main.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import os

import sys

import tkinter as tk

from tkinter import messagebox

import logging

from datetime import datetime

import subprocess

# 设置导入路径，同时支持开发环境和打包环境

def setup\_paths():

"""设置导入路径"""

if getattr(sys, 'frozen', False):

# 打包环境

base\_path = sys.\_MEIPASS if hasattr(sys, '\_MEIPASS') else os.path.dirname(sys.executable)

sys.path.append(base\_path)

else:

# 开发环境

base\_path = os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))

if base\_path not in sys.path:

sys.path.insert(0, base\_path)

return base\_path

# 初始化路径

BASE\_PATH = setup\_paths()

# 导入路径管理模块

try:

# 统一导入方式，让 Python 在 sys.path 中查找模块

from src.utils.paths import (

get\_application\_path, get\_user\_data\_dir, get\_resources\_path,

get\_icon\_path, get\_exports\_dir, get\_temp\_dir

)

from src.models.database import Database

from src.ui.main\_window import MainWindow

from src.config.logger import setup\_logger

except ImportError as e:

print(f"导入错误: {e}")

print(f"当前路径: {os.getcwd()}")

print(f"sys.path: {sys.path}")

input("按回车键退出...")

sys.exit(1)

# 预加载必要的模块

try:

# 尝试预加载必要的模块

if getattr(sys, 'frozen', False):

# 打包环境

from utils.preload import preload\_modules

else:

# 开发环境

from src.utils.preload import preload\_modules

# 执行预加载

preload\_modules()

logging.info("预加载模块完成")

except Exception as e:

print(f"预加载模块失败: {str(e)}")

logging.error(f"预加载模块失败: {str(e)}")

# 修改：添加类号前缀匹配函数，用于替换直接的类号比较

def match\_category\_code(row\_category\_code, selected\_category\_id):

"""

检查行的类号是否匹配选定的类别

Args:

row\_category\_code: Excel行中的类号（如"1-1"、"4-1-1"）

selected\_category\_id: 用户选择的类别ID（如1、4、9）

Returns:

是否匹配

"""

if not row\_category\_code:

return False

# 转换为字符串并去除空格

row\_category\_code = str(row\_category\_code).strip()

selected\_category\_id = str(selected\_category\_id).strip()

# 检查类号是否以类别ID加连字符开头，或者完全相等

# 例如，类别ID为1时，匹配"1-1"、"1-2"等，或者就是"1"

# 类别ID为4时，匹配"4-1-1"、"4-2-1"等，或者就是"4"

return row\_category\_code.startswith(f"{selected\_category\_id}-") or row\_category\_code == selected\_category\_id

# 应用程序常量

VERSION = "V1.0"

def setup\_logging():

"""配置日志记录，同时输出到控制台和文件"""

try:

# 使用Windows本地应用数据目录

import ctypes

from ctypes import wintypes

# 获取本地应用数据目录

CSIDL\_LOCAL\_APPDATA = 0x001c # 本地应用数据目录

SHGFP\_TYPE\_CURRENT = 0 # 获取当前值

buf = ctypes.create\_unicode\_buffer(wintypes.MAX\_PATH)

ctypes.windll.shell32.SHGetFolderPathW(0, CSIDL\_LOCAL\_APPDATA, 0, SHGFP\_TYPE\_CURRENT, buf)

app\_data\_dir = buf.value

# 创建应用专属日志目录

log\_dir = os.path.join(app\_data\_dir, 'ArchiMgr', 'Logs')

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

# 生成带日期的日志文件名

log\_file = os.path.join(log\_dir, f'archimgr\_{datetime.now().strftime("%Y%m%d")}.log')

except Exception as e:

# 如果获取应用数据目录失败，使用用户主目录

log\_dir = os.path.join(os.path.expanduser('~'), 'ArchiMgr', 'Logs')

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

log\_file = os.path.join(log\_dir, f'archimgr\_{datetime.now().strftime("%Y%m%d")}.log')

logging.error(f"使用备用日志目录: {log\_file}, 错误: {str(e)}")

# 记录日志文件路径

logging.info(f"日志文件路径: {log\_file}")

# 创建格式化器

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# 创建文件处理器

file\_handler = logging.FileHandler(log\_file, encoding='utf-8')

file\_handler.setLevel(logging.INFO)

file\_handler.setFormatter(formatter)

# 创建控制台处理器

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.INFO)

console\_handler.setFormatter(formatter)

# 配置根日志记录器

root\_logger = logging.getLogger()

root\_logger.setLevel(logging.INFO)

# 禁用SQLAlchemy的日志

logging.getLogger('sqlalchemy.engine').setLevel(logging.WARNING)

logging.getLogger('sqlalchemy.pool').setLevel(logging.WARNING)

# 移除所有现有的处理器

for handler in root\_logger.handlers[:]:

root\_logger.removeHandler(handler)

# 添加处理器

root\_logger.addHandler(file\_handler)

root\_logger.addHandler(console\_handler)

logging.info(f"日志文件已创建: {log\_file}")

def main():

"""主函数"""

# 在函数作用域内声明变量

db = None

root = None

is\_cleaning = False # 标记是否正在清理中

# 调试信息输出函数

def debug\_log\_msg(msg):

timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

debug\_msg = f'[{timestamp}] {msg}'

print(debug\_msg, file=sys.stderr) # 输出到标准错误，避免与GUI冲突

try:

with open('debug.log', 'a', encoding='utf-8') as f:

f.write(debug\_msg + '\n')

except Exception as e:

print(f'[ERROR] 无法写入调试日志: {e}', file=sys.stderr)

debug\_log\_msg('=' \* 50)

debug\_log\_msg('程序开始启动')

debug\_log\_msg(f'当前工作目录: {os.getcwd()}')

debug\_log\_msg(f'Python路径: {sys.executable}')

debug\_log\_msg(f'命令行参数: {sys.argv}')

debug\_log\_msg('系统路径:')

for p in sys.path:

debug\_log\_msg(f' {p}')

debug\_log\_msg(f'文件编码: {sys.getdefaultencoding()}')

debug\_log\_msg(f'文件系统编码: {sys.getfilesystemencoding()}')

setup\_logging()

def cleanup():

"""清理资源"""

nonlocal db, root, is\_cleaning

is\_cleaning = True

try:

logging.info("开始清理资源...")

if db is not None:

logging.info("正在关闭数据库连接...")

if hasattr(db, 'conn') and db.conn: # 检查是否有 conn 属性

db.conn.close() # 使用 conn 属性关闭数据库连接

logging.info("数据库连接已关闭")

else:

logging.warning("数据库连接对象不存在或已关闭")

if root is not None:

logging.info("正在销毁主窗口...")

try:

root.destroy()

except Exception as e:

logging.error(f"销毁主窗口时出错: {e}")

logging.info("资源清理完成")

except Exception as e:

logging.exception("清理资源时发生错误")

import traceback

traceback.print\_exc()

try:

# 初始化日志系统

setup\_logger()

logging.info(f"启动应用，版本: {VERSION}")

# 初始化数据库

logging.info("正在初始化数据库...")

db = Database()

logging.info("数据库初始化完成")

try:

# 设置路径 - 使用路径管理模块

app\_dir = get\_application\_path()

# 创建GUI

logging.info("正在创建主窗口...")

root = tk.Tk()

root.title(f"档案检索系统v{VERSION}")

root.geometry("1200x800")

# 设置窗口关闭时的处理

def on\_closing():

logging.info("检测到窗口关闭事件...")

cleanup()

root.quit()

root.protocol("WM\_DELETE\_WINDOW", on\_closing)

logging.info("主窗口创建完成")

# 设置程序图标

try:

icon\_path = get\_icon\_path()

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

root.iconbitmap(icon\_path)

logging.info(f"已设置程序图标: {icon\_path}")

else:

logging.warning(f"程序图标文件不存在: {icon\_path}")

except Exception as e:

logging.warning(f"设置图标失败: {str(e)}")

# 继续运行，不因图标问题而终止

# 创建主窗口

logging.info("正在初始化主窗口组件...")

main\_window = MainWindow(root, db, version=VERSION)

logging.info("主窗口组件初始化完成")

# 进入主循环

logging.info("进入主事件循环...")

root.mainloop()

logging.info("主事件循环结束")

except Exception as e:

logging.exception("创建主窗口时发生错误")

messagebox.showerror("错误", f"无法创建主窗口: {str(e)}")

raise # 重新抛出异常，让外部处理

except Exception as e:

logging.exception("程序初始化失败")

messagebox.showerror("错误", f"程序初始化失败: {str(e)}")

return 1 # 返回非零表示错误

finally:

# 只在 finally 中清理资源，如果还没有被清理过的话

if not is\_cleaning and (db is not None or root is not None):

cleanup()

logging.info("程序正常退出")

return 0 # 返回0表示成功

def register\_system(reg\_code, user\_name):

"""注册系统"""

# 验证注册信息

if is\_valid\_license(reg\_code, user\_name):

# 创建加密的许可证文件

return True, "注册成功"

return False, "注册失败"

# 分析注册信息存储机制

"""

1. 分析当前系统中存储注册信息的多个位置

2. 评估这种设计是否存在冗余

3. 提供更合理的设计建议

"""

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

main()

main\_window.py

import tkinter as tk

from tkinter import ttk

import logging

import shutil

import os

from tkinter import filedialog, messagebox, simpledialog

import pandas as pd

import subprocess

import sys

import re

import time

import json

import hashlib

import socket

import datetime

from tkinter import font as tkfont

from tkinter import scrolledtext

# 使用兼容性导入

try:

# 打包环境下的导入方式

if getattr(sys, 'frozen', False):

from utils.excel\_utils import get\_excel\_info, ExcelFileNotFound

else:

# 开发环境下的导入方式

from src.utils.excel\_utils import get\_excel\_info, ExcelFileNotFound

except ImportError as e:

import logging

logging.error(f"导入excel\_utils模块失败: {e}")

# 不直接退出，让程序继续运行并在需要时报错

class MainWindow:

def \_\_init\_\_(self, root, db=None, version="1.0"):

self.root = root

self.db = db

self.version = version

# 设置日志级别和格式 - 正式版为INFO级别

logger = logging.getLogger()

logger.setLevel(logging.INFO)

# 确保至少有一个处理器，否则添加一个

if not logger.handlers:

# 创建控制台处理器

console\_handler = logging.StreamHandler()

console\_handler.setLevel(logging.INFO)

# 设置日志格式

formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

console\_handler.setFormatter(formatter)

# 添加处理器到日志记录器

logger.addHandler(console\_handler)

# 添加文件处理器

log\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'logs')

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

log\_file = os.path.join(log\_dir, f"app\_{time.strftime('%Y%m%d')}.log")

file\_handler = logging.FileHandler(log\_file, encoding='utf-8')

file\_handler.setLevel(logging.INFO)

file\_handler.setFormatter(formatter)

logger.addHandler(file\_handler)

logging.info("初始化日志系统完成")

logging.info("=== 档案检索系统启动 ===")

logging.info(f"系统版本: {self.version}")

# 初始化文件列表和分类映射

self.all\_files = []

self.category\_mapping = {}

self.version = version

self.db = db

self.root = root

# 配置目录

self.config\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'config')

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

# 设置文件路径

self.settings\_file = os.path.join(self.config\_dir, 'user\_settings.json')

# 注册文件路径

self.registration\_file = os.path.join(self.config\_dir, 'registration.dat')

# 注册状态

self.is\_registered = False

self.current\_org = None # 当前注册单位

# 检查是否已注册

if os.path.exists(self.registration\_file) and os.path.getsize(self.registration\_file) > 0:

logging.debug(f"注册文件存在: {self.registration\_file}, 正在检查注册状态")

self.is\_registered = self.check\_registration()

else:

logging.debug(f"注册文件不存在或为空: {self.registration\_file}, 状态设为未注册")

self.is\_registered = False

logging.info(f"注册状态: {'已注册' if self.is\_registered else '未注册'}")

# 加载用户设置

self.settings = self.load\_settings()

# 导入文件的根目录（从设置中加载）

self.import\_root\_dir = self.settings.get('import\_root\_dir')

# 登录状态

self.current\_user = None

self.login\_status\_var = tk.StringVar()

self.login\_status\_var.set("未登录")

# 确保数据库中有users表并初始化管理员账号

self.init\_users\_table()

self.create\_admin\_user()

# 工具菜单引用，用于权限控制

self.tools\_menu = None

# 设置UI

self.setup\_ui()

# 初始化数据

self.init\_data()

self.current\_search\_name = None # 添加当前搜索人名的记录

self.current\_search\_id = None # 添加当前搜索编号的记录

self.has\_searched = False # 标记是否进行过搜索

# 初始状态为未登录

self.current\_user = None

self.login\_status\_var.set("未登录")

self.update\_menu\_by\_role(None)

self.update\_tools\_permission(False)

# 根据注册状态更新界面

self.update\_ui\_by\_registration()

# 获取本地IP地址

self.local\_ip = self.\_get\_local\_ip()

logging.info(f"本地IP地址: {self.local\_ip}")

def \_get\_local\_ip(self):

"""获取本地IP地址"""

try:

# 创建一个临时socket连接，获取本地IP地址

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

# 不需要真正连接

s.connect(('8.8.8.8', 80))

ip = s.getsockname()[0]

s.close()

return ip

except Exception as e:

logging.error(f"获取本地IP地址失败: {str(e)}")

return "127.0.0.1" # 返回本地回环地址作为默认值

def init\_users\_table(self):

"""初始化用户表"""

try:

self.db.cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT UNIQUE NOT NULL,

password TEXT NOT NULL,

real\_name TEXT,

role TEXT DEFAULT 'user',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

''')

self.db.conn.commit()

logging.info("用户表初始化完成")

except Exception as e:

logging.error(f"初始化用户表失败: {str(e)}")

def create\_admin\_user(self):

"""创建管理员账号"""

try:

# 检查管理员账号是否存在

self.db.cursor.execute('SELECT COUNT(\*) FROM users WHERE username = ?', ('admin',))

if self.db.cursor.fetchone()[0] == 0:

# 创建管理员账号

hashed\_password = self.hash\_password('admin123')

self.db.cursor.execute(

'INSERT INTO users (username, password, real\_name, role) VALUES (?, ?, ?, ?)',

('admin', hashed\_password, '系统管理员', 'admin')

)

self.db.conn.commit()

logging.info("已创建管理员账号")

else:

logging.info("管理员账号已存在")

except Exception as e:

logging.error(f"创建管理员账号失败: {str(e)}")

def update\_menu\_by\_role(self, role):

"""根据用户角色更新菜单"""

# 清空文件菜单

self.file\_menu.delete(0, tk.END)

if role == 'admin':

# 管理员菜单

self.file\_menu.add\_command(label="修改密码", command=self.show\_change\_password\_dialog)

self.file\_menu.add\_command(label="添加用户", command=self.show\_register\_dialog)

self.file\_menu.add\_command(label="退出登录", command=self.logout)

self.file\_menu.add\_separator()

self.file\_menu.add\_command(label="退出系统", command=self.root.quit)

# 启用工具菜单

self.update\_tools\_permission(True)

# 启用日志菜单

self.log\_menu.entryconfigure("查看操作日志", state=tk.NORMAL)

self.log\_menu.entryconfigure("导出日志", state=tk.NORMAL)

elif role == 'user':

# 普通用户菜单

self.file\_menu.add\_command(label="退出登录", command=self.logout)

self.file\_menu.add\_separator()

self.file\_menu.add\_command(label="退出系统", command=self.root.quit)

# 禁用工具菜单

self.update\_tools\_permission(False)

# 禁用日志菜单

self.log\_menu.entryconfigure("查看操作日志", state=tk.DISABLED)

self.log\_menu.entryconfigure("导出日志", state=tk.DISABLED)

else:

# 未登录菜单

self.file\_menu.add\_command(label="登录", command=self.show\_login\_dialog)

self.file\_menu.add\_separator()

self.file\_menu.add\_command(label="退出系统", command=self.root.quit)

# 禁用日志菜单

self.log\_menu.entryconfigure("查看操作日志", state=tk.DISABLED)

self.log\_menu.entryconfigure("导出日志", state=tk.DISABLED)

def update\_tools\_permission(self, is\_admin):

"""更新工具栏权限"""

if hasattr(self, 'import\_file\_btn'):

# 搜索按钮和导入文件按钮状态 - 只要登录了（无论是管理员还是普通用户）就可以使用

if self.current\_user: # 如果已登录（任何角色）

# 启用搜索按钮

if hasattr(self, 'search\_button'):

self.search\_button.config(state=tk.NORMAL)

# 启用导入文件按钮

self.import\_file\_btn.config(state=tk.NORMAL)

else: # 未登录

# 禁用搜索按钮

if hasattr(self, 'search\_button'):

self.search\_button.config(state=tk.DISABLED)

# 禁用导入文件按钮

self.import\_file\_btn.config(state=tk.DISABLED)

# 管理员特有权限

if is\_admin:

# 启用安全菜单

if hasattr(self, 'security\_menu'):

self.security\_menu.entryconfigure("档案完整性检查", state=tk.NORMAL)

self.security\_menu.entryconfigure("清理数据库", state=tk.NORMAL)

else:

# 禁用安全菜单

if hasattr(self, 'security\_menu'):

self.security\_menu.entryconfigure("档案完整性检查", state=tk.DISABLED)

self.security\_menu.entryconfigure("清理数据库", state=tk.DISABLED)

def setup\_ui(self):

"""设置用户界面"""

# 设置窗口标题

self.root.title(f"档案检索系统 {self.version}")

# 设置窗口大小和位置

window\_width = 1024

window\_height = 768

self.root.geometry(f"{window\_width}x{window\_height}")

self.root.minsize(800, 600) # 设置最小尺寸

# 将窗口居中

screen\_width = self.root.winfo\_screenwidth()

screen\_height = self.root.winfo\_screenheight()

x = (screen\_width - window\_width) // 2

y = (screen\_height - window\_height) // 2

self.root.geometry(f"{window\_width}x{window\_height}+{x}+{y}")

# 创建菜单栏

self.create\_menu()

# 创建工具栏

self.create\_toolbar()

# 初始化搜索结果状态变量

self.search\_result\_var = tk.StringVar()

self.search\_result\_var.set("就绪")

# 创建主框架

self.create\_main\_frame()

# 创建状态栏（最后创建，显示在底部）

self.create\_status\_bar()

# 确保状态栏显示与注册状态一致（调试）

logging.debug(f"初始化UI完成，注册状态: {'已注册' if self.is\_registered else '未注册'}")

if hasattr(self, 'reg\_status\_var'):

status\_text = self.reg\_status\_var.get()

logging.debug(f"状态栏显示: {status\_text}")

# 确保状态栏显示与注册状态一致

expected\_status = "已注册" if self.is\_registered else "未注册"

if expected\_status not in status\_text:

logging.warning(f"状态栏显示与注册状态不一致，强制更新")

self.update\_status\_bar()

def create\_menu(self):

"""创建菜单栏"""

menubar = tk.Menu(self.root)

self.root.config(menu=menubar)

# 文件菜单

self.file\_menu = tk.Menu(menubar, tearoff=0)

menubar.add\_cascade(label="文件", menu=self.file\_menu)

# 先添加登录选项（默认显示）

self.file\_menu.add\_command(label="登录", command=self.show\_login\_dialog)

# 帮助菜单

self.help\_menu = tk.Menu(menubar, tearoff=0)

menubar.add\_cascade(label="帮助", menu=self.help\_menu)

self.help\_menu.add\_command(label="使用说明", command=self.show\_help)

# 添加用户注册选项（未注册时可用）

self.help\_menu.add\_command(label="用户注册", command=self.show\_registration\_dialog)

# 根据注册状态设置注册菜单项的状态

# reg\_state = tk.DISABLED if self.is\_registered else tk.NORMAL

# self.help\_menu.entryconfigure("用户注册", state=reg\_state)

self.help\_menu.add\_command(label="关于", command=self.show\_about)

# 工具菜单

self.tools\_menu = tk.Menu(menubar, tearoff=0)

menubar.add\_cascade(label="工具", menu=self.tools\_menu)

self.tools\_menu.add\_command(label="打开程序安装目录", command=self.open\_install\_directory)

self.tools\_menu.add\_command(label="打开档案文件目录", command=self.open\_archive\_directory)

self.tools\_menu.add\_command(label="打开数据库位置", command=self.open\_database\_location)

# 安全菜单

self.security\_menu = tk.Menu(menubar, tearoff=0)

menubar.add\_cascade(label="安全", menu=self.security\_menu)

self.security\_menu.add\_command(label="档案完整性检查", command=self.show\_integrity\_check)

self.security\_menu.add\_command(label="清理数据库", command=self.cleanup\_database)

# 日志菜单

self.log\_menu = tk.Menu(menubar, tearoff=0)

menubar.add\_cascade(label="日志", menu=self.log\_menu)

self.log\_menu.add\_command(label="查看操作日志", command=self.show\_operation\_logs)

self.log\_menu.add\_command(label="导出日志", command=self.export\_logs)

# 用户管理菜单（初始时不显示，管理员登录后再添加）

self.user\_menu = tk.Menu(menubar, tearoff=0)

# 初始不添加到菜单栏，管理员登录后再添加

self.user\_menu\_index = None

def create\_toolbar(self):

"""创建工具栏"""

toolbar\_frame = ttk.Frame(self.root)

toolbar\_frame.pack(fill=tk.X, padx=5, pady=5)

# 左侧按钮框架

button\_frame = ttk.Frame(toolbar\_frame)

button\_frame.pack(side=tk.LEFT, padx=(0, 10))

# 导入文件按钮

self.import\_file\_btn = ttk.Button(

button\_frame,

text="导入档案",

command=self.import\_files

)

self.import\_file\_btn.pack(side=tk.LEFT)

# 右侧搜索框架

search\_frame = ttk.Frame(toolbar\_frame)

search\_frame.pack(side=tk.RIGHT, fill=tk.X, expand=True)

# 登录状态显示

login\_status\_label = ttk.Label(search\_frame, textvariable=self.login\_status\_var)

login\_status\_label.pack(side=tk.RIGHT, padx=(10, 0))

# 姓名搜索框

name\_label = ttk.Label(search\_frame, text="姓名:")

name\_label.pack(side=tk.LEFT, padx=(0, 2))

self.search\_name\_var = tk.StringVar()

name\_entry = ttk.Entry(search\_frame, textvariable=self.search\_name\_var, width=10)

name\_entry.pack(side=tk.LEFT, padx=(0, 10))

# 编号搜索框

id\_label = ttk.Label(search\_frame, text="编号:")

id\_label.pack(side=tk.LEFT, padx=(0, 2))

self.search\_id\_var = tk.StringVar()

id\_entry = ttk.Entry(search\_frame, textvariable=self.search\_id\_var, width=10)

id\_entry.pack(side=tk.LEFT, padx=(0, 10))

# 搜索按钮

self.search\_button = ttk.Button(search\_frame, text="搜索", command=self.search\_person)

self.search\_button.pack(side=tk.LEFT)

def create\_main\_frame(self):

"""创建主框架"""

# 创建主框架

main\_frame = ttk.Frame(self.root)

main\_frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

# 创建左右分隔窗口

paned = ttk.PanedWindow(main\_frame, orient=tk.HORIZONTAL)

paned.pack(fill=tk.BOTH, expand=True)

# 创建白色背景样式

style = ttk.Style()

style.configure('White.TFrame', background='white')

# 创建左侧框架（用于分类树）

self.left\_frame = ttk.Frame(paned, style='White.TFrame')

paned.add(self.left\_frame, weight=1)

# 创建右侧框架（用于文件列表）

self.right\_frame = ttk.Frame(paned)

paned.add(self.right\_frame, weight=3)

# 设置分类树

self.setup\_category\_tree()

# 设置文件列表

self.setup\_file\_list()

def setup\_category\_tree(self):

"""设置分类树"""

# 创建树形控件，使用borderwidth=0去掉边框

self.tree = ttk.Treeview(self.left\_frame, show='tree')

# 设置样式去掉边框和背景色

style = ttk.Style()

style.configure("Treeview", borderwidth=0, background='white')

# 修改选中项的背景色为蓝色，前景色（文字）为白色

style.map("Treeview",

background=[('selected', '#0078d7')],

foreground=[('selected', 'white')])

self.tree.pack(fill=tk.BOTH, expand=True)

# 添加滚动条，只在需要时显示

scrollbar = ttk.Scrollbar(self.left\_frame, orient=tk.VERTICAL, command=self.tree.yview)

# 配置滚动条和树形控件的关联

def set\_scrollbar(\*args):

# 只在内容超出可视区域时显示滚动条

if float(args[0]) == 0.0 and float(args[1]) == 1.0:

scrollbar.pack\_forget() # 隐藏滚动条

else:

scrollbar.pack(side=tk.RIGHT, fill=tk.Y) # 显示滚动条

self.tree.configure(yscrollcommand=set\_scrollbar)

# 初始不显示滚动条，等待内容加载后再决定

# 绑定选择事件

self.tree.bind('<<TreeviewSelect>>', self.on\_category\_selected)

# 从数据库加载分类

self.load\_categories\_from\_db()

def setup\_file\_list(self):

"""设置文件列表"""

# 创建文件列表框架

list\_frame = ttk.Frame(self.right\_frame)

list\_frame.pack(fill=tk.BOTH, expand=True)

# 创建Treeview，添加新的列（编号、姓名、类号、材料名称、文件名、日期、页数）

self.file\_list = ttk.Treeview(

list\_frame,

columns=('file\_id', 'person', 'class\_code', 'material\_name', 'filename', 'date', 'page\_count', 'path'),

show='headings'

)

# 设置列标题

self.file\_list.heading('file\_id', text='编号')

self.file\_list.heading('person', text='姓名')

self.file\_list.heading('class\_code', text='类号')

self.file\_list.heading('material\_name', text='材料名称')

self.file\_list.heading('filename', text='文件名')

self.file\_list.heading('date', text='日期')

self.file\_list.heading('page\_count', text='页数')

self.file\_list.heading('path', text='路径')

# 设置除路径列外的所有列为等宽

equal\_width = 100

self.file\_list.column('file\_id', width=equal\_width)

self.file\_list.column('person', width=equal\_width)

self.file\_list.column('class\_code', width=equal\_width)

self.file\_list.column('material\_name', width=equal\_width)

self.file\_list.column('filename', width=equal\_width)

self.file\_list.column('date', width=equal\_width)

self.file\_list.column('page\_count', width=equal\_width)

# 路径列保持较宽

self.file\_list.column('path', width=200)

# 添加滚动条

scrollbar = ttk.Scrollbar(list\_frame, orient=tk.VERTICAL, command=self.file\_list.yview)

self.file\_list.configure(yscrollcommand=scrollbar.set)

# 布局

self.file\_list.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

# 绑定双击事件

self.file\_list.bind('<Double-1>', self.on\_file\_double\_click)

def update\_file\_list(self, files):

"""更新文件列表"""

# 清空现有列表

self.file\_list.delete(\*self.file\_list.get\_children())

# 添加右键菜单

self.create\_file\_context\_menu()

# 对文件列表进行排序 - 按类号的数字顺序排序

def get\_sort\_key(file):

# 如果文件长度大于等于3，使用类号(第3列)进行排序

if len(file) >= 3:

class\_code = file[2] if file[2] else ""

# 将类号拆分为数字部分，并转换为整数进行排序

parts = class\_code.split('-')

# 确保至少有一个部分

if not parts:

return (0,)

# 尝试将每个部分转换为整数

try:

return tuple(int(p) if p.isdigit() else 0 for p in parts)

except (ValueError, AttributeError):

return (0,)

return (0,)

# 对文件列表进行排序

sorted\_files = sorted(files, key=get\_sort\_key)

# 添加文件到列表

for file in sorted\_files:

# 检查文件数据结构

if len(file) < 2:

logging.warning(f"文件数据结构不完整: {file}")

continue

try:

# 提取文件信息

# 根据数据库查询结果解析数据

# person\_files表结构: id, person\_name, file\_name, file\_path, category\_id, dir\_name, file\_id

if len(file) >= 7:

# 完整的person\_files记录

file\_id = file[6] if file[6] else "" # file\_id

person\_name = file[1] if file[1] else "" # person\_name

# 从文件名中提取类号和材料名称

file\_name = file[2] if file[2] else "" # file\_name

# 提取类号 - 应该是1-1, 4-1-1这种格式

class\_code = ""

material\_name = "" # 材料名称应该从Excel中读取，不硬编码

date = "" # 日期留空，应该从Excel中读取

page\_count = "" # 页数留空，应该从Excel中读取

# 使用完整类号格式 - 优先使用category\_id中的类号

class\_code = file[4] if file[4] else "" # category\_id

# 如果category\_id为空，从文件名中提取完整类号

if not class\_code:

class\_code = self.extract\_category\_num(file\_name)

# 如果仍然为空，则使用默认格式

if not class\_code:

# 从文件名中提取类号

if "-" in file\_name:

parts = file\_name.split("-")

# 提取类号部分 (如 1-1, 4-1-1)

if parts[0].isdigit():

class\_code = parts[0]

if len(parts) >= 2 and parts[1].isdigit():

class\_code += "-" + parts[1]

if len(parts) >= 3 and parts[2].isdigit():

class\_code += "-" + parts[2]

else:

# 如果文件名中没有破折号，则尝试从文件名中提取类号

# 移除文件扩展名

base\_name = os.path.splitext(file\_name)[0]

# 如果是纯数字，则直接作为类号

if base\_name.isdigit():

class\_code = base\_name

else:

# 尝试提取开头的数字部分作为类号

match = re.match(r'^(\d+)', base\_name)

if match:

class\_code = match.group(1)

else:

class\_code = base\_name

# 获取文件路径

file\_path = file[3] if file[3] else "" # file\_path

# 从Excel获取文件信息（材料名称、日期、页数）

try:

from src.utils.excel\_utils import get\_excel\_info, ExcelFileNotFound

if hasattr(self, 'import\_root\_dir') and self.import\_root\_dir and class\_code:

material\_name, date, page\_count, \_ = get\_excel\_info(self.import\_root\_dir, person\_name, file\_id, class\_code)

logging.info(f"获取Excel信息成功: 类号='{class\_code}', 材料名称='{material\_name}', 日期='{date}', 页数='{page\_count}'")

except Exception as e:

logging.error(f"获取Excel信息失败: {str(e)}")

elif len(file) == 2: # 简单的文件名和路径

file\_name = file[0] if file[0] else ""

file\_path = file[1] if file[1] else ""

# 其他字段留空

file\_id = ""

person\_name = ""

material\_name = "" # 材料名称应该从Excel中读取

date = "" # 日期留空

page\_count = "" # 页数留空

# 从文件名中提取完整类号

class\_code = self.extract\_category\_num(file\_name)

# 如果提取失败，尝试其他方法

if not class\_code:

if "-" in file\_name:

parts = file\_name.split("-")

# 提取类号部分

if parts[0].isdigit():

class\_code = parts[0]

if len(parts) >= 2 and parts[1].isdigit():

class\_code += "-" + parts[1]

if len(parts) >= 3 and parts[2].isdigit():

class\_code += "-" + parts[2]

else:

# 移除文件扩展名

base\_name = os.path.splitext(file\_name)[0]

# 如果是纯数字，则直接作为类号

if base\_name.isdigit():

class\_code = base\_name

else:

# 尝试提取开头的数字部分作为类号

match = re.match(r'^(\d+)', base\_name)

if match:

class\_code = match.group(1)

else:

class\_code = base\_name

# 从文件路径中提取人名和文件夹名

dir\_path = os.path.dirname(file\_path)

dir\_name = os.path.basename(dir\_path)

# 提取编号 - 获取文件夹名中的数字前缀

match = re.match(r'^(\d+)(.\*)', dir\_name)

if match:

file\_id = match.group(1)

# 处理姓名，去掉数字部分只保留人名

person\_name = match.group(2).strip()

else:

person\_name = dir\_name

# 从Excel获取文件信息（材料名称、日期、页数）

try:

from src.utils.excel\_utils import get\_excel\_info, ExcelFileNotFound

if hasattr(self, 'import\_root\_dir') and self.import\_root\_dir and class\_code:

material\_name, date, page\_count, \_ = get\_excel\_info(self.import\_root\_dir, person\_name, file\_id, class\_code)

logging.info(f"获取Excel信息成功: 类号='{class\_code}', 材料名称='{material\_name}', 日期='{date}', 页数='{page\_count}'")

except Exception as e:

logging.error(f"获取Excel信息失败: {str(e)}")

else:

# 尝试从现有数据中提取

file\_id = file[0] if len(file) > 0 else ""

person\_name = file[1] if len(file) > 1 else ""

class\_code = file[2] if len(file) > 2 else ""

material\_name = file[3] if len(file) > 3 else "" # 使用已有的材料名称

file\_name = file[4] if len(file) > 4 else ""

date = file[5] if len(file) > 5 else "" # 使用已有的日期

page\_count = file[6] if len(file) > 6 else "" # 使用已有的页数

file\_path = file[7] if len(file) > 7 else ""

# 确保文件路径是字符串

if not isinstance(file\_path, str):

file\_path = str(file\_path) if file\_path is not None else ""

# 插入到列表

self.file\_list.insert('', tk.END, values=(

file\_id, person\_name, class\_code, material\_name, file\_name, date, page\_count, file\_path

))

except Exception as e:

logging.error(f"处理文件记录时出错: {str(e)}, 文件: {file}")

# 更新状态栏

self.update\_status\_bar()

def create\_file\_context\_menu(self):

"""创建文件列表右键菜单"""

self.file\_context\_menu = tk.Menu(self.root, tearoff=0)

self.file\_context\_menu.add\_command(label="打开", command=self.open\_selected\_file)

self.file\_context\_menu.add\_command(label="打印", command=self.print\_selected\_file)

# 绑定右键点击事件

self.file\_list.bind("<Button-3>", self.show\_file\_context\_menu)

def show\_file\_context\_menu(self, event):

"""显示右键菜单"""

# 获取鼠标点击位置对应的项

item = self.file\_list.identify\_row(event.y)

if item:

# 选中该项

self.file\_list.selection\_set(item)

# 显示菜单

self.file\_context\_menu.post(event.x\_root, event.y\_root)

def open\_selected\_file(self):

"""打开选中的文件"""

selection = self.file\_list.selection()

if selection:

# 模拟双击事件

self.on\_file\_double\_click(None)

def print\_selected\_file(self):

"""打印选中的文件"""

selection = self.file\_list.selection()

if not selection:

return

# 获取文件路径

item = self.file\_list.item(selection[0])

values = item['values']

if len(values) < 8: # 确保有足够的列

messagebox.showerror("错误", "无法获取文件路径")

return

file\_path = values[7] # 路径在第8列

file\_id = values[0] # 编号在第1列

person\_name = values[1] # 姓名在第2列

if not os.path.exists(file\_path):

messagebox.showerror("错误", f"文件不存在: {file\_path}")

return

try:

# 打印确认

confirm = messagebox.askyesno("打印确认", f"确定要打印文件 {os.path.basename(file\_path)} 吗？")

if not confirm:

return

# 打印文件

if sys.platform.startswith('win'):

# Windows系统使用默认打印机打印

os.startfile(file\_path, "print")

else:

# 其他系统使用lpr命令打印

try:

subprocess.call(['lpr', file\_path])

except:

messagebox.showerror("错误", "当前系统不支持直接打印，请手动打开文件后打印")

# 打开文件

if sys.platform.startswith('darwin'):

subprocess.call(['open', file\_path])

else:

subprocess.call(['xdg-open', file\_path])

# 记录打印日志

if self.current\_user:

self.db.add\_log(

username=self.current\_user[1], # 使用用户名而不是整个用户元组

operation\_type="print",

target\_id=file\_id,

target\_name=person\_name,

details=f"打印文件: {os.path.basename(file\_path)}",

ip\_address=self.local\_ip

)

messagebox.showinfo("打印", "文件已发送到打印机")

except Exception as e:

logging.error(f"打印文件失败: {str(e)}")

messagebox.showerror("错误", f"打印文件失败: {str(e)}")

def on\_file\_double\_click(self, event):

"""双击文件时的处理"""

# 获取选中的项

selection = self.file\_list.selection()

if not selection:

return

# 获取文件路径

item = self.file\_list.item(selection[0])

values = item['values']

if not values or len(values) < 8: # 确保有足够的列

messagebox.showerror("错误", "无法获取文件路径")

return

file\_path = values[7] # 路径在第8列

file\_id = values[0] # 编号在第1列

person\_name = values[1] # 姓名在第2列

if not file\_path or not isinstance(file\_path, str):

messagebox.showerror("错误", "文件路径无效")

return

# 去除可能的空格

file\_path = file\_path.strip()

if not os.path.exists(file\_path):

messagebox.showerror("错误", f"文件不存在: {file\_path}")

return

try:

# 打开文件

logging.info(f"尝试打开文件: {file\_path}")

if sys.platform.startswith('win'):

os.startfile(file\_path)

elif sys.platform.startswith('darwin'):

subprocess.call(['open', file\_path])

else:

subprocess.call(['xdg-open', file\_path])

# 记录查看日志

if hasattr(self, 'current\_user') and self.current\_user:

username = self.current\_user[1] if isinstance(self.current\_user, (list, tuple)) and len(self.current\_user) > 1 else "未知用户"

self.db.add\_log(

username=username,

operation\_type="view",

target\_id=file\_id,

target\_name=person\_name,

details=f"查看文件: {os.path.basename(file\_path)}",

ip\_address=self.local\_ip

)

except Exception as e:

logging.error(f"打开文件失败: {str(e)}")

messagebox.showerror("错误", f"打开文件失败: {str(e)}")

def hash\_password(self, password):

"""对密码进行哈希加密"""

return hashlib.sha256(password.encode()).hexdigest()

def register\_user(self, username, password, real\_name):

"""注册新用户"""

try:

# 检查用户名是否已存在

self.db.cursor.execute('SELECT COUNT(\*) FROM users WHERE username = ?', (username,))

if self.db.cursor.fetchone()[0] > 0:

return False, "用户名已存在"

# 密码加密

hashed\_password = self.hash\_password(password)

# 插入新用户

self.db.cursor.execute(

'INSERT INTO users (username, password, real\_name) VALUES (?, ?, ?)',

(username, hashed\_password, real\_name)

)

self.db.conn.commit()

return True, "注册成功"

except Exception as e:

logging.error(f"注册用户失败: {str(e)}")

return False, f"注册失败: {str(e)}"

def validate\_login(self, username, password):

"""验证用户登录"""

try:

# 查询用户信息

self.db.cursor.execute('SELECT id, username, real\_name, role, password FROM users WHERE username = ?', (username,))

result = self.db.cursor.fetchone()

if result is None:

return False, None # 用户不存在

user\_id, user\_name, real\_name, role, stored\_password = result

# 验证密码

hashed\_password = self.hash\_password(password)

if hashed\_password == stored\_password:

# 记录登录日志

self.db.add\_log(

username=username,

operation\_type="login",

details="用户登录成功",

ip\_address=self.local\_ip

)

# 返回完整的用户信息元组

return True, (user\_id, user\_name, real\_name, role)

else:

return False, None # 密码错误

except Exception as e:

logging.error(f"验证登录失败: {str(e)}")

return False, None

def show\_register\_dialog(self):

"""显示用户注册对话框（仅管理员可用）"""

logging.debug("显示注册对话框")

# 检查权限

if not self.current\_user or self.current\_user[3] != 'admin':

messagebox.showerror("权限错误", "只有管理员可以添加用户")

return

# 创建注册对话框

register\_dialog = tk.Toplevel(self.root)

register\_dialog.title("添加用户")

register\_dialog.geometry("350x250") # 进一步增加对话框尺寸

register\_dialog.resizable(False, False)

register\_dialog.transient(self.root)

register\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(register\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 用户名

ttk.Label(form\_frame, text="用户名:").grid(row=0, column=0, sticky=tk.W, pady=5)

username\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=username\_var, width=20).grid(row=0, column=1, pady=5)

# 密码

ttk.Label(form\_frame, text="密码:").grid(row=1, column=0, sticky=tk.W, pady=5)

password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=password\_var, show="\*", width=20).grid(row=1, column=1, pady=5)

# 确认密码

ttk.Label(form\_frame, text="确认密码:").grid(row=2, column=0, sticky=tk.W, pady=5)

confirm\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=confirm\_password\_var, show="\*", width=20).grid(row=2, column=1, pady=5)

# 真实姓名

ttk.Label(form\_frame, text="真实姓名:").grid(row=3, column=0, sticky=tk.W, pady=5)

real\_name\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=real\_name\_var, width=20).grid(row=3, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=4, column=0, columnspan=2, pady=5)

# 注册按钮

def do\_register():

username = username\_var.get().strip()

password = password\_var.get()

confirm\_password = confirm\_password\_var.get()

real\_name = real\_name\_var.get().strip()

# 验证表单

if not username:

error\_var.set("用户名不能为空")

return

if not password:

error\_var.set("密码不能为空")

return

if password != confirm\_password:

error\_var.set("两次输入的密码不一致")

return

if not real\_name:

error\_var.set("请输入真实姓名")

return

# 注册用户

success, message = self.register\_user(username, password, real\_name)

if success:

messagebox.showinfo("添加成功", message)

register\_dialog.destroy()

else:

error\_var.set(message)

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=5, column=0, columnspan=2, pady=15) # 增加垂直间距

# 添加更大的按钮

ttk.Button(buttons\_frame, text="确定", command=do\_register, width=15).pack(side=tk.LEFT, padx=15) # 增加按钮宽度和间距

ttk.Button(buttons\_frame, text="取消", command=register\_dialog.destroy, width=15).pack(side=tk.LEFT, padx=15) # 增加按钮宽度和间距

# 居中对话框

register\_dialog.update\_idletasks()

width = register\_dialog.winfo\_width()

height = register\_dialog.winfo\_height()

x = (register\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (register\_dialog.winfo\_screenheight() // 2) - (height // 2)

register\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

register\_dialog.wait\_window()

def show\_login\_dialog(self):

"""显示登录对话框"""

logging.debug("显示登录对话框")

# 如果已经登录，显示信息并提供退出选项

if self.current\_user:

result = messagebox.askyesno("已登录",

f"当前已经以 {self.current\_user[2]} 身份登录。\n\n是否要退出登录？")

if result:

self.logout()

return

# 创建登录对话框

login\_dialog = tk.Toplevel(self.root)

login\_dialog.title("用户登录")

login\_dialog.geometry("280x180") # 增加对话框尺寸

login\_dialog.resizable(False, False)

login\_dialog.transient(self.root)

login\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(login\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 用户名

ttk.Label(form\_frame, text="用户名:").grid(row=0, column=0, sticky=tk.W, pady=5)

username\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=username\_var, width=15).grid(row=0, column=1, pady=5)

# 密码

ttk.Label(form\_frame, text="密码:").grid(row=1, column=0, sticky=tk.W, pady=5)

password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=password\_var, show="\*", width=15).grid(row=1, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=2, column=0, columnspan=2, pady=5)

# 登录按钮

def do\_login():

username = username\_var.get().strip()

password = password\_var.get()

# 验证表单

if not username or not password:

error\_var.set("用户名和密码不能为空")

return

# 验证登录

success, user = self.validate\_login(username, password)

if success:

self.current\_user = user

# 显示真实姓名

self.login\_status\_var.set(f"已登录: {user[2]}")

messagebox.showinfo("登录成功", f"欢迎回来，{user[2]}！")

# 更新菜单和权限

self.update\_menu\_by\_role(user[3])

# 确保状态栏显示正确的注册状态

self.update\_status\_bar()

login\_dialog.destroy()

else:

error\_var.set("用户名或密码错误")

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=3, column=0, columnspan=2, pady=10)

# 添加更大的按钮

ttk.Button(buttons\_frame, text="登录", command=do\_login, width=10).pack(side=tk.LEFT, padx=10)

ttk.Button(buttons\_frame, text="取消", command=login\_dialog.destroy, width=10).pack(side=tk.LEFT, padx=10)

# 居中对话框

login\_dialog.update\_idletasks()

width = login\_dialog.winfo\_width()

height = login\_dialog.winfo\_height()

x = (login\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (login\_dialog.winfo\_screenheight() // 2) - (height // 2)

login\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

login\_dialog.wait\_window()

def logout(self):

"""退出登录"""

self.current\_user = None

self.login\_status\_var.set("未登录")

logging.info("用户已退出登录")

# 清空搜索状态和右侧文件列表

self.has\_searched = False

self.current\_search\_name = None

self.current\_search\_id = None

if hasattr(self, 'file\_list'):

self.file\_list.delete(\*self.file\_list.get\_children())

# 更新菜单和权限

self.update\_menu\_by\_role(None)

# 确保状态栏显示正确的注册状态

self.update\_status\_bar()

def show\_help(self):

"""显示帮助信息"""

logging.debug("显示帮助信息")

# 创建帮助窗口

help\_window = tk.Toplevel(self.root)

help\_window.title("使用说明 - 档案检索系统 V1.0")

help\_window.geometry("800x600") # 设置更大的窗口大小

# 添加滚动条

frame = ttk.Frame(help\_window)

frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

# 添加滚动条

scrollbar = ttk.Scrollbar(frame)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

# 添加文本框

text = tk.Text(

frame,

wrap=tk.WORD,

yscrollcommand=scrollbar.set,

font=('Microsoft YaHei', 10),

padx=10,

pady=10

)

text.pack(fill=tk.BOTH, expand=True)

# 配置滚动条

scrollbar.config(command=text.yview)

# 设置帮助文本

help\_text = """档案检索系统使用说明 (v1.0)

1. 系统注册

\* 初次使用请点击"帮助"菜单中的"用户注册"

\* 输入注册码和用户名称进行注册

\* 注册成功后重启程序即可使用全部功能

\* 注册信息保存在程序目录下的registration.dat文件中

2. 用户登录

\* 默认管理员账号: admin 密码: admin123

\* 点击"文件"菜单中的"登录"选项进入登录界面

\* 管理员可以添加新用户和修改密码

3. 搜索功能

\* 登录后在右上角输入姓名或编号进行搜索

\* 不支持模糊搜索，请确保输入完整的姓名或编号

\* 当有同名档案时，系统会提示输入编号进行精确搜索

\* 搜索结果会显示材料名称、日期和页数信息

\* 支持按左侧分类目录筛选搜索结果

4. 文件浏览

\* 左侧为档案分类目录

\* 右侧显示文件列表，包含材料名称、日期和页数

\* 双击文件可以打开查看

\* 支持按列排序（点击列标题）

5. 导入功能

\* 导入档案: 导入档案文件（所有登录用户可用）

\* 支持批量导入（请选择包含PDF图片的上级目录）

\* 自动识别并导入PDF文件

\* 自动从Excel文件读取材料信息

6. 工具功能

\* 打开程序安装目录: 快速访问程序文件

\* 打开档案文件目录: 直接浏览档案文件

\* 打开数据库位置: 查看或备份数据库

\* 清理数据库(仅管理员): 清理重复和无效记录

7. 系统要求

\* 操作系统: Windows 7/10/11

\* 内存: 4GB 或更高

\* 存储空间: 至少500MB可用空间

\* 需要安装Microsoft Office或WPS Office查看文档

"""

# 插入帮助文本

text.insert(tk.END, help\_text)

# 禁用文本编辑

text.config(state=tk.DISABLED)

# 添加关闭按钮

button\_frame = ttk.Frame(help\_window)

button\_frame.pack(fill=tk.X, padx=10, pady=10)

ttk.Button(button\_frame, text="关闭", command=help\_window.destroy).pack(side=tk.RIGHT)

# 设置窗口居中

help\_window.update\_idletasks()

width = help\_window.winfo\_width()

height = help\_window.winfo\_height()

x = (help\_window.winfo\_screenwidth() // 2) - (width // 2)

y = (help\_window.winfo\_screenheight() // 2) - (height // 2)

help\_window.geometry(f'{width}x{height}+{x}+{y}')

# 设置窗口焦点

help\_window.focus\_set()

help\_window.grab\_set()

def show\_about(self):

"""显示关于信息"""

logging.debug("显示关于信息")

# 获取注册状态

is\_registered = hasattr(self, 'is\_registered') and self.is\_registered

registration\_info = f"已注册：{self.current\_org}" if is\_registered else "未注册"

# 创建关于窗口

about\_window = tk.Toplevel(self.root)

about\_window.title("关于")

about\_window.resizable(False, False)

# 设置窗口样式

style = ttk.Style()

# 主框架

main\_frame = ttk.Frame(about\_window, padding=20)

main\_frame.pack(fill=tk.BOTH, expand=True)

# 标题和版本信息

ttk.Label(

main\_frame,

text=f"档案检索系统\n\n版本: {self.version}\n{registration\_info}",

justify=tk.CENTER

).pack(pady=10)

# 分隔线

ttk.Separator(main\_frame, orient=tk.HORIZONTAL).pack(fill=tk.X, pady=10)

# 确保窗口大小合适

about\_window.update\_idletasks()

# 设置窗口大小固定为合适尺寸

about\_window.geometry("300x150")

# 设置窗口居中

about\_window.update\_idletasks()

width = 300

height = 180

x = (about\_window.winfo\_screenwidth() // 2) - (width // 2)

y = (about\_window.winfo\_screenheight() // 2) - (height // 2)

about\_window.geometry(f'{width}x{height}+{x}+{y}')

# 设置窗口焦点

about\_window.focus\_set()

about\_window.grab\_set()

def import\_categories(self):

"""导入分类"""

try:

file\_path = filedialog.askopenfilename(

title="选择分类文件",

filetypes=[("Excel files", "\*.xlsx"), ("All files", "\*.\*")]

)

if not file\_path:

return

logging.info(f"开始导入分类文件: {file\_path}")

# 读取Excel文件的所有列

df = pd.read\_excel(file\_path, engine='openpyxl')

# 清空现有分类

self.tree.delete(\*self.tree.get\_children())

self.db.cursor.execute('DELETE FROM categories')

# 清空映射

self.category\_mapping = {}

# 中文数字映射

chinese\_numbers = {

1: "一", 2: "二", 3: "三", 4: "四", 5: "五",

6: "六", 7: "七", 8: "八", 9: "九", 10: "十"

}

# 用于存储一级分类节点的字典

parent\_nodes = {}

# 手动添加所有一级分类

all\_main\_categories = [

(1, "履历材料"),

(2, "自传材料"),

(3, "鉴定、考核材料"),

(4, "学历学位、职称、学术、培训等材料"),

(5, "政审材料"),

(6, "党团材料"),

(7, "奖励材料"),

(8, "处分材料"),

(9, "工资、任免、出国、会议等材料"),

(10, "其他材料")

]

# 创建一级分类

for main\_num, category\_name in all\_main\_categories:

# 添加中文数字前缀，使用中文顿号"、"

chinese\_num = chinese\_numbers.get(main\_num, str(main\_num))

display\_text = f"{chinese\_num}、{category\_name}"

# 创建树节点

node\_id = self.tree.insert('', 'end', text=display\_text)

parent\_nodes[category\_name] = node\_id

# 存入数据库

self.db.cursor.execute(

'INSERT INTO categories (category, parent\_category, main\_category\_num, sub\_category\_num) VALUES (?, ?, ?, ?)',

(category\_name, None, main\_num, None)

)

# 添加到映射

self.category\_mapping[category\_name] = str(main\_num)

logging.debug(f"插入一级分类: {category\_name}, 主分类号: {main\_num}")

# 第4类的二级分类

subcategories\_4 = [

(4, 1, "学历学位材料"),

(4, 2, "专业技术职务材料"),

(4, 3, "科研学术材料"),

(4, 4, "培训材料")

]

# 第9类的二级分类

subcategories\_9 = [

(9, 1, "工资材料"),

(9, 2, "任免材料"),

(9, 3, "出国（境）审批材料"),

(9, 4, "会议代表材料")

]

# 合并所有二级分类

all\_subcategories = subcategories\_4 + subcategories\_9

# 创建二级分类

for main\_num, sub\_num, subcat\_name in all\_subcategories:

# 查找父分类

parent\_category = None

for name, code in self.category\_mapping.items():

if code == str(main\_num) and not isinstance(name, tuple):

parent\_category = name

break

if parent\_category and parent\_category in parent\_nodes:

# 添加阿拉伯数字前缀，使用中文顿号"、"

display\_text = f"{sub\_num}、{subcat\_name}"

# 创建树节点

child\_id = self.tree.insert(parent\_nodes[parent\_category], 'end', text=display\_text)

# 存入数据库

self.db.cursor.execute(

'INSERT INTO categories (category, parent\_category, main\_category\_num, sub\_category\_num) VALUES (?, ?, ?, ?)',

(subcat\_name, parent\_category, main\_num, sub\_num)

)

# 添加到映射

key = (parent\_category, subcat\_name)

value = f"{main\_num}-{sub\_num}"

self.category\_mapping[key] = value

logging.debug(f"插入二级分类映射: {key} -> {value}")

else:

logging.warning(f"找不到父分类(编码 {main\_num})，无法创建二级分类: {subcat\_name}")

self.db.conn.commit()

# 验证数据库中的记录并记录日志

self.db.cursor.execute('SELECT category, parent\_category, main\_category\_num, sub\_category\_num FROM categories')

all\_categories = self.db.cursor.fetchall()

logging.info(f"数据库中的分类记录: {len(all\_categories)} 条")

# 检查分类（输出到日志）

self.db.cursor.execute('SELECT category, main\_category\_num FROM categories WHERE parent\_category IS NULL ORDER BY main\_category\_num')

main\_categories = self.db.cursor.fetchall()

logging.info(f"共导入 {len(main\_categories)} 个一级分类:")

for cat, num in main\_categories:

chinese\_num = chinese\_numbers.get(num, str(num))

logging.info(f" 一级分类: {chinese\_num}、{cat}, 编码: {num}")

# 检查其子分类

self.db.cursor.execute('SELECT category, sub\_category\_num FROM categories WHERE parent\_category = ? ORDER BY sub\_category\_num', (cat,))

subcats = self.db.cursor.fetchall()

for subcat, subnum in subcats:

logging.info(f" 子分类: {subnum}、{subcat}, 子编码: {num}-{subnum}")

# 打印完整的映射字典

logging.debug("完整的分类映射字典:")

for k, v in self.category\_mapping.items():

if isinstance(k, tuple):

logging.debug(f" 二级分类: {k[0]} -> {k[1]} = {v}")

else:

logging.debug(f" 一级分类: {k} = {v}")

logging.info("分类导入完成")

messagebox.showinfo("成功", "分类导入成功！")

except Exception as e:

logging.error(f"导入分类失败: {str(e)}", exc\_info=True)

messagebox.showerror("错误", f"导入分类失败：{str(e)}")

def import\_files(self):

"""导入文件"""

try:

# 选择文件夹，使用上次的路径作为初始目录

initial\_dir = self.import\_root\_dir if self.import\_root\_dir and os.path.exists(self.import\_root\_dir) else None

folder\_path = filedialog.askdirectory(title="选择人员档案文件夹", initialdir=initial\_dir)

if not folder\_path:

return

# 保存导入目录

self.import\_root\_dir = folder\_path

logging.info(f"设置导入文件根目录: {self.import\_root\_dir}")

# 保存设置

self.save\_settings()

# 清空现有文件记录

self.db.cursor.execute('DELETE FROM person\_files')

# 遍历文件夹

imported\_count = 0

for root, \_, files in os.walk(folder\_path):

dir\_name = os.path.basename(root)

# 从目录名中提取编号和人名

# 正则表达式提取开头的连续数字作为编号

match = re.match(r'^(\d+)(.\*)', dir\_name)

file\_id = ""

person\_name = dir\_name

if match:

file\_id = match.group(1)

# 处理姓名，去掉数字部分只保留人名

person\_name = match.group(2).strip()

logging.debug(f"处理目录: {dir\_name}, 编号: {file\_id}, 人名: {person\_name}")

for file in files:

if file.startswith('.') or file.startswith('~'):

continue

file\_path = os.path.join(root, file)

# 存储绝对路径，而不是相对路径

abs\_path = os.path.abspath(file\_path)

self.db.cursor.execute('''

INSERT INTO person\_files (person\_name, file\_name, file\_path, dir\_name, file\_id)

VALUES (?, ?, ?, ?, ?)

''', (person\_name, file, abs\_path, dir\_name, file\_id))

imported\_count += 1

self.db.conn.commit()

messagebox.showinfo("成功", f"文件导入成功，共导入 {imported\_count} 个文件")

# 刷新文件列表

self.search\_person()

except Exception as e:

self.db.conn.rollback()

logging.error(f"导入文件失败: {str(e)}")

messagebox.showerror("错误", f"导入文件失败：{str(e)}")

def import\_archives(self):

"""导入档案"""

try:

# 使用上次的路径作为初始目录

initial\_dir = self.import\_root\_dir if self.import\_root\_dir and os.path.exists(self.import\_root\_dir) else os.path.join(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))), 'archives')

folder\_path = filedialog.askdirectory(

title="选择档案目录",

initialdir=initial\_dir

)

if not folder\_path:

return

# 保存导入目录

self.import\_root\_dir = folder\_path

logging.info(f"设置导入档案根目录: {self.import\_root\_dir}")

# 保存设置

self.save\_settings()

# 先清理数据库中的重复记录

self.db.cursor.execute('''

DELETE FROM person\_files

WHERE rowid NOT IN (

SELECT MIN(rowid)

FROM person\_files

GROUP BY person\_name, file\_name, file\_path

)

''')

# 遍历选择的目录

imported\_count = 0

file\_count = 0

for person\_folder in os.listdir(folder\_path):

person\_path = os.path.join(folder\_path, person\_folder)

if os.path.isdir(person\_path):

person\_name = person\_folder # 文件夹名即为人名

# 在数据库中记录人员信息

self.db.cursor.execute('''

INSERT OR REPLACE INTO persons (name, folder\_path)

VALUES (?, ?)

''', (person\_name, os.path.abspath(person\_path)))

# 记录该人下的所有文件

for root, \_, files in os.walk(person\_path):

for file in files:

# 跳过临时文件和隐藏文件

if file.startswith('~') or file.startswith('.'):

continue

file\_path = os.path.join(root, file)

abs\_path = os.path.abspath(file\_path)

# 从目录名中提取编号和人名

match = re.match(r'^(\d+)(.\*)', person\_folder)

file\_id = ""

dir\_name = person\_folder

if match:

file\_id = match.group(1)

# 如果人名不是从目录名提取的，则保持原来的人名

# person\_name = match.group(2).strip()

# 检查文件是否已存在

self.db.cursor.execute('''

SELECT COUNT(\*) FROM person\_files

WHERE person\_name = ? AND file\_name = ? AND file\_path = ?

''', (person\_name, file, abs\_path))

if self.db.cursor.fetchone()[0] == 0:

self.db.cursor.execute('''

INSERT INTO person\_files (person\_name, file\_name, file\_path, dir\_name, file\_id)

VALUES (?, ?, ?, ?, ?)

''', (person\_name, file, abs\_path, dir\_name, file\_id))

file\_count += 1

imported\_count += 1

self.db.conn.commit()

messagebox.showinfo("成功", f"成功导入 {imported\_count} 个人员的档案，共 {file\_count} 个文件！")

# 刷新文件列表

self.search\_person()

except Exception as e:

logging.error(f"导入档案失败: {str(e)}")

messagebox.showerror("错误", f"导入失败：{str(e)}")

def search\_person(self):

"""搜索人员档案"""

search\_name = self.search\_name\_var.get().strip()

search\_id = self.search\_id\_var.get().strip()

if not search\_name and not search\_id:

messagebox.showinfo("提示", "请输入姓名或编号进行搜索")

return

# 保存当前搜索条件

self.current\_search\_name = search\_name

self.current\_search\_id = search\_id

self.has\_searched = True

try:

# 构建查询条件

conditions = []

params = []

if search\_name:

conditions.append("person\_name LIKE ?")

params.append(f"%{search\_name}%")

if search\_id:

conditions.append("file\_id LIKE ?")

params.append(f"%{search\_id}%")

# 组合查询条件

query = """

SELECT

id, person\_name, file\_name, file\_path,

category\_id, dir\_name, file\_id

FROM person\_files

WHERE """ + " OR ".join(conditions) + """

ORDER BY person\_name, file\_name

"""

# 执行查询

self.db.cursor.execute(query, params)

results = self.db.cursor.fetchall()

# 检查是否有结果

if not results:

messagebox.showinfo("提示", "未找到匹配的档案")

self.file\_list.delete(\*self.file\_list.get\_children())

self.search\_result\_var.set("搜索结果: 0 个文件")

return

# 检查是否有重名情况

if search\_name and not search\_id:

names = set(row[1] for row in results) # 获取所有人名

if len(names) > 1:

self.has\_duplicate\_names = True

messagebox.showwarning("提示", "存在多个同名人员，请输入编号进一步筛选")

else:

self.has\_duplicate\_names = False

else:

self.has\_duplicate\_names = False

# 检查当前是否有选中的分类

selected\_items = self.tree.selection()

if selected\_items:

# 有选中的分类，需要进行分类筛选

selected\_item = selected\_items[0]

selected\_text = self.tree.item(selected\_item)['text']

# 获取分类代码

category\_code = self.tree.item(selected\_item)['tags'][0] if self.tree.item(selected\_item)['tags'] else ""

# 如果没有从tag中获取到分类代码，则从映射中获取

if not category\_code:

# 从显示文本中提取原始分类名称（去除数字前缀）

if '、' in selected\_text:

category\_name = selected\_text.split('、', 1)[1]

else:

category\_name = selected\_text

# 判断是一级分类还是二级分类

parent\_item = self.tree.parent(selected\_item)

is\_secondary = parent\_item != ''

if not is\_secondary and category\_name in self.category\_mapping:

category\_code = self.category\_mapping[category\_name]

elif is\_secondary:

parent\_text = self.tree.item(parent\_item)['text']

if '、' in parent\_text:

parent\_name = parent\_text.split('、', 1)[1]

else:

parent\_name = parent\_text

if (parent\_name, category\_name) in self.category\_mapping:

category\_code = self.category\_mapping[(parent\_name, category\_name)]

if category\_code:

logging.info(f"搜索时应用分类筛选: {selected\_text} (代码: {category\_code})")

# 导入match\_category\_code函数

try:

from src.main import match\_category\_code

except ImportError:

# 如果导入失败，定义一个本地版本

def match\_category\_code(row\_category\_code, selected\_category\_id):

if not row\_category\_code:

return False

row\_category\_code = str(row\_category\_code).strip()

selected\_category\_id = str(selected\_category\_id).strip()

return row\_category\_code.startswith(f"{selected\_category\_id}-") or row\_category\_code == selected\_category\_id

# 使用match\_category\_code函数过滤结果

filtered\_results = []

for file in results:

file\_name = file[2] if len(file) > 2 else ""

file\_category\_id = str(file[4]) if len(file) > 4 and file[4] is not None else "" # 确保转换为字符串

# 首先检查category\_id字段是否匹配

if file\_category\_id and match\_category\_code(file\_category\_id, category\_code):

filtered\_results.append(file)

continue

# 然后从文件名中提取类号并检查是否匹配

extracted\_category = self.extract\_category\_num(file\_name)

if extracted\_category and match\_category\_code(extracted\_category, category\_code):

filtered\_results.append(file)

continue

# 移除这个不精确的匹配条件，它会导致过度匹配

# 例如，当category\_code为"1"时，会匹配所有包含"1"的文件名

# 如"4-1-1"、"5-1"等，而不仅仅是"1-"开头的文件

# if category\_code and file\_name and str(category\_code) in file\_name:

# filtered\_results.append(file)

# 使用过滤后的结果

results = filtered\_results

logging.info(f"应用分类筛选后的结果数量: {len(results)}")

if not results:

messagebox.showinfo("提示", f"未找到属于分类 '{selected\_text}' 的档案")

self.file\_list.delete(\*self.file\_list.get\_children())

self.search\_result\_var.set("搜索结果: 0 个文件")

return

# 更新文件列表

self.update\_file\_list(results)

# 记录搜索日志

if self.current\_user:

details = f"搜索条件: 姓名={search\_name}, 编号={search\_id}, 结果数量={len(results)}"

self.db.add\_log(

username=self.current\_user[1], # 使用用户名而不是整个用户元组

operation\_type="search",

target\_name=search\_name or search\_id,

details=details,

ip\_address=self.local\_ip

)

# 更新状态栏

self.search\_result\_var.set(f"搜索结果: {len(results)} 个文件")

except Exception as e:

logging.error(f"搜索失败: {str(e)}")

messagebox.showerror("错误", f"搜索失败: {str(e)}")

def extract\_category\_num(self, filename):

"""从文件名中提取分类号"""

if not filename or not isinstance(filename, str):

return None

# 移除文件扩展名

base\_name = os.path.splitext(filename)[0]

# 文件名格式可能是: 3-1-1-会议记录.pdf 或 3-1-会议记录.pdf

parts = base\_name.split('-')

if not parts or not parts[0].isdigit():

return None

# 构建完整类号 - 包括所有数字部分

class\_parts = []

for part in parts:

if part.isdigit():

class\_parts.append(part)

else:

break

if not class\_parts:

return None

# 返回完整的类号格式，如 "1-1", "4-1-2"

return "-".join(class\_parts)

def on\_category\_selected(self, event):

"""处理分类选择事件"""

selected\_items = self.tree.selection()

if not selected\_items:

return

# 未登录时不显示任何内容

if not hasattr(self, 'current\_user') or self.current\_user is None:

self.file\_list.delete(\*self.file\_list.get\_children())

logging.info("未登录状态，点击分类不显示文件")

return

# 判断是否进行过搜索，如果没有搜索过则右侧列表为空

if not hasattr(self, 'has\_searched') or not self.has\_searched:

self.file\_list.delete(\*self.file\_list.get\_children())

logging.info("未进行搜索，不显示文件")

return

selected\_item = selected\_items[0]

selected\_text = self.tree.item(selected\_item)['text']

parent\_item = self.tree.parent(selected\_item)

# 从显示文本中提取原始分类名称（去除数字前缀）

# 例如：从"一、履历材料"提取"履历材料"

if '、' in selected\_text:

category\_name = selected\_text.split('、', 1)[1]

else:

category\_name = selected\_text

logging.info(f"选择的分类: {selected\_text}, 提取的分类名: {category\_name}")

# 清空文件列表

self.file\_list.delete(\*self.file\_list.get\_children())

# 判断是一级分类还是二级分类

is\_secondary = parent\_item != ''

# 获取分类代码

category\_code = ""

# 从树节点的tag中获取分类代码

category\_code = self.tree.item(selected\_item)['tags'][0] if self.tree.item(selected\_item)['tags'] else ""

# 如果没有从tag中获取到分类代码，则从映射中获取

if not category\_code:

# 对于一级分类，直接从映射中获取

if not is\_secondary and category\_name in self.category\_mapping:

category\_code = self.category\_mapping[category\_name]

# 对于二级分类，需要找到父分类名称

elif is\_secondary:

parent\_text = self.tree.item(parent\_item)['text']

if '、' in parent\_text:

parent\_name = parent\_text.split('、', 1)[1]

else:

parent\_name = parent\_text

# 尝试从映射中获取二级分类代码

if (parent\_name, category\_name) in self.category\_mapping:

category\_code = self.category\_mapping[(parent\_name, category\_name)]

logging.info(f"分类代码: {category\_code}, 是否为二级分类: {is\_secondary}")

# 检查是否有搜索名称

if not hasattr(self, 'current\_search\_name') or not self.current\_search\_name:

self.status\_var.set("请先输入姓名进行搜索")

return

# 检查是否有重名但未提供ID

if hasattr(self, 'has\_duplicate\_names') and self.has\_duplicate\_names and not self.current\_search\_id:

self.status\_var.set("检测到重名，请提供编号以区分")

return

# 导入match\_category\_code函数

try:

from src.main import match\_category\_code

except ImportError:

# 如果导入失败，定义一个本地版本

def match\_category\_code(row\_category\_code, selected\_category\_id):

if not row\_category\_code:

return False

row\_category\_code = str(row\_category\_code).strip()

selected\_category\_id = str(selected\_category\_id).strip()

return row\_category\_code.startswith(f"{selected\_category\_id}-") or row\_category\_code == selected\_category\_id

# 构建SQL查询 - 获取所有符合搜索条件的文件，后续再进行分类筛选

files = []

# 获取数据库连接和游标

cursor = self.db.cursor # 使用数据库对象的游标

try:

# 构建查询条件

conditions = []

params = []

if self.current\_search\_name:

conditions.append("person\_name LIKE ?")

params.append(f"%{self.current\_search\_name}%")

if self.current\_search\_id:

conditions.append("file\_id LIKE ?")

params.append(f"%{self.current\_search\_id}%")

# 组合查询条件

query = """

SELECT

id, person\_name, file\_name, file\_path,

category\_id, dir\_name, file\_id

FROM person\_files

WHERE """ + " OR ".join(conditions) + """

ORDER BY person\_name, file\_name

"""

# 执行查询

cursor.execute(query, params)

all\_files = cursor.fetchall()

logging.info(f"搜索到的所有文件数量: {len(all\_files)}")

# 使用match\_category\_code函数过滤结果，只保留与选定类别匹配的文件

filtered\_files = []

for file in all\_files:

file\_name = file[2] if len(file) > 2 else ""

file\_category\_id = str(file[4]) if len(file) > 4 and file[4] is not None else "" # 确保转换为字符串

# 首先检查category\_id字段是否匹配

if file\_category\_id and match\_category\_code(file\_category\_id, category\_code):

filtered\_files.append(file)

continue

# 然后从文件名中提取类号并检查是否匹配

extracted\_category = self.extract\_category\_num(file\_name)

if extracted\_category and match\_category\_code(extracted\_category, category\_code):

filtered\_files.append(file)

continue

# 移除这个不精确的匹配条件，它会导致过度匹配

# 例如，当category\_code为"1"时，会匹配所有包含"1"的文件名

# 如"4-1-1"、"5-1"等，而不仅仅是"1-"开头的文件

# if category\_code and file\_name and str(category\_code) in file\_name:

# filtered\_files.append(file)

logging.info(f"分类筛选后的文件数量: {len(filtered\_files)}")

# 记录过滤后的文件名

for i, file in enumerate(filtered\_files[:5]): # 只记录前5个文件

if len(file) > 2:

logging.info(f"过滤后的文件 {i+1}: {file[2]}")

# 如果没有找到匹配的文件

if not filtered\_files:

self.status\_var.set(f"未找到属于分类 '{selected\_text}' 的文件")

return

# 对于搜索到的文件，获取Excel信息并更新文件列表

excel\_files = []

for file in filtered\_files:

try:

file\_id = file[6] # file\_id

person\_name = file[1] # person\_name

file\_name = file[2] # file\_name

# 从文件名中提取完整类号

full\_category\_code = self.extract\_category\_num(file\_name)

# 如果无法从文件名提取，则使用默认格式

if not full\_category\_code:

# 如果是第四大类或第九大类，需要确保使用二级类号格式

if category\_code in ['4', '9']:

# 默认使用 "类别-1-1" 格式（如 "4-1-1"）

full\_category\_code = f"{category\_code}-1-1"

else:

# 默认使用 "类别-1" 格式（如 "1-1"）

full\_category\_code = f"{category\_code}-1"

logging.info(f"文件 {file\_name} 提取的完整类号: {full\_category\_code}")

# 使用完整类号替换简单类别号

file\_list = list(file)

file\_list[4] = full\_category\_code # 替换category\_id中的类号

# 从Excel获取文件信息（材料名称、日期、页数）

try:

from src.utils.excel\_utils import get\_excel\_info, ExcelFileNotFound

if hasattr(self, 'import\_root\_dir') and self.import\_root\_dir and full\_category\_code:

material\_name, file\_date, page\_count, \_ = get\_excel\_info(self.import\_root\_dir, person\_name, file\_id, full\_category\_code)

if material\_name or file\_date or page\_count: # 如果获取到了信息

logging.info(f"获取Excel信息成功: 类号='{full\_category\_code}', 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

# 更新材料名称、日期和页数

if material\_name:

file\_list.append(material\_name)

if file\_date:

file\_list.append(file\_date)

if page\_count:

file\_list.append(page\_count)

except Exception as e:

logging.error(f"获取Excel信息失败: {str(e)}")

excel\_files.append(tuple(file\_list))

except Exception as e:

logging.error(f"处理文件类号时出错: {str(e)}, 文件: {file}")

excel\_files.append(file) # 出错时使用原始文件信息

# 直接使用self.update\_file\_list方法更新文件列表

self.update\_file\_list(excel\_files)

# 更新状态栏

self.status\_var.set(f"找到 {len(excel\_files)} 个属于分类 '{selected\_text}' 的文件")

self.search\_result\_var.set(f"搜索结果: {len(excel\_files)} 个文件")

except Exception as e:

logging.error(f"查询分类文件时出错: {str(e)}")

self.status\_var.set(f"查询失败: {str(e)}")

# 不需要关闭游标或连接，因为我们使用的是数据库对象的游标

def update\_category\_tree(self):

"""更新分类树显示"""

self.tree.delete(\*self.tree.get\_children()) # 清空现有分类

# 从数据库获取分类

self.db.cursor.execute('SELECT category FROM categories')

categories = self.db.cursor.fetchall()

for category in categories:

self.tree.insert('', 'end', text=category[0])

def show\_operation\_logs(self):

"""显示操作日志"""

if not self.current\_user or not hasattr(self.db, 'get\_logs'):

messagebox.showerror("错误", "请先登录或系统不支持日志功能")

return

# 创建日志查询窗口

log\_window = tk.Toplevel(self.root)

log\_window.title("操作日志查询")

log\_window.geometry("900x600")

log\_window.minsize(800, 500)

# 设置窗口图标

try:

from src.utils.paths import get\_icon\_path

icon\_path = get\_icon\_path()

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

log\_window.iconbitmap(icon\_path)

except Exception as e:

logging.warning(f"设置图标失败: {str(e)}")

# 创建查询条件框架

query\_frame = ttk.Frame(log\_window, padding="10 10 10 10")

query\_frame.pack(fill=tk.X)

# 用户名筛选

ttk.Label(query\_frame, text="用户名:").grid(row=0, column=0, padx=5, pady=5, sticky=tk.W)

username\_var = tk.StringVar()

username\_entry = ttk.Entry(query\_frame, textvariable=username\_var, width=15)

username\_entry.grid(row=0, column=1, padx=5, pady=5, sticky=tk.W)

# 操作类型筛选

ttk.Label(query\_frame, text="操作类型:").grid(row=0, column=2, padx=5, pady=5, sticky=tk.W)

operation\_type\_var = tk.StringVar()

operation\_types = ["", "login", "search", "view", "print"]

operation\_type\_combo = ttk.Combobox(query\_frame, textvariable=operation\_type\_var, values=operation\_types, width=10)

operation\_type\_combo.grid(row=0, column=3, padx=5, pady=5, sticky=tk.W)

# 日期范围筛选

ttk.Label(query\_frame, text="开始日期:").grid(row=0, column=4, padx=5, pady=5, sticky=tk.W)

start\_date\_var = tk.StringVar()

start\_date\_entry = ttk.Entry(query\_frame, textvariable=start\_date\_var, width=12)

start\_date\_entry.grid(row=0, column=5, padx=5, pady=5, sticky=tk.W)

ttk.Label(query\_frame, text="结束日期:").grid(row=0, column=6, padx=5, pady=5, sticky=tk.W)

end\_date\_var = tk.StringVar()

end\_date\_entry = ttk.Entry(query\_frame, textvariable=end\_date\_var, width=12)

end\_date\_entry.grid(row=0, column=7, padx=5, pady=5, sticky=tk.W)

# 设置默认日期为今天

today = datetime.date.today().strftime("%Y-%m-%d")

start\_date\_var.set(today)

end\_date\_var.set(today)

# 创建日志列表

log\_frame = ttk.Frame(log\_window)

log\_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

# 创建Treeview显示日志

columns = ('id', 'username', 'operation\_type', 'target\_id', 'target\_name', 'details', 'operation\_time', 'ip\_address')

log\_tree = ttk.Treeview(log\_frame, columns=columns, show='headings')

# 设置列标题

log\_tree.heading('id', text='ID')

log\_tree.heading('username', text='用户名')

log\_tree.heading('operation\_type', text='操作类型')

log\_tree.heading('target\_id', text='目标ID')

log\_tree.heading('target\_name', text='目标名称')

log\_tree.heading('details', text='详情')

log\_tree.heading('operation\_time', text='操作时间')

log\_tree.heading('ip\_address', text='IP地址')

# 设置列宽

log\_tree.column('id', width=50)

log\_tree.column('username', width=80)

log\_tree.column('operation\_type', width=80)

log\_tree.column('target\_id', width=80)

log\_tree.column('target\_name', width=100)

log\_tree.column('details', width=200)

log\_tree.column('operation\_time', width=150)

log\_tree.column('ip\_address', width=120)

# 添加滚动条

scrollbar = ttk.Scrollbar(log\_frame, orient=tk.VERTICAL, command=log\_tree.yview)

log\_tree.configure(yscrollcommand=scrollbar.set)

log\_tree.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

# 按钮框架

button\_frame = ttk.Frame(log\_window, padding="10 5 10 10")

button\_frame.pack(fill=tk.X)

# 查询按钮

def query\_logs():

username = username\_var.get().strip()

operation\_type = operation\_type\_var.get()

start\_date = start\_date\_var.get().strip()

end\_date = end\_date\_var.get().strip()

# 清空现有数据

for item in log\_tree.get\_children():

log\_tree.delete(item)

# 查询日志

logs = self.db.get\_logs(

username=username if username else None,

operation\_type=operation\_type if operation\_type else None,

start\_date=start\_date if start\_date else None,

end\_date=end\_date if end\_date else None

)

# 显示日志

for log in logs:

log\_tree.insert('', tk.END, values=log)

# 更新状态

status\_var.set(f"共找到 {len(logs)} 条日志记录")

query\_btn = ttk.Button(button\_frame, text="查询", command=query\_logs)

query\_btn.pack(side=tk.LEFT, padx=5)

# 导出按钮

def export\_displayed\_logs():

# 获取当前显示的所有日志

logs = []

for item\_id in log\_tree.get\_children():

item = log\_tree.item(item\_id)

logs.append(item['values'])

if not logs:

messagebox.showinfo("提示", "没有可导出的日志")

return

# 选择保存路径

file\_path = filedialog.asksaveasfilename(

defaultextension=".xlsx",

filetypes=[("Excel文件", "\*.xlsx"), ("所有文件", "\*.\*")],

title="保存日志文件"

)

if not file\_path:

return

try:

# 创建DataFrame

df = pd.DataFrame(logs, columns=columns)

# 保存为Excel

df.to\_excel(file\_path, index=False)

messagebox.showinfo("成功", f"日志已导出到: {file\_path}")

except Exception as e:

logging.error(f"导出日志失败: {str(e)}")

messagebox.showerror("错误", f"导出失败: {str(e)}")

export\_btn = ttk.Button(button\_frame, text="导出当前结果", command=export\_displayed\_logs)

export\_btn.pack(side=tk.LEFT, padx=5)

# 关闭按钮

close\_btn = ttk.Button(button\_frame, text="关闭", command=log\_window.destroy)

close\_btn.pack(side=tk.RIGHT, padx=5)

# 状态栏

status\_var = tk.StringVar()

status\_var.set("请设置查询条件并点击查询按钮")

status\_bar = ttk.Label(log\_window, textvariable=status\_var, relief=tk.SUNKEN, anchor=tk.W)

status\_bar.pack(side=tk.BOTTOM, fill=tk.X)

# 初始加载今天的日志

query\_logs()

# 设置窗口为模态

log\_window.transient(self.root)

log\_window.grab\_set()

self.root.wait\_window(log\_window)

def export\_logs(self):

"""导出日志"""

if not self.current\_user:

messagebox.showerror("错误", "请先登录")

return

# 创建导出窗口

export\_window = tk.Toplevel(self.root)

export\_window.title("导出操作日志")

export\_window.geometry("400x300")

export\_window.minsize(400, 300)

# 设置窗口图标

try:

from src.utils.paths import get\_icon\_path

icon\_path = get\_icon\_path()

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

export\_window.iconbitmap(icon\_path)

except Exception as e:

logging.warning(f"设置图标失败: {str(e)}")

# 创建查询条件框架

query\_frame = ttk.Frame(export\_window, padding="10 10 10 10")

query\_frame.pack(fill=tk.X)

# 用户名筛选

ttk.Label(query\_frame, text="用户名:").grid(row=0, column=0, padx=5, pady=5, sticky=tk.W)

username\_var = tk.StringVar()

username\_entry = ttk.Entry(query\_frame, textvariable=username\_var, width=15)

username\_entry.grid(row=0, column=1, padx=5, pady=5, sticky=tk.W)

# 操作类型筛选

ttk.Label(query\_frame, text="操作类型:").grid(row=1, column=0, padx=5, pady=5, sticky=tk.W)

operation\_type\_var = tk.StringVar()

operation\_types = ["", "login", "search", "view", "print"]

operation\_type\_combo = ttk.Combobox(query\_frame, textvariable=operation\_type\_var, values=operation\_types, width=15)

operation\_type\_combo.grid(row=1, column=1, padx=5, pady=5, sticky=tk.W)

# 日期范围筛选

ttk.Label(query\_frame, text="开始日期:").grid(row=2, column=0, padx=5, pady=5, sticky=tk.W)

start\_date\_var = tk.StringVar()

start\_date\_entry = ttk.Entry(query\_frame, textvariable=start\_date\_var, width=15)

start\_date\_entry.grid(row=2, column=1, padx=5, pady=5, sticky=tk.W)

ttk.Label(query\_frame, text="结束日期:").grid(row=3, column=0, padx=5, pady=5, sticky=tk.W)

end\_date\_var = tk.StringVar()

end\_date\_entry = ttk.Entry(query\_frame, textvariable=end\_date\_var, width=15)

end\_date\_entry.grid(row=3, column=1, padx=5, pady=5, sticky=tk.W)

# 设置默认日期为本月第一天和今天

today = datetime.date.today()

first\_day = today.replace(day=1).strftime("%Y-%m-%d")

today\_str = today.strftime("%Y-%m-%d")

start\_date\_var.set(first\_day)

end\_date\_var.set(today\_str)

# 按钮框架

button\_frame = ttk.Frame(export\_window, padding="10 5 10 10")

button\_frame.pack(fill=tk.X, pady=10)

# 导出按钮

def do\_export():

username = username\_var.get().strip()

operation\_type = operation\_type\_var.get()

start\_date = start\_date\_var.get().strip()

end\_date = end\_date\_var.get().strip()

# 选择保存路径

file\_path = filedialog.asksaveasfilename(

defaultextension=".xlsx",

filetypes=[("Excel文件", "\*.xlsx"), ("所有文件", "\*.\*")],

title="保存日志文件"

)

if not file\_path:

return

try:

# 查询日志

logs = self.db.get\_logs(

username=username if username else None,

operation\_type=operation\_type if operation\_type else None,

start\_date=start\_date if start\_date else None,

end\_date=end\_date if end\_date else None,

limit=10000 # 导出更多记录

)

if not logs:

messagebox.showinfo("提示", "没有符合条件的日志记录")

return

# 创建DataFrame

columns = ['id', 'username', 'operation\_type', 'target\_id', 'target\_name', 'details', 'operation\_time', 'ip\_address']

df = pd.DataFrame(logs, columns=columns)

# 保存为Excel

df.to\_excel(file\_path, index=False)

messagebox.showinfo("成功", f"已导出 {len(logs)} 条日志记录到: {file\_path}")

# 记录导出日志的操作

self.db.add\_log(

username=self.current\_user[1], # 使用用户名而不是整个用户元组

operation\_type="export",

details=f"导出日志记录: {len(logs)} 条",

ip\_address=self.local\_ip

)

export\_window.destroy()

except Exception as e:

logging.error(f"导出日志失败: {str(e)}")

messagebox.showerror("错误", f"导出失败: {str(e)}")

export\_btn = ttk.Button(button\_frame, text="导出", command=do\_export)

export\_btn.pack(side=tk.LEFT, padx=5)

# 取消按钮

cancel\_btn = ttk.Button(button\_frame, text="取消", command=export\_window.destroy)

cancel\_btn.pack(side=tk.RIGHT, padx=5)

# 设置窗口为模态

export\_window.transient(self.root)

export\_window.grab\_set()

self.root.wait\_window(export\_window)

def load\_files\_from\_db(self):

"""从数据库加载所有文件"""

try:

self.db.cursor.execute('''

SELECT file\_name, file\_path

FROM person\_files

''')

self.all\_files = self.db.cursor.fetchall()

except Exception as e:

logging.error(f"加载文件失败: {str(e)}")

self.all\_files = []

def open\_install\_directory(self):

"""打开程序安装目录"""

try:

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

logging.info(f"打开程序安装目录: {install\_dir}")

if os.path.exists(install\_dir):

# 使用系统默认的文件浏览器打开目录

os.startfile(install\_dir)

else:

messagebox.showerror("错误", "程序安装目录不存在")

except Exception as e:

logging.error(f"打开程序安装目录失败: {str(e)}")

messagebox.showerror("错误", f"打开程序安装目录失败: {str(e)}")

def open\_archive\_directory(self):

"""打开档案文件目录"""

try:

# 使用当前导入目录，如果存在且有效

archive\_dir = self.import\_root\_dir

# 如果目录不存在或未设置，提示用户选择

if not archive\_dir or not os.path.exists(archive\_dir):

# 如果没有设置或目录不存在，询问用户选择目录

messagebox.showinfo("提示", "档案文件目录未设置或不存在，请选择档案文件目录")

archive\_dir = filedialog.askdirectory(title="选择档案文件目录")

if archive\_dir:

# 更新导入目录并保存设置

self.import\_root\_dir = archive\_dir

self.save\_settings()

else:

return

logging.info(f"打开档案文件目录: {archive\_dir}")

# 使用系统默认的文件浏览器打开目录

os.startfile(archive\_dir)

except Exception as e:

logging.error(f"打开档案文件目录失败: {str(e)}")

messagebox.showerror("错误", f"打开档案文件目录失败: {str(e)}")

def open\_database\_location(self):

"""打开数据库位置"""

try:

# 获取数据库文件路径

if hasattr(self.db, 'db\_path') and self.db.db\_path:

db\_dir = os.path.dirname(self.db.db\_path)

else:

# 如果没有明确的数据库路径，使用默认位置

db\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'data')

logging.info(f"打开数据库位置: {db\_dir}")

if os.path.exists(db\_dir):

# 使用系统默认的文件浏览器打开目录

os.startfile(db\_dir)

else:

messagebox.showerror("错误", "数据库目录不存在")

except Exception as e:

logging.error(f"打开数据库位置失败: {str(e)}")

messagebox.showerror("错误", f"打开数据库位置失败: {str(e)}")

def cleanup\_database(self):

"""清理数据库中的重复记录和无效记录"""

try:

# 删除重复记录

self.db.cursor.execute('''

DELETE FROM person\_files

WHERE rowid NOT IN (

SELECT MIN(rowid)

FROM person\_files

GROUP BY person\_name, file\_name, file\_path

)

''')

# 删除不存在的文件记录

self.db.cursor.execute('SELECT file\_path FROM person\_files')

for (file\_path,) in self.db.cursor.fetchall():

if not os.path.exists(file\_path):

self.db.cursor.execute('DELETE FROM person\_files WHERE file\_path = ?', (file\_path,))

self.db.conn.commit()

messagebox.showinfo("成功", "数据库清理完成！")

except Exception as e:

logging.error(f"数据库清理失败: {str(e)}")

messagebox.showerror("错误", f"清理失败：{str(e)}")

def init\_data(self):

"""初始化数据：加载分类树和文件列表"""

try:

# 加载分类树（不依赖数据库）

self.load\_categories\_from\_db()

logging.info("已加载分类树结构")

# 检查文件记录是否存在

self.db.cursor.execute('SELECT COUNT(\*) FROM person\_files')

files\_count = self.db.cursor.fetchone()[0]

if files\_count > 0:

# 如果存在文件记录，直接加载

self.load\_files\_from\_db()

logging.info(f"从数据库加载了 {files\_count} 条文件记录")

except Exception as e:

logging.error(f"初始化数据失败: {str(e)}", exc\_info=True)

messagebox.showerror("错误", f"初始化数据失败：{str(e)}")

def load\_categories\_from\_db(self):

"""加载分类树并构建映射（不依赖数据库，直接在代码中写死）"""

try:

# 清空现有树和映射

self.tree.delete(\*self.tree.get\_children())

self.category\_mapping = {}

# 中文数字映射

chinese\_numbers = {

1: "一", 2: "二", 3: "三", 4: "四", 5: "五",

6: "六", 7: "七", 8: "八", 9: "九", 10: "十"

}

# 手动定义所有一级分类

all\_main\_categories = [

(1, "履历材料"),

(2, "自传材料"),

(3, "鉴定、考核材料"),

(4, "学历学位、职称、学术、培训等材料"),

(5, "政审材料"),

(6, "党团材料"),

(7, "奖励材料"),

(8, "处分材料"),

(9, "工资、任免、出国、会议等材料"),

(10, "其他材料")

]

# 创建一级分类节点和映射

parent\_nodes = {}

for main\_num, category in all\_main\_categories:

# 添加中文数字前缀，使用中文顿号"、"

chinese\_num = chinese\_numbers.get(main\_num, str(main\_num))

display\_text = f"{chinese\_num}、{category}"

# 创建树节点，添加tags属性包含分类代码

node\_id = self.tree.insert('', 'end', text=display\_text, tags=[str(main\_num)])

parent\_nodes[category] = node\_id

# 添加一级分类映射

self.category\_mapping[category] = str(main\_num)

logging.debug(f"添加一级分类映射: {category} -> {main\_num}")

# 手动定义第4类的二级分类

subcategories\_4 = [

(4, 1, "学历学位材料"),

(4, 2, "专业技术职务材料"),

(4, 3, "科研学术材料"),

(4, 4, "培训材料")

]

# 手动定义第9类的二级分类

subcategories\_9 = [

(9, 1, "工资材料"),

(9, 2, "任免材料"),

(9, 3, "出国（境）审批材料"),

(9, 4, "会议代表材料")

]

# 合并所有二级分类

all\_subcategories = subcategories\_4 + subcategories\_9

# 创建二级分类节点和映射

for main\_num, sub\_num, category in all\_subcategories:

# 查找父分类

parent\_category = None

for name, code in self.category\_mapping.items():

if code == str(main\_num) and not isinstance(name, tuple):

parent\_category = name

break

if parent\_category and parent\_category in parent\_nodes:

# 添加阿拉伯数字前缀，使用中文顿号"、"

display\_text = f"{sub\_num}、{category}"

# 创建树节点，添加tags属性包含完整分类代码

category\_code = f"{main\_num}-{sub\_num}"

child\_id = self.tree.insert(parent\_nodes[parent\_category], 'end', text=display\_text, tags=[category\_code])

# 添加二级分类映射

key = (parent\_category, category)

value = category\_code

self.category\_mapping[key] = value

logging.debug(f"添加二级分类映射: {key} -> {value}")

# 打印完整的映射字典

logging.debug("完整的分类映射字典:")

for k, v in self.category\_mapping.items():

if isinstance(k, tuple):

logging.debug(f" 二级分类: {k[0]} -> {k[1]} = {v}")

else:

logging.debug(f" 一级分类: {k} = {v}")

except Exception as e:

logging.error(f"加载分类树失败: {str(e)}", exc\_info=True)

# 如果出错，不抛出异常，只记录日志

pass

def load\_settings(self):

"""加载用户设置"""

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

try:

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

settings = json.load(f)

logging.info(f"成功加载用户设置: {settings}")

return settings

except Exception as e:

logging.error(f"加载用户设置失败: {str(e)}")

return {}

else:

logging.info("未找到用户设置文件，将使用默认设置")

return {}

def save\_settings(self):

"""保存用户设置"""

try:

settings = {

'import\_root\_dir': self.import\_root\_dir

}

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

json.dump(settings, f, ensure\_ascii=False, indent=2)

logging.info(f"成功保存用户设置: {settings}")

except Exception as e:

logging.error(f"保存用户设置失败: {str(e)}")

def show\_change\_password\_dialog(self):

"""显示修改密码对话框"""

# 检查权限

if not self.current\_user or self.current\_user[3] != 'admin':

messagebox.showerror("权限错误", "只有管理员可以修改密码")

return

# 创建修改密码对话框

change\_pwd\_dialog = tk.Toplevel(self.root)

change\_pwd\_dialog.title("修改密码")

change\_pwd\_dialog.geometry("320x200") # 增加对话框尺寸

change\_pwd\_dialog.resizable(False, False)

change\_pwd\_dialog.transient(self.root)

change\_pwd\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(change\_pwd\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 旧密码

ttk.Label(form\_frame, text="当前密码:").grid(row=0, column=0, sticky=tk.W, pady=5)

old\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=old\_password\_var, show="\*", width=20).grid(row=0, column=1, pady=5)

# 新密码

ttk.Label(form\_frame, text="新密码:").grid(row=1, column=0, sticky=tk.W, pady=5)

new\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=new\_password\_var, show="\*", width=20).grid(row=1, column=1, pady=5)

# 确认新密码

ttk.Label(form\_frame, text="确认新密码:").grid(row=2, column=0, sticky=tk.W, pady=5)

confirm\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=confirm\_password\_var, show="\*", width=20).grid(row=2, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=3, column=0, columnspan=2, pady=5)

# 修改密码按钮

def do\_change\_password():

old\_password = old\_password\_var.get()

new\_password = new\_password\_var.get()

confirm\_password = confirm\_password\_var.get()

# 验证表单

if not old\_password or not new\_password or not confirm\_password:

error\_var.set("所有密码字段不能为空")

return

if new\_password != confirm\_password:

error\_var.set("两次输入的新密码不一致")

return

# 验证旧密码

success, \_ = self.validate\_login(self.current\_user[1], old\_password)

if not success:

error\_var.set("当前密码错误")

return

# 修改密码

try:

hashed\_password = self.hash\_password(new\_password)

self.db.cursor.execute(

'UPDATE users SET password = ? WHERE id = ?',

(hashed\_password, self.current\_user[0])

)

self.db.conn.commit()

messagebox.showinfo("成功", "密码修改成功！")

change\_pwd\_dialog.destroy()

except Exception as e:

error\_var.set(f"修改失败: {str(e)}")

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=4, column=0, columnspan=2, pady=10)

# 添加更大的按钮

ttk.Button(buttons\_frame, text="确定", command=do\_change\_password, width=10).pack(side=tk.LEFT, padx=10)

ttk.Button(buttons\_frame, text="取消", command=change\_pwd\_dialog.destroy, width=10).pack(side=tk.LEFT, padx=10)

# 居中对话框

change\_pwd\_dialog.update\_idletasks()

width = change\_pwd\_dialog.winfo\_width()

height = change\_pwd\_dialog.winfo\_height()

x = (change\_pwd\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (change\_pwd\_dialog.winfo\_screenheight() // 2) - (height // 2)

change\_pwd\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

change\_pwd\_dialog.wait\_window()

def check\_registration(self):

"""检查是否已注册"""

try:

# 使用用户管理模块的注册验证逻辑

from src.ui.user\_management import UserManager

if not hasattr(self, 'user\_manager'):

self.user\_manager = UserManager(self)

# 调用用户管理模块的检查注册方法

is\_registered = self.user\_manager.check\_registration()

if is\_registered:

# 如果注册，尝试从注册文件中获取单位名称

config\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'config')

registration\_file = os.path.join(config\_dir, 'registration.dat')

if os.path.exists(registration\_file):

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

content = f.read().strip()

# 简单解密处理

decoded\_content = ''.join([chr(ord(c) - 1) for c in content])

parts = decoded\_content.split('|')

if len(parts) >= 2:

self.current\_org = parts[1].strip()

logging.info(f"已注册单位: {self.current\_org}")

else:

self.current\_org = None

return is\_registered

except Exception as e:

logging.error(f"检查注册状态失败: {str(e)}")

logging.debug(f"异常详情:", exc\_info=True)

# 如果出错，清除单位名并返回False（默认未注册）

self.current\_org = None

return False

def create\_status\_bar(self):

"""创建状态栏"""

try:

# 创建状态栏框架

self.\_status\_bar = ttk.Frame(self.root, height=20, relief=tk.SUNKEN)

self.\_status\_bar.pack(side=tk.BOTTOM, fill=tk.X)

# 获取当前注册状态

is\_registered = hasattr(self, 'is\_registered') and self.is\_registered

# 左侧状态标签 - 用于显示注册状态

self.reg\_status\_var = tk.StringVar()

# 根据注册状态设置初始文本和颜色

if is\_registered:

status\_text = f"已注册：{self.current\_org}" if self.current\_org else "已注册"

foreground\_color = 'green'

else:

status\_text = "未注册"

foreground\_color = 'red'

self.reg\_status\_var.set(status\_text)

logging.info(f"状态栏初始化为: {status\_text}")

reg\_status\_label = ttk.Label(

self.\_status\_bar,

textvariable=self.reg\_status\_var,

anchor=tk.W,

relief=tk.SUNKEN,

padding=(5, 0, 5, 0),

width=20,

foreground=foreground\_color

)

reg\_status\_label.pack(side=tk.LEFT, padx=(0, 1))

# 中间状态标签 - 用于显示一般状态信息

self.status\_var = tk.StringVar()

self.status\_var.set("就绪")

status\_label = ttk.Label(

self.\_status\_bar,

textvariable=self.status\_var,

anchor=tk.W,

relief=tk.SUNKEN,

padding=(5, 0, 5, 0)

)

status\_label.pack(side=tk.LEFT, fill=tk.X, expand=True)

# 右侧版本信息标签

version\_label = ttk.Label(

self.\_status\_bar,

text=f"版本: {self.version}",

anchor=tk.E,

relief=tk.SUNKEN,

padding=(5, 0, 5, 0),

width=15

)

version\_label.pack(side=tk.RIGHT)

logging.info("状态栏创建成功")

except Exception as e:

logging.error(f"创建状态栏时出错: {str(e)}")

import traceback

logging.error(traceback.format\_exc())

def update\_status\_bar(self):

"""更新状态栏中的注册状态"""

try:

if hasattr(self, 'reg\_status\_var'):

logging.debug(f"更新状态栏，当前注册状态: {'已注册' if self.is\_registered else '未注册'}")

old\_status = self.reg\_status\_var.get()

if self.is\_registered:

status\_text = f"已注册：{self.current\_org}" if self.current\_org else "已注册"

self.reg\_status\_var.set(status\_text)

# 已注册状态显示为绿色

for child in self.\_status\_bar.winfo\_children():

if isinstance(child, ttk.Label) and child.cget('textvariable') == str(self.reg\_status\_var):

child.configure(foreground='green')

logging.info(f"更新状态栏: {status\_text}")

else:

self.reg\_status\_var.set("未注册")

# 未注册状态显示为红色

for child in self.\_status\_bar.winfo\_children():

if isinstance(child, ttk.Label) and child.cget('textvariable') == str(self.reg\_status\_var):

child.configure(foreground='red')

logging.info("更新状态栏: 未注册")

new\_status = self.reg\_status\_var.get()

if old\_status != new\_status:

logging.debug(f"状态栏显示已从 '{old\_status}' 更改为 '{new\_status}'")

else:

logging.warning("无法更新状态栏：状态变量不存在")

except Exception as e:

logging.error(f"更新状态栏时出错: {str(e)}")

import traceback

logging.error(traceback.format\_exc())

def update\_ui\_by\_registration(self):

"""根据注册状态更新界面"""

try:

logging.info(f"开始更新界面，注册状态: {'已注册' if self.is\_registered else '未注册'}")

# 确保状态栏已创建

if not hasattr(self, '\_status\_bar'):

self.create\_status\_bar()

# 更新状态栏显示

self.update\_status\_bar()

if not self.is\_registered:

# 未注册状态：禁用大部分功能，保留帮助菜单和退出选项

# 禁用工具栏按钮

if hasattr(self, 'import\_category\_btn'):

self.import\_category\_btn.config(state=tk.DISABLED)

if hasattr(self, 'import\_file\_btn'):

self.import\_file\_btn.config(state=tk.DISABLED)

if hasattr(self, 'search\_button'):

self.search\_button.config(state=tk.DISABLED)

# 禁用搜索框

for widget in self.root.winfo\_children():

if isinstance(widget, ttk.Frame):

for child in widget.winfo\_children():

if isinstance(child, ttk.Frame):

for subchild in child.winfo\_children():

if isinstance(subchild, ttk.Entry):

subchild.config(state=tk.DISABLED)

# 有选择地禁用文件菜单项，保留退出功能

if hasattr(self, 'file\_menu'):

# 遍历所有菜单项

for i in range(self.file\_menu.index('end') + 1):

try:

# 获取菜单项标签

label = self.file\_menu.entrycget(i, 'label')

# 如果不是退出相关选项，则禁用

if label != '退出' and label != '登录':

self.file\_menu.entryconfigure(i, state=tk.DISABLED)

except:

pass

# 禁用工具菜单

if hasattr(self, 'tools\_menu'):

for i in range(self.tools\_menu.index('end') + 1):

try:

self.tools\_menu.entryconfigure(i, state=tk.DISABLED)

except:

pass

# 禁用安全菜单

if hasattr(self, 'security\_menu'):

for i in range(self.security\_menu.index('end') + 1):

try:

self.security\_menu.entryconfigure(i, state=tk.DISABLED)

except:

pass

# 确保用户注册菜单项可用（在帮助菜单中）

menubar = self.root.nametowidget(self.root['menu'])

for i in range(menubar.index('end') + 1):

try:

if menubar.entrycget(i, 'label') == '帮助':

help\_menu = menubar.nametowidget(menubar.entrycget(i, 'menu'))

for j in range(help\_menu.index('end') + 1):

try:

# 确保所有帮助菜单项可用

help\_menu.entryconfigure(j, state=tk.NORMAL)

except:

pass

break

except:

pass

logging.info("已设置为未注册状态界面")

else:

# 已注册状态：启用所有功能

logging.info("已设置为已注册状态界面")

# 不再隐藏用户注册菜单项，始终保持可用

# menubar = self.root.nametowidget(self.root['menu'])

# for i in range(menubar.index('end') + 1):

# try:

# if menubar.entrycget(i, 'label') == '帮助':

# help\_menu = menubar.nametowidget(menubar.entrycget(i, 'menu'))

# for j in range(help\_menu.index('end') + 1):

# try:

# if help\_menu.entrycget(j, 'label') == '用户注册':

# help\_menu.entryconfigure(j, state=tk.DISABLED)

# except:

# pass

# break

# except:

# pass

except Exception as e:

logging.error(f"更新界面失败: {str(e)}")

def register\_system(self, reg\_code, user\_name):

"""注册系统"""

try:

# 使用用户管理模块的注册验证逻辑

from src.ui.user\_management import UserManager

if not hasattr(self, 'user\_manager'):

self.user\_manager = UserManager(self)

# 调用用户管理模块的注册方法

success, message = self.user\_manager.register\_system(reg\_code, user\_name)

if success:

# 更新注册状态

self.is\_registered = True

self.current\_org = user\_name # 保存当前单位名

logging.info(f"系统注册状态已更新: 已注册，单位: {self.current\_org}")

# 强制更新状态栏

if hasattr(self, 'reg\_status\_var'):

# 直接设置新状态，确保显示变化

status\_text = f"已注册：{user\_name}"

self.reg\_status\_var.set(status\_text)

# 设置颜色为绿色

for child in self.\_status\_bar.winfo\_children():

if isinstance(child, ttk.Label) and child.cget('textvariable') == str(self.reg\_status\_var):

child.configure(foreground='green')

logging.info(f"状态栏已更新为: {status\_text}")

else:

logging.warning("无法更新状态栏：状态变量不存在")

# 更新界面

self.update\_ui\_by\_registration()

# 最后显示成功消息

messagebox.showinfo("注册成功", message)

return True

else:

logging.warning(f"注册失败: {message}")

messagebox.showerror("注册失败", message)

return False

except Exception as e:

logging.error(f"注册系统失败: {str(e)}")

messagebox.showerror("注册错误", f"注册系统时发生错误：{str(e)}")

return False

def show\_registration\_dialog(self):

"""显示注册对话框"""

logging.debug("显示注册对话框")

# 创建注册对话框

reg\_dialog = tk.Toplevel(self.root)

reg\_dialog.title("系统注册")

reg\_dialog.geometry("350x200")

reg\_dialog.resizable(False, False)

reg\_dialog.transient(self.root)

reg\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(reg\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 注册码

ttk.Label(form\_frame, text="注册码:").grid(row=0, column=0, sticky=tk.W, pady=5)

reg\_code\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=reg\_code\_var, width=25).grid(row=0, column=1, pady=5)

# 用户名称

ttk.Label(form\_frame, text="用户名称:").grid(row=1, column=0, sticky=tk.W, pady=5)

user\_name\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=user\_name\_var, width=25).grid(row=1, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=2, column=0, columnspan=2, pady=5)

# 注册按钮处理函数

def do\_register():

reg\_code = reg\_code\_var.get().strip()

user\_name = user\_name\_var.get().strip()

# 验证表单

if not reg\_code:

error\_var.set("注册码不能为空")

return

if not user\_name:

error\_var.set("用户名称不能为空")

return

# 注册系统

if self.register\_system(reg\_code, user\_name):

reg\_dialog.destroy()

else:

error\_var.set("注册失败: 无效的注册码或用户名")

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=3, column=0, columnspan=2, pady=15)

# 添加按钮

ttk.Button(buttons\_frame, text="确定", command=do\_register, width=10).pack(side=tk.LEFT, padx=15)

ttk.Button(buttons\_frame, text="取消", command=reg\_dialog.destroy, width=10).pack(side=tk.LEFT, padx=15)

# 居中对话框

reg\_dialog.update\_idletasks()

width = reg\_dialog.winfo\_width()

height = reg\_dialog.winfo\_height()

x = (reg\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (reg\_dialog.winfo\_screenheight() // 2) - (height // 2)

reg\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

reg\_dialog.wait\_window()

# 状态栏相关方法已合并到上面的create\_status\_bar方法中

# 其他方法...

def some\_other\_method(self):

if not hasattr(self, 'search\_result\_var'):

self.search\_result\_var = tk.StringVar()

self.search\_result\_var.set("就绪")

# 添加安全功能相关方法

def show\_integrity\_check(self):

"""显示档案完整性检查对话框"""

try:

# 获取档案目录

archives\_dir = None

if hasattr(self, 'db') and self.db:

# 从数据库或配置中获取档案目录

try:

# 尝试从数据库获取

query = "SELECT value FROM system\_settings WHERE key = 'archives\_dir'"

result = self.db.execute\_query(query)

if result and result[0][0]:

archives\_dir = result[0][0]

except:

# 如果出错，尝试从设置中获取

if hasattr(self, 'import\_root\_dir') and self.import\_root\_dir:

archives\_dir = self.import\_root\_dir

# 导入安全模块UI

try:

# 打包环境

if getattr(sys, 'frozen', False):

from security.ui import IntegrityCheckDialog

else:

# 开发环境

from src.security.ui import IntegrityCheckDialog

# 显示对话框

dialog = IntegrityCheckDialog(self.root, archives\_dir=archives\_dir)

# 等待对话框关闭

self.root.wait\_window(dialog)

except ImportError as e:

logging.error(f"导入安全模块失败: {str(e)}")

messagebox.showerror("错误", f"无法加载安全模块: {str(e)}")

except Exception as e:

logging.exception(f"显示档案完整性检查对话框时出错: {str(e)}")

messagebox.showerror("错误", f"打开档案完整性检查工具时出错: {str(e)}")

def show\_message(self, message, message\_type="info"):

"""在状态栏显示消息

Args:

message (str): 要显示的消息

message\_type (str): 消息类型，可以是 'info', 'error', 'warning' 等

"""

try:

if hasattr(self, 'status\_var'):

self.status\_var.set(message)

logging.debug(f"状态栏消息: {message} (类型: {message\_type})")

except Exception as e:

logging.error(f"显示状态栏消息时出错: {str(e)}")

# 为status\_bar添加show\_message方法，便于直接通过status\_bar调用

@property

def status\_bar(self):

"""返回状态栏对象，并确保它有show\_message方法"""

if not hasattr(self, '\_status\_bar'):

# 如果状态栏尚未创建，返回一个空对象

class EmptyStatusBar:

def show\_message(self, message, message\_type="info"):

pass

return EmptyStatusBar()

# 为状态栏添加show\_message方法

if not hasattr(self.\_status\_bar, 'show\_message'):

self.\_status\_bar.show\_message = lambda message, message\_type="info": self.show\_message(message, message\_type)

return self.\_status\_bar

ui.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

安全模块UI组件 - 提供安全功能的图形界面组件

"""

import os

import sys

import time

import threading

import tkinter as tk

from tkinter import ttk, filedialog, messagebox, simpledialog

import logging

from datetime import datetime

# 导入安全模块

try:

from src.security.integrity import IntegrityChecker

except ImportError:

# 打包环境

from security.integrity import IntegrityChecker

class LogHandler(logging.Handler):

"""将日志输出到文本控件的处理器"""

def \_\_init\_\_(self, text\_widget):

"""初始化处理器"""

logging.Handler.\_\_init\_\_(self)

self.text\_widget = text\_widget

# 设置格式化器

formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

self.setFormatter(formatter)

def emit(self, record):

"""输出日志记录"""

msg = self.format(record)

# 在UI线程中更新文本控件

def update():

self.text\_widget.configure(state='normal')

self.text\_widget.insert(tk.END, msg + '\n')

self.text\_widget.see(tk.END)

self.text\_widget.configure(state='disabled')

# 确保在主线程中更新UI

if self.text\_widget.winfo\_exists():

self.text\_widget.after(0, update)

class IntegrityCheckDialog(tk.Toplevel):

"""档案完整性检查对话框"""

def \_\_init\_\_(self, parent, archives\_dir=None):

"""初始化对话框"""

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

self.title("档案完整性检查")

self.geometry("800x600")

self.minsize(600, 500)

# 设置模态对话框

self.transient(parent)

self.grab\_set()

# 初始化变量

self.archives\_dir = archives\_dir

self.database\_path = None

# 创建日志记录器

self.logger = logging.getLogger('security.ui.integrity')

# 创建完整性检查器

self.integrity\_checker = IntegrityChecker(

archives\_dir=self.archives\_dir,

database\_path=self.database\_path

)

# 创建UI组件

self.create\_widgets()

# 居中显示

self.center\_window()

# 运行中的线程

self.running\_thread = None

# 添加窗口关闭处理

self.protocol("WM\_DELETE\_WINDOW", self.on\_close)

def create\_widgets(self):

"""创建UI组件"""

# 主框架

main\_frame = ttk.Frame(self, padding=(10, 10, 10, 10))

main\_frame.pack(fill=tk.BOTH, expand=True)

# 创建选项框架

options\_frame = ttk.LabelFrame(main\_frame, text="档案设置", padding=(5, 5, 5, 10))

options\_frame.pack(fill=tk.X, padx=5, pady=5)

# 档案目录设置

ttk.Label(options\_frame, text="档案目录:").grid(row=0, column=0, sticky=tk.W, padx=5, pady=5)

self.dir\_var = tk.StringVar(value=self.archives\_dir or "")

dir\_entry = ttk.Entry(options\_frame, textvariable=self.dir\_var, width=50)

dir\_entry.grid(row=0, column=1, sticky=tk.EW, padx=5, pady=5)

ttk.Button(options\_frame, text="浏览...", command=self.browse\_archives\_dir).grid(row=0, column=2, padx=5, pady=5)

# 数据库文件设置

ttk.Label(options\_frame, text="数据库文件:").grid(row=1, column=0, sticky=tk.W, padx=5, pady=5)

self.db\_var = tk.StringVar(value=self.database\_path or "")

db\_entry = ttk.Entry(options\_frame, textvariable=self.db\_var, width=50)

db\_entry.grid(row=1, column=1, sticky=tk.EW, padx=5, pady=5)

ttk.Button(options\_frame, text="浏览...", command=self.browse\_database\_file).grid(row=1, column=2, padx=5, pady=5)

# 设置列权重

options\_frame.columnconfigure(1, weight=1)

# 创建操作按钮框架

buttons\_frame = ttk.Frame(main\_frame)

buttons\_frame.pack(fill=tk.X, padx=5, pady=5)

# 添加操作按钮

self.build\_btn = ttk.Button(buttons\_frame, text="构建完整性数据库", command=self.build\_database)

self.build\_btn.pack(side=tk.LEFT, padx=5)

self.verify\_btn = ttk.Button(buttons\_frame, text="验证文件完整性", command=self.verify\_integrity)

self.verify\_btn.pack(side=tk.LEFT, padx=5)

self.update\_btn = ttk.Button(buttons\_frame, text="更新完整性数据库", command=self.update\_database)

self.update\_btn.pack(side=tk.LEFT, padx=5)

self.abort\_btn = ttk.Button(buttons\_frame, text="中止操作", command=self.abort\_operation, state=tk.DISABLED)

self.abort\_btn.pack(side=tk.LEFT, padx=5)

# 创建状态框架

status\_frame = ttk.LabelFrame(main\_frame, text="状态", padding=(5, 5, 5, 5))

status\_frame.pack(fill=tk.X, padx=5, pady=5)

# 添加进度条

ttk.Label(status\_frame, text="进度:").grid(row=0, column=0, sticky=tk.W, padx=5, pady=5)

self.progress\_var = tk.DoubleVar()

self.progress\_bar = ttk.Progressbar(status\_frame, variable=self.progress\_var, length=100, mode='determinate')

self.progress\_bar.grid(row=0, column=1, sticky=tk.EW, padx=5, pady=5)

# 添加状态标签

ttk.Label(status\_frame, text="当前文件:").grid(row=1, column=0, sticky=tk.W, padx=5, pady=5)

self.status\_var = tk.StringVar(value="就绪")

status\_label = ttk.Label(status\_frame, textvariable=self.status\_var, wraplength=500)

status\_label.grid(row=1, column=1, sticky=tk.W, padx=5, pady=5)

# 设置列权重

status\_frame.columnconfigure(1, weight=1)

# 创建日志框架

log\_frame = ttk.LabelFrame(main\_frame, text="操作日志", padding=(5, 5, 5, 5))

log\_frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

# 添加日志文本框

self.log\_text = tk.Text(log\_frame, wrap=tk.WORD, width=80, height=15)

self.log\_text.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

# 添加滚动条

scrollbar = ttk.Scrollbar(log\_frame, command=self.log\_text.yview)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

self.log\_text['yscrollcommand'] = scrollbar.set

# 禁用文本编辑

self.log\_text.config(state=tk.DISABLED)

# 添加日志处理器

log\_handler = LogHandler(self.log\_text)

logging.getLogger('security').addHandler(log\_handler)

# 添加关闭按钮

close\_btn = ttk.Button(main\_frame, text="关闭", command=self.on\_close)

close\_btn.pack(side=tk.RIGHT, padx=5, pady=10)

def center\_window(self):

"""居中显示窗口"""

self.update\_idletasks()

width = self.winfo\_width()

height = self.winfo\_height()

x = (self.winfo\_screenwidth() // 2) - (width // 2)

y = (self.winfo\_screenheight() // 2) - (height // 2)

self.geometry(f'{width}x{height}+{x}+{y}')

def browse\_archives\_dir(self):

"""浏览选择档案目录"""

dir\_path = filedialog.askdirectory(title="选择档案目录")

if dir\_path:

self.dir\_var.set(dir\_path)

self.archives\_dir = dir\_path

self.integrity\_checker.archives\_dir = dir\_path

def browse\_database\_file(self):

"""浏览选择数据库文件"""

file\_path = filedialog.asksaveasfilename(

title="选择数据库文件",

defaultextension=".json",

filetypes=[("JSON文件", "\*.json"), ("所有文件", "\*.\*")]

)

if file\_path:

self.db\_var.set(file\_path)

self.database\_path = file\_path

self.integrity\_checker.database\_path = file\_path

def update\_progress(self, progress, current\_file):

"""更新进度显示"""

self.progress\_var.set(progress)

if current\_file:

# 显示相对路径

if self.archives\_dir and current\_file.startswith(self.archives\_dir):

rel\_path = os.path.relpath(current\_file, self.archives\_dir)

self.status\_var.set(rel\_path)

else:

self.status\_var.set(current\_file)

def build\_database(self):

"""构建完整性数据库"""

# 检查档案目录

if not self.archives\_dir or not os.path.isdir(self.archives\_dir):

messagebox.showerror("错误", "请选择有效的档案目录")

return

# 设置数据库路径（如果未指定）

if not self.database\_path:

config\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'config')

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

self.database\_path = os.path.join(config\_dir, 'integrity\_db.json')

self.db\_var.set(self.database\_path)

self.integrity\_checker.database\_path = self.database\_path

# 禁用按钮

self.disable\_buttons()

# 在后台线程中执行操作

self.running\_thread = threading.Thread(target=self.\_build\_database\_thread)

self.running\_thread.daemon = True

self.running\_thread.start()

# 启动进度更新定时器

self.after(100, self.update\_progress\_timer)

def \_build\_database\_thread(self):

"""构建数据库的后台线程"""

success = self.integrity\_checker.build\_database()

# 在UI线程中更新状态

self.after(0, lambda: self.on\_operation\_complete(

success,

"完整性数据库构建完成" if success else "完整性数据库构建失败"

))

def verify\_integrity(self):

"""验证文件完整性"""

# 检查数据库文件

if not self.database\_path or not os.path.exists(self.database\_path):

messagebox.showerror("错误", "请先构建或选择有效的完整性数据库文件")

return

# 禁用按钮

self.disable\_buttons()

# 在后台线程中执行操作

self.running\_thread = threading.Thread(target=self.\_verify\_integrity\_thread)

self.running\_thread.daemon = True

self.running\_thread.start()

# 启动进度更新定时器

self.after(100, self.update\_progress\_timer)

def \_verify\_integrity\_thread(self):

"""验证完整性的后台线程"""

result = self.integrity\_checker.verify\_integrity()

# 在UI线程中更新状态

self.after(0, lambda: self.on\_verification\_complete(result))

def update\_database(self):

"""更新完整性数据库"""

# 检查数据库文件

if not self.database\_path or not os.path.exists(self.database\_path):

messagebox.showerror("错误", "请先构建或选择有效的完整性数据库文件")

return

# 禁用按钮

self.disable\_buttons()

# 在后台线程中执行操作

self.running\_thread = threading.Thread(target=self.\_update\_database\_thread)

self.running\_thread.daemon = True

self.running\_thread.start()

# 启动进度更新定时器

self.after(100, self.update\_progress\_timer)

def \_update\_database\_thread(self):

"""更新数据库的后台线程"""

success = self.integrity\_checker.update\_database()

# 在UI线程中更新状态

self.after(0, lambda: self.on\_operation\_complete(

success,

"完整性数据库更新完成" if success else "完整性数据库更新失败"

))

def update\_progress\_timer(self):

"""定时更新进度"""

if self.running\_thread and self.running\_thread.is\_alive():

# 获取当前进度

progress\_info = self.integrity\_checker.get\_progress()

# 更新UI

self.update\_progress(progress\_info['progress'], progress\_info['current\_file'])

# 继续定时更新

self.after(100, self.update\_progress\_timer)

else:

# 操作已完成，恢复按钮状态

if not self.running\_thread:

self.enable\_buttons()

def on\_operation\_complete(self, success, message):

"""操作完成时的处理"""

# 启用按钮

self.enable\_buttons()

# 显示完成消息

if success:

messagebox.showinfo("完成", message)

else:

messagebox.showerror("错误", message)

# 重置进度条

self.progress\_var.set(0)

self.status\_var.set("就绪")

def on\_verification\_complete(self, result):

"""验证完成时的处理"""

# 启用按钮

self.enable\_buttons()

# 检查验证结果

if result['status'] == 'success':

# 构建报告消息

message = f"档案完整性验证完成\n\n"

message += f"验证时间: {result['verified\_time']}\n"

message += f"数据库创建时间: {result['database\_time']}\n"

message += f"档案目录: {result['archives\_dir']}\n\n"

message += f"总文件数: {result['total\_files']}\n"

message += f"验证通过: {result['verified\_files']}\n"

message += f"已修改文件: {len(result['modified\_files'])}\n"

message += f"丢失文件: {len(result['missing\_files'])}\n"

message += f"新增文件: {len(result['new\_files'])}\n\n"

# 显示详细报告

self.show\_verification\_details(result)

else:

# 显示错误消息

messagebox.showerror("验证失败", f"档案完整性验证失败: {result.get('message', '未知错误')}")

# 重置进度条

self.progress\_var.set(0)

self.status\_var.set("就绪")

def show\_verification\_details(self, result):

"""显示验证详细信息"""

# 创建详细信息窗口

details\_dialog = tk.Toplevel(self)

details\_dialog.title("档案完整性验证结果")

details\_dialog.geometry("700x500")

details\_dialog.minsize(600, 400)

# 使窗口成为模态

details\_dialog.transient(self)

details\_dialog.grab\_set()

# 创建主框架

main\_frame = ttk.Frame(details\_dialog, padding=(10, 10, 10, 10))

main\_frame.pack(fill=tk.BOTH, expand=True)

# 添加统计信息

stats\_frame = ttk.LabelFrame(main\_frame, text="验证统计", padding=(5, 5, 5, 5))

stats\_frame.pack(fill=tk.X, padx=5, pady=5)

# 格式化统计信息

stats\_text = f"验证时间: {result['verified\_time']}\n"

stats\_text += f"数据库时间: {result['database\_time']}\n"

stats\_text += f"总文件数: {result['total\_files']} "

stats\_text += f"验证通过: {result['verified\_files']} "

stats\_text += f"修改: {len(result['modified\_files'])} "

stats\_text += f"丢失: {len(result['missing\_files'])} "

stats\_text += f"新增: {len(result['new\_files'])}"

stats\_label = ttk.Label(stats\_frame, text=stats\_text, wraplength=650)

stats\_label.pack(fill=tk.X, padx=5, pady=5)

# 创建选项卡控件

notebook = ttk.Notebook(main\_frame)

notebook.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

# 添加已修改文件选项卡

modified\_frame = ttk.Frame(notebook, padding=(5, 5, 5, 5))

notebook.add(modified\_frame, text=f"已修改文件 ({len(result['modified\_files'])})")

# 添加丢失文件选项卡

missing\_frame = ttk.Frame(notebook, padding=(5, 5, 5, 5))

notebook.add(missing\_frame, text=f"丢失文件 ({len(result['missing\_files'])})")

# 添加新增文件选项卡

new\_frame = ttk.Frame(notebook, padding=(5, 5, 5, 5))

notebook.add(new\_frame, text=f"新增文件 ({len(result['new\_files'])})")

# 填充已修改文件列表

self.\_fill\_file\_list(modified\_frame, result['modified\_files'], "已修改文件")

# 填充丢失文件列表

self.\_fill\_file\_list(missing\_frame, result['missing\_files'], "丢失文件")

# 填充新增文件列表

self.\_fill\_file\_list(new\_frame, result['new\_files'], "新增文件")

# 添加关闭按钮

close\_btn = ttk.Button(main\_frame, text="关闭", command=details\_dialog.destroy)

close\_btn.pack(side=tk.RIGHT, padx=5, pady=10)

# 居中显示窗口

details\_dialog.update\_idletasks()

width = details\_dialog.winfo\_width()

height = details\_dialog.winfo\_height()

x = (details\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (details\_dialog.winfo\_screenheight() // 2) - (height // 2)

details\_dialog.geometry(f'{width}x{height}+{x}+{y}')

def \_fill\_file\_list(self, parent, file\_list, title):

"""填充文件列表"""

if not file\_list:

# 如果没有文件，显示空消息

label = ttk.Label(parent, text=f"没有{title}")

label.pack(fill=tk.X, padx=5, pady=20)

return

# 创建列表框架

frame = ttk.Frame(parent)

frame.pack(fill=tk.BOTH, expand=True)

# 创建滚动条

scrollbar = ttk.Scrollbar(frame)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

# 创建列表框

listbox = tk.Listbox(frame, yscrollcommand=scrollbar.set, font=('Microsoft YaHei', 10))

listbox.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

# 配置滚动条

scrollbar.config(command=listbox.yview)

# 添加文件到列表

for file\_path in file\_list:

listbox.insert(tk.END, file\_path)

# 双击打开文件

def on\_double\_click(event):

selection = listbox.curselection()

if selection:

file\_path = file\_list[selection[0]]

full\_path = os.path.join(result['archives\_dir'], file\_path)

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

try:

if sys.platform == 'win32':

os.startfile(full\_path)

elif sys.platform == 'darwin':

import subprocess

subprocess.call(['open', full\_path])

else:

import subprocess

subprocess.call(['xdg-open', full\_path])

except Exception as e:

self.logger.error(f"打开文件时出错: {str(e)}")

messagebox.showerror("错误", f"无法打开文件: {str(e)}")

listbox.bind('<Double-1>', on\_double\_click)

def disable\_buttons(self):

"""禁用操作按钮"""

self.build\_btn.config(state=tk.DISABLED)

self.verify\_btn.config(state=tk.DISABLED)

self.update\_btn.config(state=tk.DISABLED)

self.abort\_btn.config(state=tk.NORMAL)

def enable\_buttons(self):

"""启用操作按钮"""

self.build\_btn.config(state=tk.NORMAL)

self.verify\_btn.config(state=tk.NORMAL)

self.update\_btn.config(state=tk.NORMAL)

self.abort\_btn.config(state=tk.DISABLED)

def abort\_operation(self):

"""中止当前操作"""

if self.running\_thread and self.running\_thread.is\_alive():

# 设置中止标志

self.integrity\_checker.abort()

self.logger.info("正在中止操作...")

self.status\_var.set("正在中止...")

def on\_close(self):

"""关闭窗口时的处理"""

# 中止操作（如果正在运行）

if self.running\_thread and self.running\_thread.is\_alive():

self.integrity\_checker.abort()

self.logger.info("等待操作完成...")

self.running\_thread.join(2.0) # 等待最多2秒

# 移除日志处理器

for handler in logging.getLogger('security').handlers:

if isinstance(handler, LogHandler) and handler.text\_widget == self.log\_text:

logging.getLogger('security').removeHandler(handler)

# 关闭窗口

self.destroy()

integrity.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

档案完整性检查模块 - 提供档案文件完整性检查功能

"""

import os

import json

import logging

import hashlib

import time

from datetime import datetime

import threading

from pathlib import Path

# 导入工具模块

try:

from src.security.utils import calculate\_file\_hash, get\_file\_metadata

except ImportError:

# 打包环境

from security.utils import calculate\_file\_hash, get\_file\_metadata

class IntegrityChecker:

"""档案完整性检查类"""

def \_\_init\_\_(self, archives\_dir=None, database\_path=None):

"""

初始化档案完整性检查器

参数:

archives\_dir (str): 档案文件目录

database\_path (str): 完整性数据库文件路径

"""

# 设置档案目录

self.archives\_dir = archives\_dir

# 设置数据库路径

if database\_path:

self.database\_path = database\_path

else:

config\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'config')

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

self.database\_path = os.path.join(config\_dir, 'integrity\_db.json')

# 确保数据库目录存在

os.makedirs(os.path.dirname(self.database\_path), exist\_ok=True)

# 支持的文件扩展名

self.supported\_extensions = ['.pdf', '.jpg', '.jpeg', '.png', '.tif', '.tiff', '.doc', '.docx', '.xls', '.xlsx']

# 日志记录器

self.logger = logging.getLogger('security.integrity')

# 内部状态

self.\_is\_running = False

self.\_progress = 0

self.\_total\_files = 0

self.\_current\_file = ""

self.\_abort\_flag = False

# 线程锁

self.\_lock = threading.Lock()

def get\_progress(self):

"""获取当前进度"""

with self.\_lock:

return {

'is\_running': self.\_is\_running,

'progress': self.\_progress,

'total\_files': self.\_total\_files,

'current\_file': self.\_current\_file

}

def abort(self):

"""中止当前操作"""

with self.\_lock:

self.\_abort\_flag = True

def build\_database(self, callback=None):

"""

构建完整性数据库

参数:

callback: 进度回调函数，接收进度百分比和当前文件路径

返回:

bool: 操作是否成功

"""

if not self.archives\_dir or not os.path.isdir(self.archives\_dir):

self.logger.error(f"档案目录无效: {self.archives\_dir}")

return False

try:

with self.\_lock:

self.\_is\_running = True

self.\_progress = 0

self.\_abort\_flag = False

start\_time = time.time()

self.logger.info(f"开始构建完整性数据库，档案目录: {self.archives\_dir}")

# 扫描所有文件

all\_files = []

for root, \_, files in os.walk(self.archives\_dir):

for file in files:

file\_path = os.path.join(root, file)

if self.\_should\_process\_file(file\_path):

all\_files.append(file\_path)

with self.\_lock:

self.\_total\_files = len(all\_files)

self.logger.info(f"找到 {len(all\_files)} 个档案文件")

# 准备数据库

database = {

'created\_time': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),

'archives\_dir': self.archives\_dir,

'file\_count': len(all\_files),

'files': {}

}

# 处理每个文件

for i, file\_path in enumerate(all\_files):

# 检查是否需要中止

with self.\_lock:

if self.\_abort\_flag:

self.logger.info("操作被用户中止")

return False

self.\_current\_file = file\_path

self.\_progress = (i / len(all\_files)) \* 100 if all\_files else 100

try:

# 计算文件哈希

file\_hash = calculate\_file\_hash(file\_path)

# 获取文件元数据

metadata = get\_file\_metadata(file\_path)

# 将相对路径存储到数据库

rel\_path = os.path.relpath(file\_path, self.archives\_dir)

# 保存到数据库

database['files'][rel\_path] = {

'hash': file\_hash,

'size': metadata['size'],

'modified\_time': metadata['modified\_time'],

'created\_time': metadata['created\_time'],

'checked\_time': datetime.now().strftime('%Y-%m-%d %H:%M:%S')

}

# 调用回调函数

if callback:

callback(self.\_progress, file\_path)

except Exception as e:

self.logger.error(f"处理文件 {file\_path} 时出错: {str(e)}")

# 保存数据库

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

json.dump(database, f, ensure\_ascii=False, indent=2)

elapsed\_time = time.time() - start\_time

self.logger.info(f"完整性数据库构建完成，用时 {elapsed\_time:.2f} 秒")

return True

except Exception as e:

self.logger.exception(f"构建完整性数据库时出错: {str(e)}")

return False

finally:

with self.\_lock:

self.\_is\_running = False

self.\_progress = 100

def verify\_integrity(self, callback=None, generate\_report=True):

"""

验证档案完整性

参数:

callback: 进度回调函数

generate\_report: 是否生成报告

返回:

dict: 包含验证结果的字典

"""

if not os.path.exists(self.database\_path):

self.logger.error(f"完整性数据库不存在: {self.database\_path}")

return {'status': 'error', 'message': '完整性数据库不存在'}

try:

with self.\_lock:

self.\_is\_running = True

self.\_progress = 0

self.\_abort\_flag = False

start\_time = time.time()

self.logger.info("开始验证档案完整性")

# 加载数据库

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

database = json.load(f)

archives\_dir = database.get('archives\_dir', self.archives\_dir)

if not archives\_dir or not os.path.isdir(archives\_dir):

self.logger.error(f"数据库中的档案目录无效: {archives\_dir}")

return {'status': 'error', 'message': '档案目录无效'}

# 初始化结果

result = {

'status': 'success',

'verified\_time': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),

'database\_time': database.get('created\_time', 'unknown'),

'archives\_dir': archives\_dir,

'total\_files': database.get('file\_count', 0),

'verified\_files': 0,

'modified\_files': [],

'missing\_files': [],

'new\_files': []

}

# 获取所有当前文件

current\_files = set()

for root, \_, files in os.walk(archives\_dir):

for file in files:

file\_path = os.path.join(root, file)

if self.\_should\_process\_file(file\_path):

rel\_path = os.path.relpath(file\_path, archives\_dir)

current\_files.add(rel\_path)

# 处理数据库中的文件

db\_files = list(database.get('files', {}).keys())

with self.\_lock:

self.\_total\_files = len(db\_files)

for i, rel\_path in enumerate(db\_files):

# 检查是否需要中止

with self.\_lock:

if self.\_abort\_flag:

self.logger.info("操作被用户中止")

return {'status': 'aborted', 'message': '操作被用户中止'}

self.\_progress = (i / len(db\_files)) \* 100 if db\_files else 100

# 构造完整路径

file\_path = os.path.join(archives\_dir, rel\_path)

with self.\_lock:

self.\_current\_file = file\_path

# 调用回调函数

if callback:

callback(self.\_progress, file\_path)

# 检查文件是否存在

if not os.path.exists(file\_path):

self.logger.warning(f"文件丢失: {rel\_path}")

result['missing\_files'].append(rel\_path)

continue

try:

# 获取数据库中的哈希值

db\_hash = database['files'][rel\_path]['hash']

# 计算当前哈希值

current\_hash = calculate\_file\_hash(file\_path)

# 比较哈希值

if current\_hash != db\_hash:

self.logger.warning(f"文件已修改: {rel\_path}")

result['modified\_files'].append(rel\_path)

else:

result['verified\_files'] += 1

except Exception as e:

self.logger.error(f"验证文件 {file\_path} 时出错: {str(e)}")

# 查找新文件

db\_file\_set = set(db\_files)

for rel\_path in current\_files:

if rel\_path not in db\_file\_set:

result['new\_files'].append(rel\_path)

self.logger.info(f"发现新文件: {rel\_path}")

# 生成报告

if generate\_report:

report\_path = os.path.join(os.path.dirname(self.database\_path), 'integrity\_report.json')

with open(report\_path, 'w', encoding='utf-8') as f:

json.dump(result, f, ensure\_ascii=False, indent=2)

self.logger.info(f"完整性报告已保存到: {report\_path}")

elapsed\_time = time.time() - start\_time

self.logger.info(f"档案完整性验证完成，用时 {elapsed\_time:.2f} 秒")

return result

except Exception as e:

self.logger.exception(f"验证档案完整性时出错: {str(e)}")

return {'status': 'error', 'message': str(e)}

finally:

with self.\_lock:

self.\_is\_running = False

self.\_progress = 100

def update\_database(self, callback=None):

"""

更新完整性数据库

参数:

callback: 进度回调函数

返回:

bool: 操作是否成功

"""

if not os.path.exists(self.database\_path):

self.logger.error(f"完整性数据库不存在: {self.database\_path}")

return False

try:

with self.\_lock:

self.\_is\_running = True

self.\_progress = 0

self.\_abort\_flag = False

start\_time = time.time()

self.logger.info("开始更新完整性数据库")

# 加载数据库

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

database = json.load(f)

archives\_dir = database.get('archives\_dir', self.archives\_dir)

if not archives\_dir or not os.path.isdir(archives\_dir):

self.logger.error(f"数据库中的档案目录无效: {archives\_dir}")

return False

# 获取所有当前文件

current\_files = []

for root, \_, files in os.walk(archives\_dir):

for file in files:

file\_path = os.path.join(root, file)

if self.\_should\_process\_file(file\_path):

rel\_path = os.path.relpath(file\_path, archives\_dir)

current\_files.append((rel\_path, file\_path))

with self.\_lock:

self.\_total\_files = len(current\_files)

# 更新数据库

db\_files = set(database.get('files', {}).keys())

updated\_db = {

'created\_time': datetime.now().strftime('%Y-%m-%d %H:%M:%S'),

'archives\_dir': archives\_dir,

'file\_count': len(current\_files),

'files': {}

}

# 处理每个文件

for i, (rel\_path, file\_path) in enumerate(current\_files):

# 检查是否需要中止

with self.\_lock:

if self.\_abort\_flag:

self.logger.info("操作被用户中止")

return False

self.\_current\_file = file\_path

self.\_progress = (i / len(current\_files)) \* 100 if current\_files else 100

# 调用回调函数

if callback:

callback(self.\_progress, file\_path)

try:

# 如果文件在数据库中且未修改，则复用数据

if rel\_path in db\_files:

old\_hash = database['files'][rel\_path]['hash']

current\_hash = calculate\_file\_hash(file\_path)

if old\_hash == current\_hash:

# 复用旧数据

updated\_db['files'][rel\_path] = database['files'][rel\_path]

updated\_db['files'][rel\_path]['checked\_time'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

continue

# 计算新哈希值和元数据

file\_hash = calculate\_file\_hash(file\_path)

metadata = get\_file\_metadata(file\_path)

# 保存到数据库

updated\_db['files'][rel\_path] = {

'hash': file\_hash,

'size': metadata['size'],

'modified\_time': metadata['modified\_time'],

'created\_time': metadata['created\_time'],

'checked\_time': datetime.now().strftime('%Y-%m-%d %H:%M:%S')

}

except Exception as e:

self.logger.error(f"处理文件 {file\_path} 时出错: {str(e)}")

# 备份旧数据库

backup\_path = f"{self.database\_path}.bak"

import shutil

shutil.copy2(self.database\_path, backup\_path)

self.logger.info(f"已备份旧数据库到: {backup\_path}")

# 保存更新后的数据库

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

json.dump(updated\_db, f, ensure\_ascii=False, indent=2)

elapsed\_time = time.time() - start\_time

self.logger.info(f"完整性数据库更新完成，用时 {elapsed\_time:.2f} 秒")

return True

except Exception as e:

self.logger.exception(f"更新完整性数据库时出错: {str(e)}")

return False

finally:

with self.\_lock:

self.\_is\_running = False

self.\_progress = 100

def \_should\_process\_file(self, file\_path):

"""判断是否应该处理该文件"""

if not os.path.isfile(file\_path):

return False

# 检查文件扩展名

ext = os.path.splitext(file\_path)[1].lower()

return ext in self.supported\_extensions

file\_operations.py

import os

import logging

import re

import shutil

import subprocess

from tkinter import filedialog, messagebox

import pandas as pd

from src.utils.excel\_utils import get\_excel\_info, ExcelFileNotFound

class FileOperationManager:

"""文件操作管理类，包含所有文件操作相关的功能"""

def \_\_init\_\_(self, main\_window):

self.main\_window = main\_window

self.root = main\_window.root

self.db = main\_window.db

self.import\_root\_dir = main\_window.import\_root\_dir

def import\_files(self):

"""导入文件"""

# 检查是否已登录

if not self.main\_window.current\_user:

messagebox.showwarning("警告", "请先登录")

return

# 选择导入目录

import\_dir = filedialog.askdirectory(title="选择导入目录")

if not import\_dir:

return

# 更新导入根目录

self.main\_window.import\_root\_dir = import\_dir

self.import\_root\_dir = import\_dir

# 更新主窗口标题

if hasattr(self.main\_window, 'update\_window\_title'):

self.main\_window.update\_window\_title()

# 保存设置

self.main\_window.save\_settings()

# 扫描目录下的所有PDF文件

all\_files = []

for root, \_, files in os.walk(import\_dir):

for file in files:

if file.lower().endswith('.pdf'):

file\_path = os.path.join(root, file)

all\_files.append((file, file\_path))

if not all\_files:

messagebox.showinfo("提示", "未找到PDF文件")

return

# 更新文件列表

self.main\_window.update\_file\_list(all\_files)

# 显示导入结果

messagebox.showinfo("导入成功", f"成功导入 {len(all\_files)} 个文件")

def import\_archives(self):

"""导入档案"""

# 检查是否已登录

if not self.main\_window.current\_user:

messagebox.showwarning("警告", "请先登录")

return

# 选择导入目录

import\_dir = filedialog.askdirectory(title="选择导入目录")

if not import\_dir:

return

# 更新导入根目录

self.main\_window.import\_root\_dir = import\_dir

self.import\_root\_dir = import\_dir

# 更新主窗口标题

if hasattr(self.main\_window, 'update\_window\_title'):

self.main\_window.update\_window\_title()

# 保存设置

self.main\_window.save\_settings()

# 扫描目录下的所有子目录（每个子目录代表一个人的档案）

all\_files = []

person\_count = 0

file\_count = 0

for person\_dir in os.listdir(import\_dir):

person\_path = os.path.join(import\_dir, person\_dir)

if os.path.isdir(person\_path):

person\_count += 1

# 扫描该人档案下的所有PDF文件

for root, \_, files in os.walk(person\_path):

for file in files:

if file.lower().endswith('.pdf'):

file\_path = os.path.join(root, file)

all\_files.append((file, file\_path))

file\_count += 1

if not all\_files:

messagebox.showinfo("提示", "未找到PDF文件")

return

# 更新文件列表

self.main\_window.update\_file\_list(all\_files)

# 显示导入结果

messagebox.showinfo("导入成功", f"成功导入 {person\_count} 人的 {file\_count} 个档案文件")

def update\_file\_list(self, files):

"""更新文件列表"""

# 清空文件列表

self.main\_window.file\_list.delete(\*self.main\_window.file\_list.get\_children())

# 如果没有文件，则显示提示

if not files or len(files) == 0:

self.main\_window.status\_bar.show\_message("未找到匹配的文件", "info")

return

# 更新文件列表

for file in files:

try:

# 处理不同格式的文件数据

if len(file) >= 7:

# 完整的person\_files记录

file\_id = file[6] if file[6] else "" # file\_id

person\_name = file[1] if file[1] else "" # person\_name

file\_name = file[2] if file[2] else "" # file\_name

file\_path = file[3] if file[3] else "" # file\_path

# 使用完整类号格式

class\_code = file[4] if file[4] else "" # category\_id

# 确保使用完整类号格式 - 如果class\_code为空，尝试从文件名提取

if not class\_code:

# 从文件名中提取完整类号

class\_code = os.path.splitext(file\_name)[0]

# 从Excel获取文件信息（材料名称、日期、页数）

try:

material\_name, file\_date, page\_count, \_ = get\_excel\_info(self.import\_root\_dir, person\_name, file\_id, class\_code)

logging.info(f"获取Excel信息成功: 类号='{class\_code}', 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

except ExcelFileNotFound:

logging.warning(f"未找到Excel文件: 人名='{person\_name}', 编号='{file\_id}', 类号='{class\_code}'")

material\_name, file\_date, page\_count = '', '', ''

except Exception as e:

logging.error(f"Excel信息读取失败: {str(e)}, 类号='{class\_code}'")

material\_name, file\_date, page\_count = '', '', ''

else:

# 处理简单的(file\_name, file\_path)格式

file\_name, file\_path = file

# 从文件路径中提取人名和文件夹名

dir\_path = os.path.dirname(file\_path)

dir\_name = os.path.basename(dir\_path)

# 提取编号 - 获取文件夹名中的数字前缀

match = re.match(r'^(\d+)(.\*)', dir\_name)

file\_id = ""

person\_name = dir\_name

if match:

file\_id = match.group(1)

# 处理姓名，去掉数字部分只保留人名

person\_name = match.group(2).strip()

# 文件名就是类号（例如 3-1.pdf 的类号就是 3-1）- 保留完整格式

class\_code = os.path.splitext(file\_name)[0]

# 从Excel获取文件信息（材料名称、日期、页数）

try:

material\_name, file\_date, page\_count, \_ = get\_excel\_info(self.import\_root\_dir, person\_name, file\_id, class\_code)

logging.info(f"获取Excel信息成功: 类号='{class\_code}', 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

except ExcelFileNotFound:

logging.warning(f"未找到Excel文件: 人名='{person\_name}', 编号='{file\_id}', 类号='{class\_code}'")

material\_name, file\_date, page\_count = '', '', ''

except Exception as e:

logging.error(f"Excel信息读取失败: {str(e)}, 类号='{class\_code}'")

material\_name, file\_date, page\_count = '', '', ''

# 插入到列表

self.main\_window.file\_list.insert('', 'end', values=(

file\_id,

person\_name,

class\_code, # 使用完整类号格式

material\_name,

file\_name,

file\_date,

page\_count,

file\_path

))

except Exception as e:

logging.error(f"处理文件时出错: {str(e)}, 文件: {file}")

# 更新状态栏

item\_count = len(self.main\_window.file\_list.get\_children())

if item\_count > 0:

self.main\_window.status\_bar.show\_message(f"找到 {item\_count} 条记录", "info")

else:

self.main\_window.status\_bar.show\_message("未找到匹配的记录", "warning")

def on\_file\_double\_click(self, event):

"""处理文件双击事件"""

# 获取选中的文件

selected\_item = self.main\_window.file\_list.selection()

if not selected\_item:

return

# 获取文件路径

file\_path = self.main\_window.file\_list.item(selected\_item)['values'][-1]

if not file\_path or not os.path.exists(file\_path):

messagebox.showerror("错误", "文件不存在")

return

# 使用系统默认程序打开文件

try:

if os.name == 'nt': # Windows

os.startfile(file\_path)

elif os.name == 'posix': # macOS, Linux

subprocess.call(('xdg-open', file\_path))

logging.info(f"打开文件: {file\_path}")

except Exception as e:

logging.error(f"打开文件失败: {str(e)}")

messagebox.showerror("错误", f"打开文件失败: {str(e)}")

def search\_person(self):

"""搜索人员档案

搜索规则：

1. 只在'目录'子目录下搜索Excel文件

2. 文件名必须完全匹配"编号+姓名.xlsx"

3. 如果未找到匹配文件，只显示一次提示

"""

# 检查是否已登录

if not self.main\_window.current\_user:

messagebox.showwarning("警告", "请先登录")

return

# 获取搜索条件

search\_name = self.main\_window.search\_name\_var.get().strip()

search\_id = self.main\_window.search\_id\_var.get().strip()

logging.info(f"执行搜索：姓名='{search\_name}', 编号='{search\_id}'")

# 确保至少提供了一个搜索条件

if not search\_id and not search\_name:

error\_msg = "请输入编号或姓名进行搜索"

logging.warning(error\_msg)

if hasattr(self.main\_window, 'status\_bar'):

self.main\_window.status\_bar.show\_message(error\_msg, "warning")

return

# 记录搜索条件

self.main\_window.current\_search\_name = search\_name

self.main\_window.current\_search\_id = search\_id

self.main\_window.has\_searched = True

# 如果没有设置导入目录，提示用户

if not self.main\_window.import\_root\_dir:

error\_msg = "请先使用'导入档案'功能导入档案文件"

logging.warning(error\_msg)

if hasattr(self.main\_window, 'status\_bar'):

self.main\_window.status\_bar.show\_message(error\_msg, "warning")

messagebox.showinfo("提示", error\_msg)

return

# 如果搜索条件为空，提示用户

if not search\_name and not search\_id:

messagebox.showwarning("警告", "请输入姓名或编号")

return

logging.info(f"开始搜索: 姓名='{search\_name}', 编号='{search\_id}'")

# 清空当前列表

self.main\_window.file\_list.delete(\*self.main\_window.file\_list.get\_children())

# 构建'目录'子目录路径

import\_root\_dir = self.main\_window.import\_root\_dir

catalog\_dir = os.path.join(import\_root\_dir, '目录')

# 确保'目录'文件夹存在

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

error\_msg = f"目录不存在: {catalog\_dir}"

logging.error(error\_msg)

messagebox.showerror("错误", "未找到'目录'文件夹，请确保文件结构正确")

if hasattr(self.main\_window, 'status\_bar'):

self.main\_window.status\_bar.show\_message("错误: 未找到'目录'文件夹", "error")

return

logging.info(f"在目录中搜索: {catalog\_dir}")

# 构建预期的Excel文件名

expected\_name = f"{search\_id}{search\_name}"

expected\_filename = f"{expected\_name}.xlsx"

expected\_path = os.path.join(catalog\_dir, expected\_filename)

# 检查文件是否存在

if not os.path.exists(expected\_path):

error\_msg = f"未找到匹配的Excel文件: {expected\_filename}"

logging.warning(error\_msg)

if hasattr(self.main\_window, 'status\_bar'):

self.main\_window.status\_bar.show\_message(error\_msg, "warning")

messagebox.showinfo("提示", error\_msg)

return

# 如果找到了匹配的文件，显示文件信息

logging.info(f"找到匹配的Excel文件: {expected\_path}")

# 获取Excel文件中的所有工作表信息

try:

# 使用pandas读取Excel文件

excel\_file = pd.ExcelFile(expected\_path, engine='openpyxl')

sheet\_names = excel\_file.sheet\_names

# 处理每个工作表

for sheet\_name in sheet\_names:

try:

# 读取工作表数据

df = pd.read\_excel(excel\_file, sheet\_name=sheet\_name, engine='openpyxl', header=0)

# 将数据添加到文件列表

for \_, row in df.iterrows():

# 获取材料名称、日期和页数

material\_name = str(row.get('材料名称', '')) if pd.notna(row.get('材料名称')) else ''

file\_date = str(row.get('日期', '')) if pd.notna(row.get('日期')) else ''

page\_count = str(int(row.get('页数', ''))) if pd.notna(row.get('页数')) else ''

# 插入到列表

self.main\_window.file\_list.insert('', 'end', values=(

search\_id,

search\_name,

sheet\_name, # 使用工作表名作为类号

material\_name,

expected\_filename, # 显示Excel文件名

file\_date,

page\_count,

expected\_path # 文件路径

))

except Exception as e:

logging.error(f"处理工作表 {sheet\_name} 时出错: {str(e)}")

continue

except Exception as e:

error\_msg = f"读取Excel文件时出错: {str(e)}"

logging.error(error\_msg, exc\_info=True)

messagebox.showerror("错误", error\_msg)

return

# 更新状态栏

if hasattr(self.main\_window, 'status\_bar'):

item\_count = len(self.main\_window.file\_list.get\_children())

if item\_count > 0:

self.main\_window.status\_bar.show\_message(f"找到 {item\_count} 条记录", "info")

else:

self.main\_window.status\_bar.show\_message("未找到匹配的记录", "warning")

def extract\_category\_num(self, filename):

"""从文件名中提取分类号"""

# 去掉扩展名

name = os.path.splitext(filename)[0]

# 分类号通常是数字或数字-数字的格式

parts = name.split('-')

if not parts or not parts[0].isdigit():

return None

# 构建完整类号 - 包括所有数字部分

class\_parts = []

for part in parts:

if part.isdigit():

class\_parts.append(part)

else:

break

if not class\_parts:

return None

# 返回完整的类号格式，如 "1-1", "4-1-2"

return "-".join(class\_parts)

def open\_install\_directory(self):

"""打开程序安装目录"""

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

self.\_open\_directory(install\_dir)

def open\_archive\_directory(self):

"""打开档案文件目录"""

# 如果没有设置导入目录，则使用默认目录

if not self.main\_window.import\_root\_dir or not os.path.exists(self.main\_window.import\_root\_dir):

# 使用用户主目录作为默认目录

default\_dir = os.path.expanduser('~')

# 让用户选择目录

selected\_dir = filedialog.askdirectory(

title="选择档案文件目录",

initialdir=default\_dir

)

if not selected\_dir: # 用户取消了选择

return

# 更新导入目录

self.main\_window.import\_root\_dir = selected\_dir

self.import\_root\_dir = selected\_dir

# 保存设置

self.main\_window.save\_settings()

# 打开目录

self.\_open\_directory(self.main\_window.import\_root\_dir)

def open\_database\_location(self):

"""打开数据库位置"""

if not self.db or not hasattr(self.db, 'db\_path'):

messagebox.showwarning("警告", "数据库未初始化")

return

db\_dir = os.path.dirname(self.db.db\_path)

self.\_open\_directory(db\_dir)

def \_open\_directory(self, directory):

"""打开指定目录"""

try:

if os.name == 'nt': # Windows

os.startfile(directory)

elif os.name == 'posix': # macOS, Linux

subprocess.call(('xdg-open', directory))

logging.info(f"打开目录: {directory}")

except Exception as e:

logging.error(f"打开目录失败: {str(e)}")

messagebox.showerror("错误", f"打开目录失败: {str(e)}")

dialogs.py

import tkinter as tk

from tkinter import ttk, messagebox, simpledialog

import logging

import os

import json

import hashlib

class DialogManager:

"""对话框管理类，包含所有对话框相关的功能"""

def \_\_init\_\_(self, main\_window):

self.main\_window = main\_window

self.root = main\_window.root

self.db = main\_window.db

def show\_login\_dialog(self):

"""显示登录对话框"""

logging.debug("显示登录对话框")

# 如果已经登录，显示信息并提供退出选项

if self.main\_window.current\_user:

result = messagebox.askyesno("已登录",

f"当前已经以 {self.main\_window.current\_user[2]} 身份登录。\n\n是否要退出登录？")

if result:

self.main\_window.logout()

return

# 创建登录对话框

login\_dialog = tk.Toplevel(self.root)

login\_dialog.title("用户登录")

login\_dialog.geometry("300x180") # 增加对话框尺寸

login\_dialog.resizable(False, False)

login\_dialog.transient(self.root)

login\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(login\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 用户名

ttk.Label(form\_frame, text="用户名:").grid(row=0, column=0, sticky=tk.W, pady=5)

username\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=username\_var, width=20).grid(row=0, column=1, pady=5)

# 密码

ttk.Label(form\_frame, text="密码:").grid(row=1, column=0, sticky=tk.W, pady=5)

password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=password\_var, show="\*", width=20).grid(row=1, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=2, column=0, columnspan=2, pady=5)

# 登录按钮

def do\_login():

username = username\_var.get().strip()

password = password\_var.get()

if not username or not password:

error\_var.set("用户名和密码不能为空")

return

# 验证登录

success, user = self.main\_window.validate\_login(username, password)

if success:

self.main\_window.current\_user = user

self.main\_window.login\_status\_var.set(f"当前用户: {user[2]}")

# 更新菜单

self.main\_window.update\_menu\_by\_role(user[3])

# 关闭对话框

login\_dialog.destroy()

# 登录成功后，启用搜索按钮和导入文件按钮

self.main\_window.update\_tools\_permission(user[3] == 'admin')

logging.info(f"用户 {username} 登录成功")

else:

error\_var.set("用户名或密码错误")

logging.warning(f"用户 {username} 登录失败")

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=3, column=0, columnspan=2, pady=10)

# 添加更大的按钮

ttk.Button(buttons\_frame, text="登录", command=do\_login, width=10).pack(side=tk.LEFT, padx=10)

ttk.Button(buttons\_frame, text="取消", command=login\_dialog.destroy, width=10).pack(side=tk.LEFT, padx=10)

# 居中对话框

login\_dialog.update\_idletasks()

width = login\_dialog.winfo\_width()

height = login\_dialog.winfo\_height()

x = (login\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (login\_dialog.winfo\_screenheight() // 2) - (height // 2)

login\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

login\_dialog.wait\_window()

def show\_register\_dialog(self):

"""显示用户注册对话框（仅管理员可用）"""

logging.debug("显示注册对话框")

# 检查权限

if not self.main\_window.current\_user or self.main\_window.current\_user[3] != 'admin':

messagebox.showerror("权限错误", "只有管理员可以添加用户")

return

# 创建注册对话框

register\_dialog = tk.Toplevel(self.root)

register\_dialog.title("添加用户")

register\_dialog.geometry("350x250") # 进一步增加对话框尺寸

register\_dialog.resizable(False, False)

register\_dialog.transient(self.root)

register\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(register\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 用户名

ttk.Label(form\_frame, text="用户名:").grid(row=0, column=0, sticky=tk.W, pady=5)

username\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=username\_var, width=20).grid(row=0, column=1, pady=5)

# 密码

ttk.Label(form\_frame, text="密码:").grid(row=1, column=0, sticky=tk.W, pady=5)

password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=password\_var, show="\*", width=20).grid(row=1, column=1, pady=5)

# 确认密码

ttk.Label(form\_frame, text="确认密码:").grid(row=2, column=0, sticky=tk.W, pady=5)

confirm\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=confirm\_password\_var, show="\*", width=20).grid(row=2, column=1, pady=5)

# 真实姓名

ttk.Label(form\_frame, text="真实姓名:").grid(row=3, column=0, sticky=tk.W, pady=5)

real\_name\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=real\_name\_var, width=20).grid(row=3, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=4, column=0, columnspan=2, pady=5)

# 注册按钮

def do\_register():

username = username\_var.get().strip()

password = password\_var.get()

confirm\_password = confirm\_password\_var.get()

real\_name = real\_name\_var.get().strip()

# 验证表单

if not username:

error\_var.set("用户名不能为空")

return

if not password:

error\_var.set("密码不能为空")

return

if password != confirm\_password:

error\_var.set("两次输入的密码不一致")

return

if not real\_name:

error\_var.set("请输入真实姓名")

return

# 注册用户

success, message = self.main\_window.register\_user(username, password, real\_name)

if success:

messagebox.showinfo("添加成功", message)

register\_dialog.destroy()

else:

error\_var.set(message)

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=5, column=0, columnspan=2, pady=15) # 增加垂直间距

# 添加更大的按钮

ttk.Button(buttons\_frame, text="确定", command=do\_register, width=15).pack(side=tk.LEFT, padx=15) # 增加按钮宽度和间距

ttk.Button(buttons\_frame, text="取消", command=register\_dialog.destroy, width=15).pack(side=tk.LEFT, padx=15) # 增加按钮宽度和间距

# 居中对话框

register\_dialog.update\_idletasks()

width = register\_dialog.winfo\_width()

height = register\_dialog.winfo\_height()

x = (register\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (register\_dialog.winfo\_screenheight() // 2) - (height // 2)

register\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

register\_dialog.wait\_window()

def show\_change\_password\_dialog(self):

"""显示修改密码对话框"""

# 检查权限

if not self.main\_window.current\_user or self.main\_window.current\_user[3] != 'admin':

messagebox.showerror("权限错误", "只有管理员可以修改密码")

return

# 创建修改密码对话框

change\_pwd\_dialog = tk.Toplevel(self.root)

change\_pwd\_dialog.title("修改密码")

change\_pwd\_dialog.geometry("320x200") # 增加对话框尺寸

change\_pwd\_dialog.resizable(False, False)

change\_pwd\_dialog.transient(self.root)

change\_pwd\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(change\_pwd\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 旧密码

ttk.Label(form\_frame, text="当前密码:").grid(row=0, column=0, sticky=tk.W, pady=5)

old\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=old\_password\_var, show="\*", width=20).grid(row=0, column=1, pady=5)

# 新密码

ttk.Label(form\_frame, text="新密码:").grid(row=1, column=0, sticky=tk.W, pady=5)

new\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=new\_password\_var, show="\*", width=20).grid(row=1, column=1, pady=5)

# 确认新密码

ttk.Label(form\_frame, text="确认新密码:").grid(row=2, column=0, sticky=tk.W, pady=5)

confirm\_password\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=confirm\_password\_var, show="\*", width=20).grid(row=2, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=3, column=0, columnspan=2, pady=5)

# 修改密码按钮

def do\_change\_password():

old\_password = old\_password\_var.get()

new\_password = new\_password\_var.get()

confirm\_password = confirm\_password\_var.get()

# 验证表单

if not old\_password or not new\_password or not confirm\_password:

error\_var.set("所有密码字段不能为空")

return

if new\_password != confirm\_password:

error\_var.set("两次输入的新密码不一致")

return

# 验证旧密码

success, \_ = self.main\_window.validate\_login(self.main\_window.current\_user[1], old\_password)

if not success:

error\_var.set("当前密码错误")

return

# 修改密码

try:

hashed\_password = self.main\_window.hash\_password(new\_password)

self.db.cursor.execute(

'UPDATE users SET password = ? WHERE id = ?',

(hashed\_password, self.main\_window.current\_user[0])

)

self.db.conn.commit()

messagebox.showinfo("成功", "密码修改成功！")

change\_pwd\_dialog.destroy()

except Exception as e:

error\_var.set(f"修改失败: {str(e)}")

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=4, column=0, columnspan=2, pady=10)

# 添加更大的按钮

ttk.Button(buttons\_frame, text="确定", command=do\_change\_password, width=10).pack(side=tk.LEFT, padx=10)

ttk.Button(buttons\_frame, text="取消", command=change\_pwd\_dialog.destroy, width=10).pack(side=tk.LEFT, padx=10)

# 居中对话框

change\_pwd\_dialog.update\_idletasks()

width = change\_pwd\_dialog.winfo\_width()

height = change\_pwd\_dialog.winfo\_height()

x = (change\_pwd\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (change\_pwd\_dialog.winfo\_screenheight() // 2) - (height // 2)

change\_pwd\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

change\_pwd\_dialog.wait\_window()

def show\_registration\_dialog(self):

"""显示系统注册对话框"""

# 创建注册对话框

registration\_dialog = tk.Toplevel(self.root)

registration\_dialog.title("系统注册")

registration\_dialog.geometry("400x250")

registration\_dialog.resizable(False, False)

registration\_dialog.transient(self.root)

registration\_dialog.grab\_set()

# 表单框架

form\_frame = ttk.Frame(registration\_dialog, padding=20)

form\_frame.pack(fill=tk.BOTH, expand=True)

# 注册码

ttk.Label(form\_frame, text="注册码:").grid(row=0, column=0, sticky=tk.W, pady=5)

reg\_code\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=reg\_code\_var, width=30).grid(row=0, column=1, pady=5)

# 用户名

ttk.Label(form\_frame, text="单位名称:").grid(row=1, column=0, sticky=tk.W, pady=5)

user\_name\_var = tk.StringVar()

ttk.Entry(form\_frame, textvariable=user\_name\_var, width=30).grid(row=1, column=1, pady=5)

# 错误信息

error\_var = tk.StringVar()

error\_label = ttk.Label(form\_frame, textvariable=error\_var, foreground="red")

error\_label.grid(row=2, column=0, columnspan=2, pady=5)

# 注册按钮

def do\_register():

reg\_code = reg\_code\_var.get().strip()

user\_name = user\_name\_var.get().strip()

# 验证表单

if not reg\_code:

error\_var.set("注册码不能为空")

return

if not user\_name:

error\_var.set("单位名称不能为空")

return

# 注册系统

success, message = self.main\_window.register\_system(reg\_code, user\_name)

if success:

messagebox.showinfo("注册成功", message)

# 更新UI

self.main\_window.is\_registered = True

self.main\_window.update\_ui\_by\_registration()

registration\_dialog.destroy()

else:

error\_var.set(message)

# 创建一个按钮框架

buttons\_frame = ttk.Frame(form\_frame)

buttons\_frame.grid(row=3, column=0, columnspan=2, pady=15)

# 添加更大的按钮

ttk.Button(buttons\_frame, text="注册", command=do\_register, width=15).pack(side=tk.LEFT, padx=15)

ttk.Button(buttons\_frame, text="取消", command=registration\_dialog.destroy, width=15).pack(side=tk.LEFT, padx=15)

# 居中对话框

registration\_dialog.update\_idletasks()

width = registration\_dialog.winfo\_width()

height = registration\_dialog.winfo\_height()

x = (registration\_dialog.winfo\_screenwidth() // 2) - (width // 2)

y = (registration\_dialog.winfo\_screenheight() // 2) - (height // 2)

registration\_dialog.geometry('{}x{}+{}+{}'.format(width, height, x, y))

registration\_dialog.wait\_window()

def show\_help(self):

"""显示帮助信息"""

help\_text = """

档案检索系统使用说明

1. 登录系统

- 使用管理员账号（admin/admin123）或已注册的用户账号登录

2. 导入档案

- 点击"导入文件"按钮，选择包含PDF档案的目录

- 系统将自动扫描目录中的PDF文件并导入

3. 搜索档案

- 在搜索框中输入姓名或编号进行搜索

- 可以通过左侧分类树筛选特定类别的档案

4. 查看档案

- 双击文件列表中的档案可以打开查看

5. 管理功能（仅管理员）

- 添加用户：文件菜单 -> 添加用户

- 清理数据库：工具菜单 -> 清理数据库

"""

messagebox.showinfo("使用说明", help\_text)

def show\_about(self):

"""显示关于信息"""

about\_text = f"""

档案检索系统

版本: V1.0

本系统用于管理和检索电子档案，提供简便的导入、搜索和查看功能。

技术支持:

电话: 028-12345678

邮箱: support@example.com

"""

messagebox.showinfo("关于", about\_text)

security\_launcher.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

档案安全模块启动器

用于启动档案完整性检查工具

"""

import os

import sys

import tkinter as tk

from tkinter import messagebox

import logging

import threading

import json

import datetime

# 确保导入路径正确

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

parent\_dir = os.path.dirname(current\_dir)

if parent\_dir not in sys.path:

sys.path.insert(0, parent\_dir)

# 导入核心模块

try:

from core.integrity import IntegrityChecker

from utils.hash\_utils import calculate\_file\_hash

from utils.file\_utils import scan\_directory

from utils.config\_utils import load\_config, save\_config

from utils.logging\_utils import setup\_logger, get\_logger

# 设置日志

log\_dir = os.path.join(os.path.expanduser("~"), "ArchiMgr", "logs")

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

log\_file = os.path.join(log\_dir, f"security\_{datetime.datetime.now().strftime('%Y%m%d')}.log")

setup\_logger(log\_file)

logger = get\_logger("SecurityModule")

logger.info("安全模块启动器初始化")

except ImportError as e:

print(f"导入错误: {e}")

print(f"当前路径: {os.getcwd()}")

print(f"sys.path: {sys.path}")

input("按回车键退出...")

sys.exit(1)

class LogHandler(logging.Handler):

"""将日志输出到文本控件的处理器"""

def \_\_init\_\_(self, text\_widget):

logging.Handler.\_\_init\_\_(self)

self.text\_widget = text\_widget

self.formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')

def emit(self, record):

msg = self.formatter.format(record)

def append():

self.text\_widget.configure(state='normal')

self.text\_widget.insert(tk.END, msg + '\n')

self.text\_widget.see(tk.END)

self.text\_widget.configure(state='disabled')

# 在主线程中更新UI

self.text\_widget.after(0, append)

class IntegrityCheckApp:

"""完整性检查应用程序"""

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

self.root = root

self.root.title("档案完整性检查工具")

self.root.geometry("900x700")

self.root.minsize(800, 600)

# 配置

self.config\_file = os.path.join(os.path.expanduser("~"), "ArchiMgr", "config", "integrity\_check.json")

os.makedirs(os.path.dirname(self.config\_file), exist\_ok=True)

self.config = self.load\_config()

# 创建界面

self.create\_ui()

# 设置日志输出到文本控件

self.log\_handler = LogHandler(self.log\_text)

logger.addHandler(self.log\_handler)

logger.info("档案完整性检查工具已启动")

def load\_config(self):

"""加载配置"""

default\_config = {

"archive\_dir": "",

"database\_file": ""

}

try:

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

config = load\_config(self.config\_file)

logger.info(f"已加载配置: {self.config\_file}")

# 确保所有必要的键都存在

for key in default\_config:

if key not in config:

config[key] = default\_config[key]

return config

except Exception as e:

logger.error(f"加载配置失败: {e}")

return default\_config

def save\_config(self):

"""保存配置"""

try:

save\_config(self.config\_file, self.config)

logger.info(f"已保存配置: {self.config\_file}")

except Exception as e:

logger.error(f"保存配置失败: {e}")

def create\_ui(self):

"""创建用户界面"""

# 创建主框架

main\_frame = tk.Frame(self.root)

main\_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

# 创建顶部控制区域

control\_frame = tk.LabelFrame(main\_frame, text="控制面板", padx=10, pady=10)

control\_frame.pack(fill=tk.X, pady=(0, 10))

# 档案目录设置

dir\_frame = tk.Frame(control\_frame)

dir\_frame.pack(fill=tk.X, pady=5)

tk.Label(dir\_frame, text="档案目录:").pack(side=tk.LEFT)

self.dir\_entry = tk.Entry(dir\_frame)

self.dir\_entry.pack(side=tk.LEFT, fill=tk.X, expand=True, padx=(5, 5))

if self.config["archive\_dir"]:

self.dir\_entry.insert(0, self.config["archive\_dir"])

browse\_dir\_btn = tk.Button(dir\_frame, text="浏览...", command=self.browse\_directory)

browse\_dir\_btn.pack(side=tk.RIGHT)

# 数据库文件设置

db\_frame = tk.Frame(control\_frame)

db\_frame.pack(fill=tk.X, pady=5)

tk.Label(db\_frame, text="数据库文件:").pack(side=tk.LEFT)

self.db\_entry = tk.Entry(db\_frame)

self.db\_entry.pack(side=tk.LEFT, fill=tk.X, expand=True, padx=(5, 5))

if self.config["database\_file"]:

self.db\_entry.insert(0, self.config["database\_file"])

browse\_db\_btn = tk.Button(db\_frame, text="浏览...", command=self.browse\_database)

browse\_db\_btn.pack(side=tk.RIGHT)

# 操作按钮

btn\_frame = tk.Frame(control\_frame)

btn\_frame.pack(fill=tk.X, pady=10)

self.build\_btn = tk.Button(btn\_frame, text="构建完整性数据库", command=self.build\_database)

self.build\_btn.pack(side=tk.LEFT, padx=5)

self.verify\_btn = tk.Button(btn\_frame, text="验证文件完整性", command=self.verify\_integrity)

self.verify\_btn.pack(side=tk.LEFT, padx=5)

self.update\_btn = tk.Button(btn\_frame, text="更新完整性数据库", command=self.update\_database)

self.update\_btn.pack(side=tk.LEFT, padx=5)

# 状态栏

self.status\_var = tk.StringVar()

self.status\_var.set("就绪")

status\_bar = tk.Label(main\_frame, textvariable=self.status\_var, bd=1, relief=tk.SUNKEN, anchor=tk.W)

status\_bar.pack(side=tk.BOTTOM, fill=tk.X)

# 日志显示区域

log\_frame = tk.LabelFrame(main\_frame, text="操作日志", padx=10, pady=10)

log\_frame.pack(fill=tk.BOTH, expand=True)

# 创建滚动条

scrollbar = tk.Scrollbar(log\_frame)

scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

# 创建文本控件

self.log\_text = tk.Text(log\_frame, wrap=tk.WORD, yscrollcommand=scrollbar.set)

self.log\_text.pack(fill=tk.BOTH, expand=True)

self.log\_text.configure(state='disabled')

# 配置滚动条

scrollbar.config(command=self.log\_text.yview)

def browse\_directory(self):

"""浏览选择档案目录"""

from tkinter import filedialog

directory = filedialog.askdirectory(title="选择档案目录")

if directory:

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

self.dir\_entry.insert(0, directory)

self.config["archive\_dir"] = directory

self.save\_config()

logger.info(f"已设置档案目录: {directory}")

def browse\_database(self):

"""浏览选择数据库文件"""

from tkinter import filedialog

file\_path = filedialog.asksaveasfilename(

title="选择数据库文件",

defaultextension=".json",

filetypes=[("JSON文件", "\*.json"), ("所有文件", "\*.\*")]

)

if file\_path:

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

self.db\_entry.insert(0, file\_path)

self.config["database\_file"] = file\_path

self.save\_config()

logger.info(f"已设置数据库文件: {file\_path}")

def update\_status(self, message):

"""更新状态栏"""

self.status\_var.set(message)

def disable\_buttons(self):

"""禁用按钮"""

self.build\_btn.config(state=tk.DISABLED)

self.verify\_btn.config(state=tk.DISABLED)

self.update\_btn.config(state=tk.DISABLED)

def enable\_buttons(self):

"""启用按钮"""

self.build\_btn.config(state=tk.NORMAL)

self.verify\_btn.config(state=tk.NORMAL)

self.update\_btn.config(state=tk.NORMAL)

def build\_database(self):

"""构建完整性数据库"""

archive\_dir = self.dir\_entry.get()

database\_file = self.db\_entry.get()

if not archive\_dir or not database\_file:

messagebox.showwarning("警告", "请先设置档案目录和数据库文件")

return

self.disable\_buttons()

self.update\_status("正在构建完整性数据库...")

def task():

try:

checker = IntegrityChecker(archive\_dir, database\_file)

checker.build\_database()

self.root.after(0, lambda: self.update\_status("完整性数据库构建完成"))

except Exception as e:

logger.error(f"构建完整性数据库失败: {e}")

self.root.after(0, lambda: self.update\_status("构建失败"))

finally:

self.root.after(0, self.enable\_buttons)

threading.Thread(target=task).start()

def verify\_integrity(self):

"""验证文件完整性"""

archive\_dir = self.dir\_entry.get()

database\_file = self.db\_entry.get()

if not archive\_dir or not database\_file:

messagebox.showwarning("警告", "请先设置档案目录和数据库文件")

return

if not os.path.exists(database\_file):

messagebox.showwarning("警告", "数据库文件不存在，请先构建完整性数据库")

return

self.disable\_buttons()

self.update\_status("正在验证文件完整性...")

def task():

try:

checker = IntegrityChecker(archive\_dir, database\_file)

result = checker.verify\_integrity()

# 显示结果统计

modified\_count = len(result.get('modified', []))

missing\_count = len(result.get('missing', []))

new\_count = len(result.get('new', []))

summary = f"完整性验证完成: 修改={modified\_count}, 丢失={missing\_count}, 新增={new\_count}"

logger.info(summary)

# 显示详细结果

if modified\_count > 0:

logger.info("修改的文件:")

for item in result.get('modified', []):

logger.info(f" {item}")

if missing\_count > 0:

logger.info("丢失的文件:")

for item in result.get('missing', []):

logger.info(f" {item}")

if new\_count > 0:

logger.info("新增的文件:")

for item in result.get('new', []):

logger.info(f" {item}")

self.root.after(0, lambda: self.update\_status(summary))

except Exception as e:

logger.error(f"验证文件完整性失败: {e}")

self.root.after(0, lambda: self.update\_status("验证失败"))

finally:

self.root.after(0, self.enable\_buttons)

threading.Thread(target=task).start()

def update\_database(self):

"""更新完整性数据库"""

archive\_dir = self.dir\_entry.get()

database\_file = self.db\_entry.get()

if not archive\_dir or not database\_file:

messagebox.showwarning("警告", "请先设置档案目录和数据库文件")

return

if not os.path.exists(database\_file):

messagebox.showwarning("警告", "数据库文件不存在，请先构建完整性数据库")

return

self.disable\_buttons()

self.update\_status("正在更新完整性数据库...")

def task():

try:

checker = IntegrityChecker(archive\_dir, database\_file)

checker.update\_database()

self.root.after(0, lambda: self.update\_status("完整性数据库更新完成"))

except Exception as e:

logger.error(f"更新完整性数据库失败: {e}")

self.root.after(0, lambda: self.update\_status("更新失败"))

finally:

self.root.after(0, self.enable\_buttons)

threading.Thread(target=task).start()

def main():

"""主函数"""

root = tk.Tk()

app = IntegrityCheckApp(root)

root.mainloop()

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

main()

excel\_utils.py

import os

import logging

import sys

# 全局变量，用于存储 openpyxl 模块

openpyxl\_available = False

try:

import openpyxl

openpyxl\_available = True

except ImportError:

error\_msg = "警告: 未找到 openpyxl 模块，Excel 功能将无法正常工作"

print(error\_msg)

logging.warning(error\_msg)

# 导入 pandas

try:

import pandas as pd

# 不要全局设置引擎，而是在每次调用时指定

except ImportError as e:

print(f"警告: {str(e)}")

logging.warning(f"导入 pandas 失败: {str(e)}")

class ExcelFileNotFound(Exception):

pass

def get\_excel\_info(import\_root\_dir, person\_name, person\_id, class\_code):

"""

从Excel获取文件相关信息。

:param import\_root\_dir: Excel文件根目录

:param person\_name: 人名

:param person\_id: 编号

:param class\_code: 类号

:return: (material\_name, file\_date, page\_count, data\_list)

:raises ExcelFileNotFound: 未找到匹配的Excel文件

"""

material\_name = ""

file\_date = ""

page\_count = ""

data\_list = []

# 记录调试信息

logging.info("="\*50)

logging.info(f"开始处理Excel信息")

logging.info(f"- 根目录: {os.path.abspath(import\_root\_dir)}")

logging.info(f"- 搜索条件: 姓名='{person\_name}', 编号='{person\_id}', 类号='{class\_code}'")

# 解析sheet名 - 保留完整类号格式进行记录

original\_class\_code = class\_code

class\_parts = class\_code.split('-')

main\_code = class\_parts[0]

sheet\_code = main\_code

if main\_code in ['4', '9'] and len(class\_parts) >= 2:

sheet\_code = f"{main\_code}-{class\_parts[1]}"

# 添加详细日志

logging.info(f"解析类号 '{class\_code}' 为sheet代码 '{sheet\_code}'，原始类号 '{original\_class\_code}'")

# 映射表 - 将阿拉伯数字映射到中文数字和Excel工作表名称

sheet\_map = {

'1': '一', '2': '二', '3': '三',

'4-1': '四-1', '4-2': '四-2', '4-3': '四-3', '4-4': '四-4',

'5': '五', '6': '六', '7': '七', '8': '八',

'9-1': '九-1', '9-2': '九-2', '9-3': '九-3', '9-4': '九-4',

'10': '十'

}

# 获取对应的工作表名称

sheet\_name = sheet\_map.get(sheet\_code)

logging.info(f"映射sheet代码 '{sheet\_code}' 到工作表名称 '{sheet\_name}'")

# 如果找不到对应的工作表名称，尝试使用一级类号

if not sheet\_name and len(class\_parts) >= 2:

sheet\_name = sheet\_map.get(main\_code)

logging.info(f"尝试使用一级类号 '{main\_code}' 映射到工作表名称 '{sheet\_name}'")

if not sheet\_name:

logging.warning(f"无法将类号 '{class\_code}' 映射到工作表名称，sheet\_code='{sheet\_code}'")

return material\_name, file\_date, page\_count, data\_list

# 只搜索'目录'子目录

catalog\_dir = os.path.join(import\_root\_dir, '目录')

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

logging.warning(f"目录不存在: {catalog\_dir}")

raise ExcelFileNotFound(f"目录不存在: {catalog\_dir}")

logging.info(f"正在搜索目录: {catalog\_dir}")

# 构建预期的文件名（编号+姓名，无空格）

expected\_name = f"{person\_id}{person\_name}"

expected\_filename = f"{expected\_name}.xlsx"

logging.info(f"正在搜索Excel文件，目标文件名: {expected\_filename}")

file\_path = os.path.join(catalog\_dir, expected\_filename)

# 检查文件是否存在

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

abs\_path = os.path.abspath(file\_path)

logging.info(f"找到匹配的Excel文件: {abs\_path}")

logging.info(f"正在读取工作表: {sheet\_name}")

try:

# 传递原始完整类号

material\_name, file\_date, page\_count, data\_list = read\_excel\_file(file\_path, sheet\_name, original\_class\_code)

logging.info(f"成功读取Excel文件: {os.path.basename(file\_path)}, 工作表: {sheet\_name}")

return material\_name, file\_date, page\_count, data\_list

except Exception as e:

logging.error(f"读取Excel文件失败: {str(e)}")

raise

# 如果没有找到匹配的文件，记录日志并抛出异常

error\_msg = f"未找到匹配的Excel文件: {expected\_filename} (在目录: {catalog\_dir})"

logging.error(error\_msg)

logging.error(f"请确保文件按照'编号+姓名.xlsx'的格式命名，例如: {expected\_filename}")

raise ExcelFileNotFound(error\_msg)

def read\_excel\_file(file\_path, sheet\_name, class\_code=None, file\_index=None):

"""

读取Excel文件并返回数据

:param file\_path: Excel文件路径

:param sheet\_name: 工作表名称

:param class\_code: 类号（例如：'1', '4-1', '9-2'等）

:param file\_index: 文件索引（例如：'1-1'中的1-1）

:return: (material\_name, file\_date, page\_count, data\_list)

material\_name: 材料名称

file\_date: 文件日期

page\_count: 页数

data\_list: 包含所有行的数据列表，每行格式为 [类号, 材料名称, 日期, 页数]

"""

material\_name = ""

file\_date = ""

page\_count = ""

data\_list = []

logging.info(f"开始读取Excel文件: {os.path.abspath(file\_path)}")

logging.info(f"工作表名称: {sheet\_name}, 类号: {class\_code}, 文件索引: {file\_index}")

# 检查 openpyxl 是否可用

if not openpyxl\_available:

error\_msg = "错误: 未找到 openpyxl 模块，无法读取 Excel 文件。请安装: pip install openpyxl"

logging.error(error\_msg)

print(error\_msg)

return "", "", "", []

# 根据类号确定起始行

start\_row = 5 # 默认从第5行开始

if class\_code:

main\_code = class\_code.split('-')[0]

if main\_code in ['4', '9']: # 第四和第九类从第6行开始

start\_row = 6

logging.info(f"检测到第{main\_code}类，从第{start\_row}行开始读取")

else: # 其他类从第5行开始

logging.info(f"检测到第{main\_code}类，从第{start\_row}行开始读取")

else:

logging.warning(f"未指定类号，使用默认起始行: {start\_row}")

if not os.path.exists(file\_path):

logging.error(f"文件不存在: {file\_path}")

return material\_name, file\_date, page\_count, data\_list

wb = None

try:

# 使用openpyxl读取Excel文件

wb = openpyxl.load\_workbook(file\_path, read\_only=True, data\_only=True)

# 检查工作表是否存在

if sheet\_name not in wb.sheetnames:

logging.warning(f"工作表 '{sheet\_name}' 不存在，使用第一个可用工作表")

sheet\_name = wb.sheetnames[0]

ws = wb[sheet\_name]

# 使用固定的列索引

# A列(索引0): 类号

# B列(索引1): 材料名称

# C/D/E列(索引2/3/4): 日期（年/月/日）

# F列(索引5): 页数

name\_col = 1 # B列

date\_cols = [2, 3, 4] # C/D/E列

page\_col = 5 # F列

logging.info("使用固定列索引:")

logging.info(f"- 类号: A列(索引0)")

logging.info(f"- 材料名称: B列(索引{name\_col})")

logging.info(f"- 日期: C/D/E列(索引{date\_cols}): 年/月/日")

logging.info(f"- 页数: F列(索引{page\_col})")

logging.info(f"- 起始行: 第{start\_row}行")

# 从起始行开始读取数据

row\_count = 0

empty\_row\_count = 0

max\_empty\_rows = 7 # 连续空行的最大数量

for row\_idx, row in enumerate(ws.iter\_rows(min\_row=start\_row, values\_only=True), start=start\_row):

# 检查是否连续多行为空

if not any(cell is not None and str(cell).strip() for cell in row):

empty\_row\_count += 1

if empty\_row\_count >= max\_empty\_rows:

logging.info(f"检测到连续 {empty\_row\_count} 个空行，停止读取")

break

continue

else:

empty\_row\_count = 0

# 获取类号

row\_class = str(row[0]).strip() if row[0] is not None else ""

# 获取材料名称

material = str(row[name\_col]).strip() if len(row) > name\_col and row[name\_col] is not None else ""

# 获取日期（年/月/日）

year = str(int(row[date\_cols[0]])) if len(row) > date\_cols[0] and row[date\_cols[0]] is not None and str(row[date\_cols[0]]).strip() != '' else ""

month = str(int(row[date\_cols[1]])) if len(date\_cols) > 1 and len(row) > date\_cols[1] and row[date\_cols[1]] is not None and str(row[date\_cols[1]]).strip() != '' else ""

day = str(int(row[date\_cols[2]])) if len(date\_cols) > 2 and len(row) > date\_cols[2] and row[date\_cols[2]] is not None and str(row[date\_cols[2]]).strip() != '' else ""

# 组合日期

date\_parts = [year, month, day]

date\_str = '-'.join([d for d in date\_parts if d])

# 获取页数

pages = str(int(row[page\_col])) if len(row) > page\_col and row[page\_col] is not None and str(row[page\_col]).strip() != '' else "0"

# 添加到数据列表

data\_list.append([row\_class, material, date\_str, pages])

row\_count += 1

logging.info(f"读取完成 - 共 {row\_count} 行数据")

# 如果传入了file\_index，则根据文件索引匹配行

if file\_index and data\_list:

try:

# 计算行索引（从0开始）

# 例如：file\_index='1-1' -> row\_idx = 0 (第1个文件对应第1行数据)

# file\_index='1-2' -> row\_idx = 1 (第2个文件对应第2行数据)

file\_num = int(file\_index.split('-')[-1]) # 获取文件序号

row\_idx = file\_num - 1 # 转换为0-based索引

if 0 <= row\_idx < len(data\_list):

material\_name = data\_list[row\_idx][1] # 材料名称

file\_date = data\_list[row\_idx][2] # 日期

page\_count = data\_list[row\_idx][3] # 页数

logging.info(f"根据文件索引 '{file\_index}' 获取到数据: 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

else:

logging.warning(f"文件索引 '{file\_index}' 超出范围，有效范围: 1-{len(data\_list)}")

except (ValueError, IndexError) as e:

logging.warning(f"无效的文件索引格式: {file\_index}, 错误: {str(e)}")

# 如果传入了class\_code，则尝试匹配对应的行

elif class\_code and data\_list:

# 移除class\_code中的文件扩展名（如果有）

class\_code\_base = class\_code.split('.')[0]

logging.info(f"在Excel数据中查找类号: '{class\_code\_base}'")

# 记录所有Excel中的类号，便于调试

excel\_class\_codes = [item[0] for item in data\_list if item[0]]

logging.info(f"Excel中所有类号: {excel\_class\_codes}")

# 在数据列表中查找匹配的类号

found = False

# 首先尝试完全匹配 - 优先使用完整类号

for item in data\_list:

excel\_class\_code = item[0] # Excel中的类号

# 跳过空类号

if not excel\_class\_code:

continue

# 记录当前比较的类号

logging.info(f"比较类号: 文件='{class\_code\_base}' vs Excel='{excel\_class\_code}'")

# 完全匹配

if excel\_class\_code == class\_code\_base:

material\_name = item[1] # 材料名称

file\_date = item[2] # 日期

page\_count = item[3] # 页数

logging.info(f"找到完全匹配的类号 '{class\_code\_base}': 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

found = True

break

# 如果没有找到完全匹配，尝试其他匹配方式

if not found:

for item in data\_list:

excel\_class\_code = item[0] # Excel中的类号

# 跳过空类号

if not excel\_class\_code:

continue

# 处理第四大类和第九大类的特殊情况（三级类号）

# 例如文件名是9-2-1.pdf，Excel中记录的类号是9-2

if len(class\_code\_base.split('-')) >= 3:

# 提取前两级类号（如9-2-1变为9-2）

parts = class\_code\_base.split('-')

two\_level\_code = f"{parts[0]}-{parts[1]}"

logging.info(f"尝试二级类号匹配: 文件三级类号='{class\_code\_base}' -> 二级类号='{two\_level\_code}' vs Excel='{excel\_class\_code}'")

if excel\_class\_code == two\_level\_code:

material\_name = item[1] # 材料名称

file\_date = item[2] # 日期

page\_count = item[3] # 页数

logging.info(f"找到二级类号匹配 '{two\_level\_code}' (对应三级类号 '{class\_code\_base}'): 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

found = True

break

# 处理二级类号情况（如1-1）

# 例如文件名是1-1.pdf，Excel中可能只有1

elif len(class\_code\_base.split('-')) == 2:

# 提取一级类号（如1-1变为1）

main\_code = class\_code\_base.split('-')[0]

logging.info(f"尝试一级类号匹配: 文件二级类号='{class\_code\_base}' -> 一级类号='{main\_code}' vs Excel='{excel\_class\_code}'")

if excel\_class\_code == main\_code:

material\_name = item[1] # 材料名称

file\_date = item[2] # 日期

page\_count = item[3] # 页数

logging.info(f"找到一级类号匹配 '{main\_code}' (对应二级类号 '{class\_code\_base}'): 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

found = True

break

# 添加前缀匹配逻辑 - 如果Excel类号以class\_code\_base加连字符开头

# 例如：class\_code\_base="1"，Excel类号="1-1"、"1-2"等

elif excel\_class\_code.startswith(f"{class\_code\_base}-") or excel\_class\_code == class\_code\_base:

material\_name = item[1] # 材料名称

file\_date = item[2] # 日期

page\_count = item[3] # 页数

logging.info(f"找到匹配的类号 '{excel\_class\_code}' (对应类号 '{class\_code\_base}'): 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

found = True

break

if not found:

logging.warning(f"未找到匹配的类号: {class\_code\_base}")

# 对于第四大类和第九大类，如果未找到匹配，尝试使用第一行数据

main\_code = class\_code\_base.split('-')[0]

if main\_code in ['4', '9'] and data\_list:

logging.info(f"第{main\_code}类未找到匹配，使用第一行数据")

material\_name = data\_list[0][1] # 材料名称

file\_date = data\_list[0][2] # 日期

page\_count = data\_list[0][3] # 页数

logging.info(f"使用第一行数据: 材料名称='{material\_name}', 日期='{file\_date}', 页数='{page\_count}'")

return material\_name, file\_date, page\_count, data\_list

logging.info(f"读取完成 - 共 {row\_count} 行数据，其中 {len(data\_list)} 行非空")

# 记录提取的信息

logging.info("="\*50)

logging.info(f"提取的信息:")

logging.info(f"- 材料名称: '{material\_name}'")

logging.info(f"- 日期: '{file\_date}'")

logging.info(f"- 页数: '{page\_count}'")

logging.info(f"- 数据行数: {len(data\_list)}")

# 记录第一行数据

if data\_list and len(data\_list) > 0:

logging.info("第一行数据示例:")

for idx, value in enumerate(data\_list[0]):

logging.info(f" 列{idx}: {value}")

logging.info("="\*50)

except Exception as e:

logging.error(f"读取Excel文件失败: {str(e)}", exc\_info=True)

logging.error(f"错误发生在处理文件: {file\_path}")

logging.error(f"工作表: {sheet\_name}")

finally:

# 确保工作簿被正确关闭

if wb is not None:

wb.close()

return material\_name, file\_date, page\_count, data\_list

user\_management.py

import logging

import hashlib

import os

from tkinter import messagebox

from src.utils.pinyin\_util import get\_pinyin # 导入拼音工具，用于用户名验证

class UserManager:

"""用户管理类，包含所有用户管理相关的功能"""

def \_\_init\_\_(self, main\_window):

self.main\_window = main\_window

self.db = main\_window.db

def init\_users\_table(self):

"""初始化用户表"""

try:

self.db.cursor.execute('''

CREATE TABLE IF NOT EXISTS users (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT UNIQUE NOT NULL,

password TEXT NOT NULL,

real\_name TEXT,

role TEXT DEFAULT 'user',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

''')

self.db.conn.commit()

logging.info("用户表初始化完成")

except Exception as e:

logging.error(f"初始化用户表失败: {str(e)}")

def create\_admin\_user(self):

"""创建管理员账号"""

try:

# 检查管理员账号是否存在

self.db.cursor.execute('SELECT COUNT(\*) FROM users WHERE username = ?', ('admin',))

if self.db.cursor.fetchone()[0] == 0:

# 创建管理员账号

hashed\_password = self.hash\_password('admin123')

self.db.cursor.execute(

'INSERT INTO users (username, password, real\_name, role) VALUES (?, ?, ?, ?)',

('admin', hashed\_password, '系统管理员', 'admin')

)

self.db.conn.commit()

logging.info("已创建管理员账号")

else:

logging.info("管理员账号已存在")

except Exception as e:

logging.error(f"创建管理员账号失败: {str(e)}")

def hash\_password(self, password):

"""对密码进行哈希加密"""

return hashlib.sha256(password.encode()).hexdigest()

def register\_user(self, username, password, real\_name):

"""注册新用户"""

try:

# 检查用户名是否已存在

self.db.cursor.execute('SELECT COUNT(\*) FROM users WHERE username = ?', (username,))

if self.db.cursor.fetchone()[0] > 0:

return False, "用户名已存在"

# 验证用户名是否符合拼音规则（必须是真实姓名的小写拼音）

expected\_pinyin = get\_pinyin(real\_name).lower()

if username != expected\_pinyin:

return False, f"用户名必须是真实姓名的小写拼音，应为: {expected\_pinyin}"

# 密码加密

hashed\_password = self.hash\_password(password)

# 插入新用户

self.db.cursor.execute(

'INSERT INTO users (username, password, real\_name) VALUES (?, ?, ?)',

(username, hashed\_password, real\_name)

)

self.db.conn.commit()

return True, "注册成功"

except Exception as e:

logging.error(f"注册用户失败: {str(e)}")

return False, f"注册失败: {str(e)}"

def validate\_login(self, username, password):

"""验证登录信息"""

try:

hashed\_password = self.hash\_password(password)

self.db.cursor.execute(

'SELECT id, username, real\_name, role FROM users WHERE username = ? AND password = ?',

(username, hashed\_password)

)

user = self.db.cursor.fetchone()

if user:

return True, user

else:

return False, None

except Exception as e:

logging.error(f"验证登录失败: {str(e)}")

return False, None

def logout(self):

"""退出登录"""

self.main\_window.current\_user = None

self.main\_window.login\_status\_var.set("未登录")

# 更新菜单

self.main\_window.update\_menu\_by\_role(None)

# 禁用工具栏

self.main\_window.update\_tools\_permission(False)

logging.info("用户已退出登录")

def check\_registration(self):

"""检查是否已注册"""

try:

# 检查配置目录下的注册文件是否存在

config\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'config')

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

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

registration\_file = os.path.join(config\_dir, 'registration.dat')

if os.path.exists(registration\_file):

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

content = f.read().strip()

# 简单解密处理

decoded\_content = ''.join([chr(ord(c) - 1) for c in content])

parts = decoded\_content.split('|')

if len(parts) >= 2:

# 检查是否已注册

valid\_reg\_file = os.path.join(config\_dir, 'valid\_registration.dat')

if os.path.exists(valid\_reg\_file):

with open(valid\_reg\_file, 'r', encoding='utf-8') as vf:

valid\_content = vf.read().strip()

# 解密有效注册信息

valid\_decoded = ''.join([chr(ord(c) - 1) for c in valid\_content])

# 比对注册信息

if decoded\_content == valid\_decoded:

reg\_name = parts[1].strip()

logging.info(f"注册验证成功: {reg\_name}")

return True

# 如果没有找到有效的注册文件，返回False（未注册）

logging.info("系统未注册")

return False

except Exception as e:

logging.error(f"检查注册状态失败: {str(e)}")

# 如果出错，返回False（默认未注册）

return False

def register\_system(self, reg\_code, user\_name):

"""注册系统"""

try:

# 配置目录

config\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_)))), 'config')

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

# 验证注册码与预设的有效注册信息

valid\_reg\_file = os.path.join(config\_dir, 'valid\_registration.dat')

if not os.path.exists(valid\_reg\_file):

return False, "系统未配置有效注册信息"

# 读取有效注册信息

with open(valid\_reg\_file, 'r', encoding='utf-8') as vf:

valid\_content = vf.read().strip()

# 解密有效注册信息

valid\_decoded = ''.join([chr(ord(c) - 1) for c in valid\_content])

valid\_parts = valid\_decoded.split('|')

if len(valid\_parts) < 2:

return False, "系统注册信息格式错误"

valid\_code = valid\_parts[0].strip()

valid\_name = valid\_parts[1].strip()

# 验证输入的注册码和单位名称

if reg\_code != valid\_code or user\_name != valid\_name:

return False, "注册码或单位名称错误"

# 注册信息验证通过，保存到registration.dat

registration\_file = os.path.join(config\_dir, 'registration.dat')

# 简单加密处理

content = f"{reg\_code}|{user\_name}"

encoded\_content = ''.join([chr(ord(c) + 1) for c in content])

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

f.write(encoded\_content)

logging.info(f"系统注册成功: {user\_name}")

return True, "系统注册成功！"

except Exception as e:

logging.error(f"注册系统失败: {str(e)}")

return False, f"注册失败: {str(e)}"

def update\_ui\_by\_registration(self):

"""根据注册状态更新界面"""

try:

# 获取窗口标题控件

title = self.main\_window.root.title()

# 根据注册状态更新标题

if self.main\_window.is\_registered:

# 已注册版本

if not title.endswith("【已注册】"):

self.main\_window.root.title(f"{title} 【已注册】")

# 启用所有功能

if hasattr(self.main\_window, 'import\_file\_btn'):

self.main\_window.import\_file\_btn.config(state='normal')

# 隐藏注册选项

if hasattr(self.main\_window, 'help\_menu'):

try:

self.main\_window.help\_menu.delete("用户注册")

except:

pass

else:

# 未注册版本

if "【已注册】" in title:

self.main\_window.root.title(title.replace(" 【已注册】", ""))

# 添加水印或其他未注册标识

# 这里可以添加水印代码

# 显示注册选项

if hasattr(self.main\_window, 'help\_menu'):

try:

# 检查是否已有注册选项

has\_reg\_option = False

for i in range(self.main\_window.help\_menu.index('end') + 1):

if self.main\_window.help\_menu.entrycget(i, 'label') == "用户注册":

has\_reg\_option = True

break

if not has\_reg\_option:

# 添加注册选项

self.main\_window.help\_menu.add\_command(

label="用户注册",

command=self.main\_window.show\_registration\_dialog

)

except:

pass

logging.info(f"界面已根据注册状态更新: {'已注册' if self.main\_window.is\_registered else '未注册'}")

except Exception as e:

logging.error(f"更新界面失败: {str(e)}")

def cleanup\_database(self):

"""清理数据库中的重复记录和无效记录"""

# 检查权限

if not self.main\_window.current\_user or self.main\_window.current\_user[3] != 'admin':

messagebox.showerror("权限错误", "只有管理员可以清理数据库")

return

try:

# 删除重复记录

self.db.cursor.execute('''

DELETE FROM person\_files

WHERE rowid NOT IN (

SELECT MIN(rowid)

FROM person\_files

GROUP BY file\_path

)

''')

# 删除不存在的文件记录

self.db.cursor.execute('SELECT file\_path FROM person\_files')

for (file\_path,) in self.db.cursor.fetchall():

if not os.path.exists(file\_path):

self.db.cursor.execute('DELETE FROM person\_files WHERE file\_path = ?', (file\_path,))

self.db.conn.commit()

messagebox.showinfo("成功", "数据库清理完成！")

# 重新加载文件列表

self.main\_window.load\_files\_from\_db()

except Exception as e:

logging.error(f"清理数据库失败: {str(e)}")

messagebox.showerror("错误", f"清理数据库失败: {str(e)}")

database.py

import sqlite3

import logging

import os

import sys

class Database:

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

"""初始化数据库连接"""

# 使用路径管理模块获取数据库路径

from src.utils.paths import get\_database\_path

self.db\_path = get\_database\_path()

# 确保数据库目录存在

os.makedirs(os.path.dirname(self.db\_path), exist\_ok=True)

# 建立数据库连接

try:

self.conn = sqlite3.connect(self.db\_path)

self.cursor = self.conn.cursor()

# 创建必要的表

self.\_create\_tables()

# 执行数据库迁移，添加新列

self.\_migrate\_database()

logging.info(f"数据库连接成功: {self.db\_path}")

except Exception as e:

logging.error(f"数据库连接失败: {str(e)}", exc\_info=True)

raise

def get\_connection(self):

"""获取数据库连接对象

返回:

sqlite3.Connection: 数据库连接对象

"""

return self.conn

def \_create\_tables(self):

"""创建数据库表"""

try:

# 分类表

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS categories (

id INTEGER PRIMARY KEY AUTOINCREMENT,

category TEXT NOT NULL,

parent\_category TEXT,

main\_category\_num INTEGER, -- 主分类号（第一级）

sub\_category\_num INTEGER, -- 子分类号（第二级）

UNIQUE(category, parent\_category)

)

''')

# 人员表

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS persons (

id INTEGER PRIMARY KEY AUTOINCREMENT,

name TEXT NOT NULL UNIQUE,

folder\_path TEXT NOT NULL

)

''')

# 人员文件表

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS person\_files (

id INTEGER PRIMARY KEY AUTOINCREMENT,

person\_name TEXT NOT NULL,

file\_name TEXT NOT NULL,

file\_path TEXT NOT NULL,

category\_id INTEGER,

dir\_name TEXT, -- 目录名称（包含编号和姓名）

file\_id TEXT, -- 编号

FOREIGN KEY (person\_name) REFERENCES persons(name),

FOREIGN KEY (category\_id) REFERENCES categories(id)

)

''')

# 文件表

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS files (

id INTEGER PRIMARY KEY AUTOINCREMENT,

filename TEXT NOT NULL,

category TEXT NOT NULL,

main\_category\_num INTEGER, -- 主分类号

sub\_category\_num INTEGER, -- 子分类号

import\_time DATETIME NOT NULL,

notes TEXT

)

''')

# 操作日志表

self.cursor.execute('''

CREATE TABLE IF NOT EXISTS operation\_logs (

id INTEGER PRIMARY KEY AUTOINCREMENT,

username TEXT NOT NULL, -- 操作用户

operation\_type TEXT NOT NULL, -- 操作类型: login, search, print, view

target\_id TEXT, -- 目标ID（如档案编号）

target\_name TEXT, -- 目标名称（如档案名称、人员姓名）

details TEXT, -- 操作详情

operation\_time TIMESTAMP DEFAULT CURRENT\_TIMESTAMP, -- 操作时间

ip\_address TEXT, -- IP地址

FOREIGN KEY (username) REFERENCES users(username)

)

''')

self.conn.commit()

logging.info("数据库表创建成功")

except Exception as e:

logging.error(f"创建数据库表失败: {e}")

raise

def \_migrate\_database(self):

"""数据库迁移，添加新列"""

try:

# 检查person\_files表中是否有dir\_name列

self.cursor.execute("PRAGMA table\_info(person\_files)")

columns = [column[1] for column in self.cursor.fetchall()]

# 添加dir\_name列

if 'dir\_name' not in columns:

logging.info("添加dir\_name列到person\_files表")

self.cursor.execute("ALTER TABLE person\_files ADD COLUMN dir\_name TEXT")

# 更新现有数据的dir\_name列

self.cursor.execute('''

UPDATE person\_files

SET dir\_name = person\_name

WHERE dir\_name IS NULL

''')

# 添加file\_id列

if 'file\_id' not in columns:

logging.info("添加file\_id列到person\_files表")

self.cursor.execute("ALTER TABLE person\_files ADD COLUMN file\_id TEXT")

# 更新现有数据的file\_id列，尝试从目录名中提取编号

self.cursor.execute('''

SELECT id, person\_name FROM person\_files WHERE file\_id IS NULL

''')

rows = self.cursor.fetchall()

import re

for row\_id, person\_name in rows:

# 尝试从人名中提取编号

match = re.match(r'^(\d+)(.\*)', person\_name)

if match:

file\_id = match.group(1)

self.cursor.execute('''

UPDATE person\_files

SET file\_id = ?

WHERE id = ?

''', (file\_id, row\_id))

self.conn.commit()

logging.info("数据库迁移成功")

except Exception as e:

self.conn.rollback()

logging.error(f"数据库迁移失败: {str(e)}")

# 不抛出异常，允许程序继续运行

def add\_log(self, username, operation\_type, target\_id=None, target\_name=None, details=None, ip\_address=None):

"""添加操作日志

参数:

username (str): 操作用户名

operation\_type (str): 操作类型 (login, search, print, view)

target\_id (str, optional): 目标ID，如档案编号

target\_name (str, optional): 目标名称，如档案名称、人员姓名

details (str, optional): 操作详情

ip\_address (str, optional): IP地址

"""

try:

self.cursor.execute('''

INSERT INTO operation\_logs

(username, operation\_type, target\_id, target\_name, details, ip\_address)

VALUES (?, ?, ?, ?, ?, ?)

''', (username, operation\_type, target\_id, target\_name, details, ip\_address))

self.conn.commit()

logging.info(f"记录操作日志: 用户={username}, 操作={operation\_type}, 目标={target\_name}")

return True

except Exception as e:

self.conn.rollback()

logging.error(f"记录操作日志失败: {str(e)}")

return False

def get\_logs(self, username=None, operation\_type=None, start\_date=None, end\_date=None, limit=1000):

"""获取操作日志

参数:

username (str, optional): 按用户名筛选

operation\_type (str, optional): 按操作类型筛选

start\_date (str, optional): 开始日期，格式 YYYY-MM-DD

end\_date (str, optional): 结束日期，格式 YYYY-MM-DD

limit (int, optional): 限制返回记录数，默认1000条

返回:

list: 日志记录列表

"""

query = "SELECT \* FROM operation\_logs WHERE 1=1"

params = []

if username:

query += " AND username = ?"

params.append(username)

if operation\_type:

query += " AND operation\_type = ?"

params.append(operation\_type)

if start\_date:

query += " AND date(operation\_time) >= ?"

params.append(start\_date)

if end\_date:

query += " AND date(operation\_time) <= ?"

params.append(end\_date)

query += " ORDER BY operation\_time DESC LIMIT ?"

params.append(limit)

try:

self.cursor.execute(query, params)

return self.cursor.fetchall()

except Exception as e:

logging.error(f"获取操作日志失败: {str(e)}")

return []

utils.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

安全模块工具函数 - 提供文件哈希、元数据获取等功能

"""

import os

import hashlib

import logging

import time

from datetime import datetime

def calculate\_file\_hash(file\_path, algorithm='sha256', chunk\_size=8192):

"""

计算文件的哈希值

参数:

file\_path (str): 文件路径

algorithm (str): 哈希算法，支持 md5, sha1, sha256

chunk\_size (int): 读取文件的块大小

返回:

str: 文件的哈希值，如果出错则返回 None

"""

logger = logging.getLogger('security.utils')

if not os.path.isfile(file\_path):

logger.error(f"文件不存在: {file\_path}")

return None

try:

# 选择哈希算法

if algorithm == 'md5':

hasher = hashlib.md5()

elif algorithm == 'sha1':

hasher = hashlib.sha1()

else:

hasher = hashlib.sha256()

# 分块读取文件并更新哈希

with open(file\_path, 'rb') as f:

for chunk in iter(lambda: f.read(chunk\_size), b''):

hasher.update(chunk)

# 返回十六进制哈希值

return hasher.hexdigest()

except Exception as e:

logger.error(f"计算文件哈希时出错 ({file\_path}): {str(e)}")

return None

def get\_file\_metadata(file\_path):

"""

获取文件元数据

参数:

file\_path (str): 文件路径

返回:

dict: 包含文件元数据的字典

"""

logger = logging.getLogger('security.utils')

if not os.path.isfile(file\_path):

logger.error(f"文件不存在: {file\_path}")

return None

try:

# 获取基本文件信息

stat = os.stat(file\_path)

# 返回元数据字典

return {

'size': stat.st\_size,

'modified\_time': datetime.fromtimestamp(stat.st\_mtime).strftime('%Y-%m-%d %H:%M:%S'),

'created\_time': datetime.fromtimestamp(stat.st\_ctime).strftime('%Y-%m-%d %H:%M:%S'),

'accessed\_time': datetime.fromtimestamp(stat.st\_atime).strftime('%Y-%m-%d %H:%M:%S')

}

except Exception as e:

logger.error(f"获取文件元数据时出错 ({file\_path}): {str(e)}")

return {

'size': 0,

'modified\_time': '',

'created\_time': '',

'accessed\_time': ''

}

def verify\_file\_hash(file\_path, expected\_hash, algorithm='sha256'):

"""

验证文件哈希值是否匹配

参数:

file\_path (str): 文件路径

expected\_hash (str): 期望的哈希值

algorithm (str): 哈希算法

返回:

bool: 哈希值是否匹配

"""

current\_hash = calculate\_file\_hash(file\_path, algorithm)

return current\_hash == expected\_hash

def format\_size(size\_bytes):

"""

格式化文件大小

参数:

size\_bytes (int): 文件大小（字节）

返回:

str: 格式化后的文件大小

"""

if size\_bytes < 1024:

return f"{size\_bytes} B"

elif size\_bytes < 1024 \* 1024:

return f"{size\_bytes/1024:.2f} KB"

elif size\_bytes < 1024 \* 1024 \* 1024:

return f"{size\_bytes/(1024\*1024):.2f} MB"

else:

return f"{size\_bytes/(1024\*1024\*1024):.2f} GB"

paths.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

import os

import sys

import platform

import logging

def get\_app\_name():

"""返回应用程序名称"""

return "档案检索系统"

def get\_application\_path():

"""获取应用程序安装路径"""

if getattr(sys, 'frozen', False):

# 打包后的环境

return os.path.dirname(sys.executable)

else:

# 开发环境

return os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

def get\_user\_data\_dir():

"""获取用户数据目录，兼容不同Windows版本"""

app\_name = get\_app\_name()

# 检测操作系统

system = platform.system()

if system != "Windows":

logging.warning(f"不支持的操作系统: {system}，使用默认路径")

# 获取 AppData 路径

appdata = os.environ.get('APPDATA')

if not appdata:

# 备用方案

appdata = os.path.expanduser('~\\AppData\\Roaming')

if not os.path.exists(appdata):

# 极端情况，使用用户主目录

appdata = os.path.expanduser('~')

logging.warning(f"无法找到 AppData 目录，使用用户主目录: {appdata}")

return os.path.join(appdata, app\_name)

def get\_logs\_dir():

"""获取日志目录"""

logs\_dir = os.path.join(get\_user\_data\_dir(), 'logs')

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

return logs\_dir

def get\_database\_dir():

"""获取数据库目录"""

db\_dir = os.path.join(get\_user\_data\_dir(), 'database')

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

return db\_dir

def get\_database\_path():

"""获取数据库文件路径"""

return os.path.join(get\_database\_dir(), 'archimgr.db')

def get\_temp\_dir():

"""获取临时文件目录"""

temp\_dir = os.path.join(get\_user\_data\_dir(), 'temp')

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

return temp\_dir

def get\_config\_dir():

"""获取配置文件目录"""

config\_dir = os.path.join(get\_user\_data\_dir(), 'config')

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

return config\_dir

def get\_resources\_path():

"""获取资源文件路径，这些是只读的，保留在安装目录"""

if getattr(sys, 'frozen', False):

# 打包后的环境

# 首先检查应用程序目录下的resources

app\_resources = os.path.join(get\_application\_path(), 'resources')

if os.path.exists(app\_resources):

return app\_resources

# 然后检查应用程序目录/src/resources

src\_resources = os.path.join(get\_application\_path(), 'src', 'resources')

if os.path.exists(src\_resources):

return src\_resources

# 如果都找不到，返回默认路径

return os.path.join(get\_application\_path(), 'resources')

else:

# 开发环境

return os.path.join(get\_application\_path(), 'resources')

def get\_icon\_path():

"""获取应用图标路径"""

# 尝试不同的可能位置

possible\_paths = [

os.path.join(get\_resources\_path(), 'app.ico'),

os.path.join(get\_application\_path(), 'resources', 'app.ico'),

os.path.join(get\_application\_path(), 'src', 'resources', 'app.ico')

]

for path in possible\_paths:

if os.path.exists(path):

return path

# 如果都找不到，返回默认路径

return os.path.join(get\_resources\_path(), 'app.ico')

def get\_exports\_dir():

"""获取导出文件目录，默认放在用户的文档目录下"""

try:

# 尝试获取用户文档目录

documents = os.path.join(os.path.expanduser('~'), 'Documents')

if not os.path.exists(documents):

# 备用方案

documents = os.path.expanduser('~')

exports\_dir = os.path.join(documents, get\_app\_name(), 'exports')

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

return exports\_dir

except Exception as e:

# 如果出错，使用应用数据目录

logging.warning(f"无法访问用户文档目录，使用应用数据目录: {str(e)}")

exports\_dir = os.path.join(get\_user\_data\_dir(), 'exports')

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

return exports\_dir

pinyin\_util.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

拼音工具模块 - 提供中文转拼音功能

"""

def get\_pinyin(chinese\_str):

"""

将中文字符串转换为拼音

由于不依赖第三方库，这里只实现一个简单版本，仅返回原字符串

实际使用时建议安装pypinyin库实现完整功能

Args:

chinese\_str: 中文字符串

Returns:

拼音字符串，如果无法转换则返回原字符串

"""

try:

# 尝试导入pypinyin模块

try:

from pypinyin import lazy\_pinyin

return ''.join(lazy\_pinyin(chinese\_str))

except ImportError:

# 如果没有pypinyin模块，则返回原字符串

# 这里可以添加一个警告日志

import logging

logging.warning("无法导入pypinyin模块，无法进行中文转拼音，请安装pypinyin: pip install pypinyin")

return chinese\_str

except Exception as e:

import logging

logging.error(f"转换拼音失败: {str(e)}")

return chinese\_str

# 测试代码

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

test\_str = "张三"

pinyin = get\_pinyin(test\_str)

print(f"原字符串: {test\_str}")

print(f"拼音结果: {pinyin}")

logger.py

import os

import logging

import sys

from datetime import datetime

# 使用兼容性导入

try:

# 打包环境下的导入方式

if getattr(sys, 'frozen', False):

from utils.paths import get\_logs\_dir

else:

# 开发环境下的导入方式

from src.utils.paths import get\_logs\_dir

except ImportError as e:

print(f"日志模块导入错误: {e}")

sys.exit(1)

def setup\_logger():

"""设置日志记录器"""

# 使用路径管理模块获取日志目录

log\_dir = get\_logs\_dir()

# 设置日志格式

log\_format = '%(asctime)s - %(levelname)s - %(message)s'

date\_format = '%Y-%m-%d %H:%M:%S'

# 创建日志文件名 - 商用版使用更简洁的命名

log\_file = os.path.join(log\_dir, f'app\_{datetime.now().strftime("%Y%m%d")}.log')

# 配置根日志记录器 - 商用版使用INFO级别

logging.basicConfig(

level=logging.INFO, # 商用版使用INFO级别，减少调试信息

format=log\_format,

datefmt=date\_format,

handlers=[

logging.FileHandler(log\_file, encoding='utf-8'),

logging.StreamHandler() # 同时输出到控制台

]

)

logging.info("日志系统初始化完成")

# 导出 setup\_logger 函数

\_\_all\_\_ = ['setup\_logger']

config\_manager.py

import configparser

import os

class ConfigManager:

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

self.config = configparser.ConfigParser()

self.config\_file = 'config.ini'

def load\_config(self):

# 加载配置文件

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

self.config.read(self.config\_file, encoding='utf-8')

else:

self.create\_default\_config()

def create\_default\_config(self):

# 创建默认配置

self.config['Paths'] = {

'import\_dir': 'import/',

'archive\_dir': 'archive/'

}

self.save\_config()

def save\_config(self):

# 保存配置到文件

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

self.config.write(f)

preload.py

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

预加载必要的模块，确保它们在打包后的应用程序中可用

"""

import sys

import os

import logging

def preload\_modules():

"""预加载必要的模块"""

try:

# 预加载 openpyxl

import openpyxl

logging.info("成功加载 openpyxl 模块")

# 预加载 pandas

import pandas as pd

# 确保 pandas 可以访问 openpyxl

# 但不要全局设置引擎，因为这可能导致错误

logging.info("成功加载 pandas")

return True

except ImportError as e:

logging.error(f"预加载模块失败: {str(e)}")

return False

settings.py

# 程序的基本配置

APP\_NAME = "人事档案软件"

VERSION = "1.0.0"

# 数据库配置

DATABASE = {

'name': 'personnel.db',

'path': 'database/'

}

# 文件路径配置

PATHS = {

'import\_dir': 'import/',

'archive\_dir': 'archive/',

'logs\_dir': 'logs/'

}

# 用户配置

USERS = {

'admin': {

'password': 'admin123',

'role': 'admin'

}

}

file\_manager.py

import os

import logging

import pandas as pd

class FileManager:

@staticmethod

def import\_categories(excel\_file, db):

logging.info(f"开始导入分类: {excel\_file}")

try:

df = pd.read\_excel(excel\_file)

# 导入逻辑

except Exception as e:

logging.error(f"导入分类失败: {e}")

raise