浙江大学

本科实验报告

课程名称：	计算机网络基础
实验名称：	实现一个轻量级的 WEB 服务器
姓　　名：	颜晗
学　　院：	计算机学院
系：	计算机科学与技术
专　　业：	计算机科学与技术
学　　号：	3200105515
指导教师：	杨樾人，张泉方

年　　月　　日

浙江大学实验报告

实验名称：实现一个轻量级的 WEB 服务器　实验类型：编程实验

同组学生：rio 代码可直接用　实验地点：计算机网络实验室

一、 实验目的

深入掌握 HTTP 协议规范，学习如何编写标准的互联网应用服务器。

二、 实验内容

- 服务程序能够正确解析 HTTP 协议，并传回所需的网页文件和图片文件
- 使用标准的浏览器，如 IE、Chrome 或者 Safari，输入服务程序的 URL 后，能够正常显示服务器上的网页文件和图片
- 服务端程序界面不做要求，使用命令行或最简单的窗体即可
- 功能要求如下：

  1．服务程序运行后监听在 80 端口或者指定端口

  2．接受浏览器的 TCP 连接（支持多个浏览器同时连接）

  3．读取浏览器发送的数据，解析 HTTP 请求头部，找到感兴趣的部分

  4．根据 HTTP 头部请求的文件路径，打开并读取服务器磁盘上的文件，以 HTTP 响应格式传回浏览器。要求按照文本、图片文件传送不同的 Content-Type，以便让浏览器能够正常显示。

  5．分别使用单个纯文本、只包含文字的 HTML 文件、包含文字和图片的 HTML 文件进行测试，浏览器均能正常显示。

- 本实验可以在前一个 Socket 编程实验的基础上继续，也可以使用第三方封装好的 TCP 类进行网络数据的收发
- 本实验要求不使用任何封装 HTTP 接口的类库或组件，也不使用任何服务端脚本程序如 JSP、ASPX、PHP 等

三、 主要仪器设备

联网的 PC 机、Wireshark 软件、Visual Studio、gcc 或 Java 集成开发环境。

四、 操作方法与实验步骤

- 阅读 HTTP 协议相关标准文档，详细了解 HTTP 协议标准的细节，有必要的话使用 Wireshark 抓包，研究浏览器和 WEB 服务器之间的交互过程
- 创建一个文档目录，与服务器程序运行路径分开
- 准备一个纯文本文件，命名为 test.txt，存放在 txt 子目录下面
- 准备好一个图片文件，命名为 logo.jpg，放在 img 子目录下面
- 写一个 HTML 文件，命名为 test.html，放在 html 子目录下面，主要内容为：

```
<html>
    <head><title>Test</title></head>
    <body>
        <h1>This is a test</h1>
        <img src="img/logo.jpg">
        <form action="dopost" method="POST">
            Login:<input name="login">
            Pass:<input name="pass">
            <input type="submit" value="login">
        </form>
    </body>
</html>
```

- 将 test.html 复制为 noimg.html，并删除其中包含 img 的这一行。
- 服务端编写步骤（**需要采用多线程模式**）

a) 运行初始化，打开 Socket，监听在指定端口（**请使用学号的后 4 位作为服务器的监听端口**）

b) 主线程是一个循环，主要做的工作是等待客户端连接，如果有客户端连接成功，为该客户端创建处理子线程。该子线程的主要处理步骤是：

1. 不断读取客户端发送过来的字节，并检查其中是否连续出现了 2 个回车换行符，如果未出现，继续接收；如果出现，按照 HTTP 格式解析第 1 行，分离出方法、文件和路径名，其他头部字段根据需要读取。

✧ **如果解析出来的方法是 GET**

2. 根据解析出来的文件和路径名，读取响应的磁盘文件（该路径和服务器程序可能不在同一个目录下，需要转换成绝对路径）。如果文件不存在，第 3 步的响应消息的状态设置为 404，并且跳过第 5 步。
3. 准备好一个足够大的缓冲区，按照 HTTP 响应消息的格式先填入第 1 行(状态码=200),加上回车换行符。然后模仿 Wireshark 抓取的 HTTP 消息，填入必要的几行头部（需要哪些头部，请试验），其中不能缺少的 2 个头部是 Content-Type 和 Content-Length。Content-Type 的值要和文件类型相匹配（请通过抓包确定应该填什么），Content-Length 的值填写文件的字节大小。
4. 在头部行填完后，再填入 2 个回车换行
5. 将文件内容按顺序填入到缓冲区后面部分。

✧ **如果解析出来的方法是 POST**

6. 检查解析出来的文件和路径名，如果不是 dopost，则设置响应消息的

状态为 404，然后跳到第 9 步。如果是 dopost，则设置响应消息的状态为 200，并继续下一步。

7. 读取 2 个回车换行后面的体部内容（长度根据头部的 Content-Length 字段的指示），并提取出登录名（login）和密码（pass）的值。如果登录名是你的学号，密码是学号的后 4 位，则将响应消息设置为登录成功，否则将响应消息设置为登录失败。
8. 将响应消息封装成 html 格式，如

   <html><body>响应消息内容</body></html>
9. 准备好一个足够大的缓冲区，按照 HTTP 响应消息的格式先填入第 1 行（根据前面的情况设置好状态码），加上回车换行符。然后填入必要的几行头部，其中不能缺少的 2 个头部是 Content-Type 和 Content-Length。Content-Type 的值设置为 text/html ,如果状态码=200，则 Content-Length 的值填写响应消息的字节大小，并将响应消息填入缓冲区的后面部分，否则填写为 0。
10. 最后一次性将缓冲区内的字节发送给客户端。
11. 发送完毕后，关闭 socket，退出子线程。

c) 主线程还负责检测退出指令（如用户按退出键或者收到退出信号），检测到后即通知并等待各子线程退出。最后关闭 Socket，主程序退出。

- 编程结束后，将服务器部署在一台机器上（本机也可以）。在服务器上分别放置纯文本文件（.txt）、只包含文字的测试 HTML 文件（将测试 HTML 文件中的包含 img 那一行去掉）、包含文字和图片的测试 HTML 文件（以及图片文件）各一个。
- 确定好各个文件的 URL 地址，然后使用浏览器访问这些 URL 地址，如

http://x.x.x.x:port/dir/a.html，其中 port 是服务器的监听端口，dir 是提供给外部访问的路径，请设置为与文件实际存放路径不同，通过服务器内部映射转换。

- 检查浏览器是否正常显示页面，如果有问题，查找原因，并修改，直至满足要求
- 使用多个浏览器同时访问这些 URL 地址，检查并发性

五、 实验数据记录和处理

请将以下内容和本实验报告一起打包成一个压缩文件上传：

- **源代码：需要说明编译环境和编译方法，如果不能编译成功，将影响评分**
- **可执行文件：可运行的.exe 文件或 Linux 可执行文件**

以下实验记录均需结合屏幕截图（截取源代码或运行结果），进行文字标注（看完请删除本句）。

- 服务器的主线程循环关键代码截图（解释总体处理逻辑，省略细节部分）

主线程通过 while 不断循环，每次接受一个请求，只要连接数未超过最大值，就建立子线程处理该请求并进入下一循环准备接受请求。对于多线程处理依然十分粗糙，直接将子线程进行分离。

```
while (true){
    sockaddr_in client_addr;
    unsigned int client_addr_len = sizeof(struct sockaddr);
    int clientfd;
    if((clientfd = accept(sockfd, (struct sockaddr*)&client_addr,
                          (socklen_t*)& client_addr_len)) == -1) {
        cout<<RED << "[Error] Sockfd accept failure !!!"<<NORMAL <<endl;
        exit(1);
    }
    while (connections > 10){}// 判断连接数

    cout<< GREEN << "Handle request from address "<<
        inet_ntoa(client_addr.sin_addr) << ":" <<
        ntohs(client_addr.sin_port) << NORMAL<<endl;

    pthread_create(&subThread, NULL, handle_request, &clientfd);// 创建子线程
    pthread_detach(subThread);
    connections++; //增加一个处理线程
}
```

- 服务器的客户端处理子线程关键代码截图（解释总体处理逻辑，省略细节部分）

对于每一个客户端处理线程，首先接受发送过来的请求消息，然后解析出请求方法和请求的文件路径，通过 if 对请求方法进行分别处理，发送相应的响应包至客户端。

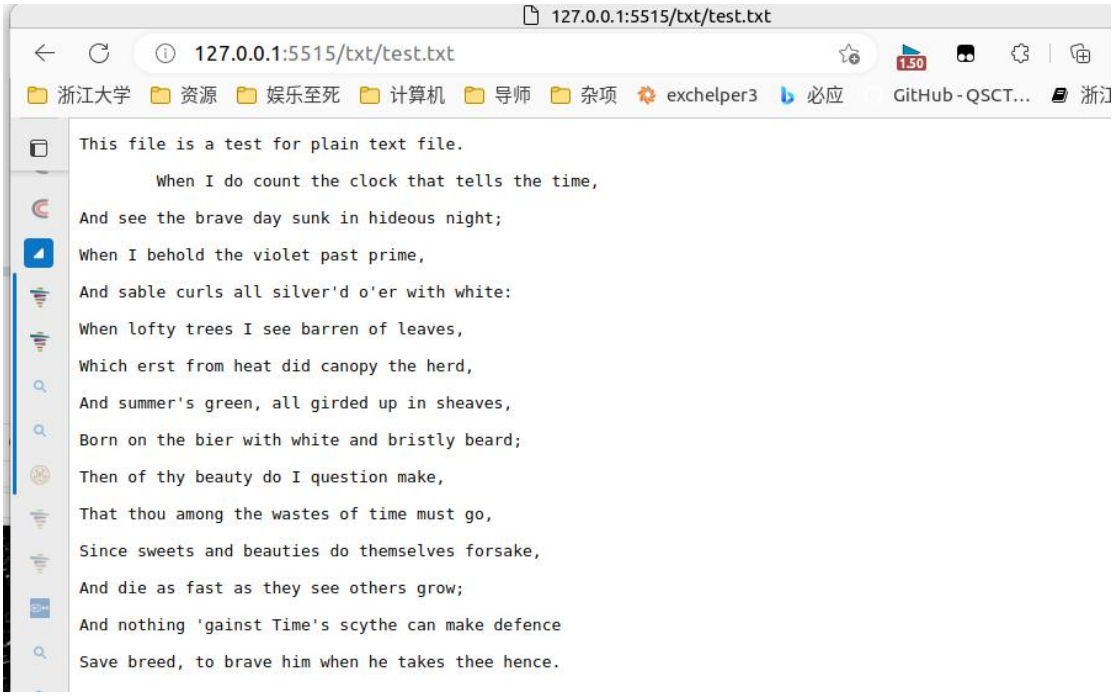
因为本次实验实现的较为简单，所以在解析方法和文件时完成了登陆信息判断等操作以及目标文件路径的获取，所以后续两种方法的处理实际上都是发送对应文件而已。

```
void* handle_request(void* arg0)
{
    int cfd = *(int*)arg0;
    char * rcvBuff = new char[BUFFSIZE];
    char * filePath = new char[BUFFSIZE];
    int rcvLength = recv(cfd, rcvBuff, BUFFSIZE, 0);

    cout << YELLOW << "Connect with socket "<< cfd<<NORMAL<<endl;
    cout << rcvBuff << endl;

    requestMethod method = getMethodandFile(rcvBuff,filePath);

    bool flag=0;
    if(requestMethod::GET == method){
        flag = sendFile(cfd, filePath);
    }
    else if(requestMethod::POST == method){
        flag = sendFile(cfd, filePath);
    }
    else {
    }
    if(!flag) cout<<RED<<"[ERROR] send file failed!!!"<<NORMAL<<endl;


    close(cfd);
    delete [] rcvBuff;
    delete [] filePath;
    cout << YELLOW <<"Disconnect with socket "<< cfd <<NORMAL<<endl<<endl;
    connections--; //减少连接计数
    return NULL;
}
```

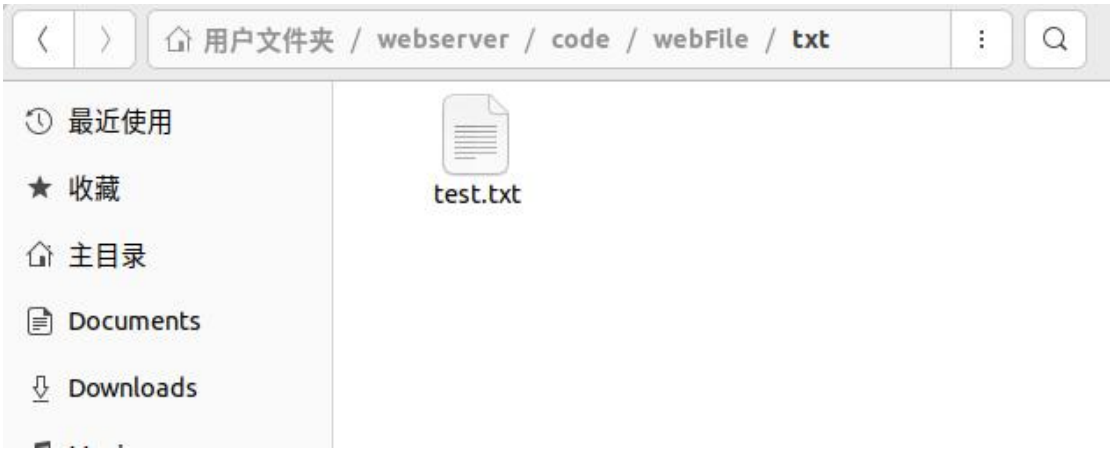
- 服务器运行后，用 netstat –an 显示服务器的监听端口

```
tcp        0      0 0.0.0.0:5515            0.0.0.0:*               LISTEN
```

- 浏览器访问纯文本文件（.txt）时，浏览器的 URL 地址和显示内容截图。



服务器上文件实际存放的路径：



服务器的相关代码片段：

访问文件分为两步，首先需要解析文件路径，然后再通过路径访问到文件并发送。

GET 方法给出的文件路径实际上已经很完整了，只是对于目录路径（例如根目录）

我们还需要主动添加 index.html，另外在路径前添加网站文件目录与程序的相对路径。

```
requestMethod getMethodandFile(char * buf, char* filepath)
{
    char method[10],uri[BUFFSIZE];
    sscanf(buf, "%s %s",method, uri);

    if(!strcmp(method,"GET")){
        strcpy(filepath, "webFile");
        strcat(filepath, uri);
        int urilen = strlen(uri);
        if(uri[urilen-1] == '/'){
            strcat(filepath,"index.html");
        }

        return requestMethod::GET;
    }
    else if(!strcmp(method,"POST")){
```

然后通过 stat 函数确认文件的存在，不存在即返回 404 报错信息，存在则再获取文件的类型和长度等信息。

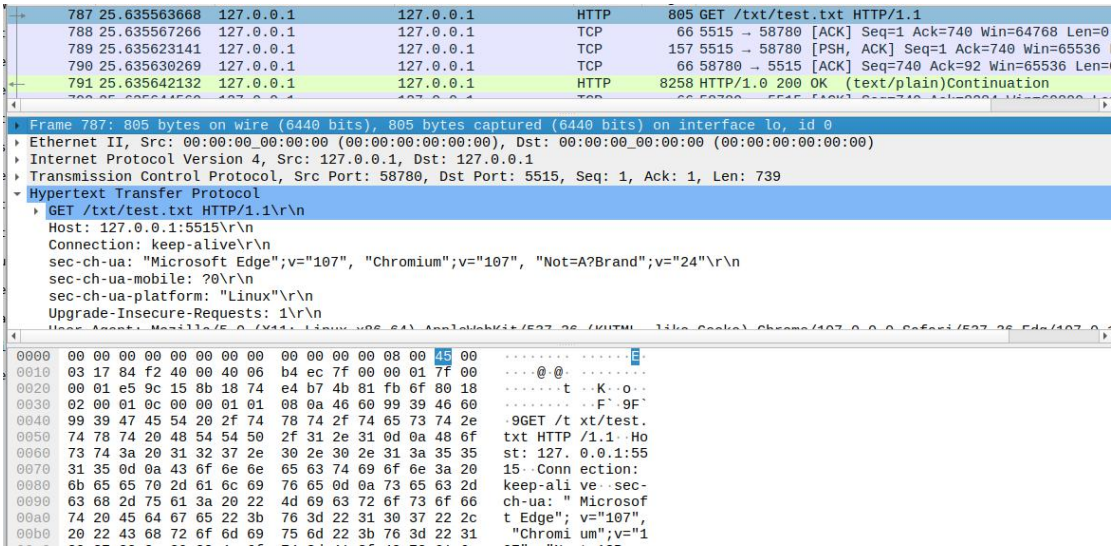
```
if((ll=stat(filepath, &sbuf)) < 0){
    char buf[1024];
    //返回404
    sprintf(buf, "HTTP/1.0 404 NOT FOUND\r\n");
    send(sockfd, buf, strlen(buf), 0);
    sprintf(buf, "Server: Tiny Web Server\r\n");
    send(sockfd, buf, strlen(buf), 0);
    sprintf(buf, "Content-Type: text/html\r\n");
    send(sockfd, buf, strlen(buf), 0);
    sprintf(buf, "\r\n");
    send(sockfd, buf, strlen(buf), 0);
    sprintf(buf, "<HTML><TITLE>Not Found</TITLE>\r\n");
    send(sockfd, buf, strlen(buf), 0);
    sprintf(buf, "<BODY><h1>404 Not Found, the file %s\r\n",filepath);
    send(sockfd, buf, strlen(buf), 0);
    sprintf(buf, "</BODY></HTML>\r\n");
    send(sockfd, buf, strlen(buf), 0);
    return false;
}
```

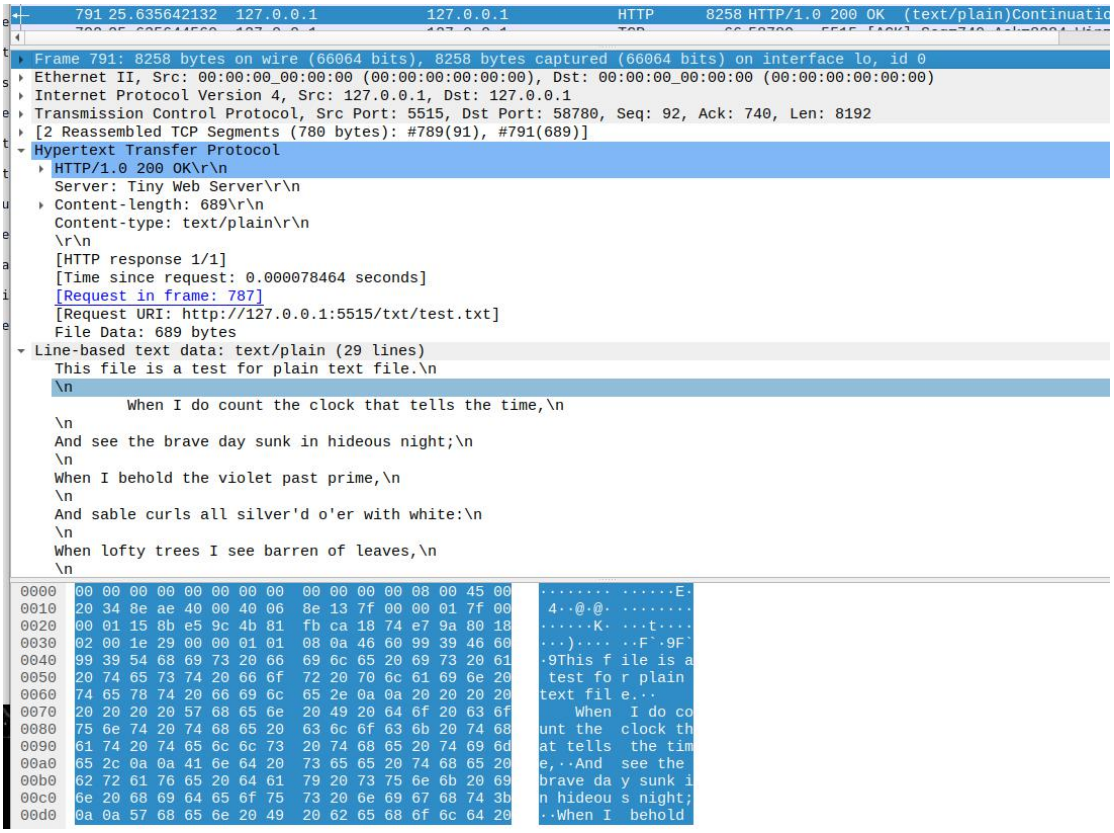
```
int fileLen = sbuf.st_size;
if (strstr(filepath, ".html"))
    strcpy(fileType, "text/html");
else if (strstr(filepath, ".jpg"))
    strcpy(fileType, "image/jpeg");
else
    strcpy(fileType, "text/plain");
```

然后将响应包的头和体分别发送出去即可。

```
sprintf(sendBuf, "HTTP/1.0 200 OK\r\n");
strcat(sendBuf, "Server: Tiny Web Server\r\n");
strcat(sendBuf, "Content-length: ");
strcat(sendBuf,to_string(fileLen).c_str());
strcat(sendBuf,"\r\n");
strcat(sendBuf, "Content-type: ");
strcat(sendBuf, fileType);
strcat(sendBuf,"\r\n\r\n");
send(sockfd, sendBuf, strlen(sendBuf), 0);

int count = 0;
while (!feof(fp))
{
    fread(fileBuf, 1,BUFFSIZE, fp);
    send(sockfd, fileBuf, BUFFSIZE, 0);
    memset(fileBuf,0,BUFFSIZE);
}
```

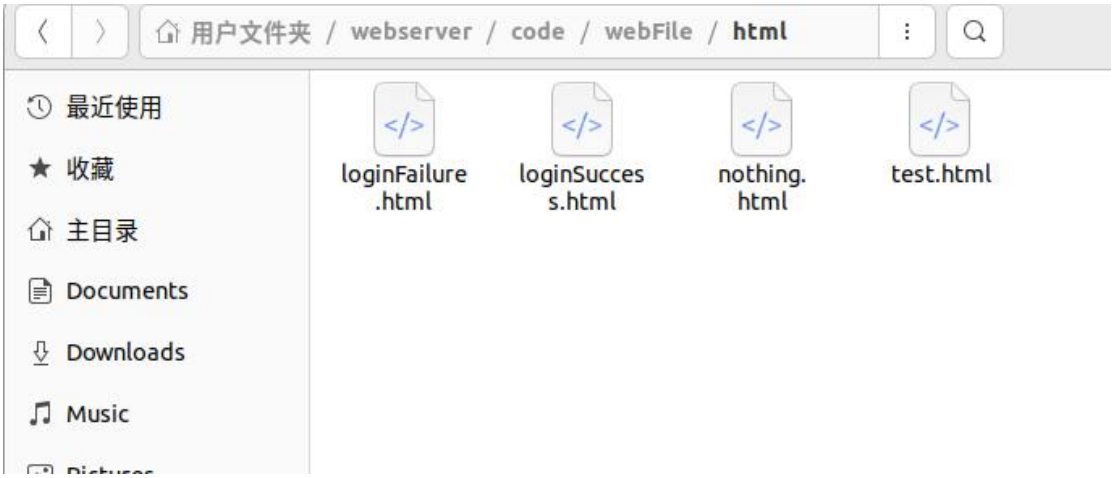
Wireshark 抓取的数据包截图（通过跟踪 TCP 流，只截取 HTTP 协议部分）：



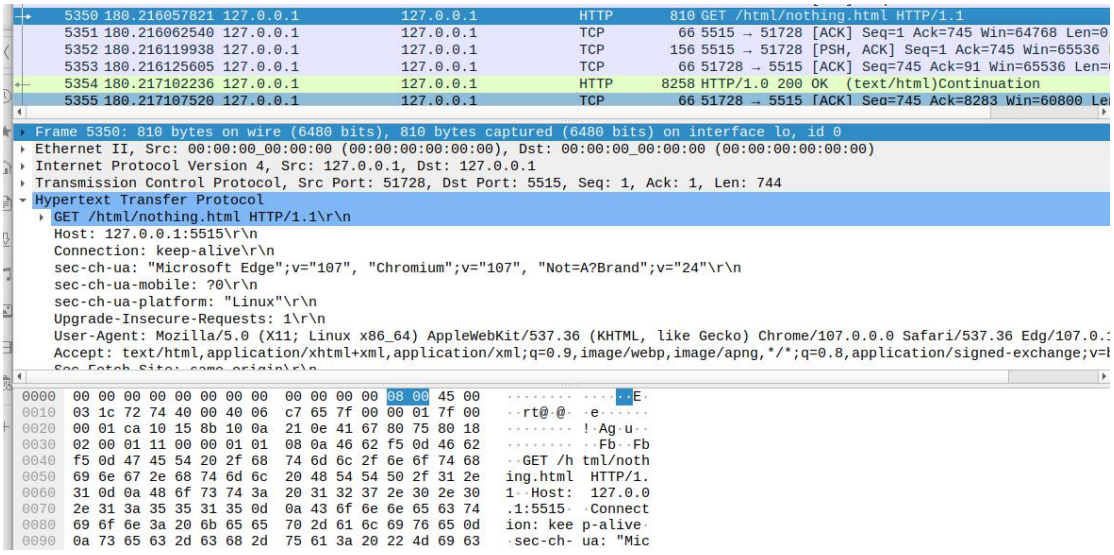


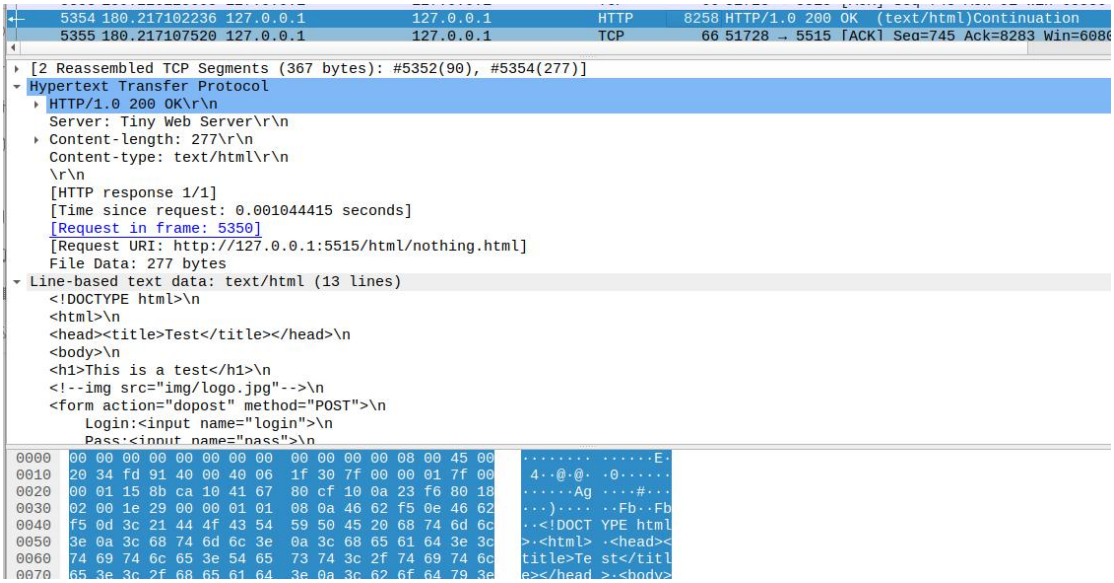
- 浏览器访问只包含文本的 HTML 文件时，浏览器的 URL 地址和显示内容截图。

服务器文件实际存放的路径：

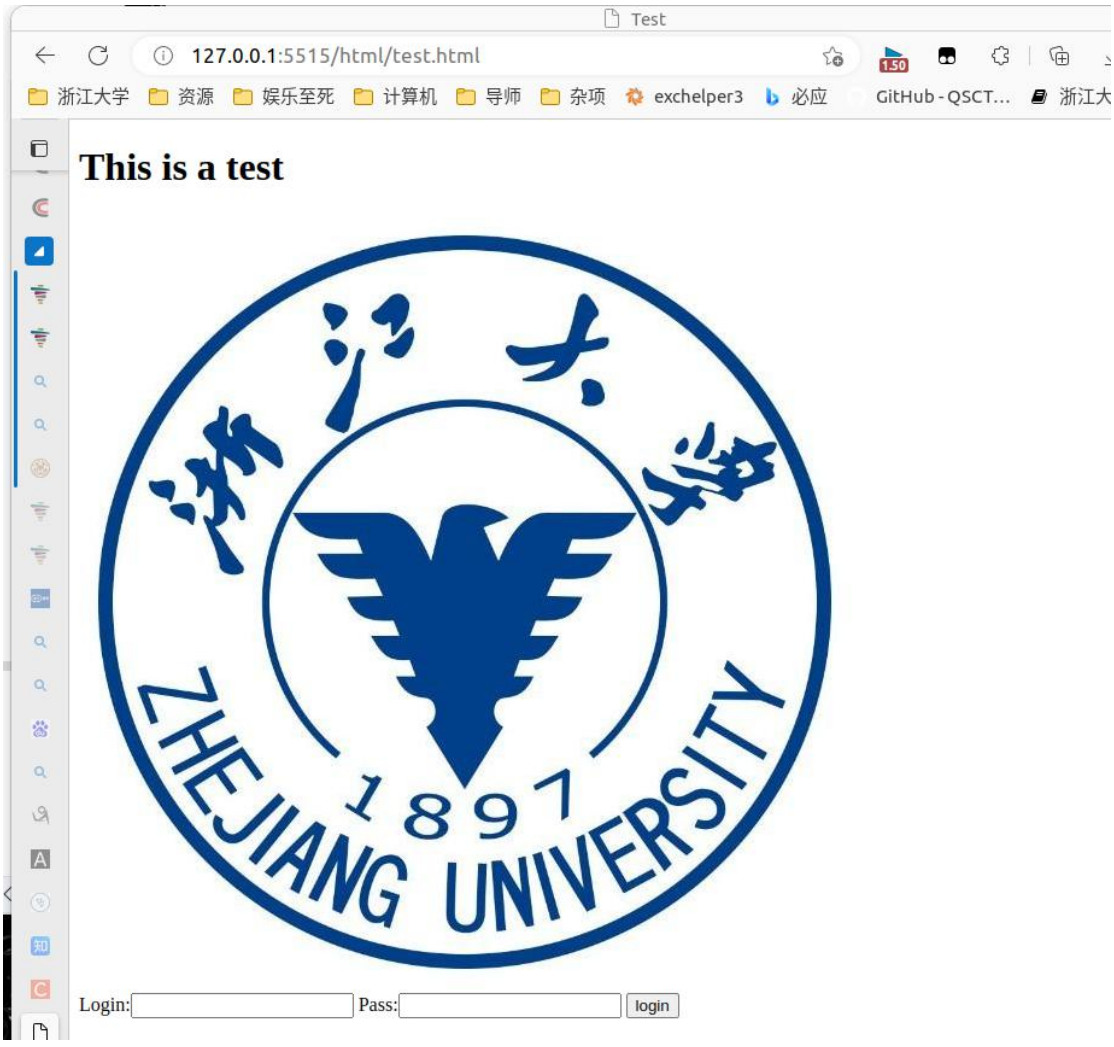


Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML 内容）：

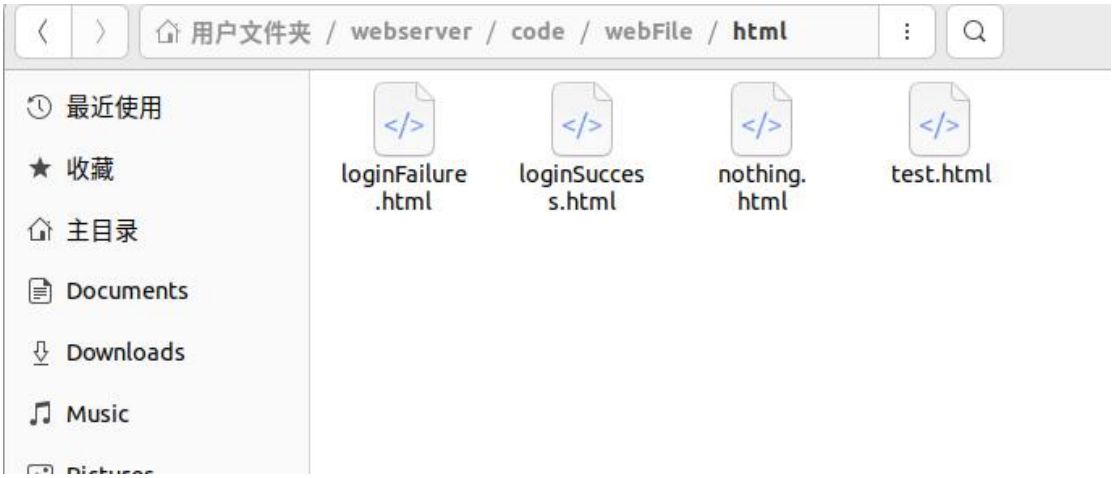




- 浏览器访问包含文本、图片的 HTML 文件时 ,浏览器的 URL 地址和显示内容截图。

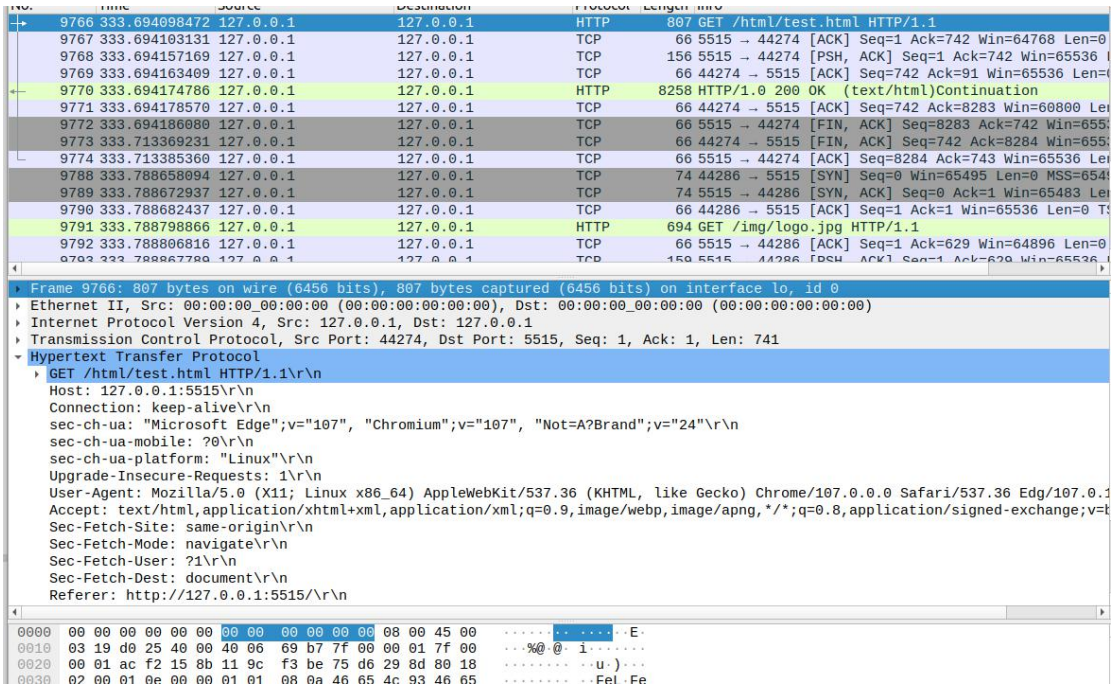


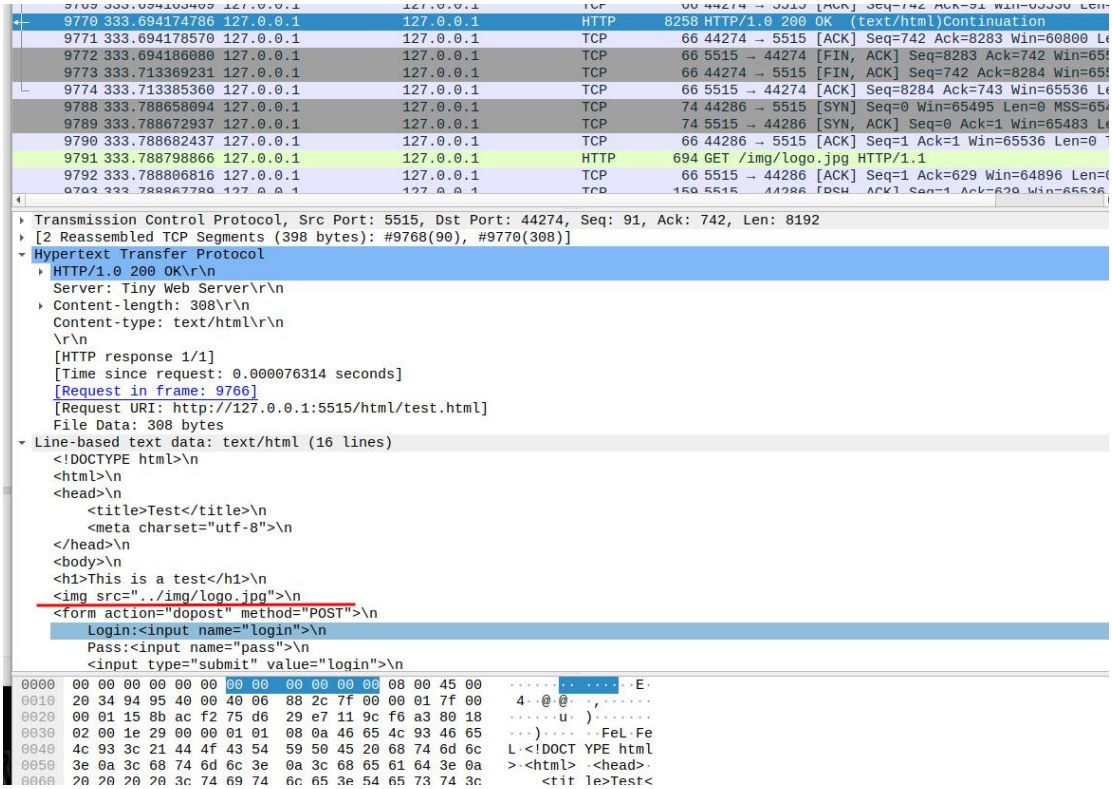
服务器上文件实际存放的路径：



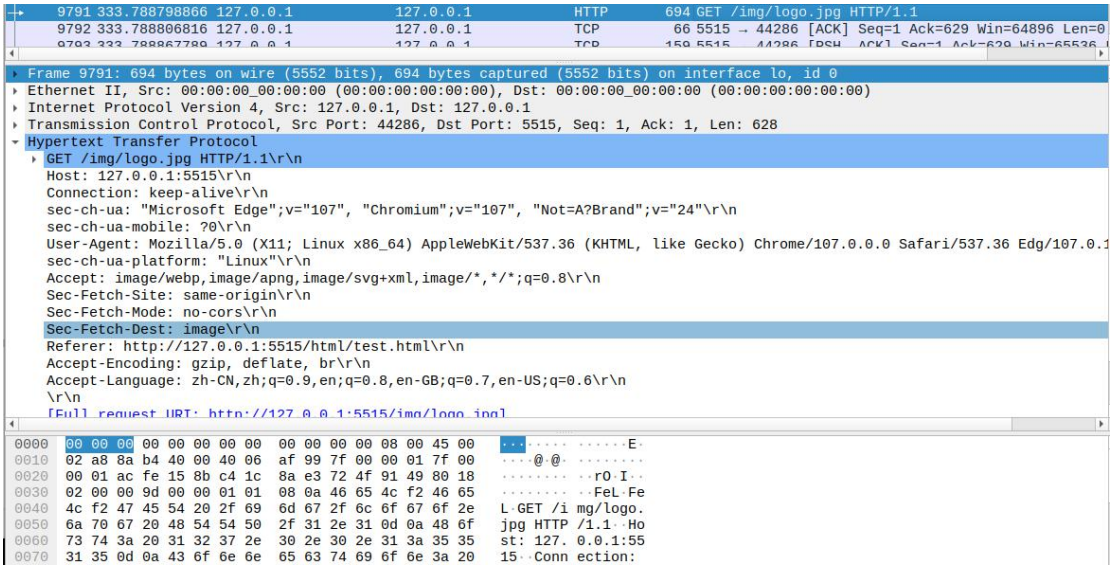
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：

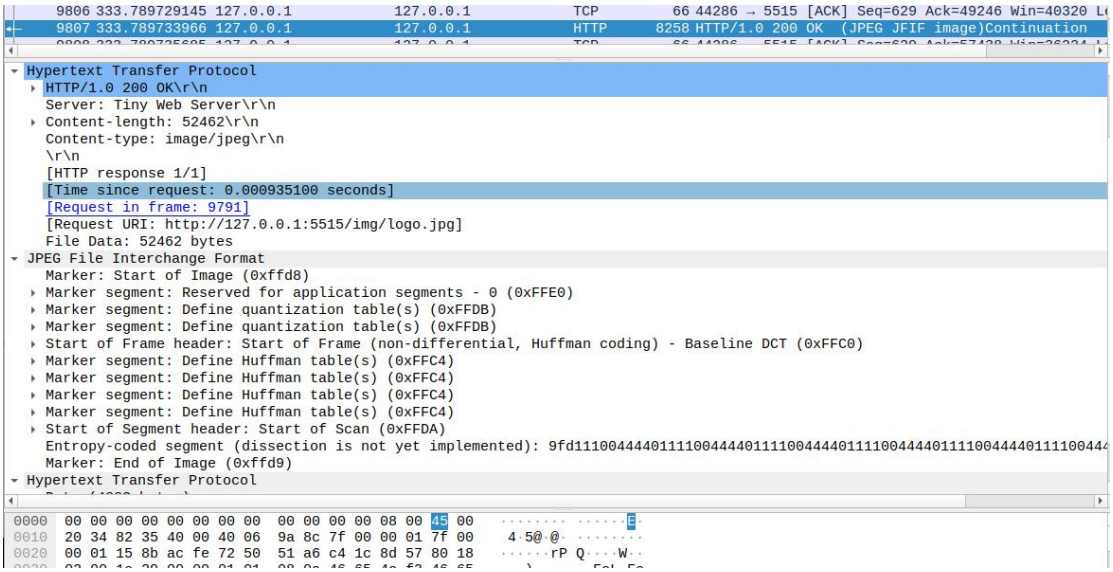
html 文件请求和响应包。




9770 333.694174786 127.0.0.1 127.0.0.1 HTTP 8258 HTTP/1.0 200 OK (text/html)Continuation
9771 333.694178570 127.0.0.1 127.0.0.1 TCP 66 44274 → 5515 [ACK] Seq=742 Ack=8283 Win=60800
9772 333.694186080 127.0.0.1 127.0.0.1 TCP 66 5515 → 44274 [FIN, ACK] Seq=8283 Ack=742 Win=65
9773 333.713369231 127.0.0.1 127.0.0.1 TCP 66 44274 → 5515 [FIN, ACK] Seq=742 Ack=8284 Win=65
9774 333.713385360 127.0.0.1 127.0.0.1 TCP 66 5515 → 44274 [ACK] Seq=8284 Ack=743 Win=65536
9788 333.788658094 127.0.0.1 127.0.0.1 TCP 74 44286 → 5515 [SYN] Seq=0 Win=65495 Len=0 MSS=65
9789 333.788672937 127.0.0.1 127.0.0.1 TCP 74 5515 → 44286 [SYN, ACK] Seq=0 Ack=1 Win=65483
9790 333.788682437 127.0.0.1 127.0.0.1 TCP 66 44286 → 5515 [ACK] Seq=1 Ack=1 Win=65536 Len=0
9791 333.788798866 127.0.0.1 127.0.0.1 HTTP 694 GET /img/logo.jpg HTTP/1.1
9792 333.788806816 127.0.0.1 127.0.0.1 TCP 66 5515 → 44286 [ACK] Seq=1 Ack=629 Win=64896 Len=0
Transmission Control Protocol, Src Port: 5515, Dst Port: 44274, Seq: 91, Ack: 742, Len: 8192
[2 Reassembled TCP Segments (398 bytes): #9768(90), #9770(308)]
Hypertext Transfer Protocol
HTTP/1.0 200 OK\r\n
Server: Tiny Web Server\r\n
Content-length: 308\r\n
Content-type: text/html\r\n
\r\n
[HTTP response 1/1]
[Time since request: 0.000076314 seconds]
[Request in frame: 9766]
[Request URI: http://127.0.0.1:5515/html/test.html]
File Data: 308 bytes
Line-based text data: text/html (16 lines)
<!DOCTYPE html>\n
<html>\n
<head>\n
<title>Test</title>\n
<meta charset="utf-8">\n
</head>\n
<body>\n
<h1>This is a test</h1>\n
<img src="../img/logo.jpg">\n
<form action="dopost" method="POST">\n
Login:<input name="login">\n
Pass:<input name="pass">\n
<input type="submit" value="login">\n
0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00
0010 20 34 94 95 40 00 40 06 88 2c 7f 00 00 01 7f 00
0020 00 01 15 8b ac f2 75 d6 29 e7 11 9c f6 a3 80 18
0030 02 00 1e 29 00 00 01 01 08 0a 46 65 4c 93 46 65
0040 4c 93 3c 21 44 4f 43 54 59 50 45 20 68 74 6d 6c
0050 3e 0a 3c 68 74 6d 6c 3e 0a 3c 68 65 61 64 3e 0a

图片文件请求和响应。


9791 333.788798866 127.0.0.1 127.0.0.1 HTTP 694 GET /img/logo.jpg HTTP/1.1
9792 333.788806816 127.0.0.1 127.0.0.1 TCP 66 5515 → 44286 [ACK] Seq=1 Ack=629 Win=64896 Len=0
Frame 9791: 694 bytes on wire (5552 bits), 694 bytes captured (5552 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 44286, Dst Port: 5515, Seq: 1, Ack: 1, Len: 628
Hypertext Transfer Protocol
GET /img/logo.jpg HTTP/1.1\r\n
Host: 127.0.0.1:5515\r\n
Connection: keep-alive\r\n
sec-ch-ua: "Microsoft Edge";v="107", "Chromium";v="107", "Not=A?Brand";v="24"\r\n
sec-ch-ua-mobile: ?0\r\n
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0 Safari/537.36 Edg/107.0.1
sec-ch-ua-platform: "Linux"\r\n
Accept: image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8\r\n
Sec-Fetch-Site: same-origin\r\n
Sec-Fetch-Mode: no-cors\r\n
Sec-Fetch-Dest: image\r\n
Referer: http://127.0.0.1:5515/html/test.html\r\n
Accept-Encoding: gzip, deflate, br\r\n
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6\r\n
\r\n
[Full request URI: http://127.0.0.1:5515/img/logo.jpg]
0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00
0010 02 a8 8a b4 40 00 40 06 af 99 7f 00 00 01 7f 00
0020 00 01 ac fe 15 8b c4 1c 8a e3 72 4f 91 49 80 18
0030 02 00 00 9d 00 00 01 01 08 0a 46 65 4c f2 46 65
0040 4c f2 47 45 54 20 2f 69 6d 67 2f 6c 6f 67 6f 2e
0050 6a 70 67 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f
0060 73 74 3a 20 31 32 37 2e 30 2e 30 2e 31 3a 35 35
0070 31 35 0d 0a 43 6f 6e 6e 65 63 74 69 6f 6e 3a 20



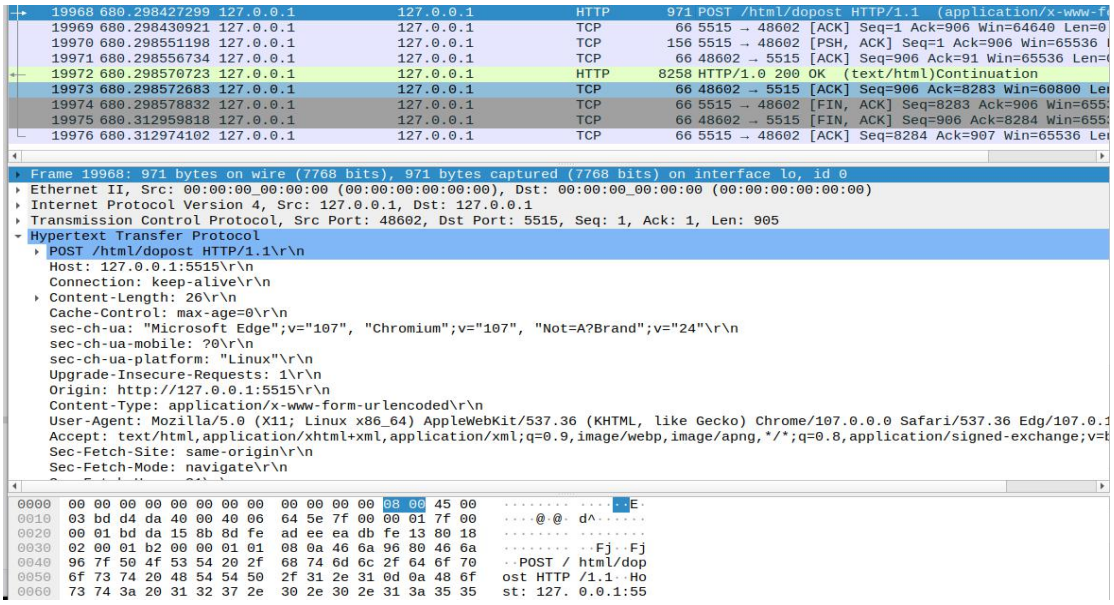
- 浏览器输入正确的登录名或密码，点击登录按钮（login）后的显示截图。

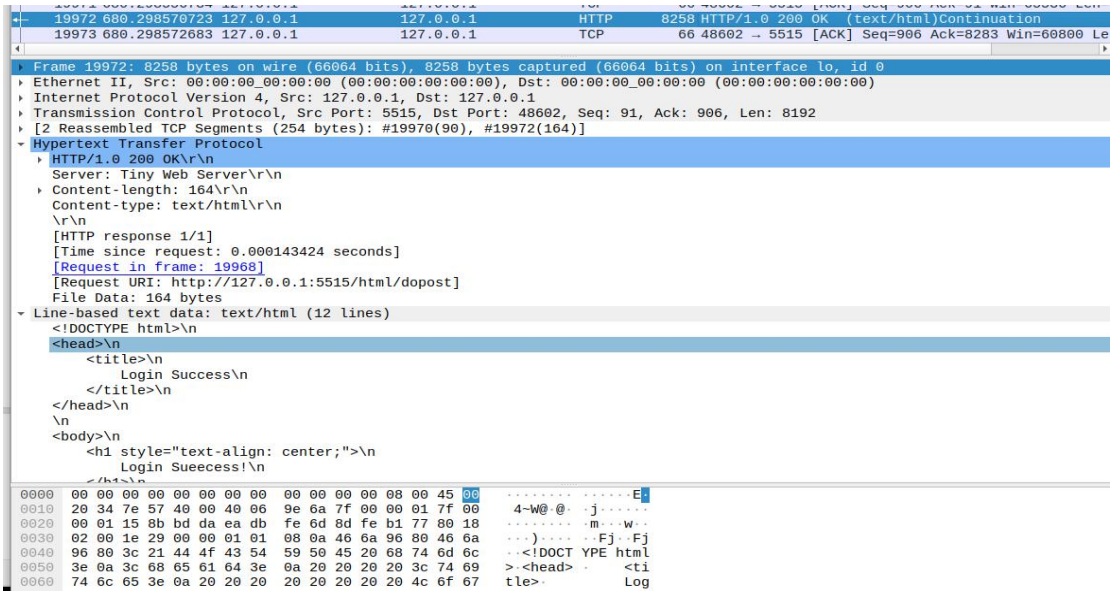
服务器相关处理代码片段：

对于服务器的 POST 请求包，通过循环读取缓冲信息，直到读取到一个单独换行符，我们就进入了体部，通过 c++字符串方法分割读取客户端发送的账号和密码并检查，返回匹配结果。

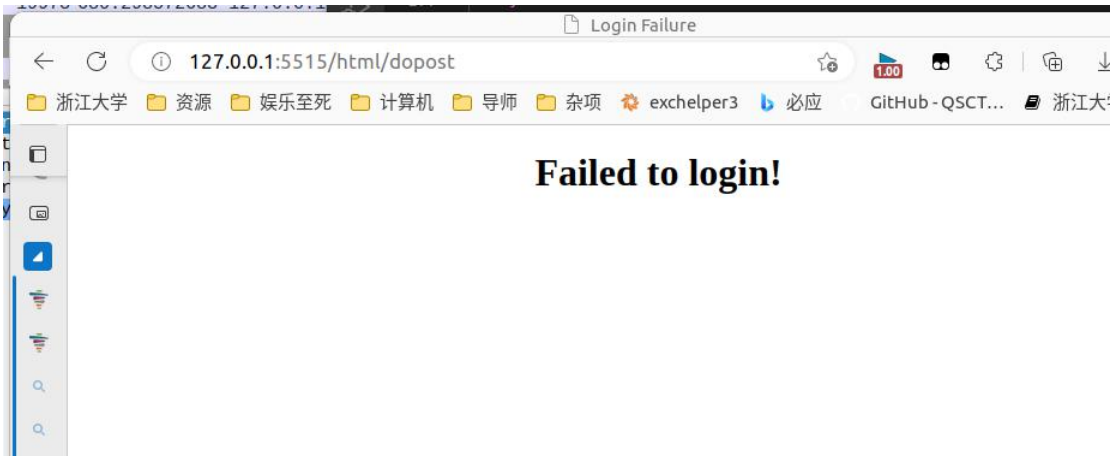
```
bool checkAccount(char *buf)
{
    stringstream in(buf);
    string blank;
    while(1){
        getline(in, blank);
        if(blank[0] == '\r'){
            break;
        }
    }
    getline(in, blank);
    int pos,mid,spos;
    pos = blank.find_first_of("=");
    spos = blank.find("=", pos+1);
    mid = blank.find("&");
    Account account;
    account.login = blank.substr(pos+1, mid-pos-1);
    account.pass = blank.substr(spos+1,blank.length()-spos-1);

    return ((account.login == "3200105515")&&(account.pass == "5515"));

}
```

Wireshark 抓取的数据包截图（HTTP 协议部分）

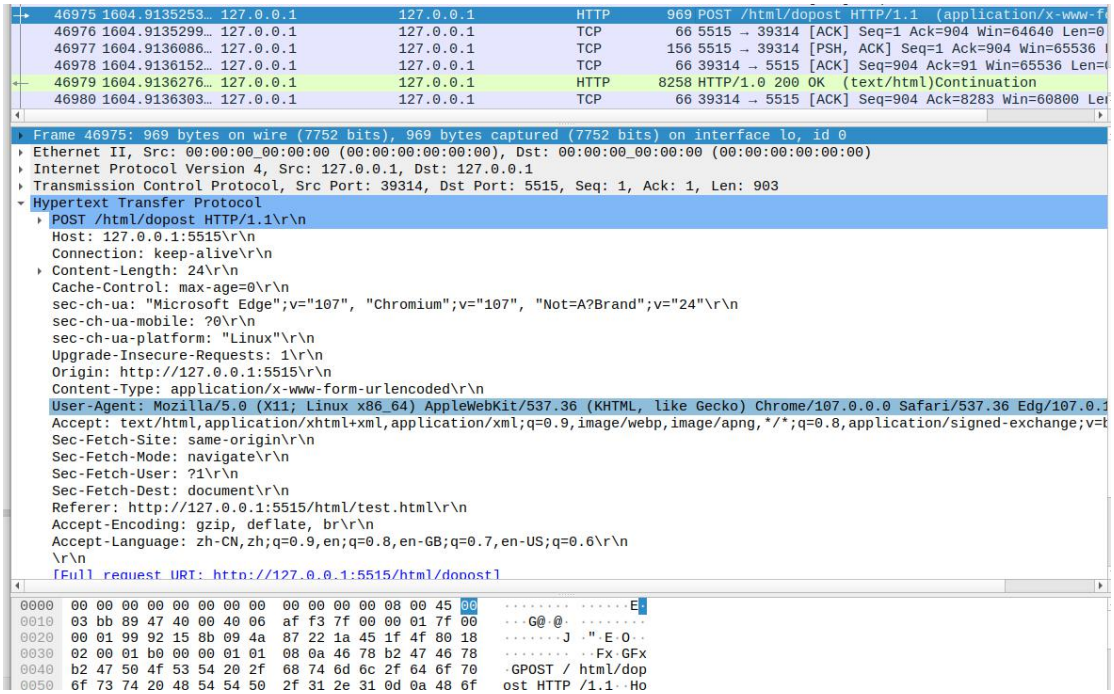


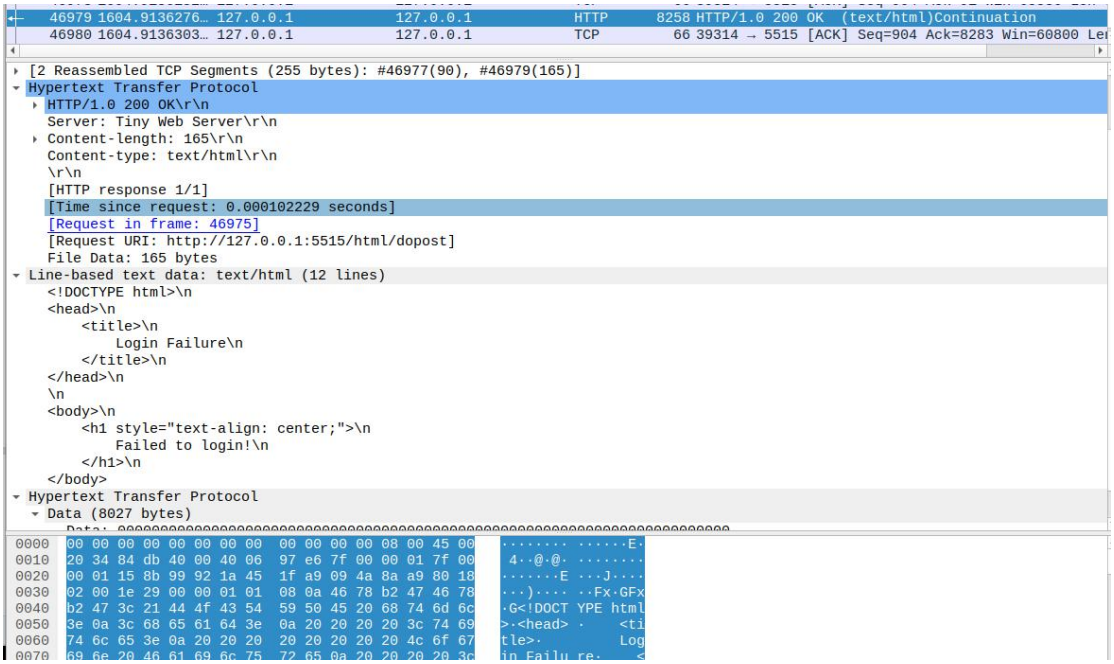


- 浏览器输入错误的登录名或密码，点击登录按钮（login）后的显示截图。

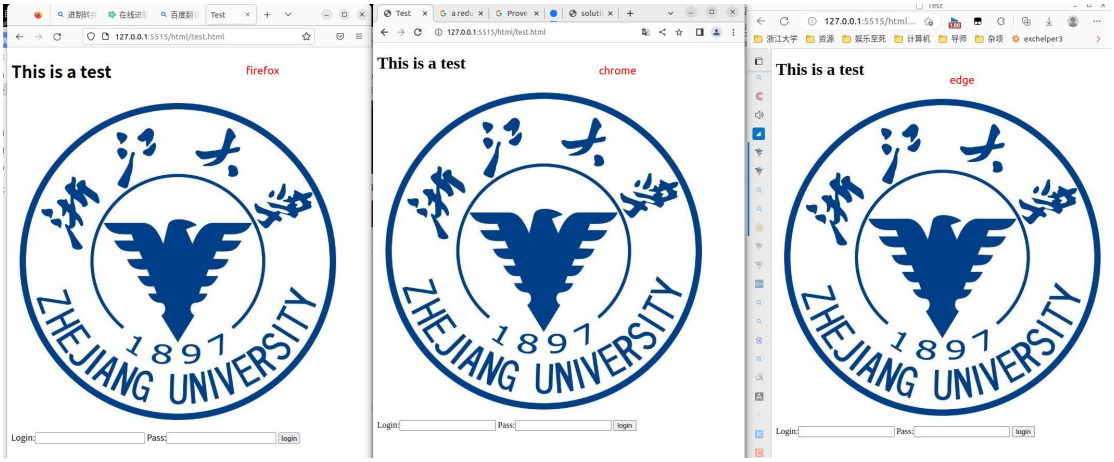


- Wireshark 抓取的数据包截图（HTTP 协议部分）

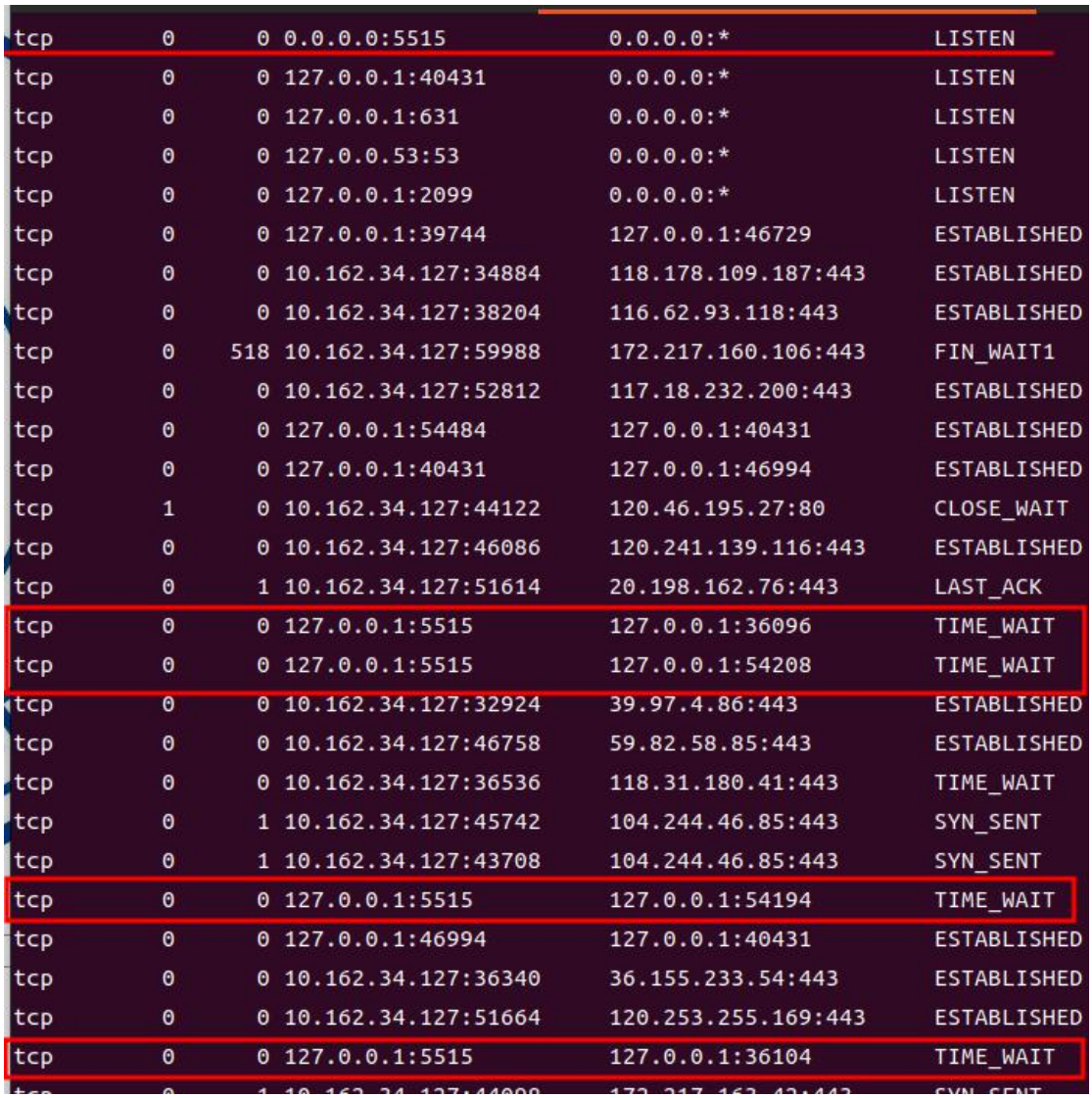




- 多个浏览器同时访问包含图片的 HTML 文件时，浏览器的显示内容截图（将浏览器窗口缩小并列）



- 多个浏览器同时访问包含图片的 HTML 文件时 ,使用 netstat –an 显示服务器的 TCP 连接（截取与服务器监听端口相关的）



tcp	0	0	0.0.0.0:5515	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:40431	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:631	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.53:53	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:2099	0.0.0.0:*	LISTEN
tcp	0	0	127.0.0.1:39744	127.0.0.1:46729	ESTABLISHED
tcp	0	0	10.162.34.127:34884	118.178.109.187:443	ESTABLISHED
tcp	0	0	10.162.34.127:38204	116.62.93.118:443	ESTABLISHED
tcp	0	518	10.162.34.127:59988	172.217.160.106:443	FIN_WAIT1
tcp	0	0	10.162.34.127:52812	117.18.232.200:443	ESTABLISHED
tcp	0	0	127.0.0.1:54484	127.0.0.1:40431	ESTABLISHED
tcp	0	0	127.0.0.1:40431	127.0.0.1:46994	ESTABLISHED
tcp	1	0	10.162.34.127:44122	120.46.195.27:80	CLOSE_WAIT
tcp	0	0	10.162.34.127:46086	120.241.139.116:443	ESTABLISHED
tcp	0	1	10.162.34.127:51614	20.198.162.76:443	LAST_ACK
tcp	0	0	127.0.0.1:5515	127.0.0.1:36096	TIME_WAIT
tcp	0	0	127.0.0.1:5515	127.0.0.1:54208	TIME_WAIT
tcp	0	0	10.162.34.127:32924	39.97.4.86:443	ESTABLISHED
tcp	0	0	10.162.34.127:46758	59.82.58.85:443	ESTABLISHED
tcp	0	0	10.162.34.127:36536	118.31.180.41:443	TIME_WAIT
tcp	0	1	10.162.34.127:45742	104.244.46.85:443	SYN_SENT
tcp	0	1	10.162.34.127:43708	104.244.46.85:443	SYN_SENT
tcp	0	0	127.0.0.1:5515	127.0.0.1:54194	TIME_WAIT
tcp	0	0	127.0.0.1:46994	127.0.0.1:40431	ESTABLISHED
tcp	0	0	10.162.34.127:36340	36.155.233.54:443	ESTABLISHED
tcp	0	0	10.162.34.127:51664	120.253.255.169:443	ESTABLISHED
tcp	0	0	127.0.0.1:5515	127.0.0.1:36104	TIME_WAIT

六、 实验结果与分析

根据你编写的程序运行效果，分别解答以下问题（看完请删除本句）：

- HTTP 协议是怎样对头部和体部进行分隔的？

  答：通过在头部和体部之间增加一个空行进行分隔。

- 浏览器是根据文件的扩展名还是根据头部的哪个字段判断文件类型的？

  答：浏览器通过头部的 Content-Type 字段判断文件类型。

- HTTP 协议的头部是不是一定是文本格式？体部呢？

答：http 协议的头部是文本格式，而体部除了文本格式还可以是音频和图片等格式的字节流。

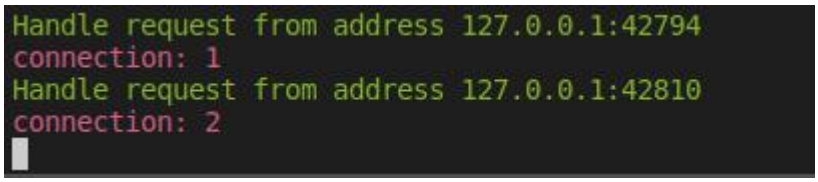
- POST 方法传递的数据是放在头部还是体部？两个字段是用什么符号连接起来的？

答：放在体部，两个字段之间通过&符号连接。

七、 讨论、心得

本次实验实现的 web 服务器功能比较简单，在经过了上一次 socket 实验后，本次实验更加不是问题。但是由于并不熟悉 socket 的底层原理和 http 协议的细节，实际上依然存在一些问题。

1. 对于 Chrome 浏览器，相比其他浏览器总是有莫名的连接存在，进入主页(index.html)时总是会产生两个连接，在获取其他资源时这两个线程才处理并结束，在新开的窗口访问主页还会产生对/favicon.ico 的访问，即使主页实际并未需要这个资源。即虽然理论上已经建立了连接并且创建了线程，但是接收不到请求信息，只有重新操作浏览器后这些线程才能接收信息，未知原因，尚未解决。



2. 图片发送错误。

图片和文本属于不同的格式，开头尚未意识到这一点时，通过字符串的操作方法对图片进行读取和发送，导致图片发送失败和不完整。