Project2 chatting Room

1 实验内容

实现一个多客户端的纯文本聊天服务器，能同时接受多个客户端的连接，并将任意一个客户端发送的文本向所有客户端（包括发送方）转发。

2 构建方法

本项目使用了 **jdk-17.0.5**，在 Ubuntu22.04 上，本项目的构建方法如下（以 `Client` 的构建为例）：

```
$ cd client
$ mkdir out
$ javac Client.java -d out
$ cd out
$ touch MAINFEST.MF
```

向 `MAINFEST.MF` 中写入以下内容：

```
Manifest-Version: 1.0
Main-Class: Client
```

继续在 `out` 目录下执行以下命令：

```
$ jar cfvm client.jar MAINFEST.MF ./Client.class
```

类似地，`Server` 的构建不再赘述。

3 设计架构

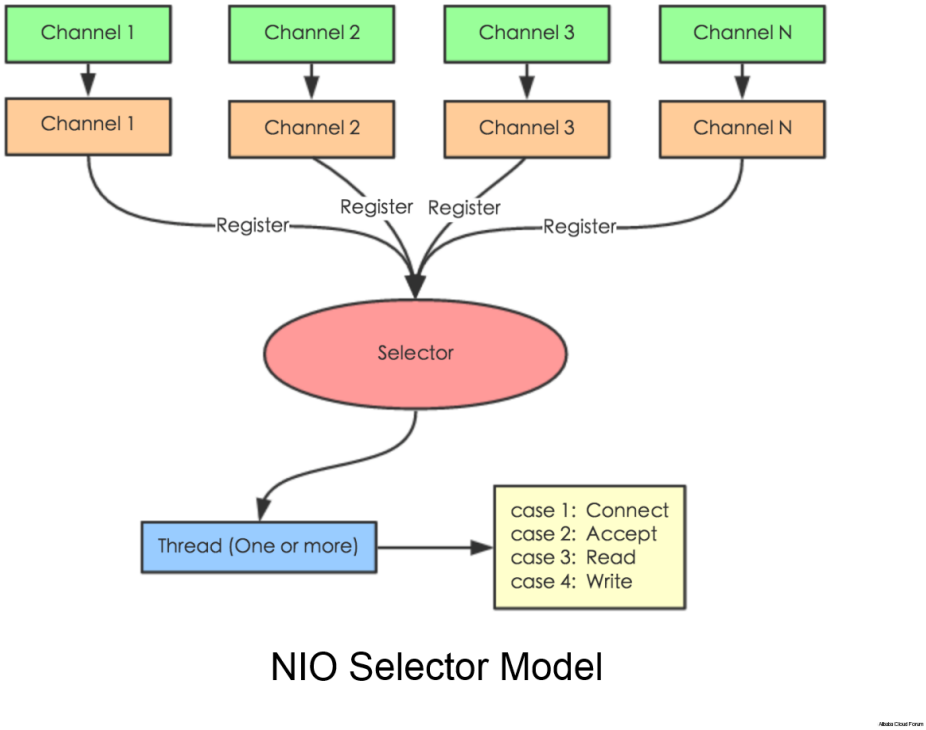
Java 实现聊天室有如下几个模式：

- **BIO 模式：**每新加一个客户端，服务端就新加一个线程
- **线程池：**基于 BIO，但规定了服务端可以打开的线程上限
- **NIO 模式：**相比于 BIO 新增了 Channel、Selector、Buffer 等抽象概念，支持面向缓冲、基于通道的 I/O 操作方法
- **AIO 模式：**采用订阅-通知模式，即应用程序向操作系统注册IO监听，然后继续做自己的事情。当操作系统发生IO事件，并且准备好数据后，在主动通知应用程序，触发相应的函数。

我们先分别分析上面几种模式的利弊：

- BIO 模式较为简单，但是在客户端很多时，服务端和客户端的线程呈线性增加，这样在客户端很多时，服务端的线程数会很多，消耗过多资源。
- 线程池是对 BIO 模式的一个约束，即约束了服务端最大可以打开的线程数，但这样做的弊端是一旦服务端达到最大可开线程数，后续的客户端就需要等待直到服务端空闲才能连接。
- NIO 则重构了原有 IO 的模型，将原先 BIO 的一**个请求一个线程**改成了一个连接一个线程，即客户端发送的连接请求都会注册到 `Selector` 上，`Selector` 轮询到连接有 I/O 请求时才启动一个线程进行处理。
- AIO 是对 NIO 的升级，服务器实现模式为一**个有效请求一个线程**，客户端的 I/O 请求都是由 OS 先完成了再通知服务器应用去启动线程进行处理。

综合考虑下，我选择了 **NIO 模式**应用于本次聊天室的开发。下面是 NIO 模式下，典型的客户端和服务端交互的架构图：



NIO Selector Model

Alibaba Cloud Forum

3.1 Server

由于我的整体设计基于 NIO，所以服务端的设计为以下几个部分：

- 首先初始化注册 `selector` 作为服务端的多路选择器
- 初始化 `ServerSocketChannel`，打开、绑定端口、**设置为非阻塞**
- 服务端刚启动，感兴趣的事件是 `OP_ACCEPT`，即接受客户端的连接
- 用一个死循环作为服务端的主线程，死循环中轮询所有的 `SelectionKey`，即事件，对不同的事件做出不同的回应。
- 事件分为以下两类：
  - `isAcceptable()`：表示现在的 Channel 的事件是接受客户端的连接。服务端需要回应客户端的连接，储存客户端的信息，同时将该 Channel 感兴趣的事件设置为 `OP_READ`，即希望接受客户端发来的信息
  - `isReadable()`：表示现在的 Channel 的事件是读取客户端发来的信息。服务端需要正确的读取客户端发来的信息并且将其转发给所有的客户端，同时将该 Channel 感兴趣的事件设置为 `OP_READ`，表示继续接受客户端发送的信息

3.2 Client

客户端和服务端的整体结构类似：

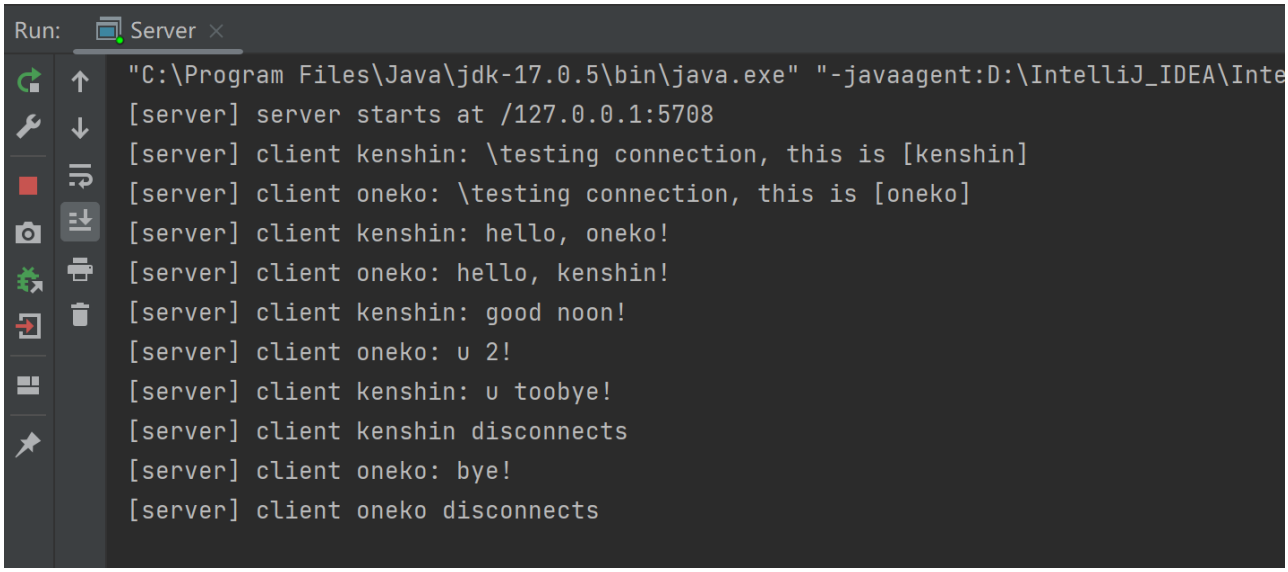
- 首先初始化注册 `selector` 作为客户端的多路选择器
- 初始化 `ServerSocketChannel`，打开、绑定端口、**设置为非阻塞**
- 客户端刚启动，感兴趣的事件是 `OP_CONNECT`，即希望连接服务器

- 用一个死循环作为客户端的主线程，死循环中轮询所有的 `SelectionKey`，对不同的事件做出不同的回应。
- 事件分为两类：
  - `isConnectable()`：表示现在的 Channel 的事件是正在和服务器进行连接。我们需要判断是否和服务器已经连接上，若连接上，则向服务器发送一条测试消息，告诉服务器连接成功和客户端的名字，同时将该 Channel 感兴趣的事件设置为 `OP_READ`，表示客户端希望接收到来自服务端的消息。注意在连接成功后，客户端新开一个线程用于发送消息。
  - `isReadable()`：表示现在的 Channel 的事件是读取服务端发来的消息。直接对消息进行解码并打印即可。

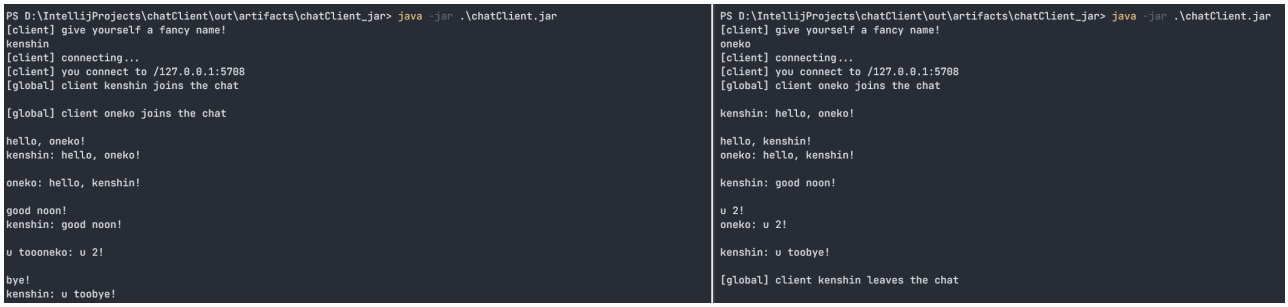
4 效果演示

为了测试方便，ip 地址均为回环地址。

4.1 服务器启动/转发



4.2 客户端启动/收发消息



5 压力测试

压力测试在 windows11 上进行，通过命令行打开多个客户端连接同一个服务器进行压力测试。首先我们需要进行一些准备工作：

5.1 修改 `chatClient.java`

修改代码让其发送消息的线程为每 0.1s 发送一条，即 1s 发送 10 条。

```
//                                  BufferedReader bufReader = new BufferedReader(new
//                                          InputStreamReader(System.in));
//                                  ByteBuffer encodedMsg = Charset.forName("GBK").encode(bufReader.readLine());

                                    ByteBuffer encodedMsg = Charset.forName("GBK").encode("test");
                                    TimeUnit.MILLISECONDS.sleep(100);
                                    sendBuf.put(encodedMsg);
```

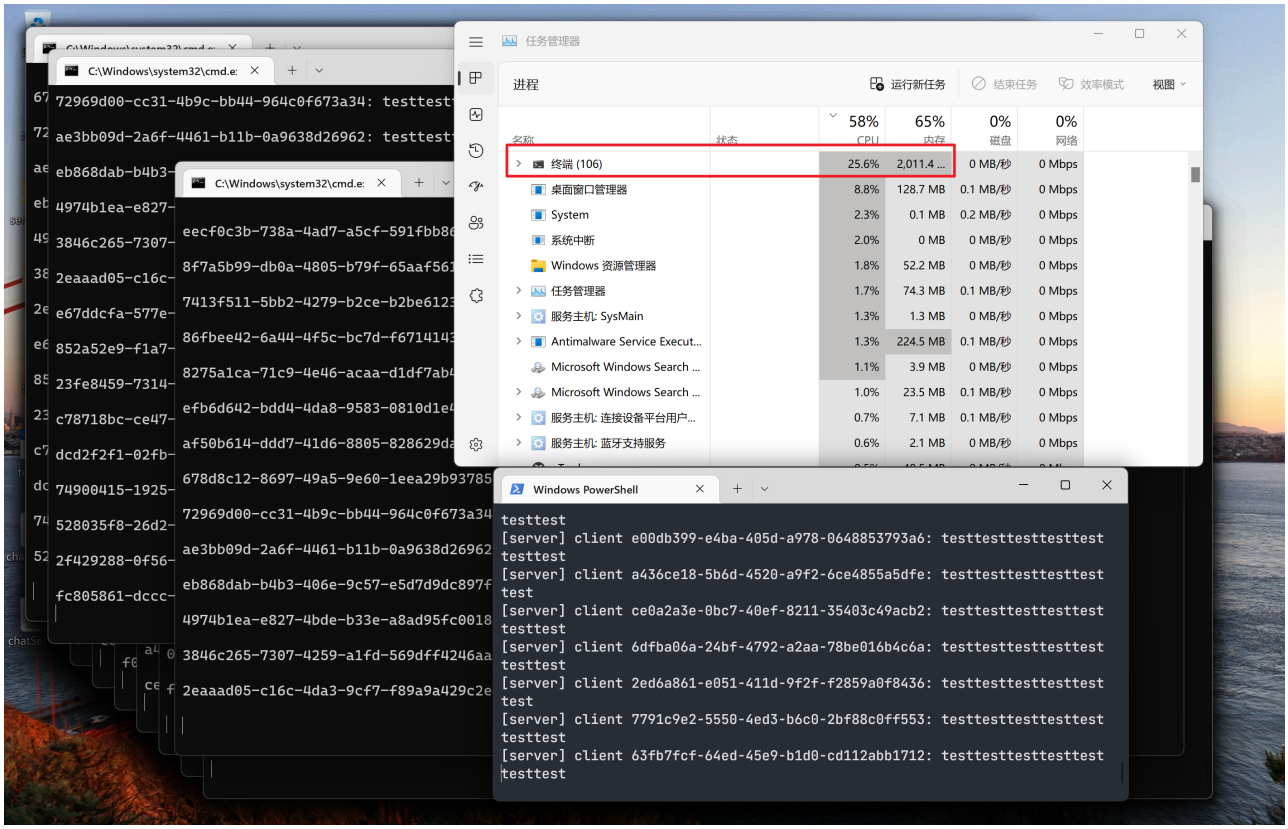
5.2 自动化测试脚本

自动化测试脚本是通过打开一个 cmd 并启动客户端，如此循环打开 50 个：

```
#include <iostream>
#include <windows.h>

int main()
{
    for (int i = 0; i < 50; i++)
    {
        system("start cmd /k \"java -jar ./chatClient.jar\"");
        Sleep(50);
    }
}
```

5.3 测试结果



50 个客户端成功启动且没有任何异常产生，在每秒共转发 25000 条消息的压力下，服务器仍然正常运行，且 cpu 占用在 25% 左右，成功通过了这个粗糙的压力测试。

PS: 客户端太多时，JVM 的内存首先会撑不住