

基于Socket的P2P聊天软件——CommuSys

总体介绍

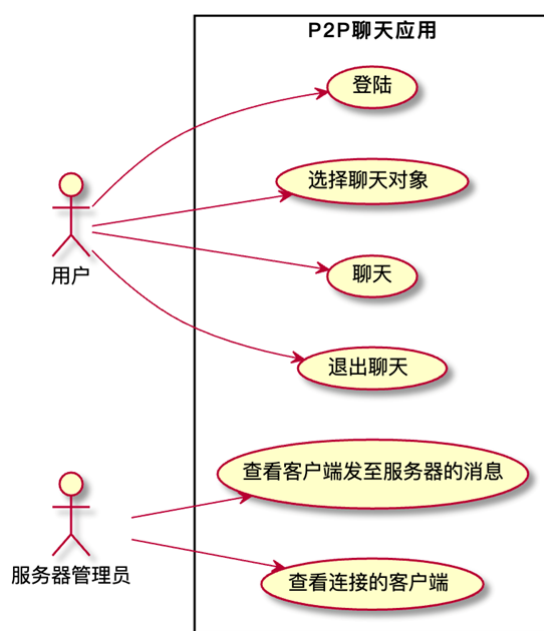
- 本项目是一个基于**C/S架构**的，基于服务器的，客户端间聊天软件
- 可以实现多个客户端间的互相通信，采用命令行的信息输入、展示方式
- 自定义了“**应用层协议**”，实现了客户端间的连接、断开与服务器的监听等功能
- 使用的是JAVA下的线程及**Socket编程**（基于TCP）

环境搭建

- JDK 16
- IDEA

需求分析

- CommuSys是一个轻量级的即时点对点的聊天系统，用户与用户可以通过服务器的消息转发互相发送消息。可以被用于公司/学校/团队内部私密交流等，也可以拓展至微信类似的大众社交软件
- 系统主要的用户（角色）主要有：**用户，服务器管理员**



功能描述

- 登陆：用户在启动客户端时直接输入自己的用户名即可登录（由于是轻量级即时聊天系统，不设置注册等繁杂流程）
- 选择聊天对象：用户可在软件中主动选择聊天对象（通过命令行/GUI界面）
- 聊天：用户直接输入聊天内容即可向之前选择的聊天对象发送信息
- 退出聊天：用户在结束聊天后可以选择退出
- 查看客户端发至服务器的消息：客户端之间的聊天通过服务器转发，所以服务器管理员可以查看客户端之间的消息记录
- 查看连接的客户端：服务器管理员可以在服务器端查看实时客户端的连接情况
- 当用户在本系统内的各种输入不符合要求规范的时候，均不会引起系统的故障，并能提示用户错误操作。

系统设计

- **消息的种类**：将消息的种类分成三种，分别为CONNECT，CONTENT，DISCONNECT

每种消息的作用：

CONNECT消息：代表的是一条客户端请求与服务器连接的消息，其中包含了连接客户端的用户名

CONTENT消息：代表的是内容消息，消息之中包含了具体的聊天内容，以及源用户名和目的用户名

DISCONNECT消息：代表的是取消连接消息，其中包含了取消连接客户端的用户名

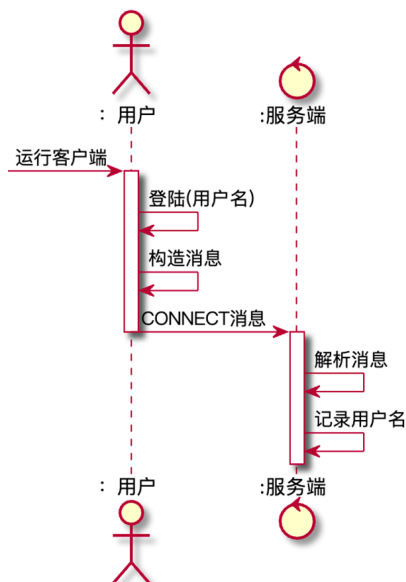
- **消息格式定义：**

当消息类型为CONNECT，DISCONNECT，格式为：消息类型 + ### + 源客户端用户名

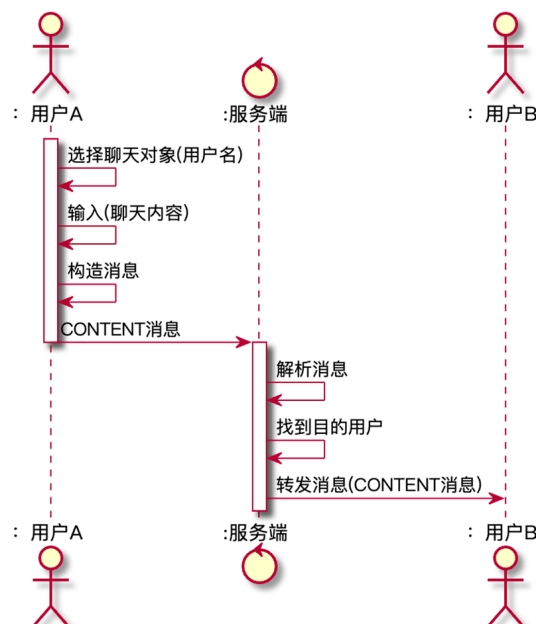
当消息类型为CONTENT，格式为：消息类型 + ### + 源客户端用户名 + ### + 目的客户端用户名 + ### + 消息内容

- **消息的交互过程**

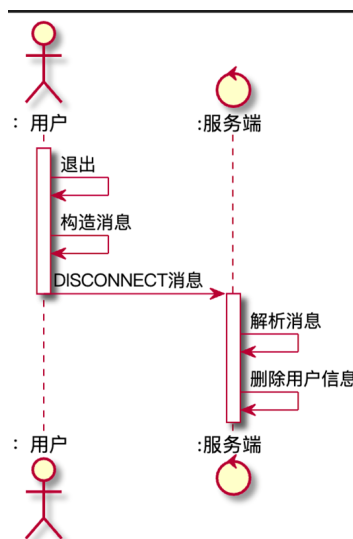
- 用户登录



- 用户聊天



- 用户退出（中断连接）



• 通信机制及模块组成

- 传输层协议：TCP
- 服务器的连接套接字选用：ServerSocketChannel配合多路复用Selector选择器，非阻塞模式，采取select方式。
- 客户端采用和服务端相同的NIO架构。
- 通过多路复用Selector选择器的select机制保证服务器端的资源得到了充分的利用，只需要首先将感兴趣的事件注册到selector，之后selector会自动帮助关注这些事件，不需要通过轮询或者多线程关注事件。并且可以通过selector的不同事件类型定制相应的解决方法。

• 模块介绍

- Message类：主要负责消息的编码，解码
- Server_table类：包含了服务器端需要使用的各种表，数据结构
- ServerHandlerBs接口：服务器处理事件的接口，包含了三个方法，分别处理接受连接，读，写，方便之后对于事件处理的定制
- ServerHandlerImpl类：继承ServerHandlerBs接口，定制其中的方法
- NioSocketServer类：服务端类，其中包含了NIO的select机制
- ClientHandlerBs接口：客户端处理事件的接口，包含了两个方法，分别处理读，写，方便之后对于事件处理的定制
- ClientHandlerImpl类：继承ServerHandlerBs接口，定制其中的方法
- NioSocketClient类：客户端类，其中包含了NIO的select机制

具体实现

• Message类

- 定义了三个静态final变量代表消息的类型，分别为0 1 2，之后构造消息时可以直接用变量表示，增强可读性。之后定义了两种构造函数，分别针对的是CONTENT类型的信息与其他两种类型
- 定义编码静态函数encode，根据消息类型的不同，将Message类的对象转化为不同的字符串。当消息类型为CONNECT，DISCONNECT，格式为：消息类型 + ### + 源客户端用户名。当消息类型为CONTENT，格式为：消息类型 + ### + 源客户端用户名 + ### + 目的客户端用户名 + ### + 消息内容

- ```

case TYPE_CONNECT, TYPE_DISCONNECT -> {
 str=message.getType()+"###"+message.getFrom();
}
case TYPE_CONTENT -> {
 str=
message.getType()+"###"+message.getTo()+"###"+message.getFrom()+"###"+me
ssage.getContent();
}

```

- 定义解码函数decode，将字符串重新转化为Message类的对象，其中采用了StringTokenizer类从字符串之中获取有效信息

- ```

case TYPE_CONNECT, TYPE_DISCONNECT -> {
    String from = stringTokenizer.nextToken();
    message1=new Message(type,from);
}
case TYPE_CONTENT -> {
    String to = stringTokenizer.nextToken();
    String from = stringTokenizer.nextToken();
    String content = stringTokenizer.nextToken();
    message1=new Message(type,from,to,content);
}
default -> throw new IllegalStateException("Unexpected value: " + type);

```

• Server_table类

服务器需要使用信息存储，之后可以拓展为和数据库交互的类并且增加更多的存储功能（比如聊天记录）。在其中目前定义了一个ConcurrentHashMap（支持高并发的哈希表），用于存储用户名-socketchannel的映射。connected表用于记录用户名与连接套接字的一对一关系，通过用户名就能找到相应的套接字，这对于消息的转发十分重要，由于在之后可能可能会有多个进程/线程同时读取修改表，所以需要支持高并发。

• ServerHandlerImpl类

- 继承ServerHandlerBs，实现了其中的方法
- handleAccept：首先获取连接的socketchannel，之后将该套接字改为非阻塞模式，最后再将对于该套接字的读事件注册到selector中以便之后的读数据
- handleRead：首先先读取Socketchannel中的数据，将其转化为Message类型。针对读到的消息类型作出不同的动作：
 - CONNECT类型：输出用户连接，并将相应的用户名与套接字加入connected表之中
 - CONTENT类型：先判断目的用户是否在connected表中，如果在则通过表中查到的socketchannel转发消息。如果不在判断目标用户是否为server，如果不为server，则需要构造一条用户不存在的信息，向源用户发送，提醒用户目标用户不存在。
 - DISCONNECT类型：关闭相应的socketchannel，输出连接断开，从connected表中删除相关的表项

- ```

case Message.TYPE_DISCONNECT -> {
 socketChannel.shutdownOutput();
 socketChannel.shutdownInput();
 socketChannel.close();
 System.out.println("连接断开.....");
 returnStr=message.getFrom()+"断开";
 Server_table.connected.remove(message.getFrom());
 return returnStr;
}

```

- **NioSocketServer类**

- 服务端的实现类

- start为服务器端的启动方法，在其中首先通过ServerSocketChannel类的open方法获取对象，之后将其绑定到8888端口，设置非阻塞模式，为serverChannel注册selector，创建服务器端事件处理器。之后新开了一个线程用于接收服务器管理员的输入

- ```

try (ServerSocketChannel serverSocketChannel =
    ServerSocketChannel.open()) {
    serverSocketChannel.socket().bind(new InetSocketAddress(8888));
    //设置为非阻塞模式
    serverSocketChannel.configureBlocking(false);
    //为serverChannel注册selector
    Selector selector = Selector.open();
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
    System.out.println("服务端开始工作：");
    ....
}

```

- 随后进入到select机制的核心，通过select方法获取一组已注册的就绪的channels（如果没有的话就阻塞在select那里）

- ```

while (flag == 1) {
 selector.select();
 System.out.println("开始处理请求：");
 //获取selectionKeys并处理
 Iterator<SelectionKey> keyIterator =
 selector.selectedKeys().iterator();
 ...}

```

- 之后通过迭代器遍历所有就绪的channels，对于每个channel相应的事件执行相应的handler函数

- **ClientHandlerImpl类**

- 继承ClientHandlerBs，实现了其中的方法
- handleRead：比服务端相对简单，在这里直接读取消息解码输出即可
- handleWrite：首先判断是否已经给服务器端发送用户名了，如果没有则构造CONNECT消息向服务器发送。之后判断是否有未处理的用户输入（用户输入都被存在Input\_cache之中），如果有则取出用户输入，如果没有直接返回。下一步判断用户输入是否为exit，如果为exit，则构造DISCONNECT消息发送至服务器端，之后关闭channel，打印断开连接。

```

if(Objects.equals(request, "exit")){
 Message message =new Message(Message.TYPE_DISCONNECT,from);
 writeBuffer.clear();

 writeBuffer.put(Message.encode(message).getBytes(StandardCharsets.UTF_8)
);
 writeBuffer.flip();
 socketChannel.write(writeBuffer);
 socketChannel.shutdownOutput();
 socketChannel.shutdownInput();
 socketChannel.close();
 System.out.println("连接断开.....");
 return;
}

```

最后判断是否为指定目的用户的命令，如果是则修改相关变量，返回。

随后判断是否已经指定消息的接收者了，如果没有则提示用户未指定接收者，返回

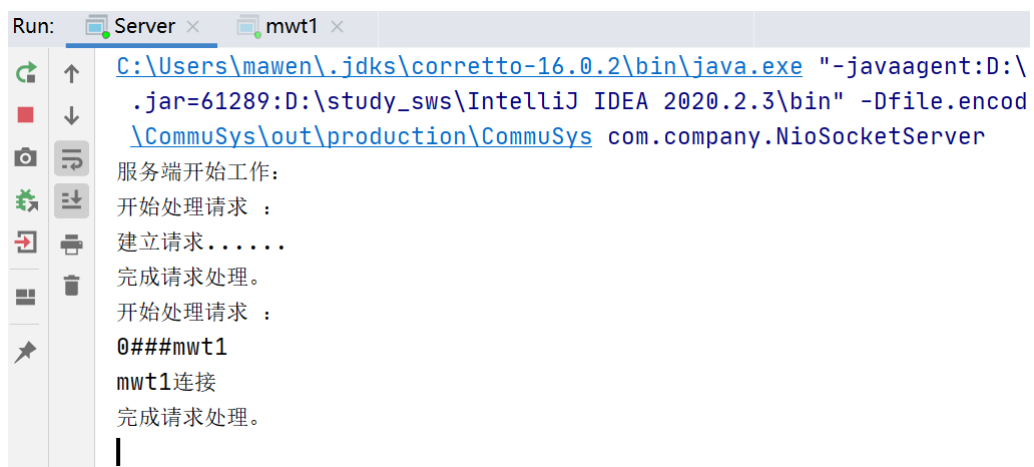
至此，可以认为用户输入为聊天内容，直接构造CONTENT消息通过channel发送至服务端即可

#### • NioSocketClient类

- start为客户器端的启动方法，在其中首先通过SocketChannel类的open方法获取一个channel，之后建立一个ip为localhost端口为8888的地址，将channel通过connect函数与这个地址建立连接，将channel设置非阻塞模式，为serverChannel注册selector，创建客户端事件处理器。
- 之后新开了一个线程用于接收用户的输入，将用户的输入添加进Input\_cache队列，方便之后的处理
- 与服务器相同，进入到select机制的核心，通过select方法获取一组已注册的就绪的channels（如果没有的话就阻塞在select那里）之后通过迭代器遍历所有就绪的channels，对于每个channel相应的事件执行相应的handler函数

## 效果展示

- 创建两个客户端，分别命名为mwt1和mwt2，创建服务器，启动系统。可以看到mwt1启动时服务器显示如下信息：



- Client1不指定发送消息的接收者直接输入消息发送，默认服务器接收。当mwt1默认发送命令hello时，可以看到服务器如下所示：

```
Run: Server x mwt1 x mwt2 x
开始处理请求 :
1###mwt1###server###hello
mwt1:hello to:server
完成请求处理。
```

- 之后再mwt1输入“to/mwt2”完成peer选择，重新发送hello，可以看到服务器如下所示：

```
Run: Server x mwt1 x mwt2 x
开始处理请求 :
1###mwt1###mwt2###hello
mwt1:hello to:mwt2
完成请求处理。
```

可以看到mwt2显示如下内容：

```
Run: Server x mwt1 x mwt2 x
C:\Users\mawen\.jdk\corretto-16.0.2\bin\java -jar 61364:D:\study_sws\IntelliJ IDEA\Commusys\out\production\Commusys com
客户端开始工作:
mwt1:hello to:mwt2

|
```

- 之后再mwt1中输入多条信息，mwt2如下所示：

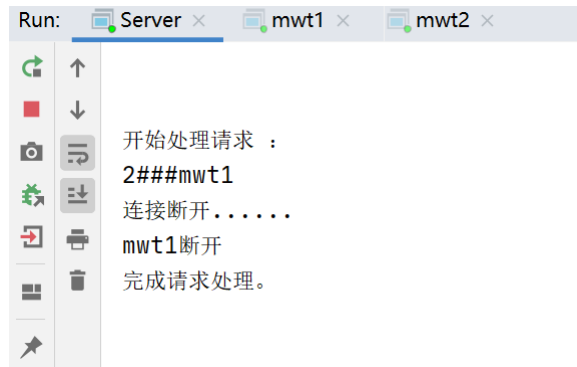
```
Run: Server x mwt1 x mwt2 x
mwt1:hi mwt2 to:mwt2
mwt1:i have alot to say to:mwt2
mwt1:can you hear me? to:mwt2
```

这标志着本系统可以完整支持消息的快速发送

- 在服务器端输入“client”查询在线客户端：

```
Run: Server x mwt1 x mwt2 x
client
mwt1
mwt2
```

- 在mwt1中输入“exit”断开连接，服务器端如下所示：



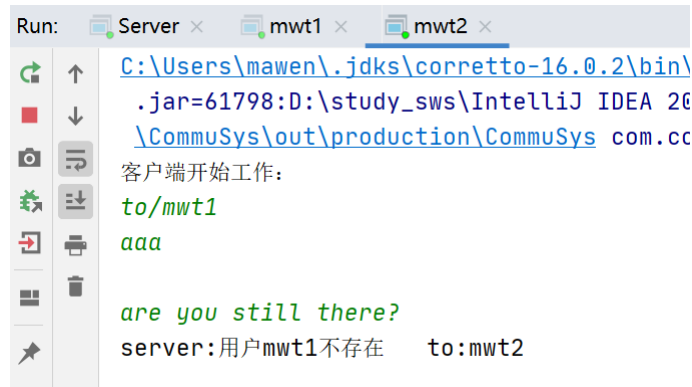
Run: Server x mwt1 x mwt2 x

```

开始处理请求 :
2###mwt1
连接断开.....
mwt1断开
完成请求处理。

```

在mwt2向mwt1发送消息，如下所示：



Run: Server x mwt1 x mwt2 x

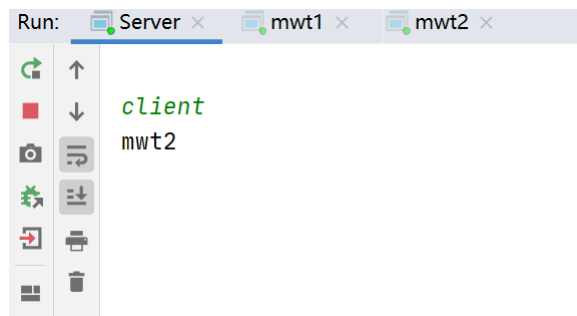
```

C:\Users\mawen\.jdk\corretto-16.0.2\bin\
.jar=61798:D:\study_sws\IntelliJ IDEA 20
\CommuSys\out\production\CommuSys com.cc
客户端开始工作:
to/mwt1
aaa

are you still there?
server:用户mwt1不存在 to:mwt2

```

可以看到报错信息。此时在服务器端输入“client”，如下所示：



Run: Server x mwt1 x mwt2 x

```

client
mwt2

```

## 项目运行方式

- git clone，生成项目
- 在run/debug configuration中创建两个客户端 apps，分别命名：“1”，“2”  
创建一个服务器app
- 运行服务器以及两个客户端apps
- 指令：
  - 客户端
    - 输入“to/NAME”来确定消息发送对象（默认为服务器）
    - 输入任意字符，回车发送
    - 输入“exit”中断连接
  - 服务器
    - 输入“client”查询客户端信息