

## 实验报告：Lab7 Socket Programming(基于 python)

课程名称：计算机网络 年级：大二

上机实践成绩：

实践

指导教师：章玥

姓名：邱吉尔

学号：10235101533

上机实践日期：

2024/12/30

### 一、需求分析

目的：实现一个简单的客户端和服务端的通信系统

功能需求：客户端应能够向服务器发送消息，服务器应能够接收并显示消息，客户端和服务端应该支持多个连接，服务端也能向客户端发送信息

输入与输出： 系统的输入是客户端输入以及从文件读取的数据

输出是服务器和客户端都能打印接收到的消息并且每次收到信息都会自动换行以区分

#### 初步设计思路：

客户端需要连接到指定的服务器 IP 和端口，发送输入的消息

服务器需要监听指定的端口，接收并打印客户端发送的消息

### 二、设计

#### 1) Server 服务器端

##### i. 实现能在标准输出打印客户端发送的消息

```
import socket # 导入 socket 模块，提供网络通信功能
```

首先导入 socket 模块，提供网络通信的基本功能，可以创建套接字、连接远程主机、接收和发送数据等

```
SERVER_PORT = 5432 # 设置服务器端口  
MAX_PENDING = 10 # 最大等待连接数  
MAX_LINE = 256 # 每次接收的最大字节数
```

设置服务器监听的端口号以及每次从客户端接收的最大字节数  
这里为了完成后续实验的要求顺便也设置了下最大连接数

```
# 创建一个 socket 对象, 使用 IPv4 地址和 TCP 协议
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error as e:
    print(f"server: socket creation failed: {e}") # 如果 socket 创建失败, 打印错误信息
    return # 退出程序
```

接着使用 `socket.socket(socket.AF_INET, socket.SOCK_STREAM)`: 创建一个 TCP 套接字 (IPv4 地址族和 TCP 协议)

`socket.AF_INET` 表示使用 IPv4 协议

`socket.SOCK_STREAM` 表示使用 TCP 协议

如果创建套接字失败, 捕获异常并打印错误信息, 程序终止

```
s.bind(('0.0.0.0', SERVER_PORT)) # 绑定到所有可用的接口
```

使用 `bind` 将服务器的套接字绑定到本地的 IP 地址和端口号  
'0.0.0.0' 表示绑定到所有可用的网络接口, 服务器可以通过任何可用的网络接口接收请求

`SERVER_PORT` 指定服务器监听的端口号, 即之前设置的 5432

```
s.listen(MAX_PENDING)
```

调用 `listen()` 方法将套接字设置为监听状态, 准备接收客户端的连接请求

`MAX_PENDING` 表示最大等待连接数。即最多允许 5 个客户端等待连接

```
print(f"Listening on port {SERVER_PORT}...")
```

输出一下正在监听哪个端口, 便于 debug

```
while True:
    # 不断地等待并接受新的连接
    try:
        # 接受连接并获取新的套接字和客户端的地址信息
        new_s, addr = s.accept()
        client_thread = threading.Thread(target=handle_client, args=(new_s, addr))
        client_thread.start()
    except socket.error as e:
        # 如果 accept 失败, 打印错误信息并继续等待其他连接
        print(f"server: accept failed: {e}")
        continue
```

`while True:` 使服务器不断接受新的客户端连接

s.accept() 阻塞直到有客户端连接到来，返回两个值：

new\_s: 一个新的套接字，用于与客户端通信

addr: 客户端的地址信息（如 IP 地址）

如果连接出现错误，会捕获异常并打印错误信息，然后继续等待新的连接

```
try:
    while True:
        # 接收来自客户端的数据，最多接收 MAX_LINE 字节
        data = new_s.recv(MAX_LINE)

        if not data:
            # 如果接收到的数据为空，表示连接已关闭
            print("Connection closed")
            break # 退出循环，关闭当前连接

        print(data.decode('utf-8'))
except socket.error as e:
    # 如果接收数据时出错，打印错误信息
    print(f"recv failed: {e}")
```

new\_s.recv(MAX\_LINE): 接收客户端发送的数据，最多接收 MAX\_LINE 字节的数据，即 256 字节

如果客户端关闭连接，recv() 返回空字节串，表示连接已关闭

data.decode('utf-8'): 将接收到的字节数据解码为字符串，使用 UTF-8 编码

## ii. 实现支持 5 个以上客户端同时发送消息并逐一打印

要使服务器能够支持多个客户端并发连接，并且能够同时接收消息并逐一打印，需要在服务器端实现多线程处理，每当一个客户端连接时，服务器会为该客户端创建一个独立的线程来处理通信，这样就能同时处理多个客户端

```
import threading
```

首先导入 threading 来实现线程相关操作

```
def handle_client(new_s, addr):
    """处理与客户端的通信"""
    print(f"Connection from {addr}") # 输出客户端的地址信息

    try:
        while True:
            # 接收来自客户端的数据，最多接收 MAX_LINE 字节
            data = new_s.recv(MAX_LINE)

            if not data:
                # 如果接收到的数据为空，表示连接已关闭
                print("Connection closed")
                break # 退出循环，关闭当前连接

            # 打印接收到的数据
            print(f"Received from {addr}: {data.decode('utf-8')}")
    except socket.error as e:
        print(f"recv failed: {e}")
    finally:
        # 关闭当前与客户端的连接
        new_s.close()
```

添加 `handle_client` 函数用于处理接收到的信息，添加输出的信息便于后续的测试

接着我们来看看 `main` 函数的改动：

```
while True:
    # 不断地等待并接受新的连接
    try:
        # 接受连接并获取新的套接字和客户端的地址信息
        new_s, addr = s.accept()
        client_thread = threading.Thread(target=handle_client, args=(new_s, addr))
        client_thread.start()
```

在每次接受到新的客户端连接后，都会使用 `threading.Thread` 为该连接创建一个独立的线程来处理，这样就可以让服务器同时处理多个客户端的请求，同时每个客户端都通过独立的线程接收和打印消息

### iii. 绑定至错误的端口号时能提示出错信息

为了实现这一功能就需要捕获 `socket.error` 异常。而 `socket.bind` 会抛出 `OSError` 或 `socket.error` 异常，如端口号已被占用，或没有足够权限绑定该端口，都会导致绑定失败  
修改代码如下：

```
try:
    s.bind(('0.0.0.0', SERVER_PORT)) # 尝试绑定到指定的端口
    print(f"Server is listening on port {SERVER_PORT}...")
except socket.error as e:
    # 如果绑定失败，打印错误信息并退出
    print(f"server: bind failed: {e}")
    s.close() # 关闭套接字
    return # 退出程序
```

如果端口绑定失败，就会打印错误信息 server: bind failed: {e}

## 2) Client 用户端

### i. 实现能从标准输入或文件接收消息

```
import socket
import sys
```

socket: 用于进行网络通信（创建套接字，连接，发送/接收数据等）。

sys: 用于处理命令行参数和系统退出。

```
SERVER_PORT = 5432 # 设置服务器端口
MAX_LINE = 256 # 设置最大消息长度
```

指定客户端连接的服务器端口号和指定最大消息长度

接着定义了 send\_message 函数，作用是将数据发送到服务器，两个参数分别是已经建立连接的 socket 以及消息的来源，可以是标准输入或文件路径，来看看函数的具体实现：

```
if source is not sys.stdin: # 如果 source 是文件对象
    with open(source, 'r') as file:
        for line in file:
            s.sendall(line.encode('utf-8'))
            print(f"Sent: {line.strip()}") # 打印已发送的内容
```

如果消息来源不是标准输入，即提供了一个文件路径，程序会打开该文件并逐行读取内容

```
else:
    # 否则从标准输入读取
    while True:
        try:
            buf = input() # 获取用户的输入数据
            if not buf: # 如果输入为空（按回车没有输入任何字符），则退出
                break
            # 将输入数据编码为 UTF-8 格式并发送
            s.sendall(buf.encode('utf-8')) # 发送数据到服务器
            print(f"Sent: {buf}") # 打印已发送的内容
        except KeyboardInterrupt:
            break # 如果收到键盘中断（Ctrl+C），则退出程序
```

如果消息来源是标准输入，则从键盘逐行读取消息

如果按下 Ctrl+C，则退出输入循环。

如果输入为空即连接两次回车，退出循环。

接着我们来看 main 函数的具体实现：

```
# 检查命令行参数是否正确
if len(sys.argv) < 2:
    print("usage: server host [filename]") # 提供正确的方式
    sys.exit(1) # 退出程序
```

首先检查命令行参数，确保至少提供了一个目标服务器地址

```
host = sys.argv[1] # 获取命令行输入的目标主机地址（即服务器的 IP 地址）

# 如果提供了文件路径，获取文件路径
filename = sys.argv[2] if len(sys.argv) == 3 else None
```

获取命令行参数中的目标主机地址，且如果提供了文件路径作为第二个命令行参数，则 filename 为该路径；如果没有，则为 None

```
# 创建一个 socket 对象，使用 IPv4 地址族和 TCP 协议
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error as e:
    print(f"client: socket creation failed: {e}") # 如果创建套接字失败，打印错误信息
    sys.exit(1) # 退出程序
```

接着创建一个套接字，指定地址族为 AF\_INET（IPv4），协议为 SOCK\_STREAM（TCP）

```
# 获取目标主机的 IP 地址
try:
    ip_address = socket.gethostbyname(host) # 通过域名解析得到目标主机的 IP 地址
except socket.gaierror as e:
    print(f"client: unknown host: {host}") # 如果主机名解析失败，打印错误信息
    sys.exit(1) # 退出程序
```

然后使用 `gethostbyname` 函数将目标主机的域名解析为 IP 地址

```
# 连接到目标服务器
try:
    s.connect((ip_address, SERVER_PORT)) # 使用获取到的 IP 地址和指定端口连接到服务器
except socket.error as e:
    print(f"client: connect failed: {e}") # 如果连接失败，打印错误信息
    s.close() # 关闭套接字
    sys.exit(1) # 退出程序
```

使用 `connect` 方法连接到目标服务器，连接的地址是 IP 地址和指定的端口，即 5432

```
send_message(s, filename if filename else sys.stdin)
```

最后调用 `send_message` 发送消息，就完成了客户端最基础的功能

## ii. 实现标准输入消息以两次回车作为结束标志

为了实现这个功能，则我们需要记录上一次的输入，只有当上一次输入以及当前输入都为空，也就是连接了两次回车后，才结束输入，所以要修改的是 `send_message` 函数：

```
else:
    last_input = ''
    while True:
        try:
            buf = input() # 获取用户的输入数据
            if not buf: # 如果输入为空（按回车没有输入任何字符）
                if not last_input: # 连续两次回车，退出
                    break
                else:
                    last_input = ''
            else:
                last_input = buf # 更新 last_input
            # 将输入数据编码为 UTF-8 格式并发送
            s.sendall(buf.encode('utf-8')) # 发送数据到服务器
```

引入了 `last_input` 用于记录上一次的输入，两次输入都为空时才 `break`

### iii. 连接至错误的 IP 地址/端口号时能提示出错信息

```
# 连接到目标服务器
try:
    s.connect((ip_address, SERVER_PORT)) # 使用获取到的 IP 地址和指定端口连接到服务器
except socket.error as e:
    print(f"client: connect failed: {e}") # 如果连接失败，打印错误信息
    s.close() # 关闭套接字
    sys.exit(1) # 退出程序
```

在 `s.connect()` 之前，添加了 `try-except` 块。这样，当无法连接到指定的 `ip` 地址或端口时，能够捕获 `socket.error` 异常并打印详细的错误信息

## 3) 整体

### i. 支持在 `localhost` 及在两台不同机器上运行

之前的用例都是在 `localhost` 上运行，肯定支持，在两台机器上运行时只要在运行 `client.py` 时输入正确的服务器地址即可，如：`python client.py 192.168.0.103`

### ii. 支持长文本消息（不少于 20KB），有缓冲区管理

要支持长文本消息，首先要将 `client` 和 `server` 端的缓冲区大小设置的大些，这里就以 4096 字节为例，先修改 `server.py`，在其中增加数据拼接逻辑：

在 `handle_client` 函数中使用循环不断调用 `recv()`

拼接收到的数据，直到客户端关闭连接或发送结束标志

以下为具体修改部分：



```
def handle_client(new_s, addr): 1个用法
    """处理与客户端的通信"""
    print(f"Connection from {addr}")

    try:
        received_data = b'' # 用于存储完整的接收到的数据
        while True:
            # 从客户端接收数据
            data = new_s.recv(BUFFER_SIZE)
            if not data:
                # 如果接收到的数据为空，表示连接已关闭
                break
            received_data += data # 拼接接收到的数据

        # 输出完整接收到的数据
        print(f"Complete message from {addr}: {received_data.decode('utf-8')}")
    except socket.error as e:
        print(f"recv failed: {e}")
    finally:
        new_s.close()
```

然后我们再修改客户端，需要自定义分块发送函数，将长于缓冲区的消息换分成最大为 4096 字节的块然后发送，另外，鉴于文本多是采用 UTF-8 编码，故显式指定发送消息的编码类型，以下为具体代码改动：

```
def send_message(s, file_or_input): 1个用法
    """发送消息到服务器"""
    if isinstance(file_or_input, str): # 如果是文件名
        try:
            with open(file_or_input, 'r', encoding='utf-8') as file:
                for line in file:
                    send_in_blocks(s, line.strip()) # 按行读取文件内容并分块发送
        except FileNotFoundError:
            print(f"File '{file_or_input}' not found.")
        except Exception as e:
            print(f"Error reading file '{file_or_input}': {e}")
```

```
def send_in_blocks(s, data): 2 用法
    """ 将数据分块发送 """
    start = 0
    while start < len(data):
        # 按 BUFFER_SIZE 分块发送数据
        end = min(start + BUFFER_SIZE, len(data))
        chunk = data[start:end]
        s.sendall(chunk.encode('utf-8'))
        start = end

    # 发送结束标志
    s.sendall(b"<END>")
```

### iii. 容错性好，无闪退

容错性方面有意在最初编写时就利用 try-except 来捕获异常，以此可以尽量避免闪退问题，以下是后面所新添加的文件不存在时的异常捕获：

```
def send_message(s, file_or_input): 1 个用法
    """ 发送消息到服务器 """
    if isinstance(file_or_input, str): # 如果是文件名
        while True:
            try:
                with open(file_or_input, 'r', encoding='utf-8') as file:
                    for line in file:
                        send_in_blocks(s, line.strip()) # 按行读取文件内容并分块发送
                    break # 文件发送成功后退出循环
            except FileNotFoundError:
                print(f"File '{file_or_input}' not found.")
                file_or_input = input("Please enter a valid file path or type 'exit' to quit: ")
                if file_or_input.lower() == 'exit':
                    print("Exiting file sending process.")
                    return
            except Exception as e:
                print(f"Error reading file '{file_or_input}': {e}")
                return
```

## 4) 附加

### i. 实现双工通信

- 1) 先来看客户端的改动： 首先为了让客户端具有接收消息的功能，需要为客户端创建一个独立的线程用于接收来自服务器的消息，

由此避免客户端在等待用户输入时阻塞接收数据，增加了 `receive_message` 函数用于在一个单独的线程中接收数据并打印，以下参考 `server.py` 完成编写：

```
def receive_message(sock): 1个用法
    """接收来自服务器的消息"""
    buffer = b'' # 用于拼接接收到的分块数据
    while True:
        try:
            data = sock.recv(BUFFER_SIZE)
            if not data:
                print("Server closed the connection.")
                break # 如果没有收到数据，服务器关闭了连接

            buffer += data
            while b"<END>" in buffer: # 检查是否接收到结束标志
                message, buffer = buffer.split(b"<END>", 1)
                clean_message = message.decode('utf-8').strip() # 解码并移除多余的空白
                print(f"Server: {clean_message}") # 仅输出消息内容
        except Exception as e:
            print(f"Error receiving message: {e}")
            break
```

然后在 `main` 函数中，启动接收线程并等待它的执行。原来发送消息的逻辑保持不变，但现在在发送消息之前启动了接收线程，确保客户端可以同时接收和发送数据，为了确保不会错过服务端的消息，因此先启动接收线程：

```
# 启动接收线程
receive_thread = threading.Thread(target=receive_message, args=(s,), daemon=True)
receive_thread.start()

# 启动发送线程
send_thread = threading.Thread(target=send_message, args=(s, filename if filename else sys.stdin), daemon=True)
send_thread.start()
```

- 2) 接着来看服务器端的改动：要让服务器也能发送消息给客户端，需要改造 `handle_client` 函数，使其可以对接收到的消息构造响应并建立一个独立线程，允许服务器主动向客户端发送消息。同时让服务器继续监听新的客户端连接，每个连接都会创建一个新的线程。这样，每个客户端的通信都能独立于其他客户端进行。

```
def server_send():
    """服务器主动发送消息给客户端"""
    try:
        while True:
            message = input("Server: ") # 服务器主动输入消息
            if message.lower() == "exit":
                print("Exiting server send mode.")
                break # 输入 exit 退出发送模式
            new_s.sendall((message + "<END>").encode('utf-8')) # 发送消息
    except socket.error as e:
        print(f"Server send error: {e}")

# 创建并启动主动发送消息的线程
send_thread = threading.Thread(target=server_send, daemon=True)
send_thread.start()
```

```
# 服务器回应
response = f"Received: {message}" # 构造回应消息
new_s.sendall((response + "<END>").encode('utf-8')) # 发送响应给客户端
```

### 三、 实现

源码见附录

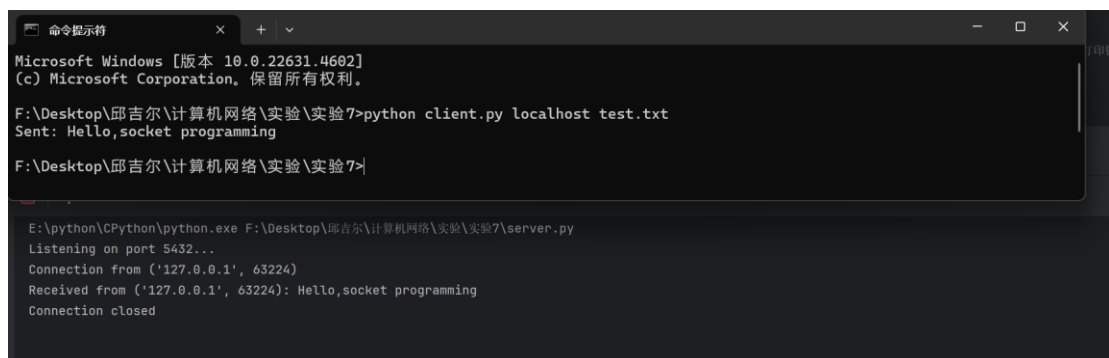
### 四、 测试

➤ 首先是基本的输入输出：

```
F:\Desktop\邱吉尔\计算机网络\实验\实验7>python client.py localhost
hello
|
```

```
Listening on port 5432...
Connection from ('127.0.0.1', 56683)
Received from ('127.0.0.1', 56683): hello
```

➤ 然后测试从文件接收信息输出：



```
命令提示符
Microsoft Windows [版本 10.0.22631.4602]
(c) Microsoft Corporation. 保留所有权利。

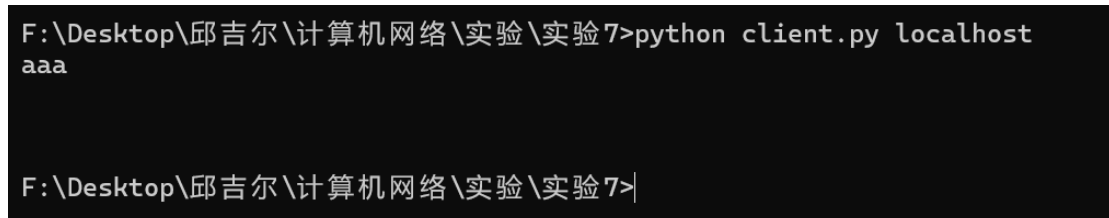
F:\Desktop\邱吉尔\计算机网络\实验\实验7>python client.py localhost test.txt
Sent: Hello,socket programming

F:\Desktop\邱吉尔\计算机网络\实验\实验7>

E:\python\CPython\python.exe F:\Desktop\邱吉尔\计算机网络\实验\实验7\server.py
Listening on port 5432...
Connection from ('127.0.0.1', 63224)
Received from ('127.0.0.1', 63224): Hello,socket programming
Connection closed
```

可以看到 client 支持从文件读入

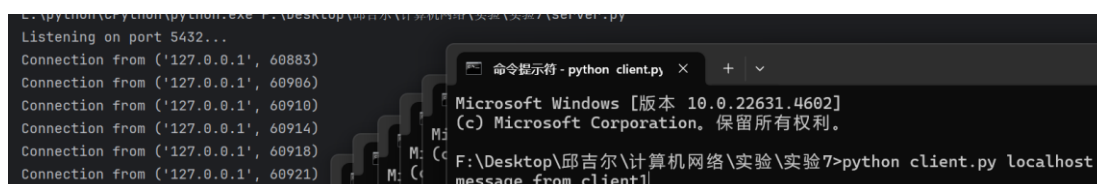
➤ 接着是两次回车作为结束输入的标志：



```
F:\Desktop\邱吉尔\计算机网络\实验\实验7>python client.py localhost
aaa

F:\Desktop\邱吉尔\计算机网络\实验\实验7>
```

➤ 接着测试是否支持 5 个以上客户端同时发送消息并逐一打印：

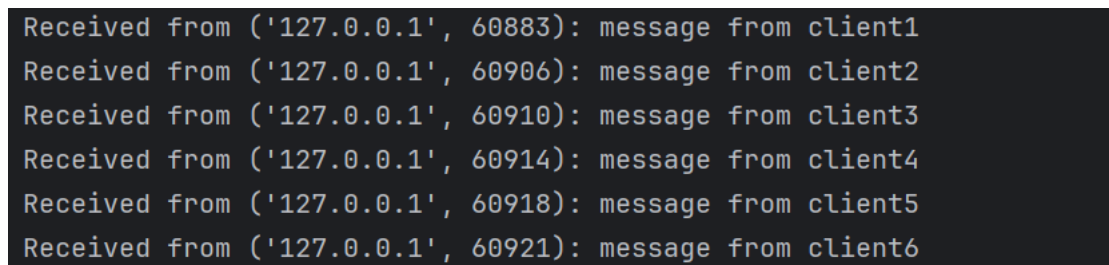


```
E:\python\CPython\python.exe F:\Desktop\邱吉尔\计算机网络\实验\实验7\server.py
Listening on port 5432...
Connection from ('127.0.0.1', 60883)
Connection from ('127.0.0.1', 60906)
Connection from ('127.0.0.1', 60910)
Connection from ('127.0.0.1', 60914)
Connection from ('127.0.0.1', 60918)
Connection from ('127.0.0.1', 60921)

命令提示符 - python client.py
Microsoft Windows [版本 10.0.22631.4602]
(c) Microsoft Corporation. 保留所有权利。

F:\Desktop\邱吉尔\计算机网络\实验\实验7>python client.py localhost
message from client1
```

由服务端打印信息（左侧）可知，已经完成同时与 6 个 client 的连接



```
Received from ('127.0.0.1', 60883): message from client1
Received from ('127.0.0.1', 60906): message from client2
Received from ('127.0.0.1', 60910): message from client3
Received from ('127.0.0.1', 60914): message from client4
Received from ('127.0.0.1', 60918): message from client5
Received from ('127.0.0.1', 60921): message from client6
```

发送信息后可以发现已经逐一打印

➤ 然后测试服务器绑定至错误的端口号时是否能提示出错误信息：

为了进行该测试我们首先需要编写一个同样占用 5432 端口的程序，这里为了偷懒（bushi）就直接把 server.py 的内容复制到了 test.py 里，然后我们先启动 test.py 再启动 server.py，就会发现 server.py 已经报错：

```
E:\python\CPython\python.exe F:\Desktop\邱吉尔\计算机网络\实验\实验7\server.py
server: bind failed: [WinError 10048] 通常每个套接字地址(协议/网络地址/端口)只允许使用一次。

进程已结束，退出代码为 0
```

➤ 那么再测试客户端连接至错误的 IP 地址/端口号时能否提示出错信息：

```
F:\Desktop\邱吉尔\计算机网络\实验\实验7>python client.py 192.168.1.256
client: unknown host: 192.168.1.256
```

➤ 接着测试可否在不同主机上运行：


```
F:\Desktop\邱吉尔\计算机网络\实验\实验7>python client.py 192.168.0.103
hello
|
```

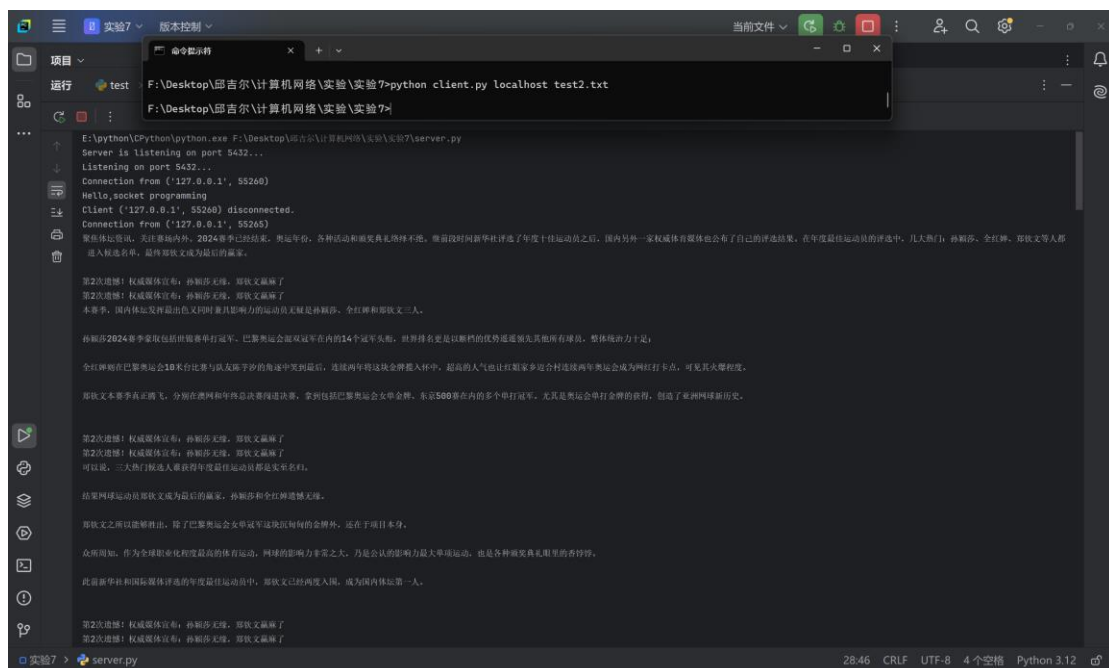
```
Server is listening on port 5432...
Listening on port 5432...
Connection from ('192.168.0.103', 56439)
Received from ('192.168.0.103', 56439): hello
```

可见可以完成

➤ 测试是否支持长文本消息传输：

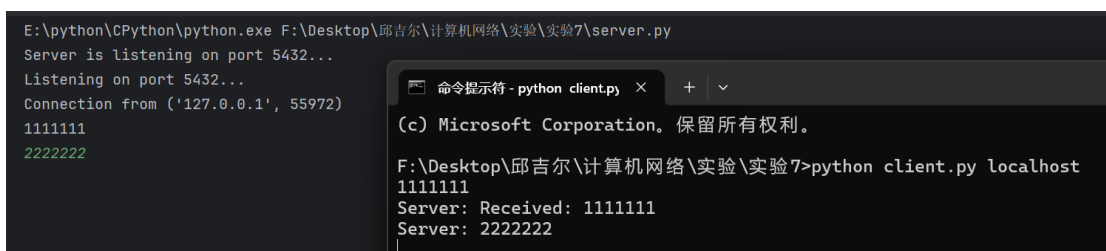
这里以这个 21KB 的文件做测试

 test2.txt	2024/12/31 22:36	文本文档	21 KB
---	------------------	------	-------



可以看到已经完整打印了出来（未完全截图）

➤ 测试是否支持双工通信:



左侧绿色字体为服务器端发送的信息，可见已经在客户端显示出来，至此已完成所有功能的测试