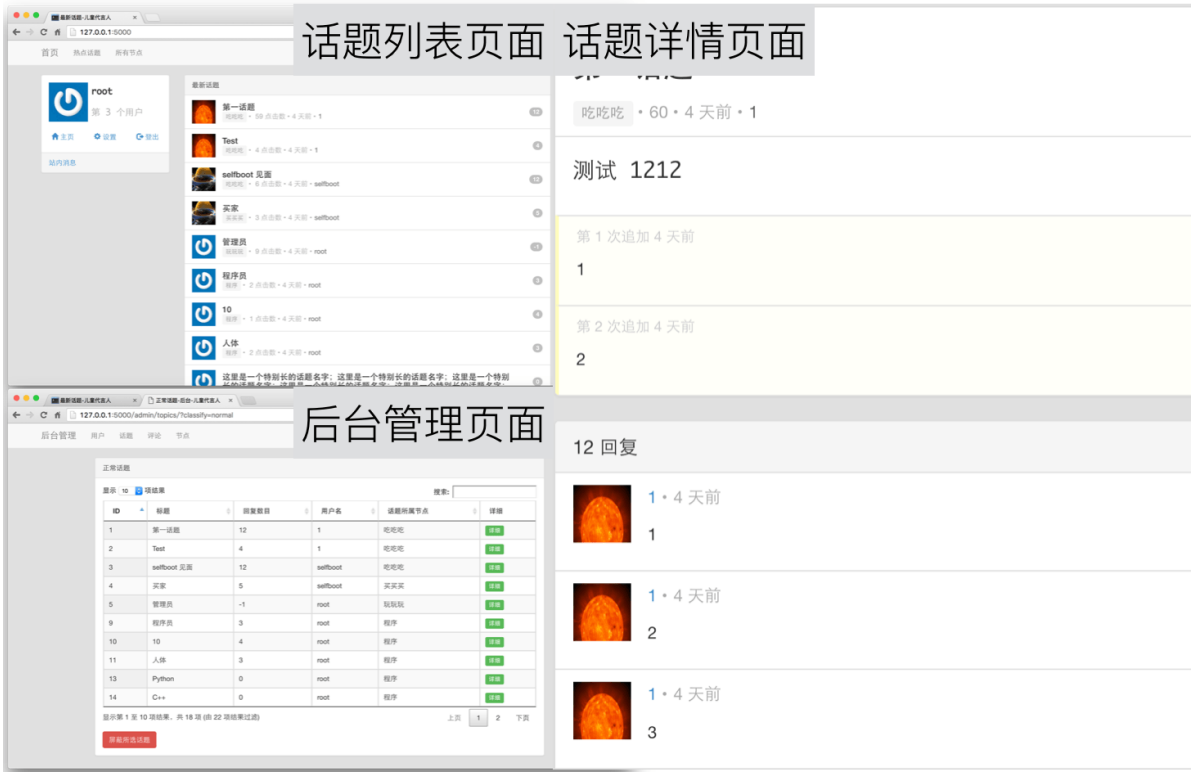
网站

https://selfboot.cn/page/5/

补充: http://mindhacks.cn/2009/03/28/effective-learning-and-memorization/

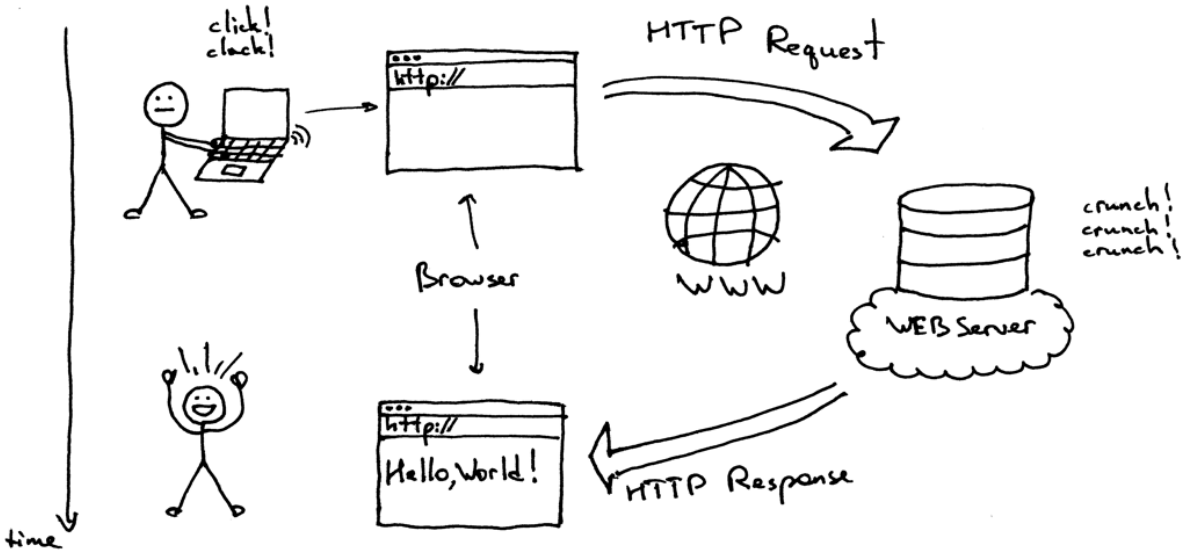
从零开始搭建论坛（一）：Web服务器与Web框架

学习一个框架最好的方式 是用框架做一个项目，在实战中理解掌握框架。用 Flask 框架，使用 Mysql 数据库做了一个论坛系统。麻雀虽小，五脏俱全，论坛效果图如下：



Web 服务器

1. 通信过程
    1. 浏览器输入URL后，浏览器先请求DNS服务器，获得请求站点的 IP 地址。
    2. 然后发送一个HTTP Request（请求）给该 IP 的主机，接着接收到服务器给的 HTTP Response（响应）
    3. 浏览器经过渲染后，以较好的效果呈现。这个过程，是Web服务器默默做贡献。
2. web服务器
    1. Web服务器是运行在物理服务器上的一个程序
        1. 它永久地等待客户端（主要是浏览器，比如Chrome，Firefox等）发送请求。
        2. 收到请求之后，它生成相应的响应并将其返回至客户端。
        3. Web服务器通过HTTP协议与客户端通信，也被称为HTTP服务器。
    2. 工作流程



Web 服务器

3. 实现一个简单的 Web 服务器。运行示例程序后，会监听本地端口 8000，在浏览器访问 http://localhost:8000 就能看到响应内容。而我们的程序也能够打印出客户端发来的请求内容，


```
Hello, I'm a simple Web Server!
```

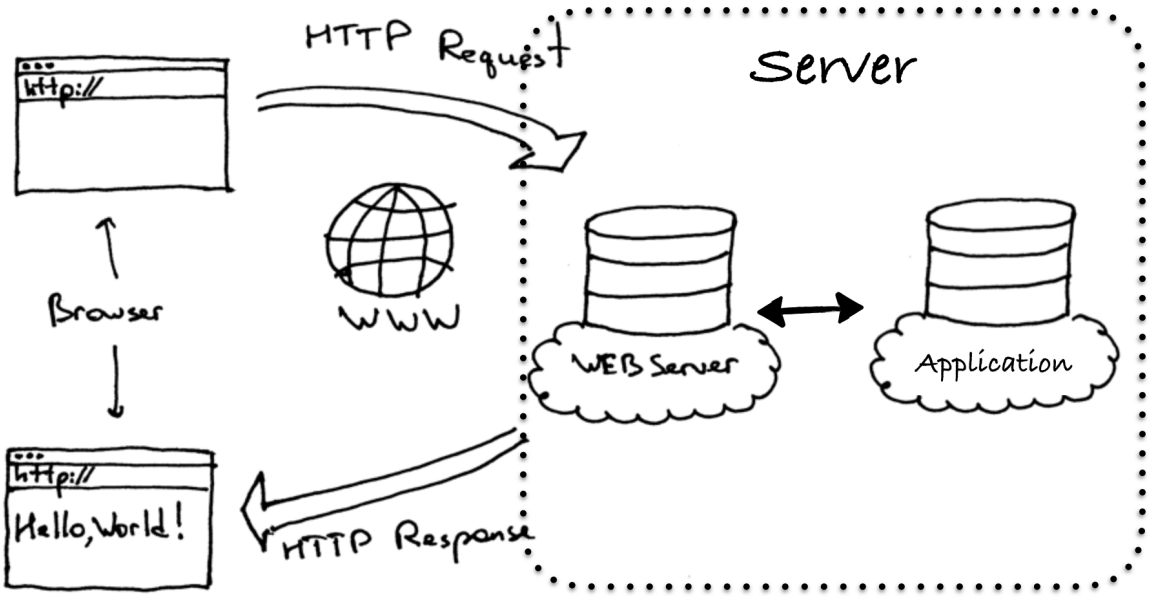
```
➜ Documents python demo.py
Serving HTTP on port 8000 ...
GET / HTTP/1.1
Host: localhost:8000
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrom
e/51.0.2704.103 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch
Accept-Language: zh-CN,zh;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: _ga=GA1.1.722326352.1465893402; PRUM_EPISODES=s=1465899827938&r=http%3A//localhost%3A5000/
```

简单Web服务器

4. Web服务器的工作原理 分4个步骤：`建立连接、请求过程、应答过程，关闭连接`
    1. 建立连接：客户机通过TCP/IP协议建立到服务器的TCP连接。
    2. 请求过程：客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档。
    3. 应答过程：服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端。由客户端解释HTML文档，在客户端屏幕上渲染图形结果。
    4. 关闭连接：客户机与服务器断开。
5. 实际的web服务器很复杂
    1. web服务器的主要工作是根据request返回response
    2. 实际中，Web 服务器远远复杂的多，要考虑的因素太多了，比如：
        1. 缓存机制：讲一些经常被访问的页面缓存起来，提高响应速度；
        2. 安全：防止黑客的各种攻击，比如 SYN Flood 攻击；
        3. 并发处理：如何响应不同客户端同时发起的请求；
        4. 日志：记录访问日至，方便做一些分析。
6. web服务器种类
    1. UNIX和LINUX平台 最广泛的免费 Web 服务器有`Apache`和`Nginx`。

Web 应用程序

1. 为什么要应用程序
    1. Web 服务器接受 Http Request，返回 Response.
    2. 很多时候 Response 不是静态文件，要一个应用程序根据 Request 生成相应的 Response。
    3. 应用程序处理业务逻辑，读取或者更新数据库，根据不同 Request 生成相应的 Response。
    4. 这不是 Web 服务器的工作. Web 服务器只负责 Http 协议层面和一些诸如并发处理，安全，日志等相关的事情。
    5. 小结
        1. 应用程序 生成动态 response
2. 应用程序
    1. 可用各种语言编写（Java, PHP, Python, Ruby等）开发
    2. 应用程序 从Web服务器接收客户端的请求，生成响应再给Web服务器，最后由Web服务器返回给客户端。
    3. 整个架构如下：



Web应用程序

4. 以 Python 为例， 用Python开发Web
    1. 最原始和直接的办法 用`CGI`标准， 1998年这种方式很流行。
    2. 设置:
        1. 首先 Web 服务器支持CGI, 且Web 服务器配置了CGI的处理程序
        2. 设置好CGI目录, 目录里添加 python 文件.
        3. 每一个 python 文件处理相应输入，生成一个 html 文件即可，如下例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

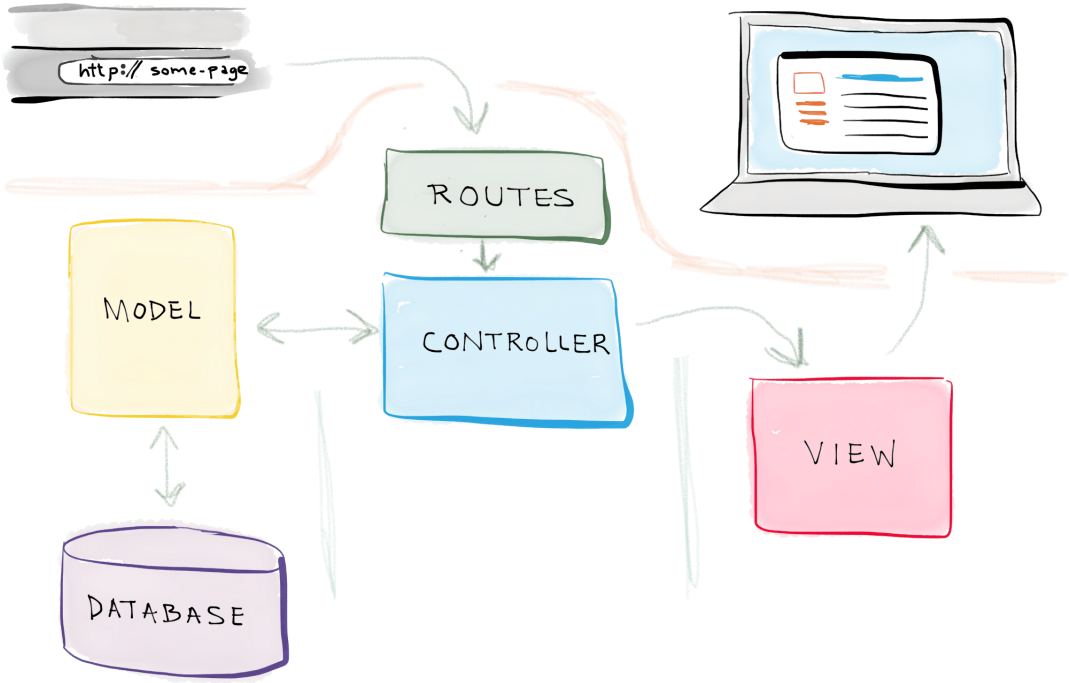
print "Content-type:text/html"
print          # 空行，告诉服务器结束头部
print '<html>'
print '<head>'
print '<meta charset="utf-8">'
print '</head>'
print '<body>'
print '<h2>Hello Word! 我是一个CGI程序</h2>'
print '</body>'
print '</html>'
```

2. 这样, 浏览器访问该文件得到Hello World 网页内容。

5. 问题
    1. CGI 写 Web 应用程序看起来很简单，每一个文件处理输入，生成html。
    2. 实际开发中，可能会遇到许多不方便的地方。比如：
        1. 每个独立的CGI脚本可能会重复写数据库连接，关闭的代码；
        2. 后端开发者会看到一堆 Content-Type 等和自己无关的 html 页面元素；

Web 框架

1. 为什么要web框架
    1. 早期开发站点 做了许多重复性劳动. 为了减少重复，避免写出庞杂，混乱的代码，提取 Web 开发的关键过程, 形成了各种 Web 框架。
    2. 有了框架，可以专注于编写清晰、易维护的代码，无需关心数据库连接之类的重复性工作。
    3. 小结: web框架 是 提取web开发的关键过程形成的
2. web框架的架构
    1. 用了 `MVC` 架构，如下图所示：



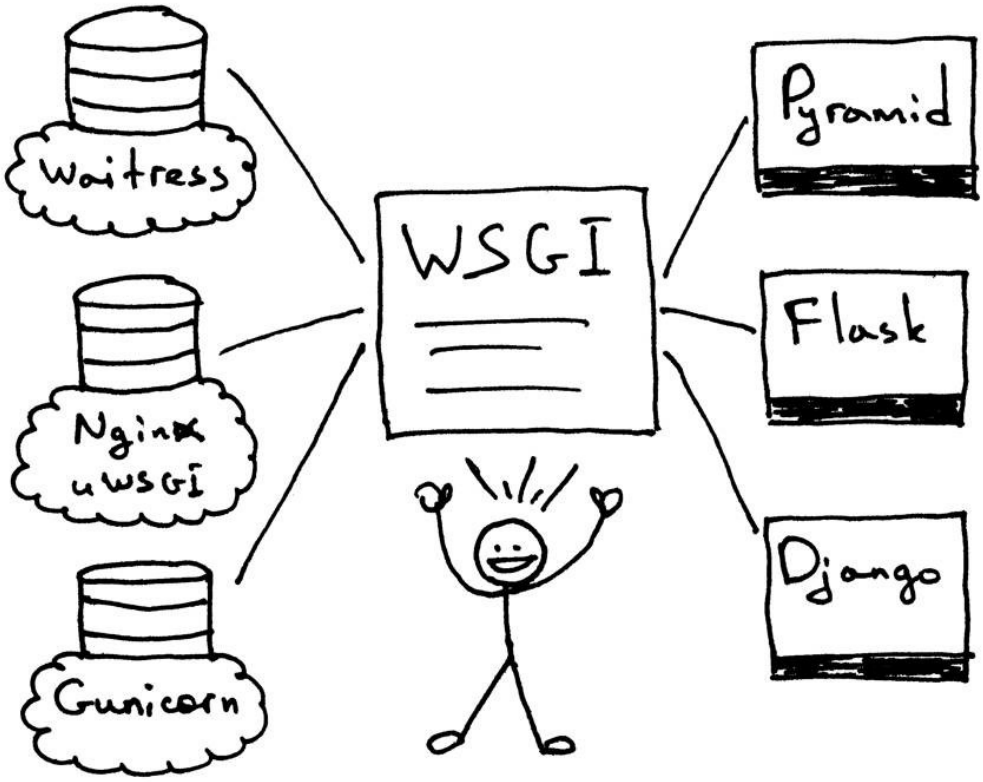
MVC 架构

2. MVC架构过程
    1. 用户输入 URL，客户端发送请求
    2. `控制器（Controller）`先拿到请求， 然后用`模型（Models）`从数据库取出需要的数据，并处理，再发送处理后的结果给`视图（View）`，
    3. 视图渲染数据生成 Html Response, 返回给客户端。
    4. 小结: 控制器, 模型, 视图
3. python web 框架 flask 为例
    1. flask框架
        1. 框架不限定架构组织应用
        2. flask 很好地支持 MVC 方式组织应用。
    2. flask框架
        1. `控制器`： flask 用`装饰器`添加路由项，如下：
            1. ```
               @app.route('/')
               def main_page():
                   pass
               ```
        2. `模型`： 作用: 从数据库中取数据
            1. ```
               @app.route('/')
               def main_page():
                   """Searches the database for entries, then displays
               them."""
                   db = get_db()
                   cur = db.execute('select * from entries order by id desc')
                   entries = cur.fetchall()
                   return render_template('index.html', entries=entries)
               ```
        3. `视图`： flask 用 jinja2 渲染页面，下面的模版文件指定了页面的样式：
            1. ```
               {% for entry in entries %}
               <li>
                 <h2>{{ entry.title }}</h2>
                 {{ entry.text|safe }}
               </li>
               {% else %}
               <li>No entries yet. Add some!</li>
               {% endfor %}
               ```

Web 服务器网关接口

1. 为什么要网关接口
    1. Python有许多的 Web 框架， 又有许多的 Web 服务器（Apache, Nginx, Gunicorn等）.
    2. 框架和Web服务器通信，它们要匹配. 框架会限制选择Web 服务器, 或Web 服务器限制选择框架，这不合理。
    3. 解决不合理的方法: 接口.
        1. 设计双方都遵守的接口
        2. 对python，就是WSGI（Web Server Gateway Interface，Web服务器网关接口）。
        3. 其他编程语言也拥有类似的接口：如Java的Servlet API和Ruby的Rack。
2. WSGI
               ```

1. WSGI 使 Web 服务器与 Web 框架不相互限制。
2. 如，用 Gunicorn 或Nginx/uWSGI 运行Django、Flask或web.py应用。

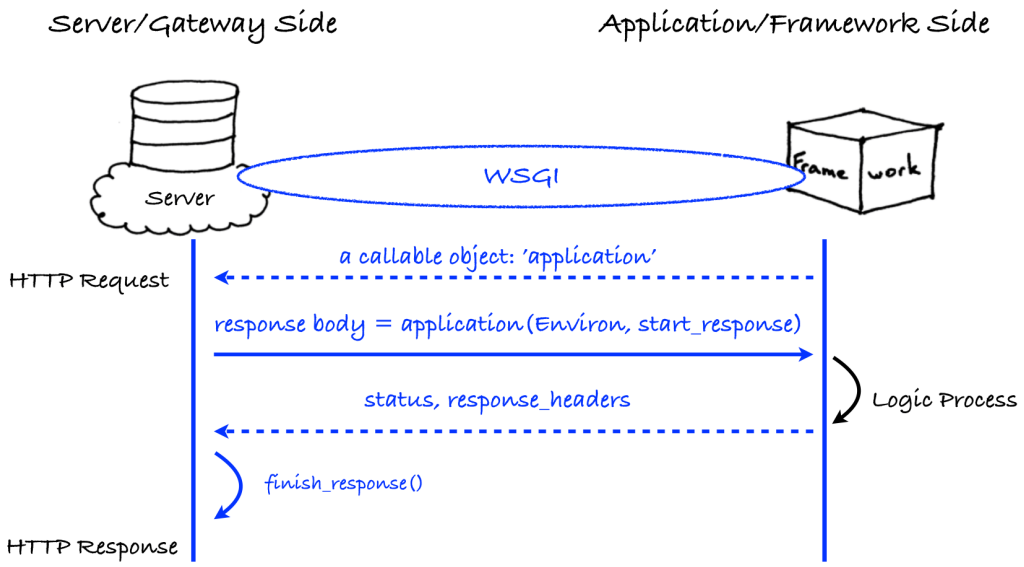


WSGI 适配

原文链接：[从零开始搭建论坛（一）：Web服务器与Web框架]

从零开始搭建论坛（二）：Web服务器网关接口

1. Java有很多 Web 框架，因为有 servlet API，任何Java Web框架写的应用程序都可以运行在任意一个 Web Server 上。
2. Python有WSGI适配Web服务器和应用程序. WSGI（Python Web Server Gateway Interface）.
3. WSGI: Web服务器和Web应用程序的桥梁
    1. Web server 接收原始 HTTP 数据，处理成统一格式后交给 Web 应用程序
    2. 应用程序 / 框架 处理业务逻辑，生成响应内容后交给服务器。
4. Web服务器和框架通过 WSGI耦合过程如下图：



WSGI Server 适配

5. WSGI耦合过程:
    1. 应用程序（网络框架）有`application`名称的可调用对象（WSGI协议没有指定如何实现这个对象）。
    2. 服务器每次接收HTTP客户端请求之后，调用`application`，调用时传递`environ`名称的字典类型的参数，和`start_response`名称的可调用对象。
    3. 框架/应用生成 HTTP状态码和HTTP响应报头，并传递二者给`start_response`，等待服务器保存。此外，框架/应用还返回响应的正文。
    4. 服务器组合状态码、响应报头和响应正文为HTTP响应，并返回给客户端（这一步不属于WSGI协议）。

下面分别从服务器端和应用程序端 看看 WSGI 是如何做适配。

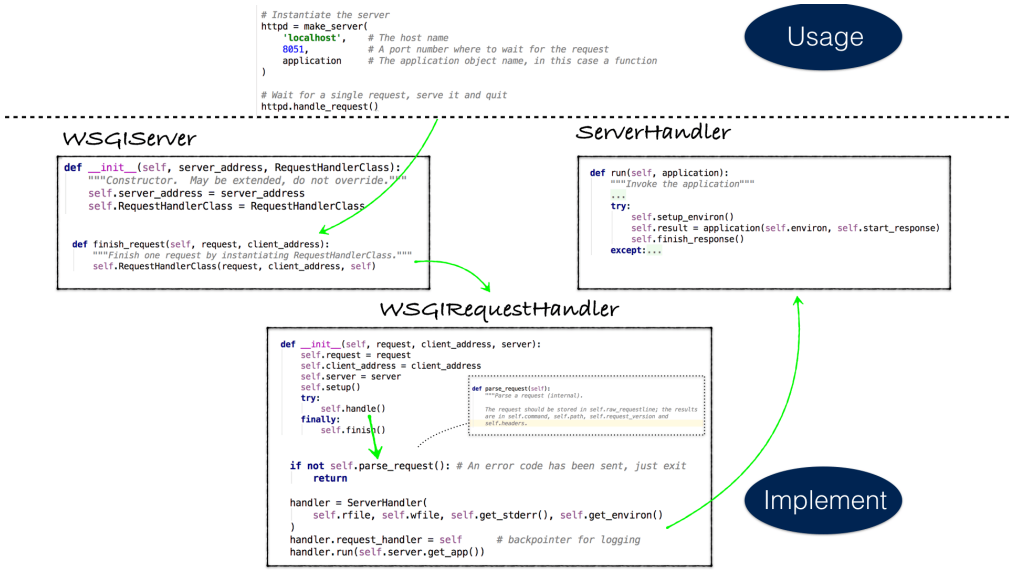
服务器端

1. 客户端（通常是浏览器）的每个HTTP请求: 请求行、消息报头、请求正文三部分.包含了本次请求的细节内容。如：
    1. Method：指出Request-URI标识的资源执行的方法，包括GET，POST 等
    2. User-Agent：客户端的操作系统、浏览器和其它属性. 把这些信息告诉服务器；
2. 服务器接收客户端HTTP请求之后，`WSGI 接口`对请求字段统一化处理，方便传给应用服务器接口（其实是给框架）。
    1. Web服务器传递哪些数据给应用程序?
        1. CGI（Common Gateway Interface，通用网关接口）详细规定. 这些数据 叫`CGI 环境变量`。
        2. WSGI 沿用 CGI 环境变量, 要求 Web 服务器创建一个字典保存CGI环境变量（将其命名为`environ`）。
        3. environ 还保存一些WSGI定义的变量，还保存一些客户端系统的环境变量，可参考 environ Variables 看看具体的变量。
3. WSGI 接口将 `environ` 交给应用程序.
    1. WSGI 规定应用程序提供`application`(一个可调用对象)， 服务器调用 application. application返回值为HTTP响应正文。
    2. 服务器调用 `application` ，要提供两个变量

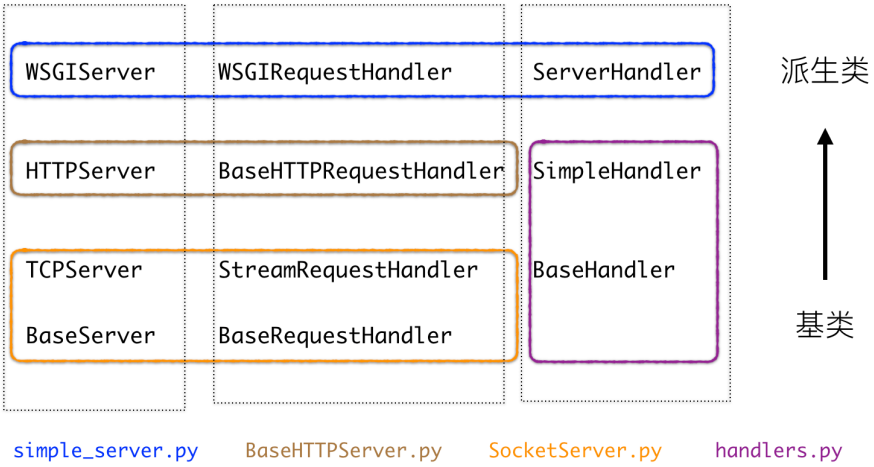
1. 一个变量字典`environ`
2. 一个可调用对象`start_response`. 它产生状态码和响应头，
3. 这样得到了完整的HTTP响应。
4. Web 服务器将响应返回给客户端，一次完整的`HTTP请求－响应`完成了。

wsgiref 分析

1. Python内置了一个实现了WSGI接口的 Web 服务器，在模块wsgiref中. 它是纯Python编写的WSGI服务器. 简单分析它的实现。
    1. 首先 用下面代码启动一个 Web 服务器：
        1. ```
           # Instantiate the server
           httpd = make_server(
               'localhost',    # The host name
               8051,           # A port number where to wait for the request
               application     # The application object name, in this case a function
           )

           # Wait for a single request, serve it and quit
           httpd.handle_request()
           ```
    2. 然后 Web服务器接收一个请求、生成 environ，然后调用 application 处理请求. 沿这条主线分析源码的调用过程. 如下图：
        1. 

           WSGI Server调用流程
    2. WSGI Server调用流程
        1. WSGIServer 是Web服务器类, 提供server_address (IP:Port) 和 WSGIRequestHandler 类初始化获得一个server对象。
        2. server对象监听响应的端口，收到HTTP请求后用 finish_request 创建一个 RequestHandler 类实例， 该实例初始化时生成一个 Handle 类实例，
        3. 然后Handle 类实例调用 run(application) 函数.
        4. run函数再 application对象来生成响应。
    3. 三个类，WSGIServer，WSGIRequestHandler，ServerHandle。
        1. 继承关系如下图所示：
           ```

1. 

   WSGI 类继承关系图

2. 此部分和 WSGI 接口关系不大，更多的是 Web 服务器的具体实现，可以忽略

   1. TCPServer 用 socket 完成 TCP 通信
   2. HTTPServer 处理 HTTP
   3. StreamRequestHandler 处理 stream socket
   4. BaseHTTPRequestHandler 处理 HTTP 层面的内容

微服务器实例

1. 实现一个微小的 Web 服务器，理解 Web 服务器端 WSGI 接口的实现。代码摘自 自己动手开发网络服务器（二），放在 gist 上， 结构如下：

   1.
```
class WSGIServer(object):
    # 套接字参数
    address_family, socket_type = socket.AF_INET, socket.SOCK_STREAM
    request_queue_size = 1

    def __init__(self, server_address):
        # TCP 服务端初始化：创建套接字，绑定地址，监听端口
        # 获取服务器地址，端口

    def set_app(self, application):
        # 获取框架提供的 application
        self.application = application

    def serve_forever(self):
        # 处理 TCP 连接：获取请求内容，调用处理函数

    def handle_request(self):
        # 解析 HTTP 请求，获取 environ，处理请求内容，返回HTTP响应结果
        env = self.get_environ()
        result = self.application(env, self.start_response)
        self.finish_response(result)

    def parse_request(self, text):
        # 解析 HTTP 请求

    def get_environ(self):
        # 分析 environ 参数，这里只是示例，实际情况有很多参数。
        env['wsgi.url_scheme']   = 'http'
        ...
        env['REQUEST_METHOD']    =  self.request_method    # GET
        ...
```

```
        return env

    def start_response(self, status, response_headers, exc_info=None):
        # 添加响应头，状态码
        self.headers_set = [status, response_headers + server_headers]

    def finish_response(self, result):
        # 返回 HTTP 响应信息

SERVER_ADDRESS = (HOST, PORT) = '', 8888

# 创建一个服务器实例
def make_server(server_address, application):
    server = WSGIServer(server_address)
    server.set_app(application)
    return server
```

2. 支持 WSGI Web服务器很多
    1. Gunicorn相当不错的一个。
    2. 它脱胎于ruby社区的Unicorn， 移植到python ，成为一个WSGI HTTP Server。
    3. 它有以下优点：
        1. 容易配置
        2. 自动管理多个worker进程
        3. 选择不同的后台扩展接口（ sync, gevent, tornado等 ）

应用程序端（框架）

1. 相比服务器端，应用程序端（即框架） 做的事情很简单.
    1. 它提供一个可调用对象（ application ）
        1. 可调用对象 可以是函数， 可以是类（下面第二个示例）或 拥有 `__call__` 方法的实例
        2. 总之, 可以接受参数 environ 和 start_response ， 返回值 被服务器 迭代即可。
    2. application对象接收服务器端传递的参数 environ 和 start_response。
2. Application 具体做什么
    1. 根据 environ 提供的 HTTP 请求的信息 处理业务，返回一个可迭代对象.
        1. 服务器端通 迭代这个对象， 获得 HTTP 响应的正文。 没有响应正文， 返回None。
    2. 调用服务器提供的 start_response，产生HTTP响应的状态码和响应头，原型如下：
        1. `def start_response(self, status, headers,exc_info=None):`
            1. Application 提供 status：一个字符串，表示HTTP响应状态字符串
            2. Application 提供 response_headers: 一个列表，
                1. 如元组：(header_name, header_value)， 表示HTTP响应的headers
            3. exc_info 可选的，出错时用，server需要返回给浏览器的信息。
3. 实现一个简单的 application, 如下所示：
    1.
```
def simple_app(environ, start_response):
    """Simplest possible application function"""
    HELLO_WORLD = "Hello world!\n"
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)
    return [HELLO_WORLD]
```

4. 类实现 application

    1. ```
       class AppClass:
           """Produce the same output, but using a class"""

           def __init__(self, environ, start_response):
               self.environ = environ
               self.start = start_response

           def __iter__(self):
               ...
               HELLO_WORLD = "Hello world!\n"
               yield HELLO_WORLD
       ```

    2. `AppClass` 类本身是 application

        1. 用 environ 和 start_response 实例化 它, 返回一个实例对象
        2. 这个实例对象 是可迭代的，符合 WSGI 对 application 的要求。

    3. 若用 AppClass 类的对象作 application

        1. 必须给类添加 `__call__` 方法，接受 environ 和 start_response 为参数，返回可迭代对象
        2. 如下所示：

            1. ```
               class AppClass:
                   """Produce the same output, but using an object"""
                   def __call__(self, environ, start_response):
                       ...
               ```

    4. 这部分涉及 python 高级特性， 如 yield 和 magic method，可以参考python语言要点来理解。

Flask 中的 WSGI

1. flask 一个轻量级的Python Web框架，符合 WSGI 规范。

    1. 最初版本只有 600 多行， 便于理解。

2. 它最初版本 关于 WSGI 接口的部分。

    1. ```
       def wsgi_app(self, environ, start_response):
           """The actual WSGI application.

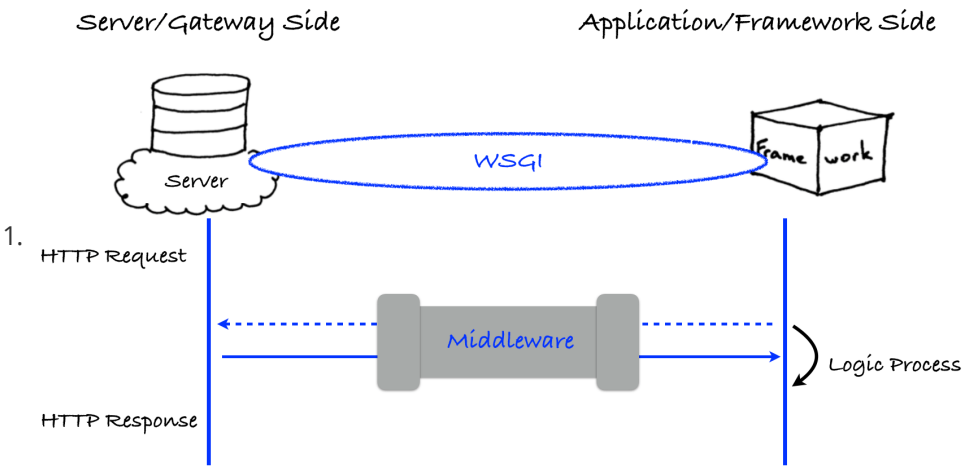
           This is not implemented in `__call__` so that middlewares can be
       applied:
               app.wsgi_app = MyMiddleware(app.wsgi_app)
           """
           with self.request_context(environ):
               rv = self.preprocess_request()
               if rv is None:
                   rv = self.dispatch_request()
               response = self.make_response(rv)
               response = self.process_response(response)
               return response(environ, start_response)


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

2. 这里的 wsgi_app 实现了 application 功能
    1. rv 是 对请求的封装
    2. response 是框架处理业务逻辑的具体函数。

中间件

1. flask 代码 wsgi_app 函数的注释中提到不直接在 `__call__` 中实现 application 部分， 为了使用 `中间件`。
2. 为什么要使用中间件?
    1. server 端调用 application处理HTTP 请求， 并返回 application 处理后的结果。这解决一般的场景了，但不完善
    2. 考虑下面的几种应用场景：
        1. 不同的请求（如不同的 URL），server 要调用不同的 application，如何选择调用哪个呢；
        2. 做负载均衡或远程处理，要用网络上其他主机的 application；
        3. 处理 application 返回的内容 才能作为 HTTP 响应；
        4. 这些场景共同点: 一些操作放在服务端还是应用（框架）端都不合适。
            1. 对应用端，这些操作应由服务器端做
            2. 对服务器端，这些操作应由应用端来。
        5. 为了处理这种情况，引入了`中间件`。
3. 中间件是什么呢？
    1. 像应用端和服务端的桥梁，沟通两边。
        1. 对服务器端，中间件的表现像是应用端
        2. 对应用端说，中间件的表现像是服务器端。
        3. 如下图所示：



中间件

中间件的实现

1. flask 框架在 Flask 类的初始化中 使用了中间件：
    1. `self.wsgi_app = SharedDataMiddleware(self.wsgi_app, { self.static_path: target })`
    2. 作用和 python 的装饰器一样，

    1. 执行 self.wsgi_app 前后执行 SharedDataMiddleware。
    2. 中间件类似python中装饰器。
3. SharedDataMiddleware 中间件
    1. 由 werkzeug 库提供
    2. 支持站点托管静态内容
    3. 支持根据不同的请求，调用不同的 application. 解决前面场景 1, 2 中的问题了。
2. DispatcherMiddleware 的实现：
    1. ```
       class DispatcherMiddleware(object):
           """Allows one to mount middlewares or applications in a WSGI
       application.
           This is useful if you want to combine multiple WSGI applications::
               app = DispatcherMiddleware(app, {
                   '/app2':        app2,
                   '/app3':        app3
               })
           """

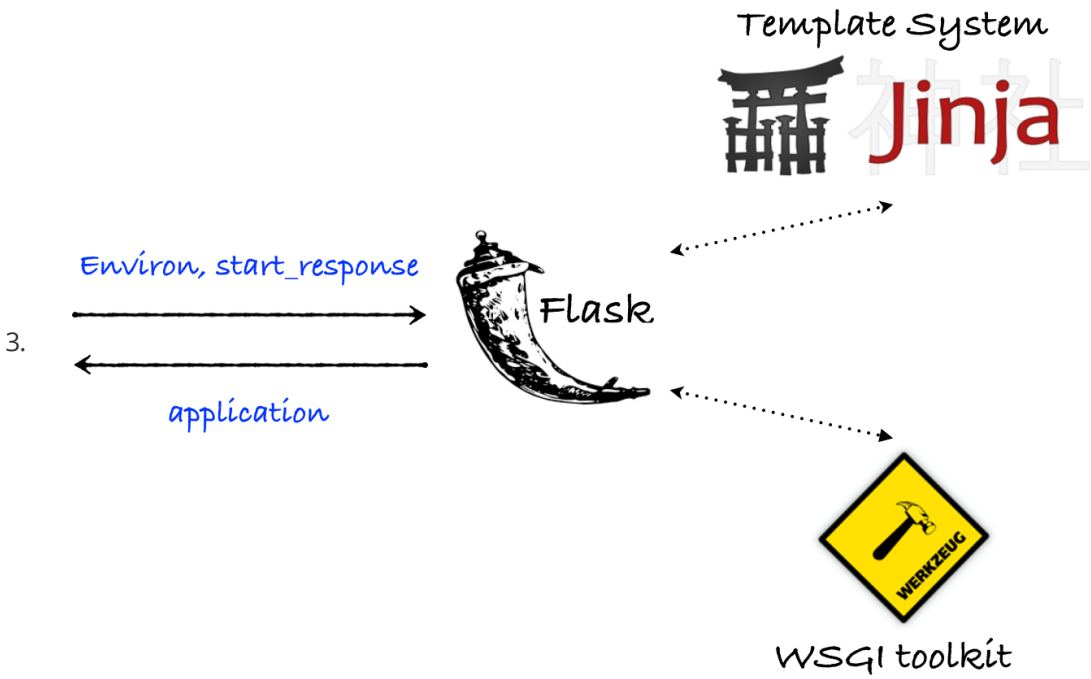
           def __init__(self, app, mounts=None):
               self.app = app
               self.mounts = mounts or {}

           def __call__(self, environ, start_response):
               script = environ.get('PATH_INFO', '')
               path_info = ''
               while '/' in script:
                   if script in self.mounts:
                       app = self.mounts[script]
                       break
                   script, last_item = script.rsplit('/', 1)
                   path_info = '/%s%s' % (last_item, path_info)
               else:
                   app = self.mounts.get(script, self.app)
               original_script_name = environ.get('SCRIPT_NAME', '')
               environ['SCRIPT_NAME'] = original_script_name + script
               environ['PATH_INFO'] = path_info
               return app(environ, start_response)
       ```
    2. 初始化中间件 要提供一个 mounts 字典，
        1. 指定不同 URL 路径到 application 的映射关系。
        2. 对一个请求，中间件检查其路径，然后选择合适的 application处理。

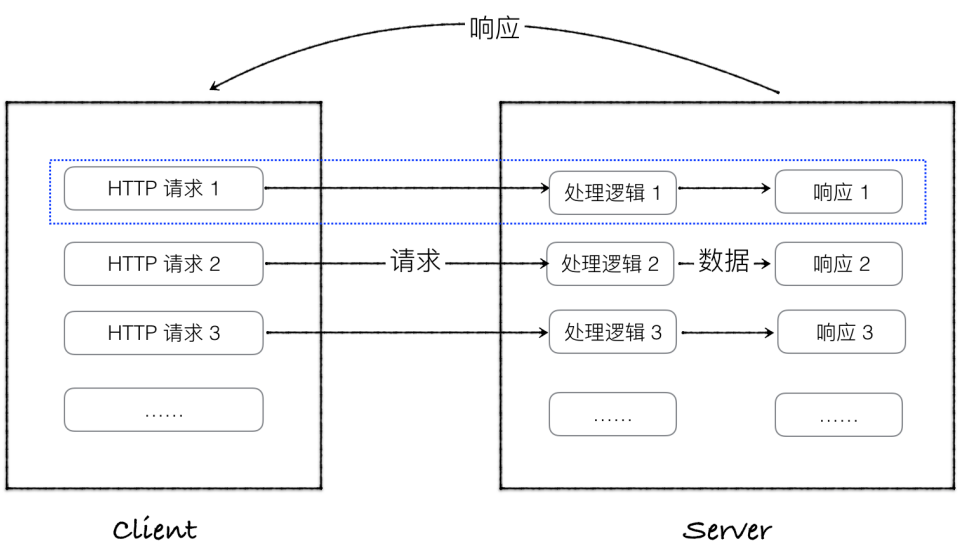
WSGI 的原理部分基本结束，下一篇我会介绍下对 flask 框架的理解。

从零开始搭建论坛（三）：Flask框架简单介绍

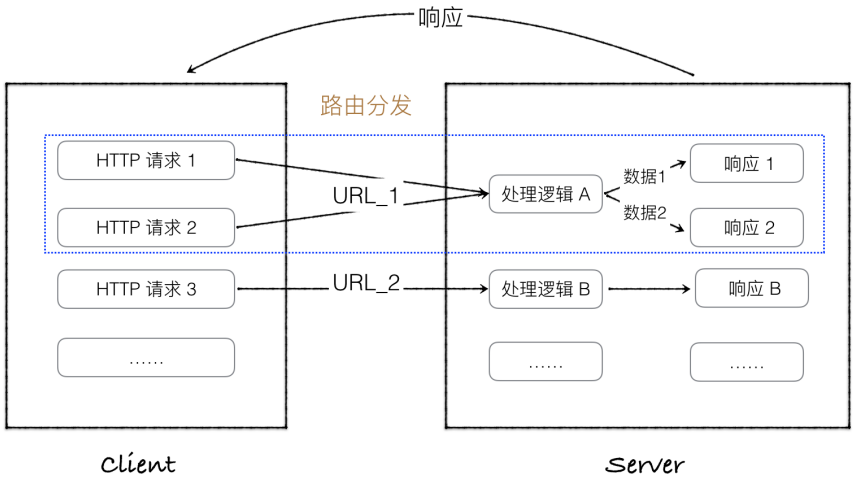
1. Web框架 将不同Web应用程序的共性部分给抽象出来，提供通用的接口，避免开发者做重复性工作
2. Flask :
    1. Web 框架
    2. 微框架，众多的拥护者，文档齐全，社区活跃度高。
       ```



Flask 框架

1. Web应用程序的一般流程。对于Web应用来说，
    1. 客户端要获取**动态资源**，会发起一个HTTP请求（如浏览器访问一个 URL），
    2. Web应用程序在后台处理相应的业务:
        1. (从数据库或计算操作等）取出数据
        2. 生成HTTP响应
    3. 如果访问静态资源，则直接返回资源即可，不用处理业务）。
    4. 过程如下
        1. 

        2. 不同的请求可能调用相同的处理逻辑
            1. 相同业务处理逻辑的 HTTP 请求可以用一类 URL 标识。
                1. 如论坛站点中，所有的获取Topic内容的请求，可以用 `topic/<topic_id>/` 这类URL表示. topic_id 区分不同的topic。
                2. 后台定义一个 `get_topic(topic_id)` 的函数，获取topic的数据
            2. 建立URL和函数的一一对应关系。即: Web开发中所谓的 路由分发.
                1. 路由分发 图示:



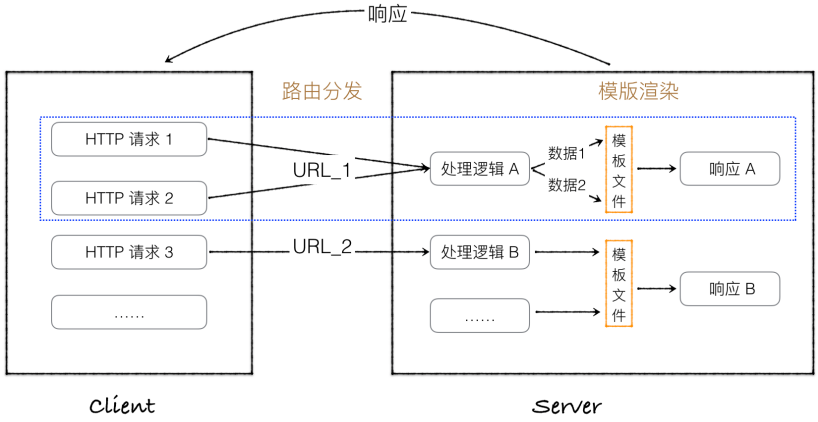
路由分发

2. Flask底层用 werkzeug 做路由分发，代码如下：

    1. ```
       @app.route('/topic/<int:topic_id>/')
       def get_topic(topic_id):
           # Do some cal or read from database
           # Get the data we need.
       ```

3. 业务逻辑函数拿到数据后， 根据这些数据生成HTTP响应（对Web应用，HTTP响应是一个HTML文件）。

    1. Web开发的做法: 提供 HTML 模板文件， 将数据传入模板， 渲染后得到 HTML 响应文件。
    2. 场景: 请求不同，但响应中数据的展示方式是相同的.
    3. 以论坛为例
        1. 不同topic, 其topic content不同，但页面展示的方式是一样的，都有标题拦，内容栏等。
        2. 对于 topic, 只需提供一个HTML模板，然后传入不同topic数据，即得到不同的HTTP响应。即**模板渲染**，
    4. 模板渲染 图示：

    1. 

    模板渲染

4. Flask 用 Jinja2 模板渲染 引擎 做模板渲染，代码如下：
       ```

1.
```
@app.route('/topic/<int:topic_id>/')
def get_topic(topic_id):
    # Do some cal or read from database
    # Get the data we need.
    return render_template('path/to/template.html',
data_needed)
```

2. 总结: Flask处理一个请求的流程
    1. 首先根据 URL 决定处理函数
    2. 然后函数操作，取得所需的数据。
    3. 再将数据传给相应的模板文件，由 Jinja2 渲染得到 HTTP 响应内容
    4. 然后由Flask返回响应内容。

Flask 入门

1. 关于 Flask 框架的学习
    1. 不建议直接读官网文档，虽然这是一手的权威资料，但并不适合初学者入手
2. 几个学习资料
    1. 汇智网flask框架教程：
        1. 精简教程, 七部分：
            1. 快速入门
            2. 路由：URL 规则与视图函数
            3. 请求、应答与会话
            4. 上下文对象：Flask 核心机制
            5. 模版：分离数据与视图
            6. 访问数据库：SQLAlchemy简介
            7. 蓝图：Flask应用组件化
        2. 总结了 Flask 最核心的内容，还提供了简单的在线练习环境，方便一边学习理论一边动手实践。
    2. 麦子学院也有一个 Flask入门 视频教程，
        1. 一共8小时的视频教程，涵盖flask web 开发的方方面面
            1. 环境的搭建
            2. flask 语法介绍
            3. 项目结构的组织
            4. flask 全球化
            5. 单元测试等内容。
        2. 视频作者有 17 年软件开发经验，曾任微软深圳技术经理及多家海外机构担任技术顾问，够牛！视频讲的也确实不错。
    3. 还可以看 Flask Web开发：基于Python的Web应用开发实战
        1. 这本有着 8.6 评分的书，相信没看完就跃跃欲试想写点什么了。
    4. Github 上当然也有 awesome-flask了，想深入学习flask的话，这里不失为一个好的资源帖。

本篇大概谈了下 Flask 的路由分发和模版渲染，下篇我们会继续讲Flask使用中的一些问题。