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：

学　　院：

系：

专　　业：

学　　号：

指导教师：

2020 年　11 月　23 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

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）

主线程通过 while 循环不断接受请求，每接收到一个请求就建立一次连接，然后创建子线程进行处理。

```
public static void main(String[] args){
    int port = 5099;
    String root = "..\\root";
    System.out.printf("port: %s\n", ""+port);
    System.out.printf("root: %s\n", root);
    try{
        ServerSocket server_socket = new ServerSocket(port);
        while(true){
            Socket socket = server_socket.accept();
            new HttpServer(socket, root).start();
        }
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

- 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）

子线程即继承了 Thread 的 HttpServer 类，构造函数会通过参数 Socket 对象打开输入输出流，然后通过 path 参数指定根目录。

```
public static class HttpServer extends Thread{
    private InputStream input;
    private OutputStream output;
    private String root;

    HttpServer(Socket socket, String path){
        try{
            input = socket.getInputStream();
            output = socket.getOutputStream();
            root = path;
        }catch(IOException e){
            e.printStackTrace();
        }
    }
```

子进程会调用被重写的 run 方法，先获取请求，得到一个 Request 对象，然后根据请求的类型进行响应。

```
@Override
public void run(){
    // System.out.println("<Start>");
    Request request = getRequest();
    if(request.method.equals("GET")){
        try {
            responseGET(request);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }else if(request.method.equals("POST")){
        try {
            responsePOST(request);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    // System.out.println("<End>");
}
```

如果接收到的请求是 GET 类型的，先检查文件是否存在或者路径是否为空，如果文件不存在或者路径为空，返回 404。

```
void responseGET(Request req) throws IOException {
    String path = req.url;
    File f = new File(root, path);
    System.out.println("A GET request for: " + f.getName());
    Response res = new Response();
    if(path.equals("/") || !f.exists()){
        res.code = 404;
        res.version = "HTTP/1.1";
        res.status = "file not found";
        res.headers.put("Content-Type", "text/html; charset=UTF-8");
        res.headers.put("Content-Length", "22");
        res.data.put("<h1>404 not found</h1>".getBytes());
        output.write(res.headBytes());
        output.write(res.data.array());
        output.flush();
        output.close();
        return;
    }
```

如果请求的文件存在，则根据文件后缀名判断文件类型，并设置响应头里的 Content-Type 字段，然后把文件的内容全部放到响应的数据里。最后将响应以字节流的形式写到输出中。

```
String ext = path.substring(path.lastIndexOf('.') + 1);
InputStream fb = new FileInputStream(f);
res.version = "HTTP/1.1";
res.code = 200;
res.status = "OK";
if(ext.equals("jpg")){
    res.headers.put("Content-Type", "image/jpeg");
}else if(ext.equals("html")){
    res.headers.put("Content-Type", "text/html; charset=UTF-8");
}else{
    res.headers.put("Content-Type", "text/plain; charset=UTF-8");
}

for(int b=fb.read(); b!=-1; b=fb.read()){
    res.data.put((byte)b);
}
fb.close();
output.write(res.headBytes());
output.write(res.data.array());
output.flush();
output.close();
return;
```

如果接收到的请求是 POST 类型，则判断请求的路径是否为 dopost，如果不是，就返回 404。

```
void responsePOST(Request req) throws IOException {
    Response res = new Response();
    System.out.printf("A POST request for: %s\n", req.url);
    if(!req.url.equals("/dopost")){
        res.version = "HTTP/1.1";
        res.code = 404;
        res.status = "file not found";
        res.headers.put("Content-Type", "text/html; charset=UTF-8");
        res.headers.put("Content-Length", "22");
        res.data.put("<h1>404 not found</h1>".getBytes());
        this.output.write(res.headBytes());
        this.output.write(res.data.array());
        this.output.flush();
        this.output.close();
        return;
    }
```

如果请求的路径是 dopost，先设置为默认登录成功。

```
String[] post = req.data.split("&");
Map<String, String> paras = new HashMap<>();
for(int i=0; i<post.length; i++){
    String para = post[i].substring(0, post[i].indexOf('='));
    String val = post[i].substring(post[i].indexOf('=') + 1);
    paras.put(para, val);
}
res.version = "HTTP/1.1";
res.code = 200;
res.status = "OK";
res.headers.put("Content-Type", "text/html; charset=UTF-8");
String msg = "<html>" +
        "<head></head>" +
        "<body>login success :)<body>" +
        "</html>";
```

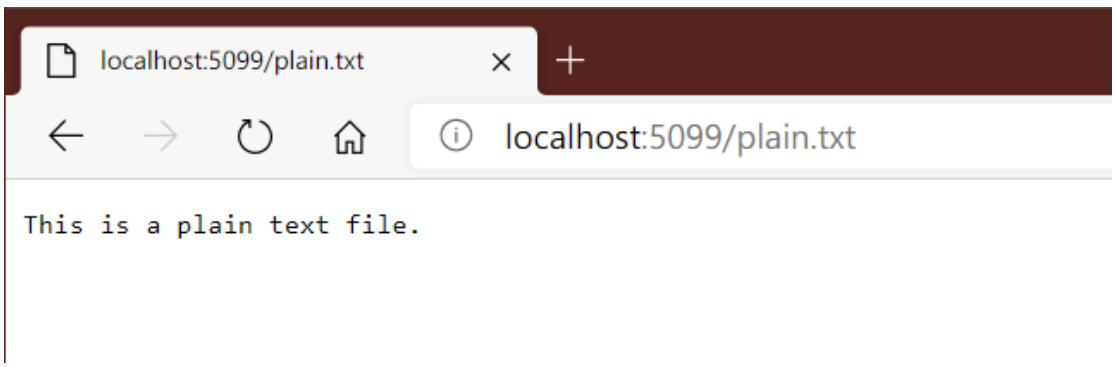
然后从请求数据中获取用户名和密码，如果有缺省，或者用户名或密码不正确，则将响应改为登陆失败。

```
if(!paras.containsKey("login") || !paras.containsKey("pass")
|| !paras.get("login").equals(login) || !paras.get("pass").equals(pass)){
    msg = "<html>" +
            "<head></head>" +
            "<body>login failed :(<body>" +
            "</html>";

}
res.headers.put("Content-Length", ""+msg.length());
res.data.put(msg.getBytes());

output.write(res.headBytes());
output.write(res.data.array());
output.flush();
output.close();
return;
```

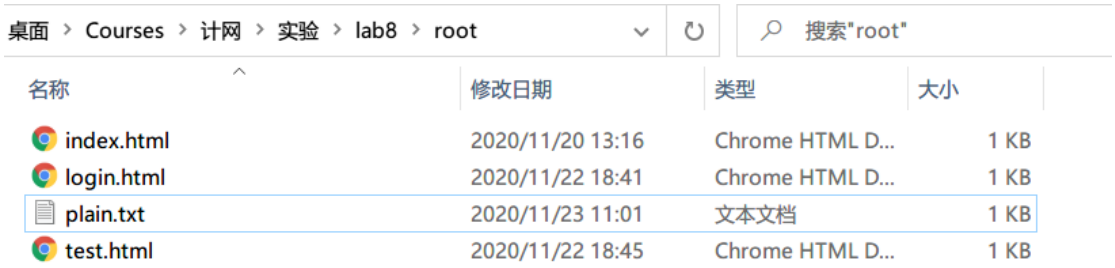
- 服务器运行后，用 netstat –an 显示服务器的监听端口

```
PS:Desktop> 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    0.0.0.0:443            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:808            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    0.0.0.0:3306           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5040           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5099           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5985           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:6646           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8082           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:9001           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:9415           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:10088          0.0.0.0:0              LISTENING
  TCP    0.0.0.0:27800          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。



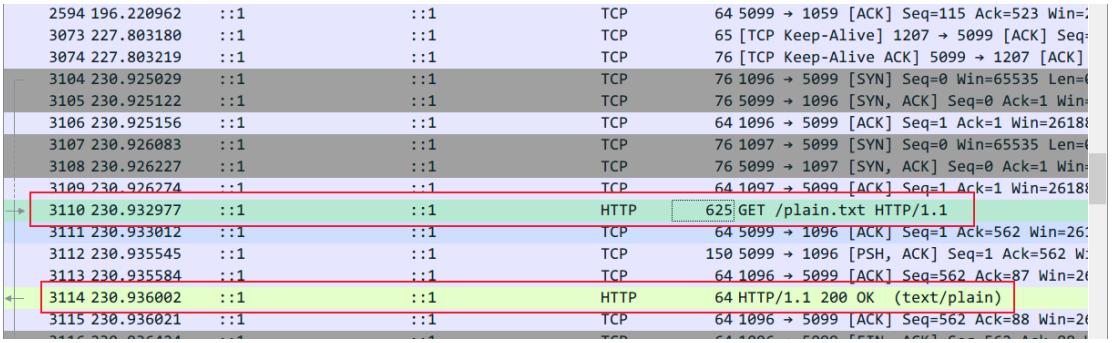
服务器上文件实际存放的路径：root/plain.txt



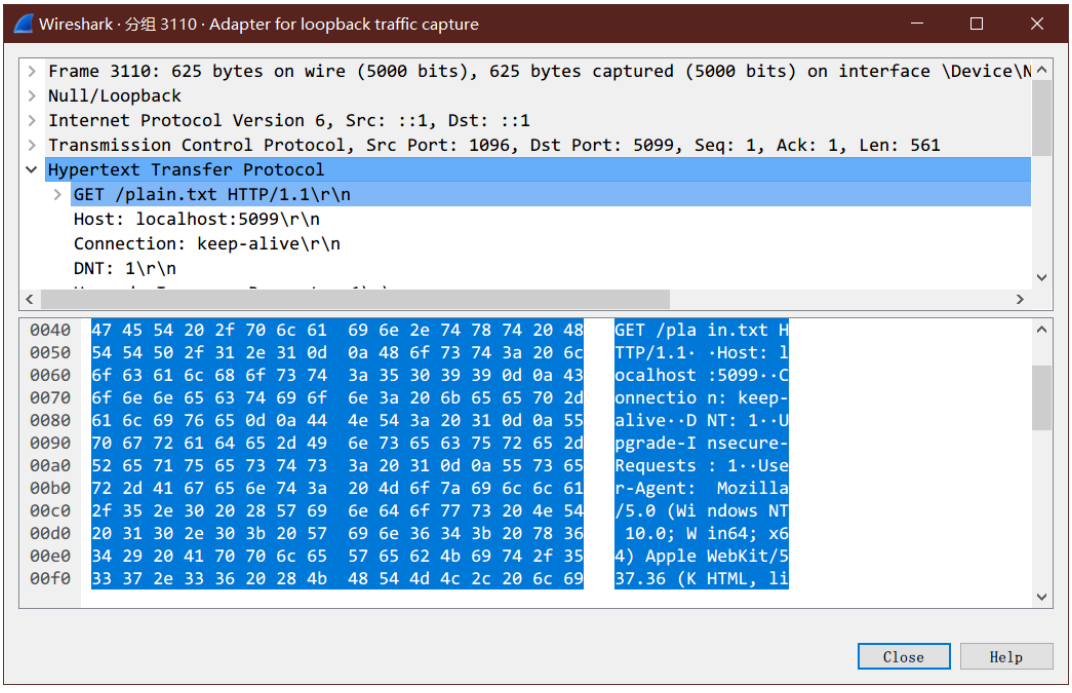
服务器的相关代码片段：

```
if(ext.equals("jpg")){
    res.headers.put("Content-Type", "image/jpeg");
}else if(ext.equals("html")){
    res.headers.put("Content-Type", "text/html; charset=UTF-8");
}else{
    res.headers.put("Content-Type", "text/plain; charset=UTF-8");
}
```

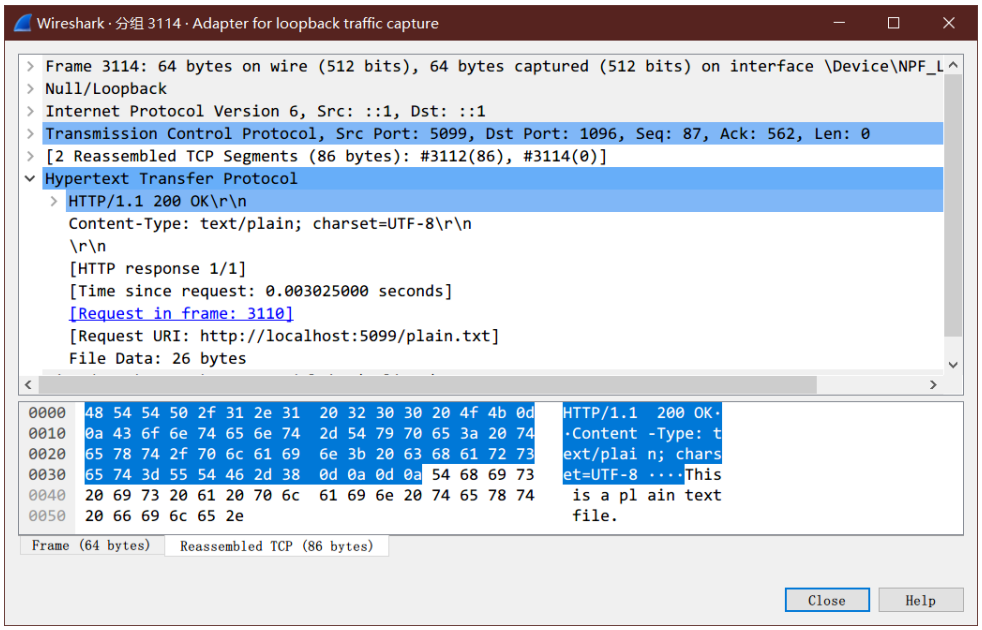
Wireshark 抓取的数据包截图（通过跟踪 TCP 流，只截取 HTTP 协议部分）：



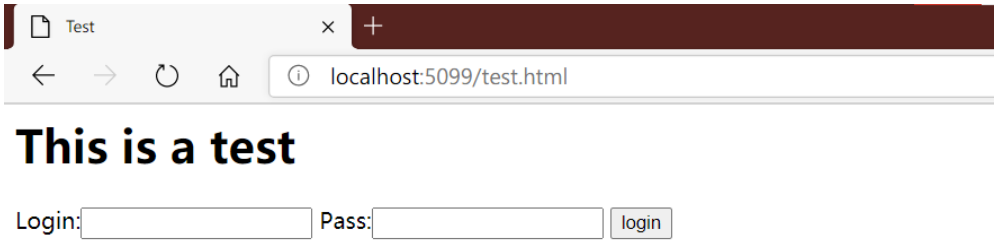
请求包：



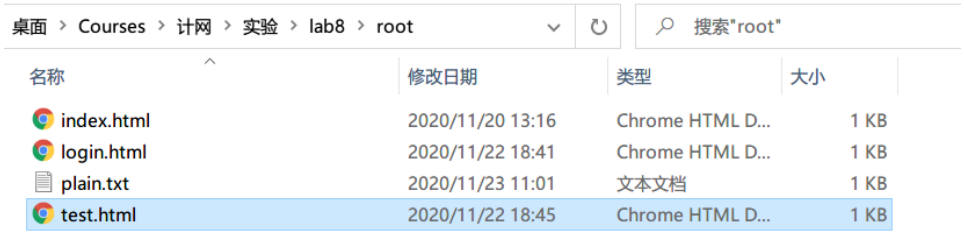
响应包：



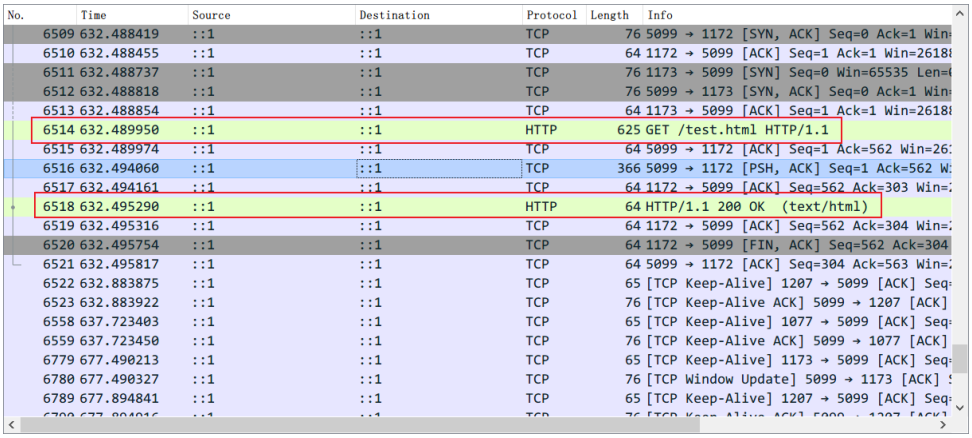
- 浏览器访问只包含文本的 HTML 文件时，浏览器的 URL 地址和显示内容截图。



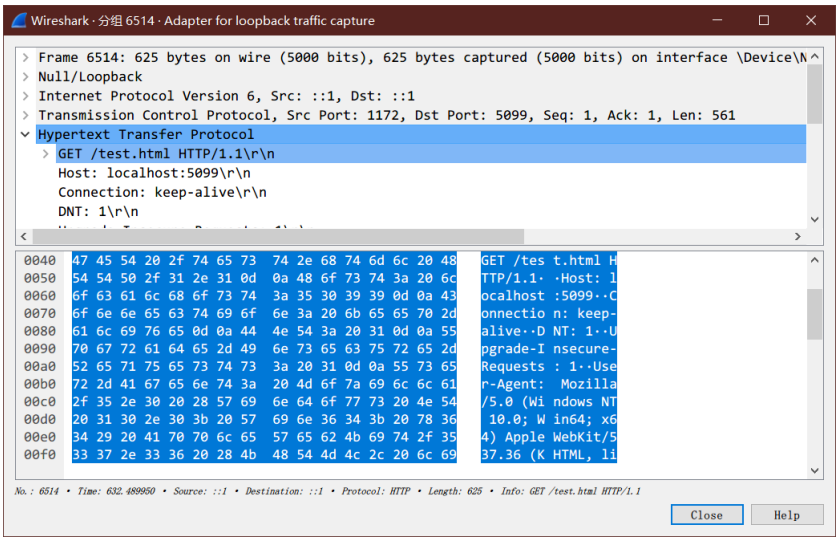
服务器文件实际存放的路径：root/test.html



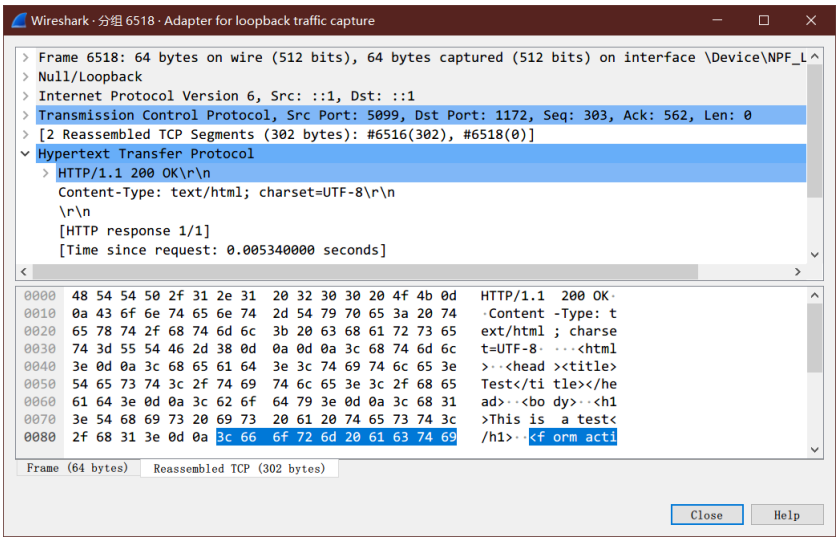
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML 内容）：



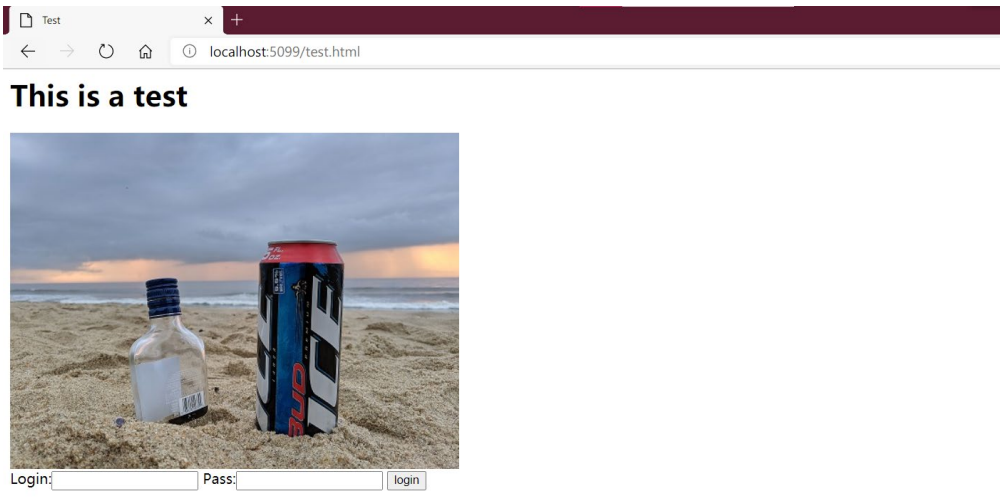
请求包：



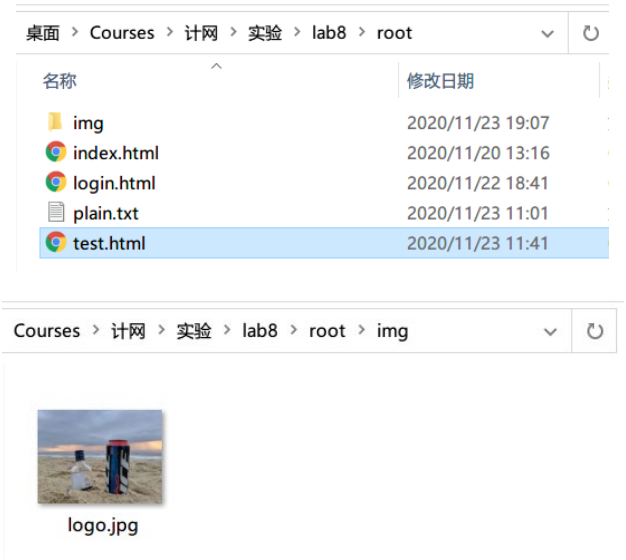
响应包：



- 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。



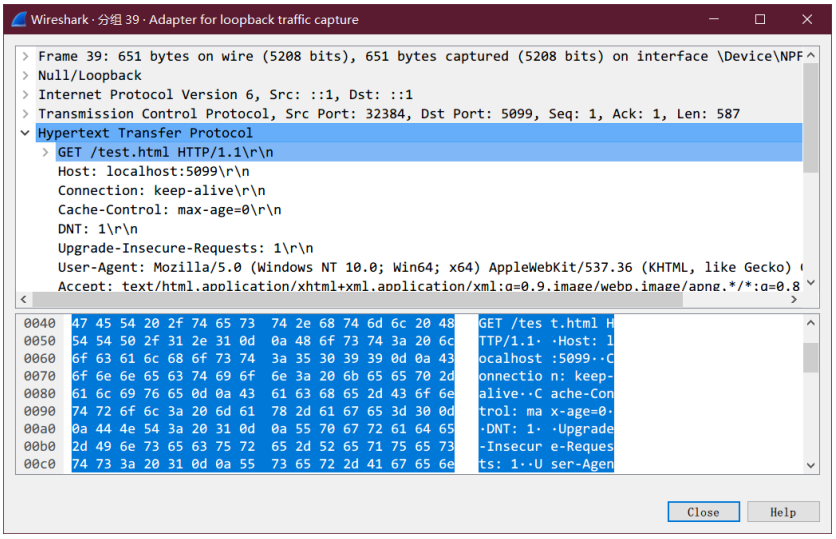
服务器上文件实际存放的路径：页面 root/test.html，图片 root/img/logo.jpg



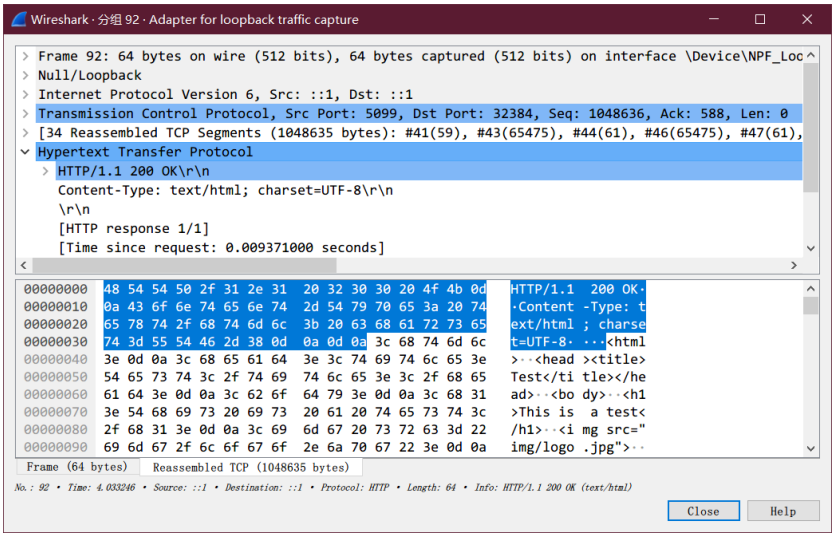
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：

No.	Time	Source	Destination	Protocol	Length	Info
33	4.020045	::1	::1	TCP	76	32384 → 5099 [SYN] Seq=0 Win=65535
34	4.020211	::1	::1	TCP	76	5099 → 32384 [SYN, ACK] Seq=0 Ack=
35	4.020328	::1	::1	TCP	64	32384 → 5099 [ACK] Seq=1 Ack=1 Win
36	4.020898	::1	::1	TCP	76	32385 → 5099 [SYN] Seq=0 Win=65535
37	4.021069	::1	::1	TCP	76	5099 → 32385 [SYN, ACK] Seq=0 Ack=
38	4.021149	::1	::1	TCP	64	32385 → 5099 [ACK] Seq=1 Ack=1 Win
39	4.023875	::1	::1	HTTP	651	GET /test.html HTTP/1.1
40	4.023932	::1	::1	TCP	64	5099 → 32384 [ACK] Seq=1 Ack=588 W
41	4.029525	::1	::1	TCP	123	5099 → 32384 [PSH, ACK] Seq=1 Ack=
42	4.029566	::1	::1	TCP	64	32384 → 5099 [ACK] Seq=588 Ack=60
43	4.029719	::1	::1	TCP	65539	5099 → 32384 [ACK] Seq=60 Ack=588
44	4.029791	::1	::1	TCP	125	5099 → 32384 [PSH, ACK] Seq=65535
45	4.029844	::1	::1	TCP	64	32384 → 5099 [ACK] Seq=588 Ack=655
46	4.029910	::1	::1	TCP	65539	5099 → 32384 [ACK] Seq=65596 Ack=5
89	4.032487	::1	::1	TCP	65539	5099 → 32384 [ACK] Seq=983100 Ack=
90	4.032551	::1	::1	TCP	125	5099 → 32384 [PSH, ACK] Seq=104857
91	4.032608	::1	::1	TCP	64	32384 → 5099 [ACK] Seq=588 Ack=104
92	4.033246	::1	::1	HTTP	64	HTTP/1.1 200 OK (text/html)
93	4.033270	::1	::1	TCP	64	32384 → 5099 [ACK] Seq=588 Ack=104
94	4.076566	::1	::1	HTTP	538	GET /img/logo.jpg HTTP/1.1
95	4.076592	::1	::1	TCP	64	5099 → 32385 [ACK] Seq=1 Ack=475 W
96	4.086473	::1	::1	TCP	64	[TCP Window Update] 32384 → 5099 [
97	4.087995	::1	::1	TCP	64	[TCP Window Update] 32384 → 5099 [
98	4.115285	::1	::1	TCP	64	32384 → 5099 [FIN, ACK] Seq=588 Ac
99	4.115334	::1	::1	TCP	64	5099 → 32384 [ACK] Seq=1048637 Ack
100	4.327860	::1	::1	TCP	109	5099 → 32385 [PSH, ACK] Seq=1 Ack=
144	4.329031	::1	::1	TCP	65539	5099 → 32385 [ACK] Seq=917550 Ack=4
145	4.329058	::1	::1	TCP	125	5099 → 32385 [PSH, ACK] Seq=983025
146	4.329080	::1	::1	TCP	64	32385 → 5099 [ACK] Seq=475 Ack=9830
147	4.329103	::1	::1	TCP	65539	5099 → 32385 [ACK] Seq=983086 Ack=4
148	4.329135	::1	::1	TCP	125	5099 → 32385 [PSH, ACK] Seq=1048561
149	4.329162	::1	::1	TCP	64	32385 → 5099 [ACK] Seq=475 Ack=1048
150	4.329449	::1	::1	HTTP	64	HTTP/1.1 200 OK (JPEG JFIF image)
151	4.329494	::1	::1	TCP	64	32385 → 5099 [ACK] Seq=475 Ack=1048
152	4.332216	::1	::1	TCP	64	[TCP Window Update] 32385 → 5099 [A
153	4.332666	::1	::1	TCP	64	32385 → 5099 [FIN, ACK] Seq=475 Ack
154	4.332707	::1	::1	TCP	64	5099 → 32385 [ACK] Seq=1048623 Ack=

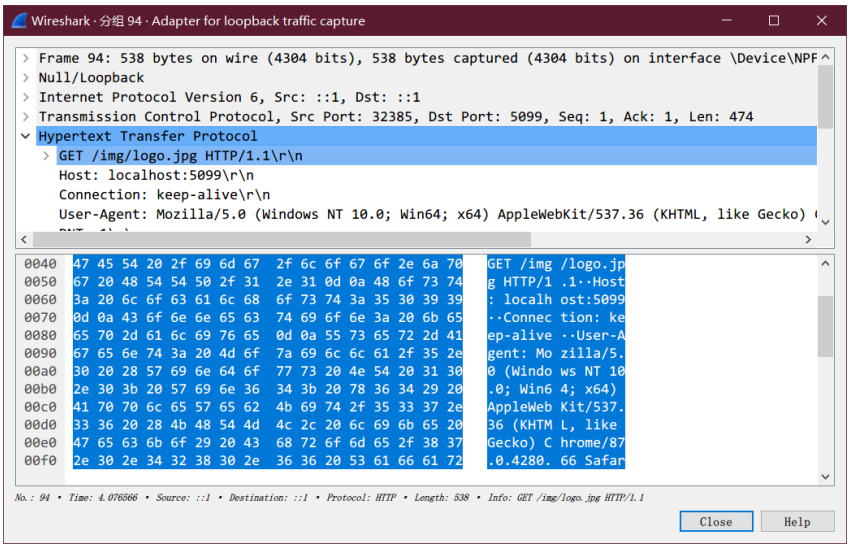
请求包（html）：



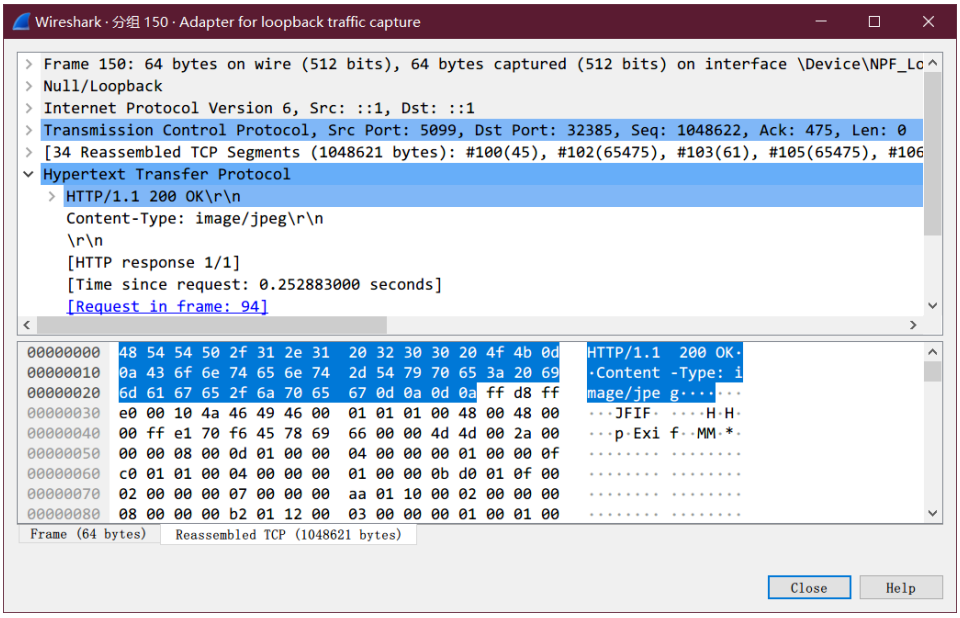
响应包（html）：



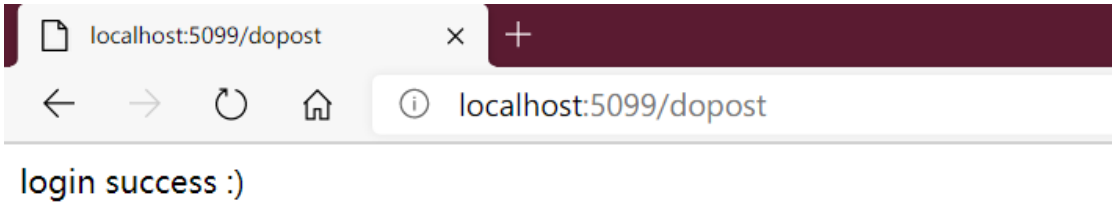
请求包（图片）：



响应包（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。



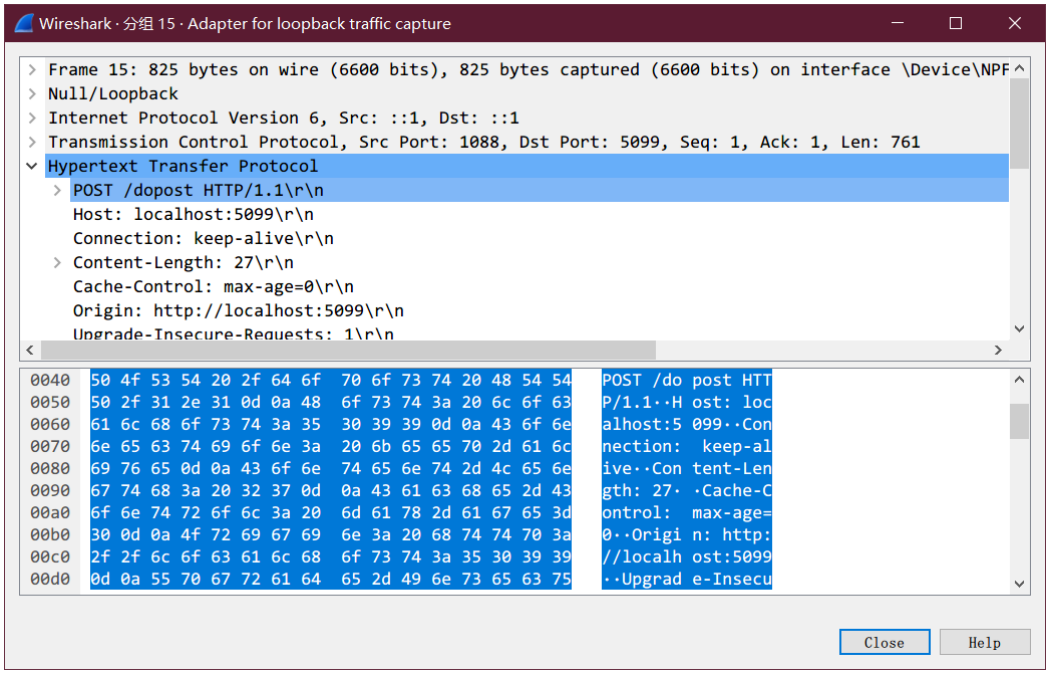
服务器相关处理代码片段：

```
String[] post = req.data.split("&");
Map<String, String> paras = new HashMap<>();
for(int i=0; i<post.length; i++){
    String para = post[i].substring(0, post[i].indexOf('='));
    String val = post[i].substring(post[i].indexOf('=') + 1);
    paras.put(para, val);
}
res.version = "HTTP/1.1";
res.code = 200;
res.status = "OK";
res.headers.put("Content-Type", "text/html; charset=UTF-8");
String msg = "<html>" +
        "<head></head>" +
        "<body>login success :)<body>" +
        "</html>";
if(!paras.containsKey("login") || !paras.containsKey("pass")
|| !paras.get("login").equals(login) || !paras.get("pass").equals(pass)){
    msg = "<html>" +
            "<head></head>" +
            "<body>login failed :(<body>" +
            "</html>";

}
res.headers.put("Content-Length", ""+msg.length());
res.data.put(msg.getBytes());
```

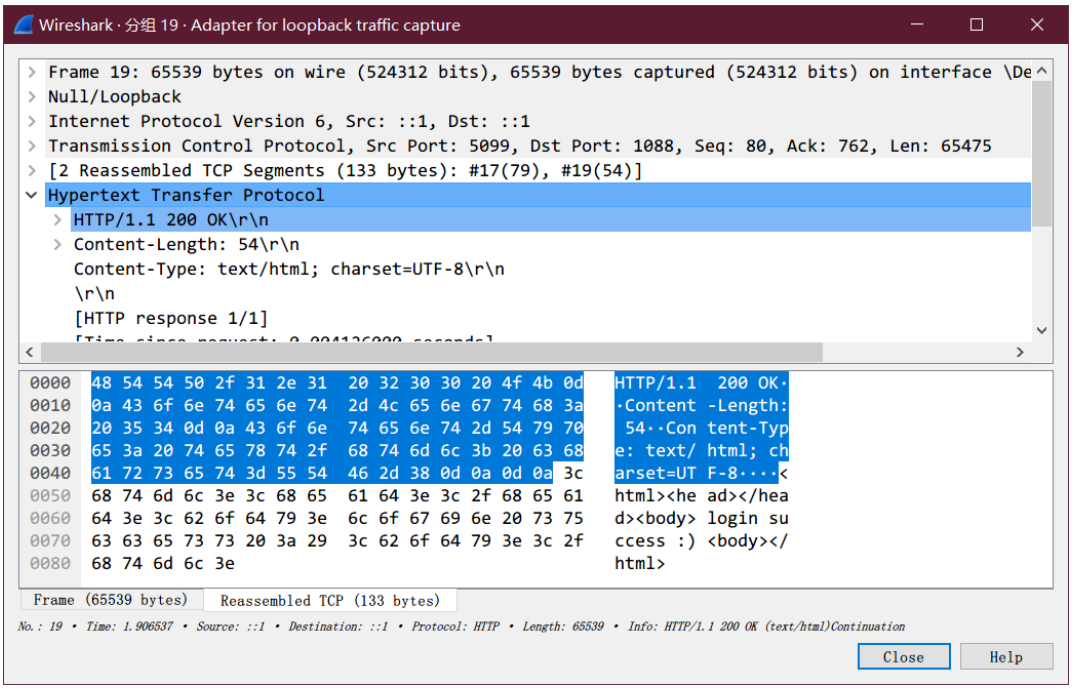
Wireshark 抓取的数据包截图（HTTP 协议部分）

No.	Time	Source	Destination	Protocol	Length	Info
15	1.902411	::1	::1	HTTP	825	POST /dopost HTTP/1.1 (application
16	1.902521	::1	::1	TCP	64	5099 → 1088 [ACK] Seq=1 Ack=762 Win
17	1.906266	::1	::1	TCP	143	5099 → 1088 [PSH, ACK] Seq=1 Ack=76
18	1.906346	::1	::1	TCP	64	1088 → 5099 [ACK] Seq=762 Ack=80 Wi
19	1.906537	::1	::1	HTTP	65539	HTTP/1.1 200 OK (text/html)Continu
20	1.906644	::1	::1	HTTP	125	Continuation
21	1.906721	::1	::1	TCP	64	1088 → 5099 [ACK] Seq=762 Ack=65616
22	1.906801	::1	::1	HTTP	65539	Continuation
23	1.906897	::1	::1	HTTP	125	Continuation
24	1.906969	::1	::1	TCP	64	1088 → 5099 [ACK] Seq=762 Ack=13115
25	1.907048	::1	::1	HTTP	65539	Continuation
26	1.907140	::1	::1	HTTP	125	Continuation

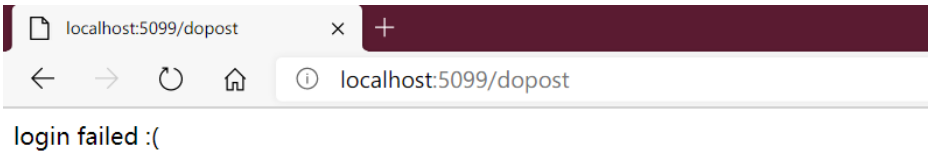
请求包：



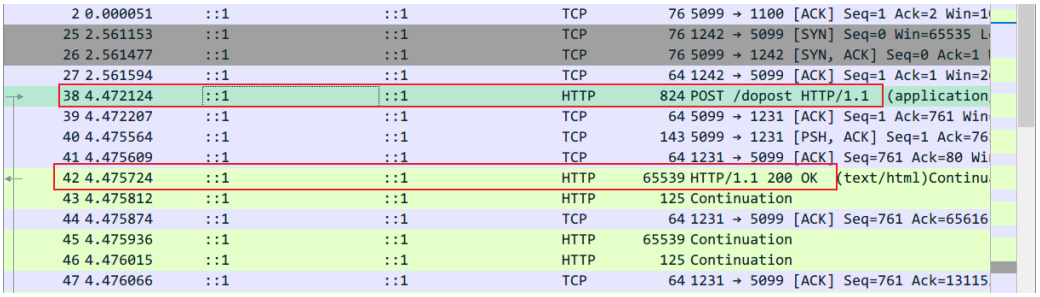
响应包：



- 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。

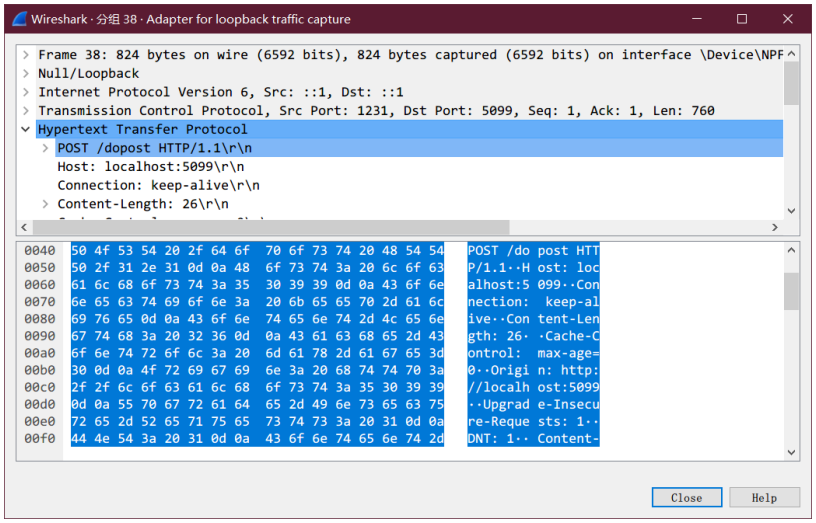


Wireshark 抓取的数据包截图（HTTP 协议部分）

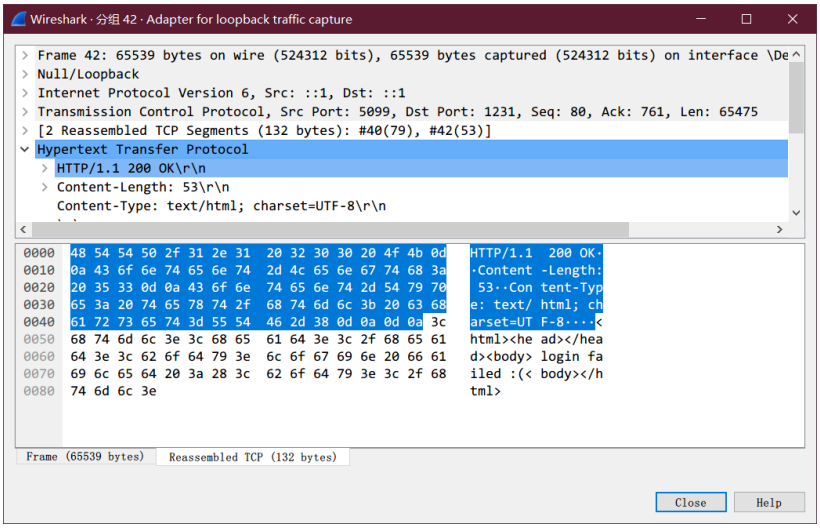


2	0.000051	::1	::1	TCP	76 5099 → 1100 [ACK] Seq=1 Ack=2 Win=1
25	2.561153	::1	::1	TCP	76 1242 → 5099 [SYN] Seq=0 Win=65535 L
26	2.561477	::1	::1	TCP	76 5099 → 1242 [SYN, ACK] Seq=0 Ack=1
27	2.561594	::1	::1	TCP	64 1242 → 5099 [ACK] Seq=1 Ack=1 Win=2
38	4.472124	::1	::1	HTTP	824 POST /dopost HTTP/1.1 (application
39	4.472207	::1	::1	TCP	64 5099 → 1231 [ACK] Seq=1 Ack=761 Win
40	4.475564	::1	::1	TCP	143 5099 → 1231 [PSH, ACK] Seq=1 Ack=76
41	4.475609	::1	::1	TCP	64 1231 → 5099 [ACK] Seq=761 Ack=80 Wi
42	4.475724	::1	::1	HTTP	65539 HTTP/1.1 200 OK (text/html)Continu
43	4.475812	::1	::1	HTTP	125 Continuation
44	4.475874	::1	::1	TCP	64 1231 → 5099 [ACK] Seq=761 Ack=65616
45	4.475936	::1	::1	HTTP	65539 Continuation
46	4.476015	::1	::1	HTTP	125 Continuation
47	4.476066	::1	::1	TCP	64 1231 → 5099 [ACK] Seq=761 Ack=13115

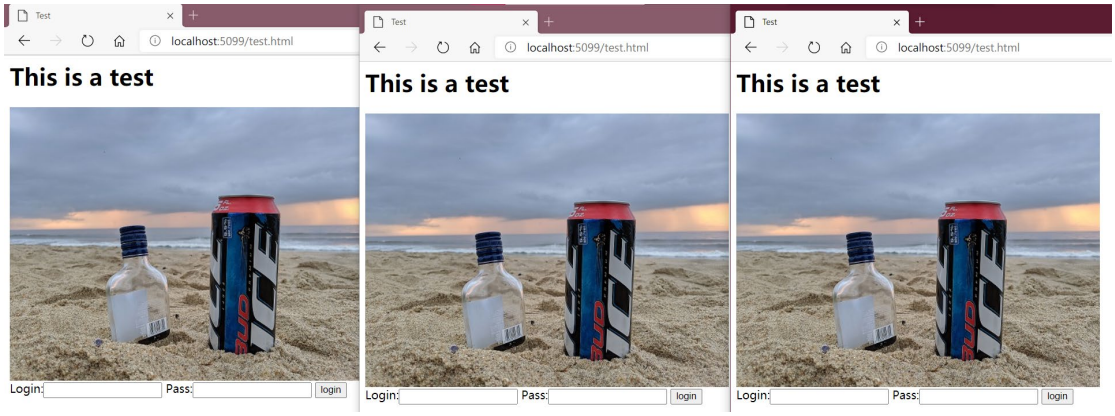
请求包：



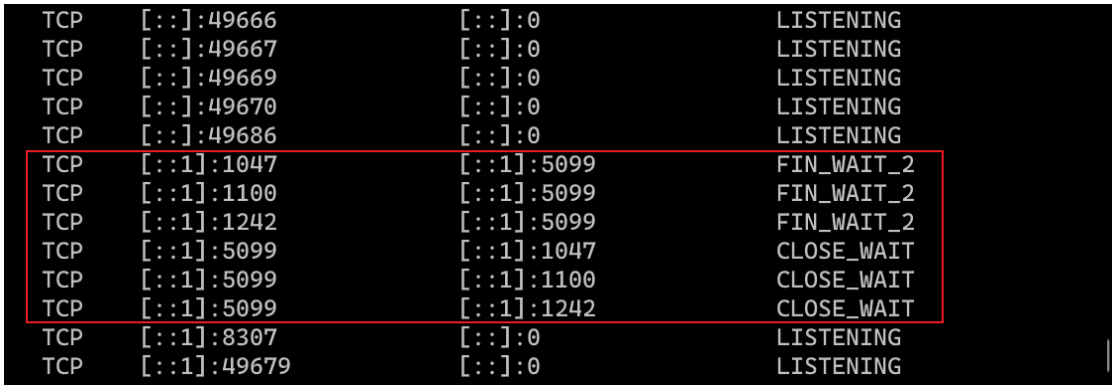
响应包：



- 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）



- 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）



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？

  HTTP 协议通过一个空行分隔头部和体部。

- 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？

  浏览器是根据 headers 中的 Content-Type 字段来判断文件类型的，比如 text/plain 表示纯文本，text/html 表示 html 页面，image/jpeg 表示 jpg 图片，等等。

- HTTP 协议的头部是不是一定是文本格式？体部呢？

  HTTP 协议的头部一定是文本格式，但是体部还可以是字节流的格式。

- 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？

  POST 方法传递的数据放在体部，两个字段用“&”符号连接，比如：

  login=admin&pass=123456

七、讨论、心得

实验中遇到的困难：

1. Wireshark 抓不到本地包

   解决方案：卸载 WinCap，安装 Ncap

2. 图片无法正常显示

   先是发现图片的字节流在传输后发生了变化，原因是 Java 的字符是 UTF-16 编码的，于是改用 ByteBuffer 进行字节流的存储；

   然后发现图片还是不能正常显示，查看控制台发现有缓冲区移除，原因是给 ByteBuffer 分配的缓冲区不够大，于是加大了给 ByteBuffer 分陪的缓冲区空间。

经验教训：

本次实验主要考察 http 会话建立的过程，包括 http 服务器的基本逻辑；编程的时候用的 socket 库等内容之前接触的比较少，但是查阅资料后可以比较熟练的应用；经过本次实验，不仅对 http 协议的理解更深了，对理论课的内容也有了进一步的认识，自己的动手能力也得到了提升，希望可以在接下来的课程学习中做的更好。