import tkinter as tk

from tkinter import ttk, filedialog, messagebox, scrolledtext

import subprocess

import os

import platform

import json

from datetime import datetime

import time

import queue

import uuid

import threading

class MainApplication(tk.Tk):

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

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

self.title("开发者工具箱")

self.geometry("525x500")

self.topmost\_state = tk.BooleanVar(value=False)

self.attributes('-topmost', self.topmost\_state.get())

self.status\_bar = StatusBar(self, self.topmost\_state)

self.status\_bar.pack(side="top", fill="x", pady=5)

self.content\_frame = ttk.Frame(self)

self.content\_frame.pack(fill="both", expand=True)

self.frames = {}

for F in (MainMenu, ADBTools, FastbootTools, LogTools):

frame = F(self.content\_frame, self)

self.frames[F.\_\_name\_\_] = frame

frame.grid(row=0, column=0, sticky="nsew")

self.show\_frame("MainMenu")

def show\_frame(self, page\_name):

frame = self.frames[page\_name]

frame.tkraise()

def run\_scrcpy(self):

script\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))

scrcpy\_dir = os.path.join(script\_dir, 'scrcpy')

scrcpy\_exe = os.path.join(scrcpy\_dir, 'scrcpy.exe')

if not os.path.exists(scrcpy\_exe):

messagebox.showerror("程序缺失", f"未找到投屏程序：\n{scrcpy\_exe}")

return

default\_adb = os.path.join(scrcpy\_dir, 'adb.exe')

adb\_path = default\_adb if os.path.exists(default\_adb) else 'adb'

try:

result = subprocess.run(

[adb\_path, 'devices'],

capture\_output=True,

text=True,

check=True

)

devices = [

line.split('\t')[0]

for line in result.stdout.splitlines()[1:]

if '\tdevice' in line

]

if not devices:

messagebox.showerror("设备未连接", "未检测到安卓设备！")

return

except (subprocess.CalledProcessError, FileNotFoundError) as e:

messagebox.showerror(

"ADB错误",

"ADB执行失败，请确保：\n1. 已安装ADB驱动\n2. 已开启USB调试\n3. 设备已授权"

)

return

try:

subprocess.Popen(

[scrcpy\_exe],

cwd=scrcpy\_dir,

creationflags=subprocess.CREATE\_NO\_WINDOW

)

messagebox.showinfo(

"投屏启动",

"投屏程序已启动，请查看设备授权提示！\n\n若窗口未出现，请检查杀毒软件拦截"

)

except Exception as e:

messagebox.showerror("启动失败", f"投屏程序启动失败：\n{str(e)}")

class StatusBar(ttk.Frame):

def \_\_init\_\_(self, parent, topmost\_var):

super().\_\_init\_\_(parent, style='Status.TFrame')

self.topmost\_var = topmost\_var

self.adb\_status = tk.StringVar(value="N")

self.fastboot\_status = tk.StringVar(value="N")

self.create\_widgets()

self.start\_auto\_check()

def create\_widgets(self):

status\_frame = ttk.Frame(self)

status\_frame.pack(side="left", padx=10)

ttk.Label(status\_frame, text="ADB连接状态:").grid(row=0, column=0)

self.lbl\_adb = ttk.Label(status\_frame, textvariable=self.adb\_status,

font=('Arial', 12, 'bold'), foreground="gray")

self.lbl\_adb.grid(row=0, column=1, padx=5)

ttk.Label(status\_frame, text="Fastboot连接状态:").grid(row=0, column=2)

self.lbl\_fastboot = ttk.Label(status\_frame, textvariable=self.fastboot\_status,

font=('Arial', 12, 'bold'), foreground="gray")

self.lbl\_fastboot.grid(row=0, column=3, padx=5)

btn\_frame = ttk.Frame(self)

btn\_frame.pack(side="right", padx=10)

ttk.Button(btn\_frame, text="重启设备",

command=self.reboot\_device).grid(row=0, column=1, padx=4)

topmost\_btn = ttk.Checkbutton(

btn\_frame,

text="置顶",

variable=self.topmost\_var,

command=self.toggle\_topmost

).grid(row=0, column=2, padx=4)

ttk.Button(btn\_frame, text="设备管理器",

command=self.open\_device\_manager).grid(row=0, column=0, padx=4)

def reboot\_device(self):

"""执行adb reboot命令"""

try:

if self.adb\_status.get() != "Y":

messagebox.showwarning("警告", "当前没有已连接的ADB设备")

return

def run\_reboot():

try:

result = subprocess.run(

["adb", "reboot"],

check=True,

stdout=subprocess.PIPE,

stderr=subprocess.PIPE,

text=True,

timeout=10

)

self.master.after(0, lambda: messagebox.showinfo(

"成功", "重启命令已发送，设备即将重启"))

except subprocess.CalledProcessError as e:

error\_msg = f"重启失败：{e.stderr.strip()}"

self.master.after(0, lambda: messagebox.showerror("错误", error\_msg))

except Exception as e:

self.master.after(0, lambda: messagebox.showerror(

"异常", f"发生未知错误：{str(e)}"))

threading.Thread(target=run\_reboot, daemon=True).start()

except Exception as e:

messagebox.showerror("错误", f"执行异常：{str(e)}")

def toggle\_topmost(self):

"""切换窗口置顶状态"""

new\_state = self.topmost\_var.get()

self.master.attributes('-topmost', new\_state)

def start\_auto\_check(self):

self.check\_status()

self.after(2000, self.start\_auto\_check)

def check\_status(self):

self.check\_adb()

self.check\_fastboot()

self.update\_colors()

def check\_adb(self):

try:

result = subprocess.run(

["adb", "devices"],

capture\_output=True,

text=True,

timeout=2

)

connected = "Y" if self.has\_connected\_device(result.stdout, 'adb') else "N"

self.adb\_status.set(connected)

except Exception as e:

print(f"ADB检测异常: {str(e)}")

self.adb\_status.set("N")

def check\_fastboot(self):

try:

result = subprocess.run(

["fastboot", "devices"],

capture\_output=True,

text=True,

timeout=2

)

connected = "Y" if self.has\_connected\_device(result.stdout, 'fastboot') else "N"

self.fastboot\_status.set(connected)

except Exception as e:

print(f"Fastboot检测异常: {str(e)}")

self.fastboot\_status.set("N")

def has\_connected\_device(self, output, mode):

valid\_lines = []

for line in output.splitlines():

line = line.strip()

if not line:

continue

if mode == 'adb' and "List of devices" in line:

continue

if ("\t" in line) or (" " in line and "fastboot" in line):

valid\_lines.append(line)

return len(valid\_lines) > 0

def update\_colors(self):

self.lbl\_adb.config(foreground="green4" if self.adb\_status.get() == "Y" else "red3")

self.lbl\_fastboot.config(foreground="green4" if self.fastboot\_status.get() == "Y" else "red3")

def open\_device\_manager(self):

system = platform.system()

try:

if system == "Windows":

os.startfile("devmgmt.msc")

elif system == "Darwin":

subprocess.run(["open", "/System/Library/CoreServices/Applications/System Information.app"])

elif system == "Linux":

subprocess.run(["gnome-control-center", "devices"])

except Exception as e:

messagebox.showerror("错误", f"无法打开设备管理器：{str(e)}")

class MainMenu(ttk.Frame):

def \_\_init\_\_(self, parent, controller):

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

self.create\_widgets(controller)

def create\_widgets(self, controller):

ttk.Label(self, text="主菜单", font=("微软雅黑", 14)).pack(pady=20)

btn\_frame = ttk.Frame(self)

btn\_frame.pack(expand=True)

buttons = [

("Adb Flash", "ADBTools"),

("Fastboot flash", "FastbootTools"),

("日志过滤", "LogTools"),

("投屏", "run\_scrcpy")

]

for text, page in buttons:

if page == "run\_scrcpy":

ttk.Button(btn\_frame, text=text, width=20,

command=controller.run\_scrcpy).pack(pady=5)

else:

ttk.Button(btn\_frame, text=text, width=20,

command=lambda p=page: controller.show\_frame(p)).pack(pady=5)

class ADBTools(ttk.Frame):

def \_\_init\_\_(self, parent, controller):

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

self.controller = controller

self.history\_file = "file\_history.json"

self.target\_history\_file = "target\_history.json"

self.file\_history = []

self.target\_history = []

self.current\_file = ""

self.current\_target = ""

self.create\_header()

self.setup\_ui()

self.load\_histories()

self.check\_environment()

def create\_header(self):

"""创建标题和返回按钮"""

header\_frame = ttk.Frame(self)

header\_frame.grid(row=0, column=0, sticky="ew", pady=5)

ttk.Label(header\_frame, text="ADB 单刷工具", font=("微软雅黑", 12)).pack(side="left", padx=10)

ttk.Button(header\_frame, text="返回主菜单",

command=lambda: self.controller.show\_frame("MainMenu")).pack(side="right", padx=10)

self.columnconfigure(0, weight=1)

def setup\_ui(self):

self.env\_frame = ttk.LabelFrame(self, text="环境检测")

self.env\_frame.grid(row=1, column=0, padx=10, pady=5, sticky="ew")

self.env\_status = ttk.Label(self.env\_frame, text="未检测")

self.env\_status.grid(row=0, column=0, padx=5, pady=2)

ttk.Button(self.env\_frame, text="检测环境", command=self.check\_environment).grid(row=0, column=1, padx=5)

ttk.Button(self.env\_frame, text="配置指南", command=self.install\_drivers).grid(row=0, column=2, padx=5)

self.file\_frame = ttk.LabelFrame(self, text="操作文件")

self.file\_frame.grid(row=2, column=0, padx=10, pady=5, sticky="ew")

self.file\_info = ttk.Label(self.file\_frame, text="未选择文件")

self.file\_info.grid(row=0, column=0, padx=5, pady=2, sticky="w")

self.history\_combo = ttk.Combobox(self.file\_frame, values=self.file\_history)

self.history\_combo.grid(row=1, column=0, padx=10, pady=2, sticky="ew")

self.history\_combo.bind("<<ComboboxSelected>>", self.on\_file\_history\_select)

ttk.Button(self.file\_frame, text="选择文件", command=self.select\_file).grid(row=1, column=1, padx=5)

self.target\_frame = ttk.LabelFrame(self, text="目标路径")

self.target\_frame.grid(row=3, column=0, padx=10, pady=5, sticky="ew")

self.target\_combo = ttk.Combobox(self.target\_frame, values=self.target\_history)

self.target\_combo.grid(row=0, column=0, padx=5, pady=2, sticky="ew")

self.target\_combo.bind("<<ComboboxSelected>>", self.on\_target\_select)

ttk.Button(self.target\_frame, text="开始刷写", command=self.start\_flash).grid(row=0, column=1, padx=5)

self.target\_frame.columnconfigure(0, weight=1)

self.output\_frame = ttk.LabelFrame(self, text="操作输出")

self.output\_frame.grid(row=4, column=0, padx=10, pady=5, sticky="nsew")

self.output\_text = tk.Text(self.output\_frame, height=15, width=70, wrap=tk.WORD)

vsb = ttk.Scrollbar(self.output\_frame, orient="vertical", command=self.output\_text.yview)

self.output\_text.configure(yscrollcommand=vsb.set)

self.output\_text.grid(row=0, column=0, sticky="nsew")

vsb.grid(row=0, column=1, sticky="ns")

self.rowconfigure(3, weight=1)

self.columnconfigure(0, weight=1)

self.output\_frame.rowconfigure(0, weight=1)

self.output\_frame.columnconfigure(0, weight=1)

def check\_environment(self):

"""检测ADB环境"""

self.insert\_output("\n=== 开始环境检测 ===\n")

try:

subprocess.run(["adb", "version"],

check=True,

stdout=subprocess.DEVNULL,

stderr=subprocess.DEVNULL,

creationflags=subprocess.CREATE\_NO\_WINDOW)

self.env\_status.config(text="环境正常", foreground="green")

self.insert\_output("检测结果：环境正常\n")

return True

except Exception as e:

self.env\_status.config(text="环境异常", foreground="red")

self.insert\_output("检测结果：环境异常\n")

self.insert\_output(f"错误信息：{str(e)}\n")

return False

def install\_drivers(self):

"""显示驱动安装指南"""

self.insert\_output("\n=== ADB环境配置指南 ===\n")

guide = """1. 下载Google USB驱动：

官方地址：https://developer.android.com/studio/run/win-usb

2. 下载Platform Tools：

下载页面：https://developer.android.com/studio/releases/platform-tools

3. 解压platform-tools到系统目录（例如：C:\\platform-tools）

4. 添加环境变量：

• 右键点击"此电脑"选择"属性"

• 进入"高级系统设置" -> "环境变量"

• 在Path中添加platform-tools目录

5. 设备连接：

• 启用USB调试模式（设置 -> 开发者选项）

• 首次连接需要在设备上授权ADB调试"""

self.insert\_output(guide + "\n")

def select\_file(self):

file\_path = filedialog.askopenfilename(filetypes=[("All Files", "\*.\*")])

if file\_path:

self.current\_file = file\_path

self.update\_file\_info()

self.add\_to\_file\_history(file\_path)

def select\_target\_dir(self):

dir\_path = filedialog.askdirectory()

if dir\_path:

self.current\_target = dir\_path

self.target\_combo.set(dir\_path)

self.add\_to\_target\_history(dir\_path)

def on\_target\_select(self, event):

self.current\_target = self.target\_combo.get()

def add\_to\_target\_history(self, path):

if path in self.target\_history:

self.target\_history.remove(path)

self.target\_history.insert(0, path)

self.target\_combo["values"] = self.target\_history[:10]

self.save\_history(self.target\_history\_file, self.target\_history)

def update\_file\_info(self):

if self.current\_file:

filename = os.path.basename(self.current\_file)

mtime = os.path.getmtime(self.current\_file)

timestamp = datetime.fromtimestamp(mtime).strftime("%Y-%m-%d %H:%M:%S")

self.file\_info.config(text=f"{filename} - 修改时间: {timestamp}")

def load\_histories(self):

if os.path.exists(self.history\_file):

try:

with open(self.history\_file, "r") as f:

self.file\_history = json.load(f)

self.history\_combo["values"] = self.file\_history

except Exception as e:

self.insert\_output(f"\n加载文件历史失败：{str(e)}\n")

if os.path.exists(self.target\_history\_file):

try:

with open(self.target\_history\_file, "r") as f:

self.target\_history = json.load(f)

self.target\_combo["values"] = self.target\_history

except Exception as e:

self.insert\_output(f"\n加载路径历史失败：{str(e)}\n")

def add\_to\_file\_history(self, path):

"""维护文件历史记录"""

if path in self.file\_history:

self.file\_history.remove(path)

self.file\_history.insert(0, path)

self.history\_combo["values"] = self.file\_history[:10]

self.save\_history(self.history\_file, self.file\_history)

def save\_history(self, filename, history):

"""保存历史记录"""

try:

with open(filename, "w") as f:

json.dump(history[:10], f)

except Exception as e:

self.insert\_output(f"\n保存历史记录失败：{str(e)}\n")

def on\_file\_history\_select(self, event):

"""文件历史选择事件"""

self.current\_file = self.history\_combo.get()

self.update\_file\_info()

def on\_target\_history\_select(self, event):

"""目标路径历史选择事件"""

self.current\_target = self.target\_history\_combo.get()

self.target\_entry.delete(0, tk.END)

self.target\_entry.insert(0, self.current\_target)

def start\_flash(self):

"""启动刷写流程"""

self.current\_target = self.target\_combo.get()

if not self.current\_file:

messagebox.showwarning("警告", "请先选择要刷写的文件！")

return

if not self.current\_target.strip():

messagebox.showwarning("警告", "请输入目标路径！")

return

self.output\_text.delete(1.0, tk.END)

self.insert\_output("=== 开始ADB刷写流程 ===\n")

try:

if self.run\_command("adb wait-for-device") != 0:

raise Exception("设备未连接")

if self.run\_command("adb root") != 0:

raise Exception("获取Root权限失败")

if self.run\_command("adb disable-verity") != 0:

self.insert\_output("验证禁用失败，尝试重启设备...\n")

self.run\_command("adb reboot")

time.sleep(20)

if self.run\_command("adb wait-for-device") != 0:

raise Exception("设备重启后未连接")

if self.run\_command("adb root") != 0:

raise Exception("重启后获取Root失败")

if self.run\_command("adb remount") != 0:

raise Exception("分区挂载失败")

push\_cmd = f'adb push "{self.current\_file}" "{self.current\_target}"'

if self.run\_command(push\_cmd) == 0:

self.insert\_output("\n✅ 文件推送成功！\n")

else:

raise Exception("文件推送失败")

except Exception as e:

self.insert\_output(f"\n❌ 操作失败: {str(e)}\n")

messagebox.showerror("错误", f"操作失败: {str(e)}")

def run\_command(self, command):

"""执行命令并返回状态码"""

self.insert\_output(f"\n>>> 执行命令: {command}\n")

try:

process = subprocess.Popen(command,

shell=True,

stdout=subprocess.PIPE,

stderr=subprocess.STDOUT,

text=True,

creationflags=subprocess.CREATE\_NO\_WINDOW)

while True:

output = process.stdout.readline()

if output == '' and process.poll() is not None:

break

if output:

self.insert\_output(output)

returncode = process.poll()

self.insert\_output(f"\n返回代码: {returncode}\n")

return returncode

except Exception as e:

self.insert\_output(f"命令执行失败: {str(e)}\n")

return -1

def insert\_output(self, text):

"""插入输出文本"""

self.output\_text.insert(tk.END, text)

self.output\_text.see(tk.END)

self.update\_idletasks()

class FastbootTools(ttk.Frame):

def \_\_init\_\_(self, parent, controller):

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

self.controller = controller

self.history\_file = "file\_history.json"

self.file\_history = []

self.current\_file = ""

self.partition\_map = {

"boot.img": "boot\_a",

"system.img": "system\_a",

"vendor.img": "vendor\_a",

"vbmeta.img": "vbmeta\_a",

"dtbo.img": "dtbo\_a",

"recovery.img": "recovery"

}

self.pad\_config = {

"frame\_padx": 8,

"frame\_pady": 3,

"widget\_padx": 3,

"widget\_pady": 2

}

self.create\_header()

self.setup\_ui()

self.load\_history()

self.check\_environment()

def create\_header(self):

"""创建标题和返回按钮"""

header\_frame = ttk.Frame(self)

header\_frame.grid(row=0,

column=0,

sticky="ew",

pady=self.pad\_config["frame\_pady"])

ttk.Label(header\_frame, text="Fastboot 单刷工具", font=("微软雅黑", 12)).pack(

side="left",

padx=self.pad\_config["widget\_padx"]

)

ttk.Button(header\_frame, text="返回主菜单",

command=lambda: self.controller.show\_frame("MainMenu")).pack(

side="right",

padx=self.pad\_config["widget\_padx"]

)

self.columnconfigure(0, weight=1)

def setup\_ui(self):

self.env\_frame = ttk.LabelFrame(self, text="环境检测")

self.env\_frame.grid(

row=1,

column=0,

padx=self.pad\_config["frame\_padx"],

pady=self.pad\_config["frame\_pady"],

sticky="ew"

)

self.env\_status = ttk.Label(self.env\_frame, text="未检测")

self.env\_status.grid(

row=0,

column=0,

padx=self.pad\_config["widget\_padx"],

pady=self.pad\_config["widget\_pady"]

)

ttk.Button(self.env\_frame, text="检测环境", command=self.check\_environment).grid(

row=0,

column=1,

padx=self.pad\_config["widget\_padx"]

)

ttk.Button(self.env\_frame, text="配置指南", command=self.install\_drivers).grid(

row=0,

column=2,

padx=self.pad\_config["widget\_padx"]

)

self.file\_frame = ttk.LabelFrame(self, text="文件操作")

self.file\_frame.grid(

row=2,

column=0,

padx=self.pad\_config["frame\_padx"],

pady=self.pad\_config["frame\_pady"],

sticky="ew"

)

self.file\_info = ttk.Label(self.file\_frame, text="未选择文件")

self.file\_info.grid(

row=0,

column=0,

padx=self.pad\_config["widget\_padx"],

pady=self.pad\_config["widget\_pady"],

sticky="w"

)

btn\_frame = ttk.Frame(self.file\_frame)

btn\_frame.grid(row=0, column=1, sticky="e")

ttk.Button(btn\_frame, text="选择文件", command=self.select\_file).pack(

side="left",

padx=self.pad\_config["widget\_padx"]

)

ttk.Button(btn\_frame, text="刷入分区", command=self.start\_flash).pack(

side="left",

padx=self.pad\_config["widget\_padx"]

)

self.history\_combo = ttk.Combobox(self.file\_frame, values=self.file\_history)

self.history\_combo.grid(

row=1,

column=0,

columnspan=2,

padx=self.pad\_config["widget\_padx"],

pady=self.pad\_config["widget\_pady"],

sticky="ew"

)

self.history\_combo.bind("<<ComboboxSelected>>", self.on\_history\_select)

self.output\_frame = ttk.LabelFrame(self, text="操作输出")

self.output\_frame.grid(

row=3,

column=0,

padx=self.pad\_config["frame\_padx"],

pady=self.pad\_config["frame\_pady"],

sticky="nsew"

)

self.output\_text = tk.Text(

self.output\_frame,

height=15,

width=70,

wrap=tk.WORD,

padx=3,

pady=3

)

vsb = ttk.Scrollbar(self.output\_frame, orient="vertical", command=self.output\_text.yview)

self.output\_text.configure(yscrollcommand=vsb.set)

self.output\_text.grid(row=0, column=0, sticky="nsew")

vsb.grid(row=0, column=1, sticky="ns")

self.rowconfigure(3, weight=1)

self.columnconfigure(0, weight=1)

self.output\_frame.columnconfigure(0, weight=1)

self.output\_frame.rowconfigure(0, weight=1)

for frame in [self.env\_frame, self.file\_frame, self.output\_frame]:

frame.grid\_configure(padx=self.pad\_config["frame\_padx"], pady=self.pad\_config["frame\_pady"])

def check\_environment(self):

self.insert\_output("\n=== 开始环境检测 ===\n")

try:

subprocess.run(["adb", "version"],

check=True,

stdout=subprocess.DEVNULL,

stderr=subprocess.DEVNULL,

creationflags=subprocess.CREATE\_NO\_WINDOW)

subprocess.run(["fastboot", "--version"],

check=True,

stdout=subprocess.DEVNULL,

stderr=subprocess.DEVNULL,

creationflags=subprocess.CREATE\_NO\_WINDOW)

self.env\_status.config(text="环境正常", foreground="green")

self.insert\_output("检测结果：环境正常\n")

return True

except Exception as e:

self.env\_status.config(text="环境异常", foreground="red")

self.insert\_output("检测结果：环境异常\n")

self.insert\_output(f"错误信息：{str(e)}\n")

return False

def install\_drivers(self):

"""显示驱动安装指南到输出窗口"""

self.insert\_output("\n=== Fastboot环境配置指南 ===\n")

guide = """1. 下载Google USB驱动：

官方地址：https://developer.android.com/studio/run/win-usb

2. 下载Platform Tools：

下载页面：https://developer.android.com/studio/releases/platform-tools

3. 解压platform-tools到系统目录（例如：C:\\platform-tools）

4. 添加环境变量：

- 右键点击"此电脑"选择"属性"

- 进入"高级系统设置" -> "环境变量"

- 在Path中添加platform-tools目录

5. 设备连接：

- 启用USB调试模式（设置 -> 开发者选项）

- 首次连接需要在设备上授权ADB调试"""

self.insert\_output(guide + "\n")

def select\_file(self):

"""选择镜像文件"""

file\_path = filedialog.askopenfilename(filetypes=[("镜像文件", "\*.img")])

if file\_path:

self.current\_file = file\_path

self.update\_file\_info()

self.add\_to\_history(file\_path)

def update\_file\_info(self):

"""更新文件信息显示"""

if self.current\_file:

filename = os.path.basename(self.current\_file)

mtime = os.path.getmtime(self.current\_file)

timestamp = datetime.fromtimestamp(mtime).strftime("%Y-%m-%d %H:%M:%S")

self.file\_info.config(text=f"{filename} - 最后修改: {timestamp}")

def add\_to\_history(self, path):

"""维护历史记录"""

if path in self.file\_history:

self.file\_history.remove(path)

self.file\_history.insert(0, path)

self.history\_combo["values"] = self.file\_history[:10]

self.save\_history()

def save\_history(self):

"""保存历史记录到文件"""

try:

with open(self.history\_file, "w") as f:

json.dump(self.file\_history[:10], f)

except Exception as e:

self.insert\_output(f"\n保存历史记录失败：{str(e)}\n")

def load\_history(self):

"""加载历史记录"""

if os.path.exists(self.history\_file):

try:

with open(self.history\_file, "r") as f:

self.file\_history = json.load(f)

self.history\_combo["values"] = self.file\_history

except Exception as e:

self.insert\_output(f"\n加载历史记录失败：{str(e)}\n")

def on\_history\_select(self, event):

"""历史记录选择事件"""

self.current\_file = self.history\_combo.get()

self.update\_file\_info()

def start\_flash(self):

"""启动自动刷写流程"""

if not self.current\_file:

messagebox.showwarning("警告", "请先选择要刷写的镜像文件！")

return

if not self.check\_environment():

return

filename = os.path.basename(self.current\_file)

partition = self.partition\_map.get(filename.lower())

if not partition:

messagebox.showerror("错误", f"无法自动识别 {filename} 对应的分区！")

return

self.output\_text.delete(1.0, tk.END)

self.insert\_output("=== 开始自动刷写流程 ===\n")

try:

self.run\_command("adb wait-for-device")

self.run\_command("adb reboot bootloader")

if not self.wait\_for\_fastboot():

messagebox.showerror("错误", "设备未进入Fastboot模式！")

return

cmd = f"fastboot flash {partition} {self.current\_file}"

self.run\_command(cmd)

self.run\_command("fastboot reboot")

self.insert\_output("\n✅ 刷写完成，设备已重启！\n")

except Exception as e:

self.insert\_output(f"\n❌ 刷写失败: {str(e)}\n")

messagebox.showerror("错误", f"刷写失败: {str(e)}")

def wait\_for\_fastboot(self, timeout=30):

"""等待设备进入fastboot模式"""

start\_time = time.time()

self.insert\_output("\n等待设备进入Fastboot模式...")

while time.time() - start\_time < timeout:

try:

result = subprocess.run(["fastboot", "devices"],

capture\_output=True,

text=True,

creationflags=subprocess.CREATE\_NO\_WINDOW)

if "fastboot" in result.stdout:

self.insert\_output("检测到Fastboot设备！\n")

return True

time.sleep(1)

except:

pass

return False

def run\_command(self, command):

"""执行命令并显示输出"""

self.insert\_output(f"\n>>> 执行命令: {command}\n")

try:

process = subprocess.Popen(command.split(),

stdout=subprocess.PIPE,

stderr=subprocess.STDOUT,

text=True,

creationflags=subprocess.CREATE\_NO\_WINDOW)

while True:

output = process.stdout.readline()

if output == '' and process.poll() is not None:

break

if output:

self.insert\_output(output)

returncode = process.poll()

self.insert\_output(f"\n返回代码: {returncode}\n")

return returncode == 0

except Exception as e:

self.insert\_output(f"命令执行失败: {str(e)}\n")

raise

def insert\_output(self, text):

"""插入文本并自动滚动"""

self.output\_text.insert(tk.END, text)

self.output\_text.see(tk.END)

self.update\_idletasks()

class LogTools(ttk.Frame):

def \_\_init\_\_(self, parent, controller):

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

self.controller = controller

self.history\_file = 'keyword\_history.txt'

self.keyword\_history = self.load\_history()

self.default\_dir = "D:/Joyboy"

os.makedirs(self.default\_dir, exist\_ok=True)

self.create\_header()

self.create\_widgets()

self.queues = {}

self.log\_windows = {}

self.processes = {}

self.running\_flags = {}

def create\_header(self):

"""创建标题和返回按钮"""

header\_frame = ttk.Frame(self)

header\_frame.grid(row=0, column=0, sticky="ew", pady=5)

ttk.Label(header\_frame, text="日志过滤", font=("微软雅黑", 12)).pack(side="left", padx=10)

ttk.Button(header\_frame, text="返回主菜单",

command=lambda: self.controller.show\_frame("MainMenu")).pack(side="right", padx=10)

self.columnconfigure(0, weight=1)

def load\_history(self):

if os.path.exists(self.history\_file):

with open(self.history\_file, 'r', encoding='utf-8') as f:

return [line.strip() for line in f.readlines()]

return []

def save\_history(self):

with open(self.history\_file, 'w', encoding='utf-8') as f:

for item in self.keyword\_history:

f.write(f"{item}\n")

def create\_widgets(self):

self.logcat\_frame = ttk.LabelFrame(self, text="Logcat配置")

self.logcat\_frame.grid(row=1, column=0, padx=10, pady=5, sticky="ew")

self.logcat\_enabled = tk.BooleanVar()

ttk.Checkbutton(self.logcat\_frame, text="启用", variable=self.logcat\_enabled).grid(row=0, column=0, padx=5, sticky="w")

ttk.Label(self.logcat\_frame, text="关键词（逗号分隔）:").grid(row=1, column=0, padx=5, sticky="w")

self.logcat\_keyword = ttk.Entry(self.logcat\_frame, width=30)

self.logcat\_keyword.grid(row=1, column=1, padx=5, sticky="ew")

self.logcat\_case = tk.BooleanVar()

ttk.Checkbutton(self.logcat\_frame, text="区分大小写", variable=self.logcat\_case).grid(row=1, column=2, padx=5, sticky="w")

self.logcat\_history\_var = tk.BooleanVar()

ttk.Checkbutton(self.logcat\_frame, text="历史记录", variable=self.logcat\_history\_var,

command=lambda: self.toggle\_history('logcat')).grid(row=2, column=0, padx=5, sticky="w")

self.logcat\_history = ttk.Combobox(self.logcat\_frame, values=self.keyword\_history, width=28)

self.logcat\_history.grid(row=2, column=1, padx=5, sticky="ew")

self.logcat\_history.grid\_remove()

ttk.Label(self.logcat\_frame, text="保存路径:").grid(row=3, column=0, padx=5, sticky="w")

self.logcat\_path = ttk.Entry(self.logcat\_frame, width=30)

self.logcat\_path.insert(0, f"{self.default\_dir}/logcat.txt")

self.logcat\_path.grid(row=3, column=1, padx=5, sticky="ew")

ttk.Button(self.logcat\_frame, text="浏览", command=lambda: self.browse('logcat')).grid(row=3, column=2, padx=5)

self.kmsg\_frame = ttk.LabelFrame(self, text="Kmsg配置")

self.kmsg\_frame.grid(row=2, column=0, padx=10, pady=5, sticky="ew")

self.kmsg\_enabled = tk.BooleanVar()

ttk.Checkbutton(self.kmsg\_frame, text="启用", variable=self.kmsg\_enabled).grid(row=0, column=0, padx=5, sticky="w")

ttk.Label(self.kmsg\_frame, text="关键词（逗号分隔）:").grid(row=1, column=0, padx=5, sticky="w")

self.kmsg\_keyword = ttk.Entry(self.kmsg\_frame, width=30)

self.kmsg\_keyword.grid(row=1, column=1, padx=5, sticky="ew")

self.kmsg\_case = tk.BooleanVar()

ttk.Checkbutton(self.kmsg\_frame, text="区分大小写", variable=self.kmsg\_case).grid(row=1, column=2, padx=5, sticky="w")

self.kmsg\_history\_var = tk.BooleanVar()

ttk.Checkbutton(self.kmsg\_frame, text="历史记录", variable=self.kmsg\_history\_var,

command=lambda: self.toggle\_history('kmsg')).grid(row=2, column=0, padx=5, sticky="w")

self.kmsg\_history = ttk.Combobox(self.kmsg\_frame, values=self.keyword\_history, width=28)

self.kmsg\_history.grid(row=2, column=1, padx=5, sticky="ew")

self.kmsg\_history.grid\_remove()

ttk.Label(self.kmsg\_frame, text="保存路径:").grid(row=3, column=0, padx=5, sticky="w")

self.kmsg\_path = ttk.Entry(self.kmsg\_frame, width=30)

self.kmsg\_path.insert(0, f"{self.default\_dir}/kmsg.txt")

self.kmsg\_path.grid(row=3, column=1, padx=5, sticky="ew")

ttk.Button(self.kmsg\_frame, text="浏览", command=lambda: self.browse('kmsg')).grid(row=3, column=2, padx=5)

self.qsee\_log\_frame = ttk.LabelFrame(self, text="qsee\_log配置")

self.qsee\_log\_frame.grid(row=3, column=0, padx=10, pady=5, sticky="ew")

self.qsee\_log\_enabled = tk.BooleanVar()

ttk.Checkbutton(self.qsee\_log\_frame, text="启用", variable=self.qsee\_log\_enabled).grid(row=0, column=0, padx=5, sticky="w")

ttk.Label(self.qsee\_log\_frame, text="关键词（逗号分隔）:").grid(row=1, column=0, padx=5, sticky="w")

self.qsee\_log\_keyword = ttk.Entry(self.qsee\_log\_frame, width=30)

self.qsee\_log\_keyword.grid(row=1, column=1, padx=5, sticky="ew")

self.qsee\_log\_case = tk.BooleanVar()

ttk.Checkbutton(self.qsee\_log\_frame, text="区分大小写", variable=self.qsee\_log\_case).grid(row=1, column=2, padx=5, sticky="w")

self.qsee\_log\_history\_var = tk.BooleanVar()

ttk.Checkbutton(self.qsee\_log\_frame, text="历史记录", variable=self.qsee\_log\_history\_var,

command=lambda: self.toggle\_history('qsee\_log')).grid(row=2, column=0, padx=5, sticky="w")

self.qsee\_log\_history = ttk.Combobox(self.qsee\_log\_frame, values=self.keyword\_history, width=28)

self.qsee\_log\_history.grid(row=2, column=1, padx=5, sticky="ew")

self.qsee\_log\_history.grid\_remove()

ttk.Label(self.qsee\_log\_frame, text="保存路径:").grid(row=3, column=0, padx=5, sticky="w")

self.qsee\_log\_path = ttk.Entry(self.qsee\_log\_frame, width=30)

self.qsee\_log\_path.insert(0, f"{self.default\_dir}/qsee\_log.txt")

self.qsee\_log\_path.grid(row=3, column=1, padx=5, sticky="ew")

ttk.Button(self.qsee\_log\_frame, text="浏览", command=lambda: self.browse('qsee\_log')).grid(row=3, column=2, padx=5)

ttk.Button(self, text="开始抓取", command=self.start).grid(row=4, column=0, pady=10, sticky="ew")

def create\_window(self, window\_id, log\_type, path):

window = tk.Toplevel(self)

window.title(f"{log\_type}日志 - {os.path.basename(path)}")

window.geometry("800x400")

text\_area = scrolledtext.ScrolledText(window, wrap=tk.WORD)

text\_area.pack(expand=True, fill='both')

ttk.Button(window, text="打开文件", command=lambda: os.startfile(path)).pack(pady=5)

self.log\_windows[window\_id] = {

'window': window,

'text\_area': text\_area,

'file': open(path, 'a', encoding='utf-8', errors='replace')

}

def toggle\_history(self, log\_type):

if log\_type == 'logcat':

if self.logcat\_history\_var.get():

self.logcat\_history.grid()

self.logcat\_history.bind('<<ComboboxSelected>>', lambda e: self.select\_history('logcat'))

else:

self.logcat\_history.grid\_remove()

elif log\_type == 'kmsg':

if self.kmsg\_history\_var.get():

self.kmsg\_history.grid()

self.kmsg\_history.bind('<<ComboboxSelected>>', lambda e: self.select\_history('kmsg'))

else:

self.kmsg\_history.grid\_remove()

else:

if self.qsee\_log\_history\_var.get():

self.qsee\_log\_history.grid()

self.qsee\_log\_history.bind('<<ComboboxSelected>>', lambda e: self.select\_history('qsee\_log'))

else:

self.qsee\_log\_history.grid\_remove()

def select\_history(self, log\_type):

if log\_type == 'logcat':

self.logcat\_keyword.delete(0, tk.END)

self.logcat\_keyword.insert(0, self.logcat\_history.get())

elif log\_type == 'kmsg':

self.kmsg\_keyword.delete(0, tk.END)

self.kmsg\_keyword.insert(0, self.kmsg\_history.get())

else:

self.qsee\_log\_keyword.delete(0, tk.END)

self.qsee\_log\_keyword.insert(0, self.kmsg\_history.get())

def browse(self, log\_type):

path = filedialog.asksaveasfilename(defaultextension=".log")

if path:

if log\_type == 'logcat':

self.logcat\_path.delete(0, tk.END)

self.logcat\_path.insert(0, path)

elif log\_type == 'kmsg':

self.kmsg\_path.delete(0, tk.END)

self.kmsg\_path.insert(0, path)

else:

self.qsee\_log\_path.delete(0, tk.END)

self.qsee\_log\_path.insert(0, path)

def start(self):

tasks = []

if self.logcat\_enabled.get():

logcat\_keyword = self.logcat\_keyword.get().strip()

tasks.append(('logcat', logcat\_keyword, self.logcat\_path.get(), self.logcat\_case.get()))

if self.kmsg\_enabled.get():

kmsg\_keyword = self.kmsg\_keyword.get().strip()

tasks.append(('kmsg', kmsg\_keyword, self.kmsg\_path.get(), self.kmsg\_case.get()))

if self.qsee\_log\_enabled.get():

qsee\_log\_keyword = self.qsee\_log\_keyword.get().strip()

tasks.append(('qsee\_log', qsee\_log\_keyword, self.qsee\_log\_path.get(), self.qsee\_log\_case.get()))

if not tasks:

messagebox.showerror("错误", "请至少选择一个日志类型")

return

for task in tasks:

log\_type, keywords, path, case = task

if not path:

messagebox.showerror("错误", "请填写保存路径")

return

if keywords and keywords not in self.keyword\_history:

self.keyword\_history.insert(0, keywords)

if len(self.keyword\_history) > 20:

self.keyword\_history.pop()

self.save\_history()

self.logcat\_history['values'] = self.keyword\_history

self.kmsg\_history['values'] = self.keyword\_history

self.qsee\_log\_history['values'] = self.keyword\_history

q = queue.Queue()

window\_id = str(uuid.uuid4())

self.queues[window\_id] = q

self.create\_window(window\_id, log\_type, path)

self.running\_flags[window\_id] = True

threading.Thread(target=self.capture, args=(log\_type, keywords, path, case, q, window\_id), daemon=True).start()

self.after(100, self.update\_display, window\_id)

def create\_window(self, window\_id, log\_type, path):

window = tk.Toplevel(self)

window.title(f"{log\_type}日志 - {os.path.basename(path)}")

window.geometry("800x400")

text\_area = scrolledtext.ScrolledText(window, wrap=tk.WORD)

text\_area.pack(expand=True, fill='both')

ttk.Button(window, text="打开文件", command=lambda: os.startfile(path)).pack(pady=5)

self.log\_windows[window\_id] = {

'window': window,

'text\_area': text\_area,

'file': open(path, 'a', encoding='utf-8')

}

window.protocol("WM\_DELETE\_WINDOW", lambda: self.close\_window(window\_id))

def close\_window(self, window\_id):

"""安全关闭窗口的流程"""

if window\_id in self.running\_flags:

self.running\_flags[window\_id] = False

if window\_id in self.processes:

self.processes[window\_id].terminate()

del self.processes[window\_id]

if window\_id in self.log\_windows:

self.log\_windows[window\_id]['file'].close()

self.log\_windows[window\_id]['window'].destroy()

del self.log\_windows[window\_id]

def capture(self, log\_type, keywords, path, case\_sensitive, q, window\_id):

keywords = [k.strip() for k in keywords.split(',')]

cmd = ['adb', 'shell', 'logcat'] if log\_type == 'logcat' else (

['adb', 'shell', 'cat', '/proc/kmsg'] if log\_type == 'kmsg'

else ['adb', 'shell', 'cat', '/proc/tzdbg/qsee\_log']

)

try:

proc = subprocess.Popen(cmd,

stdout=subprocess.PIPE,

text=True,

encoding='utf-8',

errors='replace')

self.processes[window\_id] = proc

buffer = []

last\_flush = time.time()

while self.running\_flags.get(window\_id, False):

line = proc.stdout.readline()

if not line:

time.sleep(0.1)

continue

if self.check\_filter(line, keywords, case\_sensitive):

buffer.append(line)

q.put(line)

if len(buffer) >= 100 or (time.time() - last\_flush) > 0.5:

self.log\_windows[window\_id]['file'].writelines(buffer)

buffer.clear()

last\_flush = time.time()

if proc.poll() is not None:

break

if buffer:

self.log\_windows[window\_id]['file'].writelines(buffer)

except Exception as e:

q.put(f"Error: {str(e)}")

finally:

self.close\_window(window\_id)

def check\_filter(self, line, keywords, case\_sensitive):

if not keywords:

return True

line\_check = line if case\_sensitive else line.lower()

return any(

(kw if case\_sensitive else kw.lower()) in line\_check

for kw in keywords

)

def update\_display(self, window\_id):

if window\_id not in self.running\_flags or not self.running\_flags[window\_id]:

return

try:

max\_lines = 50

processed = 0

while not self.queues[window\_id].empty() and processed < max\_lines:

line = self.queues[window\_id].get\_nowait()

self.log\_windows[window\_id]['text\_area'].insert(tk.END, line)

processed += 1

self.log\_windows[window\_id]['text\_area'].see(tk.END)

except queue.Empty:

pass

finally:

self.after(50, self.update\_display, window\_id)

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

app = MainApplication()

style = ttk.Style()

style.configure('Status.TFrame', background='#f0f0f0', borderwidth=2, relief="groove")

app.mainloop()