**HTTP协议简介**

**1. 使用谷歌/火狐浏览器分析**

在Web应用中，服务器把网页传给浏览器，实际上就是把网页的HTML代码发送给浏览器，让浏览器显示出来。而浏览器和服务器之间的传输协议是HTTP，所以：

* HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
* HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

Chrome浏览器提供了一套完整地调试工具，非常适合Web开发。

安装好Chrome浏览器后，打开Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：

## 2. http协议的分析

当我们在地址栏输入www.sina.com时，浏览器将显示新浪的首页。在这个过程中，浏览器都干了哪些事情呢？通过Network的记录，我们就可以知道。在Network中，找到www.sina.com那条记录，点击，右侧将显示Request Headers，点击右侧的view source，我们就可以看到浏览器发给新浪服务器的请求：

### 2.1 浏览器请求

#### 说明

最主要的头两行分析如下，第一行：

GET / HTTP/1.1

GET表示一个读取请求，将从服务器获得网页数据，/表示URL的路径，URL总是以/开头，/就表示首页，最后的HTTP/1.1指示采用的HTTP协议版本是1.1。目前HTTP协议的版本就是1.1，但是大部分服务器也支持1.0版本，主要区别在于1.1版本允许多个HTTP请求复用一个TCP连接，以加快传输速度。

从第二行开始，每一行都类似于Xxx: abcdefg：

Host: www.sina.com

表示请求的域名是www.sina.com。如果一台服务器有多个网站，服务器就需要通过Host来区分浏览器请求的是哪个网站。

### 2.2 服务器响应

继续往下找到Response Headers，点击view source，显示服务器返回的原始响应数据：

HTTP响应分为Header和Body两部分（Body是可选项），我们在Network中看到的Header最重要的几行如下：

HTTP/1.1 200 OK

200表示一个成功的响应，后面的OK是说明。

如果返回的不是200，那么往往有其他的功能，例如

* 失败的响应有404 Not Found：网页不存在
* 500 Internal Server Error：服务器内部出错
* ...等等...

Content-Type: text/html

Content-Type指示响应的内容，这里是text/html表示HTML网页。

请注意，浏览器就是依靠Content-Type来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠URL来判断响应的内容，所以，即使URL是http://www.baidu.com/meimei.jpg，它也不一定就是图片。

HTTP响应的Body就是HTML源码，我们在菜单栏选择“视图”，“开发者”，“查看网页源码”就可以在浏览器中直接查看HTML源码：

#### 浏览器解析过程

当浏览器读取到新浪首页的HTML源码后，它会解析HTML，显示页面，然后，根据HTML里面的各种链接，再发送HTTP请求给新浪服务器，拿到相应的图片、视频、Flash、JavaScript脚本、CSS等各种资源，最终显示出一个完整的页面。所以我们在Network下面能看到很多额外的HTTP请求。

## 3. 总结

### 3.1 HTTP请求

跟踪了新浪的首页，我们来总结一下HTTP请求的流程：

#### 3.1.1 步骤1：浏览器首先向服务器发送HTTP请求，请求包括：

方法：GET还是POST，GET仅请求资源，POST会附带用户数据；

路径：/full/url/path；

域名：由Host头指定：Host: www.sina.com

以及其他相关的Header；

如果是POST，那么请求还包括一个Body，包含用户数据

#### 3.1.1 步骤2：服务器向浏览器返回HTTP响应，响应包括：

响应代码：200表示成功，3xx表示重定向，4xx表示客户端发送的请求有错误，5xx表示服务器端处理时发生了错误；

响应类型：由Content-Type指定；

以及其他相关的Header；

通常服务器的HTTP响应会携带内容，也就是有一个Body，包含响应的内容，网页的HTML源码就在Body中。

#### 3.1.1 步骤3：如果浏览器还需要继续向服务器请求其他资源，比如图片，就再次发出HTTP请求，重复步骤1、2。

Web采用的HTTP协议采用了非常简单的请求-响应模式，从而大大简化了开发。当我们编写一个页面时，我们只需要在HTTP请求中把HTML发送出去，不需要考虑如何附带图片、视频等，浏览器如果需要请求图片和视频，它会发送另一个HTTP请求，因此，一个HTTP请求只处理一个资源(此时就可以理解为TCP协议中的短连接，每个链接只获取一个资源，如需要多个就需要建立多个链接)

HTTP协议同时具备极强的扩展性，虽然浏览器请求的是http://www.sina.com的首页，但是新浪在HTML中可以链入其他服务器的资源，比如<img src="http://i1.sinaimg.cn/home/2013/1008/U8455P30DT20131008135420.png">，从而将请求压力分散到各个服务器上，并且，一个站点可以链接到其他站点，无数个站点互相链接起来，就形成了World Wide Web，简称WWW。

### 3.2 HTTP格式

每个HTTP请求和响应都遵循相同的格式，一个HTTP包含Header和Body两部分，其中Body是可选的。

HTTP协议是一种文本协议，所以，它的格式也非常简单。

#### 3.2.1 HTTP GET请求的格式：

GET /path HTTP/1.1

Header1: Value1

Header2: Value2

Header3: Value3

每个Header一行一个，换行符是\r\n。

#### 3.2.2 HTTP POST请求的格式：

POST /path HTTP/1.1

Header1: Value1

Header2: Value2

Header3: Value3

body data goes here...

当遇到连续两个\r\n时，Header部分结束，后面的数据全部是Body。

#### 3.2.3 HTTP响应的格式：

200 OK

Header1: Value1

Header2: Value2

Header3: Value3

body data goes here...

HTTP响应如果包含body，也是通过\r\n\r\n来分隔的。

请再次注意，Body的数据类型由Content-Type头来确定，如果是网页，Body就是文本，如果是图片，Body就是图片的二进制数据。

当存在Content-Encoding时，Body数据是被压缩的，最常见的压缩方式是gzip，所以，看到Content-Encoding: gzip时，需要将Body数据先解压缩，才能得到真正的数据。压缩的目的在于减少Body的大小，加快网络传输。

## Web静态服务器-1-显示固定的页面

#coding=utf-8

import socket

def handle\_client(client\_socket):

"为一个客户端进行服务"

recv\_data = client\_socket.recv(1024).decode("utf-8")

request\_header\_lines = recv\_data.splitlines()

for line in request\_header\_lines:

print(line)

# 组织相应 头信息(header)

response\_headers = "HTTP/1.1 200 OK\r\n" # 200表示找到这个资源

response\_headers += "\r\n" # 用一个空的行与body进行隔开

# 组织 内容(body)

response\_body = "hello world"

response = response\_headers + response\_body

client\_socket.send(response.encode("utf-8"))

client\_socket.close()

def main():

"作为程序的主控制入口"

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 设置当服务器先close 即服务器端4次挥手之后资源能够立即释放，这样就保证了，下次运行程序时 可以立即绑定7788端口

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server\_socket.bind(("", 7788))

server\_socket.listen(128)

while True:

client\_socket, client\_addr = server\_socket.accept()

handle\_client(client\_socket)

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Web静态服务器-2-显示需要的页面

#coding=utf-8

import socket

import re

def handle\_client(client\_socket):

"为一个客户端进行服务"

recv\_data = client\_socket.recv(1024).decode('utf-8', errors="ignore")

request\_header\_lines = recv\_data.splitlines()

for line in request\_header\_lines:

print(line)

http\_request\_line = request\_header\_lines[0]

get\_file\_name = re.match("[^/]+(/[^ ]\*)", http\_request\_line).group(1)

print("file name is ===>%s" % get\_file\_name) # for test

# 如果没有指定访问哪个页面。例如index.html

# GET / HTTP/1.1

if get\_file\_name == "/":

get\_file\_name = DOCUMENTS\_ROOT + "/index.html"

else:

get\_file\_name = DOCUMENTS\_ROOT + get\_file\_name

print("file name is ===2>%s" % get\_file\_name) #for test

try:

f = open(get\_file\_name, "rb")

except IOError:

# 404表示没有这个页面

response\_headers = "HTTP/1.1 404 not found\r\n"

response\_headers += "\r\n"

response\_body = "====sorry ,file not found===="

else:

response\_headers = "HTTP/1.1 200 OK\r\n"

response\_headers += "\r\n"

response\_body = f.read()

f.close()

finally:

# 因为头信息在组织的时候，是按照字符串组织的，不能与以二进制打开文件读取的数据合并，因此分开发送

# 先发送response的头信息

client\_socket.send(response\_headers.encode('utf-8'))

# 再发送body

client\_socket.send(response\_body)

client\_socket.close()

def main():

"作为程序的主控制入口"

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

server\_socket.bind(("", 7788))

server\_socket.listen(128)

while True:

client\_socket, clien\_cAddr = server\_socket.accept()

handle\_client(client\_socket)

#这里配置服务器

DOCUMENTS\_ROOT = "./html"

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Web静态服务器-3-多进程

#coding=utf-8

import socket

import re

import multiprocessing

class WSGIServer(object):

def \_\_init\_\_(self, server\_address):

# 创建一个tcp套接字

self.listen\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 允许立即使用上次绑定的port

self.listen\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

# 绑定

self.listen\_socket.bind(server\_address)

# 变为被动，并制定队列的长度

self.listen\_socket.listen(128)

def serve\_forever(self):

"循环运行web服务器，等待客户端的链接并为客户端服务"

while True:

# 等待新客户端到来

client\_socket, client\_address = self.listen\_socket.accept()

print(client\_address) # for test

new\_process = multiprocessing.Process(target=self.handleRequest, args=(client\_socket,))

new\_process.start()

# 因为子进程已经复制了父进程的套接字等资源，所以父进程调用close不会将他们对应的这个链接关闭的

client\_socket.close()

def handleRequest(self, client\_socket):

"用一个新的进程，为一个客户端进行服务"

recv\_data = client\_socket.recv(1024).decode('utf-8')

print(recv\_data)

requestHeaderLines = recv\_data.splitlines()

for line in requestHeaderLines:

print(line)

request\_line = requestHeaderLines[0]

get\_file\_name = re.match("[^/]+(/[^ ]\*)", request\_line).group(1)

print("file name is ===>%s" % get\_file\_name) # for test

if get\_file\_name == "/":

get\_file\_name = DOCUMENTS\_ROOT + "/index.html"

else:

get\_file\_name = DOCUMENTS\_ROOT + get\_file\_name

print("file name is ===2>%s" % get\_file\_name) # for test

try:

f = open(get\_file\_name, "rb")

except IOError:

response\_header = "HTTP/1.1 404 not found\r\n"

response\_header += "\r\n"

response\_body = "====sorry ,file not found===="

else:

response\_header = "HTTP/1.1 200 OK\r\n"

response\_header += "\r\n"

response\_body = f.read()

f.close()

finally:

client\_socket.send(response\_header.encode('utf-8'))

client\_socket.send(response\_body)

client\_socket.close()

# 设定服务器的端口

SERVER\_ADDR = (HOST, PORT) = "", 8888

# 设置服务器服务静态资源时的路径

DOCUMENTS\_ROOT = "./html"

def main():

httpd = WSGIServer(SERVER\_ADDR)

print("web Server: Serving HTTP on port %d ...\n" % PORT)

httpd.serve\_forever()

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Web静态服务器-4-多线程

#coding=utf-8

import socket

import re

import threading

class WSGIServer(object):

def \_\_init\_\_(self, server\_address):

# 创建一个tcp套接字

self.listen\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 允许立即使用上次绑定的port

self.listen\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

# 绑定

self.listen\_socket.bind(server\_address)

# 变为被动，并制定队列的长度

self.listen\_socket.listen(128)

def serve\_forever(self):

"循环运行web服务器，等待客户端的链接并为客户端服务"

while True:

# 等待新客户端到来

client\_socket, client\_address = self.listen\_socket.accept()

print(client\_address)

new\_process = threading.Thread(target=self.handleRequest, args=(client\_socket,))

new\_process.start()

# 因为线程是共享同一个套接字，所以主线程不能关闭，否则子线程就不能再使用这个套接字了

# client\_socket.close()

def handleRequest(self, client\_socket):

"用一个新的进程，为一个客户端进行服务"

recv\_data = client\_socket.recv(1024).decode('utf-8')

print(recv\_data)

requestHeaderLines = recv\_data.splitlines()

for line in requestHeaderLines:

print(line)

request\_line = requestHeaderLines[0]

get\_file\_name = re.match("[^/]+(/[^ ]\*)", request\_line).group(1)

print("file name is ===>%s" % get\_file\_name) # for test

if get\_file\_name == "/":

get\_file\_name = DOCUMENTS\_ROOT + "/index.html"

else:

get\_file\_name = DOCUMENTS\_ROOT + get\_file\_name

print("file name is ===2>%s" % get\_file\_name) # for test

try:

f = open(get\_file\_name, "rb")

except IOError:

response\_header = "HTTP/1.1 404 not found\r\n"

response\_header += "\r\n"

response\_body = "====sorry ,file not found===="

else:

response\_header = "HTTP/1.1 200 OK\r\n"

response\_header += "\r\n"

response\_body = f.read()

f.close()

finally:

client\_socket.send(response\_header.encode('utf-8'))

client\_socket.send(response\_body)

client\_socket.close()

# 设定服务器的端口

SERVER\_ADDR = (HOST, PORT) = "", 8888

# 设置服务器服务静态资源时的路径

DOCUMENTS\_ROOT = "./html"

def main():

httpd = WSGIServer(SERVER\_ADDR)

print("web Server: Serving HTTP on port %d ...\n" % PORT)

httpd.serve\_forever()

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Web静态服务器-5-非堵塞模式

### 单进程非堵塞 模型

#coding=utf-8

from socket import \*

import time

# 用来存储所有的新链接的socket

g\_socket\_list = list()

def main():

server\_socket = socket(AF\_INET, SOCK\_STREAM)

server\_socket.setsockopt(SOL\_SOCKET, SO\_REUSEADDR , 1)

server\_socket.bind(('', 7890))

server\_socket.listen(128)

# 将套接字设置为非堵塞

# 设置为非堵塞后，如果accept时，恰巧没有客户端connect，那么accept会

# 产生一个异常，所以需要try来进行处理

server\_socket.setblocking(False)

while True:

# 用来测试

time.sleep(0.5)

try:

newClientInfo = server\_socket.accept()

except Exception as result:

pass

else:

print("一个新的客户端到来:%s" % str(newClientInfo))

newClientInfo[0].setblocking(False) # 设置为非堵塞

g\_socket\_list.append(newClientInfo)

for client\_socket, client\_addr in g\_socket\_list:

try:

recvData = client\_socket.recv(1024)

if recvData:

print('recv[%s]:%s' % (str(client\_addr), recvData))

else:

print('[%s]客户端已经关闭' % str(client\_addr))

client\_socket.close()

g\_socket\_list.remove((client\_socket,client\_addr))

except Exception as result:

pass

print(g\_socket\_list) # for test

if \_\_name\_\_ == '\_\_main\_\_':

main()

### web静态服务器-单进程非堵塞

import time

import socket

import sys

import re

class WSGIServer(object):

"""定义一个WSGI服务器的类"""

def \_\_init\_\_(self, port, documents\_root):

# 1. 创建套接字

self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 2. 绑定本地信息

self.server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.server\_socket.bind(("", port))

# 3. 变为监听套接字

self.server\_socket.listen(128)

self.server\_socket.setblocking(False)

self.client\_socket\_list = list()

self.documents\_root = documents\_root

def run\_forever(self):

"""运行服务器"""

# 等待对方链接

while True:

# time.sleep(0.5) # for test

try:

new\_socket, new\_addr = self.server\_socket.accept()

except Exception as ret:

print("-----1----", ret) # for test

else:

new\_socket.setblocking(False)

self.client\_socket\_list.append(new\_socket)

for client\_socket in self.client\_socket\_list:

try:

request = client\_socket.recv(1024).decode('utf-8')

except Exception as ret:

print("------2----", ret) # for test

else:

if request:

self.deal\_with\_request(request, client\_socket)

else:

client\_socket.close()

self.client\_socket\_list.remove(client\_socket)

print(self.client\_socket\_list)

def deal\_with\_request(self, request, client\_socket):

"""为这个浏览器服务器"""

if not request:

return

request\_lines = request.splitlines()

for i, line in enumerate(request\_lines):

print(i, line)

# 提取请求的文件(index.html)

# GET /a/b/c/d/e/index.html HTTP/1.1

ret = re.match(r"([^/]\*)([^ ]+)", request\_lines[0])

if ret:

print("正则提取数据:", ret.group(1))

print("正则提取数据:", ret.group(2))

file\_name = ret.group(2)

if file\_name == "/":

file\_name = "/index.html"

# 读取文件数据

try:

f = open(self.documents\_root+file\_name, "rb")

except:

response\_body = "file not found, 请输入正确的url"

response\_header = "HTTP/1.1 404 not found\r\n"

response\_header += "Content-Type: text/html; charset=utf-8\r\n"

response\_header += "Content-Length: %d\r\n" % (len(response\_body))

response\_header += "\r\n"

# 将header返回给浏览器

client\_socket.send(response\_header.encode('utf-8'))

# 将body返回给浏览器

client\_socket.send(response\_body.encode("utf-8"))

else:

content = f.read()

f.close()

response\_body = content

response\_header = "HTTP/1.1 200 OK\r\n"

response\_header += "Content-Length: %d\r\n" % (len(response\_body))

response\_header += "\r\n"

# 将header返回给浏览器

client\_socket.send( response\_header.encode('utf-8') + response\_body)

# 设置服务器服务静态资源时的路径

DOCUMENTS\_ROOT = "./html"

def main():

"""控制web服务器整体"""

# python3 xxxx.py 7890

if len(sys.argv) == 2:

port = sys.argv[1]

if port.isdigit():

port = int(port)

else:

print("运行方式如: python3 xxx.py 7890")

return

print("http服务器使用的port:%s" % port)

http\_server = WSGIServer(port, DOCUMENTS\_ROOT)

http\_server.run\_forever()

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Web静态服务器-6-epoll

### IO 多路复用

就是我们说的select，poll，epoll，有些地方也称这种IO方式为event driven IO。

select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。

它的基本原理就是select，poll，epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。

### epoll简单模型

import socket

import select

# 创建套接字

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 设置可以重复使用绑定的信息

s.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR,1)

# 绑定本机信息

s.bind(("",7788))

# 变为被动

s.listen(10)

# 创建一个epoll对象

epoll = select.epoll()

# 测试，用来打印套接字对应的文件描述符

# print(s.fileno())

# print(select.EPOLLIN|select.EPOLLET)

# 注册事件到epoll中

# epoll.register(fd[, eventmask])

# 注意，如果fd已经注册过，则会发生异常

# 将创建的套接字添加到epoll的事件监听中

epoll.register(s.fileno(), select.EPOLLIN|select.EPOLLET)

connections = {}

addresses = {}

# 循环等待客户端的到来或者对方发送数据

while True:

# epoll 进行 fd 扫描的地方 -- 未指定超时时间则为阻塞等待

epoll\_list = epoll.poll()

# 对事件进行判断

for fd, events in epoll\_list:

# print fd

# print events

# 如果是socket创建的套接字被激活

if fd == s.fileno():

new\_socket, new\_addr = s.accept()

print('有新的客户端到来%s' % str(new\_addr))

# 将 conn 和 addr 信息分别保存起来

connections[new\_socket.fileno()] = new\_socket

addresses[new\_socket.fileno()] = new\_addr

# 向 epoll 中注册 新socket 的 可读 事件

epoll.register(new\_socket.fileno(), select.EPOLLIN|select.EPOLLET)

# 如果是客户端发送数据

elif events == select.EPOLLIN:

# 从激活 fd 上接收

recvData = connections[fd].recv(1024).decode("utf-8")

if recvData:

print('recv:%s' % recvData)

else:

# 从 epoll 中移除该 连接 fd

epoll.unregister(fd)

# server 侧主动关闭该 连接 fd

connections[fd].close()

print("%s---offline---" % str(addresses[fd]))

del connections[fd]

del addresses[fd]

#### 说明

* EPOLLIN （可读）
* EPOLLOUT （可写）
* EPOLLET （ET模式）

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式，LT模式与ET模式的区别如下：

LT模式：当epoll检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll时，会再次响应应用程序并通知此事件。

ET模式：当epoll检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll时，不会再次响应应用程序并通知此事件。

### web静态服务器-epool

以下代码，支持http的长连接，即使用了Content-Length

import socket

import time

import sys

import re

import select

class WSGIServer(object):

"""定义一个WSGI服务器的类"""

def \_\_init\_\_(self, port, documents\_root):

# 1. 创建套接字

self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 2. 绑定本地信息

self.server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.server\_socket.bind(("", port))

# 3. 变为监听套接字

self.server\_socket.listen(128)

self.documents\_root = documents\_root

# 创建epoll对象

self.epoll = select.epoll()

# 将tcp服务器套接字加入到epoll中进行监听

self.epoll.register(self.server\_socket.fileno(), select.EPOLLIN|select.EPOLLET)

# 创建添加的fd对应的套接字

self.fd\_socket = dict()

def run\_forever(self):

"""运行服务器"""

# 等待对方链接

while True:

# epoll 进行 fd 扫描的地方 -- 未指定超时时间则为阻塞等待

epoll\_list = self.epoll.poll()

# 对事件进行判断

for fd, event in epoll\_list:

# 如果是服务器套接字可以收数据，那么意味着可以进行accept

if fd == self.server\_socket.fileno():

new\_socket, new\_addr = self.server\_socket.accept()

# 向 epoll 中注册 连接 socket 的 可读 事件

self.epoll.register(new\_socket.fileno(), select.EPOLLIN | select.EPOLLET)

# 记录这个信息

self.fd\_socket[new\_socket.fileno()] = new\_socket

# 接收到数据

elif event == select.EPOLLIN:

request = self.fd\_socket[fd].recv(1024).decode("utf-8")

if request:

self.deal\_with\_request(request, self.fd\_socket[fd])

else:

# 在epoll中注销客户端的信息

self.epoll.unregister(fd)

# 关闭客户端的文件句柄

self.fd\_socket[fd].close()

# 在字典中删除与已关闭客户端相关的信息

del self.fd\_socket[fd]

def deal\_with\_request(self, request, client\_socket):

"""为这个浏览器服务器"""

if not request:

return

request\_lines = request.splitlines()

for i, line in enumerate(request\_lines):

print(i, line)

# 提取请求的文件(index.html)

# GET /a/b/c/d/e/index.html HTTP/1.1

ret = re.match(r"([^/]\*)([^ ]+)", request\_lines[0])

if ret:

print("正则提取数据:", ret.group(1))

print("正则提取数据:", ret.group(2))

file\_name = ret.group(2)

if file\_name == "/":

file\_name = "/index.html"

# 读取文件数据

try:

f = open(self.documents\_root+file\_name, "rb")

except:

response\_body = "file not found, 请输入正确的url"

response\_header = "HTTP/1.1 404 not found\r\n"

response\_header += "Content-Type: text/html; charset=utf-8\r\n"

response\_header += "Content-Length: %d\r\n" % len(response\_body)

response\_header += "\r\n"

# 将header返回给浏览器

client\_socket.send(response\_header.encode('utf-8'))

# 将body返回给浏览器

client\_socket.send(response\_body.encode("utf-8"))

else:

content = f.read()

f.close()

response\_body = content

response\_header = "HTTP/1.1 200 OK\r\n"

response\_header += "Content-Length: %d\r\n" % len(response\_body)

response\_header += "\r\n"

# 将数据返回给浏览器

client\_socket.send(response\_header.encode("utf-8")+response\_body)

# 设置服务器服务静态资源时的路径

DOCUMENTS\_ROOT = "./html"

def main():

"""控制web服务器整体"""

# python3 xxxx.py 7890

if len(sys.argv) == 2:

port = sys.argv[1]

if port.isdigit():

port = int(port)

else:

print("运行方式如: python3 xxx.py 7890")

return

print("http服务器使用的port:%s" % port)

http\_server = WSGIServer(port, DOCUMENTS\_ROOT)

http\_server.run\_forever()

if \_\_name\_\_ == "\_\_main\_\_":

main()

### 小总结

I/O 多路复用的特点：

通过一种机制使一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，epoll()函数就可以返回。 所以, IO多路复用，本质上不会有并发的功能，因为任何时候还是只有一个进程或线程进行工作，它之所以能提高效率是因为select\epoll 把进来的socket放到他们的 '监视' 列表里面，当任何socket有可读可写数据立马处理，那如果select\epoll 手里同时检测着很多socket， 一有动静马上返回给进程处理，总比一个一个socket过来,阻塞等待,处理高效率。

当然也可以多线程/多进程方式，一个连接过来开一个进程/线程处理，这样消耗的内存和进程切换页会耗掉更多的系统资源。 所以我们可以结合IO多路复用和多进程/多线程 来高性能并发，IO复用负责提高接受socket的通知效率，收到请求后，交给进程池/线程池来处理逻辑。

## Web静态服务器-7-gevent版

from gevent import monkey

import gevent

import socket

import sys

import re

monkey.patch\_all()

class WSGIServer(object):

"""定义一个WSGI服务器的类"""

def \_\_init\_\_(self, port, documents\_root):

# 1. 创建套接字

self.server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

# 2. 绑定本地信息

self.server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

self.server\_socket.bind(("", port))

# 3. 变为监听套接字

self.server\_socket.listen(128)

self.documents\_root = documents\_root

def run\_forever(self):

"""运行服务器"""

# 等待对方链接

while True:

new\_socket, new\_addr = self.server\_socket.accept()

gevent.spawn(self.deal\_with\_request, new\_socket) # 创建一个协程准备运行它

def deal\_with\_request(self, client\_socket):

"""为这个浏览器服务器"""

while True:

# 接收数据

request = client\_socket.recv(1024).decode('utf-8')

# print(gevent.getcurrent())

# print(request)

# 当浏览器接收完数据后，会自动调用close进行关闭，因此当其关闭时，web也要关闭这个套接字

if not request:

new\_socket.close()

break

request\_lines = request.splitlines()

for i, line in enumerate(request\_lines):

print(i, line)

# 提取请求的文件(index.html)

# GET /a/b/c/d/e/index.html HTTP/1.1

ret = re.match(r"([^/]\*)([^ ]+)", request\_lines[0])

if ret:

print("正则提取数据:", ret.group(1))

print("正则提取数据:", ret.group(2))

file\_name = ret.group(2)

if file\_name == "/":

file\_name = "/index.html"

file\_path\_name = self.documents\_root + file\_name

try:

f = open(file\_path\_name, "rb")

except:

# 如果不能打开这个文件，那么意味着没有这个资源，没有资源 那么也得需要告诉浏览器 一些数据才行

# 404

response\_body = "没有你需要的文件......".encode("utf-8")

response\_headers = "HTTP/1.1 404 not found\r\n"

response\_headers += "Content-Type:text/html;charset=utf-8\r\n"

response\_headers += "Content-Length:%d\r\n" % len(response\_body)

response\_headers += "\r\n"

send\_data = response\_headers.encode("utf-8") + response\_body

client\_socket.send(send\_data)

else:

content = f.read()

f.close()

# 响应的body信息

response\_body = content

# 响应头信息

response\_headers = "HTTP/1.1 200 OK\r\n"

response\_headers += "Content-Type:text/html;charset=utf-8\r\n"

response\_headers += "Content-Length:%d\r\n" % len(response\_body)

response\_headers += "\r\n"

send\_data = response\_headers.encode("utf-8") + response\_body

client\_socket.send(send\_data)

# 设置服务器服务静态资源时的路径

DOCUMENTS\_ROOT = "./html"

def main():

"""控制web服务器整体"""

# python3 xxxx.py 7890

if len(sys.argv) == 2:

port = sys.argv[1]

if port.isdigit():

port = int(port)

else:

print("运行方式如: python3 xxx.py 7890")

return

print("http服务器使用的port:%s" % port)

http\_server = WSGIServer(port, DOCUMENTS\_ROOT")

http\_server.run\_forever()

if \_\_name\_\_ == "\_\_main\_\_":

main()