 的简单使用。

07. webob.dec.wsgify 装饰器

经过了 PasteDeploy 的路由调度，我们找到了 `nova.api.openstack.compute:APIRouterV21.factory` 这个 application 的入口，看代码知道它其实返回了 APIRouterV21 类的一个实例。

APIRouterV21 本身没有实现 `__call__` ，但它的父类 Router实现了 `__call__`

我们知道，application 必须遵从 WSGI 的规范

1. 必须接收 `environ` ， `start_response` 两个参数;
2. 必须返回 「可迭代的对象」。

但从 Router 的 `__call__` 代码来看，它并没有遵从这个规范，它不接收这两个参数，也不返回 response，而只是返回
对象，就这样我们的视线被一次又一次的转移，但没有关系，这些 `__call__` 都是外衣，只要扒掉这些外衣，我们就能

而负责扒掉这层外衣的，就是其头上的装饰器 `@webob.dec.wsgify` ，wsgify 是一个类，其 `__call__` 源码实现如下

可以看出，wsgify 在这里，会将 req 这个原始请求（dict对象）封装成 Request 对象（就是规范1里提到的 environ），
往里地执行被wsgify装饰的函数（self._route）， 得到最内部的核心application。

上面提到了规范1里的第一个参数，补充下第二个参数start_response，它是在哪定义并传入的呢?

其实这个无需我们操心，它是由 wsgi server 提供的，如果我们使用的是 wsgiref 库做为 server 的话。那这时的 start_response 就由 wsgiref 提供。

再回到 wsgify，它的作用主要是对 WSGI app 进行封装，简化wsgi app的定义与编写，它可以很方便的将一个 callable 的函数或对象，封装成一个 WSGI app。

上面，其实留下了一个问题，self._route（routes 中间件 RoutesMiddleware对象）是如何找到真正的 application呢?

带着这个问题，我们了解下 routes 是如何为我们实现第二次路由。

08. 第二次路由：中间件 routes 路由

这张图把一个 HTTP 请求粗略简单地划分为两个过程。但事实上，整个过程远比这个过程要复杂得多。

实际上在 WSGI Server 到 WSGI Application 这个过程中，我们加很多的功能（比如鉴权、URL路由），而这些功能的实现方式，我们称之为中间件。

中间件，对服务器而言，它是一个应用程序，是一个可调用对象， 有两个参数，返回一个可调用对象。而对应用程序而言，它是一个服务器，为应用程序提供了参数，并且调用了应用程序。

今天以URL路由为例，来讲讲中间件在实际生产中是如何起作用的。

当服务器拿到了客户端请求的URL，不同的URL需要交由不同的函数处理，这个功能叫做 URL Routing。

在 Nova 中是用 routes 这个库来实现对URL的的路由调度。接下来，我将从源代码处分析一下这个过程。

在routes模块里有个中间件，叫 `routes.middleware.RoutesMiddleware` ，它将接受到的 url，自动调用 `map.match()` 方法，对 url 进行路由匹配，并将匹配的结果存入request请求的环境变量 `['wsgiorg.routing_args']` ，最后会调用 `self._dispatch` （dispatch返回真正的application）返回response，最后会将这个response返回给 WSGI Server。

这个中间件的原理，看起来是挺简单的。并没有很复杂的逻辑。

但是，我在阅读 routes 代码的时候，却发现了另一个令我困惑的点。

`self._dispatch` （也就上图中的self.app）函数里，我们看到了 app，controller 这几个很重要的字眼，其是否是我苦苦追寻的 application 对象呢?

要搞明白这个问题，只要看清 match 到是什么东西?

这个 match 对象 是在 `RoutesMiddleware.__call__()` 里塞进 `req.environ` 的，它是什么东西呢，我将其打印出来。

```
{'action': u'index', 'controller': <nova.api.openstack.wsgi.ResourceV21 object at 0x6ec8910>, 'project_id': u'2

{'action': u'show', 'controller': <nova.api.openstack.wsgi.ResourceV21 object at 0x6ed9710>, 'project_id': u'2a
```


结果令人在失所望呀，这个 app 并不是我们要寻找的 Controller 对象。而是 nova.api.openstack.wsgi.ResourceV21 了就是Resource 对象。

看到这里，我有心态有点要崩了，怎么还没到 Controller？OpenStack 框架的代码绕来绕去的，没有点耐心还真的很难

既然已经开了头，没办法还得硬着头皮继续读了下去。

终于我发现，在APIRouter初始化的时候，它会去注册所有的 Resource，同时将这些 Resource 交由 routes.Mapper 关 射，所以上面提到的 routes.middleware.RoutesMiddleware 才能根据url通过 mapper.match 获取到相应的Resource。

从 Nova 代码中看出每个Resource 对应一个 Controller 对象，因为 Controller 对象本身就是对一种资源的操作集合。

通过日志的打印，可以发现 nova 管理的 Resource 对象有多么的多而杂

```
os-server-groups
os-keypairs
os-availability-zone
remote-consoles
os-simple-tenant-usage
os-instance-actions
os-migrations
os-hypervisors
diagnostics
```

```
os-agents
os-fixed-ips
os-networks
os-security-groups
os-security-groups
os-security-group-rules
flavors
os-floating-ips-bulk
os-console-auth-tokens
os-baremetal-nodes
os-cloudpipe
os-server-external-events
os-instance_usage_audit_log
os-floating-ips
os-security-group-default-rules
os-tenant-networks
os-certificates
os-quota-class-sets
os-floating-ip-pools
os-floating-ip-dns
entries
os-aggregates
os-fping
os-server-password
os-flavor-access
consoles
os-extra_specs
os-interface
os-services
servers
```


AI助手

```
limits
ips
os-cells
versions
tags
migrations
os-hosts
os-virtual-interfaces
os-assisted-volume-snapshots
os-quota-sets
os-volumes
os-volumes_boot
os-volume_attachments
os-snapshots
os-server-groups
os-keypairs
os-availability-zone
remote-consoles
os-simple-tenant-usage
os-instance-actions
os-migrations
os-hypervisors
diagnostics
os-agents
images
os-fixed-ips
os-networks
os-security-groups
```

```
os-floating-ips-bulk
os-console-auth-tokens
os-baremetal-nodes
os-cloudpipe
os-server-external-events
os-instance_usage_audit_log
os-floating-ips
os-security-group-default-rules
os-tenant-networks
os-certificates
os-quota-class-sets
os-floating-ip-pools
os-floating-ip-dns
entries
os-aggregates
os-fping
os-server-password
os-flavor-access
consoles
os-extra_specs
os-interface
os-services
servers
extensions
metadata
metadata
limits
ips
```

```
os-cells
tags
migrations
os-hosts
os-virtual-interfaces
os-assisted-volume-snapshots
os-quota-sets
os-volumes
os-volumes_boot
os-volume_attachments
os-snapshots
```


你一定很好奇，这路由是如何创建的吧，关键代码就是如下一行。如果你想要了解更多路由的创建过程，可以看一下这篇（Route总结），写得不错。

```
routes.mapper.connect("server",
                      "/{project_id}/servers/list_vm_state",
                      controller=self.resources['servers'],
                      action='list_vm_state',
                      conditions={'list_vm_state': 'GET'})
```

Copy

历尽了千辛万苦，我终于找到了 Controller 对象，知道了请求发出后，wsgi server是如何根据url找到对应的Controller（根据routes.Mapper路由映射）。

但是很快，你又会问。对于一个资源的操作（action），有很多，比如新增，删除，更新等

不同的操作要执行Controller 里不同的函数。

如果是新增资源，就调用 create()

如果是删除资源，就调用 delete()

如果是更新资源，就调用 update()

那代码如何怎样知道要执行哪个函数呢?

以/servers/xxx/action请求为例，请求调用的函数实际包含在请求的body中。

经过routes.middleware.RoutesMiddleware的 `__call__` 函数解析后，此时即将调用的Resource已经确定为哪个模块
建的Resource，而 action 参数为"action"，接下来在Resource的 `__call__` 函数里面会因为action=="action"从而
容，找出Controller中所对应的方法。

Controller在构建的过程中会由于MetaClass的影响将其所有action类型的方法填入一个字典中，key由每个 `_action_x`
`@wsgi.action('xxx')` 装饰函数给出，value为每个_action_xxx方法的名字（从中可以看出规律，在body里面请求的
即为Controller中对应调用的方法）。

之后在使用Controller构建Resource对象的过程中会向Resource注册该Controller的这个字典中的内容。这样，只需在请求的body中给出调用方法的key，然后就可以找到这个key所映射的方法，最后在Resource的__call__函数中会调用Controller类的这个函数!

其实我在上面我们打印 match 对象时，就已经将对应的函数打印出来了。

这边以 nova show（展示资源为例），来理解一下。

当你调用 nova show [uuid] 命令，novaclient 就会给 nova-api 发送一个http的请求

```
nova show 1c250b15-a346-43c5-9b41-20767ec7c94b
```

Copy

AI助手

通过打印得到的 match 对象如下

目录 ×

其中 action 就是对应的处理函数，而controller 就对应的 Resource 对象，project_id 是租户id（你可以不理会）。

继续看 ResourceV21 类里的 `__call__` 函数的代码。

图示地方，会从 environ 里获取中看到获取 action 的具体代码

我将这边的 action_args打印出来

```
{'action': 'show', 'project_id': '2ac17c7c792d45eaa764c30bac37fad9', 'id': '1c250b15-a346-43c
```

其中 action 还是是函数名，id 是要操作的资源的唯一id标识。

在 `__call__` 的最后，会 调用 `_process_stack` 方法

在图标处，get_method 会根据 action（函数名） 取得处理函数对象。


```
meth :<bound method ServersController.show of <nova.api.openstack.compute.servers.ServersController object at 0
```


最后，再执行这个函数，取得 action_result，在 `_process_stack` 会对 response 进行初步封装。

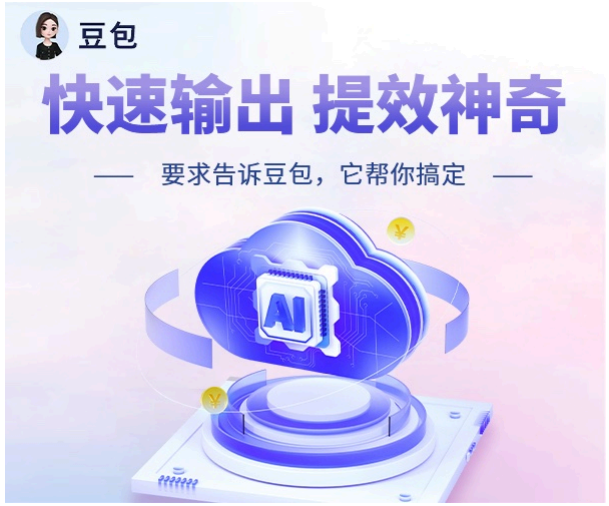
至此，一个请求从发出到响应就结束了。

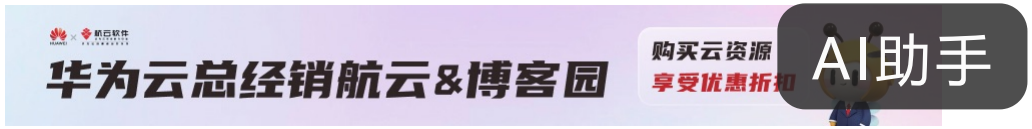
附录：参考文章

- PEP 3333 中文翻译
- nova-api源码分析（APP的调用）
- Python Route总结
- Python routes Mapper 的使用
- 详解 Paste deploy
- paste.ini 文件使用说明
- PasteDeploy 小白教程
- WSGI 两种架构图
- 伯乐在线：Python Web开发最难懂的WSGI协议
- WSGI 简介

标签: Python , WSGI , Web

posted @ 2019-06-11 12:44 王一白 阅读(15727) 评论(6) 编辑 收藏 举报





目录

· 3款.NET开源、功能强大的通讯调试工具，效率提升利器！
· 推荐一个C#轻量级矢量图形库
· .NET 9 增强 OpenAPI 规范，不再内置swagger

Powered By Cnblogs | Theme Silence v1.1.2

目录 ×

AI助手