main.py

import sys

import os

from PyQt5.QtWidgets import QApplication, QMainWindow, QFileDialog, QTreeWidget, QTreeWidgetItem, QWidget

from PyQt5.QtWidgets import QVBoxLayout, QHBoxLayout, QLabel, QLineEdit, QPushButton, QCheckBox

from PyQt5.QtWidgets import QGroupBox, QSpinBox, QProgressBar, QStatusBar, QMessageBox

from PyQt5.QtCore import Qt, QThread, pyqtSignal

from sofcrtpro.file\_scanner import scan\_directory

from sofcrtpro.code\_processor import process\_code\_files

from sofcrtpro.document\_generator import generate\_document

# 添加调试开关

DEBUG = True

def debug\_print(\*args, \*\*kwargs):

"""调试打印函数"""

if DEBUG:

print("[DEBUG]", \*args, \*\*kwargs)

class DocumentGeneratorThread(QThread):

"""文档生成线程，避免生成过程阻塞GUI"""

progress\_signal = pyqtSignal(int)

finished\_signal = pyqtSignal(str)

error\_signal = pyqtSignal(str)

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

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

self.params = params

def run(self):

try:

# 获取参数

classified\_files = self.params['classified\_files']

output\_path = self.params['output\_path']

software\_name = self.params['software\_name']

software\_version = self.params['software\_version']

font\_name\_en = self.params['font\_name\_en']

font\_name\_cn = self.params['font\_name\_cn']

lines\_per\_page = self.params['lines\_per\_page']

# 设置进度

self.progress\_signal.emit(10)

# 调试打印文件列表

debug\_print("DocumentGeneratorThread文件列表:")

for file\_type, files in classified\_files.items():

debug\_print(f"{file\_type}: {len(files)}个文件")

for f in files:

debug\_print(f" - {f}")

# 处理代码文件

merged\_content, pages, processed\_files = process\_code\_files(classified\_files, lines\_per\_page)

self.progress\_signal.emit(50)

# 打印处理后的文件列表

debug\_print(f"处理后的文件列表 ({len(processed\_files)}个):")

for f in processed\_files:

debug\_print(f" - {f}")

# 生成文档

output\_file, stats = generate\_document(

content=merged\_content,

output\_path=output\_path,

file\_list=processed\_files,

output\_format='docx',

software\_name=software\_name,

software\_version=software\_version,

font\_name\_en=font\_name\_en,

font\_name\_cn=font\_name\_cn,

font\_size=10.5

)

self.progress\_signal.emit(100)

self.finished\_signal.emit(output\_file)

except Exception as e:

debug\_print(f"文档生成错误: {str(e)}")

self.error\_signal.emit(str(e))

class SoftCopyrightApp(QMainWindow):

"""软著源代码管理器应用程序主窗口"""

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

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

self.setWindowTitle("软著源代码管理器")

self.setGeometry(100, 100, 1200, 800)

self.setup\_ui()

self.file\_data = {} # 存储文件信息的字典

def setup\_ui(self):

"""设置用户界面"""

# 主布局

main\_layout = QHBoxLayout()

# 左侧面板 - 文件列表

self.file\_tree = QTreeWidget()

self.file\_tree.setHeaderLabels(["文件", "类型", "代码行数"])

self.file\_tree.setColumnWidth(0, 500)

self.file\_tree.itemChanged.connect(self.on\_item\_changed)

# 右侧面板

right\_panel = QVBoxLayout()

# 统计信息区域

stats\_group = QGroupBox("统计信息")

stats\_layout = QVBoxLayout()

self.total\_files\_label = QLabel("总文件数: 0")

self.selected\_files\_label = QLabel("已选文件数: 0")

self.total\_lines\_label = QLabel("总代码行数: 0")

self.estimated\_pages\_label = QLabel("预计生成页数: 0")

stats\_layout.addWidget(self.total\_files\_label)

stats\_layout.addWidget(self.selected\_files\_label)

stats\_layout.addWidget(self.total\_lines\_label)

stats\_layout.addWidget(self.estimated\_pages\_label)

stats\_group.setLayout(stats\_layout)

# 参数设置区域

params\_group = QGroupBox("文档参数设置")

params\_layout = QVBoxLayout()

# 软件名称

name\_layout = QHBoxLayout()

name\_layout.addWidget(QLabel("软件名称:"))

self.name\_input = QLineEdit("SoftCopyrightPro")

name\_layout.addWidget(self.name\_input)

params\_layout.addLayout(name\_layout)

# 版本号

version\_layout = QHBoxLayout()

version\_layout.addWidget(QLabel("版本号:"))

self.version\_input = QLineEdit("v1.0")

version\_layout.addWidget(self.version\_input)

params\_layout.addLayout(version\_layout)

# 输出文件名

output\_name\_layout = QHBoxLayout()

output\_name\_layout.addWidget(QLabel("输出文件名:"))

self.output\_name\_input = QLineEdit("软著源代码")

output\_name\_layout.addWidget(self.output\_name\_input)

params\_layout.addLayout(output\_name\_layout)

# 输出路径

output\_path\_layout = QHBoxLayout()

output\_path\_layout.addWidget(QLabel("输出路径:"))

self.output\_path\_input = QLineEdit("./output")

self.output\_path\_button = QPushButton("浏览...")

self.output\_path\_button.clicked.connect(self.select\_output\_path)

output\_path\_layout.addWidget(self.output\_path\_input)

output\_path\_layout.addWidget(self.output\_path\_button)

params\_layout.addLayout(output\_path\_layout)

# 字体设置

font\_layout = QHBoxLayout()

font\_layout.addWidget(QLabel("英文字体:"))

self.en\_font\_input = QLineEdit("Courier New")

font\_layout.addWidget(self.en\_font\_input)

font\_layout.addWidget(QLabel("中文字体:"))

self.cn\_font\_input = QLineEdit("SimSun")

font\_layout.addWidget(self.cn\_font\_input)

params\_layout.addLayout(font\_layout)

# 每页最少行数

lines\_layout = QHBoxLayout()

lines\_layout.addWidget(QLabel("每页最少行数:"))

self.lines\_per\_page = QSpinBox()

self.lines\_per\_page.setRange(30, 100)

self.lines\_per\_page.setValue(50)

lines\_layout.addWidget(self.lines\_per\_page)

params\_layout.addLayout(lines\_layout)

params\_group.setLayout(params\_layout)

# 功能按钮区

buttons\_layout = QHBoxLayout()

self.select\_folder\_button = QPushButton("选择项目文件夹")

self.select\_folder\_button.clicked.connect(self.select\_project\_folder)

self.stat\_button = QPushButton("统计文件")

self.stat\_button.clicked.connect(self.stat\_files)

self.generate\_button = QPushButton("生成文档")

self.generate\_button.clicked.connect(self.generate\_document)

buttons\_layout.addWidget(self.select\_folder\_button)

buttons\_layout.addWidget(self.stat\_button)

buttons\_layout.addWidget(self.generate\_button)

# 进度条

self.progress\_bar = QProgressBar()

self.progress\_bar.setVisible(False)

# 组合右侧面板

right\_panel.addWidget(stats\_group)

right\_panel.addWidget(params\_group)

right\_panel.addLayout(buttons\_layout)

right\_panel.addWidget(self.progress\_bar)

# 组合主布局

main\_layout.addWidget(self.file\_tree, 2)

main\_layout.addLayout(right\_panel, 1)

# 设置中央窗口部件

central\_widget = QWidget()

central\_widget.setLayout(main\_layout)

self.setCentralWidget(central\_widget)

# 状态栏

self.status\_bar = QStatusBar()

self.setStatusBar(self.status\_bar)

self.status\_bar.showMessage("准备就绪")

def select\_project\_folder(self):

"""选择项目文件夹"""

folder = QFileDialog.getExistingDirectory(self, "选择项目文件夹")

if folder:

self.current\_folder = folder

self.status\_bar.showMessage(f"已选择文件夹: {folder}")

self.scan\_files(folder)

def select\_output\_path(self):

"""选择输出路径"""

folder = QFileDialog.getExistingDirectory(self, "选择输出路径")

if folder:

self.output\_path\_input.setText(folder)

def on\_item\_changed(self, item, column):

"""处理文件树项目状态变化"""

if column == 0: # 只处理第一列的勾选状态变化

file\_path = item.text(0)

is\_checked = item.checkState(0) == Qt.Checked

if file\_path in self.file\_data:

# 更新文件的选中状态

self.file\_data[file\_path]['selected'] = is\_checked

debug\_print(f"文件选择状态更新: {file\_path} -> {'选中' if is\_checked else '未选中'}")

self.update\_stats() # 更新统计信息

self.status\_bar.showMessage(f"{'选中' if is\_checked else '取消选中'}: {file\_path}")

def scan\_files(self, folder):

"""扫描文件夹中的源代码文件"""

self.status\_bar.showMessage("正在扫描文件...")

self.file\_tree.clear()

self.file\_data = {}

# 使用现有的文件扫描功能

all\_files = scan\_directory(folder)

debug\_print(f"扫描到的所有文件数: {len(all\_files)}")

# 过滤源代码文件

backend\_identifiers = {'.py', '.java', '.c', '.cpp', '.cs', '.go', '.rb', '.php', '.swift'}

frontend\_identifiers = {'.js', '.ts', '.jsx', '.tsx', '.html', '.css', '.vue', '.scss', '.less'}

code\_files = []

for file\_path in all\_files:

file\_ext = os.path.splitext(file\_path)[1].lower()

if file\_ext in backend\_identifiers or file\_ext in frontend\_identifiers:

code\_files.append(file\_path)

debug\_print(f"过滤后的源代码文件数: {len(code\_files)}")

# 暂时阻止信号，以避免每次添加项目时触发状态更新

self.file\_tree.blockSignals(True)

# 统计每个文件的行数

for file\_path in code\_files:

file\_ext = os.path.splitext(file\_path)[1].lower()

file\_type = "后端" if file\_ext in backend\_identifiers else "前端"

line\_count = self.count\_file\_lines(file\_path)

# 存储文件信息

self.file\_data[file\_path] = {

'type': file\_type,

'lines': line\_count,

'selected': True # 默认选中

}

# 添加到树控件

item = QTreeWidgetItem(self.file\_tree)

item.setText(0, file\_path)

item.setText(1, file\_type)

item.setText(2, str(line\_count))

item.setCheckState(0, Qt.Checked)

# 重新启用信号

self.file\_tree.blockSignals(False)

self.update\_stats()

self.status\_bar.showMessage(f"扫描完成，找到 {len(code\_files)} 个源代码文件")

def count\_file\_lines(self, file\_path):

"""计算文件的代码行数"""

try:

with open(file\_path, 'r', encoding='utf-8', errors='ignore') as f:

lines = f.readlines()

# 移除空行和纯注释行

code\_lines = [line for line in lines if line.strip() and not line.strip().startswith('#')]

return len(code\_lines)

except Exception as e:

debug\_print(f"计算文件行数时出错 {file\_path}: {e}")

return 0

def update\_stats(self):

"""更新统计信息"""

total\_files = len(self.file\_data)

selected\_files = sum(1 for info in self.file\_data.values() if info['selected'])

total\_lines = sum(info['lines'] for file\_path, info in self.file\_data.items() if info['selected'])

# 估算页数 (每页约50行)

estimated\_pages = total\_lines // self.lines\_per\_page.value() + 1

self.total\_files\_label.setText(f"总文件数: {total\_files}")

self.selected\_files\_label.setText(f"已选文件数: {selected\_files}")

self.total\_lines\_label.setText(f"总代码行数: {total\_lines}")

self.estimated\_pages\_label.setText(f"预计生成页数: {estimated\_pages}")

debug\_print(f"统计信息更新: 总文件数={total\_files}, 已选文件数={selected\_files}, 总代码行数={total\_lines}")

def stat\_files(self):

"""统计文件信息"""

if not hasattr(self, 'current\_folder'):

QMessageBox.warning(self, "警告", "请先选择项目文件夹")

return

self.scan\_files(self.current\_folder)

def sync\_file\_tree\_with\_data(self):

"""同步文件树与文件数据"""

debug\_print("同步文件树与文件数据")

# 遍历所有树项目

root = self.file\_tree.invisibleRootItem()

item\_count = root.childCount()

for i in range(item\_count):

item = root.child(i)

file\_path = item.text(0)

if file\_path in self.file\_data:

# 确保UI状态与数据状态一致

is\_selected = self.file\_data[file\_path]['selected']

item.setCheckState(0, Qt.Checked if is\_selected else Qt.Unchecked)

def generate\_document(self):

"""生成文档"""

if not hasattr(self, 'current\_folder') or not self.file\_data:

QMessageBox.warning(self, "警告", "请先选择项目文件夹并扫描文件")

return

# 确保文件树与数据同步

self.sync\_file\_tree\_with\_data()

# 获取参数

software\_name = self.name\_input.text()

software\_version = self.version\_input.text()

output\_name = self.output\_name\_input.text()

output\_path = os.path.join(self.output\_path\_input.text(), output\_name)

font\_name\_en = self.en\_font\_input.text()

font\_name\_cn = self.cn\_font\_input.text()

lines\_per\_page = self.lines\_per\_page.value()

# 获取选中的文件

selected\_files = {'backend': [], 'frontend': []}

unselected\_files = []

for file\_path, info in self.file\_data.items():

if info['selected']: # 确保只包含选中的文件

file\_type = 'backend' if info['type'] == "后端" else 'frontend'

selected\_files[file\_type].append(file\_path)

else:

unselected\_files.append(file\_path)

# 调试打印

debug\_print(f"选中的文件数量: 后端={len(selected\_files['backend'])}, 前端={len(selected\_files['frontend'])}")

debug\_print(f"未选中的文件数量: {len(unselected\_files)}")

if unselected\_files:

debug\_print("未选中的文件:")

for f in unselected\_files:

debug\_print(f" - {f}")

# 确认对话框，显示选中和未选中的文件数量

confirm\_msg = (f"即将生成文档，包含:\n"

f"- 后端文件: {len(selected\_files['backend'])}个\n"

f"- 前端文件: {len(selected\_files['frontend'])}个\n"

f"- 未包含文件: {len(unselected\_files)}个\n\n"

f"是否继续?")

confirm = QMessageBox.question(

self, "确认生成", confirm\_msg,

QMessageBox.Yes | QMessageBox.No

)

if confirm != QMessageBox.Yes:

self.status\_bar.showMessage("文档生成已取消")

return

# 显示进度条

self.progress\_bar.setVisible(True)

self.progress\_bar.setValue(0)

# 创建并启动文档生成线程

self.doc\_thread = DocumentGeneratorThread({

'classified\_files': selected\_files,

'output\_path': output\_path,

'software\_name': software\_name,

'software\_version': software\_version,

'font\_name\_en': font\_name\_en,

'font\_name\_cn': font\_name\_cn,

'lines\_per\_page': lines\_per\_page

})

self.doc\_thread.progress\_signal.connect(self.update\_progress)

self.doc\_thread.finished\_signal.connect(self.document\_generated)

self.doc\_thread.error\_signal.connect(self.document\_error)

self.doc\_thread.start()

self.status\_bar.showMessage("正在生成文档...")

def update\_progress(self, value):

"""更新进度条"""

self.progress\_bar.setValue(value)

def document\_generated(self, output\_file):

"""文档生成完成"""

self.progress\_bar.setVisible(False)

self.status\_bar.showMessage(f"文档生成完成: {output\_file}")

result = QMessageBox.question(

self, "生成完成",

f"文档已成功生成到: {output\_file}\n\n是否打开文档?",

QMessageBox.Yes | QMessageBox.No

)

if result == QMessageBox.Yes:

os.startfile(output\_file)

def document\_error(self, error\_msg):

"""文档生成错误"""

self.progress\_bar.setVisible(False)

self.status\_bar.showMessage("生成文档时出错")

QMessageBox.critical(self, "错误", f"生成文档时出错:\n{error\_msg}")

def main():

app = QApplication(sys.argv)

window = SoftCopyrightApp()

window.show()

sys.exit(app.exec\_())

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

main()

config\_manager.py

"""

配置管理模块

负责管理用户输入的配置参数，提供默认配置选项，验证配置的有效性。

"""

import os

import json

from typing import Dict, Any, Set, Optional

class ConfigManager:

"""配置管理类，处理用户配置和默认设置"""

def \_\_init\_\_(self, config\_file: Optional[str] = None):

"""

初始化配置管理器

Args:

config\_file: 配置文件路径，如果为None则使用默认配置

"""

# 默认配置

self.default\_config = {

"source\_dir": "",

"output\_path": "output/source\_code",

"output\_format": "docx",

"font\_name": "Courier New",

"font\_size": 10,

"lines\_per\_page": 50,

"file\_extensions": [".py", ".java", ".js", ".html", ".css", ".c", ".cpp", ".h", ".cs", ".go", ".php", ".rb"],

"exclude\_dirs": ["venv", "node\_modules", ".git", "\_\_pycache\_\_", "build", "dist"],

"backend\_identifiers": [".py", ".java", ".c", ".cpp", ".cs", ".go", ".rb", ".php"],

"frontend\_identifiers": [".js", ".ts", ".jsx", ".tsx", ".html", ".css", ".vue", ".scss", ".less"]

}

# 当前配置

self.config = self.default\_config.copy()

# 如果提供了配置文件，则加载它

if config\_file and os.path.exists(config\_file):

self.load\_config(config\_file)

def load\_config(self, config\_file: str) -> bool:

"""

从文件加载配置

Args:

config\_file: 配置文件路径

Returns:

加载是否成功

"""

try:

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

loaded\_config = json.load(f)

# 更新配置，保留默认值作为后备

for key, value in loaded\_config.items():

if key in self.config:

self.config[key] = value

return True

except Exception as e:

print(f"加载配置文件失败: {e}")

return False

def save\_config(self, config\_file: str) -> bool:

"""

保存配置到文件

Args:

config\_file: 配置文件路径

Returns:

保存是否成功

"""

try:

# 确保目录存在

os.makedirs(os.path.dirname(os.path.abspath(config\_file)), exist\_ok=True)

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

json.dump(self.config, f, ensure\_ascii=False, indent=4)

return True

except Exception as e:

print(f"保存配置文件失败: {e}")

return False

def update\_config(self, new\_config: Dict[str, Any]) -> None:

"""

更新配置

Args:

new\_config: 新的配置字典

"""

for key, value in new\_config.items():

if key in self.config:

self.config[key] = value

def get\_config(self) -> Dict[str, Any]:

"""

获取当前配置

Returns:

当前配置字典

"""

return self.config

def validate\_config(self) -> Dict[str, str]:

"""

验证配置的有效性

Returns:

错误信息字典，如果为空则表示配置有效

"""

errors = {}

# 检查源代码目录

if not self.config["source\_dir"]:

errors["source\_dir"] = "源代码目录不能为空"

elif not os.path.isdir(self.config["source\_dir"]):

errors["source\_dir"] = f"源代码目录 '{self.config['source\_dir']}' 不存在或不是一个目录"

# 检查输出格式

if self.config["output\_format"] not in ["docx", "pdf"]:

errors["output\_format"] = f"不支持的输出格式: {self.config['output\_format']}"

# 检查字体大小

if not (8 <= self.config["font\_size"] <= 14):

errors["font\_size"] = f"字体大小应在8-14磅之间，当前为: {self.config['font\_size']}"

# 检查每页行数

if self.config["lines\_per\_page"] < 10:

errors["lines\_per\_page"] = f"每页行数不应少于10行，当前为: {self.config['lines\_per\_page']}"

return errors

def get\_file\_extensions(self) -> Set[str]:

"""

获取文件扩展名集合

Returns:

文件扩展名集合

"""

return set(self.config["file\_extensions"])

def get\_backend\_identifiers(self) -> Set[str]:

"""

获取后端文件标识集合

Returns:

后端文件标识集合

"""

return set(self.config["backend\_identifiers"])

def get\_frontend\_identifiers(self) -> Set[str]:

"""

获取前端文件标识集合

Returns:

前端文件标识集合

"""

return set(self.config["frontend\_identifiers"])

def get\_exclude\_dirs(self) -> Set[str]:

"""

获取要排除的目录集合

Returns:

要排除的目录集合

"""

return set(self.config["exclude\_dirs"])

file\_utils.py

"""

文件操作工具模块

提供文件读写和目录操作的辅助功能。

"""

import os

import shutil

from typing import List, Set, Optional

def ensure\_dir\_exists(directory: str) -> bool:

"""

确保目录存在，如果不存在则创建

Args:

directory: 目录路径

Returns:

是否成功创建或目录已存在

"""

try:

os.makedirs(directory, exist\_ok=True)

return True

except Exception as e:

print(f"创建目录失败: {e}")

return False

def list\_files(directory: str, recursive: bool = True) -> List[str]:

"""

列出目录中的所有文件

Args:

directory: 目录路径

recursive: 是否递归搜索子目录

Returns:

文件路径列表

"""

files = []

if recursive:

for root, \_, filenames in os.walk(directory):

for filename in filenames:

files.append(os.path.join(root, filename))

else:

for item in os.listdir(directory):

item\_path = os.path.join(directory, item)

if os.path.isfile(item\_path):

files.append(item\_path)

return files

def filter\_files\_by\_extension(files: List[str], extensions: Set[str]) -> List[str]:

"""

按文件扩展名筛选文件

Args:

files: 文件路径列表

extensions: 文件扩展名集合

Returns:

筛选后的文件路径列表

"""

if not extensions:

return files

return [f for f in files if os.path.splitext(f)[1].lower() in extensions]

def exclude\_directories(files: List[str], exclude\_dirs: Set[str]) -> List[str]:

"""

排除指定目录中的文件

Args:

files: 文件路径列表

exclude\_dirs: 要排除的目录名集合

Returns:

排除后的文件路径列表

"""

if not exclude\_dirs:

return files

result = []

for file\_path in files:

# 检查文件路径中是否包含要排除的目录

if not any(f"/{exclude\_dir}/" in file\_path.replace("\\", "/") for exclude\_dir in exclude\_dirs):

result.append(file\_path)

return result

def get\_relative\_path(file\_path: str, base\_dir: str) -> str:

"""

获取相对于基础目录的相对路径

Args:

file\_path: 文件的绝对路径

base\_dir: 基础目录

Returns:

相对路径

"""

try:

return os.path.relpath(file\_path, base\_dir)

except:

return file\_path

def copy\_file(src: str, dst: str) -> bool:

"""

复制文件

Args:

src: 源文件路径

dst: 目标文件路径

Returns:

是否成功复制

"""

try:

# 确保目标目录存在

ensure\_dir\_exists(os.path.dirname(dst))

# 复制文件

shutil.copy2(src, dst)

return True

except Exception as e:

print(f"复制文件失败: {e}")

return False

def read\_file(file\_path: str, encoding: str = 'utf-8') -> Optional[str]:

"""

读取文件内容

Args:

file\_path: 文件路径

encoding: 文件编码

Returns:

文件内容，如果读取失败则返回None

"""

try:

with open(file\_path, 'r', encoding=encoding, errors='ignore') as f:

return f.read()

except Exception as e:

print(f"读取文件失败: {e}")

return None

def write\_file(file\_path: str, content: str, encoding: str = 'utf-8') -> bool:

"""

写入文件内容

Args:

file\_path: 文件路径

content: 要写入的内容

encoding: 文件编码

Returns:

是否成功写入

"""

try:

# 确保目录存在

ensure\_dir\_exists(os.path.dirname(file\_path))

with open(file\_path, 'w', encoding=encoding) as f:

f.write(content)

return True

except Exception as e:

print(f"写入文件失败: {e}")

return False

text\_utils.py

"""

文本处理工具模块

提供文本格式化和处理的辅助功能。

"""

import re

from typing import List, Optional

def remove\_empty\_lines(text: str) -> str:

"""

移除文本中的空行

Args:

text: 输入文本

Returns:

移除空行后的文本

"""

# 移除只包含空白字符的行

return '\n'.join(line for line in text.splitlines() if line.strip())

def remove\_empty\_lines\_from\_list(lines: List[str]) -> List[str]:

"""

移除行列表中的空行

Args:

lines: 行列表

Returns:

移除空行后的行列表

"""

return [line for line in lines if line.strip()]

def truncate\_text(text: str, max\_length: int, suffix: str = "...") -> str:

"""

截断文本到指定长度

Args:

text: 输入文本

max\_length: 最大长度

suffix: 截断后添加的后缀

Returns:

截断后的文本

"""

if len(text) <= max\_length:

return text

return text[:max\_length - len(suffix)] + suffix

def count\_non\_empty\_lines(text: str) -> int:

"""

计算文本中的非空行数

Args:

text: 输入文本

Returns:

非空行数

"""

return len([line for line in text.splitlines() if line.strip()])

def format\_file\_header(file\_path: str, line\_length: int = 80) -> str:

"""

格式化文件头部标识

Args:

file\_path: 文件路径

line\_length: 分隔线长度

Returns:

格式化后的文件头部标识

"""

separator = "=" \* line\_length

return f"{separator}\n文件: {file\_path}\n{separator}"

def split\_text\_into\_chunks(text: str, chunk\_size: int) -> List[str]:

"""

将文本分割成固定大小的块

Args:

text: 输入文本

chunk\_size: 块大小（字符数）

Returns:

文本块列表

"""

return [text[i:i + chunk\_size] for i in range(0, len(text), chunk\_size)]

def split\_lines\_into\_pages(lines: List[str], lines\_per\_page: int) -> List[List[str]]:

"""

将行列表分割成页面

Args:

lines: 行列表

lines\_per\_page: 每页的行数

Returns:

页面列表，每个页面是一个行列表

"""

pages = []

total\_lines = len(lines)

for i in range(0, total\_lines, lines\_per\_page):

end = min(i + lines\_per\_page, total\_lines)

pages.append(lines[i:end])

return pages

def sanitize\_filename(filename: str) -> str:

"""

清理文件名，移除不允许的字符

Args:

filename: 原始文件名

Returns:

清理后的文件名

"""

# 移除不允许的字符

sanitized = re.sub(r'[\\/\*?:"<>|]', '\_', filename)

# 移除前导和尾随空格

sanitized = sanitized.strip()

# 如果文件名为空，使用默认名称

if not sanitized:

sanitized = "unnamed\_file"

return sanitized

def extract\_extension(file\_path: str) -> Optional[str]:

"""

提取文件扩展名

Args:

file\_path: 文件路径

Returns:

文件扩展名（包括点），如果没有扩展名则返回None

"""

match = re.search(r'\.([^.\\/:\*?"<>|]+)$', file\_path)

if match:

return f".{match.group(1)}"

return None

file\_filter.py

#!/usr/bin/env python

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

import os

import re

import sys

import glob

import argparse

from pathlib import Path

from typing import List, Dict, Set, Tuple

class FileFilter:

"""软著文档文件筛选工具"""

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

# 通用排除规则

self.common\_exclude\_patterns = [

# 版本控制

r'\.git/', r'\.svn/', r'\.hg/',

r'\.gitignore', r'\.gitattributes',

# 编辑器/IDE

r'\.vscode/', r'\.idea/', r'\.eclipse/',

r'.\*\.sublime-project', r'.\*\.sublime-workspace',

# 操作系统

r'\.DS\_Store', r'Thumbs\.db',

# 备份和临时

r'.\*~', r'.\*\.bak', r'.\*\.swp', r'.\*\.tmp',

r'.\*\.old', r'.\*\.backup',

# 日志和缓存

r'.\*\.log', r'\.cache/',

# 敏感信息

r'.\*\.env', r'.\*\.key', r'.\*\.pem'

]

# Python项目排除规则

self.python\_exclude\_patterns = [

# 必须排除

r'\_\_pycache\_\_/',

r'.\*\.pyc', r'.\*\.pyo', r'.\*\.pyd',

r'.\*\.so', r'.\*\.dll', r'.\*\.dylib',

r'dist/', r'build/',

r'.\*\.egg-info/', r'.\*\.egg',

r'\.pytest\_cache/', r'\.coverage',

r'venv/', r'env/', r'\.venv/', r'\.env/',

]

# 前端项目排除规则

self.frontend\_exclude\_patterns = [

# 必须排除

r'node\_modules/',

r'dist/', r'build/', r'out/',

r'.\*\.min\.js', r'.\*\.min\.css',

r'\.cache/', r'coverage/',

]

# 后端项目排除规则

self.backend\_exclude\_patterns = [

# 必须排除

r'.\*\.class', r'.\*\.dll',

r'target/', r'bin/', r'obj/',

r'logs/', r'temp/', r'tmp/',

r'.\*\.bak', r'.\*\.dump',

]

# 可选排除规则

self.optional\_exclude\_patterns = {

'python': [

('\_\_init\_\_.py文件', r'\_\_init\_\_\.py'),

('安装配置文件', r'setup\.py|setup\.cfg'),

('依赖管理文件', r'requirements\.txt|Pipfile|Pipfile\.lock'),

('文档和示例目录', r'docs/|examples/')

],

'frontend': [

('依赖配置文件', r'package\.json|package-lock\.json|yarn\.lock'),

('代码风格配置文件', r'\.eslintrc|\.prettierrc'),

('构建工具配置', r'webpack\.config\.js|babel\.config\.js'),

('源码映射文件', r'.\*\.map'),

('静态资源目录', r'public/|static/'),

('环境配置文件', r'\.browserslistrc|\.nvmrc'),

('测试文件', r'.\*\.test\.js|.\*\.spec\.js')

],

'backend': [

('配置文件', r'application\.properties|web\.config'),

('依赖管理文件', r'pom\.xml|build\.gradle'),

('IDE配置目录', r'\.settings/'),

('容器配置文件', r'Dockerfile|docker-compose\.yml'),

('数据库脚本', r'.\*\.sql'),

('测试目录', r'test/|tests/')

],

'common': [

('项目文档', r'README\.md|CHANGELOG\.md'),

('许可证文件', r'LICENSE'),

('CI/CD配置文件', r'\.travis\.yml|\.github/workflows/')

]

}

# 存储所有可选规则的映射，用于显示和管理

self.all\_optional\_rules = {}

self.active\_optional\_rules = set() # 使用集合存储当前激活的可选规则

# 基础排除规则（必须排除的）

self.base\_exclude\_patterns = []

self.active\_exclude\_patterns = []

self.project\_type = None

self.source\_dir = None

def show\_menu(self):

"""显示主菜单"""

print("\n========= 软著文档文件筛选工具 =========")

print("1. 选择项目类型")

print("2. 配置排除规则")

print("3. 查看当前排除规则")

print("4. 扫描并显示筛选结果")

print("5. 导出文件列表")

print("6. 查看帮助")

print("0. 退出")

choice = input("\n请选择操作 [0-6]: ")

return choice

def select\_project\_type(self):

"""选择项目类型"""

print("\n========= 选择项目类型 =========")

print("1. Python项目")

print("2. 前端项目")

print("3. 后端项目")

print("4. 全栈项目")

print("0. 返回")

choice = input("\n请选择项目类型 [0-4]: ")

# 重置规则状态

self.base\_exclude\_patterns = []

self.active\_optional\_rules = set()

self.all\_optional\_rules = {}

if choice == '1':

self.project\_type = 'python'

self.base\_exclude\_patterns = self.common\_exclude\_patterns + self.python\_exclude\_patterns

print("已选择: Python项目")

elif choice == '2':

self.project\_type = 'frontend'

self.base\_exclude\_patterns = self.common\_exclude\_patterns + self.frontend\_exclude\_patterns

print("已选择: 前端项目")

elif choice == '3':

self.project\_type = 'backend'

self.base\_exclude\_patterns = self.common\_exclude\_patterns + self.backend\_exclude\_patterns

print("已选择: 后端项目")

elif choice == '4':

self.project\_type = 'fullstack'

self.base\_exclude\_patterns = (self.common\_exclude\_patterns +

self.python\_exclude\_patterns +

self.frontend\_exclude\_patterns +

self.backend\_exclude\_patterns)

print("已选择: 全栈项目")

# 更新活动排除规则

self.update\_active\_exclude\_patterns()

# 选择项目目录

if choice in ['1', '2', '3', '4']:

self.select\_source\_directory()

def update\_active\_exclude\_patterns(self):

"""更新活动排除规则列表"""

# 首先添加基础规则（必须排除的）

self.active\_exclude\_patterns = self.base\_exclude\_patterns.copy()

# 然后添加当前激活的可选规则

for rule\_id in self.active\_optional\_rules:

if rule\_id in self.all\_optional\_rules:

pattern = self.all\_optional\_rules[rule\_id][1]

if pattern not in self.active\_exclude\_patterns:

self.active\_exclude\_patterns.append(pattern)

def select\_source\_directory(self):

"""选择源代码目录"""

default\_dir = os.getcwd()

dir\_input = input(f"\n请输入源代码目录路径 [默认: {default\_dir}]: ")

self.source\_dir = dir\_input if dir\_input.strip() else default\_dir

if not os.path.isdir(self.source\_dir):

print(f"错误: 目录 '{self.source\_dir}' 不存在")

self.source\_dir = None

return

print(f"已选择源代码目录: {self.source\_dir}")

def configure\_exclude\_rules(self):

"""配置排除规则"""

if not self.project\_type:

print("请先选择项目类型")

return

while True:

print("\n========= 配置可选排除规则 =========")

print("当前激活的可选规则: ", end="")

if not self.active\_optional\_rules:

print("无")

else:

active\_rules = [self.all\_optional\_rules[rule\_id][0] for rule\_id in self.active\_optional\_rules if rule\_id in self.all\_optional\_rules]

print(", ".join(active\_rules))

print("\n选择操作:")

print("1. 添加排除规则")

print("2. 移除排除规则")

print("0. 返回主菜单")

op\_choice = input("\n请选择操作 [0-2]: ")

if op\_choice == '0':

return

elif op\_choice == '1':

self.add\_exclude\_rules()

elif op\_choice == '2':

self.remove\_exclude\_rules()

else:

print("无效选择，请重试")

def add\_exclude\_rules(self):

"""添加排除规则"""

# 显示可选排除规则

categories = []

if self.project\_type in ['python', 'fullstack']:

categories.append('python')

if self.project\_type in ['frontend', 'fullstack']:

categories.append('frontend')

if self.project\_type in ['backend', 'fullstack']:

categories.append('backend')

categories.append('common')

self.all\_optional\_rules = {}

option\_num = 1

print("\n可添加的排除规则:")

for category in categories:

print(f"\n--- {category.capitalize()} 项目可选排除 ---")

for name, pattern in self.optional\_exclude\_patterns[category]:

rule\_id = f"{category}\_{option\_num}"

status = "√" if rule\_id in self.active\_optional\_rules else " "

print(f"{option\_num}. [{status}] {name} ({pattern})")

self.all\_optional\_rules[rule\_id] = (name, pattern)

option\_num += 1

choice = input("\n请选择要添加的规则编号 (多个用逗号分隔，0返回): ")

if choice == '0':

return

try:

selected\_indices = [int(idx.strip()) for idx in choice.split(',') if idx.strip()]

for idx in selected\_indices:

if 1 <= idx < option\_num:

# 找到对应的规则ID

for rule\_id, (name, pattern) in self.all\_optional\_rules.items():

if rule\_id.split('\_')[1] == str(idx):

self.active\_optional\_rules.add(rule\_id)

print(f"已添加规则: {name}")

break

else:

print(f"无效的选项: {idx}")

# 更新活动排除规则

self.update\_active\_exclude\_patterns()

except ValueError:

print("输入无效，请输入数字")

def remove\_exclude\_rules(self):

"""移除排除规则"""

if not self.active\_optional\_rules:

print("当前没有激活的可选规则")

return

print("\n当前激活的排除规则:")

active\_rules = [(rule\_id, self.all\_optional\_rules[rule\_id][0])

for rule\_id in self.active\_optional\_rules

if rule\_id in self.all\_optional\_rules]

for i, (rule\_id, name) in enumerate(active\_rules, 1):

print(f"{i}. {name}")

choice = input("\n请选择要移除的规则编号 (多个用逗号分隔，0返回): ")

if choice == '0':

return

try:

selected\_indices = [int(idx.strip()) for idx in choice.split(',') if idx.strip()]

for idx in selected\_indices:

if 1 <= idx <= len(active\_rules):

rule\_id = active\_rules[idx-1][0]

name = active\_rules[idx-1][1]

self.active\_optional\_rules.remove(rule\_id)

print(f"已移除规则: {name}")

else:

print(f"无效的选项: {idx}")

# 更新活动排除规则

self.update\_active\_exclude\_patterns()

except ValueError:

print("输入无效，请输入数字")

def show\_current\_rules(self):

"""显示当前排除规则"""

if not self.project\_type:

print("请先选择项目类型")

return

print("\n========= 当前排除规则 =========")

print(f"项目类型: {self.project\_type}")

print("\n基础排除规则 (必须排除):")

for i, pattern in enumerate(self.base\_exclude\_patterns, 1):

print(f"{i}. {pattern}")

print("\n可选排除规则 (当前激活):")

if not self.active\_optional\_rules:

print("无")

else:

active\_rules = [(rule\_id, self.all\_optional\_rules[rule\_id][0], self.all\_optional\_rules[rule\_id][1])

for rule\_id in self.active\_optional\_rules

if rule\_id in self.all\_optional\_rules]

for i, (\_, name, pattern) in enumerate(active\_rules, 1):

print(f"{i}. {name} ({pattern})")

input("\n按任意键返回主菜单...")

def scan\_files(self):

"""扫描文件并应用过滤规则"""

if not self.project\_type or not self.source\_dir:

print("请先选择项目类型和源代码目录")

return [], []

all\_files = []

for root, \_, files in os.walk(self.source\_dir):

for file in files:

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

rel\_path = os.path.relpath(file\_path, self.source\_dir)

all\_files.append(rel\_path.replace('\\', '/')) # 统一使用正斜杠

included\_files = []

excluded\_files = []

for file\_path in all\_files:

excluded = False

matching\_pattern = None

for pattern in self.active\_exclude\_patterns:

if re.search(pattern, file\_path):

excluded = True

matching\_pattern = pattern

break

if excluded:

excluded\_files.append((file\_path, matching\_pattern))

else:

included\_files.append(file\_path)

return included\_files, excluded\_files

def show\_scan\_results(self):

"""显示扫描结果"""

if not self.project\_type or not self.source\_dir:

print("请先选择项目类型和源代码目录")

return

included\_files, excluded\_files = self.scan\_files()

print("\n========= 扫描结果 =========")

print(f"源代码目录: {self.source\_dir}")

print(f"项目类型: {self.project\_type}")

print(f"总文件数: {len(included\_files) + len(excluded\_files)}")

print(f"包含文件数: {len(included\_files)}")

print(f"排除文件数: {len(excluded\_files)}")

show\_details = input("\n是否显示详细文件列表? (y/n): ").lower() == 'y'

if show\_details:

print("\n--- 包含的文件 ---")

for file in sorted(included\_files):

print(file)

print("\n--- 排除的文件 ---")

for file, pattern in sorted(excluded\_files):

print(f"{file} (匹配规则: {pattern})")

def export\_file\_list(self):

"""导出文件列表"""

if not self.project\_type or not self.source\_dir:

print("请先选择项目类型和源代码目录")

return

included\_files, excluded\_files = self.scan\_files()

export\_dir = os.path.join(os.getcwd(), "export")

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

timestamp = os.path.basename(self.source\_dir)

included\_file\_path = os.path.join(export\_dir, f"{timestamp}\_included\_files.txt")

excluded\_file\_path = os.path.join(export\_dir, f"{timestamp}\_excluded\_files.txt")

with open(included\_file\_path, 'w', encoding='utf-8') as f:

f.write(f"# 包含的文件列表 - {self.project\_type} 项目\n")

f.write(f"# 源代码目录: {self.source\_dir}\n")

f.write(f"# 总计: {len(included\_files)} 文件\n\n")

for file in sorted(included\_files):

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

with open(excluded\_file\_path, 'w', encoding='utf-8') as f:

f.write(f"# 排除的文件列表 - {self.project\_type} 项目\n")

f.write(f"# 源代码目录: {self.source\_dir}\n")

f.write(f"# 总计: {len(excluded\_files)} 文件\n\n")

for file, pattern in sorted(excluded\_files):

f.write(f"{file} (匹配规则: {pattern})\n")

print(f"\n文件列表已导出到:")

print(f"- {included\_file\_path}")

print(f"- {excluded\_file\_path}")

def show\_help(self):

"""显示帮助信息"""

print("\n========= 软著文档文件筛选工具帮助 =========")

print("本工具用于帮助筛选软件著作权登记文档中应包含的源代码文件。")

print("\n使用步骤:")

print("1. 选择项目类型（Python、前端、后端或全栈项目）")

print("2. 配置可选的排除规则")

print("3. 查看当前排除规则")

print("4. 扫描并查看筛选结果")

print("5. 导出文件列表供后续处理")

print("\n项目类型说明:")

print("- Python项目: 主要包含Python源代码的项目")

print("- 前端项目: 主要包含HTML、CSS、JavaScript等前端技术的项目")

print("- 后端项目: 主要包含Java、C#等后端技术的项目")

print("- 全栈项目: 同时包含前端和后端代码的综合项目")

print("\n排除规则说明:")

print("- 必须排除: 默认排除的文件，通常是编译产物、临时文件等")

print("- 可选排除: 用户可以选择是否排除的文件，如配置文件、测试文件等")

print("- 可以添加和移除可选排除规则")

print("\n软著文档最佳实践:")

print("- 优先包含自己编写的核心业务逻辑代码")

print("- 排除第三方库、框架和工具生成的代码")

print("- 确保文档页数在合理范围内（通常30-50页为宜）")

print("- 保持文件的相对路径结构，便于理解代码组织")

input("\n按任意键返回主菜单...")

def run(self):

"""运行主程序"""

while True:

choice = self.show\_menu()

if choice == '0':

print("感谢使用，再见！")

break

elif choice == '1':

self.select\_project\_type()

elif choice == '2':

self.configure\_exclude\_rules()

elif choice == '3':

self.show\_current\_rules()

elif choice == '4':

self.show\_scan\_results()

elif choice == '5':

self.export\_file\_list()

elif choice == '6':

self.show\_help()

else:

print("无效选择，请重试")

def main():

"""主函数"""

print("软著文档文件筛选工具 v1.1")

print("用于帮助筛选软件著作权登记文档中应包含的源代码文件")

file\_filter = FileFilter()

file\_filter.run()

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

main()

run\_simple.py

"""

运行软件著作权生成工具的简单脚本 - 不依赖click库

增强版：格式优化，确保每页至少50行代码，添加页眉和页码（右上角）

支持命令行参数，可以指定源代码目录、项目名称、版本号和输出路径

"""

import os

import sys

import argparse

from sofcrtpro.file\_scanner import scan\_directory, classify\_files

from sofcrtpro.code\_processor import process\_code\_files

from sofcrtpro.document\_generator import generate\_document

def parse\_arguments():

"""

解析命令行参数

Returns:

解析后的参数对象

"""

parser = argparse.ArgumentParser(description='生成软件著作权源代码文件')

parser.add\_argument('-s', '--source', dest='source\_dir', default='.',

help='源代码目录路径 (默认: 当前目录)')

parser.add\_argument('-n', '--name', dest='software\_name', default='SoftCopyrightPro',

help='软件名称 (默认: SoftCopyrightPro)')

parser.add\_argument('-v', '--version', dest='software\_version', default='v0.1.0',

help='软件版本号 (默认: v0.1.0)')

parser.add\_argument('-o', '--output', dest='output\_path',

default='./output/SoftCopyrightPro-source-aligned',

help='输出文件路径 (默认: ./output/SoftCopyrightPro-source-aligned)')

parser.add\_argument('-f', '--format', dest='output\_format', default='docx',

choices=['docx', 'pdf'], help='输出文件格式 (默认: docx)')

parser.add\_argument('--english-font', dest='font\_name\_en', default='Courier New',

help='英文字体 (默认: Courier New)')

parser.add\_argument('--chinese-font', dest='font\_name\_cn', default='SimSun',

help='中文字体 (默认: SimSun)')

parser.add\_argument('--font-size', dest='font\_size', type=float, default=10.5,

help='字体大小 (默认: 10.5pt)')

parser.add\_argument('--lines-per-page', dest='lines\_per\_page', type=int, default=50,

help='每页最少行数 (默认: 50)')

parser.add\_argument('--list-only', action='store\_true',

help='仅列出符合条件的源代码文件，不生成文档')

parser.add\_argument('--exclude-patterns', dest='exclude\_patterns', nargs='+',

default=['test\_', 'tests', 'debug', 'generated', 'auto\_gen', 'temp', 'tmp'],

help='排除的文件名模式 (默认: test\_ tests debug generated auto\_gen temp tmp)')

parser.add\_argument('--exclude-dirs', dest='exclude\_dirs', nargs='+',

default=['.git', 'sofcrtpro.egg-info', '\_\_pycache\_\_', '.vscode', '.idea',

'venv', 'env', 'node\_modules', 'tests', 'test', 'debug', 'temp', 'tmp'],

help='排除的目录名 (默认: .git sofcrtpro.egg-info \_\_pycache\_\_ .vscode .idea venv env node\_modules tests test debug temp tmp)')

parser.add\_argument('--include-dir', dest='include\_dirs', nargs='+',

default=[],

help='指定要包含的目录，这些目录下的所有源代码文件都将被包含，即使它们匹配排除模式')

return parser.parse\_args()

def should\_exclude\_file(file\_path, exclude\_patterns, exclude\_dirs, include\_dirs):

"""

判断文件是否应该被排除

Args:

file\_path: 文件路径

exclude\_patterns: 排除的文件名模式

exclude\_dirs: 排除的目录名

include\_dirs: 指定要包含的目录

Returns:

是否应该排除

"""

# 检查文件是否在包含目录中，如果是则不排除

for include\_dir in include\_dirs:

if include\_dir in file\_path:

return False

# 检查文件是否在排除目录中

for excluded\_dir in exclude\_dirs:

if excluded\_dir in file\_path.split(os.sep):

return True

# 检查文件名是否匹配排除模式

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

for pattern in exclude\_patterns:

if pattern in file\_name:

return True

return False

def count\_file\_lines(file\_path):

"""

计算文件的代码行数（不包括空行和注释行）

Args:

file\_path: 文件路径

Returns:

代码行数

"""

try:

with open(file\_path, 'r', encoding='utf-8', errors='ignore') as f:

lines = f.readlines()

# 移除空行和纯注释行

code\_lines = [line for line in lines if line.strip() and not line.strip().startswith('#')]

return len(code\_lines)

except Exception as e:

print(f"计算文件行数时出错 {file\_path}: {e}")

return 0

def run\_simple():

"""

运行软件著作权生成工具 - 格式优化版

"""

# 解析命令行参数

args = parse\_arguments()

# 配置参数

source\_dir = args.source\_dir

output\_path = args.output\_path

output\_format = args.output\_format

font\_name\_en = args.font\_name\_en

font\_name\_cn = args.font\_name\_cn

font\_size = args.font\_size

lines\_per\_page = args.lines\_per\_page

software\_name = args.software\_name

software\_version = args.software\_version

exclude\_dirs = args.exclude\_dirs

exclude\_patterns = args.exclude\_patterns

include\_dirs = args.include\_dirs

list\_only = args.list\_only

# 确保输出目录存在

if not list\_only:

os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

print(f"扫描目录: {source\_dir}")

# 扫描文件

all\_files = scan\_directory(source\_dir)

# 排除指定目录和文件

filtered\_files = []

for file\_path in all\_files:

if not should\_exclude\_file(file\_path, exclude\_patterns, exclude\_dirs, include\_dirs):

filtered\_files.append(file\_path)

# 默认的文件扩展名

backend\_identifiers = {'.py', '.java', '.c', '.cpp', '.cs', '.go', '.rb', '.php', '.swift'}

frontend\_identifiers = {'.js', '.ts', '.jsx', '.tsx', '.html', '.css', '.vue', '.scss', '.less'}

# 筛选文件扩展名

code\_files = []

for file\_path in filtered\_files:

file\_ext = os.path.splitext(file\_path)[1].lower()

if file\_ext in backend\_identifiers or file\_ext in frontend\_identifiers:

code\_files.append(file\_path)

print(f"找到 {len(code\_files)} 个符合条件的源代码文件")

# 分类文件

classified\_files = {'backend': [], 'frontend': []}

for file\_path in code\_files:

file\_ext = os.path.splitext(file\_path)[1].lower()

if file\_ext in backend\_identifiers:

classified\_files['backend'].append(file\_path)

elif file\_ext in frontend\_identifiers:

classified\_files['frontend'].append(file\_path)

print(f"后端文件: {len(classified\_files['backend'])} 个")

print(f"前端文件: {len(classified\_files['frontend'])} 个")

# 计算每个文件的行数

file\_line\_counts = {}

total\_lines = 0

for category in ['backend', 'frontend']:

for file\_path in classified\_files[category]:

line\_count = count\_file\_lines(file\_path)

file\_line\_counts[file\_path] = line\_count

total\_lines += line\_count

# 如果仅列出文件，则打印文件列表并退出

if list\_only:

print("\n符合条件的源代码文件列表:")

print("\n后端文件:")

for i, file\_path in enumerate(sorted(classified\_files['backend']), 1):

line\_count = file\_line\_counts[file\_path]

print(f"{i}. {file\_path} - {line\_count}行")

print("\n前端文件:")

for i, file\_path in enumerate(sorted(classified\_files['frontend']), 1):

line\_count = file\_line\_counts[file\_path]

print(f"{i}. {file\_path} - {line\_count}行")

# 添加汇总信息区域，使用分隔线使其更加醒目

print("\n" + "="\*50)

print("文件列表汇总信息:")

print(f"总文件数: {len(code\_files)}个文件")

print(f"总代码行数: {total\_lines}行")

print("="\*50)

return

# 处理代码文件 - 会自动按重要性排序，去掉空行，简化文件名标识

merged\_content, pages, processed\_files = process\_code\_files(classified\_files, lines\_per\_page)

print(f"合并后的代码总行数: {len(merged\_content)}")

print(f"分页后的页数: {len(pages)}")

# 生成文档

output\_file, stats = generate\_document(

content=merged\_content,

output\_path=output\_path,

file\_list=processed\_files,

output\_format=output\_format,

software\_name=software\_name,

software\_version=software\_version,

font\_name\_en=font\_name\_en,

font\_name\_cn=font\_name\_cn,

font\_size=font\_size

)

print(f"文档已生成: {output\_file}")

print(f"文档格式:")

print(f" - 字号: {font\_size}pt")

print(f" - 行间距: 10.5pt")

print(f" - 段前间距: 0pt")

print(f" - 段后间距: 2.3pt")

print(f" - 页眉: {software\_name} {software\_version} (带横线)")

print(f" - 页码: 右上角")

print(f" - 英文字体: {font\_name\_en}（等宽）")

print(f" - 中文字体: {font\_name\_cn}（等宽）")

print(f" - 页眉字体: 与正文一致（等宽字体）")

print(f" - 页眉布局: 单行对齐，分割线紧贴页眉")

print(f" - 每页至少{lines\_per\_page}行代码")

print(f" - 简化文件名标识")

print(f" - 去掉所有空行")

print(f" - 按重要性排序(后端优先，前端其次)")

# 显示详细统计信息

print("\n详细统计信息:")

print(f"总代码行数: {stats['total\_lines']}")

print(f"总页数: {stats['total\_pages']}")

print(f"处理的文件数量: {len(processed\_files)}")

# 显示每个处理文件的详细信息

print("\n处理的文件列表(按顺序):")

for i, file\_path in enumerate(processed\_files, 1):

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

line\_count = file\_line\_counts.get(file\_path, "未知")

print(f"{i}. {file\_path} - {line\_count}行")

# 添加汇总信息区域，使用分隔线使其更加醒目

print("\n" + "="\*50)

print("文档生成汇总信息:")

print(f"总文件数: {len(processed\_files)}个文件")

print(f"总代码行数: {total\_lines}行")

print(f"总页数: {stats['total\_pages']}页")

print("="\*50)

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

run\_simple()

run\_sofcrtpro.py

"""

运行软件著作权生成工具的简单脚本

"""

import os

import sys

from sofcrtpro.cli import main

from sofcrtpro.config\_manager import ConfigManager

from sofcrtpro.file\_scanner import scan\_directory, classify\_files

from sofcrtpro.code\_processor import process\_code\_files

from sofcrtpro.document\_generator import generate\_document

from sofcrtpro.utils.file\_utils import exclude\_directories, filter\_files\_by\_extension

def run\_sofcrtpro():

"""

运行软件著作权生成工具

"""

# 配置参数

source\_dir = "."

output\_path = "./output/SoftCopyrightPro-source"

output\_format = "docx"

font\_name = "Courier New"

font\_size = 10

lines\_per\_page = 50

exclude\_dirs = [".git", "sofcrtpro.egg-info", "\_\_pycache\_\_"]

# 确保输出目录存在

os.makedirs(os.path.dirname(output\_path), exist\_ok=True)

print(f"扫描目录: {source\_dir}")

# 初始化配置管理器

config\_manager = ConfigManager()

config = {

'source\_dir': source\_dir,

'output\_path': output\_path,

'output\_format': output\_format,

'font\_name': font\_name,

'font\_size': font\_size,

'lines\_per\_page': lines\_per\_page,

'exclude\_dirs': exclude\_dirs,

'file\_extensions': None # 使用默认值

}

config\_manager.update\_config(config)

# 扫描文件

all\_files = scan\_directory(source\_dir)

# 排除指定目录

all\_files = exclude\_directories(all\_files, exclude\_dirs)

# 按扩展名筛选

filtered\_files = filter\_files\_by\_extension(all\_files, config\_manager.get\_file\_extensions())

print(f"找到 {len(filtered\_files)} 个符合条件的源代码文件")

# 分类文件

classified\_files = classify\_files(

filtered\_files,

config\_manager.get\_backend\_identifiers(),

config\_manager.get\_frontend\_identifiers()

)

print(f"后端文件: {len(classified\_files['backend'])} 个")

print(f"前端文件: {len(classified\_files['frontend'])} 个")

# 处理代码文件

merged\_content, pages = process\_code\_files(classified\_files, lines\_per\_page)

print(f"合并后的代码总行数: {len(merged\_content)}")

print(f"分页后的页数: {len(pages)}")

# 生成文档

output\_file = generate\_document(

content=merged\_content,

output\_path=output\_path,

output\_format=output\_format,

font\_name=font\_name,

font\_size=font\_size

)

print(f"文档已生成: {output\_file}")

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

run\_sofcrtpro()

setup.py

"""

SoftCopyrightPro 安装脚本

"""

from setuptools import setup, find\_packages

import os

# 读取README.md作为长描述

with open("README.md", "r", encoding="utf-8") as fh:

long\_description = fh.read()

# 读取requirements.txt中的依赖

with open("requirements.txt", "r", encoding="utf-8") as f:

requirements = f.read().splitlines()

setup(

name="sofcrtpro",

version="0.1.0",

author="SoftCopyrightPro Team",

author\_email="your.email@example.com",

description="软件著作权材料生成工具",

long\_description=long\_description,

long\_description\_content\_type="text/markdown",

url="https://github.com/yourusername/SoftCopyrightPro",

packages=find\_packages(),

classifiers=[

"Programming Language :: Python :: 3",

"Programming Language :: Python :: 3.10",

"License :: OSI Approved :: MIT License",

"Operating System :: OS Independent",

],

python\_requires=">=3.10",

install\_requires=requirements,

entry\_points={

"console\_scripts": [

"sofcrtpro=sofcrtpro.cli:main",

],

},

)

sofcrtpro\_cli.py

#!/usr/bin/env python

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

import os

import sys

import argparse

import subprocess

from pathlib import Path

def parse\_args():

"""解析命令行参数"""

parser = argparse.ArgumentParser(

description='软著文档生成工具命令行接口',

formatter\_class=argparse.RawTextHelpFormatter

)

parser.add\_argument('-i', '--input', type=str, required=True,

help='包含文件列表的文本文件路径')

parser.add\_argument('-n', '--name', type=str, required=True,

help='软件名称')

parser.add\_argument('-v', '--version', type=str, default='V1.0',

help='软件版本号 (默认: V1.0)')

parser.add\_argument('-o', '--output', type=str, required=True,

help='输出文件路径 (不含扩展名)')

parser.add\_argument('--font-size', type=float, default=10.5,

help='字体大小 (默认: 10.5)')

parser.add\_argument('--line-spacing', type=float, default=10.5,

help='行间距 (默认: 10.5)')

parser.add\_argument('--english-font', type=str, default='Courier New',

help='英文字体 (默认: Courier New)')

parser.add\_argument('--chinese-font', type=str, default='SimSun',

help='中文字体 (默认: SimSun)')

parser.add\_argument('--page-limit', type=int, default=0,

help='每页最少代码行数 (默认: 0, 表示不限制)')

parser.add\_argument('--remove-empty', action='store\_true',

help='移除空行')

parser.add\_argument('--sort-files', action='store\_true',

help='按重要性排序文件 (后端优先，前端其次)')

return parser.parse\_args()

def read\_file\_list(file\_path):

"""读取文件列表"""

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

print(f"错误: 文件 '{file\_path}' 不存在")

sys.exit(1)

files = []

with open(file\_path, 'r', encoding='utf-8') as f:

for line in f:

line = line.strip()

# 跳过注释行和空行

if not line or line.startswith('#'):

continue

files.append(line)

return files

def build\_command(args, files):

"""构建文档生成命令"""

# 基本命令

cmd = [

'python', 'run\_simple.py',

'-n', args.name,

'-v', args.version,

'-o', args.output

]

# 添加格式选项

cmd.extend(['--font-size', str(args.font\_size)])

cmd.extend(['--line-spacing', str(args.line\_spacing)])

cmd.extend(['--english-font', args.english\_font])

cmd.extend(['--chinese-font', args.chinese\_font])

# 添加其他选项

if args.page\_limit > 0:

cmd.extend(['--page-limit', str(args.page\_limit)])

if args.remove\_empty:

cmd.append('--remove-empty')

if args.sort\_files:

cmd.append('--sort-files')

# 添加文件列表

for file in files:

cmd.extend(['--include-file', file])

return cmd

def main():

"""主函数"""

print("软著文档生成工具命令行接口 v1.0")

# 解析命令行参数

args = parse\_args()

# 读取文件列表

print(f"从 '{args.input}' 读取文件列表...")

files = read\_file\_list(args.input)

print(f"找到 {len(files)} 个文件")

# 构建命令

cmd = build\_command(args, files)

cmd\_str = ' '.join(cmd)

print(f"\n将执行命令:\n{cmd\_str}")

# 确认执行

confirm = input("\n是否继续? (y/n): ").lower()

if confirm != 'y':

print("操作已取消")

sys.exit(0)

# 执行命令

print("\n正在生成文档...")

try:

result = subprocess.run(cmd, check=True, text=True, capture\_output=True)

print("\n命令执行成功!")

print(result.stdout)

except subprocess.CalledProcessError as e:

print(f"\n命令执行失败: {e}")

print(e.stderr)

sys.exit(1)

print(f"\n文档已生成: {args.output}.docx")

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

main()

cli.py

"""

命令行界面模块

提供命令行参数解析和主程序入口。

"""

import os

import sys

import click

from typing import List, Dict, Any

from sofcrtpro.config\_manager import ConfigManager

from sofcrtpro.file\_scanner import scan\_directory, classify\_files

from sofcrtpro.code\_processor import process\_code\_files

from sofcrtpro.document\_generator import generate\_document

from sofcrtpro.utils.file\_utils import exclude\_directories, filter\_files\_by\_extension

@click.command()

@click.option('--source-dir', '-s', type=click.Path(exists=True, file\_okay=False, dir\_okay=True),

help='源代码目录路径')

@click.option('--output-path', '-o', default='output/source\_code',

help='输出文件路径，不包括扩展名')

@click.option('--output-format', '-f', type=click.Choice(['docx', 'pdf']), default='docx',

help='输出格式 (docx 或 pdf)')

@click.option('--font-name', default='Courier New',

help='字体名称，推荐等宽字体如Courier New')

@click.option('--font-size', type=click.IntRange(8, 14), default=10,

help='字体大小（磅），范围8-14')

@click.option('--lines-per-page', type=click.IntRange(10, 100), default=50,

help='每页显示的代码行数')

@click.option('--extensions', '-e', multiple=True,

help='要包含的文件扩展名，例如 -e .py -e .java')

@click.option('--exclude-dirs', '-x', multiple=True,

help='要排除的目录，例如 -x venv -x node\_modules')

@click.option('--config-file', '-c', type=click.Path(exists=False),

help='配置文件路径')

@click.option('--save-config', is\_flag=True,

help='保存当前配置到配置文件')

def main(

source\_dir: str,

output\_path: str,

output\_format: str,

font\_name: str,

font\_size: int,

lines\_per\_page: int,

extensions: List[str],

exclude\_dirs: List[str],

config\_file: str,

save\_config: bool

) -> None:

"""

SoftCopyrightPro - 软件著作权材料生成工具

扫描指定目录中的源代码文件，按照后端到前端的顺序拼接代码，

并输出为符合要求格式的文档。

"""

# 初始化配置管理器

config\_manager = ConfigManager(config\_file)

# 更新配置

config\_updates = {}

if source\_dir:

config\_updates['source\_dir'] = source\_dir

if output\_path:

config\_updates['output\_path'] = output\_path

if output\_format:

config\_updates['output\_format'] = output\_format

if font\_name:

config\_updates['font\_name'] = font\_name

if font\_size:

config\_updates['font\_size'] = font\_size

if lines\_per\_page:

config\_updates['lines\_per\_page'] = lines\_per\_page

if extensions:

config\_updates['file\_extensions'] = list(extensions)

if exclude\_dirs:

config\_updates['exclude\_dirs'] = list(exclude\_dirs)

config\_manager.update\_config(config\_updates)

# 获取最终配置

config = config\_manager.get\_config()

# 验证配置

errors = config\_manager.validate\_config()

if errors:

for key, error in errors.items():

click.echo(f"配置错误 ({key}): {error}", err=True)

sys.exit(1)

# 如果需要保存配置

if save\_config and config\_file:

if config\_manager.save\_config(config\_file):

click.echo(f"配置已保存到 {config\_file}")

else:

click.echo(f"保存配置失败", err=True)

# 开始处理

click.echo(f"扫描目录: {config['source\_dir']}")

# 扫描文件

all\_files = scan\_directory(config['source\_dir'])

# 排除指定目录

all\_files = exclude\_directories(all\_files, config\_manager.get\_exclude\_dirs())

# 按扩展名筛选

filtered\_files = filter\_files\_by\_extension(all\_files, config\_manager.get\_file\_extensions())

click.echo(f"找到 {len(filtered\_files)} 个符合条件的源代码文件")

# 分类文件

classified\_files = classify\_files(

filtered\_files,

config\_manager.get\_backend\_identifiers(),

config\_manager.get\_frontend\_identifiers()

)

click.echo(f"后端文件: {len(classified\_files['backend'])} 个")

click.echo(f"前端文件: {len(classified\_files['frontend'])} 个")

# 处理代码文件

merged\_content, pages = process\_code\_files(classified\_files, config['lines\_per\_page'])

click.echo(f"合并后的代码总行数: {len(merged\_content)}")

click.echo(f"分页后的页数: {len(pages)}")

# 生成文档

output\_file = generate\_document(

content=merged\_content,

output\_path=config['output\_path'],

output\_format=config['output\_format'],

font\_name=config['font\_name'],

font\_size=config['font\_size']

)

click.echo(f"文档已生成: {output\_file}")

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

main()

code\_processor.py

"""

代码处理模块

负责读取源代码文件内容，移除空行，添加文件名标识，并按照规定顺序合并所有代码。

"""

import os

from typing import List, Dict, Tuple

def read\_file\_content(file\_path: str) -> List[str]:

"""

读取文件内容，返回所有非空行

Args:

file\_path: 文件路径

Returns:

文件内容的非空行列表

"""

try:

with open(file\_path, 'r', encoding='utf-8', errors='ignore') as f:

# 读取所有非空行

return [line.rstrip() for line in f if line.strip()]

except Exception as e:

print(f"无法读取文件 {file\_path}: {e}")

return []

def add\_file\_header(file\_path: str, content: List[str]) -> List[str]:

"""

在代码内容前添加文件名标识

Args:

file\_path: 文件路径

content: 代码内容行列表

Returns:

添加了文件名标识的代码内容

"""

# 获取相对路径作为文件标识

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

# 创建文件标识行 - 简化标识，只保留文件名

header = [file\_name]

# 合并标识和内容

return header + content

def sort\_files\_by\_importance(files: List[str]) -> List[str]:

"""

按重要性对文件进行排序

Args:

files: 文件路径列表

Returns:

排序后的文件路径列表

"""

# 定义文件重要性评分函数

def get\_importance\_score(file\_path: str) -> int:

# 文件名和路径

file\_name = os.path.basename(file\_path).lower()

file\_path\_lower = file\_path.lower()

# 初始分数

score = 0

# 根据文件名关键词增加分数

important\_keywords = ['main', 'core', 'app', 'index', 'server', 'api', 'config', 'model', 'controller']

for i, keyword in enumerate(important\_keywords):

if keyword in file\_name:

score += (10 - i) # 越靠前的关键词权重越高

# 根据目录结构增加分数

if 'core' in file\_path\_lower:

score += 5

if 'model' in file\_path\_lower:

score += 4

if 'service' in file\_path\_lower:

score += 3

if 'controller' in file\_path\_lower:

score += 2

if 'util' in file\_path\_lower:

score += 1

return score

# 按重要性分数降序排序

return sorted(files, key=get\_importance\_score, reverse=True)

def merge\_code\_files(files\_dict: Dict[str, List[str]]) -> Tuple[List[str], List[str]]:

"""

按照后端优先、前端其次的顺序合并所有代码文件

Args:

files\_dict: 包含'backend'和'frontend'键的字典，值为对应的文件路径列表

Returns:

元组(合并后的所有代码行, 处理的文件列表)

"""

merged\_content = []

processed\_files = []

# 对后端文件按重要性排序

backend\_files = sort\_files\_by\_importance(files\_dict.get('backend', []))

# 处理后端文件

for file\_path in backend\_files:

content = read\_file\_content(file\_path)

if content:

merged\_content.extend(add\_file\_header(file\_path, content))

processed\_files.append(file\_path)

# 对前端文件按重要性排序

frontend\_files = sort\_files\_by\_importance(files\_dict.get('frontend', []))

# 处理前端文件

for file\_path in frontend\_files:

content = read\_file\_content(file\_path)

if content:

merged\_content.extend(add\_file\_header(file\_path, content))

processed\_files.append(file\_path)

return merged\_content, processed\_files

def split\_into\_pages(content: List[str], lines\_per\_page: int = 50) -> List[List[str]]:

"""

将代码内容分割成固定行数的页面，确保每页不少于指定行数

Args:

content: 代码内容行列表

lines\_per\_page: 每页的最少行数

Returns:

分页后的代码内容，每个元素是一页的内容

"""

pages = []

total\_lines = len(content)

for i in range(0, total\_lines, lines\_per\_page):

end = min(i + lines\_per\_page, total\_lines)

pages.append(content[i:end])

return pages

def process\_code\_files(files\_dict: Dict[str, List[str]], lines\_per\_page: int = 50) -> Tuple[List[str], List[List[str]], List[str]]:

"""

处理所有代码文件，合并内容并分页

Args:

files\_dict: 包含'backend'和'frontend'键的字典，值为对应的文件路径列表

lines\_per\_page: 每页的最少行数

Returns:

元组(合并后的所有代码行, 分页后的代码内容, 处理的文件列表)

"""

# 合并所有代码文件

merged\_content, processed\_files = merge\_code\_files(files\_dict)

# 分页

pages = split\_into\_pages(merged\_content, lines\_per\_page)

return merged\_content, pages, processed\_files

document\_generator.py

"""

文档生成模块

负责生成Word和PDF格式的代码文档，设置等宽字体、字号和页面布局。

"""

import os

from typing import List, Optional, Dict, Tuple

from docx import Document

from docx.shared import Pt, Inches, RGBColor

from docx.enum.text import WD\_ALIGN\_PARAGRAPH

from docx.enum.section import WD\_SECTION

from docx.oxml.ns import qn

from docx.oxml import OxmlElement, parse\_xml

from docx.shared import Mm, Twips

def set\_paragraph\_spacing(paragraph, before=0, after=2.3, line=10.5):

"""

设置段落间距

Args:

paragraph: 段落对象

before: 段前间距（磅）

after: 段后间距（磅）

line: 行间距（磅）

"""

p\_format = paragraph.paragraph\_format

p\_format.space\_before = Pt(before)

p\_format.space\_after = Pt(after)

p\_format.line\_spacing = Pt(line)

def add\_page\_number(run):

"""

添加页码字段到run

Args:

run: 文本run对象

"""

fldChar1 = OxmlElement('w:fldChar')

fldChar1.set(qn('w:fldCharType'), 'begin')

instrText = OxmlElement('w:instrText')

instrText.set(qn('xml:space'), 'preserve')

instrText.text = "PAGE"

fldChar2 = OxmlElement('w:fldChar')

fldChar2.set(qn('w:fldCharType'), 'end')

run.\_r.append(fldChar1)

run.\_r.append(instrText)

run.\_r.append(fldChar2)

def add\_header\_border(paragraph):

"""

为段落添加下边框线

Args:

paragraph: 段落对象

"""

# 创建下边框

pPr = paragraph.\_p.get\_or\_add\_pPr()

pBdr = OxmlElement('w:pBdr')

bottom = OxmlElement('w:bottom')

bottom.set(qn('w:val'), 'single')

bottom.set(qn('w:sz'), '4') # 边框宽度

bottom.set(qn('w:space'), '0') # 边框与文本的间距设为0，使其紧贴文本

bottom.set(qn('w:color'), '000000') # 边框颜色

pBdr.append(bottom)

pPr.append(pBdr)

def set\_chinese\_font(run, font\_name\_cn):

"""

强制设置中文字体

Args:

run: 文本run对象

font\_name\_cn: 中文字体名称

"""

# 设置东亚文字字体（中文）

run.font.east\_asia = font\_name\_cn

# 在XML级别设置字体

r = run.\_r

rPr = r.get\_or\_add\_rPr()

# 设置东亚文字字体（中文）

eastAsia = OxmlElement('w:eastAsia')

eastAsia.set(qn('w:val'), font\_name\_cn)

# 删除任何现有的eastAsia元素

for element in rPr.findall(qn('w:eastAsia')):

rPr.remove(element)

rPr.append(eastAsia)

def create\_aligned\_header(section, software\_name, software\_version, font\_name\_en, font\_name\_cn):

"""

创建对齐的页眉，确保左右两侧高度一致，分割线紧贴页眉

Args:

section: 文档节

software\_name: 软件名称

software\_version: 软件版本号

font\_name\_en: 英文字体名称

font\_name\_cn: 中文字体名称

Returns:

页眉对象

"""

header = section.header

# 清除现有内容

for p in header.paragraphs:

p.\_element.getparent().remove(p.\_element)

p.\_p = None

p.\_element = None

# 创建单个段落用于页眉

header\_para = header.add\_paragraph()

header\_para.alignment = WD\_ALIGN\_PARAGRAPH.LEFT

header\_para.paragraph\_format.space\_before = Pt(0)

header\_para.paragraph\_format.space\_after = Pt(0)

# 添加软件名称和版本号（左侧）

left\_run = header\_para.add\_run(software\_name + " " + software\_version)

left\_run.font.size = Pt(9)

left\_run.font.name = font\_name\_en

set\_chinese\_font(left\_run, font\_name\_cn)

# 添加制表位，用于右对齐页码

tab\_stops = header\_para.paragraph\_format.tab\_stops

tab\_stop = tab\_stops.add\_tab\_stop(

Inches(section.page\_width.inches - section.left\_margin.inches - section.right\_margin.inches),

WD\_ALIGN\_PARAGRAPH.RIGHT

)

# 添加制表符和页码（右侧）

tab\_run = header\_para.add\_run("\t")

page\_run = header\_para.add\_run()

add\_page\_number(page\_run)

page\_run.font.size = Pt(9)

page\_run.font.name = font\_name\_en

set\_chinese\_font(page\_run, font\_name\_cn)

# 为页眉段落添加下边框线

add\_header\_border(header\_para)

return header

def create\_word\_document(

content: List[str],

output\_path: str,

file\_list: List[str] = None,

software\_name: str = "SoftCopyrightPro",

software\_version: str = "v0.1.0",

font\_name\_en: str = 'Courier New',

font\_name\_cn: str = 'SimSun', # 宋体

font\_size: float = 10.5, # 调整为10.5pt

page\_width: float = 8.5,

page\_height: float = 11.0,

margin: float = 1.0

) -> str:

"""

创建Word文档

Args:

content: 代码内容行列表

output\_path: 输出文件路径

file\_list: 文件列表，按处理顺序排列

software\_name: 软件名称（用于页眉）

software\_version: 软件版本号（用于页眉）

font\_name\_en: 英文等宽字体名称

font\_name\_cn: 中文等宽字体名称

font\_size: 字体大小（磅）

page\_width: 页面宽度（英寸）

page\_height: 页面高度（英寸）

margin: 页面边距（英寸）

Returns:

生成的文件路径

"""

# 创建Word文档

doc = Document()

# 设置文档默认字体

doc.styles['Normal'].font.name = font\_name\_en

doc.styles['Normal'].font.size = Pt(font\_size)

# 在XML级别设置默认字体

element = doc.styles['Normal'].\_element

rPr = element.get\_or\_add\_rPr()

# 设置默认字体

rFonts = OxmlElement('w:rFonts')

rFonts.set(qn('w:ascii'), font\_name\_en)

rFonts.set(qn('w:hAnsi'), font\_name\_en)

rFonts.set(qn('w:eastAsia'), font\_name\_cn)

rFonts.set(qn('w:cs'), font\_name\_en)

# 删除任何现有的rFonts元素

for old\_rFonts in rPr.findall(qn('w:rFonts')):

rPr.remove(old\_rFonts)

rPr.append(rFonts)

# 设置页面大小和边距

section = doc.sections[0]

section.page\_width = Inches(page\_width)

section.page\_height = Inches(page\_height)

section.left\_margin = Inches(margin)

section.right\_margin = Inches(margin)

section.top\_margin = Inches(margin)

section.bottom\_margin = Inches(margin)

# 创建对齐的页眉，确保左右两侧高度一致，分割线紧贴页眉

create\_aligned\_header(section, software\_name, software\_version, font\_name\_en, font\_name\_cn)

# 添加内容

for line in content:

paragraph = doc.add\_paragraph(line)

paragraph.style.font.name = font\_name\_en

paragraph.style.font.size = Pt(font\_size)

# 设置段落间距 - 段前0pt，段后2.3pt，行距10.5pt

set\_paragraph\_spacing(paragraph, before=0, after=2.3, line=10.5)

# 设置中文字体

if paragraph.runs:

run = paragraph.runs[0]

run.font.name = font\_name\_en # 英文字体

set\_chinese\_font(run, font\_name\_cn) # 中文字体

run.font.size = Pt(font\_size)

else:

# 如果runs为空，添加一个run

run = paragraph.add\_run(line)

run.font.name = font\_name\_en # 英文字体

set\_chinese\_font(run, font\_name\_cn) # 中文字体

run.font.size = Pt(font\_size)

# 清除段落原有文本，避免重复

paragraph.text = ""

# 确保输出目录存在

os.makedirs(os.path.dirname(os.path.abspath(output\_path)), exist\_ok=True)

# 保存文档

doc.save(output\_path)

return output\_path

def convert\_word\_to\_pdf(word\_path: str, pdf\_path: Optional[str] = None) -> str:

"""

将Word文档转换为PDF

Args:

word\_path: Word文档路径

pdf\_path: PDF输出路径，如果为None，则使用与Word文档相同的名称但扩展名为.pdf

Returns:

生成的PDF文件路径

"""

if pdf\_path is None:

pdf\_path = os.path.splitext(word\_path)[0] + '.pdf'

try:

# 尝试使用docx2pdf

from docx2pdf import convert

convert(word\_path, pdf\_path)

return pdf\_path

except ImportError:

try:

# 尝试使用win32com (仅适用于Windows)

import win32com.client

word = win32com.client.Dispatch('Word.Application')

doc = word.Documents.Open(word\_path)

doc.SaveAs(pdf\_path, FileFormat=17) # FileFormat=17 表示PDF

doc.Close()

word.Quit()

return pdf\_path

except ImportError:

print("无法将Word转换为PDF。请安装docx2pdf或确保在Windows系统上运行。")

return word\_path

def generate\_document(

content: List[str],

output\_path: str,

file\_list: List[str] = None,

output\_format: str = 'docx',

software\_name: str = "SoftCopyrightPro",

software\_version: str = "v0.1.0",

font\_name\_en: str = 'Courier New',

font\_name\_cn: str = 'SimSun', # 宋体

font\_size: float = 10.5 # 调整为10.5pt

) -> Tuple[str, Dict]:

"""

生成文档

Args:

content: 代码内容行列表

output\_path: 输出文件路径

file\_list: 文件列表，按处理顺序排列

output\_format: 输出格式，'docx'或'pdf'

software\_name: 软件名称（用于页眉）

software\_version: 软件版本号（用于页眉）

font\_name\_en: 英文等宽字体名称

font\_name\_cn: 中文等宽字体名称

font\_size: 字体大小（磅）

Returns:

元组(生成的文件路径, 文档统计信息)

"""

# 确保输出路径有正确的扩展名

base\_path = os.path.splitext(output\_path)[0]

word\_path = base\_path + '.docx'

# 创建Word文档

word\_path = create\_word\_document(

content=content,

output\_path=word\_path,

file\_list=file\_list,

software\_name=software\_name,

software\_version=software\_version,

font\_name\_en=font\_name\_en,

font\_name\_cn=font\_name\_cn,

font\_size=font\_size

)

# 计算统计信息

stats = {

'total\_lines': len(content),

'total\_pages': (len(content) + 49) // 50, # 估算页数，每页约50行

'file\_list': file\_list or []

}

# 如果需要PDF格式，则转换

if output\_format.lower() == 'pdf':

pdf\_path = base\_path + '.pdf'

return convert\_word\_to\_pdf(word\_path, pdf\_path), stats

return word\_path, stats

file\_scanner.py

"""

文件扫描与筛选模块

负责扫描指定目录下的源代码文件，根据后缀名筛选，并将文件分类为前端和后端。

"""

import os

import pathlib

from typing import List, Dict, Set, Tuple

def scan\_directory(directory: str, file\_extensions: Set[str] = None) -> List[str]:

"""

扫描指定目录及其子目录，返回符合后缀名要求的文件列表

Args:

directory: 要扫描的目录路径

file\_extensions: 要包含的文件后缀名集合，如{'.py', '.java', '.js'}

如果为None，则包含所有文件

Returns:

符合条件的文件路径列表

"""

if file\_extensions is None:

file\_extensions = set()

matched\_files = []

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

for file in files:

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

file\_ext = os.path.splitext(file)[1].lower()

# 如果file\_extensions为空或文件后缀在指定列表中

if not file\_extensions or file\_ext in file\_extensions:

matched\_files.append(file\_path)

return matched\_files

def classify\_files(

files: List[str],

backend\_identifiers: Set[str] = None,

frontend\_identifiers: Set[str] = None

) -> Dict[str, List[str]]:

"""

将文件分类为前端和后端

Args:

files: 文件路径列表

backend\_identifiers: 后端文件的标识（后缀或目录名）

frontend\_identifiers: 前端文件的标识（后缀或目录名）

Returns:

包含'backend'和'frontend'键的字典，值为对应的文件列表

"""

if backend\_identifiers is None:

# 默认后端文件后缀

backend\_identifiers = {'.py', '.java', '.c', '.cpp', '.cs', '.go', '.rb', '.php'}

if frontend\_identifiers is None:

# 默认前端文件后缀

frontend\_identifiers = {'.js', '.ts', '.jsx', '.tsx', '.html', '.css', '.vue', '.scss', '.less'}

result = {

'backend': [],

'frontend': []

}

for file\_path in files:

file\_ext = os.path.splitext(file\_path)[1].lower()

# 检查文件是否为后端文件

if file\_ext in backend\_identifiers:

result['backend'].append(file\_path)

# 检查文件是否为前端文件

elif file\_ext in frontend\_identifiers:

result['frontend'].append(file\_path)

# 如果无法确定，根据路径中的关键词进行判断

else:

path\_lower = file\_path.lower()

if any(ident in path\_lower for ident in ['backend', 'server', 'api']):

result['backend'].append(file\_path)

elif any(ident in path\_lower for ident in ['frontend', 'client', 'ui', 'web']):

result['frontend'].append(file\_path)

else:

# 无法确定的文件默认归为后端

result['backend'].append(file\_path)

return result

def get\_file\_info(files: List[str]) -> List[Tuple[str, int]]:

"""

获取文件信息，包括文件路径和代码行数（不包括空行）

Args:

files: 文件路径列表

Returns:

包含(文件路径, 行数)元组的列表

"""

file\_info = []

for file\_path in files:

try:

with open(file\_path, 'r', encoding='utf-8', errors='ignore') as f:

# 计算非空行数量

line\_count = sum(1 for line in f if line.strip())

file\_info.append((file\_path, line\_count))

except Exception as e:

print(f"无法读取文件 {file\_path}: {e}")

file\_info.append((file\_path, 0))

return file\_info