

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: 2018

上机实践成绩:

指导教师: 张召

姓名: 池欣宁

学号: 10185501409

上机实践名称: 设计开发 web 服务器与代理服务器

上机实践日期:

上机实践编号:

组号:

上机实践时间:

一、实验目的

题目 1.1:用 Java 开发一个简单的 Web 服务器, 仅能处理一个请求。

题目 1.2:用 Java 开发一个简单的 Web 代理服务器

二、实验任务

1. 开发代理服务器

2. 开发服务器

3. 进行串行并行测试

4. 服务器版本优化:加入线程池、NIO

三、使用环境

IntelliJ IDEA 2019.3.1 (Ultimate Edition)

macOS 10.15.5

Mavens-Netty:4.1.42.Final

四、实验过程

1.Web 服务器

V1.0 无线程池

Web 服务器的核心逻辑在于, Main 函数一直处于循环监听的状态, 当有一个客户端发起链接时, 便接受 socket 进行通信。

server 端需要获取所请求资源的具体文件名, 由 getResourceName 函数获得, 再通过 socket 返回给客户端。

参照 Http 报文格式对请求进行解析。

Http 报文格式:

请求方法	空格	URL	空格	协议版本	回车符	换行符	请求行
头部字段名	:	值	回车符	换行符	} 请求头部		
...							
头部字段名	:	值	回车符	换行符			
回车符	换行符						
https://blog.csdn.net/zx_emily							请求数据zx_emily

主方法:

1. server 监听 port 端口, 等待 client 端发起连接。
2. 建立 socket 连接。

```
public static void main(String [] args)throws IOException{
    /* ExecutorService executor =Executors.newFixedThreadPool(100);//线程池*/
    ServerSocket ss=new ServerSocket(8082);
    System.out.println("服务器已经监听:");
    System.out.println(ss.getInetAddress()+":"+ss.getLocalPort());
    /*Thread Pool 线程池*/
    while(!Thread.currentThread().isInterrupted()){
        System.out.println("Server 阻塞等待中");
        Socket socket=ss.accept();//创建一个连接套接字
        //为新的连接创建新的线程

        new Thread(() ->handleRequest(socket)).start();
        //executor.submit(new Thread(()->handleRequest(socket)));
    }
}
```

使用线程 `handleRequest` 作为 web 服务器处理的主要方法

```
public static void handleRequest(Socket socket){
    try{
        //使用 socket 对象中的 getInputStream。
        InputStream is=socket.getInputStream();
        //输入流对象 is 转换为字符缓冲输入流
        BufferedReader br=new BufferedReader(new InputStreamReader(is));
        String line=br.readLine();
        /*BufferedReader test */
        // System.out.println(line);
        /*parse URL*/
        String fileName =getResourceName(line);
        //如果出现 shutdown 的情况
        if(fileName.equals("/shutdown")){
            System.exit(0);
        }
        //如果出现索引省略的情况下:
        if(fileName.equals("/")){
            fileName="/index.html";//默认也转入 index.html 首页
        }
        String ROOT="/Users/chixinning/Desktop/webServer"+"Network/web";
```

```

        File file=new File(ROOT+arr[1]);
        /*如果请求文件资源正确存在*/
        if(file.exists()){
            /*先写 httpResponse 头*/
            FileInputStream fis=new FileInputStream(ROOT);
            OutputStream os=socket.getOutputStream();
            os.write("HTTP/1.1 200 OK\r\n".getBytes());
            os.write("Content-Type:text/html\r\n".getBytes());
            os.write("\r\n".getBytes()); //必须写入空行，否则浏览器不解析
            int len=0;
            byte[] bytes=new byte[1024];
            while((len=fis.read(bytes))!=-1){
                os.write(bytes,0,len);
            }
            fis.close();
            socket.close();
        }
        /*如果请求文件资源不存在，返回 404NOT FOUND 信息*/
        else{
            /*既可以写成一段话，也可以 error.html 类比 index.html 进行输出，但这里 String ErrorMessage 更符合业务逻辑*/
            String errorMessage="HTTP/1.1 404 File Not Found\r\n"+
                "Content-Type:text/html\r\n"+
                "\r\n"+
                "<h1>File Not Found</h1>";
            OutputStream os=socket.getOutputStream();
            os.write(errorMessage.getBytes());
        }
    }catch(IOException e){
        e.printStackTrace();
    }
}

```

getResourceName 方法获取文件资源路径

辅助方法类：获得文件名。

```

public static String getResourceName(String s){
    //"GET /index.html HTTP/1.1"
    String[] split=s.split(" ");
    if(split.length!=3){

```

```

        System.out.println("HTTP 方法不符合格式! 出现错误! 关闭服务器! ");
        System.exit(1);
    }
    System.out.println(s);
    return split[1];
}

```

V2.0 线程池版

单个线程的执行过程为创建、运行和销毁，所需要的时间为 t_1 、 t_2 、 t_3

线程运行的总时间 $T = t_1 + t_2 + t_3$;

假如，有些业务逻辑需要频繁的使用线程执行某些简单的任务，那么很多时间都会浪费 t_1 和 t_3 上。

为了避免这种问题，在线程池中的线程可以实现线程的复用，当线程运行完任务之后，不被销毁。

添加线程池的 webServer 代码:

```

public static void main(String [] args) throws IOException{
    ExecutorService executor = Executors.newFixedThreadPool(100); //线程池
    ServerSocket ss = new ServerSocket(8082);
    System.out.println("服务器已经监听:");
    System.out.println(ss.getInetAddress() + ":" + ss.getLocalPort());
    /*Thread Pool 线程池*/
    while(!Thread.currentThread().isInterrupted()){
        System.out.println("Server 阻塞等待中");
        Socket socket = ss.accept(); //创建一个连接套接字
        //为新的连接创建新的线程

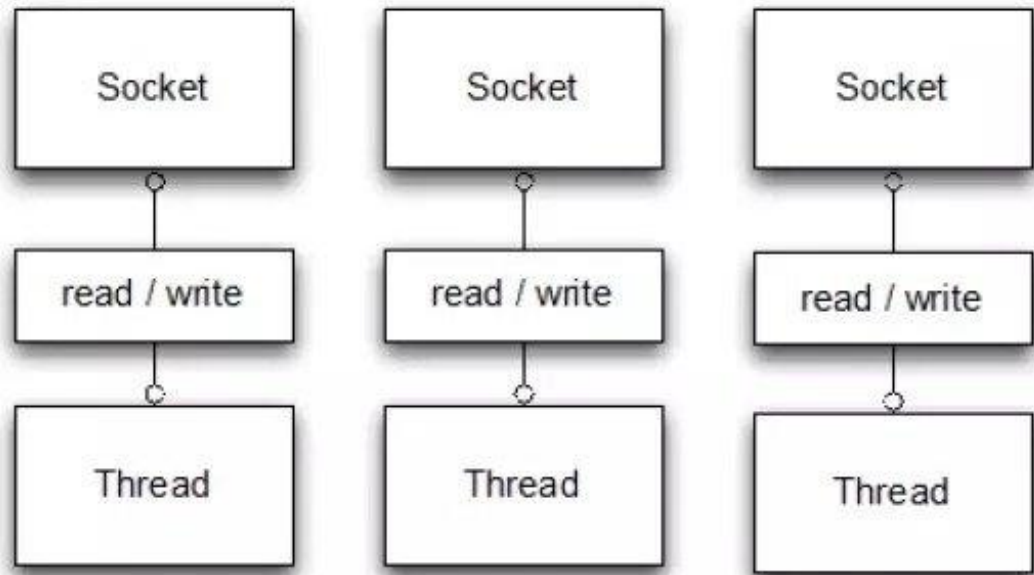
        /* new Thread() -> handleRequest(socket).start(); */
        executor.submit(new Thread(() -> handleRequest(socket)));
    }
}

```

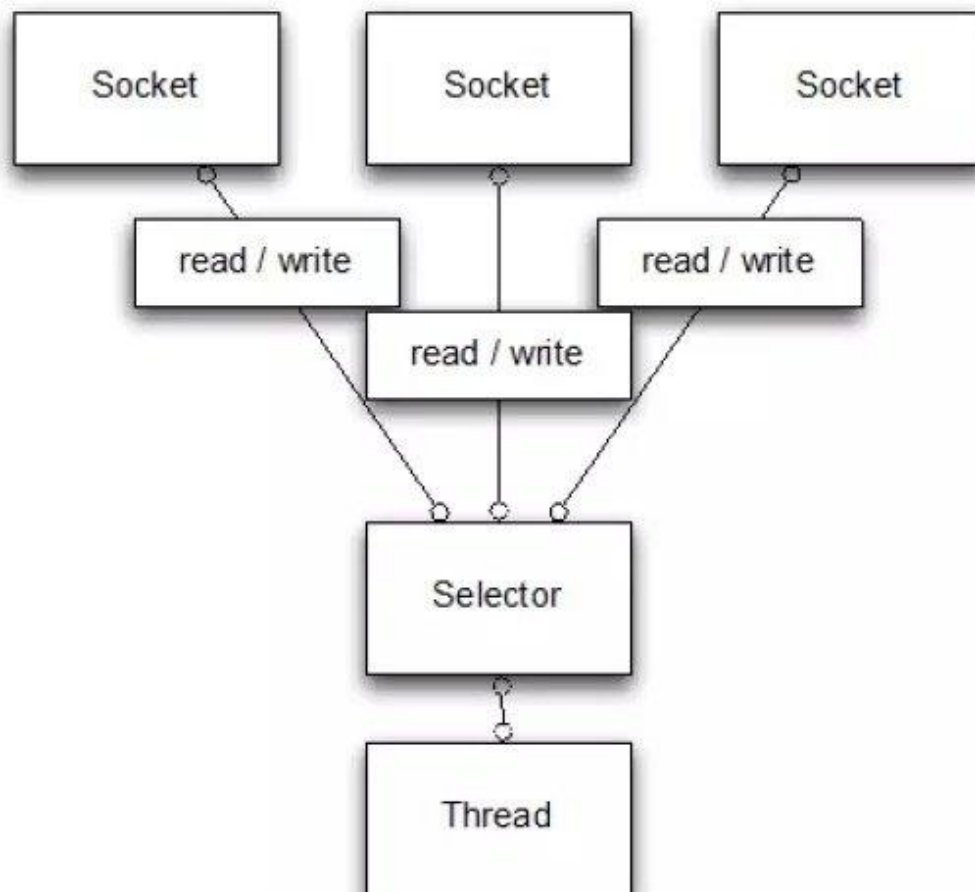
V3.0 Netty 版 (NIO)

为什么 NIO 可以提高高并发

传统 BIO 阻塞模型: (V1.0 与 V2.0 所采用的模型)



NIO 模型:



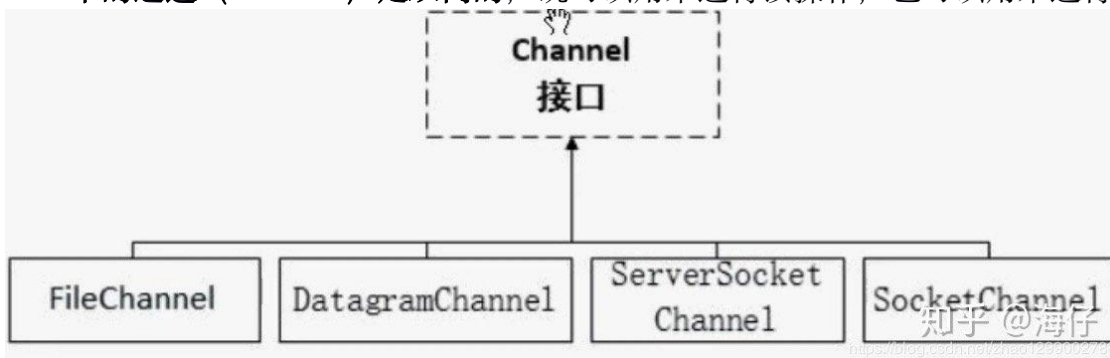
BIO 等待客户端发数据这个过程是阻塞的，这样就造成了一个线程只能处理一个请求的情况，而 CPU 性能的限制导致机器能支持的最大线程数是有限的。

NIO 的核心部分

- Channel
- Buffer
- Selector: 用于监听多个通道的事件，即使用单个线程可以监听多个数据管道



- BIO 中的 stream 是单向的，例如 `FileInputStream` 对象只能进行读取数据的操作，而 NIO 中的通道 (Channel) 是双向的，既可以用来进行读操作，也可以用来进行写操作。



NIO 使用一个线程处理大量的客户端连接。

NIO 核心 API

EventLoopGroup 和其实现类 NioEventLoopGroup

常用方法:

构造方法: `NioEventLoopGroup()`

ServerBootstrap 和 Bootstrap

`ServerBootstrap` 是 Netty 中的服务端启动助手

`Bootstrap` 是 Netty 中的客户端启动助手

利用这两个 API 完成各种配置。

Selector

检测多个注册的 Channel 上是否有时间发生，然后针对每个事件进行相应的响应处理。

这样使得只用在连接真正有读写时，才会调用函数来进行读写，大大地减少了系统开销。

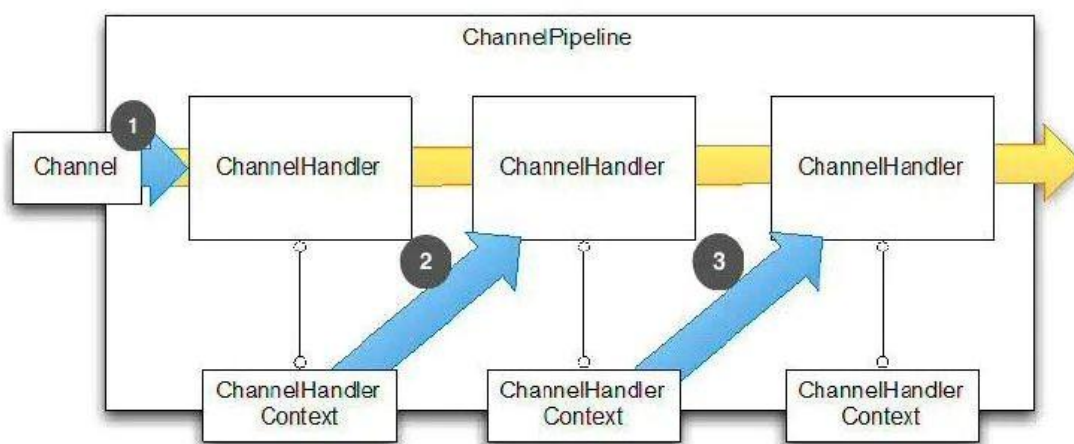
NIO 与 Netty?

Netty 是基于 NIO 的网络编程框架，使用 Netty 相当于简化和流程化了 NIO 的开发过程。

NIO 中，当一个 Socket 建立好之后，Thread 并不会阻塞去接受这个 Socket，而是将这个请求交给 Selector，Selector 会不断的去遍历所有的 Socket，一旦有一个 Socket 建立完成，他会通知 Thread，然后 Thread 处理完数据再返回给客户端——这个过程是不阻塞的，这样就能让一个 Thread 处理更多的请求了。

BIO，同步阻塞 IO，阻塞整个步骤，一个线程只处理一个连接，适用于少连接且延迟低的场景，如数据库连接。

NIO，同步非阻塞 IO，阻塞业务处理但不阻塞数据接收，适用于高并发且处理简单的场景，如聊天软件。



Channel，表示一个连接，可以理解为每一个请求，就是一个 Channel。

ChannelHandler，核心处理业务就在这里，用于处理业务请求。（在 pipeline 上实现业务功能的实现与处理）

ChannelHandlerContext，用于传输业务数据。

ChannelPipeline，用于保存处理过程需要用到的 ChannelHandler 和 ChannelHandlerContext。

这里使用 randomAccessFile 进行大综文件拷贝。

类比 httpServer Netty 的官方 demo 版

```

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 作为服务器端的通道
    .channel(NioServerSocketChannel.class)
//128 是最大线程数
    .option(ChannelOption.SO_BACKLOG,128)
    .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 {
    ///资源优雅释放
    workerGroup.shutdownGracefully();
    bossGroup.shutdownGracefully();
}
}
}

public class HttpServerInitializer extends ChannelInitializer<SocketChannel> {
    @Override
    protected void initChannel(SocketChannel socketChannel) throws Exception {
        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 业务功能实现。
    }
}

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

/*这个主要是一个文件服务器*/

```



```
public class HttpServerHandleAdapter extends SimpleChannelInboundHandler<FullHttpRequest> {
    private static final String ROOT;
    private static final File ERROR;
    static
    {
        ROOT="/Users/chixinning/Desktop/webServer/Netty/src/main/java/staticServer";//待填
        String errorPagePath=ROOT+"/error.html";
        ERROR=new File(errorPagePath);
    }
    @Override
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
FullHttpRequest fullHttpRequest) throws Exception {
        //获取 URI
        String uri=fullHttpRequest.getUri();
        if(uri.equalsIgnoreCase("/")){
            uri="/index.html";//null
        }
        String fileName=ROOT+uri;//文件地址
        if(uri.equalsIgnoreCase("/shutdown")){
            System.out.println("系统关闭");
            System.exit(0);
        }
        //根据地址构建文件
        File file=new File(fileName);
        //创建 http 响应
        HttpResponse httpResponse=new
DefaultHttpResponse(fullHttpRequest.getProtocolVersion(), HttpResponseStatus.OK);
        //设置文件格式内容

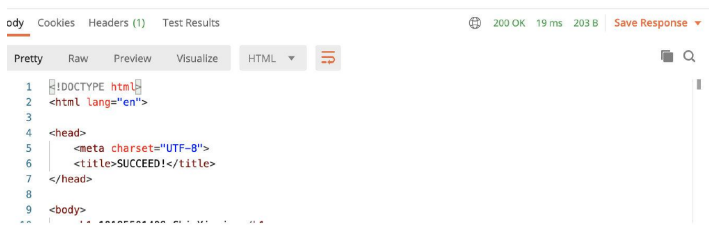
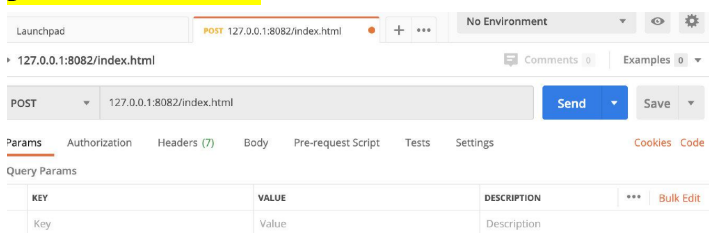
        if(fileName.endsWith(".html")){
            httpResponse.headers().set(HttpHeaders.Names.CONTENT_TYPE, "text/html; charset=UTF-8");
        }
        if(file.exists()){
            httpResponse.setStatus(HttpResponseStatus.OK);
        }
        else{
            //如果文件不存在
            file=ERROR;
            httpResponse.setStatus(HttpResponseStatus.NOT_FOUND);
        }

        System.out.println("keepAlive"+keepAlive);//Test
    }
}
```

```
//创建 randomAccess 对象
RandomAccessFile randomAccessFile=new RandomAccessFile(file,"r");
httpResponse.headers().set(HttpHeaders.Names.CONTENT_LENGTH,
randomAccessFile.length());
httpResponse.headers().set(HttpHeaders.Names.CONNECTION,
HttpHeaders.Values.KEEP_ALIVE);
channelHandlerContext.write(httpResponse);//写回 http 报文
channelHandlerContext.write(new DefaultFileRegion(randomAccessFile.getChannel(), 0,
file.length()));//写回文件
//写入文件尾部， 必须有。
channelHandlerContext.writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT);
randomAccessFile.close();//关闭文件
}
}
```

测试

postman 串行测试



v1.0 不使用线程池，仅使用线程进行接管

视图	原始数据	头
1819	全部折叠	全部展开
JSON		
id: "9d236a15-46a5-4f23-91c5-6bc31c5f4448"		
name: "并发性能测试"		
timestamp: "2020-08-03T05:51:13.868Z"		
collection_id: "96f087d9-415a-495f-b6c5-24978ed0ff16"		
folder_id: 0		
environment_id: "g"		
totalPass: 0		
totalFail: 0		
results:		
0:		
id: "e32a4fd1-8384-4594-b8ef-2fc06231a91f"		
name: "127.0.0.1:8082/index.html"		
time: 4		
responseCode:		
code: 200		
name: "OK"		
tests: {}		
testPassFailCounts: {}		
times: [...]		
allTests: [...]		
count: 1000		
TotalTime: 6106		
collection:		
requests:		
0:		
id: "e32a4fd1-8384-4594-b8ef-2fc06231a91f"		
method: "GET"		

v2.0 使用线程池

视图	原始数据	头
1819	全部折叠	全部展开
JSON		
id: "f9d384d6-3d70-4d62-bc61-57787d77a0b9"		
name: "并发性能测试"		
timestamp: "2020-08-03T05:00:21.668Z"		
collection_id: "96f087d9-415a-495f-b6c5-24978ed0ff16"		
folder_id: 0		
environment_id: "g"		
totalPass: 0		
totalFail: 0		
results:		
0:		
id: "e32a4fd1-8384-4594-b8ef-2fc06231a91f"		
name: "127.0.0.1:8082/index.html"		
time: 3		
responseCode:		
code: 200		
name: "OK"		
tests: {}		
testPassFailCounts: {}		
times: [...]		
allTests: [...]		
count: 1000		
TotalTime: 2893		
collection:		
requests:		
0:		
id: "e32a4fd1-8384-4594-b8ef-2fc06231a91f"		
method: "GET"		

v3.0 使用 netty 框架

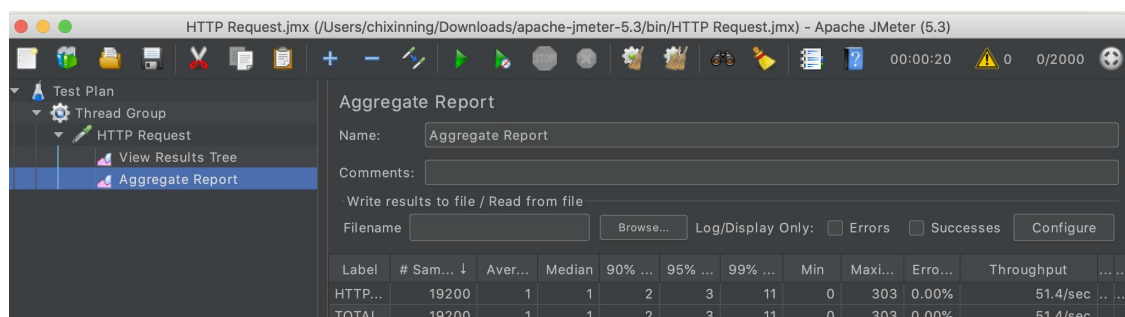
视图	原始数据	头
1819	全部折叠	全部展开
JSON		
id: "871ac2a0a-3173-4324-b0a2-4581a5327788"		
name: "并发性能测试"		
timestamp: "2020-08-03T05:45:52.228Z"		
collection_id: "96f087d9-415a-495f-b6c5-24978ed0ff16"		
folder_id: 0		
environment_id: "g"		
totalPass: 0		
totalFail: 0		
results:		
0:		
id: "e32a4fd1-8384-4594-b8ef-2fc06231a91f"		
name: "127.0.0.1:8082/index.html"		
time: 2		
responseCode:		
code: 200		
name: "OK"		
tests: {}		
testPassFailCounts: {}		
times: [...]		
allTests: [...]		
count: 100		
TotalTime: 425		
collection:		
requests:		
0:		
id: "e32a4fd1-8384-4594-b8ef-2fc06231a91f"		
method: "GET"		

可以很明显的看出使用传统 BIO 模型，对系统的负载消耗也会较大，即我的系统会出现很大的系统负载的声音/

这 3 个版本的吞吐的差异非常的明显。

JMeter 并发测试

V3.0 使用 netty 框架



Aggregate Report

Name: Aggregate Report

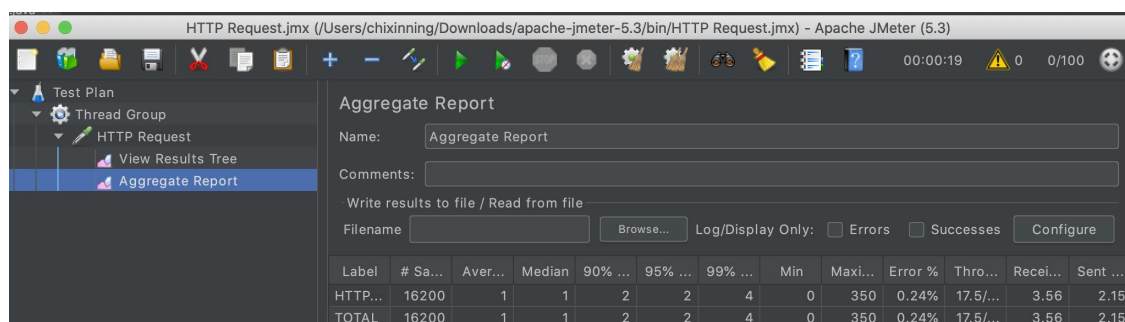
Comments:

Write results to file / Read from file

Filename: Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Label	# Sam...	Aver...	Median	90% ...	95% ...	99% ...	Min	Maxi...	Erro...	Throughput	...
HTTP...	19200	1	1	2	3	11	0	303	0.00%	51.4/sec	...
TOTAL	19200	1	1	2	3	11	0	303	0.00%	51.4/sec	...

V2.0 使用线程池



Aggregate Report

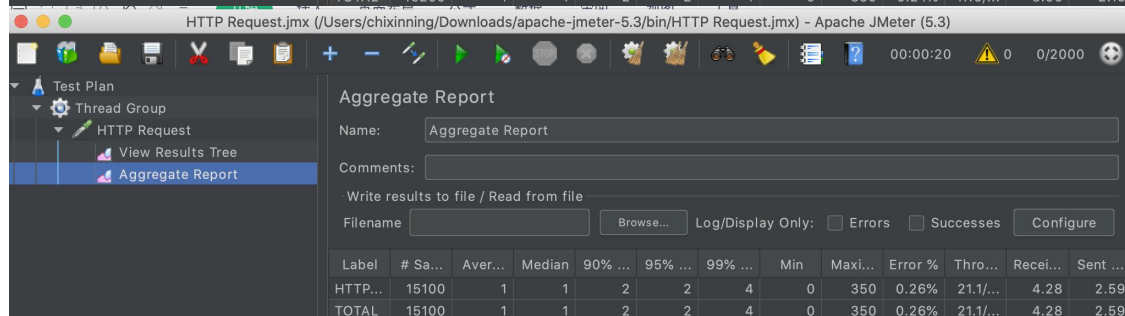
Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename: Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Label	# Sa...	Aver...	Median	90% ...	95% ...	99% ...	Min	Maxi...	Error %	Thro...	Recei...	Sent ...
HTTP...	16200	1	1	2	2	4	0	350	0.24%	17.5/...	3.56	2.15
TOTAL	16200	1	1	2	2	4	0	350	0.24%	17.5/...	3.56	2.15



Aggregate Report

Name: Aggregate Report

Comments:

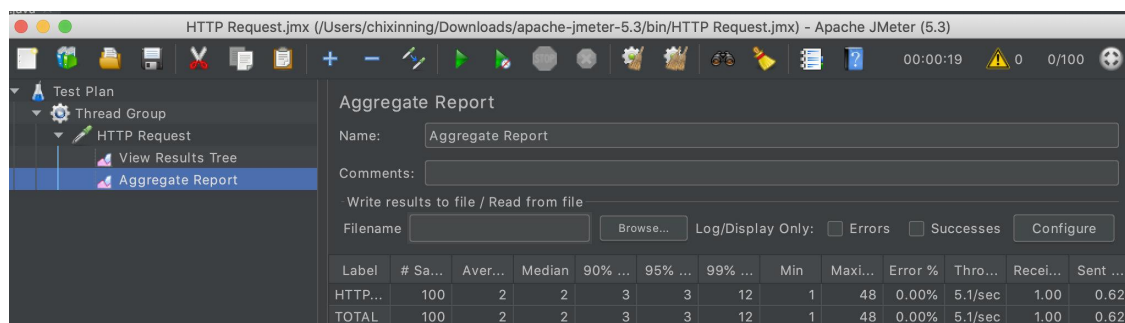
Write results to file / Read from file

Filename: Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Label	# Sa...	Aver...	Median	90% ...	95% ...	99% ...	Min	Maxi...	Error %	Thro...	Recei...	Sent ...
HTTP...	15100	1	1	2	2	4	0	350	0.26%	21.1/...	4.28	2.59
TOTAL	15100	1	1	2	2	4	0	350	0.26%	21.1/...	4.28	2.59

V1.0 传统的 BIO

ThreadUsers=100, ramp-up period:20



Aggregate Report

Name: Aggregate Report

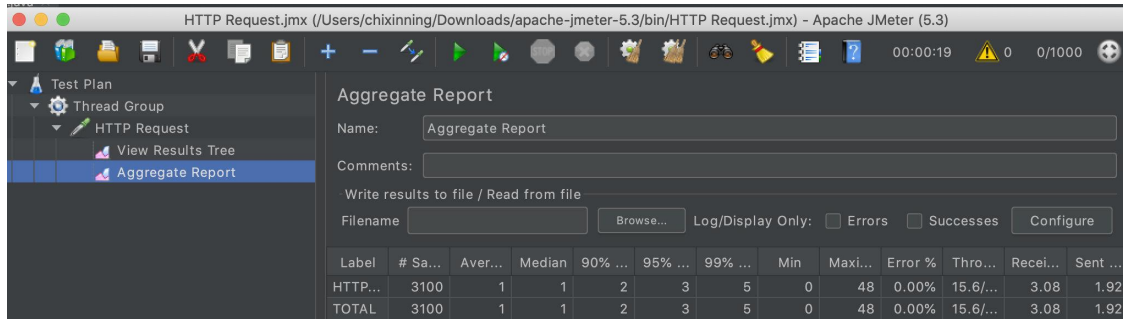
Comments:

Write results to file / Read from file

Filename: Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Label	# Sa...	Aver...	Median	90% ...	95% ...	99% ...	Min	Maxi...	Error %	Thro...	Recei...	Sent ...
HTTP...	100	2	2	3	3	12	1	48	0.00%	5.1/sec	1.00	0.62
TOTAL	100	2	2	3	3	12	1	48	0.00%	5.1/sec	1.00	0.62

ThreadUsers=3000, ramp-up period:20



Aggregate Report

Name: Aggregate Report

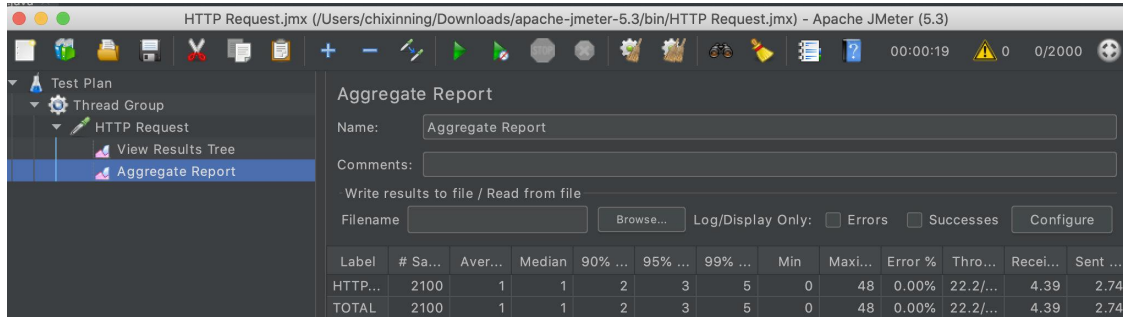
Comments:

Write results to file / Read from file

Filename: Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Label	# Sa...	Aver...	Median	90% ...	95% ...	99% ...	Min	Maxi...	Error %	Thro...	Recei...	Sent ...
HTTP...	3100	1	1	2	3	5	0	48	0.00%	15.6/...	3.08	1.92
TOTAL	3100	1	1	2	3	5	0	48	0.00%	15.6/...	3.08	1.92

ThreadUsers=2000, ramp-up period:20



Aggregate Report

Name: Aggregate Report

Comments:

Write results to file / Read from file

Filename: Browse... Log/Display Only: ☐ Errors ☐ Successes Configure

Label	# Sa...	Aver...	Median	90% ...	95% ...	99% ...	Min	Maxi...	Error %	Thro...	Recei...	Sent ...
HTTP...	2100	1	1	2	3	5	0	48	0.00%	22.2/...	4.39	2.74
TOTAL	2100	1	1	2	3	5	0	48	0.00%	22.2/...	4.39	2.74

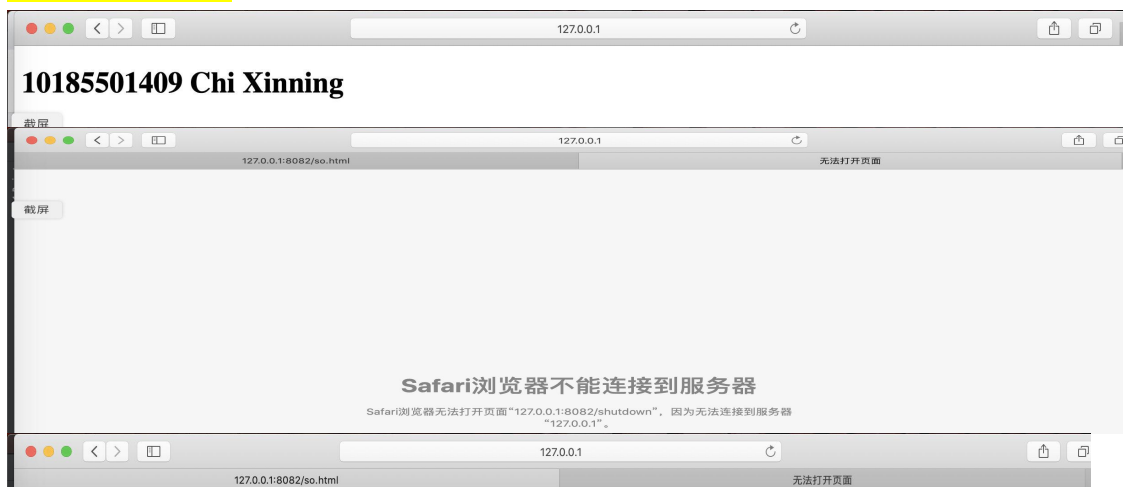
httpie 测试结果

```
[mymac:~ chixinning$ http 127.0.0.1:8082/
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>SUCCEED!</title>
</head>
<body>
<h1>10185501409 Chi Xinning</h1>

</body>
</html>
```

浏览器测试结果



File Not Found

截屏

NettyWebServer 测试

```
mymac:~ chixinning$ http 127.0.0.1:8082/index.html
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>SUCCEED!</title>
</head>
<body>
<h1>10185501409 Chi Xinning</h1>

</body>
</html>

mymac:~ chixinning$ http 127.0.0.1:8082/something.html
HTTP/1.1 404 File Not Found
Content-Length: 23
Content-Type: text/html

<h1>File Not Found</h1>

mymac:~ chixinning$ http 127.0.0.1:8082/shutdown

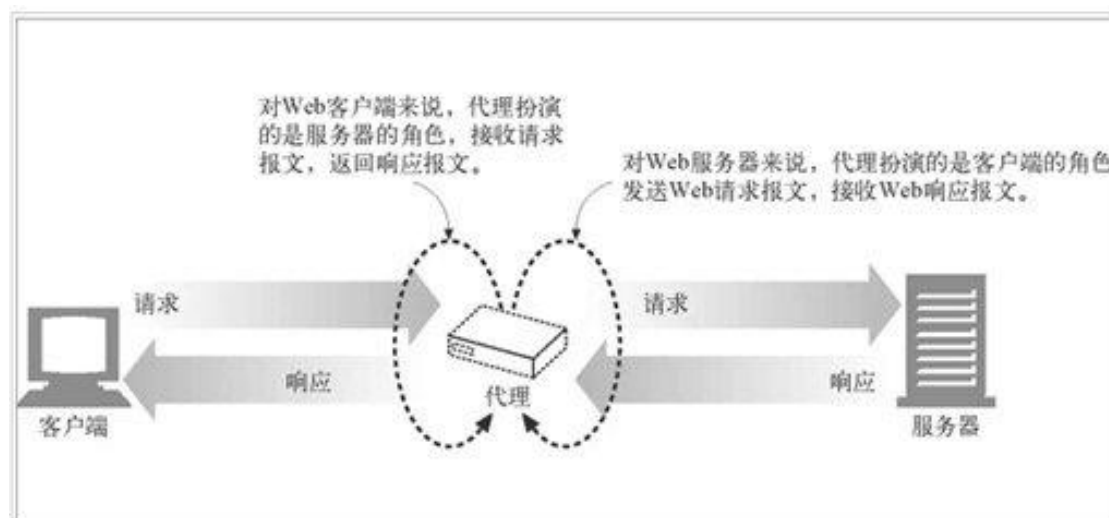
http: error: ConnectionError: ('Connection aborted.', RemoteDisconnected('Remote end closed connection without response')) while doing a GET request to URL: http://127.0.0.1:8082/shutdown

/Library/Java/JavaVirtualMachines/jdk-13.0.1...
服务器已经监听:
0.0.0.0/0.0.0.0:8082
Server阻塞等待中
Server阻塞等待中
GET /index.html HTTP/1.1
GET /index.html HTTP/1.1
Server阻塞等待中
GET /something.html HTTP/1.1
GET /something.html HTTP/1.1
Server阻塞等待中
GET /shutdown HTTP/1.1
GET /shutdown HTTP/1.1

Process finished with exit code 0
```

2.Web 代理服务器

web 代理服务器的原理:



1. 等待来自客户 (Web 浏览器) 的请求。
2. 启动一个新的线程, 以处理客户连接请求。
3. 读取浏览器请求的第一行 (该行内容包含了请求的目标 URL) 。
4. 分析请求的第一行内容, 得到目标服务器的名字和端口。
5. 打开一个通向目标服务器 (或下一个代理服务器, 如合适的话) 的 Socket。
6. 把请求的第一行发送到输出 Socket。
7. 把请求的剩余部分发送到输出 Socket。
8. 把目标 Web 服务器返回的数据发送给发出请求的浏览器。

区分 https 代理和 http 代理

http 消息直接转发

题目 1.2 V1.0 版 (使用 java 线程池)

主方法, 同 web 服务器, 等待来自客户的请求, 启动新线程以处理客户连接请求 (完成逻辑 1-2)

```
ExecutorService Socketexecutor = Executors.newFixedThreadPool(10); //线程池
ServerSocket ss = new ServerSocket(11111); //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
```

线程池, 每次建立一个连接, 都建立 handleRequest, handleRequest 承接代理的重任。

handleRequest:逻辑 3-6,part7/part8

*/*init variable*/*

```
String line = "";
InputStream clineInput = socket.getInputStream();
String tempHost="",host;
int port =80; //默认
String type=null;
OutputStream os = socket.getOutputStream();
BufferedReader br = new BufferedReader(new InputStreamReader(cline
tInput));

/*3.读取浏览器请求的第一行, 该行内容包含了请求的目标 URL*/
/*4.分析请求的第一行, 得到目标服务器的名字和端口*/
int flag=1;
StringBuilder sb =new StringBuilder();
//读取 HTTP 请求头, 拿到 HOST 请求头和 method.
/*specific code omitted */
Socket proxySocket = null; //代理间通信的 socket

//连接到目标服务器

if(host!=null&&!host.equals("")) {
    //5.打开一个通向目标服务器的 Socket.
```



```

        proxySocket = new Socket(host,port);
        OutputStream proxyOs = proxySocket.getOutputStream();//输出
        InputStream proxyIs = proxySocket.getInputStream();//输入
        /*https 不可直接转发*/
        assert type != null;
        if(type.equalsIgnoreCase("connect")) {           //https 请求的话, 告诉
客户端连接已经建立 (下面代码建立)
            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("utf-8"));
        proxyOs.flush();
        }
        //新开线程转发客户端请求至目标服务器
        ExecutorService Proxyexecutor=Executors.newFixedThreadPool(10);
        Proxyexecutor.submit(new Thread()->proxyHandler(clinetInput,proxyOs,host));

        //转发目标服务器响应至客户端
        Proxyexecutor.submit(new Thread()->proxyHandler(proxyIs,os,host));
    }
}

```

Proxyhandler:单纯消息转发.逻辑 7 的具体实现

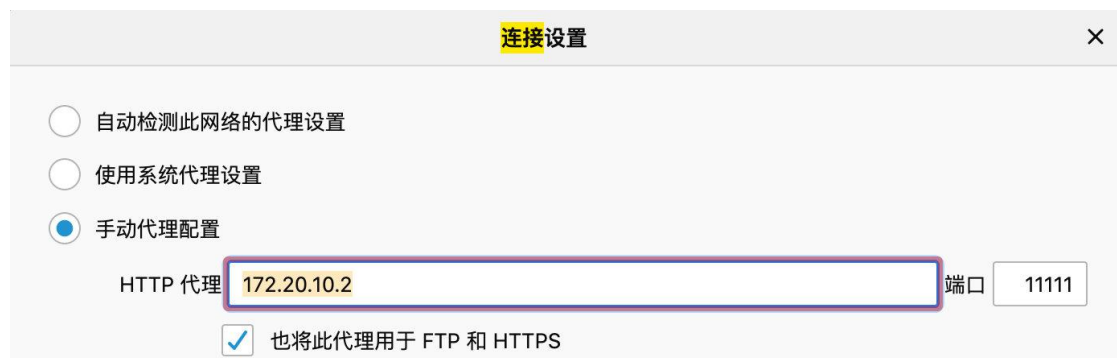
```

while(true){
    BufferedInputStream bis=new BufferedInputStream(input);
    byte[]buffer=new byte[1024];
    int lenght=-1;
    while((lenght=bis.read(buffer))!=-1){
        output.write(buffer,0,lenght);
        lenght=-1;
    }
    output.flush();
}

```

测试

更改浏览器的代理设置。



代理服务器未启动时，无法接入。

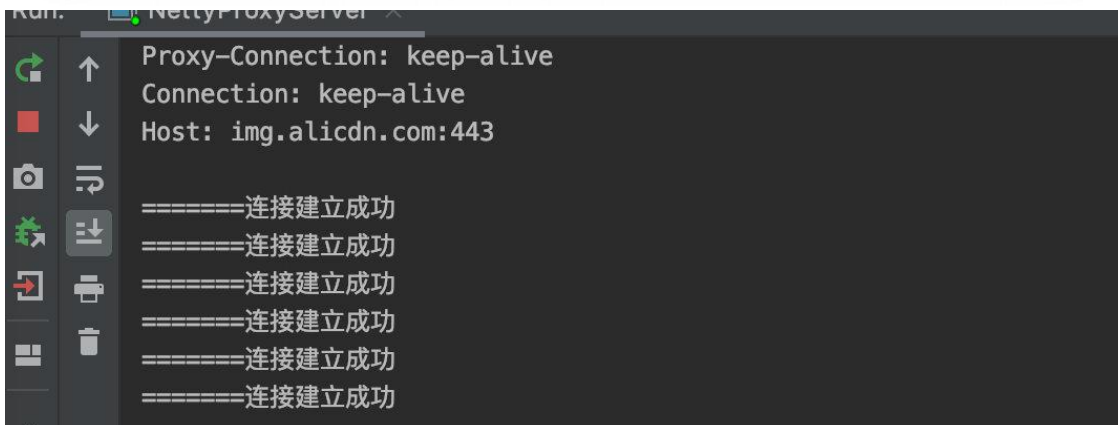


代理服务器拒绝连接

Firefox 尝试与您指定的代理服务器连接时被拒绝。

- 请检查浏览器的代理服务器设置是否正确。
- 请联系您的网络管理员以确认代理服务器工作正常。

重试



五、总结

这个实验从 0 到 1 起步，从再次复习 http 协议开始，到 socket 编程的回顾，到接触 BIO/线程池/NIO 的概念，这个实验写了快 1 个月之久。

在写 project 的过程中，才感受到编程和书本理论知识的距离。对于 java 网络编程还有很多需要学习的框架和逻辑结构，如何提高并发也只是浅尝辄止的开始接触。

自己对于 java 编程的核心逻辑掌握的还不是很熟练，有待提高和改进。

这次实验让我收获颇丰，受益匪浅。