中南大學
CENTRAL SOUTH UNIVERSITY

分布式与云计算
实验报告

学　　院：_____

专业班级：_____

学生姓名：abc

学　　号：_____

指导教师：_____

年 月 23 日

目录

一 数据包 socket 应用

实验目的

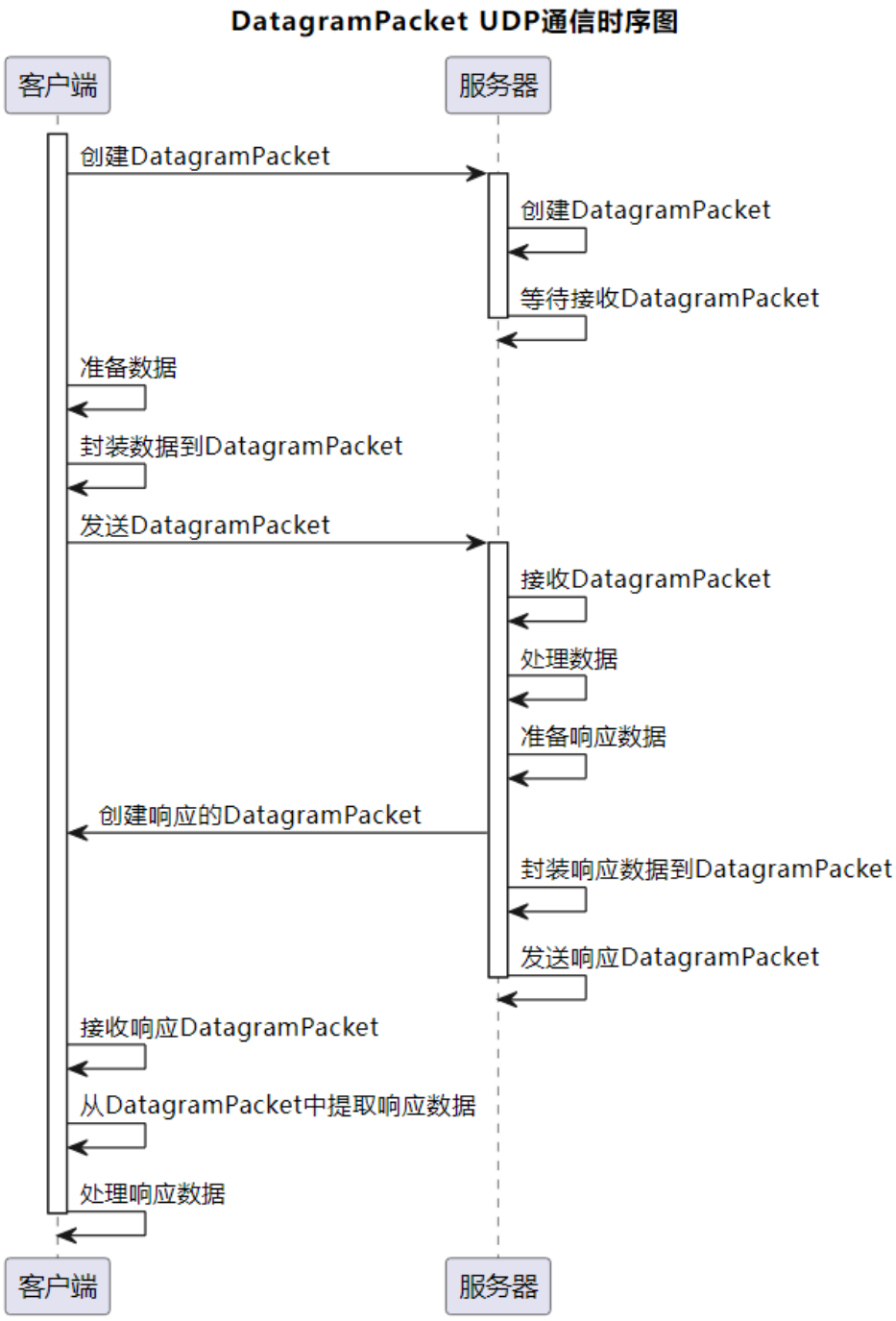
1. 理解数据包 s o c k e t 的 应 用
2. 实现数据包 s o c k e t 通信
3. 了解 J a va 并 行 编 程 的 基 本 方 法

实验要求

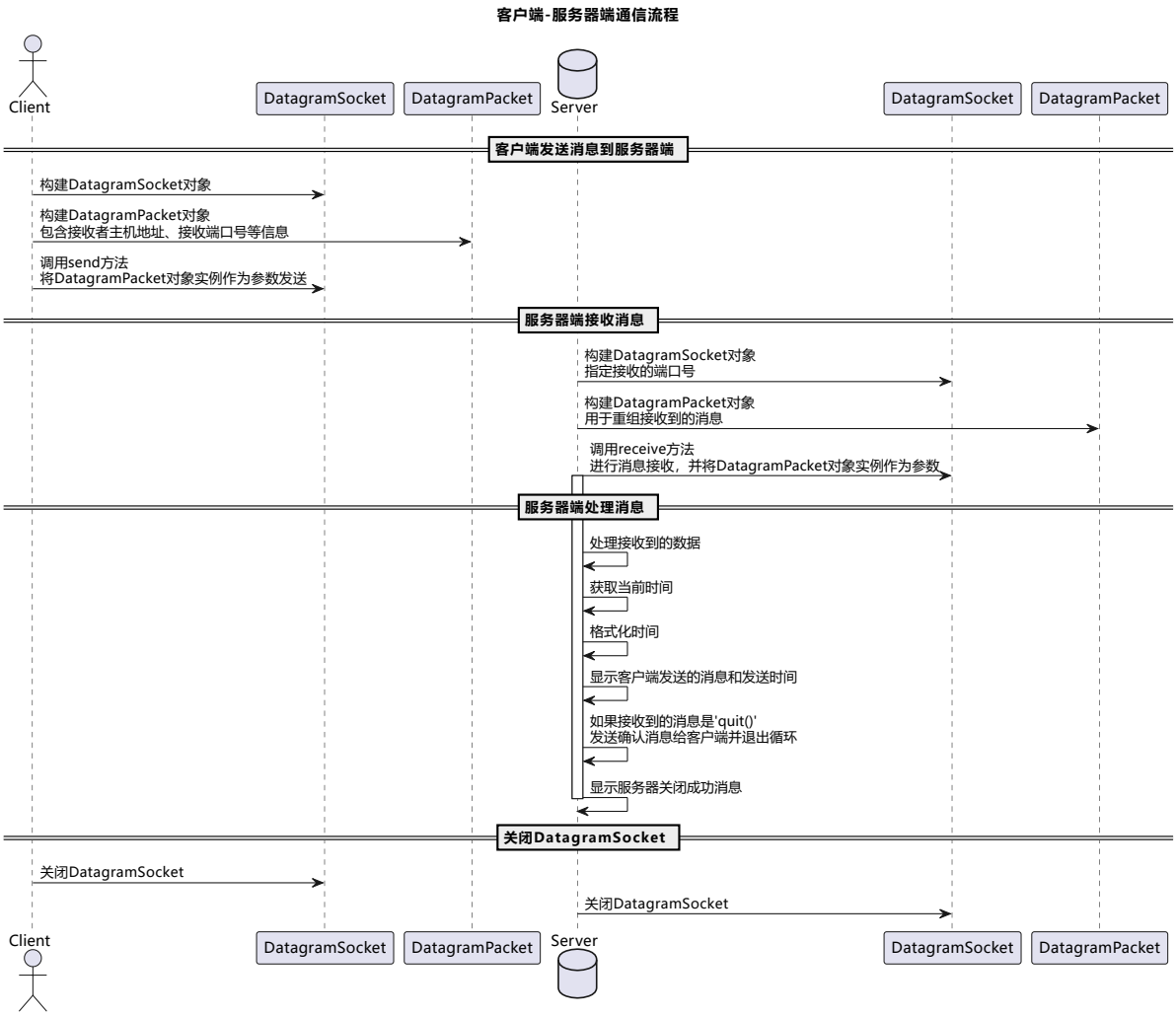
1. 预习实验指导书及教材的有关内容，了解数据包 socket 的通信原理；
2. 熟悉一种 java IDE 和程序开发过程；
3. 了解下列 Java API：Thread、Runnable；
4. 尽可能独立思考并完成实验。

实验内容

总体设计


DatagramPacket UDP通信时序图
客户端
服务器
创建DatagramPacket
创建DatagramPacket
等待接收DatagramPacket
准备数据
封装数据到DatagramPacket
发送DatagramPacket
接收DatagramPacket
处理数据
准备响应数据
创建响应的DatagramPacket
封装响应数据到DatagramPacket
发送响应DatagramPacket
接收响应DatagramPacket
从DatagramPacket中提取响应数据
处理响应数据
客户端
服务器

用例图：



构建客户端程序

（1）构建 datagramSocket 对象实例

（2）构建 DatagramPacket 对象实例，并包含接收者主机地址、接收端口号等信息

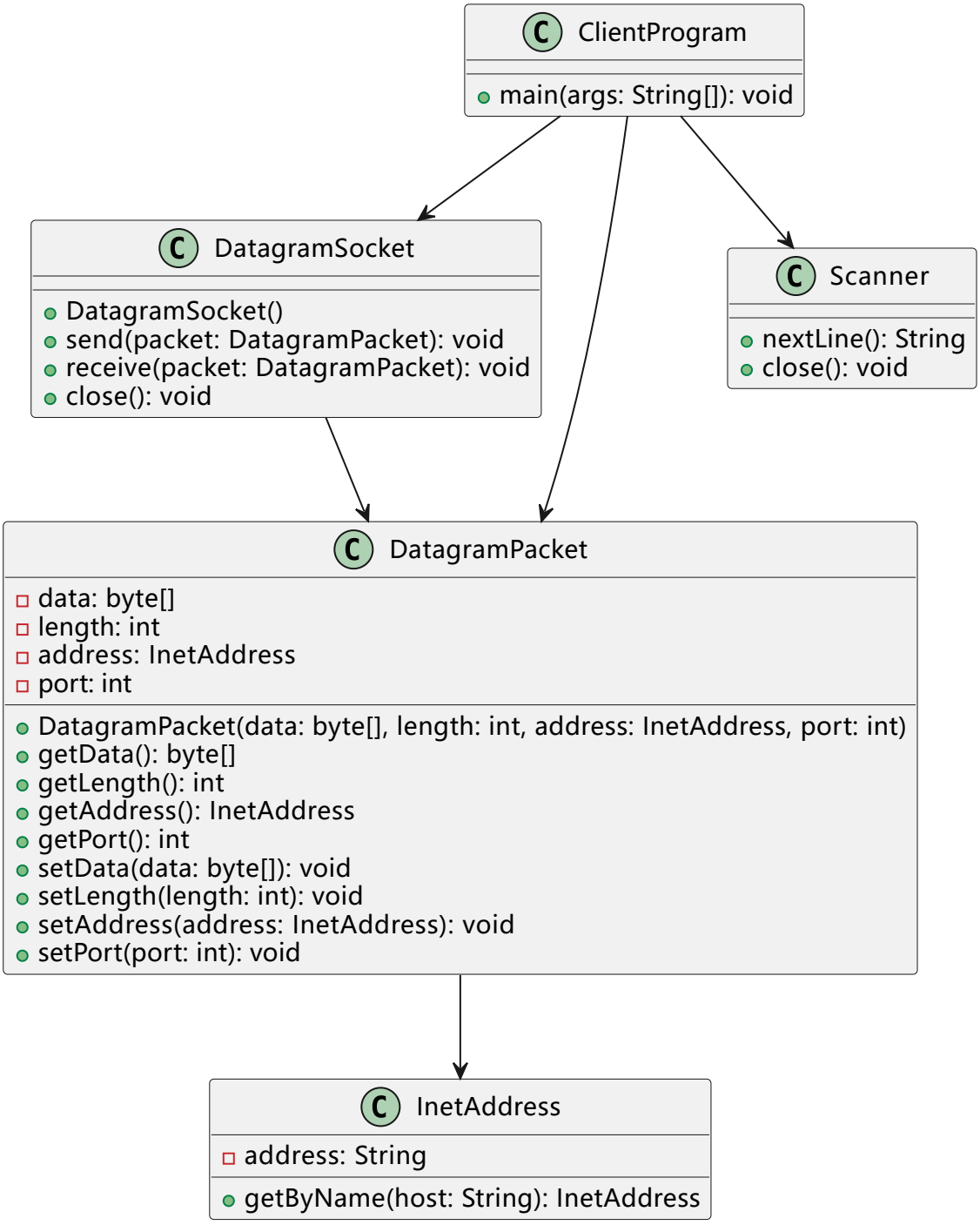
（3）调用 datagramSocket 对象实例的 send 方法，将 DatagramPacket 对象实例作为参 数发送。

伪代码如下：

```
// 1 构建 DatagramSocket 对象实例
DatagramSocket datagramSocket = new DatagramSocket();

// 2 构建 DatagramPacket 对象实例，包含接收者主机地址、接收端口号等信息
InetAddress receiverAddress = InetAddress.getByName("服务器主机地址");
int receiverPort = 12345;
byte[] data = "Hello, Server!".getBytes();
DatagramPacket datagramPacket = new DatagramPacket(data, data.length,
receiverAddress, receiverPort);
```

```
9
10 // 3 调用 datagramSocket 对象实例的 send 方法，将 DatagramPacket 对象实例作为参数发送
11 datagramSocket.send(datagramPacket);
12
13 // 关闭 DatagramSocket
14 datagramSocket.close();
15
```

具体实现：




```
1 import java.net.*;
2 import java.util.Scanner;
```

```java

public class ClientProgram {
    public static void main(String[] args) {
        try {
            // （1）构建 DatagramSocket 对象实例
            DatagramSocket datagramSocket = new DatagramSocket();

            // 获取用户输入
            Scanner scanner = new Scanner(System.in);

            while (true) {
                System.out.print("请输入要发送的消息: ");
                String message = scanner.nextLine();

                // （2）构建 DatagramPacket 对象实例，包含接收者主机地址、接收端口号等信息
                InetAddress receiverAddress = InetAddress.getByName("localhost"); // 或者 InetAddress.getByName("127.0.0.1");
                int receiverPort = 12345;
                byte[] data = message.getBytes();
                DatagramPacket datagramPacket = new DatagramPacket(data, data.length, receiverAddress, receiverPort);

                // （3）调用 datagramSocket 对象实例的 send 方法，将 DatagramPacket 对象实例作为参数发送
                datagramSocket.send(datagramPacket);

                // 如果用户输入'quit()'，退出循环
                if (message.equals("quit()")) {
                    break;
                }
            }

            // 关闭 Scanner 和 DatagramSocket
            scanner.close();
            datagramSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

构建服务器端程序

（1）构建 datagramSocket 对象实例，指定接收的端口号。

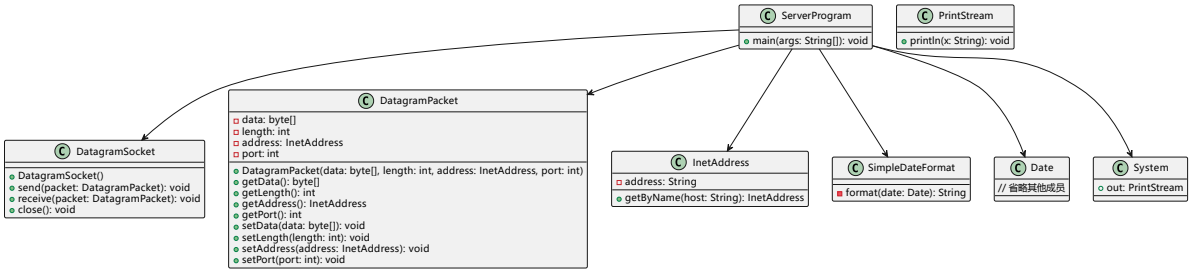
（2）构建 DatagramPacket 对象实例，用于重组接收到的消息。

（3）调用 datagramSocket 对象实例大家 receive 方法，进行消息接收，并将 DatagramPacket 对象实例作为参数。

伪代码如下：

```
// 1 构建 DatagramSocket 对象实例，指定接收的端口号
int serverPort = 12345;
DatagramSocket datagramSocket = new DatagramSocket(serverPort);

// 2 构建 DatagramPacket 对象实例，用于重组接收到的消息
byte[] receiveData = new byte[1024];
DatagramPacket receivePacket = new DatagramPacket(receiveData,
receiveData.length);

// 3 调用 datagramSocket 对象实例的 receive 方法，进行消息接收，并将 DatagramPacket
对象实例作为参数
datagramSocket.receive(receivePacket);

// 处理接收到的数据
String receivedMessage = new String(receivePacket.getData(), 0,
receivePacket.getLength());
System.out.println("Received message from client: " + receivedMessage);

// 关闭 DatagramSocket
datagramSocket.close();

```

具体实现：




```
import java.net.*;
import java.text.SimpleDateFormat;
import java.util.Date;

public class ServerProgram {
    public static void main(String[] args) {
        try {
            // （1）构建 DatagramSocket 对象实例，指定接收的端口号
            int serverPort = 12345;
            DatagramSocket datagramSocket = new DatagramSocket(serverPort);

            // 显示服务器启动成功消息
            System.out.println("服务器启动成功，等待客户端连接...");

            while (true) {
                // （2）构建 DatagramPacket 对象实例，用于重组接收到的消息
```

```
                byte[] receiveData = new byte[1024];
                DatagramPacket receivePacket = new
DatagramPacket(receiveData, receiveData.length);

                // （3）调用 datagramSocket 对象实例的 receive 方法，进行消息接收，
并将 DatagramPacket 对象实例作为参数
                datagramSocket.receive(receivePacket);

                // 处理接收到的数据
                String receivedMessage = new String(receivePacket.getData(),
0, receivePacket.getLength());

                // 获取当前时间
                Date currentTime = new Date();
                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");
                String formattedTime = dateFormat.format(currentTime);

                // 显示客户端发送的消息和发送时间
                System.out.println("从客户端收到消息(" + formattedTime + "): "
+ receivedMessage);

                // 如果接收到的消息是'quit()'，发送确认消息给客户端并退出循环
                if (receivedMessage.equals("quit()")) {
                    System.out.println("向客户端发送确认消息...");
                    byte[] confirmationData = ("服务器收到 'quit()'，退出中。" +
"\n服务器确认时间: " + formattedTime).getBytes();
                    DatagramPacket confirmationPacket = new
DatagramPacket(confirmationData, confirmationData.length,
receivePacket.getAddress(), receivePacket.getPort());
                    datagramSocket.send(confirmationPacket);

                    // 显示服务器关闭成功消息
                    System.out.println("服务器关闭成功。");
                    break;
                }
            }

            // 关闭 DatagramSocket
            datagramSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

实验结果

服务端

```
服务器启动成功，等待客户端连接...
从客户端收到消息(2023-12-23 10:39:02): hi
从客户端收到消息(2023-12-23 10:39:05): 123
从客户端收到消息(2023-12-23 10:39:09): 我和你
从客户端收到消息(2023-12-23 10:39:13): !@#
从客户端收到消息(2023-12-23 10:39:25): quit()
向客户端发送确认消息...
服务器关闭成功。
```

客户端

```
客户端启动成功！ (输入'quit()'退出)
请输入要发送的消息: hi
请输入要发送的消息: 123
请输入要发送的消息: 我和你
请输入要发送的消息: !@#
请输入要发送的消息: quit()
```

思考题

1、如何避免数据包丢失而造成的无限等待问题?

在UDP通信中，由于其无连接性和不可靠性，数据包可能会丢失。为了避免由于数据包丢失导致的无限等待问题，可以在服务器端设置一个超时机制，即通过设置`DatagramSocket`的超时时间，使其在一定时间内没有接收到数据包时抛出`SocketTimeoutException`，从而可以进行适当的处理。

代码实现，添加了超时机制：

通过调用`datagramSocket.setSoTimeout(5000)`设置了超时时间为5秒。在等待客户端连接时，如果超过5秒没有接收到数据包，就会捕获到`SocketTimeoutException`异常，然后可以在异常处理块中进行适当的处理，例如重新等待或退出循环。这样可以避免由于数据包丢失而导致的无限等待问题。

```java
import java.net.*;
import java.text.SimpleDateFormat;
import java.util.Date;

public class ServerProgram {
    public static void main(String[] args) {
        try {
            // 构建 DatagramSocket 对象实例，指定接收的端口号
            int serverPort = 12345;
```

```
            DatagramSocket datagramSocket = new DatagramSocket(serverPort);

            // 设置超时时间为5秒
            datagramSocket.setSoTimeout(5000);

            // 显示服务器启动成功消息
            System.out.println("服务器启动成功，等待客户端连接...");

            while (true) {
                // 构建 DatagramPacket 对象实例，用于重组接收到的消息
                byte[] receiveData = new byte[1024];
                DatagramPacket receivePacket = new
DatagramPacket(receiveData, receiveData.length);

                try {
                    // 调用 datagramSocket 对象实例的 receive 方法，进行消息接收，
并将 DatagramPacket 对象实例作为参数
                    datagramSocket.receive(receivePacket);
                } catch (SocketTimeoutException e) {
                    // 在超时时处理逻辑，例如重新等待或退出循环
                    System.out.println("等待客户端连接超时...");
                    continue;
                }

                // 处理接收到的数据
                String receivedMessage = new String(receivePacket.getData(),
0, receivePacket.getLength());

                // 获取当前时间
                Date currentTime = new Date();
                SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-
dd HH:mm:ss");
                String formattedTime = dateFormat.format(currentTime);

                // 显示客户端发送的消息和发送时间
                System.out.println("从客户端收到消息(" + formattedTime + "): "
+ receivedMessage);

                // 如果接收到的消息是'quit()'，发送确认消息给客户端并退出循环
                if (receivedMessage.equals("quit()")) {
                    System.out.println("向客户端发送确认消息...");
                    byte[] confirmationData = ("服务器收到 'quit()'，退出中。" +
"\n服务器确认时间: " + formattedTime).getBytes();
                    DatagramPacket confirmationPacket = new
DatagramPacket(confirmationData, confirmationData.length,
receivePacket.getAddress(), receivePacket.getPort());
                    datagramSocket.send(confirmationPacket);

                    // 显示服务器关闭成功消息
                    System.out.println("服务器关闭成功。");
                    break;
                }
            }

            // 关闭 DatagramSocket
            datagramSocket.close();
```

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

2、如何实现全双工的数据包通信?

UDP通信是无连接的、不可靠的通信协议，它本身不提供全双工通信的机制。全双工通信意味着客户端和服务器可以同时发送和接收数据。在UDP中，一个`DatagramSocket`可以被用于发送和接收数据，但它不能同时发送和接收数据。

如果需要全双工通信，可以使用TCP协议，因为TCP提供了可靠的、面向连接的通信，并且能够实现全双工通信。在TCP中，一个`Socket`可以同时拥有输入流和输出流，允许客户端和服务器在同一个连接上进行双向通信。

代码实现：

服务器端：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class TCPServer {
    public static void main(String[] args) {
        try {
            // 创建ServerSocket，指定监听的端口号
            int serverPort = 12345;
            ServerSocket serverSocket = new ServerSocket(serverPort);
            System.out.println("服务器启动，等待客户端连接...");

            // 等待客户端连接
            Socket clientSocket = serverSocket.accept();
            System.out.println("客户端连接成功");

            // 获取输入流和输出流
            BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
            PrintWriter out = new
PrintWriter(clientSocket.getOutputStream(), true);

            // 从客户端读取数据，并发送响应
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println("从客户端收到消息: " + inputLine);

                // 向客户端发送响应
                out.println("服务器收到消息: " + inputLine);

                // 如果收到"quit()"，退出循环
```

```
                if (inputLine.equals("quit()")) {
                    break;
                }
            }

            // 关闭连接
            in.close();
            out.close();
            clientSocket.close();
            serverSocket.close();
            System.out.println("服务器关闭成功");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

客户端：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;

public class TCPClient {
    public static void main(String[] args) {
        try {
            // 创建Socket，指定连接的服务器地址和端口号
            String serverAddress = "localhost";
            int serverPort = 12345;
            Socket socket = new Socket(serverAddress, serverPort);

            // 获取输入流和输出流
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(),
true);

            // 从控制台读取用户输入，发送到服务器
            BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in));
            String userInputLine;
            while ((userInputLine = userInput.readLine()) != null) {
                out.println(userInputLine);

                // 从服务器读取响应并显示在控制台上
                String response = in.readLine();
                System.out.println("从服务器收到响应: " + response);

                // 如果输入"quit()"，退出循环
                if (userInputLine.equals("quit()")) {
                    break;
                }
            }

```

```
            // 关闭连接
            in.close();
            out.close();
            userInput.close();
            socket.close();
            System.out.println("客户端关闭成功");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

二 流式 socket 应用

实验目的

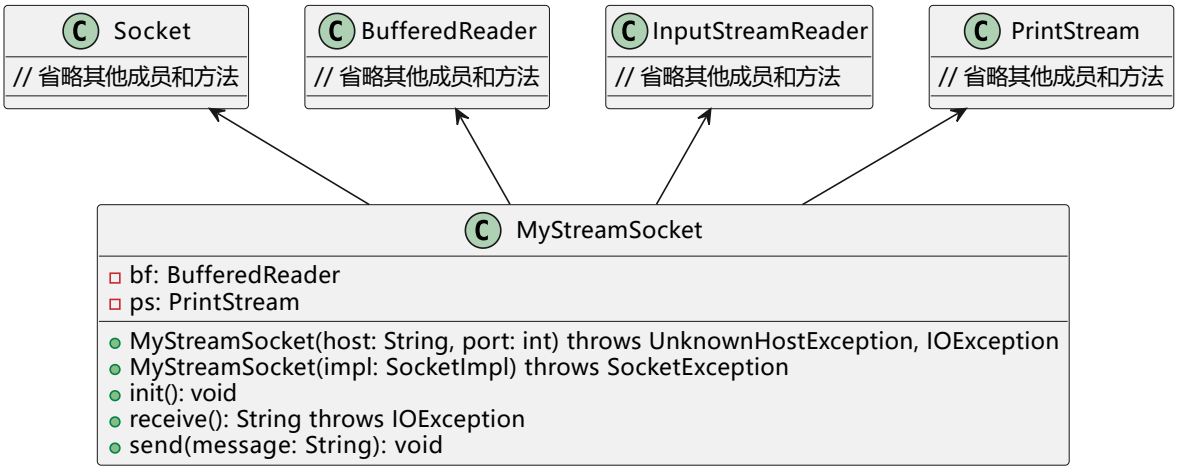
1. 理解流式 socket 的 原理
2. 实现流式 socket 通信

实验要求

1. 预习实验指导书及教材的有关内容，了解流式 socket 的通信原理；
2. 熟悉 java 环境和程序开发过程；
3. 尽可能独立思考并完成实验。

实验内容

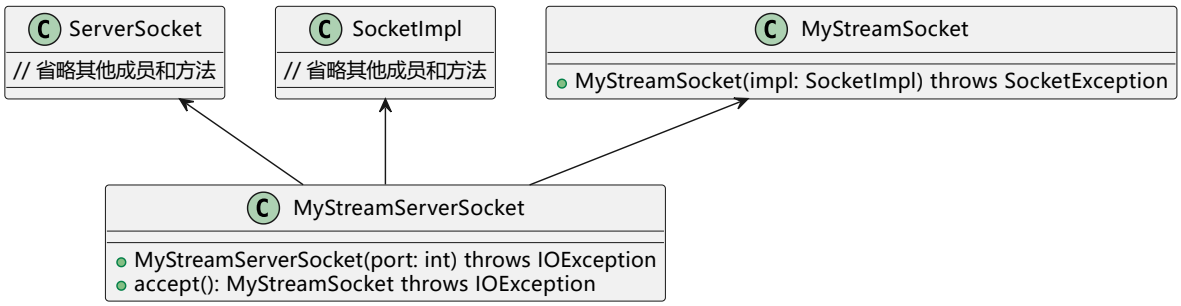
首先，需要继承自java.net.socket方法创建MyStreamSocket类，其中处理输入输出流以封装发送和接收数据的方法。输入流主要使用BufferedReader封装，输出流采用PrintStream，提供接收和发送的方法，和基于IP地址和端口号的构造函数。




```
public class MyStreamSocket extends Socket {
    // 用于从输入流中读取数据的缓冲读取器
    BufferedReader bf;

    // 用于将数据发送到输出流的打印流
    PrintStream ps;

    // 构造函数，接受主机名和端口号，可能抛出 UnknownHostException 和 IOException 异常
    public MyStreamSocket(String host, int port) throws UnknownHostException, IOException {
        super(host, port);
```

```
    }

    // 构造函数，接受 SocketImpl 对象，可能抛出 SocketException 异常
    public MyStreamSocket(SocketImpl impl) throws SocketException {
        super(impl);
    }

    // 初始化方法，可能抛出 IOException 异常
    public void init() throws IOException {
        // 初始化 BufferedReader，使用 Socket 的输入流
        bf = new BufferedReader(new InputStreamReader(getInputStream()));

        // 初始化 PrintStream，使用 Socket 的输出流
        ps = new PrintStream(getOutputStream());
    }

    // 接收方法，可能抛出 IOException 异常
    public String receive() throws IOException {
        // 从 BufferedReader 中读取一行数据并返回
        return bf.readLine();
    }

    // 发送方法，接受一个消息字符串作为参数
    public void send(String message) {
        // 使用 PrintStream 发送消息
        ps.println(message);
    }
}

```

由于在服务端，流式套接字的建立基于ServerSocket接受连接，使用ServerSocket.accept方法，返回值为Socket类型，而非我们创建的MyStreamSocket类，因此，需要另外创建类仿造jdk中ServerSocket.accept的方法，返回MyStreamSocket类型。

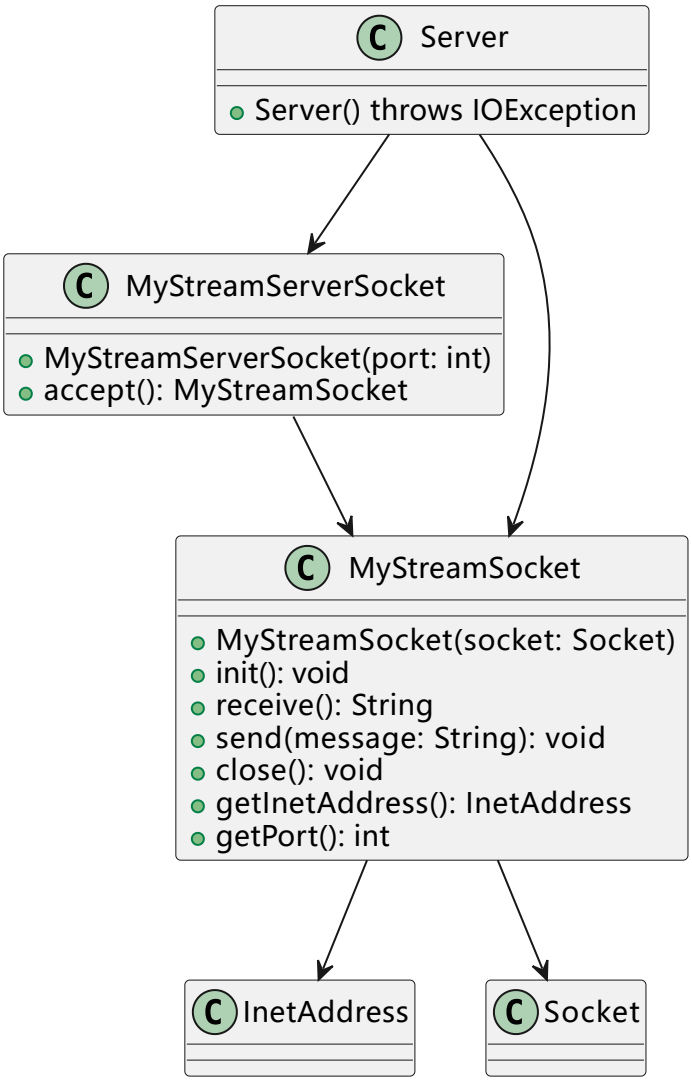



```
public class MyStreamServerSocket extends ServerSocket {

    // 构造函数，接受端口号，可能抛出 IOException 异常
    public MyStreamServerSocket(int port) throws IOException {
        super(port);
    }

    // 接受客户端连接请求，并返回一个 MyStreamSocket 对象表示连接
    public MyStreamSocket accept() throws IOException {
        // 如果套接字已关闭，抛出 SocketException 异常

```

```
        if (isClosed())
            throw new SocketException("Socket is closed");

        // 如果套接字未绑定，抛出 SocketException 异常
        if (!isBound())
            throw new SocketException("Socket is not bound yet");

        // 创建一个 MyStreamSocket 对象，使用默认的 SocketImpl
        MyStreamSocket s = new MyStreamSocket((SocketImpl) null);

        // 调用父类 ServerSocket 的 implAccept 方法接受连接
        implAccept(s);

        // 返回 MyStreamSocket 对象表示连接
        return s;
    }
}

```

服务端通过ServerSocket监听，连接时创建套接字进行通信。客户端则根据服务端的IP地址和端口号创建套接字。服务端和客户端均采用多线程来接收信息。

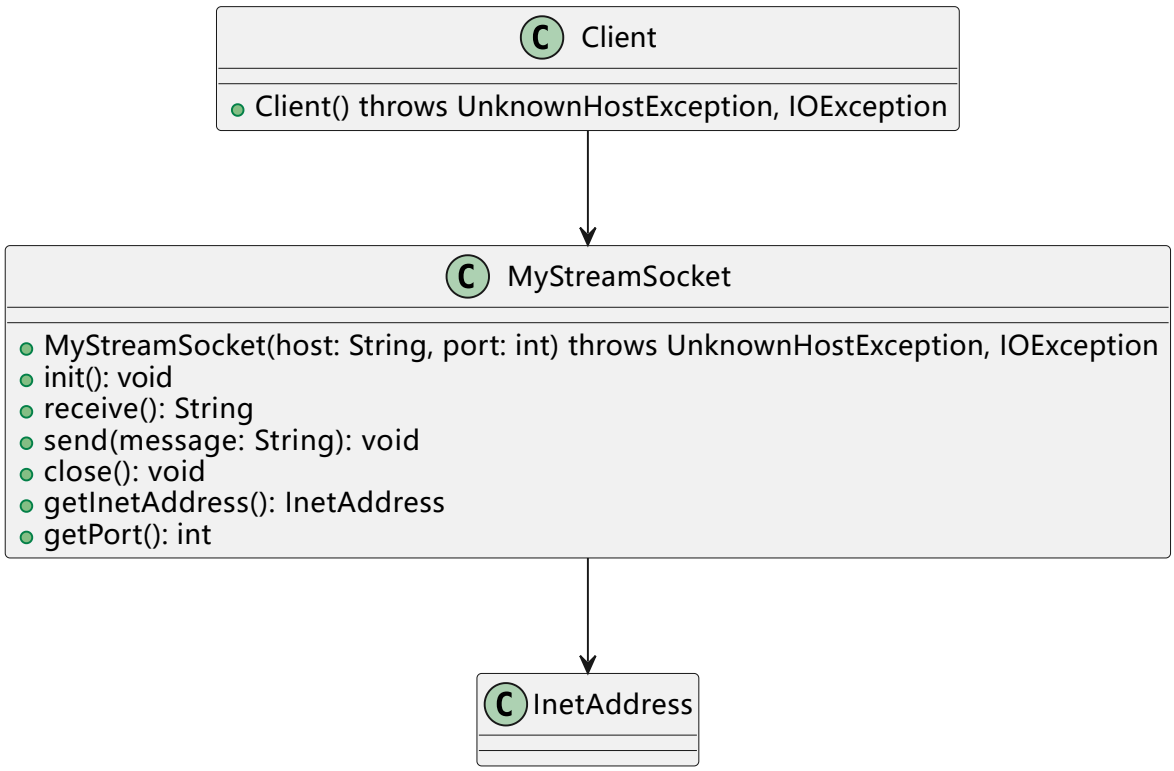
服务端




```java
public class Server {

    // 构造函数，可能抛出 IOException 异常
    public Server() throws IOException {
        // 创建 MyStreamServerSocket 对象，绑定到端口 8000
        MyStreamServerSocket ss = new MyStreamServerSocket(8000);

        // 无限循环，等待客户端连接
        while (true) {
            // 接受客户端连接请求，返回 MyStreamSocket 对象
            MyStreamSocket s = ss.accept();

            // 初始化 MyStreamSocket 对象
            s.init();

            // 创建一个新线程处理客户端消息
            new Thread() {
                public void run() {
                    // 无限循环，接收客户端消息
                    while (true) {
                        String message = null;

                        try {
                            // 接收客户端消息
                            message = s.receive();
```

```
                        } catch (IOException e) {
                            try {
                                // 关闭 MyStreamSocket 对象
                                s.close();
                            } catch (IOException e1) {
                                e1.printStackTrace();
                            }
                        }

                        // 如果收到消息，则打印客户端信息并回复相同的消息
                        if (message != null) {
                            System.out.println(
                                "Client(" + s.getInetAddress().getHostName()
+ ":" + s.getPort() + "):" + message);
                            s.send(message);
                            System.out.println("Server:" + message);
                        }
                    }
                }
            }.start();
        }
    }
}

```

客户端




```
import java.io.IOException;
import java.net.UnknownHostException;
import java.util.Scanner;

```

```
4
5  public class Client {
6
7      // 构造函数，可能抛出 UnknownHostException 和 IOException 异常
8      public Client() throws UnknownHostException, IOException {
9          // 创建 MyStreamSocket 对象，连接到服务器的 IP 地址 "127.0.0.1" 和端口号
   8000
10         MyStreamSocket s = new MyStreamSocket("127.0.0.1", 8000);
11
12         // 初始化 MyStreamSocket 对象
13         s.init();
14
15         // 创建一个新线程处理从服务器接收的消息
16         new Thread() {
17             public void run() {
18                 String message = null;
19                 // 无限循环，接收服务器消息
20                 while (true) {
21                     try {
22                         // 接收服务器消息
23                         message = s.receive();
24                     } catch (IOException e) {
25                         try {
26                             // 关闭 MyStreamSocket 对象
27                             s.close();
28                         } catch (IOException e1) {
29                             e1.printStackTrace();
30                         }
31                     }
32                     // 如果收到消息，则打印服务器信息
33                     if (message != null) {
34                         System.out.println(
35                             "Server(" + s.getInetAddress().getHostName() +
   ":" + s.getPort() + "):" + message);
36                         System.out.print("Client:");
37                     }
38                 }
39             }
40         }.start();
41
42         System.out.print("Client:");
43
44         // 无限循环，从控制台输入消息并发送给服务器
45         while (true) {
46             Scanner scanner = new Scanner(System.in);
47             String message = scanner.nextLine();
48             s.send(message);
49         }
50     }
51 }
52
```

实验结果

服务端

```
Client(127.0.0.1:52842):Hello,Server
Server:Hello,Server
Client(127.0.0.1:52842):Bye
Server:Bye
```

客户端

```
Client:Hello,Server
Server(127.0.0.1:8000):Hello,Server
Client:Bye
Server(127.0.0.1:8000):Bye
```

思考题

1、如何实现全双工的流式socket通信?

套接字本身具有接收数据和发送数据的功能，对于服务端而言，每接收到一次来自客户端的连接请求，就会针对该客户端创建一个套接字，实现一个服务端主机同时处理多个客户端的请求。

2、如何实现安全socket API?

要实现安全的Socket API，可以使用SSL/TLS协议来加密通信。使用Java中的SSLSocket和SSLServerSocket类，配置SSLContext、KeyManager和TrustManager以确保安全的密钥和证书管理。

3、如何实现1对多的并发?

对于每一个客户端主机，服务端都会建立一个套接字用于接收和发送数据，实现一对多，在服务端程序中，会使用多线程，实现对于多个客户端请求的并发处理。