华东师范大学数据科学与工程学院实验报告

课程名称：计算机网络与编程　　**年级**：2020 级　　**上机实践成绩**：

指导教师：张召　　**姓名**：张熙翔　　**学号**：10205501427

上机实践名称：Web Server　　**上机实践日期**：2022/6/20

一、实验目的

- 开发一个简单的 Web 服务器，它仅能处理一个请求。
- 开发一个简单的 Web 代理服务器。

二、实验任务

题目一：在这个编程作业中，你将用 Java 语言开发一个简单的 Web 服务器，它仅能处理一个请求，具体而言，你的 Web 服务器将：

1．当一个客户(浏览器)联系时创建一个连接套接字；

2．从这个连接接收 HTTP 请求；

3．解释该请求以确定所请求的特定文件；

4．从服务器的文件系统获得请求的文件；

5．创建一个由请求的文件组成的 HTTP 响应报文，报文前面有首部行；

6．经 TCP 连接向请求的浏览器返回响应。

具体要求：

- 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 能显示出自己的学号；
- 在浏览器中输入 localhost:8081 下其他无效路径显示 404not found；
- 在浏览器中输入 localhost:8081/shutdown 能使服务器关闭；
- 使用 postman 再次进行测试，测试 get/post 两种请求方法。

题目二：在这个编程作业中，你将用 Java 语言开发一个简单的 Web 代理服务器，让浏览器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，他生成对相同对象的一个新 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 响应时，它生成一个包括该对象的新 HTTP 响应，并发送给该客户；

3．这个代理将是多线程的，使其在同一时间能够处理多个请求。

具体要求：

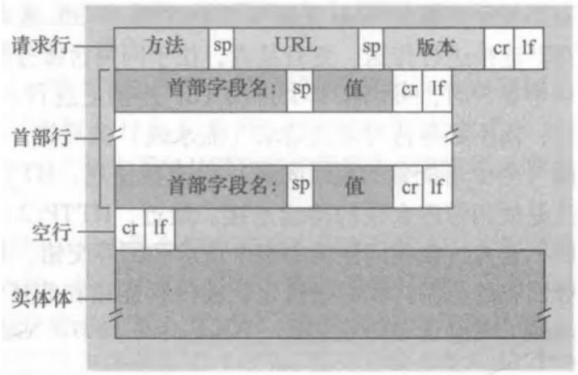
- 在题目一的代码上进行修改，使用 ServerSocket 和 Socket 进行代码实现；
- 请使多线程接管连接(最好使用线程池)；
- 请分别使用浏览器和 postman 进行代理的测试。

三、使用环境

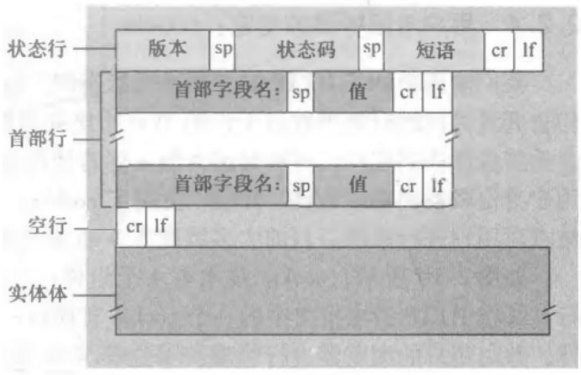
IntelliJ IDEA 2021.3.2
JDK 17
Netty 4.1.42 Final

四、实验过程

Version I：实现题目一（多线程）



HTTP 的请求报文的通用格式



HTTP 的响应报文的通用格式

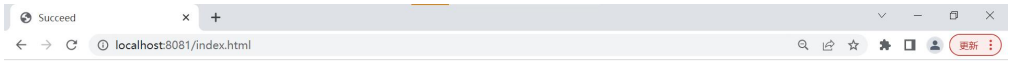
代码如下：

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
//多线程
public class WebServer {
    public static void main(String[] args) {
        try{
            //使用 serversocket 创建服务对象
            ServerSocket server = new ServerSocket(8081);
            while(true){
                try {
                    //使用服务器对象的输入流获取客户端发来的数据
                    Socket socket = server.accept();
                    new Thread(new Runnable() {
                        @Override
                        public void run() {
                            try {
                                InputStream is = socket.getInputStream();
                                //将字节流转换为字符流 效率低
                                InputStreamReader isr = new InputStreamReader(is);
                                //将字符流包装成缓冲流
                                BufferedReader br = new BufferedReader(isr);
                                String line = br.readLine();
                                System.out.println(line);
                                int flag=1;
                                //从 line 中解析文件名
                                String[] linesArr = line.split(" ");
                                //String fileName = linesArr[1].substring(1);
                                          //GET /index.html HTTP/1.1
                                String fileName2 = linesArr[1];// /index.html
                                System.out.println(fileName2);
                                String[] fileName1 = fileName2.split("/");
                                String fileName=fileName1[fileName1.length-1];
                                System.out.println(fileName);
                                if(fileName.equals("shutdown")){
                                    System.exit(0);
                                }
                                File file = new File(fileName);
                                if(file.exists()==false){
```

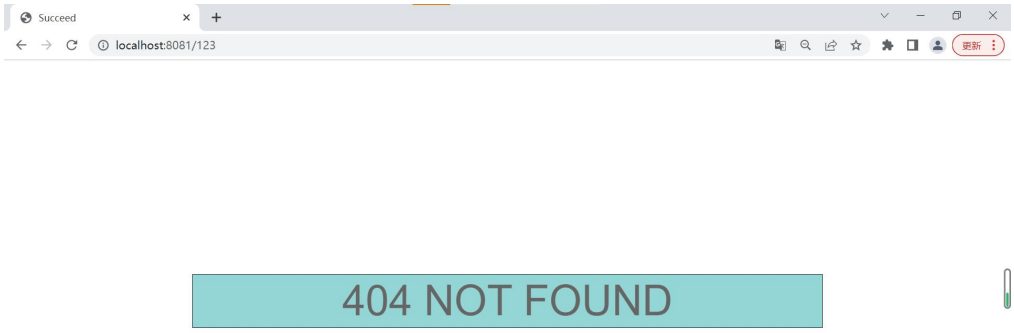
```
                                fileName = "error.html";
                                flag = 0;
                            }
                            //从磁盘读取文件
                            FileInputStream fis = new FileInputStream(fileName);
                            BufferedInputStream bis = new BufferedInputStream(fis);
                            byte[] data = new byte[1024];
                            int length = 0;
                            BufferedOutputStream bos = new
BufferedOutputStream(socket.getOutputStream());

                            //先返回响应头信息
                            if(flag == 0){
                                bos.write("HTTP/1.1 404 NOT_FOUND \r\n".getBytes());
                            }else{
                                bos.write("HTTP/1.1 200 OK \r\n".getBytes());
                            }
                            bos.write("content-type: text/html; charset=utf-8
\r\n".getBytes());

                            //响应头和数据之间有一个空行
                            bos.write("\r\n".getBytes());

                            //再返回数据
                            while((length = bis.read(data))!=-1){
                                //将读取的文件输出到网络
                                bos.write(data,0,length);
                            }
                            bos.close();
                            bis.close();
                            br.close();
                            socket.close();
                        }catch (Exception e){
                        }
                    }
                }).start();
            }catch (Exception e){
                e.printStackTrace();
            }
        }
    }catch (IOException e){
        e.printStackTrace();
    }
  }
}
```

使用浏览器和 postman 进行测试，测试 get/post 两种请求方法：

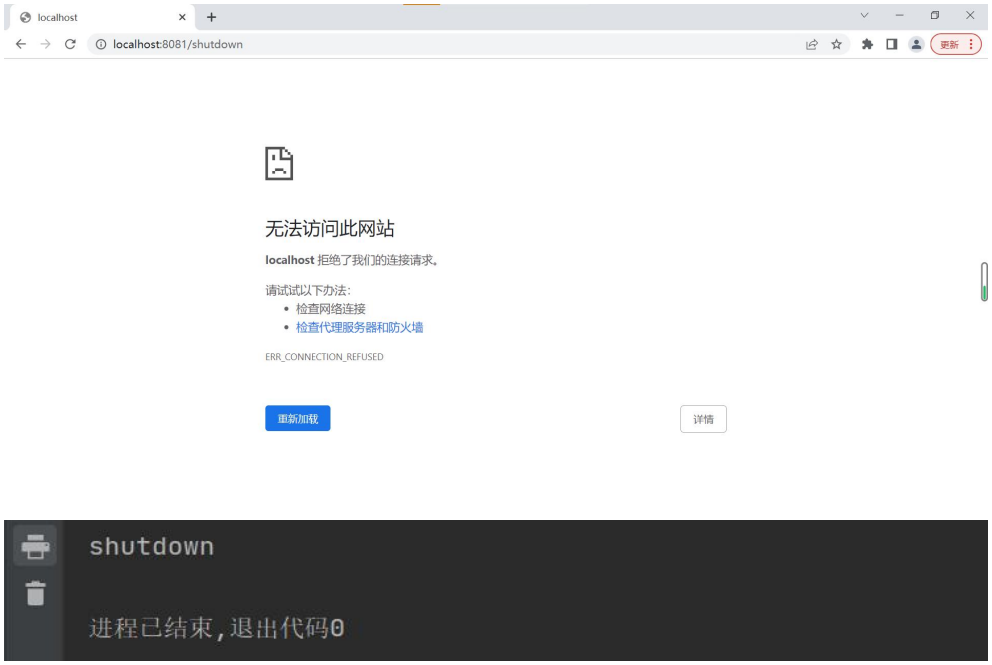
访问 localhost:8081/index.html：



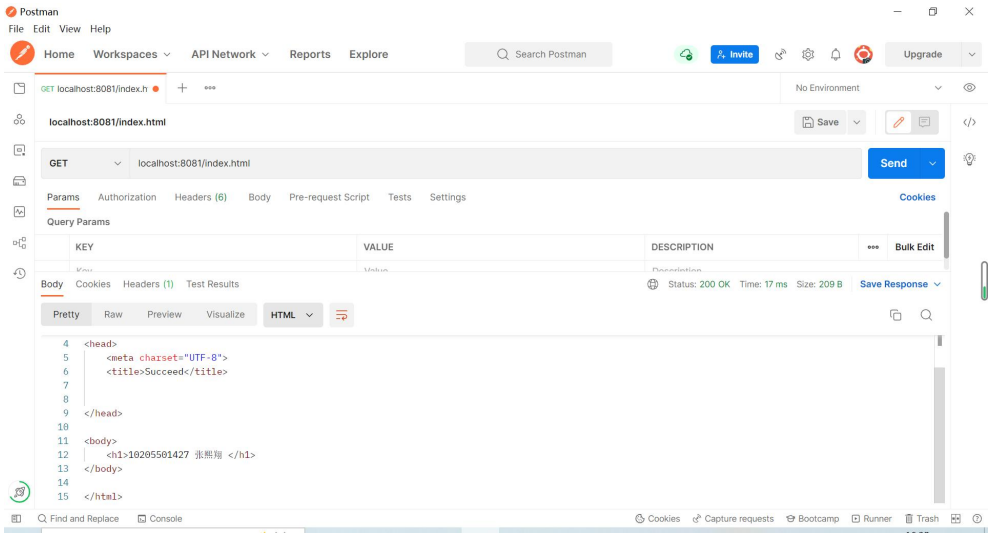
访问无效路径：



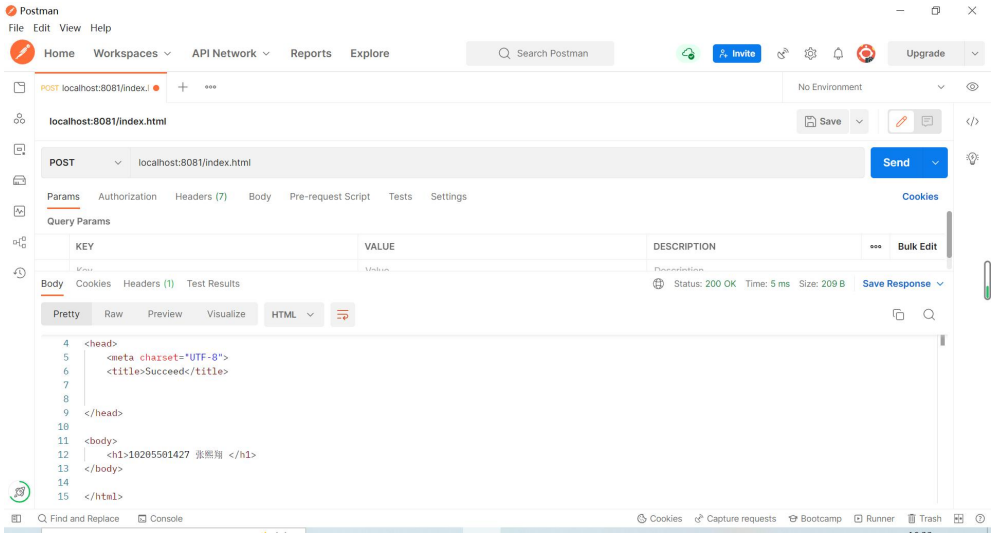
输入 localhost:8081/shutdown 关闭服务器：



GET 请求：



POST 请求：



Jmeter 压测（Ramp-Up time：20 循环次数：1）：

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	5000	1	2	2	3	3	1	21	0.00%	250.0/...	425.60	30.77
总体	5000	1	2	2	3	3	1	21	0.00%	250.0/...	425.60	30.77

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	10000	2	2	3	4	17	0	67	0.00%	499.6/...	850.44	61.48
总体	10000	2	2	3	4	17	0	67	0.00%	499.6/...	850.44	61.48

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	15000	4	2	6	16	59	0	128	0.01%	999.7/...	1701.75	123.01
总体	15000	4	2	6	16	59	0	128	0.01%	999.7/...	1701.75	123.01

Version II：（使用线程池，优化 Version I）

线程池与多线程的区别：

1、线程池是在程序运行开始，创建好的 n 个线程，并且这 n 个线程挂起等待任务的到来。而多线程是在任务到来得时候进行创建，然后执行任务。

2、线程池中的线程执行完之后不会回收线程，会继续将线程放在等待队列中；多线程程序在每次任务完成之后会回收该线程。

3、由于线程池中线程是创建好的，所以在效率上相对于多线程会高很多。

4、线程池也在高并发的情况下有着较好的性能；不容易挂掉。多线程在创建线程数较多的情况下，很容易挂掉。

代码如下：

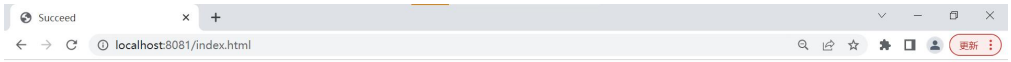
```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
//线程池
```

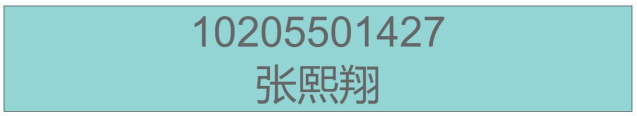
```
public class Server {
    public static void main(String[] args) {
        //构造线程池
        ThreadPoolExecutor executor = new ThreadPoolExecutor(8, 50, 400,
TimeUnit.MILLISECONDS,
                new ArrayBlockingQueue<Runnable>(2000));
        try{
            //使用 serversocket 创建服务对象
            ServerSocket server = new ServerSocket(8081);
            while(true){
                try {
                    //使用服务器对象的输入流获取客户端发来的数据
                    Socket socket = server.accept();
                    executor.execute(new ThreadPoolTask( socket ));
                }catch (Exception e){
                    e.printStackTrace();
                }
            }
        }catch (IOException e){
            e.printStackTrace();
        }
    }
}

class ThreadPoolTask implements Runnable{
    private Socket socket;

    ThreadPoolTask(Socket socket1){
        this.socket=socket1;
    }

    public void run() {
        try {
            InputStream is = null;
            try {
                is = socket.getInputStream();
            } catch (IOException e) {
                e.printStackTrace();
            }
            //将字节流转换为字符流
            InputStreamReader isr = new InputStreamReader(is);
            //将字符流包装成缓冲流
            BufferedReader br = new BufferedReader(isr);
            String line = null;
            try {
                line = br.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.out.println(line);
            //从 line 中解析文件名
            String[] linesArr = line.split(" ");
            String fileName = linesArr[1].substring(1);
            System.out.println(fileName);
            if (fileName.equals("shutdown")) {
                System.exit(0);
            }
            File file = new File(fileName);
            if (file.exists() == false) {
                fileName = "error.html";
            }
            //从磁盘读取文件
            FileInputStream fis = new FileInputStream(fileName);
            BufferedInputStream bis = new BufferedInputStream(fis);
            byte[] data = new byte[1024];
            int length = 0;
            BufferedOutputStream bos = new BufferedOutputStream(socket.getOutputStream());
```

```
            //先返回响应头信息
            bos.write("HTTP/1.1 200 OK \r\n".getBytes());
            bos.write("content-type: text/html; charset=utf-8 \r\n".getBytes());
            //响应头和数据之间有一个空行
            bos.write("\r\n".getBytes());
            //再返回数据
            while ((length = bis.read(data)) != -1) {
                //将读取的文件输出到网络
                bos.write(data, 0, length);
            }
            bos.close();
            bis.close();
            br.close();
            socket.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

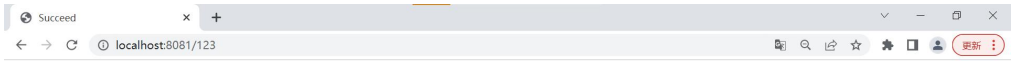
使用浏览器和 postman 进行测试，测试 get/post 两种请求方法：

访问 localhost:8081/index.html：

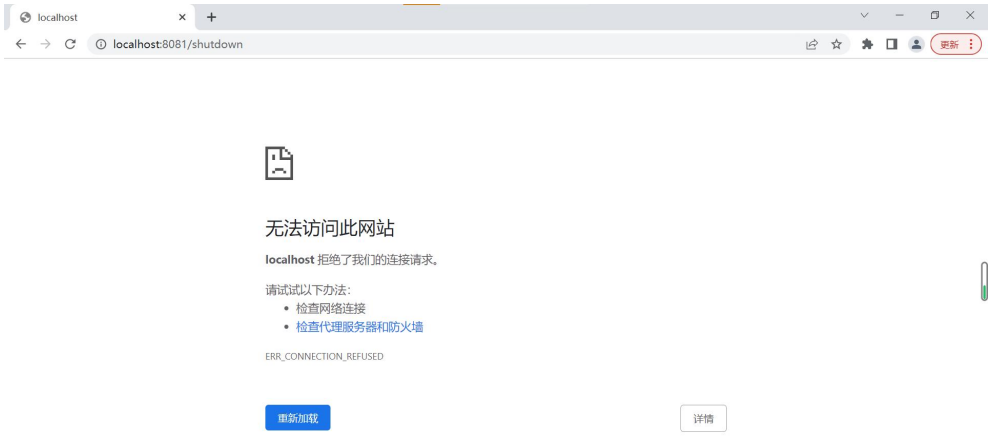




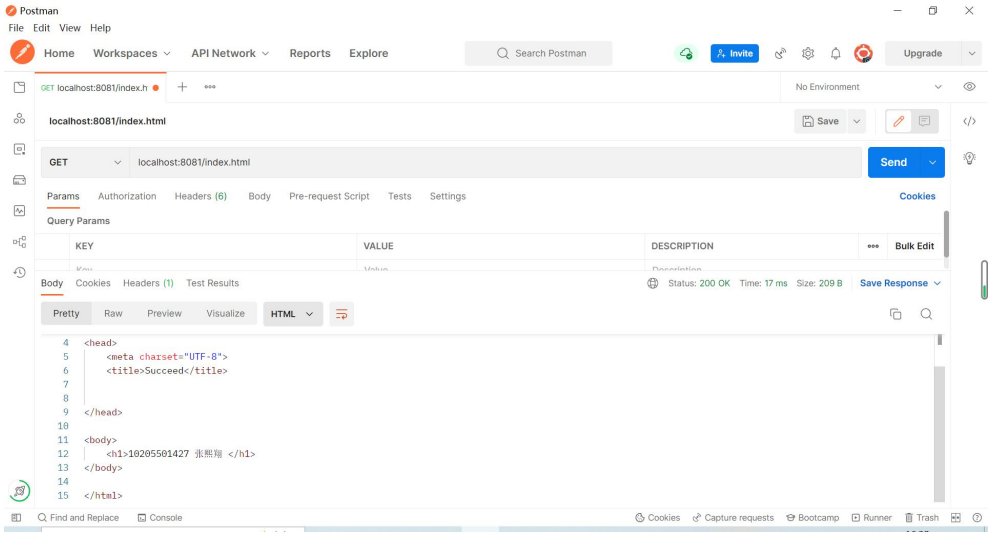
访问无效路径：



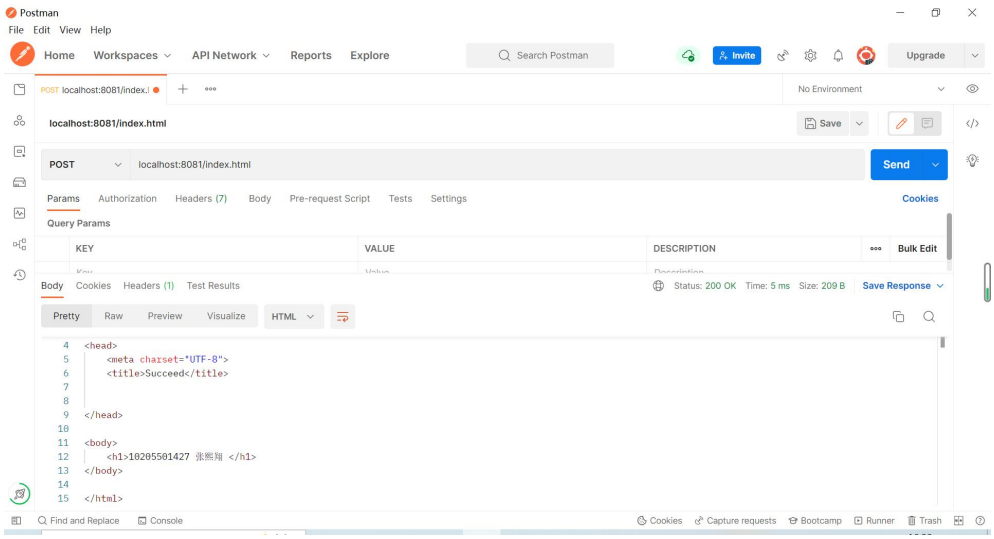
输入 localhost:8081/shutdown 关闭服务器：



GET 请求：



POST 请求：



Jmeter 压测（Ramp-Up time：20 循环次数：1）：

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	5000	1	2	2	3	3	1	18	0.00%	250.0/...	425.60	30.77
总体	5000	1	2	2	3	3	1	18	0.00%	250.0/...	425.60	30.77

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	10000	1	2	2	3	4	0	27	0.00%	500.0/...	851.12	61.53
总体	10000	1	2	2	3	4	0	27	0.00%	500.0/...	851.12	61.53

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	15000	9	2	33	58	99	0	232	0.00%	999.8/...	1701.81	123.02
总体	15000	9	2	33	58	99	0	232	0.00%	999.8/...	1701.81	123.02

Label ↑	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	20000	8	2	6	40	156	0	446	0.22%	1000....	210.41	122.83
总体	20000	8	2	6	40	156	0	446	0.22%	1000....	210.41	122.83

Version III：实现题目二（线程池）

本题是代理服务器的实现。对于客户端浏览器，代理服务器是目标 Web 服务器。就 Web 服务器而言，会把代理当成客户端，完全不知道真实客户端的存在。

首先区分 HTTP 和 HTTPS 代理流程。

HTTP 代理工作流程：

1、客户端浏览器原封不动地向代理服务器发送请求。

2、代理服务器从 Http Header 获取目标主机地址，并将请求发送到目标主机。

3、目标主机将响应发送回代理服务器。

4、代理服务器将响应发送回客户端浏览器。

HTTPS 是 HTTP 协议的安全版本，在 HTTP 上建立 SSL 加密层，对传输的数据进行加密。

HTTPS 代理工作流程：

1、浏览器首先向代理服务器发送 HTTP Connect 请求，并发送目标主机信息。

2、代理服务器与目标主机建立 TCP 链接，并以“连接已建立”回复响应浏览器。

3、浏览器将请求发送到代理服务器，代理服务器将其传输到目标主机。

4、目标主机将响应代理服务器，代理服务器将响应浏览器。

实现 http 代理服务器，其实就是代理服务器去解析 http 请求头消息找到目标服务器，然后建立代理服务器到目标服务器的 socket 连接，将 http 的全部消息全部转发给目标服务器即可。而针对 https 的代理，主要是实现牵手过程即可，这个牵手过程主要是利用 http 请求获取目标服务器地址来建立 socket 连接，并在建立连接的同时告诉客户端连接已经建立好（返回 HTTP/1.1 200 Connection Established\r\n\r\n），接下来客户端就会自动通过代理服务器建立的与目标服务器的连接发消息给目标服务器进行交换秘钥，然后，也会利用此连接传递消息，当然，传递消息时代理服务器只需要转发即可。

代码如下：

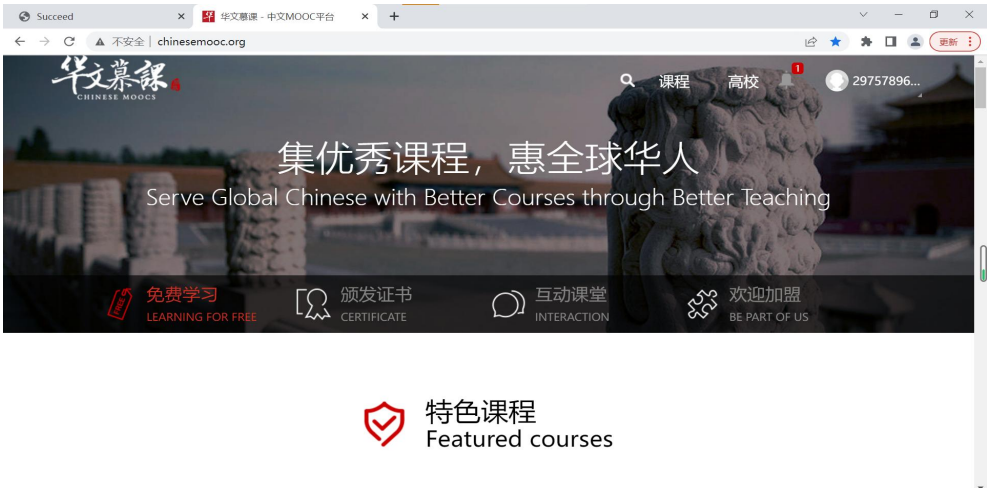
```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketTimeoutException;
//import java.nio.charset.StandardCharsets;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class test {
    public static void proxyHandler(InputStream input, OutputStream output) {
        //单纯消息转发
        try{
            while(true){
                //FileInputStream fis = new FileInputStream(fileName);
                BufferedInputStream bis=new BufferedInputStream(input);
                byte[]buffer=new byte[1024];
                int length;
                while((length=bis.read(buffer))!=-1){
                    output.write(buffer,0,length);
                    length=-1;
                }
                output.flush();
            }
        }catch (SocketTimeoutException e){
            try{
                input.close();
                output.close();
            }catch(IOException e2){
                e2.printStackTrace();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            try{
                input.close();
                output.close();
            }catch (IOException e){
                e.printStackTrace();
            }
        }
    }


    public static void handleRequest(Socket socket) {
        try {
            //socket.setSoTimeout(60*1000);//设置代理服务器与客户端的连接未活动超时时间毫秒
            String line = "";
            InputStream clientInput = socket.getInputStream();
            String tempHost="",host;
            int port =80;//默认
            String type=null;//请求方法
            OutputStream os = socket.getOutputStream();
            BufferedReader br = new BufferedReader(new InputStreamReader(clientInput));
            int flag=1;//读取浏览器的第一行
            StringBuilder sb =new StringBuilder();
            while((line = br.readLine())!=null) {
                if(flag==1) {       //获取请求行中请求方法，默认是 http
                    type = line.split(" ")[0];
                    //如果是 https，type 是 CONNECT
                    if(type==null)continue;
                }
                flag++;
                String[] s1 = line.split(": ");
                if(line.isEmpty()) {
                    break;
                }
```

```
            for(int i=0;i<s1.length;i++) {
                if(s1[i].equalsIgnoreCase("host")) {
                    //获取 host
                    tempHost=s1[i+1];
                }
            }
            sb.append(line).append("\r\n");
            //System.out.println(line);
            line=null;
        }

        sb.append("\r\n");

        if(tempHost.split(":").length>1) {
            port = Integer.parseInt(tempHost.split(":")[1]);
            //获取端口
        }

        host = tempHost.split(":")[0];
        //System.out.println(host);
        Socket proxySocket;//代理间通信的 socket

        if(host!=null&&!host.equals("")) {
            //打开一个通向目标服务器的 Socket 连接到目标服务器
            proxySocket = new Socket(host,port);
            //proxySocket.setSoTimeout(1000*60);//设置代理服务器与服务器端的连接未活动超时时间
            OutputStream proxyOs = proxySocket.getOutputStream();//代理的输出
            InputStream proxyIs = proxySocket.getInputStream();//代理的输入

            assert type != null;

            if(type.equalsIgnoreCase("connect")) {//是否为 https 请求的话
                os.write("HTTP/1.1 200 Connection Established\r\n\r\n".getBytes());
                os.flush();
            }else {//http 请求则直接转发
                proxyOs.write(sb.toString().getBytes());
                proxyOs.flush();
            }

            //新开线程转发客户端请求至目标服务器
            ExecutorService Proxyexecutor=Executors. newCachedThreadPool();
            Proxyexecutor.submit(new Thread(()->proxyHandler(clientInput,proxyOs)));
            //转发目标服务器响应至客户端
            Proxyexecutor.submit(new Thread(()->proxyHandler(proxyIs,os)));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}


public static void main(String[] args) throws IOException {
    //等待来自客户的请求 启动新线程以处理客户连接请求
    ExecutorService Socketexecutor = Executors. newCachedThreadPool();//线程池
    ServerSocket ss = new ServerSocket(8080);//监听代理代理服务器端口
    while(!Thread.currentThread().isInterrupted()){
        Socket socket=ss.accept();
        Socketexecutor.submit(new Thread(()->handleRequest(socket)));//socket 线程池
    }
}
```

手动设置代理：

代理服务器未开启，则无法访问：

开启代理服务器后，正常访问网页：



Jmeter 压测（Ramp-Up time：20 循环次数：1）：

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	250	2010	688	4584	7806	14312	241	15883	0.00%	8.1/sec	567.84	1.40
总体	250	2010	688	4584	7806	14312	241	15883	0.00%	8.1/sec	567.84	1.40

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	280	3519	914	8395	16243	30540	3	45860	1.79%	5.2/sec	356.25	0.87
总体	280	3519	914	8395	16243	30540	3	45860	1.79%	5.2/sec	356.25	0.87

Version IV：使用 Netty 尝试实现 nio

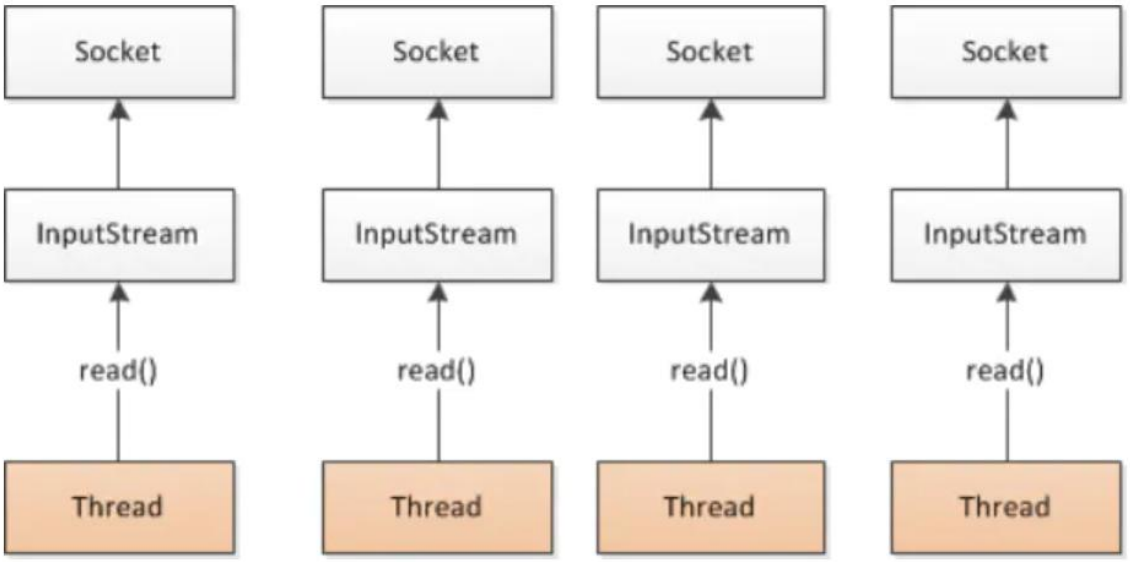
什么是 Netty？来自 netty 官网 https://netty.io/的介绍：

> Netty is *an asynchronous event-driven network application framework*
> for rapid development of maintainable high performance protocol servers & clients.

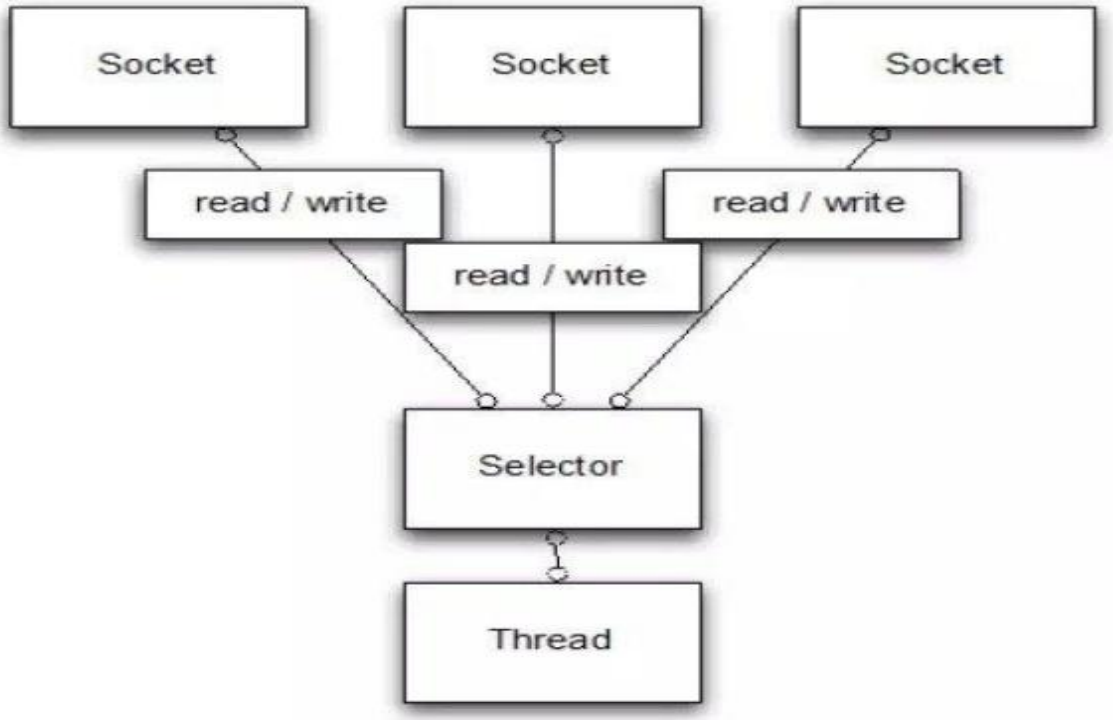
Netty 是一个异步事件驱动的网络应用框架，用于快速开发可维护的高性能协议服务器和客户端。

Netty 底层使用了 JAVA 的 NIO 技术，并在其基础上进行了性能的优化，虽然 Netty 不是单纯的 JAVA nio，但是 Netty 的底层还是基于的是 nio 技术。

BIO 模型：



NIO 模型：



BIO 和 NIO 区别在于：

1、BIO 以流的方式处理数据，NIO 以块的方式处理数据，块 IO 的效率比流 IO 高很多。

2、BIO 是阻塞的，NIO 是非阻塞的。

3、BIO 基于字节流和字符流进行操作的，而 NIO 基于 Channel（通道）和 Buffer（缓冲区）进行操作的，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector 用于监听多个通道事件，因此使用单个线程就可以监听多个客户端通道。

代码如下：

HttpServer：

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.logging.LogLevel;
import io.netty.handler.logging.LoggingHandler;

public class HttpServer {
    private final static int port=8082;
    public static void main(String[]args) throws InterruptedException{
        //BossEventLoop 负责接收客户端的连接
        EventLoopGroup bossGroup=new NioEventLoopGroup();
        //将 Socket 交给 WorkerEventLoopGroup 进行 IO 处理
        EventLoopGroup workerGroup=new NioEventLoopGroup();
        try{
            //ServerBootstrap 是服务器端启动助手
            ServerBootstrap serverBootstrap=new ServerBootstrap();

            serverBootstrap.group(bossGroup,workerGroup)
            //使用 NioServerSocketChannel 作为服务器短的通道
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG,1000)
                    .option(ChannelOption.TCP_NODELAY,true)
                    .handler(new LoggingHandler(LogLevel.INFO))
                    .childHandler(new HttpServerInitializer());
            //通道处理器添加完毕后启动服务器
            ChannelFuture channelFuture=serverBootstrap.bind(port).sync();//异步
            channelFuture.channel().closeFuture().sync();//异步

        }finally {
            ///释放资源
            workerGroup.shutdownGracefully();
            bossGroup.shutdownGracefully();
        }
    }
}
```

HttpServerInitializer：

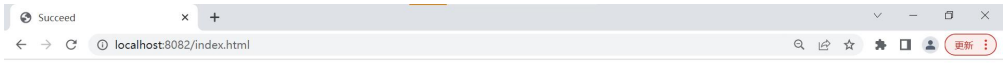
```
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.socket.SocketChannel;
import io.netty.handler.codec.http.HttpObjectAggregator;
import io.netty.handler.codec.http.HttpServerCodec;
import io.netty.handler.stream.ChunkedWriteHandler;

public class HttpServerInitializer extends ChannelInitializer<SocketChannel> {
    @Override
```

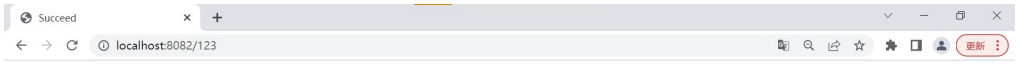
```
    protected void initChannel(SocketChannel socketChannel) {
        ChannelPipeline channelPipeline=socketChannel.pipeline();
        //将请求和应答消息编码或解码为 HTTP 消息
        channelPipeline.addLast(new HttpServerCodec());
        channelPipeline.addLast(new HttpObjectAggregator(65536));//64*1024
        channelPipeline.addLast(new ChunkedWriteHandler());//ChunkedWriteHandler 进行大规模文
件传输。
        channelPipeline.addLast(new HttpServerHandleAdapter());//FileSystem 业务工程处理器。
    }
}
```

HttpServerHandleAdapter：

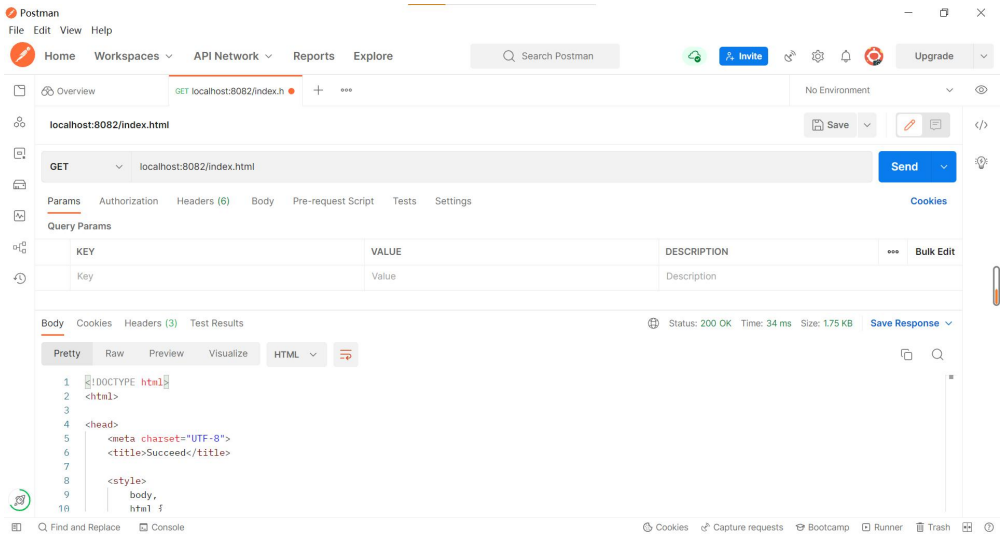
```
import io.netty.channel.*;
import io.netty.handler.codec.http.*;
import java.io.File;
import java.io.RandomAccessFile;

public class HttpServerHandleAdapter extends SimpleChannelInboundHandler<FullHttpRequest>
{
    @Override
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
FullHttpRequest fullHttpRequest) throws Exception {
        //获取 URI
        String uri=fullHttpRequest.uri();
        if(uri.equalsIgnoreCase("/") || uri.equalsIgnoreCase("/index.html")){
            uri="/index.html";//null
        }else if(uri.equalsIgnoreCase("/shutdown")){
            System.exit(0);
        }else{
            uri="/error.html";
        }
        String fileName=uri.substring(1);//文件地址
        //根据地址构建文件
        File file=new File(fileName);
        //创建 http 响应
        HttpResponse httpResponse=new DefaultHttpResponse(fullHttpRequest.protocolVersion(),
HttpResponseStatus.OK);

        //设置文件格式内容
        if(fileName.endsWith(".html")){
            httpResponse.headers().set(HttpHeaderNames.CONTENT_TYPE, "text/html;
charset=UTF-8");
        }
        if(file.exists()){
           httpResponse.setStatus(HttpResponseStatus.OK);
        }
        else{
            httpResponse.setStatus(HttpResponseStatus.NOT_FOUND);
        }
        RandomAccessFile randomAccessFile=new RandomAccessFile(file,"r");

        httpResponse.headers().set(HttpHeaderNames.CONTENT_LENGTH,
randomAccessFile.length());
        httpResponse.headers().set(HttpHeaderNames.CONNECTION, HttpHeaderValues.KEEP_ALIVE);

        channelHandlerContext.write(httpResponse);//写回 http 报文
        channelHandlerContext.write(new DefaultFileRegion(randomAccessFile.getChannel(), 0,
file.length()));//写回文件

        channelHandlerContext.writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT);
        randomAccessFile.close();//关闭文件
    }
}
```

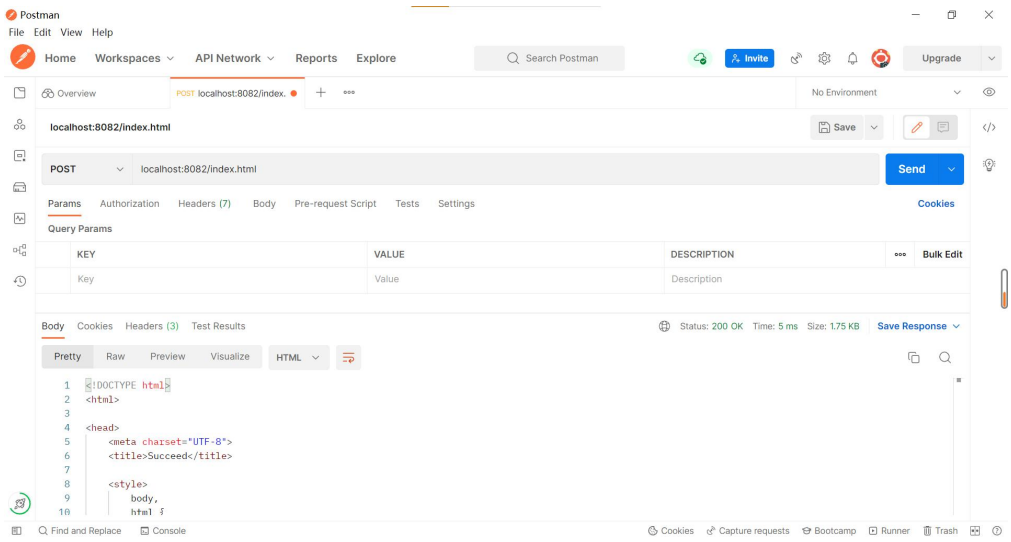
对 localhost:8082/index.html 以及无效路径测试：





Postman 测试 GET 和 POST 请求：





输入 localhost:8082/shutdown 关闭服务器：

Jmeter 压测（Ramp-Up time：20 循环次数：1）：

Label	# 样本	平均值	中位数	90% ...	95% ...	99% ...	最小值	最大值	异常 %	吞吐量	接收 K...	发送 K...
HTTP...	30000	2	1	3	8	36	0	192	0.02%	999.6/...	1701.72	122.97
总体	30000	2	1	3	8	36	0	192	0.02%	999.6/...	1701.72	122.97

五、总结

本次项目让我感受到了书本理论知识和实际应用之间的距离，对于框架的了解和对于并发的提高还有许多知识需要掌握。当然这也是一次收货颇丰的项目，http 协议，socket 编程的复习，对 BIO 和 NIO 之间区别联系的了解，以及对于 Netty 框架的尝试，在不断地优化代码，不断地改 bug 的过程中收获颇丰。