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：	姚熙源
学　　院：	计算机科学与技术学院
系：	软件工程系
专　　业：	软件工程
学　　号：	3190300677
指导教师：	董玮

2022 年　　3 月　　29 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：　-　　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**

- 编译环境
  - 虚拟机：VMWare Playstation 15 ， 操作系统：Linux
  -
```
peter@peter-virtual-machine:~$ uname -a
Linux peter-virtual-machine 5.4.0-105-generic #119-Ubuntu SMP M
on Mar 7 18:49:24 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux
```
- 编译步骤
  - 打开 terminal
  - cd /lab3 目录
  - 在 terminal 中输入 make clean（先删除所有链接文件.o 和执行文件.exe）
  - 然后再输入 make 即可生成.o 文件和可执行文件
  - 输入./webserver 即可开始 Web 服务器

- 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）

通过使用 C++的类封装来将网络连接的部分包装成 Network 类，在主线程开始时，实例化一个 Network 对象并且监听 3677 的端口然后进入主循环。在循环中，服务器不断接收来自端口 3677 的请求，由 RequestHandler 实例化的对象去获取该文件描述(file description)然后进行 Begin 方法，开始处理对客户端的 GET 或 POST 方法，处理完后再关闭此网络连接，然后进入循环又接收新的请求。

```
    Network network;
    int connfd;
    //listen to port which is my last 4 digit number of student ID
    //due to my last 4 digit number is 0677 so i change 0 -> 3 (which is my
first digit number of student ID)
    //student ID : 3190300677
    network.Listen(3677);
    cout << "[Server] Tiny Web Server Begin" << endl;
    cout << "[Server] Waiting For Request..." << endl;
    while(1){ //non stop loop for accept request and handle the request
        connfd = network.Acccept(); //accept request
        //then handle request
        RequestHandler rh(connfd);
        cout << "[Server] Connection Established! Socket ID : " << connfd <<
endl;
        rh.Begin();
        network.Close();
    }
```

- 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）

处理子线程的方法是 void handleRequest(void *arg)，传入的是 connfd(connection file description)，同时也读取发送来的请求信息，读取的分别有 method、uri 以及 version，根据获取到的请求方法（GET/POST）去执行个别方法的操作有 handleGet 和 handlePost 方法。处理完请求后就退出子线程回到主线程。

```
void* handleRequest(void *arg){
    char buf[MAXBUFFER], uri[MAXLINE], version[MAXLINE], method[MAXLINE];
    rio_t rio; //struct rio
    int connfd = *(int *)arg;
    rio_readinitb(&rio, connfd); //read buffer and reset buffer associate with
fd
    rio_readlineb(&rio, buf, MAXLINE); //read a text line (buffered)
    sscanf(buf, "%s %s %s", method, uri, version);
//output to see for debug
    cout << "[Request Method] " << method << endl;
    cout << "[Request URI] " << uri << endl;
    cout << "[Request Version] " << version << endl;
```

```
        if(!strcasecmp(method, "GET")){
          //cmp == 0 , equal to method GET
          handleGet(connfd, uri, &rio);
          pthread_exit(NULL);
        }
        else if(!strcasecmp(method, "POST")){
          //cmp == 0 , equal to method POST
          handlePost(connfd, uri, &rio);
          pthread_exit(NULL);
        }
        else{
              //服务器不支持请求的方法
          clientError(connfd, "501", "Not Implemented","This Method of Request
is Not Accepted By This Server");
          pthread_exit(NULL);
        }
}
```

- 服务器运行后，用 netstat –an 显示服务器的监听端口

  当启动 webserver 时，监听的端口是 3677（学号后四位是 0677，由于有‘0’所以就把‘0’改为学号第一位的‘3’，学号：3190300677）。

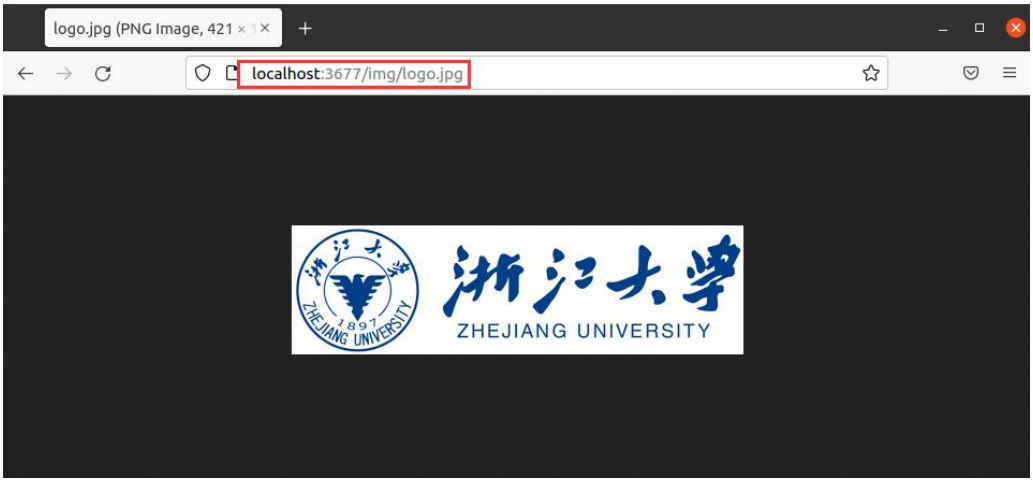
```
peter@peter-virtual-machine:~/cn/lab3$ ./webserver
[Server] Tiny Web Server Begin
[Server] Waiting For Request...
```

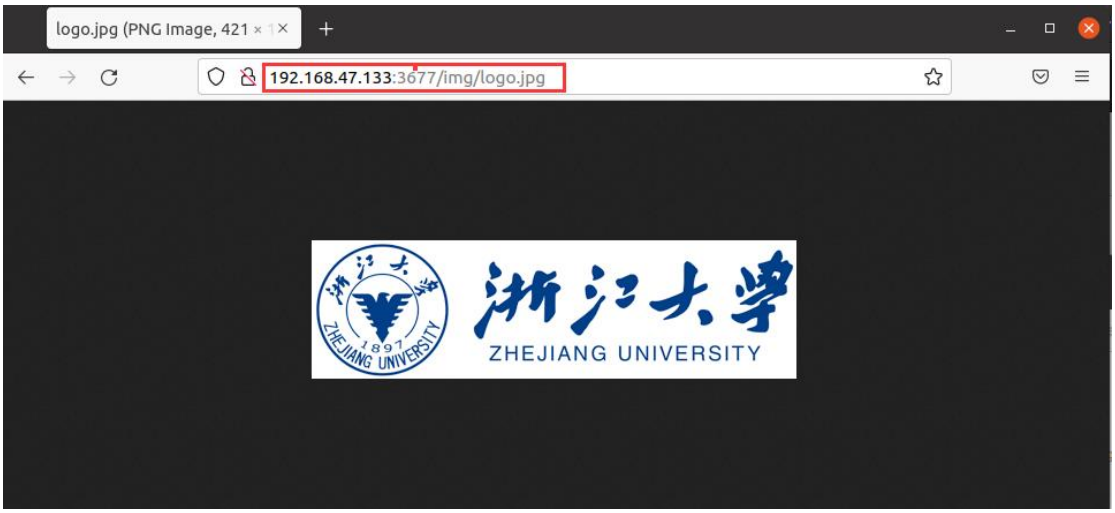
```
peter@peter-virtual-machine:~$ netstat -an | grep 3677
tcp        0      0 0.0.0.0:3677            0.0.0.0:*               LISTEN
```

- 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。

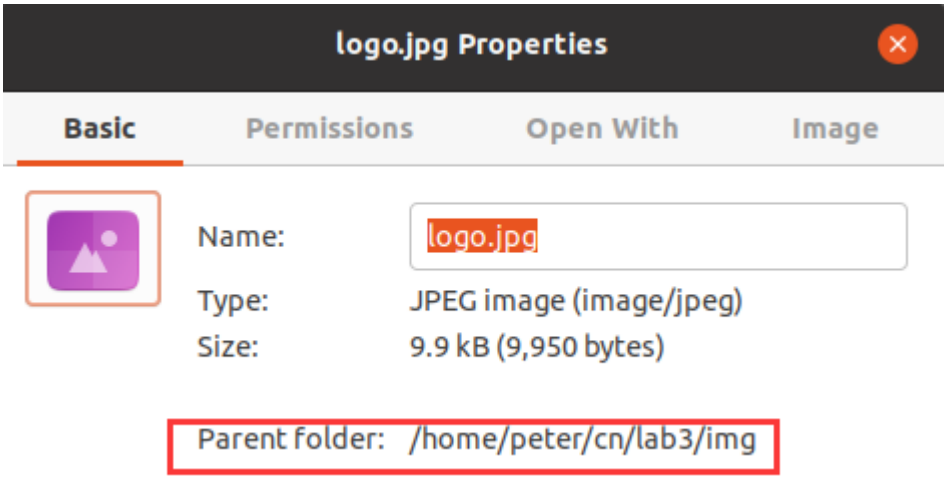
  浏览器访问 URL：localhost:3677/img/logo.jpg（本地访问）

  和 192.168.47.133:3677/img/logo.jpg（IPv4 地址）





服务器上文件实际存放的路径：



服务器上文件实际存放路径：/home/peter/cn/lab3/img

服务器的相关代码片段：

```
void handleGet(int connfd, char* uri, rio_t * rio){
    char buf[MAXBUFFER], version[MAXLINE],filename[MAXLINE],cgi_args[MAXLINE];
    int isStatic;
    struct stat sbuf;
    readRequestHeader(rio);
    //parse uri from get request
    isStatic = parseURI(uri, filename, cgi_args);
    cout << "[Request] GET Filename : " << filename << endl << endl;//debug
use
    if(stat(filename, &sbuf) < 0){
        clientError(connfd, "404", "Not Found", "Web Server Could Not Find
This File");
        return;
    }
```

```
      if(isStatic){
        //static
        if( !(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode) ){
            //is not regular file or is not owner execute permission
            clientError(connfd, "403", "Forbidden", "Web Server Could Not Read
The File");
            return;
        }
        //else serve static
        serve_static(connfd, filename, sbuf.st_size);
    }
    else{
        //dynamic
        if( !(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode) ){
            clientError(connfd, "403", "Forbidden", "Web Server Could Not Run
The CGI Program");
            return;
        }
        serve_dynamic(connfd, filename, cgi_args);
    }
}
```

handleGet 函数是处理 GET 请求，通过读取请求（readRequestHeader 函数）以及解析 URL 然后发送回应（Response）回给客户端。

```
void readRequestHeader(rio_t *rp){
    char buf[MAXBUFFER];
    rio_readlineb(rp, buf, MAXLINE);
    while(strcmp(buf, "\r\n")){
        //stop when header == \r\n
        rio_readlineb(rp, buf, MAXLINE);
        cout << "[Request Header] " << buf;
    }
}
```

readRequestHeader 函数读取请求包头

```
int parseURI(char* uri, char* filename, char* cgi_args){
    char *ptr;
    if(!strstr(uri, "cgi-bin")){
        //string cgi-bin is not inside uri (Static)
        strcpy(cgi_args, "");
        strcpy(filename, ".");
        strcat(filename, uri);
        if(uri[strlen(uri)-1] == '/'){ //if end with '/' then default
home.html filename add at behind
            strcat(filename, "home.html");
        }
        return 1;
```

```
        }
        else{
          //cgi-bin is in uri (dynamic)
          ptr = index(uri, '?');
          if(ptr){
              strcpy(cgi_args, ptr+1);
              *ptr = '\0';
          }
          else{
              strcpy(cgi_args, "");
          }
          strcpy(filename, ".");
          strcat(filename, uri);
          return 0;
      }
}
```

parseURI 函数解析请求 GET，护球相关的文件名信息。

```
//send a http response, body include local file
void serve_static(int connfd, char* filename, int filesize){
    char buf[MAXBUFFER], filetype[MAXLINE];
    getFileType(filename, filetype);
    //send response header to client
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Web Server\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, filesize);
    sprintf(buf, "%sContent-type: %s\r\n\r\n",buf, filetype);
    rio_writen(connfd, buf, strlen(buf));
    //send response body to client
    char *srcptr;
    int srcfd;
    srcfd = open(filename, O_RDONLY, 0); //open for reading only
    srcptr = (char *)Mmap(0, filesize, PROT_READ, MAP_PRIVATE, srcfd, 0);
//creates a new mapping in the virtual address space of the calling process
    //Call mmap to map the first filesize bytes of file srcfd to a private
read-only memory area starting at address srcptr
    Close(srcfd);
    rio_writen(connfd, srcptr, filesize); //Do the file transfer to client,
copy filesize bytes from the srcptr position to the client's connected fd
    Munmap(srcptr, filesize); //Free the mapped virtual memory area to avoid
memory leaks
}
```

serve_static 函数是发送 HTTP 回应和主体给浏览器

```
// Fork a child process and run a CGI program in the context of
// the child process to serve various types of dynamic content
void serve_dynamic(int connfd, char* filename, char* cgi_args){
    char buf[MAXBUFFER];
```

```
    char *emptyList[] = { NULL };
    //send response header
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    rio_writen(connfd, buf, strlen(buf));
    sprintf(buf, "Server: Web Server\r\n");
    rio_writen(connfd, buf,strlen(buf));
    //fork child
    if(Fork() == 0){ // child process made
        setenv("QUERY_STRING", cgi_args, 1); //change or add environment
variable
        Dup2(connfd, STDOUT_FILENO); //oldfd - newfd, redirect stdout to
client
        Execve(filename, emptyList, environ); // execute CGI program
    }
    Wait(NULL); //parent wait and repeat child
}
```

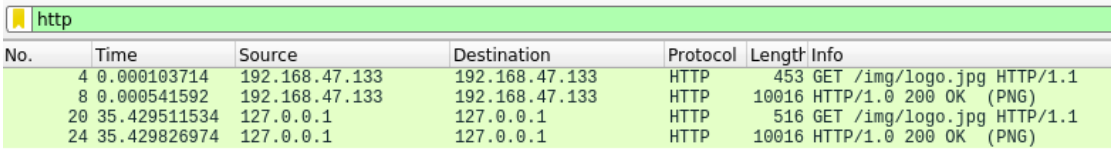
serve_static 函数是派生一个子进程并在子进程的上下文中运行一个 CGI 程序来提供各种类型的动态内容。

```
void getFileType(char* filename, char* filetype){
    if(strstr(filename, ".html")){
        strcpy(filetype, "text/html");
    }
    else if(strstr(filename,".gif")){
        strcpy(filetype, "image/gif");
    }
    else if(strstr(filename,".jpg")){
        strcpy(filetype, "image/jpeg");
    }
    else{
        //plain text
        strcpy(filetype, "text/plain");
    }
}
```

getFileType 函数是获取文件的类型比如 text/html/jpg/gif 等。

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

Wireshark 抓取数据包（HTTP 协议部分，分别有 IPv4 地址和本地 localhost 地址）



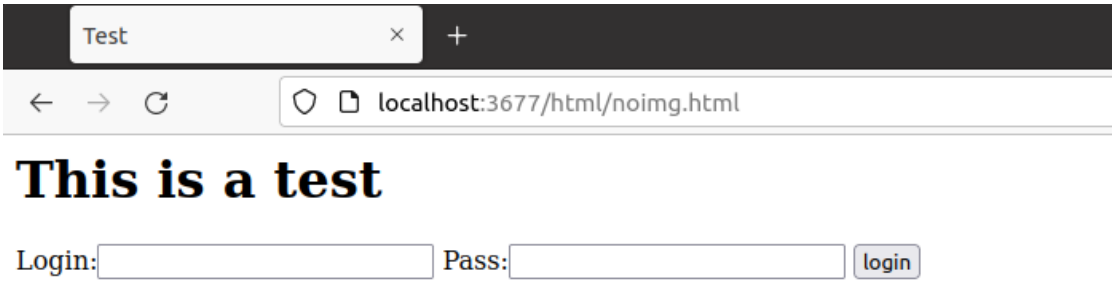
GET 请求的数据包，有 Method、URI 以及 Version

```
Hypertext Transfer Protocol
▾ GET /img/logo.jpg HTTP/1.1\r\n
  ▸ [Expert Info (Chat/Sequence): GET /img/logo.jpg HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /img/logo.jpg
    Request Version: HTTP/1.1
  Host: 192.168.47.133:3677\r\n
```

处理 GET 请求后的 Response 回应给浏览器客户

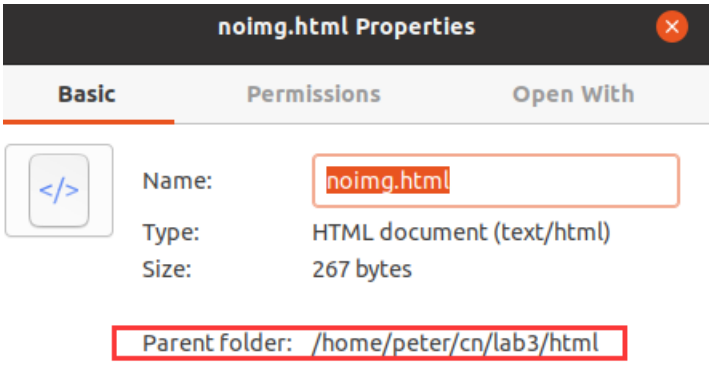
```
Hypertext Transfer Protocol
▾ HTTP/1.0 200 OK\r\n
  ▸ [Expert Info (Chat/Sequence): HTTP/1.0 200 OK\r\n]
    Response Version: HTTP/1.0
    Status Code: 200
    [Status Code Description: OK]
    Response Phrase: OK
  Server: Web Server\r\n
▸ Content-length: 9950\r\n
  Content-type: image/jpeg\r\n
  \r\n
  [HTTP response 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：

存放路径：/home/peter/cn/lab3/html



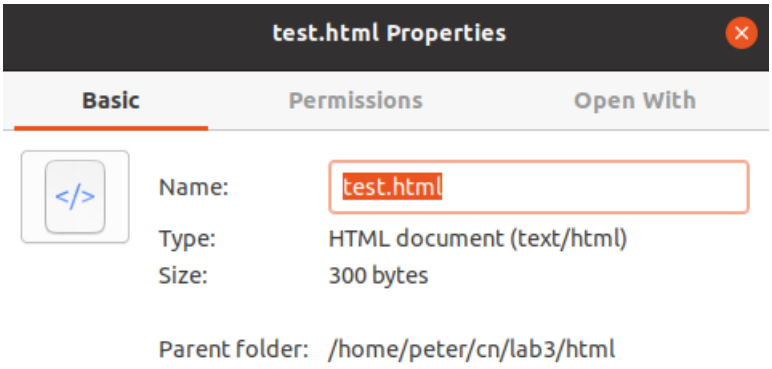
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML 内容）：

```
89 353.229926833 127.0.0.1          127.0.0.1          HTTP      519 GET /html/noimg.html HTTP/1.1
93 353.230713859 127.0.0.1          127.0.0.1          HTTP      333 HTTP/1.0 200 OK  (text/html)
```

- 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。

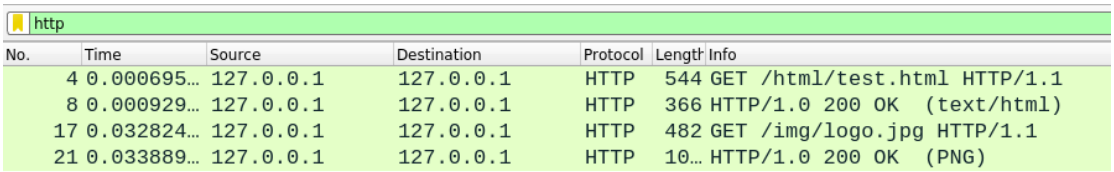
服务器上文件实际存放的路径：

存放路径：/home/peter/cn/lab3/html

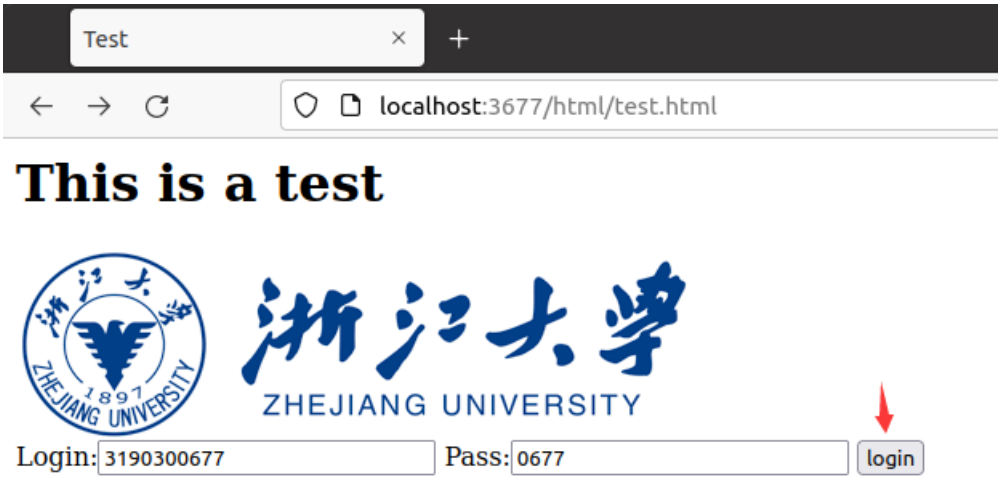


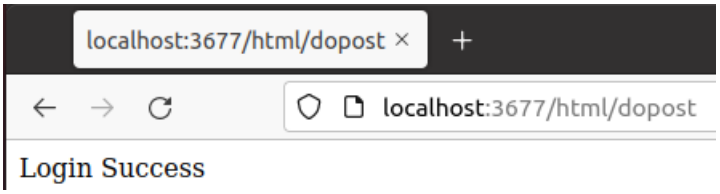
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：

先 GET 请求获取 test.html 文件然后再获取 logo.jpg 图片，由服务器返回回应请求。



- 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。





正确的登录名和密码：3190300677 和 0677

服务器相关处理代码片段：

```
void handlePost(int connfd, char* uri, rio_t * rio){
    char buf[MAXBUFFER];
    bool Login = false;
    cout << "[POST URI] : " << uri << endl;
    if(strcasecmp(uri, "/html/dopost") || strcasecmp(uri,"/dopost") < 0){
        clientError(connfd, "404", "Not Found", "This URI is Not Available In
This Server");
        return ;
    }
    else{ // uri complete
        rio_readlineb(rio, buf, MAXBUFFER);
        cout << "[Server] Header : " << buf << endl; // debug
        int contentLen;
        //reading header
        while(strcmp(buf, "\r\n")){
            //keep compare until \r\n only
            if(strstr(buf, "Content-Length:")){
                //get the pointer where contentlength at
                sscanf(buf+strlen("Content-Length:"), "%d", &contentLen);
//content length integer give to contentLen
            }
            rio_readlineb(rio, buf, MAXBUFFER);
            cout << buf; //debug
        }
        cout << "[Server] Reading Header Complete" << endl;
```

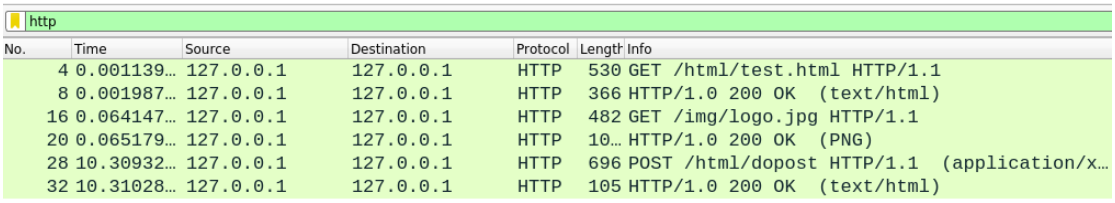
```
        //get body
        rio_readlineb(rio, buf, contentLen+1);
        cout << "[Server] Content Body : " << buf << endl;
        char login[MAXLINE], pass[MAXLINE];
        if(strstr(buf, "login=") && strstr(buf,"pass=")){
            char *ptr = strstr(buf, "login=");
            int l=0,p=0;
            while(*ptr++ != '='); //point after '=' then stop
            while(*ptr != '&'){
                //while ptr not point to '&'
                login[l++] = *ptr++; //get the login string
            }
            login[l] = 0;
            //same function use at password
            while(*ptr++ != '=');
            while(*ptr) pass[p++] = *ptr++;
            pass[p] = 0;
        }
        //check login and password correct or not,
        //login ID is student id 3190300677 and password is last 4 digit
number 0677
        cout << "Login ID : " << login << " , Password : " << pass << endl;
        if(!strcmp(login, "3190300677") && !strcmp(pass,"0677")) Login = true;
        else Login = false;
        postResponse(connfd, Login);
    }
}
```

handlePost 函数是处理 POST 请求的，从客户端浏览器读取 POST Request 后然后进行处理并发送 Response 回应给客户端浏览器。在处理过程中，服务器会判断用户所提交的登录名和密码是否都正确然后再跳转到/dopost 地址返回对应文字

```
void postResponse(int connfd, bool login){
    char buf[MAXBUFFER];
    char *successResponse = "<html><body>Login Success</body></html>";
    char *failResponse = "<html><body>Login Fail</body></html>";
    cout << "Login Success" << endl;
    //send header
    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Web Server\r\n", buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, (login ?
strlen(successResponse) : strlen(failResponse)) );
    sprintf(buf, "%sContent-type: text/html\r\n\r\n",buf);
    //Login successful
    rio_writen(connfd, buf, strlen(buf));
    if(login) rio_writen(connfd, successResponse, strlen(successResponse));
    else rio_writen(connfd, failResponse, strlen(failResponse));
}
```

postResponse 函数是发送 Response 回给客户端

Wireshark 抓取的数据包截图（HTTP 协议部分）



http

No.	Time	Source	Destination	Protocol	Length	Info
4	0.001139…	127.0.0.1	127.0.0.1	HTTP	530	GET /html/test.html HTTP/1.1
8	0.001987…	127.0.0.1	127.0.0.1	HTTP	366	HTTP/1.0 200 OK (text/html)
16	0.064147…	127.0.0.1	127.0.0.1	HTTP	482	GET /img/logo.jpg HTTP/1.1
20	0.065179…	127.0.0.1	127.0.0.1	HTTP	10…	HTTP/1.0 200 OK (PNG)
28	10.30932…	127.0.0.1	127.0.0.1	HTTP	696	POST /html/dopost HTTP/1.1 (application/x…
32	10.31028…	127.0.0.1	127.0.0.1	HTTP	105	HTTP/1.0 200 OK (text/html)

POST 请求

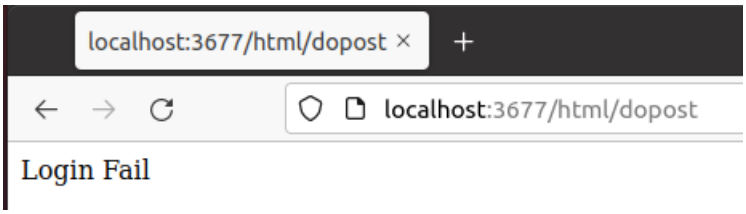
```
POST /html/dopost HTTP/1.1\r\n
  [Expert Info (Chat/Sequence): POST /html/dopost HTTP/1.1\r\n]
  Request Method: POST
  Request URI: /html/dopost
  Request Version: HTTP/1.1
Host: localhost:3677\r\n
```

POST Response

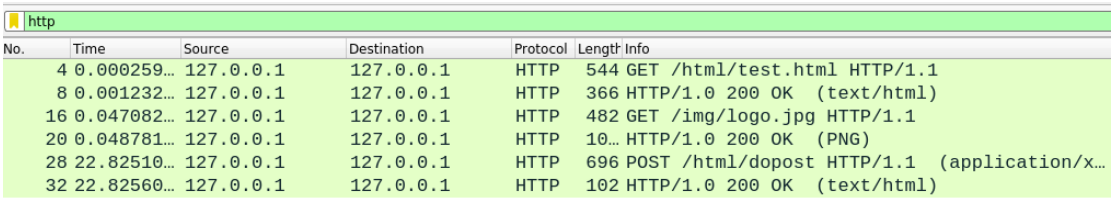
```
HTTP/1.0 200 OK\r\n
  [Expert Info (Chat/Sequence): HTTP/1.0 200 OK\r\n]
  Response Version: HTTP/1.0
  Status Code: 200
  [Status Code Description: OK]
  Response Phrase: OK
Server: Web Server\r\n
Content-length: 39\r\n
Content-type: text/html\r\n
\r\n
[HTTP response 1/1]
```

- 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。

输入错误的登录名和密码后显示“Login Fail”表示登录失败



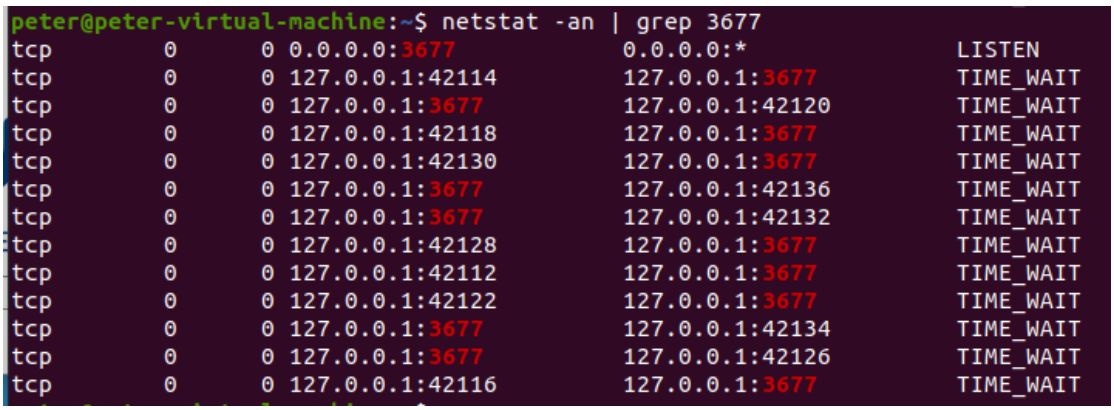
Wireshark 抓取的数据包截图（HTTP 协议部分）



- 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）

有多个连接对 3677 端口进行了连接监听，然后进入 TIME_WAIT 阶段。



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？

  头部与体部用\r\n 来进行分隔的。

- 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？

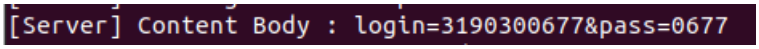
  根据头部的字段 Content-type 内容来判断文件的类型。

- 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，但体部不一定都是，要根据 Content-type 的内容来缺点，此次的实验中就包含了.jpg/.html 等的格式

- 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？

  数据是放在体部，两个字段之间使用‘&’符号来连接起来的。



七、 讨论、心得

在执行本次实验前，我参考了一个博主的文章，是关于搭建网络服务器的解析，学习了网络服务器的基本实施方法，对 GET 和 POST 请求有了更深入的理解。这次的实验算是在实验 2 的基础上添加了在浏览器上展现服务器和客户端的交互方法，实验 2 只是通过终端(terminal)在交互，而此实验是通过浏览器。本次实验参考且应用了 csapp 的库并使用 Rio 函数包来对浏览器与服务器进行交互，有了这个库可以让我们更方便地执行 I/O，所以这个库对本次实验的帮助很大。所以本次的实验难度不算大，是在实验 2 的基础上进行升级，主要是要解析请求的头部与体部然后再回应给浏览器（传输数据）。通过这次的实验，学会了浏览器与服务器的交互以及解析请求的数据包。

参考链接：

【1】 https://blog.csdn.net/qq_20480611/article/details/50982688 CSAPP Tiny web 服务器源码分析及搭建运行

【2】 https://www.cnblogs.com/cknightx/p/7518085.html 健壮的网络编程 IO 函数-RIO

包

【3】 https://www.softwaretestinghelp.com/cpp-makefile-tutorial/ Makefile 例子