廣東工業大學

课程设计报告

课程名称	计算机网络
学生学院	计算机学院
专业班级	计算机科学与技术 4 班
指导教师	梁路
学生姓名	黄钰竣
学　　号	3117004568

2019 年 7 月 2 日

目录

一、程序开发基础知识

（一） P2P 原理

1． P2P 的概念定义

P2P 为 peer-to-peer 的简写，“Peer”在英语里有“对等者、伙伴、对端”的意义。因此，从字面上，P2P 可以理解为对等计算或对等网络，在学术界统一称为对等网络（Peer-to-Peer networking）。P2P 是相对于“客户-服务器”（C/S）模式来说的一种网络交换方式。

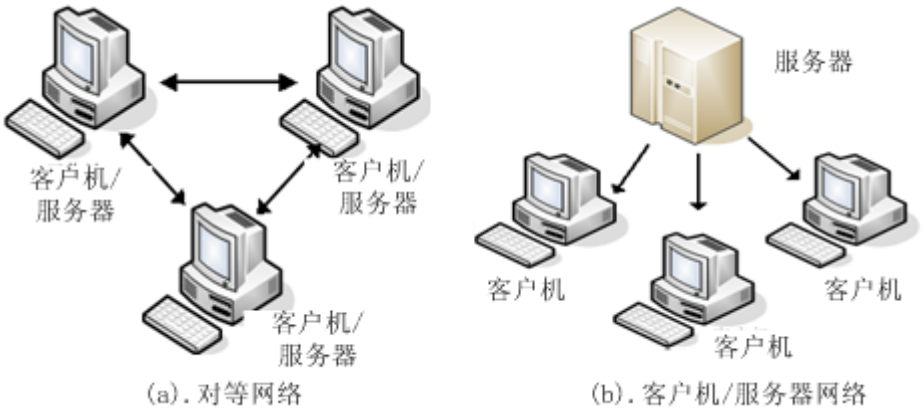
在“客户-服务器”模式中，数据从客户端发送后必须通过专门的服务器进行处理和分发给客户端想发送给的那一方，在这种方式下，逻辑上可以将服务器（组）当作各客户连接的中心节点，各客户都从服务器获得数据，所以服务器也往往需要有较为强大的硬件支持。

P2P 则是相对于“客户-服务器”方式的网络信息交换方式。P2P 主张不加区分服务器和客户的方式进行数据交换，每一台主机既是服务器向连接它的主机提供服务，同时也是服务的请求方，获取其他主机提供的服务。

2． P2P 的优缺点分析

在客户主机数量较小，或者提供服务所需要的总体 CPU 能力、内存大小、网络带宽限制较低的情况下，使用 P2P 的网络信息交换方式可以充分发挥庞大的终端资源。另外，P2P 也是解决“客户-服务器”连接方式的中心化等特点，所面临的一系列可扩展性差、系统健壮较低、负载不均衡等问题的绝佳方式。

但是同时 P2P 连接方式也面临网络安全、影响客户计算机性能以及较难实现数据备份、恢复等统一管理等问题。



3． P2P 对等网络的基本结构

P2P 主要有三种基本的结构分别为集中式对等网络、无结构分布式网络、结构化分布式网络。

集中式对等网络是基于中央目录服务器的，在主机需要向其他主机发送数据的时候，先向目录服务器进行查询服务，获取发送的必要信息，但是传输的内容无需再经过中央服务器。这种方式可以认为是仍带有一点中心色彩的对等网络。Napster、QQ 等软件使用这种结构。

无结构分布式网络则直接通过与相邻节点的通信从而介入网络。它与集中式对等网络的明显区别在于它是完全去中心化的，即不存在向目录服务器获取消息接收方的地址信息，而是通过发送查询包的机制来搜索需要的信息、资源。这种方式类似于获取路由的 OSPF 协议。Gnutella 等软件使用这种结构。

结构化分布式网络，是近几年基于分布式哈希表（Distributed Hash Table）技术的研究成果。它的基本思想是将网络中所有的资源整理成一张巨大的表，表内包含资源的关键字和所存放结点的地址，然后将这张表分割后分别存储到网络中的每一结点中去。当用户在网络中搜索相应的资源时，它将能发现存储与关键词对应的哈希表内容所存放的结点，在该结点中存储了包含所需资源的结点地址，然后发起搜索的结点根据这些地址信息，与对应结点连接并传输资源。这种方式类似与使用一层层的子网将网络进行不断的划分最后找到指定的主机。

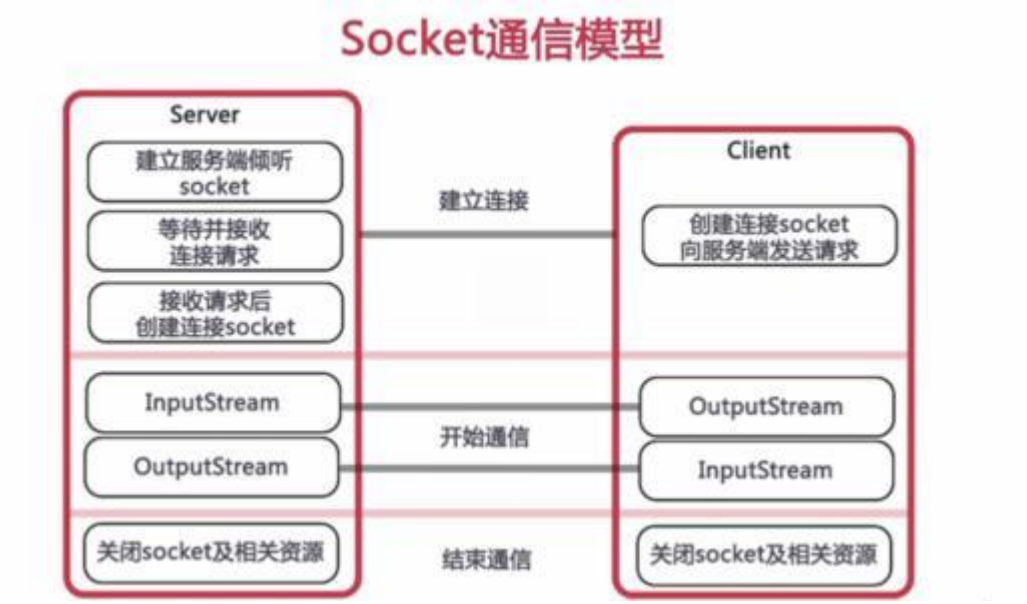
（二） Java 的 socket 原理与 NIO

1． socket 的概念定义

网络上的两个程序通过一个双向的通信连接实现数据的交换，这个连接的一端称为一个 socket。根据教材《计算机网络》的定义可以理解为是一个由“IP 地址”以及“端口号”两个信息组成的通信端口，是 TCP 协议里确定一条 TCP 连接起始端或者终点端的必须要素。但是事实上，Socket 是一个多词义的名词，它还可以指允许网络程序访问连接网络协议的应用编程接口 API（Application Programming Interface），即通过使用编写程序使用该接口定义的方法、函数可以实现网络的通信连接。另外，socket 这个词还可以指 Socket API 里面的一个函数名等。

2． Java 的 socket 原理

Java 的 Socket 原理实际上就是使用 TCP 协议进行网络连接通信，其实质就是使用了 Java 编程语言实现了一系列能够使用 TCP 协议进行通信的程序接口。如图下图即为 socket 的传统 I/O 通信模型。



3． NIO

NIO 是在 JDK1.4 中新加入的类，N 代表 New，IO 分别对应 Input 和 output，即输入输出流。NIO 的新是相对于旧的 I/O 而言的。NIO 引入了一种基于通道和缓冲区的 I/O 方式，相对于传统的 I/O 而言是一种同步非阻塞的 I/O 模型。传统的 I/O 在通信的过程中往往需要单独开辟一个线程出来，用于处理读取和接受连接中的信息，即一条与其他主机的连接通信等价于一个单独线程资源的消耗。而 NIO 则通过使用 Selector 这个能够对各个通道进行轮询 IO 事件（就是连接的通道中有信息发送过来，这个信息根据类别可以分为 Acceptabel 允许接受连接事件、Readable 允许读取连接通道事件），当这些事件发生，即可以被 selector 监测到并交由相应的程序来处理，从而实现了通过一个线程管理多条连接。

4． Java 的 Swing UI 界面程序设计

Swing 是 Java 为图形界面应用开发提供的一组工具包，是 Java 基础类的一部分。Swing 包含了构建图形界面（GUI）的各种组件，如：窗口、标签、按钮、文本框等。Swing 提供了许多比 AWT 更好的屏幕显示元素，使用纯 Java 实现，能够更好的兼容跨平台运行。同时使用 Swing UI 界面程序设计还可以使用 Java 附加的小程序 windowManager 实现图形化编程，通过对 UI 组件的图形化拖拽、属性设置使得 Java 程序的 UI 设计变得更加已用。

二、设计思路

（一） 设计题目的基本要求

设计题目：基于 P2P 的局域网即时通信系统
已知技术参数和设计要求：
（1）掌握 P2P 原理。
（2）实现一个图形用户界面局域网内的消息系统。
（3）用户界面：界面上包括对等方列表；消息显示列表；消息输入框；文件传输进程显示及操作按钮或菜单。
（4）功能：建立一个局域网内的简单的 P2P 消息系统，程序既是服务器又是客户，服务器端口（自拟服务器端口号并选定）。

a. 用户注册及对等方列表的获取：对等方 A 启动后，用户设置自己的信息（用户名，所在组）；扫描网段中在线的对等方（服务器端口打开），向所有在线对等方的服务端口发送消息，接收方接收到消息后，把对等方 A 加入到自己的用户列表中，并发应答消息；对等方 A 把回应消息的其它对等方加入用户列表。双方交换的消息格式自己根据需要定义，至少包括用户名、IP 地址。

b. 发送消息和文件：用户在列表中选择用户，与用户建立 TCP 连接，发送文件或消息。

（二） 具体设计思路

在对题目的要求进行仔细分析后，我认为这个基于 P2P 的局域网通信系统按照功能以及实现逻辑可以分为四大部分，即用户图形界面 UI，专门负责处理通道信息的通道操作类，数据报文的自定义与解析，为通道操作类提供必要数据的一些工具类，各种耗时操作的线程，以及内容提供器。

1．用户界面 UI

这一部分需要完成的是登陆界面（输入用户名）以及主体的信息通信界面的图形设计，编程实现相关组件的相应事件。程序中相关的类名为 Login、MainFrame、ConnectFrame。

2．通道操作类

这一部分是整个程序的核心。通道操作类主要是实现底层的通信处理，例如向通道写入自定义的通信报文/文件、从通道读取报文/文件信息 以及文件的读取和保存功能。这一部分为底层的数据操作类，总体来说就是将通道的信息进行读取与写入操作，面向的层级是通信的数据流。值得一提的是，这里将使用 NIO 的处理方式，通过 1 到 2 个线程实现对多个连接通道的管理。程序中主要实现的

类名为Handler。

3．数据报文的自定义与解析

为了更好的管理通道中信息的类别、发送方，需要自定义在数据通道中传输的报文格式。本程序使用的信息报文主要有6类，分别是新启动用户信息交换报文、已启动用户信息交换报文、文件发送请求报文、文件允许发送报文、文件拒绝接受报文、文件确认接收报文、聊天信息报文。报文均已特定字符开头，通过检测报文开头即可确认报文的种类。若接受到的报文不是以这6类报文的特定字符串开头则说明接受到的是文件。程序中涉及的类名/接口名：HeadString（报文特定开头字符串）、Encoder（发送信息编码）。

4．基本工具类

基本工具类包含一些可能会用到的数据处理工具，例如自定义通信报文的编码和解码工具、获取本地IP地址和生成局域网内最大最小主机IP地址的IP工具。程序中主要的类名有：IPtool、Encoder。

5．各类耗时线程

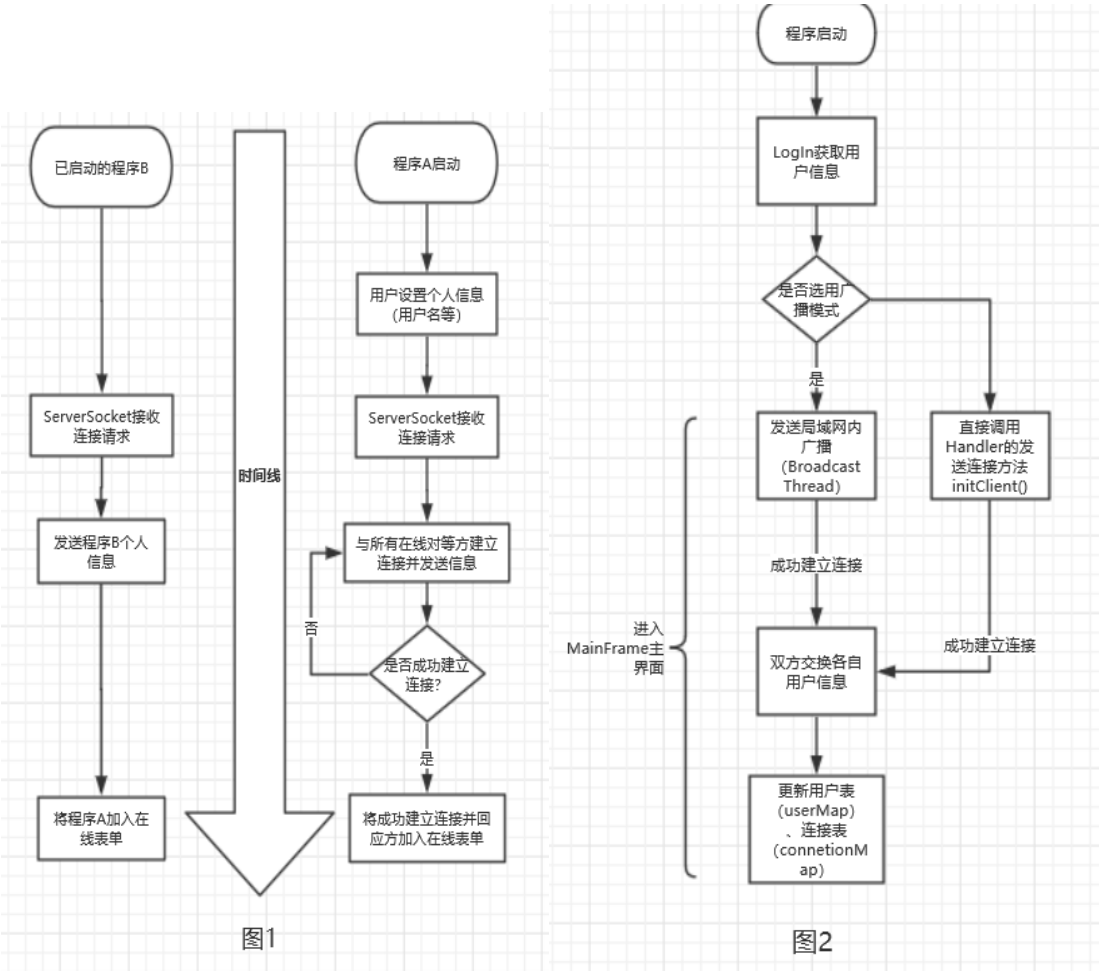
除了主要的UI操作主线程外，还需要将一些耗时的操作另外开辟线程来处理，防止UI线程阻塞对用户体验造成不利影响。程序涉及的各类耗时线程的操作分别有：广播建立连接过程、数据（包括文件和自定义报文）读取、进度条进度绘制。程序中主要的类名有：ServerThread、UpdateThread、BroadcastThread。

6．内容提供器

内容提供器主要是为所有组件提供用户信息存储，包含用户自己的个人信息（IP 地址、子网掩码、端口号、用户名）、已连接用户的信息 以及这些连接用户对应的信息通道。内容提供器只提供本次程序的非持续性存储，只记录程序运行过程中的信息数据。

三、程序流程图

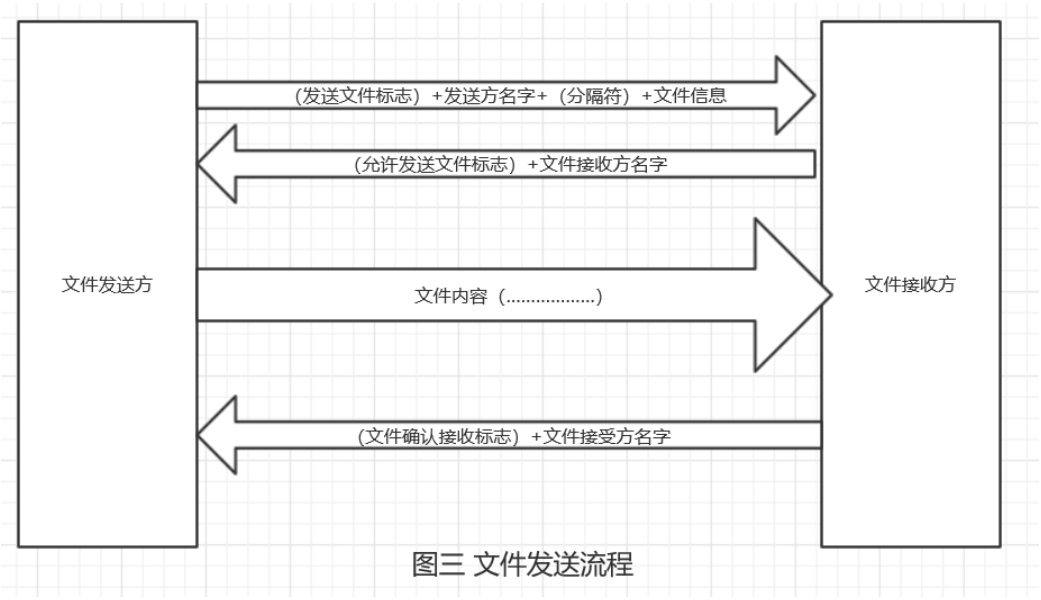
如下图一为程序的总体工作流程图。



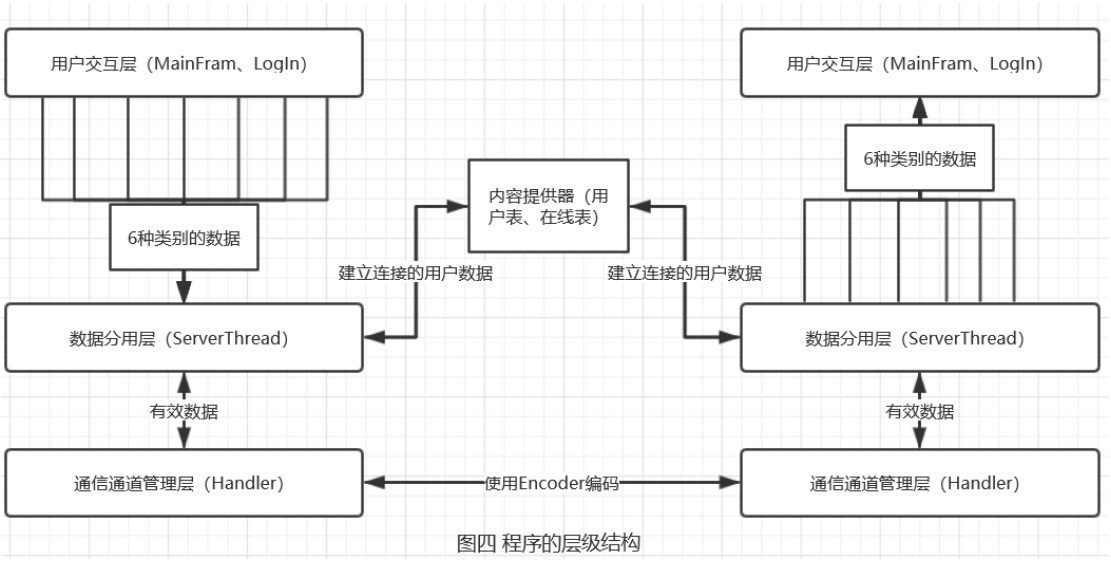
如图二，为程序启动的程序调用类、方法的简易流程

如图三为文件发送流程。文件发送为了获取文件的信息，通过多次准备信息的发送从而实现文件收发方对文件情况的实时了解。

图四为程序中各层次的逻辑划分



图三 文件发送流程

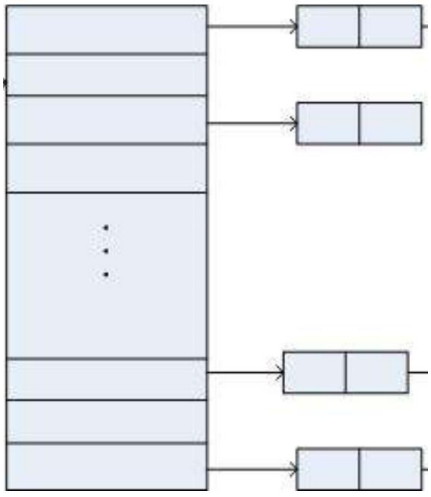


图四 程序的层级结构

四、关键数据结构

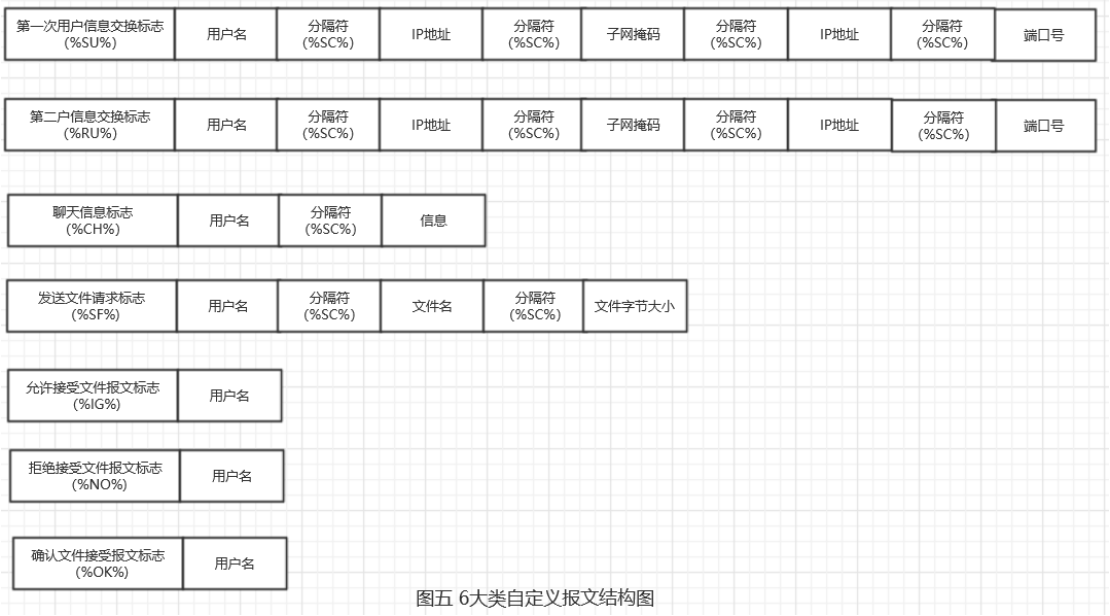
程序中主要使用了自定义的继承于Ｈａｓｈｍａｐ的ＯｎｌｉｎｅＭａｐ来存储用户信息。该数据结构具有哈希表通过键值对存储和查询的特征，以及集合的键值不可重复性，对外部仍然显现出类似哈希表的程序接口。哈希表的结构如下图所示，表中要求不能有相同值的一项。

程序中主要由内容提供器负责操作和存储多个数据表，其中在线用户表、在线连接表、历史记录表均采用这种方式存储。



在用户数据存储方面，设置了一个用户信息的存储类 User，记录用户的用户名、IP 地址、子网掩码以及程序端口号等信息。

另外，程序中自定义的传输报文结构如图五所示



图五 6大类自定义报文结构图

五、关键性代码

个人觉得程序的关键代码主要为初始化服务器和客户端部分的代码（都属于Ｈａｎｄｌｅｒ实现的部分）。这部分主要作用为初始化ｓｅｌｅｃｔｏｒ等重要变量，创建通道

另外在主线程外的通道信息监控线程ＳｅｒｖｅｒＴｈｒｅａｄ部分中的ｈａｎｄｌｅｒ（）函数也是整个程序的关键代码。这部分包含了针对不同类型的报文进行相应不同的操作。由于篇幅限制，ＳｅｒｖｅｒＴｈｒｅａｄ的这部分不显示出来。

```java
/************** 客户端部分接口 ***************/
// 创建客户端，并进行初始化
public boolean initClient(User ifo,String ip,int port) throws IOException {

    //初始化自己的用户名字
    myName = ifo.name;

    // 创建socketChannel，并绑定端口
    SocketChannel clientChannel = SocketChannel.open();
    clientChannel.configureBlocking(false);
    // 创建临时选择器，最后需要关闭,（！！！！这里的Selector与通道是唯一对应的所以可以这样用！！！！）
    Selector selectorT = Selector.open();
    clientChannel.register(selectorT, SelectionKey.OP_CONNECT);
    // 连接服务端socket
    InetSocketAddress otherAddress = new InetSocketAddress(ip, port);
    clientChannel.connect(otherAddress);
    // 完成三次握手中的第三次，必不可少！
    try {
        selectorT.select();
        Set<SelectionKey> selectionKeys = selectorT.selectedKeys();
        for (SelectionKey key : selectionKeys) {
            if (key.isConnectable()) {
                SocketChannel client = (SocketChannel) key.channel();
                if (client.isConnectionPending()) {
                    client.finishConnect();
                    // 传送个人信息给对方
                    sendMessage(Encoder.encodeIfo1(ifo), client);
                }
            }
        }
        selectionKeys.clear();
        // 关闭临时选择器
        selectorT.close();
        // 将已经建立连接的通道交由服务端监听，处理请求
        selector.reg(clientChannel, SelectionKey.OP_READ);
    } catch (IOException e) {
        //如果该IP地址连接不成功，finishConnect方法就会抛出异常，在这里捕获
        System.out.println(ip+"连接失败");
        return false;
    }
    return true;

}

/************** 服务端部分接口 ***************/
// 开启服务器，并做相关初始化
public void initServer(int port) throws IOException {
    // 创建serverSocketChannel ，监听端口
    ServerSocketChannel serverChannel = ServerSocketChannel.open();
    // 连接地址
    InetSocketAddress localAddress = new InetSocketAddress(IPtool.getMyIP(), port);
    serverChannel.socket().bind(localAddress);
    // 设置非阻塞模式
    serverChannel.configureBlocking(false);
    // 注册选择器
    try {
        selector = new MySelector();
        selector.reg(serverChannel, SelectionKey.OP_ACCEPT);
    }catch(IOException e) {
        e.printStackTrace();
        System.out.println("注册选择器出错！");
    }
    System.out.println("服务器开始工作！");
}
```

六、开发过程中遇到的问题及解决办法

实际编程中发现记录的问题总共有１１条，但是由于在撰写报告的时候部分问题已经讨论到并给出解决方法了（例如：传送文件与传送消息的衔接问题、个人信息互传死循环问题等），所以这里仅筛选个人认为容易出现且未提即的几个问题。

（一） selector 的方法锁问题

由于本程序使用的是 NIO 进行数据流的读写操作，所以不可避免的就是使用 NIO 的重要类——selector。在程序开发的过程中，连接的发起方在建立连接后需要重新注册通道的监听器（即 selector）为监听线程（ＳｅｒｖｅｒＴｈｒｅａｄ）中指定的监听器，从而使得监听线程能够监听并处理这条通道的事件。但在程序实际运行的过程中，会发现注册监听器操作所在的线程出现阻塞现象，程序无法完成对监听线程的监听器进行注册，自然也就无法成功的接受信息。

经过查阅关于 NIO 使用 selector 的资料，发现原来 selector 的 select（）方法和通道的 register（）注册监听器的方法存在冲突，即 selector 在监听通道的调用 select 方法的过程中，任何通道想要调用 register 方法，为通道注册该监听器都会发生线程阻塞的问题，网上具体的说法为“register 方法和 select 方法共有一把锁”。所以如果直接使用 NIO 接口提供的 selector，注册事件必须要 select（）方法返回的时候才能开始完成，体现程序出来的效果为，直到 selector 监听的所有通道内有什么新的事件，使得 select 方法的锁被释放转而去处理事件后，才能为原有客户端建立的通道注册该监听器，导致了这段时间用户传输数据的丢失。解决方法为自定义一个 selector 类，里面附带一个布尔变量以及同步方法用于处理 selector 阻塞的问题。代码如下：

```
public class MySelector {
    private volatile boolean mark = false;
    private final Selector selector;

    public MySelector() throws IOException {
        this.selector = Selector.open();
    }

    public Set<SelectionKey> selectedKeys(){
        return selector.selectedKeys();
    }

    public synchronized SelectionKey reg(SelectableChannel channel,int op)
            throws ClosedChannelException{
        mark = true;
        selector.wakeup();
        SelectionKey result = channel.register(selector, op);
        mark = false;
        return result;
    }

    public int select() throws IOException{
        while(true) {
            if(mark == true)
                continue;
            int result = selector.select();
            return result;
        }
    }
```

（二） 线程同步问题

这里的问题主要体现在主线程，即 UI 图形用户操作线程，在访问内容提供器所提供的用户数据时有可能与另一个负责监听通道并更新内容提供其中的数据表的线程不同步的问题，用户可能使用的是未更新的数据。针对这个问题，通过时监听线程具备绘制信息提示框以及更新用户表能力的方式，让用户知晓数据的变化。

（三） 对等方的连接断开问题

这部分主要还是通过对对通道读写操作是否抛出异常来判断，所以最好的解决方法是将读写进行集中处理，以便于在某个特定的位置接收到方法抛出的连接异常并及时处理，通知用户。本程序中在对通道的数据读取中实现了对通道读操作的集中处理，但是对于发送消息这部分的异常处理未设置专门的处理方法，但是考虑到程序大部分的操作以接受、处理数据为主，所以对于信息发送过程中连接中断的问题不是很突出。

另外，在程序发送广播的时候，需要判断是否成功建立连接的问题。这里主要通过处理 clientInit 方法里面调用的 finishConnect 方法抛出的异常来实现，即如果连接不成功 finishConnect 方法将抛出异常。

（四） 自连问题

就是在广播的时候仍会再次向本地的 IP 地址发送连接请求，导致程序出现自己跟自己聊天发文件的奇怪现象，解决方法也较为简单，就是在生成广播 IP 地址时，自动忽略掉自己的 IP 地址。

七、程序中待解决的问题及改进方向

首先，我认为自己的程序在界面设设计上还是过于简单、粗暴，没有给用户一种美观、简洁的使用视觉。这方面还需要多学习参照前端界面设计的案例和知识。

其次，程序的健壮性还需进一步的调试调高，在开发以及测试的过程中都由我一个人完成，许多程序运行的逻辑、程序运行过程中的各种特例都仅是在自己测试的范围内，可能仍有许多未知的错误需要被发现和改正。

最后，不得不提的是程序在所在局域网内广播的过程，主要还是依靠一个一个 IP 地址去发送连接请求，每个连接建立以及发现无法连接的过程都需要消耗时间，进而导致广播时间的总时间较长。在完成课程设计的程序代码编写与同一个选题的同学交流后，我发现如果可以通过 IP 多播的方式，向局域网内的所有主机发送个人 IP 地址及个人信息，再由各在线终端从 IP 多播地址获取这些关

键信息，进而发送直接的对点连接的方式，在局域网内搜寻可连接主机这一耗时的问题将能得到较好的解决。这也仅是一个突发的思路，未作进一步深入的实现过程思考。