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：　徐铭

学　　院：　计算机学院

系：

专　　业：　计算机科学与技术

学　　号：　3210102037

指导教师：　陆魁军

2023 年　12　月　22　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）

```
void Start() {
    /* Process the clients */
    while(!exit_requested) {
        /* Create the Client Socket */
        sockaddr_in clientAddress{};
        int clientAddrLen = sizeof(clientAddress);
        SOCKET clientSocket = accept(serverSocket, reinterpret_cast<struct sockaddr*>(&clientAddress), &clientAddrLen);
        if (clientSocket == INVALID_SOCKET) {
            perror("Error accepting connection");
            continue;
        }
        CreateThread(NULL, 0, ThreadHandle, reinterpret_cast<LPVOID>(&clientSocket), 0, nullptr);
    }
    if (serverSocket != INVALID_SOCKET) closesocket(serverSocket);
    WSACleanup();
}
```

上面的代码包含一个无限循环，负责处理客户端连接。在循环内部，我使用了 accept 函数接受客户端连接，并在成功时创建一个新的套接字(clientSocket)以处理连接的客户端的通信。如果 accept 返回 INVALID_SOCKET，则打印错误消息并继续下一次迭代。对于每个成功接受的连接，我会使用 CreateThread 函数创建一个新线程，调用 ThreadHandle 函数处理与新的客户端的通信。最终，在服务器循环退出时（当 exit_requested 标志为 true 时），

我会关闭主服务器套接字，并使用 WSACleanup()清理 Winsock API。

- 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）

```
DWORD WINAPI ThreadHandle(LPVOID lpParam) {
    char buffer[1024 * 1024] = {0};
    SOCKET clientSocket = *(static_cast<SOCKET *>(lpParam));
    struct sockaddr_in HTTPAddress = {0};
    int HTTPAddrLen = sizeof(HTTPAddress);
    if (getpeername(clientSocket, (struct sockaddr *)&HTTPAddress, &HTTPAddrLen) == SOCKET_ERROR) {
        perror("Error getting client address");
        closesocket(clientSocket);
        return -1;
    }
    std::cout << "Client connected: " << inet_ntoa(HTTPAddress.sin_addr) << ":" << HTTPAddress.sin_port << std::endl;
    while (true) {
        if( recv( clientSocket, buffer, sizeof(buffer), 0) < 0 ) {
            printf( "Receive Error from client!\n" );
            closesocket(clientSocket);
            return -1;
        }
        std::string request(buffer);
        std::istringstream ss(request);
        std::string method, file, headers;
        ss >> method >> file >> headers;

        if (method == "GET") {
            GETHandle(clientSocket, file);
        } else if (method == "POST") {
            POSTHandle(clientSocket, file, request);
            break;
        } else {
            UnknownMethodHandle(clientSocket);
        }
    }
    std::cout << "Thread " << clientSocket << " exiting." << std::endl;
    closesocket(clientSocket);
    return 0;
}
```

首先我创建了一个用于存储接收数据的缓冲区 buffer；然后，通过将参数 lpParam 强制类型转换为 SOCKET 类型，获取客户端套接字 clientSocket；接着，我使用 getpeername 函数获取客户端的地址信息，并在控制台输出连接的客户端的 IP 地址和端口号。

接下来，我们进入一个无限循环来处理客户端的具体请求，不断接收来自客户端的数据。如果接收过程中出现错误，会打印错误消息，并关闭客户端套接字，然后退出线程。接收到数据后，将其解析，使用字符串流的方式从接收到的 request 这种获取 HTTP 请求的方法、文件和头部信息。

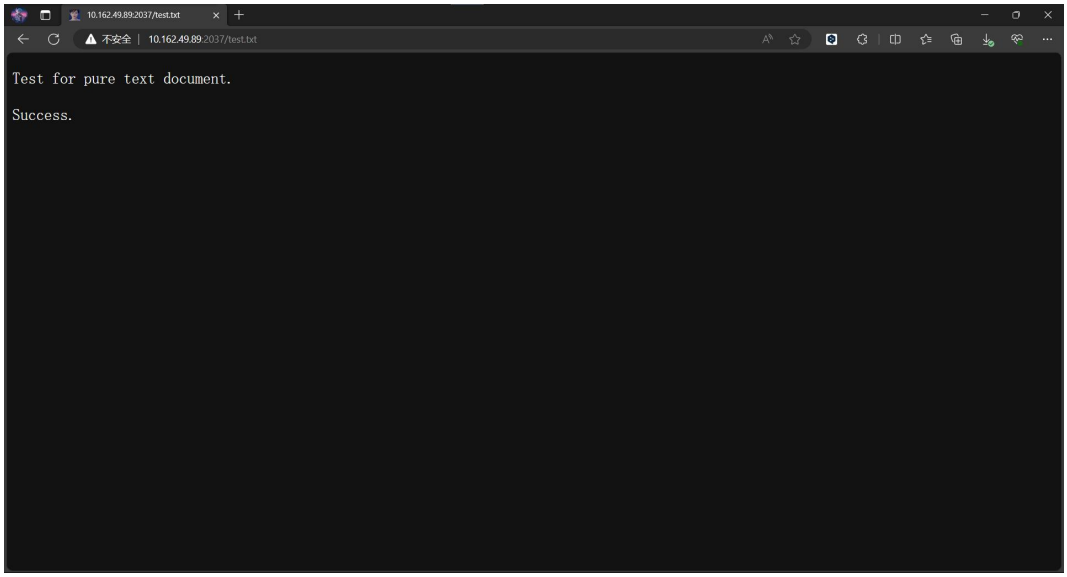
根据解析得到的 HTTP 方法，分别调用相应的处理函数：GETHandle 用于处理 GET 请求，POSTHandle 用于处理 POST 请求，而对于未知的 HTTP 方法，调用 UnknownMethodHandle 进行处理。

如果我们处理的是 POST 请求，那处理完后，线程打印一条消息表示正在退出，并关闭客户端套接字，最后返回 0 以结束线程；其他请求则继续接受这个客户端的数据，并判断新的请求是什么类型，即进入下一个循环

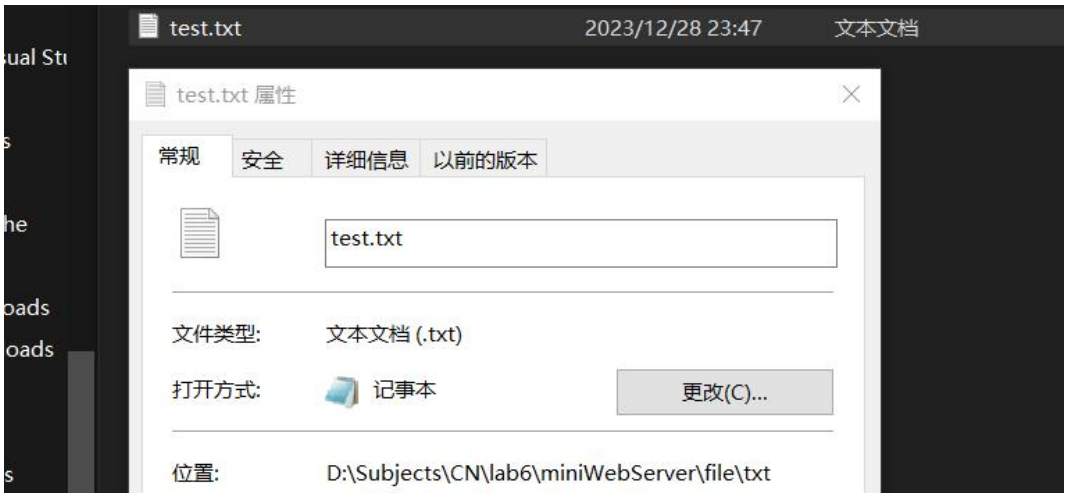
- 服务器运行后，用 netstat –an 显示服务器的监听端口

```
PS C:\Users\admin> 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    0.0.0.0:445            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    0.0.0.0:2037           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:2179           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5357           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5426           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:7890           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:17861          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：

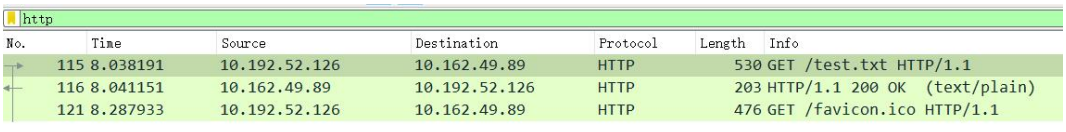


服务器的相关代码片段：

```
void GETHandle(SOCKET clientSocket, std::string url) {
    std::cout << "handle GET request!" << std::endl;
    // analyse the request
    std::string response_type;
    std::string filepath = "..";
    if (url.find(".txt") != std::string::npos) {
        response_type = "text/plain";
        filepath += "/file/txt";
    } else if (url.find(".html") != std::string::npos) {
        response_type = "text/html";
        filepath += "/file/html";
    } else if (url.find(".jpg") != std::string::npos) {
        response_type = "image/jpg";
        filepath += "/file/img";
    } else if (url.find(".ico") != std::string::npos) {
        response_type = "image/ico";
        filepath += "/img";
    }
    filepath += url;

    // read the files
    std::string response;
    std::ifstream fileStream(filepath, std::ios::binary);
    if (fileStream.is_open()) {
        std::cout << "file found" << std::endl;
        response.clear();
        std::ostringstream fileContentStream;
        fileContentStream << fileStream.rdbuf();
        std::string fileContent = fileContentStream.str();

        // 构建HTTP响应
        std::ostringstream responseStream;
        responseStream << "HTTP/1.1 200 OK\r\n";
        responseStream << "Connection: keep-alive\r\n";
        responseStream << "Content-Length: " << fileContent.size() << "\r\n";
        responseStream << "Server: csr_http1.1\n";

        if (url.find(".jpg") != std::string::npos) {
            responseStream << "Accept-Ranges: bytes\r\n";
        }

        responseStream << "Content-Type: " << response_type << "\r\n";
        responseStream << "\r\n";
        responseStream << fileContent;

        // 发送HTTP响应
        response = responseStream.str();
    } else {
        // 文件不存在，返回404 Not Found
        std::cout << "file found failed!!!!!" << std::endl;
        response.clear();
        response = "HTTP/1.1 404 Not Found\r\n";
        response += "Content-Type: text/html;charset=utf-8\r\n";
        response += "Content-Length: 84\r\n\r\n";
        response += "<html><body><h1>404 Not Found</h1><p>From server: URL is wrong.</p></body></html>\r\n";
    }
    send(clientSocket, response.c_str(), response.size(), 0);
    fileStream.close(); // 记得关闭文件
    fileStream.clear();
}
```

对获取到的 GET 请求中的请求行进行分解，获得 URL 中请求的文件内容，据此再取服务器本地磁盘文件中寻找对应的文件，找到后组装响应消息；

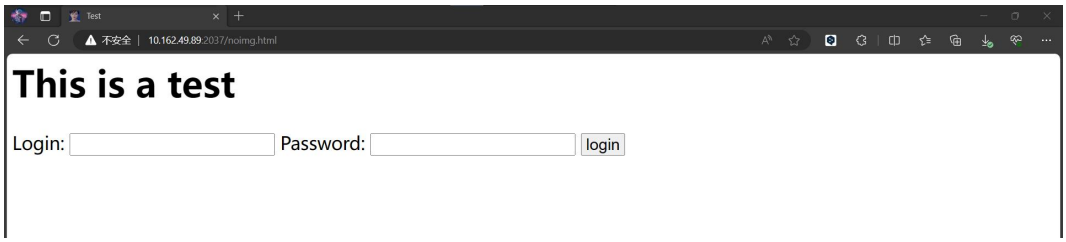
Wireshark 抓取的数据包截图（通过跟踪 TCP 流，只截取 HTTP 协议部分）：



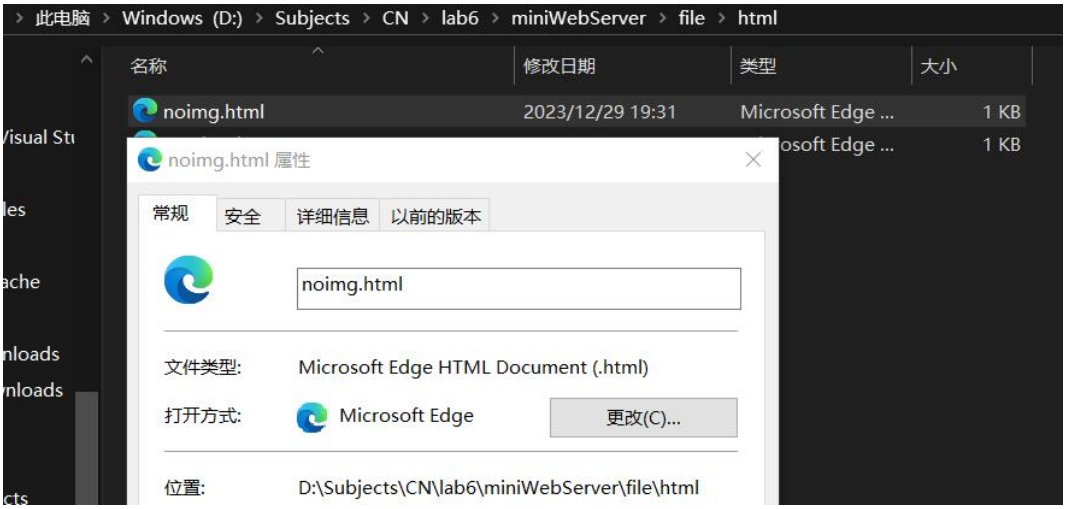
http

No.	Time	Source	Destination	Protocol	Length	Info
115	8.038191	10.192.52.126	10.162.49.89	HTTP	530	GET /test.txt HTTP/1.1
116	8.041151	10.162.49.89	10.192.52.126	HTTP	203	HTTP/1.1 200 OK (text/plain)
121	8.287933	10.192.52.126	10.162.49.89	HTTP	476	GET /favicon.ico HTTP/1.1

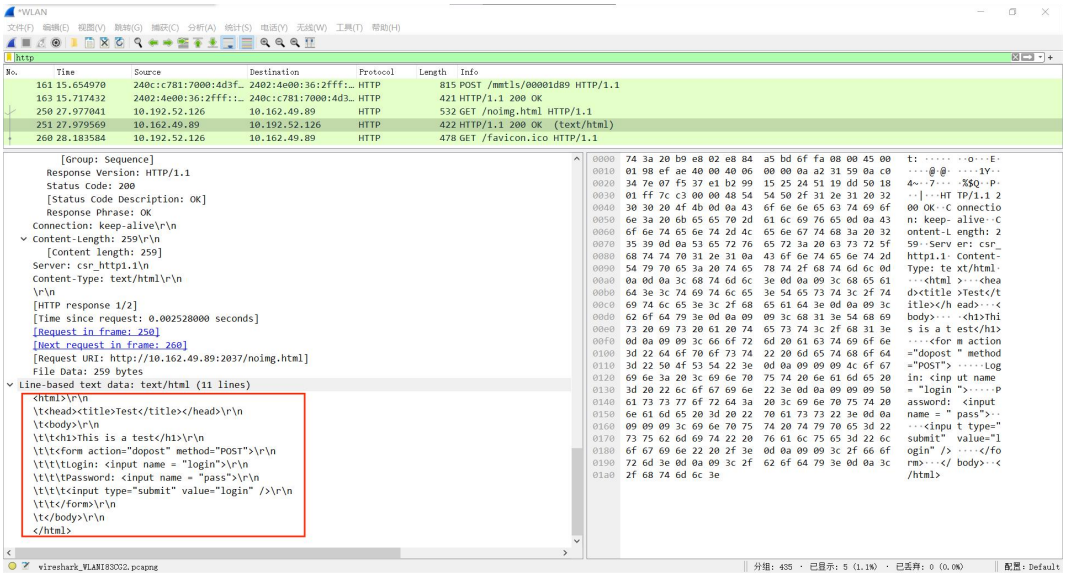
- 浏览器访问只包含文本的 HTML 文件时，浏览器的 URL 地址和显示内容截图。



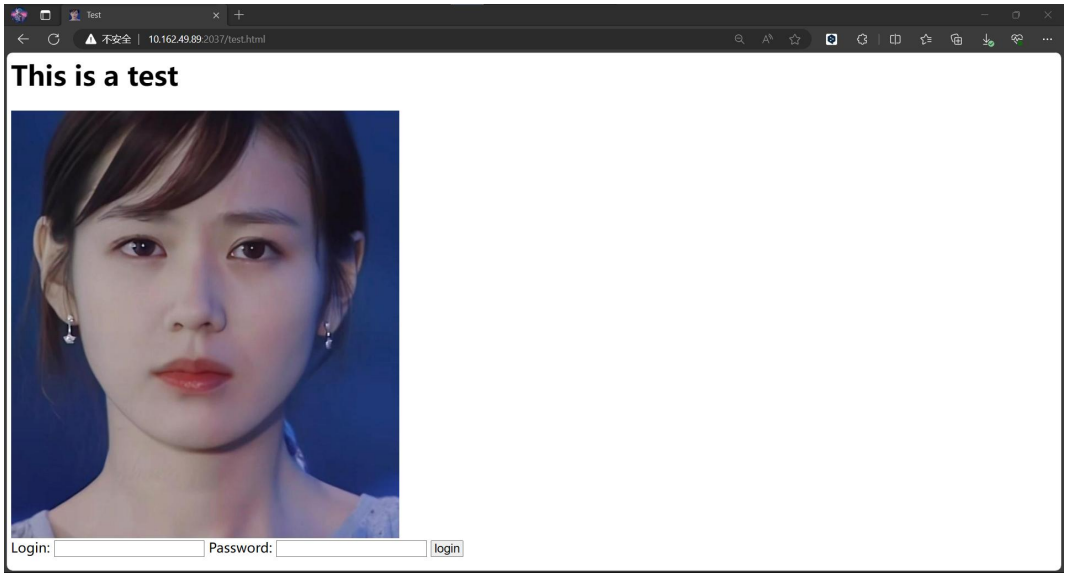
服务器文件实际存放的路径：



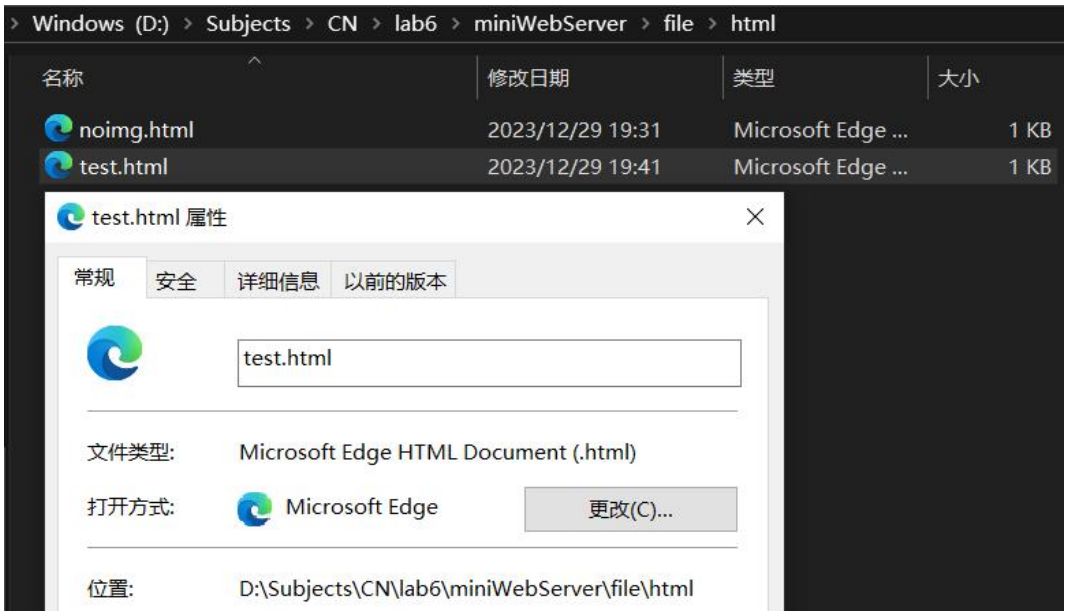
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML 内容）：



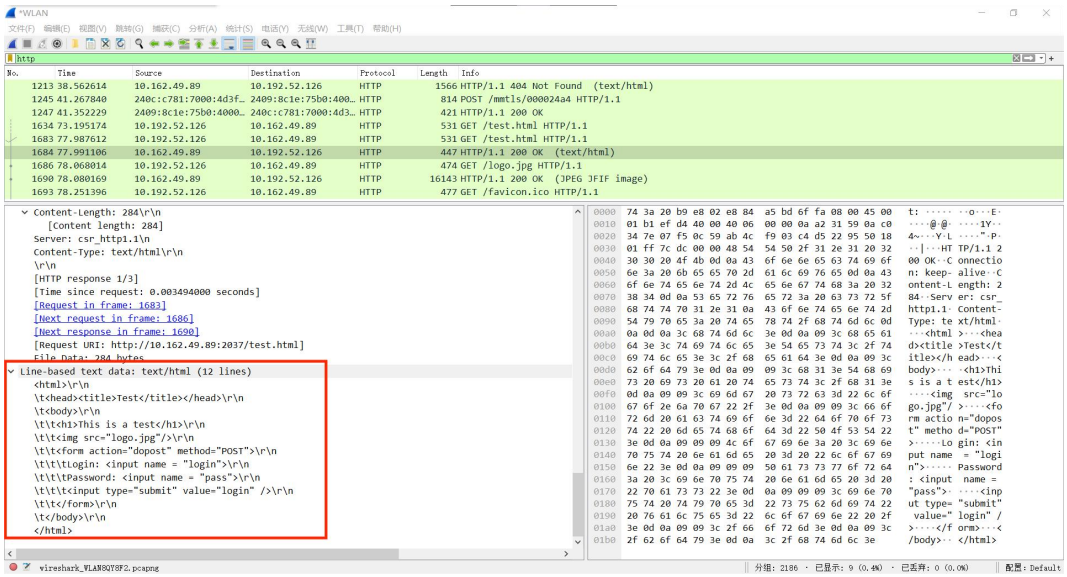
- 浏览器访问包含文本、图片的 HTML 文件时，浏览器的 URL 地址和显示内容截图。

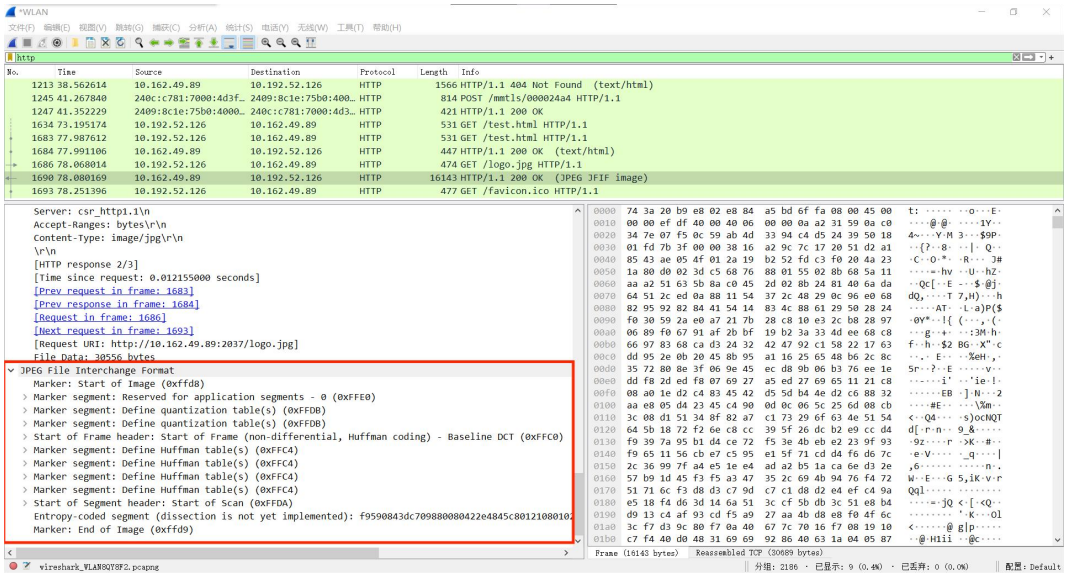


服务器上文件实际存放的路径：

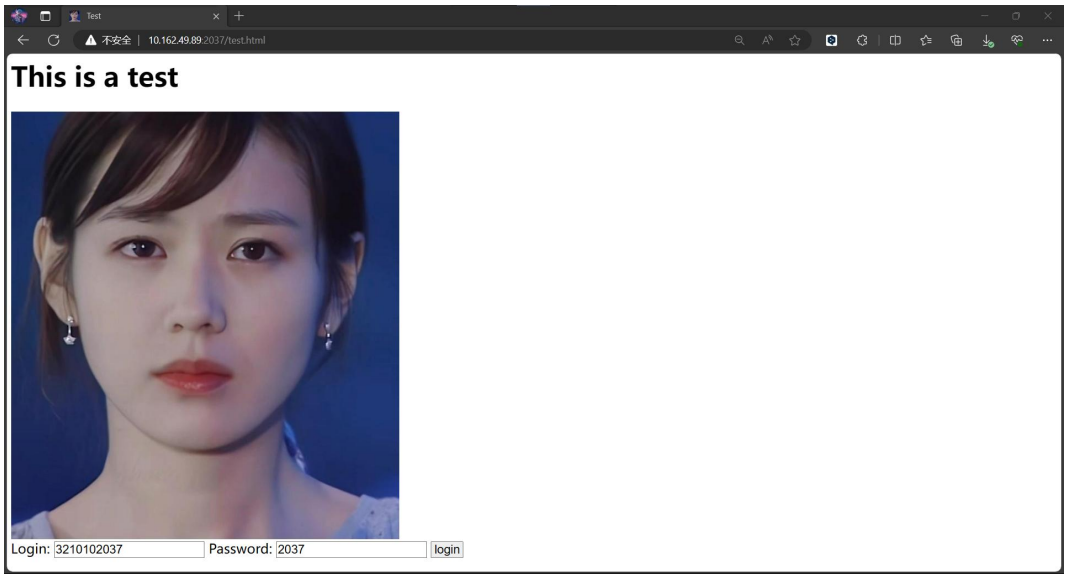


Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：





- 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。



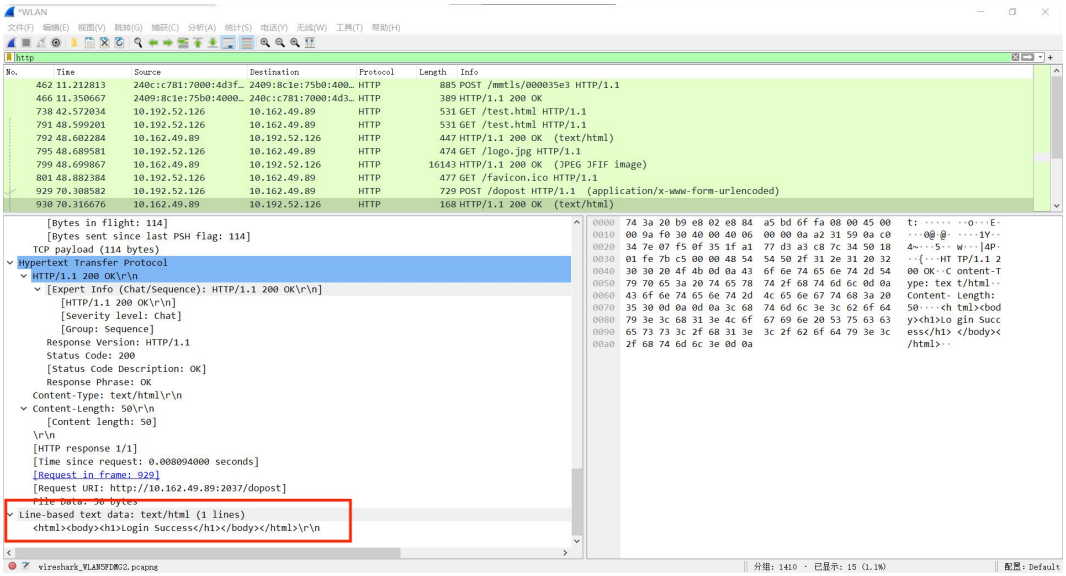
服务器相关处理代码片段：

```
void POSTHandle(SOCKET clientSocket, std::string url, std::string request) {
    std::cout << "handle POST request!" << std::endl;
    std::cout << request << std::endl;
    std::string response;
    std::string content, loginName, passWord;
    if (url.find("dopost") != std::string::npos) {
        response.clear();
        response = "HTTP/1.1 200 OK\r\n";
        response += "Content-Type: text/html\r\n";
        response += "Content-Length: ";
        // get the post content
        content = request.substr(request.find("login=") + 6);
        loginName = content.substr(0, content.find("&"));
        passWord = content.substr(content.find("&") + 1 + 5);
        std::cout << loginName << "  " << passWord << std::endl;
        if (loginName == std::to_string(LOGIN) && passWord == std::to_string(PASSWORD)) {
            std::string SuccessMsg = "<html><body><h1>Login Success</h1></body></html>\r\n";
            response += std::to_string(SuccessMsg.length());
            response += "\r\n\r\n";
            response += SuccessMsg;
        } else {
            std::string FailMsg = "<html><body><h1>Login Fail</h1></body></html>\r\n";
            response += std::to_string(FailMsg.length());
            response += "\r\n\r\n";
            response += FailMsg;
        }
    } else {
        response.clear();
        response = "HTTP/1.1 404 Not Found\r\n";
        response += "Content-Type: text/html;charset=utf-8\r\n";
        response += "Content-Length: 0\r\n\r\n";
    }
    send(clientSocket, response.c_str(), response.size(), 0);
}
```

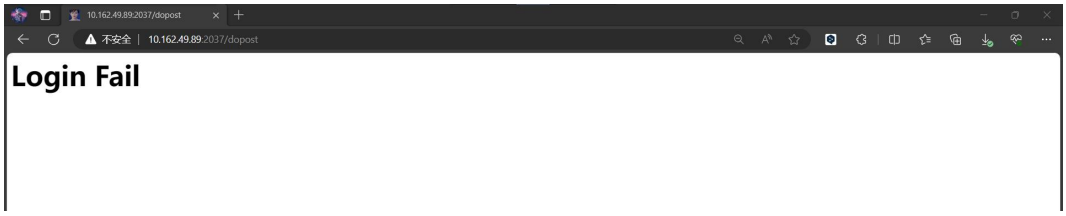
这里我们需要处理 POST 请求，首先根据请求头中的 URL 判断这是否是“dopost”请求，如果是继续进行分解，将 request 中的 Content 部分找到，并根据 submit 的内容格式分解出相应的登录名和密码，比较后组装相应的响应消息；

否则，组装 404 的响应消息。

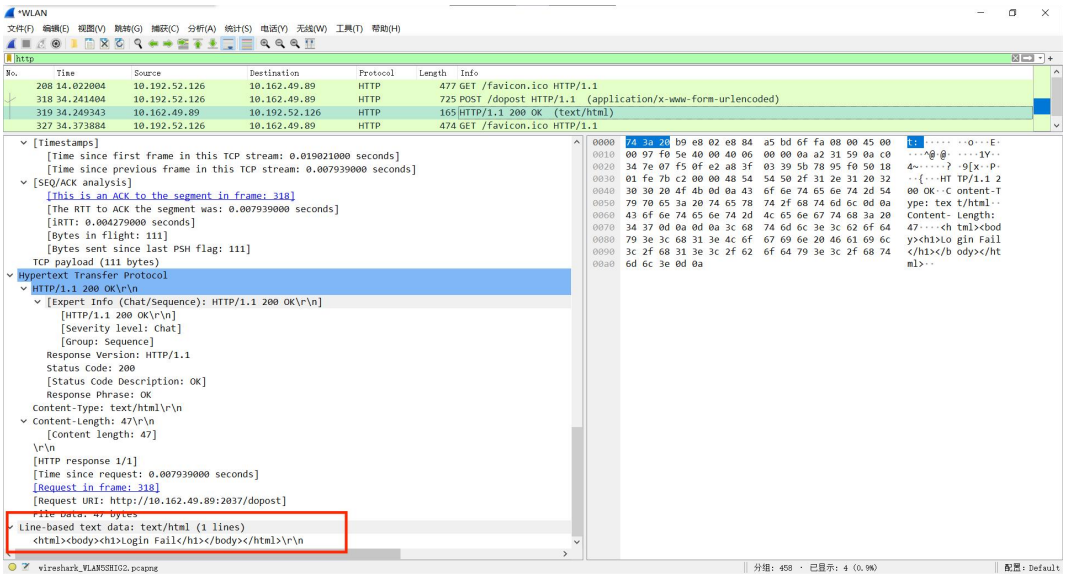
Wireshark 抓取的数据包截图（HTTP 协议部分）



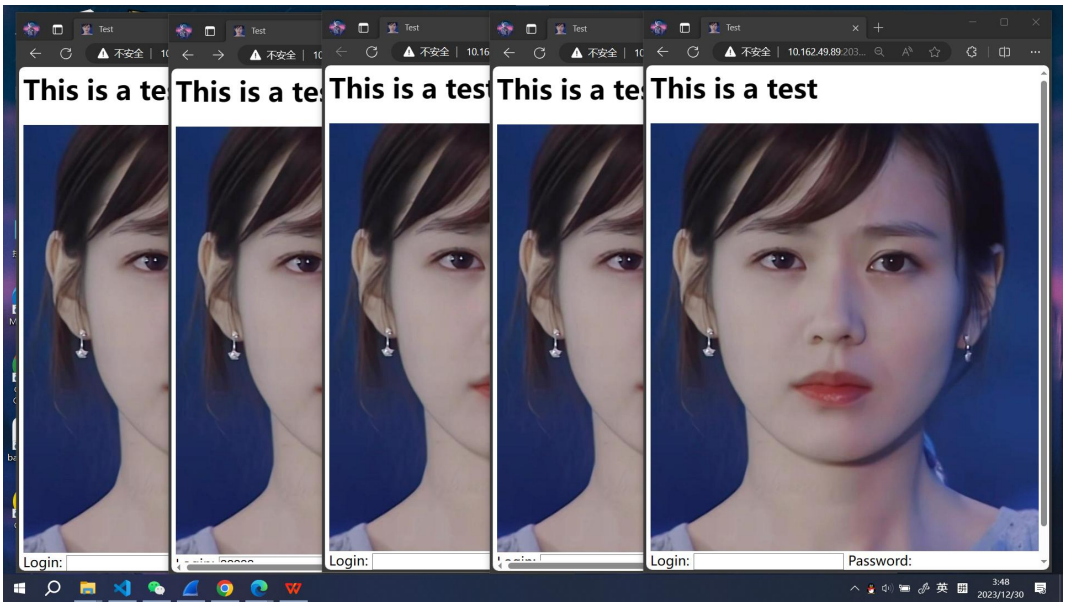
- 浏览器输入错误的登录名或密码，点击登录按钮（login）后的显示截图。



- 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    0.0.0.0:2037           0.0.0.0:0              LISTENING
TCP    10.162.49.89:2037      10.162.49.89:54982     CLOSE_WAIT
TCP    10.162.49.89:2037      10.162.49.89:55176     CLOSE_WAIT
TCP    10.162.49.89:2037      10.162.49.89:55205     ESTABLISHED
TCP    10.162.49.89:2037      10.162.49.89:55379     ESTABLISHED
TCP    10.162.49.89:2037      10.162.49.89:55382     ESTABLISHED
TCP    10.162.49.89:2037      10.162.49.89:55383     ESTABLISHED
TCP    10.162.49.89:2037      10.162.49.89:55384     ESTABLISHED
TCP    10.162.49.89:2037      10.162.49.89:55385     ESTABLISHED
TCP    10.162.49.89:2037      10.192.52.126:3884     CLOSE_WAIT
TCP    10.162.49.89:2037      10.192.52.126:4067     ESTABLISHED
TCP    10.162.49.89:54982     10.162.49.89:2037      FIN_WAIT_2
TCP    10.162.49.89:55176     10.162.49.89:2037      FIN_WAIT_2
TCP    10.162.49.89:55205     10.162.49.89:2037      ESTABLISHED
TCP    10.162.49.89:55379     10.162.49.89:2037      ESTABLISHED
TCP    10.162.49.89:55382     10.162.49.89:2037      ESTABLISHED
TCP    10.162.49.89:55383     10.162.49.89:2037      ESTABLISHED
TCP    10.162.49.89:55384     10.162.49.89:2037      ESTABLISHED
TCP    10.162.49.89:55385     10.162.49.89:2037      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？

  在 HTTP 协议中，头部（Header）和主体（Body）之间通过一个空行来进行分隔。这个空行是由回车符（CR，ASCII 码为 13）和换行符（LF，ASCII 码为 10）组成的，即"\r\n"。

  对于头部中的请求（应答）行、请求（应答）头等部分，也用“\r\n”

分隔，因此头部和主体之间实际上由“\r\n\r\n”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？

  根据头部的 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？

  头部是纯文本格式，但是体部不一定，也可以有“.jpg”格式，需要根据 Content-Type 来确定。

- 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？

  放在体部；

  两个字段是使用“&”连接的。

七、 讨论、心得

建议参考操作系统的实验设计，对整体代码给出一个框架，而不是全部由学生自己写。