Webpy

2024年8月17日　9:35

一、简介

Python Web开发中，服务端程序可以分为两个部分
一个是服务器程序：负责把客户端请求接收、整理（Server）
二是应用程序：负责具体的逻辑处理（Application）

中间件就是我们之间的一些组件，为了进一步实现某些功能

而为了方便应用程序的开发，我们把常用的功能封装起来，称为各种Web开发框架，例如Django、Flask、Webpy

不同的框架有不同的开发方式，但是无论如何，开发出的程序都需要与服务器程序进行配合，才能为用户提供服务。这样的话，服务器就需要给不同的框架提供不同的支持，这样是很混乱的，非常不好

这个时候，标准化就尤为重要，我们可以设立一个标准，只要服务器程序支持这个标准，框架也支持这个标准，这样就实现了服务器程序与框架的配合使用。这样服务器就可以支持更多支持标准的框架。框架也就支持多标准的服务器

Python Web 开发中，这个标准就是Python Web Server Gateway Interface 即 WSGI，Web服务器的网关接口协议

WSGI是作为Web服务器与Web应用程序或应用框架之间的一种低级的接口，它是基于现存的CGI标准设计的。

WSGI将Web组件分为三类。Web服务器（Server）、中间件（Middleware）、Web应用程序（Application）

Web.py是一款轻量级的python开发框架

二、安装

测试web.py是否安装完成

```
import web
urls = (
    '/(.*)','hello',
)
app = web.application(urls,globals())
class hello:
    def GET(self,name):
        if not name:
            name = 'World!'
        return 'Hello' + name

if __name__ == '__main__':
    app.run()

    # app.wsgifunc()
```

三、URL映射

URL映射就是一个URL请求由那块代码（类或函数）来处理。

URL的调度模式被定义在元组中

元组的格式是 'URL路径','处理类'

URL映射有三种类型：
1、URL完全匹配

2、URL模糊匹配
3、URL带组匹配

四、webpy架构分析

整个项目可以分为三层：应用层、中间层、网络层

应用层负责处理应用逻辑，并暴露除用户app的一个wsgi调用接口
中间层提供应用层和网络层的对接服务，用户可以在这里选择使用不同的服务器模式来执行 wsgi app

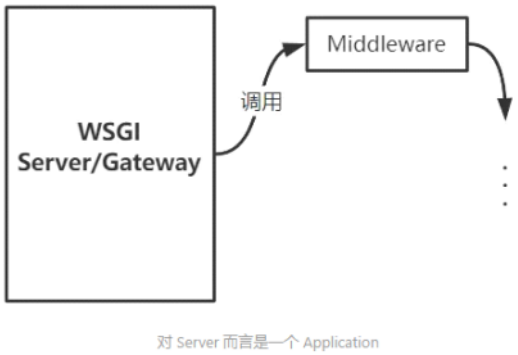
应用框架的主要逻辑在 application.py 文件中，

用户对服务器请求之后，服务器将请求相关信息进行处理(主要是得到 wsgi 调用所需要的两个参数 environ和 start response)并传给 wsgi方法进行调用，这个wsgi方法就是wsgifunc()函数

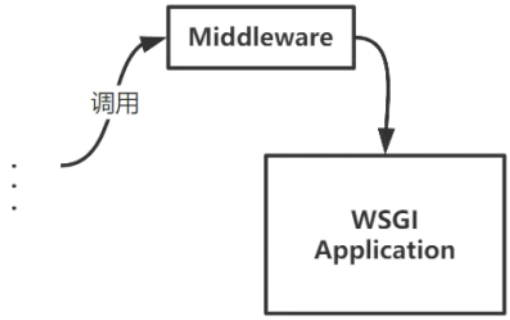
五、中间层解析

有些功能介于服务器程序和应用程序之间，例如服务器拿到了客户端请求的 URL，不同的 URL需要交给不同的函数处理，这个功能叫做 URL Routing，这个功能就可以放在二者中间实现，这个中间层就是middleware.

其实就是中间件，对服务器来说，中间层就是应用框架。对于application来说，中间件就是服务器



对 Server 而言是一个 Application



对 Application 而言是一个 Server

如上图，如果 Middleware 的执行完成，需要执行下一层 Middleware或者 Application，就将 environ 和 start response 作为参数调用下一层，

对于下一层看来，它的行为与 Server 无异。

Middleware 本身对于前后的概念是透明的，这也意味着在 Server 和Application 之间可以有多层 Middleware 嵌套，这样的好处在于它可以将逻辑拆分开，使得每一层处理其专业化的过程，同时允许随意修改中间件的调用顺序，甚至编写逻辑实现上层的调用管理，中间件的设计使得这些解决方案能够被灵活地支持。

Middleware 对服务器程序和应用是透明的，也就是说，服务器程序以为它就是应用程序，而应用程序以为它就是服务器。所以 middleware 需要把自己伪装成一个服务器，接受应用程序，调用它，同时middleware还需要把自己伪装成一个应用程序，传给服务器程序。

六、application模块解析

```
import web
urls = (
    '/(.*)','hello',
)
app = web.application(urls,globals())
class hello:
    def GET(self,name):
        if not name:
            name = 'World!'
        return 'Hello' + name

if __name__ == '__main__':
    app.run()

    # app.wsgifunc()
```

这个模块主要是实现了 WSGI兼容的接口，以便应用程序能够被WSGI应用服务器调用。WSGI是WebServer GatewayInterface(服务器网关接口)的缩写。WSGI是为 Python 语言定义的 Web 服务器和 Web应用框架之间的一种简单而通用的接口。

比如后面我们会使用到wsgiserver(uWSGI)，它需要和 wsgiapplication交互(webpy)，uWSGI需要将过来的请求转发给webpy处理，那么这个时候他们二者的交互就需要一个统一的规范，这个规范就是WSGI.

- **接口的使用，使用 web.py 自带的 HTTP Server**

  app.run()的调用时初始化各种 WCGI 接口(CGI 为通用网关接口描述了服务器和请求处理程序之间传输数据的一种标准，可以让客户端从网页浏览器向服务器发起请求)，并启动一个内置的HTTP服务器和这些接口对接

  app.run()函数是之间运行webpy自身拥有的服务器，该服务器只能用作测试，能处理的并发量较小

- **与WSGI应用服务器对接**

  如果应用需要与WSGI服务器进行对接（uWSGI），而不是webpy自己的小型服务器，就不能使用app.run()。

  应用入口的代码应该为 app.wsgifunc()

在这种场景下，应用的代码不需要启动 HTTP 服务器，而是实现一个WSGI兼容的接口供 WSGI服务器调用。web.py 框架为我们实现了在这样的接口，所以我们只需要调用 application =app.wsgifunc()就可以了这里所得到的 application 变量就是 WSGI 接口。

- **WSGI接口的实现分析**

获得一个接口主要的代码就两行

```
app = web.application(urls,globals())
application = app.wsgifunc()
```

1、web.application的实例化

（1）通过web.application()函数完成实例化。初始化这个实例需要传入两个参数，URL路由元组和globals()的返回结果

（2）URL元组就是不同的URL请求调用那块代码解决

（3）globals()函数会返回一个类似字典对象，包含当前空间的所有变量、函数、类以及模块，键是这些东西的名称，值是响应对象。目的是可以通过键值对来调用该空间下的所有内容（比如数据库的连接，配置之类的）

（4）另外还可以传递第三个变量，autoreload，用来指定是否需要自动重新导入Python模块，这个在调试的时候非常有用

2、application的初始代码块如下：

主要内容使用init_mapping()函数，就是将传入的URL元组进行解析，生成一个列表，存到成员变量mapping中将获取的全局变量，赋值给成员变量 fvars

```
class application:
    def __init__(self, mapping=(), fvars={}, autoreload=None):
        if autoreload is None:
            autoreload = web.config.get("debug", False)
        self.init_mapping(mapping)
        self.fvars = fvars
        self.processors = []

        self.add_processor(loadhook(self._load))
        self.add_processor(unloadhook(self._unload))

        if autoreload:
```

```
def init_mapping(self, mapping):
    self.mapping = list(utils.group(mapping, 2))
```

```
urls = ("/", "Index",
    "/hello/(.*)", "Hello",
    "/world", "World")
```

如果用户初始化时传递的元组是这样的，那么调用 init_mapping 之后：

```
self.mapping = [["/", "Index"],
        ["/hello/(.*)", "Hello"],
        ["/world", "World"]]
```

添加了两个处理器

```
self.add_processor(loadhook(self._load))
self.add_processor(unloadhook(self._unload))
```

用于加载一些变量，ctx就是一个全局的成员变量，包括路径之类的

```
def _load(self):
    web.ctx.app_stack.append(self)
```

```
def _unload(self):
    web.ctx.app_stack = web.ctx.app_stack[:-1]

    if web.ctx.app_stack:
        # this is a sub-application, revert ctx to earlier state.
        oldctx = web.ctx.get("_oldctx")
        if oldctx:
            web.ctx.home = oldctx.home
            web.ctx.homepath = oldctx.homepath
            web.ctx.path = oldctx.path
            web.ctx.fullpath = oldctx.fullpath
```

3、wsgifunc()函数分析

执行这个函数时，返回一个wsgi兼容的函数，并且该函数内部实现了URL路由等功能。

除开内部函数的定义，wsgifunc的定义非常简单，如果没有实现任何中间件，那么就是自己返回其内部定义的wsgi函数

for循环挨个套用中间件，然后返回内部定义的wsgi函数

```
def wsgifunc(self, *middleware):
    """Returns a WSGI-compatible function for this application."""

    for m in middleware:
        wsgi = m(wsgi)

    return wsgi
```

简化的wsgifunc函数

wsgi()函数分析

这个函数就是我们之前写的application。他接收的也是两个参数 environ、start_response
我们来将该函数划分为5个部分

```
def wsgi(env, start_resp):
    # clear threadlocal to avoid inteference of previous requests
    #------1---------------
    self._cleanup()
    #------2---------------
    self.load(env)
    #--------------------3--------------------------
    try:
        # allow uppercase methods only
        if web.ctx.method.upper() != web.ctx.method:
            raise web.nomethod()
        result = self.handle_with_processors()
        if result and hasattr(result, "__next__"):
            result = peep(result)
        else:
            result = [result]
    except web.HTTPError as e:
        result = [e.data]
    def build_result(result):
        for r in result:
            if isinstance(r, bytes):
                yield r
            else:
                yield str(r).encode("utf-8")
    result = build_result(result)
    #--------------------3--------------------------
    #--------------------4--------------------------
    status, headers = web.ctx.status, web.ctx.headers
    start_resp(status, headers)
    def cleanup():
        self._cleanup()
        yield b""  # force this function to be a generator
    return itertools.chain(result, cleanup())
    #--------------------4--------------------------
#----------5--------------
for m in middleware:
    wsgi = m(wsgi)
#----------5--------------
return wsgi
```

（1）self._cleanup()，清除数据

这个函数内部调用clear_all()，清除了所有的数据，避免内存泄漏

```
def _cleanup(self):
    # Threads can be recycled by WSGI servers.
    # Clearing up all thread-local state to avoid interefereing with subsequent requests.
    utils.ThreadedDict.clear_all()
```

（2）self.load(env)，函数内部使用env中的参数初始化web.ctx变量。

```
def load(self, env):
    """Initializes ctx using env."""
    ctx = web.ctx
    ctx.clear()
    ctx.status = "200 OK"
    ctx.headers = []
    ctx.output = ""
    ctx.environ = ctx.env = env
    ctx.host = env.get("HTTP_HOST")

    if env.get("wsgi.url_scheme") in ["http", "https"]:
        ctx.protocol = env["wsgi.url_scheme"]
    elif env.get("HTTPS", "").lower() in ["on", "true", "1"]:
        ctx.protocol = "https"
    else:
        ctx.protocol = "http"
    ctx.homedomain = ctx.protocol + "://" + env.get("HTTP_HOST", "[unknown]")
    ctx.homepath = os.environ.get("REAL_SCRIPT_NAME", env.get("SCRIPT_NAME", ""))
    ctx.home = ctx.homedomain + ctx.homepath
    # @@ home is changed when the request is handled to a sub-application.
    # @@ but the real home is required for doing absolute redirects.
    ctx.realhome = ctx.home
    ctx.ip = env.get("REMOTE_ADDR")
    ctx.method = env.get("REQUEST_METHOD")   请求方法
    try:
        ctx.path = bytes(env.get("PATH_INFO"), "latin1").decode("utf8")
    except UnicodeDecodeError:  # If there are Unicode characters...
        ctx.path = env.get("PATH_INFO")
```

（3）第三段代码主要是调用handle_with_processors()，这个函数会对请求的URL进行路由，找到合适的类或子应用来处理请求。也会调用添加的处理器（在初始化的时候添加）来处理其它的事情。

内部最核心的代码就是调用handle()函数进行处理

```
def handle_with_processors(self):
    def process(processors):
        try:
            if processors:
                p, processors = processors[0], processors[1:]
                return p(lambda: process(processors))
            else:
                return self.handle()
        except web.HTTPError:
            raise
        except (KeyboardInterrupt, SystemExit):
            raise
        except:
            print(traceback.format_exc(), file=web.debug)
            raise self.internalerror()

    # processors must be applied in the resvere order. (??)
    return process(self.processors)
```

handle()函数内部 调用 match()函数，参数mapping就是在init_mapping()函数中初始化的URL元组，path是本次请求的路径。目的就是根据请求的路由来匹配处理的方法（类或者是函数）

```
def handle(self):
    fn, args = self._match(self.mapping, web.ctx.path)
    return self._delegate(fn, self.fvars, args)
```

对于返回的结果，有三种处理方式。

① 返回一个可迭代对象，则进行安全迭代处理

② 返回其他值，则创建一个列表来进行存放

③ 如果抛出了一个 HTTPError异常，则将异常中的数据e.data封装成一个列表

最后得到一个 result列表

（4）第四段代码，会对第三步返回的result列表进行字符串化处理，得到HTTP Response(HTTP响应)的body部分。然后根据WSGI的规范做下面的两件事情

① 调用start_response()函数

② 将result结果转换为一个迭代器

（5）循环嵌套中间件。

最后返回一个wsgi函数

所以 application = app.wsgifunc() 的作用就是把wsgi函数赋值给application变量，这样应用服务器就可以采用WSGI的标准可我们的应用框架进行对接

上面所有的代码都是分析我们的application是如何初始化，生成接口，以便于和WSGI应用服务器进行对接的

七、处理HTTP请求

上面第四点分析的代码说明了 webpy 框架是如何实现 WSGI兼容接口的，所以我们已经大概了解了 HTTP 请求到达框架以及从框架返回给应用服务器的流程，现在了解一下框架内部是如何调用我们的应用代码来实现一个个请求的处理。

1、loadhook 和 unloadhook 装饰器

这两个函数是真实处理器的函数的装饰器函数，装饰后得到的处理器分别对应请求处理前 和 请求处理后

```
self.add_processor(loadhook(self._load))
self.add_processor(unloadhook(self._unload))
```

```
def loadhook(h):
    def processor(handler):
        h()
        return handler()

    return processor
```

loadhook()函数返回一个函数 processor，他会确保先调用你提供的处理器函数 h ，然后再调用后续的操作函数 handler

unloadhook()函数也返回一个processor。他会先调用参数传递进来的hanlder，然后再调用你提供的处理器函数

2、handle_with_processors函数

```
def handle_with_processors(self):
        def process(processors):
            try:
                if processors:
                    p, processors = processors[0], processors[1:]
                    return p(lambda: process(processors))
                else:
                    return self.handle()
            except web.HTTPError:
                raise
            except (KeyboardInterrupt, SystemExit):
                raise
            except:
                print(traceback.format_exc(), file=web.debug)
                raise self.internalerror()
        # processors must be applied in the resvere order. (??)
        return process(self.processors)
```

当框架开始执行这个函数的时候，是逐个执行这些处理器的。这个函数非常复杂，最核心的部分采用了递归实现，也就是规定了处理器的执行顺序。

前面提到过，在初始化 application 实例的时候会添加两个处理器到

```
self.add_processor(loadhook(self._load))
self.add_processor(unloadhook(self._unload))
```

执行的顺序： （1）self._load()
（2）self.handle()
（3）self._unload()

如果后面还有更多的装饰器，也是按照这种方式执行下去。对于loadhook 装饰的处理器，**先添加的先执行**，对于unloadhook 装饰的处理器，**后添加的先执行**。最后执行handle 函数。

3、handle()函数

在所有的 load 处理器执行完之后，就会执行self.handle()函数，其内部会调用我们写的应用代码。比如返回个hello,world 之后的。

```
def handle(self):
    fn, args = self._match(self.mapping, web.ctx.path)
    return self._delegate(fn, self.fvars, args)
```

（1）_match()函数

这个函数的参数中 mapping 就是 self.mapping，是 URL 路由映射表;value 是web.ctx.path，是本次请求的路径。这个函数遍历self.mapping，根据映射关系中处理对象的类型来处理。

返回值是处理请求的类，和一些参数

```
def _match(self, mapping, value):
        for pat, what in mapping:
            if isinstance(what, application): # 位置 1
                if value.startswith(pat):
                    f = lambda: self._delegate_sub_application(pat, what)
                    return f, None
                else:
                    continue
            elif isinstance(what, str):   # 位置 2
                what, result = utils.re_subm(r'^%s\Z' % (pat,), what, value)
            else:  # 位置 3
                result = utils.re_compile(r'^%s\Z' % (pat,)).match(value)
            if result:  # it's a match
                return what, [x for x in result.groups()]
        return None, None
```

① 位置1，处理对象是一个 application 实例，也就是一个子应用，则返回一个匿名函数，该匿名函数会调用 self._delegate_sub_application 进行处理，

② 位置 2，如果处理对象是一个字符串则调用 utils.re_subm进行处理，这里会把 value(也就是 web.ctx.path)中的和 pat 匹配的部分替换成 what(也就是我们指定的一个 URL 模式的处理对象字符串)，然后返回替换后的结果以及匹配的项(是一个re.MatchObject 实例)。路由匹配。

③ 位置3，如果是其他情况，比如直接指定一个类对象作为处理对象。

如果 result 非空，则返回处理对象和一个参数列表(这个参数列表就是传递给我们实现的 GET 等函数的参数)

（2）_delegate()函数

从_match函数返回的结果会作为参数传递给_delegate()函数

```
def handle(self):
    fn, args = self._match(self.mapping, web.ctx.path)
    return self._delegate(fn, self.fvars, args)
```

其中：

① fn:是要处理当前请求的对象，一般是个类名 比如说 login

② args:是要传递给请求处理对象的参数 比如说登录时的账号密码（userid、pwd）

③ self.fvars:是实例化 application 时的全局名称空间，会用于查找处理对象

```
def _delegate(self, f, fvars, args=[]):
    def handle_class(cls):
        meth = web.ctx.method
        if meth == "HEAD" and not hasattr(cls, meth):
            meth = "GET"
        if not hasattr(cls, meth):
            raise web.nomethod(cls)
        tocall = getattr(cls(), meth)
        return tocall(*args)
    if f is None:
        raise web.notfound()
    elif isinstance(f, application):
        return f.handle_with_processors()
    elif isclass(f):
        return handle_class(f)
    elif isinstance(f, str):
        if f.startswith("redirect "):
            url = f.split(" ", 1)[1]
            if web.ctx.method == "GET":
                x = web.ctx.env.get("QUERY_STRING", "")
                if x:
```

```
                url += "?" + x
            raise web.redirect(url)
        elif "." in f:
            mod, cls = f.rsplit(".", 1)
            mod = __import__(mod, None, None, [""])
            cls = getattr(mod, cls)
        else:
            cls = fvars[f]
        return handle_class(cls)
    elif hasattr(f, "__call__"):
        return f()
    else:
        return web.notfound()
```

这个函数主要是根据参数f的类型来做出不同的处理

- f为空，则返回 302 NOT Found
- f是一个 application 实例，则调用子应用的handle_with_processors()处理
- f是一个类对象，则调用内部函数handle_class
- f是一个字符串，则进行重定向处理，或者获取要处理请求的类名后，调用 handle class 进行处理(我们写的代码一般是在这个分支下被调用)
- f是一个可调用对象，直接调用
- 其他情况返回 302 NOT Found