Flask源代码分析 0.11

2016年12月2日
16:12

以一个简单的打印“Hello World!”为例分析Flask结构,代码示例如下:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return '<h1>Hello World!</h1>'

if __name__ == '__main__':
    app.run(debug=True)
```

1. run分析

如果是直接执行这个文件，将会调用app.run , app 是Flask 的一个实例， run是其中的一个method

run()主要是调用run_simple函数， （run 的define 位于Flask的app.py中）

```
def run(self, host=None, port=None, debug=None, **options):
    from werkzeug.serving import run_simple
    ...
    try:
        run_simple(host, port, self, **options)
    finally:
        self._got_first_request = False
```

从run_simple 开始进入werkzeug

run_simple()主要是调用inner() 函数, inner 函数的define 如下:（位于werkzeug的serving.py， ）

```
    def inner():
        srv = make_server(hostname, port, application, threaded,
                          processes, request_handler,
                          passthrough_errors, ssl_context,
                          fd=fd)
        srv.serve_forever()
```

make_server() 在假设(threaded = False，processes = 1）时， 返回BaseWSGIServer实例

```
def make_server(host=None, port=None, app=None, threaded=False, processes=1,
                request_handler=None, passthrough_errors=False, ssl_context=None, fd=None):
    return BaseWSGIServer(host, port, app, request_handler,
                          passthrough_errors, ssl_context, fd=fd)
```

class BaseWSGIServer(HTTPServer, object) ， BaseWSGIServer 继承自 HTTPServe

```
    def serve_forever(self):
        self.shutdown_signal = False
        try:
            HTTPServer.serve_forever(self)
        except KeyboardInterrupt:
            pass
        finally:
            self.server_close()
```

srv.serve_forever()其实是BaseWSGIServer类中的serve_forever()方法，然后我们发现BaseWSGIServer类继承了HTTPServer类，且BaseWSGIServer的serve_forever()方法中调用了HTTPServer的serve_forever()方法。找到HTTPServer类，如下代码：

```
class HTTPServer(SocketServer.TCPServer):
    allow_reuse_address = 1
    def server_bind(self):
        SocketServer.TCPServer.server_bind(self)
        host, port = self.socket.getsockname()[:2]
        self.server_name = socket.getfqdn(host)
        self.server_port = port
```

HTTPServer类中并没有serve_forever()方法，且这个类继承了 SocketServer.TCPServer，我们再找到TCPServer类，然而它也没有serve_forever()方法，且这个类继承了BaseServer类，所以再去BaseServer里面看看，如下代码：

```
class BaseServer
def serve_forever(self, poll_interval=0.5):
    self.__is_shut_down.clear()
    try:
        while not self.__shutdown_request:
            r, w, e = _eintr_retry(select.select, [self], [], [], poll_interval)
            if self in r:
                self._handle_request_noblock()
    finally:
        self.__shutdown_request = False
```

```
        self.__is_shut_down.set()
```

总结：

run-〉run_simple -> make_server (1)创建 BaseWSGIServer实例，(2)调用 BaseWSGIServer中的 serve_forever

srv.serve_forever()其实是调用了BaseServer里面的serve_forever()方法，它接受一个参数poll_interval，用于表示select轮询的时间。然后进入一个无限循环，调用select方式进行网络IO监听。也就是说app.run()启动的是一个BaseWSGIServer，该服务通过一层一层的继承创建socket来进行网络监听，等待客户端连接。

整理一下相关server类的继承关系，如下：

BaseWSGIServer-->HTTPServer-->SocketServer.TCPServer-->BaseServer

从上面的类继承关系，我们可以很容易的理解，因为Flask是一个Web框架，所以需要一个HTTP服务，而HTTP服务是基于TCP服务的，而TCP服务最终会有一个基础服务来处理socket。这一条线都能够解释的通。

源文档 <https://segmentfault.com/a/1190000005788124>

2.server接受requst 返回response

Flask没有自定义请求处理类，使用了WSGI库的WSGIRequestHandler。

```
class WSGIRequestHandler(BaseHTTPRequestHandler, object):
```

1.
```
def handle_one_request(self):
    """Handle a single HTTP request."""
    self.raw_requestline = self.rfile.readline()
    if not self.raw_requestline:
        self.close_connection = 1
    elif self.parse_request():
        return self.run_wsgi()
```

2.
```
def run_wsgi(self):
    def write(data):
        assert headers_set, 'write() before start_response'
        if not headers_sent:
            status, response_headers = headers_sent[:] = headers_set
            try:
                code, msg = status.split(None, 1)
            except ValueError:
                code, msg = status, ""
            self.send_response(int(code), msg)
            header_keys = set()
            for key, value in response_headers:
                self.send_header(key, value)
                key = key.lower()
                header_keys.add(key)
            if 'content-length' not in header_keys:
                self.close_connection = True
                self.send_header('Connection', 'close')
            if 'server' not in header_keys:
                self.send_header('Server', self.version_string())
            if 'date' not in header_keys:
                self.send_header('Date', self.date_time_string())
            self.end_headers()

        assert isinstance(data, bytes), 'applications must write bytes'
        self.wfile.write(data)
        self.wfile.flush()

    def start_response(status, response_headers, exc_info=None):
        if exc_info:
            try:
                if headers_sent:
                    reraise(*exc_info)
            finally:
                exc_info = None
        elif headers_set:
            raise AssertionError('Headers already set')
        headers_set[:] = [status, response_headers]
        return write

    def execute(app):
        application_iter = app(environ, start_response)  app是传入的 Flask实例，因为内部定义了__call__ 函数，所以可以被调用
        try:
            for data in application_iter:
                write(data)
            if not headers_sent:
                write(b'')
        finally:
            if hasattr(application_iter, 'close'):
                application_iter.close()
            application_iter = None

    try:
        execute(self.server.app)
```

```
        application_iter = None
    try:
        execute(self.server.app)
    except (socket.error, socket.timeout) as e:
        self.connection_dropped(e, environ)
    except Exception:
        if self.server.passthrough_errors:
            raise
        from werkzeug.debug.tbtools import get_current_traceback
        traceback = get_current_traceback(ignore_system_exceptions=True)
        try:
            # if we haven't yet sent the headers but they are set
            # we roll back to be able to set them again.
            if not headers_sent:
                del headers_set[:]
            execute(InternalServerError())
        except Exception:
            pass
        self.server.log('error', 'Error on request:\n%s',
                        traceback.plaintext)
```

3.
```
def __call__(self, environ, start_response):
    """Shortcut for :attr:`wsgi_app`."""
    return self.wsgi_app(environ, start_response)
```

4.
```
def wsgi_app(self, environ, start_response):
    ctx = self.request_context(environ)
    ctx.push()
    error = None
    try:
        try:
            response = self.full_dispatch_request()
        except Exception as e:
            error = e
            response = self.make_response(self.handle_exception(e))
        return response(environ, start_response)
    finally:
        if self.should_ignore_error(error):
            error = None
        ctx.auto_pop(error)
```

5.
```
def full_dispatch_request(self):
    """Dispatches the request and on top of that performs request
    pre and postprocessing as well as HTTP exception catching and
    error handling.

    .. versionadded:: 0.7
    """
    self.try_trigger_before_first_request_functions()
    try:
        request_started.send(self)
        rv = self.preprocess_request()
        if rv is None:
            rv = self.dispatch_request()
    except Exception as e:
        rv = self.handle_user_exception(e)
    response = self.make_response(rv)
    response = self.process_response(response)
    request_finished.send(self, response=response)
    return response
```

make_response:
Converts the return value from a view function to a real
response object that is an instance of :attr:`response_class`.

6.
```
def dispatch_request(self):
    """Does the request dispatching.  Matches the URL and returns the
    return value of the view or error handler.  This does not have to
    be a response object.  In order to convert the return value to a
    proper response object, call :func:`make_response`.

    .. versionchanged:: 0.7
       This no longer does the exception handling, this code was
       moved to the new :meth:`full_dispatch_request`.
    """
    req = _request_ctx_stack.top.request
    if req.routing_exception is not None:
        self.raise_routing_exception(req)
    rule = req.url_rule
    # if we provide automatic options for this URL and the
    # request came with the OPTIONS method, reply automatically
    if getattr(rule, 'provide_automatic_options', False) \
       and req.method == 'OPTIONS':
        return self.make_default_options_response()
    # otherwise dispatch to the handler for that endpoint
    return self.view_functions[rule.endpoint](**req.view_args)
```

@app.route() 是一个装饰器， 内部调用

```
def route(self, rule, **options):
    def decorator(f):
        self.add_url_rule(rule, endpoint, f, **options)
        return f
    return decorator
```

```
add_url_rule:
  def add_url_rule(self, rule, endpoint=None, view_func=None, **options):
    self.view_functions[endpoint] = view_fun
```

```
    and req.method == 'OPTIONS':
        return self.make_default_options_response()
    # otherwise dispatch to the handler for that endpoint
    return self.view_functions[rule.endpoint](**req.view_args)
```

总结：

1. handle_one_request->run_wsgi -> execute->app(environ, start_response) (Flask.__call__) -> wsgi_app ->full_dispatch_request-> dispatch_request 最终在 dispatch_request 中调用 view_functions , 得到 view_functions的返回值，
2. view_function 的返回值在 full_dispatch_requset 中通过make_reponse 转换为reponse

@app.route 分析：

3. router 是一个装饰器，内部调用add_url_rule，rule对应url， f对应视图函数 ， options 存method的字典

```
def route(self, rule, **options):
    def decorator(f):
        self.add_url_rule(rule, endpoint, f, **options)
        return f
    return decorator
```

4. add_url_rule:

```
def add_url_rule(self, rule, endpoint=None, view_func=None, **options):
    methods = options.pop('methods', None)
    methods = set(item.upper() for item in methods)
    rule = self.url_rule_class(rule, methods=methods, **options)
    self.url_map.add(rule)
```

5. 其中url_rule_class = Rule, url_map =Map, Rule和Map分别调用的是werkzeug.routing 的Rule/Map