华东师范大学数据科学与工程学院实验报告

课程名称：计算机网络与编程	**年级：**大一	**上机实践成绩：**
指导教师：张召	**姓名：**林子骥	**学号：**10225501460
上机实践名称：Lab8: 基于 TCP 的 Socket 编程优化		**上机实践日期：**5.1
上机实践编号：No.8	**组号：**	**上机实践时间：**

一、 实验目的

- 对数据发送和接收进行优化
- 实现信息共享
- 熟悉阻塞 I/O 与非阻塞 I/O

二、 实验任务

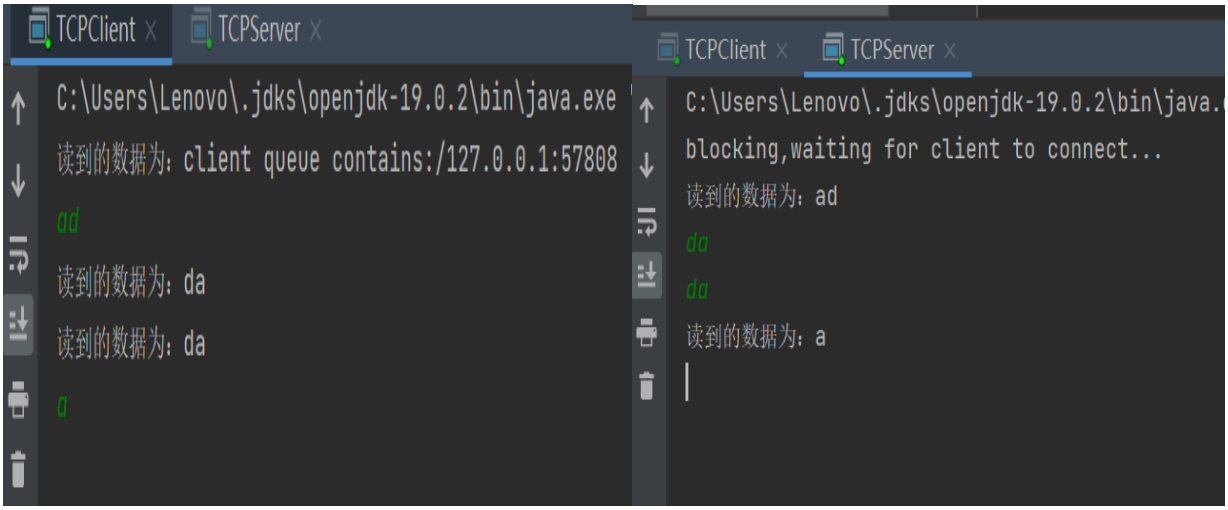
- 将数据发送与接收并行，实现全双工通信
- 实现服务端向所有客户端广播消息
- 了解非阻塞 I/O

三、 使用环境

- IntelliJ IDEA JDK
- 版本: Java 19

四、 实验过程

Task1



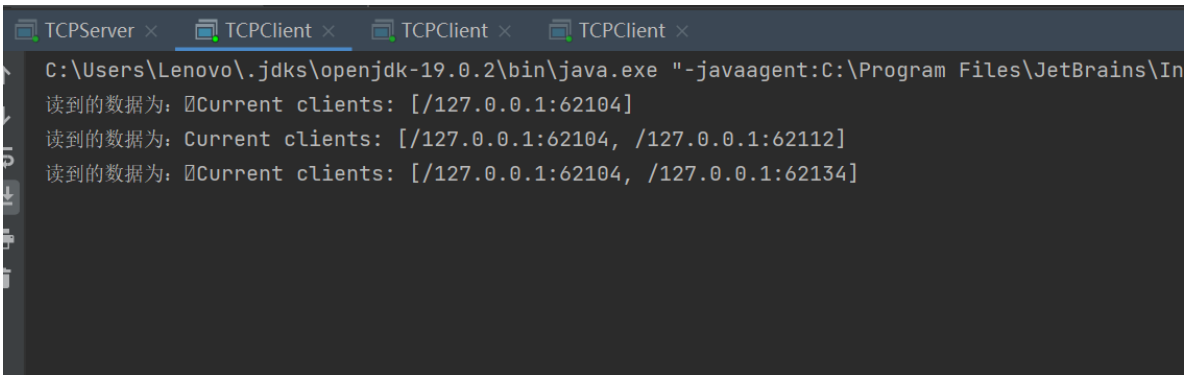

```
//处理从客户端读取的数据
class ServerReadHandler extends Thread {
    private final BufferedReader bufferedReader;
    ServerReadHandler(InputStream inputStream) {
        this.bufferedReader = new BufferedReader(new
InputStreamReader(inputStream));
```

```
    }
    @Override
    public void run() {
        try {
            while (true) {
                // 拿到客户端一条数据
             String str = bufferedReader.readLine();
                if (str == null) {
                    System.out.println("读到的数据为空");
                    break;
                } else {
                    System.out.println("读到的数据为：" + str);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

// 处理向客户端写数据的线程
class ServerWriteHandler extends Thread {
    private final PrintWriter printWriter;
    private final Scanner sc;
    ServerWriteHandler(OutputStream outputStream) {
        this.printWriter = new PrintWriter(new OutputStreamWriter(outputStream),
true);
        this.sc = new Scanner(System.in);
    }
    void send(String str){
        this.printWriter.println(str);
    }

    @Override
    public void run() {
        while (sc.hasNext()) {
            // 拿到控制台数据
          String str = sc.next();
            send(str);
        }
    }
}

class ServerHandler extends Thread {
    private Socket socket;
    private final ServerReadHandler serverReadHandler;
    private final ServerWriteHandler serverWriteHandler;
    ServerHandler(Socket socket) throws IOException{
        this.socket = socket;
        this.serverReadHandler = new ServerReadHandler(socket.getInputStream());
        this.serverWriteHandler = new
ServerWriteHandler(socket.getOutputStream());
    }

    @Override
    public void run() {
        super.run();
        serverReadHandler.start();
        serverWriteHandler.start();
```

```
    }
}
```

```
try {
        client.startConnection("127.0.0.1", port);
}
catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        Thread.sleep(1000);
    }catch (InterruptedException e){
        e.printStackTrace();
    }
    client.stopConnection();
}
```

Task2



第一行输出表示了一个客户端已经连接；

第二行输出表示了有两个客户端已经连接；

第三行输出表示其中一个客户端断开连接，另一个新的客户端建立了连接。

TCPServer

```
public void start(int port) throws IOException {
// 1. 创建服务器端 Socket，即 ServerSocket，监听指定端?
        serverSocket = new ServerSocket(port);
// 2. 调?accept()?法开始监听，阻塞等待客户端的连接
       System.out.println("blocking,waiting for client to connect...");
        for (;;) {
            Socket socket = serverSocket.accept();
            System.out.println("has accepted:"+socket.getRemoteSocketAddress());
            clientsSocketList.add(socket);
            sendMessageToAllClients();
            ClientHandler ch = new ClientHandler(socket);
            ch.start();
        }
    }

    private void sendMessageToAllClients() {
        List<String> remoteSocketAddressList = new ArrayList<String>();
        Iterator<Socket>iterator = clientsSocketList.iterator();
        while(iterator.hasNext()) {
            Socket socket = iterator.next();
```

```
            try{
                socket.getOutputStream().write(0);

remoteSocketAddressList.add(socket.getRemoteSocketAddress().toString());
            }catch (IOException e){
                iterator.remove();
            }
        }

        String message = "Current clients: " +
remoteSocketAddressList.toString();
        for (Socket socket : clientsSocketList) {
            try {
                PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);
                out.println(message);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

TcpClient 增加函数

```
public  void getMessageOfQueue() throws IOException{
    BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
    String rec;
    while ((rec = in.readLine())!=null){
        System.out.println(rec);
    }
}
```

Task3

运行 NIOServer 和 TCPClient 后，可以发现以下不同之处：

1. NIOServer 使用了非阻塞 I/O，可以同时处理多个客户端连接，而 TCPServer 使用了阻塞 I/O，需要等待每个客户端连接完成才能继续处理下一个连接。

2. NIOServer 使用了 select()轮询方式，可以同时监听多个通道的读写事件，而 TCPServer 只能在一个通道上进行读写操作。

3. NIOServer 使用了单个线程处理多个连接，而 TCPServer 需要为每个连接创建一个新的线程，造成线程资源的浪费。

当有并发的 1 万个客户端想要建立连接时，在阻塞 I/O 的 TCPServer 可能会存在以下问题：

1. 阻塞 I/O 模型会导致每个客户端连接都需要等待前面的连接完成才能建立连接，因此会出现大量的连接请求排队等待的情况，造成连接建立的延迟和响应时间的增加。

2. 每个连接都需要单独创建一个线程，当连接数达到一定规模时，线程资源会被耗尽，导致无法继续处理新的连接请求，甚至可能导致系统崩溃。

3. 阻塞 I/O 模型无法应对高并发场景，因为每个连接的读写操作都会阻塞整个线程，造成系统资源的浪费，降低系统的吞吐量。

Task4

在 NIOServer 代码中，selector 和 channel 的方法有很多封装了底层的系统调用，以下

是各个方法与系统调用的对应关系：

Selector.open()：调用的是 epoll 系统调用。

channel.configureBlocking(false)：在内核中调用了 fcntl 系统调用将文件描述符设置为非阻塞模式。

SelectionKey.OP_READ：相当于注册监听读取事件，调用了 epoll_ctl 添加监听给定文件描述符上的读取事件。

SelectionKey.OP_READ | SelectionKey.OP_WRITE：相当于注册监听读写事件，同上添加监听给定文件描述符的可读可写的事件。

selector.select()：在 Linux 中对应的是 epoll_wait 系统调用。

Set<SelectionKey> selectedKeys = selector.selectedKeys()：获取已经就绪的通道的 SelectionKey 集合，而这个集合其实就是在之前调用 select 方法时返回的已经准备好的时间集合。

selectionKey.isReadable/isWritable()：判断是否可以读/写，调用的是内核中相应文件描述符是否可读可写。

channel.read(buffer)/channel.write(buffer)：读/写数据，这些方法最终也会在内核里使用类似 read/write 等系统调用来实现具体的读写操作。

Task5

```
public class NIOClient {
    private static int BYTE_LENGTH = 64;
    public static void main(String[] args) throws IOException {
        SocketChannel channel = SocketChannel.open(new
InetSocketAddress("localhost", 9091));
        //set non blocking
        channel.configureBlocking(false);
        // the message sent
        String message = "Hello Server";
        //convert to byte
        ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());
        channel.write(buffer);
        ByteBuffer inBuffer = ByteBuffer.allocate(BYTE_LENGTH);
        while (true) {
            int numRead = channel.read(inBuffer);
            //连接断开
          if (numRead == -1) {
                System.out.println("off the connection");
                break;
            }
            if (numRead > 0) {
                byte[] data = new byte[numRead];
                inBuffer.flip();
                inBuffer.get(data,0,numRead);
                channel.write(inBuffer);
                System.out.println("print out message:" + new String(data));
                inBuffer.clear();
            }
        }
        channel.close();
    }

    private void read(SelectionKey key) throws IOException {
        SocketChannel channel = (SocketChannel) key.channel();
```

```
        ByteBuffer buffer = ByteBuffer.allocate(BYTE_LENGTH);
        int numRead = -1;
        numRead = channel.read(buffer);
        if (numRead == -1) {
            Socket socket = channel.socket();
            SocketAddress remoteAddr = socket.getRemoteSocketAddress();
            System.out.println("连接关闭: " + remoteAddr);
            channel.close();
            key.cancel();
            return;
        }
        byte[] data = new byte[numRead];
        System.arraycopy(buffer.array(), 0, data, 0, numRead);
        System.out.println("服务端已收到消息: " + new String(data));
    }
}
```

NIOServer

```
//回写消息
key.attach("hello client".getBytes());
SocketChannel socketchannel = (SocketChannel) key.channel();
byte[] bytes = (byte[]) key.attachment();
socketchannel.write(ByteBuffer.wrap(bytes));
```

总结

数据发送与接收并行，实现全双工通信的总结：

原理：数据发送与接收并行，实现全双工通信的原理是通过使用两个线程分别处理数据的发送和接收，使得数据的发送和接收可以同时进行，从而实现全双工通信。

实现过程：

1.需要创建两个线程，一个线程负责数据的发送，一个线程负责数据的接收。

2.在发送线程中，需要将要发送的数据存储到发送缓冲区中，并通过系统调用向目标主机发送数据。

3.在接收线程中，需要通过系统调用监听网络端口，等待接收数据，并将接收到的数据存储到接收缓冲区中。

4.发送线程和接收线程可以通过共享内存或消息队列等方式进行数据交互和同步。

优点：

1.数据发送与接收并行，实现全双工通信可以提高网络通信的效率和速度，使得数据的发送和接收可以同时进行，降低了网络延迟和响应时间。

2.数据发送与接收并行，实现全双工通信可以提高系统的并发处理能力，可以同时处理多个连接，提高系统的吞吐量。

应用场景：

高并发场景：如服务器端的网络编程等。

实时通信场景：如视频会议、在线游戏等。

大数据传输场景：如文件传输、数据备份等。

以下是非阻塞 I/O 的详细总结

非阻塞 I/O 是一种 I/O 模型，相对于阻塞 I/O，它的特点是在读写操作时不会阻塞整个进程或线程，而是立即返回，告知调用者当前操作是否完成。

原理：非阻塞 I/O 的实现原理是基于操作系统提供的非阻塞 I/O 接口。在使用非阻塞 I/O 时，应用程序首先要将 I/O 通道设置为非阻塞模式，然后通过轮询或事件通知方式等待 I/O 操作完成。

优点：

1.非阻塞 I/O 可以提高系统的并发处理能力，可以同时处理多个客户端连接，提高系统的吞吐量。

2.非阻塞 I/O 可以避免 I/O 操作阻塞整个进程或线程，提高系统的响应速度，降低系统的延迟。

3.非阻塞 I/O 可以减少线程或进程的创建和销毁，降低系统的开销，提高系统的效率。

不足：

1.非阻塞 I/O 的实现比较复杂，需要对 I/O 操作进行轮询或事件通知等处理，增加了代码的复杂度和难度。

2.非阻塞 I/O 的性能受到操作系统和硬件的影响比较大，不同的操作系统和硬件对非阻塞 I/O 的支持程度不同。

应用场景：

高并发场景：如服务器端的网络编程等。

长连接场景：如即时通讯、游戏等。

多任务场景：如爬虫、批处理等。