



**东北大学秦皇岛分校**  
**计算机与通信工程学院**  
**计算机网络课程设计**

设计题目 基于 Java 和多线程 Socket  
的 Telnet 协议客户端

专业名称	<u>计算机科学与技术</u>
班级学号	<u>计科 1802-35-20188068</u>
学生姓名	<u>孔天欣</u>
指导教师	<u>徐长明</u>
设计时间	<u>2021 年 6 月 14 日—2021 年 6 月 18 日</u>

# 课程设计任务书

专业：计算机科学与技术 学号：20188068 学生姓名（签名）：

设计题目：

## 一、设计实验条件

1. 基于 Windows 10 的开发环境
2. 阿里云 Ubuntu 云服务器

## 二、设计任务及要求

1. 实现连接和访问云服务器远程主机；
2. 实现网络虚拟终端 NVT；
3. 实现选项协商功能；
4. 实现 TELNET 协议和服务器交互的基本功能；
5. 实现 TELNET 内置功能。

## 三、设计报告的内容

1. 设计题目与设计任务（设计任务书）
2. 前言（绪论）(设计的目的、意义等)
3. 设计主体（各部分设计内容、分析、结论等）
4. 结束语（设计的收获、体会等）
5. 参考资料

## 四、设计时间与安排

- 1、设计时间： 1 周
- 2、设计时间安排：

熟悉实验设备、收集资料： 2 天

设计图纸、实验、计算、程序编写调试： 2 天

编写课程设计报告： 2 天

答辩： 1 天

## 目 录

前 言.....	3
1 设计概述.....	4
1.1 设计要求 .....	4
1.2 设计内容 .....	4
2 设计原理.....	4
2.1 远程登录服务协议 Telnet .....	4
2.2 虚拟终端 NVT .....	5
2.3 选项协商 .....	6
2.4 通信方式 .....	7
2.5 内置功能模式 .....	7
3 设计思路.....	8
3.1 整体 Telnet 结构设计 .....	8
3.2 虚拟终端 NVT 设计 .....	9
3.3 选项协商方法设计 .....	9
3.4 通信方式设计.....	10
4 设计实现.....	11
4.1 整体 Telnet 结构实现 .....	11
4.2 虚拟终端 NVT 实现 .....	11
4.3 选项协商的实现 .....	13
4.4 通信方式的实现 .....	15
5 设计测试与分析.....	20
5.1 测试环境.....	20
5.2 测试内容 .....	20
6 结束语.....	24
参考文献 .....	25
附 录.....	26

## 基于 Java 和多线程 Socket 的 Telnet 协议客户端

### 前 言

计算机网络课程不仅需要对书本上的理论知识有所深刻掌握，同时还需要将理论转化为代码实践。此外，只有通过代码实践，才能加深对原本理论的理解。因此进行计算机网络课程设计是非常有必要的。计算机网络课程设计有利于提高设计者对计算机网络知识的理解，同时也更能掌握程序设计编程语言的应用。

本文根据 RFC 854 协议设计了 Telnet 客户端程序。根据 RFC 854 协议要求，该 Telnet 通过 NVT ASCII、一次一个字符的半双工通信模式和状态机化的 Telnet 状态和选项协商逻辑，能够实现虚拟终端、选项协商、远程登录和控制主机、紧急停止的功能。此外，该程序还提供了 Telnet 的内置功能模式，允许用户进行程序内部选项配置。同时，本文配置了阿里云 Ubuntu 云服务器的 Telnet 服务端程序作为测试对象，并成功使用本地 Telnet 客户端程序和对方进行连接、选项协商和交互的全过程测试。综上所述，本文基于的 Telnet 客户端程序实现了 Telnet 协议的基本功能。

本文首先对该 Telnet 程序进行全局概述，然后介绍该程序所用到的技术原理和知识节点，再详细叙述设计思路和设计具体的代码实现，最后通过设计测试和分析对本程序逐个功能进行测试和验证。

## 1 设计概述

### 1.1 设计要求

TELNET 协议允许用户用一台终端来访问远程的主机，它允许终端于主机之间以半双工的方式交换信息，可参阅 RFC864[6-13]。本次设计要求实现 TELNET 协议的基本功能。

### 1.2 设计内容

本文基于 Java,Socket 和多线程技术实现了 Telnet 的客户端程序,根据 RFC 854 协议要求,该 Telnet 通过 NVT ASCII、一次一个字符的半双工通信模式和状态机化的 Telnet 状态和选项协商逻辑,能够实现虚拟终端、选项协商、远程登录和控制主机、进入紧急停止模式的功能。此外,该程序还提供了 Telnet 的内置功能模式,允许用户进行程序内部选项调整。同时,本文配置了阿里云 Ubuntu 云服务器的 Telnet 服务端程序作为测试对象,并成功使用本地 Telnet 客户端程序和对方进行连接、选项协商和交互的全过程测试。综上所述,本文基于的 Telnet 客户端程序实现了 Telnet 协议的基本功能。

## 2 设计原理

### 2.1 远程登录服务协议 Telnet

Telnet (Teletype Network) 开发于 1969 年,是一种应用层协议,使用于互联网或局域网中,通过虚拟终端的形式,提供半双工、以字符流为主的命令行交互功能。它是互联网远程登录服务的标准协议和主要方式,常用于服务器的远程控制,可供用户在本地主机运行远程主机上的工作。它利用客户端、服务端的 Telnet 程序和虚拟终端 NVT 实现 TCP 数据包和 Telnet 控制信息在本地和远程主机的交互,从而达到远程控制的效果<sup>[1]</sup>。

在使用 Telnet 客户端时,用户首先在电脑运行 Telnet 客户端程序,使用 Telnet 协议连接到服务器,然后输入账户名和口令以验证身份。用户通过在本地主机的输入命令,让已连接的远程主机运行用户提供的命令,如同直接在远程主机的控制台上进行字符流输入和执行一样。

## 2.2 虚拟终端 NVT

### 1. NVT

网络虚拟终端 NVT (Network Virtual Terminal) 是带有键盘和显示器的虚拟设备。用户输入产生的数据被发送到服务器进程, 服务器进程回送的响应则输出到虚拟终端的显示器上。客户端和服务端都必须把它们的物理终端和 NVT 进行相互转换。无论客户进程所在的终端类型为何种, Telnet 程序都将它转换为 NVT 格式。同时, 不考虑服务器终端类型, Telnet 程序必须能够把 NVT 格式转换为本终端所能够支持的格式。

### 2. NVT ASCII

在 NVT 格式中, NVT ASCII 代表 7 位的 ASCII 字符集, 在客户端和服务端进行消息传输时, 客户端和服务端互相传送的数据都将被转为 NVT ASCII 格式。

NVT ASCII 分为命令字节和数据字节两种, 其中命令字节的最高位为 1, 数据字节的最高位为 0。其余低 7 位则是实际的命令或者数据内容。在字节流进入 Telnet 程序时, Telnet 程序通过检查每个字节的最高位来判断对方发送的是命令字节还是数据字节。所有 Telnet 命令至少包含一个两到三字节序列, 如果是命令字节, 那么 Telnet 程序接收到的第一个最高位为 1 的字节必须是 IAC (11111111), 否则不会识别后续给出的命令选项细节。如果是数据字节, 则等待用户输入完成并执行任务。

根据 RFC 854 协议, Telnet 提供的部分 NVT ASCII 命令字节集如表 2.1 所示。

表 2.1 NVT ASCII 命令集<sup>[2]</sup>

名称	字节码	功能
DM	242	数据标记
IP	244	中止对方进程
AO	245	中止对方输出
AVT	246	试探活动连接
EC	247	擦除字符
EL	248	擦除行
GA	249	继续进行
SB	250	子选项协商
WILL	251	选项协商 WILL
WONT	252	选项协商 WONT
DO	253	选项协商 DO

名称	字节码	功能
DONT	254	选项协商 DONT
IAC	255	解释为命令

### 2.3 选项协商

当客户端成功连接到远程主机时，Telnet 连接双方首先进行交互的信息是选项协商数据。其中，任何一方都可以主动发送选项协商请求给对方，并要求对方对此请求做出回应。只有双方协商完毕后，双方才会正式发送实际的数据。

选项协商包含了一串字节流，在字节流中，每个处理选项协商的命令都是三字节序列（子选项除外）：第一个字节是 IAC，第二个字节是 WILL，WONT，DO，DONT 命令的其中之一，第三个字节是协商的具体内容。

第一个字节 IAC 是所有协商选项开头必须携带的标识，以表明后续字节是命令的详细解释。

第二个字节中的四个命令分别对应的含义见表 2.2。此项字节表达出发送方对后续第三个字节协商选项的意愿，并要求通信对方也要使用对应的三字节序列进行回复以完成该项目的协商内容。根据 RFC854 协议规定，对于 WILL 和 DO 命令而言，客户端有权同意与否，但如果服务端发送 WONT 和 DONT 命令，那么客户端只能同意服务端的要求，不得反驳，详细的双方交互可能性见表 2.3。

以回显功能的开启为例，服务端 Telnet 程序发送 IAC WILL ECHO，表达出服务端将要激活字符的回显功能，客户端 Telnet 若回答 IAC DO ECHO，则表示客户端同意对方开启此功能，若回答 IAC DONT ECHO，则表示客户端不同意对方激活此功能。

表 2.2 选项协商第二字节请求含义

命令	含义
WILL	发送方本身将激活选项
WONT	发送方本身将禁止选项
DO	发送端让接收端激活选项
DONT	发送端让接收端禁止选项

表 2.3 第二字节双方选项协商情况

发送方命令	发送方意愿	接收方命令	接收方意愿
WILL	将激活选项	DO	同意
WILL	将激活选项	DONT	不同意

发送方命令	发送方意愿	接收方命令	接收方意愿
DO	让对方激活选项	WILL	同意
DO	让对方激活选项	WONT	不同意
WONT	将禁止选项	DONT	只能同意
DONT	让对方禁止选项	WONT	只能同意

第三个字节则是协商的选项细节，例如流量控制、窗口大小、数据收发方式的协商等，具体内容见表 2.4。

表 2.4 协商选项细节<sup>[2]</sup>

选项字节码	功能
1	回显
3	抑制继续进行
5	状态
6	定时标记
24	终端类型
31	窗口大小
32	终端速率
33	远程流量控制
34	行方式
36	环境变量

## 2.4 通信方式

Telnet 的服务端和客户端之间共有 3 种通信方式。分别为一次一个字符、行模式和准行模式通信方式。由于大多数主流 Telnet 程序均基于一次一个字符通信模式进行构建，因此本项目使用该模式实现双方通信。

一次一个字符模式中，用户所键入的每个字符都单独发送到服务端 Telnet 程序。同时，服务端 Telnet 程序会对用户键入的绝大多数字符进行回显（除了 Ctrl，Caps Lock 键等），而本地终端则不会显示用户输入的字符。如果要使用该模式，那么在协商选项阶段中，客户端 Telnet 程序需要同意对方激活回显功能和抑制继续进行功能。

## 2.5 内置功能模式

在本项目中，用户通过同时键入“Ctrl”和“]”键来开启 Telnet 客户端内置功能模式，开启该功能后，用户输入的字符不再被发送至服务端服务器，而是在本地进行处理。本项目提供的 Telnet 内置功能命令如表 2.5 所示。



表 2.5 Telnet 内置功能命令

命令	功能
help	内置功能和命令帮助
status	当前连接状态，如 IP 地址，端口等
stop	要求远程主机停止执行当前命令
close	关闭和远程 Telnet 服务端的连接
exit	退出内置功能模式

### 3 设计思路

#### 3.1 整体 Telnet 结构设计

用户在登入 Telnet 客户端后，进行的整体流程图如图 3.1 所示。图中可见，用户需要输入 telnet 命令来启动 Telnet 客户端，如果用户输入的 IP 地址开启了 Telnet 服务，则进行选项协商，否则提醒用户要求重新输入正确格式或者开通 Telnet 服务的 IP 地址，选项协商完毕后，用户需要输入远程终端对应的账户和密码，若核对无误，则可以正式实行远程控制终端。

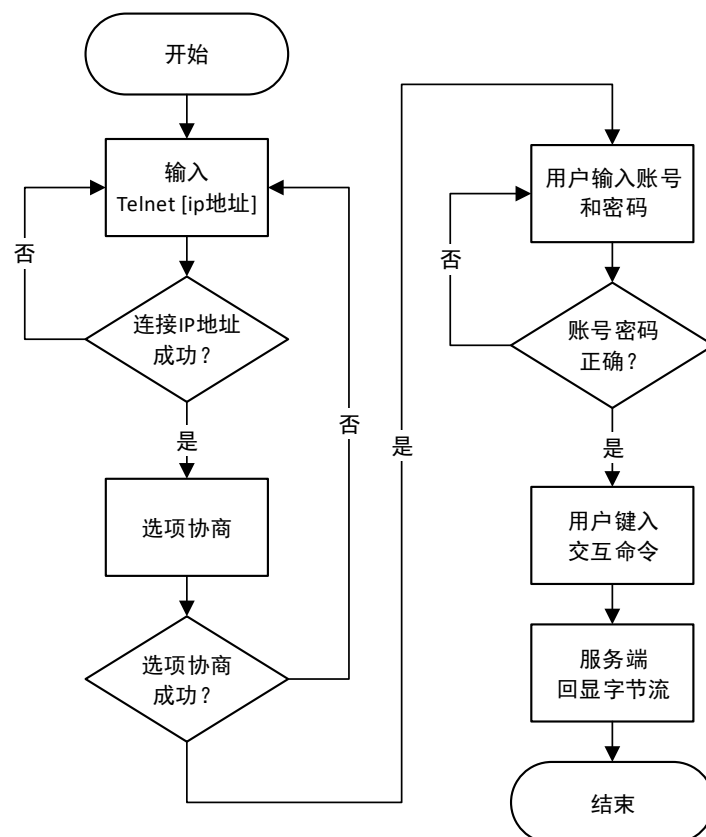


图 3.1 Telnet 客户端流程图

### 3.2 虚拟终端 NVT 设计

对于虚拟终端而言，Windows10 提供的命令提示符可以完成虚拟终端的背景版功能。但需尽可能实现就在 Linux 终端控制的效果，比如输入 CTRL+C 可以使得服务端结束当前正在执行命令，输入退格可以让服务端回显退格等，此外，还需要添加 CTRL+J 键实现切换内部模式的功能。因此对于 NVT ASCII 而言，虚拟终端设计程序需依次解决下述问题。

1. 将用户键入的键盘码转换为 NVT ASCII 码；
2. 将用户键入的特定组合键转换为特定行为；
3. 将用户键入的特定组合键转换为专用命令。

对于问题 1 而言，可以通过编写程序，通过监控来接收用户敲入的键盘码和 ASCII 字符进行一一映射，再将转换后的字符发送给服务端。

对于问题 2 而言，可以通过钩子函数监控用户先后输入的两个键，如果都命中程序代码的预先设定，则调用相关方法，例如监控到 ctrl 和 j 键后，调用执行内部模式的方法来进入 Telnet 内部模式。

对于问题 3 而言，同问题 2，但需要给服务端发送特定的 IAC 开头的命令令其执行。

### 3.3 选项协商方法设计

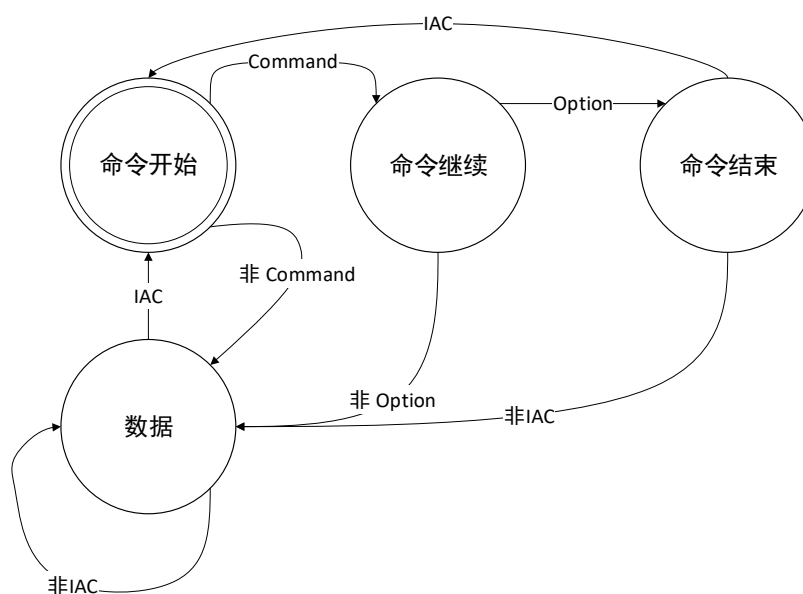


图 3.2 字节流识别状态机

在客户端和服务端初次连接时，需要进行选项协商。选项协商方法通过设计状

态机来完成，如图 3.2 所示。图中可见，选项协商的时候，客户端会收到一连串字节流，这串字节流既可能包含命令，也可能包含数据。当开始识别一个个字节时，如果第一个字节是 IAC，则进一步识别后续的命令字节（Command），如果第二个字节是命令，则进一步识别下一个选项字节（Option）。如果识别的第一个字节不是 IAC 或者中途遇到非命令字节，则识别为服务端回显的普通数据，输出到终端。

### 3.4 通信方式设计

如图 3.3 的 Telnet 模式状态机所示，Telnet 客户端设计为 5 个模式，分别为未连接模式、未选项协商模式、交互模式、内置功能模式和紧急停止模式，用户登入 Telnet 客户端时，Telnet 的状态为未连接；但用户键入 telnet 命令时，程序会尝试连接用户键入的 IP，若连接成功，则转变状态未选项协商模式，此时开始选项协商，若协商成功，则进入交互模式。如果用户在此时键入 Ctrl+]，则进入 Telnet 内置功能模式，此时可以执行 Telnet 内部提供的命令（见表 2.5）。如果键入 ctrl+C，则进入紧急停止模式，要求服务端停止当前命令执行。

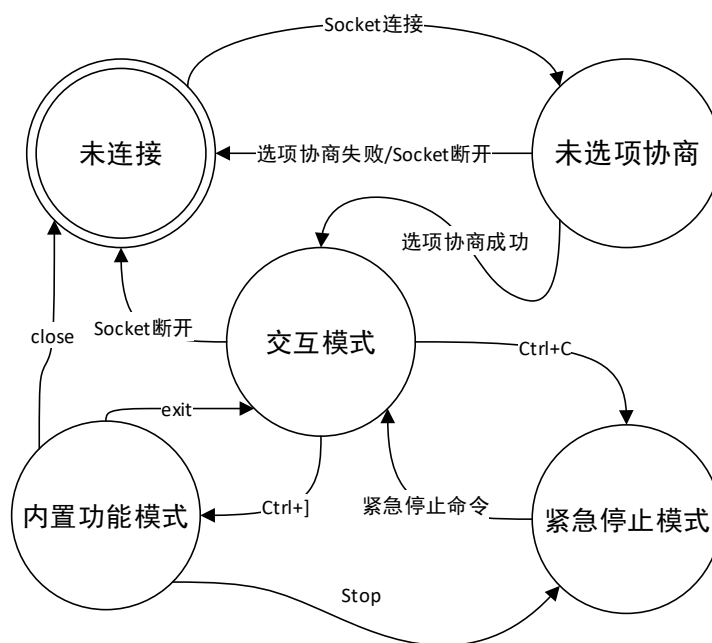


图 3.3 Telnet 模式状态机

一次一个字符的通信方式要求用户键入一个字符后，键盘监控程序便捕获到用户的输入字符，并将其转变为 NVT ASCII 格式，然后交由 Telnet 客户端执行。Telnet 客户端将接收到的字符通过 Socket 发送 TCP 数据包到对方的服务端，接着开辟新线程，通过读取输入流的方式来等待服务端的响应。此后对方的服务端会回显用

户输入的字符，在本地终端则不输出用户输入的字符。在用户敲下回车键后，服务端接收到回车键，则开始执行命令，并将执行结果以字节流形式返回显示到本地终端。

## 4 设计实现

### 4.1 整体 Telnet 结构实现

本项目的 Telnet 客户端主要基于 Java 语言和多线程 Socket 技术、全局钩子监控函数实现，可以和远程任何开启 Telnet 服务功能的主机进行 Telnet 连接和通信。在本项目中，将 IP 地址为 106.13.48.34 的阿里云云服务器开启 Telnet 服务功能，并令本项目的 Telnet 客户端程序与其进行连接和交互。

在项目工程文件中，主要有 TelnetBoot，TelnetClient，AssignedNumber 和 KeyBoardListener 四个类，它们各自对应的功能如下。

#### 1. TelnetBoot 类

枢纽类。该类是 Telnet 客户端的启动类，作为其他所有类的中枢，负责令 KeyBoardListener 类识别和转换用户键入，并要求 Telnet 核心类做出相应的行为模式。

#### 2. TelnetClient 类

该类是 Telnet 客户端的核心类，基于 Socket 和多线程技术实现了 Telnet 的基本功能。

#### 3. AssignedNumber 类

该类提供了 RFC 854 规定的命令和选项字节集。

#### 4. KeyBoardListener 类

该类基于钩子监控函数，负责将用户键入的字符格式转为 NVT ASCII 格式。同时还负责根据用户的键入的特殊组合键要求 Telnet 核心类改变其行为模式。

### 4.2 虚拟终端 NVT 实现

#### 1. NVT ASCII 转换

虚拟终端 NVT 的实现主要基于 Windows 10 操作系统自带的 CMD 控制台图形化控制台程序和本项目编写的 KeyBoardListener 类，CMD 控制台本身可以作为虚拟终端的图形化界面，令用户如同在实际进行操作 Linux 的终端界面一般；而

KeyListener 类将用户键入的值转化为 NVT ASCII 数据格式，屏蔽了本机 Windows 系统和服务端 Linux 系统之间字符格式的差异。

在 KeyListener 类中，通过实现 NativeKeyListener 接口，并注册全局钩子函数，以达到检测用户键入的效果，当用户按下键盘时，钩子会捕获到用户输入键对应的 raw code，对于一般键而言（如 a, b 等），raw code 就是 ASCII 值，此时无需进行转换，可以直接进行发送；而对于特殊键（如 Ctrl, /等）而言，它的 rawCode 并不是 ASCII 值，因此需要对此进行转换。实现这种转换功能的函数为 convertRawCodeToASCII，它的具体实现细节见代码清单 4.1。

代码清单 4.1 转换 rawCode 至 ASCII 格式

---

```
private int convertRawCodeToASCII(int rawCode) {
    switch (rawCode) {
        // -键
        case 189:
            rawCode = 45;
            break;
        // . 键
        case 190:
            rawCode = 46;
            break;
        // ] 键
        case 221:
            rawCode = 93;
            break;
        /// 键
        // ....更多映射略
    }
    return rawCode;
}
```

---

## 2. 特定组合键识别

在本程序中，nativeKeyPressed 函数负责监控用户的键入键，如果遇到特定组合键，则改变 Telnet 的模式或者要求 Telnet 执行特定操作，例如进入 Telnet 内部功能模式，或者要求服务端 Telnet 程序进行特定的操作等；如果是普通的输入，则将其转为 NVT ASCII 格式，并设置 flag 标记为 true，允许枢纽类进行字符的后续处理。相关实现见代码清单 4.2。

代码清单 4.2 监控键盘按下处理函数

```
@Override
public void nativeKeyPressed(NativeKeyEvent nativeKeyEvent) {
    int rawCode = nativeKeyEvent.getRawCode();
    // Ctrl + C 紧急停止对方的正在执行的命令
    if (telnetClient.telnetMode == 2 && ch == 17 && rawCode == 67) {
        telnetClient.emergencyStop();
    }
    // CTRL + ] 进入 telnet 模式
    if (ch == 17 && rawCode == 221) {
        telnetClient.telnetMode = 1;
    }
    rawCode = convertRawCodeToASCII(rawCode);
    ch = (char) rawCode;
    flag = true;
}
```

### 4.3 选项协商的实现

客户端和服务端初次连接时，首先需要进行选项协商，才能实现后续的数据流传输。在 `TelnetClient` 类中设置有 `int` 类型的属性 `telnetMode`，并设置不同的状态，如表 4.1 所示。其中，`telnetMode` 的初态是 `NOT_CONNECTED`。

表 4.1 `telnetMode` 状态

属性名	值	状态
<code>NOT_CONNECTED</code>	-1	尚未连接
<code>NOT_NEGOTIATED</code>	0	尚未选项协商
<code>INNER_MODE</code>	1	Telnet 内置功能模式
<code>INTERACTIVE_MODE</code>	2	交互模式
<code>EMERGENCY_STOP</code>	3	紧急停止模式

当用户键入连接 `telnet` 命令完成和远程服务端连接时，`telnetMode` 的状态则由 `NOT_CONNECTED` 转为 `NOT_NEGOTIATED` 模式，当客户端程序检测到 `telnetMode` 为尚未协商状态时，则会执行状态协商功能。

状态协商开始后，由于服务端的大多数协商命令（包括子选项）都是毫无必要的，因此只对特定的协商命令进行指定的回应。在本项目中，为实现一次一个字符的通信模式，对于服务端 `Telnet` 程序发送的 `IAC WILL ECHO` 和 `IAC WILL SUPPRESS_GO_AHEAD` 命令一律应答以同意；对于服务端发来的 `DONT` 和

WONT 命令，则根据 RFC 854 协议规定，一律回答以 WONT 和 DONT；其他的协商请求则予以拒绝，因为暂无协商的必要性。这些功能由 TelnetClient 类中的 optionToNegotiate 实现，为实现简单，采用了简化的状态机模式。详见代码清单 4.3。其中，responseNegotiation 对象是动态数组，用于存放应答的命令集。

代码清单 4.3 选项协商

---

```
private int optionToNegotiate(byte[] msg, int msgLength, int offset) {
    int i = offset + 1;
    if (i + 1 < msgLength) {
        byte command = msg[i];
        byte option = msg[i + 1];
        switch (command) {
            // 拒绝 DO 命令
            case AssignedNumber.Command.DO:
                responseNegotiation.add(AssignedNumber.IAC);
                responseNegotiation.add(AssignedNumber.Command.WONT);
                responseNegotiation.add(option);
                break;
            // RFC 854
            case AssignedNumber.Command.DONT:
                responseNegotiation.add(AssignedNumber.IAC);
                responseNegotiation.add(AssignedNumber.Command.WONT);
                responseNegotiation.add(option);
                break;
            case AssignedNumber.Command.WILL:
                switch (option) {
                    // 同意回显
                    case AssignedNumber.Option.ECHO:
                        responseNegotiation.add(AssignedNumber.IAC);
                        responseNegotiation.add(AssignedNumber.Command.DO);
                        responseNegotiation.add(AssignedNumber.Option.ECHO);
                        break;
                    // 同意抑制继续进行
                    case AssignedNumber.Option.SUPPRESS_GO_AHEAD:
                        responseNegotiation.add(AssignedNumber.IAC);
                        responseNegotiation.add(AssignedNumber.Command.DO);
                        responseNegotiation.add(AssignedNumber.Option.SUPPRESS
GO_AHEAD);
                        break;
                    // 其他拒绝
                    default:
                        responseNegotiation.add(AssignedNumber.IAC);
                        responseNegotiation.add(AssignedNumber.Command.DONT);
```

---

---

```
        responseNegotiation.add(option);
    }
    break;
// RFC 854
case AssignedNumber.Command.WONT:
    responseNegotiation.add(AssignedNumber.IAC);
    responseNegotiation.add(AssignedNumber.Command.DONT);
    responseNegotiation.add(option);
    break;
// 拒绝子选项协商
case AssignedNumber.Command.SB:
    responseNegotiation.add(AssignedNumber.IAC);
    responseNegotiation.add(AssignedNumber.Command.DONT);
    responseNegotiation.add(option);
    break;
default:
    }
}
return offset + 3;
}
```

---

当服务端和客户端协商完毕后, telnetMode 将由 NOT\_NEGOTIATED 模式转变到 INTERACTIVE\_MODE 模式, 即和服务端正式进行交互的模式, 双方开始传输真正的数据。

#### 4.4 通信方式的实现

双方协商完毕后, 则根据事先双方约定, 开始一次一个字符的通信模式。用户在键入字符后, 字符便传输到远程主机, 远程主机负责回显字符, 而本地终端不负责显示用户键入的字符。当用户键入回车后, 远程主机执行命令, 并返回一串命令执行结果的字符串, 在本地终端回显。用户还有可能输入特殊组合键进入特殊模式。

##### 1. Telnet 行为模式状态机

TelnetBoot 类和 TelnetClient 类则负责实现上述交互的各种可能性。在 TelnetClient 类中, 用户首先需进行 telnet 命令输入, 才能连接到远程主机, 而选项协商的过程对用户是透明的, 协商完毕之后就进入正式交互状态, 此时用户输入的一系列字符会在远程主机上执行。当用户输入特定组合键时会转变内部功能模式或紧急停止模式。上述状态行为的改变和对应字符的处理均由 telnetMode 属性



决定，因此 analyseInputLine 函数实现了根据 telnetMode 属性改变 telnet 行为的 telnetMode 状态机，当 telnetMode 状态发生改变时，会对用户键入的字符进行不同的处理，如代码清单 4.4 所示。

代码清单 4.4 Telnet 行为模式状态机

---

```
public void analyseInputLine(String input) {  
    // 识别未连接模式  
    if (telnetMode == NOT_CONNECTED && isValidTelnetConnectionInput(input)) {  
        String host = input.split(" ")[1];  
        System.out.println("Trying " + host + " ...");  
        // 连接远程主机和选项协商  
        connect(host);  
        // 转为交互模式  
        telnetMode = INTERACTIVE_MODE;  
        // 识别内部模式  
    } else if (telnetMode == INNER_MODE) {  
        // 进入内部模式  
        handleInnerCommand();  
        telnetMode = tcpSocket == null ? NOT_CONNECTED : INTERACTIVE_MODE;  
        // 如果 socket 已经关闭  
    } else if (tcpSocket.isClosed()) {  
        telnetMode = NOT_CONNECTED;  
        // 紧急停止模式  
    } else if (telnetMode == EMERGENCY_STOP) {  
        // 发送令对方服务端停止执行当前指令的命令  
        emergencyStop();  
        // 转为交互模式  
        telnetMode = INTERACTIVE_MODE;  
    } else {  
        // 交互模式  
        sendMessage(input.getBytes());  
        receiveMessage();  
    }  
}
```

---

## 2. 枢纽类 TelnetBoot

在 TelnetBoot 类中，用户首先输入 telnet [ip 地址]命令完成连接。之后该类会监控 KeyBoardListener 类中的 flag 是否为 true，若否，则等待，因为此时用户尚未键入。然后 KeyBoardListener 类开启对用户键入键盘行为监控，并将 NVT ASCII 格式下的键入字符传递给此类，并置 flag 为 true。TelnetBoot 类再将字符交由 TelnetCilent 类进行发送和接受回显处理，置 flag 为 false，等待用户的下一次键入。

代码清单 4.5 则展示了上述过程。

代码清单 4.5 TelnetBoot 枢纽函数

---

```
public void start() throws InterruptedException, IOException {
    new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
    Scanner scanner = new Scanner(System.in);
    TelnetClient telnetClient = new TelnetClient();
    KeyBoardListener keyBoardListener = new KeyBoardListener(telnetClient);
    while (true) {
        System.out.println("Enter telnet [ip_address] to continue :");
        KeyBoardListener.unregisterNativeHook();
        // 用户输入 telnet 命令
        telnetClient.analyseInputLine(scanner.nextLine());
        KeyBoardListener.registerNativeHook();
        do {
            // 如果用户尚未键入字符，则等待
            if (!keyBoardListener.flag) {
                Thread.sleep(10);
            } else {
                // 接受并处理用户输入的字符
                String s = String.valueOf(keyBoardListener.ch);
                keyBoardListener.flag = false;
                telnetClient.analyseInputLine(s);
            }
            // 如果没有断开连接
        } while (telnetClient.telnetMode != -1);
    }
}
```

---

### 3. 多线程并行收发字节流

当字符传递给 TelnetClient 类进行处理时，TelnetClient 类调用 sendMessage 函数获取 socket 输出流，填充入该字符，并强制清空缓冲区以立即发送数据，之后调用 receiveMessage 函数接受服务端的回显字符或字符串。receiveMessage 函数基于多线程模式进行，每当一个字符发送完毕后，都会开辟一个新线程进行回显数据的接收，同时处理回显数据。这是因为服务端发送回显字节流的时间间隔不等，而且字节流发送次数不止一次，而客户端在等待服务端返回数据时，整个程序会陷入阻塞，导致此时程序无法处理用户的后续键入字符。开辟新线程可以并行处理用户的键入字符和接收服务端的回显字节流。相关具体实现如代码清单 4.6 所示。在代码中，还设置了无用线程回收器 collectUselessThreads()函数，当线程数过多时会对 threadList 里的线程进行一一清理。

代码清单 4.6 多线程接收回显字节流

---

```
private Queue<Thread> threadList = new LinkedList<>();

private void receiveMessage() {
    // 开辟新线程
    Runnable runnable = new Runnable() {
        @Override
        public void run() {
            byte[] msg = new byte[MESSAGE_LENGTH];
            int msgLength;
            do {
                // 接收数据
                msgLength = getMessageFromServer(msg);
                // 处理数据
                handleMessageStream(msg, msgLength);
            } while (msgLength != -1);
        }
    };
    // 执行新线程
    Thread thread = new Thread(runnable);
    thread.start();
    threadList.add(thread);
    // 线程满足 10 的倍数则进行回收，减少资源占用
    if (threadList.size() % 10 == 0) {
        collectUselessThreads();
    }
}
```

---

#### 4. 滑动指针处理字节流

对服务端回显的字节流则处理如下，设置指针 pos 指向用户传回的字节流头部，判断当前指向的字节是否是 IAC 命令，如果是，则执行处理 Telnet 命令的函数，命令处理完毕后，返回 pos 指针最新位置。否则滑动 pos 指针，依次在本地终端输出 pos 指向字节流对应的 UTF-8 格式字符串，在显示字符串时如果再次遇到 IAC 命令，则返回 pos 指针最新位置。指针 pos 会在处理中不断移动至字节流尾部，到达尾部后，即表明已经处理完这段字节流。之后如果有应答命令，那么将应答命令的集合发送至服务端 Telnet 程序，发送完毕后，集合会清空，并等待在下一个字节流到来时中识别命令和填充响应字节命令集。上述流程的具体实现的函数为 handleMessageStream，如代码清单 4.7 所示。msgLength 是字节流中有效字节的长度。

代码清单 4.7 处理服务端回显字节流

---

```
private void handleMessageStream(byte[] msg, int msgLength) {
    int pos = 0;
    while (pos < msgLength) {
        // IAC 开头的处理协商命令，命令处理完毕，更新 pos
        if (msg[pos] == AssignedNumber.IAC) {
            pos = optionToNegotiate(msg, msgLength, pos);
        } else {
            // 本地终端输出字符串，若遇到命令，更新 pos
            pos = echo(msg, msgLength, pos);
        }
    }
    // 发送应答命令
    if (!responseNegotiation.isEmpty()) {
        sendMessage(responseNegotiation);
        responseNegotiation.clear();
    }
}
```

---

## 5. 紧急停止服务端现有执行

当用户键入组合键时，如果是 Ctrl+C，那么 TelnetClient 的 telnetMode 将转变模式为 EMERGENCY\_STOP。此时将执行 emergencyStop 函数，该函数适用于服务端正在执行的命令进入无限循环不中止，导致用户平常的键入无法令其处理后续命令，远程控制就此失效的情况。具体实现流程如下，首先客户端给服务端发送一个 TCP 紧急数据包，数据包的内容是 Telnet 命令中的 DM<sup>[3]</sup>，然后紧接着给服务端发送 IAC IP 命令。服务端收到紧急数据包的 DM 命令后，则无视用户此前的普通键入字节流，一直翻阅后续的输入字节流直到找到 IAC IP 命令并执行，此后服务端则会中止当前正在执行的命令。用户又恢复了对服务端的远程控制能力。emergencyStop 函数的具体实现如代码清单 4.8 所示。

代码清单 4.8 紧急停止函数

---

```
public void emergencyStop() {
    // 发送紧急 TCP 数据包 DM
    tcpSocket.sendUrgentData((int) AssignedNumber.Command.DM);
    // 发送停止进程命令 IAC IP
    responseNegotiation.add(AssignedNumber.IAC);
    responseNegotiation.add(AssignedNumber.Command.IP);
    sendMessage(responseNegotiation);
    responseNegotiation.clear();
}
```

---

## 5 设计测试与分析

## 5.1 测试环境

本次测试的 Telnet 客户端是基于 JDK1.8 开发的 maven 项目，运行在 Windows 10 操作系统的 CMD 命令提示符上；它安装有 jnativehook 依赖，是一个钩子函数库。Telnet 服务端是基于阿里云 Ubuntu 云服务器的 openbsd-inetd 和 telnetd 服务，同时设置安全组开放端口为 23，并且暂时关闭了系统防火墙。测试时网络畅通，用户在本地启动 Telnet 客户端。

## 5.2 测试内容

## 1. 用户登入 Telnet 客户端

CMD 输入 `mvn exec:java -Dexec.mainClass="com.strutnut.telnet.TelnetBoot" -q` 并回车，以启动 Telnet 客户端程序。此后，用户输入 `telnet 106.13.48.34` 命令和远程云服务器的 Telnet 服务连接，如图 5.1 所示。

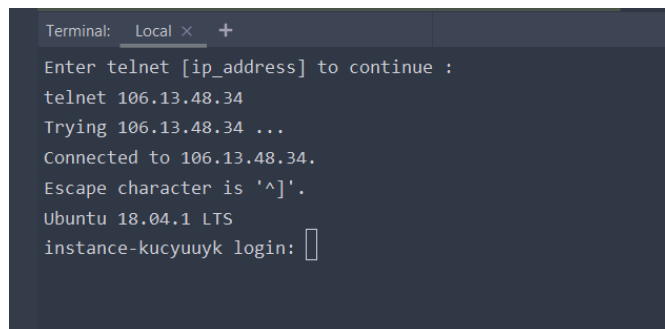


图 5.1 Telnet 连接远程主机

## 2. 选项协商

通过开启双方传输数据的输出函数，可见双方选项协商的过程，如图 5.2 所示，request 是服务端发送给客户端的字节流，response 是客户端发给服务端的字节流。

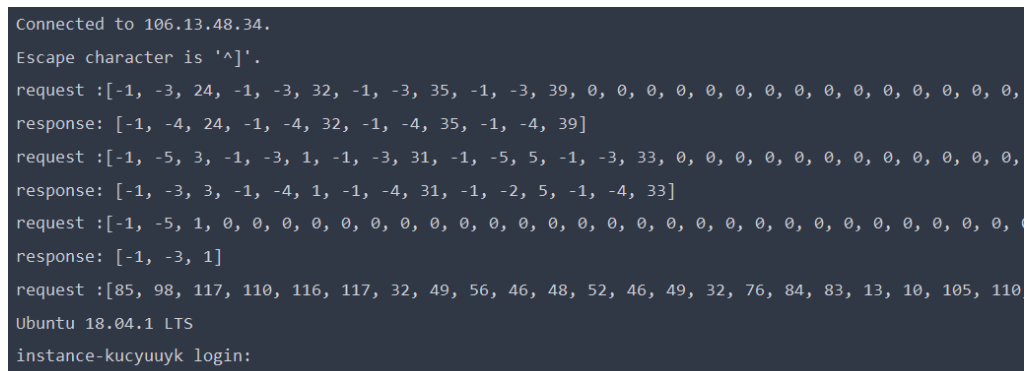


图 5.2 选项协商

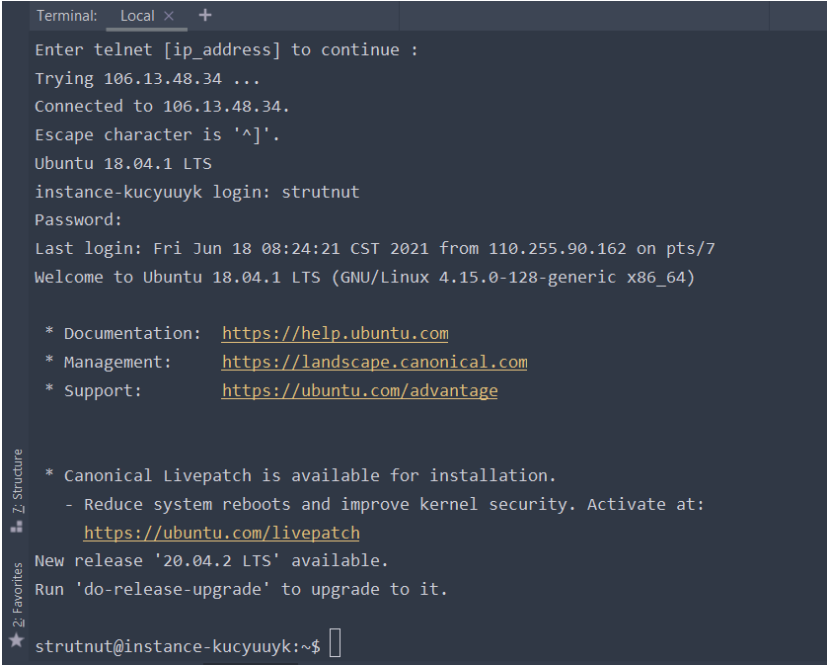
图 5.2 中可见,第一次连接时,服务端 Telnet 程序发送了-1, -3, 24, -1, -3, 32, -1, -3, 35, -1, -3, 39 的字节流。由于 Java 中的 byte 类型是 8 位带正负号,故最高位为符号位,因此 $-1 + 256 = 255$ ,  $-3 + 256 = 252$ , 以此类推。

服务端发送的命令字节流经翻译后,意为: IAC DO 终端类型, IAC DO 终端速率, IAC DO 显示位置, IAC DO 新环境选择, 此后客户端程序回应-1, -4, 24, -1, -4, 32, -1, -4, 35, -1, -4, 39。意为: IAC WONT 终端类型, IAC WONT 终端速率, IAC WONT 显示位置, IAC WONT 新环境选择; 上述操作表明客户端拒绝了服务端希望客户端进行的所有选项, 因为对其来说并非必要。

服务端再次发来命令字节流-1, -5, 3, -1, -3, 1, -1, -3, 31, -1, -5, 5, -1, -3, 33, 意为: IAC WILL 抑制继续进行, IAC DO 回显, IAC DO 窗口大小, IAC WILL 状态, IAC DO 远程流量控制, IAC WILL 客户端程序回应-1, -3, 3, -1, -4, 1, -1, -4, 31, -1, -2, 5, -1, -4, 33, 意为 IAC DO 抑制继续进行, IAC WONT 回显, IAC WONT 窗口大小, IAC DONT 状态, IAC WONT 远程流量控制。上述操作表明客户端允许服务端进行一次一个字节的通信模式, 以及不同意客户端本身进行回显字符, 对于其他无关紧要的选项, 则一律拒绝后续协商。

最后服务端发来命令字节流-1,-5,1, 意为: IAC WILL 回显, 客户端回应-1,-3,1, 意为: IAC DO 回显。上述操作表明客户端同意服务端本身进行回显字符的操作。

### 3 用户输入账户名和密码



```
Terminal: Local x +
Enter telnet [ip_address] to continue :
Trying 106.13.48.34 ...
Connected to 106.13.48.34.
Escape character is '^]'.
Ubuntu 18.04.1 LTS
instance-kucyuuyk login: strutnut
Password:
Last login: Fri Jun 18 08:24:21 CST 2021 from 110.255.90.162 on pts/7
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-128-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch
New release '20.04.2 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

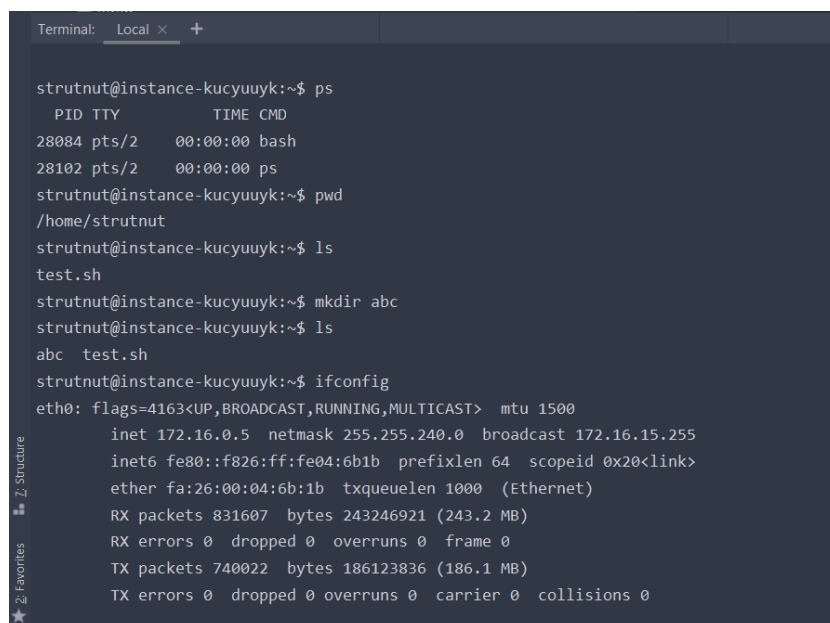
strutnut@instance-kucyuuyk:~$
```

图 5.3 用户登录成功

连接完毕后，用户需要输入账户名和密码才能真正远程控制云服务器终端，如图 5.3 所示，在 instance-kucyuuyk login 一行中，strutnut 是用户名。在 Password 一行中，由于此时远程 Telnet 服务端关闭了回显功能，因此用户输入的密码字符串并没有显示在本地终端。当用户输入完毕密码后，则服务端则返回了远程终端登录成功的欢迎界面，并等待用户输入下一个命令。

#### 4. 常用命令测试

图 5.4 给出了 Linux 终端常用的命令，例如 ls, pwd, mkdir, ifconfig 等，可以看到，用户输入的字符都得到了回显，并且用户要求执行的任务也均得到了完整和正确的返回结果。

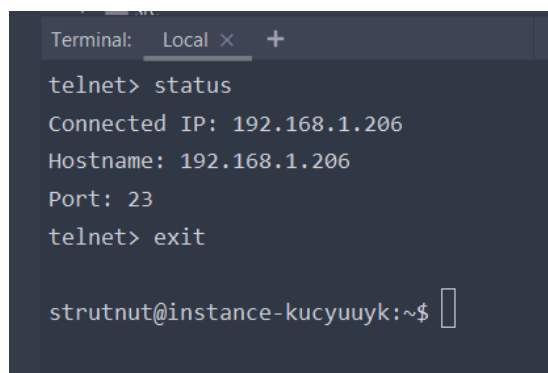


```
Terminal: Local x +
strutnut@instance-kucyuuyk:~$ ps
  PID TTY          TIME CMD
 28084 pts/2    00:00:00 bash
 28102 pts/2    00:00:00 ps
strutnut@instance-kucyuuyk:~$ pwd
/home/strutnut
strutnut@instance-kucyuuyk:~$ ls
test.sh
strutnut@instance-kucyuuyk:~$ mkdir abc
strutnut@instance-kucyuuyk:~$ ls
abc test.sh
strutnut@instance-kucyuuyk:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 172.16.0.5  netmask 255.255.240.0  broadcast 172.16.15.255
    inet6 fe80::f826:ff:fe04:6b1b  prefixlen 64  scopeid 0x20<link>
    ether fa:26:00:04:6b:1b  txqueuelen 1000  (Ethernet)
    RX packets 831607  bytes 243246921 (243.2 MB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 740022  bytes 186123836 (186.1 MB)
    TX errors 0  dropped 0  overruns 0  carrier 0  collisions 0
```

图 5.4 常用命令测试

#### 5. Telnet 内置功能模式测试

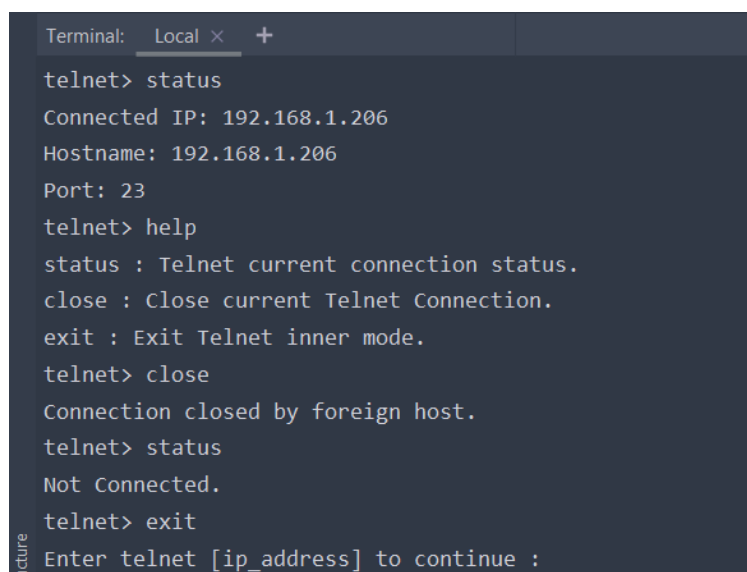
当用户按下组合键 CTRL+]时，就会进入 Telnet 内置模式，图 5.5 和图 5.6 展示了 Telnet 中 help, status, close 和 exit 命令的实现效果。



```
Terminal: Local x +
telnet> status
Connected IP: 192.168.1.206
Hostname: 192.168.1.206
Port: 23
telnet> exit

strutnut@instance-kucyuuyk:~$
```

图 5.5 Telnet 未断开连接

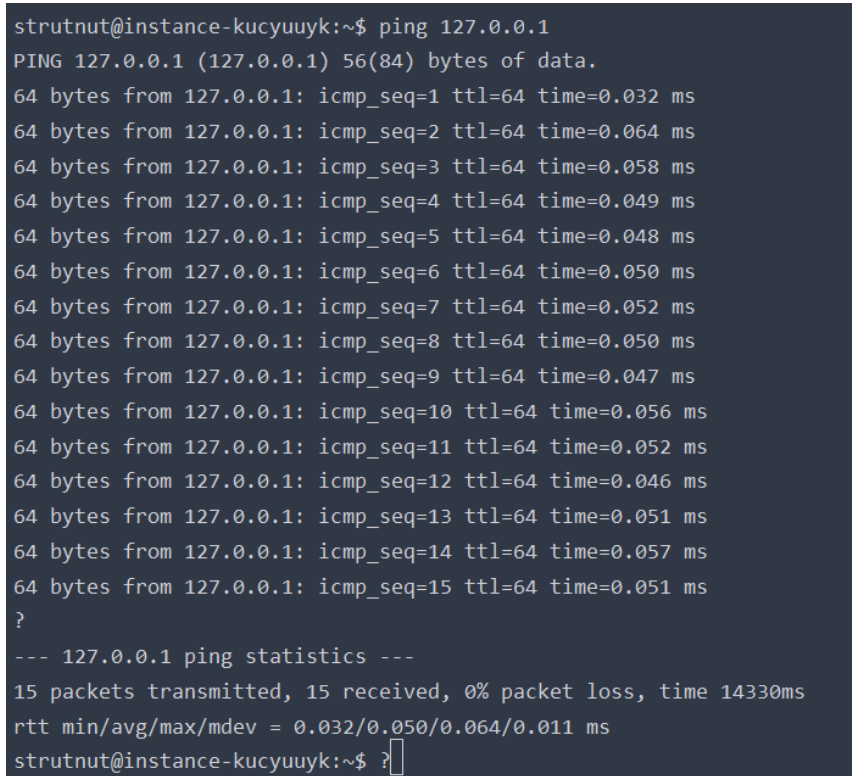


```
Terminal: Local x +
telnet> status
Connected IP: 192.168.1.206
Hostname: 192.168.1.206
Port: 23
telnet> help
status : Telnet current connection status.
close : Close current Telnet Connection.
exit : Exit Telnet inner mode.
telnet> close
Connection closed by foreign host.
telnet> status
Not Connected.
telnet> exit
Enter telnet [ip_address] to continue :
```

图 5.6 Telnet 断开连接

## 6. 紧急停止模式测试

如图 5.7 所示，用户输入 ping 任务后，服务端不断返回 ping 的数据行，并且没有停止的迹象，导致用户无法输入后续命令。因此，用户在服务端返回第 15 个数据包（icmp\_seq=15）后按下 Ctrl+C（终端出现“?”符号），强行中止了服务端正在执行的 ping 任务，服务端给出了 ping 命令的总结报告后成功结束任务，此后，用户又可以继续发送新的任务给服务端。



```
strutnut@instance-kucyuuyk:~$ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.032 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.064 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.049 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.048 ms
64 bytes from 127.0.0.1: icmp_seq=6 ttl=64 time=0.050 ms
64 bytes from 127.0.0.1: icmp_seq=7 ttl=64 time=0.052 ms
64 bytes from 127.0.0.1: icmp_seq=8 ttl=64 time=0.050 ms
64 bytes from 127.0.0.1: icmp_seq=9 ttl=64 time=0.047 ms
64 bytes from 127.0.0.1: icmp_seq=10 ttl=64 time=0.056 ms
64 bytes from 127.0.0.1: icmp_seq=11 ttl=64 time=0.052 ms
64 bytes from 127.0.0.1: icmp_seq=12 ttl=64 time=0.046 ms
64 bytes from 127.0.0.1: icmp_seq=13 ttl=64 time=0.051 ms
64 bytes from 127.0.0.1: icmp_seq=14 ttl=64 time=0.057 ms
64 bytes from 127.0.0.1: icmp_seq=15 ttl=64 time=0.051 ms
?
--- 127.0.0.1 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14330ms
rtt min/avg/max/mdev = 0.032/0.050/0.064/0.011 ms
strutnut@instance-kucyuuyk:~$ ?
```

图 5.7 按下 CTRL+C 紧急停止



## 6 结束语

本次课程设计的内容是 Telnet 客户端，需要阅读 RFC 856 以了解其中的原理，但囿于个人水平有限，在刚开始进行程序设计的时候遇到了许多困难，但在同学和老师的热情帮助下，通过查阅互联网相关知识资料和计算机网络相关书籍的 Telnet 部分，本人成功了解了 Telnet 客户端程序实现的各种原理，以及具体的协议格式等，成功克服了各种困难，最终完成了 Telnet 客户端程序。

尽管本人设计的 Telnet 程序已经能够实现大量基本命令功能，但限于时间仓促有限，一些特殊的任务功能尚未能实现，例如 vi/vim 命令下的文本编辑器等。此外，字节流收发有小概率引发延迟短时间较高的问题，可通过设置缓冲区解决。同时还可能存在一些潜在的 bug 未能测试得出。

作为大学四年最后一次课程设计，这次课程设计内容通过将课程理论知识转化为代码实践，大大加深了本人对 Telnet 协议原理的理解，一定程度上提高了本人对于计算机网络学科的掌握程度，因此收获颇丰。明年就要进行毕业论文和相关作品的设计，希望本次课程设计能够对此有所启发。

## 参考文献

- [1] JamesF.Kurose, KeithW.Ross. 计算机网络:自顶向下方法[M]. 机械工业出版社, 2009.
- [2] W.RICHARDSTEVENS. TCP/IP 详解.卷 1,协议[M]. 机械工业出版社, 2000:295-315.
- [3] 王罡, 林立志. 基于 Windows 的 TCP/IP 编程[M]. 清华大学出版社, 2002.
- [4] 金瑜 王建勇 杨湘. 计算机网络实验教程(普通高等教育十二五规划教材.高等学校计算机[M]. 科学出版社, 2013.
- [5] AeleenFrisch, 弗里斯科比, 刘颖. Unix 与 Linux 系统管理[M]. 清华大学出版社, 2006.

## 附 录

本节包含 Telnet 客户端程序的所有源代码，下述代码在基于 JDK1.8，MAVEN 依赖配置 jnativehook2.1.0，IDEA 2019.3，Windows 10 环境下运行、测试通过。

### 1. TelnetBoot 类

---

```
package com.strutnut.telnet;

import java.io.IOException;
import java.util.Scanner;

public class TelnetBoot {

    public void start() throws InterruptedException, IOException {
        new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
        Scanner scanner = new Scanner(System.in);
        TelnetClient telnetClient = new TelnetClient();
        KeyBoardListener keyBoardListener = new KeyBoardListener(telnetClient);
        while (true) {
            System.out.println("Enter telnet [ip_address] to continue :");
            KeyBoardListener.unregisterNativeHook();
            // 用户输入 telnet 命令
            telnetClient.analyseInputLine(scanner.nextLine());
            KeyBoardListener.registerNativeHook();
            do {
                // 如果用户尚未键入，则等待
                if (!keyBoardListener.flag) {
                    Thread.sleep(10);
                } else {
                    // 接受并处理用户输入的字符
                    String s = String.valueOf(keyBoardListener.ch);
                    keyBoardListener.flag = false;
                    telnetClient.analyseInputLine(s);
                }
                // 如果没有断开连接
            } while (telnetClient.telnetMode != -1);
        }
    }

    public static void main(String[] args) throws InterruptedException, IOException {
        new TelnetBoot().start();
    }
}
```

---

---

## 2. AssignedNumber 类

---

```
package com.strutnut.telnet;

public class AssignedNumber {

    public static final byte IAC = (byte) 255;

    public static class Command {
        public static final byte WILL = (byte) 251;
        public static final byte WONT = (byte) 252;
        public static final byte DO = (byte) 253;
        public static final byte DONT = (byte) 254;
        public static final byte DM = (byte) 242;
        public static final byte IP = (byte) 244;
        public static final byte SB = (byte) 250;
        public static final byte EL = (byte) 248;
        public static final byte EC = (byte) 247;
    }

    public static class Option {
        public static final byte ECHO = (byte) 1;
        public static final byte SUPPRESS_GO_AHEAD = (byte) 3;
    }
}
```

---

## 3. TelnetClient 类

---

```
package com.strutnut.telnet;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.*;

public class TelnetClient {

    private Socket tcpSocket;

    InputStream telnetInputStream;

    OutputStream telnetOutputStream;

    private static final int PORT = 23;
```

---

---

```
private static final int MESSAGE_LENGTH = 1024 * 10;

private static final int DELAY_TIME = 50;

/**
 * -1 尚未连接模式
 * 0 尚未协商模式
 * 1 telnet 模式
 * 2 和服务端交互模式
 * 3 紧急停止模式
 */
public int telnetMode = NOT_CONNECTED;

private static final int NOT_CONNECTED = -1;
private static final int NOT_NEGOTIATED = 0;
private static final int INNER_MODE = 1;
private static final int INTERACTIVE_MODE = 2;
private static final int EMERGENCY_STOP = 3;

public void analyseInputLine(String input) throws IOException, InterruptedException {
    // 未连接模式
    if (telnetMode == NOT_CONNECTED && isValidTelnetConnectionInput(input)) {
        String host = input.split(" ")[1];
        System.out.println("Trying " + host + " ...");
        connect(host);
        telnetMode = INTERACTIVE_MODE;
    } else if (telnetMode == INNER_MODE) {
        handleInnerCommand();
        telnetMode = tcpSocket == null ? NOT_CONNECTED : INTERACTIVE_MODE;
        // 如果 socket 已经关闭
    } else if (tcpSocket.isClosed()) {
        telnetMode = NOT_CONNECTED;
        // 紧急停止模式
    } else if (telnetMode == EMERGENCY_STOP) {
        emergencyStop();
        telnetMode = INTERACTIVE_MODE;
    } else {
        // 交互模式
        sendMessage(input.getBytes());
        receiveMessage();
    }
}
```

---

---

```

private void handleInnerCommand() throws IOException, InterruptedException {
    boolean f = false;
    KeyBoardListener.unregisterNativeHook();
    Scanner scanner = new Scanner(System.in);
    System.out.println();
    while (true) {
        System.out.print("telnet> ");
        String command = scanner.nextLine();
        if (!f) {
            new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
            f = true;
        }
        switch (command) {
            case "help":
                System.out.println("status : Telnet current connection status.");
                System.out.println("close : Close current Telnet Connection.");
                System.out.println("exit : Exit Telnet inner mode.");
                break;
            case "status":
                if (tcpSocket != null) {
                    System.out.println("Connected          IP:          "          +
tcpSocket.getLocalAddress().getHostAddress());
                    System.out.println("Hostname:          "          +
tcpSocket.getLocalAddress().getHostName());
                    System.out.println("Port: " + tcpSocket.getPort());
                } else {
                    System.out.println("Not Connected.");
                }
                break;
            case "close":
                collectUselessThreads();
                System.out.println("Connection closed by foreign host.");
                telentMode = NOT_CONNECTED;
                try {
                    tcpSocket.close();
                    tcpSocket = null;
                } catch (IOException e) {
                    e.printStackTrace();
                }
                break;
            case "exit":
                if (tcpSocket != null) {
                    sendMessage("\n".getBytes());
                    receiveMessage();

```

---

---

```
        }
        KeyBoardListener.registerNativeHook();
        return;
    case "stop":
        emergencyStop();
        break;
    default:
        System.out.print("\r");
        break;
    }
}

}

public void emergencyStop() {

    try {
        tcpSocket.sendUrgentData((int) AssignedNumber.Command.DM);
    } catch (IOException e) {
        e.printStackTrace();
    }
    responseNegotiation.add(AssignedNumber.IAC);
    responseNegotiation.add(AssignedNumber.Command.IP);
    sendMessage(responseNegotiation);
    responseNegotiation.clear();
}

private void connect(String host) {
    try {
        this.tcpSocket = new Socket(host, PORT);
        this.telnetInputStream = tcpSocket.getInputStream();
        this.telnetOutputStream = tcpSocket.getOutputStream();
        System.out.println("Connected to " + host + ".");
        System.out.println("Escape character is '^['.");
        telnetMode = NOT_NEGOTIATED;
        receiveMessage();
    } catch (IOException e) {
        System.out.println("Unable to connect to remote host: Connection timed out.");
    }
}

private Queue<Thread> threadList = new LinkedList<>();

private void receiveMessage() {
    Runnable runnable = new Runnable() {
```

---

---

```
@Override
public void run() {
    int msgLength;
    do {
        byte[] msg = new byte[MESSAGE_LENGTH];
        // 接收数据
        msgLength = getMessageFromServer(msg);
//        System.out.println("request :"+ Arrays.toString(msg));
        // 处理数据
        handleMessageStream(msg, msgLength);
    } while (msgLength != -1);
}

};
// 执行新线程
Thread thread = new Thread(runnable);
thread.start();
//    threadList.add(thread);
// 线程满足 10 的倍数则进行回收, 减少资源占用
if (threadList.size() % 10 == 0) {
    collectUselessThreads();
}
}

private void collectUselessThreads() {
    Iterator<Thread> iterator = threadList.iterator();
    while (iterator.hasNext()) {
        iterator.next().interrupt();
        iterator.remove();
    }
    System.gc();
}

private void handleMessageStream(byte[] msg, int msgLength) {
    int pos = 0;
    while (pos < msgLength) {
        // IAC 开头的处理协商命令, 命令处理完毕, 更新 pos
        if (msg[pos] == AssignedNumber.IAC) {
            pos = optionToNegotiate(msg, msgLength, pos);
        } else {
            // 本地终端输出字符串, 若遇到命令, 更新 pos
            pos = echo(msg, msgLength, pos);
        }
    }
}

if (!responseNegotiation.isEmpty()) {
```

---



---

```
        sendMessage(responseNegotiation);
        responseNegotiation.clear();
    }
}

private void sendMessage(List<Byte> list) {
    byte[] bytes = convertListToBytes(list);
    sendMessage(bytes);
}

private void sendMessage(byte[] msg) {
    try {
        if (telnetMode == NOT_NEGOTIATED) {
            telnetOutputStream.write(msg);
            telnetOutputStream.flush();
        } else {
            for (byte b : msg) {
                telnetOutputStream.write(b);
                telnetOutputStream.flush();
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private int getMessageFromServer(byte[] msg) {
    try {
        return telnetInputStream.read(msg);
    } catch (IOException e) {
    }
    return -1;
}

private void delay(int time) {
    try {
        Thread.currentThread().sleep(time);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

private byte[] convertListToBytes(List<Byte> list) {
    int i = 0;
```

---

```

byte[] listBytes = new byte[list.size()];
for (Byte b : list) {
    listBytes[i++] = b.byteValue();
}
return listBytes;
}

private int echo(byte[] msg, int msgLength, int offset) {
    int i = offset;
    while (i < msgLength && msg[i] != AssignedNumber.IAC) {
        i++;
    }
    String msgStr = new String(msg, offset, i - offset);
    System.out.print(msgStr);
    if (msgStr.contains("logout")) {
        telnetMode = NOT_CONNECTED;
    }
    return i;
}

private List<Byte> responseNegotiation = new ArrayList<>();

/**
 * 选项协商
 */
private int optionToNegotiate(byte[] msg, int msgLength, int offset) {
    int i = offset + 1;
    if (i + 1 < msgLength) {
        byte command = msg[i];
        byte option = msg[i + 1];
        switch (command) {
            case AssignedNumber.Command.DO:
                responseNegotiation.add(AssignedNumber.IAC);
                responseNegotiation.add(AssignedNumber.Command.WONT);
                responseNegotiation.add(option);
                break;
            case AssignedNumber.Command.DONT:
                responseNegotiation.add(AssignedNumber.IAC);
                responseNegotiation.add(AssignedNumber.Command.WONT);
                responseNegotiation.add(option);
                break;
            case AssignedNumber.Command.WILL:
                switch (option) {
                    case AssignedNumber.Option.ECHO:

```

---

```
        responseNegotiation.add(AssignedNumber.IAC);
        responseNegotiation.add(AssignedNumber.Command.DO);
        responseNegotiation.add(AssignedNumber.Option.ECHO);
        break;
    case AssignedNumber.Option.SUPPRESS_GO_AHEAD:
        responseNegotiation.add(AssignedNumber.IAC);
        responseNegotiation.add(AssignedNumber.Command.DO);

responseNegotiation.add(AssignedNumber.Option.SUPPRESS_GO_AHEAD);
        break;
    default:
        responseNegotiation.add(AssignedNumber.IAC);
        responseNegotiation.add(AssignedNumber.Command.DONT);
        responseNegotiation.add(option);
    }
    break;
case AssignedNumber.Command.WONT:
    responseNegotiation.add(AssignedNumber.IAC);
    responseNegotiation.add(AssignedNumber.Command.DONT);
    responseNegotiation.add(option);
    break;
case AssignedNumber.Command.SB:
    responseNegotiation.add(AssignedNumber.IAC);
    responseNegotiation.add(AssignedNumber.Command.DONT);
    responseNegotiation.add(option);
    break;
    default:
    }
}
return offset + 3;
}

/**
 * IP 地址格式的正则表达式
 */
private static final String IP_REGEX = "([1-9]|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])" +
    "(\\.([1-9]|[1-9]\\d|1\\d{2}|2[0-4]\\d|25[0-5])){3}";

/**
 * IP 地址格式验证
 */
private boolean isValidTelnetConnectionInput(String input) {
    String[] s = input.split(" ");
    if (s.length > 1 && input.startsWith("telnet")) {
```

---

---

```

        String connectHost = s[1];
        return s[1].equals("localhost") || connectHost.matches(IP_REGEX);
    }
    return false;
}

public static void main(String[] args) throws IOException, InterruptedException {
    // 测试一些命令的返回结果协商
    TelnetClient telnetClient = new TelnetClient();
    telnetClient.analyseInputLine("telnet 106.13.48.34");
    telnetClient.responseNegotiation.add(AssignedNumber.IAC);
    telnetClient.responseNegotiation.add((byte) 248);
    telnetClient.sendMessage(telnetClient.responseNegotiation);
    byte[] msg = new byte[1024];
    int[] msgIn = new int[1024];
    telnetClient.getMessageFromServer(msg);
    for (int i = 0; i < msg.length; i++) {
        msgIn[i] = msg[i] + 256;
    }
    System.out.println(Arrays.toString(msgIn));
}
}

```

---

#### 4. KeyBoardListener 类

---

```

package com.strutnut.telnet;

import org.jnativehook.GlobalScreen;
import org.jnativehook.NativeHookException;
import org.jnativehook.keyboard.NativeKeyEvent;
import org.jnativehook.keyboard.NativeKeyListener;

import java.util.logging.Level;
import java.util.logging.Logger;

public class KeyBoardListener implements NativeKeyListener {
    private static final Logger logger =
        Logger.getLogger(GlobalScreen.class.getPackage().getName());

    private TelnetClient telnetClient;

    public KeyBoardListener(TelnetClient telnetClient) {
        this.telnetClient = telnetClient;
        logger.setLevel(Level.OFF);
        GlobalScreen.addNativeKeyListener(this);
    }
}

```

---

---

```
}

public char ch = 0;

public boolean flag = false;

@Override
public void nativeKeyTyped(NativeKeyEvent nativeKeyEvent) {
}

@Override
public void nativeKeyPressed(NativeKeyEvent nativeKeyEvent) {
    int rawCode = nativeKeyEvent.getRawCode();
    //      System.out.println(rawCode);

    // ctrl + c 紧急停止对方的无尽输出
    // 67
    if (telnetClient.telnetMode == 2 && ch == 17 && rawCode == 88) {
        telnetClient.emergencyStop();
    }
    // CTRL + ] 进入 telnet 模式
    if (ch == 17 && rawCode == 221) {
        telnetClient.telnetMode = 1;
    }

    rawCode = convertRawCodeToASCII(rawCode);
    ch = (char) rawCode;
    //      System.out.println(rawCode);
    //      System.out.println(ch);
    flag = true;
}

@Override
public void nativeKeyReleased(NativeKeyEvent nativeKeyEvent) {
}

private int convertRawCodeToASCII(int rawCode) {
    // 大写字母转小写字母
    if (rawCode > 64 && rawCode < 91) {
        rawCode += 32;
    }
    switch (rawCode) {
        // -键
```

---

---

```
        case 189:
            rawCode = 45;
            break;
            // . 键
        case 190:
            rawCode = 46;
            break;
            // ] 键
        case 221:
            rawCode = 93;
            break;
            /// 键
        case 191:
            rawCode = 47;
            break;
            // Ctrl 键
        case 162:
            rawCode = 17;
            break;
            // | 键
        case 220:
            rawCode = 124;
            break;
    }
    return rawCode;
}

public static void registerNativeHook() {
    try {
        GlobalScreen.registerNativeHook();
    } catch (NativeHookException e) {
        e.printStackTrace();
    }
}

public static void unregisterNativeHook() {
    try {
        GlobalScreen.unregisterNativeHook();
    } catch (NativeHookException e) {
        e.printStackTrace();
    }
}
}
```

---

## 5. Test 类（该类是测试类，正式运行不纳入）

---

```
package com.strutnut.telnet;

import org.jnativehook.NativeHookException;

import java.io.*;
import java.net.Socket;
import java.util.Arrays;
import java.util.Scanner;

public class Test {

    public static void main(String[] args) throws NativeHookException, InterruptedException {
        Test test = new Test();
        test.test03();
    }

    public void test06() {
        try {
            Process process = Runtime.getRuntime().exec("tasklist");
            InputStreamReader reader = new InputStreamReader(process.getInputStream());
            LineNumberReader line = new LineNumberReader(reader);
            StringBuilder response = new StringBuilder();
            String str;
            while ((str = line.readLine()) != null) {
                response.append(str).append("\n");
            }
            System.out.print(response.toString());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void test05() {
        try {
            Socket socket = new Socket("localhost", 8080);
            InputStream inputStream = socket.getInputStream();
            OutputStream outputStream = socket.getOutputStream();
            while (true) {
                byte[] content = new byte[1024];
                inputStream.read(content);
                System.out.println(Arrays.toString(content));
                outputStream.write(new byte[]{111, 112, 113});
                outputStream.flush();
            }
        }
    }
}
```

---

---

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

    private void test04() {
        Scanner scanner = new Scanner(System.in);
        System.out.println(scanner.hasNext());
    }

    private void test03() {
        KeyBoardListener keyBoardListener = new KeyBoardListener(new TelnetClient());
    }

    private void test02() throws InterruptedException, NativeHookException, IOException {
        TelnetBoot telnetBoot = new TelnetBoot();
        telnetBoot.start();
    }

    /**
     * 255, 253, 24, 255, 253, 32, 255, 253, 35, 255, 253, 39
     */
    private void test01() {
        try {
            Socket socket = new Socket("192.168.1.221", 23);
            InputStream inputStream = socket.getInputStream();
            byte[] bytes = new byte[1024];
            int len = inputStream.read(bytes);

            int[] res = new int[1024];
            for (int i = 0; i < len; i++) {
                res[i] = bytes[i];
                if (bytes[i] < 0) {
                    res[i] += 256;
                }
            }
            System.out.println(Arrays.toString(res));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

---