HTTP

> `HTTP`协议属于建立在`TCP`协议中的应用层上的一种协议，`HTTP`协议以客户端请求和服务端应答为标准
>
> 浏览器通常被人称为客户端，`Web`服务器常被称作服务端。`HTTP`协议常用端口为`80`，客户端首先通过`80`端口向`HTTP`服务端发起请求，建立`TCP`连接；之后进行`HTTP`数据传输。
>
> `bye the way`：加持`ssl`或`tls`加密协议的`HTTP`协议

协议	解释
HTTP	应用层
SSL/TLS	加密层
TCP	传输层
IP	传输层
数据链路层	物理层

> 浏览器作为客户端访问服务器之后
>
> 取到所有所需的数据
>
> **立即断开`TCP`连接**
>
> 整个`HTTP`连接过程非常短，所以人们也常称`HTTP`协议为**无连接**的协议，每一次`HTTP`的请求，都是重新开启一个新的连接，而不是在一个历史连接状态下持续工作

资源定位标识符

> 发起`HTTP`请求的内容资源由**统一资源标识符**（`Uniform Resource Identifiers`）`URI`进行表示
>
> 其中资源定位标识符主要有三种

- `URI`

  > 统一资源标识符（`uniform resourceidentifier`）
  >
  > 唯一表示一个资源

- `URN`

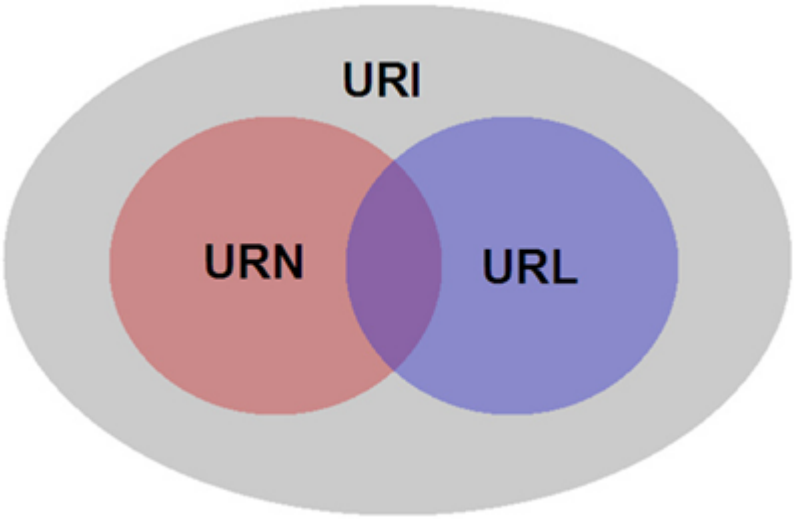
  > 统一资源命名（`uniform resource name`）
  >
  > 通过名字来表示资源

- `URL`

  > 统一资源定位器（`uniform resource locaotor`）
  >
  > 一个具体的`URI`标示，可以通过该标识获取访问到对应资源

> 其中如果已经是一个`url`，那么肯定是一个`url`标识

> 但是一个`url`并不一定是一个`url`；还可能是一个`urn`
>
> `url`是`url`的一种表现形式，就像下面这幅图



- `ftp://ftp.is.co.za/rfc/rfc1808.txt` (also a URL because of the protocol)
- `http://www.ietf.org/rfc/rfc2396.txt` (also a URL because of the protocol)
- `ldap://[2001:db8::7]/c=GB?objectClass?one` (also a URL because of the protocol)
- `mailto:John.Doe@example.com` (also a URL because of the protocol)
- `news:comp.infosystems.www.servers.unix` (also a URL because of the protocol)
- `tel:+1-816-555-1212`
- `telnet://192.0.2.16:80/` (also a URL because of the protocol)
- `urn:oasis:names:specification:docbook:dtd:xml:4.1.2`

> 以上这些都是`URI`，但是只有那些提供了访问机制的属于`URL`

URL

> `URL`是资源标识符
>
> 而`URL`不光具有标识性，还需要具有定位性，用来更加具体描述资源的具体位置，并且还会指明获取资源所采用的协议
>
> 一个`URL`包含，主机名称（`IP`或域名）、端口号（默认为`80`）、路径和查询字符串`5`部分

```
http://www.example.com:80/aaa/test.jpg?id=1
```

- 网络协议为：`http`
- 主机：`www.example.com`
- 端口：`80`
- 路径`aaa/test.jpg`
- 查询字符串参数：`id=1`

URN

> `URN`也是`URI`的一种表现形式，包括名字（给定的命名空间）
>
> 但是不包括访问的方式

> 一个URN就像这样：www.example.com/aaa/test.jpg

统一资源定位表示器

protocol://host[:port]/path/…/[?query-string][#anchor]
http://www.example/abc/index.html?id=1#index

连接部分	介绍
protocol	访问协议，常见为：http、https、ftp、rsync等；
host	服务器的IP地址或域名；
port	服务器的端口，http默认为80；
path	访问资源在服务器的路径；
query-string	访问时提交给服务器的参数及字符串；
anchor	锚点符，返回资源中进行定位所使用；

Request请求

> 当我们使用HTTP协议访问某个连接时，首先需要向服务器提交一个Request请求；
>
> 一般当我们使用浏览器访问web服务时，由浏览器完成这个工作，Request消息分为三部分，分别包括：Request Line、Request Header、Body

- 下面为HTTP协议在Request时的消息体构造

请求方法	空	URL	空	协议版本	\r回车	\n换行	Request Line
头部字段	:冒号	对应字段	\r回车	\n换行	Request	Header	
头部字段	:冒号	对应字段	\r回车	\n换行	Request	Header	
\r回车	\n换行				Request	Header	
其他数据	Body						

```
#request
"GET / HTTP/1.1\r\n" + \
"Host: 127.0.0.1\r\n"+ \
"Accept: text/html\r\n" + \
"\r\n"
```

- 这是一个通过curl命令获取到的访问web服务时的信息

```
* About to connect() to baidu.com port 80 (#0)
*   Trying 123.125.115.110...
* Connected to baidu.com (123.125.115.110) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.29.0
> Host: baidu.com
> Accept: */*
>
```

请求方式

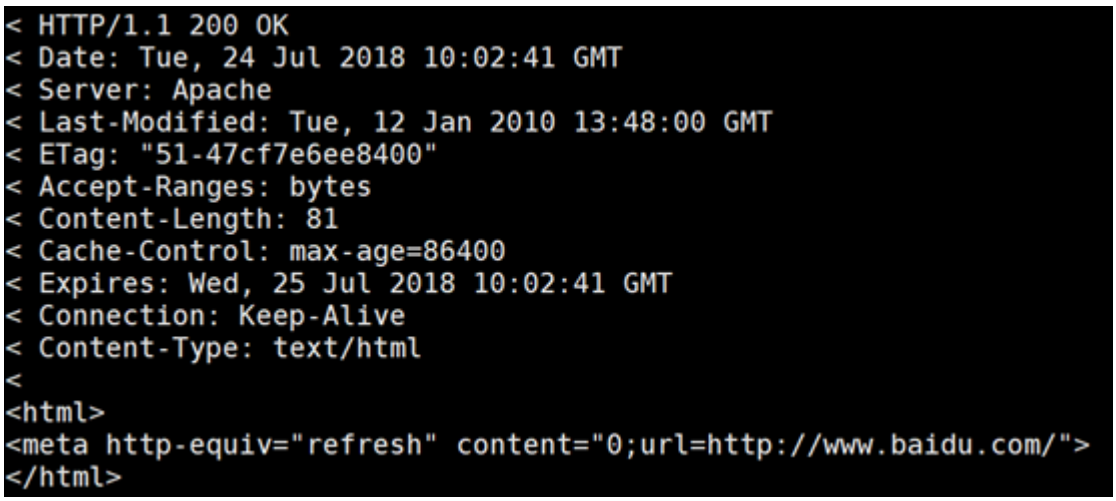
请求方式	解释
`GET`	获取服务端数据；
`POST`	向服务端提交数据；
`PUT`	向服务端更新数据；
`DELETE`	删除服务端通过 `Request-URL` 所标示的资源；
`TRACE`	测试服务端是否可以接收到 `Request` 请求；
`CONNECT`	以管道方式连接代理服务器；
`OPTIONS`	返回服务器所支持的其他 `HTTP` 请求方法；
`HEAD`	与 `GET` 方法类似，但不返回服务器响应时的消息体；

Response响应

> 服务器接收到之后，会向我们返回一个 `Response` 响应，浏览器接收到了 `Response` 之后
>
> 也同样会帮助我们对信息进行解析，之后我们就可以看到对应的 `web` 页面或者是获取到的资源

- 下面是 `HTTP` 协议在 `Response` 响应时的消息体构造

响应版本	空	状态码	空	状态信息	`\r` 回车	Response Line
头部字段	:冒号	对应字段	`\r` 回车	`\n` 换行	`Response`	`Header`
头部字段	:冒号	对应字段	`\r` 回车	`\n` 换行	`Response`	`Header`
其他	头部	字段	属性			
`\r` 回车	`\n` 换行					
响应	数据	`body`				




```
#response
'''
< HTTP/1.1 200 OK
< Date: Tue, 24 Jul 2018 10:02:41 GMT
< Server: Apache
…
< Content-Type: text/html
<
<html>
<meta http-equiv="refresh" content="0;
url=http://www.baidu.com/">
</html>
'''
```

Response状态码

> 关于`HTTP`中`Response`返回的状态码详解
>
> 这个码常用来描述`web`服务器对于用户请求动作的返回状态，有如下这些

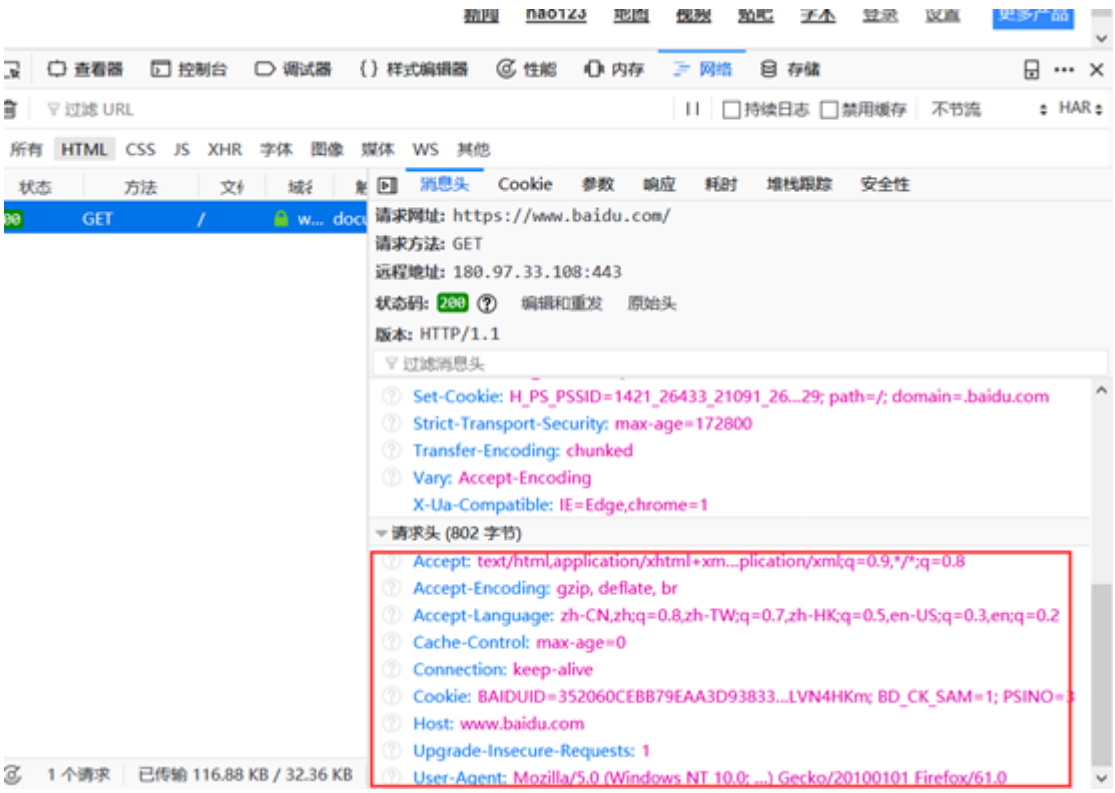
CODE	状态码英文标示	意义
100	Continue	`HTTP/1.1` 中新增状态码，表示客户端可以继续请求 `HTTP` 服务器
101	Switching Protocols	服务端切换请求协议，切换到 `HTTP` 协议新版本
200	OK	客户端的请求服务端正常完成
301	Moved Permanently	客户端请求的资源已被永久移动到新的 `URL`
302	Found	客户顿请求的资源被临时移动，客户端继续使用原有 `URL`；常用于三方登录之后的跳转
304	Not Modified	请求的资源未被修改，可以继续访问原 `URL` 常用于缓存
400	Bad Request	客户端的请求语法错误，或无法解析请求
401	Unauthorized	请求需要经过身份验证
402	Payment Required	保留状态码，为以后使用做准备的呢
403	Forbidden	服务端直接拒绝客户端的请求
404	Not Found	客户端请求的资源找不到
405	Method Not Allowed	客户端请求的方式不被允许
406	Not Acceptable	客户端请求的内容无法正常完成
499	Client has closed Connection	服务端处理该请求花费了太长的时间
500	Internal Server Error	服务端内部错误；可能是因为 `web` 服务配置文件读取错误，也可能是因为用户权限等等问题导致
502	Bad Geteway	服务端内部错误；服务端错误的网关
503	Service Unavailable	服务端无法响应客户端的请求
504	Gateway Time-out	服务端处理请求超时；或者可能是访问的网管超时；
505	HTTP Version not Supported	客户端请求的 `HTTP` 协议版本无法被服务端支持；可能是你的浏览器太古老了

请求|响应报文字段

> 经常，我们会以浏览器作为客户端访问 `web` 服务
>
> 这里我们以火狐浏览器为例，说明在访问时的字段属性

- 我们可以在访问站点时，在浏览器中按 `F12` 打开开发者工具，如图所示

- 在下方，我们点击进入第一个`get`方法的行，会看到右边额外展开了访问时的详细信息



> 此时我们看到的请求头，就是当我们通过浏览器访问一个`URL`时，会向服务器发送的主要数据
>
> 我们经常叫这个为`Request Headers`，请求头；
>
> 在请求头中，维护了很多字段信息

request-header

- 常见的`HTTP`请求头`request`中字段的详细解释如下

> 参考地址：https://en.wikipedia.org/wiki/List_of_HTTP_header_fields

request-header	describe
Accept: text/html	指定客户端支持接收的数据类型（`MIME`类型）
Accept-Charset: utf-8	指定客户端可接受的字符集，缺省表示任何字符集都可以接受
Accept-Encoding: gzip, deflate	指定客户端支持的Web服务器返回内容压缩编码类型
Accept-Language: zh-CN	指定客户端可以接受的语言
Authorization:Basic xxx=	`HTTP`认证的权限
Cache-Control: no-cache	指定所有缓存机制必须遵从的指令
Connection: keep-alive	表示是否需要持久连接（`HTTP1.1`协议中默认进行持久连接）
Content-Length: 123	请求的内容长度
Content-Type: text/html	用于在`post`及`put`时请求的媒体类型
Cookie: $XXX	保存在该请求域名下的`COOKIE`值信息，常用于表示用户的历史访问记录
Date: Tue, 15 Nov	请求发送的日期时间
Referer: ...	访问到这个站点之前的地址，用来指定访问从何处而来
User-Agent: ...	指定当前访问者的用户信息，经常为一些浏览器、操作系统属性

response-header

- 常见的`HTTP`响应头`response`中字段的详细解释如下

response-header	describe
Age: 775	从原始服务器到代理缓存形成的时间估算（秒为单位）
Allow: GET,HEAD	指定资源的有效请求行为，不允许请求则返回`405`
Cache-Control: max-age=3600	缓存机制，`max-age`参数未缓存有效时间（秒为单位）
Content-Encoding: gzip	响应内容的支持的压缩编码类型
Content-Disposition: attachment;	可下载资源属性（`MIME`二进制格式），提供文件下载的对话框，并默认文件名为`filename`
Content-Length: 348	响应的数据长度
Content-Language: en-US	响应的数据语言格式
Content-Location: /index.html	响应资源的替代地址
Content-Type: text/html;charset=utf-8	响应数据的`MIME`类型
Date: Tue, 15 Nov...	服务器发送响应时的日期时间
Expires: Tue, 15 Nov...	响应过期的日期时间
Server: Nginx	`Web`服务器名称
Set-Cookie: username=test; Max-Age=3600;	设置`HTTP`响应中的`cookie`值

HTTP支持的MIME类型

> 经常服务端会返回很多类型的内容给到客户端，这就需要在返回的过程中在`Response`中指明返回的文件类型
>
> 然后浏览器等各种客户端工具可以通过解析在`Response`中涉及到的`MIME`类型，从而对各种文件使用已有应用程序来进行解释
>
> 在`HTTP`协议中，`MIME`类型被定义在`Conetent-Type字`段处

- `application/msexcel`
  > 微软`Execl`文件；`.xls .xla`
- `application/mshelp`
  > 微软`windows`帮助文件；`.hlp .chm`
- `application/mspowerpoint`
  > 微软`PowerPoint`文件；`.ppt .ppz .pps .pot`
- `application/msword`
  > 微软`Word`文件；`.doc .dot`
- `application/octet-stream`

> 二进制可执行文件；`.exe`

- `application/pdf`

> `Adobe PDF`文件；`.pdf`

- `application/post******`

> `Adobe post***`文件；`.ai .eps .ps`

- `application/rtf`

> 微软`rtf`文件；.rtf

- `application/x-httpd-php`

> `PHP`文件；`.php .phtml`

- `application/x-java******`

> 服务端的`java***`文件；`.js`

- `application/x-shockwave-flash`

> `Shockwave-Flash`文件；`.swf .cab`

- `application/zip`

> `ZIP`存档文件；`.zip`

- `audio/basic`

> 音频文件；`.au .snd`

- `audio/mpeg`

> `MPEG`文件；`.mp3`

- `audio/x-midi`

> `MIDI`文件；`.mid .midi`

- `audio/x-mpeg`

> `MPEG`文件；`.mp2`

- `audio/x-wav`

> `Wav`文件；`.wav`

- `image/gif`

> `Gif`文件；`.gif`

- image/jpeg

> `JPEG`文件；`.jpeg .jpg .jpe`

- `image/x-windowdump`

> X-Windows存储文件；.xwd

- `text/css`

  > css样式表文件；.css

- `text/html`

  > html类型；.html .htm .shtml

- `text/java******`

  > `Java***`文件；`.js`

- `text/plain`

  > 文本类型；`.txt`

- `video/mpeg`

  > `MPEG`文件；`.mpeg .mpg .mpe`

- `video/vnd.rn-realvideo`

  > 在线播放视频文件；`.rmvb`

- `video/quicktime`

  > `Quicktime`文件，苹果公司提供的视频文件格式；`.qt .mov`

- `video/vnd.vivo`

  > `Vivo`文件；`.viv .vivo`

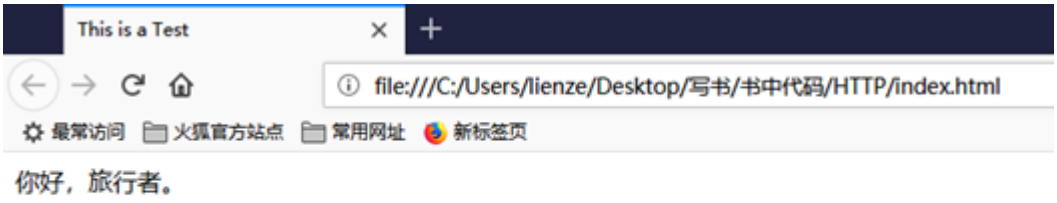
手动构造B/S

> 现在我们将通过实现协议的方式并且结合`TCP`套接字来实现一个简单的`WEB`客户端及服务端

- 先编写一个简单的网页

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>This is a Test</title>
</head>
<body>
    <div>你好，旅行者。</div>
</body>
</html>
```

- 这个页面长这样



> 要注意在浏览器中，我们访问的地址该`html`文件的绝对路径，并不是通过一个网络中的`URL`地址来访问

- 编写一个`tcp`客户端，来进行页面的返回

```
# httpserver.py
import socket

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

with open('index.html',encoding='utf-8') as fp:
    html_data = fp.read()

msg = "HTTP/1.1 200 OK \r\n" + \
      "Server: Nice\r\n" + \
      "Content-Type: text/html; charset=utf-8\r\n" + \
      "Content-Length: %d\r\n" % len(html_data) + \
      "\r\n" + \
      html_data

s.bind(('',80))
s.listen(5)
print('[+] wait for connection...')
while True:
    try:
        c,c_addr = s.accept()
        print('[+] from:%s' % (c_addr,))
        print('[+] data:\n%s' % c.recv(1024).decode())
        c.send(msg.encode())
        c.close()
    except KeyboardInterrupt:
        break
        print('[+] Server close...')
```

> 在这段代码中，我们实现了一个`web`服务端，用来返回一个`index.html`页面
>
> 首先通过`with…as…`语句打开`html`文件并保存到`html_data`变量中，以备接下来客户端访问时返回
>
> 接着在构造`HTTP`协议`Response`字符串`msg`，其中声明协议为`HTTP1.1`版本，并且提供返回的数据类型为`text/html`，以及返回数据的长度

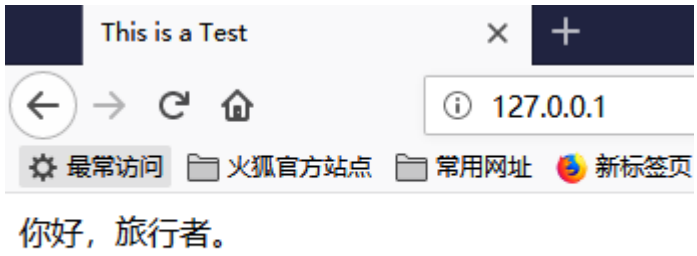
> 通过创建`TCP`套接字，并且绑定所有可用`IP`地址及端口`80`
>
> 因为浏览器打开连接时默认会通过`80`端口发起请求

> 服务端在接收到客户端的连接时，打印来访者地址及提交的数据，然后返回我们构造好的`HTTP`协议字符串数据，返回之后便关闭来访者套接字，模拟实现`http`协议短链接状态

- 运行服务端代码

```
C:\Users\lienze\Desktop\写书\书中代码\HTTP>python3 HTTP服务端.py
[+] wait for connection...
```

- 浏览器访问本地回环路由地址：`127.0.0.1`



- 右键点击查看网站源代码

> 在这里我们也可以在服务器的运行窗口下查看到，当浏览器通过`127.0.0.1`地址访问时给出的`Request`头信息

```
C:\Users\lienze\Desktop\写书\书中代码\HTTP>python3 HTTP服务端.py
[+] wait for connection...
[+] from: ('127.0.0.1', 58433)
[+] data:
GET / HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:61.0) Gecko/20100101 Firefox/61.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

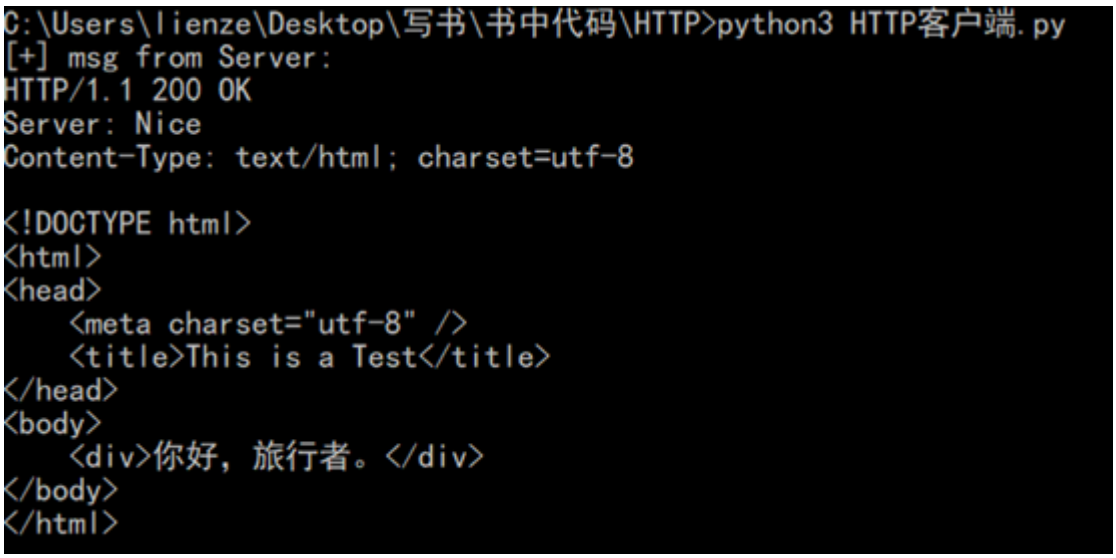
> 当然，我们除了编写简单的服务端代码，我们还可以自己实现一个类似浏览器的客户端
>
> 用来访问对应的`Web`服务，那么需要做的事情，其实也很简单，只需要构建对应的`Request`头部数据即可

```python
#http_client.py
import socket

c = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
msg = "GET / HTTP/1.1\r\n" + \
      "Host: 127.0.0.1\r\n"+ \
      "Accept: text/html\r\n" + \
      "\r\n"

c.connect(('127.0.0.1',80))
c.send(msg.encode())
data = c.recv(1024).decode()
print('[+] msg from Server:\n%s' % data)
```

> 我们可以通过我们自己编写的类似浏览器客户端的代码来访问我们自己编写的浏览器服务端



总结

> 这里我们的示例代码关于`HTTP`协议的`Request`以及`Response`都只是构造了一个简单字符串而已。
>
> 在正式的`Web`服务器实现中，需要我们分析客户端发来的`Request`来对应构造`Response`返回头数据
>
> 而客户端也要解析服务器返回的`Response`数据来给用户友好展示或处理数据