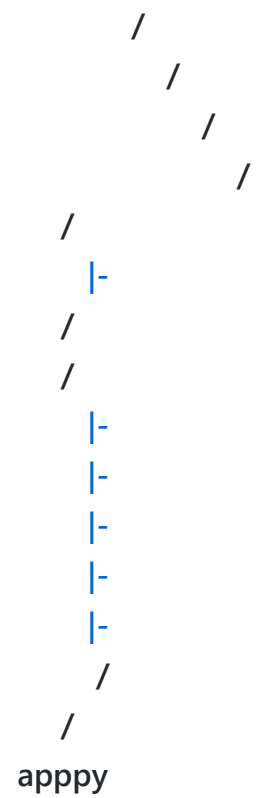
•

Contents



to top

```python
import os
import uuid
import threading
from flask import Flask, request, render_template, jsonify, send_file
from werkzeug.utils import secure_filename
from config import config
from utils.file_utils import FileUtils
from utils.compression import CompressionHandler
from utils.image_processor import ImageProcessor
from utils.pdf_generator import PDFGenerator

# 创建Flask应用
app = Flask(__name__)
app.config.from_object(config['default'])

# 全局变量存储处理状态
processing_status = {}
processing_results = {}

class ProcessingTask:
    """处理任务类"""

    def __init__(self, task_id):
        self.task_id = task_id
        self.status = "等待开始"
        self.progress = 0
        self.current_step = ""
        self.result_files = []
        self.error = None

    def update_status(self, status, progress=None, step=None):
        """更新任务状态"""
        self.status = status
        if progress is not None:
            self.progress = progress
        if step:
            self.current_step = step

        # 更新全局状态
        processing_status[self.task_id] = {
            'status': self.status,
            'progress': self.progress,
            'current_step': self.current_step,
            'error': self.error
        }

def process_compressed_file(task_id, file_path, output_dir):
    """处理压缩文件的主函数"""
    task = ProcessingTask(task_id)
    processing_status[task_id] = {
        'status': '等待开始',
        'progress': 0,
```

```
        'current_step': '',
        'error': None
    }

    try:
        # 步骤1: 创建临时目录
        task.update_status("创建临时目录", 5, "初始化")
        temp_dir = os.path.join(app.config['TEMP_FOLDER'], f"temp_{task_id}")
        os.makedirs(temp_dir, exist_ok=True)

        # 步骤2: 递归解压
        task.update_status("开始解压文件", 10, "解压")
        compression_handler = CompressionHandler()
        compression_handler.set_status_callback(
            lambda msg, prog=None: task.update_status(msg, prog, "解压")
        )

        extracted_files = compression_handler.recursive_extract(file_path, temp_dir)

        if not extracted_files:
            task.update_status("解压失败，没有找到文件", 100, "错误")
            task.error = "解压失败，没有找到文件"
            return

        task.update_status(f"解压完成，找到 {len(extracted_files)} 个文件", 30, "解压完
成")

        # 步骤3: 收集和排序图片
        task.update_status("收集图片文件", 40, "图片处理")
        image_processor = ImageProcessor()
        image_processor.set_status_callback(
            lambda msg, prog=None: task.update_status(msg, prog, "图片处理")
        )

        image_groups = image_processor.collect_and_sort_images(temp_dir)

        if not image_groups:
            task.update_status("没有找到图片文件", 100, "错误")
            task.error = "没有找到图片文件"
            return

        task.update_status(f"找到 {len(image_groups)} 个包含图片的文件夹", 50, "图片收集
完成")

        # 步骤4: 处理图片
        task.update_status("处理图片文件", 60, "图片优化")
        processed_image_groups = {}
        for folder_path, image_paths in image_groups.items():
            processed_images = image_processor.process_image_group(image_paths,
temp_dir)
            processed_image_groups[folder_path] = processed_images

        task.update_status("图片处理完成", 70, "图片处理完成")
```

```
        # 步骤5: 生成PDF
        task.update_status("生成PDF文件", 80, "PDF生成")
        pdf_generator = PDFGenerator()
        pdf_generator.set_status_callback(
            lambda msg, prog=None: task.update_status(msg, prog, "PDF生成")
        )

        # 创建PDF输出目录
        pdf_output_dir = os.path.join(output_dir, f"pdfs_{task_id}")
        os.makedirs(pdf_output_dir, exist_ok=True)

        # 生成PDF
        generated_pdfs = pdf_generator.generate_pdfs_by_folder(
            processed_image_groups,
            pdf_output_dir,
            base_name="converted"
        )

        if not generated_pdfs:
            task.update_status("PDF生成失败", 100, "错误")
            task.error = "PDF生成失败"
            return

        task.update_status(f"成功生成 {len(generated_pdfs)} 个PDF文件", 90, "PDF生成完
成")

        # 步骤6: 打包结果
        task.update_status("打包结果文件", 95, "打包")
        zip_output_path = os.path.join(output_dir, f"result_{task_id}.zip")

        if pdf_generator.create_pdf_package(list(generated_pdfs.values()),
zip_output_path):
            task.result_files = [zip_output_path]
            task.update_status("处理完成", 100, "完成")
        else:
            task.update_status("打包失败", 100, "错误")
            task.error = "打包失败"

        # 存储结果
        processing_results[task_id] = {
            'pdf_files': list(generated_pdfs.values()),
            'zip_file': zip_output_path if os.path.exists(zip_output_path) else None
        }

    except Exception as e:
        task.update_status(f"处理失败: {str(e)}", 100, "错误")
        task.error = str(e)
    finally:
        # 清理临时文件（保留输出文件供下载）
        try:
            # 清理上传文件和临时解压目录
            FileUtils.safe_remove(file_path)  # 删除上传的原始文件
```

```
            if 'temp_dir' in locals():
                FileUtils.safe_remove(temp_dir)  # 删除临时解压目录
        except Exception as cleanup_error:
            print(f"清理临时文件失败: {cleanup_error}")

@app.route('/')
def index():
    """主页"""
    return render_template('index.html')

@app.route('/upload', methods=['POST'])
def upload_file():
    """文件上传接口"""
    try:
        # 检查文件是否存在
        if 'file' not in request.files:
            return jsonify({'error': '没有选择文件'}), 400

        file = request.files['file']

        # 检查文件名
        if file.filename == '':
            return jsonify({'error': '没有选择文件'}), 400

        # 检查文件类型
        if not FileUtils.allowed_file(file.filename, app.config['ALLOWED_EXTENSIONS']):
            return jsonify({'error': '不支持的文件格式'}), 400

        # 创建必要的目录
        FileUtils.create_directories()

        # 生成任务ID
        task_id = str(uuid.uuid4())

        # 保存上传的文件
        filename = secure_filename(file.filename)
        file_path = os.path.join(app.config['UPLOAD_FOLDER'], f"{task_id}_{filename}")
        file.save(file_path)

        # 检查文件大小
        file_size = FileUtils.get_file_size(file_path)
        if file_size > 1024:  # 1GB限制
            os.remove(file_path)
            return jsonify({'error': '文件大小超过1GB限制'}), 400

        # 启动后台处理任务
        output_dir = app.config['OUTPUT_FOLDER']
        thread = threading.Thread(
            target=process_compressed_file,
            args=(task_id, file_path, output_dir)
        )
        thread.daemon = True
        thread.start()
```

```
        return jsonify({
            'task_id': task_id,
            'message': '文件上传成功，开始处理'
        })

    except Exception as e:
        return jsonify({'error': f'上传失败: {str(e)}'}), 500

@app.route('/status/<task_id>')
def get_status(task_id):
    """获取处理状态"""
    if task_id in processing_status:
        status_info = processing_status[task_id]

        # 检查是否完成且有结果
        if status_info['status'] == '处理完成' and task_id in processing_results:
            result = processing_results[task_id]
            status_info['download_url'] = f'/download/{task_id}'
            status_info['pdf_count'] = len(result.get('pdf_files', []))
            status_info['pdf_list_url'] = f'/download/list/{task_id}'

        return jsonify(status_info)
    else:
        return jsonify({'error': '任务不存在'}), 404

@app.route('/download/<task_id>')
def download_result(task_id):
    """下载处理结果（ZIP包）"""
    if task_id in processing_results:
        result = processing_results[task_id]
        zip_file = result.get('zip_file')

        if zip_file and os.path.exists(zip_file):
            filename = f"converted_pdfs_{task_id}.zip"
            return send_file(zip_file, as_attachment=True, download_name=filename)

    return jsonify({'error': '文件不存在或尚未完成'}), 404

@app.route('/download/pdf/<task_id>/<int:pdf_index>')
def download_single_pdf(task_id, pdf_index):
    """下载单个PDF文件"""
    if task_id in processing_results:
        result = processing_results[task_id]
        pdf_files = result.get('pdf_files', [])

        if 0 <= pdf_index < len(pdf_files):
            pdf_file = pdf_files[pdf_index]
            if os.path.exists(pdf_file):
                filename = os.path.basename(pdf_file)
                return send_file(pdf_file, as_attachment=True, download_name=filename)

    return jsonify({'error': 'PDF文件不存在'}), 404
```

```python
@app.route('/download/list/<task_id>')
def list_pdf_files(task_id):
    """获取PDF文件列表"""
    if task_id in processing_results:
        result = processing_results[task_id]
        pdf_files = result.get('pdf_files', [])

        file_list = []
        for i, pdf_file in enumerate(pdf_files):
            if os.path.exists(pdf_file):
                file_info = {
                    'index': i,
                    'filename': os.path.basename(pdf_file),
                    'size': os.path.getsize(pdf_file),
                    'download_url': f'/download/pdf/{task_id}/{i}'
                }
                file_list.append(file_info)

        return jsonify({'pdf_files': file_list})

    return jsonify({'error': '任务不存在'}), 404

@app.route('/cleanup', methods=['POST'])
def cleanup_files():
    """清理临时文件"""
    try:
        # 清理上传文件
        FileUtils.cleanup_old_files(app.config['UPLOAD_FOLDER'])

        # 清理临时文件
        FileUtils.cleanup_old_files(app.config['TEMP_FOLDER'])

        # 清理输出文件（保留时间更长）
        FileUtils.cleanup_old_files(app.config['OUTPUT_FOLDER'], hours_old=48)

        return jsonify({'message': '清理完成'})
    except Exception as e:
        return jsonify({'error': f'清理失败: {str(e)}'}), 500

@app.route('/cleanup/task/<task_id>', methods=['POST'])
def cleanup_task_files(task_id):
    """清理指定任务的所有文件"""
    try:
        FileUtils.cleanup_task_files(
            task_id,
            app.config['UPLOAD_FOLDER'],
            app.config['TEMP_FOLDER'],
            app.config['OUTPUT_FOLDER']
        )
        return jsonify({'message': f'任务 {task_id} 文件清理完成'})
    except Exception as e:
        return jsonify({'error': f'清理失败: {str(e)}'}), 500
```

```
if __name__ == '__main__':
    # 创建必要的目录
    FileUtils.create_directories()

    # 启动应用
    app.run(debug=True, host='0.0.0.0', port=5000)
```

cleanuppy

to top

```python
#!/usr/bin/env python3
"""
文件清理脚本
用于清理上传文件、临时文件和旧的输出文件
"""

import os
import time
from config import config
from utils.file_utils import FileUtils

def cleanup_all_files():
    """清理所有临时和旧文件"""
    print("开始清理所有文件...")

    config_obj = config['default']

    # 确保目录存在
    FileUtils.create_directories()

    # 清理上传文件夹（保留24小时内的文件）
    print("清理上传文件夹...")
    FileUtils.cleanup_old_files(config_obj.UPLOAD_FOLDER, hours_old=24)

    # 清理临时文件夹（保留12小时内的文件）
    print("清理临时文件夹...")
    FileUtils.cleanup_old_files(config_obj.TEMP_FOLDER, hours_old=12)

    # 清理输出文件夹（保留48小时内的文件）
    print("清理输出文件夹...")
    FileUtils.cleanup_old_files(config_obj.OUTPUT_FOLDER, hours_old=48)

    print("文件清理完成！")

def cleanup_task_files(task_id):
    """清理指定任务的所有文件"""
    print(f"清理任务 {task_id} 的文件...")

    config_obj = config['default']
    FileUtils.cleanup_task_files(
        task_id,
        config_obj.UPLOAD_FOLDER,
        config_obj.TEMP_FOLDER,
        config_obj.OUTPUT_FOLDER
    )

def show_file_stats():
    """显示文件统计信息"""
    print("\n文件统计信息:")
    print("-" * 40)

    config_obj = config['default']
```

```
    for folder_name, folder_path in [
        ("上传文件夹", config_obj.UPLOAD_FOLDER),
        ("临时文件夹", config_obj.TEMP_FOLDER),
        ("输出文件夹", config_obj.OUTPUT_FOLDER)
    ]:
        if os.path.exists(folder_path):
            files = os.listdir(folder_path)
            total_size = 0

            for file in files:
                file_path = os.path.join(folder_path, file)
                if os.path.isfile(file_path):
                    total_size += os.path.getsize(file_path)

            print(f"{folder_name}: {len(files)} 个文件, {total_size / (1024*1024):.2f}
MB")
        else:
            print(f"{folder_name}: 目录不存在")

if __name__ == '__main__':
    print("压缩包转PDF工具 - 文件清理脚本")
    print("=" * 50)

    # 显示当前文件状态
    show_file_stats()

    # 询问用户要执行的操作
    print("\n请选择操作:")
    print("1. 清理所有临时和旧文件")
    print("2. 清理指定任务的文件")
    print("3. 仅显示文件统计")

    choice = input("\n请输入选择 (1-3): ").strip()

    if choice == '1':
        cleanup_all_files()
    elif choice == '2':
        task_id = input("请输入任务ID: ").strip()
        if task_id:
            cleanup_task_files(task_id)
        else:
            print("无效的任务ID")
    elif choice == '3':
        show_file_stats()
    else:
        print("无效的选择")

    print("\n操作完成! ")
```

configpy

to top

```python
import os

class Config:
    # 基础配置
    SECRET_KEY = os.environ.get('SECRET_KEY') or 'dev-secret-key'

    # 文件上传配置
    MAX_CONTENT_LENGTH = 1024 * 1024 * 1024  # 1GB
    UPLOAD_FOLDER = 'uploads'
    TEMP_FOLDER = 'temp'
    OUTPUT_FOLDER = 'outputs'

    # 允许的压缩文件扩展名
    ALLOWED_EXTENSIONS = {
        'zip', 'tar', 'gz', 'bz2', 'rar', '7z', 'tar.gz', 'tar.bz2'
    }

    # 允许的图片文件扩展名
    ALLOWED_IMAGE_EXTENSIONS = {
        'jpg', 'jpeg', 'png', 'gif', 'bmp', 'webp'
    }

    # PDF配置
    PDF_PAGE_SIZE = 'A4'
    PDF_ORIENTATION = 'portrait'  # portrait 或 landscape

    # 清理配置（小时）
    CLEANUP_INTERVAL = 24  # 24小时后清理临时文件

class DevelopmentConfig(Config):
    DEBUG = True

class ProductionConfig(Config):
    DEBUG = False

config = {
    'development': DevelopmentConfig,
    'production': ProductionConfig,
    'default': DevelopmentConfig
}
```

flask_zip_to_pdf_project_planmd

to top

Flask压缩包转PDF项目详细计划

项目概述

创建一个Python Flask Web应用，能够处理嵌套压缩包，提取其中的图片文件，并按文件夹分组生成A4竖排的PDF文件。

技术规格

技术栈

- 后端框架: Python Flask
- 压缩包处理: zipfile, tarfile, rarfile, py7zr
- 图片处理: Pillow (PIL)
- **PDF**生成: img2pdf
- 文件类型检测: python-magic
- 前端: HTML5, CSS3, JavaScript (原生)
- 进度显示: Server-Sent Events (SSE)

支持的文件格式

- 压缩格式: .zip, .tar.gz, .tar.bz2, .rar, .7z
- 图片格式: .webp, .jpg, .jpeg, .png, .gif, .bmp

项目架构

系统架构图

```
graph TB
    subgraph "前端界面"
        A[文件上传表单] --> B[进度显示]
        B --> C[结果下载]
    end

    subgraph "Flask后端"
        D[文件接收] --> E[文件验证]
        E --> F[递归解压]
        F --> G[图片收集]
        G --> H[PDF生成]
        H --> I[结果打包]
    end

    subgraph "文件系统"
        J[uploads/] --> K[temp/]
        K --> L[outputs/]
    end

    A --> D
    I --> C
    F --> M[状态更新]
    M --> B
```

目录结构

```
zip_to_pdf_app/
├── app.py                    # Flask主应用
├── requirements.txt          # Python依赖包
├── config.py                 # 配置文件
├── utils/                    # 工具模块
│   ├── __init__.py
│   ├── file_utils.py         # 文件处理工具
│   ├── compression.py        # 压缩包处理
│   ├── image_processor.py    # 图片处理
│   └── pdf_generator.py      # PDF生成
├── static/
│   ├── css/
│   │   └── style.css         # 样式文件
│   └── js/
│       └── main.js           # 前端JavaScript
├── templates/
│   └── index.html            # 主页面模板
├── uploads/                  # 上传文件临时存储
├── temp/                     # 解压临时目录
└── outputs/                  # 生成的PDF文件
```

核心功能模块详细设计

1. 文件上传模块 (app.py)

```
# 主要功能：
- 接收multipart/form-data文件上传
- 文件大小验证（最大1GB）
- 文件类型验证（支持的压缩格式）
- 实时上传进度显示
- 文件存储到uploads目录
```

2. 递归解压模块 (utils/compression.py)

```
# 主要功能：
- 自动检测压缩包格式
- 递归解压嵌套压缩包
- 保持原始目录结构
- 实时解压状态跟踪
- 临时文件管理
```

3. 图片处理模块 (utils/image_processor.py)

```
# 主要功能：
- 图片文件识别和过滤
- 按文件名自然排序
- 图片格式转换（如webp转png）
- 图片质量优化
- 按文件夹分组图片
```

4. PDF生成模块 (utils/pdf_generator.py)

```
# 主要功能：
- A4竖排页面设置
- 图片自适应页面大小
- 按文件夹生成多个PDF
- PDF文件命名和存储
- 生成结果打包
```

5. 前端界面模块

```
<!-- 主要功能： -->
- 拖拽文件上传界面
- 实时进度条显示
- 解压状态实时更新
- 下载链接生成
- 响应式设计
```

详细实现步骤

阶段1: 项目基础设置 (预计时间: 1小时)

1. 创建项目目录结构
2. 配置requirements.txt依赖包
3. 设置Flask应用基础配置
4. 创建必要的目录结构

阶段2: 文件上传功能 (预计时间: 2小时)

1. 实现Flask文件上传路由
2. 添加文件验证逻辑
3. 实现上传进度显示
4. 测试文件上传功能

阶段3: 递归解压功能 (预计时间: 3小时)

1. 实现多种压缩格式支持
2. 开发递归解压算法
3. 添加解压状态跟踪

4. 测试嵌套压缩包处理

阶段4: 图片处理和PDF生成 (预计时间: 2小时)

1. 实现图片文件识别和排序
2. 开发PDF生成逻辑
3. 按文件夹分组处理
4. 测试PDF生成功能

阶段5: 前端界面开发 (预计时间: 2小时)

1. 创建用户友好的上传界面
2. 实现实时进度更新
3. 添加下载功能
4. 优化用户体验

阶段6: 集成测试和优化 (预计时间: 2小时)

1. 端到端功能测试
2. 性能优化和错误处理
3. 内存泄漏检查
4. 用户体验优化

关键算法设计

递归解压算法

```
def recursive_extract(file_path, extract_to, status_callback):
    """
    递归解压压缩包
    """
    # 1. 检测文件类型
    # 2. 解压到临时目录
    # 3. 遍历解压后的文件
    # 4. 对每个文件检测是否为压缩包
    # 5. 如果是压缩包，递归调用
    # 6. 更新解压状态
```

图片收集和排序算法

```python
def collect_and_sort_images(root_dir):
    """
    收集并排序图片文件
    """
    # 1. 遍历目录结构
    # 2. 识别图片文件
    # 3. 按文件夹分组
    # 4. 按文件名自然排序
    # 5. 返回分组后的图片列表
```

配置要求

系统要求

- Python 3.8+
- 足够磁盘空间（至少2GB空闲）
- 内存：建议4GB+

Python依赖包

```
Flask==2.3.3
Pillow==10.0.0
img2pdf==0.4.4
python-magic==0.4.27
rarfile==4.0
py7zr==0.20.4
```

测试计划

功能测试

1. 单层压缩包处理
2. 多层嵌套压缩包处理
3. 多种图片格式支持
4. 大文件处理（接近1GB）
5. 错误文件处理

性能测试

1. 解压速度测试
2. 内存使用监控
3. 并发处理测试
4. 磁盘空间管理

部署说明

本地开发运行

```
# 安装依赖
pip install -r requirements.txt

# 运行应用
python app.py
```

访问地址

http://localhost:5000

风险管理和应对措施

技术风险

1. **内存溢出**: 使用流式处理，及时清理临时文件
2. **文件损坏**: 添加文件验证和错误恢复机制
3. **格式不支持**: 提供清晰的错误提示信息

用户体验风险

1. **长时间等待**: 提供详细的进度信息
2. **操作复杂**: 简化界面设计，提供明确指引

成功标准

- 支持所有指定的压缩和图片格式
- 正确处理嵌套压缩包
- 按文件夹生成正确的PDF文件
- 提供良好的用户体验
- 稳定处理1GB以内的文件

这个计划涵盖了项目的所有关键方面，从技术实现到用户体验，确保项目能够成功交付。

READMEmd

to top

Flask压缩包转PDF工具

一个基于Python Flask的Web应用，能够处理嵌套压缩包，提取其中的图片文件，并按文件夹分组生成A4竖排的PDF文件。

功能特性

- ☑ **支持多种压缩格式**: ZIP, TAR, RAR, 7Z (包括.tar.gz, .tar.bz2)

- ☑ **嵌套压缩包处理**: 自动递归解压多层嵌套的压缩包
- ☑ **多种图片格式支持**: JPG, PNG, WEBP, GIF, BMP
- ☑ **按文件夹分组**: 保持原始目录结构，按文件夹生成多个PDF
- ☑ **保持图片顺序**: 按文件名自然排序，保持图片原始顺序
- ☑ **A4竖排PDF**: 生成标准A4尺寸的PDF文件
- ☑ **实时进度显示**: 上传、解压、处理、生成全过程进度显示
- ☑ **大文件支持**: 支持最大1GB的压缩包
- ☑ **用户友好界面**: 现代化的Web界面，支持拖拽上传
- ☑ **灵活的下载选项**: 支持完整包下载和单个PDF文件下载
- ☑ **自动文件清理**: 处理完成后自动清理临时文件，避免磁盘占用

项目结构

```
zip_to_pdf_app/
├── app.py                    # Flask主应用
├── cleanup.py                # 文件清理脚本
├── requirements.txt          # Python依赖包
├── config.py                 # 配置文件
├── README.md                 # 项目说明文档
├── utils/                    # 工具模块
│   ├── __init__.py
│   ├── file_utils.py         # 文件处理工具
│   ├── compression.py        # 压缩包处理
│   ├── image_processor.py    # 图片处理
│   └── pdf_generator.py      # PDF生成
├── static/
│   ├── css/
│   │   └── style.css         # 样式文件
│   └── js/
│       └── main.js           # 前端JavaScript
├── templates/
│   └── index.html            # 主页面模板
├── uploads/                  # 上传文件临时存储
├── temp/                     # 解压临时目录
└── outputs/                  # 生成的PDF文件
```

安装和运行

1. 安装依赖

```
pip install -r requirements.txt
```

2. 运行应用

```
python app.py
```

3. 访问应用

打开浏览器访问: http://localhost:5000

使用说明

1. **上传压缩包**: 点击选择文件或拖拽压缩包到上传区域
2. **等待处理**: 系统会自动解压、处理图片并生成PDF
3. **下载结果**: 处理完成后选择下载方式：
   - **完整包下载**: 下载包含所有PDF的ZIP文件
   - **单独下载**: 在PDF文件列表中单独下载需要的PDF

下载结果

处理完成后，系统提供多种下载方式：

1. 完整下载

- 下载包含所有PDF文件的ZIP压缩包
- 适合批量下载所有转换结果

2. 单独下载

- 查看生成的PDF文件列表
- 单独下载每个PDF文件
- 适合只需要部分文件的情况

下载功能特点：

- **ZIP包下载**: 一键下载所有PDF文件
- **PDF列表**: 显示每个PDF文件的详细信息（文件名、大小）
- **单独下载**: 点击即可下载单个PDF文件
- **响应式设计**: 在手机和电脑上都能良好显示

文件清理机制

自动清理

- **上传文件**: 处理完成后立即删除原始上传文件
- **临时文件**: 处理完成后立即删除解压的临时文件
- **输出文件**: 保留48小时供下载，之后自动清理

手动清理

使用清理脚本管理文件：

```
# 运行清理脚本
python cleanup.py

# 选项：
# 1. 清理所有临时和旧文件
# 2. 清理指定任务的文件
# 3. 显示文件统计信息
```

API清理接口

- `POST /cleanup` - 清理所有临时文件
- `POST /cleanup/task/<task_id>` - 清理指定任务的所有文件

技术栈

- 后端: Python Flask
- 文件处理: zipfile, tarfile, rarfile, py7zr
- 图片处理: Pillow (PIL)
- PDF生成: img2pdf
- 文件类型检测: mimetypes (Python内置)
- 前端: HTML5, CSS3, JavaScript (原生)

配置说明

在 `config.py` 中可以修改以下配置：

- 文件大小限制: 默认1GB
- 支持的压缩格式: ZIP, TAR, RAR, 7Z等
- 支持的图片格式: JPG, PNG, WEBP等
- PDF页面设置: A4竖排
- 临时文件清理: 24小时自动清理

API接口

POST /upload

文件上传接口

- 参数: file (压缩包文件)
- 返回: task_id (任务ID)

anchor=true>GET /status/<task_id>

获取处理状态

- 返回: 处理进度、状态信息

anchor=true>GET /download/<task_id>

下载处理结果（ZIP包）

- 返回: PDF文件包ZIP

anchor=true>GET /download/list/<task_id>

获取PDF文件列表

- 返回: PDF文件信息列表

/ anchor=true>GET /download/pdf/<task_id>/<pdf_index>

下载单个PDF文件

- 返回: 指定的PDF文件

POST /cleanup

清理临时文件

anchor=true>POST /cleanup/task/<task_id>

清理指定任务的所有文件

处理流程

1. **文件上传**: 接收并验证压缩包文件
2. **递归解压**: 自动解压所有嵌套压缩包
3. **图片收集**: 按文件夹分组收集图片文件
4. **图片处理**: 格式转换和尺寸优化
5. **PDF生成**: 按文件夹生成多个PDF文件
6. **结果打包**: 将所有PDF打包成ZIP文件
7. **文件下载**: 提供多种下载选项
8. **自动清理**: 处理完成后清理临时文件

注意事项

- 确保有足够的磁盘空间处理大文件
- 临时文件会自动清理，避免磁盘空间占用
- 支持中文字符的文件名和路径
- 建议在本地网络环境中使用

故障排除

常见问题

1. 文件上传失败
    - 检查文件大小是否超过1GB限制
    - 检查文件格式是否支持
2. 解压失败
    - 检查压缩包是否损坏
    - 检查是否受密码保护
3. **PDF生成失败**
    - 检查图片文件是否损坏
    - 检查磁盘空间是否充足
4. 磁盘空间不足
    - 运行清理脚本释放空间
    - 手动删除不需要的输出文件

日志查看

应用运行时的错误信息会在控制台输出，可以据此进行故障诊断。

许可证

MIT License

贡献

欢迎提交Issue和Pull Request来改进这个项目。

requirementstxt

to top

```
Flask==2.3.3
Pillow==10.0.0
img2pdf==0.4.4
rarfile==4.0
py7zr==0.20.4
Werkzeug==2.3.7
```

runpy

to top

```python
#!/usr/bin/env python3
"""
Flask压缩包转PDF工具 - 启动脚本
"""

import os
import sys
import webbrowser
import threading
import time
from utils.file_utils import FileUtils

def check_dependencies():
    """检查必要的依赖包"""
    required_packages = [
        'flask',
        'pillow',
        'img2pdf',
        'python-magic',
        'rarfile',
        'py7zr'
    ]

    missing_packages = []

    for package in required_packages:
        try:
            __import__(package.replace('-', '_'))
        except ImportError:
            missing_packages.append(package)

    if missing_packages:
        print("缺少必要的依赖包:")
        for package in missing_packages:
            print(f"  - {package}")
        print("\n请运行以下命令安装依赖:")
        print("pip install -r requirements.txt")
        return False

    return True

def open_browser():
    """在浏览器中打开应用"""
    time.sleep(2)  # 等待应用启动
    webbrowser.open('http://localhost:5000')

def main():
    """主函数"""
    print("=" * 50)
    print("Flask压缩包转PDF工具")
    print("=" * 50)
```

```
    # 检查依赖
    print("检查依赖包...")
    if not check_dependencies():
        sys.exit(1)

    # 创建必要目录
    print("创建项目目录...")
    FileUtils.create_directories()

    # 导入并启动Flask应用
    try:
        from app import app

        print("启动Flask应用...")
        print("应用地址: http://localhost:5000")
        print("按 Ctrl+C 停止应用")
        print("-" * 50)

        # 在浏览器中打开应用
        browser_thread = threading.Thread(target=open_browser)
        browser_thread.daemon = True
        browser_thread.start()

        # 启动Flask应用
        app.run(debug=True, host='0.0.0.0', port=5000, use_reloader=False)

    except KeyboardInterrupt:
        print("\n应用已停止")
    except Exception as e:
        print(f"启动失败: {e}")
        sys.exit(1)

if __name__ == '__main__':
    main()
```

static\js\mainjs

to top

```javascript
// 主要JavaScript功能增强

class ZipToPDFApp {
    constructor() {
        this.currentTaskId = null;
        this.statusCheckInterval = null;
        this.initializeEventListeners();
    }

    initializeEventListeners() {
        // 文件拖拽事件
        const uploadArea = document.getElementById('uploadArea');
        const fileInput = document.getElementById('fileInput');

        if (uploadArea && fileInput) {
            this.setupDragAndDrop(uploadArea, fileInput);
            this.setupFileInput(fileInput);
        }

        // 清理按钮事件（如果有的话）
        const cleanupBtn = document.getElementById('cleanupBtn');
        if (cleanupBtn) {
            cleanupBtn.addEventListener('click', () => this.cleanupFiles());
        }
    }

    setupDragAndDrop(uploadArea, fileInput) {
        // 阻止默认拖拽行为
        ['dragenter', 'dragover', 'dragleave', 'drop'].forEach(eventName => {
            uploadArea.addEventListener(eventName, this.preventDefaults, false);
            document.body.addEventListener(eventName, this.preventDefaults, false);
        });

        // 高亮拖拽区域
        ['dragenter', 'dragover'].forEach(eventName => {
            uploadArea.addEventListener(eventName, () => {
                uploadArea.classList.add('dragover');
            }, false);
        });

        ['dragleave', 'drop'].forEach(eventName => {
            uploadArea.addEventListener(eventName, () => {
                uploadArea.classList.remove('dragover');
            }, false);
        });

        // 处理文件放置
        uploadArea.addEventListener('drop', (e) => {
            const files = e.dataTransfer.files;
            if (files.length > 0) {
                this.handleFile(files[0]);
            }
```

```
        }, false);
    }

    setupFileInput(fileInput) {
        fileInput.addEventListener('change', (e) => {
            if (e.target.files.length > 0) {
                this.handleFile(e.target.files[0]);
            }
        });
    }

    preventDefaults(e) {
        e.preventDefault();
        e.stopPropagation();
    }

    async handleFile(file) {
        try {
            this.resetUI();

            // 验证文件
            if (!this.validateFile(file)) {
                return;
            }

            // 显示文件信息
            this.showFileInfo(file);

            // 显示进度容器
            this.showProgressContainer('正在上传文件...');

            // 上传文件
            const formData = new FormData();
            formData.append('file', file);

            const response = await fetch('/upload', {
                method: 'POST',
                body: formData
            });

            const data = await response.json();

            if (data.error) {
                this.showError(data.error);
                return;
            }

            this.currentTaskId = data.task_id;
            this.showProgressContainer('文件上传成功，开始处理...');

            // 开始轮询状态
            this.startStatusPolling();
```

```
        } catch (error) {
            this.showError('上传失败: ' + error.message);
            console.error('File upload error:', error);
        }
    }

    validateFile(file) {
        const allowedExtensions = ['zip', 'tar', 'gz', 'bz2', 'rar', '7z', 'tar.gz',
'tar.bz2'];
        const fileExtension = file.name.split('.').pop().toLowerCase();
        const doubleExtension = file.name.split('.').slice(-2).join('.').toLowerCase();

        if (!allowedExtensions.includes(fileExtension) &&
!allowedExtensions.includes(doubleExtension)) {
            this.showError('不支持的文件格式。请上传ZIP、TAR、RAR或7Z格式的文件。');
            return false;
        }

        // 验证文件大小 (1GB = 1024MB)
        if (file.size > 1024 * 1024 * 1024) {
            this.showError('文件大小超过1GB限制。');
            return false;
        }

        return true;
    }

    showFileInfo(file) {
        const fileSizeMB = (file.size / (1024 * 1024)).toFixed(2);

        // 可以在这里添加显示文件信息的逻辑
        console.log(`文件信息: ${file.name}, 大小: ${fileSizeMB} MB`);
    }

    showProgressContainer(message) {
        const progressContainer = document.getElementById('progressContainer');
        const statusMessage = document.getElementById('statusMessage');

        if (progressContainer && statusMessage) {
            progressContainer.style.display = 'block';
            statusMessage.style.display = 'block';
            statusMessage.textContent = message;
        }
    }

    async startStatusPolling() {
        if (this.statusCheckInterval) {
            clearInterval(this.statusCheckInterval);
        }

        this.statusCheckInterval = setInterval(async () => {
            try {
                const response = await fetch(`/status/${this.currentTaskId}`);
```

```
                const data = await response.json();

                if (data.error) {
                    this.stopStatusPolling();
                    this.showError(data.error);
                    return;
                }

                this.updateProgress(data);

                if (data.status === '完成') {
                    this.stopStatusPolling();
                    this.showResult(data);
                } else if (data.error) {
                    this.stopStatusPolling();
                    this.showError(data.error);
                }

            } catch (error) {
                console.error('Status check error:', error);
            }
        }, 1000);
    }

    stopStatusPolling() {
        if (this.statusCheckInterval) {
            clearInterval(this.statusCheckInterval);
            this.statusCheckInterval = null;
        }
    }

    updateProgress(data) {
        const progressFill = document.getElementById('progressFill');
        const progressText = document.getElementById('progressText');
        const statusText = document.getElementById('statusText');
        const statusMessage = document.getElementById('statusMessage');

        if (progressFill) progressFill.style.width = data.progress + '%';
        if (progressText) progressText.textContent = data.progress + '%';
        if (statusText) statusText.textContent = data.current_step;
        if (statusMessage) statusMessage.textContent = data.status;
    }

    showResult(data) {
        const progressContainer = document.getElementById('progressContainer');
        const resultContainer = document.getElementById('resultContainer');
        const downloadBtn = document.getElementById('downloadBtn');

        if (progressContainer) progressContainer.style.display = 'none';
        if (resultContainer) resultContainer.style.display = 'block';

        if (downloadBtn && data.download_url) {
            downloadBtn.href = data.download_url;
```

```
            // 添加点击事件统计
            downloadBtn.addEventListener('click', () => {
                this.trackDownload(data.pdf_count || 1);
            });
        }

        // 显示成功消息
        this.showSuccessMessage(`成功生成 ${data.pdf_count || 1} 个PDF文件`);
    }

    showError(message) {
        const progressContainer = document.getElementById('progressContainer');
        const errorMessage = document.getElementById('errorMessage');

        if (progressContainer) progressContainer.style.display = 'none';
        if (errorMessage) {
            errorMessage.style.display = 'block';
            errorMessage.textContent = message;
        }
    }

    showSuccessMessage(message) {
        // 可以在这里添加成功消息的显示逻辑
        console.log('Success:', message);
    }

    resetUI() {
        const elements = {
            progressContainer: document.getElementById('progressContainer'),
            errorMessage: document.getElementById('errorMessage'),
            resultContainer: document.getElementById('resultContainer'),
            progressFill: document.getElementById('progressFill'),
            progressText: document.getElementById('progressText'),
            statusText: document.getElementById('statusText')
        };

        // 隐藏所有状态容器
        if (elements.progressContainer) elements.progressContainer.style.display =
'none';
        if (elements.errorMessage) elements.errorMessage.style.display = 'none';
        if (elements.resultContainer) elements.resultContainer.style.display = 'none';

        // 重置进度
        if (elements.progressFill) elements.progressFill.style.width = '0%';
        if (elements.progressText) elements.progressText.textContent = '0%';
        if (elements.statusText) elements.statusText.textContent = '等待开始';

        // 停止轮询
        this.stopStatusPolling();
        this.currentTaskId = null;
    }
```

```
    async cleanupFiles() {
        try {
            const response = await fetch('/cleanup', {
                method: 'POST'
            });

            const data = await response.json();

            if (data.error) {
                alert('清理失败: ' + data.error);
            } else {
                alert('临时文件清理完成');
            }
        } catch (error) {
            alert('清理请求失败: ' + error.message);
        }
    }

    trackDownload(pdfCount) {
        // 可以在这里添加下载统计逻辑
        console.log(`用户下载了 ${pdfCount} 个PDF文件`);
    }
}

// 工具函数
const Utils = {
    formatFileSize(bytes) {
        if (bytes === 0) return '0 Bytes';

        const k = 1024;
        const sizes = ['Bytes', 'KB', 'MB', 'GB'];
        const i = Math.floor(Math.log(bytes) / Math.log(k));

        return parseFloat((bytes / Math.pow(k, i)).toFixed(2)) + ' ' + sizes[i];
    },

    formatTime(seconds) {
        const minutes = Math.floor(seconds / 60);
        const remainingSeconds = seconds % 60;

        if (minutes > 0) {
            return `${minutes}分${remainingSeconds}秒`;
        } else {
            return `${remainingSeconds}秒`;
        }
    },

    debounce(func, wait) {
        let timeout;
        return function executedFunction(...args) {
            const later = () => {
                clearTimeout(timeout);
                func(...args);
```

```
            };
            clearTimeout(timeout);
            timeout = setTimeout(later, wait);
        };
    }
};

// 页面加载完成后初始化应用
document.addEventListener('DOMContentLoaded', () => {
    window.zipToPDFApp = new ZipToPDFApp();

    // 添加页面卸载时的清理
    window.addEventListener('beforeunload', () => {
        if (window.zipToPDFApp) {
            window.zipToPDFApp.stopStatusPolling();
        }
    });
});

// 导出供其他脚本使用
if (typeof module !== 'undefined' && module.exports) {
    module.exports = { ZipToPDFApp, Utils };
}
```

templates\indexhtml

to top

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>压缩包转PDF工具</title>
    <style>
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            min-height: 100vh;
            padding: 20px;
        }

        .container {
            max-width: 800px;
            margin: 0 auto;
            background: white;
            border-radius: 15px;
            box-shadow: 0 20px 40px rgba(0,0,0,0.1);
            overflow: hidden;
        }

        .header {
            background: linear-gradient(135deg, #4facfe 0%, #00f2fe 100%);
            color: white;
            padding: 30px;
            text-align: center;
        }

        .header h1 {
            font-size: 2.5em;
            margin-bottom: 10px;
            font-weight: 300;
        }

        .header p {
            font-size: 1.1em;
            opacity: 0.9;
        }

        .content {
            padding: 40px;
        }

        .upload-area {
```

```
            border: 3px dashed #4facfe;
            border-radius: 10px;
            padding: 40px;
            text-align: center;
            margin-bottom: 30px;
            transition: all 0.3s ease;
            background: #f8f9fa;
        }

        .upload-area:hover {
            border-color: #667eea;
            background: #e9ecef;
        }

        .upload-area.dragover {
            border-color: #667eea;
            background: #e3f2fd;
            transform: scale(1.02);
        }

        .upload-icon {
            font-size: 4em;
            color: #4facfe;
            margin-bottom: 20px;
        }

        .upload-text {
            font-size: 1.2em;
            color: #495057;
            margin-bottom: 15px;
        }

        .file-input {
            display: none;
        }

        .upload-btn {
            background: linear-gradient(135deg, #4facfe 0%, #00f2fe 100%);
            color: white;
            border: none;
            padding: 12px 30px;
            border-radius: 25px;
            font-size: 1.1em;
            cursor: pointer;
            transition: all 0.3s ease;
        }

        .upload-btn:hover {
            transform: translateY(-2px);
            box-shadow: 0 10px 20px rgba(79, 172, 254, 0.3);
        }

        .upload-btn:disabled {
```

```
            background: #6c757d;
            cursor: not-allowed;
            transform: none;
            box-shadow: none;
        }

        .progress-container {
            margin-top: 30px;
            display: none;
        }

        .progress-bar {
            width: 100%;
            height: 20px;
            background: #e9ecef;
            border-radius: 10px;
            overflow: hidden;
            margin-bottom: 10px;
        }

        .progress-fill {
            height: 100%;
            background: linear-gradient(135deg, #4facfe 0%, #00f2fe 100%);
            border-radius: 10px;
            transition: width 0.3s ease;
            width: 0%;
        }

        .progress-text {
            display: flex;
            justify-content: space-between;
            font-size: 0.9em;
            color: #6c757d;
        }

        .status-message {
            background: #e7f3ff;
            border-left: 4px solid #4facfe;
            padding: 15px;
            margin: 20px 0;
            border-radius: 5px;
            display: none;
        }

        .result-container {
            margin-top: 30px;
            display: none;
        }

        .download-btn {
            background: linear-gradient(135deg, #28a745 0%, #20c997 100%);
            color: white;
            border: none;
```

```
            padding: 15px 40px;
            border-radius: 25px;
            font-size: 1.2em;
            cursor: pointer;
            transition: all 0.3s ease;
            text-decoration: none;
            display: inline-block;
        }

        .download-btn:hover {
            transform: translateY(-2px);
            box-shadow: 0 10px 20px rgba(40, 167, 69, 0.3);
        }

        .pdf-list-container {
            margin-top: 30px;
            display: none;
        }

        .pdf-list {
            background: #f8f9fa;
            border-radius: 10px;
            padding: 20px;
            max-height: 300px;
            overflow-y: auto;
        }

        .pdf-item {
            display: flex;
            justify-content: space-between;
            align-items: center;
            padding: 12px 15px;
            background: white;
            border-radius: 8px;
            margin-bottom: 10px;
            border: 1px solid #e9ecef;
            transition: all 0.3s ease;
        }

        .pdf-item:hover {
            border-color: #4facfe;
            box-shadow: 0 2px 8px rgba(79, 172, 254, 0.1);
        }

        .pdf-info {
            flex: 1;
        }

        .pdf-name {
            font-weight: 500;
            color: #495057;
            margin-bottom: 5px;
        }
```

```
        .pdf-size {
            font-size: 0.9em;
            color: #6c757d;
        }

        .pdf-download-btn {
            background: linear-gradient(135deg, #4facfe 0%, #00f2fe 100%);
            color: white;
            border: none;
            padding: 8px 16px;
            border-radius: 20px;
            font-size: 0.9em;
            cursor: pointer;
            text-decoration: none;
            transition: all 0.3s ease;
        }

        .pdf-download-btn:hover {
            transform: translateY(-1px);
            box-shadow: 0 5px 15px rgba(79, 172, 254, 0.3);
        }

        .error-message {
            background: #f8d7da;
            border-left: 4px solid #dc3545;
            color: #721c24;
            padding: 15px;
            margin: 20px 0;
            border-radius: 5px;
            display: none;
        }

        .supported-formats {
            background: #f8f9fa;
            padding: 20px;
            border-radius: 10px;
            margin-top: 30px;
        }

        .formats-grid {
            display: grid;
            grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
            gap: 15px;
            margin-top: 15px;
        }

        .format-item {
            background: white;
            padding: 10px;
            border-radius: 5px;
            text-align: center;
            border: 1px solid #dee2e6;
```

```
        }

        .format-label {
            font-weight: bold;
            color: #495057;
        }

        .format-ext {
            color: #6c757d;
            font-size: 0.9em;
        }

        .loading {
            display: inline-block;
            width: 20px;
            height: 20px;
            border: 3px solid #f3f3f3;
            border-top: 3px solid #4facfe;
            border-radius: 50%;
            animation: spin 1s linear infinite;
        }

        @keyframes spin {
            0% { transform: rotate(0deg); }
            100% { transform: rotate(360deg); }
        }

        @media (max-width: 768px) {
            .container {
                margin: 10px;
            }

            .content {
                padding: 20px;
            }

            .header h1 {
                font-size: 2em;
            }

            .pdf-item {
                flex-direction: column;
                align-items: flex-start;
                gap: 10px;
            }

            .pdf-download-btn {
                align-self: flex-end;
            }
        }
    </style>
</head>
<body>
```

```
    <div class="container">
        <div class="header">
            <h1>压缩包转PDF工具</h1>
            <p>上传压缩包，自动解压并将图片转换为PDF</p>
        </div>

        <div class="content">
            <div class="upload-area" id="uploadArea">
                <div class="upload-icon">🗀</div>
                <div class="upload-text">拖拽文件到此处或点击选择文件</div>
                <p style="color: #6c757d; margin-bottom: 20px;">支持 ZIP, TAR, RAR, 7Z
等格式</p>
                <input type="file" id="fileInput" class="file-input"
accept=".zip,.tar,.gz,.bz2,.rar,.7z,.tar.gz,.tar.bz2">
                <button class="upload-btn"
onclick="document.getElementById('fileInput').click()">选择文件</button>
            </div>

            <div class="progress-container" id="progressContainer">
                <div class="progress-bar">
                    <div class="progress-fill" id="progressFill"></div>
                </div>
                <div class="progress-text">
                    <span id="progressText">0%</span>
                    <span id="statusText">等待开始</span>
                </div>
                <div class="status-message" id="statusMessage"></div>
            </div>

            <div class="error-message" id="errorMessage"></div>

            <div class="result-container" id="resultContainer">
                <div style="text-align: center;">
                    <h3 style="color: #28a745; margin-bottom: 20px;">处理完成！</h3>
                    <a href="#" class="download-btn" id="downloadBtn">下载PDF文件包</a>
                </div>

                <div class="pdf-list-container" id="pdfListContainer" style="margin-
top: 30px; display: none;">
                    <h4 style="color: #495057; margin-bottom: 15px;">生成的PDF文件：
</h4>
                    <div class="pdf-list" id="pdfList"></div>
                </div>
            </div>

            <div class="supported-formats">
                <h3>支持的文件格式</h3>
                <div class="formats-grid">
                    <div class="format-item">
                        <div class="format-label">压缩格式</div>
                        <div class="format-ext">ZIP, TAR, RAR, 7Z</div>
                    </div>
                    <div class="format-item">
```

```
                    <div class="format-label">图片格式</div>
                    <div class="format-ext">JPG, PNG, WEBP, GIF, BMP</div>
                </div>
                <div class="format-item">
                    <div class="format-label">最大文件</div>
                    <div class="format-ext">1 GB</div>
                </div>
                <div class="format-item">
                    <div class="format-label">输出格式</div>
                    <div class="format-ext">A4 PDF</div>
                </div>
            </div>
        </div>
    </div>
</div>

<script>
    let currentTaskId = null;
    let statusCheckInterval = null;

    // 获取DOM元素
    const uploadArea = document.getElementById('uploadArea');
    const fileInput = document.getElementById('fileInput');
    const progressContainer = document.getElementById('progressContainer');
    const progressFill = document.getElementById('progressFill');
    const progressText = document.getElementById('progressText');
    const statusText = document.getElementById('statusText');
    const statusMessage = document.getElementById('statusMessage');
    const errorMessage = document.getElementById('errorMessage');
    const resultContainer = document.getElementById('resultContainer');
    const downloadBtn = document.getElementById('downloadBtn');
    const pdfListContainer = document.getElementById('pdfListContainer');
    const pdfList = document.getElementById('pdfList');

    // 拖拽事件处理
    uploadArea.addEventListener('dragover', (e) => {
        e.preventDefault();
        uploadArea.classList.add('dragover');
    });

    uploadArea.addEventListener('dragleave', () => {
        uploadArea.classList.remove('dragover');
    });

    uploadArea.addEventListener('drop', (e) => {
        e.preventDefault();
        uploadArea.classList.remove('dragover');

        const files = e.dataTransfer.files;
        if (files.length > 0) {
            handleFile(files[0]);
        }
    });
```

```
        // 文件选择事件
        fileInput.addEventListener('change', (e) => {
            if (e.target.files.length > 0) {
                handleFile(e.target.files[0]);
            }
        });

        // 处理文件上传
        function handleFile(file) {
            // 重置状态
            resetUI();

            // 验证文件类型
            const allowedExtensions = ['zip', 'tar', 'gz', 'bz2', 'rar', '7z',
'tar.gz', 'tar.bz2'];
            const fileExtension = file.name.split('.').pop().toLowerCase();

            if (!allowedExtensions.includes(fileExtension) &&

!allowedExtensions.includes(file.name.split('.').slice(-2).join('.').toLowerCase())) {
                showError('不支持的文件格式。请上传ZIP、TAR、RAR或7Z格式的文件。');
                return;
            }

            // 验证文件大小 (1GB = 1024MB)
            if (file.size > 1024 * 1024 * 1024) {
                showError('文件大小超过1GB限制。');
                return;
            }

            // 显示进度容器
            progressContainer.style.display = 'block';
            statusMessage.style.display = 'block';
            statusMessage.textContent = '正在上传文件...';

            // 创建FormData并上传
            const formData = new FormData();
            formData.append('file', file);

            fetch('/upload', {
                method: 'POST',
                body: formData
            })
            .then(response => response.json())
            .then(data => {
                if (data.error) {
                    showError(data.error);
                    return;
                }

                currentTaskId = data.task_id;
                statusMessage.textContent = '文件上传成功，开始处理...';
```

```
            // 开始轮询状态
            startStatusPolling();
        })
        .catch(error => {
            showError('上传失败: ' + error.message);
        });
    }

    // 开始轮询处理状态
    function startStatusPolling() {
        statusCheckInterval = setInterval(() => {
            fetch(`/status/${currentTaskId}`)
                .then(response => response.json())
                .then(data => {
                    if (data.error) {
                        clearInterval(statusCheckInterval);
                        showError(data.error);
                        return;
                    }

                    // 更新进度
                    progressFill.style.width = data.progress + '%';
                    progressText.textContent = data.progress + '%';
                    statusText.textContent = data.current_step;
                    statusMessage.textContent = data.status;

                    // 检查是否完成
                    if (data.status === '处理完成') {
                        clearInterval(statusCheckInterval);
                        showResult(data);
                    } else if (data.error) {
                        clearInterval(statusCheckInterval);
                        showError(data.error);
                    }
                })
                .catch(error => {
                    console.error('状态检查失败:', error);
                });
        }, 1000); // 每秒检查一次
    }

    // 显示结果
    function showResult(data) {
        progressContainer.style.display = 'none';
        resultContainer.style.display = 'block';

        if (data.download_url) {
            downloadBtn.href = data.download_url;
        }

        // 如果有PDF列表URL，加载PDF文件列表
        if (data.pdf_list_url) {
```

```
            loadPdfList(data.pdf_list_url);
        }
    }

    // 加载PDF文件列表
    function loadPdfList(pdfListUrl) {
        fetch(pdfListUrl)
            .then(response => response.json())
            .then(data => {
                if (data.error) {
                    console.error('加载PDF列表失败:', data.error);
                    return;
                }

                if (data.pdf_files && data.pdf_files.length > 0) {
                    displayPdfList(data.pdf_files);
                }
            })
            .catch(error => {
                console.error('加载PDF列表失败:', error);
            });
    }

    // 显示PDF文件列表
    function displayPdfList(pdfFiles) {
        pdfList.innerHTML = '';

        pdfFiles.forEach(pdfFile => {
            const pdfItem = document.createElement('div');
            pdfItem.className = 'pdf-item';

            const pdfInfo = document.createElement('div');
            pdfInfo.className = 'pdf-info';

            const pdfName = document.createElement('div');
            pdfName.className = 'pdf-name';
            pdfName.textContent = pdfFile.filename;

            const pdfSize = document.createElement('div');
            pdfSize.className = 'pdf-size';
            pdfSize.textContent = formatFileSize(pdfFile.size);

            pdfInfo.appendChild(pdfName);
            pdfInfo.appendChild(pdfSize);

            const downloadBtn = document.createElement('a');
            downloadBtn.className = 'pdf-download-btn';
            downloadBtn.href = pdfFile.download_url;
            downloadBtn.textContent = '下载';
            downloadBtn.target = '_blank';

            pdfItem.appendChild(pdfInfo);
            pdfItem.appendChild(downloadBtn);
```

```
                pdfList.appendChild(pdfItem);
            });

            pdfListContainer.style.display = 'block';
        }

        // 格式化文件大小
        function formatFileSize(bytes) {
            if (bytes === 0) return '0 Bytes';
            const k = 1024;
            const sizes = ['Bytes', 'KB', 'MB', 'GB'];
            const i = Math.floor(Math.log(bytes) / Math.log(k));
            return parseFloat((bytes / Math.pow(k, i)).toFixed(2)) + ' ' + sizes[i];
        }

        // 显示错误信息
        function showError(message) {
            progressContainer.style.display = 'none';
            errorMessage.style.display = 'block';
            errorMessage.textContent = message;
        }

        // 重置UI状态
        function resetUI() {
            progressContainer.style.display = 'none';
            errorMessage.style.display = 'none';
            resultContainer.style.display = 'none';
            pdfListContainer.style.display = 'none';
            progressFill.style.width = '0%';
            progressText.textContent = '0%';
            statusText.textContent = '等待开始';

            if (statusCheckInterval) {
                clearInterval(statusCheckInterval);
            }
        }

        // 页面加载时重置状态
        window.addEventListener('load', resetUI);
    </script>
</body>
</html>
```

utils\compressionpy

to top

```python
import os
import zipfile
import tarfile
import rarfile
import py7zr
import tempfile
from pathlib import Path
from utils.file_utils import FileUtils

class CompressionHandler:
    """压缩包处理类"""

    def __init__(self):
        self.extracted_files = []
        self.status_callback = None

    def set_status_callback(self, callback):
        """设置状态回调函数"""
        self.status_callback = callback

    def _update_status(self, message, progress=None):
        """更新处理状态"""
        if self.status_callback:
            self.status_callback(message, progress)

    def recursive_extract(self, file_path, extract_to, depth=0, max_depth=10):
        """
        递归解压压缩包

        Args:
            file_path: 压缩包文件路径
            extract_to: 解压目标目录
            depth: 当前递归深度
            max_depth: 最大递归深度

        Returns:
            list: 所有解压出的文件路径列表
        """
        if depth > max_depth:
            self._update_status(f"达到最大递归深度 {max_depth}，停止解压")
            return []

        self._update_status(f"开始解压: {os.path.basename(file_path)}")

        extracted_files = []

        try:
            # 根据文件类型选择解压方法
            if zipfile.is_zipfile(file_path):
                extracted_files.extend(self._extract_zip(file_path, extract_to))
            elif tarfile.is_tarfile(file_path):
                extracted_files.extend(self._extract_tar(file_path, extract_to))
```

```
            elif file_path.lower().endswith('.rar'):
                extracted_files.extend(self._extract_rar(file_path, extract_to))
            elif file_path.lower().endswith('.7z'):
                extracted_files.extend(self._extract_7z(file_path, extract_to))
            else:
                self._update_status(f"不支持的压缩格式: {file_path}")
                return []

            # 检查解压出的文件是否也是压缩包
            for extracted_file in extracted_files[:]:  # 使用副本进行迭代
                if FileUtils.is_compressed_file(extracted_file):
                    self._update_status(f"发现嵌套压缩包:
{os.path.basename(extracted_file)}")

                    # 创建子目录用于解压嵌套压缩包
                    nested_extract_to = os.path.join(
                        extract_to,
                        f"nested_{Path(extracted_file).stem}"
                    )
                    os.makedirs(nested_extract_to, exist_ok=True)

                    # 递归解压
                    nested_files = self.recursive_extract(
                        extracted_file, nested_extract_to, depth + 1, max_depth
                    )
                    extracted_files.extend(nested_files)

                    # 删除已解压的嵌套压缩包文件
                    try:
                        os.remove(extracted_file)
                    except:
                        pass

            self._update_status(f"解压完成: {os.path.basename(file_path)}",
progress=100)
            return extracted_files

        except Exception as e:
            self._update_status(f"解压失败 {file_path}: {str(e)}")
            return []

    def _extract_zip(self, file_path, extract_to):
        """解压ZIP文件"""
        extracted_files = []
        try:
            with zipfile.ZipFile(file_path, 'r') as zip_ref:
                file_list = zip_ref.namelist()
                total_files = len(file_list)

                for i, file_info in enumerate(file_list):
                    # 跳过目录
                    if file_info.endswith('/'):
                        continue
```

```
                    # 解压文件
                    zip_ref.extract(file_info, extract_to)
                    extracted_path = os.path.join(extract_to, file_info)
                    extracted_files.append(extracted_path)

                    # 更新进度
                    progress = (i + 1) / total_files * 100
                    self._update_status(f"解压ZIP文件: {file_info}", progress)

            return extracted_files
        except Exception as e:
            raise Exception(f"ZIP解压失败: {str(e)}")

    def _extract_tar(self, file_path, extract_to):
        """解压TAR文件（包括.tar.gz, .tar.bz2）"""
        extracted_files = []
        try:
            mode = 'r'
            if file_path.endswith('.gz'):
                mode = 'r:gz'
            elif file_path.endswith('.bz2'):
                mode = 'r:bz2'

            with tarfile.open(file_path, mode) as tar_ref:
                members = tar_ref.getmembers()
                total_files = len(members)

                for i, member in enumerate(members):
                    # 跳过目录
                    if member.isdir():
                        continue

                    # 解压文件
                    tar_ref.extract(member, extract_to)
                    extracted_path = os.path.join(extract_to, member.name)
                    extracted_files.append(extracted_path)

                    # 更新进度
                    progress = (i + 1) / total_files * 100
                    self._update_status(f"解压TAR文件: {member.name}", progress)

            return extracted_files
        except Exception as e:
            raise Exception(f"TAR解压失败: {str(e)}")

    def _extract_rar(self, file_path, extract_to):
        """解压RAR文件"""
        extracted_files = []
        try:
            with rarfile.RarFile(file_path) as rar_ref:
                file_list = rar_ref.namelist()
                total_files = len(file_list)
```

```
                for i, file_info in enumerate(file_list):
                    # 跳过目录
                    if file_info.endswith('/') or file_info.endswith('\\'):
                        continue

                    # 解压文件
                    rar_ref.extract(file_info, extract_to)
                    extracted_path = os.path.join(extract_to, file_info)
                    extracted_files.append(extracted_path)

                    # 更新进度
                    progress = (i + 1) / total_files * 100
                    self._update_status(f"解压RAR文件: {file_info}", progress)

            return extracted_files
        except Exception as e:
            raise Exception(f"RAR解压失败: {str(e)}")

    def _extract_7z(self, file_path, extract_to):
        """解压7z文件"""
        extracted_files = []
        try:
            with py7zr.SevenZipFile(file_path, mode='r') as seven_zip_ref:
                file_list = seven_zip_ref.getnames()
                total_files = len(file_list)

                for i, file_info in enumerate(file_list):
                    # 解压文件
                    seven_zip_ref.extract(targets=[file_info], path=extract_to)
                    extracted_path = os.path.join(extract_to, file_info)
                    extracted_files.append(extracted_path)

                    # 更新进度
                    progress = (i + 1) / total_files * 100
                    self._update_status(f"解压7z文件: {file_info}", progress)

            return extracted_files
        except Exception as e:
            raise Exception(f"7z解压失败: {str(e)}")
```

utils\file_utilspy

to top

```python
import os
import shutil
import mimetypes
from pathlib import Path
from config import config

class FileUtils:
    """文件处理工具类"""

    @staticmethod
    def allowed_file(filename, allowed_extensions):
        """检查文件扩展名是否允许"""
        return '.' in filename and \
               filename.rsplit('.', 1)[1].lower() in allowed_extensions

    @staticmethod
    def get_file_type(file_path):
        """使用mimetypes检测文件类型"""
        try:
            # 使用mimetypes库检测文件类型
            file_type, encoding = mimetypes.guess_type(file_path)
            return file_type
        except Exception as e:
            print(f"检测文件类型失败: {e}")
            return None

    @staticmethod
    def is_compressed_file(file_path):
        """检查是否为压缩文件"""
        file_type = FileUtils.get_file_type(file_path)
        if file_type:
            return any(compressed_type in file_type for compressed_type in [
                'application/zip',
                'application/x-tar',
                'application/gzip',
                'application/x-bzip2',
                'application/x-rar',
                'application/x-7z-compressed'
            ])
        return False

    @staticmethod
    def is_image_file(file_path):
        """检查是否为图片文件"""
        file_type = FileUtils.get_file_type(file_path)
        if file_type:
            return file_type.startswith('image/')
        # 备用检查：通过文件扩展名
        ext = Path(file_path).suffix.lower()[1:]
        return ext in config['default'].ALLOWED_IMAGE_EXTENSIONS

    @staticmethod
```

```python
    def natural_sort_key(s):
        """自然排序键函数"""
        import re
        return [int(text) if text.isdigit() else text.lower()
                for text in re.split(r'(\d+)', str(s))]

    @staticmethod
    def create_directories():
        """创建必要的目录"""
        directories = [
            config['default'].UPLOAD_FOLDER,
            config['default'].TEMP_FOLDER,
            config['default'].OUTPUT_FOLDER
        ]

        for directory in directories:
            os.makedirs(directory, exist_ok=True)

    @staticmethod
    def cleanup_old_files(directory, hours_old=24):
        """清理指定目录中超过指定时间的文件"""
        import time
        current_time = time.time()

        for filename in os.listdir(directory):
            file_path = os.path.join(directory, filename)
            if os.path.isfile(file_path):
                file_time = os.path.getmtime(file_path)
                if current_time - file_time > hours_old * 3600:
                    try:
                        os.remove(file_path)
                        print(f"已清理文件: {file_path}")
                    except Exception as e:
                        print(f"清理文件失败 {file_path}: {e}")

    @staticmethod
    def get_file_size(file_path):
        """获取文件大小（MB）"""
        try:
            size_bytes = os.path.getsize(file_path)
            return size_bytes / (1024 * 1024)  # 转换为MB
        except OSError:
            return 0

    @staticmethod
    def safe_remove(path):
        """安全删除文件或目录"""
        try:
            if os.path.isfile(path):
                os.remove(path)
                print(f"已删除文件: {path}")
            elif os.path.isdir(path):
                shutil.rmtree(path)
```

```
                print(f"已删除目录: {path}")
        except Exception as e:
            print(f"删除失败 {path}: {e}")

    @staticmethod
    def cleanup_task_files(task_id, upload_folder, temp_folder, output_folder):
        """清理指定任务的所有相关文件"""
        print(f"开始清理任务 {task_id} 的文件...")

        # 清理上传文件
        for filename in os.listdir(upload_folder):
            if filename.startswith(task_id):
                file_path = os.path.join(upload_folder, filename)
                FileUtils.safe_remove(file_path)

        # 清理临时目录
        temp_dir = os.path.join(temp_folder, f"temp_{task_id}")
        if os.path.exists(temp_dir):
            FileUtils.safe_remove(temp_dir)

        # 清理PDF输出目录
        pdf_dir = os.path.join(output_folder, f"pdfs_{task_id}")
        if os.path.exists(pdf_dir):
            FileUtils.safe_remove(pdf_dir)

        # 清理ZIP输出文件
        zip_file = os.path.join(output_folder, f"result_{task_id}.zip")
        if os.path.exists(zip_file):
            FileUtils.safe_remove(zip_file)

        print(f"任务 {task_id} 文件清理完成")

    @staticmethod
    def cleanup_all_temp_files():
        """清理所有临时文件"""
        print("开始清理所有临时文件...")

        config_obj = config['default']

        # 清理上传文件夹
        for filename in os.listdir(config_obj.UPLOAD_FOLDER):
            file_path = os.path.join(config_obj.UPLOAD_FOLDER, filename)
            FileUtils.safe_remove(file_path)

        # 清理临时文件夹
        for item in os.listdir(config_obj.TEMP_FOLDER):
            item_path = os.path.join(config_obj.TEMP_FOLDER, item)
            FileUtils.safe_remove(item_path)

        print("所有临时文件清理完成")
```

utils\image_processorpy

to top

```python
import os
from pathlib import Path
from PIL import Image
from utils.file_utils import FileUtils

class ImageProcessor:
    """图片处理类"""

    def __init__(self):
        self.status_callback = None

    def set_status_callback(self, callback):
        """设置状态回调函数"""
        self.status_callback = callback

    def _update_status(self, message, progress=None):
        """更新处理状态"""
        if self.status_callback:
            self.status_callback(message, progress)

    def collect_and_sort_images(self, root_dir):
        """
        收集并排序图片文件，按文件夹分组

        Args:
            root_dir: 根目录路径

        Returns:
            dict: 按文件夹分组的图片路径字典 {folder_path: [sorted_image_paths]}
        """
        self._update_status("开始收集图片文件...")

        image_groups = {}

        # 遍历目录结构
        for root, dirs, files in os.walk(root_dir):
            image_files = []

            # 过滤图片文件
            for file in files:
                file_path = os.path.join(root, file)
                if FileUtils.is_image_file(file_path):
                    image_files.append(file_path)

            # 如果有图片文件，按自然顺序排序
            if image_files:
                # 使用自然排序
                image_files.sort(key=FileUtils.natural_sort_key)
                image_groups[root] = image_files

        self._update_status(f"找到 {len(image_groups)} 个包含图片的文件夹")
        return image_groups
```

```
    def convert_to_supported_format(self, image_path, output_dir):
        """
        将图片转换为PDF支持的格式

        Args:
            image_path: 原始图片路径
            output_dir: 输出目录

        Returns:
            str: 转换后的图片路径
        """
        try:
            # 打开图片
            with Image.open(image_path) as img:
                # 获取文件信息
                filename = Path(image_path).stem
                output_path = os.path.join(output_dir, f"{filename}.png")

                # 如果已经是PNG格式且不需要转换，直接返回原路径
                if image_path.lower().endswith('.png'):
                    return image_path

                # 转换图片格式为PNG（PDF生成最兼容的格式）
                if img.mode in ('P', 'RGBA'):
                    # 处理带透明度的图片
                    img = img.convert('RGBA')
                else:
                    img = img.convert('RGB')

                # 保存为PNG格式
                img.save(output_path, 'PNG', optimize=True)

                return output_path

        except Exception as e:
            self._update_status(f"图片转换失败 {image_path}: {str(e)}")
            return None

    def optimize_image_for_pdf(self, image_path, max_size=(2480, 3508)):
        """
        优化图片以适应PDF页面（A4尺寸）

        Args:
            image_path: 图片路径
            max_size: 最大尺寸 (宽, 高)，默认A4尺寸(2480x3508像素)

        Returns:
            str: 优化后的图片路径
        """
        try:
            with Image.open(image_path) as img:
                # 获取原始尺寸
```

```
                original_width, original_height = img.size

                # 计算缩放比例
                width_ratio = max_size[0] / original_width
                height_ratio = max_size[1] / original_height
                scale_ratio = min(width_ratio, height_ratio, 1.0)  # 不超过原始尺寸

                # 如果不需要缩放，直接返回
                if scale_ratio >= 1.0:
                    return image_path

                # 计算新尺寸
                new_width = int(original_width * scale_ratio)
                new_height = int(original_height * scale_ratio)

                # 调整图片大小
                resized_img = img.resize((new_width, new_height),
Image.Resampling.LANCZOS)

                # 保存优化后的图片（覆盖原文件）
                resized_img.save(image_path, optimize=True)

                self._update_status(f"优化图片尺寸: {original_width}x{original_height} -
> {new_width}x{new_height}")
                return image_path

        except Exception as e:
            self._update_status(f"图片优化失败 {image_path}: {str(e)}")
            return image_path  # 返回原路径，不中断流程

    def process_image_group(self, image_group, output_dir):
        """
        处理一组图片，进行格式转换和优化

        Args:
            image_group: 图片路径列表
            output_dir: 输出目录

        Returns:
            list: 处理后的图片路径列表
        """
        processed_images = []
        total_images = len(image_group)

        for i, image_path in enumerate(image_group):
            # 更新进度
            progress = (i + 1) / total_images * 100
            self._update_status(f"处理图片: {Path(image_path).name}", progress)

            # 转换图片格式
            converted_path = self.convert_to_supported_format(image_path, output_dir)
            if converted_path:
                # 优化图片尺寸
```

```
                optimized_path = self.optimize_image_for_pdf(converted_path)
                processed_images.append(optimized_path)
            else:
                # 如果转换失败，尝试直接使用原图
                self._update_status(f"使用原图: {Path(image_path).name}")
                processed_images.append(image_path)

        return processed_images

    def get_image_info(self, image_path):
        """
        获取图片信息

        Args:
            image_path: 图片路径

        Returns:
            dict: 图片信息字典
        """
        try:
            with Image.open(image_path) as img:
                return {
                    'format': img.format,
                    'size': img.size,
                    'mode': img.mode,
                    'filename': Path(image_path).name
                }
        except Exception as e:
            return {
                'format': 'Unknown',
                'size': (0, 0),
                'mode': 'Unknown',
                'filename': Path(image_path).name,
                'error': str(e)
            }
```

utils\pdf_generatorpy

to top

```python
import os
import img2pdf
from pathlib import Path
from utils.file_utils import FileUtils

class PDFGenerator:
    """PDF生成类"""

    def __init__(self):
        self.status_callback = None

    def set_status_callback(self, callback):
        """设置状态回调函数"""
        self.status_callback = callback

    def _update_status(self, message, progress=None):
        """更新处理状态"""
        if self.status_callback:
            self.status_callback(message, progress)

    def generate_pdf_from_images(self, image_paths, output_pdf_path, page_size='A4'):
        """
        从图片列表生成PDF

        Args:
            image_paths: 图片路径列表
            output_pdf_path: 输出PDF路径
            page_size: 页面尺寸

        Returns:
            bool: 是否成功生成
        """
        try:
            if not image_paths:
                self._update_status("没有图片可生成PDF")
                return False

            self._update_status(f"开始生成PDF: {Path(output_pdf_path).name}")

            # 验证所有图片文件都存在
            valid_images = []
            for img_path in image_paths:
                if os.path.exists(img_path):
                    valid_images.append(img_path)
                else:
                    self._update_status(f"图片文件不存在: {img_path}")

            if not valid_images:
                self._update_status("没有有效的图片文件")
                return False

            # 设置PDF页面参数
```

```
            pdf_layout_fun = self._get_pdf_layout_function(page_size)

            # 生成PDF
            with open(output_pdf_path, "wb") as pdf_file:
                pdf_file.write(img2pdf.convert(
                    valid_images,
                    layout_fun=pdf_layout_fun
                ))

            # 验证生成的PDF文件
            if os.path.exists(output_pdf_path) and os.path.getsize(output_pdf_path) >
0:
                self._update_status(f"PDF生成成功: {Path(output_pdf_path).name}")
                return True
            else:
                self._update_status("PDF文件生成失败")
                return False

        except Exception as e:
            self._update_status(f"PDF生成失败: {str(e)}")
            return False

    def _get_pdf_layout_function(self, page_size):
        """获取PDF布局函数"""
        if page_size.upper() == 'A4':
            # A4尺寸: 210mm x 297mm
            return img2pdf.get_layout_fun((img2pdf.mm_to_pt(210),
img2pdf.mm_to_pt(297)))
        elif page_size.upper() == 'LETTER':
            # Letter尺寸: 215.9mm x 279.4mm
            return img2pdf.get_layout_fun((img2pdf.mm_to_pt(215.9),
img2pdf.mm_to_pt(279.4)))
        else:
            # 默认使用A4
            return img2pdf.get_layout_fun((img2pdf.mm_to_pt(210),
img2pdf.mm_to_pt(297)))

    def generate_pdfs_by_folder(self, image_groups, output_dir, base_name="output"):
        """
        按文件夹分组生成多个PDF

        Args:
            image_groups: 按文件夹分组的图片字典 {folder_path: [image_paths]}
            output_dir: 输出目录
            base_name: 基础文件名

        Returns:
            dict: 生成的PDF文件路径字典 {folder_name: pdf_path}
        """
        generated_pdfs = {}
        total_groups = len(image_groups)

        for i, (folder_path, image_paths) in enumerate(image_groups.items()):
```

```
            # 更新进度
            progress = (i + 1) / total_groups * 100
            folder_name = Path(folder_path).name or "root"
            self._update_status(f"生成PDF: {folder_name}", progress)

            # 生成PDF文件名
            pdf_filename = f"{base_name}_{folder_name}.pdf"
            pdf_path = os.path.join(output_dir, pdf_filename)

            # 生成PDF
            if self.generate_pdf_from_images(image_paths, pdf_path):
                generated_pdfs[folder_name] = pdf_path

        return generated_pdfs

    def create_pdf_package(self, pdf_files, output_zip_path):
        """
        将多个PDF文件打包成ZIP文件

        Args:
            pdf_files: PDF文件路径列表
            output_zip_path: 输出ZIP文件路径

        Returns:
            bool: 是否成功打包
        """
        import zipfile

        try:
            self._update_status("正在打包PDF文件...")

            with zipfile.ZipFile(output_zip_path, 'w', zipfile.ZIP_DEFLATED) as zipf:
                for pdf_path in pdf_files:
                    if os.path.exists(pdf_path):
                        # 使用相对路径存储
                        arcname = Path(pdf_path).name
                        zipf.write(pdf_path, arcname)

            # 验证生成的ZIP文件
            if os.path.exists(output_zip_path) and os.path.getsize(output_zip_path) >
0:
                self._update_status(f"PDF打包成功: {Path(output_zip_path).name}")
                return True
            else:
                self._update_status("PDF打包失败")
                return False

        except Exception as e:
            self._update_status(f"PDF打包失败: {str(e)}")
            return False

    def get_pdf_info(self, pdf_path):
        """
```

```
        获取PDF文件信息

        Args:
            pdf_path: PDF文件路径

        Returns:
            dict: PDF信息字典
        """
        try:
            file_size = os.path.getsize(pdf_path)
            file_size_mb = file_size / (1024 * 1024)

            return {
                'filename': Path(pdf_path).name,
                'file_size': file_size,
                'file_size_mb': round(file_size_mb, 2),
                'created_time': os.path.getctime(pdf_path)
            }
        except Exception as e:
            return {
                'filename': Path(pdf_path).name,
                'error': str(e)
            }
```

utils__init__py

to top

```
# 工具模块包
```