

## 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：21 级

上机实践成绩：

指导教师：张召

姓名：杨茜雅

学号：10215501435

上机实践名称：Lab08

上机实践日期：2023. 4. 21

上机实践编号：

组号：

上机实践时间：2023. 4. 21

### 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

Task 1: 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。

```
package Task1;

import java.io.*;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

public class TCPClient {
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;
    private Thread receiveThread;

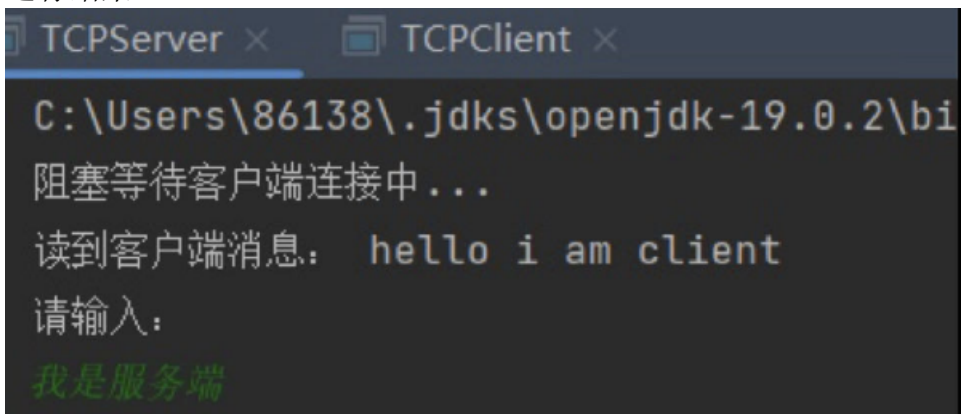
    public void startConnection(String ip, int port) throws IOException {
        clientSocket = new Socket(ip, port);
        out = new PrintWriter(new OutputStreamWriter(clientSocket.getOutputStream(), StandardCharsets.UTF_8), autoFlush: true);
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream(), StandardCharsets.UTF_8));
        receiveThread = new Thread() -> {
            try {
                while (true) {
                    String msg = in.readLine();
                    if (msg == null) {
                        System.out.println("服务器已断开连接");
                        break;
                    }
                }
                System.out.println("收到服务端消息: " + msg);
            }
        };
    }
}
```

```
        System.out.println("收到服务端消息: " + msg);
    }
} catch (IOException e) {
    e.printStackTrace();
}
});
receiveThread.start();
}

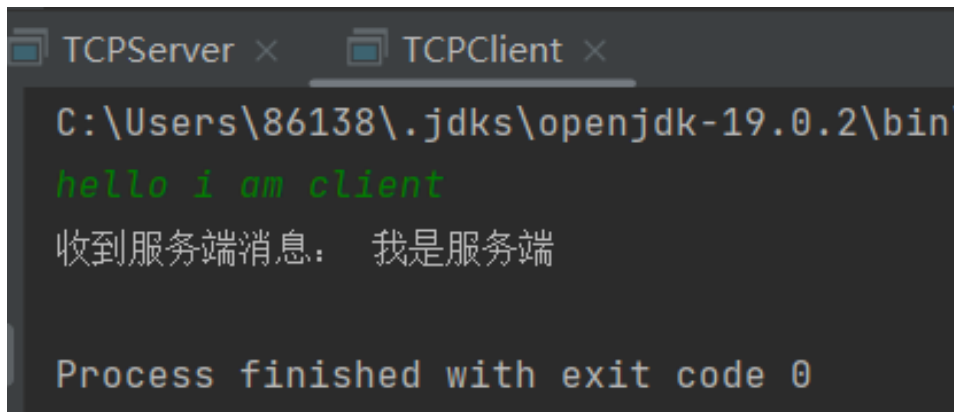
public void stopConnection() {
    try {
        if (in != null) in.close();
        if (out != null) out.close();
        if (clientSocket != null) clientSocket.close();
        if (receiveThread != null) receiveThread.interrupt();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws IOException {
    int port = 9091;
    TCPClient client = new TCPClient();
    Scanner scanner = new Scanner(System.in);
    try {
        client.startConnection("127.0.0.1", port);
        while (true) {
            String msg = scanner.nextLine();
            if (msg.equalsIgnoreCase("quit")) {
                break;
            }
            client.sendMessage(msg);
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        scanner.close();
        client.stopConnection();
    }
}
```

运行结果:



```
TCPServer x TCPClient x
C:\Users\86138\.jdk\openjdk-19.0.2\bin
阻塞等待客户端连接中...
读到客户端消息: hello i am client
请输入:
我是服务端
```



```
TCPServer x TCPClient x
C:\Users\86138\.jdk\openjdk-19.0.2\bin\
hello i am client
收到服务端消息: 我是服务端
Process finished with exit code 0
```

Task 2: 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。

TCPServer:

```
package Task2;

import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;

public class TCPServer{

    private ServerSocket serverSocket;
    private ArrayList<Socket> clients;

    public void start(int port) throws IOException {

        serverSocket = new ServerSocket(port);
        clients = new ArrayList<>();

        while(true){

            Socket clientSocket = serverSocket.accept();
            clients.add(clientSocket);
            System.out.println("客户端" + clientSocket.getRemoteSocketAddress() + "已连接");

            StringBuilder sb = new StringBuilder("当前已连接的客户端地址: ");
            for(Socket socket : clients){
                sb.append(socket.getRemoteSocketAddress()).append(",");
            }
            sb.delete(sb.length() - 2, sb.length());
            sendMessageToAllClients(sb.toString());

            BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream(),
                StandardCharsets.UTF_8));

            clientSocket.close();
            clients.remove(clientSocket);
            System.out.println("客户端" + clientSocket.getRemoteSocketAddress() + "已断开连接");
        }
    }
}
```

```
private void sendMessageToAllClients(String message) throws IOException{
    for (Socket socket : clients){
        PrintWriter out = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),
            StandardCharsets.UTF_8),
            autoFlush: true);
        out.println(message);
    }
}

public void stop(){
    try{
        if (serverSocket != null){
            serverSocket.close();
        }
        for (Socket socket : clients){
            if (socket != null){
                socket.close();
            }
        }
    }catch (IOException e){
        e.printStackTrace();
    }
}

public static void main(String[] args){
    int port = 9091;
    TCPServer server = new TCPServer();
    try{
        server.start(port);
    } catch (IOException e){
        e.printStackTrace();
    } finally{
        server.stop();
    }
}
```

### TCPClient:

```
package Task2;

import java.io.*;
import java.net.Socket;
import java.nio.charset.StandardCharsets;

public class TCPClient{

    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;

    public void startConnection(String ip, int port) throws IOException{
        clientSocket = new Socket(ip, port);
        out = new PrintWriter(new OutputStreamWriter(clientSocket.getOutputStream(), StandardCharsets.UTF_8), autoFlush: true);
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream(), StandardCharsets.UTF_8));
    }

    public String sendMessage(String msg) throws IOException{
        out.println(msg);
        String resp = in.readLine();
        return resp;
    }

    public void stopConnection(){
        try{
            if (in != null) in.close();
            if (out != null) out.close();
            if (clientSocket != null) clientSocket.close();
        }
    }
}
```

```
}catch (IOException e){
    e.printStackTrace();
}
}

public static void main(String[] args){
    int port = 9091;
    TCPClient client = new TCPClient();
    try{
        client.startConnection( ip: "127.0.0.1", port);
        String response = client.sendMessage( msg: "用户名: 10215501435;");
        System.out.println(response);
    }catch (IOException e) {
        e.printStackTrace();
    }finally{
        client.stopConnection();
    }
}
```

运行结果:

```
客户端 /127.0.0.1: 69221 已连接
客户端 /127.0.0.1: 69221 已断开连接
客户端 /127.0.0.1: 69227 已连接
客户端 /127.0.0.1: 69227 已断开连接
客户端 /127.0.0.1: 69233 已连接
客户端 /127.0.0.1: 69233 已断开连接
```

Task 3: 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。

代码运行结果:

```
服务端启动
连接成功
服务端接收到消息: 用户名: 10215501435;
```

TCPServer和NIOServer的不同之处:

TCP Server 和 NIO Server 都是网络编程中的服务器程序实现方式, 二者的主要区别在于其底层的 I/O 处理方式。具体如下:

- 1、TCP Server 采用的是阻塞式 (Blocking) I/O 方式, 也就是说当 Socket 进行读写操作时, 线程会被阻塞, 直到该操作完成。
- 2、NIO Server 采用的是非阻塞式 (Non-blocking) I/O 方式。当 Socket 进行读写操作时, 线程不会被阻塞, 而是将其读写操作委托给内核完成, 通过轮询方式检查 Socket 连接事件并进行处理。这种方式可以充分利用 CPU 资源, 使同一线程可以处理多个连接请求。



因此,从上述介绍中可以看出,TCP Server 相对于 NIO Server 更加简单易用,但性能和扩展性方面局限性更大,支持的并发连接数少;而 NIO Server 则可以处理更多的并发连接,并提供更好的性能和扩展性。但在编码复杂度和处理方式上相对更加复杂。因此,在实践中需要根据需求和具体情况选择使用哪种服务器实现方式。

### 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

当1万个客户端同时想要建立TCP连接时,TCPServer可能会存在以下问题:

- 1、**资源耗尽:** 要处理1万个客户端连接请求,TCPServer需要同时创建、维护和管理几万个 socket 连接,这将耗尽服务器的 CPU、内存、网络带宽等资源,导致性能下降甚至系统崩溃。
- 2、**连接请求丢失:** 当有大量连接请求同时涌入时,TCPServer可能会出现连接请求丢失的情况,因为服务端在处理请求时只能顺序处理,在等待处理一个请求时可能会错过一些连接请求。
- 3、**连接队列溢出:** 通过 backlog 参数设置的连接队列是有限制的,当同时涌入的连接请求超过队列长度时,新的连接请求将被服务器忽略或者拒绝。
- 4、**并发处理瓶颈:** 即使处理器和内存资源够用,TCP Server 也可能存在并发处理瓶颈,处理请求的线程或进程总是处于繁忙状态,无法及时响应新的请求,进一步影响系统性能。
- 5、**同步阻塞,处理效率低下:** 在处理连接请求时,TCPServer采用的是同步阻塞 I/O 模型,每一个连接请求都需要等待当前传输的结束,这会导致TCPServer处理的效率比较低,无法抵御峰值请求量的冲击。

因此,当遇到1万个并发客户端连接请求时,使用TCPServer容易导致系统性能下降、请求丢失、连接队列溢出以及无法及时响应客户端连接等问题。为了提高网络服务的处理能力,处理大量的并发请求,应该采用更加高效、低延迟、大吞吐量的框架,例如 NIO 框架。

### Task 4:

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。

```
服务端已启动
已连接: /127.0.0.1: 63423
服务端已收到消息: 用户名: 10215501435;
```

该代码中的I/O 多路复用使用了 Java NIO 库中的 Selector 类和 SelectionKey类,调用了以下几个系统调用API:

1. `select()`: `select` 函数是最常用的 I/O 多路复用函数,支持在单个线程中同时监听多个文件描述符的 I/O 事件。使用 `select` 函数需要手动维护文件描述符集合,由操作系统在内核中维护等待的数据,再由使用者进行轮询。
2. `poll()`: `poll` 函数是与 `select` 函数类似的 I/O 多路复用函数,它也可以同时监听多个文件描述符的 I/O 事件。不同于 `select` 函数需要手动维护描述符集合,`poll` 函数采用传入一个指针指向内核分配的开辟的数组的方式维护文件描述符集合。
3. `accept()`: 可以获取连接请求,并返回一个新的文件描述符,应用程序可以基于该文件描述符与新的客户端建立连接并进行通信。

4. `read()`: 当一个文件描述符变得可读时, 应用程序可以使用 `read()` 函数从这个文件描述符中读取数据, 调用该函数后, 将从指定文件描述符中读取指定字节数的数据, 并将这些数据缓冲区复制到应用程序的缓冲区中。

5. `write()`: 当一个文件描述符变得可写时, 应用程序可以使用 `write()` 函数向这个文件描述符写入数据, 将数据缓冲区的数据写入操作系统的缓冲区中, 操作系统将保证数据最终会被写入到指定的文件描述符中。

在 I/O 多路复用模型中, `Selector` 类主要用于管理多个 `Channel`, 以实现更高效的 I/O 数据传输, 并减少 CPU 使用率和系统资源占用率。

## 五、总结

本次实验难度较大, 通过本次试验, 我了解了阻塞 I/O 和非阻塞 I/O 以及 `NIO` 与 `TCP` 的区别和特点, 但是很多内容和代码实现还是没有完全清楚, 写作业时遇到很多困难, 后续的学习中会更加注意。