《Java 应用技术》课程实验报告——WEB 服务器

应承峻 3170103456

1. 实验描述

以 Socket 建立一 Web 服务器，除普通 WEB 功能外，可处理简单嵌入程序，比如表达式计算。当文件扩展名为.htm 时，如实显示(静态网页)；当文件扩展名为.zup 时，标识<%、%>之间为程序语句，程序的语法自行设计，也可只是简单表达式。如网页代码为：<%12+5%>则实际显示为：17

其中<%、%>为程序嵌入标记。嵌入程序的语法规范自行确定，可参考 asp 或 jsp。

- 以 IIS 或 TOMCAT 方式均可。
- 可考虑写表单处理、上传、下载组件。

2. 实验思路

本实验类似于 IIS 的方式通过 Socket 建立 Web 服务器。

程序在 main 函数中，创建了一个 Socket 对象，并与 8080 端口绑定，然后通过一个 while 循环持续监听是否有请求，如果有连接则分配一个 HttpServer 对象处理请求。

```
public static void main(String[] args) {
    try {
        ServerSocket serverSocket = new ServerSocket( port: 8080);
        while(true){
            Socket socket = serverSocket.accept();
            new HttpServer(socket, rootPath: "e:/MyServerFiles/").start();
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

由于该进程需要能够同时处理多个请求，因此，我们设计的 HttpServer 类应该是继承于 Thread 类，并重写其 run 的方法。

```
public class HttpServer extends Thread {

    private InputStream req;
    private OutputStream res;
    private String root;

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

    @Override
    public void run() {
        try {
            response(getReqPath());
        } catch (IOException | ParseException e) {
            e.printStackTrace();
        }
    }
```

当 HttpServer 类的一个对象被创建的时候，首先先通过 Socket 打开输入输出流，然后执行 run 中的 getReqPath()方法得到请求的资源的名称，再通过 response()方法将请求结果通过输出流返回到客户端。

由于 HTTP 的请求报文通常由请求方法、请求 URL、HTTP 协议及版本以及报文头和报文体组成，而请求的 URL 又位于请求报文的第一行，因此我们只需要得到请求报文中第一行的第二个单词即可得到请求的资源。具体的实现方法是，先通过 InputStreamReader 将字节流转换成字符流，然后通过 BufferReader 来提高字符流读取的效率，再通过 BufferReader 的 readline()方法读取第一行，通过 split 方法进行单词分割，从而得到请求文件的路径。

```
/**
 * 获取请求的文件
 *
 * @return 请求文件路径
 * @apiNote HTTP请求报文中第一行为：请求方法 请求URL HTTP协议及版本。  eg: POST /index.htm HTTP/1.1
 * @apiNote HTTP请求中还有报文头和报文体
 */
private String getReqPath() throws IOException, ParseException {
    //通过字符流和缓冲区读入请求
    BufferedReader reader = new BufferedReader(new InputStreamReader(req));
    //分割参数
    String line = reader.readLine();
    String[] list = line.split( regex: " ");
    if (list.length != 3) throw new ParseException("HTTP request header parse Error!", 0);
    return list[1];
}
```

HTTP 响应则由 HTTP 协议版本、状态码等信息组成。对于一个文件的请求，首先需要判断该文件是否存在，如果不存在则返回 404，如果存在则根据文件的类型判断，如果是 zup 文件则需要调用 parseZupFile()方法对标签进行处理。

```
/**
 * 将响应结果和文件返回给客户端
 *
 * @param filepath 响应的文件路径
 */
private void response(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
        BufferedReader reader = new BufferedReader(new FileReader(file));  //得到文件字节流
        StringBuffer fileBuffer = new StringBuffer();
        StringBuffer resBuffer = new StringBuffer();
        String line;
        while ((line = reader.readLine()) != null) {
            fileBuffer.append(line).append("\r\n");
        }
        if (getPostFixName(file.getName()).equals(".zup")) fileBuffer = parseZupFile(fileBuffer);
        resBuffer.append("HTTP/1.1 200 ok \r\n")
                .append("Content-Language:zh-CN \r\n")
                .append("Content-Type:text/html;charset=UTF-8 \r\n")
                .append("Content-Length:").append(fileBuffer.toString().getBytes().length).append("\r\n")
                .append("\r\n").append(fileBuffer.toString());
        res.write(resBuffer.toString().getBytes());
        res.flush();
        res.close();
    } else {  //文件不存在
        StringBuffer err = new StringBuffer();
        err.append("HTTP/1.1 404 file not found \r\n").append("Content-Type:text/html \r\n")
                .append("Content-Length:20 \r\n").append("\r\n").append("<h1>404 Not Found</h1>");
        res.write(err.toString().getBytes()); //输出字节流
        res.flush();
        res.close();
    }
}
```

对标签的处理并不是特别复杂，由于<% %>标签不允许嵌套，所以只需要通过一个 while 循环，不断地寻找存在的<% %>标签，找到一个则将其用表达式计算的结果替换掉即可。

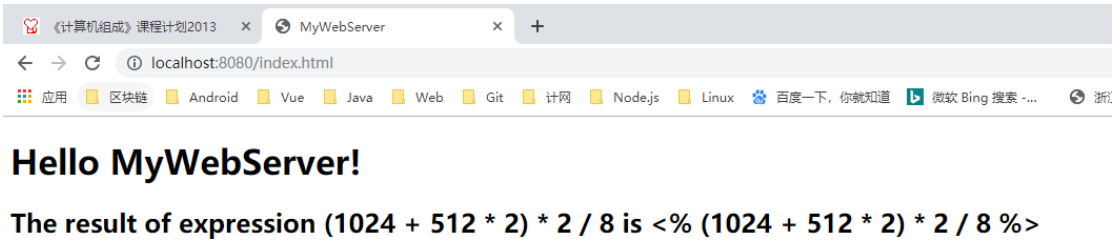
```
public static StringBuffer parseZupFile(StringBuffer buffer) {
    int openTag;
    while ((openTag = buffer.indexOf("<% ")) >= 0) {
        int closeTag = buffer.indexOf(" %>");
        if (closeTag < 0) break;
        String content = buffer.substring(openTag + 2, closeTag);
        try {
            buffer.replace(openTag, end: closeTag + 3, String.valueOf(Expr.calc(content.trim())));
        } catch (ScriptException e) {
            buffer.replace(openTag, end: closeTag + 3, str: "");
        }
    }
    return buffer;
}
```

3. 实验结果

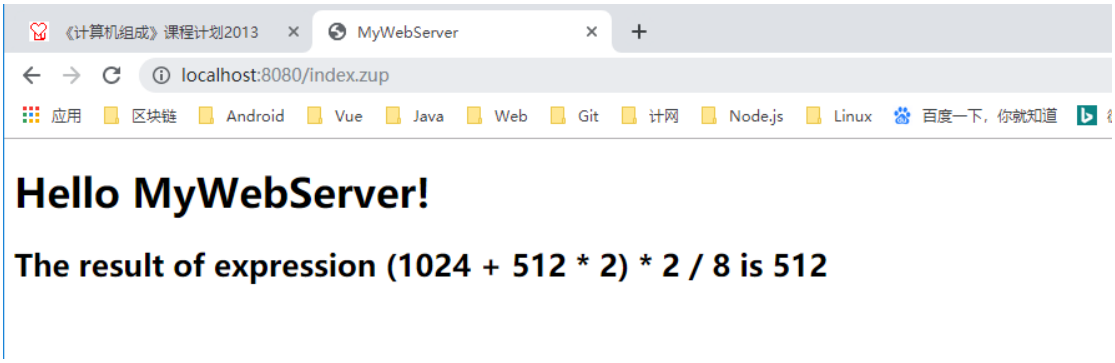
在本次实验中，我们通过验证两个文件”index.html”和”index.zup”来判断程序是否能够运行，两个文件的内容相同，均为：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>MyWebServer</title>
</head>
<body>
<h1>Hello MyWebServer!</h1>
<h2>The result of expression (1024 + 512 * 2) * 2 / 8 is <% (1024 + 512 * 2) * 2 / 8   %> </h2>
</body>
</html>
```

当启动程序并在浏览器中输入 localhost:8080/index.html 时，运行结果如下：



而在浏览器中输入 localhost:8080/index.html 时，运行结果如下：



如果文件不存在则返回 404 Not Found：

由此完成测试，程序能够实现在.zup 文件中对标签内容进行解析。

4. 实验心得

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，使用 Socket 对象来实现 WEB 服务器的搭建。实验中遇到的一个比较棘手的问题是，当请求 zup 文件时，能够在控制台中正常输出解析后的网页而无法在网页中输出，请求 html 文件则是正常的。通过 F12 调出 chorome 控制台排查后发现了“ERR_CONTENT_LENGTH_MISMATCH”这一错误，查阅网上资料得到，CONTENT_LENGTH 代表的是响应体中发送内容的长度或大小（即我们发送的 index 文件的大小），当实际发送的大小超出 CONTENT_LENGTH 时，浏览器只接收 CONTENT_LENGTH 长度的部分，而将超出部分截断，而实际发送的长度不足时，浏览器则会一直等待剩下的字节直到超时。再对程序进行排查发现原来程序中 CONTENT_LENGTH 的来源是 file.length()，当我们请求 zup 文件时，通过函数对该文件中标签进行替换，所以导致实际长度变小，即实际长度<CONTENT_LENGTH，从而导致了这种情况的发生，因此只需替换成 fileBuffer.toString().getBytes().length 即可。

```
resBuffer.append("HTTP/1.1 200 ok \r\n")
        .append("Content-Language:zh-CN \r\n")
        .append("Content-Type:text/html;charset=UTF-8 \r\n")
        .append("Content-Length:").append(fileBuffer.toString().getBytes().length).append("\r\n")
        .append("\r\n").append(fileBuffer.toString());
```

。