华东师范大学数据科学与工程学院期末项目报告

课程名称：计算机网络与编程	年级：2021	上机实践成绩：
指导教师：张召	姓名：唐小卉	学号：10215501437
上机实践名称： WebServer & ProxyServer		上机实践日期：2023.6.12
上机实践编号：	组号：	上机实践时间：2023.6.12

一、题目要求

实现 Web Server

题目 1.1：请使用 Java 语言开发一个简单的 Web 服务器，它能处理 HTTP 请求。具体而言，你的 Web 服务器将：

1. 当一个客户端（浏览器）联系时创建一个连接套接字；
2. 从这个连接套接字接受 HTTP 请求；
3. 解释该请求以确定所请求的特定文件；
4. 从文件系统中获得请求的文件；
5. 创建一个由请求的文件组成的 HTTP 响应报文；
6. 经 TCP 连接向请求的浏览器返回响应。

功能要求:

请使用 ServerSocket 和 Socket 进行代码实现；

请使用多线程接管连接；

在浏览器中输入localhost:8081/index.html 能显示自己的学号信息（编写简单的 index.html）；

在浏览器中输入localhost:8081 下其他无效路径，浏览器显示 404 not found；

在浏览器中输入localhost:8081/shutdown 能使服务器关闭；

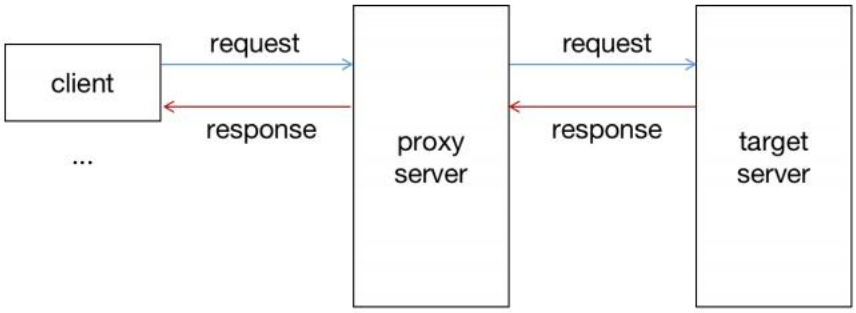
使用 postman 进行测试，测试 get 和 post 两种请求方法。

题目 1.2：

在题目 1.1 的基础上实现代理服务器，让浏览器请求经过你的代理来请求 Web 对象。具体而言：

1. 当你的代理服务器从一个浏览器接收到对某个对象的 HTTP 请求时，它生成对相同对象的一个新的 HTTP 请求并向初始服务器发送；
2. 当该代理从初始服务器接收到具有该对象的 HTTP 相应时，它生成一个包括该对象的新的 HTTP 响应，并发送给该客户。

代理服务器图示如下：



功能要求:

请在题 1.1 的代码上进行修改，使用 ServerSocket 和 Socket 进行代码实现；

请分别使用浏览器（可设置浏览器代理）和 postman，并进行代理测试。

代理服务器图示如下：

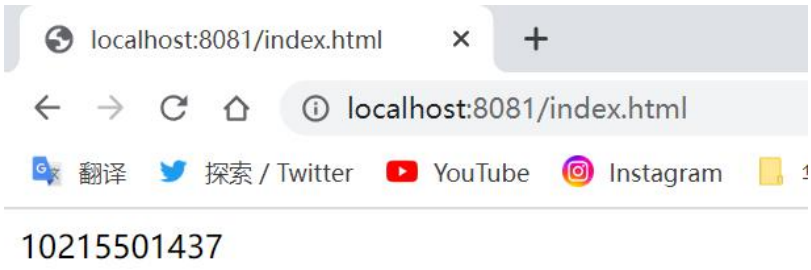
性能测试

使用 JMeter 进行压测，在保证功能完整的前提下测试每秒响应的请求数。

Bonos (optional)：分析当前能支持同时连接的最大数，使用学习过的 NIO 修改代码使服务器能同时支持并发的 1000 个连接。（注意 JMeter 中的集合点设置）

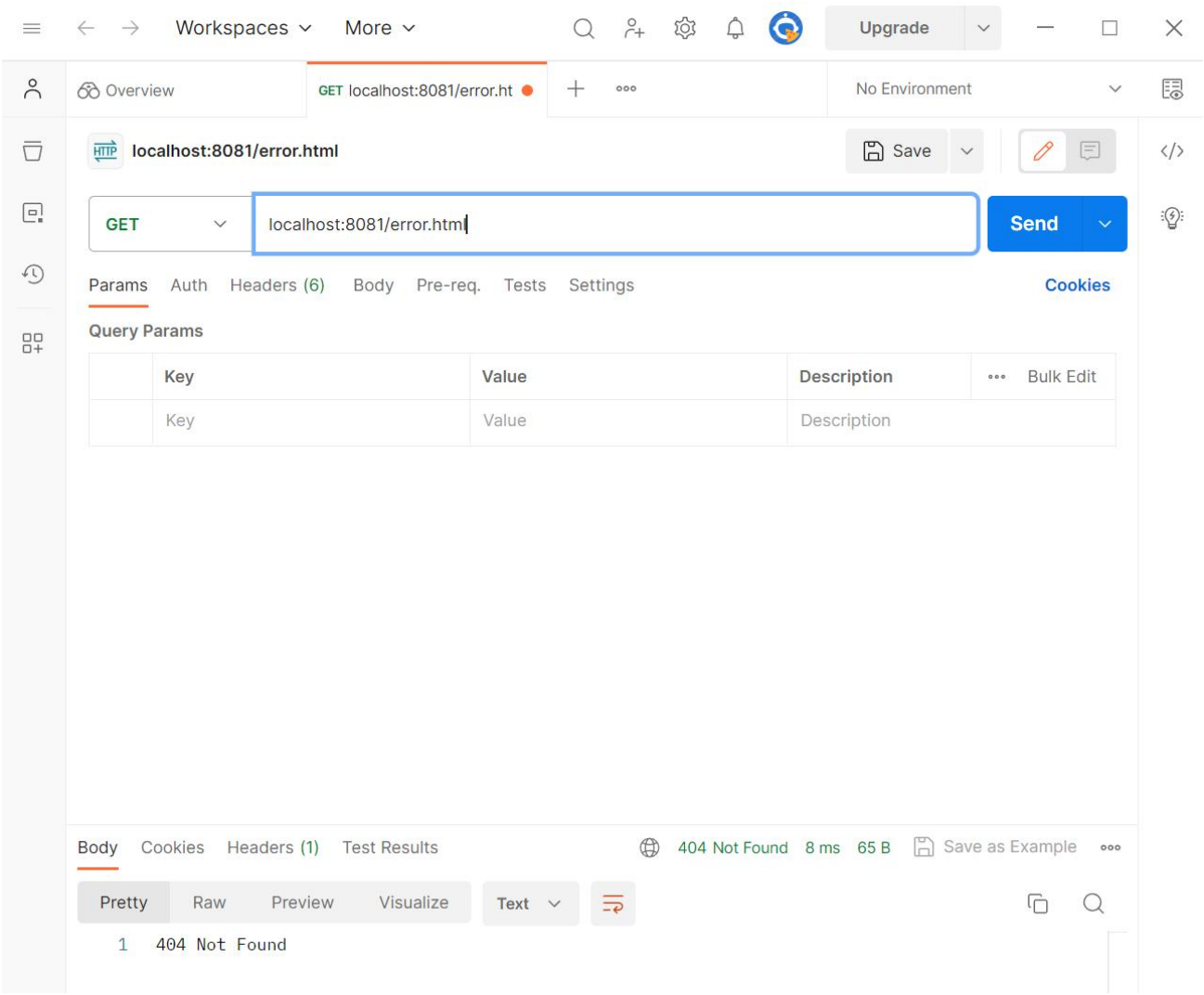
二、 功能实现情况

1.浏览器访问 localhost:8081/index.html，显示自己的学号信息：

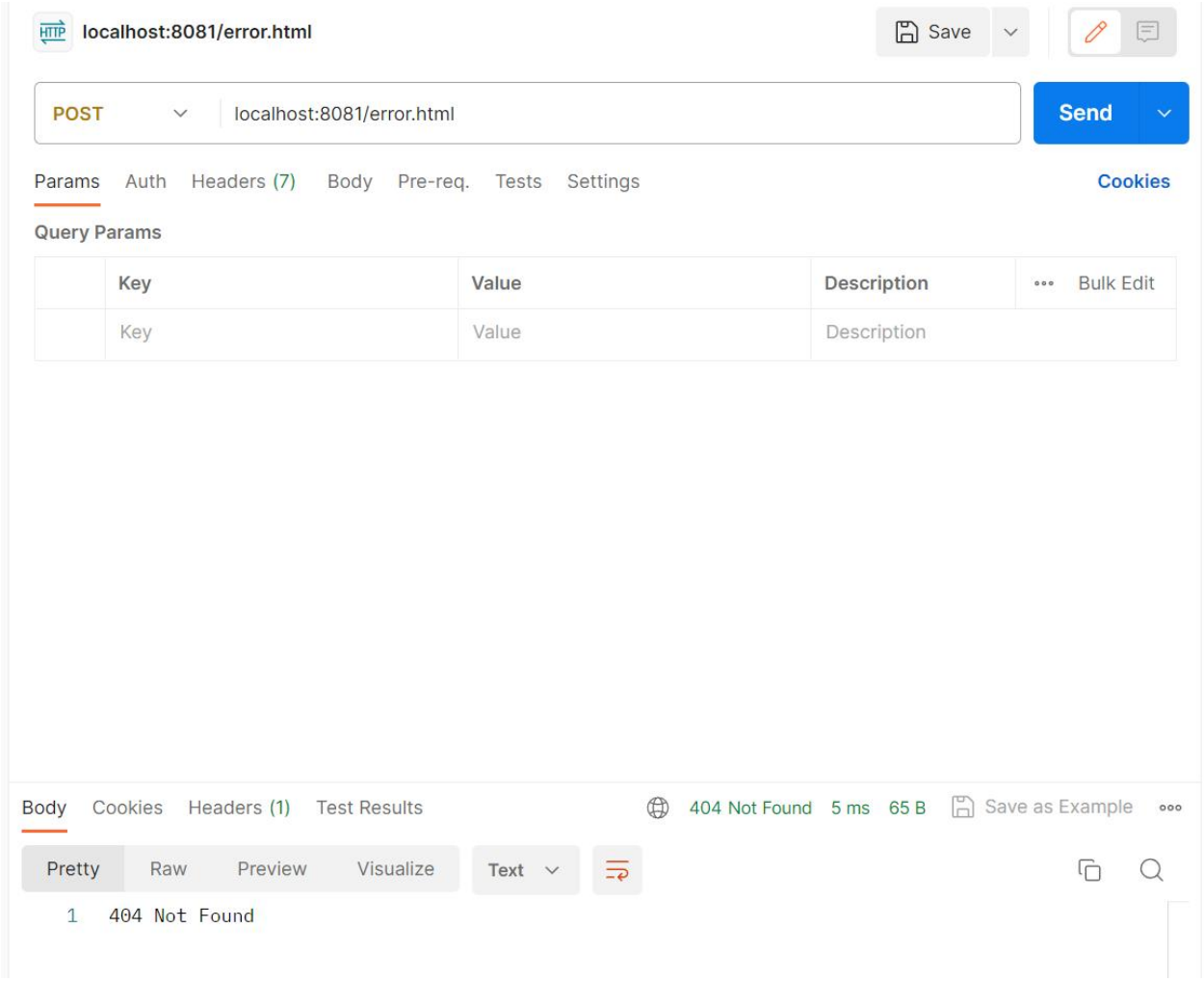


2.使用 postman 测试 get/post，访问 localhost:8081 下其他无效路径，显示 404 not found

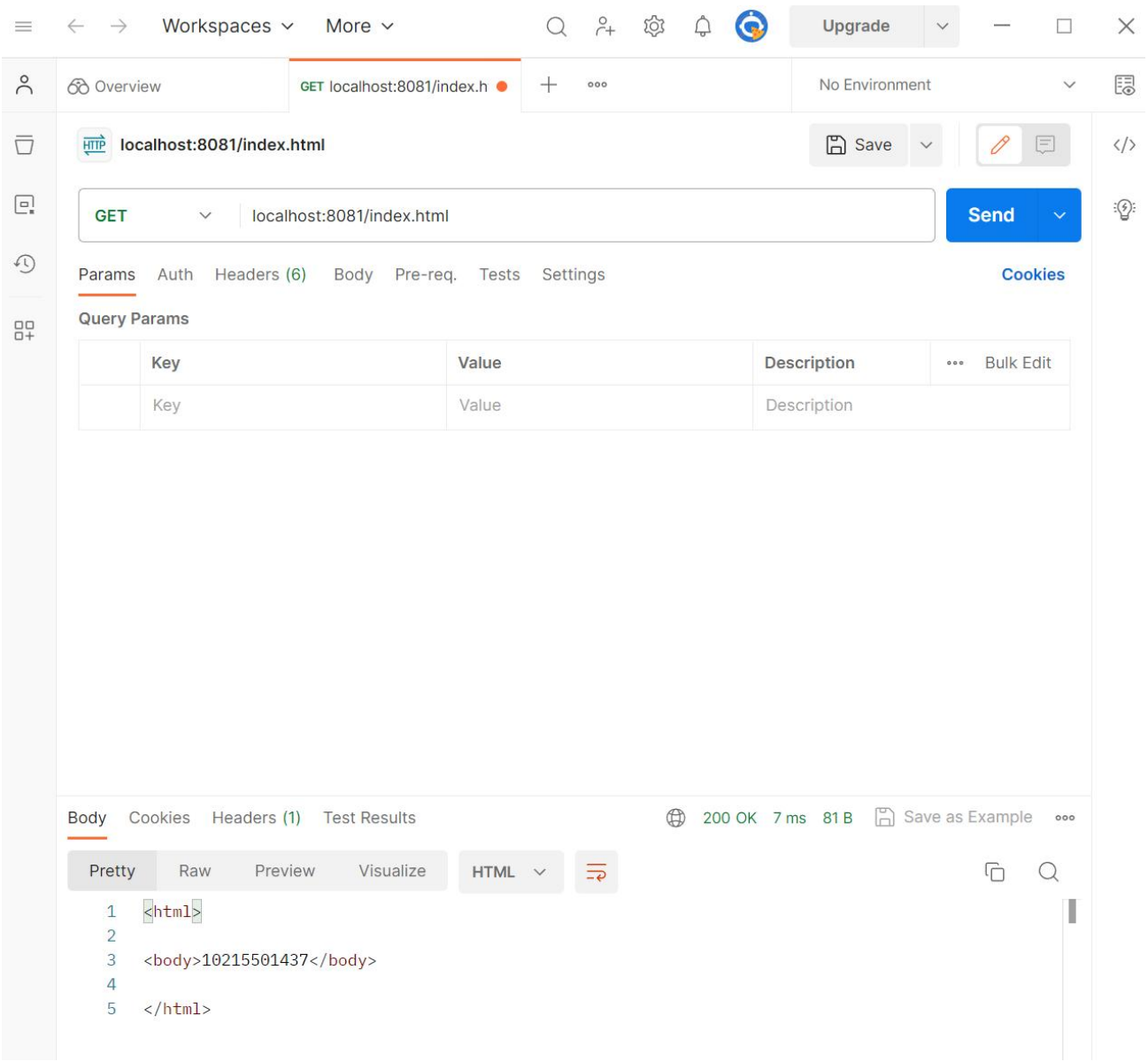
GET：


Workspaces
More
Upgrade
Overview
GET localhost:8081/error.ht
No Environment
localhost:8081/error.html
Save
GET
localhost:8081/error.html
Send
Params
Auth
Headers (6)
Body
Pre-req.
Tests
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (1)
Test Results
404 Not Found
8 ms
65 B
Save as Example
Pretty
Raw
Preview
Visualize
Text
1 404 Not Found

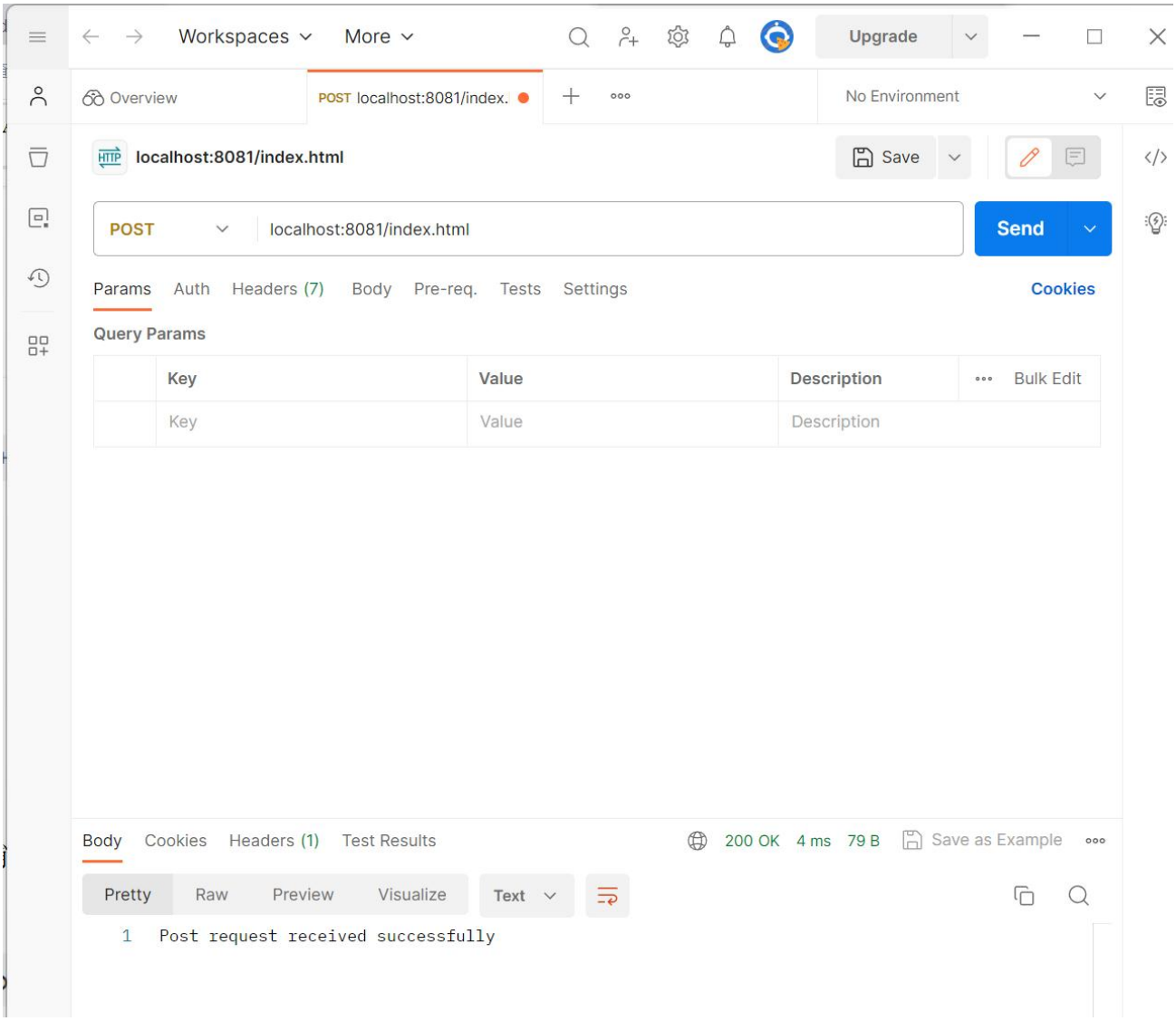
POST：



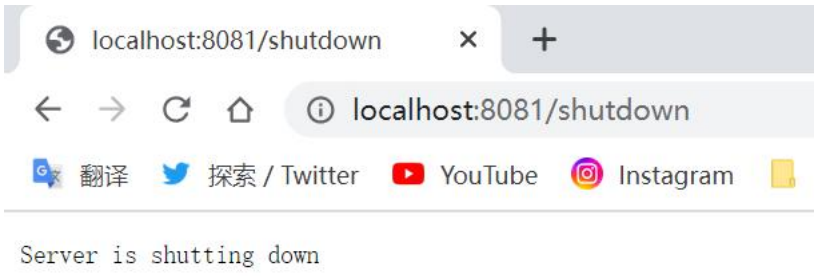
同时，如果访问 index.html，GET 请求会返回 HTTP 相应报文：



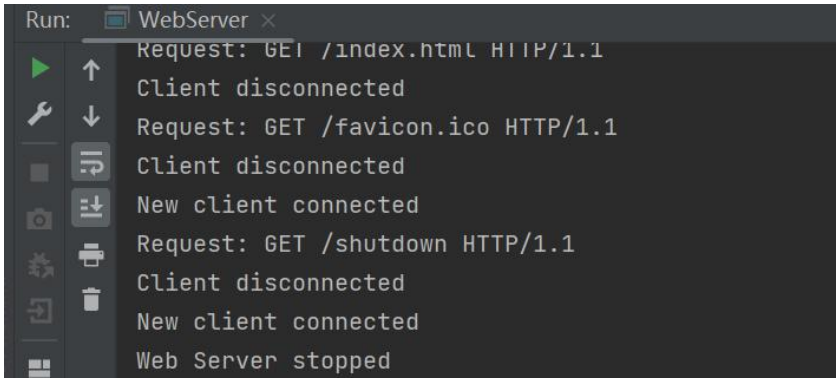
在代码中，我设置如果访问 index.html（有效路径）会返回 Post request received successfully，如果访问非有效路径，则会返回 404 not found。测试结果如下：



3.浏览器访问 localhost:8081/shutdown 能使服务器关闭

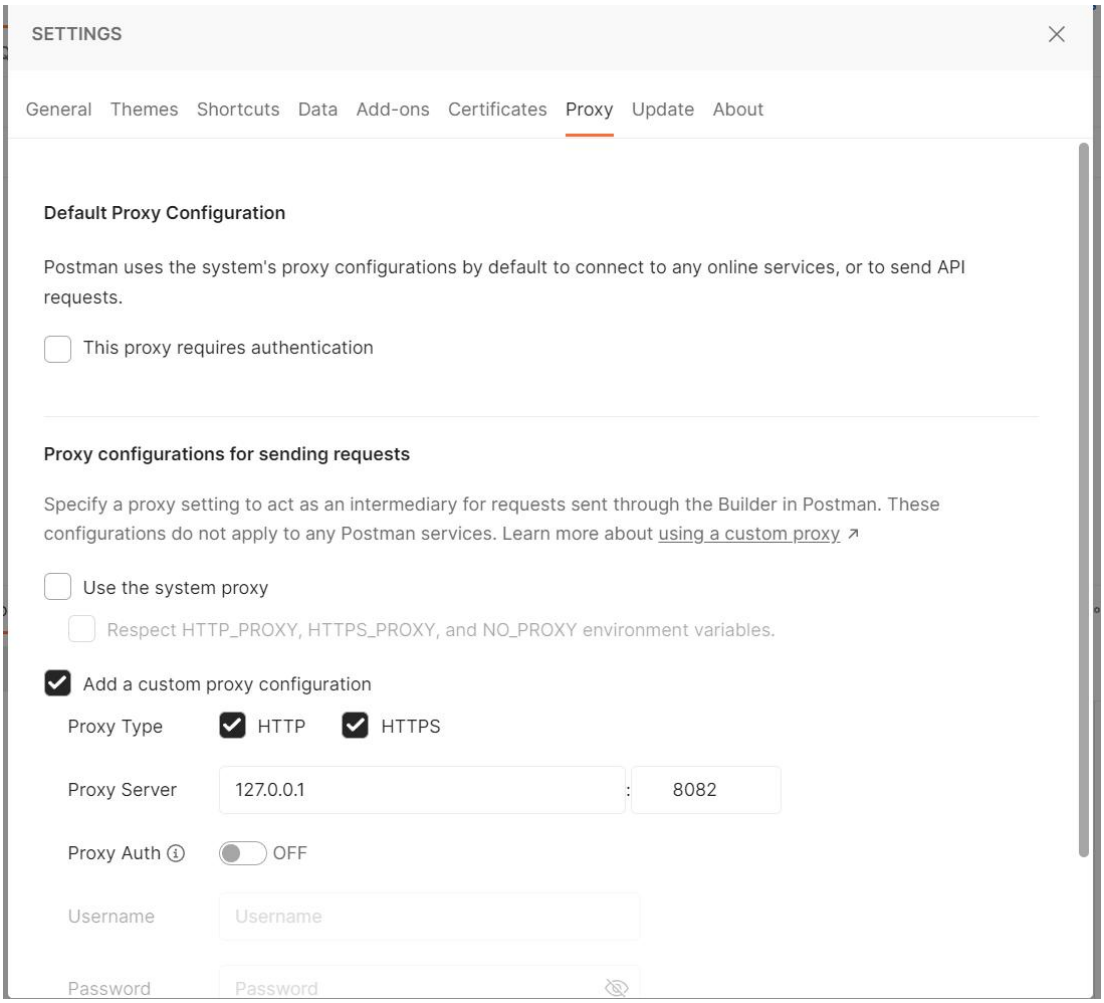


同时可以观察到 IJ 的 WebServer.java 终止运行了

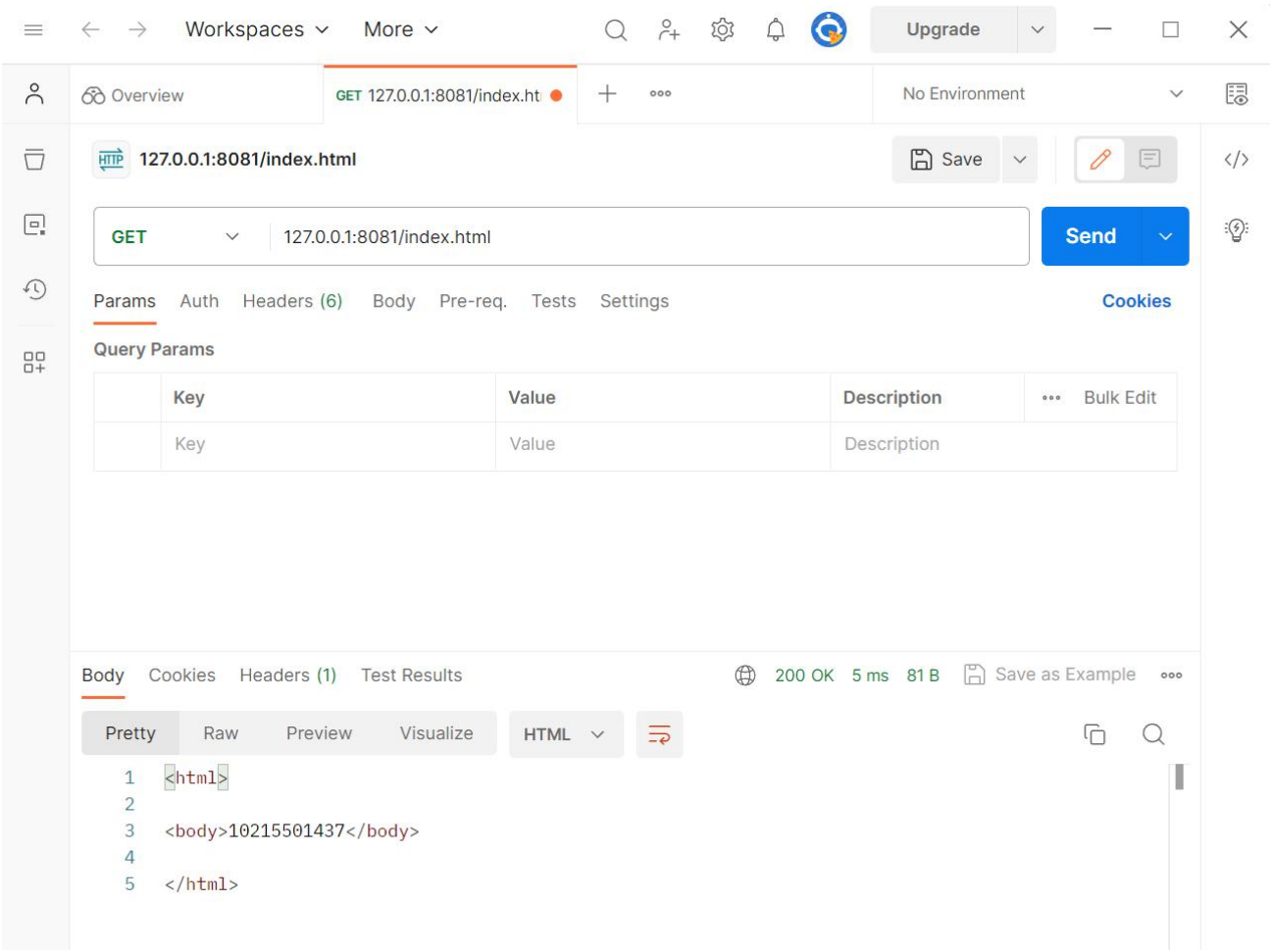


4.使用 postman 设置代理，启动第一题的 httpserver（监听在 8081 端口），和第二题的 proxyserver，使用 postman 测试是否能成功经过代理访问 localhost:8081/index.html，在 postman 界面显示自己的学号信息

首先，在 postman 的 settings 中设置代理：

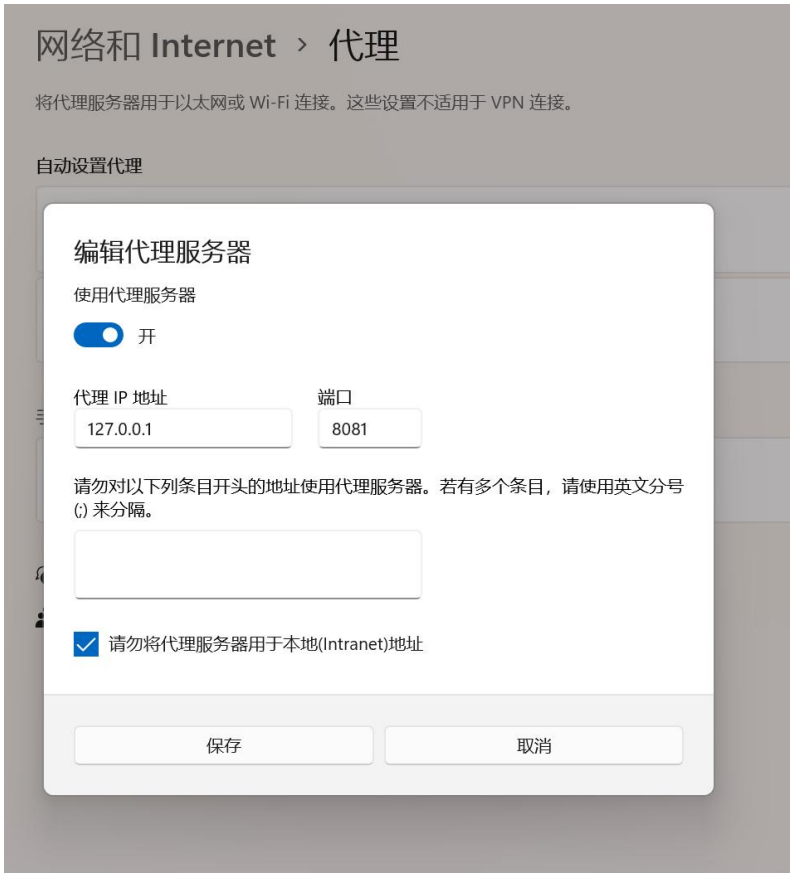


之后发送 GET 请求，结果如图所示：



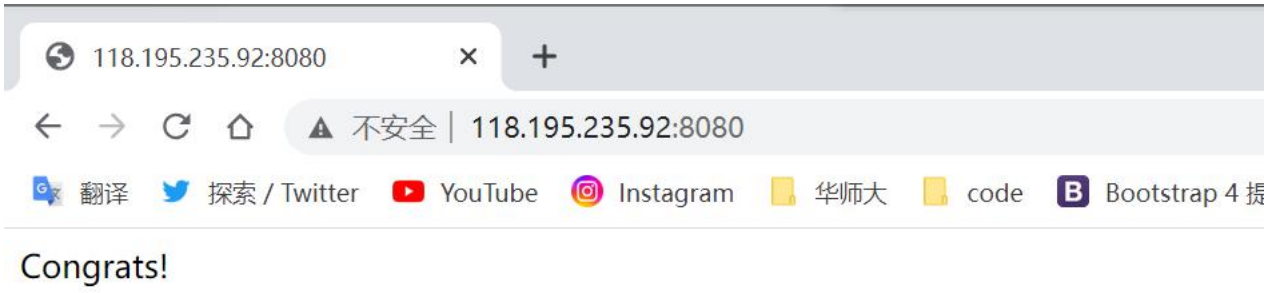
5.使用系统提供的代理，启动第二题的 proxyserver，使用浏览器测试是否能成功访问 http://www.baidu.com/search/error.html

首先，设置浏览器代理：

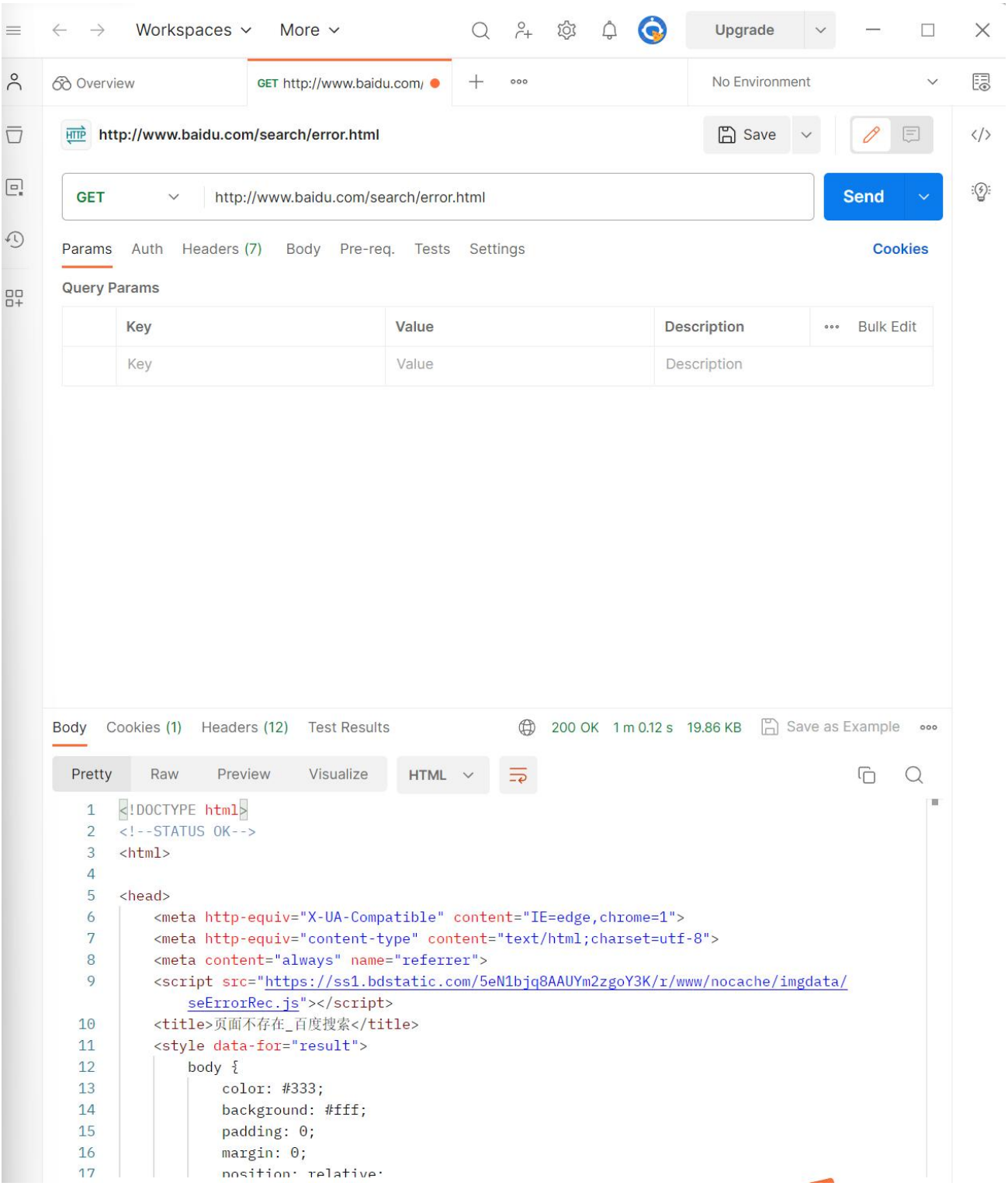


访问结果如下所示

给定网址：



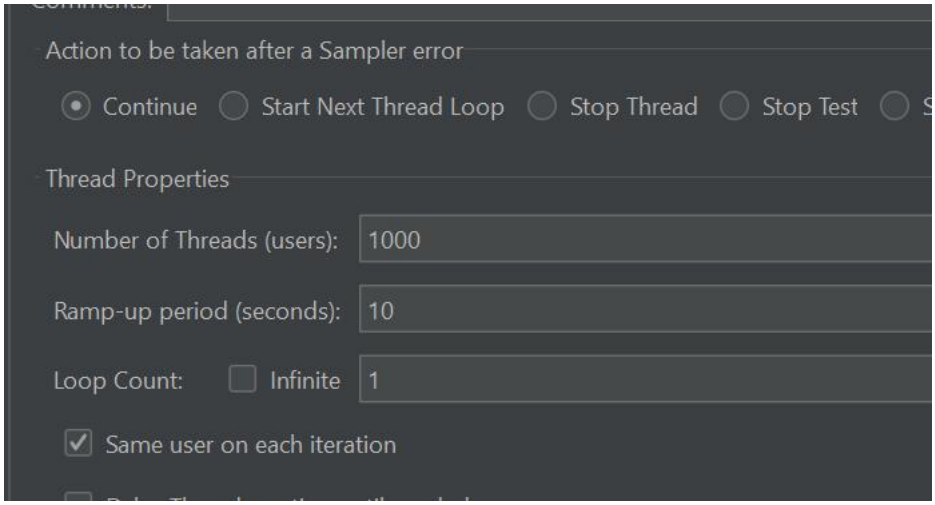
6.使用 postman 设置代理，启动第二题的 proxyserver，使用 postman 测试是否能成功访问考察时给出的地址

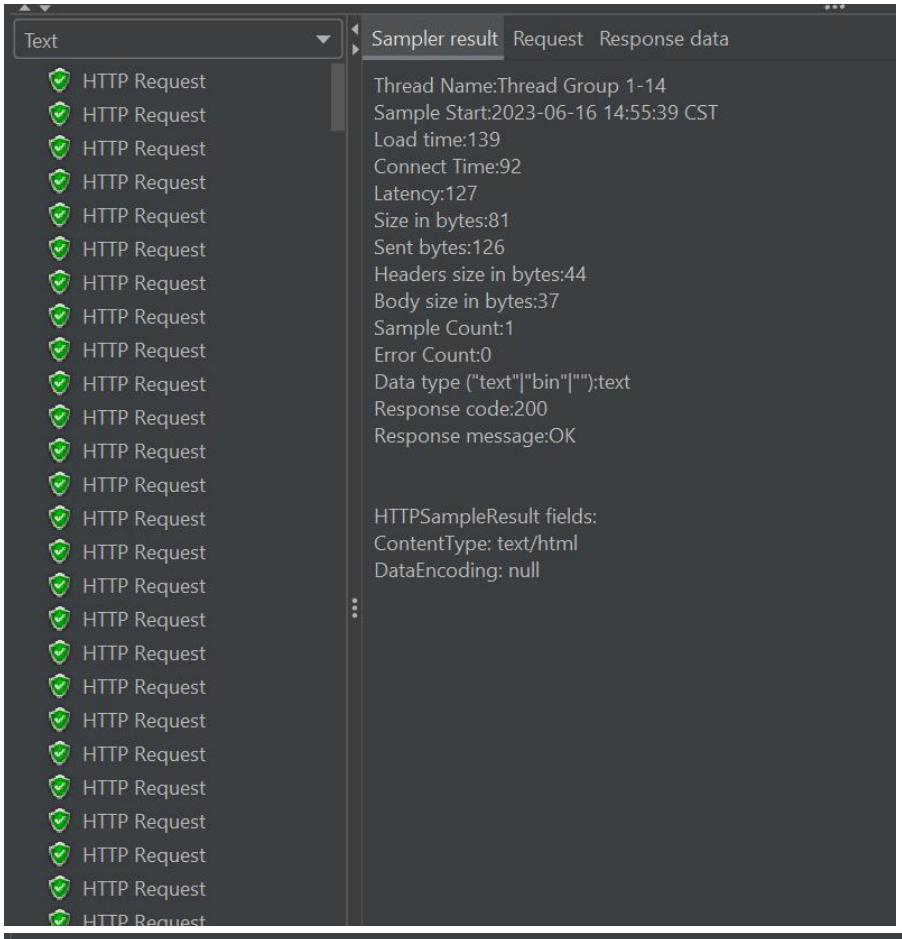


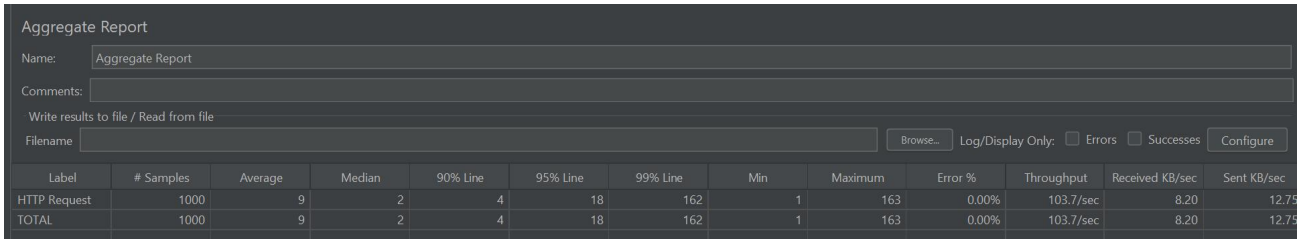
三、性能测试情况

1.使用 JMeter 测试，得出聚合结果

2.【基本要求】参数为(1000,10)时，吞吐 100/sec 左右 ，错误率 0~10%(10')；







Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/sec	Sent KB/sec
HTTP Request	1000	9	2	4	18	162	1	163	0.00%	103.7/sec	8.20	12.75
TOTAL	1000	9	2	4	18	162	1	163	0.00%	103.7/sec	8.20	12.75

四、总结

通过本次对 WebServer 和 ProxyServer 的编写，我学到了以下知识：

1.编写基本的代理服务器：你学会了如何编写一个简单的代理服务器，它能够接收客户端的请求并将其转发给初始服务器，然后将初始服务器的响应返回给客户端。

2.使用 Java Socket 进行网络通信：你熟悉了使用 Java Socket 类进行网络通信的基本步骤，包括创建 Socket 对象、获取输入流和输出流等。

3.处理 HTTP 请求和响应：你了解了如何解析 HTTP 请求报文，包括请求方法、请求路径等信息，并且学会了构造和发送 HTTP 响应报文。

4.多线程编程：你学会了如何使用多线程来处理多个客户端请求，每个请求在独立的线程中进行处理，从而实现并发处理能力。

在实验过程中，我也遇到了一些困难，比如在使用 JMeter 进行压力测试时，你可能需要熟悉 JMeter 的配置和使用方法，我对如何配置目标地址、端口和路径，以及如何分析测试结果和性能指标并不了解，出现了端口监听错误的情况；同时最开始写 WebServer 的时候因为代码错误导致无法关闭服务器，向助教请教了之后，在 cmd 中找到了占用 8081 窗口的 PID 并 kill 掉了。在检测代码逻辑的时候我还学会了利用浏览器的“检查”功能，在 Network 里面找到了报文的内容。

总之，通过这个过程，我不仅学到了如何编写代理服务器和处理 HTTP 请求，还锻炼了调试和问题解决的能力，同时也了解了性能测试和压力测试的基本概念和工具的使用。这些知识和经验将为我在网络编程和性能优化方面打下坚实的基础。

附录：WebServer 和 ProxyServer 的代码

WebServer：

```java
package Final;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

no usages
public class WebServer {
    2 usages
    private static boolean isRunning = true; // 布尔值控制服务器是否继续运行，如果执行shutdown就停止运行

    // 从指定长端口接受客户端连接请求并创建一个新的线程处理每个客户端的请求
    public static void main(String[] args) {
        int port = 8081; // 服务器监听的端口号
        // try就像一个网，把try{}里面的代码所抛出的异常都网住，然后把异常交给catch{}里面的代码去处理。
        try {
            // 创建一个服务器端Socket，鉴定指定的端口
            ServerSocket serverSocket = new ServerSocket(port);
            System.out.println("Web Server listening on port " + port);

            //当服务器仍然在运行的时候
            while (isRunning) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected");

                // 创建一个新的线程来处理客户端请求
                Thread thread = new Thread(new ClientHandler(clientSocket));
                thread.start();
            }

            serverSocket.close();
            System.out.println("Web Server stopped");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```java
// ClientHandler是一个内部类，实现了Runnable接口，用于处理客户端请求的线程
1 usage
static class ClientHandler implements Runnable {
    4 usages
    private Socket clientSocket; // 新的Socket用于表示与客户端的连接

    // 初始化并传入客户端的Socket对象
    1 usage
    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    // 表示线程的执行逻辑
    @Override
    public void run() {
        try {
            // 获取输入流和输出流
            BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream());

            // 读取请求报文的第一行
            String requestLine = in.readLine();
            System.out.println("Request: " + requestLine);

            // 判空处理(POST请求需要判断)
            if (requestLine == null) {
                System.out.println("Invalid request");
                return; // 终止当前请求处理
            }

            // 解析请求方法和请求路径
            String[] requestParts = requestLine.split( regex: " "); // 用空格拆分请求行
            String method = requestParts[0]; // 存储请求方法
            String path = requestParts[1]; // 存储请求路径
```

```
// 处理GET请求
if (method.equals("GET")) {
    if (path.equals("/index.html")) {
        // 构造HTTP响应报文
        String response = "HTTP/1.1 200 OK\r\n" +
                "Content-Type: text/html\r\n" +
                "\r\n" +
                "<html><body>10215501437</body></html>";

        // 发送响应
        out.write(response);
        out.flush();
    } else if (path.equals("/shutdown")) {
        // 关闭服务器
        String response = "HTTP/1.1 200 OK\r\n" +
                "Content-Type: text/plain\r\n" +
                "\r\n" +
                "Server is shutting down";

        out.write(response);
        out.flush();
        isRunning = false;
    } else {
        // 构造404响应报文
        String response = "HTTP/1.1 404 Not Found\r\n" +
                "Content-Type: text/plain\r\n" +
                "\r\n" +
                "404 Not Found";

        // 发送响应
        out.write(response);
        out.flush();
    }
```

```
            } else if (method.equals("POST")) {
                // 处理POST请求
                if (path.equals("/index.html")) {
                    // 处理有效路径的逻辑
                    String response = "HTTP/1.1 200 OK\r\n" +
                            "Content-Type: text/plain\r\n" +
                            "\r\n" +
                            "Post request received successfully";

                    // 发送响应
                    out.write(response);
                    out.flush();
                } else {
                    // 构造404响应报文
                    String response = "HTTP/1.1 404 Not Found\r\n" +
                            "Content-Type: text/plain\r\n" +
                            "\r\n" +
                            "404 Not Found";

                    // 发送响应
                    out.write(response);
                    out.flush();
                }
            }

            // 关闭连接
            clientSocket.close();
            System.out.println("Client disconnected");
        } catch (IOException e) {
            e.printStackTrace();
        }// catch用来捕捉可能发生的IO异常并打印异常堆栈跟踪
    }
}
}
```

ProxyServer：

```java
package Final;

import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;

no usages
public class ProxyServer {
//    private static final String SERVER_HOST = "127.0.0.1";//初始服务器的主机名
    2 usages
    private static final String SERVER_HOST = "180.101.50.188"; // 初始服务器的主机名
    1 usage
    private static final int SERVER_PORT = 80; // 初始服务器的端口号

    public static void main(String[] args) {
        int port = 8082; // 代理服务器监听的端口号

        try {
            // 创建代理服务器的Serversocket对象，用于接收客户端的连接请求
            ServerSocket serverSocket = new ServerSocket(port);
            System.out.println("Proxy Server is listening on port " + port);

            while (true) {
                // 等待客户端连接，并返回与客户端通信的Socket对象。
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected");

                // 创建一个新的线程来处理客户端请求
                Thread thread = new Thread(new ClientHandler(clientSocket));
                thread.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
1 usage
static class ClientHandler implements Runnable {
    4 usages
    private Socket clientSocket;// 新的Socket用于表示与客户端的连接

    // 初始化并传入客户端的Socket对象
    1 usage
    public ClientHandler(Socket clientSocket) { this.clientSocket = clientSocket; }

    @Override
    public void run() {
        try {
            // 获取输入流和输出流
            BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream())); // 从客户端socket读取输入流
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream()); // 向客户端socket输出流写入数据

            // 读取请求报文的第一行
            String requestLine = in.readLine();
            System.out.println("Request: " + requestLine); // 用于调试

            // 创建与初始服务器的连接
            Socket serverSocket = new Socket(SERVER_HOST, SERVER_PORT);
            // 从初始服务器Socket的输入流中获取数据
            BufferedReader serverIn = new BufferedReader(new InputStreamReader(serverSocket.getInputStream()));
            // 通过初始服务器Socket的输出流发送数据
            PrintWriter serverOut = new PrintWriter(serverSocket.getOutputStream());

            // 发送与初始服务器相同的HTTP请求
            serverOut.println(requestLine);
            serverOut.println("Host: " + SERVER_HOST);
            serverOut.println(); // 头部和请求体之间的空行，表示请求结束
            serverOut.flush();
```

```
            // 读取初始服务器的响应并转发给客户端
            String line;
            while ((line = serverIn.readLine()) != null) {
                out.println(line);
            } // 从初始服务器的输入流读取响应数据，并将其逐行写入客户端的输出流，实现数据的转发
            out.flush(); // 刷新客户端的输出流，确保数据发送到客户端

            // 关闭Socket
            clientSocket.close();
            serverSocket.close();

            System.out.println("Client disconnected");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```