一、正则表达式

1.1 re 模块操作

在 Python 中需要通过正则表达式对字符串进行匹配的时候，可以使用一个模块，名字为 re

1. re 模块的使用过程

```
#coding=utf-8

# 导入 re 模块
import re

# 使用 match 方法进行匹配操作
result = re.match(正则表达式,要匹配的字符串)

# 如果上一步匹配到数据的话，可以使用 group 方法来提取数据
result.group()
```

2. re 模块示例(匹配以 itcast 开头的语句)

```
#coding=utf-8

import re

result = re.match("itcast","itcast.cn")

result.group()
```

运行结果为:

```
itcast
```

3. 说明

- re.match() 能够匹配出以 xxx 开头的字符串

1.2 匹配单个字符

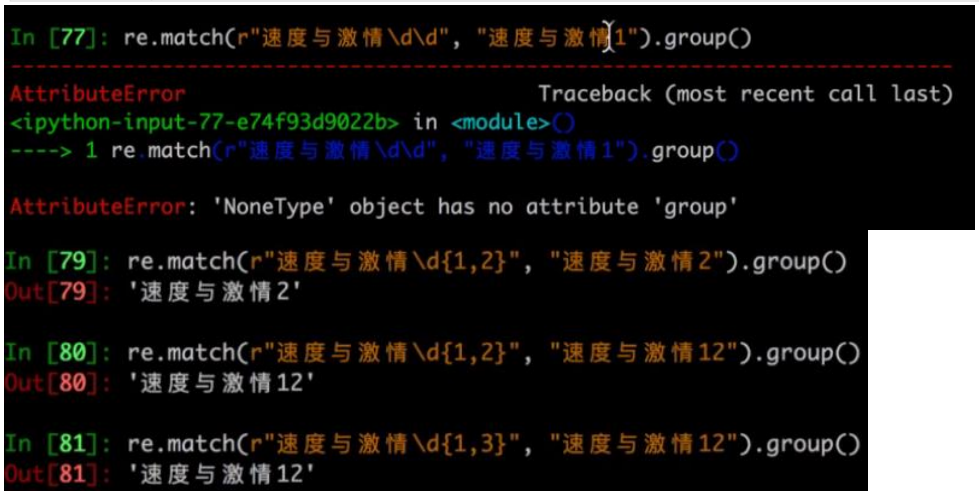
在上一小节中，了解到通过 re 模块能够完成使用正则表达式来匹配字符串本小节，将要讲解正则表达式的单字符匹配

字符	功能
.	匹配任意1个字符（除了\n）
[ ]	匹配[ ]中列举的字符
\d	匹配数字，即0-9
\D	匹配非数字，即不是数字
\s	匹配空白，即 空格，tab键
\S	匹配非空白
\w	匹配单词字符，即a-z、A-Z、0-9、_
\W	匹配非单词字符

1.3 匹配多个字符

匹配多个字符的相关格式

字符	功能
*	匹配前一个字符出现0次或者无限次，即可有可无
+	匹配前一个字符出现1次或者无限次，即至少有1次
?	匹配前一个字符出现1次或者0次，即要么有1次，要么没有
{m}	匹配前一个字符出现m次
{m,n}	匹配前一个字符出现从m到n次



```
In [95]: re.match(r"\d{3,4}-?\d{8}", "0532-12345678").group()
Out[95]: '0532-12345678'

In [96]: re.match(r"\d{3,4}-?\d{7,8}", "0532-12345678").group()
Out[96]: '0532-12345678'

In [97]: re.match(r"\d{3,4}-?\d{7,8}", "0532-1234567").group()
Out[97]: '0532-1234567'
```

1.4 匹配开头结尾

字符	功能
^	匹配字符串开头
$	匹配字符串结尾

示例1：$

需求：匹配163.com的邮箱地址

```
#coding=utf-8

import re

email_list = ["xiaoWang@163.com", "xiaoWang@163.comheihei", ".com.xiaowang@qq.com"]

for email in email_list:
    ret = re.match("[\w]{4,20}@163\.com", email)
    if ret:
        print("%s 是符合规定的邮件地址,匹配后的结果是:%s" % (email, ret.group()))
    else:
        print("%s 不符合要求" % email)
```

运行结果:

```
xiaoWang@163.com 是符合规定的邮件地址,匹配后的结果是:xiaoWang@163.com
xiaoWang@163.comheihei 是符合规定的邮件地址,匹配后的结果是:xiaoWang@163.com
.com.xiaowang@qq.com 不符合要求
```

完善后

```
email_list = ["xiaoWang@163.com", "xiaoWang@163.comheihei", ".com.xiaowang@qq.com"]

for email in email_list:
    ret = re.match("[\w]{4,20}@163\.com$", email)
    if ret:
        print("%s 是符合规定的邮件地址,匹配后的结果是:%s" % (email, ret.group()))
    else:
        print("%s 不符合要求" % email)
```

运行结果：

```
xiaoWang@163.com 是符合规定的邮件地址,匹配后的结果是:xiaoWang@163.com
xiaoWang@163.comheihei 不符合要求
.com.xiaowang@qq.com 不符合要求
```

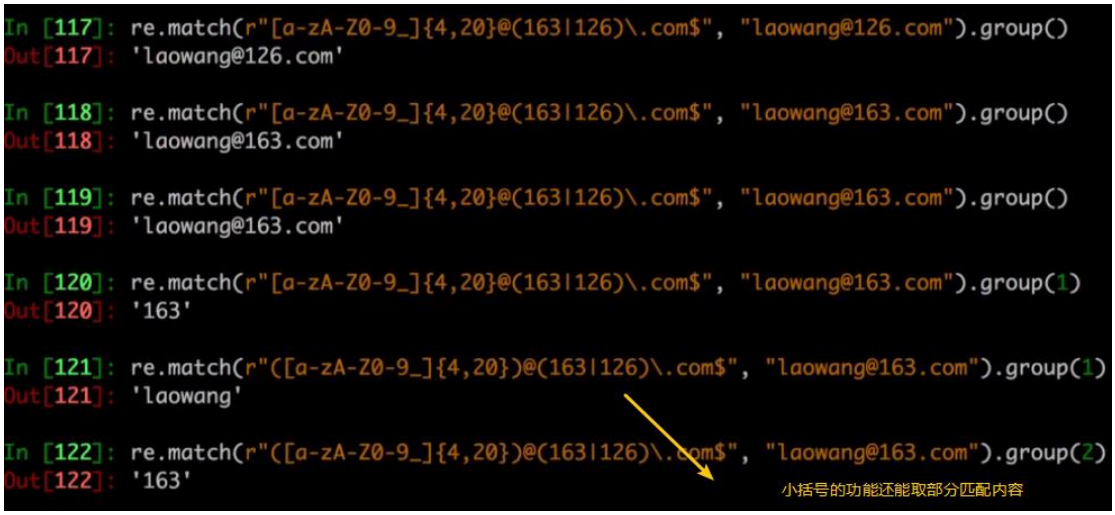
匹配邮箱地址.py

```
import re


email_list = ["abclhq2005@163.com", "abclhq2005@126.com", ".kdjla.fjdk"]

for email in email_list:

        ret = re.match("[a-zA-z0-9]+@(163|126)\.com$", email)

        if ret:
                print("%s 是符合邮件规范的，邮件地址：%s" % (email, ret.group()))
        else:
                print("%s 不符合邮件规范" % email)
```

```
python@lhq:~/Desktop/正则表达式$ python 匹配邮箱地址.py
abclhq2005@163.com 是符合邮件规范的，邮件地址：abclhq2005@163.com
abclhq2005@126.com 是符合邮件规范的，邮件地址：abclhq2005@126.com
.kdjla.fjdk 不符合邮件规范
```

1.5 匹配分组

字符	功能
|	匹配左右任意一个表达式
(ab)	将括号中字符作为一个分组
`\num`	引用分组num匹配到的字符串
`(?P<name>)`	分组起别名
(?P=name)	引用别名为name分组匹配到的字符串



提取区号和电话号码

```
>>> ret = re.match("([^-]*)-(\d+)","010-12345678")
>>> ret.group()
'010-12345678'
>>> ret.group(1)
'010'
>>> ret.group(2)
'12345678'
```

示例4：\number

需求：匹配出 `<html><h1>www.itcast.cn</h1></html>`

```
#coding=utf-8

import re

labels = ["<html><h1>www.itcast.cn</h1></html>", "<html><h1>www.itcast.cn</h2></html>"]

for label in labels:
    ret = re.match(r"<(\w*)><(\w*)>.*</\2></\1>", label)
    if ret:
        print("%s 是符合要求的标签" % ret.group())
    else:
        print("%s 不符合要求" % label)
```

运行结果：

```
<html><h1>www.itcast.cn</h1></html> 是符合要求的标签
<html><h1>www.itcast.cn</h2></html> 不符合要求
```

示例5：(?P<name>) (?P=name)

需求：匹配出 <html><h1>www.itcast.cn</h1></html>

```
#coding=utf-8

import re

ret = re.match(r"<(?P<name1>\w*)><(?P<name2>\w*)>.*</(?P=name2)></(?P=name1)>", "<html><h1>www.itc
ret.group()

ret = re.match(r"<(?P<name1>\w*)><(?P<name2>\w*)>.*</(?P=name2)></(?P=name1)>", "<html><h1>www.itc
ret.group()
```

注意：(?P<name>) 和 (?P=name) 中的字母p大写

注意：(?P<name>) 和 (?P=name) 中的字母p大写

运行结果：

```
>>> #coding=utf-8
...
>>> import re
>>>
>>> ret = re.match(r"<(?P<name1>\w*)><(?P<name2>\w*)>.*</(?P=name2)></(?P=name1)>", "<html><h1>www.itcast.cn</
h1></html>")
>>> ret.group()
'<html><h1>www.itcast.cn</h1></html>'
>>>
>>> ret = re.match(r"<(?P<name1>\w*)><(?P<name2>\w*)>.*</(?P=name2)></(?P=name1)>", "<html><h1>www.itcast.cn</
h2></html>")
>>> ret.group()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
>>>
```

1.6 re 模块的高级用法

1. search

需求：匹配出文章阅读的次数

```
#coding=utf-8
import re

ret = re.search(r"\d+", "阅读次数为 9999")
ret.group()
```

运行结果：

```
'9999'
```

```
In [136]: re.search(r"\d+", "阅读次数为 9999").group()
Out[136]: '9999'

In [137]: re.search(r"\d+", "阅读次数为 9999, 点赞数为:100").group()
Out[137]: '9999'
```

2. findall

需求：统计出python、c、c++相应文章阅读的次数

```
#coding=utf-8
import re

ret = re.findall(r"\d+", "python = 9999, c = 7890, c++ = 12345")
print(ret)
```

运行结果：

```
['9999', '7890', '12345']
```

3. sub 将匹配到的数据进行替换

需求：将匹配到的阅读次数加1

方法1：

```
#coding=utf-8
import re

ret = re.sub(r"\d+", '998', "python = 997")
print(ret)
```

运行结果：

```
python = 998
```

方法2：

```
#coding=utf-8
import re

def add(temp):
    strNum = temp.group()
    num = int(strNum) + 1
    return str(num)

ret = re.sub(r"\d+", add, "python = 997")
print(ret)

ret = re.sub(r"\d+", add, "python = 99")
print(ret)
```

运行结果：

```
python = 998
python = 100
```

4. split 根据匹配进行切割字符串，并返回一个列表

需求：切割字符串“info:xiaoZhang 33 shandong”

```
#coding=utf-8
import re

ret = re.split(r":| ","info:xiaoZhang 33 shandong")
print(ret)
```

运行结果：

```
['info', 'xiaoZhang', '33', 'shandong']
```

1.7 python 贪婪和非贪婪

Python 里数量词默认是贪婪的（在少数语言里也可能是默认非贪婪），总是尝试匹配尽可能多的字符；

非贪婪则相反，总是尝试匹配尽可能少的字符。

在"*","?","+","{m,n}"后面加上？，使贪婪变成非贪婪。

```
>>> s="This is a number 234-235-22-423"
>>> r=re.match(".+(\d+-\d+-\d+-\d+)",s)
>>> r.group(1)
'4-235-22-423'
>>> r=re.match(".+?(\d+-\d+-\d+-\d+)",s)
>>> r.group(1)
'234-235-22-423'
>>>
```

正则表达式模式中使用到通配字，那它在从左到右的顺序求值时，会尽量“抓取”满足匹配最长字符串，在我们上面的例子里面，“.+”会从字符串的启始处抓取满足模式的最长字符，其中包括我们想得到的第一个整型字段的中的大部分，“\d+”只需一位字符就可以匹配，所以它匹配了数字“4”，而“.+”则匹配了从字符串起始到这个第一位数字 4 之前的所有字符。

解决方式：非贪婪操作符“？”，这个操作符可以用在"*","+","?"的后面，要求正则匹配的越少越好。

```
>>> re.match(r"aa(\d+)","aa2343ddd").group(1)
'2343'
>>> re.match(r"aa(\d+?)","aa2343ddd").group(1)
'2'
>>> re.match(r"aa(\d+)ddd","aa2343ddd").group(1)
'2343'
>>> re.match(r"aa(\d+?)ddd","aa2343ddd").group(1)
'2343'
>>>
```

二、HTTP 协议简介

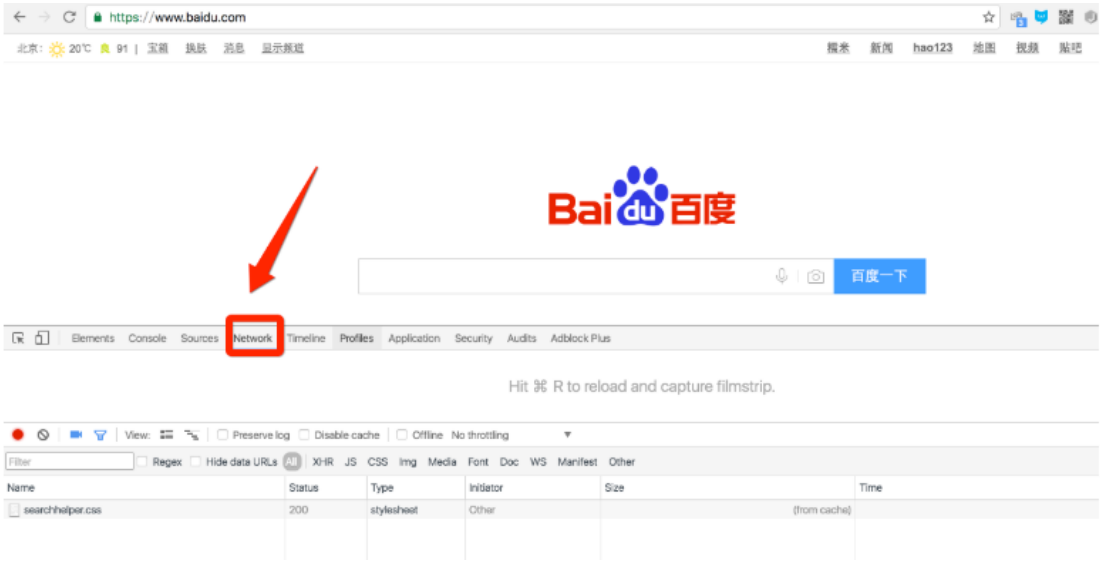
2.1. 使用谷歌/火狐浏览器分析

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。

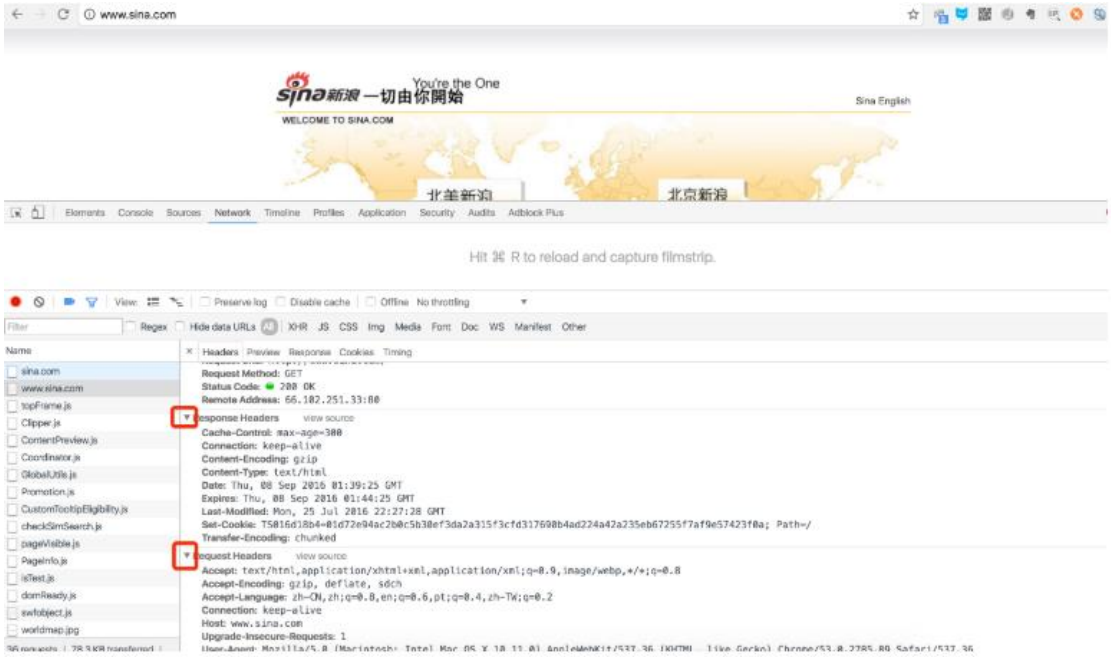
安装好 Chrome 浏览器后，打开 Chrome，在菜单中选择“视图”，“开发者”，“开发者工具”，就可以显示开发者工具：



说明

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

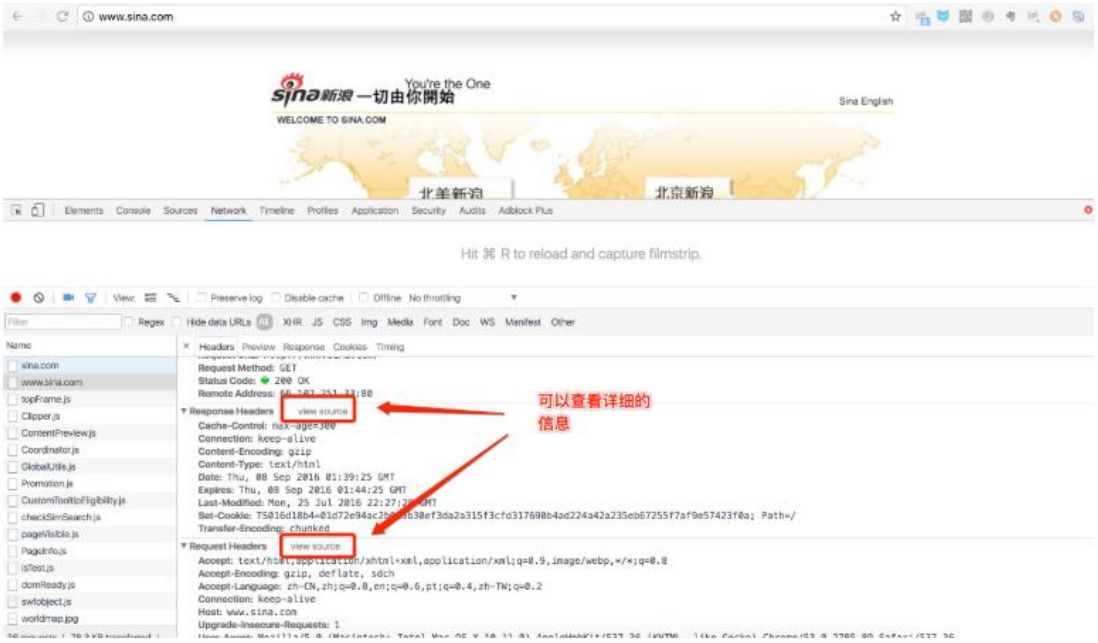
我们点 Network，确保第一个小红灯亮着，Chrome 就会记录所有浏览器和服务器之间的通信：

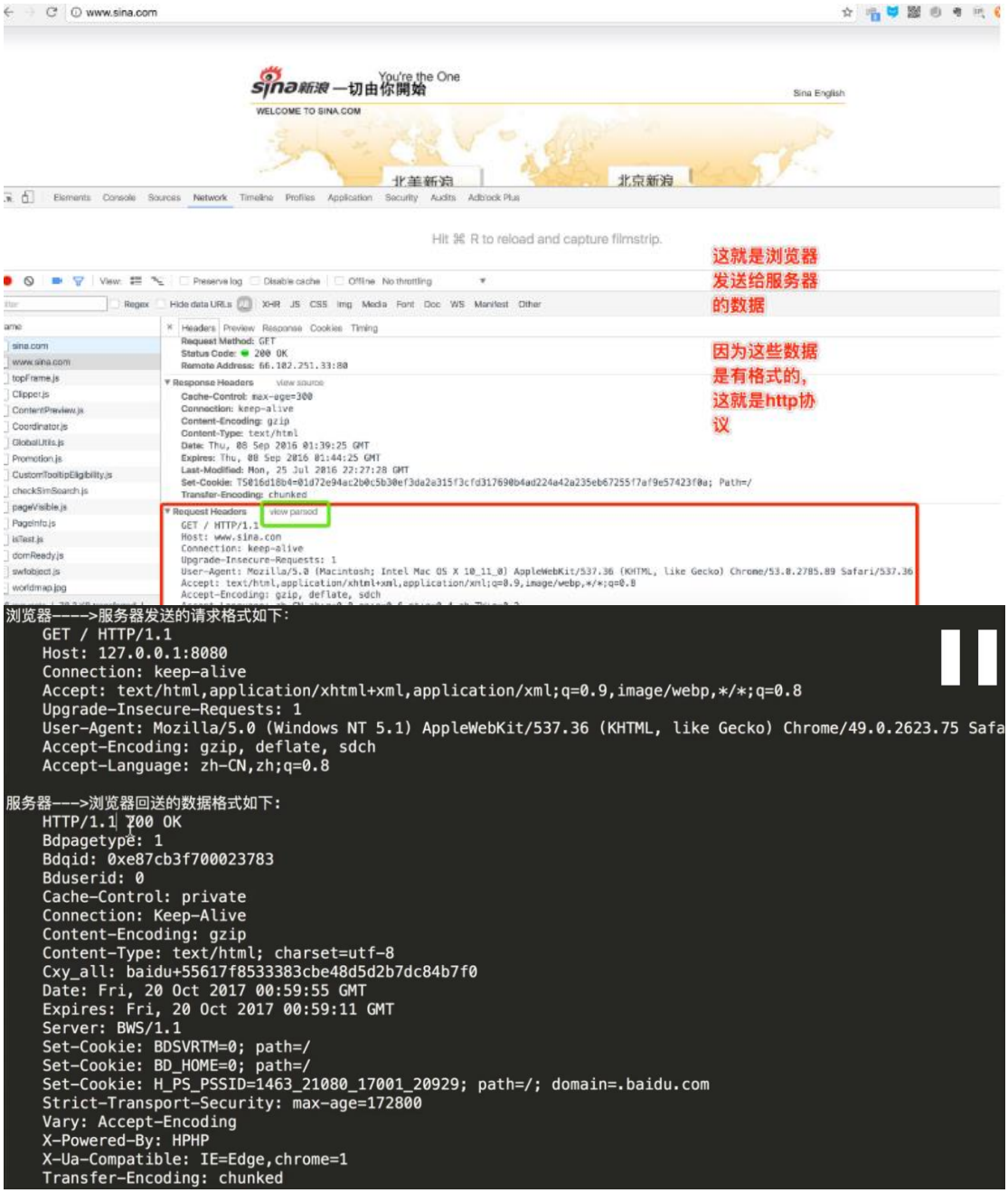


2.2. http 协议的分析

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：

2.2.1 浏览器请求






```
浏览器---->服务器发送的请求格式如下:
    GET / HTTP/1.1
    Host: 127.0.0.1:8080
    Connection: keep-alive
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
    Upgrade-Insecure-Requests: 1
    User-Agent: Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/49.0.2623.75 Safa
    Accept-Encoding: gzip, deflate, sdch
    Accept-Language: zh-CN,zh;q=0.8

服务器---->浏览器回送的数据格式如下:
    HTTP/1.1 200 OK
    Bdpagetype: 1
    Bdqid: 0xe87cb3f700023783
    Bduserid: 0
    Cache-Control: private
    Connection: Keep-Alive
    Content-Encoding: gzip
    Content-Type: text/html; charset=utf-8
    Cxy_all: baidu+55617f8533383cbe48d5d2b7dc84b7f0
    Date: Fri, 20 Oct 2017 00:59:55 GMT
    Expires: Fri, 20 Oct 2017 00:59:11 GMT
    Server: BWS/1.1
    Set-Cookie: BDSVRTM=0; path=/
    Set-Cookie: BD_HOME=0; path=/
    Set-Cookie: H_PS_PSSID=1463_21080_17001_20929; path=/; domain=.baidu.com
    Strict-Transport-Security: max-age=172800
    Vary: Accept-Encoding
    X-Powered-By: HPHP
    X-Ua-Compatible: IE=Edge,chrome=1
    Transfer-Encoding: chunked
```

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。

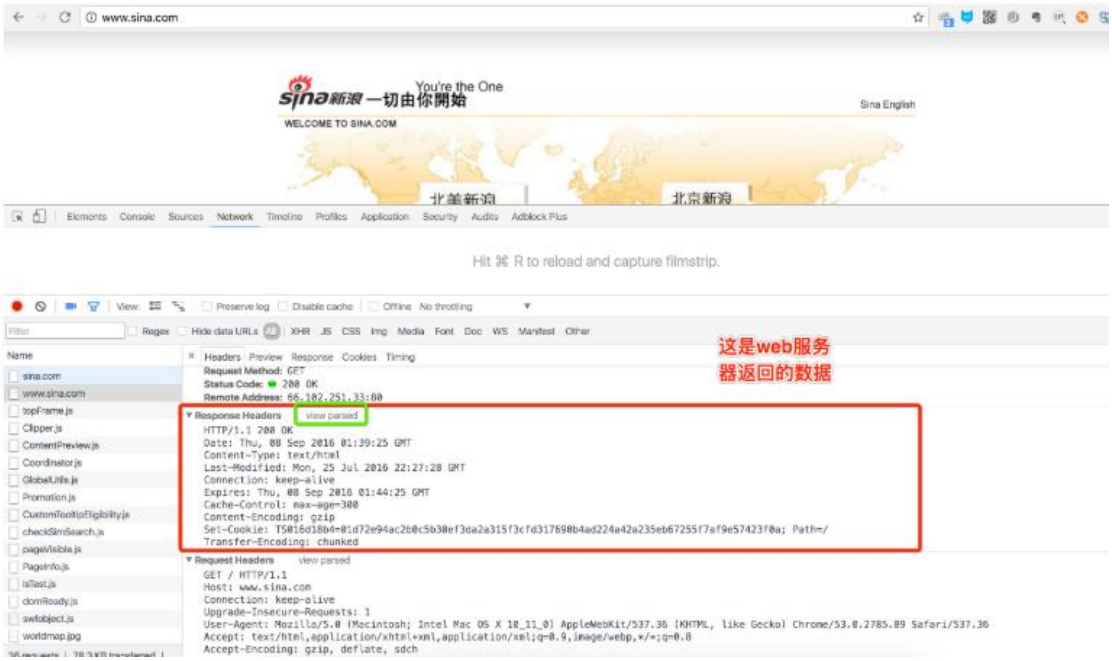
从第二行开始，每一行都类似于 Xxx: abcdefg：

```
    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。

2.2.2 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

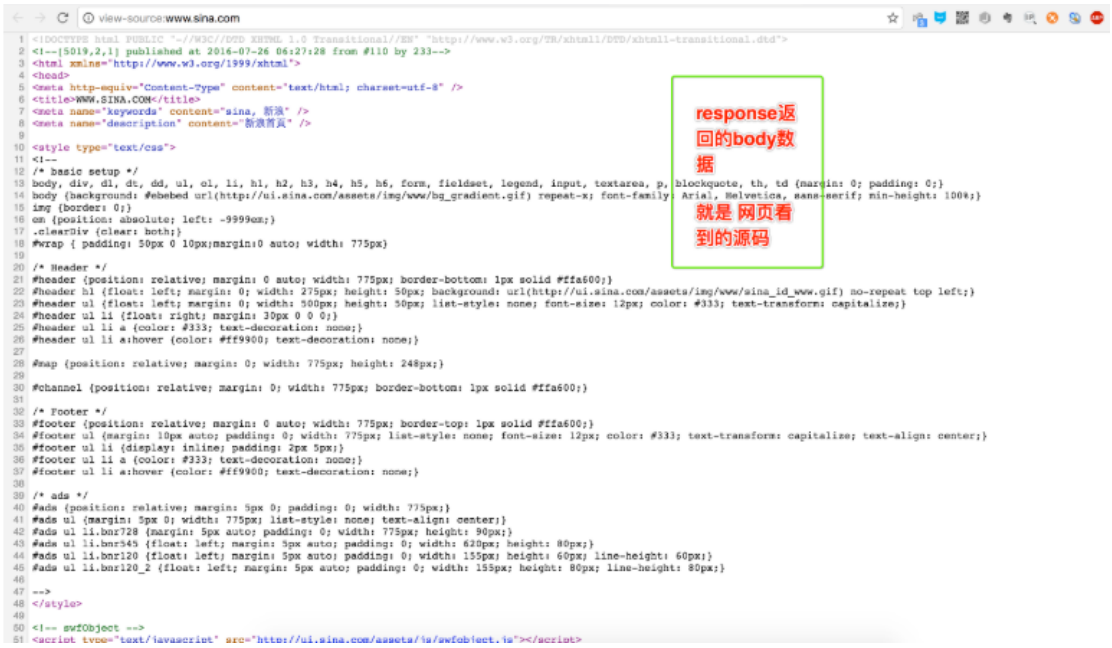
- 失败的响应有 404 Not Found：网页不存在
- 500 Internal Server Error：服务器内部出错
- ...等等...

```
Content-Type: text/html
```

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

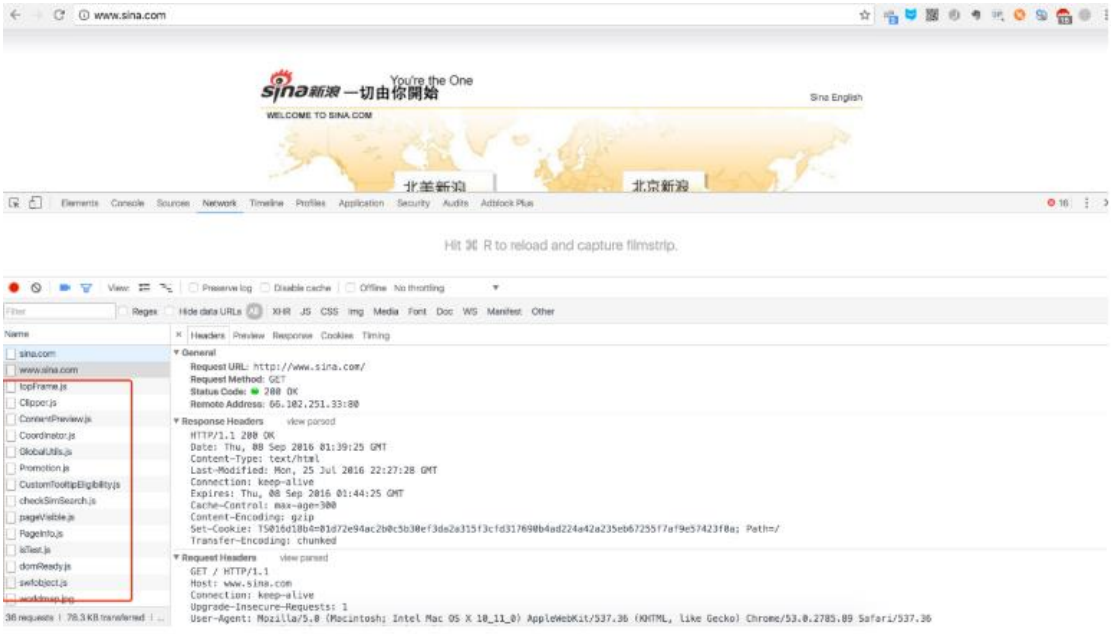
请注意，浏览器就是依靠 Content-Type 来判断响应的内容是网页还是图片，是视频还是音乐。浏览器并不靠 URL 来判断响应的内容，所以，即使 URL 是 http://www.baidu.com/meimei.jpg，它也不一定就是图片。

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：



2.2.3 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。



2.3. 总结

2.3.1 HTTP 请求

跟踪了新浪的首页，我们来总结一下 HTTP 请求的流程：

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

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；

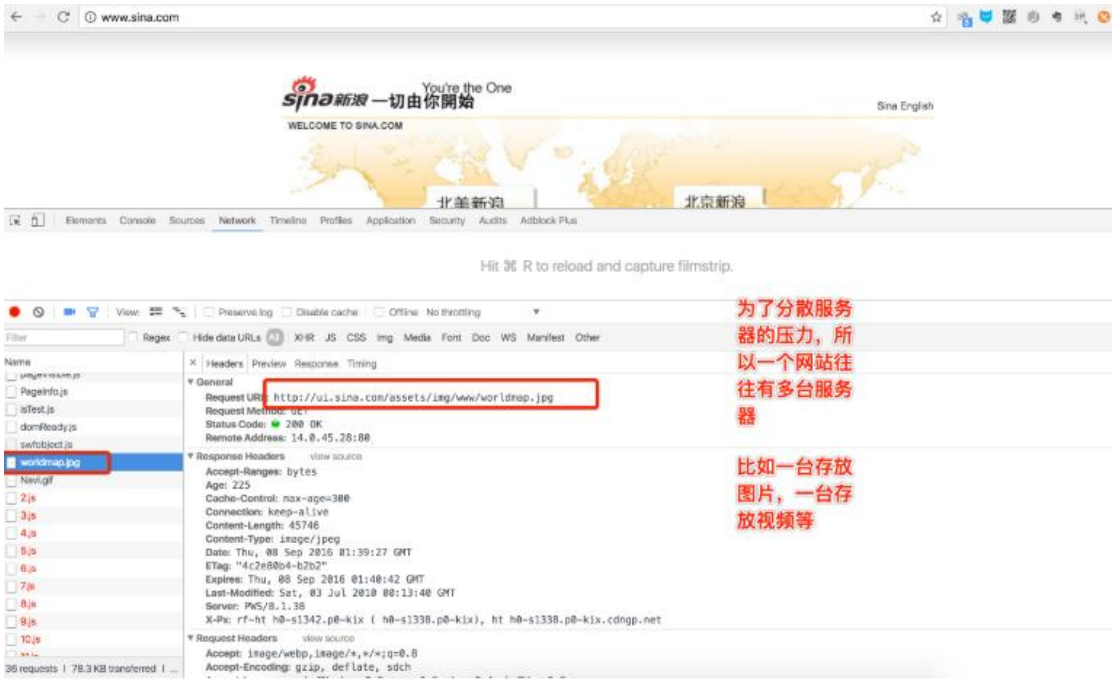
以及其他相关的 Header；

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。

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。



2.3.2 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。

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。

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。

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。

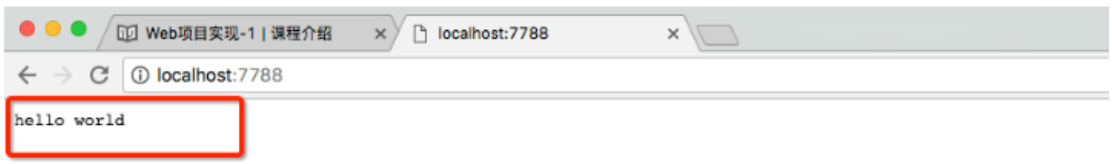
三、Web 服务器

3.1 Web 静态服务器-显示固定的页面

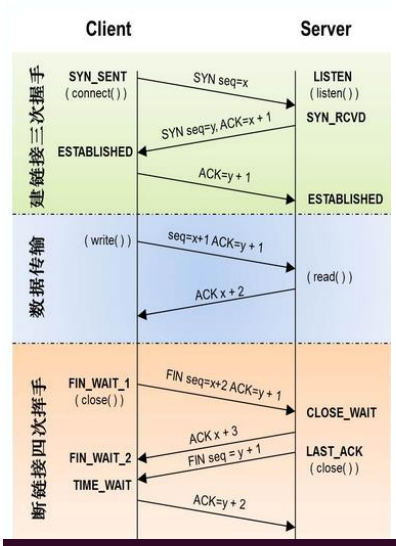
```python
import socket


def service_client(socket):
    """为客户端服务"""

    # 1.接收浏览器发送的请求，即http请求
    # GET / HTTP/1.1
    request = socket.recv(1024)
    print(request)

    # 2.返回http格式数据，给浏览器
    # 2.1 发送header
    response = "HTTP/1.1 200 OK\r\n"
    response += "\r\n"
    # 2.2 发送body
    response += "<h1>hahaha</h1>"

    socket.send(response.encode("utf-8"))
    # 关闭套接字
    socket.close()


def main():
    """用来完成整体的控制"""
    # 1.创建套接字
    tcp_server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # 2.绑定
    tcp_server_socket .bind(("", 7890))

    # 3.变为监听套接字
    tcp_server_socket.listen(128)

    while True:
        # 4.等待客户端链接
        new_socket, client_addr = tcp_server_socket.accept()

        # 5.为客户端服务
        service_client(new_socket)

    # 6.关闭套接字
    tcp_server_socket.close()


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

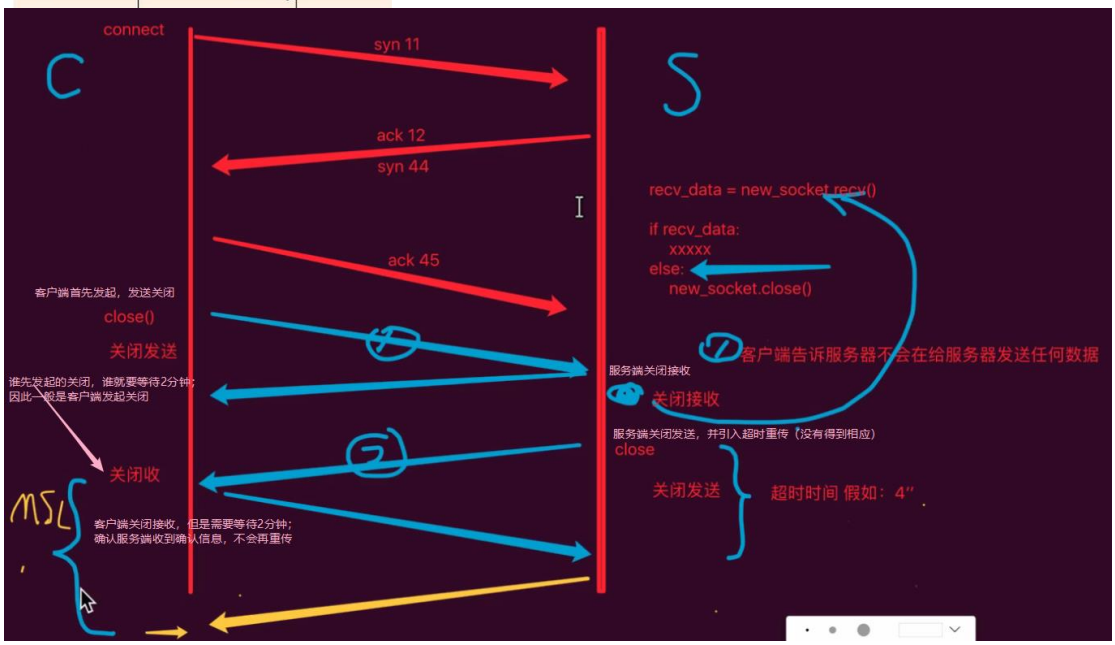
发送header

客户端



3.2 tcp 的 3 次握手和 4 次挥手





3.3 Web 静态服务器-显示需要的页面

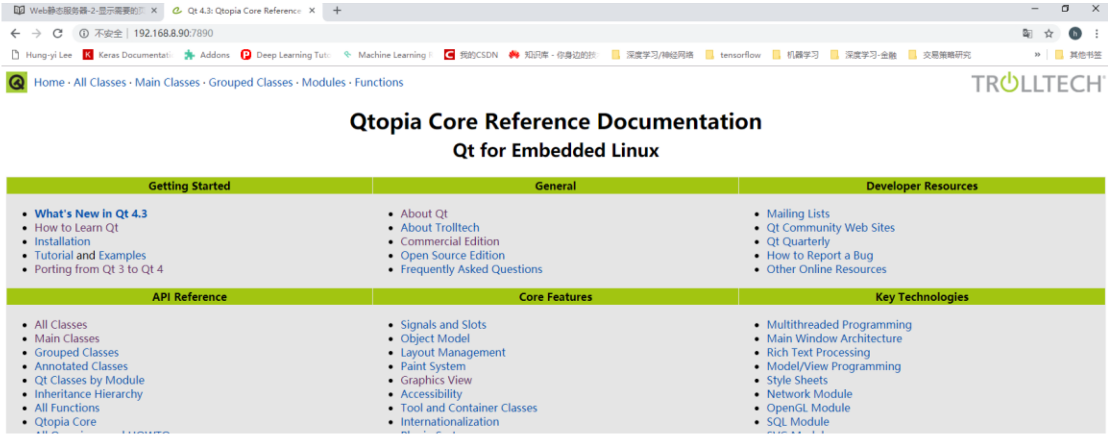
```python
import socket
import re


def service_client(socket):
    """为客户端服务"""

    # 1.接收浏览器发送的请求，即http请求
    # GET / HTTP/1.1
    request = socket.recv(1024).decode("utf-8")
    # print(">>"*50)
    # print(request)

    request_lines = request.splitlines()
    print("")
    print(">>"*50)
    print(request_lines)

    # GET /classic.css HTTP/1.1
    # get post put del
    ret = re.match(r"[^/]+(/[^ ]*)", request_lines[0])
    if ret:
        file_name = ret.group(1)
        if file_name == '/':
            file_name = "/index.html"
        print("*"*50, file_name)
    else:
        file_name = None

    try:
        f = open("./html" + file_name, "rb")
        html_content = f.read()
        f.close()
    except:
        response = "HTTP/1.1 404 NOT FOUND\r\n"
        response += "\r\n"
        response += "-----file not found------"
        # 发送header
        socket.send(response.encode("utf-8"))
    else:
        # 2.返回http格式数据，给浏览器
        # 2.1 发送header
        response = "HTTP/1.1 200 OK\r\n"
        response += "\r\n"

        # 发送header
        socket.send(response.encode("utf-8"))
        # 发送html
        socket.send(html_content)

    # 关闭套接字
    socket.close()


def main():
    """用来完成整体的控制"""
    # 1.创建套接字
    tcp_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
    tcp_server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # 2.绑定
    tcp_server_socket .bind(("", 7890))

    # 3.变为监听套接字
    tcp_server_socket.listen(128)

    while True:
        # 4.等待客户端链接
        new_socket, client_addr = tcp_server_socket.accept()

        # 5.为客户端服务
        service_client(new_socket)

    # 6.关闭套接字
    tcp_server_socket.close()


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


服务端：

```
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>>>
['GET / HTTP/1.1', 'Host: 192.168.8.90:7890', 'Connection: keep-alive', 'Upgrade-Insecure-Requests: 1', 'User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.110 Safari/537.36', 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8', 'Accept-Encoding: gzip, deflate', 'Accept-Language: zh-CN,zh;q=0.9', '']
************************************************** /index.html

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>>>>>>>>>
['GET /classic.css HTTP/1.1', 'Host: 192.168.8.90:7890', 'Connection: keep-alive', 'User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/70.0.3538.110 Safari/537.36', 'Accept: text/css,*/*;q=0.1', 'Referer: http://192.168.8.90:7890/', 'Accept-Encoding: gzip, deflate', 'Accept-Language: zh-CN,zh;q=0.9', '']
************************************************** /classic.css
```

需要提取的网址

客户端：



3.4 Web 静态服务器-多进程

```python
def main():
        """用来完成整体的控制"""
        # 1.创建套接字
        tcp_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
        tcp_server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        # 2.绑定
        tcp_server_socket .bind(("", 7890))

        # 3.变为监听套接字
        tcp_server_socket.listen(128)

        while True:
                # 4.等待客户端链接
                new_socket, client_addr = tcp_server_socket.accept()

                # 5.为客户端服务
                p = multiprocessing.Process(target=service_client, args=(new_socket,))
                p.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
                new_socket.close()

        # 6.关闭套接字
        tcp_server_socket.close()

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

多进程

多进程进行了套接字拷贝，如果父进程不关闭，子进程关闭，套接字不会关闭

3.5 Web 静态服务器-多线程

```python
def main():
    """用来完成整体的控制"""
    # 1.创建套接字
    tcp_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
    tcp_server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # 2.绑定
    tcp_server_socket .bind(("", 7890))

    # 3.变为监听套接字
    tcp_server_socket.listen(128)

    while True:
        # 4.等待客户端链接
        new_socket, client_addr = tcp_server_socket.accept()

        # 5.为客户端服务
        p = threading.Thread(target=service_client, args=(new_socket,))
        p.start()

    # 6.关闭套接字
    tcp_server_socket.close()

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

线程不用关闭套接字，因为线程不会拷贝套接字

3.5 Web 静态服务器-多协程

```python
def main():
    """用来完成整体的控制"""
    # 1.创建套接字
    tcp_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
    tcp_server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # 2.绑定
    tcp_server_socket .bind(("", 7890))

    # 3.变为监听套接字
    tcp_server_socket.listen(128)

    while True:
        # 4.等待客户端链接
        new_socket, client_addr = tcp_server_socket.accept()

        # 5.为客户端服务
        gevent.spawn(service_client, new_socket)

    # 6.关闭套接字
    tcp_server_socket.close()
```

gevent

3.6 Web 静态服务器-单进程单线程非堵塞模式

```python
def main():
    """用来完成整体的控制"""
    # 1.创建套接字
    tcp_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
    tcp_server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # 设置套接字非堵塞
    tcp_server_socket.setblocking(False)

    # 2.绑定
    tcp_server_socket .bind(("", 7890))

    # 3.变为监听套接字
    tcp_server_socket.listen(128)

    client_socket_list = []
    while True:
        time.sleep(0.5)
        try:
            # 4.等待客户端链接--创建客户端套接字
            new_socket, client_addr = tcp_server_socket.accept()
        except Exception as ret:
            # 因为设置套接字为非堵塞，因此不等待客户端到来，如果没有客户端到来，则accept会出现异常
            print("---没有新的客户端到来---")
            pass
        else:
            print("---只要没有产生异常，意味着新的客户端到来---")
            new_socket.setblocking(False)
            client_socket_list.append(new_socket)

        # 5.为客户端服务
        for client_socket in client_socket_list:
            try:
                recv_data = client_socket.recv(1024)
            except Exception as ret:
                print("---这个客户端没有发送过来数据---")
            else:
                if recv_data:
                    # 对方发送过来数据
                    print("---客户端发送过来数据---")
                    print(client_addr, recv_data)
                else:
                    # 对方调用close导致了recv返回为空
                    client_socket.close()
                    client_socket_list.remove(client_socket)
                    print("---客户端已经关闭---")

    # 6.关闭套接字
    tcp_server_socket.close()
```

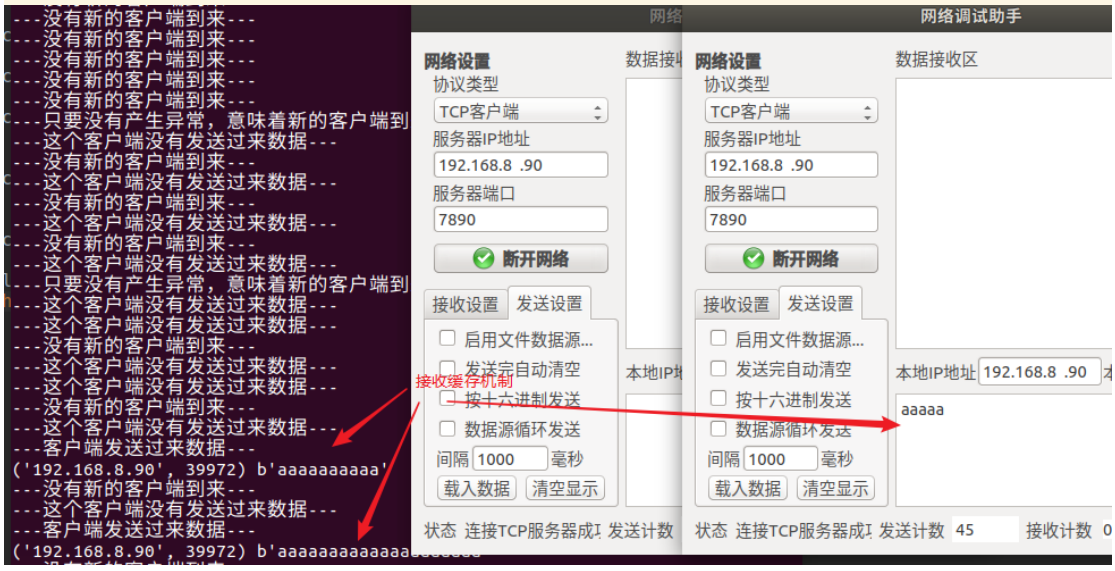
套接字设置为非堵塞

因为设置为非堵塞，当没有连接时会产生异常；

客户套接字设置为非堵塞

没有收到数据报异常

接收客户套接字不报异常的两种情况



同时处理两个客户端，同时因为程序里面有等待，导致了快速发送数据时，接收缓存。

3.7 tcp 长连接和短连接

TCP 在真正的读写操作之前，server 与 client 之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接时它们可以释放这个连接，连接的建立通过三次握手，释放则需要四次握手，所以说每个连接的建立都是需要资源消耗和时间消耗的。

3.7.1. TCP 短连接

模拟一种 TCP 短连接的情况:

- client 向 server 发起连接请求
- server 接到请求，双方建立连接
- client 向 server 发送消息
- server 回应 client
- 一次读写完成，此时双方任何一个都可以发起 close 操作

在步骤 5 中，一般都是 client 先发起 close 操作。当然也不排除有特殊的情况。

从上面的描述看，短连接一般只会在 client/server 间传递一次读写操作！

3.7.2. TCP 长连接

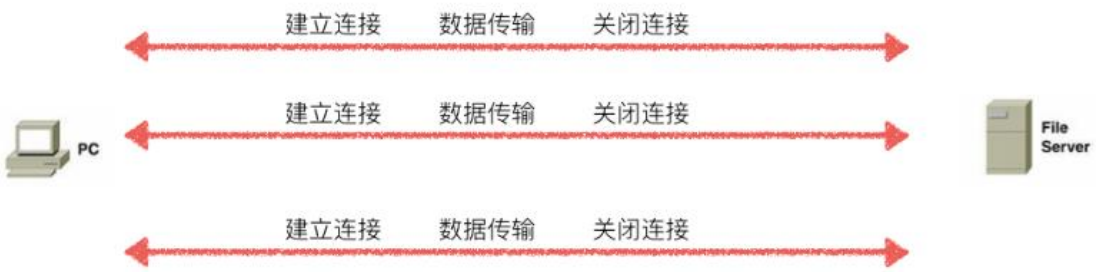
再模拟一种长连接的情况:

- client 向 server 发起连接
- server 接到请求，双方建立连接
- client 向 server 发送消息
- server 回应 client
- 一次读写完成，连接不关闭
- 后续读写操作...
- 长时间操作之后 client 发起关闭请求

3.7.3. TCP 长/短连接操作过程

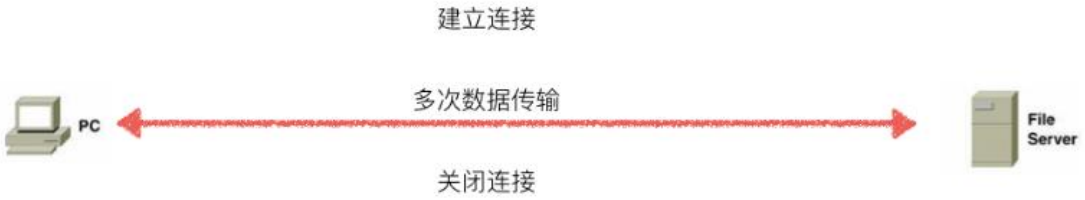
1 短连接的操作步骤是：

建立连接——数据传输——关闭连接...建立连接——数据传输——关闭连接



2 长连接的操作步骤是：

建立连接——数据传输...（保持连接）...数据传输——关闭连接



4. TCP 长/短连接的优点和缺点

- 长连接可以省去较多的 TCP 建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户来说，较适用长连接。
- client 与 server 之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server 早晚有扛不住的时候，这时候 server 端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致 server 端服务受损；
- 如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。
- 短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。
- 但如果客户请求频繁，将在 TCP 的建立和关闭操作上浪费时间和带宽。

5. TCP 长/短连接的应用场景

- 长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况。每个 TCP 连接都需要三次握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，再次处理时直接发送数据包就 OK 了，不用建立 TCP 连接。
- 例如：数据库的连接用长连接，如果用短连接频繁的通信会造成 socket 错误，而且频繁的 socket 创建也是对资源的浪费。
- 而像 WEB 网站的 http 服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像 WEB 网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

6. 之前代码分析

之前的多进程-多线程-多协程版本，用的处理函数：

```python
def service_client(socket):
    """为客户端服务"""

    # 1.接收浏览器发送的请求，即http请求
    # GET / HTTP/1.1
    request = socket.recv(1024).decode("utf-8")
    # print(">>"*50)
    # print(request)

    request_lines = request.splitlines()
    print("")
    print(">>"*50)
    print(request_lines)

    # GET /classic.css HTTP/1.1
    # get post put del
    ret = re.match(r"[^/]+(/[^ ]*)", request_lines[0])
    if ret:
        file_name = ret.group(1)
        if file_name == '/':
            file_name = "/index.html"
        print("*"*50, file_name)
    else:
        file_name = None

    try:
        f = open("./html" + file_name, "rb")
        html_content = f.read()
        f.close()
    except:
        response = "HTTP/1.1 404 NOT FOUND\r\n"
        response += "\r\n"
        response += "-----file not found------"
        # 发送header
        socket.send(response.encode("utf-8"))
    else:
        # 2.返回http格式数据，给浏览器
        # 2.1 发送header
        response = "HTTP/1.1 200 OK\r\n"
        response += "\r\n"

        # 发送header
        socket.send(response.encode("utf-8"))
        # 发送html
        socket.send(html_content)

    # 关闭套接字
    socket.close()
```

接收到数据之后，客户端都关闭套接字，则为短链接；

3.8 Web 静态服务器-单进程单线程非堵塞-长连接模式

```python
def main():
    """用来完成整体的控制"""
    # 1.创建套接字
    tcp_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
    tcp_server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # 设置套接字为非堵塞
    tcp_server_socket.setblocking(False)

    # 2.绑定
    tcp_server_socket .bind(("", 7890))

    # 3.变为监听套接字
    tcp_server_socket.listen(128)

    client_socket_list = list()

    while True:                                                    单进程-单线程-非堵塞模式
        # 4.等待客户端链接
        try:
            new_socket, client_addr = tcp_server_socket.accept()
        except Exception as ret:
            # 没有链接报异常
            pass
        else:
            new_socket.setblocking(False)
            client_socket_list.append(new_socket)

        # 5.为客户端服务
        for client_socket in client_socket_list:
            try:
                recv_data = client_socket.recv(1024).decode('utf-8')
            except Exception as ret:
                # 客户端没有收到数据报异常
                pass
            else:
                if recv_data:
                    service_client(client_socket, recv_data)
                else:
                    # 收到数据为空，说明客户端套接字关闭链接
                    print("---客客户端关闭链接---")
                    client_socket.close()
                    client_socket_list.remove(client_socket)

    # 6.关闭套接字
    tcp_server_socket.close()
```

```python
def service_client(socket, request):
    """为客户端服务"""

    request_lines = request.splitlines()
    print("")
    print(">>"*50)
    print(request_lines)

    # GET /classic.css HTTP/1.1
    # get post put del
    ret = re.match(r"[^/]+(/[^ ]*)", request_lines[0])
    if ret:
        file_name = ret.group(1)
        if file_name == '/':
            file_name = "/index.html"
        print("*"*50, file_name)
    else:
        file_name = None

    try:
        f = open("./html" + file_name, "rb")
        html_content = f.read()
        f.close()
    except:
        print("-----发送404----")
        response = "HTTP/1.1 404 NOT FOUND\r\n"
        response += "\r\n"
        response += "-----file not found------"
        # 发送header
        socket.send(response.encode("utf-8"))
    else:
        print("----发送202-----")
        response_body = html_content

        response_header = "HTTP/1.1 200 OK\r\n"
        response_header += "content-length: %d\r\n" % len(response_body)
        response_header += "\r\n"

        response = response_header.encode('utf-8') + response_body

        # 发送header
        socket.send(response)
```

长连接时，必须返回长度，
告诉客户端该数据长度后，
客户端才知道接收数据完毕

3.9 Web 静态服务器-epoll

什么是 epoll

epoll 是什么？按照 man 手册的说法：是为处理大批量句柄而作了改进的 poll。当然，这不是 2.6 内核才有的，它是在 2.5.44 内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)，它几乎具备了之前所说的一切优点，被公认为 Linux2.6 下性能最好的多路 I/O 就绪通知方法。

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。

```python
def service_client(socket, request):
    """为客户端服务"""

    request_lines = request.splitlines()
    print("")
    print(">>"*50)
    print(request_lines)

    # GET /classic.css HTTP/1.1
    # get post put del
    ret = re.match(r"[^/]+(/[^ ]*)", request_lines[0])
    if ret:
        file_name = ret.group(1)
        if file_name == '/':
            file_name = "/index.html"
        print("*"*50, file_name)
    else:
        file_name = None

    try:
        f = open("./html" + file_name, "rb")
        html_content = f.read()
        f.close()
    except:
        print("-----发送404----")
        response = "HTTP/1.1 404 NOT FOUND\r\n"
        response += "\r\n"
        response += "-----file not found------"
        # 发送header
        socket.send(response.encode("utf-8"))
    else:
        print("----发送202-----")
        response_body = html_content

        response_header = "HTTP/1.1 200 OK\r\n"
        response_header += "content-length: %d\r\n" % len(response_body)
        response_header += "\r\n"

        response = response_header.encode('utf-8') + response_body

        # 发送header
        socket.send(response)
```

```python
def main():
    """用来完成整体的控制"""
    # 1.创建套接字
    tcp_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
    tcp_server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    # 2.绑定
    tcp_server_socket .bind(("", 7890))

    # 3.变为监听套接字
    tcp_server_socket.listen(128)

    # 创建一个epoll对象，操作系统和应用程序共享内存空间
    epl = select.epoll()
    # 将监听套接字对应的fd注册到epoll中
    epl.register(tcp_server_socket.fileno(), select.EPOLLIN)

    fd_event_dict = dict()

    while True:

        fd_event_list = epl.poll()  # 默认会堵塞，直到os监测到数据到来，通过事件通知，告诉程序，此时才会解堵塞

        # [(fd,event), (套接字对应的文件描述符， 这个文件描述符到底是什么事件 例如 可以调用recv接收), (), (),...]
        for fd, event in fd_event_list:

            if fd == tcp_server_socket.fileno():
                # 等待客户端链接
                new_socket, client_addr = tcp_server_socket.accept()
                # 注册客户端套接字
                epl.register(new_socket.fileno(), select.EPOLLIN)
                fd_event_dict[new_socket.fileno()] = new_socket

            elif event == select.EPOLLIN:
                # 判断已经链接的客户端是否有数据发送过来
                client_socket = fd_event_dict[fd]
                recv_data = client_socket.recv(1024).decode('utf-8')
                if recv_data:
                    service_client(client_socket, recv_data)
                else:
                    # 收到数据为空，说明客户端套接字关闭链接
                    print("---客客户端关闭链接---")
                    client_socket.close()
                    epl.unregister(fd)
                    del fd_event_dict[fd]

    # 6.关闭套接字
    tcp_server_socket.close()
```


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。

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。

参考资料

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

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