Lab8: 基于TCP的Socket编程优化

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

1. 完善数据发送与接收并行

- 在Lab7的实验中，仅能支持客户端发送一行数据然后服务端回写一行数据，功能有限。本节将尝试将数据发送与接收并行，客户端和服务端可任意地发送和接收数据。
- 增加ClientReadHandler类

```
// 处理从客户端读数据的线程
class ClientReadHandler extends Thread {
    private final BufferedReader bufferedReader;

    ClientReadHandler(InputStream inputStream) {
        this.bufferedReader = new BufferedReader(new InputStreamReader(inputStream,
StandardCharsets.UTF_8));
    }

    @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();
        }
    }
}
```

- 增加ClientWriteHandler类

```
// 处理向客户端写数据的线程
class ClientWriteHandler extends Thread {
    private final PrintWriter printWriter;
    private final Scanner sc;

    ClientWriteHandler(OutputStream outputStream) {
        this.printWriter = new PrintWriter(new OutputStreamWriter(outputStream, StandardCharsets.UTF_8), 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);
        }
    }
}
```

- 增加ClientHandler类

```
class ClientHandler extends Thread {
    private Socket socket;
    private final ClientReadHandler clientReadHandler;
    private final ClientWriteHandler clientWriteHandler;

    ClientHandler(Socket socket) throws IOException{
        this.socket = socket;
        this.clientReadHandler = new ClientReadHandler(socket.getInputStream());
        this.clientWriteHandler = new ClientWriteHandler(socket.getOutputStream());
    }

    @Override
    public void run() {
        super.run();
        clientReadHandler.start();
        clientWriteHandler.start();
    }
}
```

- 修改TCPServer类代码段

```
for (;;) {
    Socket socket = serverSocket.accept();
    ClientHandler ch = new ClientHandler(socket);
    ch.start();
}
```

Task1: 继续修改TCPClient类，使其发送和接收并行，达成如下效果，当服务端和客户端建立连接后，无论是服务端还是客户端均能随时从控制台发送消息、将接收的信息打印在控制台，将修改后的TCPClient代码附在实验报告中，并展示运行结果。

2. 实现消息共享

Task2: 修改TCPServer和TCPClient类，达成如下效果，每当有新的客户端和服务端建立连接后，服务端向当前所有建立连接的客户端发送消息，消息内容为当前所有已建立连接的Socket对象的`getRemoteSocketAddress()`的集合，请测试客户端加入和退出的情况，将修改后的代码附在实验报告中，并展示运行结果。

3. 阻塞I/O与非阻塞I/O

- 请先阅读以下资料：https://mp.weixin.qq.com/s/CCFG3rFUBLpWrLbAV_9qiQ
- 给出在用户态模拟I/O多路复用的服务端NIOServer

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class NIOServer {
    private static List<SocketChannel> channelList = new ArrayList<>();
    private static int BYTE_LENGTH = 64;

    public static void main(String[] args) throws IOException {
        // ServerSocketChannel与serverSocket类似
        ServerSocketChannel serverSocket = ServerSocketChannel.open();
        serverSocket.socket().bind(new InetSocketAddress(9091));
        // 设置ServerSocketChannel为非阻塞
        serverSocket.configureBlocking(false);
        System.out.println("服务端启动");

        for (;;) {
            // accept方法不阻塞
            SocketChannel socketChannel = serverSocket.accept();
            if (socketChannel != null) {
                System.out.println("连接成功");
                socketChannel.configureBlocking(false);
                channelList.add(socketChannel);
            }
            // 遍历连接进行数据读取
            Iterator<SocketChannel> iterator = channelList.iterator();
            while (iterator.hasNext()) {
                SocketChannel sc = iterator.next();
                ByteBuffer byteBuffer = ByteBuffer.allocate(BYTE_LENGTH);
                // read方法不阻塞
                int len = sc.read(byteBuffer);
                // 如果有数据，把数据打印出来
                if (len > 0) {
                    System.out.println("服务端接收到消息：" + new
String(byteBuffer.array()));
                } else if (len == -1) {
                    // 如果客户端断开，把socket从集合中去掉
                    iterator.remove();
                    System.out.println("客户端断开连接");
                }
            }
        }
    }
}
```

Task3: 尝试运行NIOServer并运行TCPClient，观察TCPServer和NIOServer的不同之处，并说明当有并发的1万个客户端(C10K)想要建立连接时，在Lab7中实现的TCPServer可能会存在哪些问题。

- 给出在内核态实现I/O多路复用的服务端NIOServer

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
```

```java
import java.util.Iterator;
import java.util.Set;

public class NIOServer {
    private static int BYTE_LENGTH = 64;
    private Selector selector;

    public static void main(String[] args) throws IOException {
        try {
            new NIOServer().startServer();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void startServer() throws IOException {
        this.selector = Selector.open();
        // ServerSocketChannel与serverSocket类似
        ServerSocketChannel serverSocket = ServerSocketChannel.open();
        serverSocket.socket().bind(new InetSocketAddress(9091));
        // 设置无阻塞
        serverSocket.configureBlocking(false);
        // 将channel注册到selector
        serverSocket.register(this.selector, SelectionKey.OP_ACCEPT);
        System.out.println("服务端已启动");

        for (;;) {
            // 操作系统提供的非阻塞I/O
            int readyCount = selector.select();
            if (readyCount == 0) {
                continue;
            }

            // 处理准备完成的fd
            Set<SelectionKey> readyKeys = selector.selectedKeys();
            Iterator iterator = readyKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = (SelectionKey) iterator.next();

                iterator.remove();

                if (!key.isValid()) {
                    continue;
                }

                if (key.isAcceptable()) {
                    this.accept(key);
                } else if (key.isReadable()) {
                    this.read(key);
                } else if (key.isWritable()) {
                }
            }
        }
    }

    private void accept(SelectionKey key) throws IOException {
        ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
        SocketChannel channel = serverChannel.accept();
        channel.configureBlocking(false);
        Socket socket = channel.socket();
        SocketAddress remoteAddr = socket.getRemoteSocketAddress();
        System.out.println("已连接: " + remoteAddr);
        // 监听读事件
        channel.register(this.selector, SelectionKey.OP_READ);
    }

    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));
    }
}
```

Task4: 尝试运行上面提供的NIOServer，试猜测该代码中的I/O多路复用调用了你操作系统中的哪些API，并给出理由。

Task5 (Bonus): 编写基于NIO的NIOClient，当监听到和服务器建立连接后向服务端发送"Hello Server"，当监听到可读时将服务端发送的消息打印在控制台中。（自行补全NIOServer消息回写）