后端是什么

读《 SQL 反模式》
https://zhuanlan.zhihu.com/p/37534634

数据库第一二三范式到底在说什么
https://zhuanlan.zhihu.com/p/20028672

https://github.com/pingcap/awesome-database-learning

https://github.com/alibaba/p3c/tree/master/p3c-gitbook/MySQL%E6%95%B0%E6%8D%AE%E5%BA%93

我在前面梳理了前端一些知识，最后看到网友的一句总结，让我十分感叹，
今天 npm， 明天 yarn， 后天 pnpm
今天 rollup， 明天 webpack，后天 vite
今天 angular， 明天 vue ，后天 react
今天 moment，明天 dayjs ，后天 data-fns
今天 redux， 明天 mobx，后天 hook+context
今天 express ，明天 koa，后天 nestjs

有没有可能我全要！前端发展令人望而止步。

后端要做什么

之前我重新又学了一遍http协议，发现后端的核心就在于此。详情如下：

> ## 1.http协议是什么
>
> 英文名：HyperText Transfer Protocol，中文名：超文本传输协议
>
> **超文本传输协议**，它可以拆成三个部分，分别是：超文本，传输和协议
>
> - 超文本
>   - 所谓文本（Text），就表示 HTTP 传输的不是 TCP/UDP 这些底层协议里被切分的杂乱无章的二进制包（datagram），而是完整的、有意义的数据，可以被浏览器、服务器这样的上层应用程序处理。在互联网早期，“文本”只是简单的字符文字，但发展到现在，文本的涵义已经被大大地扩展了，图片、音频、视频、甚至是压缩包，在 HTTP 眼里都可以算做是“文本。所谓超文本，就是超越了普通文本的文本，它是

> 文字、图片、音频和视频等的混合体，最关键的是含有“超链接”，能够从一个超文本跳跃到另一个超文本，形成复杂的非线性、网状的结构关系。对于超文本，我们最熟悉的就应该是 HTML 了，它本身只是纯文字文件，但内部用很多标签定义了对图片、音频、视频等的链接，再经过浏览器的解释，呈现在我们面前的就是一个含有多种视听信息的页面

- 传输
  - 数据虽然是在 A 和 B 之间传输，但并没有限制只有 A 和 B 这两个角色，允许中间有**中转**或者**接力**
- 协议
  - 双方必须共同遵从的一组约定，它应该包括语法，语义，同步规则和错误处理
  - http是一种网络传输协议
  - HTTP **协议是一个双向协议**(A<=>B)
  - HTTP **不是一个孤立的协议**
    - 在互联网世界里，HTTP 通常跑在 TCP/IP 协议栈之上，依靠 IP 协议实现寻址和路由、TCP 协议实现可靠数据传输、DNS 协议实现域名查找、SSL/TLS 协议实现安全通信。此外，还有一些协议依赖于 HTTP，例如 WebSocket、HTTPDNS 等。这些协议相互交织，构成了一个协议网，而 HTTP 则处于中心地位

Web 服务器

http的响应方

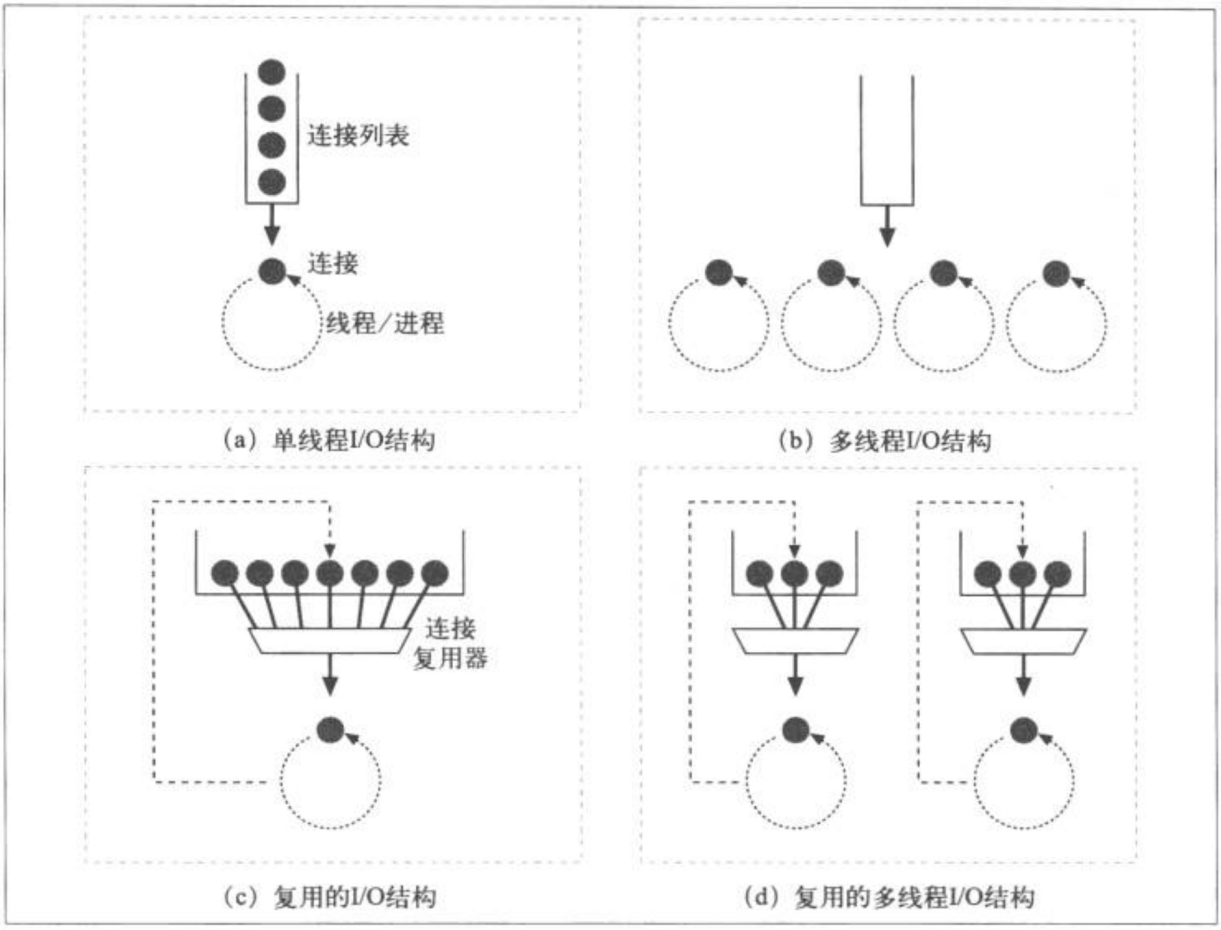
当我们谈到Web 服务器时有两个层面的含义：硬件和软件。

硬件含义就是物理形式或云形式的机器，在大多数情况下它可能不是一台服务器，而是利用反向代理、负载均衡等技术组成的庞大集群。但从外界看来，它仍然表现为一台机器，但这个形象是虚拟的。

软件含义的 Web 服务器可能我们更为关心，它就是提供 Web 服务的应用程序，通常会运行在硬件含义的服务器上。它利用强大的硬件能力响应海量的客户端 HTTP 请求，处理磁盘上的网页、图片等静态文件，或者把请求转发给后面的 Tomcat、Node.js 等业务应用，返回动态的信息

为什么说核心在于htpp协议，前端（C）和后端（S）属于C/S（Client-Server）架构的服务，准确的来讲现在的前后端分离应该是B/S（Browser/Server）架构。大致的功能如下：

- 服务器（Server）关注数据处理。用集群、分布式、微服务等技术来提高处理速度和稳定性。
- 客户端（Client）关注用户界面。因为客户端直接与人交互，涉及到人们的各种需求，所以发展的非常丰富：桌面应用、浏览器、手机App、智能穿戴设备App、小程序等等。
  关于B/S和C/S的区别，我比较认同，B/S是C/S的一种实现方式。
  既然要请求和响应数据，那就必须涉及到传输的协议，所以不论是前端还是后端，都要学习HTTP协议，一个要知道如何发请求，一个要知道怎么接受请求以及返回所请求的数据。
  我们每天上网，HTTP协议我们其实每天都在使用，可能当我们习惯性的访问网站的时候，直接WWW，其实默认的背后，都会在前面加上`https://`标识协议。
  如果觉得理论枯燥，可以看看下面形象科普网络知识，为什么需要：
- https://mp.weixin.qq.com/s?__biz=Mzk0MjE3NDE0Ng==&mid=2247489907&idx=1&sn=a296cb42467cab6f0a7847be32f52dae&chksm=c2c663def5b1eac84b664c8c1cadf1c8ec23ea2e57e48e04add9b833c841256fc9449b62c0ec&scene=21#wechat_redirect
- https://www.zhihu.com/question/39541968
  我们已经知道用什么传输的协议，在来看看另外一个核心的东西就是IO，详细可参考：https://zhang-jane.github.io/post/c3d01f49.html#socket。由于IO涉及到操作系统的核心，所以不同操作系统IO模型也不一样。
  服务器处理的模型：



a. 单线程的模型，服务器一次只能处理一个请求，直到完成为止。它就像一个队列，排队一个个处理，缺点很明显。

b. 多进程以及多线程的任务，可以同时处理多个请求，但是假设有大量的千万百万的，会对服务器资源消耗殆尽。

c. 复用IO模型，同时监听所有在连接上的活动，当连接的状态发生变化的时候，就进行处理，处理之后，放回到连接器中，等待下一次变化。这样只有有事情的时候才会去进行连接处理，没事就会空闲等待。

d. 复用的多线程IO模型，利用多核cpu和多线程同时处理。

简单看看web服务器处理的流程图：

在实现 HTTP 事务时所应执行的步骤。

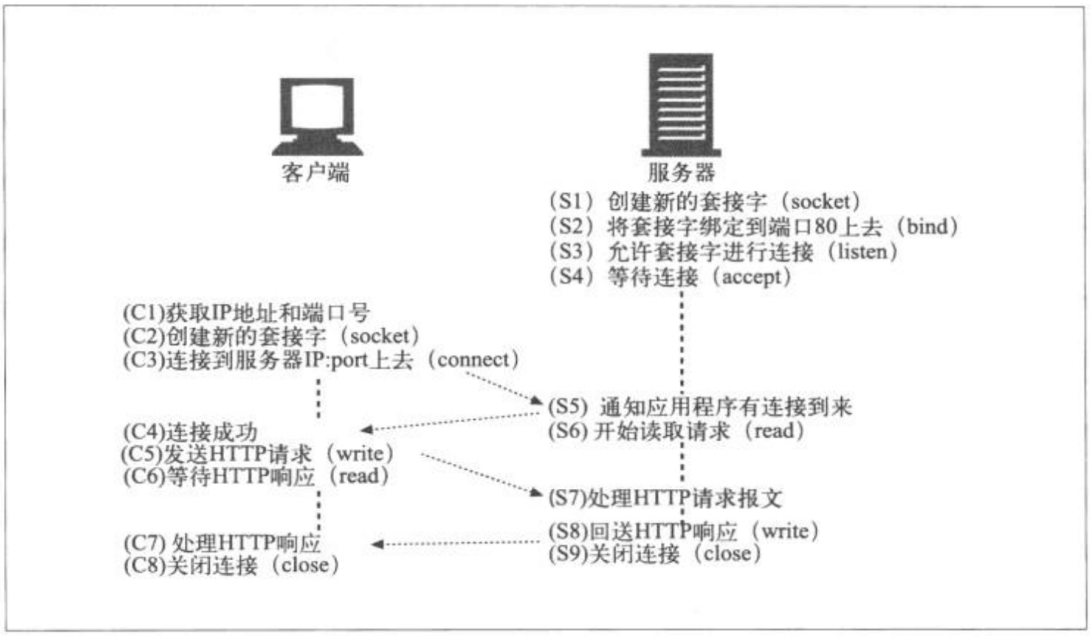
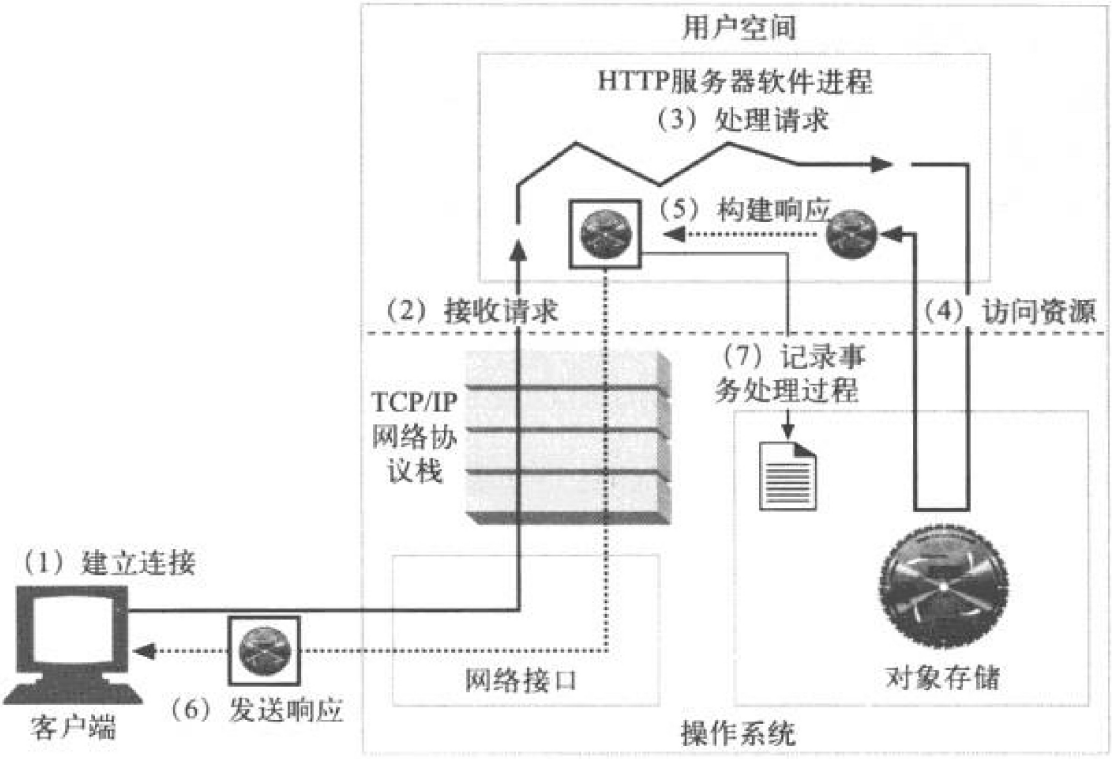


图 4-6　TCP 客户端和服务器是如何通过 TCP 套接字接口进行通信的

我们从 Web 服务器等待连接（参见图 4-6，S4）开始。客户端根据 URL 判定出 IP 地址和端口号，并建立一条到服务器的 TCP 连接（参见图 4-6，C3）。建立连接可能要花费一些时间，时间长短取决于服务器距离的远近、服务器的负载情况，以及因特网的拥挤程度。

一旦建立了连接，客户端就会发送 HTTP 请求（参见图 4-6，C5），服务器则会读取请求（参见图 4-6，S6）。一旦服务器获取了整条请求报文，就会对请求进行处理，执行所请求的动作（参见图 4-6，S7），并将数据写回客户端。客户端读取数据（参见图 4-6，C6），并对响应数据进行处理（参见图 4-6，C7）。

4.2 对TCP性能的考虑



了解了一堆原理，接下来看看如何具体实现。

scoket编程

https://mp.weixin.qq.com/s/VazobOgY9QVaADpEur81ng
`socket`进行编程。

于是第一步就是创建个关于TCP的`socket`。就像下面这样。

```
sock_fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

这个方法会返回`socket_fd`，它是socket文件的句柄，是个数字，相当于socket的身份证号。

得到了`socket_fd`之后，对于服务端，就可以依次执行`bind()`,`listen()`,`accept()`方法，然后坐等客户端的连接请求。

对于客户端，得到`socket_fd`之后，你就可以执行`connect()`方法向服务端发起建立连接的请求，此时就会发生TCP三次握手。

code
先从python开始

在学习python后端框架的时候，他有一个python独有的概念叫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，翻译https://github.com/mainframer/PEP333-zh-CN#goal。

为什么要WSGI（pep3333）

Python 目前拥有各种各样的 Web 应用进程框架，例如 Zope、 Quixote、 Webware、 SkunkWeb、 PSO 和 Twisted Web——仅举几例 [1]。如此广泛的选择对于 Python 新手来说可能是个问题，因为一般来说，他们对 Web 框架的选择会限制他们对可用 Web 服务器的选择，反之亦然。

相比之下，尽管 Java 有同样多的可用 Web 应用进程框架，但 Java 的“servlet” API (https://www.liaoxuefeng.com/wiki/1252599548343744/1304265949708322)使用任何 Java Web 应用进程框架编写的应用进程可以在支持 servlet API 的任何 Web 服务器上运行。

这种 API 在 Python 的 Web 服务器中的可用性和广泛使用——无论这些服务器是用 Python 编写的（例如 Medusa）、嵌入 Python（例如 mod_python）还是通过网关协议（例如 CGI、FastCGI 等）调用 Python – 将框架的选择与 Web 服务器的选择分开，让用户自由选择适合他们的配对，同时让框架和服务器开发人员能够专注于他们喜欢的专业领域。

因此，本 PEP 提出了 Web 服务器和 Web 应用进程或框架之间的简单通用接口：Python Web 服务器网关接口 (WSGI)。

wsgi提供我们的标准详情：
WSGI 接口有两个方面：“服务器”或“网关”方面，以及“应用进程”或“框架”方面。服务器端调用应用进程端提供的可调用对象。如何提供该对象的细节取决于服务器或网关。假定某些服务器或网关将要求应用进程的部署者编写一个简短的脚本来创建服务器或网关的实例，并为其提供应用进程对象。其他服务器和网关可以使用配置文档或其他机制来指定应从何处导入或以其他方式获取应用进程对象。

Details:
The application object must accept two positional arguments. For the sake of illustration, we have named them environ and start_response, but they are not required to have these names. A server or gateway must invoke the application object using positional (not keyword) arguments.(E.g. by calling result = application(environ, start_response) as shown above.)

The environ parameter is a dictionary object, containing CGI-style environment variables. This object must be a builtin Python dictionary (not a subclass, UserDict or other dictionary emulation), and the application is allowed to modify the dictionary in any way it desires. The dictionary must also include certain WSGI-required variables (described in a later section), and may also include server-specific extension variables, named according to a convention that will be described below.

The start_response parameter is a callable accepting two required positional arguments, and one optional argument. For the sake of illustration, we have named these arguments status, response_headers, and exc_info, but they are not required to have these names, and the application must invoke the start_response callable using positional arguments (e.g. start_response(status, response_headers)).
The status parameter is a status string of the form "999 Message here", and response_headers is a list of (header_name, header_value) tuples describing the

HTTP response header. The optional exc_info parameter is described below in the sections on The start_response() Callable and Error Handling. It is used only when the application has trapped an error and is attempting to display an error message to the browser.

The start_response callable must return a write(body_data) callable that takes one positional parameter: a bytestring to be written as part of the HTTP response body. (Note: the write() callable is provided only to support certain existing frameworks' imperative output APIs; it should not be used by new applications or frameworks if it can be avoided. See the Buffering and Streaming section for more details.)

When called by the server, the application object must return an iterable yielding zero or more bytestrings. This can be accomplished in a variety of ways, such as by returning a list of bytestrings, or by the application being a generator function that yields bytestrings, or by the application being a class whose instances are iterable. Regardless of how it is accomplished, the application object must always return an iterable yielding zero or more bytestrings.

The server or gateway must transmit the yielded bytestrings to the client in an unbuffered fashion, completing the transmission of each bytestring before requesting another one. (In other words, applications should perform their own buffering. See the Buffering and Streaming section below for more on how application output must be handled.)

The server or gateway should treat the yielded bytestrings as binary byte sequences: in particular, it should ensure that line endings are not altered. The application is responsible for ensuring that the bytestring(s) to be written are in a format suitable for the client. (The server or gateway may apply HTTP transfer encodings, or perform other transformations for the purpose of implementing HTTP features such as byte-range transmission. See Other HTTP Features, below, for more details.)

If a call to len(iterable) succeeds, the server must be able to rely on the result being accurate. That is, if the iterable returned by the application provides a working **len**() method, it must return an accurate result. (See the Handling the Content-Length Header section for information on how this would normally be used.)

If the iterable returned by the application has a close() method, the server or gateway must call that method upon completion of the current request, whether the request was completed normally, or terminated early due to an application

error during iteration or an early disconnect of the browser. (The close() method requirement is to support resource release by the application. This protocol is intended to complement PEP 342's generator support, and other common iterables with close() methods.)

Applications returning a generator or other custom iterator should not assume the entire iterator will be consumed, as it may be closed early by the server.

(Note: the application must invoke the start_response() callable before the iterable yields its first body bytestring, so that the server can send the headers before any body content. However, this invocation may be performed by the iterable's first iteration, so servers must not assume that start_response() has been called before they begin iterating over the iterable.)

Finally, servers and gateways must not directly use any other attributes of the iterable returned by the application, unless it is an instance of a type specific to that server or gateway, such as a "file wrapper" returned by wsgi.file_wrapper (see Optional Platform-Specific File Handling). In the general case, only attributes specified here, or accessed via e.g. the PEP 234 iteration APIs are acceptable.

- 1.它是一个可调用的对象，在python中应该是是类，或者实现了**call**方法的类的实例
- 2.这个可调用的对象应该接受，environ(包含了http请求信息的dict对象)和start_response(可调用对象，三个参数，status-状态码，response_headers-包含信息的二元组列表，exc_info-异常信息)
- 上代码，最简单的WSGI server为Python自带的wsgiref.simple_server
  - `from wsgiref.simple_server import make_server`
  - `from webob import Request, Response` 封装请求和响应的数据
  - `from webob.dec import wsgify` 装饰器，把一个普通函数或者类封装成application，需要传入一个request，返回一个response
  - `def simple_app(environ, start_response):`
  - `httpd = make_server("0.0.0.0", 9000, simple_app)`

  demo：

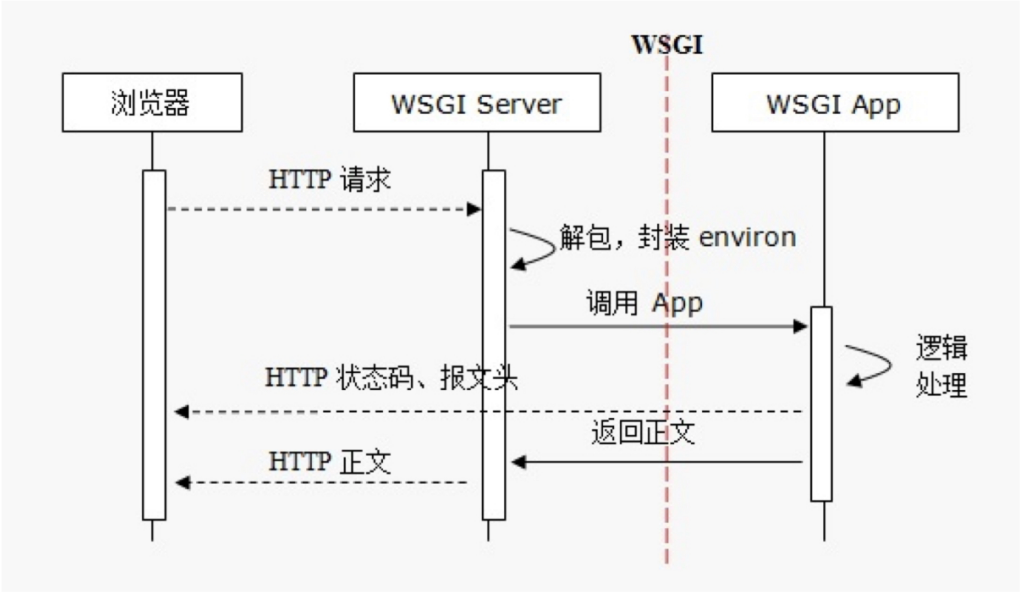
```python
from wsgiref.simple_server import make_server,
demo_app
```

```
HELLO_WORLD = b"Hello world!\n"
server_ip = "127.0.0.1"
port = 9991

# 自己实现application
def simple_app(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type',
'text/plain')]
    start_response(status, response_headers)
    return [HELLO_WORLD]

server = make_server(server_ip, port, demo_app)

server.serve_forever() # handle_request() 一次
"""
http 127.0.0.1:9991
HTTP/1.0 200 OK
Content-Length: 13
Content-type: text/plain
Date: Wed, 28 Dec 2022 08:12:32 GMT
Server: WSGIServer/0.2 CPython/3.9.0

Hello world!

"""
```

```
HELLO_WORLD = b"Hello world!\n"

def simple_app(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type',
'text/plain')]
    start_response(status, response_headers)
```

```
    return [HELLO_WORLD]

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

    (Note: 'AppClass' is the "application" here, so
calling it
    returns an instance of 'AppClass', which is then
the iterable
    return value of the "application callable" as
required by
    the spec.

    If we wanted to use *instances* of 'AppClass' as
application
    objects instead, we would have to implement a
'__call__'
    method, which would be invoked to execute the
application,
    and we would need to create an instance for use by
the
    server or gateway.
    """

    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
```

WSGI



可能你已经注意到：
The WSGI interface has two sides: the “server” or “gateway” side, and the “application” or “framework” side.
WSGI提供了服务或者网关和应用或框架。如上图所示，WSGI服务来处理浏览器发送过来的请求，WSGI应用来处理逻辑，比如crud，权限管理等

总结

web server：

- 实际上，tcp服务器，监听在特定的端口等待连接
- 支持http协议，能够对http请求的报文进行处理，也能够把http响应的报文返回
- 遵循WSGI协议标准
  application：
- 本身是可调用的对象
- 返回包含正文的可迭代对象
- 遵循WSGI协议标准

python编写一个简单的web服务器

https://ruslanspivak.com/lsbaws-part1/

why not a golang WSGI implementation

https://stackoverflow.com/questions/20771890/why-not-a-golang-wsgi-implementation

GO的gin

https://heary.cn/posts/%E4%BB%8E%E6%BA%90%E7%A0%81%E7%90%86%E8%A7%A3Gin%E6%A1%86%E6%9E%B6%E5%8E%9F%E7%90%86/#%E8%AF%B7%E6%B1%82%E4%B8%8E%E5%93%8D%E5%BA%94%E8%BF%87%E7%A8%8B

网关是什么

在计算机网络中，网关（英语：Gateway）是转发其他服务器通信数据的服务器，接收从客户端发送来的请求时，它就像自己拥有资源的源服务器一样对请求进行处理。有时客户端可能都不会察觉，自己的通信目标是一个网关。

区别于路由器（由于历史的原因，许多有关TCP/IP的文献曾经把网络层使用的路由器（英语：Router）称为网关，在今天很多局域网采用都是路由来接入网络，因此现在通常指的网关就是路由器的IP），经常在家庭中或者小型企业网络中使用，用于连接局域网和互联网。

API网关也是随着对传统庞大的单体应用（All in one）拆分为众多的微服务（Microservice）以后，所引入的统一通信管理系统。用于运行在外部http请求与内部rpc服务之间的一个流量入口，实现对外部请求的协议转换、参数校验、鉴权、切量、熔断、限流、监控、风控等各类共性的通用服务。