简易版xshell项目目录

[1 系统功能描述 4](#_Toc28017)

[1.1 系统具有的功能描述 4](#_Toc13298)

[1.2 所用技术与模块描述 4](#_Toc31765)

[1.3 ssh暗影坤手概述 4](#_Toc17450)

[2 系统功能实现 5](#_Toc28705)

[2.1 服务器信息管理的功能设计与实现 5](#_Toc28295)

[2.2 连接服务器的功能设计与实现 5](#_Toc32677)

[2.3 添加服务器的功能设计与实现 7](#_Toc11727)

[2.4 删除服务器的功能设计与实现 9](#_Toc11126)

[2.5 修改服务器的功能设计与实现 9](#_Toc21502)

[2.6 Linux基础命令的功能设计与实现 11](#_Toc2076)

[2.7 vim编辑器功能设计与实现 12](#_Toc27671)

[2.8 文件远程上传下载的功能设计与实现 16](#_Toc25301)

[2.9 服务器监控功能设计与实现 18](#_Toc27514)

[2.10 批量下载日志文件和查看日志功能设计与实现 28](#_Toc19486)

[2.11界面窗口的实现 30](#_Toc14054)

[2.12 打包成可执行文件的实现 33](#_Toc6951)

[3 系统完成效果 33](#_Toc24718)

[3.1 服务器信息结果展示 33](#_Toc23049)

[3.2 连接服务器结果展示 33](#_Toc28659)

[3.3 添加服务器信息结果展示 33](#_Toc18640)

[3.4 删除服务器信息结果展示 34](#_Toc18778)

[3.5 修改服务器信息结果展示 35](#_Toc21895)

[3.6 Linux交互命令结果展示 35](#_Toc14868)

[3.7 vim编辑器结果展示 36](#_Toc6629)

[3.8 文件上传下载结果展示 37](#_Toc6135)

[3.9 服务器监控功能结果展示 40](#_Toc13151)

[3.10 批量下载日志文件和查看日志文件结果展示 41](#_Toc5564)

[3.11窗口界面结果展示 42](#_Toc12441)

[3.12 打包成可执行文件的结果展示 43](#_Toc15126)

[4 总结与分析 44](#_Toc26513)

1 系统功能描述

1.1 系统具有的功能描述

（1）连接服务器：用户可以通过系统连接到远程服务器，系统支持多个服务器配置，并且可以方便地管理这些配置。

（2）执行命令：用户可以在连接成功后，在界面中输入命令，并执行在远程服务器上执行。执行结果会显示在界面中，包括标准输出和标准错误。

（3）上传和下载文件：用户可以通过系统上传本地文件到远程服务器，或者下载远程服务器上的文件到本地。

（4）服务器监控：系统可以监控远程服务器的 CPU 使用率、内存使用率和进程列表，实时显示在界面上。

（5）查看日志文件：用户可以通过系统查看远程服务器上的日志文件，并且可以选择下载到本地进行查看。

（6）Vim编辑文件：用户可以在界面上编辑远程服务器上的文件，编辑后可以保存到服务器上。

1.2 所用技术与模块描述

（1）JSON模块（json）：用于存储和加载配置信息、与服务器交换数据等。

（2）操作系统路径模块（os.path）：主要用于处理文件路径，比如检查文件是否存在、获取文件或目录的详细信息等。

（3）正则表达式模块（re）：Python的正则表达式模块，用于字符串匹配和替换。

（4）套接字模块（socket）：提供了网络通信的底层接口，允许应用程序在网络上发送和接收数据。

（5）子进程模块（subprocess）：允许从Python脚本中启动新的应用程序，连接到它们的输入/输出/错误管道，并获取它们的返回码。

（6）线程模块（threading）：提供了对线程的基本支持，线程是轻量级的进程，允许在Python程序中并发执行代码。

（7）时间模块（time）：提供了与时间相关的各种功能，如获取当前时间、执行时间延迟等。

（8）Tkinter简单对话框模块（tkinter.simpledialog）：Tkinter GUI库的一部分，提供了简单的对话框功能，如输入对话框、消息框等。

（9）Paramiko模块（paramiko）：一个实现SSHv2协议的Python库，支持连接和操作SSH服务器。

（10）Watchdog事件模块（watchdog.events）：watchdog库的一部分，提供了文件系统事件（如文件创建、修改、删除等）的接口。

（11）Watchdog观察者模块（watchdog.observers）：watchdog库的一部分，提供了跨平台的文件系统事件观察者实现。

1.3 ssh暗影坤手概述

一个简单易用的SSH管理工具，允许用户通过图形化界面实现对虚拟机（VM）的基本操作。通过集成SSH协议，该工具能够提供远程登录、命令执行、文件传输等核心功能，为用户提供便捷、安全的虚拟机管理体验。

2 系统功能实现

2.1 服务器信息管理的功能设计与实现

将所有服务器信息保存在json文件中，调用时借助函数加载文件，读取数据。

代码展示如下：

def load\_server(self):  
 with open('server.json', "r", encoding='utf-8') as fp:  
 self.servers = json.load(fp)  
 print(f"获取到的服务器信息：{self.servers}")  
  
def save\_server(self):  
 with open('server.json', "w", encoding='utf-8') as fp:  
 fp.write(json.dumps(self.servers))

2.2 连接服务器的功能设计与实现

（1）读取json文件中的数据，将ip、端口、用户名等信息传到connect中连接服务器。

代码展示如下：

def connecting(self):  
 self.connected = None  
 self.ssh.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy)  
 try:  
 # 连接到服务器  
 self.ssh.connect(hostname=self.server["ip"], port=self.server["port"], username=self.server["username"],  
 password=self.server["password"])  
 print(f"【success】连接到服务器【{self.server['ip']}】成功！！！")  
 self.connected = True  
 self.get\_pwd()  
  
 return True  
 except Exception as e:  
 print(f"【errors】连接到服务器【{self.server['ip']}】失败！！！")  
 self.connected = False  
 return False

（2）创建连接指定服务器函数，并且创建线程拥堵单独执行连接操作，不影响其他操作的执行，创建显示函数将连接成功或失败信息展示到界面，以便用户清楚服务器是否已经连接成功。

代码展示如下：

# 连接到指定的服务器  
def connect(self, server):  
 print("连接到指定的服务器")  
 self.mssh.server = server  
 print(f"服务器信息：{self.mssh.server}")  
  
 socket.setdefaulttimeout(5)  
  
 # 在单独的线程中执行连接操作  
 thread\_connect = threading.Thread(target=self.connect\_thread, args=(server,))  
 thread\_connect.start()  
  
 self.pop\_win.destroy()  
  
def connect\_thread(self, server):  
 result = self.mssh.connecting()  
 if result:  
 self.is\_connected = True  
 self.update\_connection\_status(server, True)  
 else:  
 self.is\_connected = False  
 self.update\_connection\_status(server, False)  
#服务器连接展示  
def update\_connection\_status(self, server, connected):  
 if connected:  
 self.label\_server.config(text=f"服务器{server['ip']}连接成功", foreground="green")  
 else:  
 self.label\_server.config(text=f"服务器{server['ip']}连接失败", foreground="red")

2.3 添加服务器的功能设计与实现

（1）首先创建函数用于添加服务器的操作，然后输入服务器的IP地址，账户名和密码添加服务器，最后将服务器信息保存到json文件中。

代码展示如下：

def add\_server(self, ip, port, username, password, description):  
 self.servers.append({  
 "description": description,  
 "ip": ip,  
 "port": port,  
 "username": username,  
 "password": password  
 })  
 self.save\_server()

1. 创建新建函数，用于与用户交互的图形化界面，让用户输入服务器的IP地址，账户名和密码添加服务器信息。

代码展示如下：

# 新建连接  
def new\_connect\_ui(self):  
 self.pop\_win = tk.Toplevel(self.root)  
 # 设置标题和大小  
 self.pop\_win.title("新建连接")  
 self.pop\_win.geometry("300x200+600+250")  
 frame = tk.Frame(self.pop\_win)  
 frame.grid()  
  
 # IP  
 ip\_label = tk.Label(frame, text="服务器IP地址:")  
 ip\_label.grid(row=0, column=0)  
 self.ip\_entry = tk.Entry(frame)  
 self.ip\_entry.grid(row=0, column=1)  
  
 # Port  
 port\_label = tk.Label(frame, text="服务器端口号:")  
 port\_label.grid(row=1, column=0)  
 self.port\_entry = tk.Entry(frame)  
 self.port\_entry.grid(row=1, column=1)  
  
 # Username  
 username\_label = tk.Label(frame, text="用户名:")  
 username\_label.grid(row=2, column=0)  
 self.username\_entry = tk.Entry(frame)  
 self.username\_entry.grid(row=2, column=1)  
  
 # Password  
 password\_label = tk.Label(frame, text="密码:")  
 password\_label.grid(row=3, column=0)  
 self.password\_entry = tk.Entry(frame, show="\*")  
 self.password\_entry.grid(row=3, column=1)  
  
 # Description  
 description\_label = tk.Label(frame, text="备注信息:")  
 description\_label.grid(row=4, column=0)  
 self.description\_entry = tk.Entry(frame)  
 self.description\_entry.grid(row=4, column=1)  
  
 # Button  
 submit\_button = tk.Button(frame, text="保存", command=self.save\_server\_info)  
 submit\_button.grid(row=5, columnspan=2)

1. 获取用户所输入的信息，将服务器信息保存到文件。

代码展示如下：

def save\_server\_info(self):  
 ip = self.ip\_entry.get()  
 port = self.port\_entry.get()  
 username = self.username\_entry.get()  
 password = self.password\_entry.get()  
 description = self.description\_entry.get()  
  
 # 调用 add\_server 函数，将获取到的信息添加到服务器列表中  
 ssh.add\_server(ip=ip, port=int(port), username=username, password=password, description=description)  
 messagebox.showinfo("成功","服务器信息已保存！！")  
 # print("服务器信息已保存")

2.4 删除服务器的功能设计与实现

创建删除函数，当用户点击确定时，调用remove移除服务器信息，并且返回删除成功提示信息。

代码展示如下：

def del\_connect(self, server):  
 print("删除指定的服务器")  
 # 输出服务器信息  
 print(f"服务器信息：{server}")  
  
 # 确认是否删除服务器  
 confirm = messagebox.askyesno("确认删除", f"确认删除服务器 {server['ip']} 吗？")  
 if confirm:  
 # 从服务器列表中删除指定的服务器  
 if server in self.mssh.servers:  
 self.mssh.servers.remove(server)  
 self.pop\_win.destroy()  
 self.open\_connect\_ui()  
 messagebox.showinfo("删除成功", f"服务器 {server['ip']} 已成功删除！")  
 else:  
 messagebox.showerror("错误", "服务器不存在，无法删除")  
  
 # 保存更新后的服务器列表到配置文件中  
 self.mssh.save\_server()

2.5 修改服务器的功能设计与实现

（1）创建修改函数，用于与用户交互的图形化界面，让用户修改服务器的IP地址、账户名和密码等服务器信息。

代码展示如下：

def modify\_server(self, server):  
 print("修改服务器信息")  
 self.mssh.server = server.copy() # 添加这一行来复制服务器信息  
 self.pop\_win\_modify = tk.Toplevel(self.root)  
 frame = tk.Frame(self.pop\_win\_modify)  
 frame.grid()  
  
 # IP  
 ip\_label = tk.Label(frame, text="服务器IP地址:")  
 ip\_label.grid(row=0, column=0)  
 self.ip\_entry = tk.Entry(frame, textvariable=tk.StringVar(value=server['ip']))  
 self.ip\_entry.grid(row=0, column=1)  
  
 # Port  
 port\_label = tk.Label(frame, text="服务器端口号:")  
 port\_label.grid(row=1, column=0)  
 self.port\_entry = tk.Entry(frame, textvariable=tk.StringVar(value=str(server['port'])))  
 self.port\_entry.grid(row=1, column=1)  
  
 # Username  
 username\_label = tk.Label(frame, text="用户名:")  
 username\_label.grid(row=2, column=0)  
 self.username\_entry = tk.Entry(frame, textvariable=tk.StringVar(value=server['username']))  
 self.username\_entry.grid(row=2, column=1)  
  
 # Password  
 password\_label = tk.Label(frame, text="密码:")  
 password\_label.grid(row=3, column=0)  
 self.password\_entry = tk.Entry(frame, show="\*", textvariable=tk.StringVar(value=server['password']))  
 self.password\_entry.grid(row=3, column=1)  
  
 # Description  
 description\_label = tk.Label(frame, text="备注信息:")  
 description\_label.grid(row=4, column=0)  
 self.description\_entry = tk.Entry(frame, textvariable=tk.StringVar(value=server['description']))  
 self.description\_entry.grid(row=4, column=1)  
  
 # Button  
 submit\_button = tk.Button(frame, text="保存", command=self.save\_modified\_server)  
 submit\_button.grid(row=5, columnspan=2)

（2）获取修改后的服务器信息，保存到json文件中。

代码展示如下：

def save\_modified\_server(self):  
 modified\_server = {  
 "ip": self.ip\_entry.get(),  
 "port": int(self.port\_entry.get()),  
 "username": self.username\_entry.get(),  
 "password": self.mssh.server["password"], # 假设密码不变，保持原值  
 "description": self.description\_entry.get(),  
 }  
 index = self.mssh.servers.index(self.mssh.server)  
 self.mssh.servers[index] = modified\_server  
 messagebox.showinfo("成功", "服务器信息已成功修改！")  
 self.pop\_win\_modify.destroy()  
 self.open\_connect\_ui()  
 self.mssh.save\_server()

2.6 执行交互式命令的功能设计与实现

主要调用paramiko库的exec\_command方法执行命令，同时获取标准输入、标准输出和标准错误。

（1）基本命令cd的实现

创建exec方法主要用于执行远程命令，并处理用户输入的路径，确保命令相对于当前路径执行。方法接收用户输入的命令，执行命令（若是"cd"命令则更新当前路径），获取命令执行结果，并返回执行结果。

代码展示如下：

def exec(self, command):  
 if command.startswith("cd"):  
 tmp\_path = command.split(" ")[-1]  
 if tmp\_path.startswith("/"):  
 # 绝对路径处理  
 self.path = tmp\_path  
 else:  
 # todo 如果用户输入的是一个相对路径呢？  
 print("用户输入的是相对路径")  
 if tmp\_path.startswith(".."):  
 self.path = self.path + "/" + tmp\_path  
 else:  
 self.path = self.path + tmp\_path  
 else:  
 # 根据当前的路径去执行指令  
 command = f"cd {self.path} && {command}"  
 print(f"->输入指令({self.path})：{command}")  
 stdin, stdout, stderr = self.ssh.exec\_command(command)  
 result = stdout.read().decode("utf-8")  
 print(f"<-返回：{result}")  
 # 用户输入的路径里包含.. ，通过pwd刷新当前路径  
 if command.find("..") > 0:  
 self.get\_pwd()  
 return result

1. 创建函数用于获取当前路径

代码展示如下：

# 使用pwd更新当前路径  
def get\_pwd(self):  
 stdin, stdout, stderr = self.ssh.exec\_command(f"cd {self.path} && pwd")  
 result = stdout.read().decode("utf-8")  
 self.path = result.strip()

1. 交互式命令实现

通过SSH与远程服务器进行交互,用户可以输入命令并发送到远程服务器，同时应用会读取并显示来自远程服务器的输出。所有的输入/输出处理都在单独的线程中进行，以保持GUI的响应性。

代码展示如下：

def \_read\_channel(self, channel,output\_callback):  
 """在独立线程中读取通道输出，并调用回调函数处理"""  
 while True:  
 if channel.recv\_ready():  
 data = channel.recv(1024)  
 output\_callback(data)

def handle\_output(self, data):  
 # 先解码数据  
 decoded\_data = data.decode("utf-8")  
 # 去除颜色转义序列  
 cleaned\_data = re.sub(r'\x1B\[([0-?]\*[ -/]\*[@-~])', '', decoded\_data)  
 # 插入到文本框的末尾  
 self.txt\_result.insert(tk.END, cleaned\_data)  
 # 确保滚动条跟随到最后  
 self.txt\_result.see(tk.END)

def okk(self,event):  
 self.command.set('q')  
 self.ok(self)

def ok(self, event):  
 command = self.command.get()  
 if not self.mssh.connected:  
 self.txt\_result.insert(tk.END, "未连接到服务器！\n")  
 return  
 # 如果ssh\_channel尚未创建，创建一个新的  
 if self.mssh\_channel is None:  
 self.mssh\_channel = self.mssh.ssh.invoke\_shell()  
 self.mssh\_channel.set\_combine\_stderr(True)  
 try:  
 # 使用ssh\_channel发送命令  
 self.mssh\_channel.send(f"{command}\n")  
 # 创建线程监听输出并调用UI提供的回调函数  
 t = threading.Thread(target=self.mssh.\_read\_channel, args=(self.mssh\_channel, self.handle\_output))  
 t.daemon = True  
 t.start()  
 # 设置延时清空输入框  
 self.root.after(2000, lambda: self.command.set(""))  
 except Exception as e:  
 self.txt\_result.insert(tk.END, f"执行错误: {str(e)}\n")  
  
 # 清空输入框  
 self.command.set("")

2.7 vim编辑器功能设计与实现

利用watchdog模块监控文件系统的更改，如文件的创建、修改、移动或删除等，并在这些事件发生时执行自定义操作。

（1）定义watch\_and\_upload方法用于监视本地文件的修改并在远程服务器上进行相应的上传操作。

代码展示如下：

def watch\_and\_upload(self, local\_file\_path, remote\_full\_path):  
 class FileModifiedHandler(FileSystemEventHandler):  
 def \_\_init\_\_(self, ssh\_handler, remote\_full\_path):  
 self.ssh\_handler = ssh\_handler  
 self.remote\_full\_path = remote\_full\_path  
  
 def on\_modified(self, event):  
 if event.is\_directory or not event.src\_path == local\_file\_path:  
 return  
 # messagebox.showinfo("文件被修改", f"文件被修改：{event.src\_path}")  
 try:  
 self.ssh\_handler.connecting() # 重新连接SSH（如果需要）  
 sftp = self.ssh\_handler.ssh.open\_sftp()  
 sftp.put(local\_file\_path, self.remote\_full\_path)  
 sftp.close()  
 # messagebox.showinfo("文件上传成功", f"vim修改成功: {local\_file\_path} -> {self.remote\_full\_path}")  
 except Exception as e:  
 messagebox.showerror("文件上传失败", f"文件上传失败: {e}")  
  
 event\_handler = FileModifiedHandler(self, remote\_full\_path)  
 observer = Observer()  
 observer.schedule(event\_handler, path=os.path.dirname(local\_file\_path), recursive=False)  
 observer.start()  
  
 try:  
 # 等待用户操作或其他信号来停止监视  
 # 这里使用time.sleep作为示例，但建议使用更好的机制，如事件或信号  
 while True:  
 time.sleep(1)  
 except KeyboardInterrupt:  
 observer.stop()  
  
 observer.join()

（2）创建vim\_edit 函数用于在vscode中编辑文件。首先提示用户输入要编辑的文件名，并获取远程文件的路径，用户选择本地目录，并构造本地文件的完整路径，连接到远程服务器，并创建 SFTP 客户端，检查远程文件是否存在，如果不存在则在远程服务器上创建空文件。下载文件到本地，并启动文件监视线程 watch\_and\_upload，然后调用子进程打开文件编辑器。

def vim\_edit(self):  
 remote\_file = simpledialog.askstring("vim", "请输入要编辑的完整远程文件路径:")  
 if remote\_file is None: # 用户点击“取消”或关闭对话框  
 return  
 elif not remote\_file: # 用户输入了空字符串  
 messagebox.showerror("错误", "请输入文件路径！")  
 return  
  
 # 检查用户是否输入了以'/'开头的绝对路径  
 if not remote\_file.startswith('/'):  
 messagebox.showerror("错误", "请输入绝对路径，例如：/home/user/file.txt")  
 return  
  
 # 临时文件目录  
 tmp = filedialog.askdirectory(title="选择下载到的本地目录")  
 if not tmp: # 用户取消了选择目录  
 return  
  
 local\_file\_path = os.path.join(tmp, os.path.basename(remote\_file))  
  
 try:  
 self.connecting()  
 sftp = self.ssh.open\_sftp()  
  
 # 检查远程文件是否存在  
 try:  
 sftp.stat(remote\_file)  
 except FileNotFoundError:  
 # 如果不存在，则在远程服务器上创建空文件  
 with sftp.open(remote\_file, 'wb') as f:  
 pass # 写入空内容或者模板内容  
  
 # 下载文件到本地  
 sftp.get(remote\_file, local\_file\_path)  
 sftp.close()  
  
 # 启动文件监视线程  
 watch\_thread = threading.Thread(target=self.watch\_and\_upload, args=(local\_file\_path, remote\_file))  
 watch\_thread.start()  
  
 # 打开文件编辑器  
 subprocess.run(['Code', local\_file\_path], shell=True)  
  
 except Exception as e:  
 messagebox.showerror("文件下载失败", f"文件下载失败：{e}")  
 print(f"文件下载失败：{e}")

2.8 文件远程上传下载的功能设计与实现

（1）文件上传功能实现

首先检查是否已连接到服务器。如果未连接到服务器，则显示错误消息并返回。如果已连接到服务器，则调用filedialog.askopenfilename()方法请求用户选择要上传的文件。接下来，如果用户选择了文件，则提取文件名并创建一个SFTP客户端。然后，计算远程路径，并在控制台打印文件和远程路径信息。最后，使用SFTP客户端将本地文件上传到远程路径，并显示上传成功的消息框。如果用户未选择文件，则输出“请选择要上传的文件”。如果出现任何异常，则显示文件上传失败的错误消息。

代码展示如下：

# 上传文件  
def upload\_file(self):  
 if not self.mssh.connected:  
 messagebox.showerror("错误", "请先连接到服务器!")  
 return  
 else:  
 self.file\_path = filedialog.askopenfilename()  
 try:  
 if self.file\_path:  
 # 获取文件名  
 file\_name = self.file\_path.split("/")[-1]  
  
 # 创建 SFTP 客户端  
 sftp\_client = self.mssh.ssh.open\_sftp()  
 remote\_path = f"{self.mssh.path}/{file\_name}"  
 print(f"文件：{self.file\_path},上传到：{remote\_path}")  
 # 上传文件  
 sftp\_client.put(self.file\_path, remote\_path)  
 messagebox.showinfo("成功", "上传成功")  
 sftp\_client.close()  
 else:  
 print("请选择要上传的文件")  
 except Exception as e:  
 messagebox.showerror("错误", f"文件上传失败: {e}")

#判断是否连接服务器

def vim\_edit(self):  
 if not self.mssh.connected:  
 messagebox.showerror("错误", "请先连接到服务器!")  
 return  
 self.mssh.vim\_edit()

（2）文件下载功能实现

使用tk中的simpledialog.askstring 获取用户输入的要下载文件名，filedialog.askdirectory获取用户所选目录的完整路径，创建一个SFTP连接，并调用get方法来下载文件，调用完关闭SFTP连接，弹窗提示用户下载成功。

代码展示如下：

def download\_file(self):  
 #tk.simpledialog.askstring 的主要用途是获取用户输入的文本信息  
 remote\_file = tk.simpledialog.askstring("输入文件名", "请输入要下载的文件名:")  
 if remote\_file is None: # 用户点击“取消”或关闭对话框  
 return  
 elif not remote\_file: # 用户输入了空字符串  
 messagebox.showerror("错误", "请输入文件名！")  
 return  
 #filedialog.askdirectory返回的是用户所选目录的完整路径，而不是文件名  
 local\_path = filedialog.askdirectory(title="选择下载到的本地目录")  
 if local\_path:  
 try:  
 sftp = self.ssh.open\_sftp()  
 local\_file\_path = os.path.join(local\_path, remote\_file)  
 sftp.get(remote\_file, local\_file\_path)  
 sftp.close()  
 messagebox.showinfo("下载成功", f"文件下载成功: {remote\_file} -> {local\_path}")  
 except Exception as e:  
 messagebox.showerror("下载失败", f"文件下载失败: {e}")

#下载文件  
def downloader(self):  
 if not self.mssh.connected:  
 messagebox.showerror("错误", "请先连接到服务器!")  
 return  
 self.mssh.download\_file()

2.9 服务器监控功能设计与实现

通过SSH连接到远程服务器，并在界面上展示服务器的进程信息、系统资源利用情况和磁盘使用情况，同时处理连接状态的变化。

（1）进程展示的实现

创建get\_process方法通过执行"ps -ef"命令获取进程信息并展示在界面上的表格中。如果连接失败或执行命令出错，会通过弹出窗口显示错误信息。 代码展示如下：

def get\_process(self):  
 if not self.mssh.connected:  
 messagebox.showerror("错误", "请先连接到服务器!")  
 return  
  
 self.pop\_win = tk.Toplevel(self.root)  
 self.pop\_win.title("进程")  
 self.pop\_win.geometry("800x600+600+250")  
  
 table\_frame = ttk.Frame(self.pop\_win)  
 table\_frame.pack(padx=10, pady=10, fill="both", expand=True)  
  
 server = self.mssh.server  
 ip = server['ip']  
 username = server['username']  
 password = server['password']  
  
 try:  
 client = paramiko.SSHClient()  
 client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())  
 client.connect(ip, 22, username, password)  
 stdin, stdout, stderr = client.exec\_command("ps -ef")  
 result = stdout.read().decode("utf-8").strip().split("\n")  
  
 # 创建表格  
 table = ttk.Treeview(table\_frame, columns=(  
 "user", "pid", "cpu", "mem", "vsz", "rss", "tty", "stat", "start", "time", "command"), show="headings")  
 table.heading("user", text="用户")  
 table.heading("pid", text="进程ID")  
 table.heading("cpu", text="CPU%")  
 table.heading("mem", text="内存%")  
 table.heading("vsz", text="虚拟内存")  
 table.heading("rss", text="常驻内存")  
 table.heading("tty", text="终端")  
 table.heading("stat", text="状态")  
 table.heading("start", text="启动时间")  
 table.heading("time", text="CPU时间")  
 table.heading("command", text="命令")  
 table.pack(side="left", fill="both", expand=True)  
  
 # 添加滚动条  
 scrollbar = ttk.Scrollbar(table\_frame, orient="vertical", command=table.yview)  
 scrollbar.pack(side="right", fill="y")  
 table.configure(yscrollcommand=scrollbar.set)  
  
 # 填充表格数据  
 for line in result[1:]:  
 parts = line.split()  
 if len(parts) >= 10:  
 user, pid, cpu, mem, vsz, rss, tty, stat, start, time, \*command = parts  
 command = " ".join(command)  
 table.insert("", "end", values=(user, pid, cpu, mem, vsz, rss, tty, stat, start, time, command))  
  
 # 调整列宽度  
 table.column("#0", width=0, stretch="no") # 隐藏第一列  
 table.column("user", width=80, anchor="w")  
 table.column("pid", width=60, anchor="e")  
 table.column("cpu", width=60, anchor="e")  
 table.column("mem", width=60, anchor="e")  
 table.column("vsz", width=100, anchor="e")  
 table.column("rss", width=100, anchor="e")  
 table.column("tty", width=60, anchor="w")  
 table.column("stat", width=60, anchor="w")  
 table.column("start", width=120, anchor="w")  
 table.column("time", width=80, anchor="e")  
 table.column("command", width=300, anchor="w")  
  
 # 自动调整行高  
 def fixed\_map(option):  
 return [elm for elm in style.map("Treeview", query\_opt=option) if elm[:2] != ("!disabled", "!selected")]  
  
 style = ttk.Style()  
 style.map("Treeview", foreground=fixed\_map("foreground"), background=fixed\_map("background"))  
  
 except paramiko.AuthenticationException:  
 messagebox.showerror("错误", "认证失败,请验证您的凭据")  
 except paramiko.SSHException as sshException:  
 messagebox.showerror("错误", f"无法建立SSH连接: {sshException}")  
 except paramiko.BadHostKeyException as badHostKeyException:  
 messagebox.showerror("错误", f"无法验证服务器的主机密钥: {badHostKeyException}")  
 except Exception as e:  
 messagebox.showerror("错误", f"发生错误: {e}")  
 finally:  
 if 'client' in locals() and client:  
 client.close() # 确保连接被关闭  
  
 # 当窗口关闭时,确保没有遗留的引用或连接  
 self.pop\_win.protocol("WM\_DELETE\_WINDOW", self.on\_closing)

（2）CPU、内存、网络等信息展示的实现

创建create\_widgets方法制作界面上的标签、进度条和按钮。用于展示CPU、内存、网络等信息，并提供查看磁盘使用情况的按钮。使用update\_info方法用于在界面上更新系统信息，包括CPU利用率、内存使用情况和网络速率。reset\_system\_monitoring方法重置系统监控状态，将CPU信息、内存信息和网络信息重置为初始状态。show\_disk\_usage展示磁盘使用情况，并创建一个新窗口展示磁盘信息。

代码展示如下：

def create\_widgets(self, frame):  
 self.cpu\_label = ttk.Label(frame, text="CPU信息：")  
 self.cpu\_label.grid()  
  
 self.cpu\_progress = ttk.Progressbar(frame, orient='horizontal', length=200, mode='determinate')  
 self.cpu\_progress.grid(pady=(0, 10))  
  
 self.mem\_label = ttk.Label(frame, text="Memory Usage: 0 MB / 0 MB")  
 self.mem\_label.grid()  
  
 self.mem\_progress = ttk.Progressbar(frame, orient='horizontal', length=200, mode='determinate')  
 self.mem\_progress.grid(pady=(0, 5))  
  
 self.net\_label = ttk.Label(frame, text="网络信息：")  
 self.net\_label.grid(pady=5)  
  
 self.disk\_button = tk.Button(frame, text="查看磁盘使用情况", command=self.show\_disk\_usage)  
 self.disk\_button.grid(pady=(5, 5))

def update\_info(self, frame):  
 self.create\_widgets(frame) # 创建标签  
 # 定义进度条样式  
 style = ttk.Style()  
 style.theme\_use("default")  
 style.configure("green.Horizontal.TProgressbar", foreground='green', background='green')  
 style.configure("yellow.Horizontal.TProgressbar", foreground='yellow', background='yellow')  
 style.configure("red.Horizontal.TProgressbar", foreground='red', background='red')  
 prev\_connected = self.mssh.connected # 记录上一次的连接状态  
 while True:  
 if "ip" in self.mssh.server:  
 if self.mssh.connected == None:  
 result\_connect = (f"服务器{self.mssh.server['ip']}连接中", "red")  
 self.reset\_system\_monitoring() # 重置系统监控状态  
  
 elif self.mssh.connected:  
 result\_connect = (f"服务器{self.mssh.server['ip']}连接成功", "green")  
 # -----------------------------系统信息------------------------------------------------  
 server = self.mssh.server  
 ip = server['ip']  
 username = server['username']  
 password = server['password']  
 self.client = paramiko.SSHClient()  
 self.client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())  
 self.client.connect(ip, 22, username, password)  
  
 # 执行远程命令获取内存信息  
 stdin, stdout, stderr = self.client.exec\_command("free -m")  
 result = stdout.read().decode('utf-8')  
 # 获取系统内存信息  
 lines = result.split('\n')  
 mem\_info = lines[1].split()  
 total\_mem = int(mem\_info[1])  
 used\_mem = int(mem\_info[2])  
 mem\_usage = used\_mem / total\_mem \* 100  
  
 # 获取网络速率  
 stdin1, stdout1, stderr1 = self.client.exec\_command("ip a")  
 result = stdout1.read().decode("utf-8")  
 # 使用正则表达式提取数据包和字节数信息  
 matches = re.findall(r'ens160.\*?RX packets (\d+) bytes (\d+).\*?TX packets (\d+) bytes (\d+)',  
 result, re.DOTALL)  
 if matches:  
 rx\_packets, rx\_bytes, tx\_packets, tx\_bytes = matches[0]  
 else:  
 rx\_packets, rx\_bytes, tx\_packets, tx\_bytes = 0, 0, 0, 0  
 time.sleep(1)  
 stdin2, stdout2, stderr3 = self.client.exec\_command("ip a")  
 result = stdout2.read().decode("utf-8")  
 matches = re.findall(r'ens160.\*?RX packets (\d+) bytes (\d+).\*?TX packets (\d+) bytes (\d+)',  
 result, re.DOTALL)  
 if matches:  
 rx\_packets, new\_rx\_bytes, tx\_packets, new\_tx\_bytes = matches[0]  
 else:  
 rx\_packets, new\_rx\_bytes, tx\_packets, new\_tx\_bytes = 0, 0, 0, 0  
  
 # 计算速率  
 upload\_speed = ((int(new\_tx\_bytes) - int(tx\_bytes)) / 1024) # MB/s，  
 download\_speed = ((int(new\_rx\_bytes) - int(rx\_bytes)) / 1024)  
  
 # 获取cpu利用率  
 stdin4, stdout4, stderr4 = self.client.exec\_command("top -bn 1")  
 result = stdout4.read().decode("utf-8")  
 cpu = 0  
 for line in result.split("\n"):  
 if line.startswith("%Cpu(s)"):  
 cpu\_us = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[0])[0]  
 cpu\_sy = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[1])[0]  
 cpu\_ni = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[2])[0]  
 cpu\_id = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[3])[0]  
 cpu\_wa = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[4])[0]  
 cpu\_hi = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[5])[0]  
 cpu\_si = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[6])[0]  
 cpu\_st = re.findall(r'\d+\.\d+', line.split(":")[1].split(",")[7])[0]  
 sum = float(cpu\_st) + float(cpu\_si) + float(cpu\_hi) + float(cpu\_wa) + float(cpu\_id) + float(  
 cpu\_ni) + float(cpu\_sy) + float(cpu\_us)  
 cpu = sum - float(cpu\_id)  
  
 # 设置进度条样式  
 if cpu > 80:  
 cpu\_style = "red.Horizontal.TProgressbar"  
 elif cpu > 50:  
 cpu\_style = "yellow.Horizontal.TProgressbar"  
 else:  
 cpu\_style = "green.Horizontal.TProgressbar"  
  
 if mem\_usage > 80:  
 mem\_style = "red.Horizontal.TProgressbar"  
 elif mem\_usage > 50:  
 mem\_style = "yellow.Horizontal.TProgressbar"  
 else:  
 mem\_style = "green.Horizontal.TProgressbar"  
  
 # 更新标签  
 self.cpu\_label.config(text=f"CPU信息：{cpu:.2f}%")  
 self.cpu\_progress.config(style=cpu\_style, value=cpu)  
 self.mem\_label.config(text=f"Memory Usage: {used\_mem} MB / {total\_mem} MB ({mem\_usage:.2f}%)")  
 self.mem\_progress.config(style=mem\_style, value=mem\_usage)  
 self.net\_label.config(text=f"网络信息：↑{upload\_speed:.2f}MB/s \t ↓{download\_speed:.2f}MB/s")  
  
 else:  
 result\_connect = (f"服务器{self.mssh.server['ip']}连接失败", "red")  
 self.reset\_system\_monitoring() # 重置系统监控状态  
 else:  
 result\_connect = (f"服务器未选择", "black")  
 self.reset\_system\_monitoring() # 重置系统监控状态  
  
 # 检查连接状态是否发生变化  
 if self.mssh.connected != prev\_connected:  
 self.reset\_system\_monitoring() # 重置系统监控状态  
 prev\_connected = self.mssh.connected  
 self.label\_server.config(text=f"{result\_connect[0]}", foreground=result\_connect[1])  
 time.sleep(1)  
  
def reset\_system\_monitoring(self):  
 # 重置系统监控状态  
 self.cpu\_label.config(text="CPU信息：")  
 self.cpu\_progress.config(style="", value=0)  
 self.mem\_label.config(text="Memory Usage: 0 MB / 0 MB")  
 self.mem\_progress.config(style="", value=0)  
 self.net\_label.config(text="网络信息：")  
  
def show\_disk\_usage(self):  
 if not self.mssh.connected:  
 messagebox.showerror("错误", "请先连接到服务器!")  
 return  
  
 try:  
 self.mssh.connecting()  
 stdin, stdout, stderr = self.mssh.ssh.exec\_command("df -h")  
 disk\_info = stdout.read().decode("utf-8").strip().split("\n")  
  
 # 创建新窗口  
 disk\_window = tk.Toplevel(self.root)  
 disk\_window.title("磁盘使用情况")  
  
 # 创建表格  
 table\_frame = ttk.Frame(disk\_window)  
 table\_frame.pack(padx=10, pady=10, fill="both", expand=True)  
  
 # 创建表格标题  
 table = ttk.Treeview(table\_frame, columns=("filesystem", "size", "used", "avail", "use%", "mounted\_on"),  
 show="headings")  
 table.heading("filesystem", text="文件系统")  
 table.heading("size", text="总大小")  
 table.heading("used", text="已用空间")  
 table.heading("avail", text="可用空间")  
 table.heading("use%", text="已用百分比")  
 table.heading("mounted\_on", text="挂载点")  
 table.pack(side="left", fill="both", expand=True)  
  
 # 添加滚动条  
 scrollbar = ttk.Scrollbar(table\_frame, orient="vertical", command=table.yview)  
 scrollbar.pack(side="right", fill="y")  
 table.configure(yscrollcommand=scrollbar.set)  
  
 # 填充表格数据  
 for line in disk\_info[1:]:  
 parts = line.split()  
 if len(parts) >= 6:  
 filesystem, size, used, avail, use\_percent, mounted\_on = parts  
 table.insert("", "end", values=(filesystem, size, used, avail, use\_percent, mounted\_on))  
  
 # 调整列宽度  
 table.column("#0", width=0, stretch="no") # 隐藏第一列  
 table.column("filesystem", width=100, anchor="w")  
 table.column("size", width=80, anchor="e")  
 table.column("used", width=80, anchor="e")  
 table.column("avail", width=80, anchor="e")  
 table.column("use%", width=80, anchor="e")  
 table.column("mounted\_on", width=200, anchor="w")  
  
 # 自动调整行高  
 def fixed\_map(option):  
 return [elm for elm in style.map("Treeview", query\_opt=option) if elm[:2] != ("!disabled", "!selected")]  
  
 style = ttk.Style()  
 style.map("Treeview", foreground=fixed\_map("foreground"), background=fixed\_map("background"))  
  
 except Exception as e:  
 messagebox.showerror("错误", f"无法获取磁盘使用情况: {e}")

2.10 批量下载日志文件和查看日志功能设计与实现

（1）批量下载日志文件的实现

首先，检查是否已连接到远程服务器，然后设置远程目录为 /var/log。接下来，它通过SSH连接到服务器并打开SFTP客户端。然后，它检查远程目录是否存在，并获取远程目录中的文件列表。接着，它要求用户选择下载到的本地目录，并将选择的本地目录设置为类属性 local\_log\_dir。然后，它遍历远程目录中的文件，筛选以 .log 结尾的文件，并将这些文件下载到本地目录。最后，它关闭SFTP连接，并显示成功下载的文件数量。 代码展示如下：

def download\_logs(self):  
 if not self.mssh.connected:  
 messagebox.showerror("错误", "请先连接到服务器!")  
 return  
  
 # 将远程目录设置为 /var/log  
 remote\_dir = "/var/log"  
  
 try:  
 self.mssh.connecting()  
 sftp = self.mssh.ssh.open\_sftp()  
  
 # 检查远程目录是否存在  
 try:  
 sftp.stat(remote\_dir)  
 except IOError as e:  
 messagebox.showerror("错误", f"远程目录 {remote\_dir} 不存在")  
 return  
  
 remote\_files = sftp.listdir(remote\_dir)  
  
 local\_dir = filedialog.askdirectory(title="选择下载到的本地目录")  
 if local\_dir:  
 self.local\_log\_dir = local\_dir  
 else:  
 return  
  
 downloaded\_count = 0  
 for filename in remote\_files:  
 # 检查文件是否以 ".log" 结尾  
 if filename.endswith(".log"):  
 remote\_path = f"{remote\_dir}/{filename}"  
 local\_path = f"{local\_dir}/{filename}"  
 sftp.get(remote\_path, local\_path)  
 downloaded\_count += 1  
 print(f"下载成功: {remote\_path} -> {local\_path}")  
  
 sftp.close()  
 print(f"共下载了 {downloaded\_count} 个文件")  
 if downloaded\_count > 0:  
 messagebox.showinfo("成功", f"成功下载了 {downloaded\_count} 个日志文件")  
 except Exception as e:  
 print(f"下载失败: {e}")  
 messagebox.showerror("错误", f"下载日志文件失败: {e}")

（2）查看日志功能的实现 先检查是否已连接到远程服务器并下载了日志文件。接着，它尝试从 local\_log\_dir 属性获取之前选择的本地目录，如果该属性不存在，则要求用户选择下载日志文件所在的目录。然后，它要求用户选择要查看的日志文件，并使用VSCode打开选定的日志文件。

代码展示如下：

def view\_log(self):  
 if not self.mssh.connected:  
 messagebox.showerror("错误", "请先连接服务器并下载日志文件!")  
 return  
  
 # 从 download\_logs 函数中获取之前选择的本地目录  
 try:  
 local\_dir = self.local\_log\_dir  
 except AttributeError:  
 local\_dir = filedialog.askdirectory(title="选择下载日志文件所在的目录")  
 if not local\_dir:  
 return  
  
 log\_file = filedialog.askopenfilename(initialdir=local\_dir, title="选择要查看的日志文件",  
 filetypes=(("Log Files", "\*.log"),))  
 if log\_file:  
 try:  
 # 使用 VSCode 打开日志文件  
 subprocess.run(['Code', log\_file], shell=True)  
 except Exception as e:  
 messagebox.showerror("错误", f"无法打开文件: {e}")

2.11界面窗口的实现

创建一个具有菜单、标签、输入框和文本框的用户界面(UI)，以及一个后台线程来更新系统信息。同时，通过不同的菜单项和按钮，用户可以执行连接、上传文件、下载文件、查看进程、扩展功能等操作。 代码展示如下：

def \_\_init\_\_(self, ssh):  
 print("初始化")  
 self.is\_connected = False  
 self.mssh = ssh  
  
 # 主应用程序窗口  
 self.root = tk.Tk()  
 frame = ttk.Frame(self.root)  
 self.root.geometry("800x550")  
  
 # 设置窗口标题（可选）  
 self.root.title("ssh 暗影坤手")  
  
 # 使用 grid 布局  
 self.root.grid\_rowconfigure(0, weight=1)  
 self.root.grid\_columnconfigure(0, weight=1)  
  
 # 布局，网格布局  
 # 创建一个框架容器  
 frame = ttk.Frame(self.root)  
 frame.grid(sticky="nsew")  
  
 # 添加菜单项 连接：新建连接 打开连接 操作：-- 帮助：--  
 menu\_top = tk.Menu(self.root)  
  
 # 连接菜单  
 menu\_link = tk.Menu(menu\_top)  
 menu\_top.add\_cascade(label="连接", menu=menu\_link)  
 menu\_link.add\_command(label="新建连接", command=self.new\_connect\_ui)  
 menu\_link.add\_command(label="打开连接", command=self.open\_connect\_ui)  
  
 # 操作菜单  
 menu\_func = tk.Menu(menu\_top)  
 menu\_top.add\_cascade(label="操作", menu=menu\_func)  
 menu\_func.add\_command(label="上传文件", command=self.upload\_file)  
 menu\_func.add\_command(label="下载文件", command=self.downloader)  
 menu\_func.add\_command(label="vim编辑文件", command=self.vim\_edit)  
  
 # 帮助菜单  
 menu\_help = tk.Menu(menu\_top)  
 menu\_top.add\_cascade(label="帮助", menu=menu\_help)  
 menu\_help.add\_command(label="查看进程", command=self.get\_process)  
 # menu\_help.add\_command(label="查看磁盘使用情况", command=self.view\_disk\_usage)  
  
 # 扩展菜单  
 menu\_ext = tk.Menu(menu\_top)  
 menu\_top.add\_cascade(label="扩展", menu=menu\_ext)  
 menu\_ext.add\_command(label="批量下载日志文件", command=self.download\_logs)  
 menu\_ext.add\_command(label="查看日志文件", command=self.view\_log)  
  
 # 展示信息的label  
 self.label\_server = ttk.Label(frame, text="服务器信息：")  
 self.label\_server.grid(row=0, column=0, sticky="w", padx=10, pady=5)  
  
 # 输入指令框  
 self.command = tk.StringVar()  
 entry\_command = ttk.Entry(frame, width=60, textvariable=self.command)  
 entry\_command.grid(row=1, column=0, sticky="ew", padx=10, pady=5)  
  
 # 执行指令的按钮  
 entry\_command.bind("<Return>", self.ok) # 当按下Enter键时执行命令  
  
 # 展示结果  
 self.txt\_result = tk.Text(frame)  
 self.txt\_result.grid(row=2, column=0, sticky="nsew", padx=10, pady=5)  
 self.root.config(menu=menu\_top)  
  
 # 文本框背景/前景颜色  
 self.txt\_result.config(bg="#262626", fg="#D9D9D9")  
 self.root.config(menu=menu\_top)  
  
 # 导入 ttk 后  
 style = ttk.Style()  
  
 # 设置 ttk 主题  
 style.theme\_use("clam")  
  
 # 定制按钮样式  
 style.configure("TButton", foreground="white", background="#404040")  
 style.map("TButton",  
 foreground=[("active", " #CCCCCC"), ("disabled", "#666666")],  
 background=[("active", "#404040"), ("disabled", "#262626")])  
  
 # 让文本框自动扩展以填充剩余空间  
 frame.grid\_rowconfigure(2, weight=1)  
 frame.grid\_columnconfigure(0, weight=1)  
 frame.grid\_rowconfigure(3, weight=0) # 信息监控面板不需要扩展  
  
 # thread\_info=threading.Thread(target=self.update\_info)  
 thread\_info = threading.Thread(target=self.update\_info, args=(frame,))  
 thread\_info.start()  
  
 # 将主应用程序挂起  
 self.root.protocol("WM\_DELETE\_WINDOW", self.on\_closing\_main)  
 self.root.mainloop()

2.12 打包成可执行文件的实现

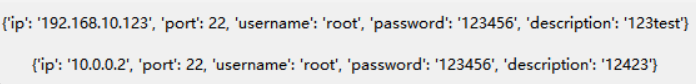
PyInstaller将Python程序打包成独立的可执行文件，包括所有必要的依赖项。

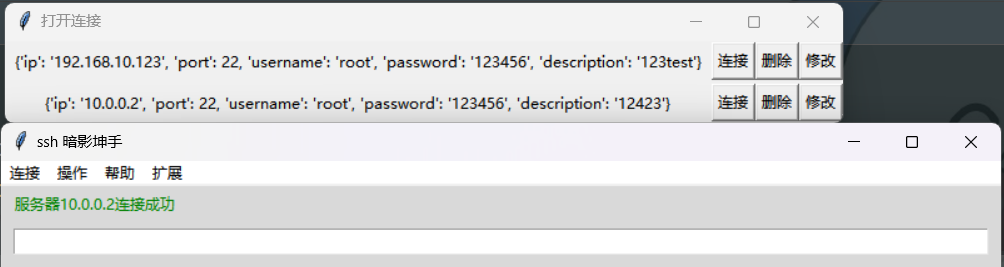
在终端执行以下命令：

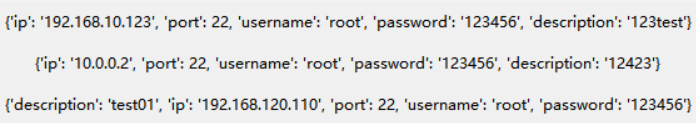
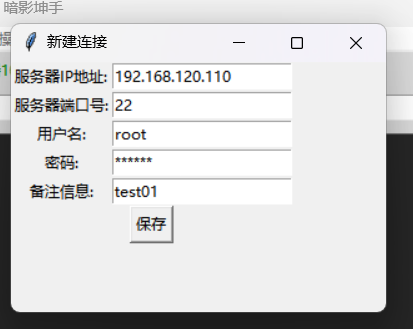
pip install pyinstaller

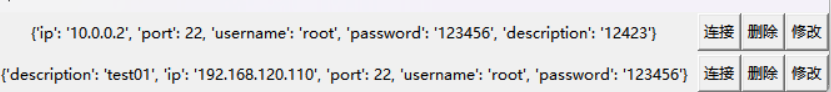
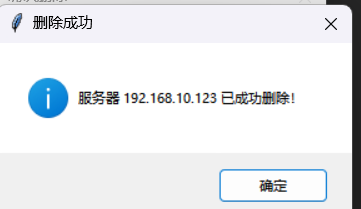
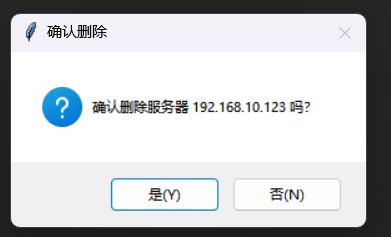
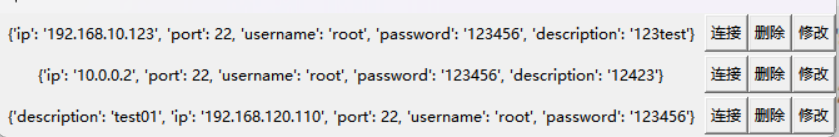
Pyinstaller -F -w -i ssh.ico -p D:\python\Lib\site-packages --onefile --add-data "server.json;./" demotest.py

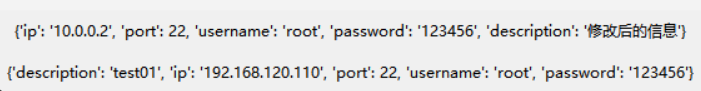
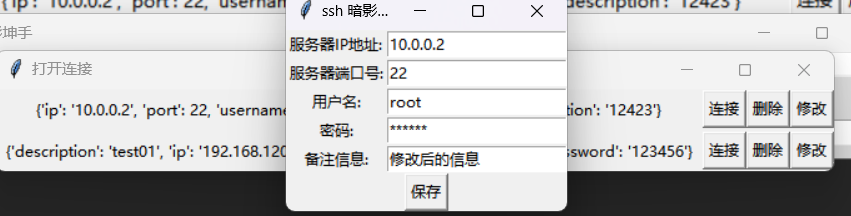
3 系统完成效果

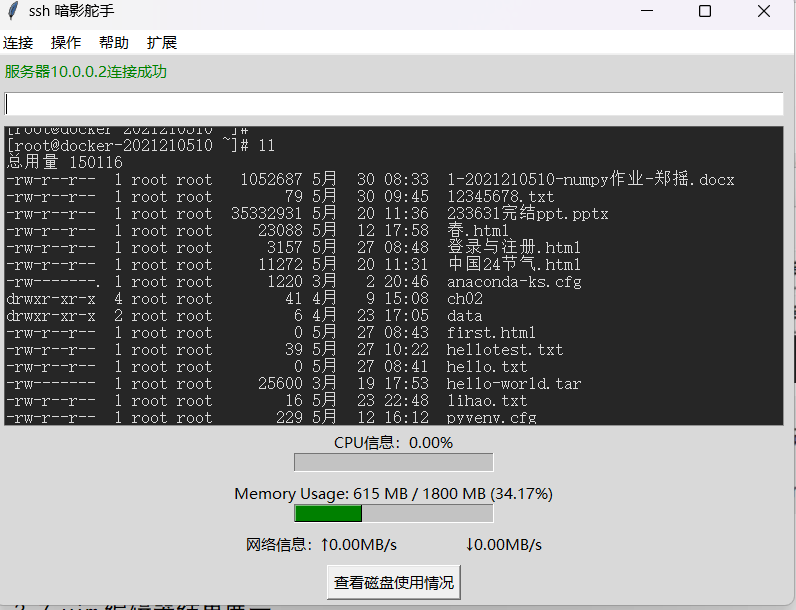
3.1 服务器信息结果展示

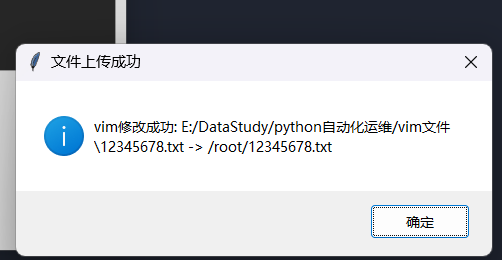
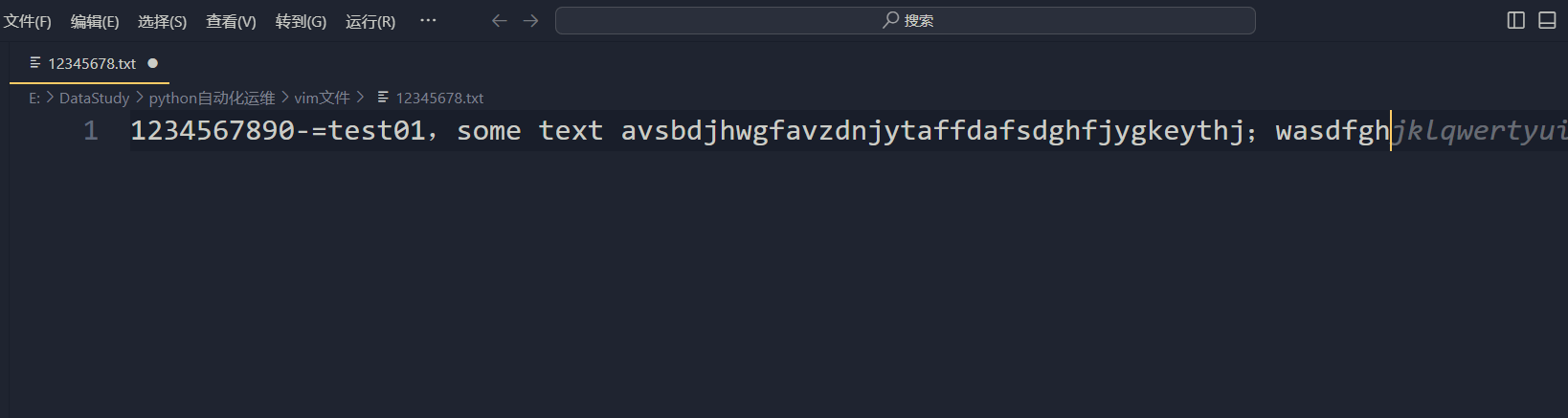
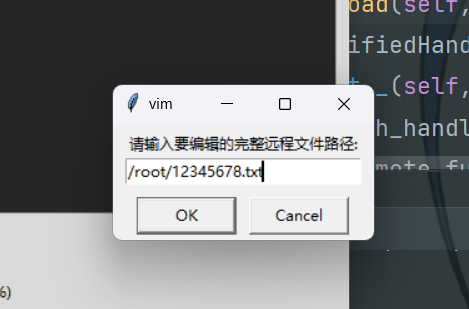
3.2 连接服务器结果展示

3.3 添加服务器信息结果展示

3.4 删除服务器信息结果展示

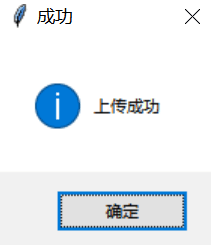
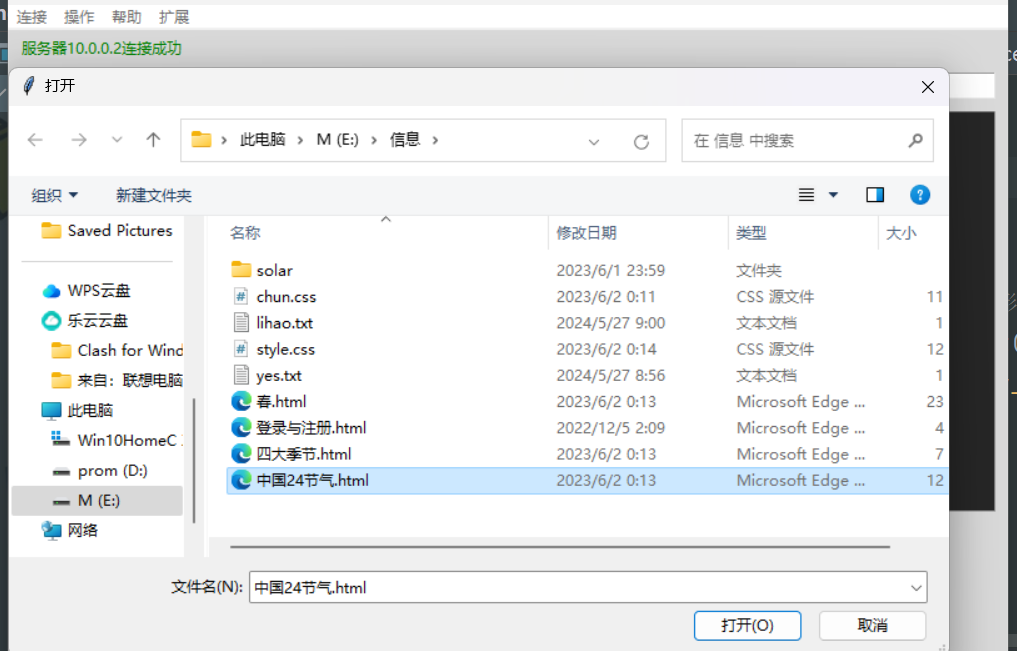
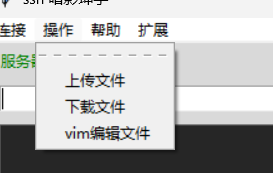
3.5 修改服务器信息结果展示

3.6 执行交互式交互命令结果展示

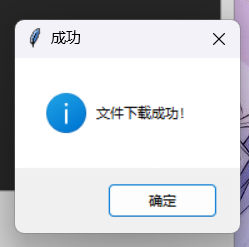
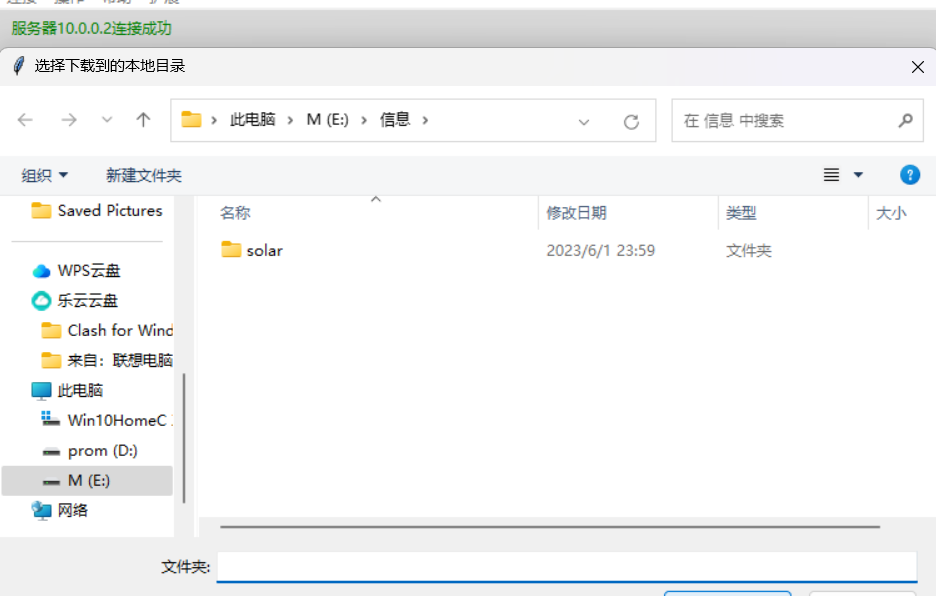
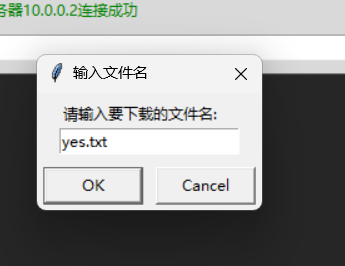
3.7 vim编辑器结果展示

3.8 文件上传下载结果展示

（1）上传文件



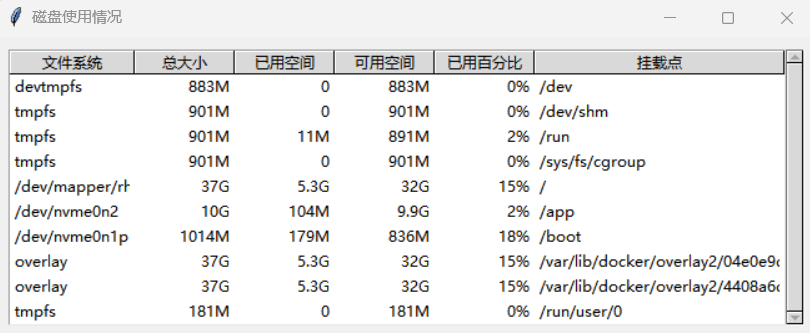
（2）下载文件

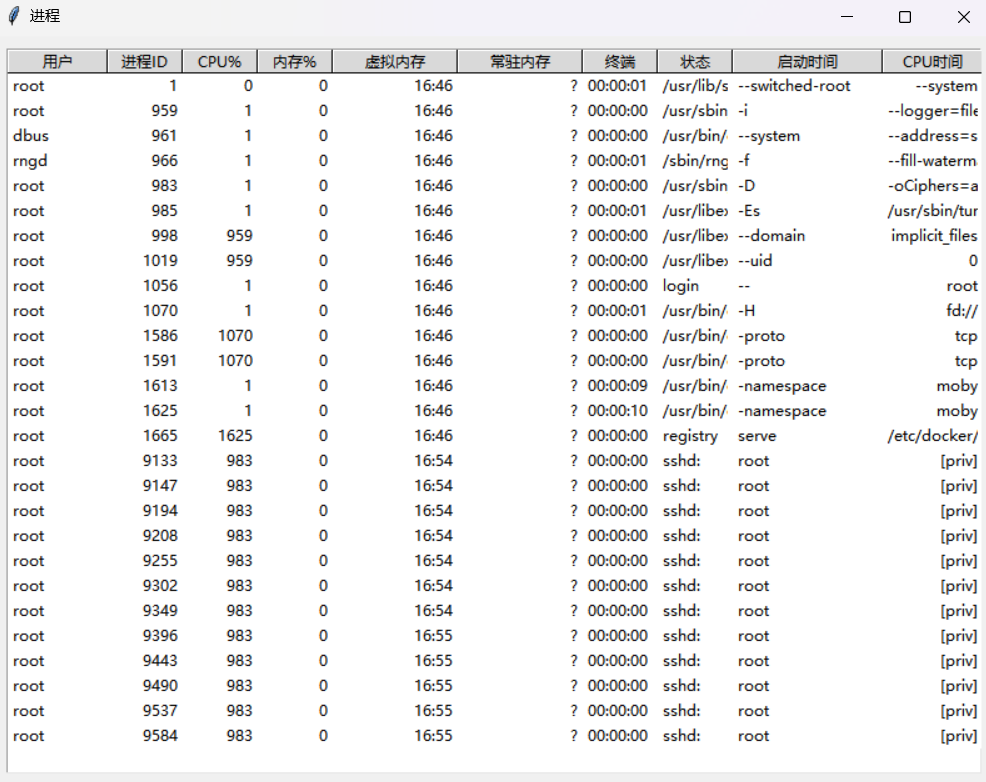


3.9 服务器监控功能结果展示

（1）cpu、内存、网络信息展示

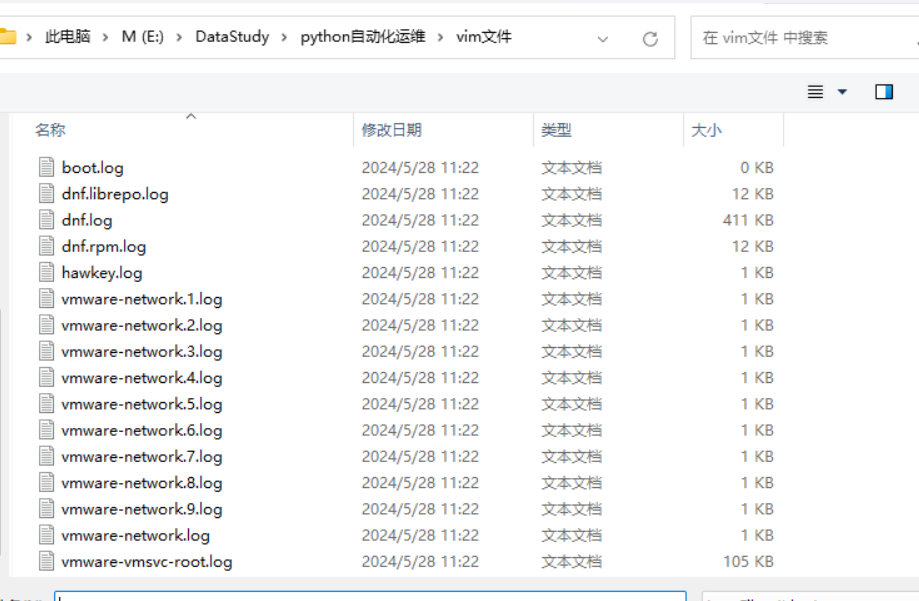
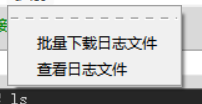


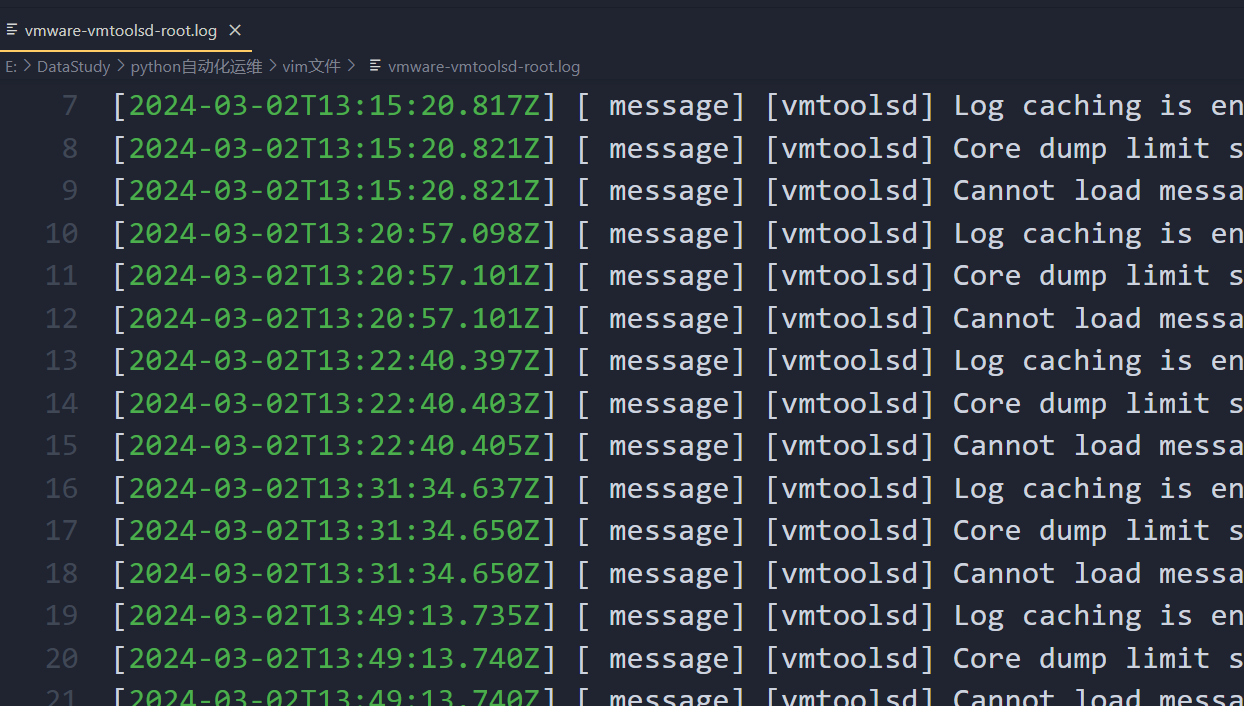
（2）磁盘使用情况展示

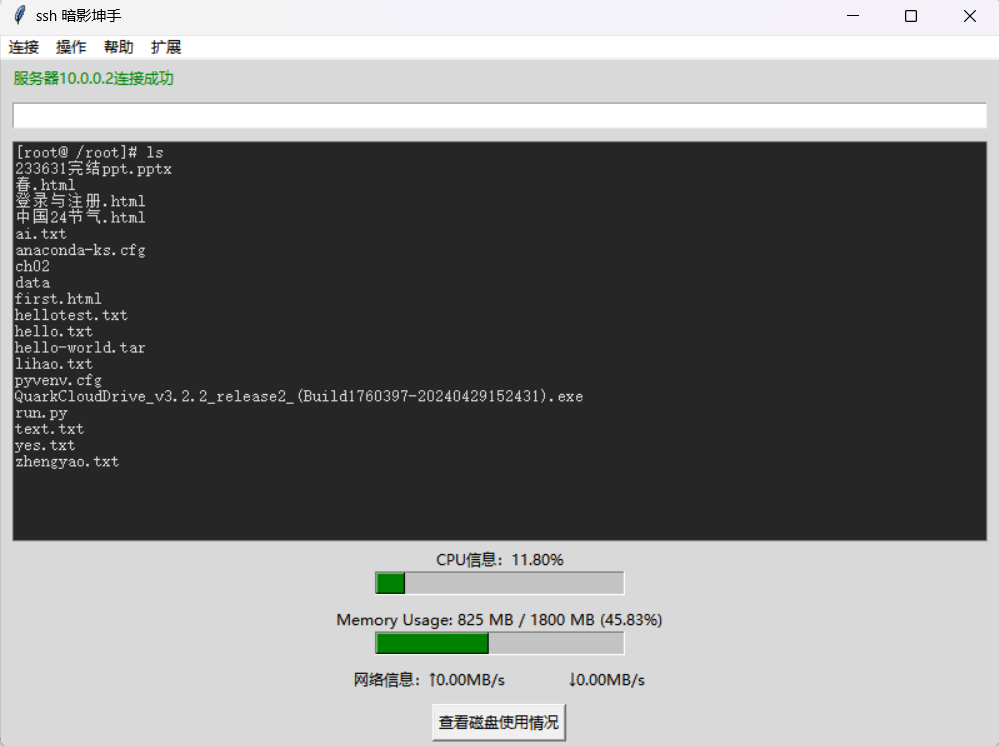
（3）进程信息

3.10 批量下载日志文件和查看日志文件结果展示

（1）批量下载文件展示



（2）查看日志信息

3.11窗口界面结果展示

3.12 打包成可执行文件的结果展示