

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： 沈子衿

学 院： 计算机学院

系： 软件工程

专 业： 软件工程

学 号： 3160104734

指导教师： 董玮

2018 年 11 月 17 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: 林宇翔 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
- 本实验可单独完成或组成两人小组。若组成小组, 则一人负责编写服务器 GET 方法的响应, 另一人负责编写 POST 方法的响应和服务器主线程。

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（该路径和服务端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）

```
while (1) {
    printf("Waiting for client to enter...\n");
    //试图接受一个连接，这个连接是阻塞的，没有新的连接进入会一直阻塞在那里
    clientSocket = accept(serverSocket, (sockaddr*)&serverChannel,
&len);

    //如果在阻塞的过程中出现了错误，则返回错误
    if (clientSocket < 0) {
        printf("ACCEPT_ERROR\n");
    }
    else {
        // 接收成功，也就算连接成功了
        printf("Connection successful!\n");
        memset(buffer, 0, sizeof(buffer));
```

```

        int ret;
        // 此时客户端必然会发来请求（页面或文本），这时使用 recv 阻塞函
数监听这一请求
        ret = recv(clientSocket, buffer, BUFFER_SIZE, 0);
        if (ret == SOCKET_ERROR) {
            printf("Fail to receive due to SOCKET_ERROR
(ret<0)\n");
        }
        // 出现 0，连接没有出现异常，一般情况下是客户端主动关闭连接
        else if (ret == 0) {
            printf("A sudden disconnection occur(ret=0)\n");
        }
        else {
            printf("Reception Sucessful\n");
            for (int i = 0; i < MAX; i++) {
                if (!isActive[i]) {
                    isActive[i] = true;
                    // 构造处理特定客户端信息的主线程
                    message msg(buffer, &isActive[i], clientSocket,
i);

                    t[i] = new std::thread(handleMessage, msg);
                    break;
                }
            }
        }
    }
}

```

这段代码的逻辑同 socket 实验服务端的代码逻辑类似。首先，程序调用一个名为 `accept` 的函数监听在此前设定的 `serverChannel` 上（其中包括之前在 `bind` 操作绑定的 ip 地址以及端口），`accept` 函数阻塞地接收从客户端发过来的连接请求，一旦接收到请求，服务器端输出连接成功的信息，随即接收从客户端发来的第一条信息（这个过程同样也是阻塞的），一旦接收到信息，则证明客户端与服务器端的联系是正常的（三次握手的原则），此时开辟一个新线程，将处理该客户端来往消息的业务代码放到该线程中，完成一次同客户端之间的连接。

- 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）

```

while (1) {
    if ((i >= strlen(msg.data.c_str()) || msg.data[i] == '\n') &&
(msg.data[i + 2] == '\n'))break;
    if (msg.data[i] == ' ') {
        if (isGetMethod) {
            data = temp;
            isGetMethod = false;

```

```

        break;
    }
    //如果是 GET 请求，由于需要进一步解析 GET 请求的路径，这里直接将
isGetMethod 置为 true
    //等到 i 移动到路径之前的空格时将包内剩下的内容全部储存到 data
中，进入下一个代码段
    else if (temp == "GET") {
        RESTfulmethod = temp;
        isGetMethod = true;
    }
    //如果是 POST，由于信息在 body 中，不需要提前结束循环
    else if (temp == "POST") {
        RESTfulmethod = temp;
        isPostMethod = true;
    }
    temp = "";
}
else if(msg.data[i] != '\n'){
    temp = temp + msg.data[i];
}
++i;
}
if (RESTfulmethod == "POST") {...}
else if (RESTfulmethod == "GET" && data != "") {...}
}

```

这段代码的主要逻辑体现在将请求的头部与 body 分离。考虑到这一点，我省略了较为复杂的 post 和 get 解析代码，重点介绍头部的解析过程。首先，将整个请求包 msg 放入无限循环中进行遍历，如未检测到空白字符，则将当前字符加入 temp 中；如果遇到空格，则检测当前的 type 是否为 get 和 post，如果是其中一个，将相应的 flag 置为 true，进入指定的语句块处理。对于 get，由于其请求头的特殊性（需要解析路径）。因此在解析出 get 之后，循环终止，未处理的头交由专门处理 get 语句的代码段解析。而本题中的 post 不需要解析路径，因此可以让该循环将头部解析完毕，只让后面的 post 语句块处理 body 中的键值对即可。

- 服务器运行后，用 netstat -an 显示服务器的监听端口

```

C:\Users\沈子衿>netstat -an
活动连接
 协议 本地地址           外部地址           状态
TCP    0.0.0.0:80          0.0.0.0:0          LISTENING

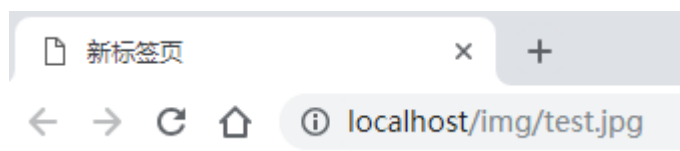
```

运行服务器之后，发现在列表的第一个，80 端口处于 LISTENING 状态，此即为服务器端的监听端口。

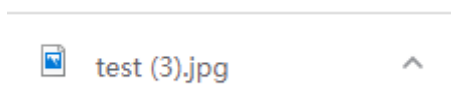
- 浏览器访问图片文件（.jpg）时，浏览器的 URL 地址和显示内容截图。

在本地浏览器中输入 `http://localhost/img/test.jpg`，访问无图片的网页。在这里我遇到了一个值得探究的情况。

这一情况出现的背景是：虽然我们完全可以通过将 `content-type` 设置为 `text/html`，然后将图片路径包装到一个 `html` 中再发送到客户端（因为这样也可以实现在浏览器端的预览），但这与本小题的题意相悖，故我们通过将 `header` 中的 `content-type` 设置为 `img/jpeg` 的方式来直接访问图片。



首先在 chrome 浏览器地址栏中输入地址，点击访问



发现浏览器并没有打开图片预览，而是将图片下载到了本地



使用本地照片查看器打开图片，可以正常显示，说明图片信息成功地从服务器传输到了客户端。

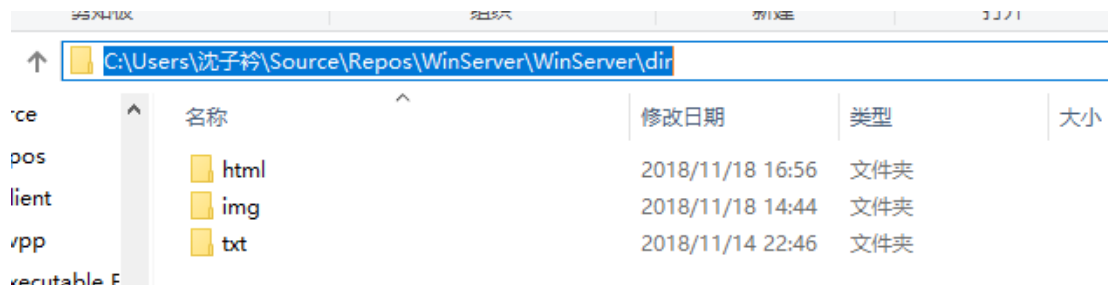
但是，当我使用手机端的 safari 和 Internet Explorer 访问时，则直接打开了图片预览：



另使用 Microsoft Edge 复现以上测试，情况和 chrome 一样，图片被直接下载而非以网页形式预览。

经网上查阅资料可知，服务器端资源文件在客户端的展现形式主要由浏览器和前端代码决定。考虑到已经实现了基本功能，这里便没有做进一步调整。

服务器上文件实际存放的路径：



在根目录的 dir 文件夹下,相对路径为\dir\img\test.jpg.

服务器的相关代码片段：

```
void SendMsg(std::string path, Message msg, int msgType = 1){
```



```

//从本地读取文件流
std::ifstream in(path, std::ios::binary);
int sp;
if (!in) {
    // 未找到路径, 则返回 HTTP:404 报错
    msg.data = "HTTP/1.1 404 Not Found\n";
    printf("404: No such Path or Directory\n");
}
else {
    in.seekg(0, std::ios_base::end);
    sp = in.tellg();
    char length[20];
    sprintf(length, "%d", sp);
    msg.data = "HTTP/1.1 200 OK\n";
    if (msgType == 1) // 如果是文本或 html 信息, messageType 会被置为
1
        msg.data += "Content-Type:
text/html; charset=utf-8\nContent-Length: ";
    else { // 如果是图片信息, messageType 会被置为 0
        printf("this is an img\n");
        msg.data += "Content-Type: img/jpeg; \nContent-Length: ";
    }
    msg.data += length;
    msg.data += "\n\n";
    int total_size = 0;
    int r = send(msg.clientSocket, msg.data.c_str(),
strlen(msg.data.c_str()), 0);
    if (r == SOCKET_ERROR) {
        printf("send failed\n");
        *msg.live = false;
        return;
    }
    else {
        printf("send success\n");
    }
    char buffer[100];
    int s = sp + strlen(msg.data.c_str()) + 1;
    int len = sp;
    total_size = 0;
    in.clear();
    in.seekg(0, std::ios_base::beg);

    while (len > 0) {
        memset(buffer, 0, sizeof(buffer));

```

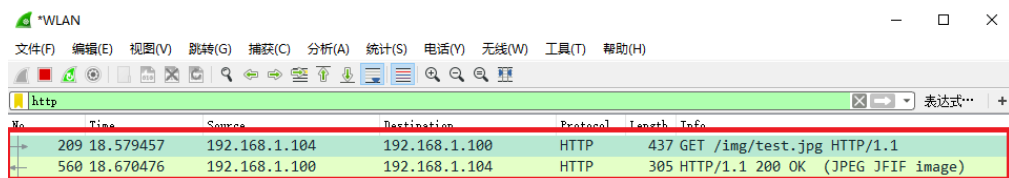
```

        int size = sizeof(buffer) < len ? sizeof(buffer) : len;
        total_size += size;
        len -= size;
        in.read(buffer, size);
        int r = send(msg.clientSocket, buffer, size, 0);

        if (r == SOCKET_ERROR) {
            printf("send failed\n");
            *msg.live = false;
            return;
        }
    }
}
}
}

```

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

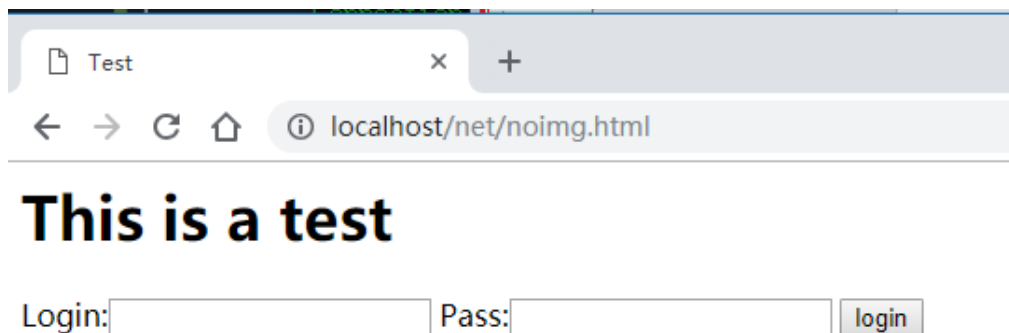


No.	Time	Source	Destination	Protocol	Length	Info
209	18.579457	192.168.1.104	192.168.1.100	HTTP	437	GET /img/test.jpg HTTP/1.1
560	18.670476	192.168.1.100	192.168.1.104	HTTP	305	HTTP/1.1 200 OK (JPEG JFIF image)

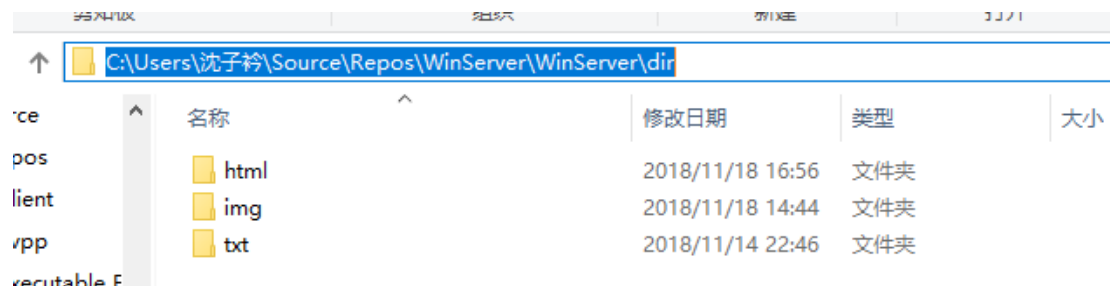
如图，是我使用寝室路由器子网下的另一个设备 192.168.1.104 请求服务器一次，请求数据包和响应数据包的信息。

- 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。

在本地浏览器中输入 <http://localhost/net/noimg.html>，访问无图片的网页：

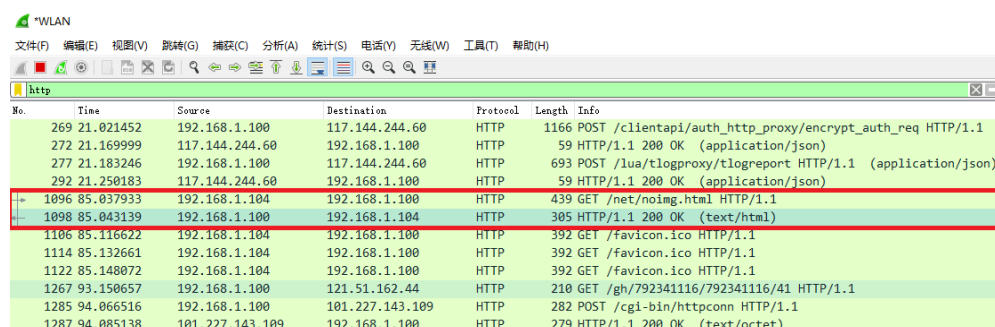


服务器文件实际存放的路径：

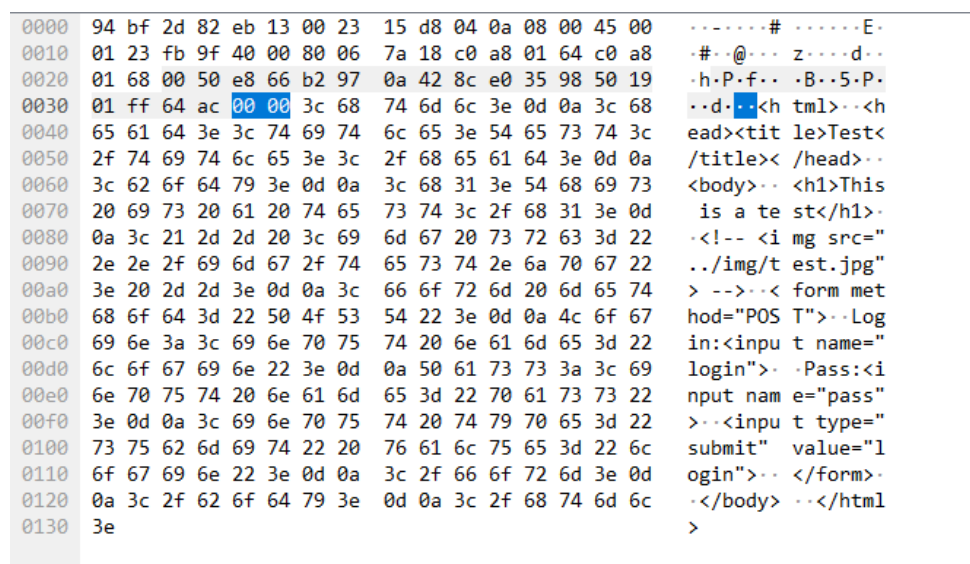


相对路径为 dir/html/noimg.html

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



如图，是我使用寝室路由器子网下的另一个设备 192.168.1.104 请求服务器一次，请求数据包和响应数据包的信息。之所以使用手机测试数据包，是因为资料显示 wireshark 不能捕获本地回路数据包。

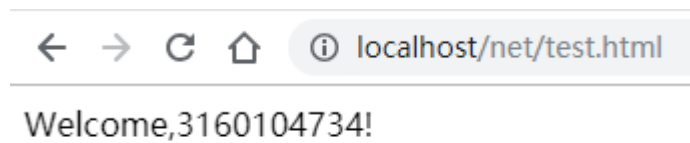


如图是数据包内容，可以清晰看出所包含的 html 文本。

- 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。

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。

点击登陆：



服务器相关处理代码片段：

```
if (RESTfulmethod == "POST") {

    bool startToEnterUsername = false;
    bool startToEnterPassword = false;
    std::string name = "";
    std::string password = "";
    temp = "";
    for (int j = i + 3; j <= strlen(msg.data.c_str()); j++) {
        if (msg.data[j] == '&'amp;' || msg.data[j] == '=' || j ==
strlen(msg.data.c_str())) {
            std::cout << temp << std::endl;
            //进入用户名字段
            if (startToEnterUsername) {
                if (temp == "3160104734") {
                    name = temp;
                    password = "4734";
                }
                else
                    password = "";
                startToEnterUsername = false;
            }
            //开始进入密码字段
            else if (startToEnterPassword) {
                // 如果密码和用户名匹配
                if (temp == password && temp != "") {
                    std::cout << "temp=" << temp << " " << "paw="
<< password << std::endl;
                    char response[200];
                    strcpy(response, "<html><body>Welcome,");
                    strcat(response, name.c_str());
                    strcat(response, "!</body></html>\n");
                    int len = strlen(response);
                    char length[20];
                    sprintf(length, "%d", len);
                    msg.data = "HTTP/1.1 200 OK\n";
```

```

        msg.data += "Content-Type:
text/html;charset=utf-8\nContent-Length: ";
        msg.data += length;
        msg.data += "\n\n";
        msg.data += response;
        printf("%s\n", msg.data.c_str());
        int r = send(msg.clientSocket,
msg.data.c_str(), 10000, 0);

        if (r == SOCKET_ERROR) {
            printf("send failed\n");
            *msg.live = false;
            return;
        }
        printf("send success\n");
        *msg.live = false;
        return;
    }
    // 如果密码和用户名不匹配
    else {
        // 构建响应头
        char response[200];
        strcpy(response, "<html><body>login failed:
Wrong user or password</body></html>\n");
        int len = strlen(response);
        char length[20];
        sprintf(length, "%d", len);
        msg.data = "HTTP/1.1 200 OK\n";
        msg.data += "Content-Type:
text/html;charset=utf-8\nContent-Length: ";
        msg.data += length;
        msg.data += "\n\n";
        msg.data += response;
        printf("%s\n", msg.data.c_str());
        int ret = send(msg.clientSocket,
msg.data.c_str(), 10000, 0);
        if (ret == SOCKET_ERROR) {
            printf("send failed\n");
            *msg.live = false;
            return;
        }
        printf("send success\n");
        *msg.live = false;
        return;
    }
}

```

```

    }
    startToEnterPassword = false;
}
else if (temp == "login")
    startToEnterUsername = true;
else if (temp == "pass")
    startToEnterPassword = true;
if (j == data.size())break;
temp = "";
}
else {
    temp = temp + msg.data[j];
}
}
*msg.live = false;
return;
}

```

Wireshark 抓取的数据包截图（HTTP 协议部分）

The screenshot shows a Wireshark capture of an HTTP POST request. The packet list pane highlights packet 1687, which is a POST request to /net/test.html. The packet details pane shows the HTTP form URL encoded data. The packet bytes pane shows the raw data with a red box highlighting the login and password fields.

No.	Time	Source	Destination	Protocol	Length	Info
14	0.936143	192.168.1.100	223.111.187.174	HTTP	331	GET /pc/qqclient/sfile/index.json HTTP/1.1
18	0.964990	223.111.187.174	192.168.1.100	HTTP	283	HTTP/1.1 200 OK (application/json)
362	24.754084	192.168.1.104	192.168.1.100	HTTP	438	GET /net/test.html HTTP/1.1
364	24.760949	192.168.1.100	192.168.1.104	HTTP	296	HTTP/1.1 200 OK (text/html)
373	24.826603	192.168.1.104	192.168.1.100	HTTP	450	GET /img/test.jpg HTTP/1.1
729	24.995894	192.168.1.100	192.168.1.104	HTTP	340	HTTP/1.1 200 OK (JPEG JFIF image)
952	30.438773	192.168.1.100	125.78.252.152	HTTP	671	GET /gchatpic_new/9CF082D5882ED878C3360FAFD48EB2D290D490D463EF3746263880095 HTTP/1.1
1039	30.723525	125.78.252.152	192.168.1.100	HTTP	1053	HTTP/1.1 200 OK (image/jpeg)
1079	34.942362	192.168.1.104	192.168.1.100	HTTP	72	POST /net/test.html HTTP/1.1 (application/x-www-form-urlencoded)
1177	40.676482	192.168.1.104	192.168.1.100	HTTP	438	GET /net/test.html HTTP/1.1
1179	40.683966	192.168.1.100	192.168.1.104	HTTP	296	HTTP/1.1 200 OK (text/html)
1187	40.711197	192.168.1.104	192.168.1.100	HTTP	450	GET /img/test.jpg HTTP/1.1
1503	40.779083	192.168.1.100	192.168.1.104	HTTP	353	HTTP/1.1 200 OK (JPEG JFIF image)
1687	49.486045	192.168.1.104	192.168.1.100	HTTP	72	POST /net/test.html HTTP/1.1 (application/x-www-form-urlencoded)
1689	49.500510	192.168.1.100	192.168.1.104	HTTP	590	HTTP/1.1 200 OK (text/html)Continuation
1832	60.213047	192.168.1.100	125.78.252.155	HTTP	671	GET /gchatpic_new/2C6360808C5F641E9553158E10E50C071E8BA1D6223A7B74380F07E65 HTTP/1.1
1841	60.243856	125.78.252.155	192.168.1.100	HTTP	1321	HTTP/1.1 200 OK (image/jpeg)

Frame 1687: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface 0
 Ethernet II, Src: Apple_82:eb:13 (94:bf:2d:82:eb:13), Dst: IntelCor_d8:04:0a (00:23:15:d8:04:0a)
 Internet Protocol Version 4, Src: 192.168.1.104, Dst: 192.168.1.100
 Transmission Control Protocol, Src Port: 60060, Dst Port: 80, Seq: 530, Ack: 1, Len: 18
 [2 Reassembled TCP Segments (547 bytes): #1686(529), #1687(18)]
 Hypertext Transfer Protocol
 HTML Form URL Encoded: application/x-www-form-urlencoded

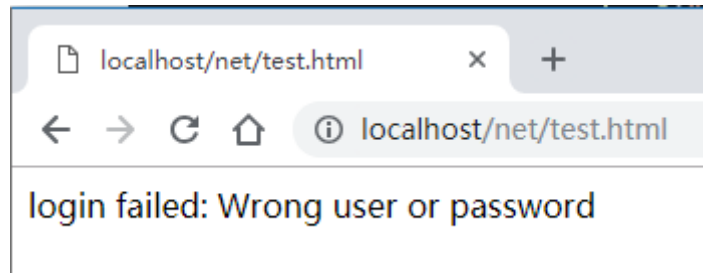
```

0000  00 23 15 d8 04 0a 94 bf 2d 82 eb 13 08 00 45 00  .#.....E.
0010  00 3a 00 00 40 00 40 06 b6 a1 c0 a8 01 68 c0 a8  :..@..@.....h..
0020  01 64 ea 9c 00 50 4b 72 e3 7a 75 6c 11 b2 50 18  -d...PKr..zul..P.
0030  08 00 88 fd 00 00 6c 6f 67 69 6e 3d 31 32 33 26  ....lo gin=123&
0040  70 61 73 73 3d 31 32 33  pass=123

```

- 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。

Login: Pass:

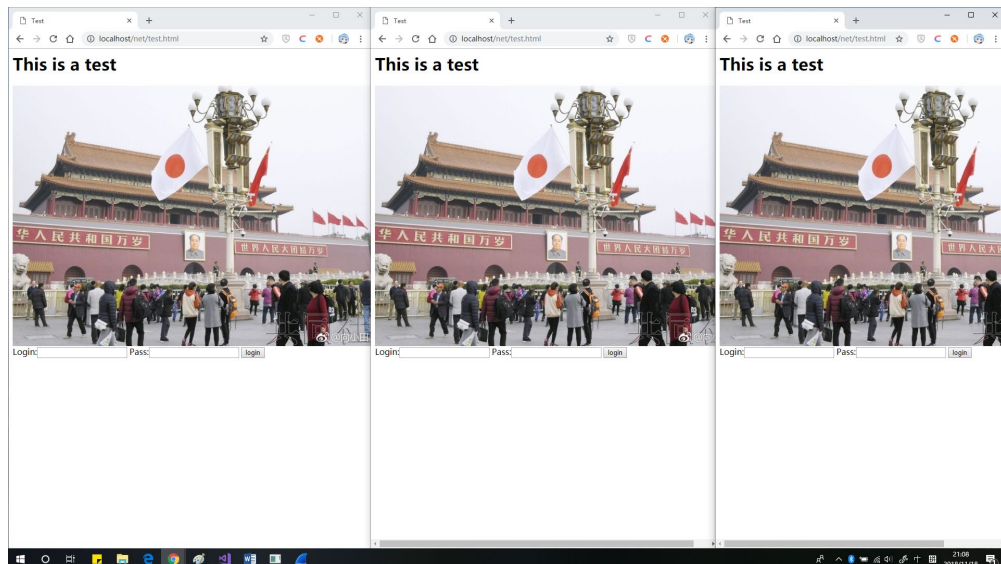


- Wireshark 抓取的数据包截图（HTTP 协议部分）

A screenshot of the Wireshark network protocol analyzer. The 'http2' filter is applied. The packet list shows four packets related to an HTTP GET request and its response.

No.	Source	Destination	Protocol	Length	Info
250	48.391471	192.168.1.101	HTTP	439	GET /net/noimg.html HTTP/1.1
252	48.395345	192.168.1.100	HTTP	305	HTTP/1.1 200 OK (text/html)
418	59.794072	192.168.1.101	HTTP	73	POST /net/noimg.html HTTP/1.1 (application/x
420	59.803122	192.168.1.100	HTTP	1514	HTTP/1.1 200 OK (text/html)Continuation

- 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）

A screenshot of the output of the `netstat -an` command. It shows a list of TCP connections to the server's IP address 127.0.0.1 on port 80. The connections are in various states, including CLOSE_WAIT, TIME_WAIT, and ESTABLISHED.

Protocol	Local Address	Foreign Address	State
TCP	127.0.0.1:80	127.0.0.1:7210	CLOSE_WAIT
TCP	127.0.0.1:80	127.0.0.1:7337	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7338	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7339	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7341	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7343	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7346	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7347	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7350	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7353	TIME_WAIT
TCP	127.0.0.1:80	127.0.0.1:7354	ESTABLISHED

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？
根据对截获的字符串格式的 http 协议头部分析，头部和体部是用三个换行符分割的。这也是我代码中分隔头部和体部的根据。
- 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？是
根据头部的 content-type 来判断类型的。如 html 或纯文本，其类型为 txt/html；而
对于 jpg 图片，其类型为 img/jpeg。这也是我在服务器端代码中区分普通文本、
html 和图片的根据。
- HTTP 协议的头部是不是一定是文本格式？体部呢？
头是文本格式，体部的格式由头部的特定字段决定，但整个 http 数据包均以二进制的形式传输。
- 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？
POST 传输的数据毫无疑问是放在体部，在存在多键值对且格式为 text 的情况下，
两个字段是通过 ‘&’ 符号连接在一起的，并且中间不能有多余的空格。

七、 讨论、心得

在本次实验中，我通过搭建简易的 web 服务器，复习了实验 2 中学习的 socket 编程相关知识，系统学习了 HTTP 协议概念、HTTP 协议请求头、HTTP 协议数据包以及 TCP 协议相关知识，对如何使用 C++ 发送、接受请求以及如何处理 HTTP 请求头与数据包有了较为清楚的了解。此外，我在实验 2 的基础上学习了先进的 C++11 线程库的用法，简化了编程，提高了效率，受益匪浅。

本次实验由于涉及模块多，开发周期长，具有一定挑战性。我遇到的主要挑战有两点：第一，是对多线程互斥和同步的处理；第二，是对不同网络环境下兼容性的处理。我尽自己所能克服了这些问题，成效显著，也了解了更多知识点中的细节，为未来完成期末课程设计和展示打下了坚实而有力的基础。