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：	应承峻
学　　院：	计算机学院
系：	计算机系
专　　业：	软件工程
学　　号：	3170103456
指导教师：	高艺

2019 年 12 月 15 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：　夏林轩　　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**
- 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 死循环不断接收请求，当接收到请求时建立连接，然后创建子线程进行处理。

```
public class Main {

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket( port: 3456);
            while(true){
                Socket socket = serverSocket.accept();
                new HttpServer(socket,  rootPath: "e:/txt/").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）

  子线程处理的 HttpServer 类继承了 Thread 类，构造函数的作用是通过 Socket 对象打开输入流 req 和输出流 res，然后通过 root 指定存储的根目录。

```
public class HttpServer extends Thread {

    private InputStream req;
    private OutputStream res;
    private String root;
    private static final int BUFFER_SIZE = 4096;  //4MB
    private static final String MYLOGIN = "3170103456";
    private static final String MYPASS = "3456";

    public HttpServer(Socket socket, String rootPath) {
        try {
            req = socket.getInputStream();
            res = socket.getOutputStream();
            root = rootPath;
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

然后子进程将会调用 run 方法，先通过 getReqHeader 来解析请求头（如果时 POST 请求还有请求体），得到一个自定义的 Request 对象（包含 URL、Method 和 Params 三个属性），然后根据具体的请求类型来执行相应的处理函数，执行完毕后退出线程。

```
@Override
public void run() {
    try {
        System.out.println(Thread.currentThread().getName() + ": 开始处理请求...");
        Request request = getReqHeader();
        if (request.getMethod() == Request.Method.GET) {
            responseGET(request.getUrl());
        } else if (request.getMethod() == Request.Method.POST) {
            responsePOST(request);
        } else throw new ParseException("Method " + request.getMethod() + " not supported!", 0);
        System.out.println(Thread.currentThread().getName() + ": 执行结束...");
    } catch (IOException | ParseException e) {
        e.printStackTrace();
    } finally {
        //noinspection deprecation
        stop();
    }
}
```

处理 GET 请求的代码如下，需要注意判断请求的文件是否存在以及请求文件的类型。

```
private void responseGET(String filepath) throws IOException {
    File file = new File(root, filepath);
    System.out.println("An user request for file: " + file.getName());
    if (!filepath.equals("/") && file.exists()) {  //文件存在
        String extension;
        InputStream in = new FileInputStream(file);
        byte[] bytes = new byte[BUFFER_SIZE];  //缓冲区
        int len;  //每次读入缓冲区的长度
        res.write("HTTP/1.1 200 OK\r\n".getBytes());
        extension = getPostFixName(filepath);  //获取文件扩展名
        if (extension.equals(".jpg")) {  //设置MIME消息类型
            res.write("content-type:image/jpg\r\n".getBytes());
        } else if (extension.equals(".html")) {
            res.write("content-type:text/html;charset=UTF-8\r\n".getBytes());
        } else {
            res.write("content-type:text/plain;charset=UTF-8\r\n".getBytes());
        }
        res.write("\r\n".getBytes());
        while ((len = in.read(bytes)) != -1) {  //输出资源
            res.write(bytes, off: 0, len);
        }
        res.flush();
        res.close();
    } else {  //文件不存在
        String err = "HTTP/1.1 404 file not found \r\n" + "Content-Type:text/html;charset=UTF-8\r\n" +
                "Content-Length:22 \r\n" + "\r\n" + "<h1>404 Not Found</h1>";
        res.write(err.getBytes()); //输出字节流
        res.flush();
        res.close();
    }
}
```

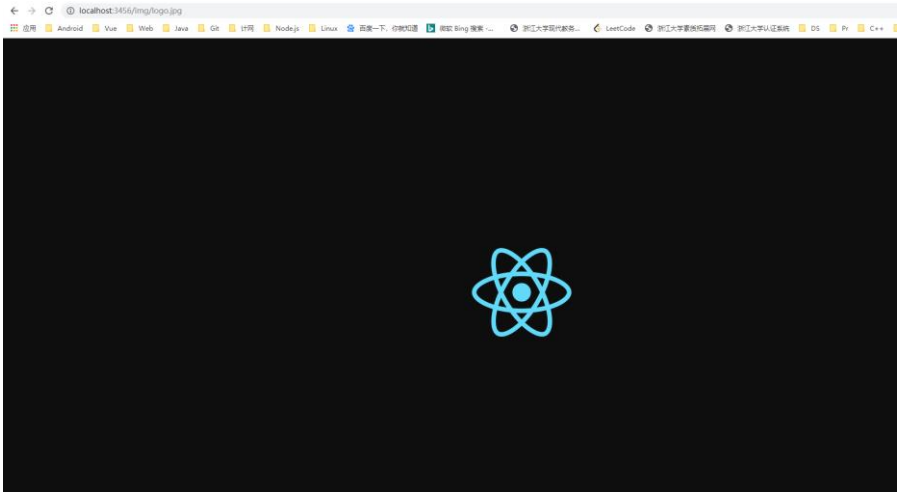
处理 POST 请求的代码：

```
private void responsePOST(Request request) throws IOException {
    if (request.getUrl().equals("/dopost") && request.params.containsKey("login") && request.params.containsKey("pass"
        String msg;
        String login = (String) request.params.get("login");
        String pass = (String) request.params.get("pass");
        if (login.equals(MYLOGIN) && pass.equals(MYPASS)) msg = "<html><head><meta charset=\"UTF-8\"></head><body>登录
        else msg = "<html><head><meta charset=\"UTF-8\"></head><body>登录失败!</body></html>";
        msg = "HTTP/1.1 200 OK\r\n" + "Content-Type:text/html \r\n" +
                "Content-Length:" + msg.length() + "\r\n" + "\r\n" + msg;
        res.write(msg.getBytes()); //输出字节流
        res.flush();
        res.close();
    } else {
        String err = "HTTP/1.1 404 file not found \r\n" + "Content-Type:text/html \r\n" +
                "Content-Length:22 \r\n" + "\r\n" + "<h1>404 Not Found</h1>";
        res.write(err.getBytes()); //输出字节流
        res.flush();
        res.close();
    }
}
```

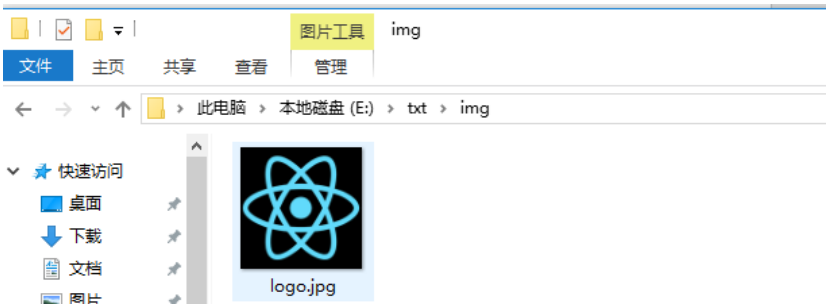
- 服务器运行后，用 netstat –an 显示服务器的监听端口

```
C:\Users\Nonehyo>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    0.0.0.0:902            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:912            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1080           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1536           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1537           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1538           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1539           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1541           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1544           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1549           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:3000           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:3306           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:3456           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:5357           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:7250           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:19531          0.0.0.0:0              LISTENING
```

- 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。



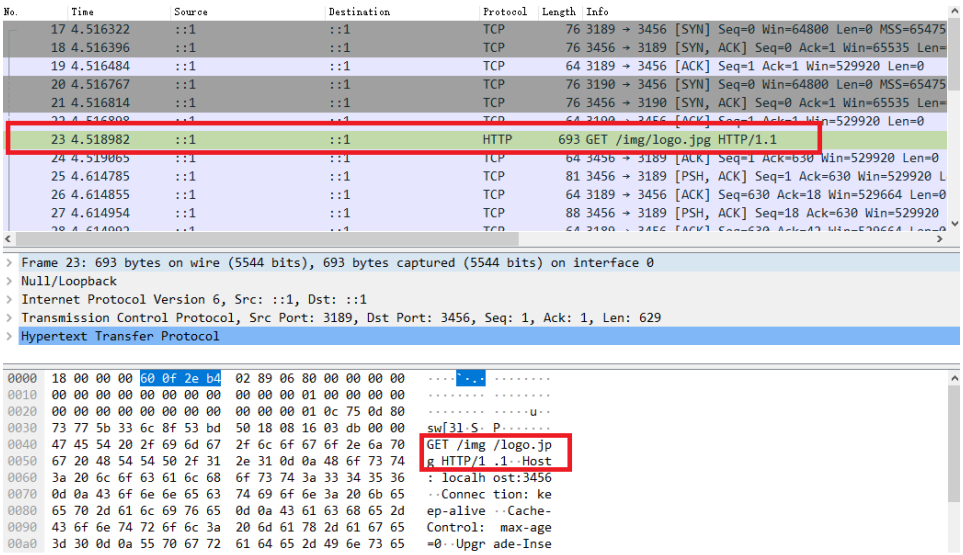
服务器上文件实际存放的路径：e:/txt/img/logo.jpg



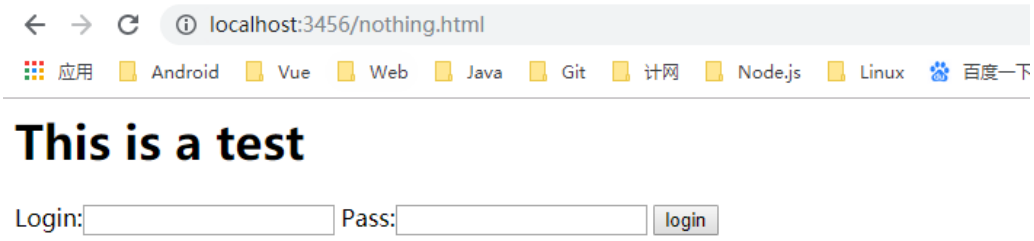
服务器的相关代码片段：

```
private void responseGET(String filepath) throws IOException {
    File file = new File(root, filepath);
    System.out.println("An user request for file: " + file.getName());
    if (!filepath.equals("/") && file.exists()) {  //文件存在
        String extension;
        InputStream in = new FileInputStream(file);
        byte[] bytes = new byte[BUFFER_SIZE];  //缓冲区
        int len;  //每次读入缓冲区的长度
        res.write("HTTP/1.1 200 OK\r\n".getBytes());
        extension = getPostFixName(filepath);  //获取文件扩展名
        if (extension.equals(".jpg")) {  //设置MIME消息类型
            res.write("content-type:image/jpg\r\n".getBytes());
        } else if (extension.equals(".html")) {
            res.write("content-type:text/html;charset=UTF-8\r\n".getBytes());
        } else {
            res.write("content-type:text/plain;charset=UTF-8\r\n".getBytes());
        }
        res.write("\r\n".getBytes());
        while ((len = in.read(bytes)) != -1) {  //输出资源
            res.write(bytes, off: 0, len);
        }
        res.flush();
        res.close();
```

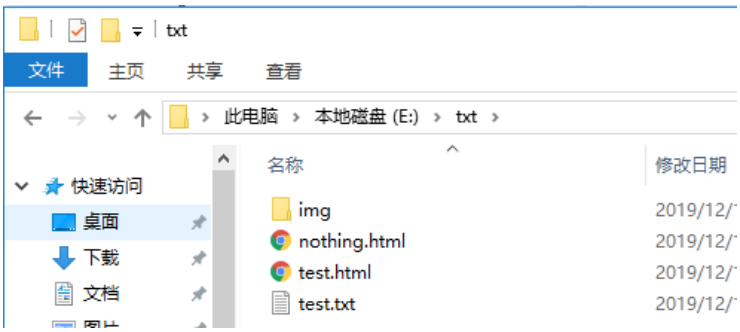
Wireshark 抓取的数据包截图（通过跟踪 TCP 流，只截取 HTTP 协议部分）：



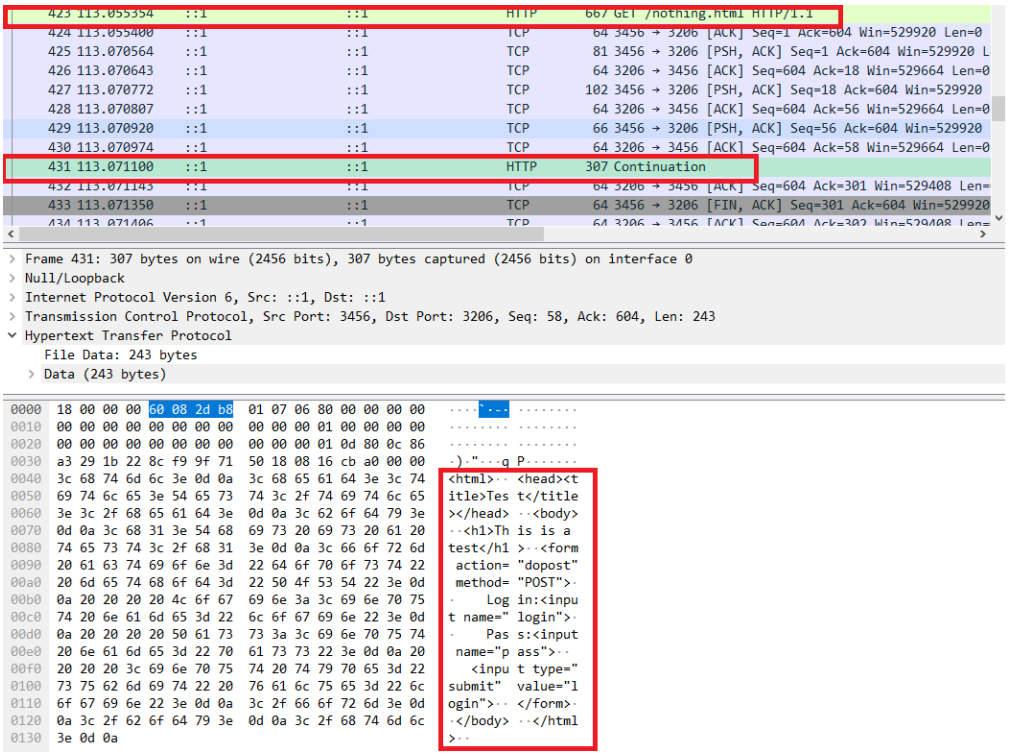
- 浏览器访问只包含文本的 HTML 文件时，浏览器的 URL 地址和显示内容截图。



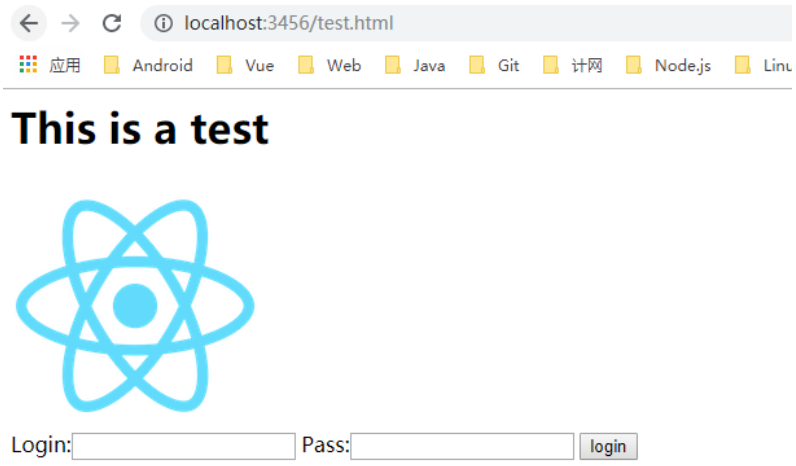
服务器文件实际存放的路径：e:/txt/nothing.html



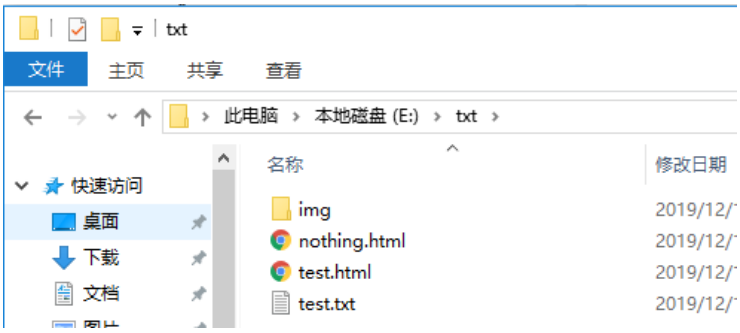
Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML 内容）：



- 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。

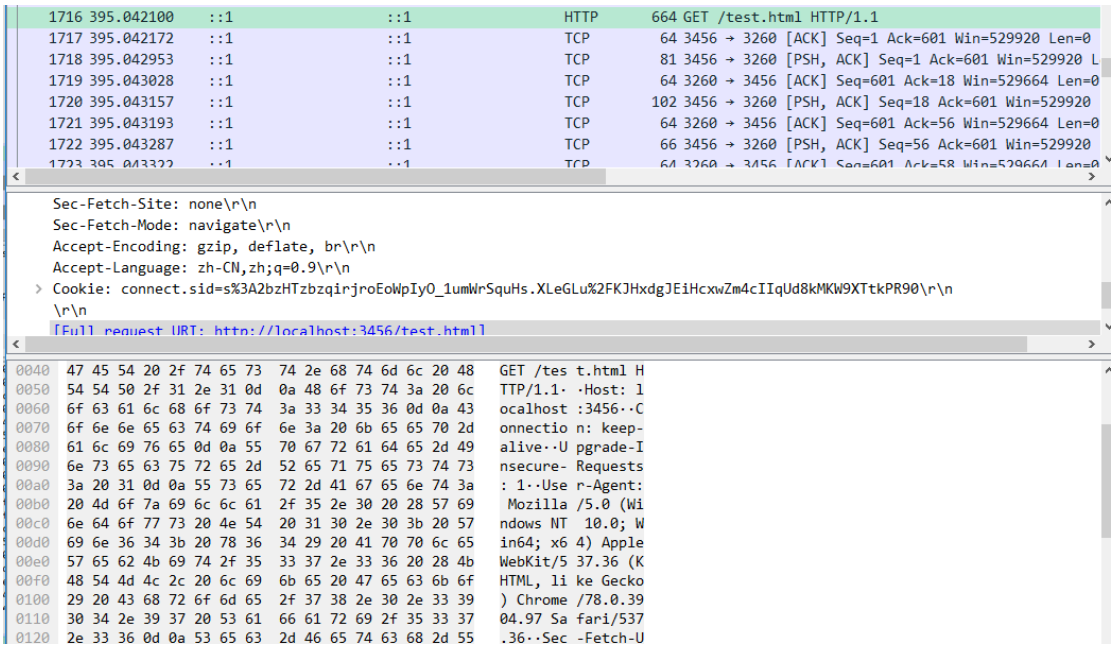


服务器上文件实际存放的路径：e:/txt/test.html

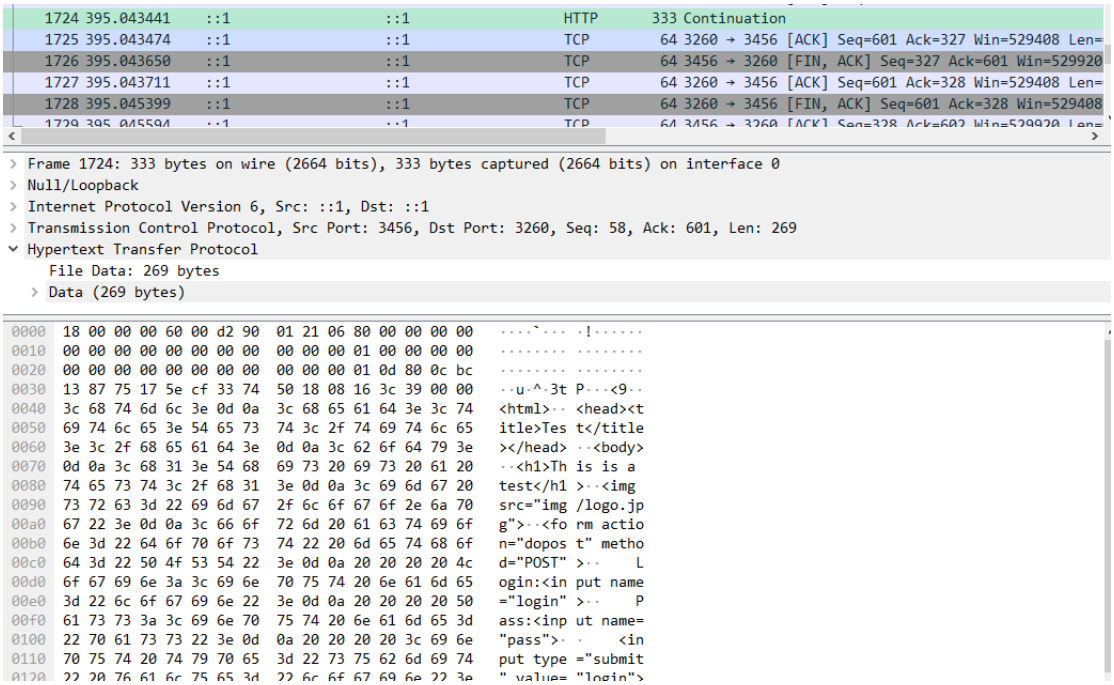


Wireshark 抓取的数据包截图（只截取 HTTP 协议部分，包括 HTML、图片文件的部分内容）：

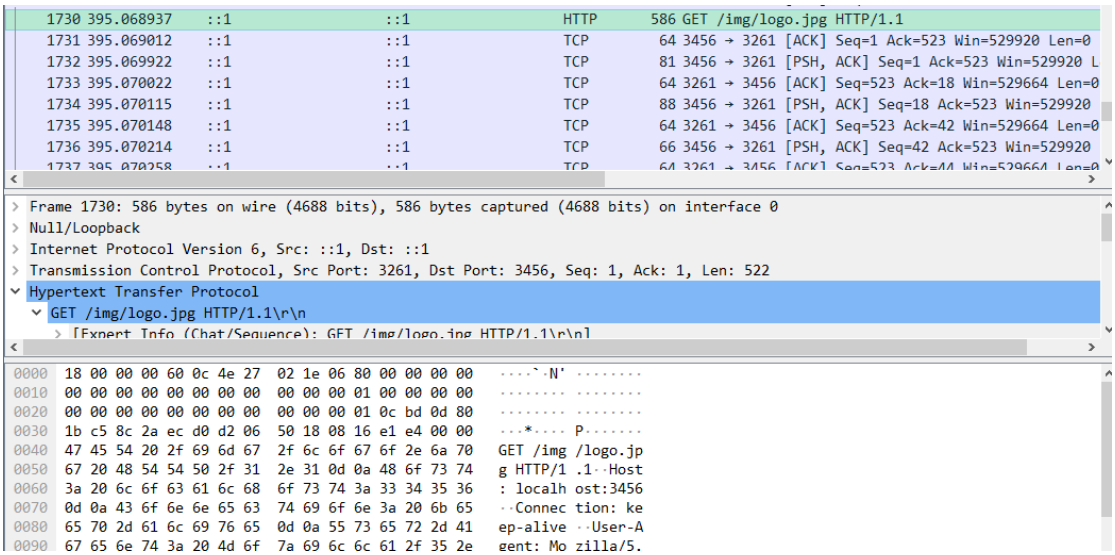
HTTP 请求



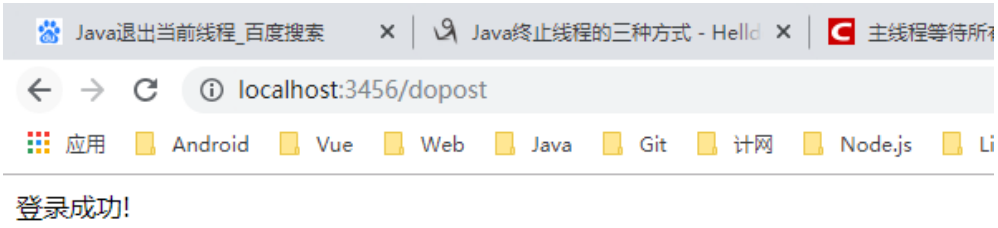
网页：



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：

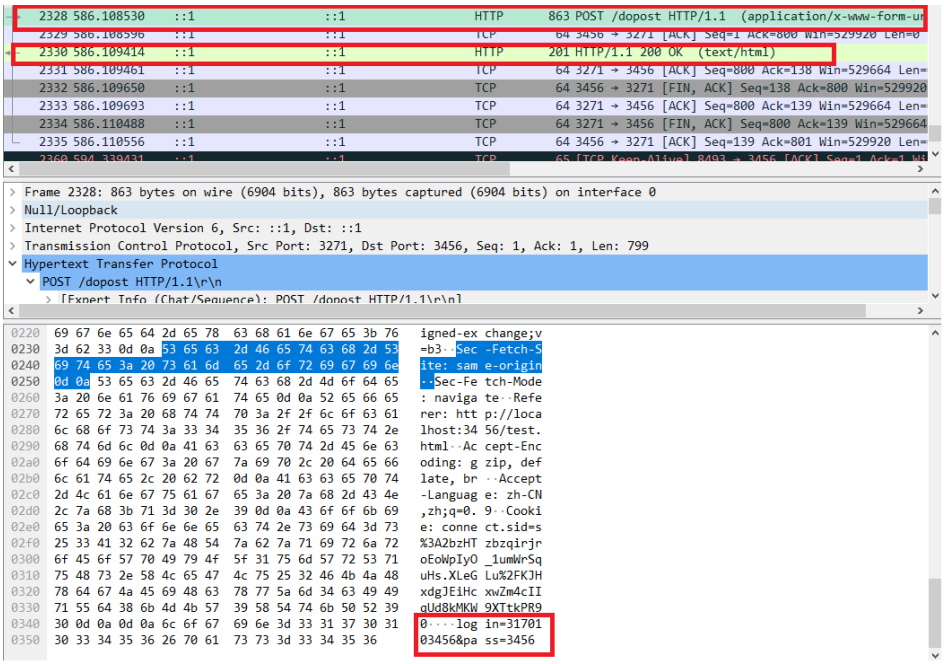
getReqHeader 方法用于解析出 POST 请求中的键值对

```
private Request getReqHeader() throws IOException, ParseException, NumberFormatException {
    BufferedReader reader = new BufferedReader(new InputStreamReader(req)); //通过字符流和缓冲区读入请求
    String line = reader.readLine();
    String[] list = line.split( regex: " "); //分割参数
    if (list.length != 3) throw new ParseException("HTTP request header parse Error!", 0);
    Request request = new Request(list[1], Request.Method.valueOf(list[0].trim()));
    if (request.getMethod() == Request.Method.POST) {  //POST请求还需要抓取请求体
        int contentLength = 0;
        while ((line = reader.readLine()) != null) {
            if ("".equals(line)) {   //判断是否读到请求头末尾
                break;
            } else if (line.contains("Content-Length")) {  //计算长度
                contentLength = Integer.parseInt(line.substring(line.indexOf("Content-Length") + 16));
            }
        }
        char[] buf;
        if (contentLength != 0) {
            buf = new char[contentLength];
            reader.read(buf,  off: 0, contentLength);
            String[] params = new String(buf).split( regex: "&");  //分割键值对
            for (String item : params) {
                String key = item.substring(0, item.indexOf("="));  //获取键值对
                String val = item.substring(item.indexOf("=") + 1);
                request.params.put(key, val);
            }
        }
    }
    //System.out.println("OK");
    return request;
}
```

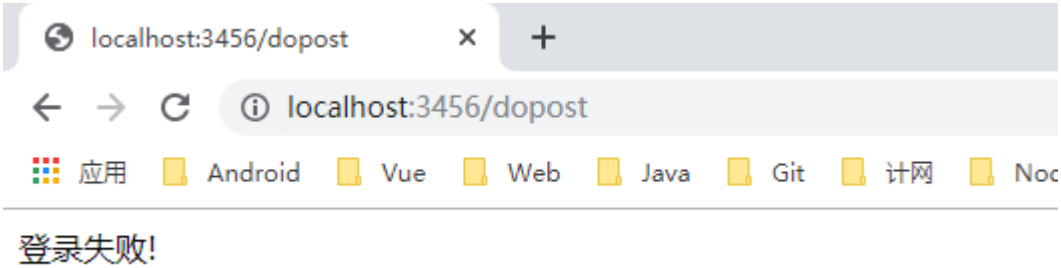
验证并响应给客户端：

```
private void responsePOST(Request request) throws IOException {
    if (request.getUrl().equals("/dopost") && request.params.containsKey("login") && request.params.containsKey("pass")) {
        String msg;
        String login = (String) request.params.get("login");
        String pass = (String) request.params.get("pass");
        if (login.equals(MYLOGIN) && pass.equals(MYPASS)) msg = "<html><head><meta charset=\"UTF-8\"></head><body>登录成功!</body></html>";
        else msg = "<html><head><meta charset=\"UTF-8\"></head><body>登录失败!</body></html>";
        msg = "HTTP/1.1 200 OK\r\n" + "Content-Type:text/html \r\n" +
                "Content-Length:" + msg.length() + "\r\n" + "\r\n" + msg;
        res.write(msg.getBytes()); //输出字节流
        res.flush();
        res.close();
    } else {
        String err = "HTTP/1.1 404 file not found \r\n" + "Content-Type:text/html \r\n" +
                "Content-Length:22 \r\n" + "\r\n" + "<h1>404 Not Found</h1>";
        res.write(err.getBytes()); //输出字节流
        res.flush();
        res.close();
    }
}
```

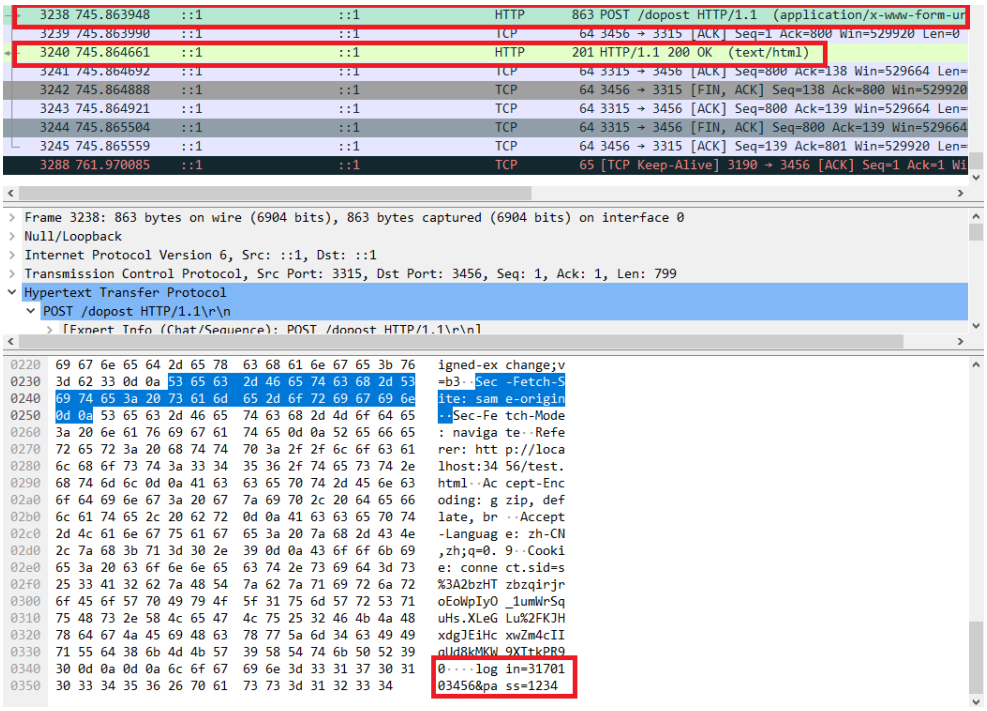
Wireshark 抓取的数据包截图（HTTP 协议部分）



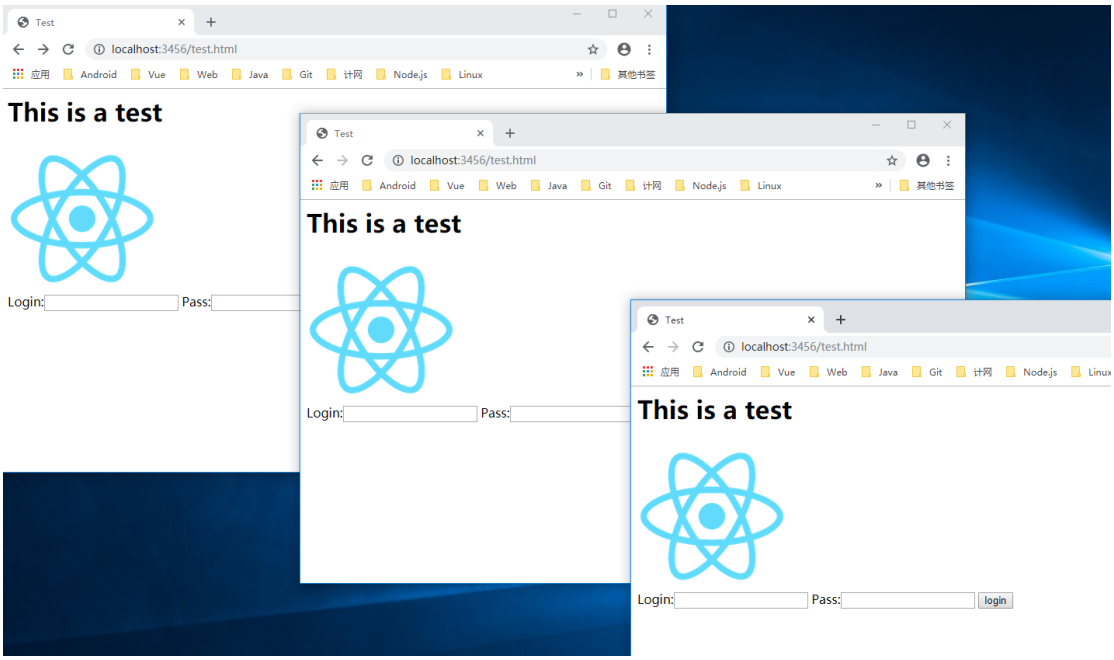
- 浏览器输入错误的登录名或密码，点击登录按钮（login）后的显示截图。



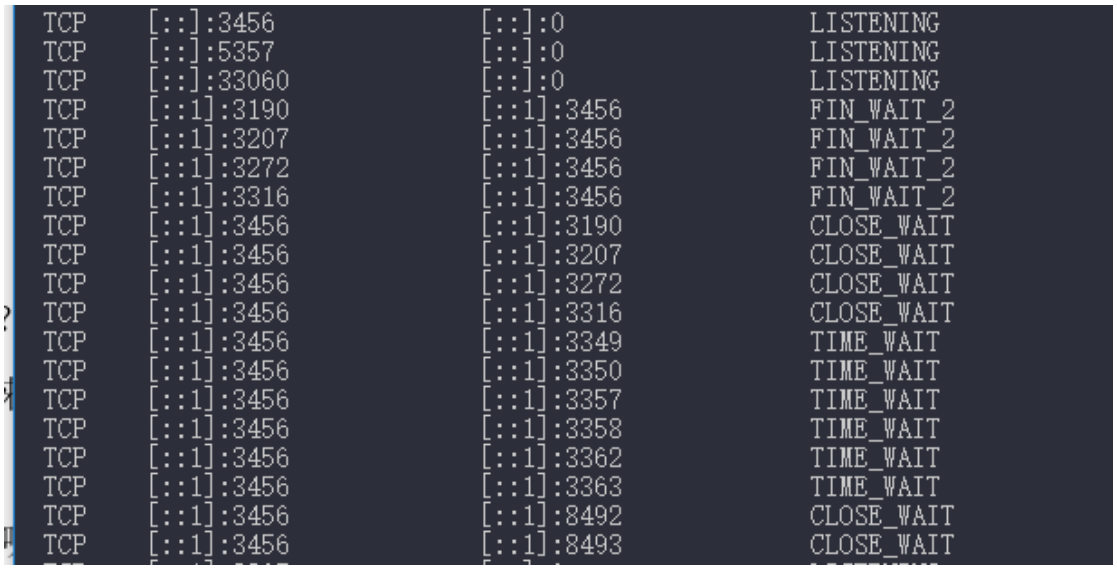
Wireshark 抓取的数据包截图（HTTP 协议部分）



- 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    [::]:3456              [::]:0                 LISTENING
TCP    [::]:5357              [::]:0                 LISTENING
TCP    [::]:33060             [::]:0                 LISTENING
TCP    [::1]:3190             [::1]:3456             FIN_WAIT_2
TCP    [::1]:3207             [::1]:3456             FIN_WAIT_2
TCP    [::1]:3272             [::1]:3456             FIN_WAIT_2
TCP    [::1]:3316             [::1]:3456             FIN_WAIT_2
TCP    [::1]:3456             [::1]:3190             CLOSE_WAIT
TCP    [::1]:3456             [::1]:3207             CLOSE_WAIT
TCP    [::1]:3456             [::1]:3272             CLOSE_WAIT
TCP    [::1]:3456             [::1]:3316             CLOSE_WAIT
TCP    [::1]:3456             [::1]:3349             TIME_WAIT
TCP    [::1]:3456             [::1]:3350             TIME_WAIT
TCP    [::1]:3456             [::1]:3357             TIME_WAIT
TCP    [::1]:3456             [::1]:3358             TIME_WAIT
TCP    [::1]:3456             [::1]:3362             TIME_WAIT
TCP    [::1]:3456             [::1]:3363             TIME_WAIT
TCP    [::1]:3456             [::1]:8492             CLOSE_WAIT
TCP    [::1]:3456             [::1]:8493             CLOSE_WAIT
```

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？

  答：HTTP 协议的请求头和请求体通过空行来分隔。

- 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？

  答：浏览器是根据头部的 Content-Type 字段来判断文件类型的，如常见的 Content-Type 值有

  - text/html ：HTML 格式
  - text/plain ：纯文本格式
  - text/xml ： XML 格式
  - image/jpeg ：jpg 图片格式
  - image/png：png 图片格式

- HTTP 协议的头部是不是一定是文本格式？体部呢？

  答：协议的头部是文本格式，体部除了文本格式外，还可以是字节流的方式，比如音频、图片等数据的传输。

- 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

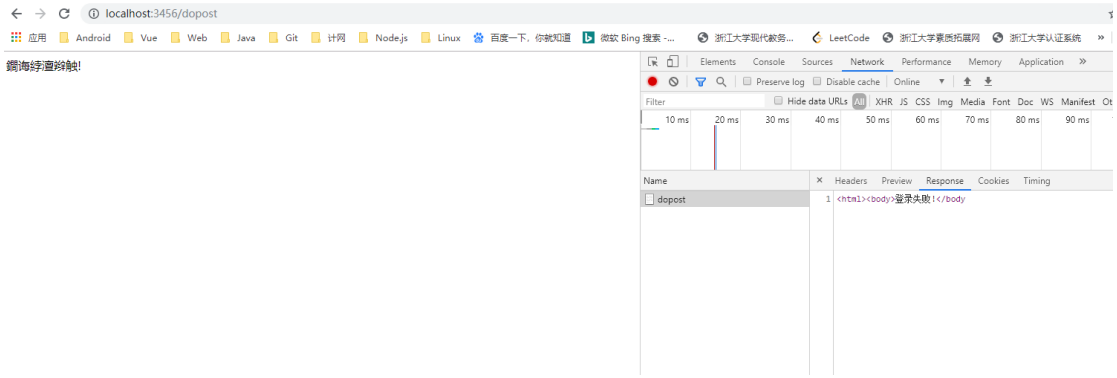
接起来的？

答：放在体部，如“login=123&pass=12345”，两个字段通过&符号连接。

七、 讨论、心得

在本次实验中，我通过了前一段时间所学的计算机网络的相关知识以及 Java 知识，使用 Socket 对象实现了一个轻量级 WEB 服务器的搭建。在实验的过程中主要遇到如下问题：

1. 在反馈登录结果后，如下图所示，可以看到客户端出现了乱码现象，而通过 Chrome 浏览器自带的抓包插件可以看到，服务端返回的 HTML 网页并没有乱码，因此猜测是字符集出现了问题导致解析时出现乱码，故只需要在返回的 HTML 代码中指定字符集即可： <html><head><meta charset=\"UTF-8\"></head><body>登录失败!</body></html>



2. 在实验过程中，解析 POST 的请求体是一个比较有难度的操作，刚开始我是通过 readline 方法来逐行从输入流 req 中读取请求内容，但是出现了读取始终不停止的问题，查阅资料得，当服务端在读取输入流时，只要不发回响应就不会停止，也就是始终会等待客户端的输入流，为了解决这一问题可以通过 Content-Length 的值来来指定读取的内容而不是通过 while 循环读取。实现代码如下：

```
if (request.getMethod() == Request.Method.POST) {  //POST请求还需要抓取请求体
    int contentLength = 0;
    while ((line = reader.readLine()) != null) {
        if ("".equals(line)) {   //判断是否读到请求头末尾
            break;
        } else if (line.contains("Content-Length")) {  //计算长度
            contentLength = Integer.parseInt(line.substring(line.indexOf("Content-Length") + 16));
        }
    }
    char[] buf;
    if (contentLength != 0) {
        buf = new char[contentLength];
        reader.read(buf,  off: 0, contentLength);
        String[] params = new String(buf).split( regex: "&");  //分割键值对
        for (String item : params) {
            String key = item.substring(0, item.indexOf("="));  //获取键值对
            String val = item.substring(item.indexOf("=") + 1);
            request.params.put(key, val);
        }
    }
}
```