

**FTP实验**

学号： 2022011599

专业： 计算机科学与技术

姓名： 黄思源

# 目录

1 实验概述

1.1 实验目的

1.2 实验内容

2 FTP 实验实现

2.1 基本功能介绍

2.2 套接口函数的应用

2.3 文件操作函数的应用

2.4 客户端关键函数与功能描述

2.5 服务器功能实现与关键函数

3 FTP实验结果展示

3.1 客户端命令

4 Web 服务器实现

4.1 技术架构

4.2 核心功能

4.3 界面展示

5 爬虫部分实现

5.1 技术架构

5.2 核心功能

6 思考题解答

7 实验结论

8 附录

8.1 GitHub 项目链接

8.2 client.cpp

8.3 server.cpp

8.4 app.py

8.5 index.html

8.6 scraper.py1 实验概述

## 1.1 实验目的

本次实验旨在通过在 Linux 系统上使用 C/C++ 语言实现 FTP 客户端与服务器，深入理解 FTP 协议的原理与细节，掌握 Socket 接口编程和 TCP/IP 网络应用开发技能。同时，基于 Web 技术实现文件传输系统，并开发简易网络爬虫，提升综合网络编程能力。

## 1.2 实验内容

实现 FTP 客户端与服务器，支持控制通道与数据通道分离，实现get、put、pwd、dir等基本命令，支持主动 / 被动模式及登录验证。

使用 Python Flask 框架搭建 Web 文件传输系统，实现文件上传、下载、目录管理等功能。

基于 Socket 和 BeautifulSoup 开发网络爬虫，抓取指定网页的文本和图片资源并保存。

# 2 FTP 实验实现

## 2.1 基本功能介绍

本实验实现的 FTP 系统遵循 RFC959 规范，客户端与服务器通过 TCP 协议建立连接：

控制通道：固定端口（服务器 21 端口），用于传输 FTP 命令（如登录、文件操作指令）。

数据通道：动态端口，用于文件传输或目录列表获取，支持主动模式（服务器主动连接客户端）和被动模式（客户端主动连接服务器）。

支持的核心命令包括：

connect <host> <port>：连接到服务器

login <username> <password>：登录

ls/dir：列出当前目录内容

cd <directory>：切换目录

get <filename>：从服务器下载文件

put <filename>：上传文件到服务器

passive：设置客户端为被动模式（默认）

active：设置客户端为主动模式

quit：断开连接并退出

## 2.2 套接口函数的应用

socket()： 创建套接字，指定通信域（AF\_INET）、套接字类型（SOCK\_STREAM，表示 TCP 连接）和协议。

bind()： 将套接字绑定到本地的 IP 地址和端口号。服务器端使用此函数绑定到固定的监听端口（如 21），被动模式下服务器的数据连接监听套接字也会绑定到随机端口。

listen()： 服务器端套接字进入监听状态，等待客户端连接请求。

accept()： 服务器端接受客户端的连接请求，返回一个新的已连接套接字，用于与特定客户端进行通信。在被动模式下，服务器的数据连接监听套接字也会使用此函数接受客户端的数据连接请求。

connect()： 客户端使用此函数向服务器发起连接请求，建立控制连接。在主动模式下，服务器会使用此函数向客户端发起数据连接。在被动模式下，客户端会使用此函数向服务器发起数据连接。

send() / recv()： 用于在已建立的套接字上发送和接收数据。控制连接用于发送命令和接收响应，数据连接用于传输文件数据或目录列表数据。

setsockopt()： 用于设置套接字选项，例如 SO\_REUSEADDR 和 SO\_REUSEPORT，以允许端口重用，避免 TIME\_WAIT 状态导致的端口占用问题。

close()： 关闭套接字，释放相关资源。

## 2.3 文件操作函数的应用

文件传输：

使用fopen()、fread()、fwrite()实现文件读写，结合sendfile()优化大文件传输效率。

传输前通过stat()获取文件元数据（如大小、修改时间）。

目录操作：

使用opendir()、readdir()遍历目录，chdir()切换工作目录，mkdir()创建目录。

## 2.4 客户端关键函数与功能描述

客户端程序 (ftp\_client/client.cpp) 提供了一个交互式命令行界面，负责解析用户输入并与服务器进行通信。

main() 函数： 客户端程序的入口点，负责套接字初始化、连接服务器、用户登录流程，以及进入主循环以接收用户输入并调用相应的命令处理函数。

send\_command(int sock, const std::string& command)： 核心通信函数，负责向服务器的控制连接发送完整的 FTP 命令（自动添加 \r\n 终止符），并等待接收服务器的响应字符串。

open\_passive\_data\_connection\_client(const std::string& ip, int port)： 在被动模式下，客户端根据服务器返回的 IP 和端口，主动发起数据连接。

open\_active\_data\_listener\_client(std::string& client\_ip, int& client\_port)： 在主动模式下，客户端打开一个监听套接字，等待服务器来连接，并将监听的 IP 和端口返回给服务器。

handle\_list\_command(int sock)： 处理用户输入的 ls 或 dir 命令。根据当前设置的模式（主动或被动），先与服务器协商数据连接（发送 PASV 或 PORT），然后发送 LIST 命令，并通过建立的数据连接接收目录列表数据并显示。

handle\_get\_command(int sock, const std::string& filename)： 处理 get <filename> 命令。协商数据连接后，发送 RETR <filename> 命令，并通过数据连接接收文件内容并保存到本地文件。

handle\_put\_command(int sock, const std::string& filename)： 处理 put <filename> 命令。协商数据连接后，发送 STOR <filename> 命令，并通过数据连接读取本地文件内容并发送到服务器。

set\_passive\_mode() / set\_active\_mode()： 客户端内部的辅助函数，用于切换客户端的数据传输模式标志位，影响后续数据传输命令的行为。

quit\_command(int sock)： 发送 QUIT 命令给服务器，接收服务器的 221 Goodbye 响应，然后关闭控制连接并终止客户端程序。

login\_user(int sock, const std::string& username, const std::string& password)： 负责客户端的登录流程，发送 USER 和 PASS 命令，并根据服务器响应判断登录是否成功。

change\_directory(int sock, const std::string& path)： 处理 cd <directory> 命令，发送 CWD <directory> 命令给服务器。

## 2.5 服务器功能实现与关键函数

服务器程序 (ftp\_server/server.cpp) 负责监听客户端连接，管理会话状态，并响应客户端的 FTP 命令。

main() 函数： 服务器程序的入口点。负责创建主监听套接字（21 端口），绑定地址，进入监听状态。在主循环中，它使用 accept() 接收新的客户端连接，并为每个新连接创建一个独立的线程 (client\_handler\_thread) 来处理。

client\_handler\_thread(void \*socket\_desc)： 每个客户端连接的处理线程函数。它负责初始化 ClientState，发送欢迎消息，然后进入循环，持续接收、解析客户端命令，并调用相应的处理函数。

ClientState 结构体： 用于存储每个客户端的会话状态，包括其控制套接字、当前工作目录、登录状态、用户名、数据连接相关信息（如 IP、端口、模式、数据套接字文件描述符）。

send\_response(int client\_socket, const std::string& response)： 向指定客户端的控制连接发送 FTP 响应字符串。

handle\_user(ClientState& client\_state, const std::string& username)： 处理 USER 命令，保存客户端提供的用户名。

handle\_pass(ClientState& client\_state, const std::string& password)： 处理 PASS 命令，根据预设的用户数据库验证用户名和密码，更新客户端的登录状态。

open\_passive\_data\_listener(int control\_socket, std::string& server\_ip, int& data\_port)： 处理 PASV 命令。服务器打开一个随机高位端口并监听，将服务器的 IP 和该端口信息格式化后返回给客户端。

accept\_passive\_data\_connection(int data\_listener\_sock)： 在被动模式下，服务器在 open\_passive\_data\_listener 建立的监听套接字上接受客户端的数据连接请求。

open\_active\_data\_connection(int control\_socket, const std::string& client\_ip, int client\_port)： 处理 PORT 命令。服务器根据客户端提供的 IP 和端口，主动发起数据连接。

handle\_list\_data(ClientState& client\_state)： 处理 LIST 命令。根据当前模式，建立数据连接，遍历 client\_state.current\_dir 中的文件和目录，将列表数据通过数据连接发送给客户端。

handle\_retr\_data(ClientState& client\_state, const std::string& filename)： 处理 RETR 命令。建立数据连接，打开指定文件，将文件内容通过数据连接发送给客户端。

handle\_stor\_data(ClientState& client\_state, const std::string& filename)： 处理 STOR 命令。建立数据连接，在服务器端创建或打开文件，从数据连接接收文件内容并写入文件。

handle\_pwd(int client\_socket, const std::string& current\_dir)： 处理 PWD 命令，返回客户端当前在服务器上的工作目录。

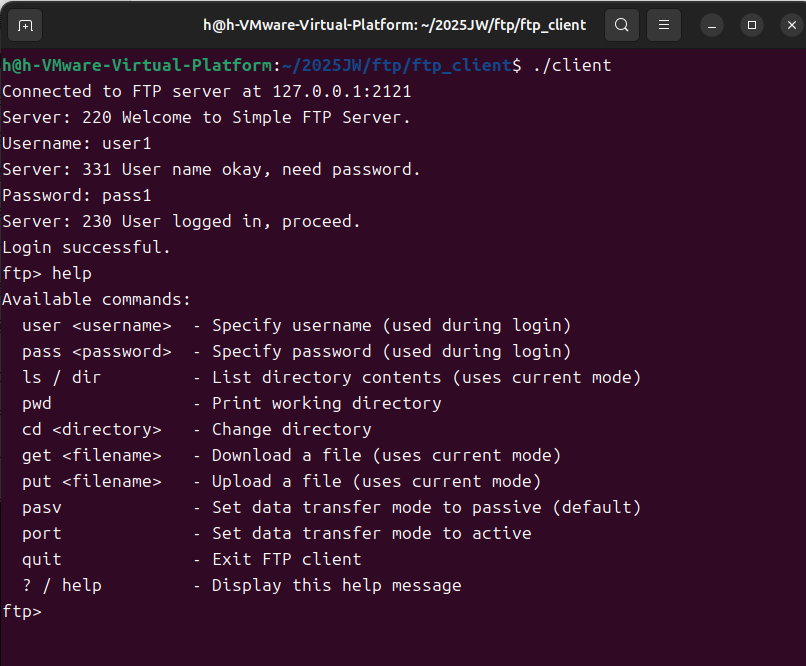
handle\_cwd(int client\_socket, std::string& current\_dir, const std::string& path)： 处理 CWD 命令，改变客户端在服务器上的工作目录。

handle\_quit(ClientState& client\_state)： 处理 QUIT 命令，发送 221 Goodbye 响应，并负责清理客户端资源，关闭套接字。

# 3 FTP实验结果展示

## 3.1 客户端命令

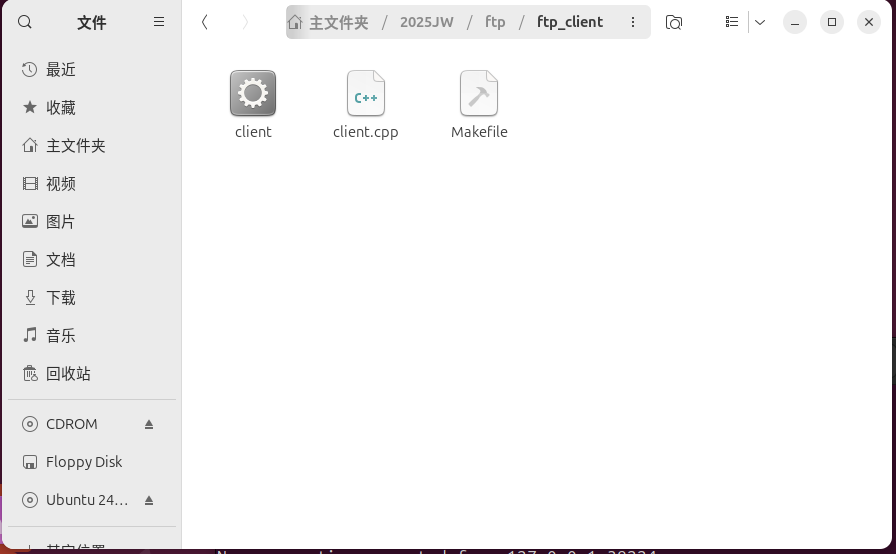
所有命令展示（使用help可以得到所有指令的提示）



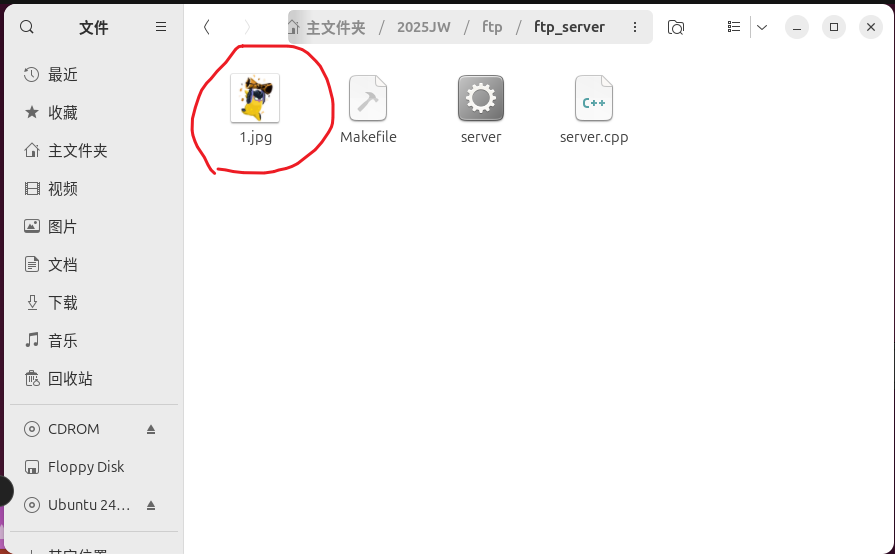
下面只演示关键功能（get、put）部分，其他功能可以自行下载我的代码尝试。

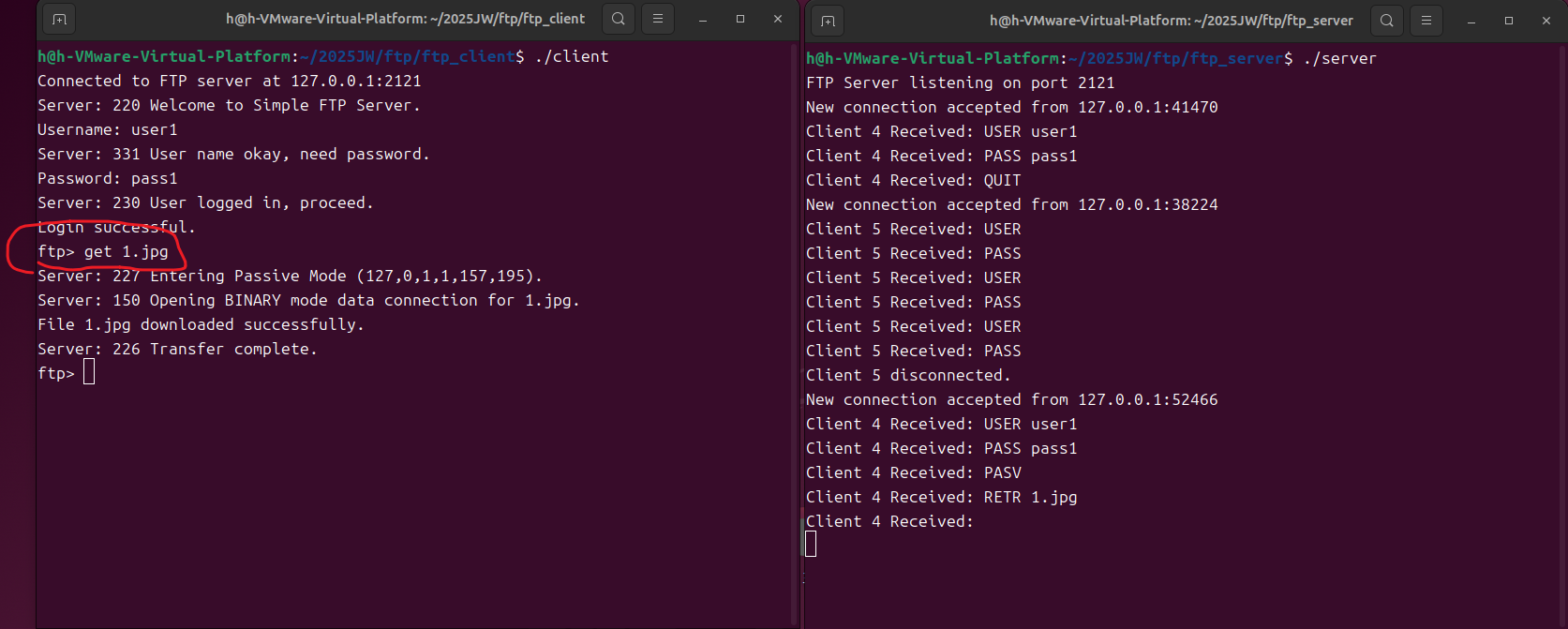
Get之前

客户端的文件

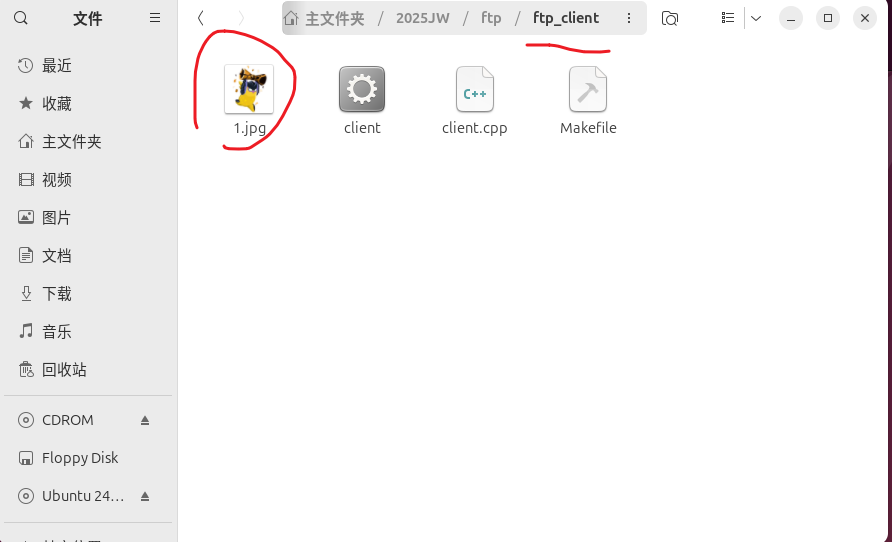


服务端的文件，有一个1.jpg，接下来我们会在客户端通过get指令获取它



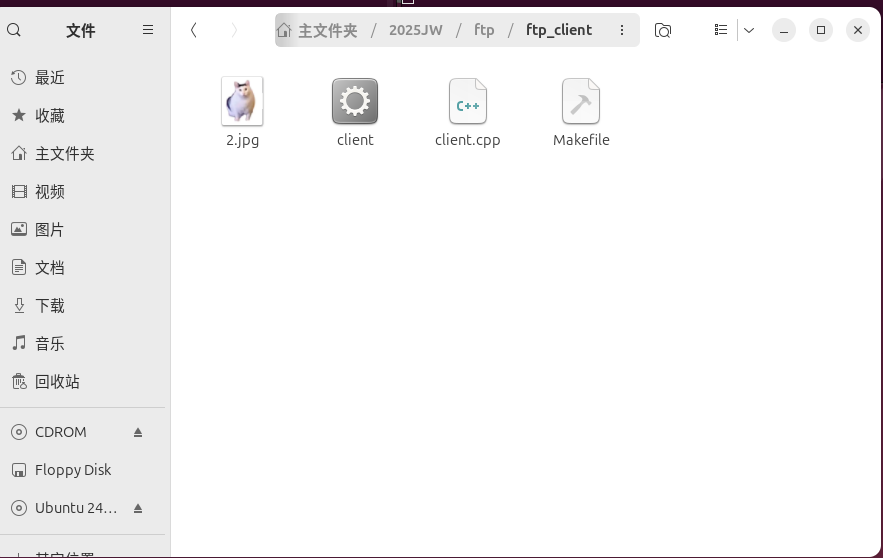


可以看到，输入get 1.jpg指令后，客户端获取到该图片了。

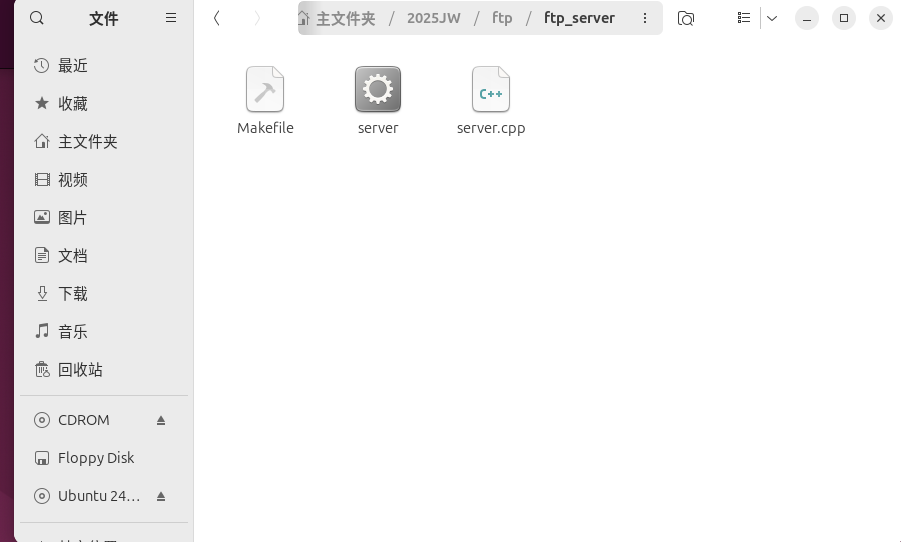


同理，下面演示put。

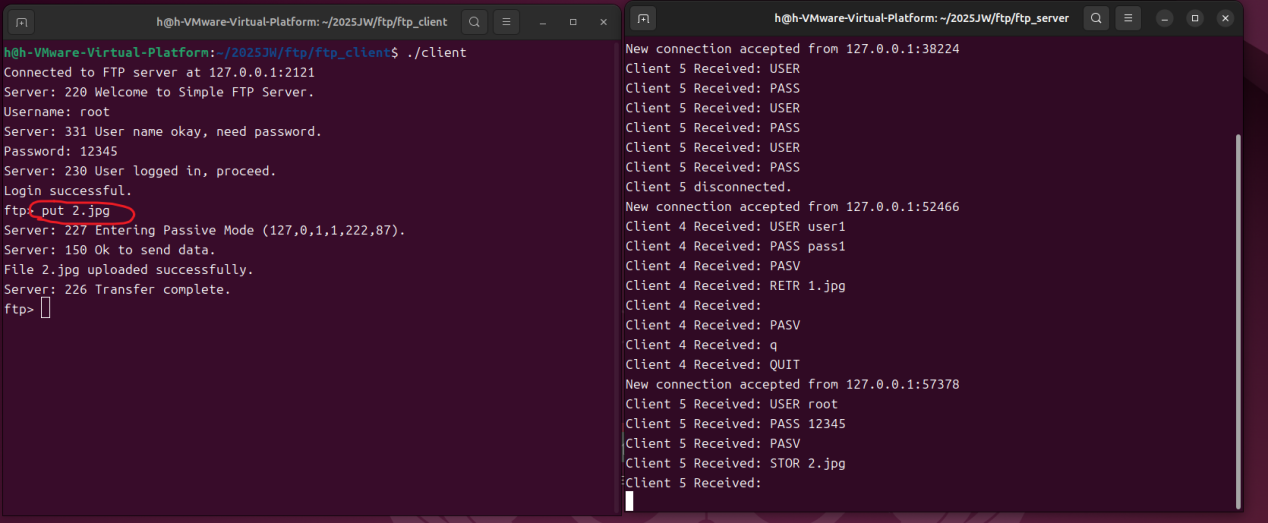
这是put之前，

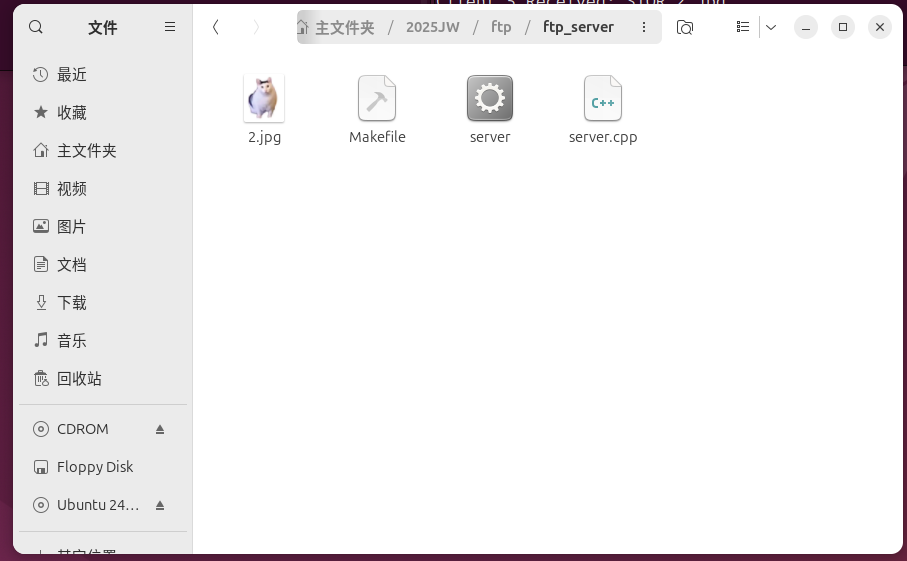
客户端的文件

服务器端的文件



我们会通过客户端的put指令，将2.jpg文件从客户端传给服务器端。





# 4 Web 服务器实现

## 4.1 技术架构

后端：Python Flask，实现轻量级 Web 服务。

前端：HTML/CSS+JavaScript，提供文件上传下载界面（upload.html、listfile.jsp）。

## 4.2 核心功能

文件列表与目录导航：

显示服务器指定 uploads 目录下当前路径的文件和子目录列表。

支持点击子目录名称进入下一级目录，并通过“返回上一级”链接回到父目录。

动态调整显示路径，提供清晰的用户界面。

文件上传：

通过 Web 表单接收用户选择的文件。

将文件安全地保存到服务器上的当前浏览目录中。

支持文件类型过滤 (ALLOWED\_EXTENSIONS) 和文件大小限制 (MAX\_CONTENT\_LENGTH)。

文件下载：

为列表中的文件提供明确的“下载”按钮。

点击下载按钮时，强制浏览器将文件下载到本地，而不是直接在浏览器中打开。

目录创建：

允许用户在当前目录下创建新的空文件夹。

对输入的文件夹名称进行安全处理，防止非法字符。

文件/目录删除：

为文件和空目录提供删除功能。

在删除前有 JavaScript 确认提示，增强用户体验和安全性。

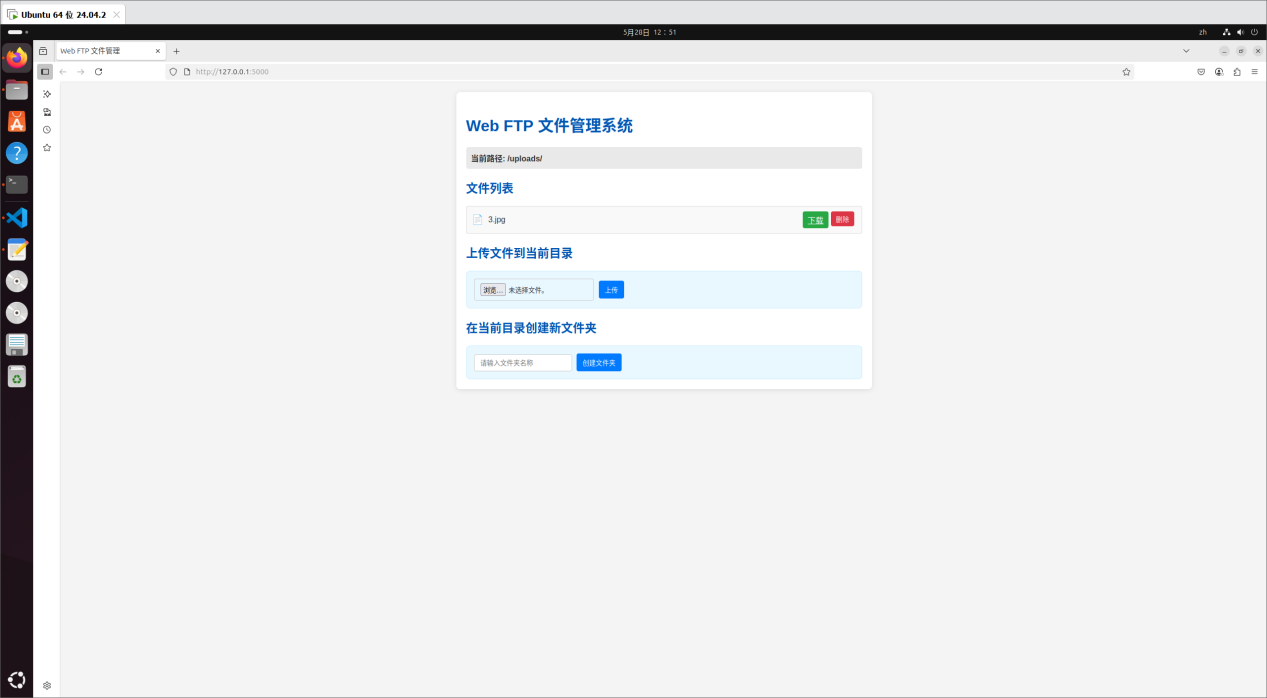
安全防护：

secure\_filename：清洗文件名，防止恶意文件名导致的问题。

safe\_path\_join：严格检查所有文件操作的路径，确保操作始终限制在 UPLOAD\_FOLDER 内部，有效防止了路径遍历攻击（如 ../ 访问服务器其他文件）。

用户反馈：

通过 Flask 的 flash 消息机制，在页面顶部显示操作成功、失败或警告等提示信息。

4.3 界面展示  


# 5 爬虫部分实现

## 5.1 技术架构

核心网络通信： Python socket 模块直接进行 TCP 连接，发送和接收原始 HTTP 请求和响应数据。

HTTP 协议实现： 手动构建 HTTP GET 请求头，并解析服务器返回的原始 HTTP 响应（包括状态行、响应头和响应体）。

HTML 解析： BeautifulSoup4 库用于解析 HTML 响应体，方便地提取网页中的文本内容和指定标签（如 <img>）的属性。

XML 解析器： lxml 是 BeautifulSoup4 默认推荐和使用的解析器之一，提供高性能的解析能力。

文件系统操作： Python 的 os 模块和内置的文件操作（open, write）用于将抓取到的数据（文本、图片）保存到本地文件。

URL 处理： urllib.parse 模块（虽然代码中可能没直接显示但通常会用到）用于解析和构建 URL。

## 5.2核心功能

底层 HTTP GET 请求：

通过 Socket 连接到目标 Web 服务器的 80 端口（HTTP 默认端口）。

手动构造并发送符合 HTTP/1.0 或 HTTP/1.1 规范的 GET 请求字符串。

原始 HTTP 响应接收与解析：

通过 Socket 接收服务器返回的原始字节流。

手动解析响应头和响应体，分离 HTTP 头部和实际内容。

处理可能的编码问题（如 UTF-8）。

网页文本内容提取：

使用 BeautifulSoup 解析 HTML 响应体，提取可见的纯文本内容。

图片 URL 提取与下载：

在解析后的 HTML 中查找所有 <img> 标签。

提取图片的 src 属性（URL）。

如果图片 URL 是 HTTP 协议，则再次通过 Socket 发送 GET 请求下载图片数据。

将下载的图片保存到本地指定目录。

本地数据存储：

将抓取到的网页文本内容保存为 .txt 文件。

将下载的图片保存为原始格式的文件。

错误处理：

处理网络连接失败、HTTP 状态码异常等情况。

（根据实现细节）可能包含对重定向、相对路径 URL 的处理。

# 6 思考题解答

## 6.1 为何 FTP 使用两个 TCP 连接？

控制与数据分离：控制通道传输命令和响应（如用户认证、文件操作指令），数据通道专注于文件传输，提高协议灵活性和效率。

并发处理：可在不中断控制连接的情况下，并行处理多个数据传输任务。

## 6.2 主动模式与被动模式的区别？

主动模式：服务器主动连接客户端数据端口（默认 20 端口），适用于服务器位于公网、客户端在内网的场景，但可能被防火墙拦截。

被动模式：客户端主动连接服务器临时开放的数据端口，适用于客户端位于 NAT 或防火墙后的场景，兼容性更强。

## 6.3 下载大量小文件速度慢的原因及改进？

原因：每个文件传输需新建 / 关闭数据连接，TCP 三次握手和四次挥手带来额外开销。

改进方法：

使用MLSD/MLST等批量目录查询命令减少交互次数；

启用 TCP\_NODELAY 选项禁用 Nagle 算法，减少小包延迟；

采用二进制传输模式（bin命令）替代 ASCII 模式，避免字符转换开销。

# 7 实验结论

通过本次实验，深入掌握了 FTP 协议的底层原理与 Socket 编程实现，理解了网络应用中协议分层和连接管理的设计思想。同时，通过 Web 文件系统和网络爬虫的开发，熟悉了前后端分离架构和 HTTP 协议的实际应用。实验中遇到的防火墙配置、端口冲突等问题，通过查阅 RFC 文档和调试 Socket 状态得以解决，进一步提升了问题分析与解决能力。

# 8 附录

## 8.1 GitHub 项目链接

[https://github.com/FishCoderovo/2025JW\_Network\_Projects](https://github.com/FishCoderovo/2025JW_Network_Projects" \t "https://www.doubao.com/chat/_blank)

ftp/目录：FTP 客户端与服务器代码；

web/目录：Web 文件传输系统代码；

crawler/目录：网络爬虫代码。

## 8.2 client.cpp

FTP 客户端实现代码

#include <iostream>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string>

#include <cstring>

#include <fstream>

#include <sstream>

#include <vector>

#include <algorithm> // For std::replace

#include <random> // For random port generation (for active mode)

#include <chrono> // For high-resolution clock (for random seed)

#include <netdb.h> // For gethostname, gethostbyname

#define SERVER\_IP "127.0.0.1" // Default server IP address (localhost)

#define CONTROL\_PORT 2121 // FTP控制连接端口

// Global state for data connection

bool passive\_mode\_enabled = true; // Default to passive mode

int active\_data\_listener\_sock = -1; // For active mode: client's listening socket for data

// Function to send a command and receive a response

std::string send\_command(int sock, const std::string& command) {

std::string full\_command = command + "\r\n";

send(sock, full\_command.c\_str(), full\_command.length(), 0);

char buffer[4096] = {0}; // Increased buffer size for potentially longer responses

memset(buffer, 0, sizeof(buffer));

int valread = recv(sock, buffer, sizeof(buffer) - 1, 0); // -1 to ensure null termination

if (valread <= 0) {

std::cerr << "Server disconnected or error during recv." << std::endl;

return ""; // Indicate error

}

return std::string(buffer);

}

// Function to open a data connection (for passive mode client: client connects to server)

int open\_passive\_data\_connection\_client(const std::string& ip, int port) {

int data\_sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (data\_sock == -1) {

std::cerr << "Data socket creation failed." << std::endl;

return -1;

}

sockaddr\_in data\_addr;

data\_addr.sin\_family = AF\_INET;

data\_addr.sin\_port = htons(port);

if (inet\_pton(AF\_INET, ip.c\_str(), &data\_addr.sin\_addr) <= 0) {

std::cerr << "Invalid IP address for data connection." << std::endl;

close(data\_sock);

return -1;

}

if (connect(data\_sock, (struct sockaddr \*)&data\_addr, sizeof(data\_addr)) < 0) {

perror("data connect failed");

close(data\_sock);

return -1;

}

return data\_sock;

}

// Function to open a data listener for active mode client (client listens, server connects to client)

int open\_active\_data\_listener\_client(std::string& client\_ip, int& client\_port) {

int data\_listener\_sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (data\_listener\_sock == -1) {

std::cerr << "Active data listener socket creation failed." << std::endl;

return -1;

}

// Allow reuse of address and port

int opt = 1;

if (setsockopt(data\_listener\_sock, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT, &opt, sizeof(opt)) < 0) {

perror("setsockopt active listener failed");

close(data\_listener\_sock);

return -1;

}

sockaddr\_in data\_addr;

data\_addr.sin\_family = AF\_INET;

data\_addr.sin\_addr.s\_addr = INADDR\_ANY; // Listen on any available interface

data\_addr.sin\_port = 0; // Let the system assign an ephemeral port

if (bind(data\_listener\_sock, (struct sockaddr \*)&data\_addr, sizeof(data\_addr)) < 0) {

perror("active bind failed");

close(data\_listener\_sock);

return -1;

}

socklen\_t addr\_len = sizeof(data\_addr);

getsockname(data\_listener\_sock, (struct sockaddr\*)&data\_addr, &addr\_len);

client\_port = ntohs(data\_addr.sin\_port);

if (listen(data\_listener\_sock, 1) < 0) { // Only allow one pending connection from server

perror("active listen failed");

close(data\_listener\_sock);

return -1;

}

// Get client's IP address (important for NAT scenarios)

// For local testing, gethostname and gethostbyname usually give 127.0.0.1 or local LAN IP.

// In a real-world scenario with NAT, this would require UPnP or a specific external IP configuration.

char hostbuffer[256];

gethostname(hostbuffer, sizeof(hostbuffer));

struct hostent \*host\_entry;

host\_entry = gethostbyname(hostbuffer);

if (host\_entry == NULL) {

perror("gethostbyname failed");

close(data\_listener\_sock);

return -1;

}

client\_ip = inet\_ntoa(\*((struct in\_addr\*)host\_entry->h\_addr\_list[0]));

return data\_listener\_sock;

}

// Function to handle LIST/DIR command

void handle\_list\_command(int sock) {

std::string response;

int data\_sock = -1;

if (passive\_mode\_enabled) {

response = send\_command(sock, "PASV");

std::cout << "Server: " << response;

if (response.rfind("227", 0) == 0) {

size\_t start = response.find('(');

size\_t end = response.find(')');

if (start != std::string::npos && end != std::string::npos) {

std::string params\_str = response.substr(start + 1, end - start - 1);

std::vector<int> parts;

std::stringstream ss(params\_str);

std::string segment;

while(std::getline(ss, segment, ',')) {

parts.push\_back(std::stoi(segment));

}

if (parts.size() == 6) {

std::string data\_ip = std::to\_string(parts[0]) + "." + std::to\_string(parts[1]) + "." + std::to\_string(parts[2]) + "." + std::to\_string(parts[3]);

int data\_port = parts[4] \* 256 + parts[5];

data\_sock = open\_passive\_data\_connection\_client(data\_ip, data\_port);

} else {

std::cerr << "Failed to parse PASV response.\n";

}

} else {

std::cerr << "Invalid PASV response format.\n";

}

} else {

std::cerr << "Server denied PASV mode or sent invalid response.\n";

}

} else { // Active mode

std::string client\_ip;

int client\_data\_port = 0;

active\_data\_listener\_sock = open\_active\_data\_listener\_client(client\_ip, client\_data\_port);

if (active\_data\_listener\_sock != -1) {

std::string ip\_str = client\_ip;

std::replace(ip\_str.begin(), ip\_str.end(), '.', ',');

int p1 = client\_data\_port / 256;

int p2 = client\_data\_port % 256;

std::string port\_command = "PORT " + ip\_str + "," + std::to\_string(p1) + "," + std::to\_string(p2);

response = send\_command(sock, port\_command);

std::cout << "Server: " << response;

if (response.rfind("200", 0) != 0) {

std::cerr << "Server denied PORT command or sent invalid response.\n";

close(active\_data\_listener\_sock);

active\_data\_listener\_sock = -1;

return;

}

// Now server will connect to client's active\_data\_listener\_sock

} else {

std::cerr << "Failed to set up active data listener.\n";

return;

}

}

if (data\_sock != -1 || active\_data\_listener\_sock != -1) {

response = send\_command(sock, "LIST"); // Send LIST command over control connection

std::cout << "Server: " << response; // Expecting 150 response

if (response.rfind("150", 0) == 0) {

if (!passive\_mode\_enabled) { // In active mode, client needs to accept the connection

data\_sock = accept(active\_data\_listener\_sock, NULL, NULL);

close(active\_data\_listener\_sock); // Close listener after accepting

active\_data\_listener\_sock = -1;

if (data\_sock < 0) {

perror("accept active data connection failed");

std::cerr << "Failed to accept data connection from server.\n";

return;

}

}

char data\_buffer[4096];

memset(data\_buffer, 0, sizeof(data\_buffer));

int bytes\_read = recv(data\_sock, data\_buffer, sizeof(data\_buffer) - 1, 0);

if (bytes\_read > 0) {

std::cout << "Directory Listing:\n" << data\_buffer << std::endl;

} else if (bytes\_read == 0) {

std::cout << "Server sent empty directory listing.\n";

} else {

perror("recv data failed");

std::cerr << "Error receiving directory listing.\n";

}

close(data\_sock);

// Get final response from server (e.g., 226 Directory send OK)

response = send\_command(sock, ""); // Send dummy to prompt final response

std::cout << "Server: " << response;

} else {

std::cerr << "Server denied LIST command or data connection setup failed.\n";

}

} else {

std::cerr << "Failed to establish data connection.\n";

}

}

// Function to handle GET command (download)

void handle\_get\_command(int sock, const std::string& filename) {

std::string response;

int data\_sock = -1;

if (passive\_mode\_enabled) {

response = send\_command(sock, "PASV");

std::cout << "Server: " << response;

if (response.rfind("227", 0) == 0) {

size\_t start = response.find('(');

size\_t end = response.find(')');

if (start != std::string::npos && end != std::string::npos) {

std::string params\_str = response.substr(start + 1, end - start - 1);

std::vector<int> parts;

std::stringstream ss(params\_str);

std::string segment;

while(std::getline(ss, segment, ',')) {

parts.push\_back(std::stoi(segment));

}

if (parts.size() == 6) {

std::string data\_ip = std::to\_string(parts[0]) + "." + std::to\_string(parts[1]) + "." + std::to\_string(parts[2]) + "." + std::to\_string(parts[3]);

int data\_port = parts[4] \* 256 + parts[5];

data\_sock = open\_passive\_data\_connection\_client(data\_ip, data\_port);

}

}

}

} else { // Active mode

std::string client\_ip;

int client\_data\_port = 0;

active\_data\_listener\_sock = open\_active\_data\_listener\_client(client\_ip, client\_data\_port);

if (active\_data\_listener\_sock != -1) {

std::string ip\_str = client\_ip;

std::replace(ip\_str.begin(), ip\_str.end(), '.', ',');

int p1 = client\_data\_port / 256;

int p2 = client\_data\_port % 256;

std::string port\_command = "PORT " + ip\_str + "," + std::to\_string(p1) + "," + std::to\_string(p2);

response = send\_command(sock, port\_command);

std::cout << "Server: " << response;

if (response.rfind("200", 0) != 0) {

close(active\_data\_listener\_sock);

active\_data\_listener\_sock = -1;

std::cerr << "Server denied PORT command.\n";

return;

}

} else {

std::cerr << "Failed to set up active data listener.\n";

return;

}

}

if (data\_sock != -1 || active\_data\_listener\_sock != -1) {

response = send\_command(sock, "RETR " + filename); // Send RETR command

std::cout << "Server: " << response; // Expecting 150 response

if (response.rfind("150", 0) == 0) {

if (!passive\_mode\_enabled) { // In active mode, client needs to accept the connection

data\_sock = accept(active\_data\_listener\_sock, NULL, NULL);

close(active\_data\_listener\_sock);

active\_data\_listener\_sock = -1;

if (data\_sock < 0) {

perror("accept active data connection failed");

std::cerr << "Failed to accept data connection from server.\n";

return;

}

}

std::ofstream outfile(filename, std::ios::binary);

if (outfile.is\_open()) {

char data\_buffer[4096];

int bytes\_read;

while ((bytes\_read = recv(data\_sock, data\_buffer, sizeof(data\_buffer), 0)) > 0) {

outfile.write(data\_buffer, bytes\_read);

}

if (bytes\_read < 0) {

perror("recv file data failed");

std::cerr << "Error receiving file data.\n";

} else {

std::cout << "File " << filename << " downloaded successfully.\n";

}

outfile.close();

} else {

std::cerr << "Failed to open local file for writing: " << filename << std::endl;

}

close(data\_sock);

// Get final response from server (e.g., 226 Transfer complete)

response = send\_command(sock, ""); // Dummy command to get final response

std::cout << "Server: " << response;

} else {

std::cerr << "Server denied RETR command or data connection setup failed.\n";

}

} else {

std::cerr << "Failed to establish data connection.\n";

}

}

// Function to handle PUT command (upload)

void handle\_put\_command(int sock, const std::string& filename) {

std::ifstream infile(filename, std::ios::binary);

if (!infile.is\_open()) {

std::cerr << "Failed to open local file for reading: " << filename << std::endl;

return;

}

std::string response;

int data\_sock = -1;

if (passive\_mode\_enabled) {

response = send\_command(sock, "PASV");

std::cout << "Server: " << response;

if (response.rfind("227", 0) == 0) {

size\_t start = response.find('(');

size\_t end = response.find(')');

if (start != std::string::npos && end != std::string::npos) {

std::string params\_str = response.substr(start + 1, end - start - 1);

std::vector<int> parts;

std::stringstream ss(params\_str);

std::string segment;

while(std::getline(ss, segment, ',')) {

parts.push\_back(std::stoi(segment));

}

if (parts.size() == 6) {

std::string data\_ip = std::to\_string(parts[0]) + "." + std::to\_string(parts[1]) + "." + std::to\_string(parts[2]) + "." + std::to\_string(parts[3]);

int data\_port = parts[4] \* 256 + parts[5];

data\_sock = open\_passive\_data\_connection\_client(data\_ip, data\_port);

}

}

}

} else { // Active mode

std::string client\_ip;

int client\_data\_port = 0;

active\_data\_listener\_sock = open\_active\_data\_listener\_client(client\_ip, client\_data\_port);

if (active\_data\_listener\_sock != -1) {

std::string ip\_str = client\_ip;

std::replace(ip\_str.begin(), ip\_str.end(), '.', ',');

int p1 = client\_data\_port / 256;

int p2 = client\_data\_port % 256;

std::string port\_command = "PORT " + ip\_str + "," + std::to\_string(p1) + "," + std::to\_string(p2);

response = send\_command(sock, port\_command);

std::cout << "Server: " << response;

if (response.rfind("200", 0) != 0) {

close(active\_data\_listener\_sock);

active\_data\_listener\_sock = -1;

std::cerr << "Server denied PORT command.\n";

return;

}

} else {

std::cerr << "Failed to set up active data listener.\n";

return;

}

}

if (data\_sock != -1 || active\_data\_listener\_sock != -1) {

response = send\_command(sock, "STOR " + filename); // Send STOR command

std::cout << "Server: " << response; // Expecting 150 response

if (response.rfind("150", 0) == 0) {

if (!passive\_mode\_enabled) { // In active mode, client needs to accept the connection

data\_sock = accept(active\_data\_listener\_sock, NULL, NULL);

close(active\_data\_listener\_sock);

active\_data\_listener\_sock = -1;

if (data\_sock < 0) {

perror("accept active data connection failed");

std::cerr << "Failed to accept data connection from server.\n";

return;

}

}

char data\_buffer[4096];

while (!infile.eof()) {

infile.read(data\_buffer, sizeof(data\_buffer));

ssize\_t bytes\_sent = send(data\_sock, data\_buffer, infile.gcount(), 0);

if (bytes\_sent < 0) {

perror("send file data failed");

std::cerr << "Error sending file data.\n";

break;

}

}

infile.close();

close(data\_sock);

std::cout << "File " << filename << " uploaded successfully.\n";

// Get final response from server (e.g., 226 Transfer complete)

response = send\_command(sock, ""); // Dummy command to get final response

std::cout << "Server: " << response;

} else {

std::cerr << "Server denied STOR command or data connection setup failed.\n";

}

} else {

std::cerr << "Failed to establish data connection.\n";

}

}

int main() {

int sock = 0;

struct sockaddr\_in serv\_addr;

char buffer[1024] = {0};

if ((sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0) {

std::cout << "\n Socket creation error \n";

return -1;

}

serv\_addr.sin\_family = AF\_INET;

serv\_addr.sin\_port = htons(CONTROL\_PORT);

// Convert IP address from text to binary form

if (inet\_pton(AF\_INET, SERVER\_IP, &serv\_addr.sin\_addr) <= 0) {

std::cout << "\nInvalid address/ Address not supported \n";

return -1;

}

if (connect(sock, (struct sockaddr \*)&serv\_addr, sizeof(serv\_addr)) < 0) {

std::cout << "\nConnection Failed \n";

return -1;

}

std::cout << "Connected to FTP server at " << SERVER\_IP << ":" << CONTROL\_PORT << std::endl;

// Receive welcome message

memset(buffer, 0, sizeof(buffer));

recv(sock, buffer, 1024, 0);

std::cout << "Server: " << buffer;

std::string username, password;

std::string response;

// Login process

while (true) {

std::cout << "Username: ";

std::getline(std::cin, username);

response = send\_command(sock, "USER " + username);

std::cout << "Server: " << response;

if (response.rfind("331", 0) != 0) { // Expected "331 User name okay"

std::cerr << "Unexpected server response during USER command." << std::endl;

close(sock);

return -1;

}

std::cout << "Password: ";

std::getline(std::cin, password);

response = send\_command(sock, "PASS " + password);

std::cout << "Server: " << response;

if (response.rfind("230", 0) == 0) { // Expected "230 User logged in"

std::cout << "Login successful." << std::endl;

break;

} else if (response.rfind("530", 0) == 0) { // "530 Not logged in"

std::cout << "Login failed. Please try again." << std::endl;

} else {

std::cerr << "Unexpected server response during PASS command." << std::endl;

close(sock);

return -1;

}

}

std::string command\_line;

while (true) {

std::cout << "ftp> ";

std::getline(std::cin, command\_line);

std::string command\_name;

std::string argument;

size\_t first\_space = command\_line.find(' ');

if (first\_space != std::string::npos) {

command\_name = command\_line.substr(0, first\_space);

argument = command\_line.substr(first\_space + 1);

} else {

command\_name = command\_line;

}

// Convert command name to uppercase for case-insensitive comparison

for (char &c : command\_name) {

c = toupper(c);

}

if (command\_name == "QUIT") {

response = send\_command(sock, "QUIT");

std::cout << "Server: " << response;

break;

} else if (command\_name == "PASV") {

passive\_mode\_enabled = true;

std::cout << "Passive mode enabled. All data transfers will use PASV.\n";

// The actual PASV command to the server will be sent by LIST/GET/PUT handlers

} else if (command\_name == "PORT") {

passive\_mode\_enabled = false;

std::cout << "Active mode enabled. All data transfers will use PORT.\n";

// The actual PORT command to the server will be sent by LIST/GET/PUT handlers

} else if (command\_name == "LS" || command\_name == "DIR") {

handle\_list\_command(sock);

} else if (command\_name == "GET") {

handle\_get\_command(sock, argument);

} else if (command\_name == "PUT") {

handle\_put\_command(sock, argument);

} else if (command\_name == "?" || command\_name == "HELP") {

std::cout << "Available commands:\n";

std::cout << " user <username> - Specify username (used during login)\n";

std::cout << " pass <password> - Specify password (used during login)\n";

std::cout << " ls / dir - List directory contents (uses current mode)\n";

std::cout << " pwd - Print working directory\n";

std::cout << " cd <directory> - Change directory\n";

std::cout << " get <filename> - Download a file (uses current mode)\n";

std::cout << " put <filename> - Upload a file (uses current mode)\n";

std::cout << " pasv - Set data transfer mode to passive (default)\n";

std::cout << " port - Set data transfer mode to active\n";

std::cout << " quit - Exit FTP client\n";

std::cout << " ? / help - Display this help message\n";

}

else {

response = send\_command(sock, command\_line);

std::cout << "Server: " << response;

}

}

close(sock);

return 0;

}

## 8.3 server.cpp

#include <iostream>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#include <string>

#include <cstring>

#include <pthread.h>

#include <map>

#include <fstream>

#include <filesystem>

#include <sstream>

#include <vector>

#include <algorithm> // For std::replace

#include <netdb.h> // For gethostname, gethostbyname

#define CONTROL\_PORT 2121 // FTP 控制连接端口

namespace fs = std::filesystem; // For C++17 filesystem operations

// Simple user database

std::map<std::string, std::string> users = {

{"user1", "pass1"},

{"admin", "adminpass"},

{"root","12345"}

};

// Structure to hold client-specific state

struct ClientState {

int control\_socket;

std::string current\_dir;

bool logged\_in;

std::string username;

// Data connection related

int data\_socket\_fd; // For active mode: client's accepted data socket; for passive mode: server's listening data socket

std::string data\_ip; // For active mode: client's IP to connect to

int data\_port; // For active mode: client's port to connect to

bool passive\_mode; // true for PASV (server listens), false for PORT (server connects to client)

ClientState() : control\_socket(-1), current\_dir(""), logged\_in(false), username(""),

data\_socket\_fd(-1), data\_ip(""), data\_port(0), passive\_mode(false) {}

};

// Map to store client states, protected by a mutex

std::map<int, ClientState> client\_states;

pthread\_mutex\_t client\_states\_mutex = PTHREAD\_MUTEX\_INITIALIZER; // Mutex for client\_states map

// Function to send a response to the client

void send\_response(int client\_socket, const std::string& response) {

send(client\_socket, response.c\_str(), response.length(), 0);

}

// Function to open a data connection for active mode (server connects to client)

int open\_active\_data\_connection(int control\_socket, const std::string& client\_ip, int client\_port) {

int data\_sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (data\_sock == -1) {

perror("data socket creation failed");

send\_response(control\_socket, "425 Can't open data connection.\r\n");

return -1;

}

sockaddr\_in data\_addr;

data\_addr.sin\_family = AF\_INET;

data\_addr.sin\_port = htons(client\_port);

if (inet\_pton(AF\_INET, client\_ip.c\_str(), &data\_addr.sin\_addr) <= 0) {

send\_response(control\_socket, "425 Invalid IP address for data connection.\r\n");

close(data\_sock);

return -1;

}

if (connect(data\_sock, (struct sockaddr \*)&data\_addr, sizeof(data\_addr)) < 0) {

perror("data connect failed");

send\_response(control\_socket, "425 Can't open data connection.\r\n");

close(data\_sock);

return -1;

}

return data\_sock;

}

// Function to set up a listener for passive mode (server listens, client connects to server)

int open\_passive\_data\_listener(int control\_socket, std::string& server\_ip, int& data\_port) {

int data\_listener\_sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (data\_listener\_sock == -1) {

send\_response(control\_socket, "421 Service not available, can't open data listener.\r\n");

return -1;

}

// Allow reuse of address and port

int opt = 1;

if (setsockopt(data\_listener\_sock, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT, &opt, sizeof(opt)) < 0) {

perror("setsockopt passive listener failed");

close(data\_listener\_sock);

send\_response(control\_socket, "421 Service not available, setsockopt failed.\r\n");

return -1;

}

sockaddr\_in data\_addr;

data\_addr.sin\_family = AF\_INET;

data\_addr.sin\_addr.s\_addr = INADDR\_ANY;

data\_addr.sin\_port = 0; // Let the system assign an ephemeral port

if (bind(data\_listener\_sock, (struct sockaddr \*)&data\_addr, sizeof(data\_addr)) < 0) {

perror("passive bind failed");

send\_response(control\_socket, "421 Service not available, bind failed.\r\n");

close(data\_listener\_sock);

return -1;

}

socklen\_t addr\_len = sizeof(data\_addr);

getsockname(data\_listener\_sock, (struct sockaddr\*)&data\_addr, &addr\_len);

data\_port = ntohs(data\_addr.sin\_port);

if (listen(data\_listener\_sock, 1) < 0) { // Only allow one pending connection for passive mode

perror("passive listen failed");

send\_response(control\_socket, "421 Service not available, listen failed.\r\n");

close(data\_listener\_sock);

return -1;

}

// Get server's IP address (important for NAT scenarios)

char hostbuffer[256];

gethostname(hostbuffer, sizeof(hostbuffer));

struct hostent \*host\_entry;

host\_entry = gethostbyname(hostbuffer);

if (host\_entry == NULL) {

perror("gethostbyname failed");

send\_response(control\_socket, "421 Service not available, cannot determine server IP.\r\n");

close(data\_listener\_sock);

return -1;

}

server\_ip = inet\_ntoa(\*((struct in\_addr\*)host\_entry->h\_addr\_list[0]));

// Format the PASV response: 227 Entering Passive Mode (h1,h2,h3,h4,p1,p2)

// IP: h1.h2.h3.h4, Port: p1\*256 + p2

std::string ip\_str = server\_ip;

std::replace(ip\_str.begin(), ip\_str.end(), '.', ',');

int p1 = data\_port / 256;

int p2 = data\_port % 256;

std::string pasv\_response = "227 Entering Passive Mode (" + ip\_str + "," + std::to\_string(p1) + "," + std::to\_string(p2) + ").\r\n";

send\_response(control\_socket, pasv\_response);

return data\_listener\_sock;

}

// Function to accept a data connection for passive mode

int accept\_passive\_data\_connection(int data\_listener\_sock) {

sockaddr\_in client\_data\_addr;

socklen\_t client\_data\_addrlen = sizeof(client\_data\_addr);

int new\_data\_sock = accept(data\_listener\_sock, (struct sockaddr\*)&client\_data\_addr, &client\_data\_addrlen);

if (new\_data\_sock < 0) {

perror("accept passive data connection failed");

return -1;

}

return new\_data\_sock;

}

// Handle LIST command using data connection

void handle\_list\_data(ClientState& client\_state) {

int data\_sock = -1;

if (client\_state.passive\_mode) {

send\_response(client\_state.control\_socket, "150 Opening ASCII mode data connection for file list.\r\n");

data\_sock = accept\_passive\_data\_connection(client\_state.data\_socket\_fd);

close(client\_state.data\_socket\_fd); // Close the listener after accepting

client\_state.data\_socket\_fd = -1; // Reset to indicate listener is closed

} else { // Active mode

send\_response(client\_state.control\_socket, "150 Opening ASCII mode data connection for file list.\r\n");

data\_sock = open\_active\_data\_connection(client\_state.control\_socket, client\_state.data\_ip, client\_state.data\_port);

}

if (data\_sock == -1) {

// Error response already sent by open\_active\_data\_connection or accept\_passive\_data\_connection

return;

}

std::string file\_list;

try {

for (const auto& entry : fs::directory\_iterator(client\_state.current\_dir)) {

file\_list += entry.path().filename().string() + "\r\n";

}

send(data\_sock, file\_list.c\_str(), file\_list.length(), 0);

send\_response(client\_state.control\_socket, "226 Directory send OK.\r\n");

} catch (const fs::filesystem\_error& e) {

send\_response(client\_state.control\_socket, "550 Failed to list directory.\r\n");

}

close(data\_sock);

}

// Handle RETR command using data connection

void handle\_retr\_data(ClientState& client\_state, const std::string& filename) {

int data\_sock = -1;

if (client\_state.passive\_mode) {

send\_response(client\_state.control\_socket, "150 Opening BINARY mode data connection for " + filename + ".\r\n");

data\_sock = accept\_passive\_data\_connection(client\_state.data\_socket\_fd);

close(client\_state.data\_socket\_fd);

client\_state.data\_socket\_fd = -1;

} else {

send\_response(client\_state.control\_socket, "150 Opening BINARY mode data connection for " + filename + ".\r\n");

data\_sock = open\_active\_data\_connection(client\_state.control\_socket, client\_state.data\_ip, client\_state.data\_port);

}

if (data\_sock == -1) {

return;

}

std::string full\_path = client\_state.current\_dir + "/" + filename;

std::ifstream file(full\_path, std::ios::binary);

if (file.is\_open()) {

char buffer[4096];

while (!file.eof()) {

file.read(buffer, sizeof(buffer));

ssize\_t bytes\_sent = send(data\_sock, buffer, file.gcount(), 0);

if (bytes\_sent < 0) {

perror("send file data failed");

send\_response(client\_state.control\_socket, "426 Connection closed; transfer aborted.\r\n");

break;

}

}

file.close();

send\_response(client\_state.control\_socket, "226 Transfer complete.\r\n");

} else {

send\_response(client\_state.control\_socket, "550 Failed to open file.\r\n");

}

close(data\_sock);

}

// Handle STOR command using data connection

void handle\_stor\_data(ClientState& client\_state, const std::string& filename) {

int data\_sock = -1;

if (client\_state.passive\_mode) {

send\_response(client\_state.control\_socket, "150 Ok to send data.\r\n");

data\_sock = accept\_passive\_data\_connection(client\_state.data\_socket\_fd);

close(client\_state.data\_socket\_fd);

client\_state.data\_socket\_fd = -1;

} else {

send\_response(client\_state.control\_socket, "150 Ok to send data.\r\n");

data\_sock = open\_active\_data\_connection(client\_state.control\_socket, client\_state.data\_ip, client\_state.data\_port);

}

if (data\_sock == -1) {

return;

}

std::string full\_path = client\_state.current\_dir + "/" + filename;

std::ofstream file(full\_path, std::ios::binary);

if (file.is\_open()) {

char buffer[4096];

int bytes\_received;

while ((bytes\_received = recv(data\_sock, buffer, sizeof(buffer), 0)) > 0) {

file.write(buffer, bytes\_received);

}

if (bytes\_received < 0) {

perror("recv file data failed");

send\_response(client\_state.control\_socket, "426 Connection closed; transfer aborted.\r\n");

} else {

send\_response(client\_state.control\_socket, "226 Transfer complete.\r\n");

}

file.close();

} else {

send\_response(client\_state.control\_socket, "550 Failed to create file.\r\n");

}

close(data\_sock);

}

// Handle PWD command

void handle\_pwd(int client\_socket, const std::string& current\_dir) {

std::string response = "257 \"" + current\_dir + "\" is current directory.\r\n";

send\_response(client\_socket, response);

}

// Handle CWD command

void handle\_cwd(int client\_socket, std::string& current\_dir, const std::string& path) {

fs::path new\_path = fs::path(current\_dir) / path; // Resolve relative path first

// Attempt to canonicalize the path, handling ".." and symlinks

try {

fs::path canonical\_path = fs::canonical(new\_path);

if (fs::is\_directory(canonical\_path)) {

current\_dir = canonical\_path.string();

send\_response(client\_socket, "250 Directory successfully changed.\r\n");

} else {

send\_response(client\_socket, "550 Not a directory.\r\n");

}

} catch (const fs::filesystem\_error& e) {

send\_response(client\_socket, "550 Failed to change directory: " + std::string(e.what()) + "\r\n");

}

}

// Thread function to handle a single client connection

void \*client\_handler\_thread(void \*socket\_desc) {

int client\_socket = \*(int\*)socket\_desc;

delete (int\*)socket\_desc; // Free the dynamically allocated memory

// Initialize client state

pthread\_mutex\_lock(&client\_states\_mutex);

client\_states[client\_socket] = ClientState(); // Use default constructor for initialization

client\_states[client\_socket].control\_socket = client\_socket;

client\_states[client\_socket].current\_dir = fs::current\_path().string(); // Set initial directory

ClientState& client\_state = client\_states[client\_socket]; // Get a reference to the client's state

pthread\_mutex\_unlock(&client\_states\_mutex);

char buffer[1024] = {0};

send\_response(client\_socket, "220 Welcome to Simple FTP Server.\r\n");

while (true) {

memset(buffer, 0, sizeof(buffer));

int valread = recv(client\_socket, buffer, 1024, 0);

if (valread <= 0) {

std::cout << "Client " << client\_socket << " disconnected." << std::endl;

break;

}

std::string command\_line(buffer);

// Remove trailing \r\n

if (!command\_line.empty() && command\_line.back() == '\n') command\_line.pop\_back();

if (!command\_line.empty() && command\_line.back() == '\r') command\_line.pop\_back();

std::cout << "Client " << client\_socket << " Received: " << command\_line << std::endl;

std::string command\_name;

std::string argument;

size\_t first\_space = command\_line.find(' ');

if (first\_space != std::string::npos) {

command\_name = command\_line.substr(0, first\_space);

argument = command\_line.substr(first\_space + 1);

} else {

command\_name = command\_line;

}

// Convert command name to uppercase for case-insensitive comparison

for (char &c : command\_name) {

c = toupper(c);

}

if (command\_name == "USER") {

client\_state.username = argument;

send\_response(client\_socket, "331 User name okay, need password.\r\n");

} else if (command\_name == "PASS") {

if (users.count(client\_state.username) && users[client\_state.username] == argument) {

client\_state.logged\_in = true;

send\_response(client\_socket, "230 User logged in, proceed.\r\n");

} else {

client\_state.logged\_in = false;

send\_response(client\_socket, "530 Not logged in.\r\n");

}

} else if (command\_name == "QUIT") {

send\_response(client\_socket, "221 Goodbye.\r\n");

break;

} else if (!client\_state.logged\_in && !(command\_name == "USER" || command\_name == "PASS")) {

send\_response(client\_socket, "530 Not logged in.\r\n");

} else if (command\_name == "PORT") {

// Parse PORT command: PORT h1,h2,h3,h4,p1,p2

std::vector<int> parts;

std::stringstream ss(argument);

std::string segment;

while(std::getline(ss, segment, ',')) {

parts.push\_back(std::stoi(segment));

}

if (parts.size() == 6) {

client\_state.data\_ip = std::to\_string(parts[0]) + "." + std::to\_string(parts[1]) + "." + std::to\_string(parts[2]) + "." + std::to\_string(parts[3]);

client\_state.data\_port = parts[4] \* 256 + parts[5];

client\_state.passive\_mode = false; // Set to active mode

send\_response(client\_socket, "200 PORT command successful.\r\n");

} else {

send\_response(client\_socket, "501 Syntax error in parameters or arguments.\r\n");

}

} else if (command\_name == "PASV") {

// In PASV, server listens for a connection

std::string server\_ip;

int passive\_port = 0;

int data\_listener\_sock = open\_passive\_data\_listener(client\_socket, server\_ip, passive\_port);

if (data\_listener\_sock != -1) {

client\_state.data\_socket\_fd = data\_listener\_sock; // Store the listener socket

client\_state.passive\_mode = true; // Set to passive mode

}

// Response already sent by open\_passive\_data\_listener

}

// Data transfer commands now use data connection

else if (command\_name == "LIST" || command\_name == "NLST" || command\_name == "DIR") {

// Check if data connection details are set (either PORT or PASV was used)

if (client\_state.data\_socket\_fd != -1 || (!client\_state.passive\_mode && !client\_state.data\_ip.empty() && client\_state.data\_port != 0)) {

handle\_list\_data(client\_state);

} else {

send\_response(client\_socket, "425 Use PORT or PASV first.\r\n");

}

} else if (command\_name == "RETR") {

if (client\_state.data\_socket\_fd != -1 || (!client\_state.passive\_mode && !client\_state.data\_ip.empty() && client\_state.data\_port != 0)) {

handle\_retr\_data(client\_state, argument);

} else {

send\_response(client\_socket, "425 Use PORT or PASV first.\r\n");

}

} else if (command\_name == "STOR") {

if (client\_state.data\_socket\_fd != -1 || (!client\_state.passive\_mode && !client\_state.data\_ip.empty() && client\_state.data\_port != 0)) {

handle\_stor\_data(client\_state, argument);

} else {

send\_response(client\_socket, "425 Use PORT or PASV first.\r\n");

}

}

// Other non-data transfer commands

else if (command\_name == "PWD" || command\_name == "XPWD") {

handle\_pwd(client\_socket, client\_state.current\_dir);

} else if (command\_name == "CWD" || command\_name == "XCWD") {

handle\_cwd(client\_socket, client\_state.current\_dir, argument);

} else if (command\_name == "SYST") {

send\_response(client\_socket, "215 UNIX Type: L8\r\n"); // Standard response for SYST

} else if (command\_name == "TYPE") {

// TYPE A (ASCII) or TYPE I (Binary Image)

if (argument == "A" || argument == "I") {

send\_response(client\_socket, "200 Type set to " + argument + ".\r\n");

} else {

send\_response(client\_socket, "504 Command not implemented for that parameter.\r\n");

}

}

else {

send\_response(client\_socket, "500 Unknown command.\r\n");

}

}

// Clean up client state on disconnect

pthread\_mutex\_lock(&client\_states\_mutex);

if (client\_state.data\_socket\_fd != -1) {

close(client\_state.data\_socket\_fd);

}

client\_states.erase(client\_socket);

pthread\_mutex\_unlock(&client\_states\_mutex);

close(client\_socket);

pthread\_exit(NULL);

}

int main() {

int server\_fd, new\_socket;

struct sockaddr\_in address;

int opt = 1;

int addrlen = sizeof(address);

// Create socket file descriptor

if ((server\_fd = socket(AF\_INET, SOCK\_STREAM, 0)) == 0) {

perror("socket failed");

exit(EXIT\_FAILURE);

}

// Forcefully attach socket to the port, even if in TIME\_WAIT state

if (setsockopt(server\_fd, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT, &opt, sizeof(opt))) {

perror("setsockopt");

exit(EXIT\_FAILURE);

}

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = INADDR\_ANY; // Listen on all available interfaces

address.sin\_port = htons(CONTROL\_PORT);

// Bind the socket to the specified port

if (bind(server\_fd, (struct sockaddr \*)&address, sizeof(address)) < 0) {

perror("bind failed");

exit(EXIT\_FAILURE);

}

// Listen for incoming connections

if (listen(server\_fd, 5) < 0) { // Max 5 pending connections

perror("listen");

exit(EXIT\_FAILURE);

}

std::cout << "FTP Server listening on port " << CONTROL\_PORT << std::endl;

while (true) {

if ((new\_socket = accept(server\_fd, (struct sockaddr \*)&address, (socklen\_t\*)&addrlen)) < 0) {

perror("accept");

// Non-fatal error, continue listening for other connections

continue;

}

std::cout << "New connection accepted from " << inet\_ntoa(address.sin\_addr) << ":" << ntohs(address.sin\_port) << std::endl;

pthread\_t thread\_id;

int \*p\_new\_sock = new int; // Dynamically allocate memory for socket descriptor

\*p\_new\_sock = new\_socket;

if (pthread\_create(&thread\_id, NULL, client\_handler\_thread, (void\*)p\_new\_sock) < 0) {

perror("could not create thread");

close(new\_socket); // Close the socket if thread creation fails

delete p\_new\_sock; // Free allocated memory

continue;

}

pthread\_detach(thread\_id); // Detach the thread to clean up resources automatically

}

close(server\_fd);

return 0;

}

## 8.4 app.py

web部分主程序

import os

from flask import Flask, request, redirect, url\_for, render\_template, send\_from\_directory, flash

from werkzeug.utils import secure\_filename

from urllib.parse import unquote

app = Flask(\_\_name\_\_)

# 配置上传文件的总根目录，所有文件都将保存在这个目录下

# 确保这个目录存在，如果不存在Flask会报错

UPLOAD\_FOLDER = os.path.join(os.path.dirname(os.path.abspath(\_\_file\_\_)), 'uploads')

ALLOWED\_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif', 'zip', 'rar', '7z', 'doc', 'docx', 'xls', 'xlsx', 'ppt', 'pptx', 'mp4', 'mp3', 'json', 'xml', 'html', 'css', 'js', 'py'} # 增加更多常见文件类型

app.config['UPLOAD\_FOLDER'] = UPLOAD\_FOLDER

app.config['MAX\_CONTENT\_LENGTH'] = 16 \* 1024 \* 1024 # 限制文件大小为16MB (可根据需求调整)

app.secret\_key = 'your\_super\_secret\_key\_here' # 非常重要！用于flash消息，生产环境应使用更长、更安全的密钥

# 辅助函数：检查文件扩展名是否允许

def allowed\_file(filename):

return '.' in filename and \

filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

# 辅助函数：确保路径在允许的上传目录下且安全

def safe\_path\_join(base\_path, \*paths):

# 构建目标路径的绝对路径

target\_path = os.path.join(base\_path, \*paths)

abs\_base\_path = os.path.abspath(base\_path)

abs\_target\_path = os.path.abspath(target\_path)

# 检查目标路径是否以基准路径开头，防止跳出限定目录

# os.path.commonpath 返回两个或多个路径的共同最长路径前缀

# 如果共同前缀不是abs\_base\_path本身，说明abs\_target\_path在abs\_base\_path之外

common\_prefix = os.path.commonpath([abs\_base\_path, abs\_target\_path])

if common\_prefix != abs\_base\_path:

print(f"SECURITY ALERT: Attempted to access path outside UPLOAD\_FOLDER. Base: {abs\_base\_path}, Target: {abs\_target\_path}, Common: {common\_prefix}")

return None # 路径不安全

return abs\_target\_path

@app.route('/', defaults={'path': ''})

@app.route('/<path:path>')

def index(path):

# 解码URL路径，因为path参数可能包含URL编码（如空格被编码为%20）

current\_path = unquote(path).strip('/') # 移除路径开头的斜杠，防止与后续join冲突

full\_current\_path = safe\_path\_join(app.config['UPLOAD\_FOLDER'], current\_path)

if full\_current\_path is None or not os.path.exists(full\_current\_path):

flash('Requested path does not exist or is invalid.', 'error')

return redirect(url\_for('index')) # 重定向到根目录

if not os.path.isdir(full\_current\_path):

# 如果请求的路径不是一个目录（而是一个文件），则重定向到其父目录

flash('Requested path is a file, not a directory. Redirecting to parent.', 'info')

# os.path.dirname(current\_path) 获取当前路径的父目录部分

return redirect(url\_for('index', path=os.path.dirname(current\_path).strip('/')))

files = []

directories = []

try:

# 列出当前目录下的所有文件和目录

with os.scandir(full\_current\_path) as entries:

for entry in entries:

if entry.name.startswith('.'): # 忽略隐藏文件/目录

continue

if entry.is\_dir():

directories.append(entry.name)

else:

files.append(entry.name)

except Exception as e:

flash(f'Error listing directory: {e}', 'error')

return redirect(url\_for('index'))

# 对文件和目录进行排序

directories.sort()

files.sort()

# 构建返回上一级目录的路径

parent\_path = os.path.dirname(current\_path).strip('/') if current\_path else None

# 传递给模板的数据

return render\_template('index.html',

current\_path=current\_path,

files=files,

directories=directories,

parent\_path=parent\_path)

@app.route('/upload', methods=['POST'])

def upload\_file():

# 获取当前路径（从隐藏字段传递过来）

current\_path = request.form.get('current\_path', '').strip('/')

target\_dir = safe\_path\_join(app.config['UPLOAD\_FOLDER'], current\_path)

if target\_dir is None:

flash('Invalid upload path.', 'error')

return redirect(url\_for('index', path=current\_path))

# 确保目标上传目录存在，如果不存在则创建

if not os.path.exists(target\_dir):

try:

os.makedirs(target\_dir)

except OSError as e:

flash(f"Error creating directory {target\_dir}: {e}", 'error')

return redirect(url\_for('index', path=current\_path))

if 'file' not in request.files:

flash('No file part', 'error')

return redirect(url\_for('index', path=current\_path))

file = request.files['file']

if file.filename == '':

flash('No selected file', 'error')

return redirect(url\_for('index', path=current\_path))

if file and allowed\_file(file.filename):

filename = secure\_filename(file.filename)

# 检查文件名是否已存在，如果存在可以考虑重命名或提示用户

file\_path = os.path.join(target\_dir, filename)

if os.path.exists(file\_path):

flash(f'File "{filename}" already exists. Overwriting.', 'warning') # 提示用户

try:

file.save(file\_path)

flash(f'File "{filename}" uploaded successfully to "/uploads/{current\_path}"', 'success')

except Exception as e:

flash(f'Error saving file: {e}', 'error')

else:

flash('File type not allowed or no file selected.', 'error')

return redirect(url\_for('index', path=current\_path))

@app.route('/download/<path:filepath>')

def download\_file(filepath):

# 解码URL路径

decoded\_filepath = unquote(filepath)

# 确保文件路径在UPLOAD\_FOLDER内部

full\_file\_path = safe\_path\_join(app.config['UPLOAD\_FOLDER'], decoded\_filepath)

if full\_file\_path is None: # safe\_path\_join返回None表示路径不安全

flash('Invalid download path.', 'error')

return redirect(url\_for('index', path=os.path.dirname(decoded\_filepath).strip('/')))

if not os.path.exists(full\_file\_path):

flash('File not found.', 'error')

return redirect(url\_for('index', path=os.path.dirname(decoded\_filepath).strip('/')))

if os.path.isdir(full\_file\_path): # 防止下载目录

flash('Cannot download directories.', 'error')

return redirect(url\_for('index', path=decoded\_filepath.strip('/')))

# 从父目录发送文件

directory = os.path.dirname(full\_file\_path)

filename = os.path.basename(full\_file\_path)

try:

# as\_attachment=True 强制浏览器下载而不是显示

return send\_from\_directory(directory, filename, as\_attachment=True)

except Exception as e:

flash(f'Error downloading file: {e}', 'error')

return redirect(url\_for('index', path=os.path.dirname(decoded\_filepath).strip('/')))

@app.route('/create\_dir', methods=['POST'])

def create\_dir():

current\_path = request.form.get('current\_path', '').strip('/')

dir\_name = request.form.get('dir\_name')

if not dir\_name:

flash('Directory name cannot be empty.', 'error')

return redirect(url\_for('index', path=current\_path))

# 清理目录名，防止特殊字符或路径遍历

secure\_dir\_name = secure\_filename(dir\_name)

if not secure\_dir\_name:

flash('Invalid directory name after sanitization. Avoid special characters.', 'error')

return redirect(url\_for('index', path=current\_path))

full\_path\_to\_create = safe\_path\_join(app.config['UPLOAD\_FOLDER'], current\_path, secure\_dir\_name)

if full\_path\_to\_create is None:

flash('Invalid path for directory creation.', 'error')

return redirect(url\_for('index', path=current\_path))

try:

os.makedirs(full\_path\_to\_create, exist\_ok=True) # exist\_ok=True 避免目录已存在时报错

flash(f'Directory "{secure\_dir\_name}" created successfully in "/uploads/{current\_path}"', 'success')

except OSError as e:

flash(f'Error creating directory: {e}', 'error')

return redirect(url\_for('index', path=current\_path))

@app.route('/delete\_item/<path:item\_path>', methods=['POST'])

def delete\_item(item\_path):

decoded\_item\_path = unquote(item\_path)

full\_item\_path = safe\_path\_join(app.config['UPLOAD\_FOLDER'], decoded\_item\_path)

if full\_item\_path is None: # safe\_path\_join返回None表示路径不安全

flash('Invalid deletion path.', 'error')

return redirect(url\_for('index', path=os.path.dirname(decoded\_item\_path).strip('/')))

if not os.path.exists(full\_item\_path):

flash('Item not found.', 'error')

return redirect(url\_for('index', path=os.path.dirname(decoded\_item\_path).strip('/')))

try:

if os.path.isdir(full\_item\_path):

if not os.listdir(full\_item\_path): # 检查目录是否为空

os.rmdir(full\_item\_path) # rmdir 只能删除空目录

flash(f'Directory "{os.path.basename(decoded\_item\_path)}" deleted successfully.', 'success')

else:

flash(f'Error deleting directory "{os.path.basename(decoded\_item\_path)}": Directory is not empty. Only empty directories can be deleted.', 'error')

else:

os.remove(full\_item\_path)

flash(f'File "{os.path.basename(decoded\_item\_path)}" deleted successfully.', 'success')

except OSError as e:

flash(f'Error deleting item "{os.path.basename(decoded\_item\_path)}": {e}', 'error')

return redirect(url\_for('index', path=os.path.dirname(decoded\_item\_path).strip('/')))

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

# 确保 uploads 目录存在

if not os.path.exists(app.config['UPLOAD\_FOLDER']):

os.makedirs(app.config['UPLOAD\_FOLDER'])

app.run(debug=True, host='0.0.0.0') # 允许从虚拟机外部访问

## 8.5 index.html

web部分前端页面

<!DOCTYPE html>

<html lang="zh-CN">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Web FTP 文件管理</title>

<style>

body {

font-family: Arial, sans-serif;

margin: 20px;

background-color: #f4f4f4;

color: #333;

}

.container {

max-width: 800px;

margin: 0 auto;

background-color: #fff;

padding: 20px;

border-radius: 8px;

box-shadow: 0 2px 10px rgba(0,0,0,0.1);

}

h1, h2 {

color: #0056b3;

}

.current-path {

background-color: #e9e9e9;

padding: 10px;

border-radius: 5px;

margin-bottom: 20px;

font-weight: bold;

display: flex;

justify-content: space-between;

align-items: center;

}

.current-path a {

text-decoration: none;

color: #007bff;

font-weight: normal;

}

.current-path a:hover {

text-decoration: underline;

}

.file-list ul {

list-style: none;

padding: 0;

}

.file-list li {

background-color: #f9f9f9;

border: 1px solid #ddd;

padding: 10px;

margin-bottom: 5px;

border-radius: 4px;

display: flex; /\* 使用 Flexbox 布局 \*/

align-items: center; /\* 垂直居中对齐项目 \*/

justify-content: space-between; /\* 左右两端对齐内容 \*/

}

.file-list li .item-info { /\* 包裹图标和名称 \*/

display: flex;

align-items: center;

flex-grow: 1; /\* 允许占据剩余空间 \*/

text-decoration: none; /\* 移除文件名的下划线 \*/

color: #333; /\* 文件名颜色 \*/

}

.file-list li .item-info a { /\* 目录名称链接的样式 \*/

color: #007bff;

text-decoration: none;

margin-left: 5px; /\* 目录名与图标的间距 \*/

}

.file-list li .item-info a:hover {

text-decoration: underline;

}

.file-icon {

margin-right: 10px;

color: #555;

font-size: 1.2em; /\* 增大图标 \*/

}

.file-icon.folder::before { content: '📁'; }

.file-icon.file::before { content: '📄'; }

/\* 操作按钮组 \*/

.actions {

display: flex;

gap: 5px; /\* 按钮之间的间距 \*/

}

.upload-form, .create-dir-form, .delete-form {

background-color: #e9f7ff;

border: 1px solid #cceeff;

padding: 15px;

border-radius: 8px;

margin-top: 20px;

}

input[type="file"], input[type="text"], input[type="submit"], button {

padding: 8px 12px;

border-radius: 4px;

border: 1px solid #ccc;

margin-right: 5px;

}

input[type="submit"], button {

background-color: #007bff;

color: white;

border: none;

cursor: pointer;

}

input[type="submit"]:hover, button:hover {

background-color: #0056b3;

}

.messages {

list-style: none;

padding: 0;

margin-top: 10px;

}

.messages li {

padding: 10px;

margin-bottom: 10px;

border-radius: 5px;

font-weight: bold;

}

.messages .success {

background-color: #d4edda;

color: #155724;

border-color: #c3e6cb;

}

.messages .error {

background-color: #f8d7da;

color: #721c24;

border-color: #f5c6cb;

}

.messages .info {

background-color: #d1ecf1;

color: #0c5460;

border-color: #bee5eb;

}

.delete-btn {

background-color: #dc3545;

color: white;

border: none;

padding: 5px 10px;

border-radius: 4px;

cursor: pointer;

}

.delete-btn:hover {

background-color: #c82333;

}

.download-btn { /\* 新增下载按钮样式 \*/

background-color: #28a745; /\* 绿色 \*/

color: white;

border: none;

padding: 5px 10px;

border-radius: 4px;

cursor: pointer;

}

.download-btn:hover {

background-color: #218838;

}

</style>

</head>

<body>

<div class="container">

<h1>Web FTP 文件管理系统</h1>

<div class="current-path">

<span>当前路径: /uploads/{{ current\_path }}</span>

{% if parent\_path is not none %}

<a href="{{ url\_for('index', path=parent\_path) }}">返回上一级</a>

{% endif %}

</div>

<ul class="messages">

{% for category, message in get\_flashed\_messages(with\_categories=true) %}

<li class="{{ category }}">{{ message }}</li>

{% endfor %}

</ul>

<h2>文件列表</h2>

<div class="file-list">

<ul>

{% if directories %}

{% for dir in directories %}

<li>

<div class="item-info">

<span class="file-icon folder"></span>

<a href="{{ url\_for('index', path=current\_path + '/' + dir) }}">{{ dir }}</a>

</div>

<div class="actions">

<form action="{{ url\_for('delete\_item', item\_path=current\_path + '/' + dir) }}" method="post" style="display:inline;">

<button type="submit" class="delete-btn" onclick="return confirm('确定要删除目录 {{ dir }} 吗？空目录才能被删除。');">删除</button>

</form>

</div>

</li>

{% endfor %}

{% endif %}

{% if files %}

{% for file in files %}

<li>

<div class="item-info">

<span class="file-icon file"></span>

<span>{{ file }}</span>

</div>

<div class="actions">

<a href="{{ url\_for('download\_file', filepath=file if not current\_path else current\_path + '/' + file) }}" class="download-btn">下载</a>

<form action="{{ url\_for('delete\_item', item\_path=file if not current\_path else current\_path + '/' + file) }}" method="post" style="display:inline;">

<button type="submit" class="delete-btn" onclick="return confirm('确定要删除文件 {{ file }} 吗？');">删除</button>

</form>

</div>

</li>

{% endfor %}

{% else %}

{% if not directories %}

<li>当前目录为空。</li>

{% endif %}

{% endif %}

</ul>

</div>

<h2>上传文件到当前目录</h2>

<div class="upload-form">

<form action="{{ url\_for('upload\_file') }}" method="post" enctype="multipart/form-data">

<input type="file" name="file" required>

<input type="hidden" name="current\_path" value="{{ current\_path }}">

<input type="submit" value="上传">

</form>

</div>

<h2>在当前目录创建新文件夹</h2>

<div class="create-dir-form">

<form action="{{ url\_for('create\_dir') }}" method="post">

<input type="text" name="dir\_name" placeholder="请输入文件夹名称" required>

<input type="hidden" name="current\_path" value="{{ current\_path }}">

<input type="submit" value="创建文件夹">

</form>

</div>

</div>

</body>

</html>

## 8.6 scraper.py

爬虫部分代码

import socket

import re

import os

import sys

from urllib.parse import urlparse

from bs4 import BeautifulSoup

# 定义下载保存目录

DOWNLOADS\_DIR = 'downloads'

def create\_downloads\_dir():

"""创建下载目录"""

if not os.path.exists(DOWNLOADS\_DIR):

os.makedirs(DOWNLOADS\_DIR)

print(f"Created download directory: {DOWNLOADS\_DIR}")

# 简单的文件名清理函数，避免使用werkzeug.utils.secure\_filename

def simple\_secure\_filename(filename):

"""

简单的文件名清理，替换掉可能导致问题的字符，

并确保只保留字母、数字、点、下划线和连字符。

"""

# 替换非法字符为下划线

filename = re.sub(r'[^a-zA-Z0-9\_.-]', '\_', filename)

# 替换连续下划线为单个

filename = re.sub(r'\_\_+', '\_', filename)

# 移除开头和结尾的特殊字符

filename = filename.strip('.\_-')

if not filename:

filename = "untitled" # 避免空文件名

return filename

def get\_http\_response\_via\_socket(url):

"""

通过Socket发送HTTP GET请求并接收响应。

处理HTTP头和内容分离，并尝试处理Content-Length。

"""

parsed\_url = urlparse(url)

host = parsed\_url.netloc

path = parsed\_url.path if parsed\_url.path else '/'

# 默认使用80端口，如果协议是http

port = 80

# 对于HTTPS，此简单示例不直接支持

if parsed\_url.scheme == 'https':

print("HTTPS not directly supported by raw socket in this example. Please use HTTP URLs.")

return None, None

try:

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.settimeout(10) # 设置超时

s.connect((host, port))

request\_line = f"GET {path} HTTP/1.1\r\n"

headers = f"Host: {host}\r\nConnection: close\r\nUser-Agent: SimpleSocketScraper/1.0\r\n\r\n"

request = (request\_line + headers).encode('utf-8')

s.sendall(request)

# 接收响应

response\_bytes = b''

while True:

chunk = s.recv(4096)

if not chunk:

break

response\_bytes += chunk

s.close()

# 分离头部和正文

header\_end = response\_bytes.find(b'\r\n\r\n')

if header\_end == -1:

print(f"Error: Could not find end of headers for {url}")

return None, None

headers\_raw = response\_bytes[:header\_end].decode('latin-1') # 头部通常是latin-1编码

body\_raw = response\_bytes[header\_end + 4:] # +4 for '\r\n\r\n'

headers\_dict = {}

for line in headers\_raw.split('\r\n'):

if ':' in line:

key, value = line.split(':', 1)

headers\_dict[key.strip().lower()] = value.strip()

# 检查Content-Length，并尝试确保接收完整内容

content\_length\_str = headers\_dict.get('content-length')

if content\_length\_str:

content\_length = int(content\_length\_str)

if len(body\_raw) < content\_length:

print(f"Warning: Body received ({len(body\_raw)} bytes) is less than Content-Length ({content\_length} bytes) for {url}")

return headers\_dict, body\_raw

except socket.error as e:

print(f"Socket error for {url}: {e}")

return None, None

except Exception as e:

print(f"An unexpected error occurred for {url}: {e}")

return None, None

def save\_text\_to\_file(text\_content, filename):

"""将文本内容保存到文件"""

filepath = os.path.join(DOWNLOADS\_DIR, filename)

with open(filepath, 'w', encoding='utf-8') as f:

f.write(text\_content)

print(f"Text content saved to: {filepath}")

def download\_image\_via\_socket(image\_url, image\_filename):

"""

通过Socket下载图片并保存。

"""

parsed\_url = urlparse(image\_url)

host = parsed\_url.netloc

path = parsed\_url.path if parsed\_url.path else '/'

port = 80

# 对于HTTPS，此简单示例不直接支持

if parsed\_url.scheme == 'https':

print(f"Skipping HTTPS image: {image\_url}")

return

try:

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

s.settimeout(10)

s.connect((host, port))

request\_line = f"GET {path} HTTP/1.1\r\n"

headers = f"Host: {host}\r\nConnection: close\r\nUser-Agent: SimpleSocketScraper/1.0\r\n\r\n"

request = (request\_line + headers).encode('utf-8')

s.sendall(request)

# 接收响应

response\_bytes = b''

while True:

chunk = s.recv(4096)

if not chunk:

break

response\_bytes += chunk

s.close()

# 分离头部和正文

header\_end = response\_bytes.find(b'\r\n\r\n')

if header\_end == -1:

print(f"Error: Could not find end of headers for image {image\_url}")

return

headers\_raw = response\_bytes[:header\_end].decode('latin-1')

body\_raw = response\_bytes[header\_end + 4:]

# 可以解析headers\_raw来获取Content-Length等信息，但此处直接保存整个body

filepath = os.path.join(DOWNLOADS\_DIR, image\_filename)

with open(filepath, 'wb') as f:

f.write(body\_raw)

print(f"Image downloaded and saved to: {filepath}")

except socket.error as e:

print(f"Socket error downloading image {image\_url}: {e}")

except Exception as e:

print(f"An unexpected error occurred downloading image {image\_url}: {e}")

def main():

if len(sys.argv) < 2:

print("Usage: python scraper.py <URL>")

sys.exit(1)

target\_url = sys.argv[1]

create\_downloads\_dir()

print(f"Grabbing data from: {target\_url}")

headers, body\_bytes = get\_http\_response\_via\_socket(target\_url)

if headers is None or body\_bytes is None:

print("Failed to get HTTP response.")

return

content\_type = headers.get('content-type', '').lower()

if 'text/html' in content\_type:

# 尝试解码HTML内容

charset = 'utf-8' # 默认UTF-8

match = re.search(r'charset=([\w-]+)', content\_type)

if match:

charset = match.group(1)

try:

html\_content = body\_bytes.decode(charset)

except UnicodeDecodeError:

print(f"Could not decode with {charset}, trying utf-8...")

html\_content = body\_bytes.decode('utf-8', errors='ignore')

except Exception as e:

print(f"Error decoding HTML: {e}, trying utf-8 with ignore errors...")

html\_content = body\_bytes.decode('utf-8', errors='ignore')

soup = BeautifulSoup(html\_content, 'lxml')

# 抓取文本信息

page\_title = soup.title.string if soup.title else 'No Title'

text\_filename = simple\_secure\_filename(f"{page\_title[:50]}\_text.txt") # 限制文件名长度

save\_text\_to\_file(soup.get\_text(), text\_filename)

# 抓取图片信息并下载

image\_tags = soup.find\_all('img')

print(f"Found {len(image\_tags)} image tags.")

for i, img in enumerate(image\_tags):

img\_src = img.get('src')

if img\_src:

# 完善图片URL（处理相对路径）

if not img\_src.startswith(('http://', 'https://', '//')):

# 简单拼接，更复杂的应使用urljoin

if img\_src.startswith('/'): # 根相对路径

img\_url = urlparse(target\_url).scheme + "://" + urlparse(target\_url).netloc + img\_src

else: # 相对路径

# 获取基础URL的目录部分

base\_url\_parsed = urlparse(target\_url)

base\_path = os.path.dirname(base\_url\_parsed.path)

# 确保路径以斜杠结尾，以便正确拼接

if base\_path and not base\_path.endswith('/'):

base\_path += '/'

elif not base\_path: # 如果是根路径

base\_path = '/'

img\_url = base\_url\_parsed.scheme + "://" + base\_url\_parsed.netloc + base\_path + img\_src

elif img\_src.startswith('//'): # 协议相对路径

img\_url = urlparse(target\_url).scheme + ":" + img\_src

else:

img\_url = img\_src

# 提取文件名

img\_name = os.path.basename(urlparse(img\_url).path)

if not img\_name or '.' not in img\_name: # 如果路径末尾没有文件名或没有扩展名，给一个默认名

# 尝试从Content-Type获取扩展名，或者使用通用jpg

content\_type\_img = headers.get('content-type', '').lower()

if 'image/' in content\_type\_img:

ext = content\_type\_img.split('/')[-1]

img\_name = f"image\_{i}.{ext}"

else:

img\_name = f"image\_{i}.jpg" # 默认jpg

img\_filename = simple\_secure\_filename(f"img\_{i}\_{img\_name}") # 加入索引防止重名

print(f"Attempting to download image: {img\_url}")

download\_image\_via\_socket(img\_url, img\_filename)

else:

print(f"Content-Type is {content\_type}, not HTML. Only text/html is processed for text/image extraction.")

# 如果是其他类型，可以直接保存原始文件

file\_extension = content\_type.split('/')[-1] if '/' in content\_type else 'bin'

filename = simple\_secure\_filename(f"downloaded\_content.{file\_extension}")

filepath = os.path.join(DOWNLOADS\_DIR, filename)

with open(filepath, 'wb') as f:

f.write(body\_bytes)

print(f"Non-HTML content saved to: {filepath}")

if \_\_name\_\_ == "\_\_main\_\_":

main()