计算机网络项目设计文档

> *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1.1 V1.0 无线程池

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

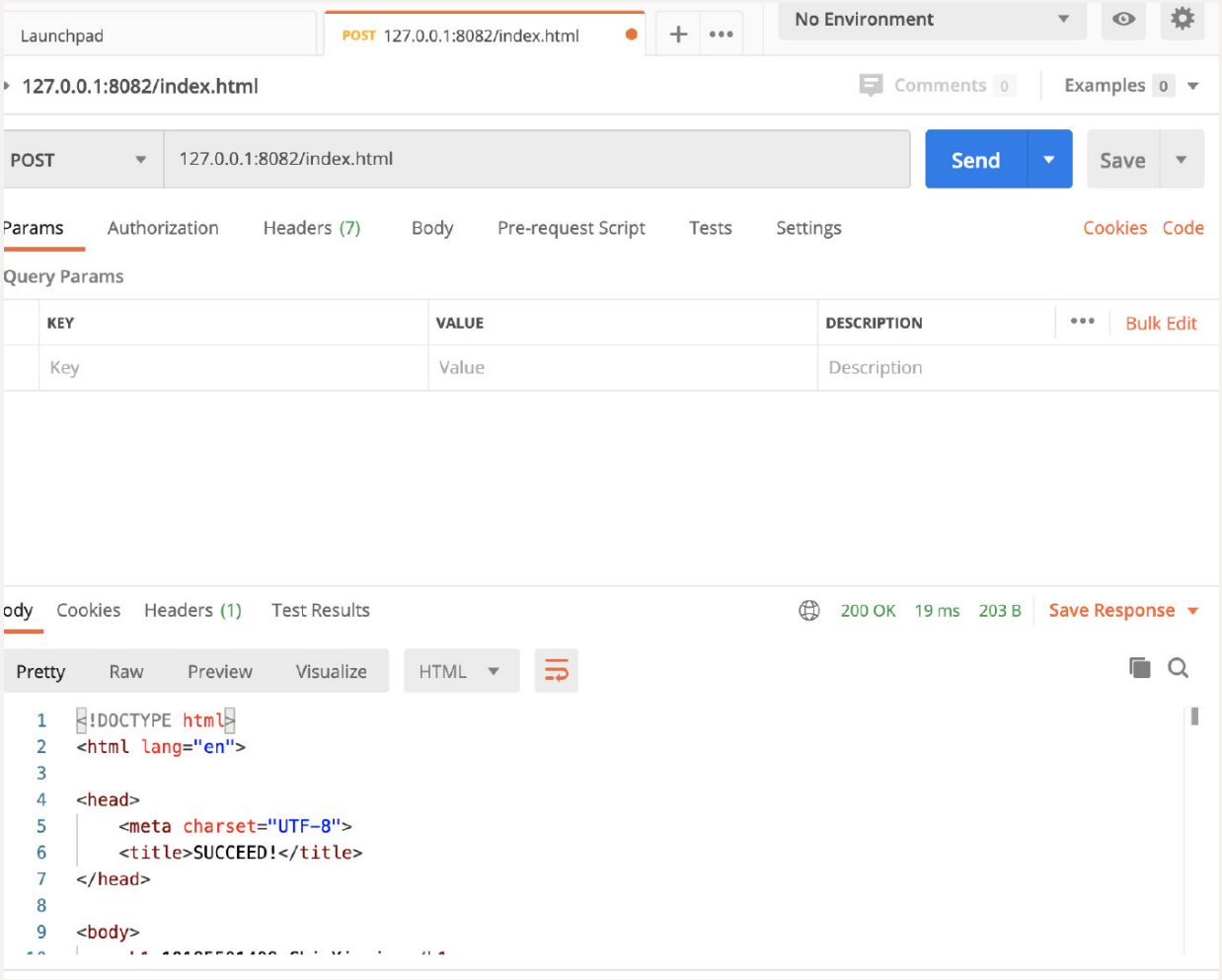
```
    public static void handleRequest(Socket socket){
        try{
            //使用socket对象中的getInputStream。
            InputStream is=socket.getInputStream();
            //输入流对象is转换为字符缓冲输入流
            BufferedReader br=new BufferedReader(new
InputStreamReader(is));
            String line=br.readLine();
            /*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();
        }
    }
```

Http报文格式：

测试结果：

Postman POST测试，GET于1000个并发测试时完成。

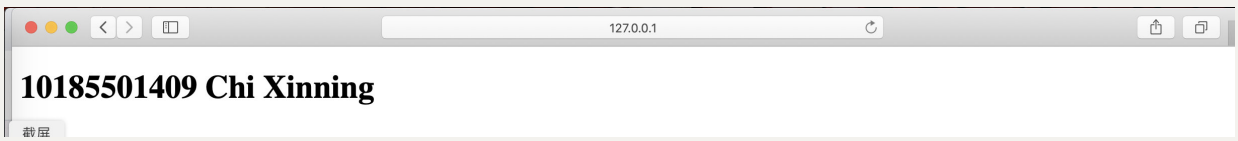

Launchpad
POST 127.0.0.1:8082/index.html
No Environment
127.0.0.1:8082/index.html
Comments 0
Examples 0
POST
127.0.0.1:8082/index.html
Send
Save
Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings
Cookies
Code
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
Key
Value
Description
ody
Cookies
Headers (1)
Test Results
200 OK
19 ms
203 B
Save Response
Pretty
Raw
Preview
Visualize
HTML
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>SUCCEED!</title>
</head>
<body>

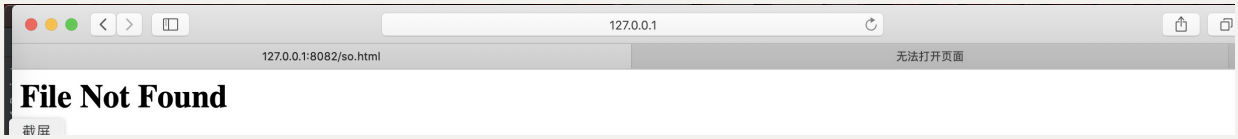
httpie测试结果

```
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>
```

- 状态行：HTTP 版本号、状态码和状态值
- 请求头部：c.f.`Header`是一个个的key-value值。
- 空行：`os.write("\r\n".getBytes()); //必须写入空行，否则浏览器不解析`用来分割header和数据
- 请求数据：`os.write()`一段html。

浏览器测试结果





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];
    }
```

题目1.1 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。

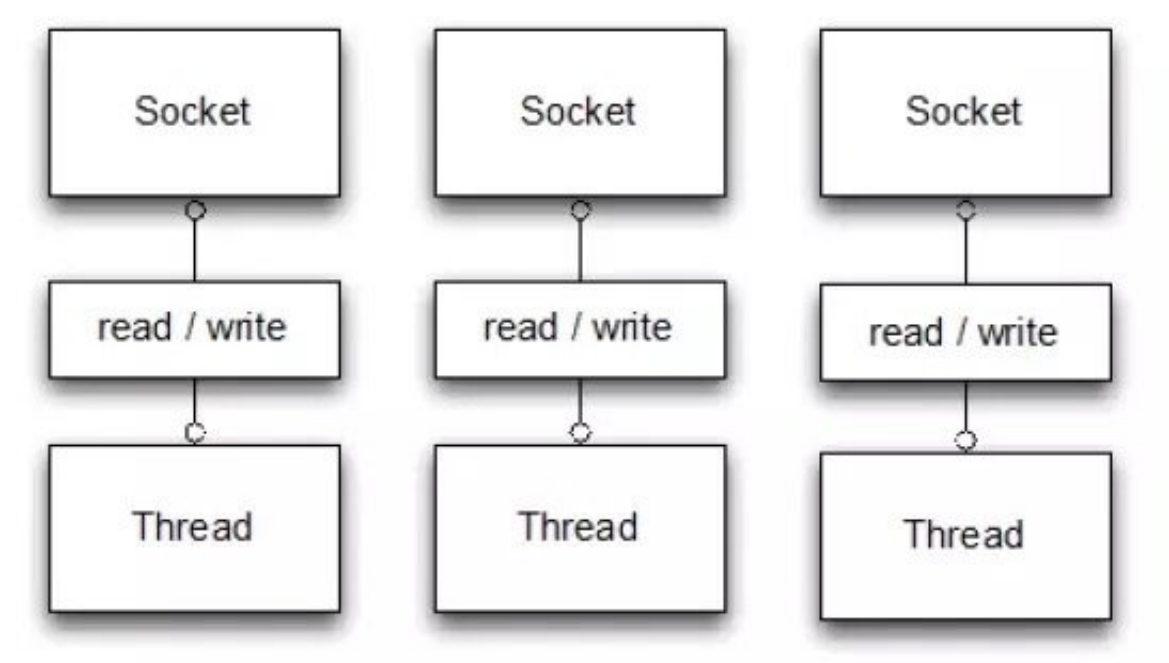
添加线程池的webServer代码

```
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)));
         }
```

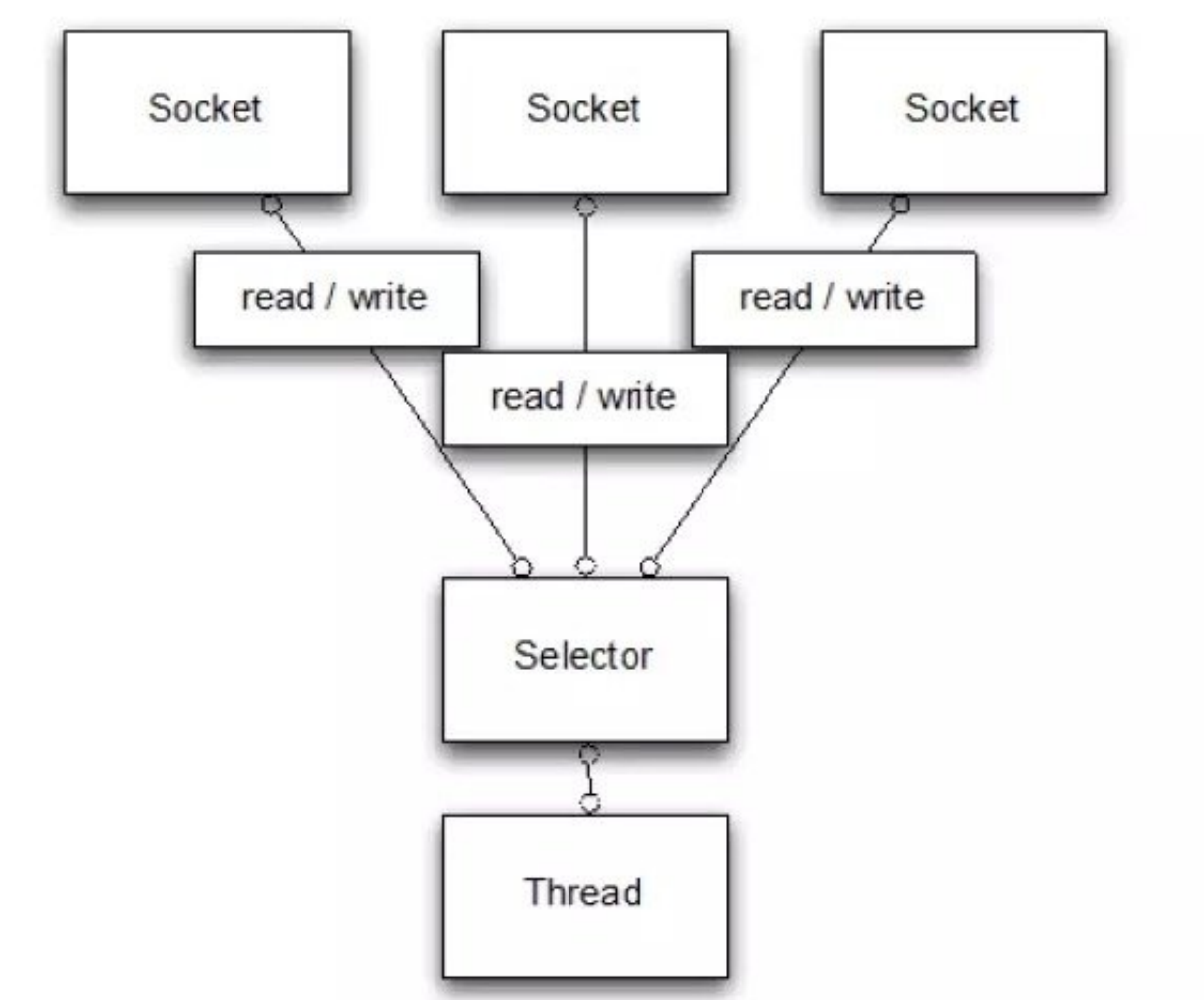
题目1.1 V3.0 Netty版(NIO)

为什么NIO可以提高高并发

传统BIO阻塞模型:



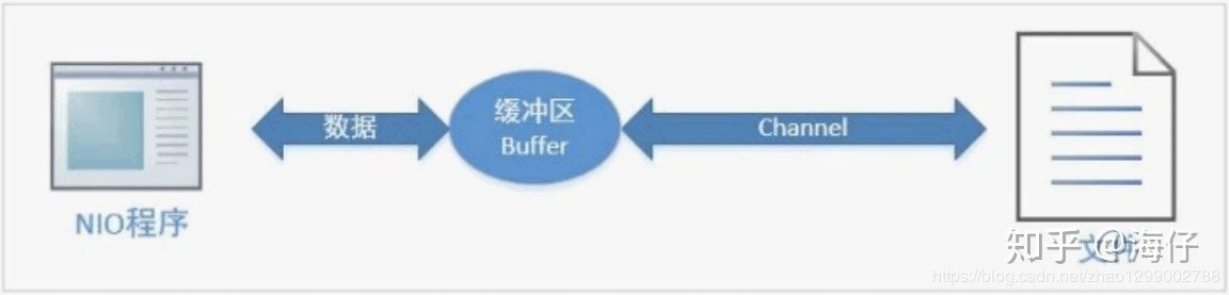
NIO模型：



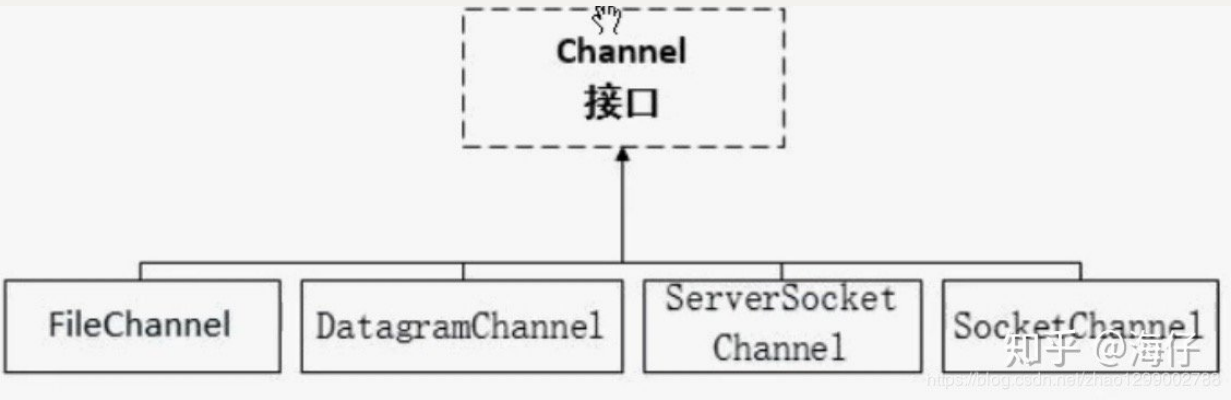
BIO等待客户端发数据这个过程是阻塞的，这样就造成了一个线程只能处理一个请求的情况，而CPU性能的限制导致机器能支持的最大线程数是有限的。

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

EventLoopGroup是一组EventLoop的抽象，Netty为了更好的利用多核CPU资源，一般会有多个EventLoop同时工作，每个EventLoop维护着一个Selector实例。

在Netty服务器端编程中，我们一般都需要提供两个EventLoopGroup，例如：BossEventLoopGroup和WorkderEventLoopGroup。

通常一个服务端口即一个ServerSocketChannel对应一个Selector和一个EventLoop线程。BossEventLoop负责接收客户端的连接并将SocketChannel交给WorkerEventLoopGroup来进行IO处理，

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。

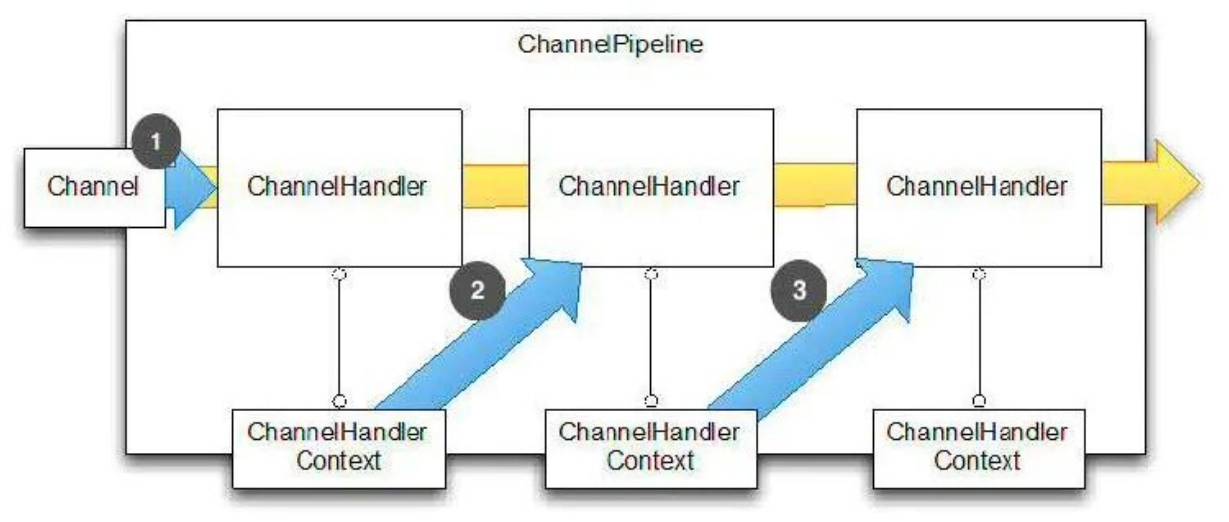
NIO与Netty?

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作为服务器短的通道
                    .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。
    }
}
```

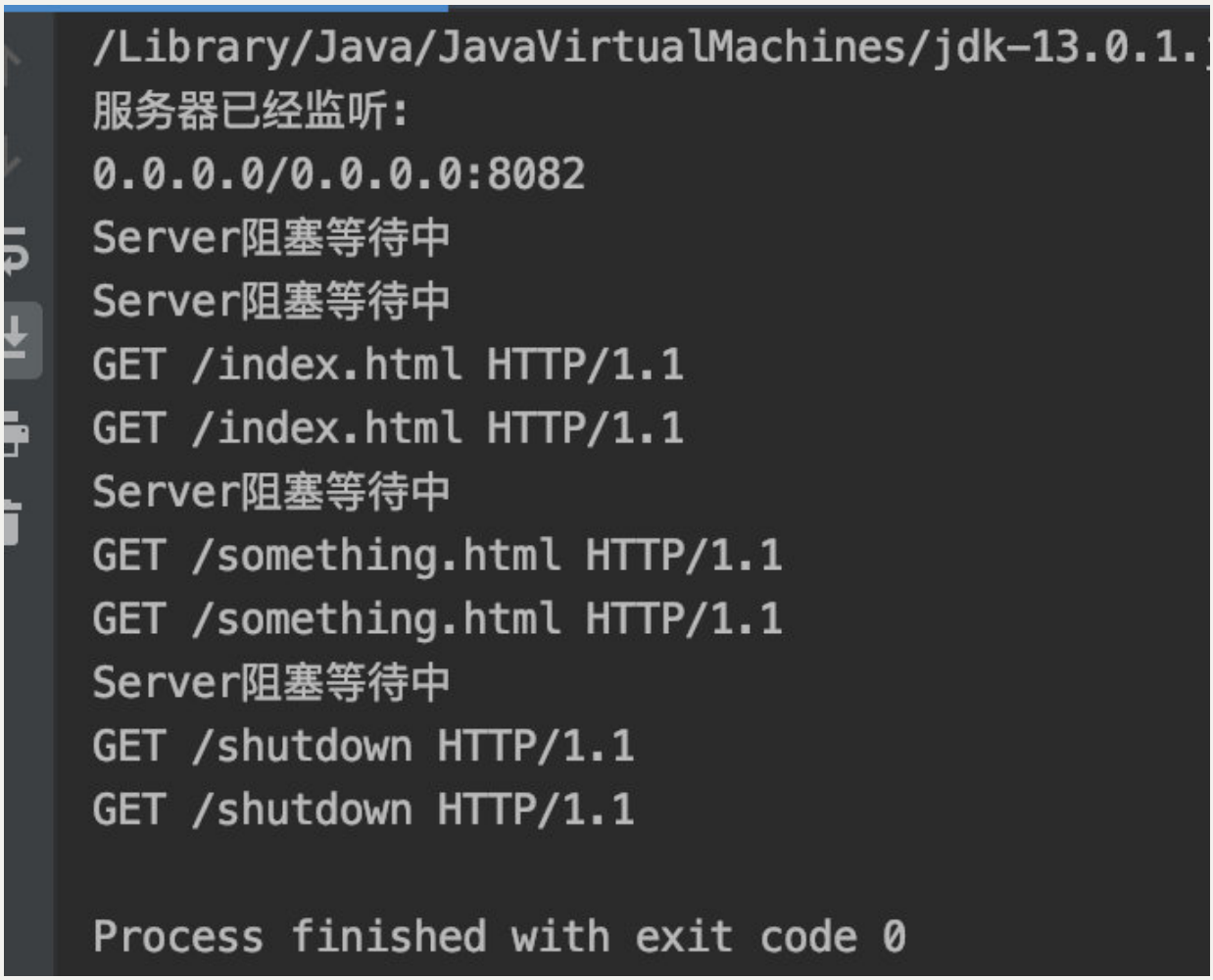
```
package staticServer;
import io.netty.channel.*;
import io.netty.handler.codec.http.*;
import java.io.File;
import java.io.RandomAccessFile;

/*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_LAS
T_CONTENT);
        randomAccessFile.close();//关闭文件
    }
}
```

测试

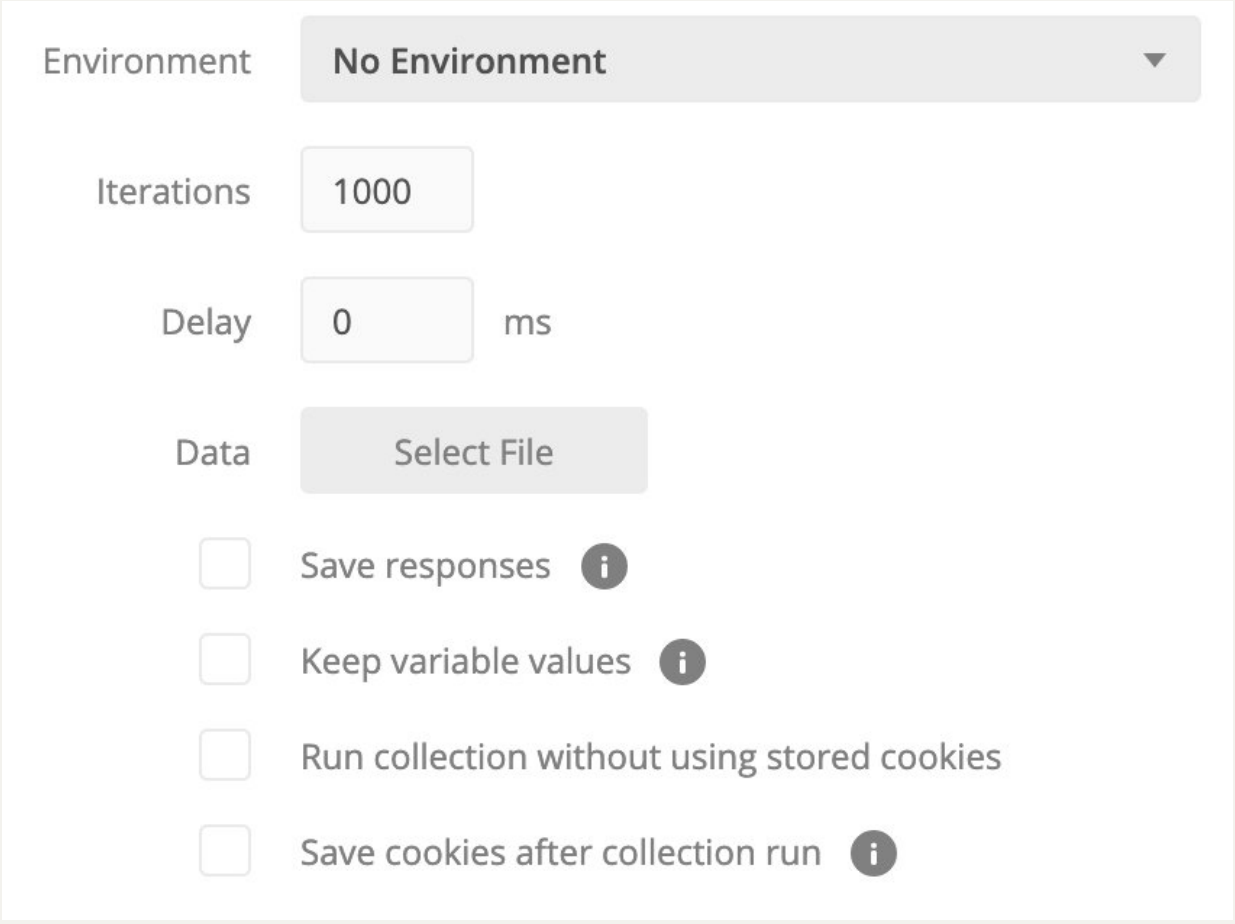
```
[mymac:~ chixinning$ http 127.0.0.1:8082/index.html
HTTP/1.1 200 OK
Content-Type: text/html

<!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 connectio
 without response')) while doing a GET request to URL: http://127.0.0.1:8082/shutdown
```

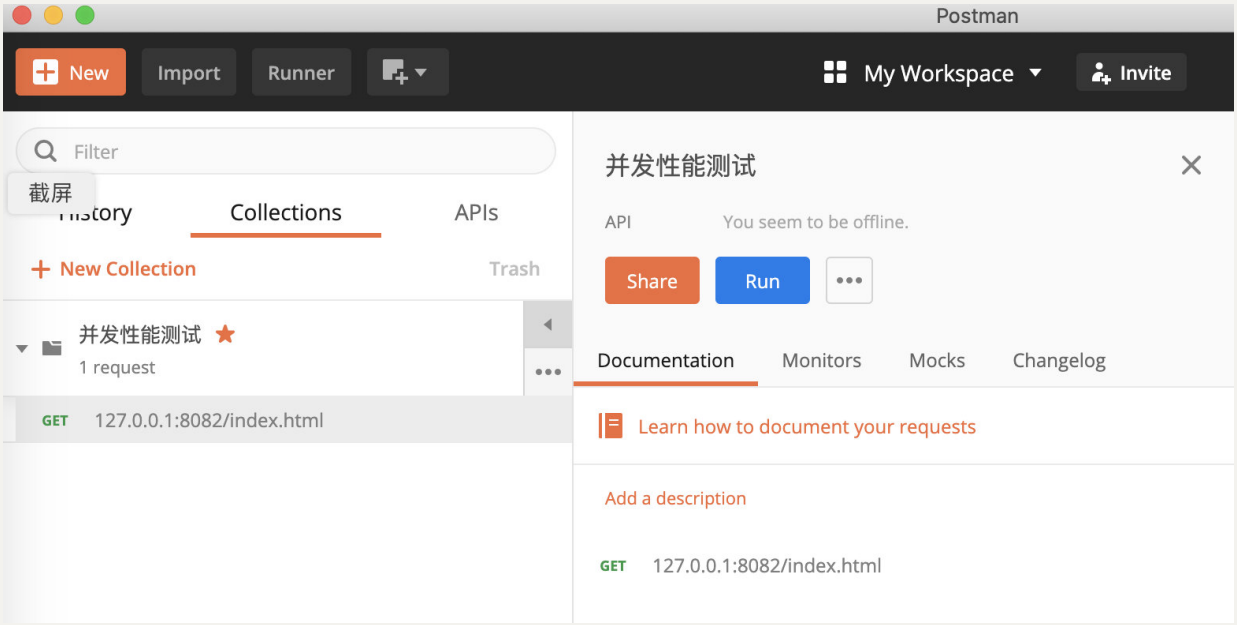


性能测试

测试环境

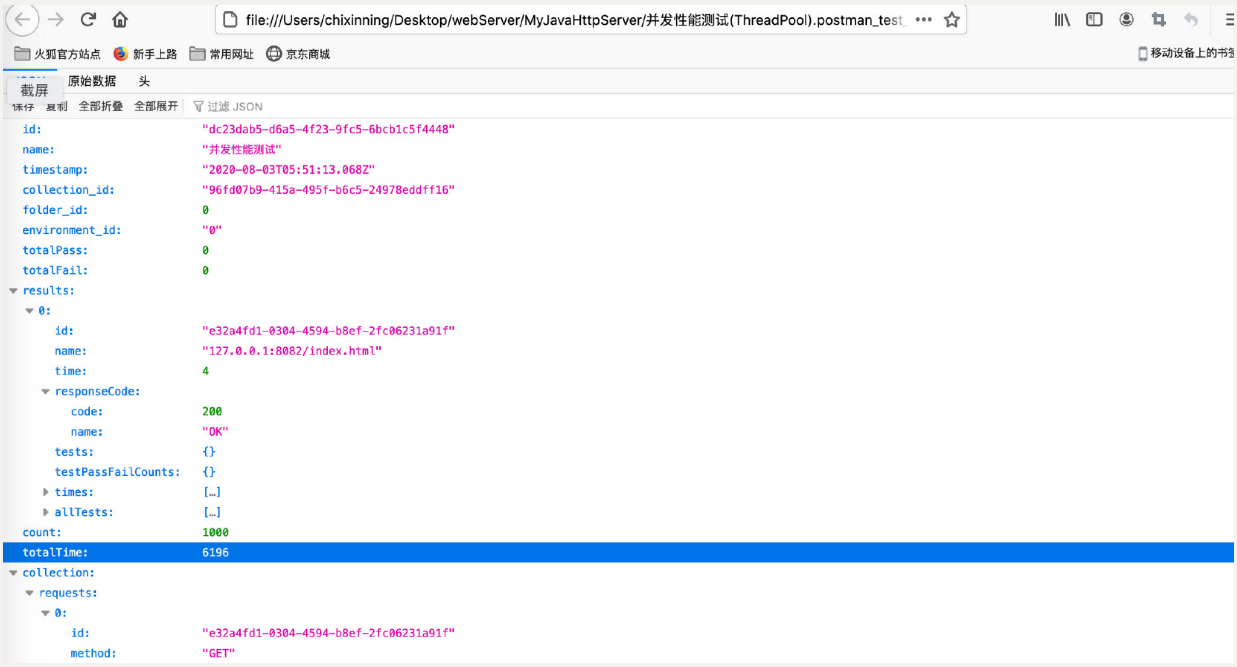
Postman串联测试





使用`postman`进行服务器性能并发测试以后：

v1.0不使用线程池，仅使用线程进行接管



v2.0 使用线程池

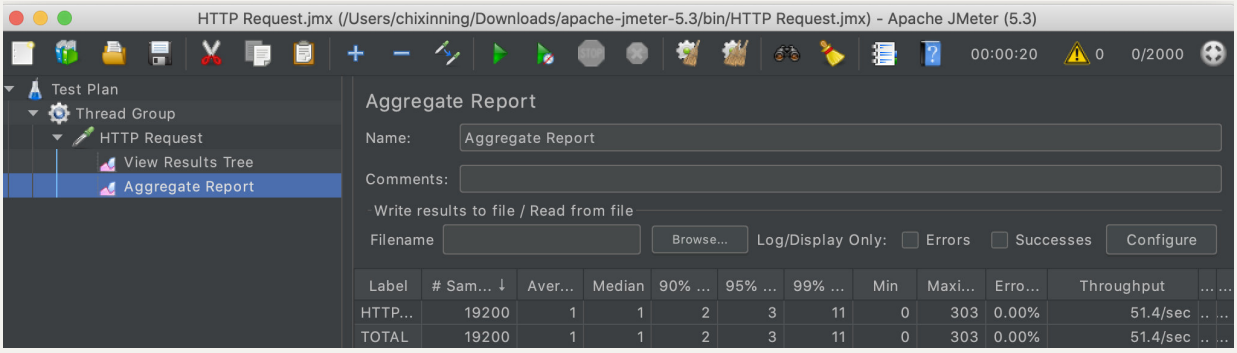
可以很明显的看出使用传统BIO模型，对系统的负载消耗也会较大，即我的系统会出现很大声的系统负载的声音/

v3.0使用netty框架

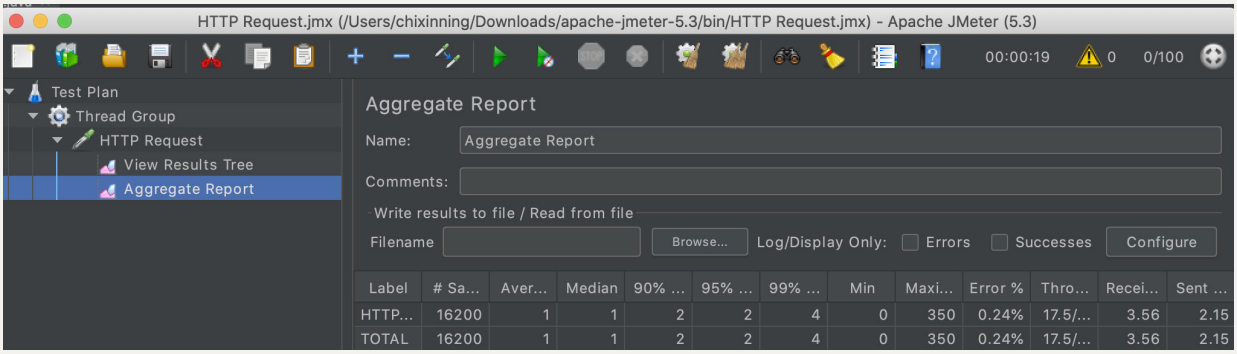
这3个版本的吞吐的差异非常的明显。

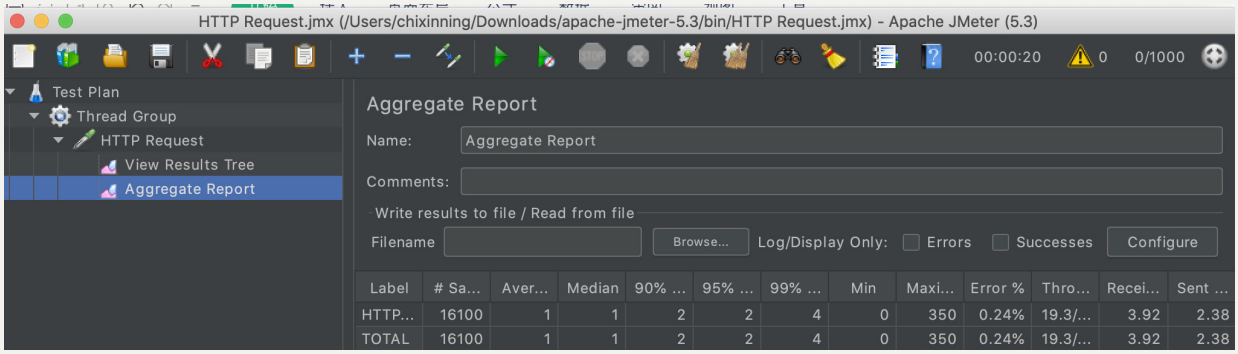
JMeter并发测试

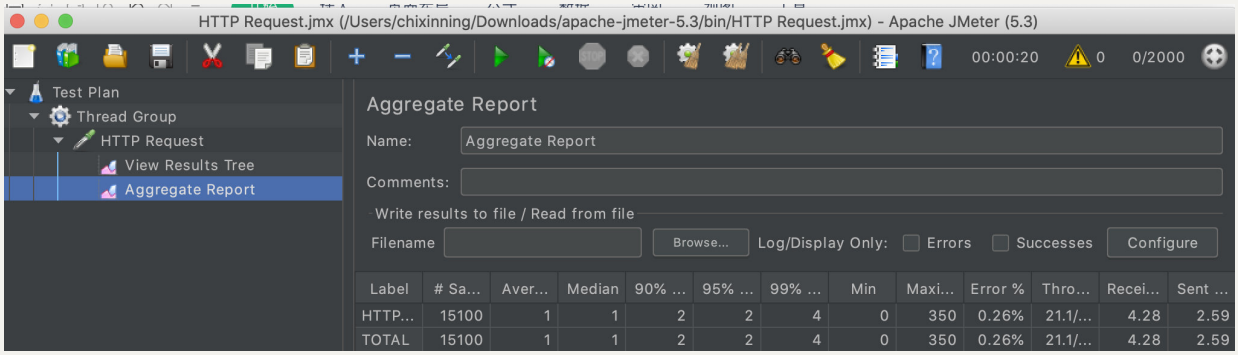
V3.0 使用netty框架



V2.0使用线程池

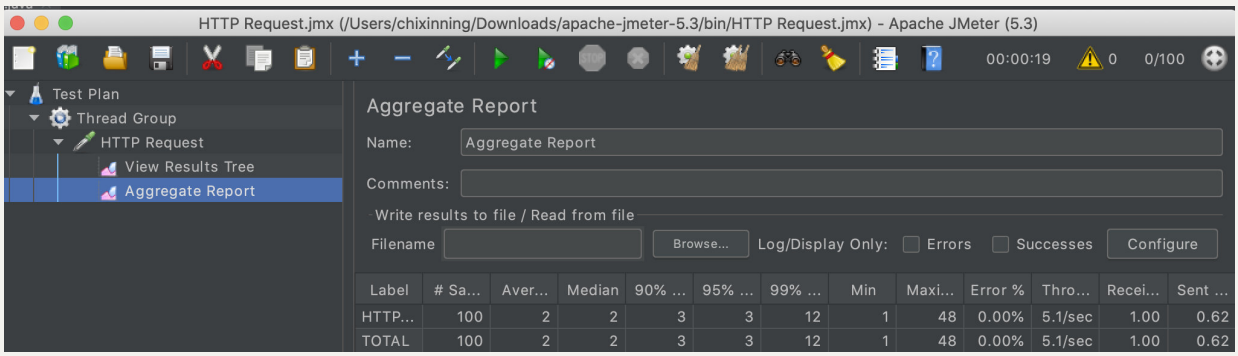




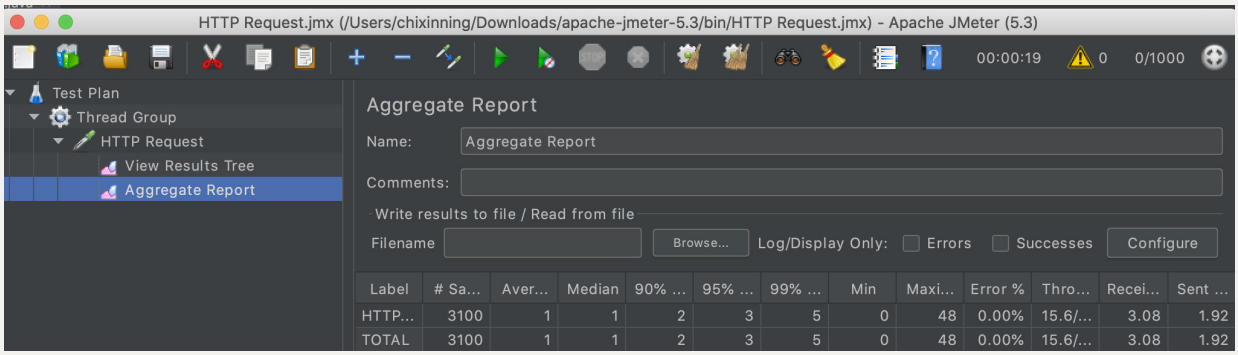


V1.0传统的BIO

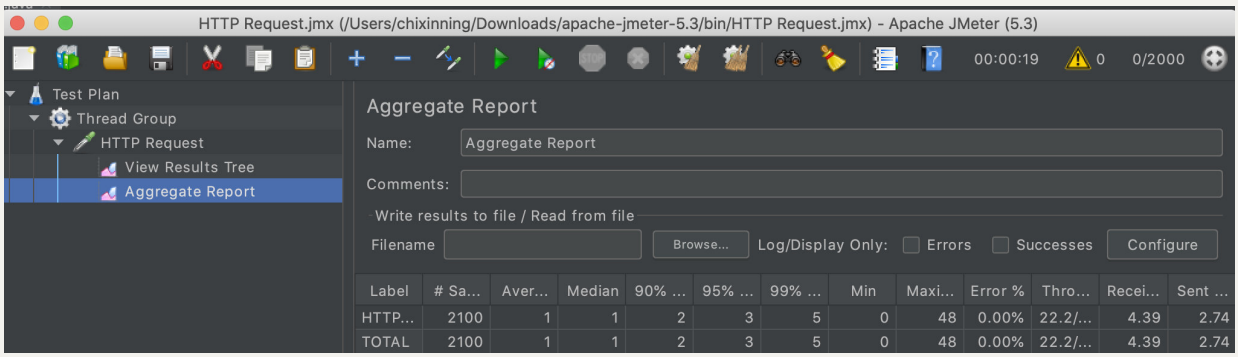
ThreadUsers=100，ramp-up period:20



ThreadUsers=1000，ramp-up period:20



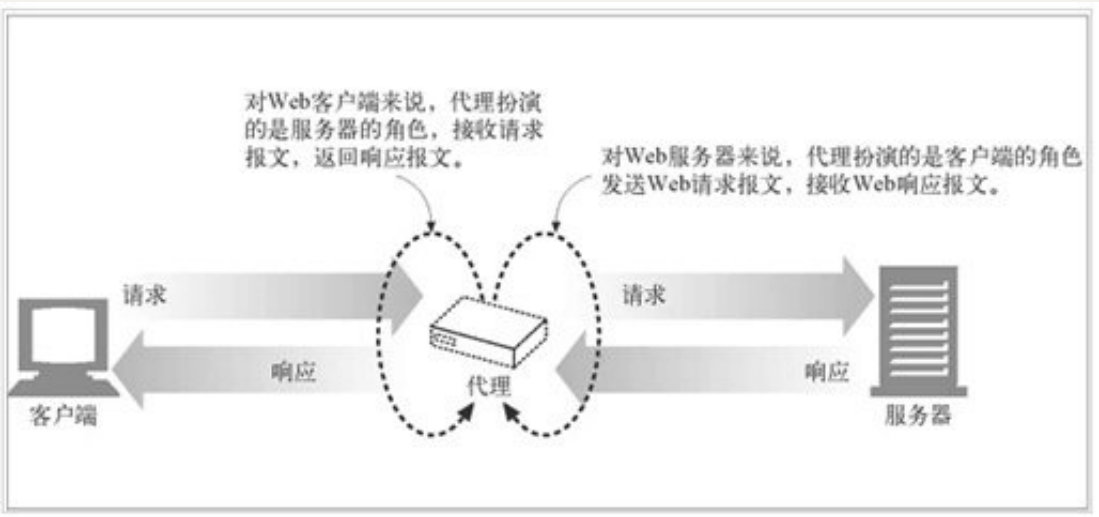
ThreadUsers=2000，ramp-up period:20



> *题目1.2:用Java开发一个简单的Web代理服务器*

题目1.2:用Java开发一个简单的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的请求（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。

httpRequest报文格式样例。

```
GET /search?hl=zh-CN&source=hp&q=domety&aq=f&oq= HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint,
application/msword, application/x-silverlight, application/x-
shockwave-flash,
Referer: <a
href="http://www.google.cn/">http://www.google.cn/</a>
Accept-Language: zh-cn
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
SV1; .NET CLR 2.0.50727; TheWorld)
Host: <a href="http://www.google.cn">www.google.cn</a>
Connection: Keep-Alive
Cookie:
PREF=ID=80a06da87be9ae3c:U=f7167333e2c3b714:NW=1:TM=1261551909
:LM=1261551917:S=ybYcq2wpfefs4V9g;
NID=31=ojj8d-
IygaEtSxLgaJmqSjVhCspkviJrB6omjamNrSm8lZhKy_yMfO2M4QMRKcH1g0iQ
v9u-2hfBW7bUFwVh7pGaRUb0RnHcJU37y-
FxlRugatx63JLv7CWMD6UB_O_r
```

handleRequest:逻辑3-6,part7/part8

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

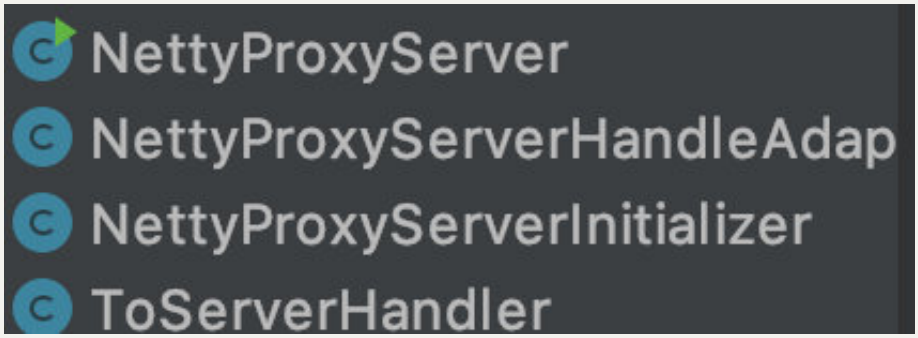
```
while(true){
    BufferedInputStream bis=new BufferedInputStream(input);
    byte[]buffer=new byte[1024];
    int lenght=-1;
    while((lenght=bis.read(buffer))!=-1){
        output.write(buffer,0,lenght);
        lenght=-1;
    }
    output.flush();
```

题目1.2V2.0Netty版(能力有限，代码只是模仿，很多细节没有考虑到)

Netty的整体实现逻辑跟线程池大致类似，只是netty使用NIO同步非阻塞可以增加并发量。

注意代理需要区分http代理和https代理

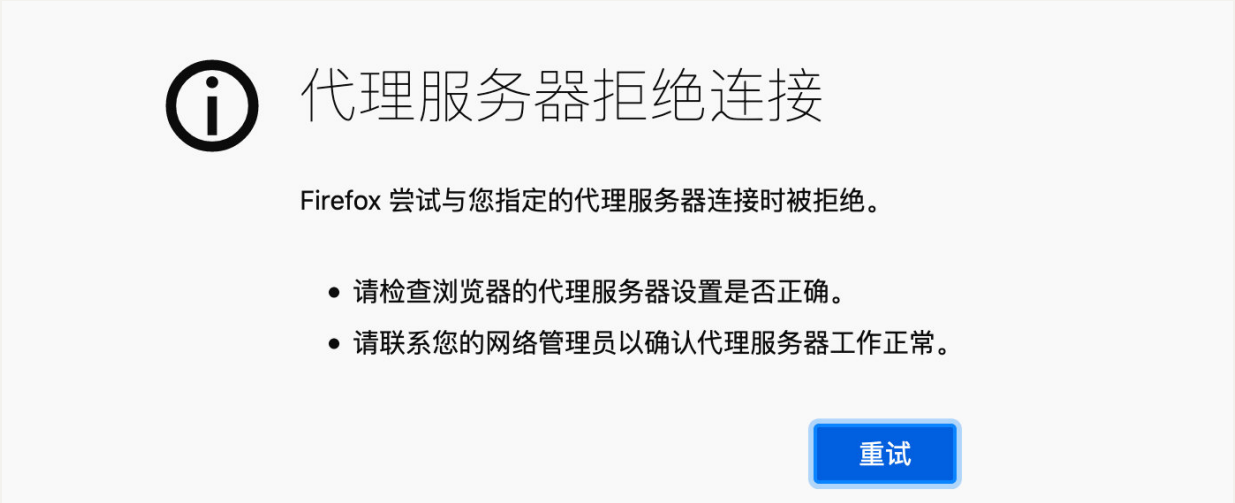
NettyProxyServer的文件结构如下：

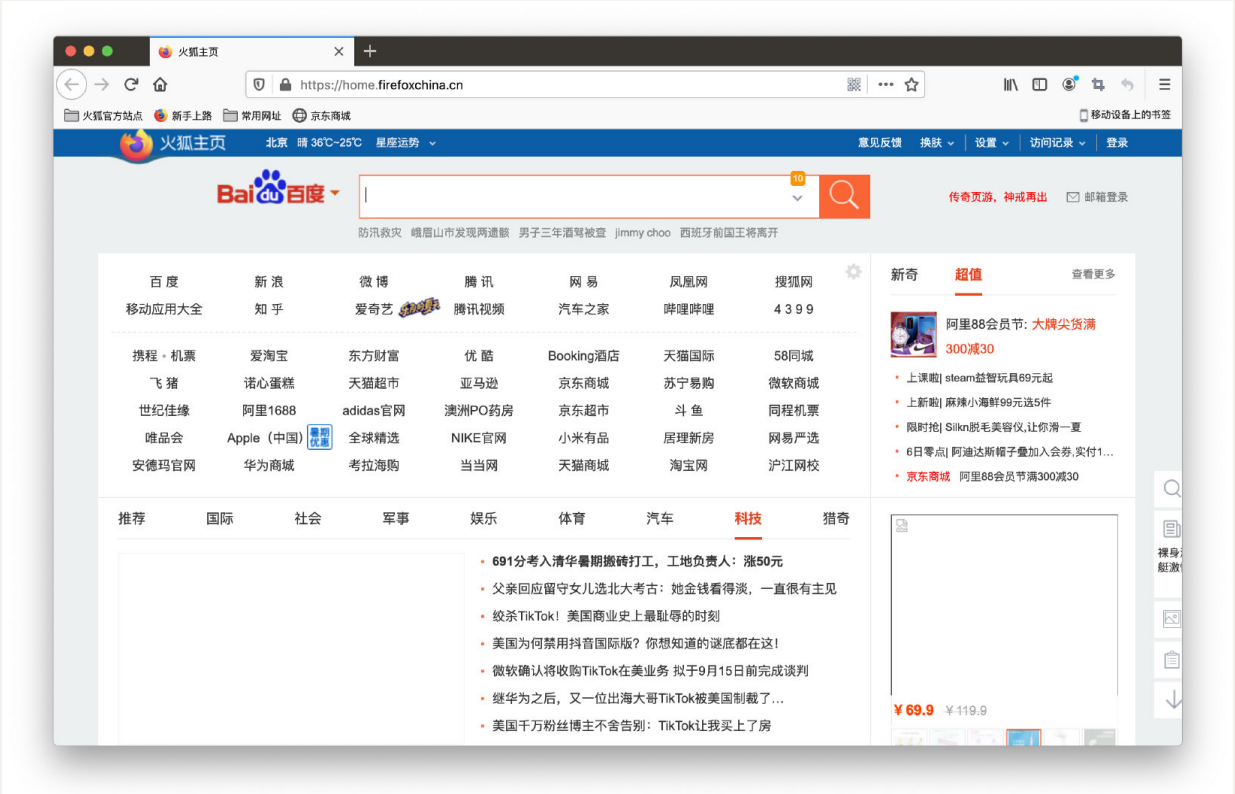


ProxyServer测试

更改浏览器的代理设置。

代理服务器未启动时，无法接入。





总结与感想

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。