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：**

```
laylalaisyc@ubuntu:~$ uname -a
Linux ubuntu 4.13.0-39-generic #44~16.04.1-Ubuntu SMP Thu Apr 5 16:43:10 UTC 201
8 x86_64 x86_64 x86_64 GNU/Linux
```

- **编译方法：**

1. 进入文件夹 p3_http 目录；
2. 输入：make clean
3. 输入：make
4. 输入：./server

- 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）

```
 int main(int argc, char** argv)
{
    NetworkManager nm;
    int connfd;

    /* Start listening on port 1224 */
    nm.Listen(1224);
    std::cout << "Web Server started.. Waiting for request" <<
std::endl;

    /* Infinity loop on accepting request and handling */
    while (1)
    {
        connfd = nm.Accept();
        RequestManager rm(connfd);
        std::cout << "Connection Established, socket id = " << connfd
<< std::endl;
        rm.Run();
        nm.Close();
    }

    return 0;
}
```

在服务器主线程中，我们使用 C++面向对象编程，将网络的连接部分包装成了 NetworkManager 类，在主线程的开始，我们实例化一个对象，并监听 1224 接口。

随后，进入主循环，并实例化一个 RequestManager，并传入连接的文件描述符。随后，RequestManager 进行 run 方法，进行对客户端 GET 或 POST 方法的相应，之后网络关闭这个连接。

- 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）

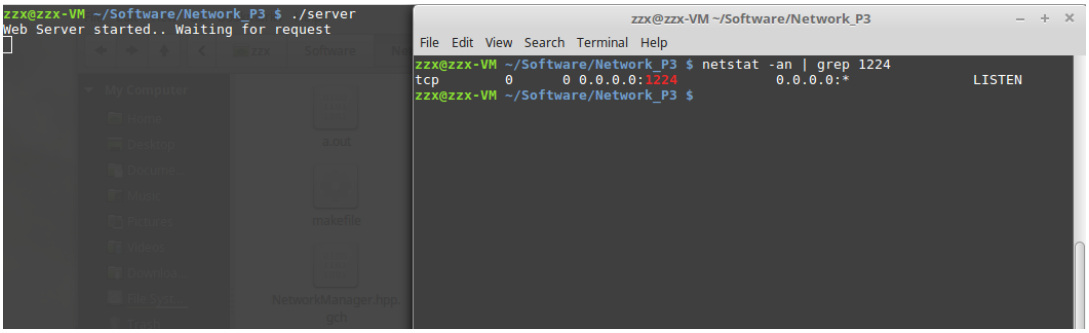
```
 void* do_request(void * arg)
{
    char buf[MAX_LINE], method[MAX_LINE], uri[MAX_LINE],
version[MAX_LINE];
    rio_t rio;
    int connfd = *(int *)arg;
    rio_readinitb(&rio, connfd);
    rio_readlineb(&rio, buf, MAX_LINE);
    sscanf(buf, "%s %s %s", method, uri, version);

    /* Debug */
    std::cout << method << std::endl;
    std::cout << uri << std::endl;
```

```
    std::cout << version << std::endl;
    if (!strcasecmp(method, "GET"))
    {
        HandleGet();
        pthread_exit(NULL);
    }
    else if (!strcasecmp(method, "POST"))
    {
        HandlePost(connfd, uri, &rio);
        pthread_exit(NULL);
    }
    else
    {
        iClient_error(connfd, "501", "Not implemented", "This
kind of request is not supported in this server");
        pthread_exit(NULL);

    }
}
```

子线程是 do_request, 接受连接的 file descriptor, 同时读取发送来的信息，读取 method, uri, 以及 version，根据获取到请求行的 GET、POST 等不同指令进行不同的操作。

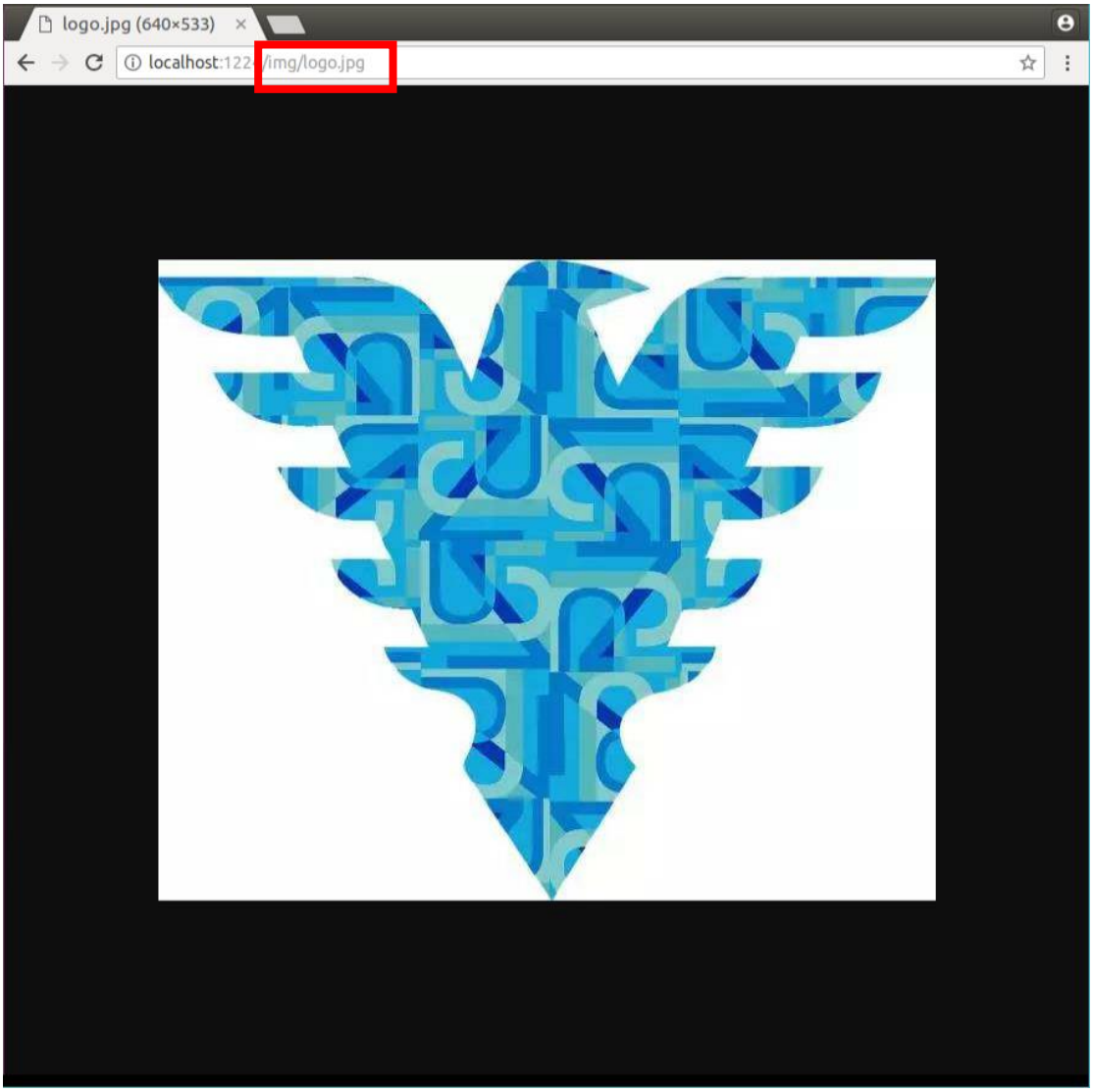
操作结束之后退出线程，回到主线程。

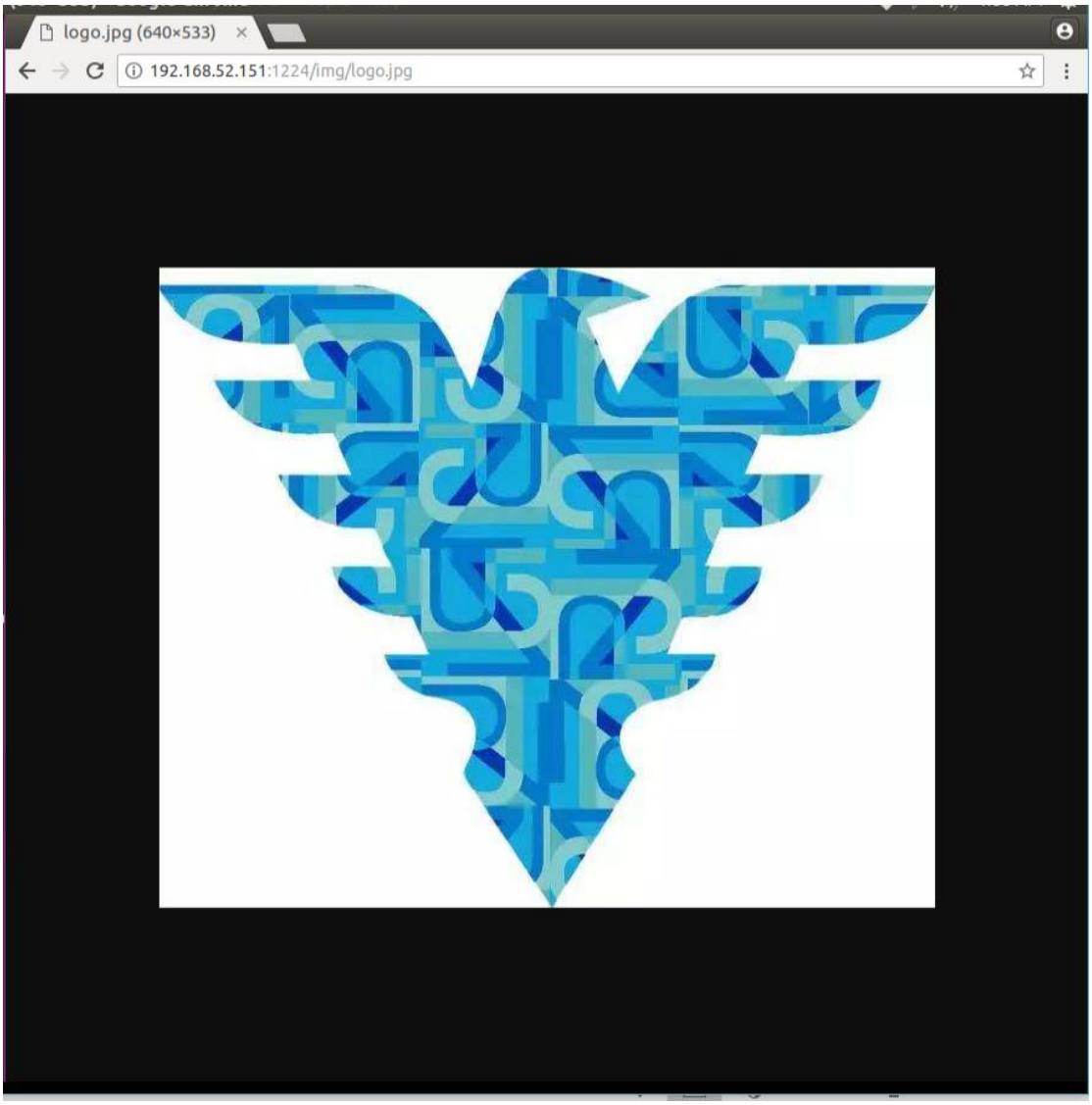
- 服务器运行后，用 netstat –an 显示服务器的监听端口



可以看到，当我们启动 server 时，监听的确实是 1224 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。


logo.jpg (640×533)
localhost:122
/img/logo.jpg



服务器上文件实际存放的路径：

在服务器上文件实际存放的位置在/dir/img 文件夹中，但是 uri 中的位置只在/img 路径下。

服务器的相关代码片段：

```
/* HandleGet Function: handle GET request from client */
void HandleGet(int connfd, char* uri, rio_t* rio)
{
```

```
    struct stat sbuf;
    char buf[MAX_LINE], version[MAX_LINE];
    char filename[MAX_LINE];

    // read request
    read_requesthdrs(rio);

    // parse URI from GET request
    parse_uri(uri, filename);

    // check if file exist
    if(stat(filename, &sbuf) < 0){
        iClient_error(connfd, "404", "Not found", "Cannot find
this file");
        return;
    }
    // static content: check if is readable
    if (!(S_ISREG(sbuf.st_mode)) || !(S_IXUSR & sbuf.st_mode))
{
        iClient_error(connfd, "403", "Forbidden", "Cannot read
the file");
        return;
    }

    get_response(connfd, filename, sbuf.st_size);
}
```

HandleGet 为处理 GET 的主要函数，完成从客户读取 request 到发送 response 给客户的所有过程。

```
* read_requesthdrs:  only read requst head  */
static void read_requesthdrs(rio_t *rp){
    char buf[MAX_LINE];

    rio_readlineb(rp, buf, MAX_LINE);    // read request to
buffer
    while(strcmp(buf, "\r\n")){          // end of request head
        rio_readlineb(rp, buf, MAX_LINE);
        printf("%s", buf);            // print request head
    }
    return;
}
```

read_requesthdrs 该函数用来读取用户 request 包头。

```
/* parse_uri:
```

```
    假设静态内容的主目录就是当前目录；可执行文件的主目录是./cgi-
bin/，任何包含该字符串的 URI 都认为是对动态内容的请求
    将 URI 解析为一个文件名和一个可选的 CGI 参数字符串
*/
static void parse_uri(char *uri, char *filename){
    char *ptr;

    strcpy(filename, "./dir");                  // begin
convert
    strcat(filename, uri);                   // end convert
    if(uri[strlen(uri)-1] == '/'){           // slash check: if
'/', show default file
        strcat(filename, "html/test.html"); // append default
    }
    return;
}
```

parse_uri 完成请求的解析，从而获得相关文件名等信息。

```
/* get_filetype:
    derive file type from file name
*/
void get_filetype(char *filename, char *filetype){
    if(strstr(filename, ".html")){           // text/html
        strcpy(filetype, "text/html");
    }
    else if(strstr(filename, ".jpg")){       // image/jpeg
        strcpy(filetype, "image/jpeg");
    }
    else{
        strcpy(filetype, "text/plain");
    }
}
```

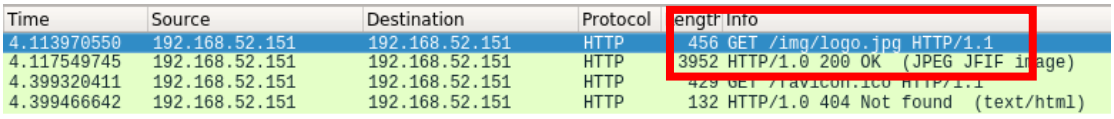
get_filetype 用来获取客户要求的文件类型。

```
/* get_response: send response header from server to client */
/* refer to csapp */
void get_response(int fd, char *filename, int filesize){
    int srcfd;
    char *srcp, filetype[MAX_LINE], buf[MAX_LINE];

    // get filetype
    get_filetype(filename, filetype);

    // send response head
    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(fd, buf, strlen(buf));

    // send response body to client
    srcfd = Open(filename, O_RDONLY, 0); // open
    srcp = (char *)Mmap(0, filesize, PROT_READ, MAP_PRIVATE,
srcfd, 0); // mmap: 将被请求文件映射到一个虚拟存储器空间，

    // 调用 mmap 将文件 srcfd 的前 filesize 个字节映射到

    // 一个从地址 srcp 开始的私有只读存储器区域
    rio_writen(fd, srcp, filesize);          // 执行到客户端的实
际文件传动，拷贝从 srcp 位置开始的 filesize 个字节到客户端的已连接描
述符

    Close(srcfd);
    Munmap(srcp, filesize);                  // 释放了映射的虚拟
存储器区域，避免潜在的存储器泄露
}
```

get_response 为完成解析后，服务器向客户端发送相关信息。

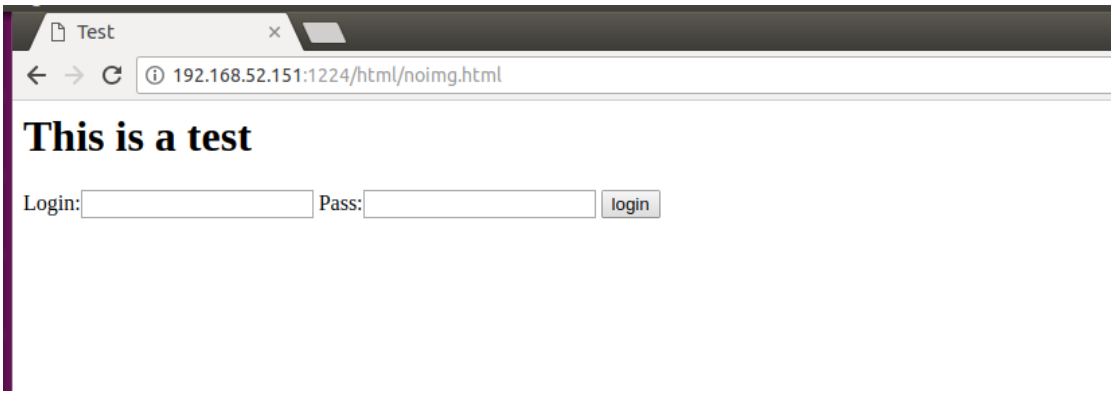
Wireshark 抓取的数据包截图（通过跟踪 TCP 流，只截取 HTTP 协议部分）：



Time	Source	Destination	Protocol	engtr	Info
4.113970550	192.168.52.151	192.168.52.151	HTTP	456	GET /img/logo.jpg HTTP/1.1
4.117549745	192.168.52.151	192.168.52.151	HTTP	3952	HTTP/1.0 200 OK (JPEG JFIF image)
4.399320411	192.168.52.151	192.168.52.151	HTTP	429	GET /favicon.ico HTTP/1.1
4.399466642	192.168.52.151	192.168.52.151	HTTP	132	HTTP/1.0 404 Not found (text/html)

在本机上进行实验，从中可以看到当客户端及浏览器请求图片时，会 request 相关内容。

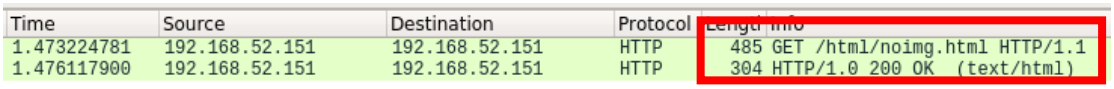
- 浏览器访问只包含文本的 HTML 文件时，浏览器的 URL 地址和显示内容截图。



服务器文件实际存放的路径：

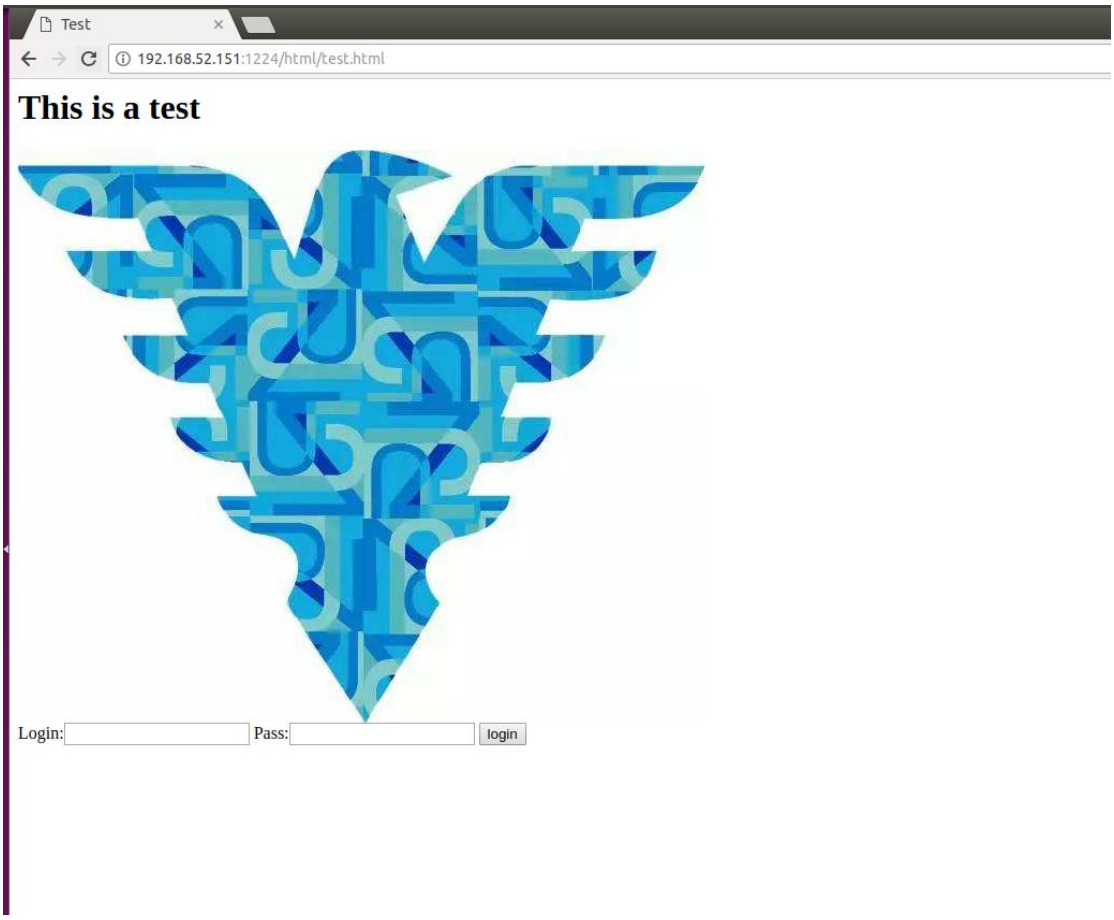
在服务器上文件实际存放的位置在/dir/html 文件夹中，但是 uri 中的位置只在/html 路径下。

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

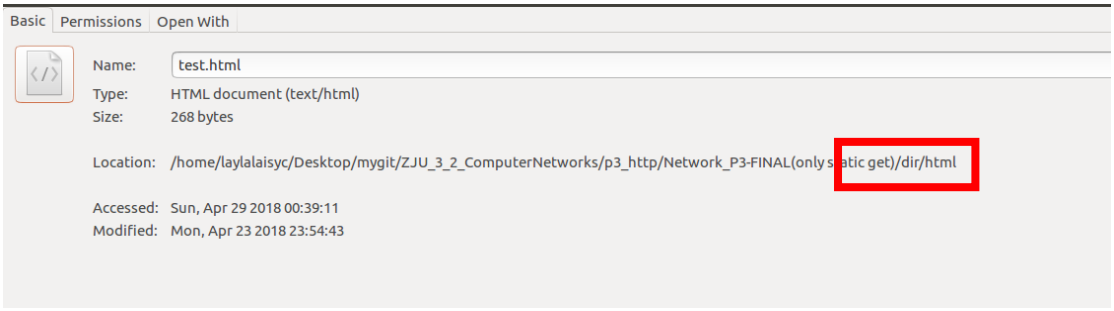


在本机上进行实验，从中可以看到当客户端及浏览器请求 html 时，会 request 相关内容。

- 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。

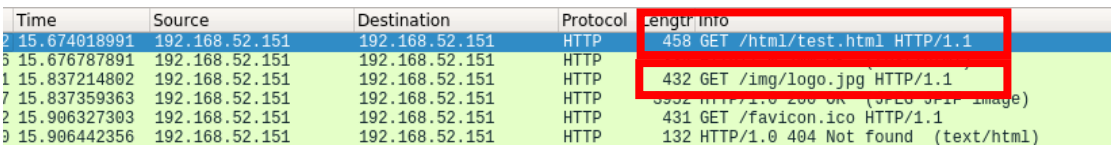


服务器上文件实际存放的路径：



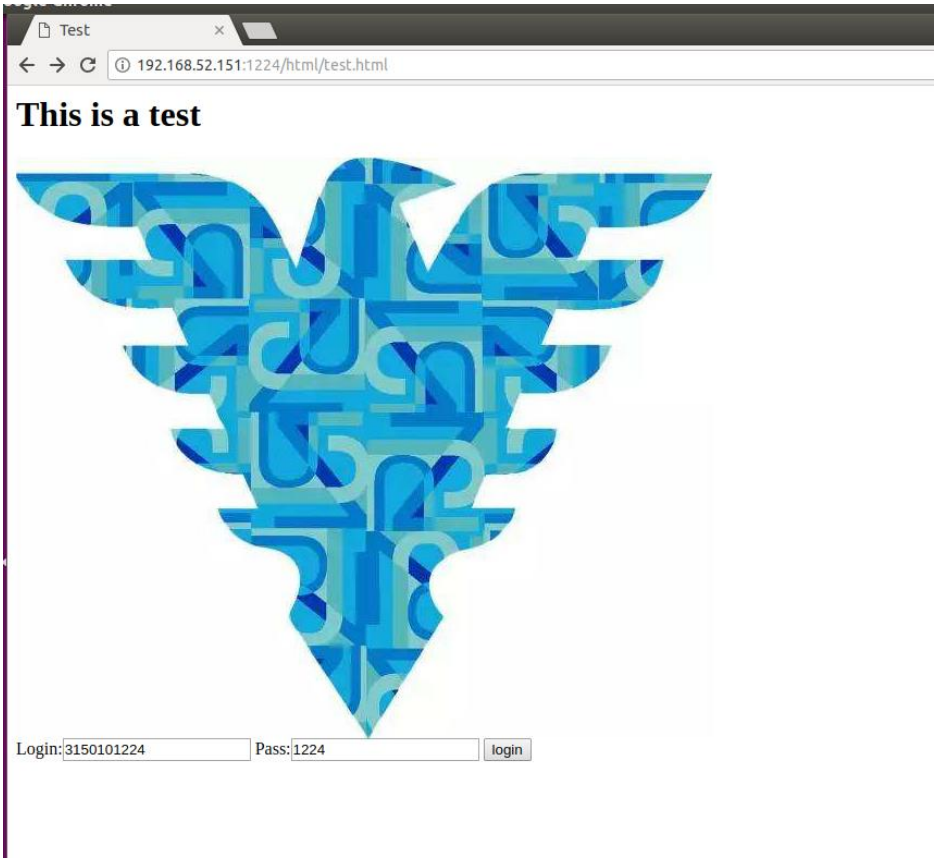
在服务器上文件实际存放的位置在/dir/html 文件夹中，但是 uri 中的位置只在/html 路径下。

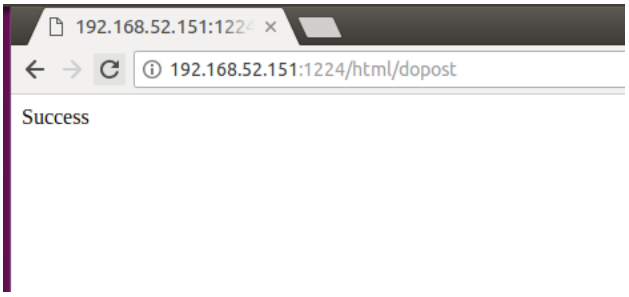
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：



在本机上进行实验，从中可以看到当客户端及浏览器请求 html 时，会 request 相关内容。

- 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。





正确的登录名和密码为：

Login：3150101224

Pass：1224

服务器相关处理代码片段：

```
void HandlePost(int connfd, char * uri, rio_t* rio)
{
    char buf[MAX_LINE];
    bool log = false;
    if (strcasecmp(uri, "/html/dopost") && strcasecmp(uri, "/dopost"))
    {
```

```
            iClient_error(connfd, "404",    "Not found", "This kind of resource is not
available in this server");
            return;
        }
        else
        {
            int contentlength;
            rio_readlineb(rio, buf, MAX_LINE);
            std::cout << buf;

            /* Read till \r\n appear */
            while (strcmp(buf, "\r\n"))
            {
                if (strstr(buf, "Content-Length:"))
                    sscanf(buf+strlen("Content-Length:"), "%d", &contentlength);
                rio_readlineb(rio, buf, MAX_LINE);
                std::cout << buf;
            }
            std::cout << "Finish reading header" << std::endl;
            /* Now we get the body */
            rio_readlineb(rio, buf, contentlength+1);
            /* Output the body */
            std::cout << "Body:" << buf << std::endl;
            char login[MAX_LINE], pass[MAX_LINE];
            if (strstr(buf, "login=") && strstr(buf, "pass="))
            {
                char * p = strstr(buf, "login=");
                int li = 0, pi = 0;
                while (*p++ != '=');
                while (*p != '&')
                {
                    login[li++] = *p++;
                }
                login[li] = 0;
                while (*p++ != '=');
                while (*p)
                {
                    pass[pi++] = *p++;
                }
                pass[pi] = 0;
            }
            /* Login status check */
            std::cout << "login = " << login << "pass = " << pass << std::endl;
            if (!strcmp(login, "3150101224") && !strcmp(pass, "1224"))
```

```
                log = true;
            else
                log = false;

            /* Return status */
            post_response(connfd, log);
        }
}
```

HandlePost 为处理 POST 的主要函数，完成从客户读取 request 到发送 response 给客户的所有过程。

```
void post_response(int connfd, bool log)
{
    char buf[MAX_LINE];
    char * successmsg = "<html><body>Success</body></html>";
    char * failmsg = "<html><body>Fail</body></html>";

    sprintf(buf, "HTTP/1.0 200 OK\r\n");
    sprintf(buf, "%sServer: Server ",buf);
    sprintf(buf, "%sContent-length: %d\r\n", buf, log ? strlen(successmsg) :
strlen(failmsg) );
    sprintf(buf, "%sContent-type: text/html\r\n\r\n", buf);

    /* Log in successful */

    rio_writen(connfd, buf, strlen(buf));
    if (log)
    rio_writen(connfd, successmsg, strlen(successmsg));
    else
    rio_writen(connfd, failmsg, strlen(failmsg));

}
```

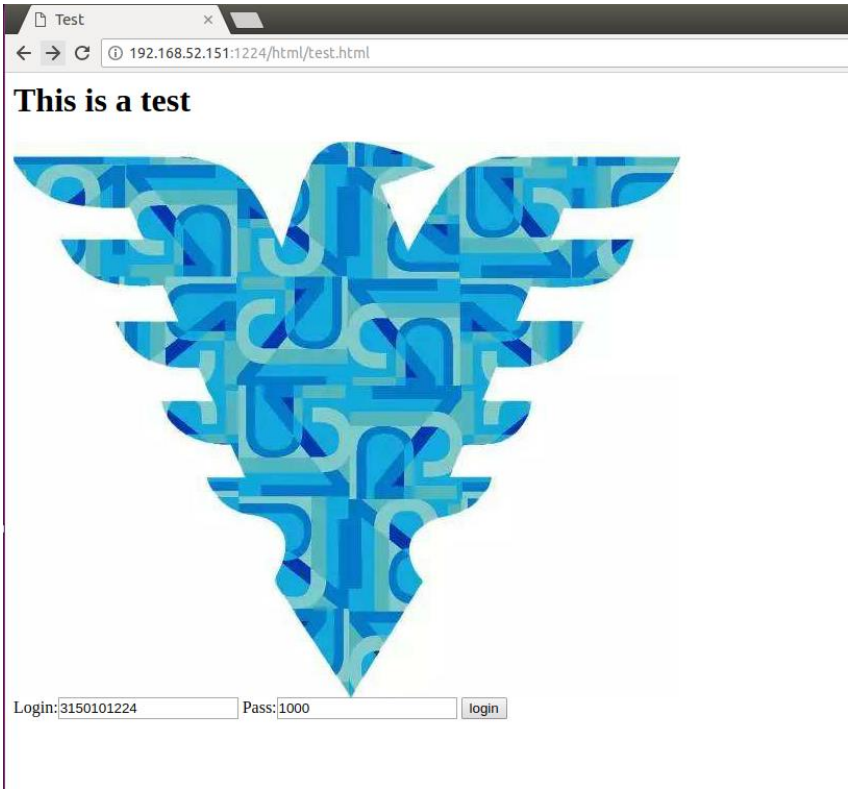
post_response 为完成解析后，服务器向客户端发送相关信息

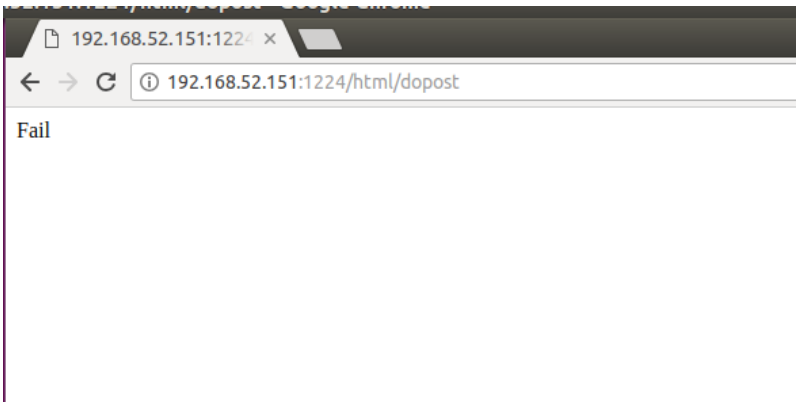
Wireshark 抓取的数据包截图（HTTP 协议部分）

Time	Source	Destination	Protocol	Length	Info
0.000689124	192.168.52.151	192.168.52.151	HTTP	458	GET /html/test.html HTTP/1.1
0.000780620	192.168.52.151	192.168.52.151	HTTP	334	HTTP/1.0 200 OK (text/html)
0.136794026	192.168.52.151	192.168.52.151	HTTP	432	GET /img/logo.jpg HTTP/1.1
0.136968371	192.168.52.151	192.168.52.151	HTTP	3952	HTTP/1.0 200 OK (JPEG JFIF image)
5.553275958	192.168.52.151	192.168.52.151	HTTP	665	POST /html/dopost HTTP/1.1 (application/x-www-form-urlencoded)
5.553452243	192.168.52.151	192.168.52.151	HTTP	66	HTTP/1.0 200 OK (text/html)

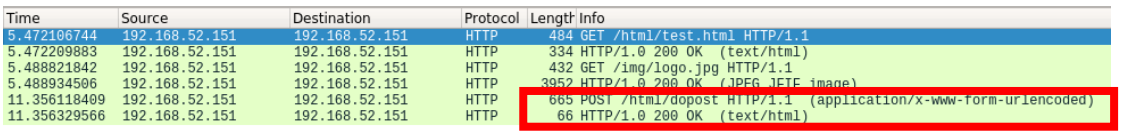
在本机上进行实验，从中可以看到当客户端及浏览器请求 html 并登陆时时，会 request 相关内容并调用 POST。

- 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。





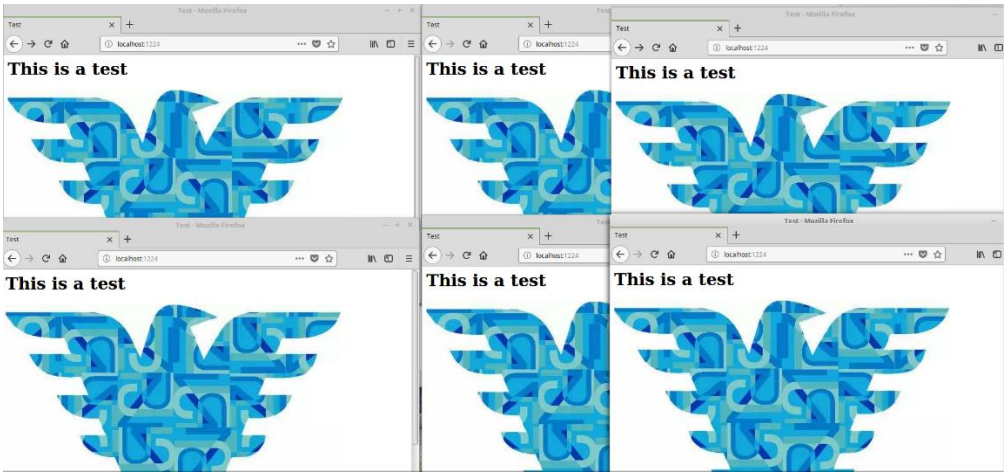
- Wireshark 抓取的数据包截图（HTTP 协议部分）



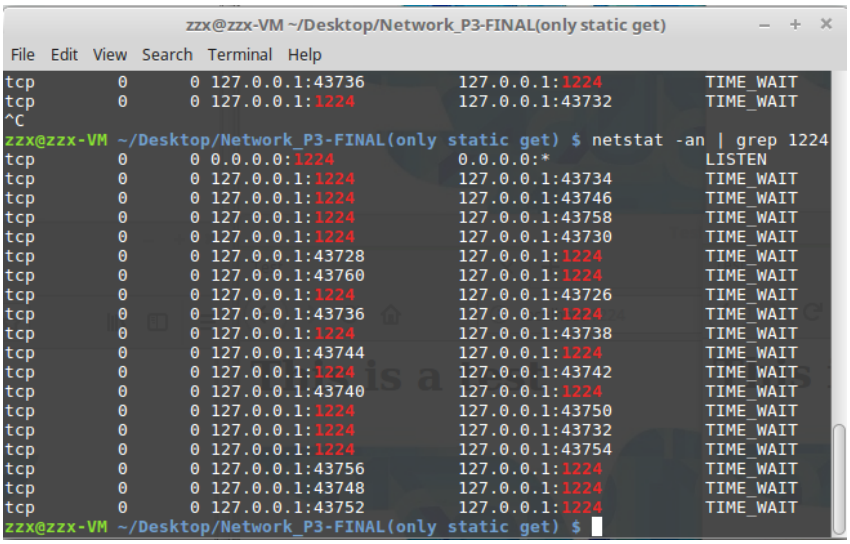
Time	Source	Destination	Protocol	Length	Info
5.472106744	192.168.52.151	192.168.52.151	HTTP	484	GET /html/test.html HTTP/1.1
5.472209883	192.168.52.151	192.168.52.151	HTTP	334	HTTP/1.0 200 OK (text/html)
5.488821842	192.168.52.151	192.168.52.151	HTTP	432	GET /img/logo.jpg HTTP/1.1
5.488934506	192.168.52.151	192.168.52.151	HTTP	3952	HTTP/1.0 200 OK (JPEG JFIF image)
11.356118409	192.168.52.151	192.168.52.151	HTTP	665	POST /html/dopost HTTP/1.1 (application/x-www-form-urlencoded)
11.356329566	192.168.52.151	192.168.52.151	HTTP	66	HTTP/1.0 200 OK (text/html)

在本机上进行实验，从中可以看到当客户端及浏览器请求 html 并登陆时时，会 request 相关内容并调用 POST。

- 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）


This is a test
This is a test
This is a test
This is a test
This is a test
This is a test

- 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）



可以看到，确实有多个连接对 1224 端口进行了连接监听，并进入 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？

  头部与体部用空行\r\n 进行分割。

- 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？

  根据头部字段 Content-type 的内容来判断文件类型。

- 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，要根据 Content-type 来进行确定，在本次实验中包含了.jpg 等其他格式。

- 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？

  放在体部，两个字段之间是用&符号连接起来的。

七、 讨论、心得

本次实验主要是参考并使用了 csapp 库，一开始对 GET 和 POST 的区别理解不够深入，所以有所混淆。后来通过网上查阅资料理解两者的不同：GET 把参数包含在 URL 中，POST 通过 request body 传递参数；GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包。在实验二 socket 的基础上完成本次实验，总体来说觉得难度不是非常大，重点在于对 request 的头部和体部的解析以及传输。