**#服务器代码**

import socket

import threading

import time

host = '192.168.3.110' # 服务器IP地址

prot = 2024 # 服务器端口号

# 存储所有已连接的客户端，key是客户端的标识符，value是套接字对象

clients = {}

all\_id = [] # 用于保存所有客户端标识符的列表

clients\_lock = threading.Lock() # 锁，用于线程安全地访问clients字典

# 接收并处理客户端消息的函数

def tcplink\_recv(client\_socket, address):

global clients

global all\_id

client\_id = None # 用于存储客户端的标识符

try:

# 接收客户端的标识符

client\_id = client\_socket.recv(1024).decode('utf-8')

print(f"连接到一个客户端，发送来的标识符为：{client\_id}")

# 将客户端信息存入字典，确保线程安全

with clients\_lock:

clients[client\_id] = client\_socket

all\_id.append(client\_id)

# 不断接收客户端发送的数据

while True:

try:

data = client\_socket.recv(1024).decode('utf-8')

if not data: # 客户端关闭连接

print(f"客户端{client\_id}断开连接")

break

# 打印接收到的消息，并将消息转发给所有连接的客户端

print(f"接收到来自{client\_id}的消息:{data}")

message = client\_id + ": " + data + '\n'

for i in clients.values():

i.send(message.encode('utf-8')) # 广播消息

except Exception as e:

print(f"客户端{client\_id}在中途异常断开连接:{e}")

break

except Exception as e:

print(f"客户端{client\_id}在发送标识连接时异常断开连接:{e}")

# 客户端断开连接时，移除客户端信息

with clients\_lock:

if client\_id and client\_id in clients:

del clients[client\_id]

all\_id.remove(client\_id)

# 关闭与客户端的连接

client\_socket.close()

# 主函数，创建服务器并接受客户端连接

def tcplink():

global clients

global all\_id

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 创建TCP套接字

server\_socket.bind((host, prot)) # 绑定地址和端口

server\_socket.listen(10) # 监听最大连接数为10

print("等待客户端连接...")

while True:

try:

client\_socket, address = server\_socket.accept() # 接受客户端连接

print(f"连接来自{address}")

# 为每个客户端创建独立线程处理接收数据

t = threading.Thread(target=tcplink\_recv, args=(client\_socket, address))

t.start()

except ConnectionResetError:

print(f"客户端{address}异常断开连接")

continue

except KeyboardInterrupt:

print("服务器关闭")

break

server\_socket.close()

print(f"客户端对象{address}关闭")

# 服务器主程序入口

if \_\_name\_\_ == '\_\_main\_\_':

t = threading.Thread(target=tcplink) # 启动服务器线程

t.start()

Socket编程（socket）：用于在客户端与服务器之间传输数据，基于TCP协议建立可靠连接。

多线程（threading）：使用多线程处理多个客户端连接，实现并发处理。

线程锁（Lock）：在多线程访问共享资源（如clients字典）时，确保线程安全。

**#login.py**

import sys

from PyQt6.QtWidgets import QApplication, QLineEdit, QPushButton, QWidget

from PyQt6 import uic

from room import talk # 导入聊天界面类

class login(QWidget):

def \_\_init\_\_(self):

super().\_\_init\_\_()

uic.loadUi("./login.ui", self) # 加载UI界面文件

self.name = self.findChild(QLineEdit, 'log\_name') # 获取用户名输入框

self.conn = self.findChild(QPushButton, 'login') # 获取登录按钮

self.conn.clicked.connect(self.jump) # 绑定点击事件，执行跳转到聊天界面

# 登录后的跳转操作

def jump(self):

self.tt = talk(name=self.name.text()) # 获取用户名，并创建聊天窗口

self.tt.show() # 显示聊天窗口

self.close() # 关闭登录窗口

PyQt6：用于开发图形用户界面（GUI），使客户端界面更加友好。

UI界面设计：通过uic.loadUi加载.ui文件，这种方式便于设计师和开发者分别进行界面设计与逻辑编程。

**#chat,py**

import time

from PyQt6.QtWidgets import QApplication, QLineEdit, QPushButton, QWidget, QTextEdit

from PyQt6 import uic

from PyQt6.QtCore import QThread, pyqtSignal

import socket

# 接收消息的线程类

class recv\_Thread(QThread):

finished = pyqtSignal(str) # 信号，用于更新UI

def \_\_init\_\_(self, tcpc):

super().\_\_init\_\_()

self.tcpc = tcpc # TCP连接# 保存TCP连接对象，tcpc 是通过 socket 连接到服务器的套接字

def run(self):

while True:

try:

data = self.tcpc.recv(1024) # 接收数据# 从套接字接收数据，最多接收1024字节

if data:

self.finished.emit(data.decode('utf-8')) # 触发信号更新UI# 如果接收到数据，发射信号，传递数据

else:

break

except Exception as e:

print(f"data\_error")

break

# 聊天窗口类

class talk(QWidget):

def \_\_init\_\_(self, name):

super().\_\_init\_\_()

uic.loadUi("./talk.ui", self) # 加载聊天界面UI

self.send = self.findChild(QPushButton, 'send') # 获取发送按钮

self.message = self.findChild(QTextEdit, 'message') # 获取消息输入框

self.all = self.findChild(QTextEdit, 'all') # 获取聊天记录框

self.send.clicked.connect(self.send\_mes) # 绑定发送按钮事件

self.name = name # 用户名

self.tcpc = None # TCP连接

self.recv\_th = None # 接收消息线程

self.sock\_conn() # 连接服务器

def sock\_conn(self):

server\_ip = '192.168.3.110' # 服务器IP

server\_port = 2024 # 服务器端口

try:

self.tcpc = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 创建TCP套接字

self.tcpc.connect((server\_ip, server\_port)) # 连接服务器

client\_id = self.name # 发送用户名作为标识

self.tcpc.send(client\_id.encode('utf-8'))

print("send\_name")

except Exception as e:

print(f"error {e}")

# 启动接收消息的线程

self.recv\_th = recv\_Thread(self.tcpc)

self.recv\_th.finished.connect(self.update\_ui) # 连接信号，更新UI

self.recv\_th.start()

# 更新聊天界面

def update\_ui(self, data):

self.all.append(data) # 显示收到的消息

# 发送消息函数

def send\_mes(self):

if not self.tcpc:

print("没有连接服务器")

return

try:

message = self.message.toPlainText() # 获取消息文本

if message:

self.tcpc.send(message.encode('utf-8')) # 发送消息

time.sleep(0.05) # 防止过快发送

else:

print("empty")

except ConnectionAbortedError:

print("连接被中止")

return -1

except ConnectionResetError:

print("连接被重置")

return -1

except Exception as e:

print(f"发送数据时发生异常: {e}")

return -1

# 关闭时处理资源清理

def closeEvent(self, event):

if self.recv\_th:

self.recv\_th.stop() # 停止接收线程

self.recv\_th.wait() # 等待线程结束

if self.tcpc:

self.tcpc.close() # 关闭TCP连接

event.accept() # 关闭事件

PyQt6多线程（QThread）：用于后台线程处理接收消息，避免界面卡顿。

信号与槽（Signal & Slot）：通过信号finished触发界面的更新，保证UI线程的安全。

**#main.py**

import sys

from PyQt6.QtWidgets import QApplication, QLineEdit, QPushButton, QWidget

from log import login # 导入登录界面

if \_\_name\_\_ == '\_\_main\_\_':

app = QApplication(sys.argv) # 创建Qt应用

log = login() # 创建登录窗口

log.show() # 显示登录窗口

sys.exit(app.exec()) # 进入应用事件循环

PyQt6应用程序生命周期：通过app.exec()启动Qt事件循环，使应用可以响应用户操作。

这个recv\_Thread类是一个用于接收数据的后台线程类，它的主要作用是从服务器（或客户端）接收数据，并将接收到的数据通过信号传递给主线程。它继承自 PyQt6 的 QThread 类，因此可以在后台独立运行，不会阻塞主线程。

详细解释：

1. 继承自QThread

QThread是 PyQt6 中用于创建后台线程的类。通过继承QThread，我们可以将长时间运行的任务放到子线程中执行，从而避免阻塞主线程，确保 GUI 界面的流畅性。

2.finished信号

finished是一个自定义信号（pyqtSignal）。在 PyQt 中，信号和槽（Signal & Slot）机制被用来进行线程间或组件间的通信。这个信号会在接收到数据后触发，并传递接收到的消息（str 类型）。

通过信号，可以把接收到的数据从 recv\_Thread 线程传递到主线程的 UI 组件进行显示，避免直接在子线程中更新 UI 界面（线程不安全）。

3. \_\_init\_\_ 构造函数

4. run 方法

run 方法是线程启动后会自动执行的方法。在 QThread 中，run 方法用于放置线程需要执行的任务。

这个方法不断从 tcpc 套接字中接收数据，使用 self.tcpc.recv(1024) 从连接中读取最多 1024 字节的数据。

如果成功接收到数据，则通过 self.finished.emit(data.decode('utf-8')) 发射 finished 信号，并将数据作为参数传递。这里 data.decode('utf-8') 是将接收到的字节数据解码为字符串。

如果接收不到数据（例如服务器关闭连接或其他异常），则会跳出循环，结束线程。

如果在接收数据的过程中出现任何异常，会捕获并打印错误信息，之后退出线程。

5. 线程与主线程通信

通过 pyqtSignal，recv\_Thread 线程可以将接收到的数据传递给主线程。

主线程（通常是UI线程）可以通过连接 finished 信号的槽函数来更新 UI（例如，在聊天界面显示新收到的消息）。这种方式避免了直接在子线程中更新UI，确保线程安全。