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：	李向
学　　院：	计算机学院
系：	计算机科学与工程学系
专　　业：	计算机科学与技术
学　　号：	3180106148
指导教师：	黄正谦

2021 年　1 月　11 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：无　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**

- 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）

```
//主线程循环调用accept()，阻塞等待接受客户端连接
while (1) {
    sServer = accept(sListen, (struct sockaddr*)&saClient, &addrlen);
    //如果接收到的socket句柄无效，则继续循环
    if (sServer == INVALID_SOCKET)
        continue;
    //如果接收到有效的socket句柄，则在客户端列表中增加一个新客户端的项目，并记录下该客户端句柄和连接状态、端口，然后创建一个子线程后继续循环。
    else {
        //客户端列表中的元素个数加一
        Clientnum++;
        //记录下该客户端socket句柄、编号、ip地址、端口和连接状态，保存到tempinfo中
        tempinfo.Cli_socket = sServer;
        tempinfo.Cli_id = Clientnum;
        tempinfo.Cli_ip = inet_ntoa(saClient.sin_addr);
        tempinfo.Cli_port = ntohs(saClient.sin_port);
        tempinfo.Cli_state = UP;

        //将刚刚的记录加入Clientlist中
        Clientlist.push_back(tempinfo);

        //服务端输出相应的提示信息
        cout << endl;
        cout << "*********************************************************************" << endl;
        cout << "接受了" << tempinfo.Cli_id << "号客户端的连接请求！该客户端信息如下：" << endl;
        cout << "SOCKET: " << tempinfo.Cli_socket << endl;
        cout << "IP: " << tempinfo.Cli_ip << endl;
        cout << "PORT: " << tempinfo.Cli_port << endl;
        cout << "*********************************************************************\n" << endl;
```

```
    //服务端输出相应的提示信息
    cout << endl;
    cout << "*****************************************************************" << endl;
    cout << "接受了" << tempinfo.Cli_id << "号客户端的连接请求！该客户端信息如下：" << endl;
    cout << "SOCKET：" << tempinfo.Cli_socket << endl;
    cout << "IP：" << tempinfo.Cli_ip << endl;
    cout << "PORT：" << tempinfo.Cli_port << endl;
    cout << "*****************************************************************\n" << endl;

    //创建一个子线程处理后续与该客户端的通信
    thread Childthread(Interact_with_Client, tempinfo);
    Childthread.detach();
}


{
    lock_guard<mutex> lk(Lock_for_shouldquit);
    if (shouldquit)
        break;
}
```

服务端通过一个 while(1)循环来不断监听客户端的连接请求，首先调用 accept()来接受客户端连接，如果接收到的 socket 句柄无效，则继续循环；如果接收到有效的 socket 句柄，则在客户端列表中增加一个新客户端的项目，并记录下该客户端句柄和连接状态、端口，再输出刚刚连接的客户端的相应信息，之后创建一个子线程处理后续与该客户端的通信，最后在互斥锁的保护下检查当前是否应该退出服务端程序。

- 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）

```
//子线程执行的函数，用于与已经连接的客户端的后续交互
void Interact_with_Client(Client_info curclient) {
    char* request = new char[BUFFERSIZE];       //接收缓冲区
    char* respond = new char[BUFFERSIZE];       //发送缓冲区

    SOCKET now_socket = curclient.Cli_socket;
    int now_id = curclient.Cli_id;
    fstream file;           //文件标志符
    int enter;
    string url, root_path, method, version;
    root_path = "D:/Webserver_root";            //Webserver保存文件的根目录

    int ret = recv(now_socket, request, BUFFERSIZE, 0);
    while(1) {
        if (ret == SOCKET_ERROR)//接收失败
        {
            cout << "##### Error: recv()函数执行出错！#####" << endl;
            lock_guard<mutex> lk(Lock_for_Clientlist);
            Clientlist[now_id - 1].Cli_state = DOWN;
            closesocket(now_socket);//关闭套接字
            break;
        }
        else {
```

服务端的客户端处理子线程先创建了合适大小的请求缓冲区和响应缓冲区，定义了文件的根路径，接着循环调用 recv()函数接收从客户端发来的请求数据包，当接收失败时输出错误信息并跳出循环。

```
else {
    string HTTP_request(request);
    stringstream header;
    header.str(HTTP_request);
    //查找CRLF（回车换行）
    if (HTTP_request.find("\r\n") != HTTP_request.npos) {
        //查找是否是GET方法
        if (HTTP_request.find("GET") != HTTP_request.npos) {
            //解析HTTP请求头
            header >> method >> url >> version;
            //分析url
            if (url == "/QUIT") {

                cout << "##### 客户端发送了退出信号！#####" << endl;
                {
                    lock_guard<mutex> lk(Lock_for_Clientlist);
                    Clientlist[now_id - 1].Cli_state = DOWN;
                }
                closesocket(now_socket);//关闭套接字
                {
                    lock_guard<mutex> lk(Lock_for_shouldquit);
                    shouldquit = 1;
                }
                break;
            }
            string file_path;
            file_path = root_path + url;
            cout << "##### "<< now_id << "号客户端发来GET请求！#####" << endl;
            cout << "##### FILEPATH:" << file_path << " #####" << endl;
```

当接收成功则在接收到的数据包中查找是否有 CRLF，如果找到，再分别查找是否有“GET”或者“POST”，如果找到则分情况处理。其中对于方法为 GET 的处理中比较特殊的是，当解析到的 URL 为“/QUIT”的时候，说明客户端发出信号想要关闭服务器，此时服务端要输出相应的提示信息，通知并等待各子线程退出（将 shouldquit 的值置为 1）。最后关闭 Socket，主程序退出。

```
//打开文件
file.open(file_path, std::ios::binary | std::ios::in);

//如果请求的文件不存在
if (!file)
{
    cout << "##### ERROR: No such file existed: " + file_path + "! #####" << endl;
    string error_info;  //存储响应消息
    error_info = "HTTP/1.1 404 Not Found\r\nContent-Type: text/html;charset=utf-8\r\nContent-Length: 84\r\n\r\n";
    error_info += "<html><body><h1>404 Not Found</h1><p>From server: URL is wrong.</p></body></html>\r\n";
    strcpy(respond, error_info.c_str());
    send(now_socket, respond, BUFFERSIZE, 0);
    closesocket(now_socket);
    break;
}
//请求的文件存在
else {
    string response_header;     //保存HTML响应头
    response_header = "HTTP/1.1 200 OK\r\nContent-Type: ";

    long left, right;
    left = file.tellg();                    //获取当前读指针位置保存到left
    file.seekg(0, ios::end);                //定位读指针位置为文件结尾
    right = file.tellg();                   //获取当前读指针位置保存到right
    char* content = new char[right - left];
    file.seekg(0, ios::beg);                //重新定位读指针位置为文件开头
    file.read(content, right - left);       //读取文件内容
    file.close();                           //关闭文件

    string type;
```

```
string type;
if (url.find(".txt") != url.npos) {          //文本文件
    type = "text/plain;charset=utf-8\r\n";
    response_header += type;
}
else if (url.find(".html") != url.npos) {    //HTML文件
    type = "text/html;charset=utf-8\r\n";
    response_header += type;
}
else if (url.find(".jpg") != url.npos) {     //jpg图片文件
    type = "image/jpeg\r\n";
    response_header += type;
}
else {
    type = "none";
}

if (type != "none") {
    response_header += "Content-Length: ";
    ostringstream ss;
    ss << right - left;
    string length = ss.str();
    response_header += length;
    response_header += "\r\n\r\n";                    //两个回车换行表示头部结束
    int size = response_header.length();
    memcpy(respond, response_header.c_str(), size);//头部信息
    memcpy(respond + size, content, right - left);//文件信息

    send(now_socket, respond, BUFFERSIZE, 0);
    closesocket(now_socket);
```

对于方法为 GET 的处理，首先根据请求数据包中的 URL 字段与服务端存储文件的根路径组合得到文件的完整路径。然后根据这个路径尝试打开相应的文件，如果打开失败，则首先输出错误信息，接着组装响应数据包，响应消息的状态设置为 404，其他字段也相应设置好，最后调用 send()函数将响应数据包发给客户端，并关闭与该客户端的 socket 连接；如果成功打开，则响应头的状态码设置为 200，之后根据 URL 中的内容判断文件的类型并装入响应头（Content-Type），接着将文件的字节长度也装入响应头（Content-Length），在头部行填完后，再填入 2 个回车换行，最后组装好头部内容和文件内容，调用 send()将整个数据包发回客户端。

```
//查找是否是POST方法
if (HTTP_request.find("POST") != HTTP_request.npos) {
    string response_body;
    string response_head;
    header >> method >> url >> version;
    //输出提示信息
    cout << "##### " << now_id << "号客户端发来POST请求！#####" << endl;
    cout << "##### URL:" << url << " #####" << endl;

    //如果路径名不是dopost
    if (url != "/dopost") {
        cout << "##### ERROR: 路径名不是dopost！ #####" << endl;
        response_head = "HTTP/1.1 404 Not Found\r\nContent-Type: text/html;charset=utf-8\r\nContent-Length: 0\r\n\r\n";
        memcpy(respond, response_head.c_str(), 85);
        send(now_socket, respond, BUFFERSIZE, 0);
        closesocket(now_socket);
        break;
    }

    //如果路径名是dopost
    else {
        int cont_len = HTTP_request.find("Content-Length:");
        string tmp;
        header.clear();
        header.str(HTTP_request.substr(cont_len));
        header >> tmp >> cont_len;                //抽取content-length

        int start = HTTP_request.find("\r\n\r\n");
```

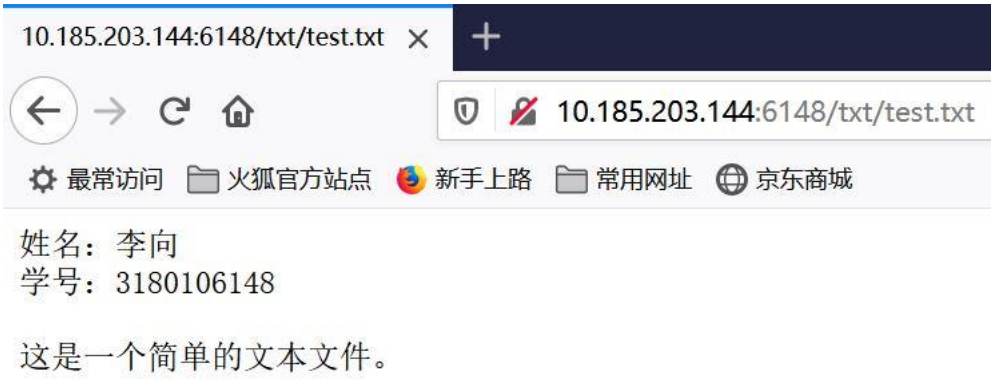
```
if (start != HTTP_request.npos) {
    string body;
    body = HTTP_request.substr(start + 4, cont_len);    //获取体部

    int pos1 = body.find("login=");
    int pos2 = body.find("&pass=");
    if (pos1 != body.npos && pos2 != body.npos && pos1 < pos2) {
        string login = body.substr(pos1 + 6, pos2 - pos1 - 6);
        string pswd = body.substr(pos2 + 6);

        if (login == LOGIN && pswd == PASS) {           //登录成功
            response_body = "<html><body>Login Success, Congratulations!</body></html>";
            response_head = "HTTP/1.1 200 OK\r\nContent-Type: text/html;charset=utf-8\r\nContent-Length: 58\r\n\r\n";
        }
        else {                                          //登录失败
            response_body = "<html><body>Login Failed, Please check your account and password!</body></html>";
            response_head = "HTTP/1.1 200 OK\r\nContent-Type: text/html;charset=utf-8\r\nContent-Length: 80\r\n\r\n";
        }

        response_head += response_body;
        int size = response_head.length();
        memcpy(respond, response_head.c_str(), size);
        send(now_socket, respond, BUFFERSIZE, 0);               //发送响应报文
        closesocket(now_socket);
        break;
    }
}
```

对于方法为 POST 的处理，首先检查请求数据包中的 URL 字段是否为“/dopost”，如果不是，则首先输出错误信息，接着组装响应数据包，响应消息的状态设置为 404，其他字段也相应设置好，最后调用 send()函数将响应数据包发给客户端，并关闭与该客户端的 socket 连接；如果是，则响应头的状态码设置为 200，之后抽取请求数据包中头部 Content-Length 字段的值，然后抽取体部，并在体部中抽取登录名（login）和密码（pass）的值，接着通过比对判断登录是否成功，并组装不同的响应数据包（包括头部和体部），最后调用 send()将整个数据包发回客户端。

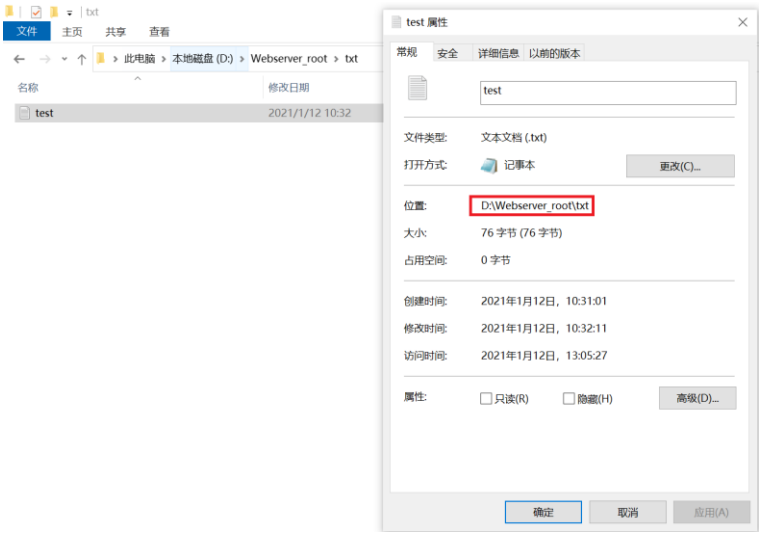
- 服务器运行后，用 netstat –an 显示服务器的监听端口

```
C:\Users\86525>netstat -an

活动连接

  协议   本地地址          外部地址        状态
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:443            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:902            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:912            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:6148           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:7680           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49664          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49665          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49666          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49667          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49668          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49669          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49671          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:53166          0.0.0.0:0              LISTENING
```

- 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：

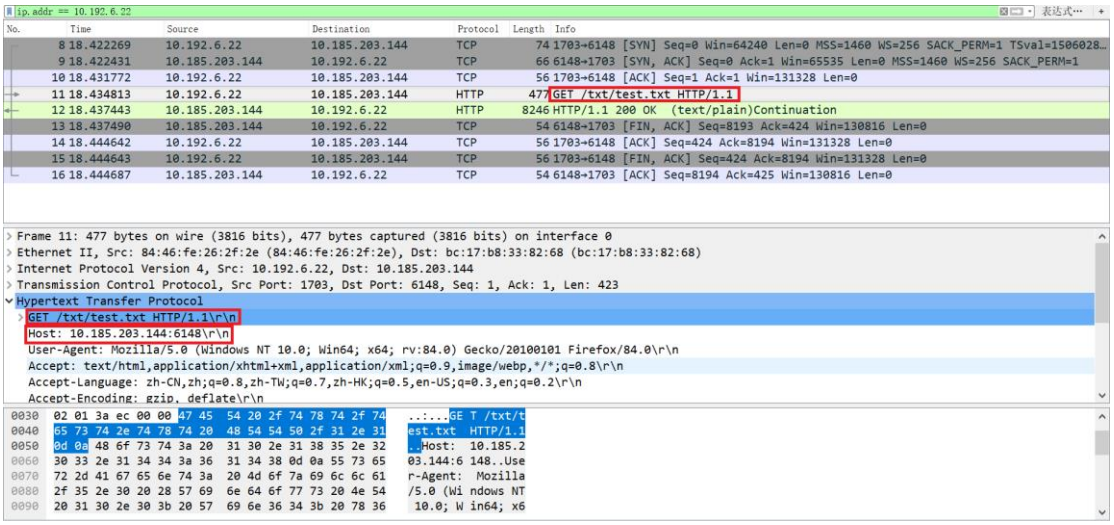


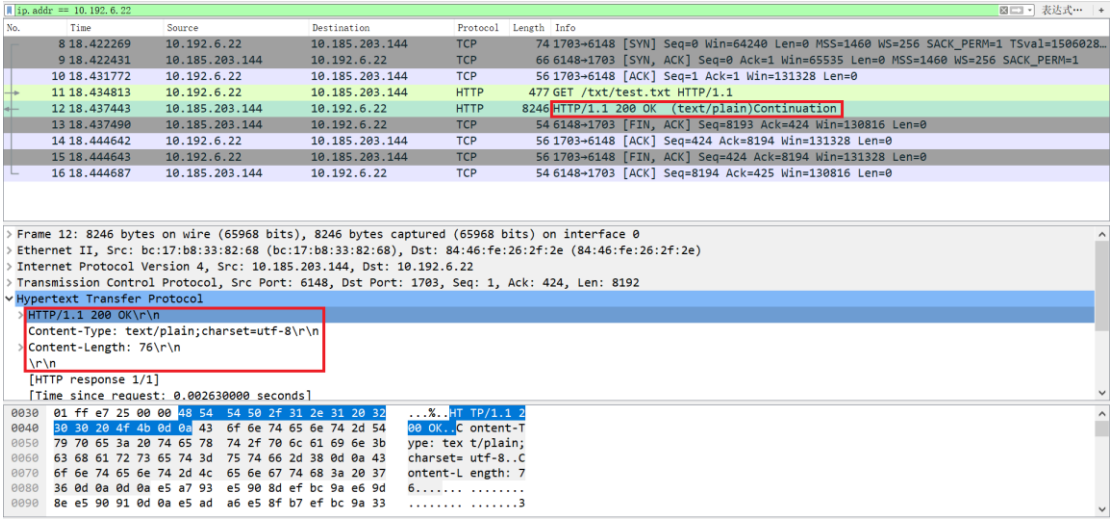
服务器的相关代码片段：

```
if (url.find(".txt") != url.npos) {          //文本文件
    type = "text/plain;charset=utf-8\r\n";
    response_header += type;
}
```

```
if (type != "none") {
    response_header += "Content-Length: ";
    ostringstream ss;
    ss << right - left;
    string length = ss.str();
    response_header += length;
    response_header += "\r\n\r\n";                //两个回车换行表示头部结束
    int size = response_header.length();
    memcpy(respond, response_header.c_str(), size);//头部信息
    memcpy(respond + size, content, right - left);//文件信息

    send(now_socket, respond, BUFFERSIZE, 0);
    closesocket(now_socket);
    break;
}
```

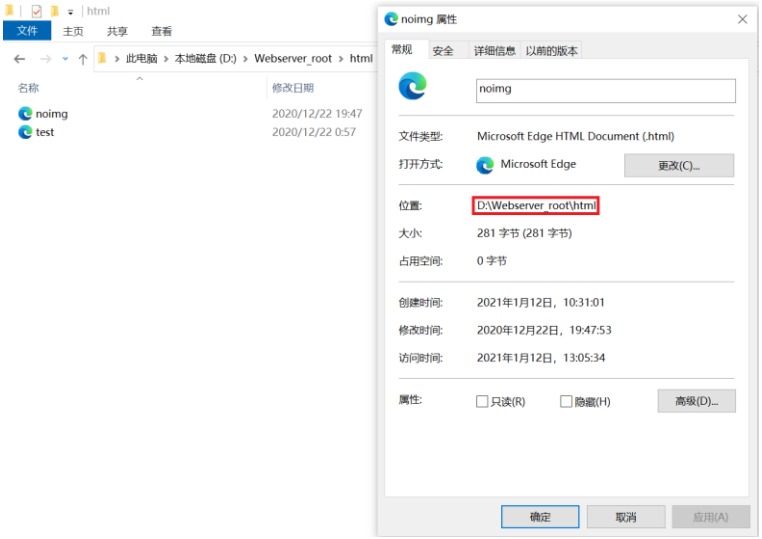
Wireshark 抓取的数据包截图（通过跟踪 TCP 流，只截取 HTTP 协议部分）：



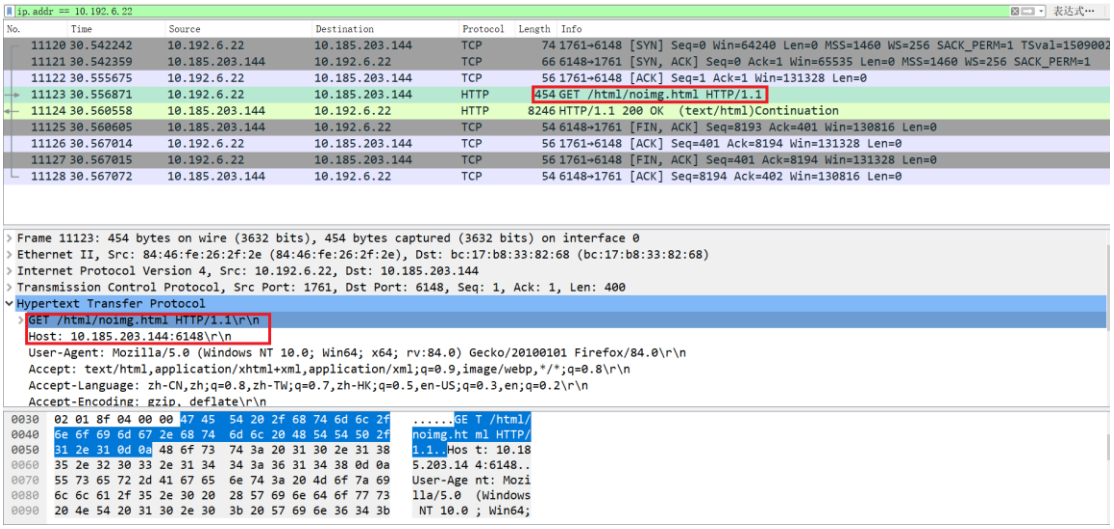


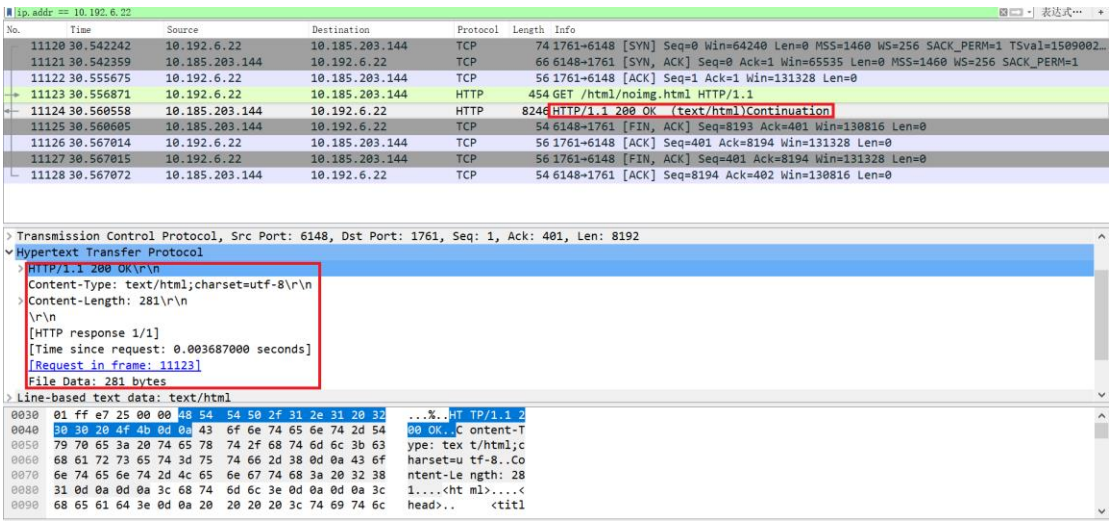
- 浏览器访问只包含文本的 HTML 文件时，浏览器的 URL 地址和显示内容截图。

服务器文件实际存放的路径：

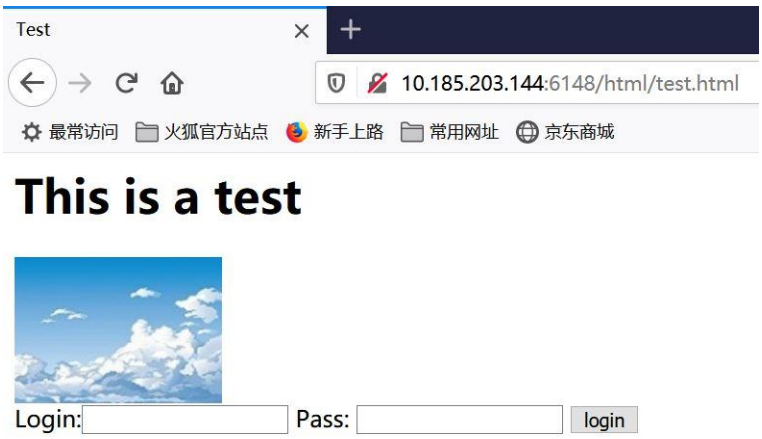


Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML 内容）：

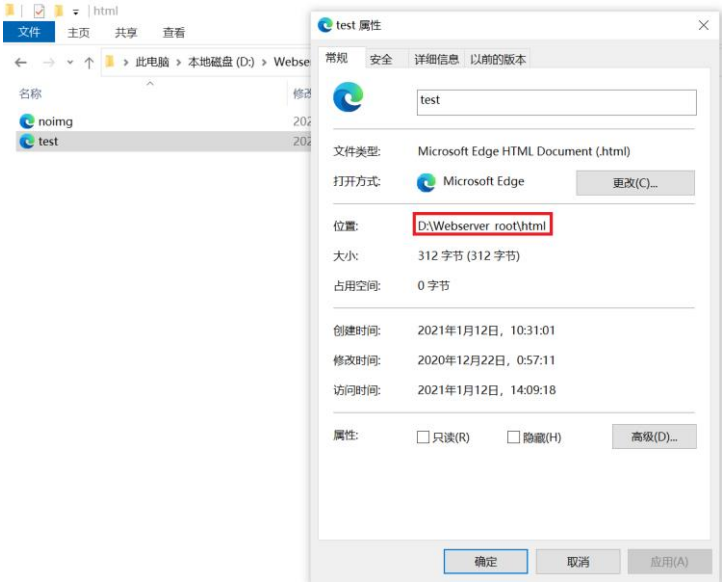


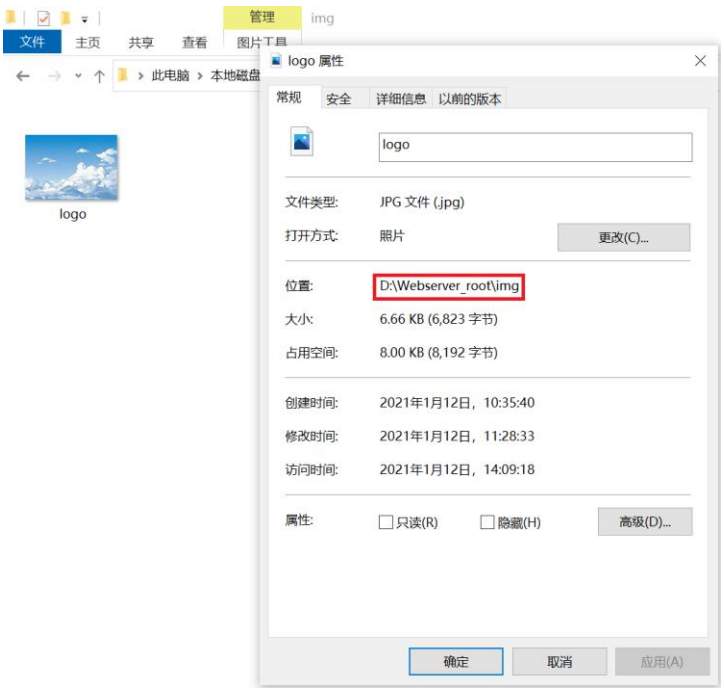


- 浏览器访问包含文本、图片的 HTML 文件时，浏览器的 URL 地址和显示内容截图。

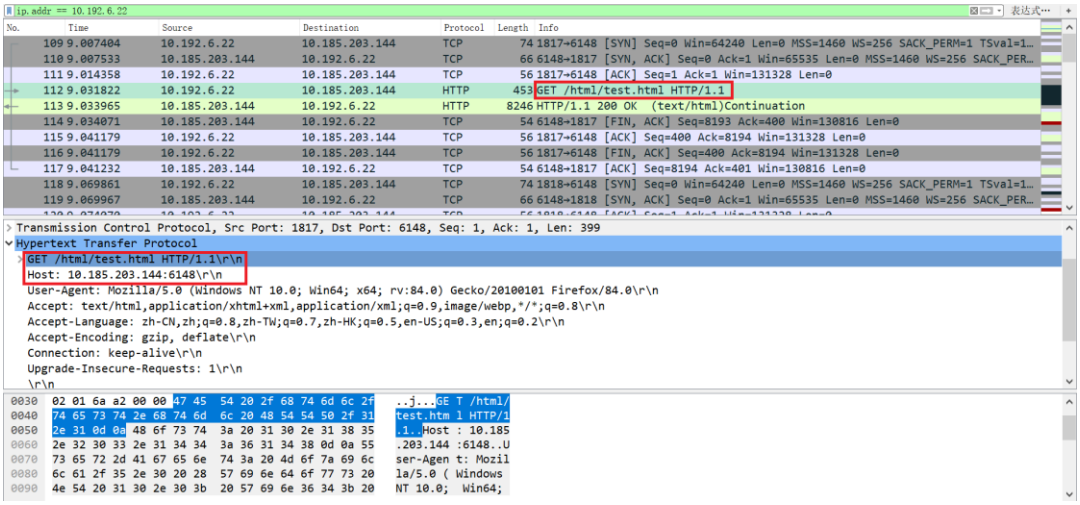


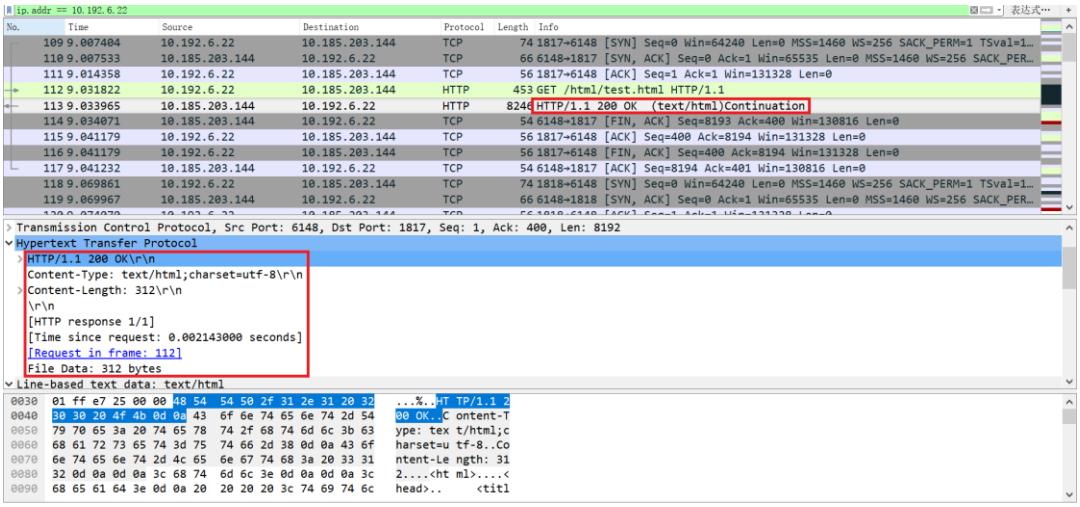
服务器上文件实际存放的路径：

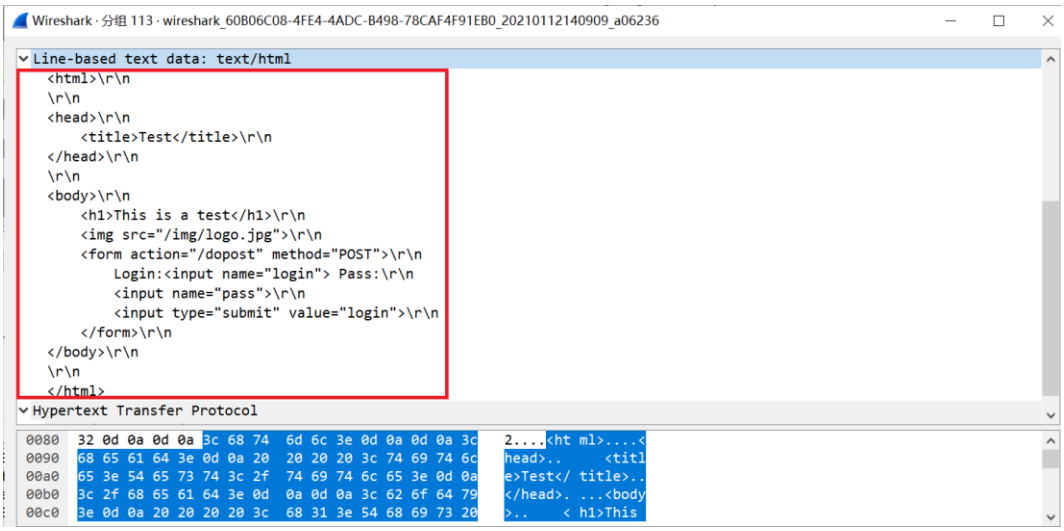


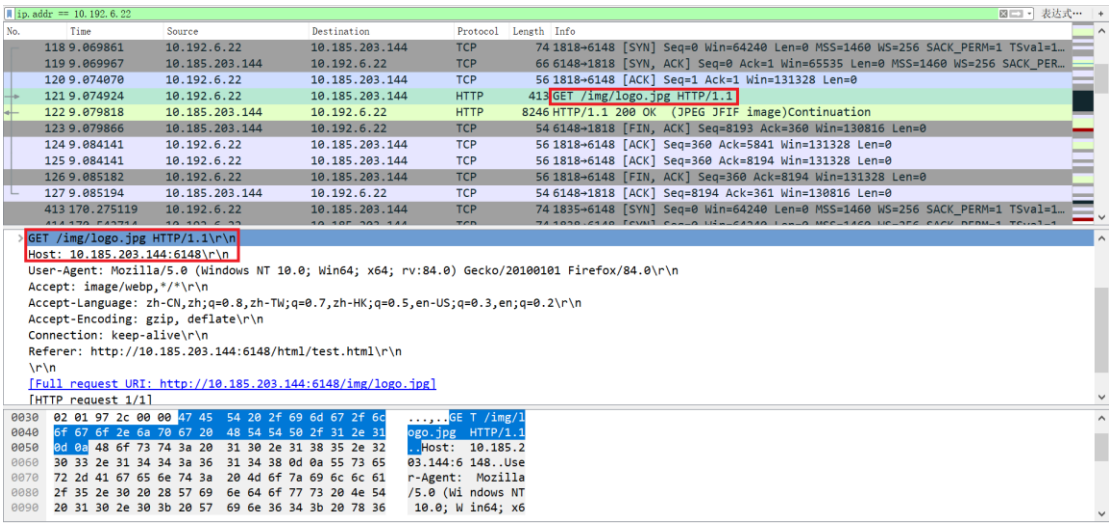


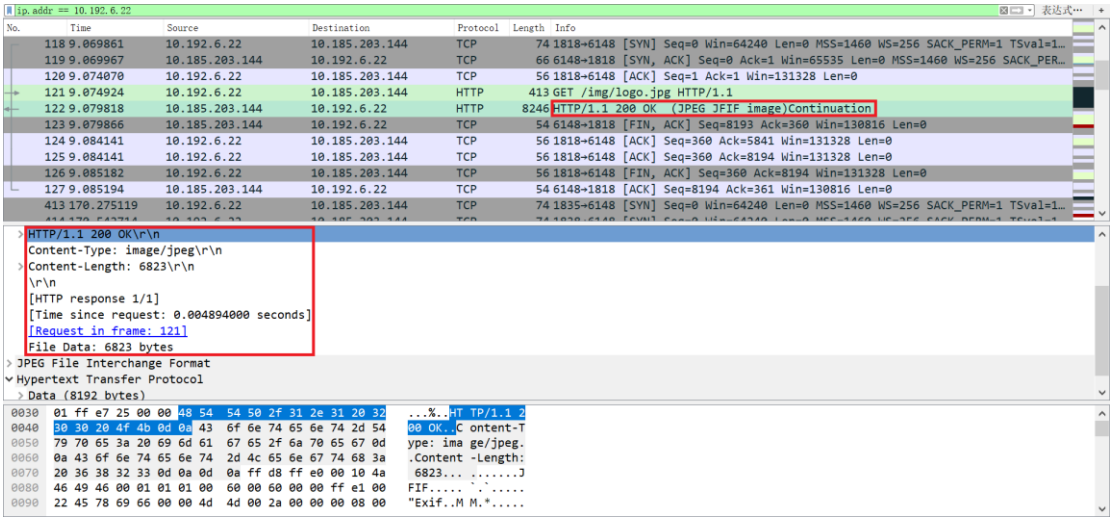
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：

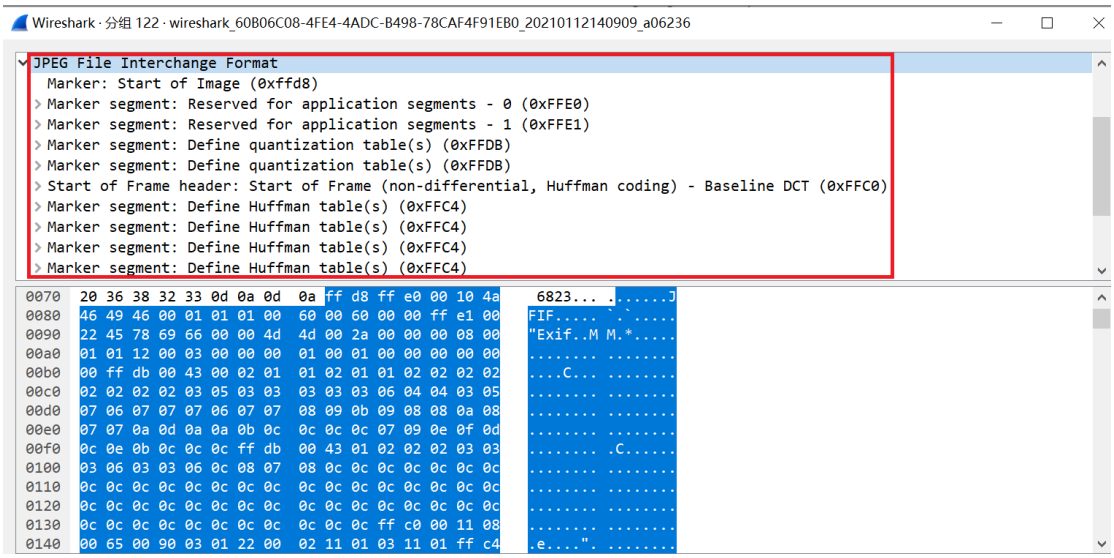



ip.addr == 10.192.6.22
No. Time Source Destination Protocol Length Info
109 9.007404 10.192.6.22 10.185.203.144 TCP 74 1817→6148 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=1…
110 9.007533 10.185.203.144 10.192.6.22 TCP 66 6148→1817 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PER…
111 9.014358 10.192.6.22 10.185.203.144 TCP 56 1817→6148 [ACK] Seq=1 Ack=1 Win=131328 Len=0
112 9.031822 10.192.6.22 10.185.203.144 HTTP 453 GET /html/test.html HTTP/1.1
113 9.033965 10.185.203.144 10.192.6.22 HTTP 8246 HTTP/1.1 200 OK (text/html)Continuation
114 9.034071 10.185.203.144 10.192.6.22 TCP 54 6148→1817 [FIN, ACK] Seq=8193 Ack=400 Win=130816 Len=0
115 9.041179 10.192.6.22 10.185.203.144 TCP 56 1817→6148 [ACK] Seq=400 Ack=8194 Win=131328 Len=0
116 9.041179 10.192.6.22 10.185.203.144 TCP 56 1817→6148 [FIN, ACK] Seq=400 Ack=8194 Win=131328 Len=0
117 9.041232 10.185.203.144 10.192.6.22 TCP 54 6148→1817 [ACK] Seq=8194 Ack=401 Win=130816 Len=0
118 9.069861 10.192.6.22 10.185.203.144 TCP 74 1818→6148 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=1…
119 9.069967 10.185.203.144 10.192.6.22 TCP 66 6148→1818 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PER…
Transmission Control Protocol, Src Port: 6148, Dst Port: 1817, Seq: 1, Ack: 400, Len: 8192
Hypertext Transfer Protocol
HTTP/1.1 200 OK\r\n
Content-Type: text/html;charset=utf-8\r\n
Content-Length: 312\r\n
\r\n
[HTTP response 1/1]
[Time since request: 0.002143000 seconds]
[Request in frame: 112]
File Data: 312 bytes
Line-based text data: text/html
0030 01 ff e7 25 00 00 48 54 54 50 2f 31 2e 31 20 32 ...%..HT TP/1.1 2
0040 30 30 20 4f 4b 0d 0a 43 6f 6e 74 65 6e 74 2d 54 00 OK..C ontent-T
0050 79 70 65 3a 20 74 65 78 74 2f 68 74 6d 6c 3b 63 ype: tex t/html;c
0060 68 61 72 73 65 74 3d 75 74 66 2d 38 0d 0a 43 6f harset=u tf-8..Co
0070 6e 74 65 6e 74 2d 4c 65 6e 67 74 68 3a 20 33 31 ntent-Le ngth: 31
0080 32 0d 0a 0d 0a 3c 68 74 6d 6c 3e 0d 0a 0d 0a 3c 2....<ht ml>....<
0090 68 65 61 64 3e 0d 0a 20 20 20 20 3c 74 69 74 6c head>.. <titl

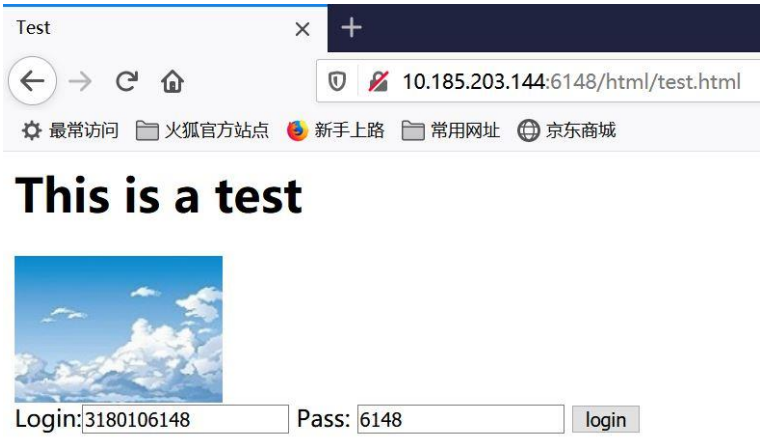

Wireshark · 分组 113 · wireshark_60B06C08-4FE4-4ADC-B498-78CAF4F91EB0_20210112140909_a06236
Line-based text data: text/html
<html>\r\n
\r\n
<head>\r\n
<title>Test</title>\r\n
</head>\r\n
\r\n
<body>\r\n
<h1>This is a test</h1>\r\n
<img src="/img/logo.jpg">\r\n
<form action="/dopost" method="POST">\r\n
Login:<input name="login"> Pass:\r\n
<input name="pass">\r\n
<input type="submit" value="login">\r\n
</form>\r\n
</body>\r\n
\r\n
</html>
Hypertext Transfer Protocol
0080 32 0d 0a 0d 0a 3c 68 74 6d 6c 3e 0d 0a 0d 0a 3c 2....<ht ml>....<
0090 68 65 61 64 3e 0d 0a 20 20 20 20 3c 74 69 74 6c head>.. <titl
00a0 65 3e 54 65 73 74 3c 2f 74 69 74 6c 65 3e 0d 0a e>Test</ title>..
00b0 3c 2f 68 65 61 64 3e 0d 0a 0d 0a 3c 62 6f 64 79 </head>. ...<body
00c0 3e 0d 0a 20 20 20 20 3c 68 31 3e 54 68 69 73 20 >.. < h1>This


ip.addr == 10.192.6.22
No. Time Source Destination Protocol Length Info
118 9.069861 10.192.6.22 10.185.203.144 TCP 74 1818→6148 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=1…
119 9.069967 10.185.203.144 10.192.6.22 TCP 66 6148→1818 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PER…
120 9.074070 10.192.6.22 10.185.203.144 TCP 56 1818→6148 [ACK] Seq=1 Ack=1 Win=131328 Len=0
121 9.074924 10.192.6.22 10.185.203.144 HTTP 413 GET /img/logo.jpg HTTP/1.1
122 9.079818 10.185.203.144 10.192.6.22 HTTP 8246 HTTP/1.1 200 OK (JPEG JFIF image)Continuation
123 9.079866 10.185.203.144 10.192.6.22 TCP 54 6148→1818 [FIN, ACK] Seq=8193 Ack=360 Win=130816 Len=0
124 9.084141 10.192.6.22 10.185.203.144 TCP 56 1818→6148 [ACK] Seq=360 Ack=5841 Win=131328 Len=0
125 9.084141 10.192.6.22 10.185.203.144 TCP 56 1818→6148 [ACK] Seq=360 Ack=8194 Win=131328 Len=0
126 9.085182 10.192.6.22 10.185.203.144 TCP 56 1818→6148 [FIN, ACK] Seq=360 Ack=8194 Win=131328 Len=0
127 9.085194 10.185.203.144 10.192.6.22 TCP 54 6148→1818 [ACK] Seq=8194 Ack=361 Win=130816 Len=0
413 170.275119 10.192.6.22 10.185.203.144 TCP 74 1835→6148 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1 TSval=1…
GET /img/logo.jpg HTTP/1.1\r\n
Host: 10.185.203.144:6148\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101 Firefox/84.0\r\n
Accept: image/webp,*/*\r\n
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\r\n
Accept-Encoding: gzip, deflate\r\n
Connection: keep-alive\r\n
Referer: http://10.185.203.144:6148/html/test.html\r\n
\r\n
[Full request URI: http://10.185.203.144:6148/img/logo.jpg]
[HTTP request 1/1]
0030 02 01 97 2c 00 00 47 45 54 20 2f 69 6d 67 2f 6c ...,..GE T /img/l
0040 6f 67 6f 2e 6a 70 67 20 48 54 54 50 2f 31 2e 31 ogo.jpg HTTP/1.1
0050 0d 0a 48 6f 73 74 3a 20 31 30 2e 31 38 35 2e 32 ..Host: 10.185.2
0060 30 33 2e 31 34 34 3a 36 31 34 38 0d 0a 55 73 65 03.144:6 148..Use
0070 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69 6c 6c 61 r-Agent: Mozilla
0080 2f 35 2e 30 20 28 57 69 6e 64 6f 77 73 20 4e 54 /5.0 (Wi ndows NT
0090 20 31 30 2e 30 3b 20 57 69 6e 36 34 3b 20 78 36 10.0; W in64; x6





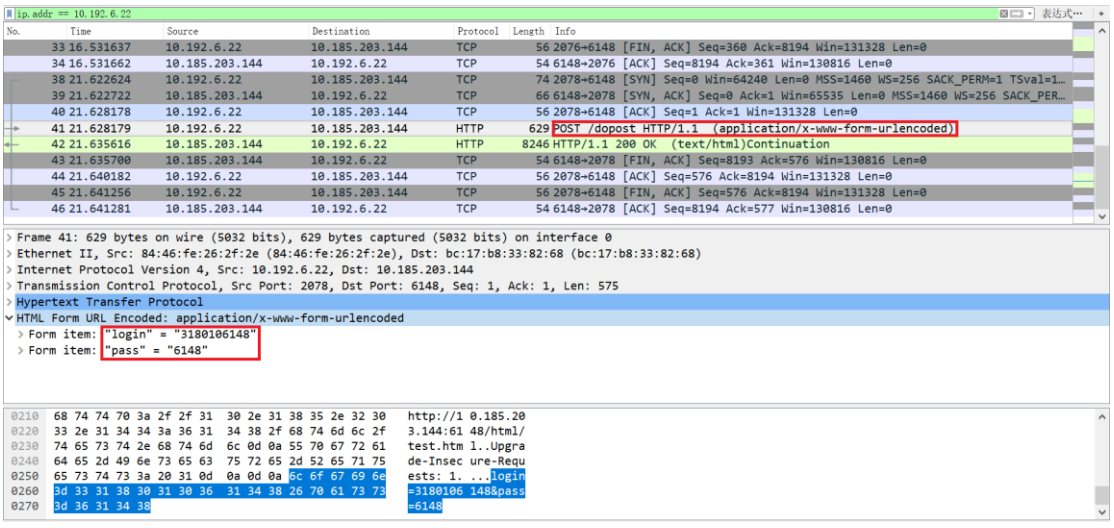
- 浏览器输入正确的登录名或密码，点击登录按钮（login）后的显示截图。

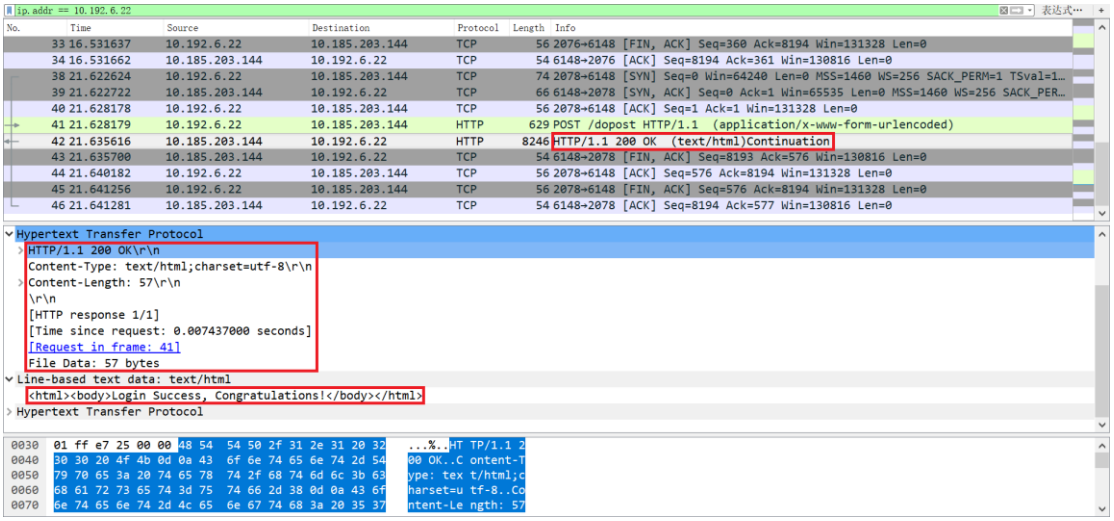


服务器相关处理代码片段：

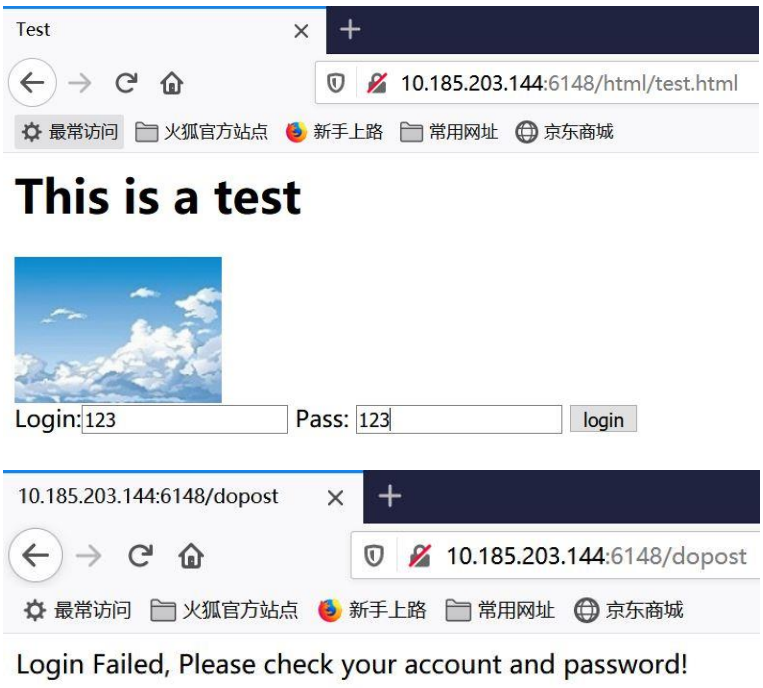
```
//如果路径名是dopost
else {
    int cont_len = HTTP_request.find("Content-Length:");
    string tmp;
    header.clear();
    header.str(HTTP_request.substr(cont_len));
    header >> tmp >> cont_len;                  //抽取content-length

    int start = HTTP_request.find("\r\n\r\n");
    if (start != HTTP_request.npos) {
        string body;
        body = HTTP_request.substr(start + 4, cont_len);    //获取体部

        int pos1 = body.find("login=");
        int pos2 = body.find("&pass=");
        if (pos1 != body.npos && pos2 != body.npos && pos1 < pos2) {
            string login = body.substr(pos1 + 6, pos2 - pos1 - 6);
            string pswd = body.substr(pos2 + 6);

            if (login == LOGIN && pswd == PASS) {           //登录成功
                response_body = "<html><body>Login Success, Congratulations!</body></html>";
                response_head = "HTTP/1.1 200 OK\r\nContent-Type: text/html;charset=utf-8\r\nContent-Length: 57\r\n\r\n";
            }
            else {                                          //登录失败
                response_body = "<html><body>Login Failed, Please check your account and password!</body></html>";
                response_head = "HTTP/1.1 200 OK\r\nContent-Type: text/html;charset=utf-8\r\nContent-Length: 80\r\n\r\n";
            }

            response_head += response_body;
```

Wireshark 抓取的数据包截图（HTTP 协议部分）

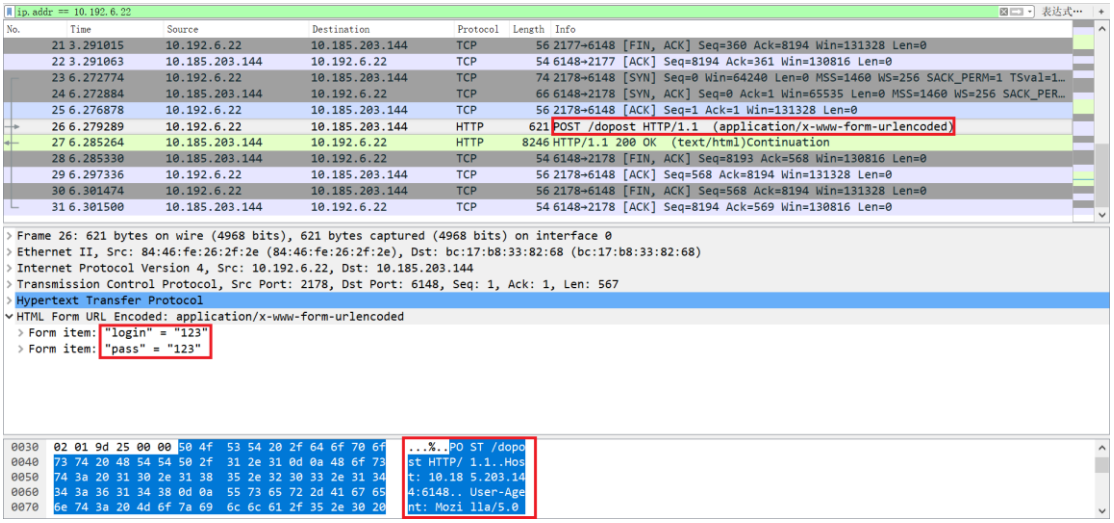


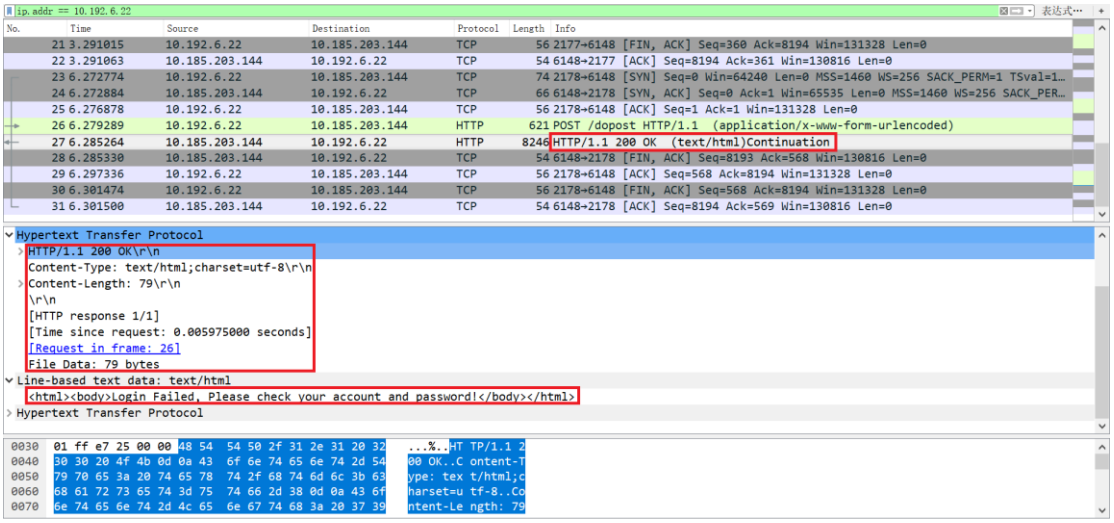


- 浏览器输入错误的登录名或密码，点击登录按钮（login）后的显示截图。

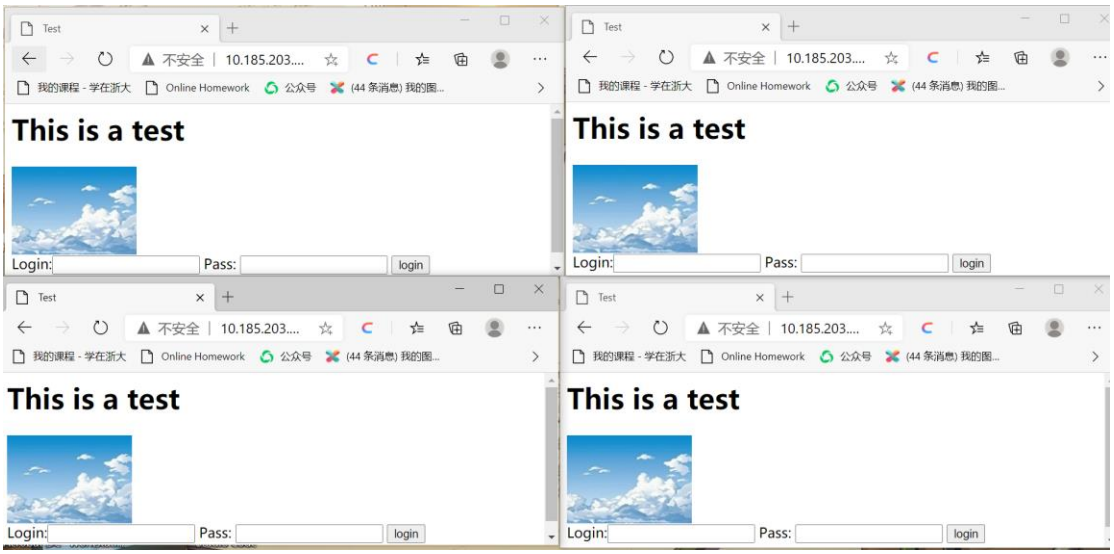


- Wireshark 抓取的数据包截图（HTTP 协议部分）





- 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）

```
C:\Users\86525>netstat -an

活动连接

  协议   本地地址          外部地址        状态
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:443            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:902            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:912            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:6148           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:7680           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49664          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49665          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49666          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49667          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49668          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49669          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:49671          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:53166          0.0.0.0:0              LISTENING
  TCP    10.185.203.144:139     0.0.0.0:0              LISTENING
  TCP    10.185.203.144:6148    10.192.6.22:2259       TIME_WAIT
  TCP    10.185.203.144:6148    10.192.6.22:2267       TIME_WAIT
  TCP    10.185.203.144:6148    10.192.6.22:2271       TIME_WAIT
  TCP    10.185.203.144:6148    10.192.6.22:2272       TIME_WAIT
  TCP    10.185.203.144:6148    10.192.6.22:2275       TIME_WAIT
  TCP    10.185.203.144:6148    10.192.6.22:2276       TIME_WAIT
  TCP    10.185.203.144:6148    10.192.6.22:2279       TIME_WAIT
  TCP    10.185.203.144:6148    10.192.6.22:2280       TIME_WAIT
  TCP    10.185.203.144:7680    10.185.196.208:63129   ESTABLISHED
  TCP    10.185.203.144:7680    10.185.196.208:63152   ESTABLISHED
```

六、 实验结果与分析

- HTTP 协议是怎样对头部和体部进行分隔的？

  答：通过两个连续的 CRLF（回车换行）进行分隔，也即用一个空行来分隔。

- 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：浏览器是根据头部的 Content-Type 字段来判断文件类型的，通过查阅资料我了解到常见的 Content-Type 值有：

text/html ： HTML 格式

text/plain ： 纯文本格式

text/xml ： XML 格式

image/jpeg ： jpg 图片格式

image/png: png 图片格式

- HTTP 协议的头部是不是一定是文本格式？体部呢？

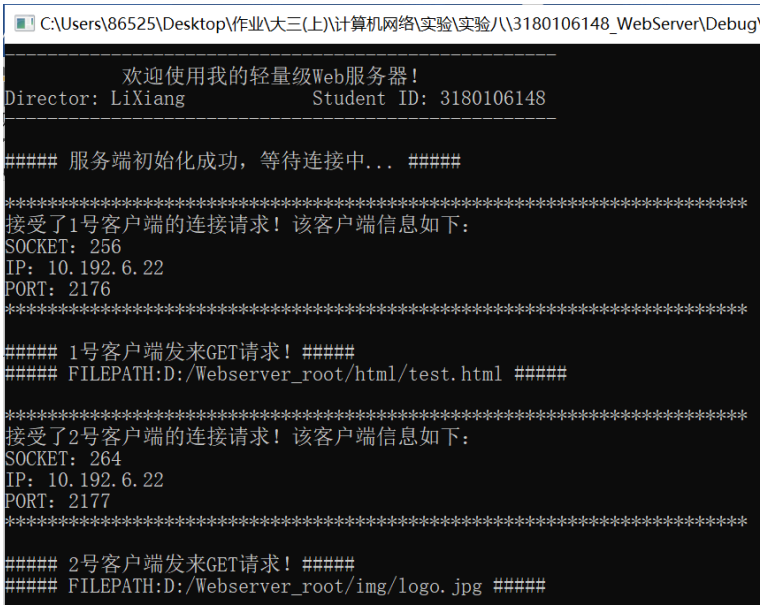
  答：HTTP 协议的头部一定是文本格式，但体部不一定是文本格式，还可以是字节流的方式，以字节流的方式来完成非文本数据如音频、图片等数据的传输。

- 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：POST 方法传递的数据是放在体部，两个字段是用‘&’符号连接起来的。

七、 讨论、心得

1. 每次通过浏览器请求资源时服务器都会接收到两个相邻 PORT 号的客户端连接请求，在请求带有图片的 html 时也会发现前一个用来 GET html 文件，后一个用来 GET jpg 图片文件，可以猜测浏览器图片和文字的加载是异步进行的。



2. 在处理 POST 请求的实验过程中，起初我通过 readline 方法来逐行从输入流中读取请求内容，但是出现了读取始终不停止的问题。查阅资料后得知，当服务端在读取输入流时，只要不发回响应就不会停止，也就是始终会等待客户端的输入流，为了解决这一问题我通过首先抽取出请求中 Content-Length 的值来确定要读取内容的长度而不是循环读取，解决了这个问题
3. 在处理对于带有图片的 html 的请求时，起初发现每次图片一开始加载之后，服务器端就自动退出了，图片根本加载不出来。找了很久的 bug 才发现是因为自己设置的接收缓冲区的大小只有 8K，而图片有 50 多 K，缓冲区溢出导致了程序退出。经过修改了图片的尺寸之后解决了这个问题。
4. 起初两个 html 文件按照实验指导提供的代码编写均出现了问题，仔细检查之后发现，原来的 html 中的图片相关代码为 src=“img/logo.jpg”，实际上此时请求图片时的 URL 是“/html/img/logo.jpg”，故把 html 文本改为 src=“/img/logo.jpg”，同理在提交表单时，由于最开始写的<form action=“dopost” method=“POST”>导致提交时跳转的 URL 是“/html/dopost”而不是“/dopost”，也需要修改。