

# Lab2 实现一个轻量级的WEB服务器

姓名:陈希典  
学号:3210102362  
指导老师:郑扣根  
日期:2023年10月24日  
同组学生:无

## 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
- 准备好一个图片文件，命名为logo.jpg，放在img子目录下
- 写一个HTML文件，命名为test.html，放在html子目录下，主要内容为：

```
<html>
  <head><title>Test</title></head>
  <body>
    <h1>This is a test</h1>
    
    <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

- 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

### 服务器的主线程循环

```
int main(){
    std::signal(SIGINT, signalHandler); // 登记一个信号,用来退出服务器
    Init();// 套接字初始化

    while(serverRunning){

        SOCKET* clientSocket = new(SOCKET);
        struct sockaddr_in clientAddr;
        int clientAddrLength = sizeof(clientAddr);
        // 主线程循环调用accept 直到返回一个有效的socket句柄
        *clientSocket = accept(serverSocket, (struct sockaddr*)&clientAddr, &clientAddrLength);
        // 获取客户端地址信息,记录下该客户端句柄和连接状态、端口。
        char* clientIP = inet_ntoa(clientAddr.sin_addr);
        int clientPort = ntohs(clientAddr.sin_port);
        int temidx = clients_map.size();
        clients_map[temidx] = ClientInfo(clientSocket, clientIP, clientPort);

        // 然后创建一个子线程后继续调用accept()
        thread clientThread(clientHandle,clientSocket,temidx);
        clientThread.detach();
    }
    cout <<"Finish!" << endl;
    return 0;
}
```

循环调用accept函数接收到一个client,建立连接,同时登记其信息,创建一个处理子线程

同时使用一个原子量atomic serverRunning(true); 和登记一个信号量 SIGINT来检测用户输入或者系统发来的中断信号,并且在检测到中断信号时关闭所有子线程,切断服务器

```

// Global flag to indicate server termination
atomic<bool> serverRunning(true);

void signalHandler(int signal)
{
    if (signal == SIGINT)
    {
        serverRunning = false;
        cout << "BYEBYE!" << endl;
        closesocket(serverSocket);
        WSACleanup();
    }
}

```

## 服务器的客户端处理子线程

```

//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。
void clientHandle(SOCKET* clientSocketptr, int clientIdx){

    cout << "client " << clientIdx << clients_map[clientIdx].IP << " connected!" << endl;

    char recvBuffer[2048] = {0};
    bool HandleFinish = false;
    while(!HandleFinish & serverRunning){ // serverRunning 用来通知子线程退出
        int recv_handler = recv(*clientSocketptr, recvBuffer, 2048, 0);
        if (recv_handler == SOCKET_ERROR) {
            std::cerr << "Failed to read data from socket. Error code: "
                << WSAGetLastError() << std::endl;
            closesocket(*clientSocketptr);
            WSACleanup();
            return ;
        }
        else if (recv_handler == 0) {
            std::cout << "client disconnected." << std::endl;
            closesocket(*clientSocketptr);
            WSACleanup();
            return ;
        }
        else {
            if(checkDoubleEnter(recvBuffer,100)){
                parse_request_header(*clientSocketptr, recvBuffer);
                HandleFinish = true;
            }
        }
    }
    Sleep(1); // 很重要,因为HTML还有后续请求,这需要一定时间相应
    closesocket(*clientSocketptr);
    clients_map.erase(clientIdx);
}

```

循环调用recv函数,每次都接受一个包,或者一组包,放到缓冲区,然后检查连接情况,如果是正常收到一个包,并发现其出现了两个回车换行符,就按照HTTP协议规定的格式进行拆包。

```

void parse_request_header(SOCKET clientSocket, char * request) {

    // ofstream outputfile("output.txt");
    // outputfile << string(request);
    // outputfile.close();

    std::istringstream request_stream(request);
    std::string request_line;

    // Read the first line of the request, which contains the request method, URL, and HTTP version
    std::getline(request_stream, request_line);

    // Parse the request line
    std::string request_method;
    std::string request_url;
    std::string http_version;
    std::istringstream request_line_stream(request_line);
    request_line_stream >> request_method >> request_url >> http_version;

    // Output the parsed information
    std::cout << "Request Method: " << request_method << std::endl;
    std::cout << "Request URL: " << request_url << std::endl;
    std::cout << "HTTP Version: " << http_version << std::endl;

    if(request_method == "GET"){
        GEThandle(request_url,clientSocket);
    }else if(request_method == "POST"){
        POSThandle(request_url,clientSocket,request);
    }
}

```

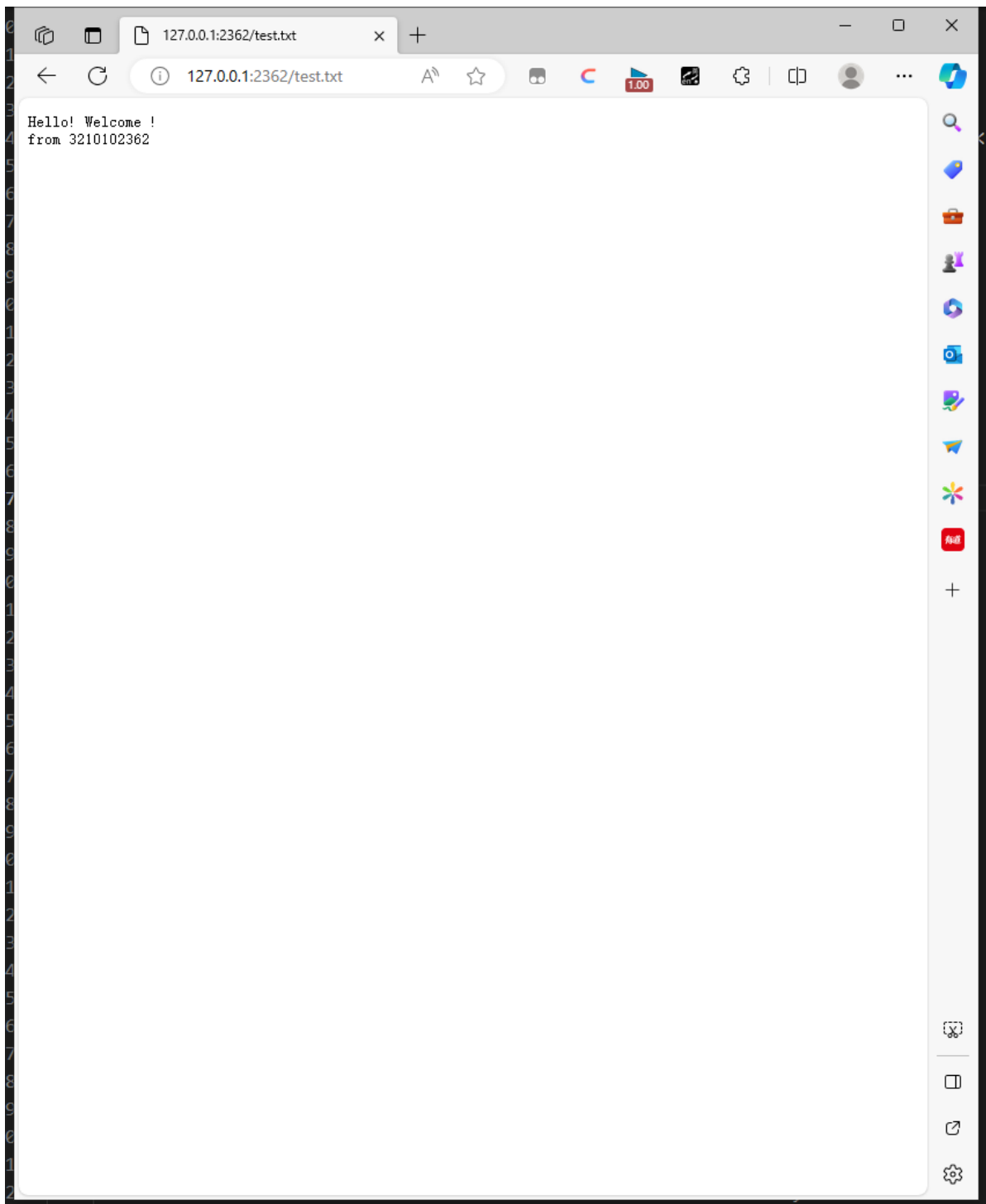
进行拆包和调用GET或者POST的处理函数,最后发包,结束一次对话

最后子线程在完成发送任务或者是收到中断信号时候退出,收回连接的Socket句柄

## 服务器运行后,用netstat -an显示服务器的监听端口

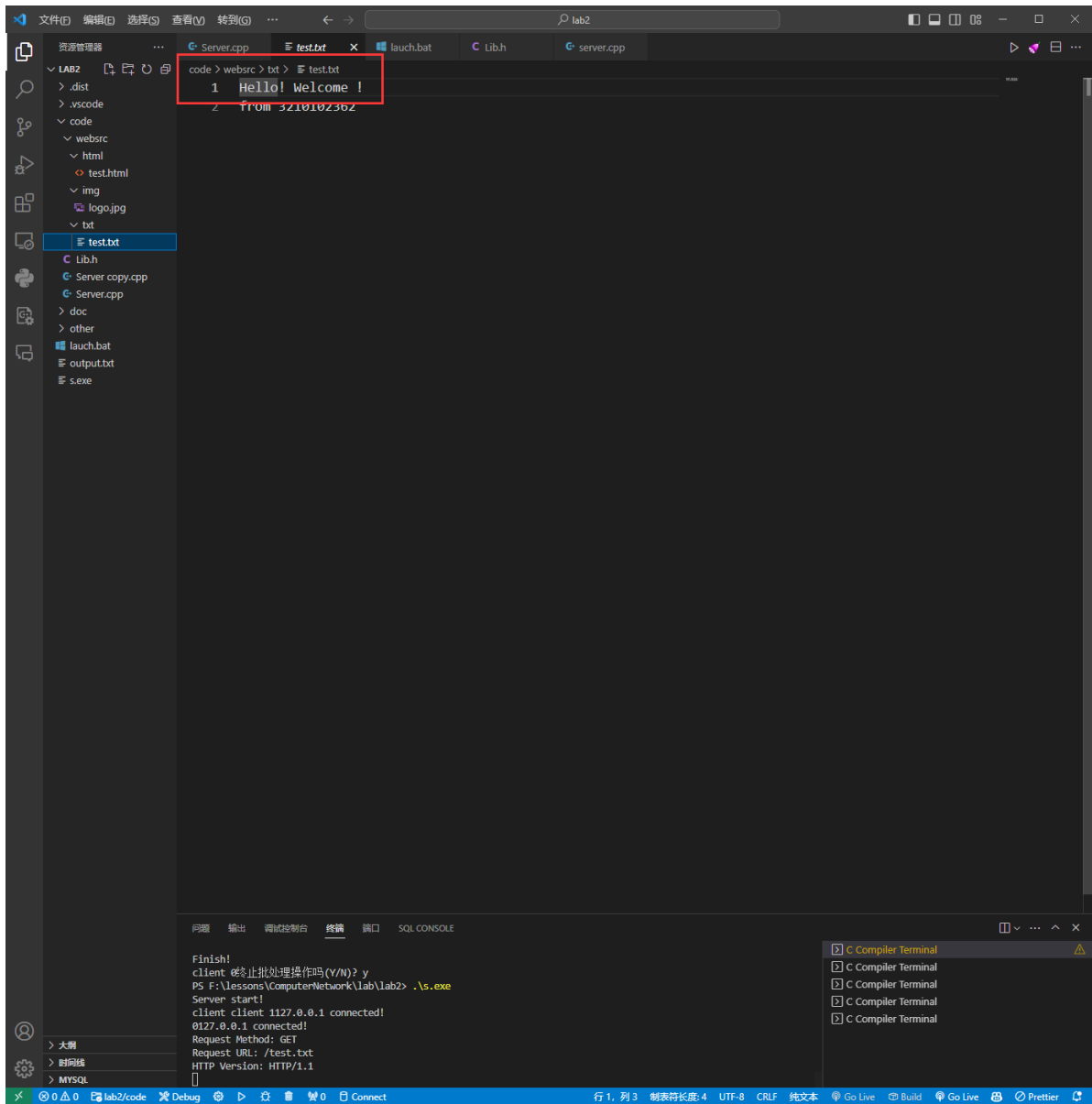
协议	本地地址	外部地址	状态
TCP	0.0.0.0:135	0.0.0.0:0	LISTENING
TCP	0.0.0.0:445	0.0.0.0:0	LISTENING
TCP	0.0.0.0:902	0.0.0.0:0	LISTENING
TCP	0.0.0.0:912	0.0.0.0:0	LISTENING
TCP	0.0.0.0:2362	0.0.0.0:0	LISTENING
TCP	0.0.0.0:3306	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5040	0.0.0.0:0	LISTENING
TCP	0.0.0.0:5091	0.0.0.0:0	LISTENING
TCP	0.0.0.0:7680	0.0.0.0:0	LISTENING
TCP	0.0.0.0:7890	0.0.0.0:0	LISTENING
TCP	0.0.0.0:33060	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	0.0.0.0:49668	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49671	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49680	0.0.0.0:0	LISTENING
TCP	0.0.0.0:49692	0.0.0.0:0	LISTENING
TCP	0.0.0.0:60760	0.0.0.0:0	LISTENING
TCP	10.162.117.167:139	0.0.0.0:0	LISTENING
TCP	10.162.117.167:50063	112.13.119.52:443	ESTABLISHED
TCP	10.162.117.167:51088	223.252.199.69:6003	ESTABLISHED
TCP	10.162.117.167:51836	36.155.197.24:443	CLOSE_WAIT
TCP	10.162.117.167:52046	120.232.65.165:80	ESTABLISHED
TCP	10.162.117.167:52070	183.194.237.168:443	CLOSE_WAIT
TCP	10.162.117.167:52071	36.155.197.24:443	CLOSE_WAIT
TCP	10.162.117.167:52072	36.155.197.24:443	CLOSE_WAIT

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：

F:\lessons\ComputerNetwork\lab\lab2\code\websrc\txt\test.txt



服务器的相关代码片段：

```

void parse_request_header(SOCKET clientSocket, char * request) {

    std::istringstream request_stream(request);
    std::string request_line;

    // Read the first line of the request, which contains the request method, URL, and HTTP version
    std::getline(request_stream, request_line);

    // Parse the request line
    std::string request_method;
    std::string request_url;
    std::string http_version;
    std::istringstream request_line_stream(request_line);
    request_line_stream >> request_method >> request_url >> http_version;

    // Output the parsed information
    std::cout << "Request Method: " << request_method << std::endl;
    std::cout << "Request URL: " << request_url << std::endl;
    std::cout << "HTTP Version: " << http_version << std::endl;

    if(request_method == "GET"){
        GEThandle(request_url, clientSocket);
    }else if(request_method == "POST"){
        POSThandle(request_url, clientSocket, request);
    }
}

```

首先会收到一个请求,这里保存在一个request缓冲区之中,然后服务器对其进行解析,发现是GET请求,于是调用GEThandle函数

在GEThandle函数之中,分别进行如下的内容解释:

一个是解析URL,然后组装响应消息,然后根据解析得来的URL寻找实际的文件位置,并打开文件



```

//解析出来的方法是GET
void GEThandle(string request_url, SOCKET clientSocket){
    // Define the content type based on the file extension
    std::string content_type;
    if (request_url.find(".html") != std::string::npos) {
        content_type = "text/html";
    } else if (request_url.find(".jpg") != std::string::npos) {
        content_type = "image/jpeg";
    } else if (request_url.find(".txt") != std::string::npos) {
        content_type = "text/plain";
    } else {
        content_type = "text/plain"; // Default content type (plain text)
    }

    // Check if the request URL is empty, and if so, send a "hello" response
    if (request_url.empty() | request_url == "/") {
        const char* response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: ";
        send(clientSocket, response, strlen(response), 0);
        return;
    }

    // Prepare the file path
    std::string filePath;
    if (request_url == "/test.html") {
        filePath = "code/websrc/html/test.html";
    } else if (request_url == "/logo.jpg") {
        filePath = "code/websrc/img/logo.jpg";
    } else if (request_url == "/test.txt") {
        filePath = "code/websrc/txt/test.txt";
    } else {
        // Handle 404 Not Found for other URLs
        const char* response = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nFile ";
        send(clientSocket, response, strlen(response), 0);
        return;
    }
}

```

打开需要发送的文件,写入到response String之中,最后和组装好的响应头进行一次拼接,存在一个 malloc 来的大缓冲区里面,然后把写好的响应包发出

```
// Use C standard library functions to read the file
FILE* file = fopen(filePath.c_str(), "rb");
if (file) {
    // Determine the content length for the response
    fseek(file, 0, SEEK_END);
    long contentLength = ftell(file);
    // cout << " contentLength " << contentLength << endl;
    fseek(file, 0, SEEK_SET);

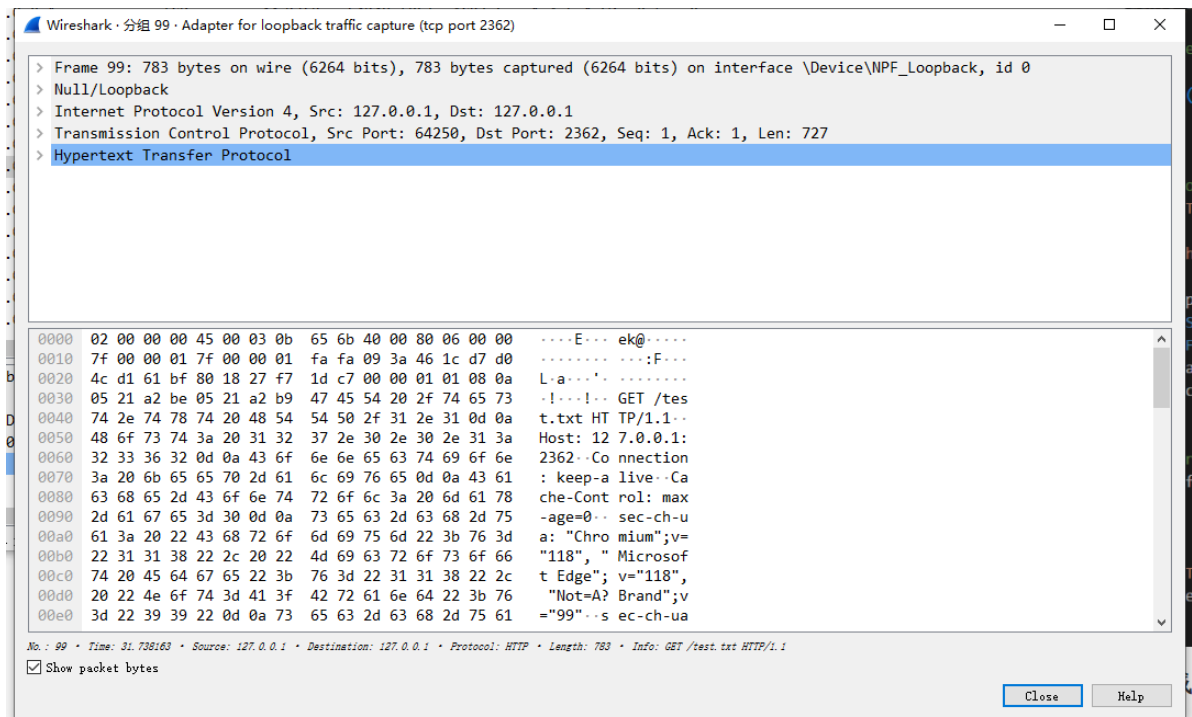
    // Construct the HTTP response with proper headers
    std::string response = "HTTP/1.1 200 OK\r\n";
    response += "Content-Type: " + content_type + "\r\n";
    response += "Content-Length: " + std::to_string(contentLength) + "\r\n\r\n";

    int headerlen = strlen(response.c_str());
    char* sendbuff = new char[SENDBUFFSIZE];
    memset(sendbuff, 0, SENDBUFFSIZE);
    while (fread(sendbuff + headerlen, 1, contentLength, file) > 0);
    memcpy(sendbuff, response.c_str(), headerlen);

    // Send the response header
    send(clientSocket, sendbuff, headerlen + contentLength + 10, 0);
    fclose(file);
} else {
    // Handle file not found
    const char* response = "HTTP/1.1 404 Not Found\r\nContent-Type: text/plain\r\n\r\nFile";
    send(clientSocket, response, strlen(response), 0);
}
}
```

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

首先在浏览器中输入对应地址：[127.0.0.1:2362/test.txt](http://127.0.0.1:2362/test.txt)



发出GET包

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/web
Sec-Fetch-Site: cross-site\r\n
Sec-Fetch-Mode: navigate\r\n
Sec-Fetch-User: ?1\r\n
Sec-Fetch-Dest: document\r\n
Accept-Encoding: gzip, deflate, br\r\n
Accept-Language: zh,en;q=0.9,en-US;q=0.8,en-GB;q=0.7\r\n
\r\n
```

[Full request URI: http://127.0.0.1:2362/test.txt]

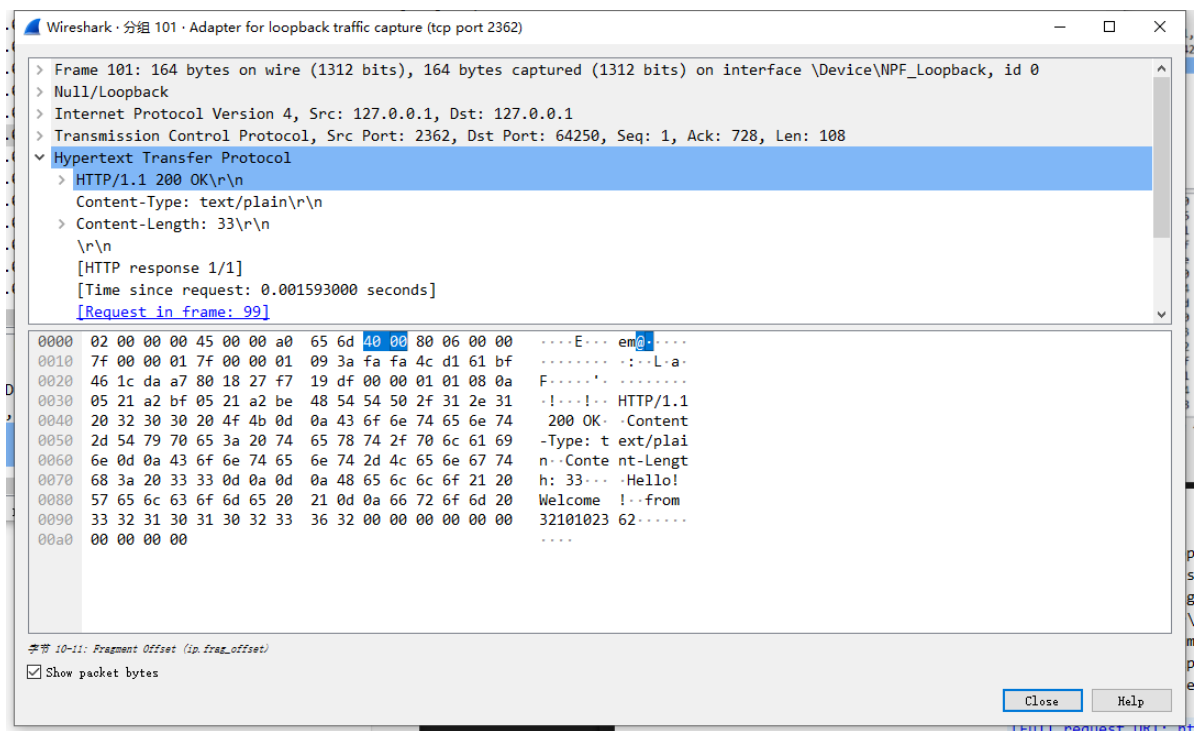
[HTTP request 1/1]

[Response in frame: 101]

20	3b 71 3d 30 2e 38 2c 61 70 70 6c 69 63 61 74 69	;q=0.8,application
30	6f 6e 2f 73 69 67 6e 65 64 2d 65 78 63 68 61 6e	on/sign d-exchan

内容就是GET请求

然后收到服务器发送回来的响应包,发送内容为

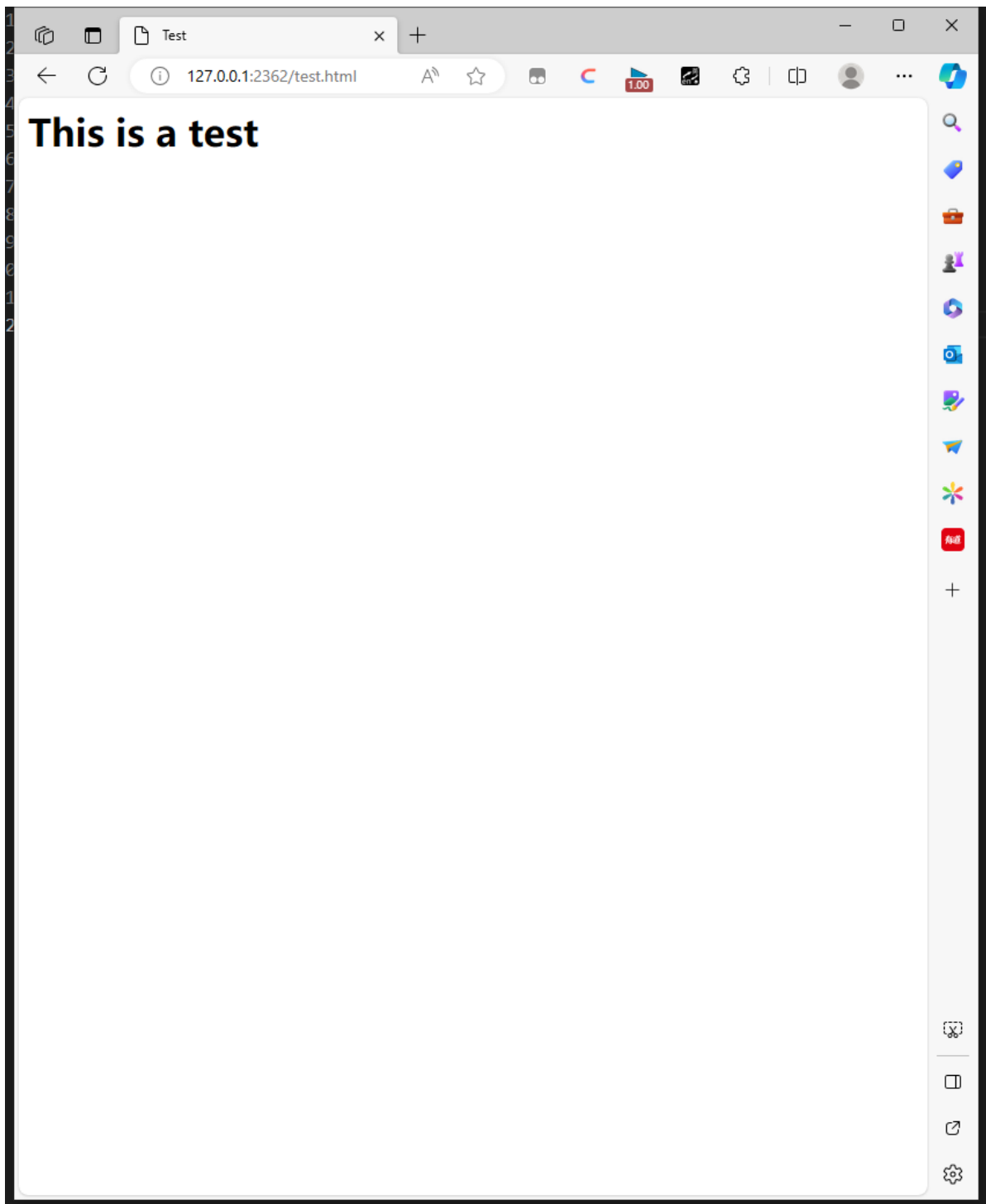


浏览器访问只包含文本的HTML文件时，浏览器的URL地址和显示内容截图。

首先我们先修改一下HTML文件

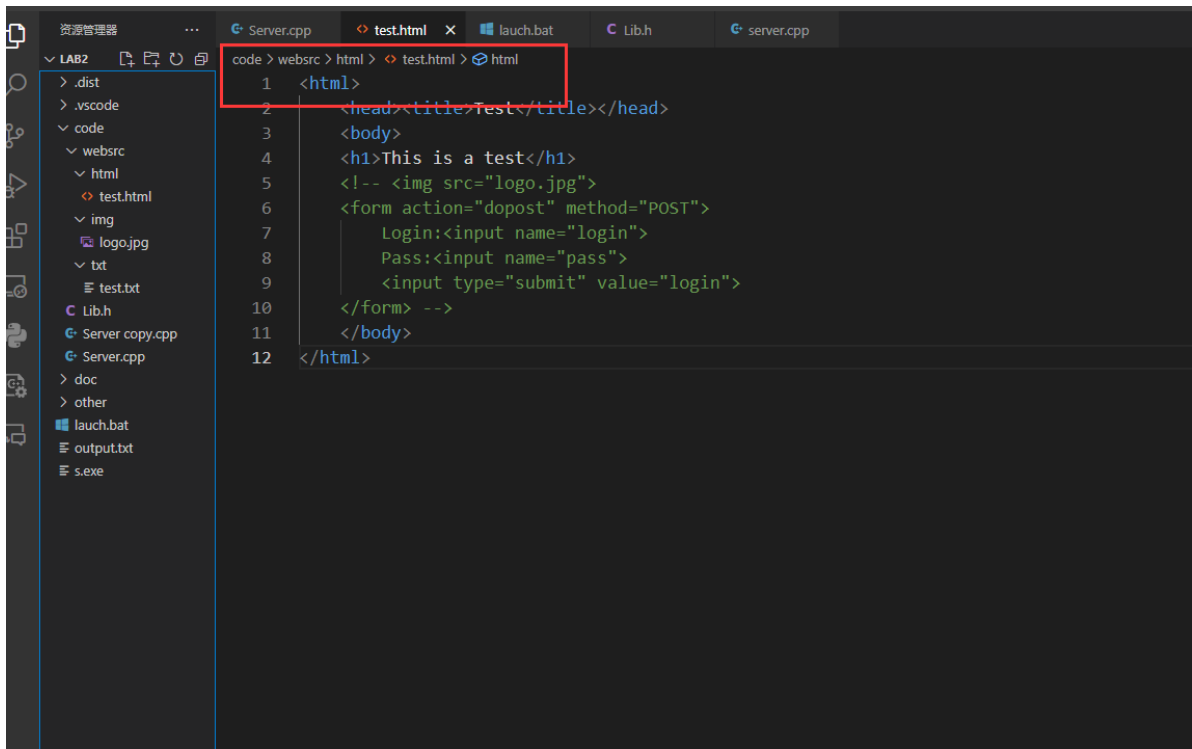
```
7 website / html / > test.html / > html
1 <html>
2   <head><title>Test</title></head>
3   <body>
4     <h1>This is a test</h1>
5     <!-- 
6     <form action="dopost" method="POST">
7       Login:<input name="login">
8       Pass:<input name="pass">
9       <input type="submit" value="login">
10    </form> -->
11  </body>
12</html>
```

URL = [Test](#)



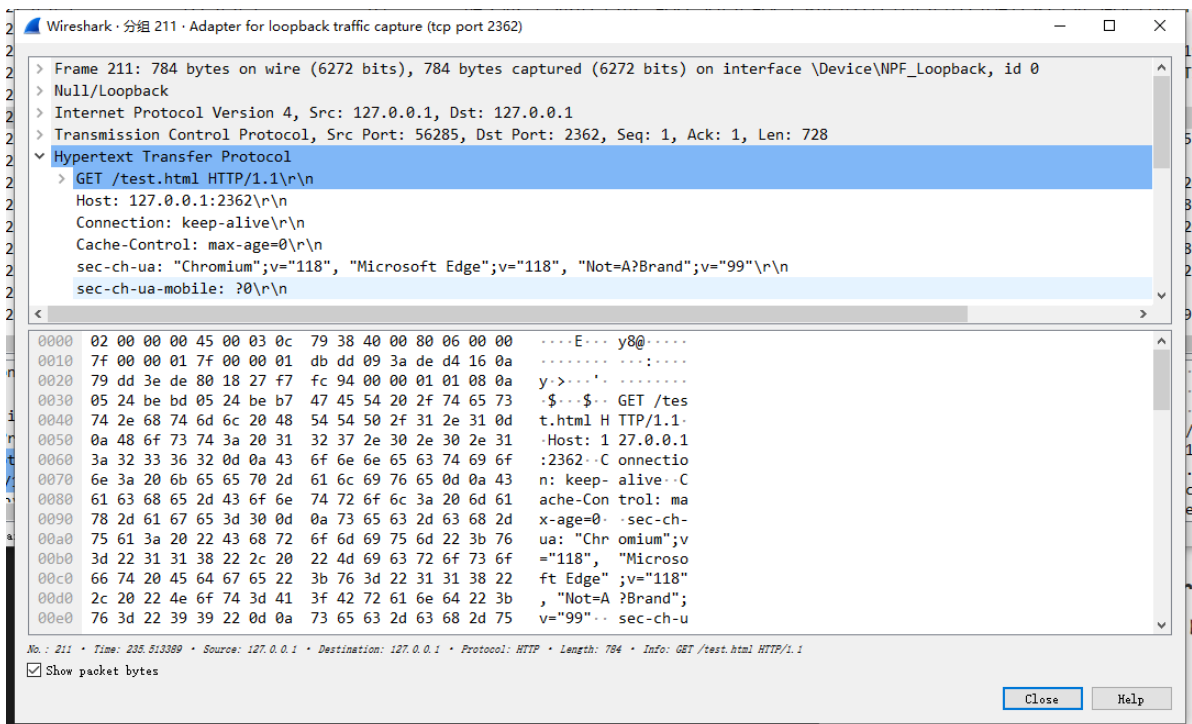
服务器文件实际存放的路径：

F:\lessons\ComputerNetwork\lab\lab2\code\websrc\html\test.html



## Wireshark抓取的数据包截图 HTTP协议部分


浏览器访问该URL,此时发送一个GET请求

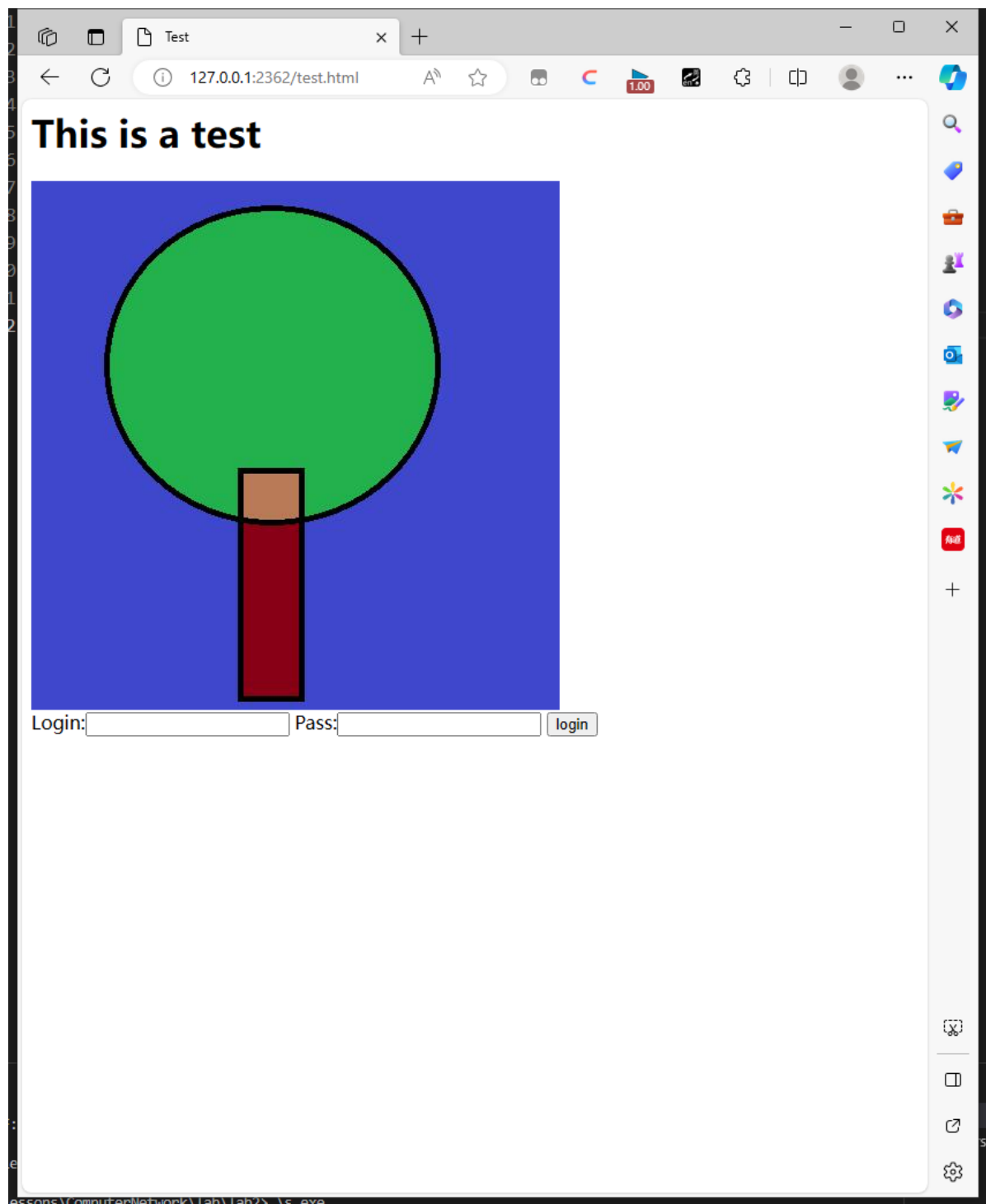


随机服务器发回响应包,其中包含一个HTML文件

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。

```
1 <html>
2   <head><title>Test</title></head>
3   <body>
4     <h1>This is a test</h1>
5     
6     <form action="dopost" method="POST">
7       Login:<input name="login">
8       Pass:<input name="pass">
9       <input type="submit" value="login">
10    </form>
11  </body>
12 </html>
```

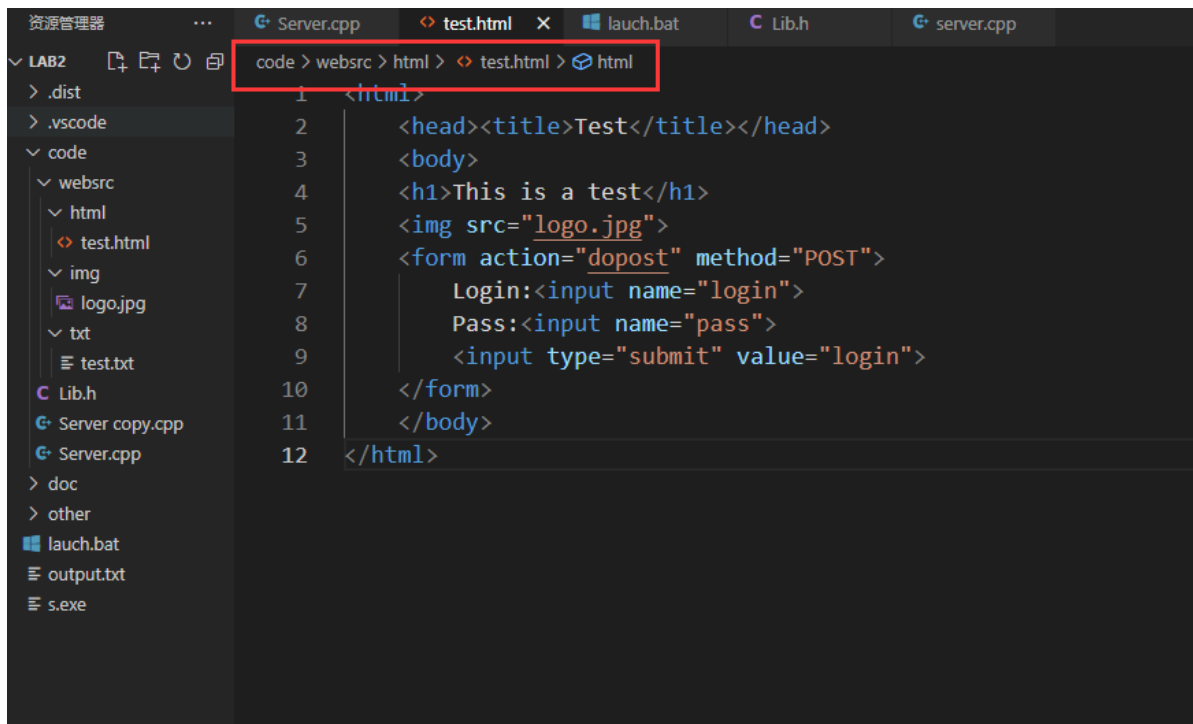
 127.0.0.1:2362/test.html



服务器上文件实际存放的路径：

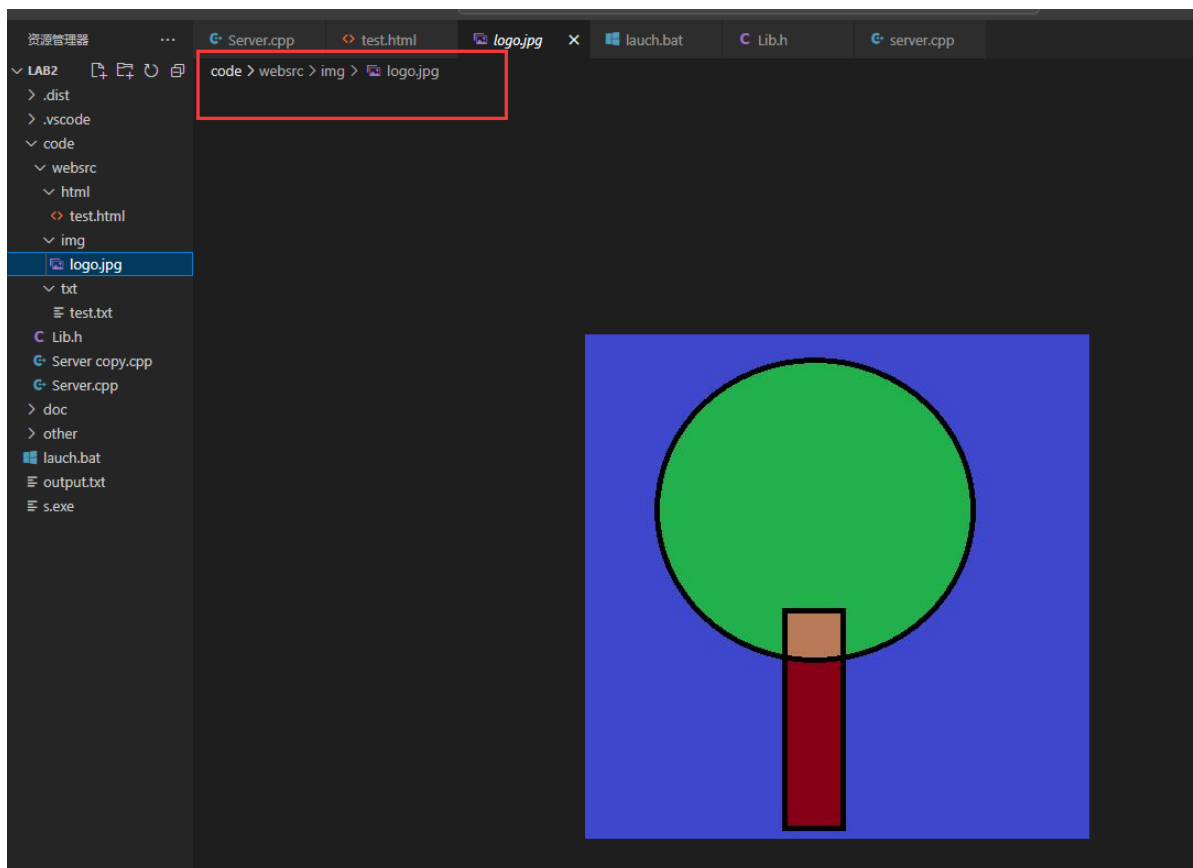
html的路径F:\lessons\ComputerNetwork\lab\lab2\code\websrc\html\test.html





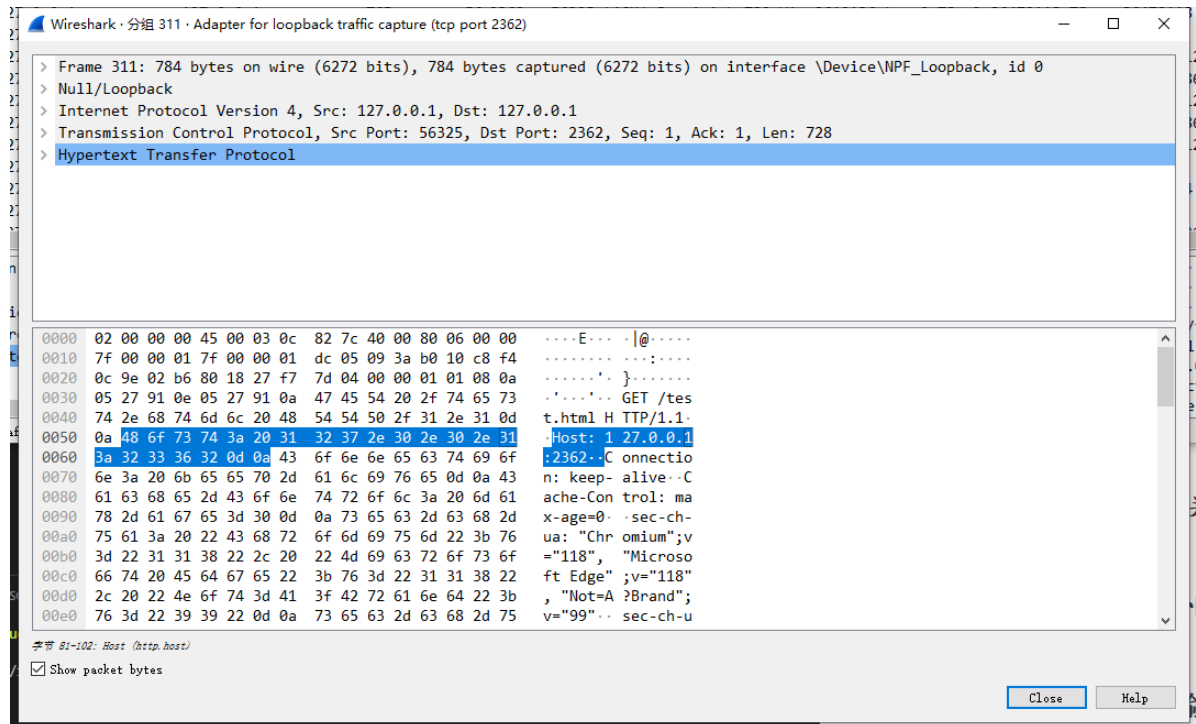
img的路径

F:\lessons\ComputerNetwork\lab\lab2\code\websrc\img\logo.jpg

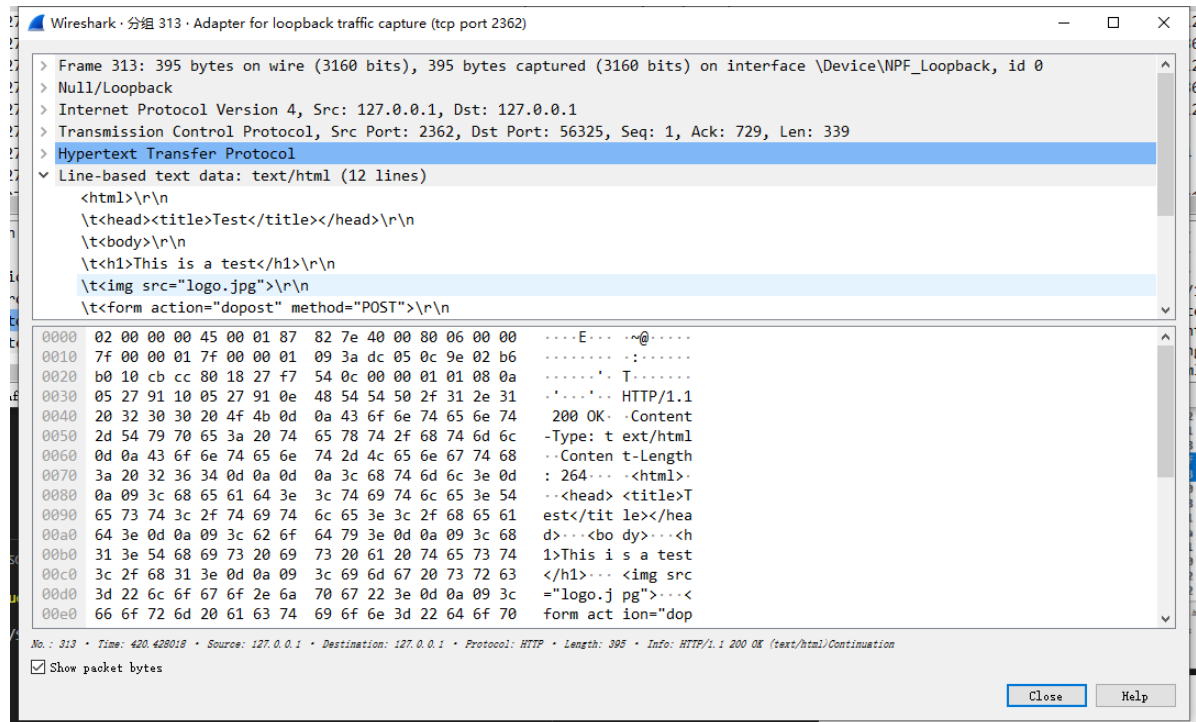


## Wireshark抓取的数据包截图（只截取HTTP协议部分，包括HTML、图片文件的部分内容）

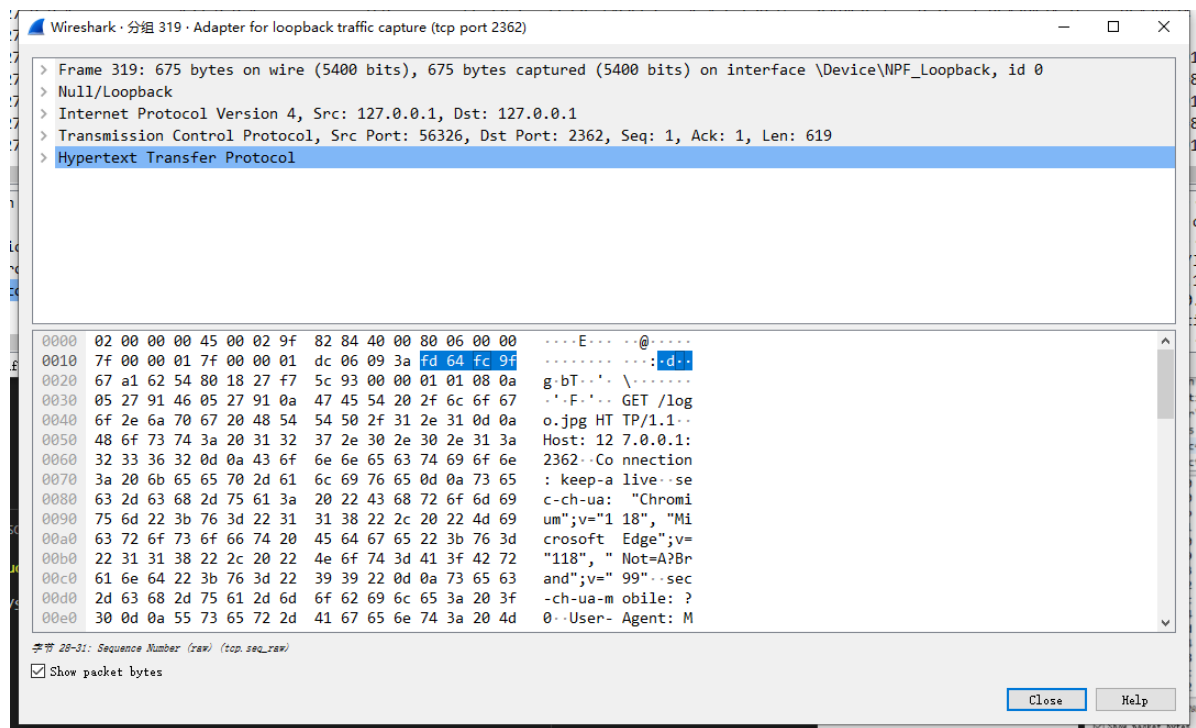
浏览器访问服务器,发送一个GET请求,请求内容是HTML文件



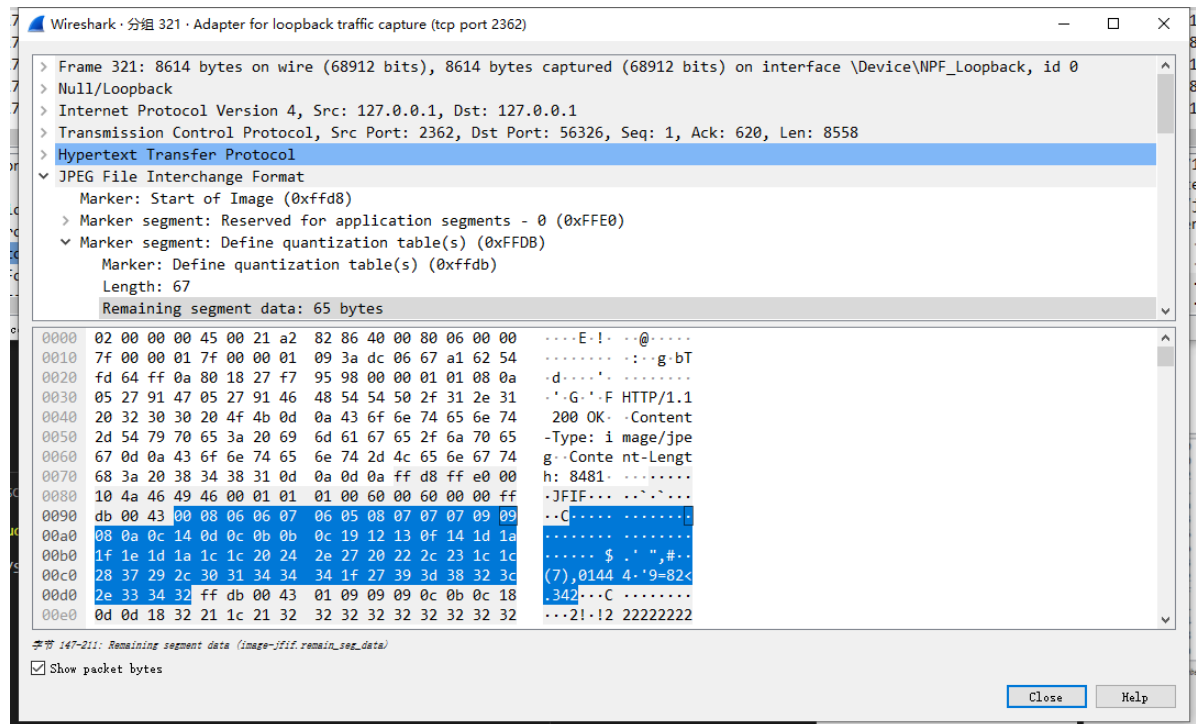
接下来服务器先相应HTML请求,发回一个相应包



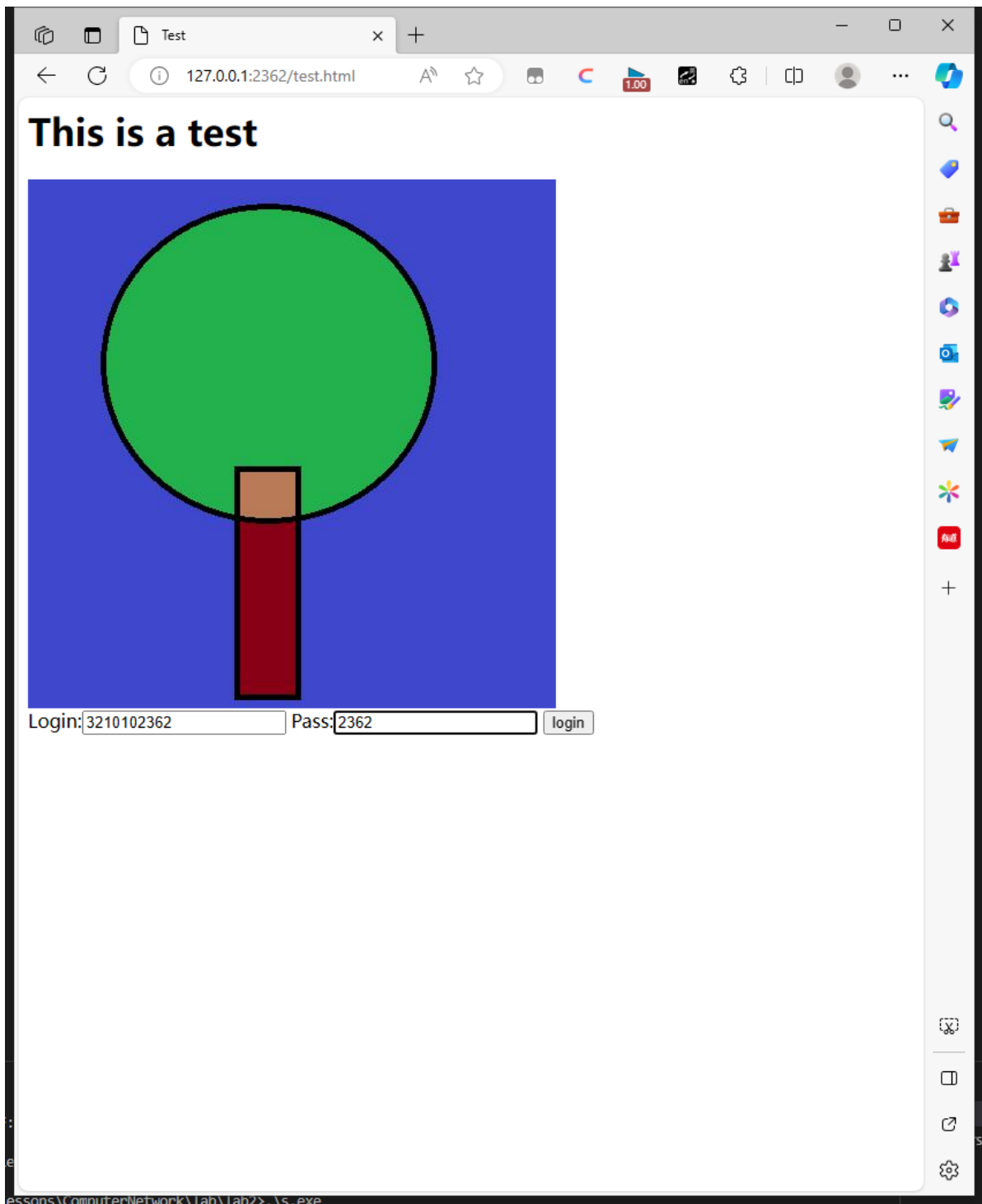
紧接着浏览器发出了一个GET请求,内容是索要logo.img

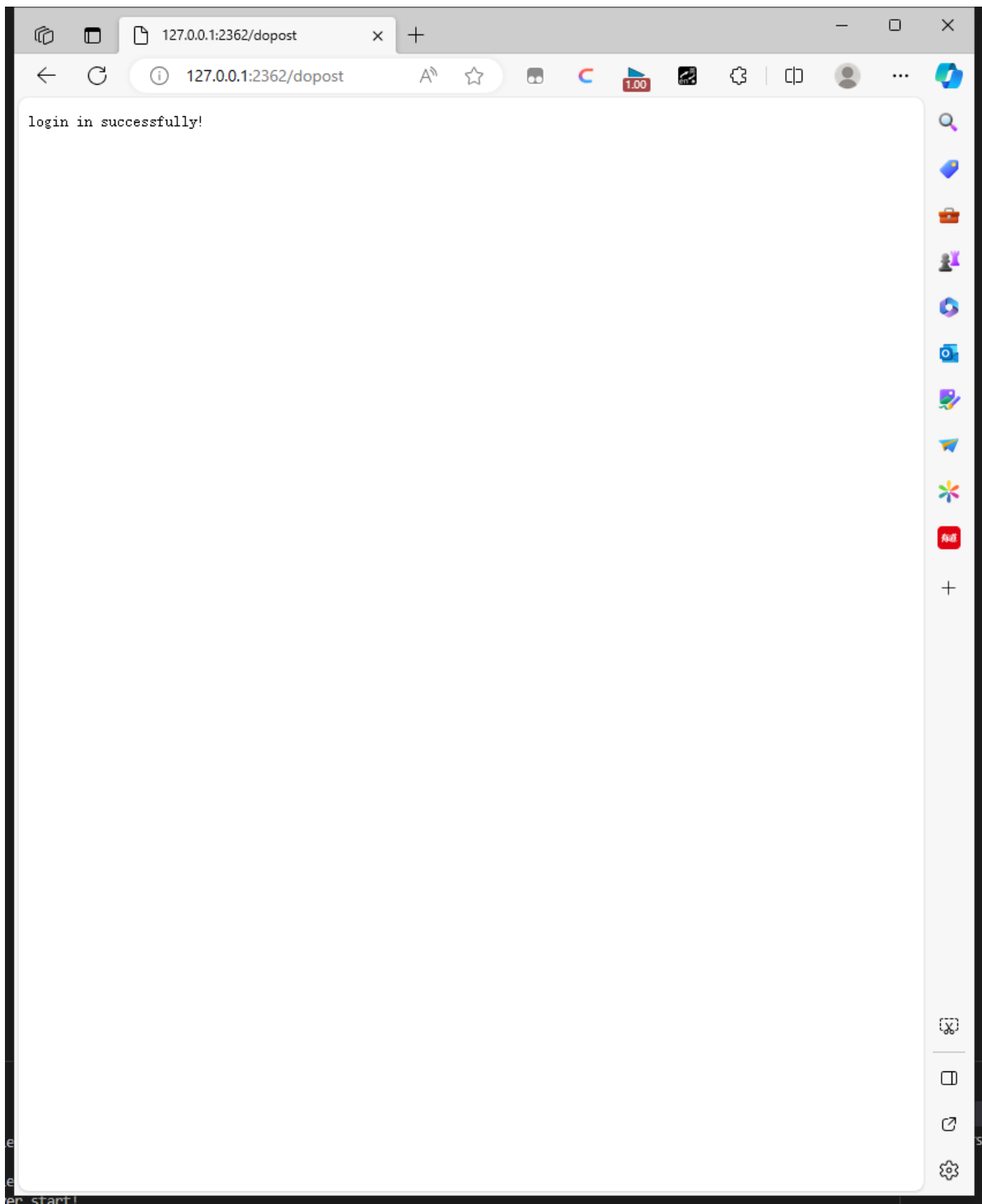


然后服务器回传该图片



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：

这里需要处理POST请求

首先需要解析包头,查询请求的方法是POST,且请求的URL是dopost

```

void parse_request_header(SOCKET clientSocket, char * request) {

    std::istringstream request_stream(request);
    std::string request_line;

    // Read the first line of the request, which contains the request method, URL, and HTTP version
    std::getline(request_stream, request_line);

    // Parse the request line
    std::string request_method;
    std::string request_url;
    std::string http_version;
    std::istringstream request_line_stream(request_line);
    request_line_stream >> request_method >> request_url >> http_version;

    // Output the parsed information
    std::cout << "Request Method: " << request_method << std::endl;
    std::cout << "Request URL: " << request_url << std::endl;
    std::cout << "HTTP Version: " << http_version << std::endl;

    if(request_method == "GET"){
        GEThandle(request_url,clientSocket);
    }else if(request_method == "POST"){
        POSThandle(request_url,clientSocket,request);
    }
}

```

然后交给POSThandle函数,首先做的是查询是不是dopost,否则发送一个400 Bad Request

如果是Dopost,则进行解析,查找到连续的两个回车换行符,后面就是正文

由于正文是Content-Type: application/x-www-form-urlencoded

所以可以直接按照键值对的方法进行解析,这里开了一个String数组,为formdata,首先对正文进行分割,然后以此读出键值对,这里只会有两个键值对,一个是login=XXX 一个是pass=XXX

这么写的可拓展性比较好,相较于直接进行字符串匹配,而且效率也会更高

```

18
19 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
20
21 Origin: http://127.0.0.1:2362
22
23 Content-Type: application/x-www-form-urlencoded
24
25 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,app
26
27 Sec-Fetch-Site: same-origin
28
29 Sec-Fetch-Mode: navigate
30
31 Sec-Fetch-User: ?1
32
33 Sec-Fetch-Dest: document
34
35 Referer: http://127.0.0.1:2362/test.html
36
37 Accept-Encoding: gzip, deflate, br
38
39 Accept-Language: zh,en;q=0.9,en-US;q=0.8,en-GB;q=0.7
40
41
42 login=321&pass=321
43

```

```

}
void POSThandle(const std::string& request_url, SOCKET clientSocket, char *p) {
    bool isDopost = (request_url == "/dopost");

    string recvBuffer(p);

    // cout << recvBuffer ;
    if (isDopost) {
        // 查找请求体结束标志 "\r\n\r\n" 的位置
        std::size_t bodyEndPos = recvBuffer.find("\r\n\r\n");
        if (bodyEndPos != std::string::npos) {
            // 从请求体结束标志位置开始解析表单字段
            std::string contentBody = recvBuffer.substr(bodyEndPos + 4);
            // cout << contentBody ;
            // 使用 & 符号分割表单字段
            std::vector<std::string> formData;
            std::vector<std::string> Values;
            size_t startPos = 0;
            while (startPos < contentBody.size()) {
                size_t endPos = contentBody.find("&", startPos);
                if (endPos == std::string::npos) {
                    endPos = contentBody.size();
                }
                formData.push_back(contentBody.substr(startPos, endPos - startPos));
                startPos = endPos + 1;
            }

            // 解析表单字段的键和值
            for (const std::string& field : formData) {
                size_t equalsPos = field.find("=");
                if (equalsPos != std::string::npos) {
                    std::string key = field.substr(0, equalsPos);
                    std::string value = field.substr(equalsPos + 1);
                    Values.push_back(value);
                }
            }
        }
    }
}

```

```

// 解析表单字段的键和值
for (const std::string& field : formData) {
    size_t equalsPos = field.find("=");
    if (equalsPos != std::string::npos) {
        std::string key = field.substr(0, equalsPos);
        std::string value = field.substr(equalsPos + 1);
        Values.push_back(value);
    }
}

if(Values[0] == "3210102362" && Values[1] == "2362"){
    const char* response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 10\r\n";
    send(clientSocket, response, strlen(response), 0);
}else{
    const char* response = "HTTP/1.1 200 OK\r\nContent-Type: text/plain\r\nContent-Length: 10\r\n";
    send(clientSocket, response, strlen(response), 0);
}
} else {
    // 未找到请求体结束标志, 处理错误
    const char* errorResponse = "HTTP/1.1 400 Bad Request\r\nContent-Type: text/plain\r\nContent-Length: 10\r\n";
    send(clientSocket, errorResponse, strlen(errorResponse), 0);
}
} else {
    // 处理其他情况或错误
    const char* errorResponse = "HTTP/1.1 400 Bad Request\r\nContent-Type: text/plain\r\nContent-Length: 10\r\n";
    send(clientSocket, errorResponse, strlen(errorResponse), 0);
}
}
}

```

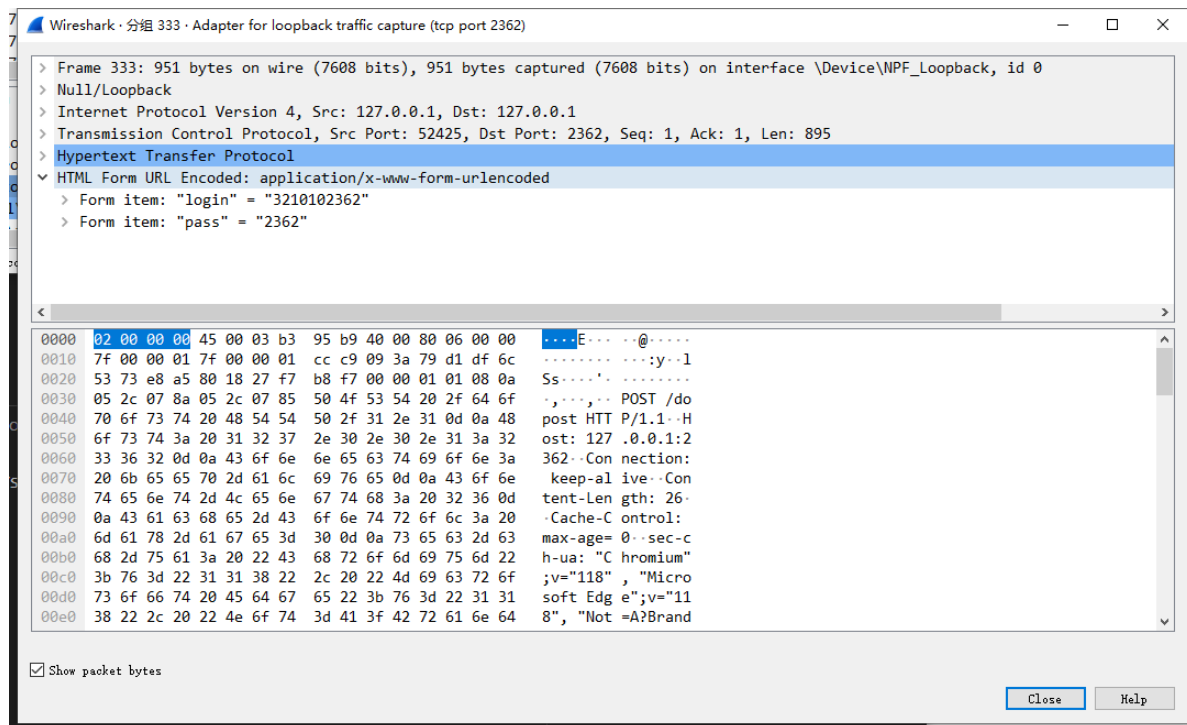
如果密码和账户正确,就发送一个登陆成功的包

否则发送一个登录失败的包

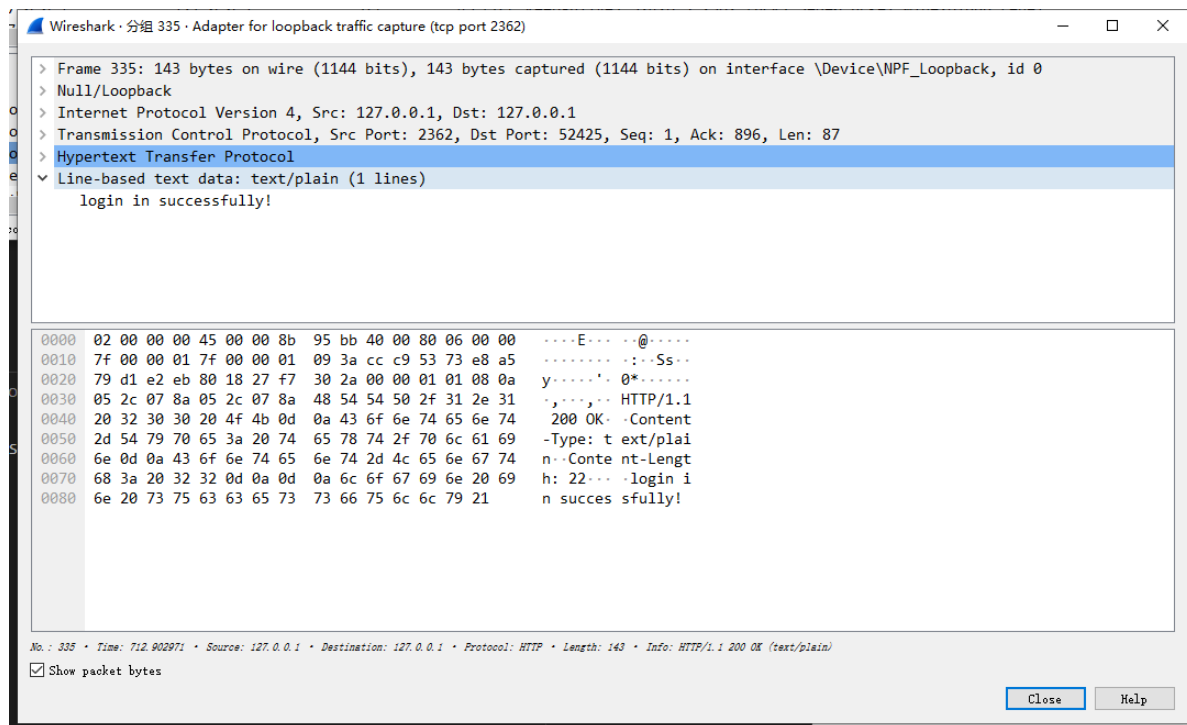
其他情况,比如没有找到请求结束标志,或者是其他类型非Dopost的包都发送400 Bad Request

## Wireshark抓取的数据包截图（HTTP协议部分）

点击login之后,浏览器发送了一个POST请求,url部分表示是dopost,携带信息有两个,一个是login账户信息,一个是pass密码值



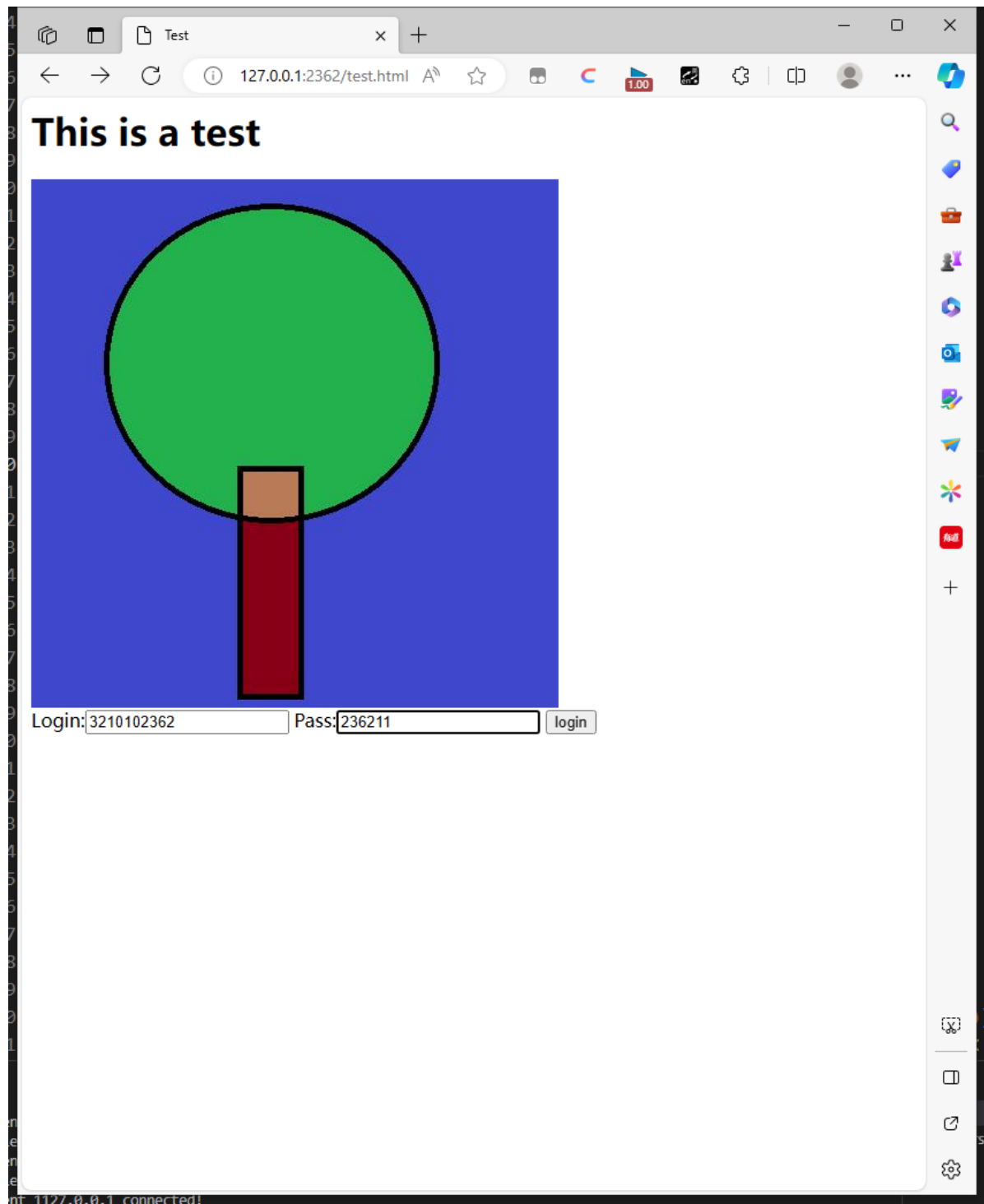
随机服务器收到请求,做出判断,发回一个登陆成功信息包

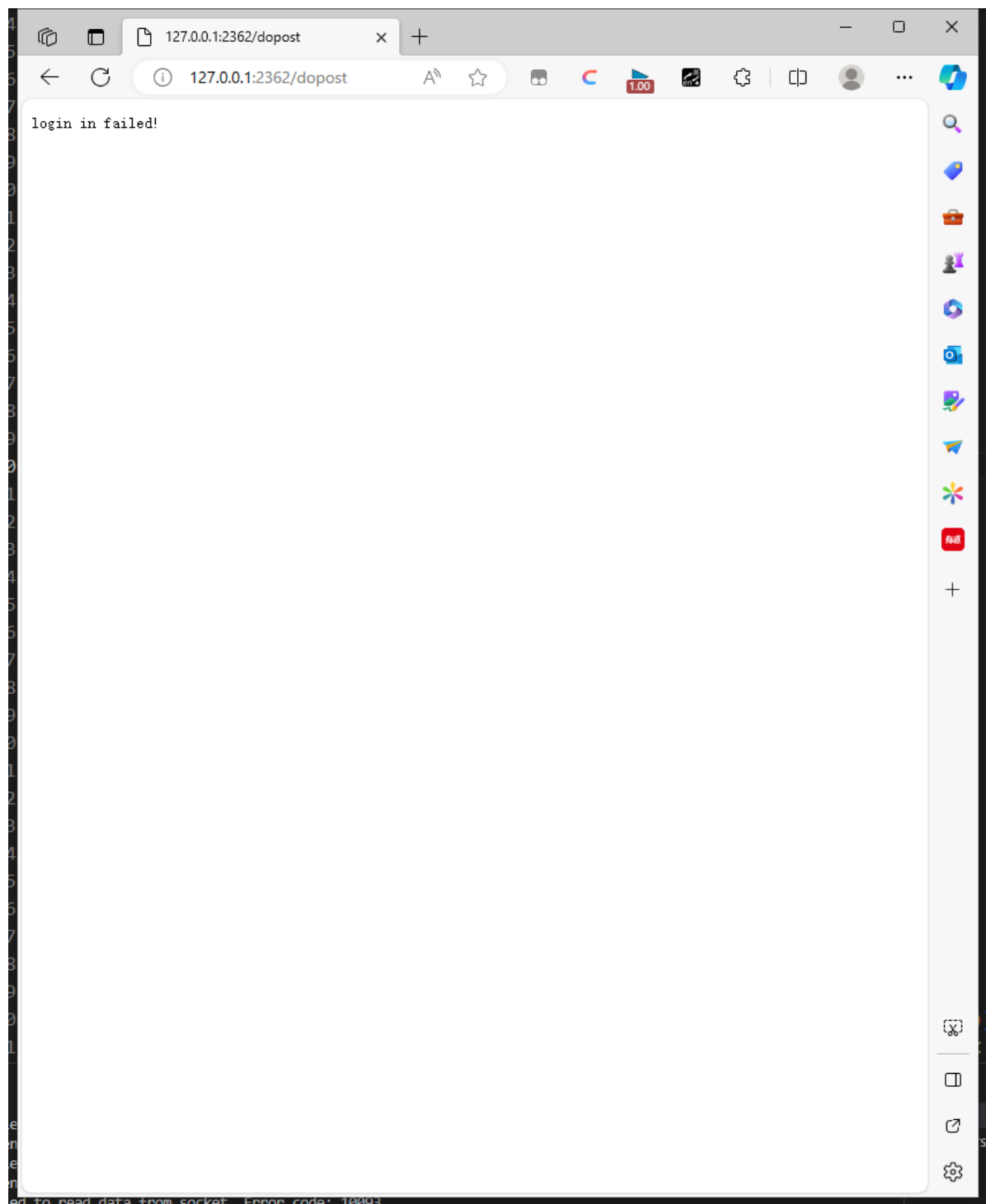


浏览器输入错误的登录名或密码, 点击登录按钮（login）后的显示截图。



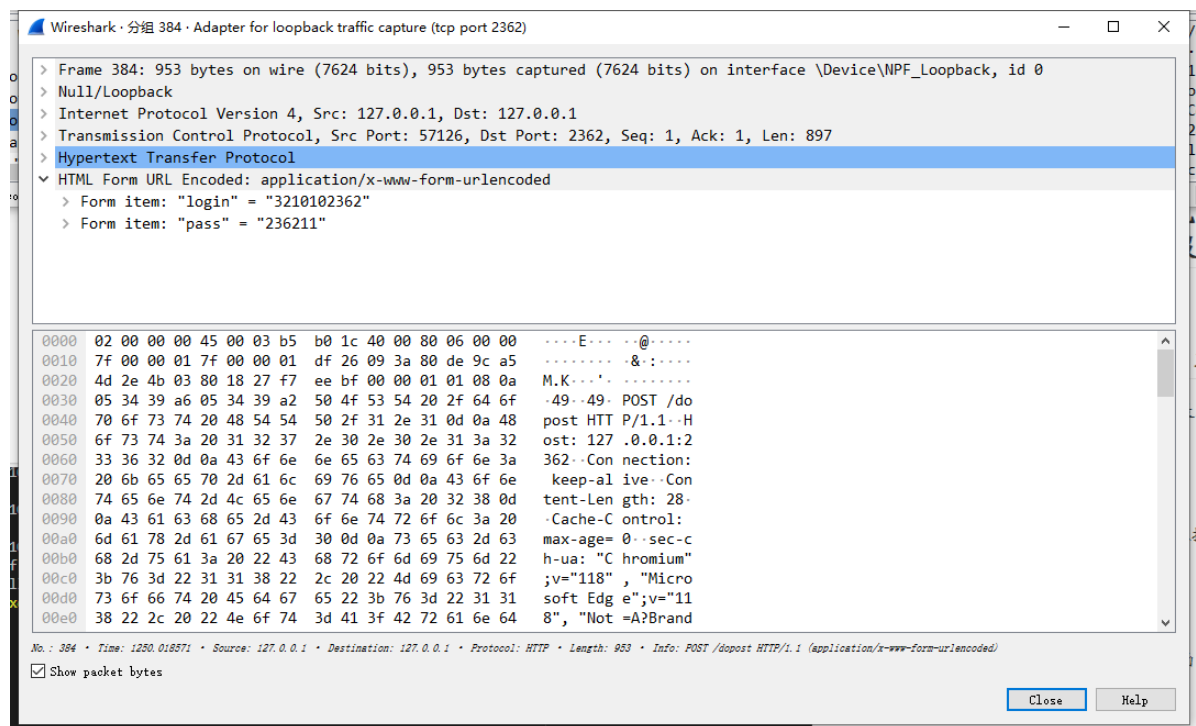
输错密码



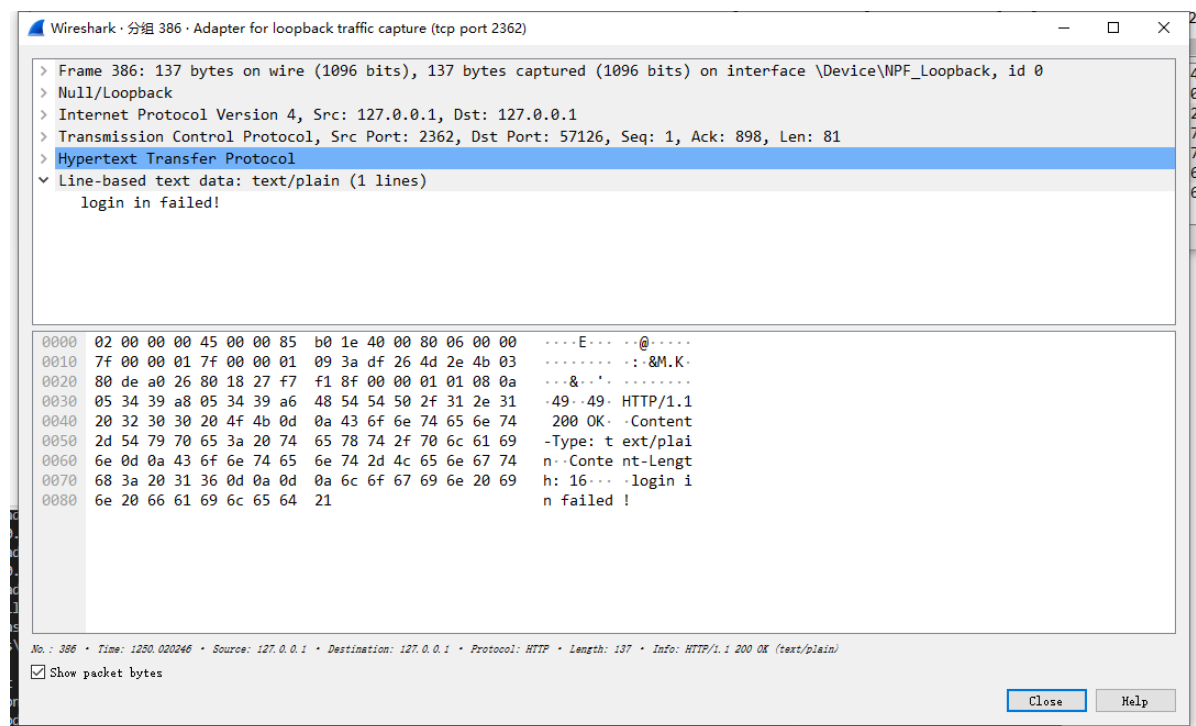


## Wireshark抓取的数据包截图（HTTP协议部分）

发送了一个dopost请求, 可以看到密码是错的

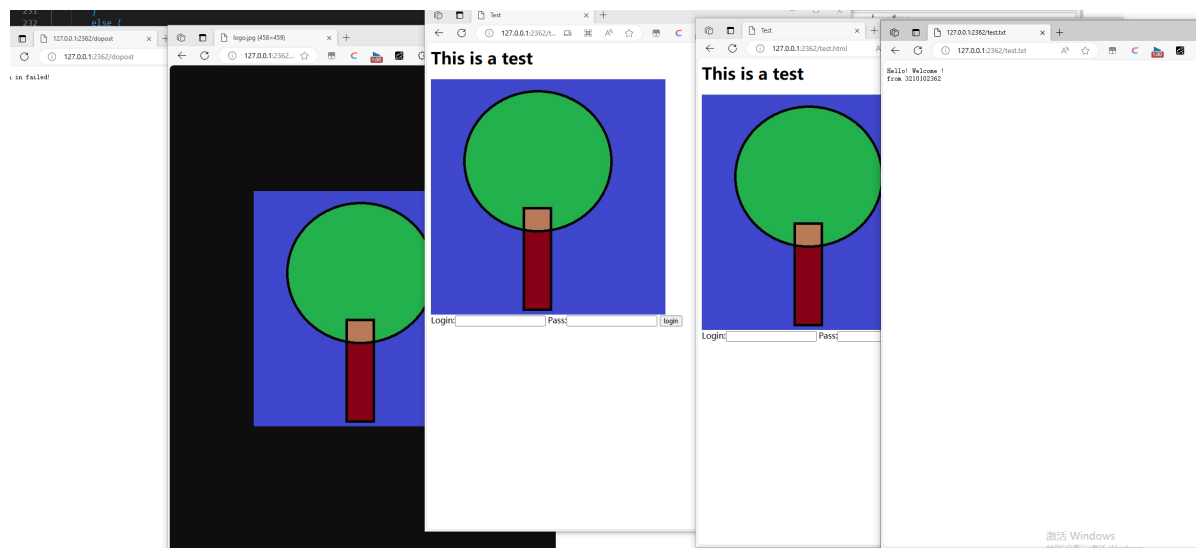


随后服务器回传一个包含登录失败信息的响应包



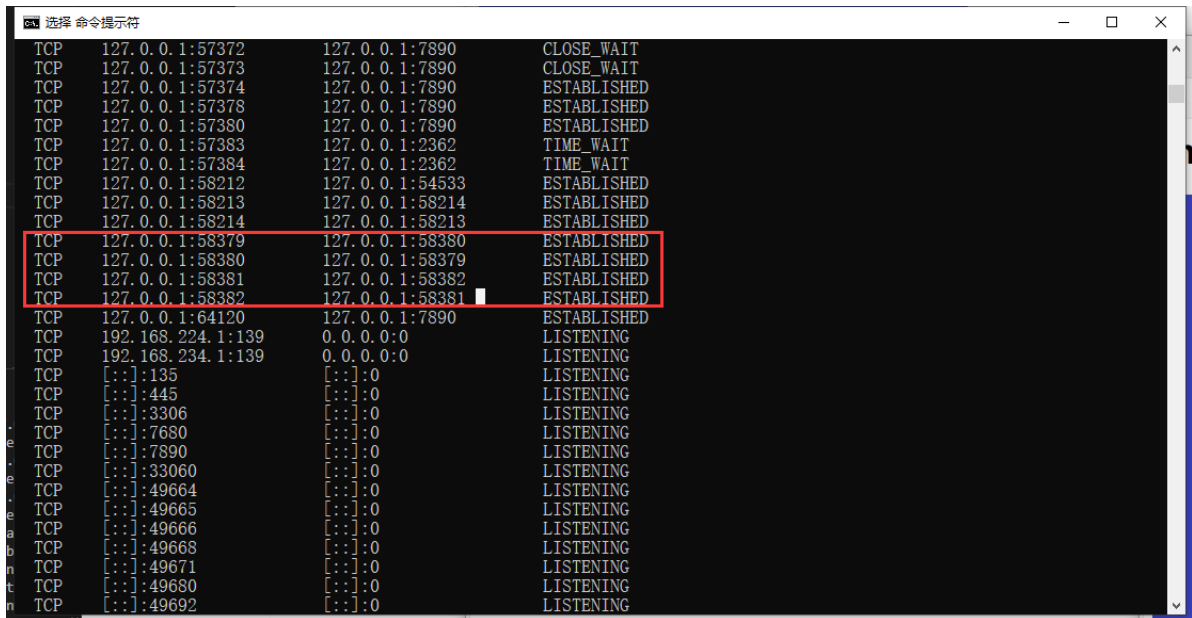
多个浏览器同时访问包含图片的HTML文件时，浏览器的显示内容截图（将浏览器窗口缩小并列）

同时打开多个窗口



```
PS F:\lessons\ComputerNetwork\lab\lab2> .\s.exe
Server start!
client client 1127.0.0.1 connected!
0127.0.0.1 connected!
Request Method: POST
Request URL: /dopost
HTTP Version: HTTP/1.1
client 1127.0.0.1 connected!
Request Method: GET
Request URL: /logo.jpg
HTTP Version: HTTP/1.1
client 0127.0.0.1 connected!
Request Method: GET
Request URL: /test.html
HTTP Version: HTTP/1.1
Request Method: GET
Request URL: /logo.jpg
HTTP Version: HTTP/1.1
client 0127.0.0.1 connected!
client 1127.0.0.1 connected!
Request Method: GET
Request URL: /test.txt
HTTP Version: HTTP/1.1
client 1127.0.0.1 connected!
Request Method: GET
Request URL: /test.html
HTTP Version: HTTP/1.1
Request Method: GET
Request URL: /logo.jpg
HTTP Version: HTTP/1.1
```

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）



## 实验结果与分析

- HTTP协议是怎样对头部和体部进行分隔的？

使用一个回车换行符"`\r\n`"对头部和体部进行分割

所以当我们检查连续的"`\r\n\r\n`"便可以找到头部和体部的交界处

HTTP协议使用一个空白行（CRLF，即回车符"`\r`"和换行符"`\n`"的组合）来这个空白行标志着头部的结束和消息体的开始。

HTTP请求和响应的基本结构如下：

### 1. 请求消息：

```
makefileCopy code请求行
头部字段1： 值1
头部字段2： 值2
...
头部字段N： 值N
(空白行)
请求体
```

### 2. 响应消息：

```
makefileCopy code状态行
头部字段1： 值1
头部字段2： 值2
...
头部字段N： 值N
(空白行)
响应体
```

在请求消息中，"请求行"包含了请求的方法（GET、POST、等）、URL和HTTP协议版本。

- 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？

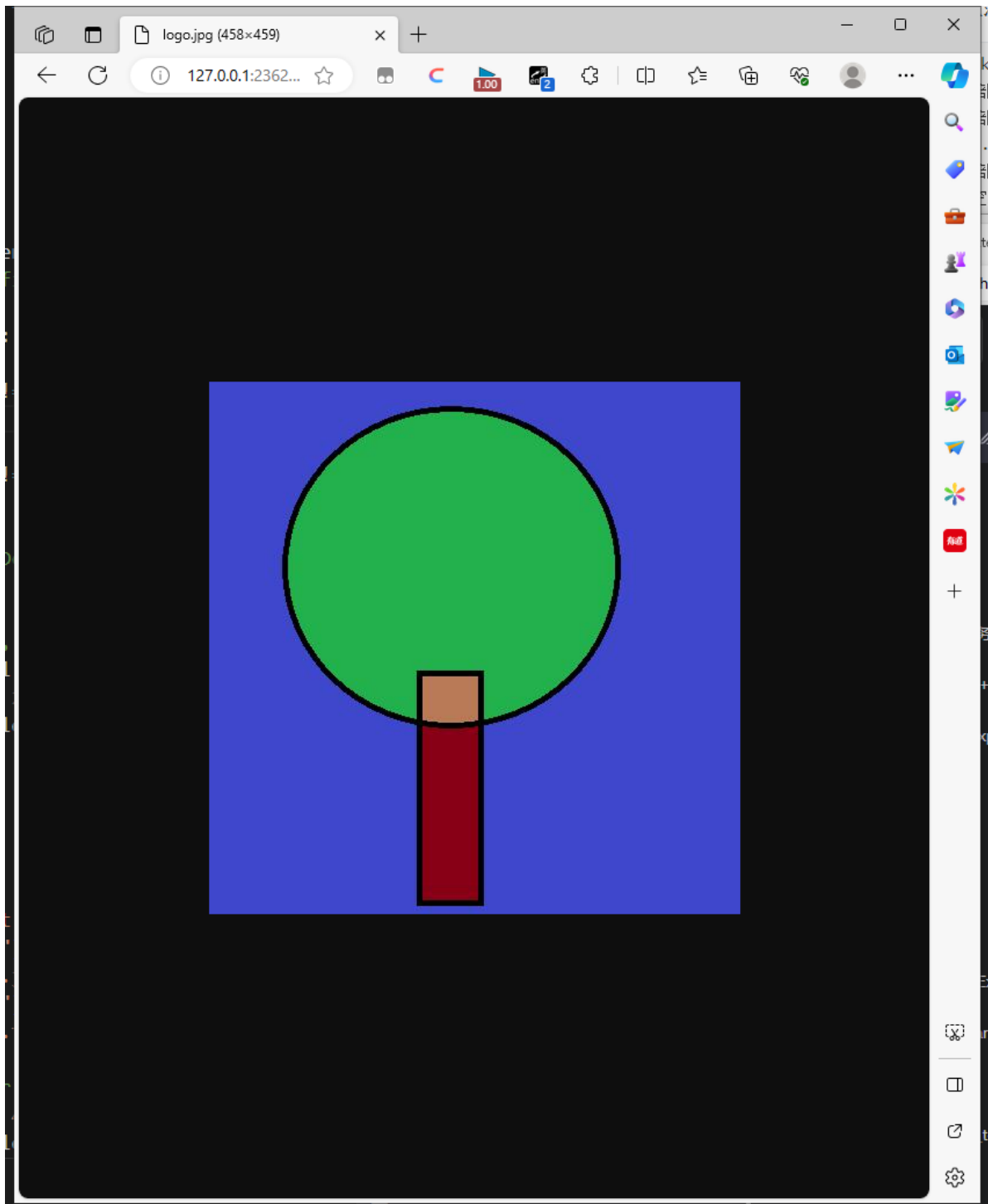
1. 文件扩展名： 浏览器会查看URL中文件的扩展名（例如 `.html`、`.jpg`、`.pdf` 等）来猜测文件类型。这是一种常见的方法，尤其在没有其他信息可用的情况下。例如，如果URL以 `.html` 结尾，浏

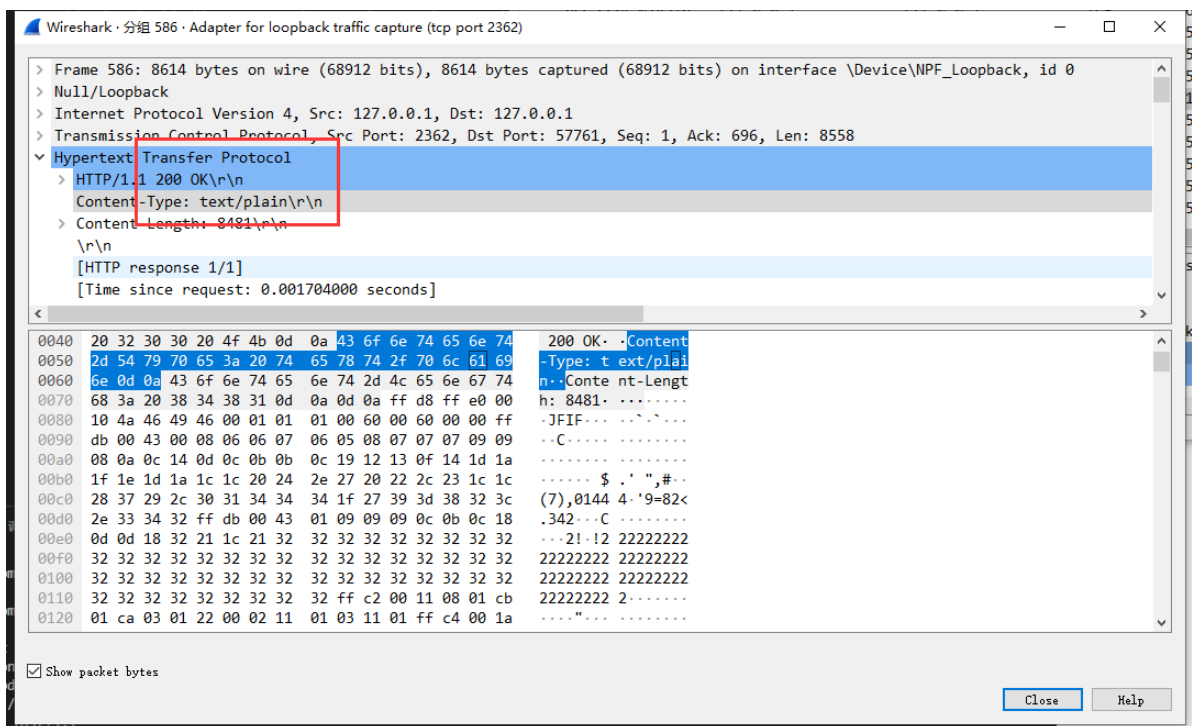
浏览器通常会认为它是HTML网页。

2. **HTTP头部字段：** HTTP响应头部包括一个名为"Content-Type"的字段，它指定了响应中的内容类型（MIME类型）。浏览器会检查这个字段，以确保正确解释和显示响应的内容。如果服务器正确设置了"Content-Type"字段，浏览器会优先使用这个信息来确定文件类型。

对代码进行修改,更改Content-Type

```
content_type = "text/html";  
} else if (request_url.find(".jpg") != std::string::npos) {  
    content_type = "text/plain";  
    // content_type = "image/jpeg";  
} else if (request_url.find(".txt") != std::string::npos) {  
    content_type = "text/plain";  
}
```

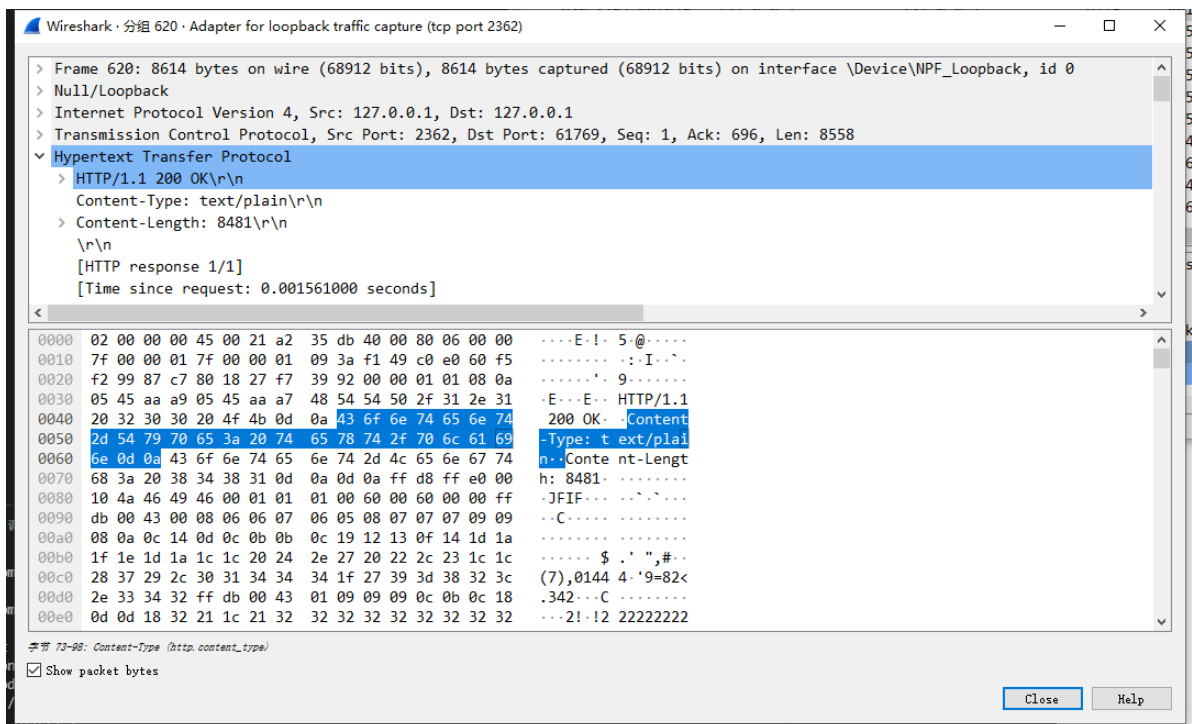
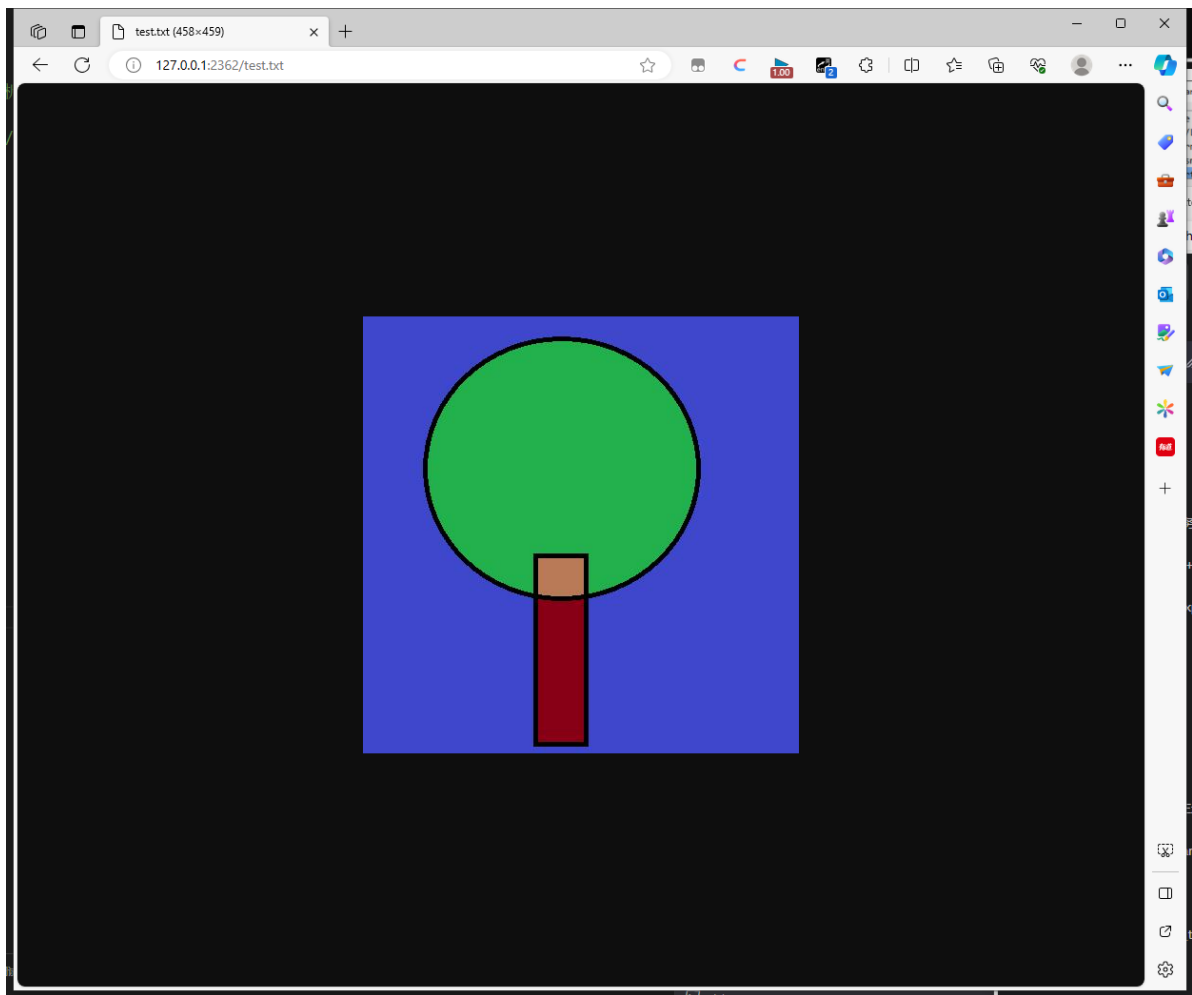




可以看到即便ContentType是错的,还是能够正确显示图片

下面我们改一下扩展名,这里我们发送对应文本的请求,传递一个图片,此时渲染类型是text/plain,但是内容其实是一个图片

```
// Prepare the file path
std::string filePath;
if (request_url == "/test.html") {
    filePath = "code/websrc/html/test.html";
} else if (request_url == "/logo.jpg") {
    // filePath = "code/websrc/txt/test.txt";
    filePath = "code/websrc/img/logo.jpg";
} else if (request_url == "/test.txt") {
    filePath = "code/websrc/img/logo.jpg";
    // filePath = "code/websrc/txt/test.txt";
} else {
```



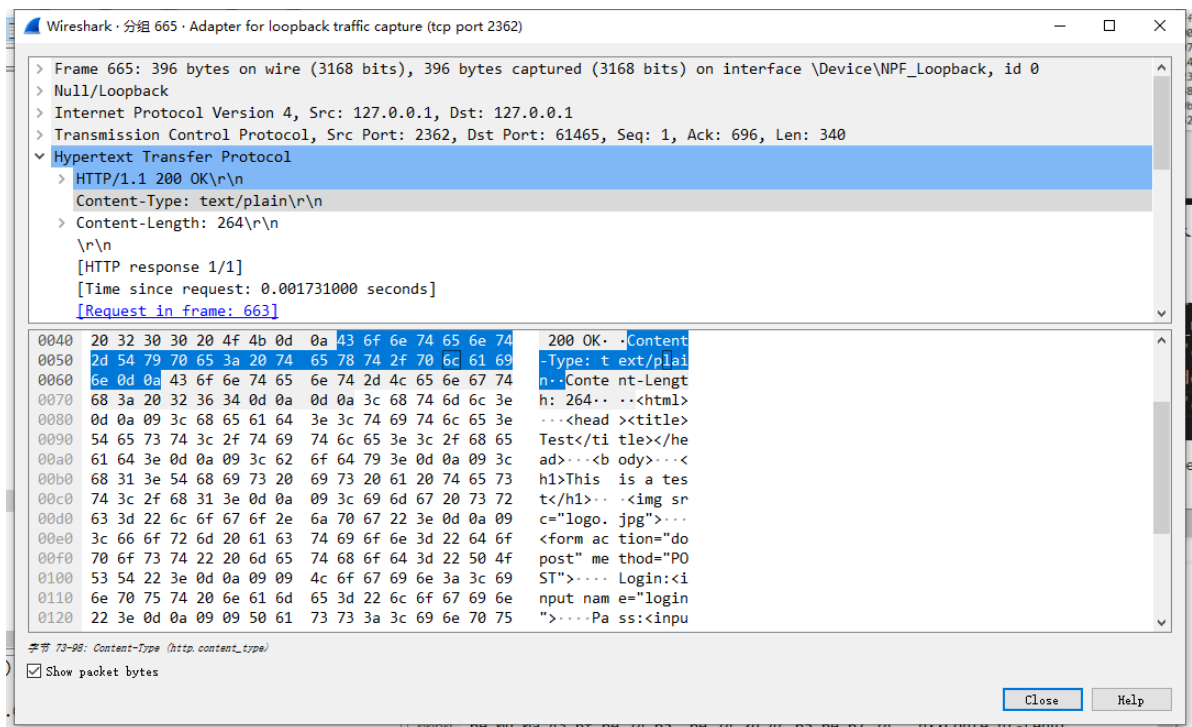
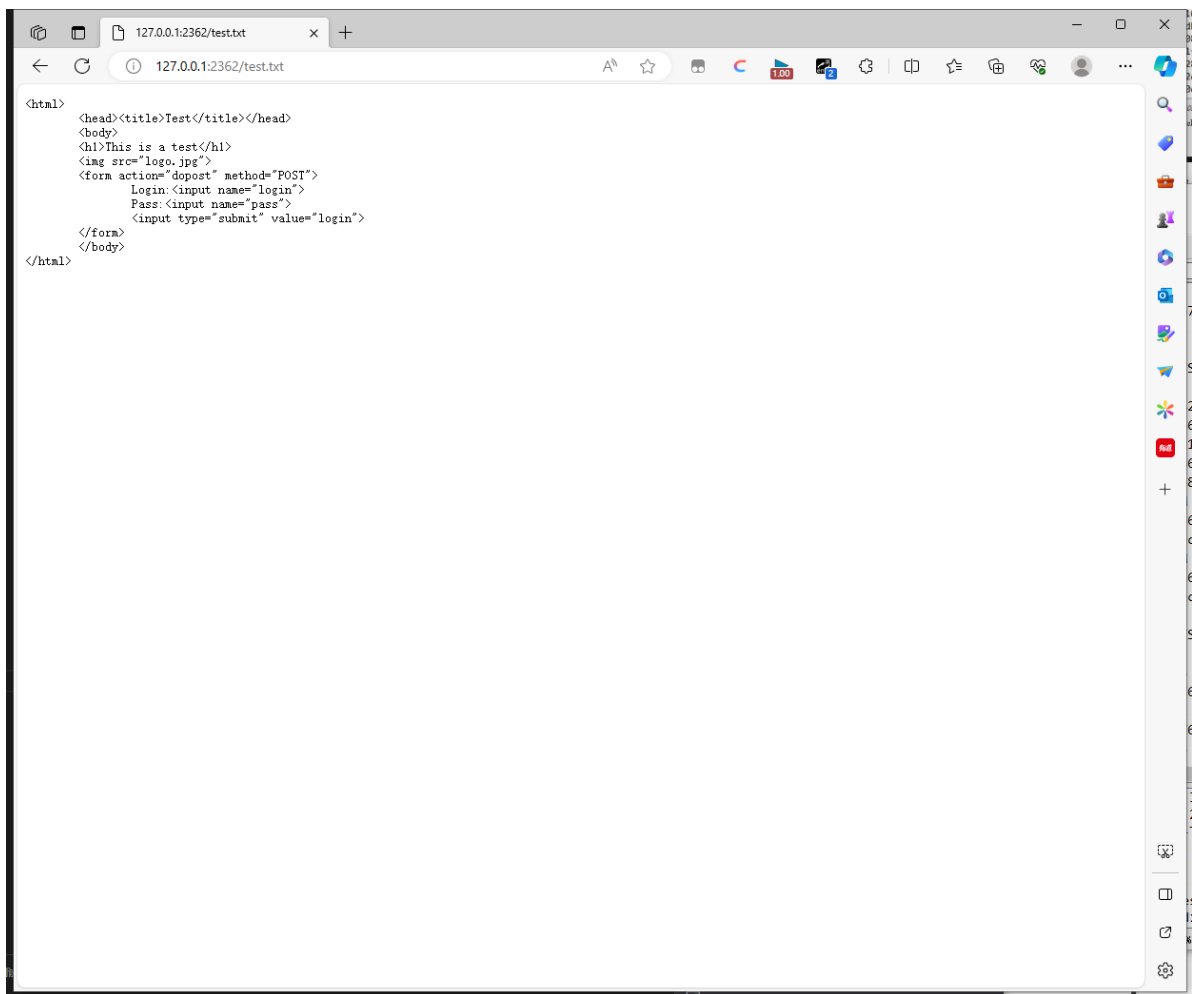
可以看到即使是真的传了一个图片作为文本,还是被解释为了图片,并且成功的渲染了出来



```

} else if (request_url == "/test.txt") {
    // filePath = "code/websrc/img/logo.jpg";
    filePath = "code/websrc/html/test.html";
    // filePath = "code/websrc/txt/test.txt";
} else {

```



这里我们传了一个html文件,但是把ContentType设置为text/palin,可以看到浏览器并没有真的对其进行相应的渲染,而是直接作为文本内容显示了出来

由此可见HTTP是同时通过ContentType和拓展名来保证显示内容的正确性的

- HTTP协议的头部是不是一定是文本格式？体部呢？

协议头部是文本格式

体部通常是文本格式,比如HTML文件和test文件,当然也可以直接传输二进制数据,或者进行压缩和加密等再编码,比如传递图像

- 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？

POST方法传递的数据放在体部,需要靠"\r\n\r\n"查到后从后面读取

两个字段用&连接

## 七 讨论 心得

本次实验相比于上一个实验更近一步,而且非常有趣,我终于搞懂了日常的上网操作究竟是怎么实现的,实际去拆包,解析GET和POST请求的过程出了不少差错,比如:

文本乱码:



这个是因为传递的文本使用了UTF8编码,而传递过程中没有显式的表明,或者在包头做声明,考虑到同步客户端和服务端的编码方式比较麻烦,于是后面的内容通信都是用英文了.

打开文件失败,这里是服务器无法访问img src

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

# This is a test



Login:  Pass:

事实上是先传HTML文件,然后进行二次渲染,二次渲染的时候,服务器会根据img src后面的内容作为url再请求一次,所以这个路径不是相对于服务器的真实路径,而是一个url地址,需要解析的,所以我改成实际的url就可以了。

接受缓冲区不够大 导致拆包错误.这个是我后面才发现的问题,本来以为是需要用char\* 而不能封装给String

,实际上是在栈区开不了太大的缓冲区,而需要使用malloc申请.而且一定要注意处理String的时候不要设置为读到换行符就结束.多去把实际String存储的内容打印出来,然后再做相应的检查.

本次实验的收获还是挺多的,再次复习了一下基于Socket编程的内容,同时总算了解了实际HTTP协议的通信过程,对GET和POST拆包算是实在的掌握了,每一个包头都是自己拿着String一点点拼出来的