# 实验内容与环境描述

## 实验内容

使用Socket API通信实现文件传输**客户端**和**服务器端**2个程序，客户端发送文件传输请求,服务器端将文件数据发送给客户端，要求至少能传输1个文本文件和1个图片文件。客户端指定服务器的IP地址（127.0.0.1）和文件名。为防止重名,客户端将收到的文件**改名**后保存在目标目录下。

输出要求：

* 客户端：新文件名、传输总字节数，或者差错报告；
* 服务器端：客户端的IP地址和端口号，发送的文件数据总字节数，必要的差错报告(如文件不存在、连接中断等)。

## 实验环境

* 操作系统：Windows 11 Home
* 编程语言：Python 3.12.7
* 依赖库：本实验使用Python标准库中的 socket、os、argparse、threading、logging、ssl和datetime模块。
* 开发工具（IDE）：PyCharm
* 端口配置：使用9000作为服务器端的服务端口，客户端端口随机分配。

# 软件设计

## 数据结构

### TCPFileServer类（Server.py）

* **self.host**（str）：服务器监听的IP地址，默认为'0.0.0.0'，表示监听所有可用的网络接口；
* **self.port**（int）：服务器监听的端口号，默认为9000；
* **self.use\_ssl**（bool）：标志位，指示是否启用SSL/TLS加密通信；
* **self.certfile**（str）：SSL证书文件的路径，仅当self.use\_ssl为True时有效；
* **self.keyfile**（str）：SSL私钥文件的路径，仅当self.use\_ssl为True时有效；
* **self.sock**（socket对象）：服务器的监听套接字。在初始化时为None，在start()方法中创建并绑定；
* **self.running**（bool）：服务器运行状态标志，控制主循环的开启和关闭；
* **logging**（Logger对象）：用于记录服务器运行日志，将日志输出到server\_access.log文件。

### TCPFileClient类（Client.py）

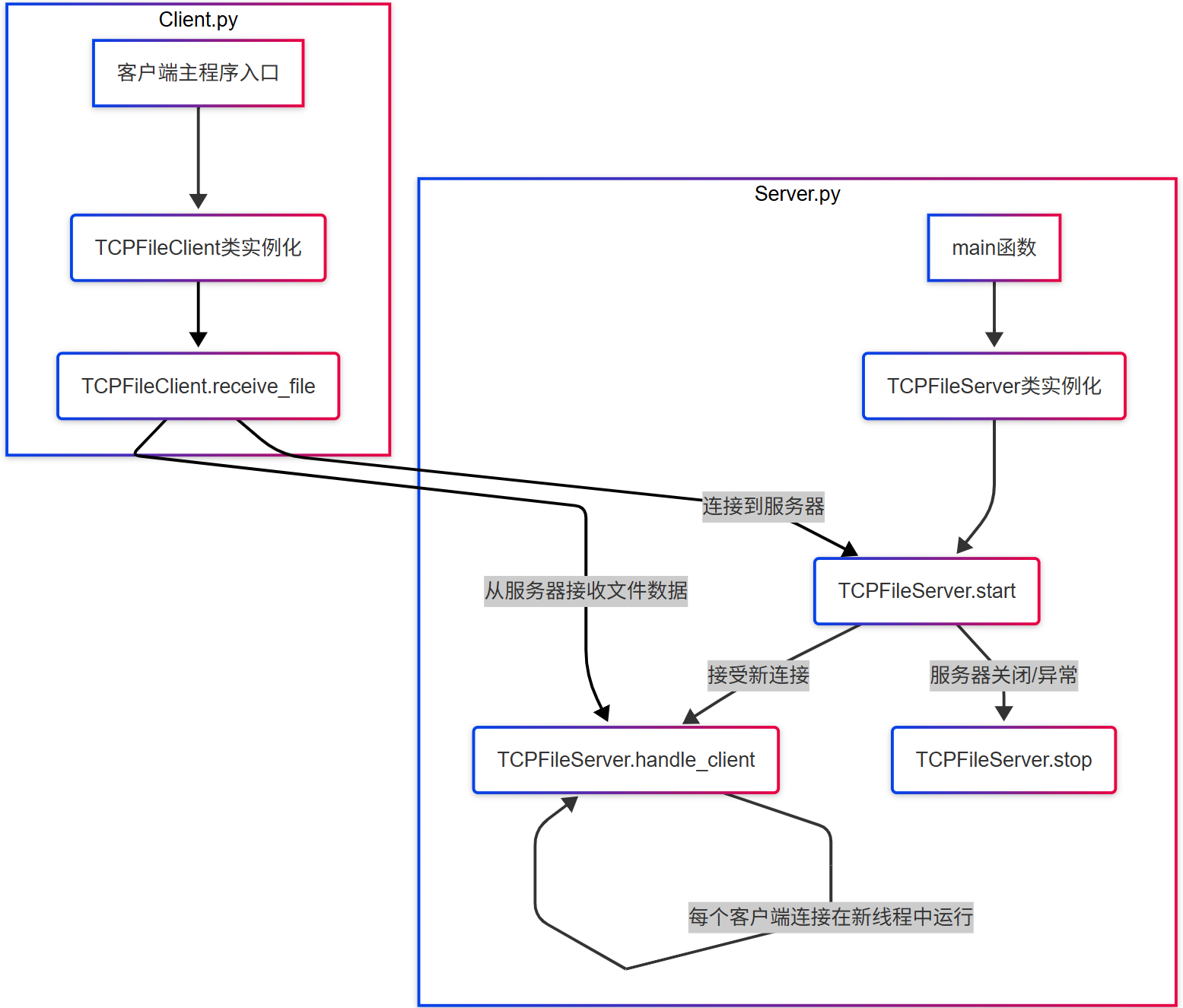
* **self.server\_ip**（str）：目标服务器的IP地址；
* **self.filename**（str）：客户端请求下载的文件名。

### 临时变量与数据传输格式

* **conn**（**socket对象）**：在服务器端，每个客户端连接被接受后，**accept()**方法返回的新套接字对象，用于与特定客户端进行通信（当使用SSL时，它会被封装**ssl.SSLSocket**对象）；
* **addr**（tuple）：客户端的地址信息，包含IP地址和端口号；
* **data**（**字节串**）：用于接收网络传输的数据，如客户端发送的文件名或文件内容；
* **chunk**（**字节串**）：文件读写和网络发送/接收时使用的每次操作的固定大小数据块（4096字节）。
* 文件名传输协议：
  + 客户端发送文件名时，将文件名编码为UTF-8并附加换行符；
  + 服务器接收到文件名后，会对其进行解码和清理。
* 文件传输响应协议：
  + 服务器向客户端发送文件传输状态和文件大小时，使用“**SUCCESS:<filesize>**”或“**ERROR:<message>**”格式，后跟换行符；
  + **filesize**为文件的字节大小。
* 日志记录数据：日志文件（**erver\_access.log**）中记录的信息格式为 **%(asctime)s [%(levelname)s] %(message)s**，时间格式为**YYYY-MM-DD HH:M:S**。

## 模块结构

### 程序调用流程图



### 服务器端（Server.py）

* **TCPFileServer**类：
  + **\_\_init\_\_(self, host, port, use\_ssl, certfile, keyfile)**：初始化服务器实例，设置监听地址、端口、SSL配置和日志；
  + **start(self)**：启动服务器，创建并绑定监听套接字，进入主循环等待和接受客户端连接。在此方法中处理SSL上下文的创建和应用，无参数；
  + **stop(self)**：停止服务器，关闭监听套接字并清理资源，无参数；
  + **handle\_client(self, conn, addr)**：在一个新线程中执行的函数，负责处理单个客户端连接。包括接收文件名、验证文件路径、读取文件内容并将其发送给客户端。同时负责异常处理和日志记录：
    - conn：已连接的客户端套接字对象；
    - addr：客户端的IP地址和端口元组；
* **main()**：程序的入口点，无参数。

1. 使用argparse解析命令行参数（host, port, ssl, cert, key），创建服务器文件存储目录 (./files\_in\_server)；
2. 实例化 TCPFileServer 对象并调用其 start() 方法；
3. 包含 try-except-finally 块以处理 KeyboardInterrupt 并确保服务器正常停止。

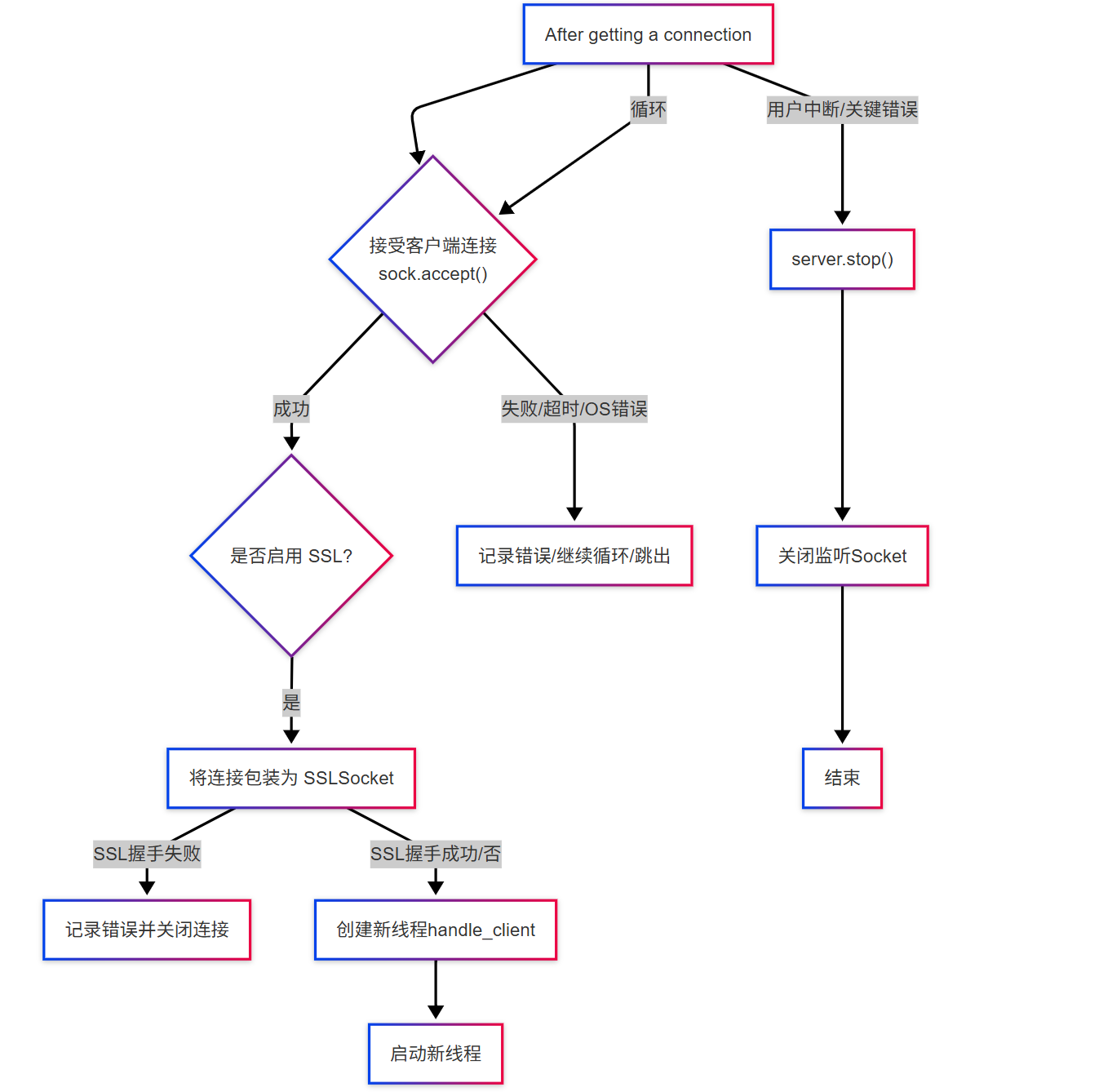
### 客户端（Client.py）

* **TCPFileClient**类：
  + **\_\_init\_\_(self, server\_ip, filename)**：初始化客户端实例，设置目标服务器IP和要请求的文件名；
  + **receive\_file(self)**：连接到服务器，发送请求的文件名，接收服务器响应（文件大小或错误信息），然后接收文件数据并写入本地文件（文件名将被重命名为包含日期和时间戳的格式）。
* **if \_\_name\_\_ == '\_\_main\_\_':**块：程序的入口点。

1. 本实验中硬编码服务器IP为 "127.0.0.1"；
2. 提示用户输入要下载的文件名；
3. 根据用户输入实例化TCPFileClient对象并调用receive\_file()方法；
4. 输入 'quit' 退出当前程序。

## 算法流程

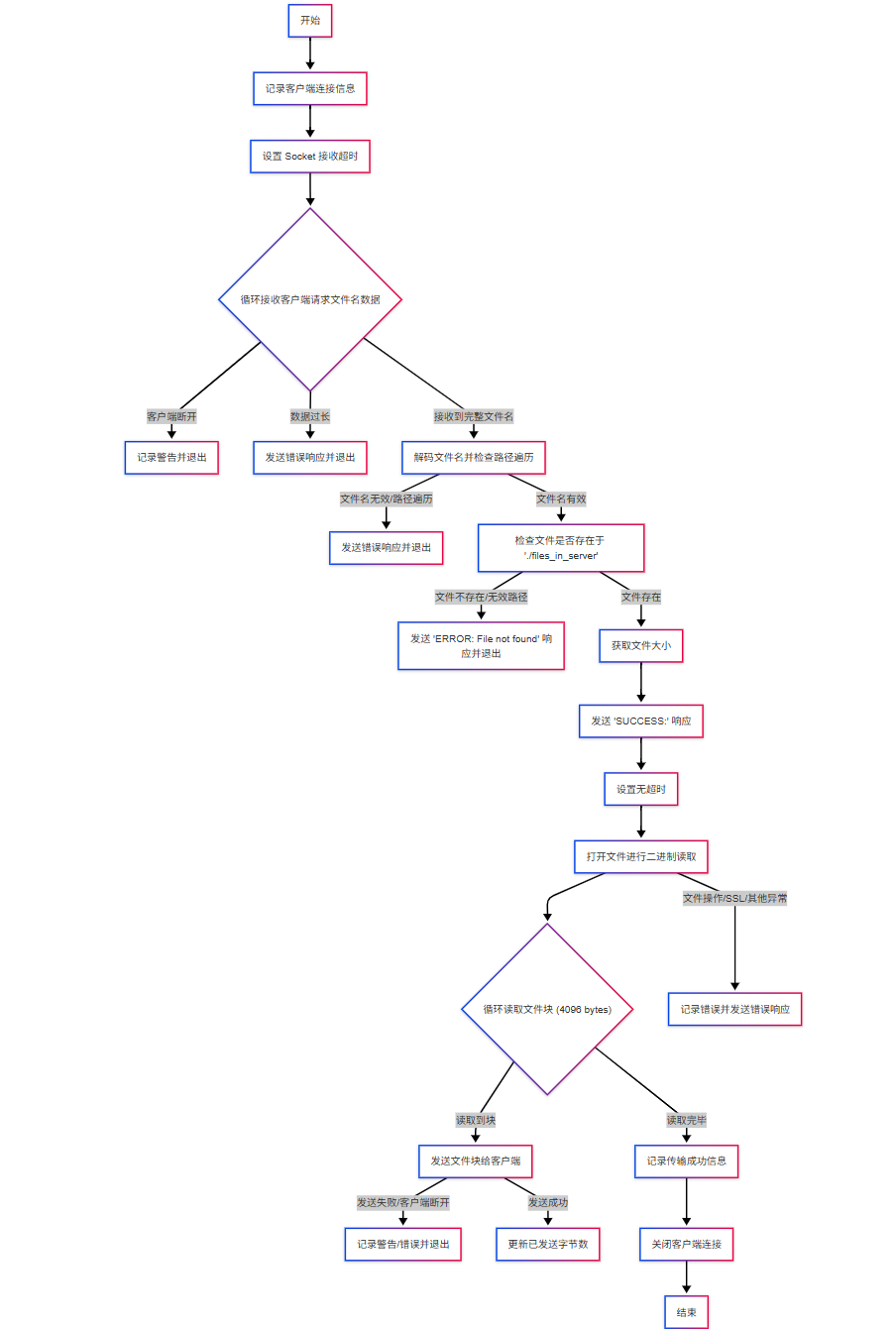
### 服务器端主流程



**服务器端主流程负责启动、监听、客户端连接接受及资源清理。**

程序启动后，解析命令行参数并创建文件存储目录；接着实例TCPFileServe对象并调用其start()方法，该方法会创建并绑定监听Socket，设置端口重用，然后进入主循环开始监听连接；如果启用SSL，服务器会加载证书和密钥，若加载失败则停止；在主循环中，服务器阻塞接受客户端连接，如果启用SSL则将连接包装为SSLSocket，若SSL握手失败则关闭连接；对于每个成功连接的客户端，服务器会创建一个新线程来调用handle\_client方法进行独立处理；主循环持续等待新连接，直到用户中断或发生关键错误，此时会调用server.stop()关闭监听Socket并结束程序。

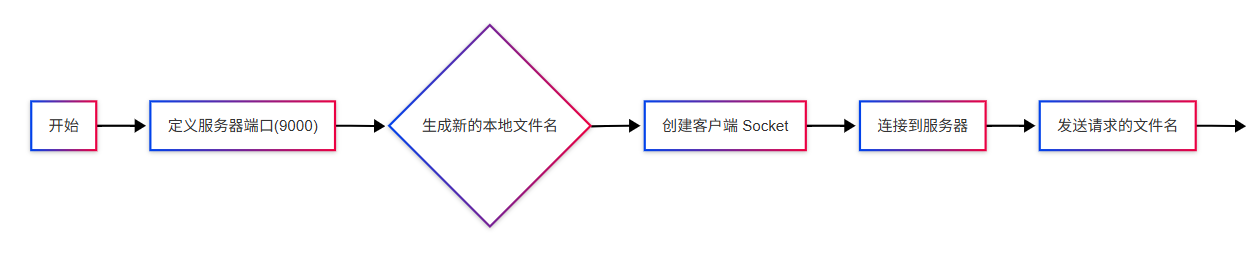
### 服务器端handle\_client()

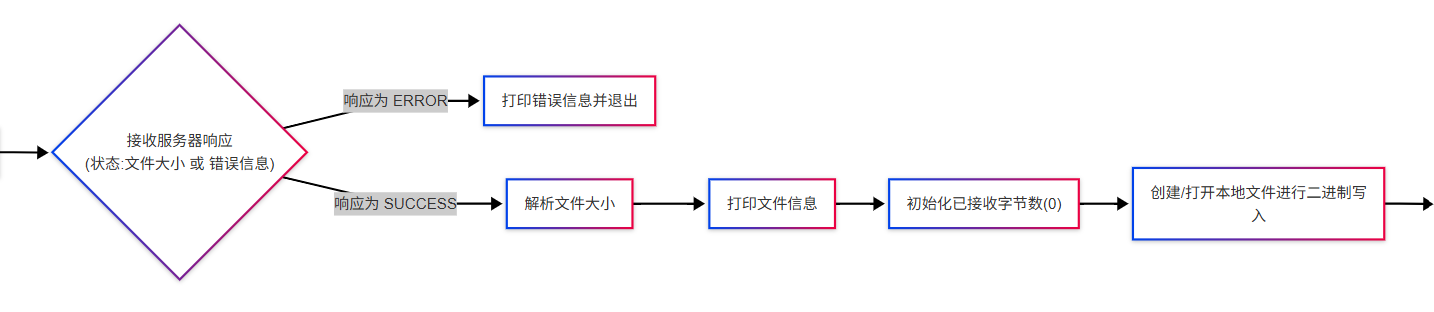


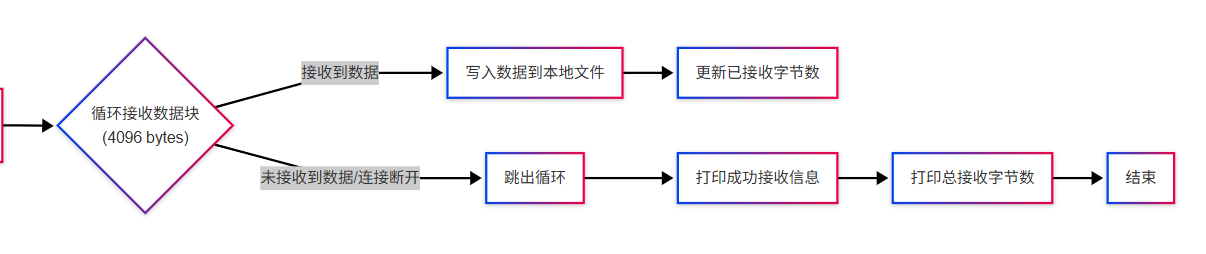
**handle\_client()方法在线程中运行，处理单个客户端的请求和文件传输。**

线程启动后，首先记录客户端连接信息并设置Socket接收超时（保证资源可用）；接着，它循环接收客户端请求的文件名数据，如果客户端断开或数据过长则记录错误并退出；成功接收文件名后，进行解码和路径遍历安全检查，若文件名无效则发送错误响应；如果文件存在且路径合法，服务器获取文件大小并发送"**SUCCESS:<filesize>**"响应，然后取消Socket超时，打开文件以4096字节分块循环读取并发送给客户端；在传输过程中，如果发送失败或客户端断开，则记录警告并退出；文件读取完毕后，记录传输成功信息并关闭客户端连接。

### 客户端（仅解读receive\_file子程序）







**receive\_file()方法是客户端核心逻辑，负责请求、接收和保存文件。**

客户端首先定义服务器端口并生成一个唯一的本地文件名用于保存下载；接着，它创建TCP套接字并连接到服务器，然后发送请求的文件名；客户端等待并接收服务器的响应，如果响应为错误则打印错误信息并退出；若响应成功，客户端解析文件大小，打印接收提示，初始化接收字节计数器，并创建本地文件准备写入；随后，客户端循环从网络接收4096字节的数据块并写入本地文件，同时更新已接收字节数；当不再接收到数据（表示传输完成或连接断开）时，跳出循环，打印文件接收成功信息和总接收字节数，最后结束文件接收过程。

## 实现要点

### TCP连接建立与管理

* **服务器端**：使用socket.socket()创建 TCP 套接字，通过 bind() 绑定 IP 和端口，listen(5)监听，accept()接受连接，设置 socket.SO\_REUSEADDR允许端口重用。
* **客户端**： 使用socket.socket()创建TCP套接字，并通过 connect()连接服务器。

### 并发处理

服务器端采用**多线程**模型。accept()接受连接后，为每个客户端创建新的**Thread**，调用**self.handle\_client()**，实现并发文件传输。

### 文件传输协议

* + - * 文件名请求：客户端发送UTF-8编码的文件名，后跟换行符；
      * 响应与文件大小：服务器响应**SUCCESS:<filesize>**或**ERROR:<message>**，后跟换行符，客户端据此预知文件大小或处理错误；
      * 数据传输：文件内容以4096字节分块传输，循环收发直到完成。

### 错误处理与鲁棒性

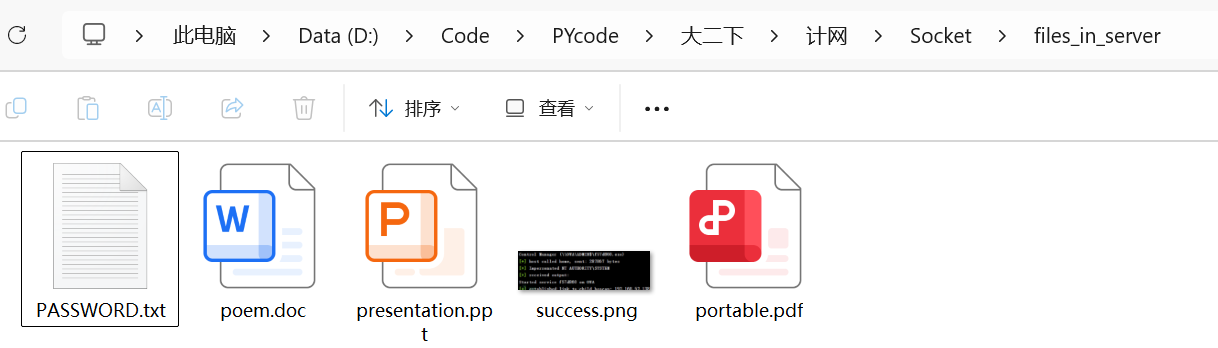
* + - * **连接管理**：客户端使用with socket.socket(...)确保套接字自动关闭；
      * **超时机制**：服务器端接收文件名时设置10秒超时，文件传输时取消超时；
      * **安全管理**：
        + **文件路径安全**：服务器端检查文件名中的路径遍历字符（..、/、\），确保只访问指定目录下的文件；
        + **文件操作错误**：捕获FileNotFoundError、IOError等，返回相应错误信息；
        + **网络异常**：捕获ConnectionResetError（客户端强制关闭）、BrokenPipeError（发送时对端断开）、ConnectionAbortedError（客户端中止连接）等连接中断异常，记录警告并关闭连接；
        + **通用异常捕获**：使用try...except Exception捕获未预料的错误，提高稳定性。

### 日志记录

服务器端使用logging模块记录详细运行信息（连接、请求、传输状态、错误、警告）到server\_access.log，级别为 INFO。

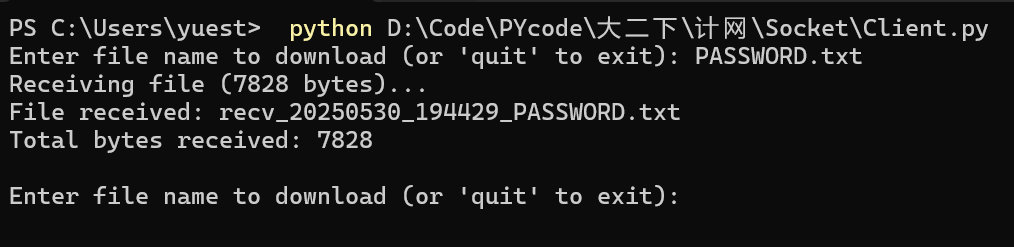
# 实验结果演示与分析

## 功能测试

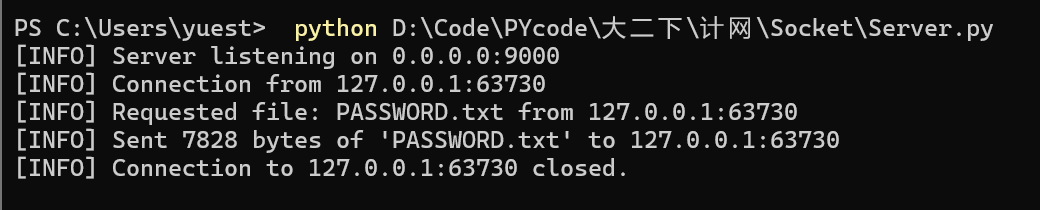


### 传输文本文件

客户端：

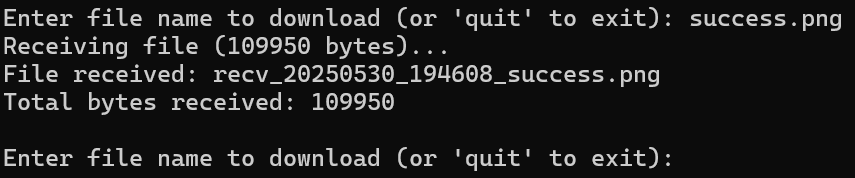


服务器端：

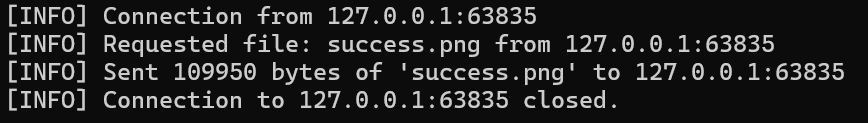


### 传输图片文件

客户端：

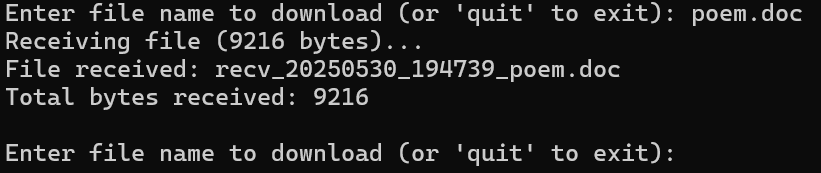


服务器端：

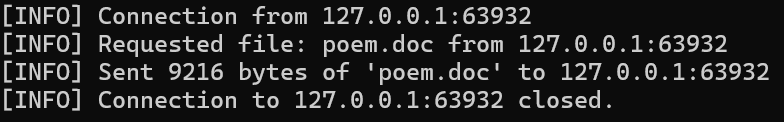


### 传输word文档

客户端：

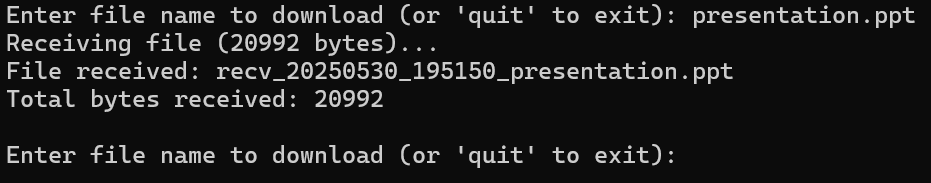


服务器端：

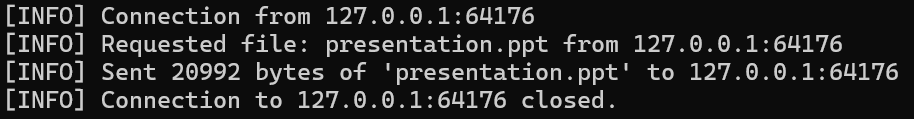


### 传输ppt文件

客户端：

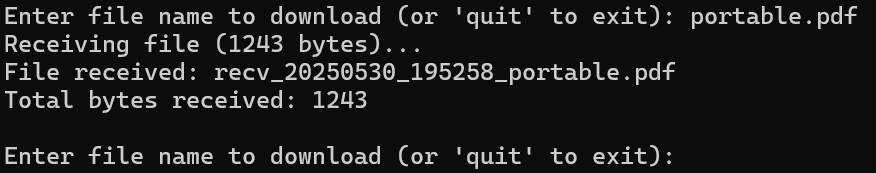


服务器端：

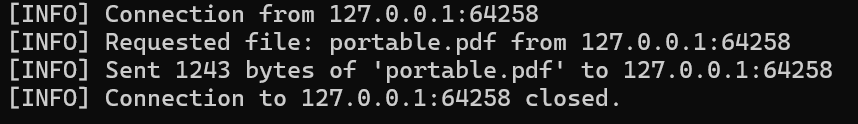


### 传输pdf文件

客户端：



服务器端：



## 差错处理功能解释

### 日志文件

* + - * 作用： 在服务器启动时配置了一个基本的日志记录系统；
      * **filename='server\_access.log'**：指定所有日志信息将写入到名为 server\_access.log 的文件中，这使得服务器的运行状态和错误信息可以持久化保存，即使终端窗口关闭或服务器重启，历史记录依然存在；
      * **level=logging.INFO**：设置日志记录的最低级别为INFO。这意味着INFO、WARNING、ERROR和CRITICAL级别的消息都会被记录，而DEBUG级别的消息则不会；
      * **format**和**datefmt**：定义了日志消息的格式，包含时间戳、日志级别和消息内容，使得日志更易读和分析。

### 服务器启动阶段的错误处理

* **功能：**负责捕获并处理服务器在**持续监听和接受客户端连接过程中**可能遇到的错误，包括套接字接受连接时的操作系统错误（如文件描述符耗尽）、超时，以及在主循环中发生的任何其他未预料的异常；
* **目的**：确保服务器即使在处理单个连接请求时遇到问题，也能保持持续运行，不影响其他连接。

### SSL 配置的错误处理

* + - * **作用：**专门处理 SSL 证书和密钥加载过程中的错误。
      * 检查文件是否存在：如果未提供证书或密钥文件，会打印错误并阻止服务器启动；
      * **ssl.SSLError**：捕获 SSL 库特有的错误，例如证书格式不正确、密码错误等；
      * **FileNotFoundError**：捕获证书或密钥文件不存在的错误。

### 客户端连接处理的错误处理

负责捕获并处理**单个客户端连接生命周期内**可能发生的所有错误。这包括从接收文件名到文件传输过程中的各种网络错误（如客户端超时、连接重置）、文件操作错误（如文件未找到、读取失败）、SSL错误，以及在处理特定客户端时发生的任何其他异常。其核心目的是隔离每个客户端的错误，防止一个客户端的问题影响到其他客户端或服务器的整体稳定性，并确保客户端套接字在处理结束后被正确关闭。

# 实验总结和心得体会

## 问题与解决方法

### 对接问题

* + - * 问题描述：当两部分代码进行交互时，无法完成任何文件传输，一直显示"ERROR: File 'PASSWORD.txt' not found or invalid path."的错误信息。
      * 问题分析：首先考虑文件内容不匹配，检查后确认无误；然后交流思路发现客户端首先请求一个response但是服务器端并未有此处理，因此只能显示未成功找到文件。
      * 解决办法：在服务端增加response生成步骤用于确认传递状态。

### 文件请求问题

* + - * 问题描述：在命令行中无法成功请求服务器文件，即使文件存在。最初尝试输入PASSWORD.txt，但收到"ERROR: File 'PASSWORD.txt' not found or invalid path."的错误信息，文件路径不匹配导致文件找不到。
      * 问题分析：服务器端Server.py代码中，文件路径是通过 os.path.join("./files\_in\_server", filename) 构建的，这意味着服务器期望文件位于与 Server.py 脚本同级目录下的 files\_in\_server 文件夹内。然而，最初运行 Server.py 时，它没有在预期工作目录下执行，导致无法找到D:\Code\PYcode\大二下\计网\Socket\files\_in\_server下的文件。
      * 解决办法：在命令行中切换到脚本所在目录后运行：在启动Server.py和Client.py之前，先使用cd命令切换到它们各自所在的目录。

## 反思与不足

尽管本次实验取得了预期的效果，但在实践过程中也依然有一些可以改进和深入探索的方面：

* + - * **错误处理的粒度与健壮性提升**：虽然在多个try-except块中捕获了多种异常，但有些通用 Exception 捕获可能过于宽泛，导致难以精确识别和处理特定类型的错误。未来可以考虑更细致地划分异常类型，针对性地进行处理，从而提供更准确的错误信息和恢复机制。对于一些临时的网络波动或资源不足，可以考虑引入重试机制，而不是直接报错或断开连接，以提高服务的韧性。
      * **安全性考量不足**：当前的路径遍历预防虽然初步有效，但对于更复杂的恶意文件名构造（例如unicode编码、双重编码、短文件名等）可能仍有漏洞。在生产环境中，需要采用更严格的文件名白名单校验或更复杂的路径规范化算法来彻底杜绝此类风险。
      * **传输加密的默认启用**： 虽然实现了SSL选项，但并未强制启用。在传输敏感数据时，应默认启用加密，并考虑证书的自动化管理和吊销机制。
      * **性能优化与资源管理**：当前的文件传输是同步进行的，对于单个大文件传输，可能会阻塞线程。可以考虑引入异步I/O或线程池/进程池等更高级的并发模型，以更高效地利用系统资源，尤其是在处理大量并发连接或超大文件传输时。对sock.recv()和sock.send()的缓冲区大小的选择相对固定（4096字节）。在不同网络环境下，调整缓冲区大小可能会显著影响传输效率。未来可以考虑动态调整或提供配置选项。
      * **可维护性与可配置性**：服务器的端口、SSL证书/密钥路径等参数目前主要通过命令行参数配置。可以考虑引入配置文件（如YAML, INI），使得部署和管理更加灵活，并允许更复杂的配置选项。同时，日志记录虽然已实现，但日志轮转、日志级别动态调整等高级功能尚未涉及。在长期运行的服务中，日志文件的管理是一个重要方面。
      * 客户端交互的增强：客户端的错误提示相对简单。可以提供更详细的错误信息，指导用户进行排查（例如“文件不存在，请检查文件名”）。