客户端核心代码解释

```
private void setUpNetworking() {
        // open a SocketChannel to the server
        SocketChannel svr = null;
        try {
            svr = SocketChannel.open(new InetSocketAddress("192.168.118.128",
8080));
            //用open这个静态方法创建SocketChannel对象，传入一个包含IP地址和端口号的
InetSocketAddress对象
            //用于连接服务器。
            System.out.println("连接成功");
            BufferedReader br = new BufferedReader(new
InputStreamReader(svr.socket().getInputStream()));
            //BufferedReader对象用于接收服务器返回的字节流，转成字符流并包装。
            System.out.println("现在可以向服务器发送消息，输入-1退出");
            String input = " ";
            Scanner sc = new Scanner(System.in);
            //从终端中读入用户输入
            while (!"-1".equals(input)) {
                //如果这里用==，是比较两个字符串引用指向的内存地址是否相同，很显然不同（即使值
都是“-1”）
                //String对象重写了equals方法用于比较两个字符串的内容是否相同，所以这里要用
equals
                input = sc.nextLine();
                //该方法会读入一行文本直到遇到换行符，input本身不会包括换行符
                sendMessage(svr, input);
                System.out.println(br.readLine());
                //输出服务器返回的消息
            }
            System.out.println("与服务器的连接已断开....");
            svr.close();
            sc.close();
        } catch (IOException e) {
            System.out.println("与服务器交互失败");
            //throw new RuntimeException(e);
        }

    }
    public void sendMessage(SocketChannel svr, String msg) throws IOException {//
发送消息给服务器
        svr.write(StandardCharsets.UTF_8.encode(msg+"\n"));
        //先将字符串编码成字节数组，并通过write方法以字节流的形式写入到连接通道
（SocketChannel）中
        //在服务端的readline方法是一个阻塞方法，需要读到换行符才会返回值，但是nextline方法不
会读入换行符，需要手动添加
    }
```

服务器核心代码解释

```
public class SimpleChatServer {
    public final List<PrintWriter> clientWriters = new ArrayList<>();
    //这个列表用于存储每一个PrintWriter对象，当服务器要将消息广播给各个客户端的时候就可以调用
这些PrintWriter的Println方法
    public static void main(String[] args) {
        new SimpleChatServer().go();
    }
```

```
public void go(){
  //创建一个通道用于和客户端之间的通信
  SocketChannel clientchannel = null;
  int id=-1;
   try{
      //服务器创建一个ServerSocketChannel并绑定到一个特定端口
    ServerSocketChannel serverChannel = ServerSocketChannel.open();
    serverChannel.bind(new InetSocketAddress(8080));
    while(true){
     clientchannel = serverChannel.accept();
     //一直监听，直到客户端连接
     System.out.println("connect successfully with
"+clientchannel.socket().getRemoteSocketAddress());
     PrintWriter writer = new PrintWriter(new
OutputStreamWriter(clientchannel.socket().getOutputStream()));
     clientWriters.add(writer);
        //每一个与客户端连接服务器，都创建一个PrintWriter对象，保存在列表中
     id = id + 1;
     Thread t = new Thread(new ClientHandler(clientchannel,id));//创建线程
      //id用于标注线程的编号，方便从列表中删除对应的PrintWriter

     t.start();//将线程变为就绪状态
        }
    }catch(IOException e){
        e.printStackTrace();
      }
    }
```

```
   public class ClientHandler implements Runnable{
       //通过实现Runnalbe接口创建一个线程，需要重写run（）方法
        private SocketChannel socketChannel;
        public int id;
        public ClientHandler(SocketChannel s,int id){
            //有参构造器，传入通道和id
            socketChannel = s;
            this.id = id;
        }
       //重写run（）方法
        public void run(){
        try{
     BufferedReader br = new BufferedReader(new
InputStreamReader(socketChannel.socket().getInputStream()));
//BufferedReader用于获取客户端发来的消息
            String line;
```

```
            while((line = br.readLine()) != null){//按行读取
                if("-1".equals(line)) {//读到-1则断开连接
                 System.out.println("disconnect with "+socketChanne
l.socket().getRemo teSocketAddr ess()) ;
                 break;
                }else{
                    System.out.println("Received message:"+line);
                    tellEveryone(line);//将这条消息广播给所有客户端
                }

            }
         }catch(IOException e){
                System.err.println("failed to read");
          }finally{
            clientWriters.remove(id);
            //将对应的PrintWriter对象从列表中移除
            try{
                socketChannel.close();//如果关闭不成功也会抛出异常，需要捕获
            }catch(IOException e){
                System.err.println("fail to close socket");
                e.printStackTrace();
            }
           }
        }
    }
```

1.Socket和SocketChannel的一些区别

Socket 是阻塞模式的，这意味着当程序调用如 read()或 write() 这样的方法时，如果操作没有完成（例如，没有数据可读或数据尚未完全写入），那么线程将会被挂起直到操作完成。

SocketChannel支持非阻塞模式，这允许程序在没有数据可读或写入的情况下继续执行其他任务。

Socket使用传统的字节流模型（InputStream和 OutputStream）。

SocketChannel 使用基于缓冲区的模型（ByteBuffer）

所以上面服务器读取客户端的信息用的是Socket的输入流socket().getInputStream()，用也可以用ByteBuffer

```
        ByteBuffer buffer = ByteBuffer.allocate(256);//创建一个容量为256字节的缓冲区
        int bytesRead = clientChannel.read(buffer);//将数据读入到缓冲区
        if (bytesRead > 0) {
            buffer.flip();
            //写入数据的时候读写头指向最后一个位置，因此需要flip（翻转）将读写头指向第一个位
置，从缓冲区的起始点开始读取

            byte[] data = new byte[buffer.remaining()];
            //返回的是从当前位置（position）到界限（limit）之间的剩余字节数。确保不会超出界
限
            buffer.get(data); // 从 buffer 中获取数据读取到字节数组中
            String message = new String(data, StandardCharsets.UTF_8);//将字节数组
转成字符串，并设置编码为UTF-8
            System.out.println("Received message: " + message);
```

SocketChannel在实例化的时候会创建一个对等的Socket对象，两者之间可以互相得到

```
Socket socket = soketChannel.socket();
SocketChannel socketChannel = socket.getChannel();
```

2.进程和线程

1. 一个进程可以启动多个线程
2. 线程A和线程B，堆内存 和 方法区 内存共享。但是 栈内存独立，一个线程一个栈。，每个栈和每个栈之间，互不干扰，各自执行各自的。
3. 五个状态：新建状态，就绪状态，运行状态，阻塞状态，死亡状态
4. 实现线程的几种方法：
    1. 继承java.lang.Thread 重写run方法

       启动：

       ```
       MyThread t = new MyThread();          *// 启动线程*

        *//t.run(); // 不会启动线程，不会分配新的分支栈。（这种方式就是单线程。）*

        t.start();
       ```

    2. 实现java.lang.Runnable接口，实现run方法.

       启动

       ```
       // 实现Runnable接口
       public class MyRunnable implements Runnable {
           public void run(){
           }
       }
       // 创建线程对象
       Thread t = new Thread(new MyRunnable());
       // 启动线程
       t.start();
       ```

    3. sleep方法，Thread.sleep(1000);

       让当前线程进入休眠，进入阻塞状态，放弃占有CPU时间片，让给其它线程使用。

       void interrupt 终止线程的睡眠