

深圳大学实验报告

课程名称： 互联网编程

实验项目名称： 多线程/线程池 TCP 服务器端程序设计

学院： 计算机与软件学院

专业： 腾班

指导教师： 周宇

报告人： 叶茂林 学号： 2021155015 班级： 腾班

实验时间： 2023.4.16-2023.4.29

实验报告提交时间： 2023.4.29

教务处制

一、实验目的与内容：

目的：熟悉 java 线程编程技术，掌握线程技术在 JAVA 互联网通信程序中的应用。

内容要求：

1. 多线程 TCP 服务器（30 分）：

设计编写一个 TCP 服务器端程序，需使用多线程处理客户端的连接请求。客户端与服务端之间的通信内容，以及服务器端的处理功能等可自由设计拓展，无特别限制和要求。

2. 线程池 TCP 服务器（30 分）：

设计编写一个 TCP 服务器端程序，需使用线程池处理客户端的连接请求。客户端与服务端之间的通信内容，以及服务器端的处理功能等可自由设计拓展，无特别限制和要求，但应与第 1 项要求中的服务器功能一致，便于对比分析。

3. 比较分析不同编程技术对服务器性能的影响（20 分）：

自由编写客户端程序和设计测试方式，对 1 和 2 中的服务器端程序进行测试，分析比较两个服务器的并发处理能力。

4. 设计编写可重用的服务器日志程序模块，日志记录的内容和日志存储方式可自定（比如可以记录客户端的连接时间、客户端 IP 等，日志存储为.TXT 或.log 文件等），分别在 1 和 2 的服务器程序中调用该日志程序模块，使多线程 TCP 服务器和线程池 TCP 服务器都具备日志功能，注意线程之间的同步操作处理。（20 分）

注意：

1. 实验报告中需要有实验结果的截屏图像。

二、实验过程和代码与结果

1. 给出满足内容要求 1 的程序源码及运行结果，简述思路或实验过程。

首先编写通讯功能，对 Thread 派生子类，覆盖其 run() 方法，从 socket 连接的输入流中读取信息并输出，如图 1 所示，附件已含源代码。

```
class MultiThread extends Thread {
    3 个用法
    private Socket socket = null;

    1 个用法
    public MultiThread(Socket socket) { this.socket = socket; }

    public void run() {
        try {
            BufferedReader input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            String message = null;
            while ((message = input.readLine()) != null) {
                System.out.println(message);
            }
            input.close();
            socket.close();
        } catch (IOException error) {
            error.printStackTrace();
        }
    }
}
```

图 1

然后编写 TCP 服务器，创建 Server Socket 在端口 8888 监听，将最大同时连接数量设置为

10000, 然后等待客户端发起连接, 使用多线程处理客户端的连接请求, 每收到一个连接请求就创建一个新的线程, 如图 2 所示, 附件已含源代码。

```
public class MultithreadingServer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket( port: 8888, backlog: 10000);
            while (true) {
                Socket client = serverSocket.accept();
                new MultiThread(client).start();
            }
        } catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }
}
```

图 2

编写一个简单的客户端对其进行测试, 对服务器发起 1000 次连接, 每个连接重复发送 1000 条信息, 测试结果如图 3 所示, 通信功能正常, 多线程响应连接正常。

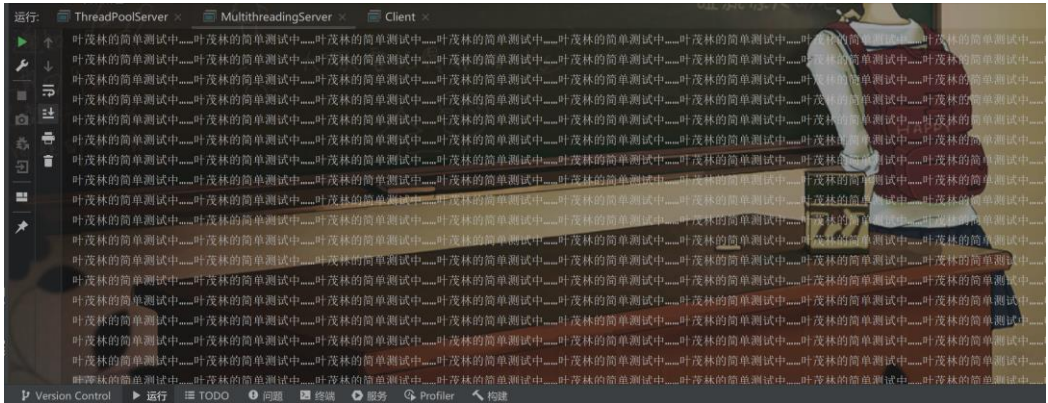


图 3

2. 给出满足内容要求 2 的程序源码及运行结果, 简述思路或实验过程。

首先编写通讯功能, 实现 Runnable 接口, 从 socket 连接的输入流中读取信息并输出, 如图 4 所示, 附件已含源代码。

```
class ThreadPooTask implements Runnable{
    3个用法
    private Socket socket=null;
    1个用法
    public ThreadPooTask(Socket socket) { this.socket=socket; }
    public void run(){
        try{
            BufferedReader input=new BufferedReader(new InputStreamReader(socket.getInputStream()));
            String message=null;
            while((message=input.readLine())!=null){
                System.out.println(message);
            }
            input.close();
            socket.close();
        }catch (IOException error){
            error.printStackTrace();
        }
    }
}
```

图 4

然后编写 TCP 服务器，创建 Server Socket 在端口 9999 监听，将最大同时连接数量设置为 10000，使用线程池处理客户端的连接请求，利用 Executor Service 构造一个线程池，线程数固定为 200，然后等待客户端发起连接，每收到一个客户端连接就交给线程池处理，如图 5 所示，附件已含源代码。

```
public class ThreadPoolServer {
    public static void main(String[] args) {
        ExecutorService executorService= Executors.newFixedThreadPool( nThreads: 200);
        try{
            ServerSocket serverSocket=new ServerSocket( port: 9999, backlog: 10000);
            while(true){
                Socket client=serverSocket.accept();
                executorService.execute(new ThreadPoolTask(client));
            }
        } catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }finally {
            executorService.shutdown();
        }
    }
}
```

图 5

用编写好的客户端程序对其进行测试，创建 1000 个线程进行连接，每个连接重复发送 1000 条信息，测试结果如图 6 所示，通讯功能正常，线程池响应正常。

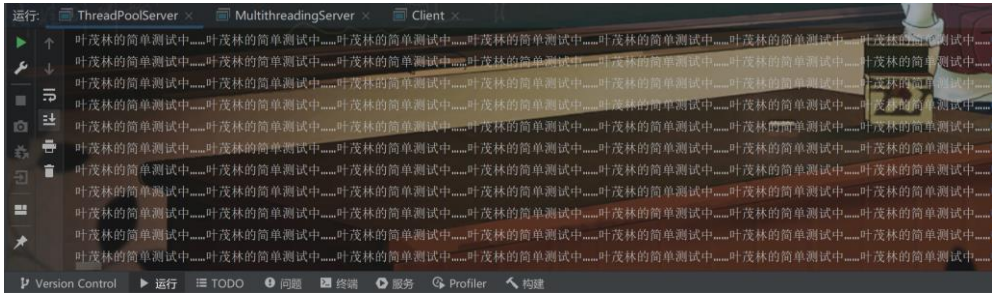


图 6

3.按内容要求 3 给出测试结果及分析对比情况，阐述测试方法或过程。

编写一个压力测试程序，测试多线程服务器和线程池服务器在高并发时的表现，实现 run() 方法，发起连接请求，为了保证多线程并发访问 long 类型变量时的线程安全性，使用线程安全的 AtomicLong 类来记录服务器响应的时间，如图 7 所示，附件已含源代码。

```
public void run() {
    try {
        Socket socket = new Socket();
        SocketAddress socketAddress=new InetSocketAddress(InetAddress.getLocalHost(), port);
        long start=System.currentTimeMillis();
        socket.connect(socketAddress);
        while(!socket.isConnected()){
            socket.close();
            long end=System.currentTimeMillis();
            if(port==9999)
                timePool.addAndGet( delta: end-start);
            else timeMulti.addAndGet( delta: end-start);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit( status: -1);
        }
    }
}
```

图 7

主程序同时向两个服务器发起多个连接,连接规模从 1000 个连接请求开始一直增加到 10000 个,通过在短时间内发起大量连接请求来对服务器进行压力测试,如图 8 所示。

```
public class TestServerClient {
    public static void main(String[] args) {
        for (int power = 1; power <= 10; power++) {
            int scale = 2000*power ;
            for (int i = 0; i < 2000; i++) {
                if(i%2==0)
                    new TestTask( port: 9999).run();
                else new TestTask( port: 8888).run();
            }
            System.out.printf("线程池: 规模为%d消耗时间%d毫秒\n", scale/2, TestTask.timePool.get());
            System.out.printf("多线程: 规模为%d消耗时间%d毫秒\n", scale/2, TestTask.timeMulti.get());
        }
    }
}
```

图 8

测试过程数据如图 9 所示。

```
运行: ThreadPoolServer x MultithreadingServer x TestServerClient x
"C:\Program Files\Java\jdk-18.0.1.1\bin\java.exe" "-javaagent:C:\Program Files\JetBrains
线程池: 规模为1000消耗时间142毫秒
多线程: 规模为1000消耗时间719毫秒
线程池: 规模为2000消耗时间269毫秒
多线程: 规模为2000消耗时间1405毫秒
线程池: 规模为3000消耗时间551毫秒
多线程: 规模为3000消耗时间1556毫秒
线程池: 规模为4000消耗时间735毫秒
多线程: 规模为4000消耗时间1825毫秒
线程池: 规模为5000消耗时间1046毫秒
多线程: 规模为5000消耗时间1952毫秒
线程池: 规模为6000消耗时间1234毫秒
多线程: 规模为6000消耗时间2248毫秒
线程池: 规模为7000消耗时间1487毫秒
多线程: 规模为7000消耗时间2573毫秒
线程池: 规模为8000消耗时间1907毫秒
多线程: 规模为8000消耗时间2715毫秒
线程池: 规模为9000消耗时间2207毫秒
多线程: 规模为9000消耗时间3025毫秒
线程池: 规模为10000消耗时间2458毫秒
多线程: 规模为10000消耗时间3368毫秒
进程已结束,退出代码0
```

图 9

分析两个服务器的表现情况,如图 10 所示,可见在处理大量短任务(如处理网络请求)的情况下,使用线程池可以避免频繁地创建、销毁线程所带来的开销,因此会更快一些。

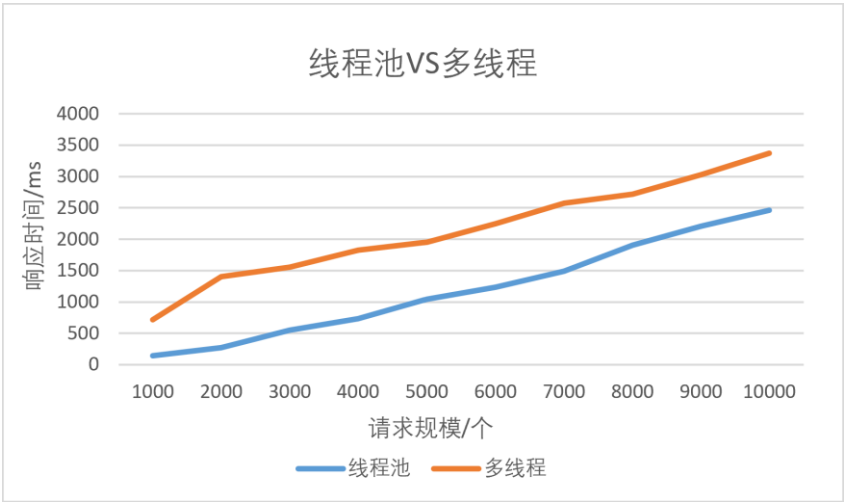


图 10

4.给出满足内容要求 4 的实验结果，包括源码及两个服务器增加了日志功能后，日志记录运行结果。

编写可重用的服务器日志程序模块，定义 Logger 类，记录用户登录的 IP 地址和登录时间，根据传入的文件路径写入 txt 文件，由于多个线程可能会同时访问日志记录，所以使用 synchronize 同步锁修饰写入文件方法，保证每个线程对共享资源的访问按照一定顺序进行，保证线程安全性，如图 11 所示，附件已含源代码。

```
public class Logger {
    2个用法
    private final String log;
    2个用法
    private final String filePath;
    2个用法
    public Logger(String IP, Date date,String filePath){
        log=IP+"Login at "+date.toString()+"\n";
        this.filePath=filePath;
        keep();
    }
    1个用法
    private synchronized void keep(){
        try {
            BufferedWriter bufferedWriter=new BufferedWriter(new FileWriter(filePath, append(true)));
            bufferedWriter.write(log);
            bufferedWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

图 11

在多线程服务器增加日志功能，将客户端的连接 IP 和连接时间以及日志文件保存地址的路径传入 Logger，如图 12 所示。

```
public class MultithreadingServer {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket( port: 8888, backlog: 10000);
            while (true) {
                Socket client = serverSocket.accept();
                new MultiThread(client).start();
                new Logger(client.getInetAddress().getHostAddress(), new Date(), filePath: "LogMultithreadingServer.txt");
            }
        } catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
    }
}
```

图 12

同样的，在线程池服务器增加日志功能，如图 13 所示。

```
public class ThreadPoolServer {
    public static void main(String[] args) {
        ExecutorService executorService= Executors.newFixedThreadPool( nThreads: 200);
        try{
            ServerSocket serverSocket=new ServerSocket( port: 9999, backlog: 10000);
            while(true){
                Socket client=serverSocket.accept();
                executorService.execute(new ThreadPoolTask(client));
                new Logger(client.getInetAddress().getHostAddress(),new Date(), filePath: "LogThreadPoolServer.txt");
            }
        } catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }finally {
            executorService.shutdown();
        }
    }
}
```

图 13

两个服务器的日志记录结果如图 14 所示。

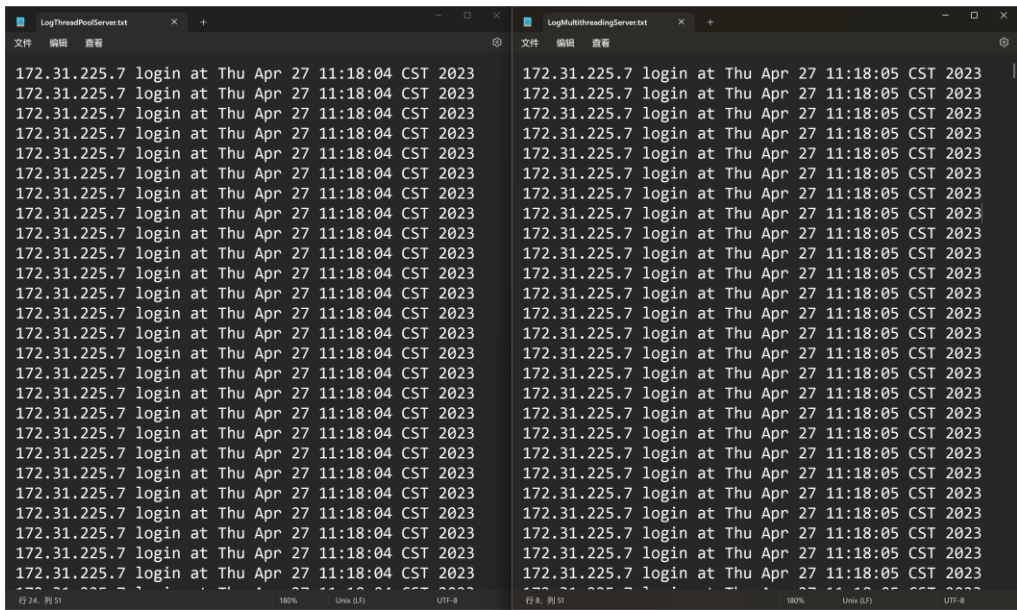


图 14

三、实验总结

本次实验我成功搭建了分别使用多线程和线程池处理客户端的连接请求的 TCP 服务器端程序，并编写了可重用的服务器日志模块。

在实验的过程中，我发现使用多线程和线程池都能够提高服务器的并发处理能力，但两者的实现方式和工作原理有很大的不同。多线程方式下，每个客户端连接请求都会新建一个线程来处理，这种方式操作简单易懂，但会带来大量线程的创建和销毁开销。线程池则可以复用已经创建好的线程，减少线程数量和线程创建和销毁所带来的开销。而在测试比较中，我发现线程池 TCP 服务器的并发处理能力明显优于多线程 TCP 服务器。

此外，我也意识到了日志记录在服务器端的重要性。通过设计编写可重用的服务器日志程序模块，并在多线程 TCP 服务器和线程池 TCP 服务器中调用，可以使服务器具备完善的日志记录功能，方便服务器维护和故障排查。

在设计客户端进行压力测试的时候，一开始用线程池跑的，连接的线程从 1000 跑到 5000，但是多次报错 `java.net.BindException: Address already in use`，如图 15 所示。

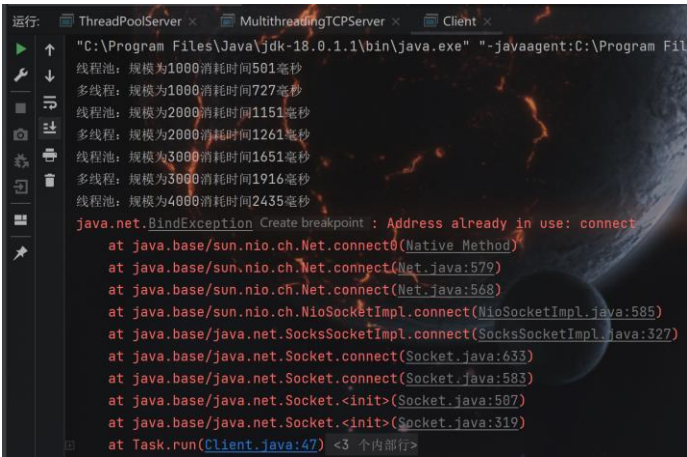
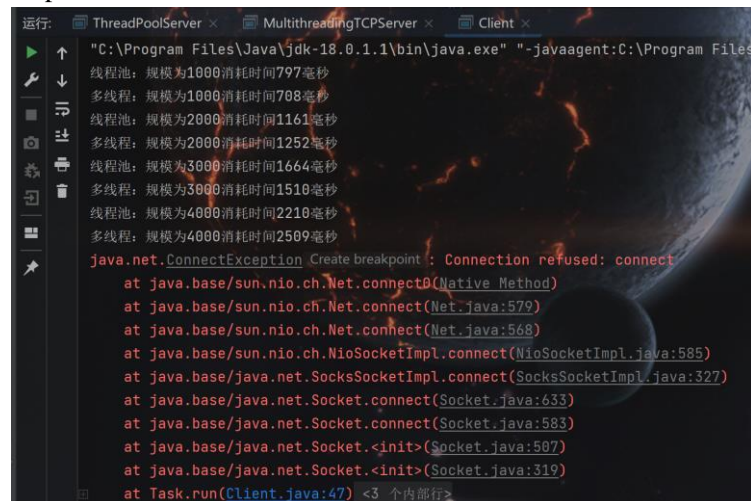


图 15

搜遍全网，有说端口被别的线程占用的，因为短时间内 new socket 操作过多，而 socket.close() 操作不能立即释放绑定的端口，而是把端口设置为 TIME_WAIT 状态，要过 240s 才释放，在命令行用 netstat -na 命令查看，果然，非常多端口处于 TIME_WAIT 状态，网上绝大部分让我修改系统对可使用端口数的限制和加快被占用端口的释放速度，实际我试了很多遍，修改系统对可使用端口数的限制只能缓解端口被占用的问题，而加快被占用端口的释放速度没有什么用处，还是存在大量的 TIME_WAIT。

没办法，只能修改系统配置，增加可用端口缓解一下，后来又出现了新的报错信息：java.net.ConnectException: Connection refused: connect，如图 16 所示。



```
运行: ThreadPoolServer x MultithreadingTCPServer x Client x
"C:\Program Files\Java\jdk-18.0.1\bin\java.exe" "-javaagent:C:\Program Files
线程池: 规模为1000消耗时间797毫秒
多线程: 规模为1000消耗时间708毫秒
线程池: 规模为2000消耗时间1161毫秒
多线程: 规模为2000消耗时间1252毫秒
线程池: 规模为3000消耗时间1664毫秒
多线程: 规模为3000消耗时间1510毫秒
线程池: 规模为4000消耗时间2210毫秒
多线程: 规模为4000消耗时间2509毫秒
java.net.ConnectException Create breakpoint : Connection refused: connect
    at java.base/sun.nio.ch.Net.connect0(Native Method)
    at java.base/sun.nio.ch.Net.connect(Net.java:579)
    at java.base/sun.nio.ch.Net.connect(Net.java:568)
    at java.base/sun.nio.ch.NioSocketImpl.connect(NioSocketImpl.java:585)
    at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:327)
    at java.base/java.net.Socket.connect(Socket.java:633)
    at java.base/java.net.Socket.connect(Socket.java:583)
    at java.base/java.net.Socket.<init>(Socket.java:507)
    at java.base/java.net.Socket.<init>(Socket.java:319)
    at Task.run(Client.java:47) <3 个内部行>
```

图 16

网上搜到的结果说是端口号被占用的结果，真是苦笑不得，估计是和前面一样的问题，能查到的解决方法都用过了，还是没有解决，我自己已经折腾了几天了，只能请求老师帮助了，后来还是不行，我还是在不断测试各种方法修改我的客户端，修改测试方法，突然想到，端口不够用的问题是在端口使用规模大的情况才会出现，我的两个服务器都会占用大量的端口，而我的客户端又使用了线程池，这进一步加剧了端口紧缺，于是最后我决定放弃客户端使用线程池进行测试，采用普通循环，这样照样可以在短时间内发起大量连接请求，达到压力测试的目的，这样我的请求数量可以达到 10000 以上。

通过这次实验，我对多线程和线程池有了更深入的了解，也对 TCP 连接和线程的使用有了更深刻的理解，总的来说，这是一次非常棒的体验。

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字： 年 月 日</p>	
<p>备注：</p>	

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。