

## python 高级

### HTTP 协议简介

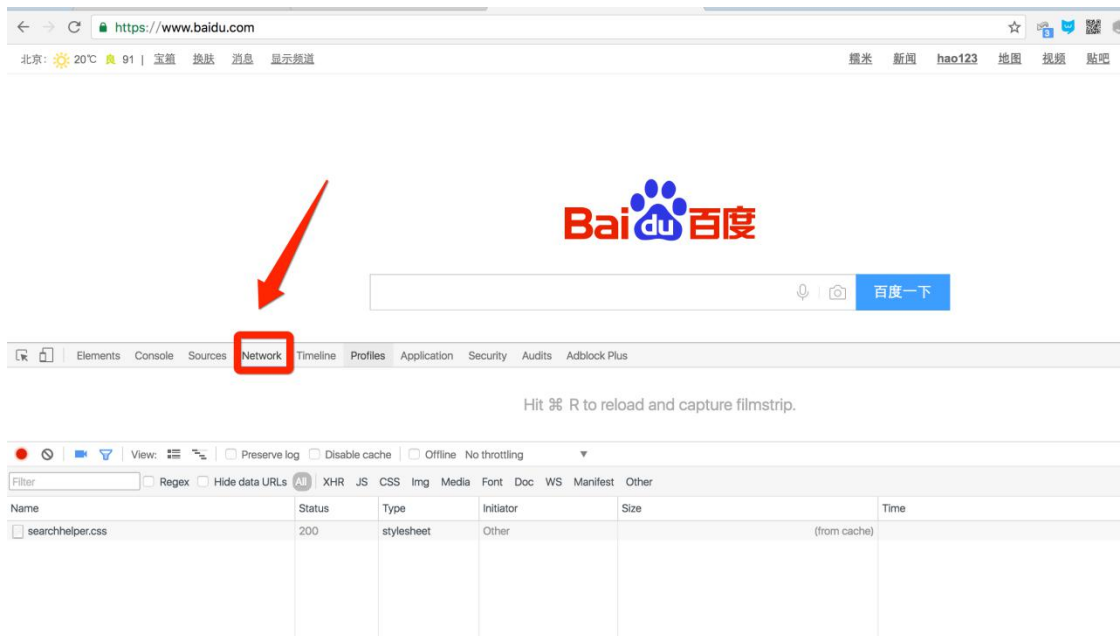
#### 1. 使用谷歌/火狐浏览器分析

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。

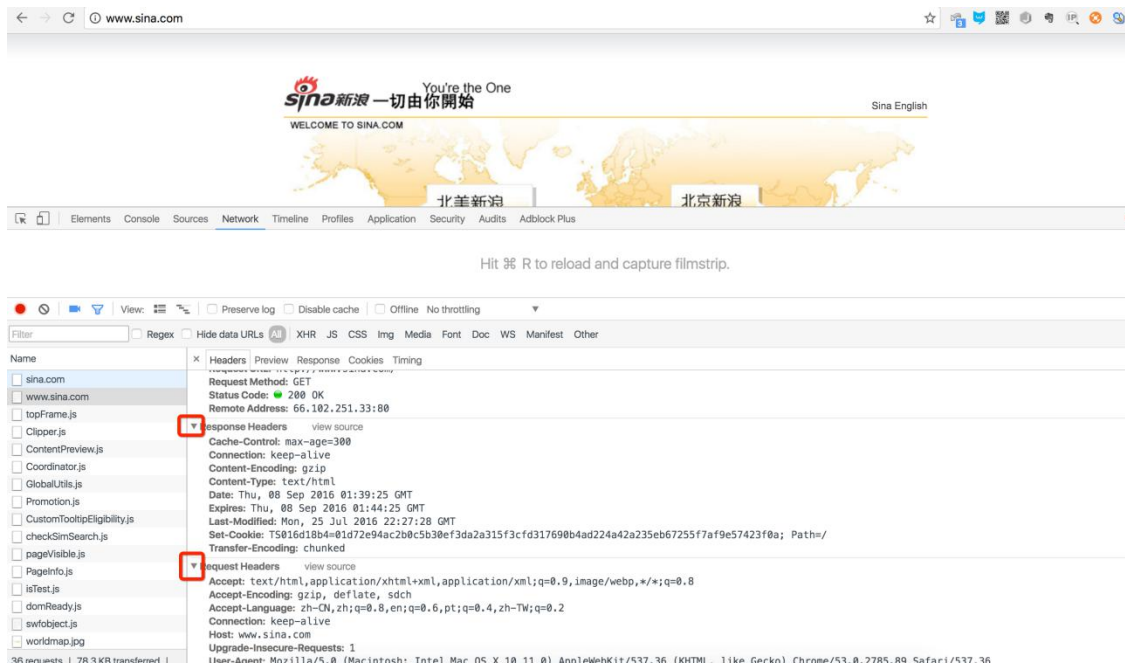
安装好 Chrome 浏览器后，打开 Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



#### 说明

- Elements 显示网页的结构
- Network 显示浏览器和服务器的通信

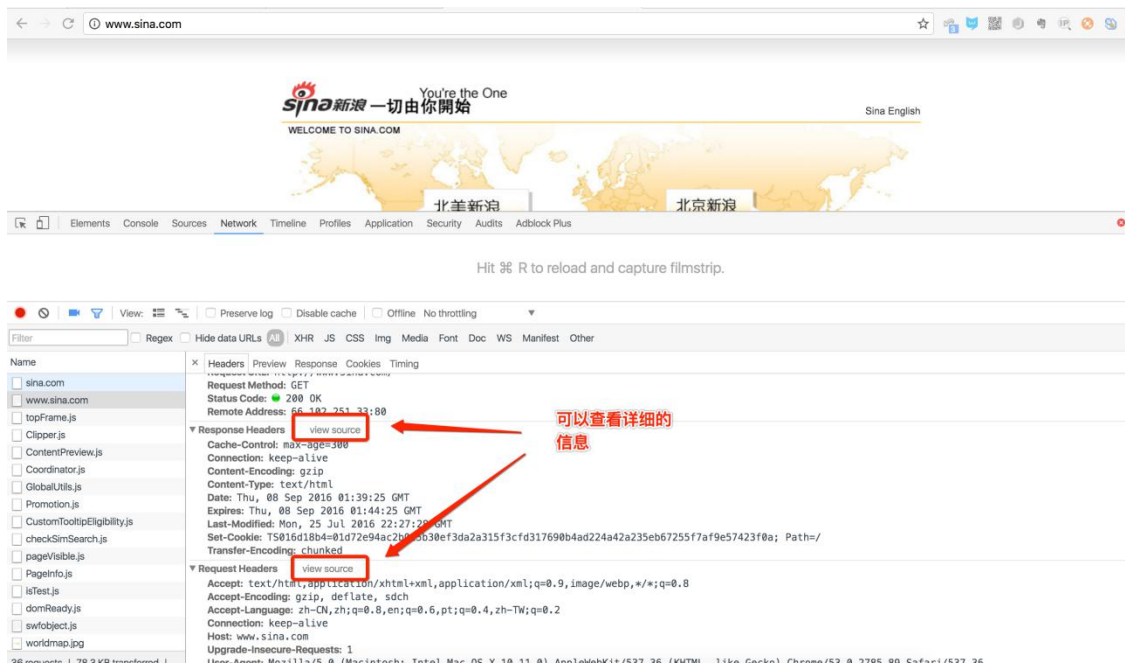
我们点 Network，确保第一个小红灯亮着，Chrome 就会记录所有浏览器和服务器的通信：

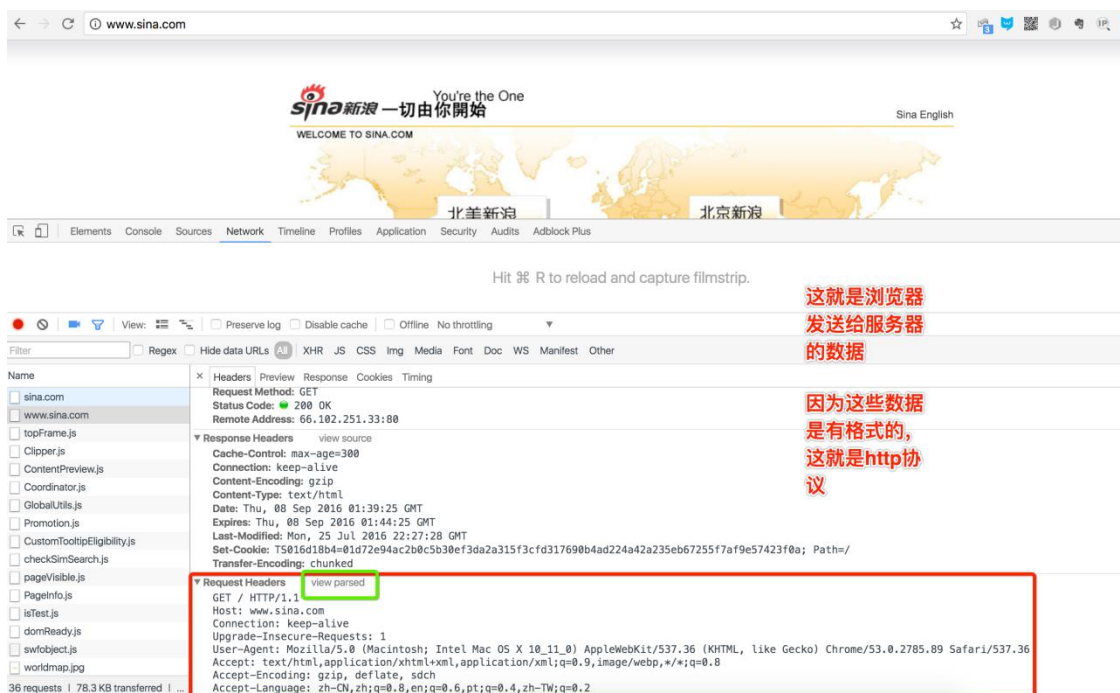


## 2. http 协议的分析

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 请求复用同一个 TCP 连接，以加快传输速度。

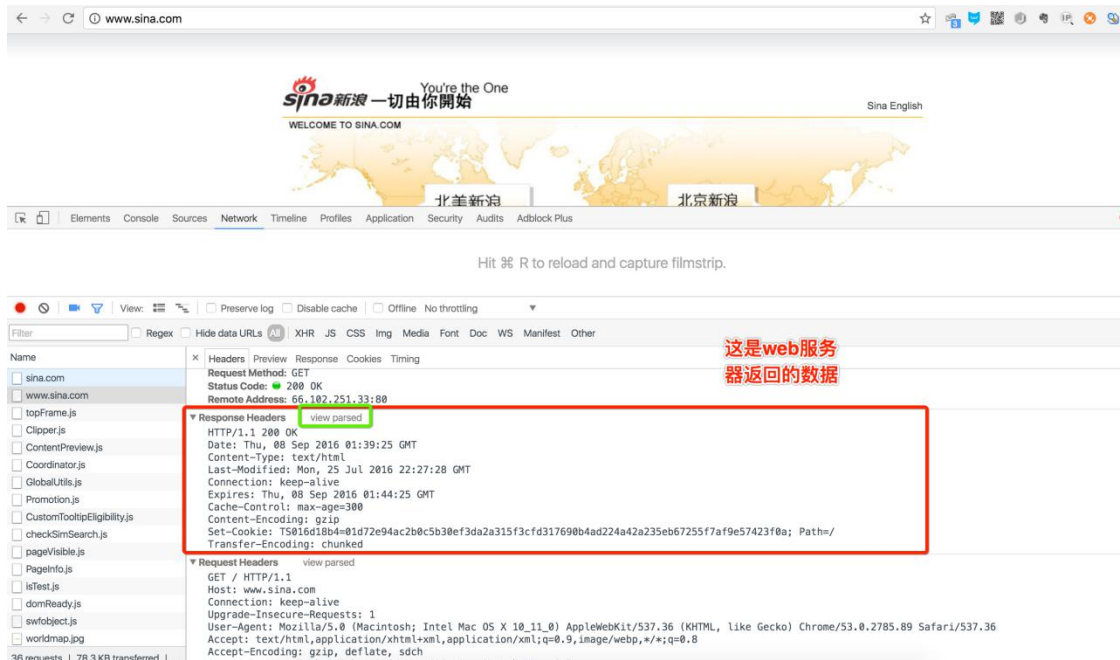
从第二行开始，每一行都类似于 Xxx: abcdefg:

**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

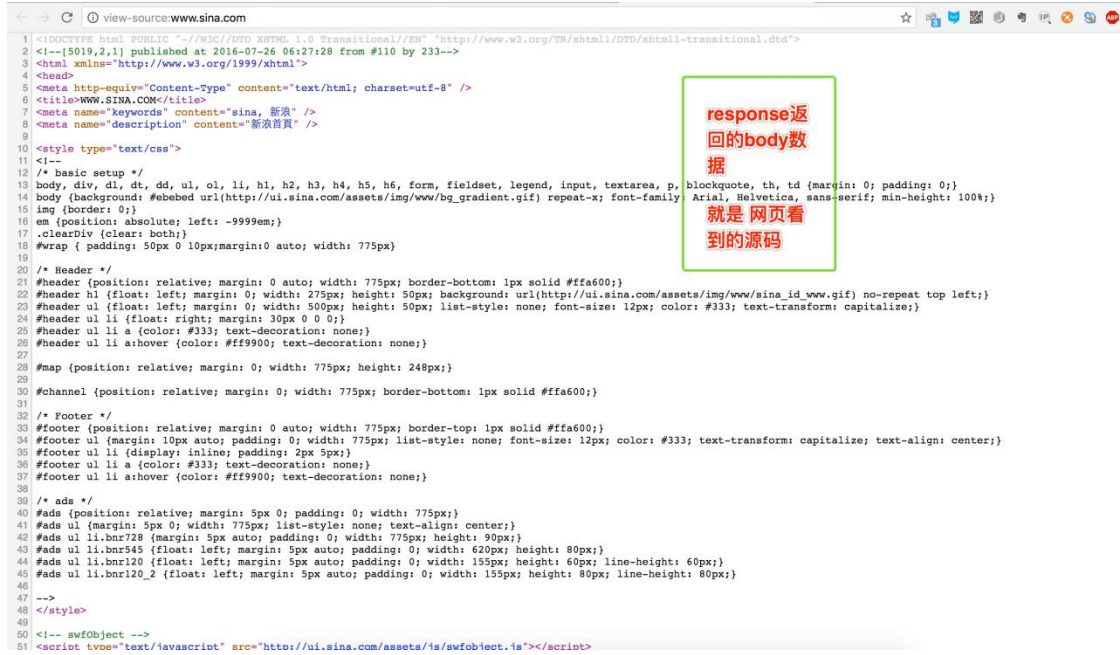
- 失败的响应有 404 Not Found：网页不存在
- 500 Internal Server Error：服务器内部出错
- ...等等...

Content-Type: text/html

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

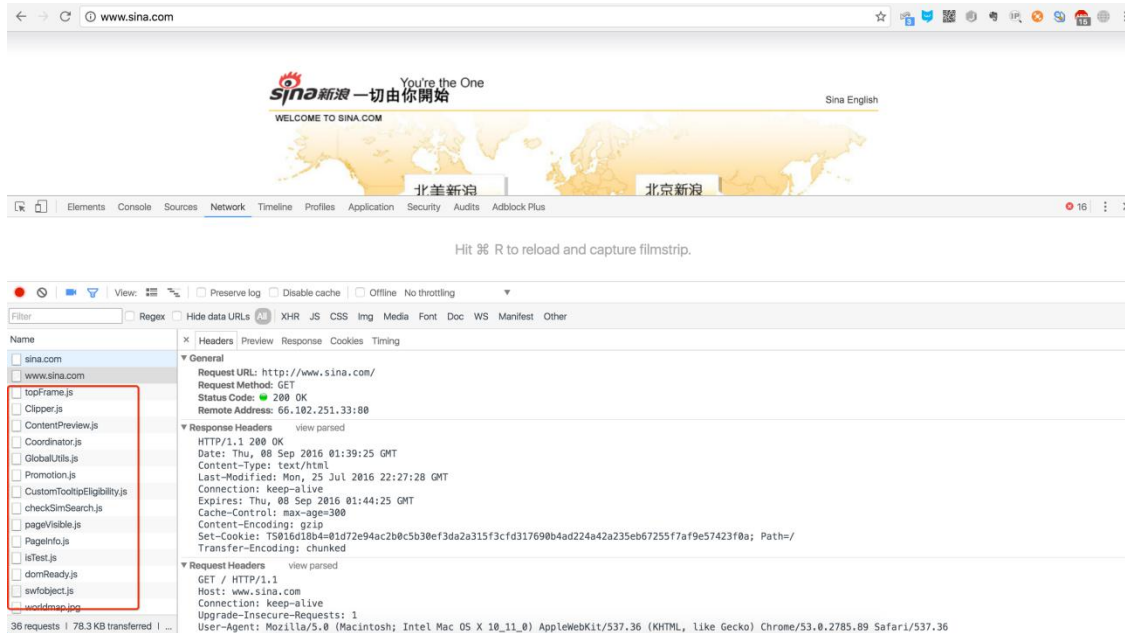
请注意，浏览器就是依靠 Content-Type 来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠 URL 来判断响应的内容，所以，即使 URL 是 <http://www.baidu.com/meimei.jpg>，它也不一定是图片。

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、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。

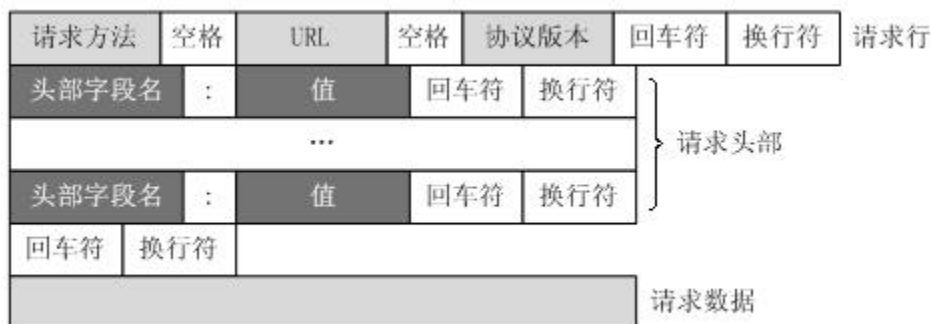


## http 协议的结束符

突然想起很久之前一次面试，面试官问我，当请求头没有 content-length 时，怎么知道请求体结束了？



http 的 header 和 body 之间空行分割的，又因为每个头部项是以 `\r\n` 作为结束符，所以，数据流中是以 `\r\n\r\n` 来分割解析请求头(响应头)与请求体（响应体）的。如下图所示：



那么怎么知道（请求体）响应体结束了呢？http 协议规定，响应头的字段 `content-length` 用来表示响应体长度大小，但是，有可能发送请求头时，并不能知道完整的响应体长度（比如当响应数据太大，服务端流式处理的情况），这时需要设置请求头 `Transfer-Encoding: chunked`，使用数据块的方式传输，数据块格式如下图所示：



每个数据块分为两个部分：数据长度和数据内容，以 `\r\n` 分割，最后长度为 0 的数据块，内容为空行（`\r\n`），表示没有数据再传输了，响应结束。需要注意的是，此时，`content-length` 不应该被设置，就算设置了，也会被忽略掉。

回到最开始的那个问题，我当时对 http 协议不太清楚，回答不上来，那位面试官就告诉我，可以使用 `\r\n\r\n` 来判断，现在看来，他说的并不严谨。首先，http 协议并没有规定请求体（响应体）要以 `\r\n\r\n` 作为结束符，其次，很重要的一点是，响应体（请求体）的内容是多种多样的，你没法做限制，当数据内容包含 `\r\n\r\n` 时，显然解析出来的响应体就是不全的。

当然，如果是自己实现 http 服务端的话，怎么兼容这种情况呢？

如果是短连接的话，比较简单，连接关闭就表示数据传输完成了。如果是长连接的话，一种不太优雅的方式就是使用超时机制，当读取超过一定时间，就认为数据已经传输完成。

总之，判断数据（块）结束最严谨的方式是计算长度，而不是使用结束符，但是，一般可控的场景下（双方约定），还是可以选择使用结束符来判断的，这样实现起来会更简洁。此时，为了防止内容中包含约定的结束符，导致数据内容被提前截断，客户端可以在发送数据时先对内容中的约定结束符进行编码。

加入

<https://blog.csdn.net/u012375924/article/details/82806617>

Cache-Control 字段

## 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；

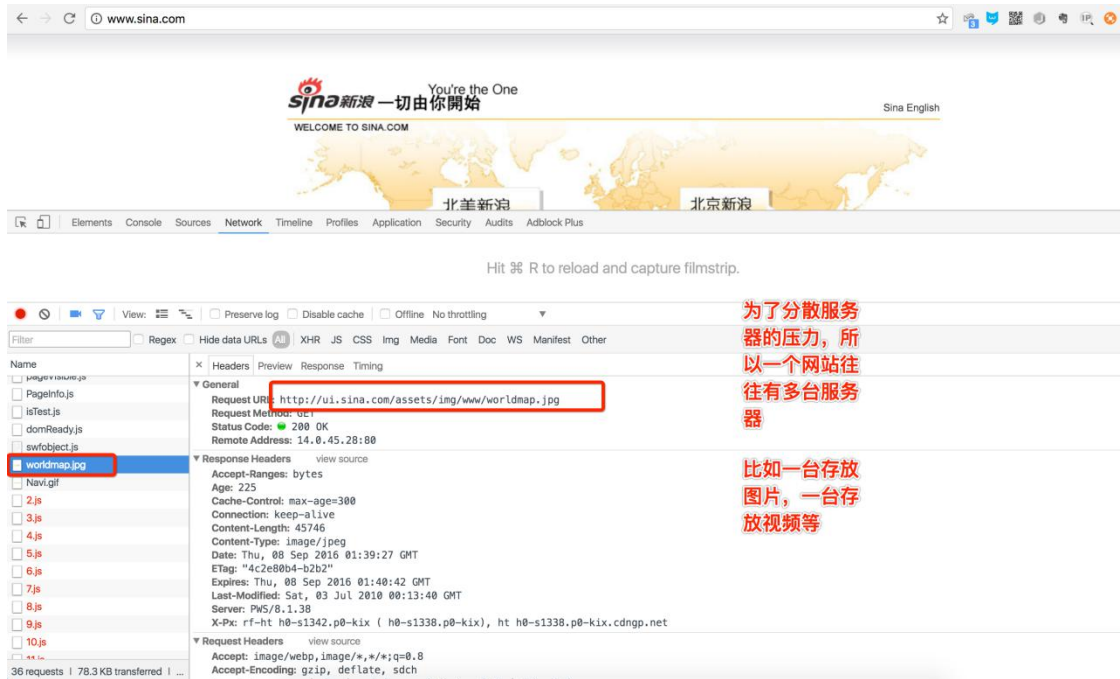
以及其他相关的 Header；

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 ``，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。

总结

<https://www.cnblogs.com/lmh001/p/10443339.html>

HTTP 请求的类型

GET 查询

POST 新增

PUT 修改

DELETE 删除

响应状态码

<https://zhuanlan.zhihu.com/p/66062179>

## 4.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)

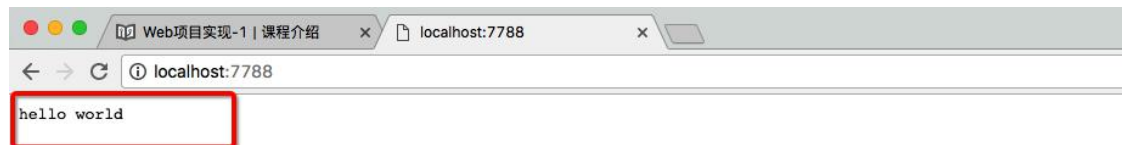
if __name__ == "__main__":
    main()
```

## 服务器端

```
dongge@hogan Desktop$ python 04-web服务器-静态页面.py
GET / HTTP/1.1
Host: localhost:7788
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6,pt;q=0.4,zh-TW;q=0.2

GET /favicon.ico HTTP/1.1
Host: localhost:7788
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36
Accept: */*
Referer: http://localhost:7788/
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: zh-CN,zh;q=0.8,en;q=0.6,pt;q=0.4,zh-TW;q=0.2
```

## 客户端



## 5.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====".encode('utf-8')

    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'))
```

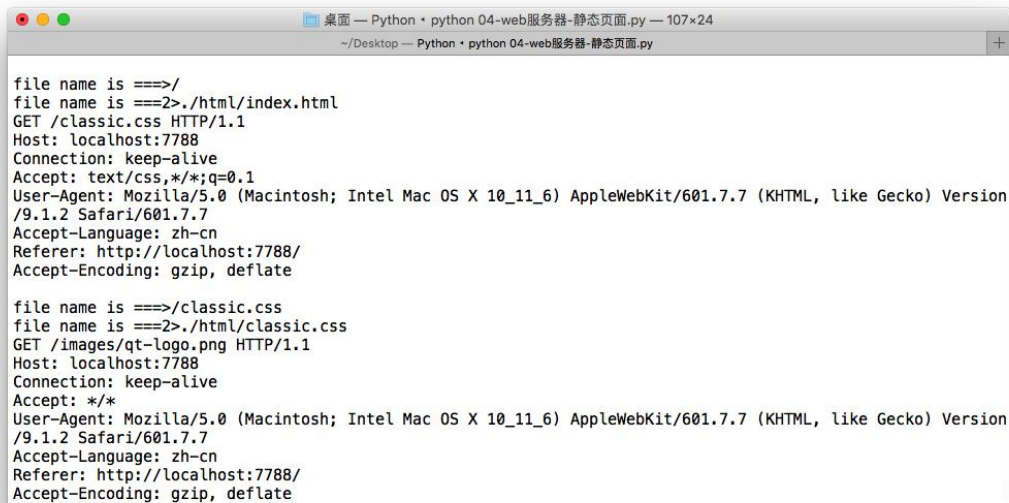
```
# 再发送 body
client_socket.send(response_body)
client_socket.close()

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()
```

## 服务器端



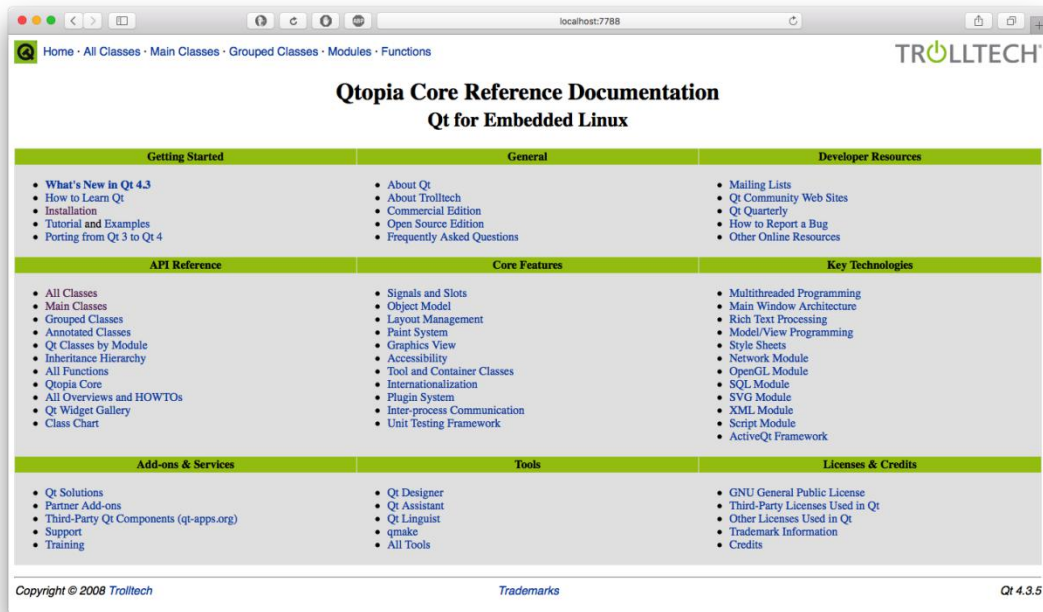
```
桌面 — Python • python 04-web服务器-静态页面.py — 107x24
~/Desktop — Python • python 04-web服务器-静态页面.py

file name is ==>/
file name is ==2>./html/index.html
GET /classic.css HTTP/1.1
Host: localhost:7788
Connection: keep-alive
Accept: text/css,*/*;q=0.1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/601.7.7 (KHTML, like Gecko) Version
/9.1.2 Safari/601.7.7
Accept-Language: zh-cn
Referer: http://localhost:7788/
Accept-Encoding: gzip, deflate

file name is ==>/classic.css
file name is ==2>./html/classic.css
GET /images/qt-logo.png HTTP/1.1
Host: localhost:7788
Connection: keep-alive
Accept: */*
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/601.7.7 (KHTML, like Gecko) Version
/9.1.2 Safari/601.7.7
Accept-Language: zh-cn
Referer: http://localhost:7788/
Accept-Encoding: gzip, deflate
```



## 客户端



## 6.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_REUSE
ADDR, 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，等待客户端的连接并为客户端服务"
        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.handleReq
uest, 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 ==>%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()
```

## 7.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_REUSE
ADDR, 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，等待客户端的连接并为客户端服务"
        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, a
rgs=(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()
```



## 8.Web 静态服务器-5-非堵塞模式

### 1.1 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()
```

## 1.2web 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_REUSE  
ADDR, 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"([^\s/]*)([^\s]+)", 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') + respo
nse_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()
```

## 9.Web 静态服务器-6-epoll（epoll 只有 Linux 平台支持）

### 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。

### 2.1epoll 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.E
POLLET)

    # 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。

## 2.2web 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_REUSE
EADDR, 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"([^\s/]*)([^\s]+)", 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。

## 参考资料

- 如果了解下 `epoll` 在 Linux 中的实现过程可以参考:  
<http://blog.csdn.net/xiajun07061225/article/details/9250579>



## 10.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_REUSE
EADDR, 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:
    client_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"([^\s]*) ([^\s]+)", 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(respo
nse_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()
```

## 11.知识扩展-C10K 问题

参考文章：

《单台服务器并发 TCP 连接数到底可以有多少》

<http://www.52im.net/thread-561-1-1.html>

《上一个 10 年，著名的 C10K 并发连接问题》 <http://www.52im.net/thread-566-1-1.html>