

# 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：2022 级      上机实践成绩：

指导教师：张召      姓名：李芳      学号：10214602404

上机实践名称：期末项目      上机实践日期：2024.06.20

上机实践编号：Final      组号：      上机实践时间：

## 一、题目要求

a) 实现具有以下两功能的 HTTP 服务器：

功能1(10')	功能2(20')
浏览器访问 localhost:8080/index.html 正确显示自己的学号姓名	浏览器访问 localhost:8080/ 以反向代理形式正确返回院网首页

b) 进行性能测试：

对简易 HTTP 服务器，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。

(加分项，可选项) 使用 Java NIO 库提升系统性能

二、功能实现情况（整体思路：先逐个实现单个任务，最后按照补充要求合并成一个程序）

### 储备知识：

#### ✧ NIO 类：处理非阻塞式 I/O 操作

1. Channel（通道）：到 I/O 设备（如文件或网络套接字）的开放连接。

● 通道类：

FileChannel：用于文件 I/O。

SocketChannel：用于网络套接字 I/O。

ServerSocketChannel：用于服务器套接字 I/O。

DatagramChannel：用于 UDP 连接。

- 方法:

**open():** 打开一个通道。

**close():** 关闭通道。

**read(ByteBuffer dst):** 从通道读取数据到缓冲区。

**write(ByteBuffer src):** 将缓冲区数据写入通道。

2. Buffer (缓冲区): 存储数据的容器, 可以在通道和应用程序之间传输数据。

- 缓冲区类:

**ByteBuffer:** 用于字节数据。

CharBuffer: 用于字符数据。

IntBuffer、FloatBuffer 等用于不同基本数据类型。

- 方法:

**allocate(int capacity):** 分配一个新的缓冲区。

**put(...):** 向缓冲区写数据。

**get(...):** 从缓冲区读数据。

**flip():** 切换缓冲区从写模式到读模式。

**clear():** 清空缓冲区, 为下一次写入做准备。

**remaining():** 返回当前位置到限制之间的元素数。

3. Selector (选择器): 监听多个通道的事件 (如连接、读、写事件), 单线程可以管理多个通道。

方法:

**open():** 打开一个选择器。

**select():** 阻塞直到至少有一个通道准备就绪。

**selectedKeys():** 返回已选择的键集。

**register(Selector sel, int ops):** 将通道注册到选择器, 并指定监听的事件类型

✧ **线程池：执行大量异步任务而不需要频繁地创建和销毁线程**

java.util.concurrent 类：

1. Executor 接口：基本的线程池接口

方法：

- **void execute(Runnable command): 提交一个任务执行**

2. ExecutorService 接口：提供管理和控制线程池的方法

方法：

- **Future<?> submit(Runnable task): 提交一个任务并返回一个 Future，表示任务的执行结果。**
- **<T> Future<T> submit(Callable<T> task): 提交一个有返回值的任务，并返回一个 Future。**
- **void shutdown(): 启动有序关闭，不接受新任务。**
- **List<Runnable> shutdownNow(): 试图停止所有正在执行的任务，并返回等待执行的任务列表。**
- **boolean isShutdown(): 判断线程池是否已关闭。**
- **boolean isTerminated(): 判断所有任务是否已终止。**

3. Executors 工具类：创建各种类型线程池的静态工厂方法。

方法：

- **ExecutorService newFixedThreadPool(int nThreads): 创建一个固定大小的线程池。**
- **ExecutorService newCachedThreadPool(): 创建一个可缓存的线程池。**
- **ScheduledExecutorService newScheduledThreadPool(int corePoolSize): 创建一个支持定时和周期性任务的线程池。**

4. ThreadPoolExecutor 类：可以自定义线程池的核心线程数、最大线程数、线程空闲时间、任务队列等。

✧ **关键字：可以定义类成员变量的访问权限和属性**

1. **private**：声明类的私有成员，这些成员只能在类的内部访问，不能在类的外部直接访问。

2. `final`: 声明常量或不变对象。对于变量来说, `final` 表示该变量只能被赋值一次, 一旦赋值后就不能修改。
3. `private final`: 该成员变量是私有的, 并且只能被赋值一次。
4. `private static`: 该成员变量是私有的, 并且是静态的, 即属于类而不是某个实例。
5. `private static final`: 该成员变量是私有的、静态的, 并且只能被赋值一次。

## 功能实现:

### 1. 简易 HTTP 服务器 (这部分只介绍 `RequestTask` 类)

为了实现客户端请求 `localhost:8080/index.html` 之后, 显示我的学号及姓名, 我创建了一个继承 `Runnable` 接口的 `RequestTask` 类, 它主要处理网络连接中客户端的请求, 并返回一个固定的 HTTP 响应: 学号、姓名、服务器。它将被后面整合代码中的 `requestSolver` 调用交给线程池运行:

#### ➤ 思路解析

首先, 它需要接收一个 `SocketChannel` 对象并将其存储在类的私有字段 `socketChannel` 中, 覆盖重写 `run` 方法:

生成固定的 HTTP 响应, 包括状态行、头部和主体部分; 然后, 将字符串响应内容转换为字节数组, 并将其包装在 `ByteBuffer` 类型的 `responseBuffer` 中, 并通过 `SocketChannel` 发送给客户端, 此处使用 `while` 循环确保所有数据都被发送; 发送完成后, 关闭 `Channel` 释放资源。

#### ■ 代码展示

```
class RequestTask implements Runnable { 2个用法
    private final SocketChannel socketChannel; 3个用法
    public RequestTask(SocketChannel socketChannel) { 1个用法
        this.socketChannel = socketChannel;
    }

    @Override
    public void run() {
        try {
            String response = "HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=UTF-8\r\n\r\n10214602404 李芳<br><br>简易 HTTP 服务器";
            ByteBuffer responseBuffer = ByteBuffer.wrap(response.getBytes(StandardCharsets.UTF_8));
            while (responseBuffer.hasRemaining()) {
                socketChannel.write(responseBuffer);
            }
            socketChannel.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

#### ➤ 代码优化

为了提升吞吐量，我对刚才的代码进行优化：

首先，因为响应内容是固定的，所以我使用静态 ByteBuffer 缓存响应内容，这样，响应内容只需要经历一次转换成字节数组并包装的过程，不需要每个 RequestTask 实例都创建一个新的 ByteBuffer，重复去编码和包装，减少了每次任务运行开销还有内存分配、垃圾回收的负担。

其次，为了线程安全，防止多线程共享 responseBuffer 可能导致的位置指针混乱、数据不一致或异常等并发问题，在 run() 函数中，使用 duplicate() 方法创建 responseBuffer 副本，每个任务使用 responseBuffer 的副本进行写操作。

### ■ 代码展示

```
class RequestTask implements Runnable { 2个用法
    private final SocketChannel socketChannel; 3个用法
    // 响应字节数组包装到静态Buffer中
    private static final ByteBuffer responseBuffer = ByteBuffer.wrap(("HTTP/1.1 200 OK\r\nContent-Type: text/html; charset=UTF-8\r\n\r\n10214602404 李芳<br><br>简易 HTTP 服务器"
    ).getBytes(StandardCharsets.UTF_8));

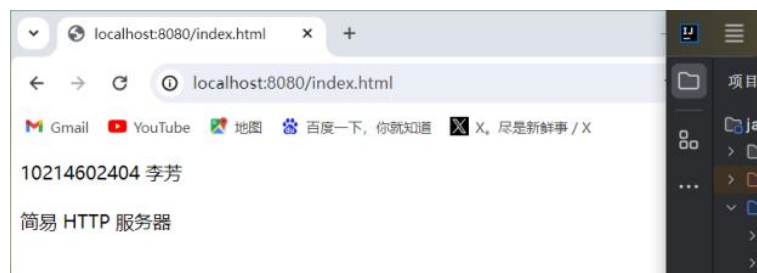
    public RequestTask(SocketChannel socketChannel) { 1个用法
        this.socketChannel = socketChannel;
    }

    @Override
    public void run() {
        try {
            // 创建responseBuffer的副本，避免多线程发生异常
            ByteBuffer buffer = responseBuffer.duplicate();
            socketChannel.write(buffer);
            socketChannel.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

### ➤ 结果展示

■ 输入：localhost:8080/index.html

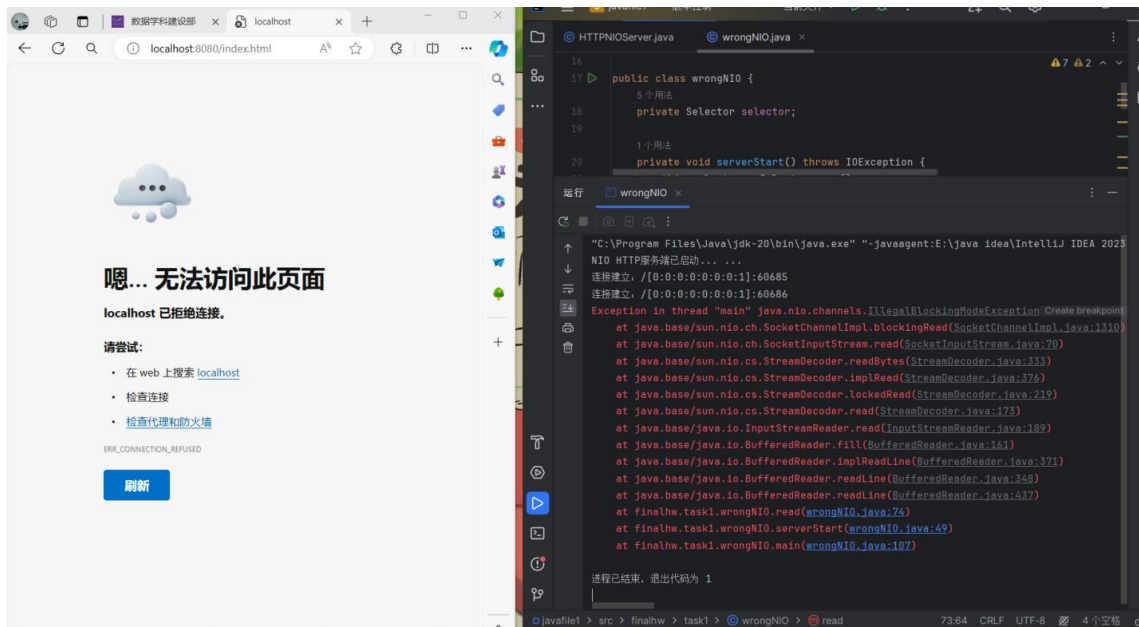
■ 网页显示：



### ➤ 出错反思

一开始，使用了 `BufferedReader` 这个阻塞式读取的类，在一个非阻塞模式的 `SocketChannel` 上执行阻塞式读取操作，导致无法正常进行数据读取和输出。

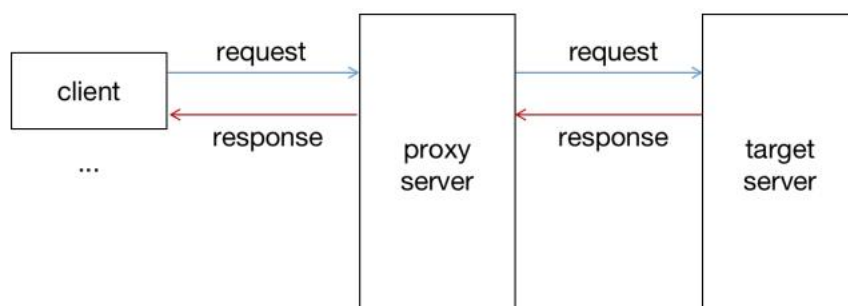
解决方法：直接通过 `channel` 实现非阻塞式 I/O 读取数据。



## 2. 实现反向代理（这部分只介绍 `TranspondTask` 类）

为了实现客户端请求 `localhost:8080/` 之后返回 `dase.550w.host` 的网页内容，实现反向代理，像上个方法一样，我创建了一个继承 `Runnable` 接口的 `TranspondTask` 类，它用来在两个 `SocketChannel` 之间转发 HTTP 请求和响应：从客户端 `SocketChannel` 接收 HTTP 请求，将其转发到目标服务器 `SocketChannel`，然后再将服务器的响应转发回客户端。

### ➤ 反向代理原理：



### ➤ `curl` 语句深入探究 `dase.550w.host` 请求头：

因为中途实现反向代理功能时，服务端打印输出到控制台的信息显示，目标服务器写入通道的响

```

Windows PowerShell
(base) PS C:\Users\HUAMEI> curl -v http://dase.550w.host
详细信息: GET http://dase.550w.host/ with 0-byte payload
详细信息: received 1-byte response of content type text/html

StatusCode      : 200
StatusDescription: OK
Content          : <!DOCTYPE html>
                  <html class="webplus-main" >
                  <head>
                  <meta charset="utf-8">
                  <meta name="renderer" content="webkit" />
                  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
                  <meta name="v...
RawContent       : HTTP/1.1 200 OK
                  Frame-Options: SAMEORIGIN
                  Accept-Ranges: bytes
                  Content-Type: text/html
                  Date: Wed, 12 Jun 2024 10:59:30 GMT
                  Server: Caddy/nginx
                  Set-Cookie: cookie=1VYDTLOV7486gxQpntQPUhGH09v7kyjU...
Forms            : {}
Headers          : [[Frame-Options, SAMEORIGIN], [Accept-Ranges, bytes], [Content-Type, text/html], [Date, Wed, 12 Jun
                  2024 10:59:30 GMT]...]
Images           : @{{innerHTML=; innerText=; outerHTML=; outerText=; tagName=IMG; alt=; src="/_upload/tpl/15/01/5377/template5377/images/Logodat
                  a.png}}; @{{innerHTML=; innerText=; outerHTML=; outerText=; tagName=IMG; alt=; src="/_upload/tpl/15/01/5377/template5377/images/more
                  .png}}; @{{innerHTML=; innerText=; outerHTML=; outerText=; tagName=IMG; width=3
                  20; src="/_upload/article/images/b9/11/35d0fae04932995d6555888f1e2/cae292ed-a0c6-436b-b360-808bc56e
                  a56.jpg}}; @{{innerHTML=; innerText=; outerHTML=; outerText=; tagName=IMG; widt
                  h=320; src="/_upload/article/images/cc/79/81ec5fb04974a3fb195e289f770e/b1a334b7-4ada-4fec-a2f2-c5518c
                  2aa9c0.png}}...}
InputFields      : @{{innerHTML=; innerText=; outerHTML=<input name="keyword" class="search-title" onfocus="if(this.val
                  ue == '') { this.value = ''; }" onblur="if(this.value == '') { this.value = ''; }" type="text" place
                  holder="Search" value="">; outerText=; tagName=INPUT; name=keyword; class=search-title; onfocus=if(t
                  his.value == '') { this.value = ''; }"; onblur="if(this.value == '') { this.value = ''; }"; type=text;
                  placeholder=Search; value=; @{{innerHTML=; innerText=; outerHTML=<input name="submit" class="search-
                  submit" type="submit" value="">; outerText=; tagName=INPUT; name=submit; class=search-submit; type=s
                  ubmit; value=}}
Links            : @{{innerHTML=; innerText=
                  ; outerHTML=<a title="e&#x00c5&#x27e9;ã&#x27e9;" href="/main.htm"></a>; outerText=; tagName=A; titl
                  e=e&#x00c5&#x27e9;ã&#x27e9;" href="/main.htm"}, @{{innerHTML=EN; innerText=EN
                  ; outerHTML=<a href="/EN/list.htm" target="_self">EN</a>; outerText=EN; tagName=A; href=/EN/list.htm
                  ; target=_self}, @{{innerHTML=e&#x27e9;" href="http://d
                  ase.ecnu.edu.cn/main.htm" target=_self">e&#x27e9;" href="http://dase.ecnu.edu.cn/main.htm; target=_self}, @{{innerHTML=<a class="menu-link" href="http://d
                  ase.ecnu.edu.cn/main.htm" target=_self">e&#x27e9;" href="http://dase.ecnu.edu.cn/main.htm; target=_self}, @{{innerHTML=<a class="menu-link" href="/41472/list.htm" target=_self">e&#x27e9;" href="/41472/list.htm; target=_self"}...}
ParsedHtml       : System.__ComObject
RawContentLength : 36865

```

首先，它接收 `SocketChannel` 和请求字符串 `requestStr`，并将它们存储在类的私有字段中。

第一步，打开一个到目标服务器的 SocketChannel 配置为非阻塞模式，连接到目标服务器地址 dase.550w.host 的 80 端口，使用 while 循环等待连接完成：

第三步，分配一个接收响应 `ByteBuffer`，使用 `StringBuilder` 构建完整的响应。这一步中，为了持续读取目标服务器的响应，使用 `while (true)`，将读取到的 `ByteBuffer` 数据写入到客户端的 `SocketChannel`，及时清空缓冲区以便下次读取。

最后，在 try-catch 块中处理可能出现的 IOException，在 finally 块中关闭 socketChannel 释放



资源。

## ■ 代码展示

```
class TranspondTask implements Runnable { 2个用法
    private final SocketChannel socketChannel; 3个用法
    private final String requestStr; 2个用法
    public TranspondTask(SocketChannel socketChannel, String requestStr) { 1个用法
        this.socketChannel = socketChannel;
        this.requestStr = requestStr;
    }
    @Override
    public void run() {
        try (SocketChannel serverChannel = SocketChannel.open()) {
            serverChannel.configureBlocking(false);
            serverChannel.connect(new InetSocketAddress("dase.550w.host", port: 80));

            while (!serverChannel.finishConnect()) {
            }

            ByteBuffer requestBuffer = ByteBuffer.wrap(requestStr.getBytes(StandardCharsets.UTF_8));
            while (requestBuffer.hasRemaining()) {
                serverChannel.write(requestBuffer);
            }
            //serverChannel.shutdownOutput();
            ByteBuffer buffer = ByteBuffer.allocate(capacity: 4096);
            StringBuilder responseBuilder = new StringBuilder();

            while (true) {
                int bytesRead = serverChannel.read(buffer);
                if (bytesRead == -1) {
                    break;
                } else if (bytesRead > 0) {
                    buffer.flip();
                    byte[] data = new byte[bytesRead];
                    buffer.get(data);
                    responseBuilder.append(new String(data, StandardCharsets.UTF_8));
                    buffer.flip();
                    while (buffer.hasRemaining()) {
                        socketChannel.write(buffer);
                    }
                    buffer.clear();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                socketChannel.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

## ➤ 代码优化

为了降低响应时间过长，导致下一次请求无法被处理，网页一直处于缓冲状态，我对代码进行了



优化:

首先,我新增了响应时间阈值 `RESPONSE_TIME_THRESHOLD_MS`, 设置为 10 秒。在处理响应时增加了时间检查, 如果响应时间超过阈值, 将中断读取过程。避免长时间等待无响应的情况, 提高系统的稳定性和响应速度。

其次,我在结尾使用 `shutdownOutput` 函数, 及时关闭输出流, 减少资源消耗。

## ■ 代码展示

```
class TranspondTask implements Runnable { 2个用法
    private final SocketChannel socketChannel; 3个用法
    private final String requestStr; 2个用法
    // 响应时间阈值为10秒
    private final long RESPONSE_TIME_THRESHOLD_MS = 10000; 1个用法
```

... ..

```
StringBuilder responseBuilder = new StringBuilder();

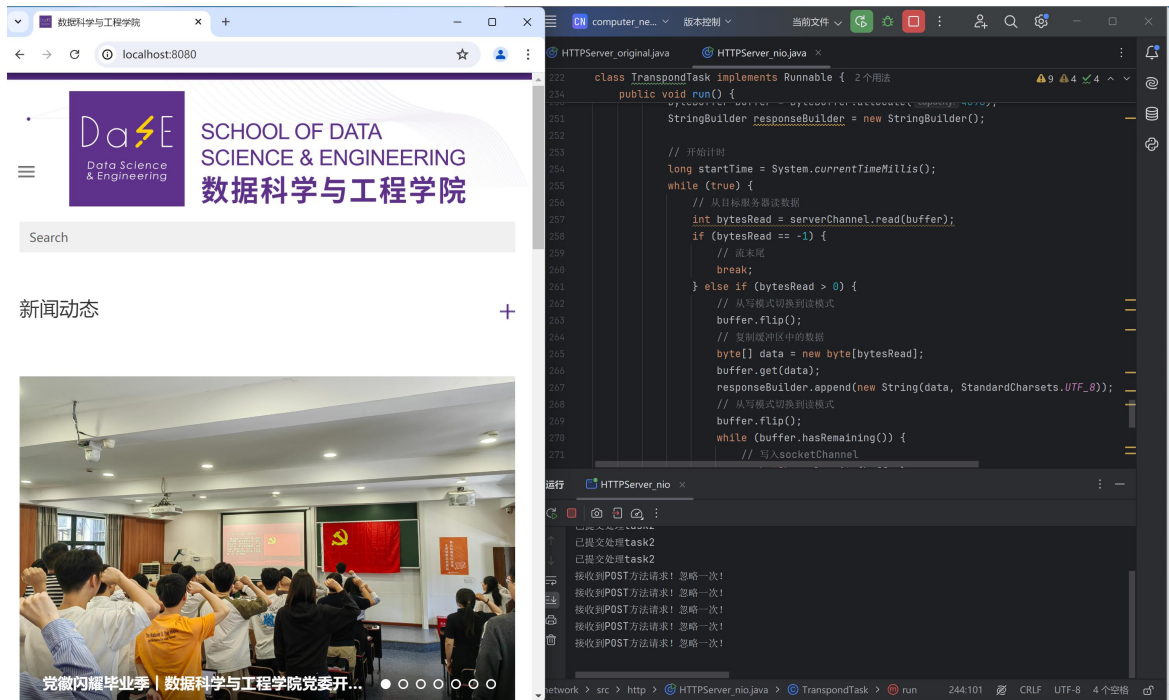
// 开始计时
long startTime = System.currentTimeMillis();
while (true) {
    // 从目标服务器读数据
    int bytesRead = serverChannel.read(buffer);
    if (bytesRead == -1) {
        // 流末尾
        break;
    } else if (bytesRead > 0) {
        // 从写模式切换到读模式
        buffer.flip();
        // 复制缓冲区中的数据
        byte[] data = new byte[bytesRead];
        buffer.get(data);
        responseBuilder.append(new String(data, StandardCharsets.UTF_8));
        // 从写模式切换到读模式
        buffer.flip();
        while (buffer.hasRemaining()) {
            // 写入socketChannel
            socketChannel.write(buffer);
        }
        // 清除, 以便下一次读取
        buffer.clear();
    }
    // 检查响应时间
    if (System.currentTimeMillis() - startTime > RESPONSE_TIME_THRESHOLD_MS) {
        break;
    }
}

// 关闭serverChannel输出
serverChannel.shutdownOutput();
} catch (IOException e) {
```

## ➤ 结果展示

■ 输入: localhost:8080/

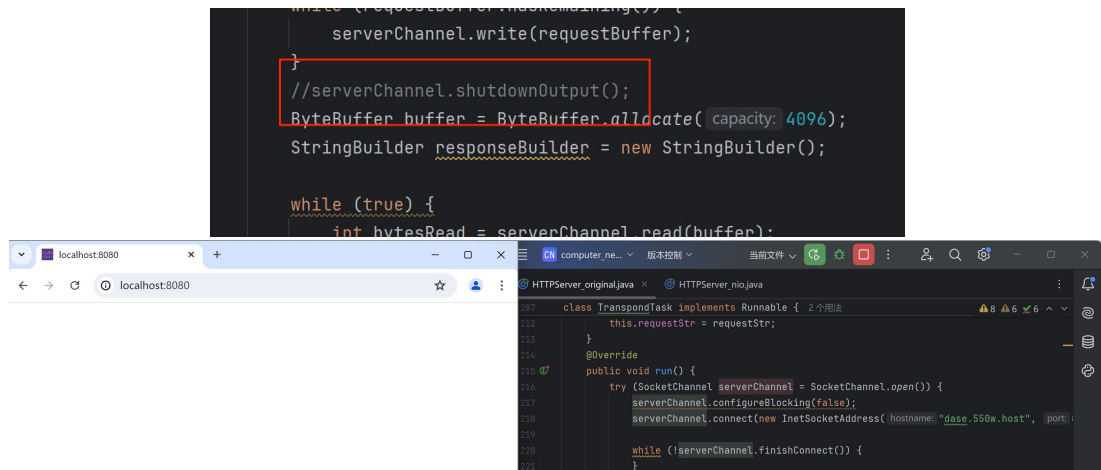
■ 网页显示:

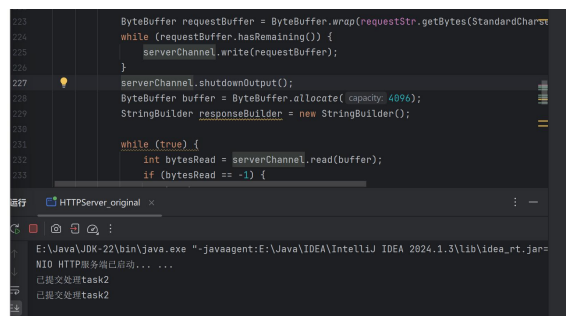


## ➤ 出错反思

一开始, 将 `serverChannel.shutdownOutput()` 写在了服务器发送响应之前, 导致一直显示回写到客户端的响应为空, 无法正常显示网页, 一直加载不出。

解决方法: 注释掉 `serverChannel.shutdownOutput()`, 把它移动到最后传输完目标服务器响应数据之后。





### 3. 合并程序（这部分介绍主类 HTTPServer\_nio）

#### ➤ 思路解析

#### ➤ 声明：

首先声明四个私有成员变量，包括服务器通道，选择器，服务器运行标志，线程池及线程池容量。

这是服务器运行所用到的成员变量的作用范围和生命周期。也直接显示了我的初始解题思路就是使用 NIO 非阻塞模式。

#### ➤ Init () 初始化函数：

接着，对 NIO 服务器进行初始化，打开服务器通道并且打开选择器，利用 configure 设置非阻塞模式，绑定到接收到的 port 端口以及本地 127.0.0.1IP 地址上，设置感兴趣事件为可接受事件。

#### ■ 代码展示

```
public class HTTPServer_nio {
    // 服务器Socket通道
    private ServerSocketChannel server; 9个用法
    // 选择器
    private Selector selector; 9个用法
    // 运行状态标志
    private boolean running = false; 4个用法
    // 初始化线程池（最多15个：14核+1）
    private ExecutorService executor = Executors.newFixedThreadPool(nThreads: 15); 6个用法

    // 初始化非阻塞NIO服务器
    public void init(int port) throws IOException { 1个用法
        // 打开服务器Socket通道，监听新的TCP连接
        this.server = ServerSocketChannel.open();
        // 打开选择器，监视channel的I/O事件
        this.selector = Selector.open();
        // 服务器通道配置为非阻塞模式
        // 非阻塞时，I/O立即返回，不阻塞线程
        // 服务器可以同时处理多个连接，不用等待单个I/O操作直到完成
        server.configureBlocking(false);
        // 服务器通道绑定到8080，监听端口上的连接请求
        // InetAddress: (IP地址, 端口号)的Socket地址实现类
        server.bind(new InetAddress(port));
        // 服务器通道注册到选择器上，指定感兴趣的事件类型为接受连接事件: SelectionKey.OP_ACCEPT
        server.register(selector, SelectionKey.OP_ACCEPT);
    }
}
```

## ➤ Start()启动函数:

直接使用 lambda 实现 runnable 接口，这是服务器的启动步骤，线程池多线程处理了 main 任务：等待连接的 I/O 事件，接着利用迭代器循环遍历可连接的客户端通道，并立刻注册为可读事件，方便进入可读判断中，读取客户端通道中的数据并对 request 进行处理，这是就会调用请求处理函数 requestsolver()，要及时将遍历过的建从迭代器中移除。最后把 main 线程任务交给线程池处理。

## ■ 代码展示

```
// 启动非阻塞NIO服务器
public void start() throws IOException{
    if (server == null || selector == null) {
        // 未初始化
        throw new RuntimeException("服务器未初始化");
    }
    if (running) {
        // 已经运行
        throw new RuntimeException("重复启动服务器");
    }

    // 服务器开始运行
    this.running = true;
    System.out.println("NIO HTTP服务端已启动...");

    // Lambda表达式实现Runnable接口
    Runnable main = () -> {
        while (running) {
            try {
                // 阻塞等待I/O事件
                selector.select();
                // 创建选择器中准备好的通道的键集
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                // 创建键集的迭代器
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while (iterator.hasNext()) {
                    SelectionKey current = iterator.next();

                    // 可接受新的连接
                    if (current.isAcceptable()) {
                        // 接受新的连接并创建新的Socket通道
                        SocketChannel channel = server.accept();
                        //System.out.println("连接成功" + channel.getRemoteAddress());
                        // 新通道配置为非阻塞模式，注册到选择器上，指定感兴趣读操作
                        channel.configureBlocking(false);
                        channel.register(selector, SelectionKey.OP_READ);
                    }
                    // 可读
                    if (current.isReadable()) {
                        // 将通道转换为SocketChannel，处理读操作
                        SocketChannel channel = (SocketChannel) current.channel();
                        requestSolver(channel);
                    }
                    // 从键集中移除当前键
                    iterator.remove();
                }
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    };
    // 主任务给executor执行
    executor.execute(main);
}
```

### ➤ RequestSolver()请求处理函数:

这个函数直接对想要合并的两个功能进行分类处理，对从通道中读取到的请求进行字符串转化后，先排除读取字符为空现象；然后将请求字符串按空格分裂，假设的到的分裂字符串数小于 2，证明请求出错；正常情况下，分裂后的第一个字符串是方法，因为题干要求 POST 方法不处理，所以这边还要事先判断跳过 POST 方法；分裂后的第二个字符串是路径，如果路径为 /index.html 就会创建第一个 RequestTask 类的任务，即响应我的学号和姓名信息；其他情况就会创建第二个 TranspondTask 任务，即进行反向代理请求，这些任务的处理都会交给线程池完成。

### ■ 代码展示

```
public void requestSolver(SocketChannel channel) { 1个用法
    try {
        // 分配非直接缓冲区
        ByteBuffer byteBuffer = ByteBuffer.allocate(capacity: 1024);
        // 创建拼接字符串实例
        StringBuilder builder = new StringBuilder();

        // 从通道中读取数据到byteBuffer
        while (channel.read(byteBuffer) > 0) {
            // 缓冲区从写模式切换到读模式
            byteBuffer.flip();
            // byteBuffer中的字节解码为字符串，给StringBuilder
            builder.append(StandardCharsets.UTF_8.decode(byteBuffer));
            // 清除缓冲区
            byteBuffer.clear();
        }
        String requestStr = builder.toString();

        // 移除请求字符串首尾的空白字符后是空的
        if (requestStr.trim().isEmpty()) {
            return;
        }
        // 请求字符串按空格分裂
        String[] mainArgs = requestStr.split(regex: " ");
        if (mainArgs.length < 2) {
            System.out.println("异常请求，忽略！");
            return;
        }
        // 获取方法，并判断POST
        String method = mainArgs[0];
        if (method.equals("POST")) {
            System.out.println("接收到POST方法请求！忽略一次！");
            return;
        }
        // 获取路径，并分别处理
        String path = mainArgs[1];
        if (path.equals("/index.html")) {
            RequestTask task = new RequestTask(channel);
            executor.execute(task);
        } else {
            String[] lines = requestStr.split(regex: "\r\n");
            StringBuilder modifiedRequest = new StringBuilder();
        }
    }
}
```

```

        for (String line : lines) {
            if (line.startsWith("Host: ")) {
                line = "Host: dase.550w.host";
            }
            modifiedRequest.append(line).append("\r\n");
        }
        modifiedRequest.append("\r\n");

        TranspondTask task = new TranspondTask(channel, modifiedRequest.toString());
        System.out.println("已提交处理task2");
        executor.execute(task);
    }
} catch (IOException e) {
    throw new RuntimeException(e);
}
}

```

### ➤ close()服务器终止函数&main()主函数:

这个函数是配合主函数一起使用的，倘若，主函数中，创建服务器、初始化、启动的三部曲中任何一步出错，就会在 trycatch 块抛出异常后，调用 close()进行服务器终止，及时关闭资源。

#### ■ 代码展示

```

private void close() { 1个用法
    // 停止服务器的主循环
    running = false;
    // 关闭服务器通道
    if (server != null && server.isOpen()) {
        try {
            server.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    // 关闭选择器
    if (selector != null && selector.isOpen()) {
        try {
            selector.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    // 关闭线程池，不接受新任务，等提交的任务完成
    if (executor != null && !executor.isShutdown()) {
        executor.shutdown();
    }
}

public static void main(String[] args) {
    HTTPServer_nio server = new HTTPServer_nio();
    try {
        server.init(port: 8080);
        server.start();
    } catch (IOException e) {
        e.printStackTrace();
        server.close();
    }
}

```

## 三、性能测试情况

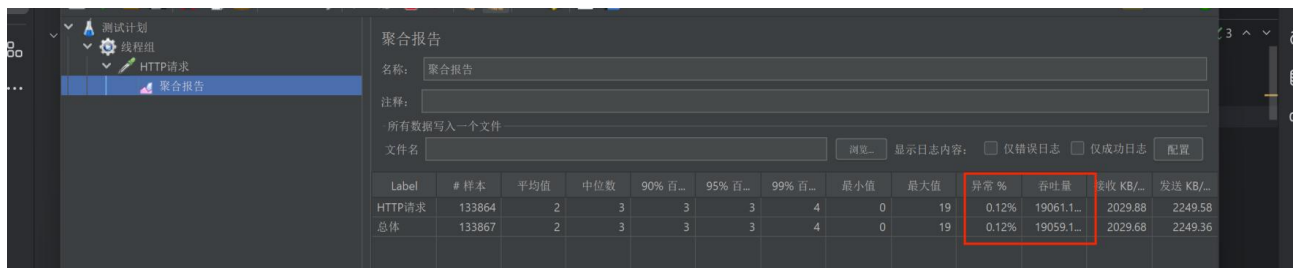
### ➤ 操作流程



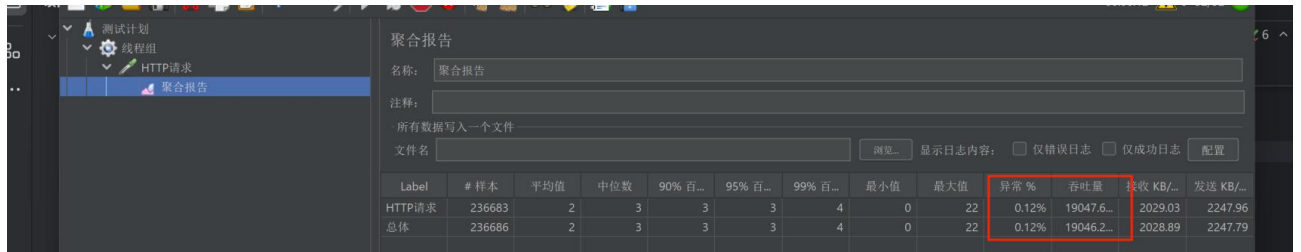




- 保证正确率情况下，线程数量设置为 52，测试时，吞吐量稳定在 19000/sec 以上：

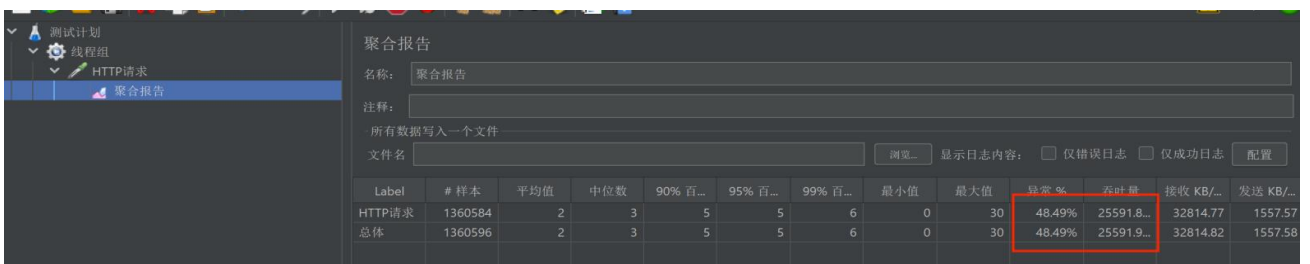


Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 KB/...	发送 KB/...
HTTP请求	133864	2	3	3	3	4	0	19	0.12%	19061.1...	2029.88	2249.58
总体	133867	2	3	3	3	4	0	19	0.12%	19059.1...	2029.68	2249.36



Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 KB/...	发送 KB/...
HTTP请求	236683	2	3	3	3	4	0	22	0.12%	19047.6...	2029.03	2247.96
总体	236686	2	3	3	3	4	0	22	0.12%	19046.2...	2028.89	2247.79

- 在增大线程数为 70，目标为 25000/sec 情况下，错误率较高，大概一般正常，一半异常：



Label	# 样本	平均值	中位数	90% 百...	95% 百...	99% 百...	最小值	最大值	异常 %	吞吐量	接收 KB/...	发送 KB/...
HTTP请求	1360584	2	3	5	5	6	0	30	48.49%	25591.8...	32814.77	1557.57
总体	1360596	2	3	5	5	6	0	30	48.49%	25591.9...	32814.82	1557.58

## 四、总结

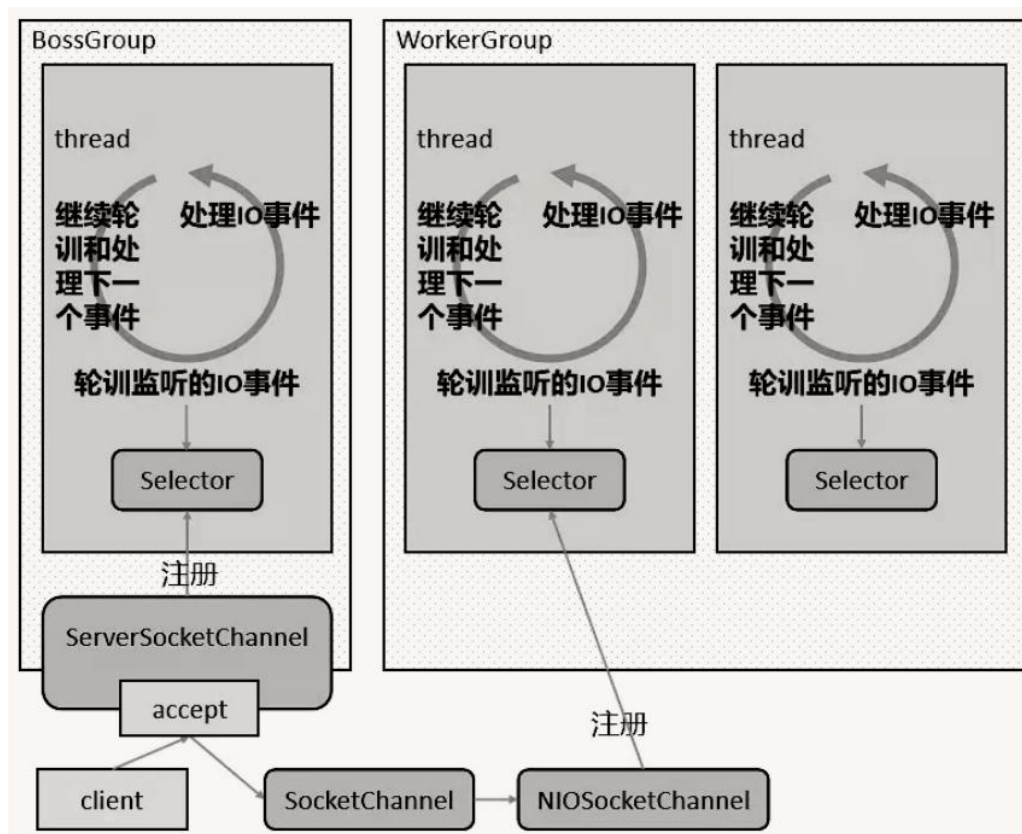
本次期末作业，我在了解了实验内容和要求，浏览了助教给的代码示例之后，直接选择了用 NIO 非阻塞式处理，因为这部分我在之前的实验课，学的有些不牢固，想通过期末作业进一步巩固一下。在这次实验前，我先深入了解了 JAVANIO 实现的原理、类、方法、思路等，明确了它与 IO 的区别：

1. BIO 是面向流的，NIO 是面向缓冲区的。BIO 每次从流中读一个或多个字节，直至读取所有字节，没有被缓存。NIO 数据读取到稍后处理的缓冲区，增加了处理的灵活性。前提是检查该缓冲区中是否包含需要处理的数据，还要确保不覆盖未处理的数据。
2. BIO 是阻塞的，当一个线程调用 read()或 write()时，被阻塞，直到有一些数据被读取，或数据完全写入。NIO 为非阻塞，直至数据可以读取写入之前，该线程是灵活的。因此，多线程时可以在其它通道上执行 BIO 操作。

3. NIO 允许一个单独的线程来监视多个通道，注册多个通道但只使用一个选择器来“选择”通道并对事件进行处理。

同时，在实现简易 HTTP 服务器以及反向代理的过程中，进一步熟悉了 nio 框架以及整体的流程代码如何编写，也帮我复习了多线程相关的代码操作，包括 runnable 接口，lambda 表达式简易创建线程等等，另外，在对类的成员变量进行声明时，我一度搞混了关键词的作用，查阅资料后，才搞清楚他们之间排列组合的作用。

另外，在完成本次实验时，规定不能使用外部库进行处理之前，我初始计划使用 netty 框架，因为它是基于 NIO 的一个异步事件驱动的网络应用框架，能够快速开发可维护的高性能协议服务器和客户端，大致流程如下：



我会在课后，使用 netty 框架再进行一次代码实现，争取能够实现功能，提升一下自己的实践能力。

最后，本次实验十分具有挑战性，但是对我来说，意义很大，因为我之前一直片面的认为大学学习知识都偏理论，没有落地的实际感，但是计网这门学科真的让我改变了这个想法，借助 JAVA 这门语

言，我现在已经可以实现简单的 HTTP 服务器了，在看到网页能够正常显示和被代理时，内心很喜悦。我记得老师和学长也说，可以自己试着实现一个简单的聊天功能，这个我接下来暑假空闲时，也会去尝试一下。大作业实现过程中，我也遇到了一些困难，中间也请教了学长一些问题，非常感谢两位学长不厌其烦地帮助我解决计网实验课中的问题，也感谢老师的倾囊相授，这门课我的收获颇丰，谢谢！