外挂知识库系统

- 在 AI（特别是大语言模型，LLM）领域，**外挂知识库**（External Knowledge Base）指的是 **模型之外的知识存储**。
- 这类系统会把 **用户文档、业务数据、FAQ 等** 单独存放在数据库或向量库里。
- AI 在回答问题时，会先从这个知识库里检索相关信息，再结合模型本身的能力生成答案。
- 常见名字：**RAG（Retrieval-Augmented Generation，检索增强生成）**。
-

文档先存入数据库/向量库，再根据问题检索相关片段，传给 AI → 这就是典型的 **外挂知识库系统**。

- 优点：可扩展，文档再多也能高效检索。
- 缺点：需要额外搭建数据库和检索逻辑。

存入数据库 + 检索 + AI 生成 → 才是 **标准外挂知识库系统**。

如果先做 **向量化 + 检索（只喂相关片段）**，再传给 AI，就是完整的外挂知识库系统。

外挂知识库系统，就是在大模型（如 ChatGPT、DeepSeek、Claude 等）的原始知识之外，**外挂一套外部资料**，让 AI 在回答问题时能引用这些资料，生成更准确、更定制化的回答。

它常被称为 **RAG（Retrieval-Augmented Generation，检索增强生成）**。

基本原理

1. **准备知识库**
    - 把文档（PDF、网页、数据库里的资料等）收集起来。
    - 通常会切片、向量化，存进向量数据库。
2. **接收用户问题**
    - 用户输入问题，比如“合同第 5 条规定了什么？”
3. **检索相关内容**
    - 从知识库里找出最相关的文档片段。
4. **拼接上下文**
    - 把【用户问题 + 检索到的文档片段】一起传给 AI。
5. **生成回答**
    - AI 结合外部知识，给出最终答案。

特点

- **外挂**：知识不在 AI 模型内部，而是你额外挂上去的。
- **可更新**：只要更新知识库，不需要重新训练模型。
- **可控**：可以限定 AI 只能依据你提供的资料回答。

（做向量化 + 检索）

这是比较标准的 **RAG（检索增强生成）** 模式。

- **流程**
    1. **文档预处理**：切片（按段落、句子分割）。
    2. **向量化**：用 Embedding 模型把每个片段转为向量。
    3. **存储**：放进向量数据库（如 FAISS、Milvus、Weaviate、Pinecone）。
    4. **检索**：用户提问时，把问题向量化，在数据库里找最相似的片段。
    5. **拼接 Prompt**：把问题 + 检索到的片段一起传给 AI。

6. **AI 回答**。

- **优点**：适合大规模文档，速度快，回答更精准。
- **缺点**：需要多用一套数据库和 embedding 模型。

区别总结表

方面	文本化	文本向量化
输入	文档、图片、网页等	文本（纯文字）
输出	字符串文本	数值向量
目的	获取文字信息	表示语义、方便计算机处理
示例	PDF → “这是文章内容”	“这是文章内容” → [0.12, -0.33, ...]
应用	数据清洗、存档	相似度搜索、问答系统、推荐系统

文本化 → 拿到文字

文本向量化 → 把文字变成数字向量让 AI 能“理解”
从 PDF → 文本 → 向量 → AI 处理。

这个脚本批量把 `pdf_folder` 下的每个 PDF 转成纯文本（`.txt`）。先尝试用 `pdfplumber` 提取文本层，若提取不到（或为空），就把 PDF 每页转成图片然后用 `pytesseract` 做 OCR。并行处理多个 PDF（`ThreadPoolExecutor`）并用 `tqdm` 显示进度条。

关键库说明

-

`os`：文件路径和目录操作。

-
-

`pdfplumber`：从 PDF 中提取“文本层”（对于可复制文本的 PDF 最好）。

-
-

`pdf2image.convert_from_path`：把 PDF 页转换为 PIL 图片（需要安装 Poppler 二进制）。

-
-

pytesseract：Tesseract OCR 的 Python 封装（需要安装 Tesseract 可执行程序和语言包）。

-
-

concurrent.futures.ThreadPoolExecutor、as_completed：并行提交任务并获取完成结果。

-
-

tqdm：漂亮的命令行进度条。

-

配置区解释

```
pdf_folder = r"D:\pdf"            # 待处理 PDF 所在文件夹（原始字符串 r"..." 可避免反斜杠转义）
output_folder = r"D:\py-output"   # 输出 TXT 的目录
ocr_lang = "chi_sim+eng"          # OCR 使用的语言（Tesseract 的语言包名，多个用 + 连接）
max_workers = 4                   # 并行处理的最大线程数（建议 = CPU 核心数）
```

os.makedirs(output_folder, exist_ok=True)：确保输出目录存在（若不存在则创建）。

---

process_pdf(pdf_path) 函数详解

这个函数负责处理单个 PDF，并返回一个描述性字符串（成功或错误信息）。

1.

文件名与输出路径

2.

```
filename = os.path.basename(pdf_path)
txt_filename = os.path.splitext(filename)[0] + "_output.txt"
txt_path = os.path.join(output_folder, txt_filename)
```

生成输出文件名，例如 demo.pdf → demo_output.txt。

1.

尝试用 pdfplumber 提取文本层

2.

```
full_text = ""try:
    with pdfplumber.open(pdf_path) as pdf:
        for page in pdf.pages:
            text = page.extract_text()
            if text:
                full_text += text + "\n"except Exception as e:
    return f"[错误] pdfplumber 处理 {filename} 失败: {e}"
```

- page.extract_text()：返回该页的字符串，可能为 None（比如 PDF 是扫描图像而非嵌入文本）。

-
-

注意：脚本在 pdfplumber 出现任何异常时直接 return，这会跳过后面的 OCR 步骤 —— 如果你想在 pdfplumber 失败时仍尝试 OCR，需要把异常处理改成记录错误但继续执行 OCR（见改进建议）。

-

1.

如果没有文本（空字符串），使用 OCR

2.

```
if not full_text.strip():
    try:
        pages = convert_from_path(pdf_path)
        for page in pages:
            text = pytesseract.image_to_string(page, lang=ocr_lang)
            full_text += text + "\n"
    except Exception as e:
        return f"[错误] OCR 处理 {filename} 失败: {e}"
```

-

convert_from_path 把 PDF 每页变成 PIL Image 的列表（**可能占大量内存**，尤其是多页大 PDF）。

-
-

pytesseract.image_to_string(page, lang=ocr_lang)：对图片执行 OCR。

-
-

ocr_lang 要确保安装了对应的 Tesseract 语言包（chi_sim、eng 等）。

-

1.

写入输出文件

2.

```
with open(txt_path, "w", encoding="utf-8") as f:
    f.write(full_text)return f"[完成] {filename} → {txt_filename}"
```

-

使用 utf-8 写入，保证中文正确保存。

-

---

main() 函数详解（控制流程与并发）

```
pdf_files = [
    os.path.join(pdf_folder, f)
    for f in os.listdir(pdf_folder)
    if f.lower().endswith(".pdf")
]
```

从目录读取所有 .pdf 文件（注意：os.listdir 只列基本名，不递归子目录）。

若没有 PDF 则打印并退出。

并发部分：

```
with ThreadPoolExecutor(max_workers=max_workers) as executor:
    future_to_pdf = {executor.submit(process_pdf, pdf): pdf for pdf in pdf_files}
    for future in tqdm(as_completed(future_to_pdf), total=len(pdf_files), desc="处理进度"):
        results.append(future.result())
```

- executor.submit(process_pdf, pdf)：异步提交任务，返回 Future。
- as_completed(...)：当某个任务完成就产生一个 future（返回顺序是任务完成顺序，而非原始文件顺序）。
- tqdm(...)：把 as_completed 包装为带进度条的迭代器，total=len(pdf_files) 用来显示总进度。
- 最后打印 results，其中每项是 process_pdf 返回的字符串（完成或错误信息）。

入口：

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

常见的 Python 入口保护，方便脚本直接运行或被导入时不自动执行。

常见问题与注意事项（运行前必读）

1. **外部依赖：**
2.

- Tesseract 可执行程序必须安装（Windows 上通常在 C:\Program Files\Tesseract-OCR\tesseract.exe），并且要安装 chi_sim、eng 等语言包。
- pdf2image 需要 poppler（Windows 上需下载 Poppler 并指定 poppler_path）。
- Python 包：pip install pdfplumber pdf2image pytesseract pillow tqdm。

3. **路径设置示例（Windows）：**

4.

```
import pytesseract
pytesseract.pytesseract.tesseract_cmd = r"C:\Program
Files\Tesseract-OCR\tesseract.exe"

pages = convert_from_path(pdf_path, dpi=200,
poppler_path=r"C:\path\to\poppler\bin")
```

1. **内存/性能：**convert_from_path 会一次把所有页图片加载到内存，若 PDF 很大会爆内存。可以改为按页转换（page_numbers=[i] 或 first_page/last_page）逐页处理。
2.
3. **并发模型：**OCR 是 CPU 密集型（调用本地 C 程序），用 ThreadPoolExecutor 可能因 GIL 或 Python 层调度不是最优。建议：

4.

-

  如果 OCR 调用会释放 GIL（调用外部进程），ThreadPoolExecutor 可能表现良好；

  -
  -

  若发现 CPU 利用率低或速度不理想，改用 ProcessPoolExecutor（多进程）通常更适合 CPU 密集型任务。

  -

5.

错误处理：目前如果 pdfplumber.open 抛异常，函数就 return 并跳过 OCR。更稳健的方式是在 pdfplumber 抛异常时记录错误但继续尝试 OCR。

6.
7.

输出顺序：results 的顺序是任务完成的顺序，不是原始文件顺序。如果你需要按原始文件顺序输出，可以在收集时把 future_to_pdf 的映射用起来排序或等所有完成后按 pdf_files 顺序读取结果。

8.
9.

OCR 精度：可以通过 pytesseract.image_to_string(image, lang=..., config='--psm 6') 等参数微调（--psm、--oem 等）。

10.
11.

跳过已处理文件：如果输出文件已存在且你不想重复处理，可以在 main() 里过滤掉已经存在的 _output.txt。

12.

推荐的简单改进示例（更稳健的处理 pdfplumber 与 OCR 回退）

```
# （示例片段：只展示逻辑）try:
    with pdfplumber.open(pdf_path) as pdf:
        for page in pdf.pages:
```

```
            text = page.extract_text() or ""
            full_text += text + "\n"except Exception as e:
    # 记录但不直接返回，让 OCR 继续作为回退
    print(f"pdfplumber 出错（将尝试 OCR）: {e}")
if not full_text.strip():
    # OCR 逐页转换以降低内存占用
    from pdf2image import convert_from_path
    for i in range(num_pages):  # 先可通过 pdfplumber 获取页数，或单页转换循环
        page_img = convert_from_path(pdf_path, first_page=i+1, last_page=i+1,
poppler_path=...)
        text = pytesseract.image_to_string(page_img[0], lang=ocr_lang)
        full_text += text + "\n"
```

（上面只是示例逻辑，实际实现时注意拿到页数或逐页迭代）

环境安装简要提示

- Windows 安装 Tesseract：下载安装器后把路径加入系统 PATH，或在脚本中设 `pytesseract.pytesseract.tesseract_cmd`。
- Poppler（Windows）：从官网/第三方下载 zip，解压后把 `bin` 路径作为 `poppler_path` 传给 `convert_from_path`。
- pip 包：`pip install pdfplumber pdf2image pytesseract pillow tqdm`

这段脚本的作用是**逐页优先使用 PDF 的文本层，若某页没有文本则对该页做 OCR**，最后把每页的结果（来自文本层或 OCR）写到一个 `.txt` 文件。它设计成逐页回退 OCR（`convert_from_path` 每次只渲染一页），以避免一次性把整本 PDF 渲染到内存导致 OOM。

下面把代码按模块逐步解释，并指出细节、潜在问题和可行的改进建议。

依赖与常量

```
import pdfplumberfrom pdf2image import convert_from_pathimport pytesseractfrom PIL import ImageOpsfrom tqdm import tqdmimport gcimport os
```

-

pdfplumber：提取 PDF 的文本层（适合“可选中文本”的 PDF）。

-
-

pdf2image.convert_from_path：借助 Poppler 把 PDF 渲染为 PIL Image。

-
-

pytesseract：调用本地 Tesseract 做 OCR。

-
-

PIL.ImageOps：导入了但在当前代码中没有被使用；常用于图像增强/反相/自动对比度等预处理。

-
-

tqdm：命令行进度条。

-
-

gc：手动触发垃圾回收以帮助释放图像对象占用的内存。

-
-

os：文件/目录操作。

-

接着是路径与参数配置：

```
pytesseract.pytesseract.tesseract_cmd = r"D:\tesseract-ocr\tesseract.exe"
POPPLER_PATH = r"D:\Poppler\poppler-25.07.0\Library\bin"
PDF = r"D:\pdf\BB60C-User-Manual.pdf"
```

```
OUTTXT = r"D:\pdf\BB60C-User-Manual_output.txt"
OCR_LANG = "chi_sim+eng"
```

- tesseract_cmd：若 Tesseract 不在系统 PATH，需要显式指定可执行程序位置（Windows 常见）。
-
-
- POPPLER_PATH：pdf2image 在 Windows 上通常需要显式传入 Poppler bin 目录（或把其加入 PATH）。
-
-
- PDF/OUTTXT：输入 PDF 路径与输出 TXT 路径。
-
-
- OCR_LANG：Tesseract 要用到的语言包（用 + 连接多个语言），需要确保这些 .traineddata 已安装到 Tesseract 的 tessdata。
-

---

preprocess_image(img) 函数

```
def preprocess_image(img):
    # 简单预处理：灰度 + 自适应二值（这里用固定阈值示例）
    gray = img.convert("L")
    # 二值化（阈值 150）
    bw = gray.point(lambda x: 0 if x < 150 else 255, "1")
    return bw
```

- 把图片转为灰度（L 模式），然后用固定阈值 150 二值化（返回 mode='1' 的二值图像）。
-
-

目的：去噪、增强字符对比，有时能提高 OCR 精度。

-
-

注意：二值化并非总是更好；对低对比度或彩色字体，反而可能丢失信息。可以替换为自适应二值、放大、去噪、开闭运算等更复杂预处理（OpenCV 可做更多操作）。

-

主处理流程（详细）

```
with pdfplumber.open(PDF) as pdf:
    num_pages = len(pdf.pages)
    out_lines = []
    for i in tqdm(range(num_pages), desc="处理页"):
        page = pdf.pages[i]
        page_text = page.extract_text()
        if page_text and page_text.strip():
            out_lines.append(f"=== Page {i+1} (text layer) ===\n{page_text}\n")
        else:
            # 逐页渲染并 OCR
            images = convert_from_path(PDF, first_page=i+1, last_page=i+1,
                                       dpi=300, poppler_path=POPPLER_PATH)
            img = images[0]
            try:
                img = preprocess_image(img)            # 可选预处理
                ocr = pytesseract.image_to_string(img, lang=OCR_LANG,
config="--psm 3")
                out_lines.append(f"=== Page {i+1} (OCR) ===\n{ocr}\n")
            finally:
                # 立即释放图片对象
                try:
                    img.close()
                except Exception:
                    pass
                del images
                gc.collect()
```

逐步说明：

1.

with pdfplumber.open(PDF) as pdf: 打开 PDF（上下文管理器保证关闭）。

2.
3.

num_pages = len(pdf.pages)：获取总页数。

4.
5.

for i in tqdm(range(num_pages), desc="处理页"):: 用 tqdm 显示进度，i 是 0-based 页索引。

6.
7.

对每一页：page.extract_text()：尝试提取文本层。如果该页是可复制的文本，这里会返回字符串；若是扫描图像通常返回 None 或空字符串。

if page_text and page_text.strip()::若文本非空（去掉纯空白），则把该页的文本以 === Page X (text layer) === 的标识追加到 out_lines 列表(这样输出文件里能区分哪些来自文本层,哪些来自 OCR)。

否则只针对当前页做 OCR：用 convert_from_path(..., first_page=i+1, last_page=i+1, dpi=300, poppler_path=...) 单页渲染（convert_from_path 的页码是 1-based，所以传 i+1）。逐页渲染节省峰值内存。

-
-

dpi=300：渲染分辨率常用 300 dpi（比 200 更有利于 OCR，但占用更多资源）。

-
-

images = convert_from_path(...) 返回一个列表，这里取 images[0]。

-
-

img = preprocess_image(img)：对图像做预处理（可选），然后 pytesseract.image_to_string(img, lang=OCR_LANG,

config="--psm 3") 执行 OCR。--psm 3 指定页面分割模式（PSM），不同 PSM 值对不同文档类型效果差异大，可调参（常用 3/4/6 等）。

finally 里做资源释放：

8.

- img.close()：关闭 PIL Image 句柄（注意：如果 img 已被 preprocess_image 替换为新的 Image，close() 调用是关闭当前变量指向的对象）。

- del images 和 gc.collect()：删除引用并强制垃圾回收，帮助释放内存（因为图像对象占用较多内存，Python 的垃圾回收可能延迟释放）。

写入文件与结束

```
os.makedirs(os.path.dirname(OUTTXT), exist_ok=True)with open(OUTTXT, "w", encoding="utf-8") as f:
    f.writelines(out_lines)print("处理完成，输出文件：", OUTTXT)
```

- os.makedirs(..., exist_ok=True)：确保输出目录存在。

- open(..., encoding="utf-8")：用 UTF-8 写文件（适合中文）。

- writelines(out_lines)：把 out_lines 列表写入文件（每项已经包含换行符）

- 最后打印输出文件路径。

好的，这段代码实现了一个**批量 PDF 文本提取工具**，能够自动判断 PDF 是否为可复制文本，如果不是则使用 OCR（Tesseract）识别，再将结果写入 TXT 文件。它还使用了**多线程并发处理**以加快处理速度，并显示进度条。下面我分模块详细解释。

---

1 导入依赖

```
import osimport pdfplumberfrom pdf2image import convert_from_pathimport pytesseractfrom concurrent.futures import ThreadPoolExecutor, as_completedfrom tqdm import tqdm
```

-

os：文件路径操作和目录管理。

-
-

pdfplumber：读取 PDF 并提取文本层，适合可复制文本的 PDF。

-
-

pdf2image.convert_from_path：将 PDF 页面渲染为图像，为 OCR 做准备。

-
-

pytesseract：调用本地 Tesseract OCR，将图像转为文本。

-
-

ThreadPoolExecutor, as_completed：多线程处理 PDF，提高批量处理效率。

-
-

tqdm：显示进度条，方便查看批量处理进度。

-

---

2 配置

```
pdf_folder = r"D:\pdf"
output_folder = r"D:\py-output"
ocr_lang = "chi_sim+eng"
max_workers = 4
os.makedirs(output_folder, exist_ok=True)
```

-

pdf_folder：待处理 PDF 所在文件夹。

-
-

output_folder：提取的文本输出路径。

-
-

ocr_lang：OCR 使用的语言，这里支持简体中文和英文。

-
-

max_workers：线程数量，一般等于 CPU 核心数。

-
-

os.makedirs(..., exist_ok=True)：确保输出目录存在，如果不存在就创建。

-

---

3 单个 PDF 处理函数

```
def process_pdf(pdf_path):
    filename = os.path.basename(pdf_path)
    txt_filename = os.path.splitext(filename)[0] + "_output.txt"
    txt_path = os.path.join(output_folder, txt_filename)

    full_text = ""
```

-

获取 PDF 文件名，生成输出 TXT 文件名。

-
-

full_text 用于存储整本 PDF 的文本内容。

-

3.1 尝试提取文本层

```
try:
    with pdfplumber.open(pdf_path) as pdf:
        for page in pdf.pages:
            text = page.extract_text()
            if text:
                full_text += text + "\n"except Exception as e:
    return f"[错误] pdfplumber 处理 {filename} 失败: {e}"
```

-

  打开 PDF，每页提取文本。

-
-

  如果某页没有文本 (None) 则跳过。

-
-

  出现异常时返回错误信息。

-

3.2 如果文本为空，使用 OCR

```
if not full_text.strip():
    try:
        pages = convert_from_path(pdf_path)
        for page in pages:
            text = pytesseract.image_to_string(page, lang=ocr_lang)
            full_text += text + "\n"
    except Exception as e:
        return f"[错误] OCR 处理 {filename} 失败: {e}"
```

-

  判断 PDF 是否为空白（即 pdfplumber 提取不到文本）。

-
-

使用 convert_from_path 渲染 PDF 为图像。

-
-

每页图像用 pytesseract OCR 转为文本。

-
-

OCR 出现异常返回错误信息。

-

3.3 写入 TXT 文件

```
with open(txt_path, "w", encoding="utf-8") as f:
    f.write(full_text)return f"[完成] {filename} → {txt_filename}"
```

-

将整本 PDF 的文本写入输出 TXT 文件。

-
-

返回处理结果字符串，用于日志打印。

-

---

4 主函数

```
def main():
    pdf_files = [
        os.path.join(pdf_folder, f)
        for f in os.listdir(pdf_folder)
        if f.lower().endswith(".pdf")
    ]

    if not pdf_files:
        print("❌ 没有找到 PDF 文件！")
        return
```

-

扫描 PDF 文件夹，筛选 .pdf 文件。

-
-

如果没有 PDF 文件则打印提示并退出。

-

4.1 并发处理

```
results = []with ThreadPoolExecutor(max_workers=max_workers) as
executor:
    future_to_pdf = {executor.submit(process_pdf, pdf): pdf for pdf in
pdf_files}

    for future in tqdm(as_completed(future_to_pdf),
total=len(pdf_files), desc="处理进度"):
        results.append(future.result())
```

-

`ThreadPoolExecutor(max_workers=max_workers)` 创建线程池。

-
-

`executor.submit(process_pdf, pdf)`：提交每个 PDF 给线程池执行。

-
-

`as_completed(future_to_pdf)`：迭代已完成任务，保证进度条实时更新。

-
-

每个完成的任务结果加入 `results` 列表。

-

4.2 打印处理结果

```
print("\n===== 处理结果 =====")for r in results:
    print(r)
```

-

打印每个 PDF 的处理结果，包含 [完成] 或 [错误]。

-

---

5 运行入口

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

-

Python 脚本入口，保证模块导入时不会自动执行。

-
-

直接运行脚本时调用 `main()`。

-

✓ 总结

这段代码实现了一个**高效、批量 PDF 文本提取工具**，特点：

1.

优先文本层提取，仅在无文本时才使用 OCR。

2.
3.

支持批量处理文件夹下所有 PDF。

4.
5.

并发加速，可同时处理多个 PDF。

6.
7.

带进度条，方便监控。

8.
9.

容错设计，每个 PDF 出错不会影响其他 PDF。

10.

改进点：OCR 可以加入逐页预处理（灰度/二值化）提升准确率。

`convert_from_path` 默认 DPI 可调整（如 300dpi 提升 OCR 精度）。

对大 PDF 可考虑分页 OCR 或限制内存占用。

输出 TXT 文件可以包含页码或 OCR 标记，方便后续查阅。

异常处理可以更详细，比如记录错误到日志文件。

如果你需要，我可以帮你把这段脚本改造成**逐页混合策略（有文本就用文本，无文本就 OCR），并且加入进度条和内存优化版本**，性能更稳健。我帮你改造成一个**逐页混合策略**版本，特点如下：

-

逐页处理：先用 pdfplumber 提取文本层，如果当前页为空再做 OCR。

-
-

内存优化：OCR 时每次只渲染一页，处理完及时释放图片对象。

-
-

进度条：显示整体 PDF 处理进度。

-
-

输出文件：每页结果加上页码和来源标记（文本层 / OCR）。

-
-

批量处理：可以处理文件夹下所有 PDF，同时支持多线程。

-

改进点说明：

1.

逐页混合策略：对每页优先用文本层，只有无文本时才 OCR。

2.
3.

内存优化：每次只渲染一页 PDF，处理完释放图片对象。

4.
5.

输出标识：每页都带 `=== Page X (text layer/OCR) ===`，方便查看来源。

6.
7.

多线程批量处理：处理文件夹下所有 PDF，支持并行，提高速度。

8.
9.

异常容错：每个 PDF 都 try/except 包裹，出错不影响其他 PDF

我来给你**逐行详细解释这个增强版逐页 OCR 批量处理 PDF 的代码**，让你清楚每个模块的功能和设计思路。

---

1 导入依赖

```
import osimport pdfplumberfrom pdf2image import convert_from_pathimport pytesseractfrom concurrent.futures import ThreadPoolExecutor, as_completedfrom tqdm import tqdmfrom PIL import Image, ImageOps, ImageFilterimport gc
```

-

`os`：操作文件路径、创建目录等。

-
-

`pdfplumber`：提取 PDF 的文本层。

-
-

`pdf2image.convert_from_path`：将 PDF 页渲染成图像，为 OCR 做准备。

-
-

`pytesseract`：Python 封装 Tesseract OCR，将图像转成文本。

-
-

`ThreadPoolExecutor`、`as_completed`：多线程处理多个 PDF 文件，提高批量处理效率。

-
-

`tqdm`：进度条显示。

-
-

`PIL.Image`, `ImageOps`, `ImageFilter`：图像预处理（灰度、二值化、自动对比度、去噪）。

-
-

`gc`：手动触发垃圾回收，释放内存，避免处理大 PDF 时内存占用过高。

-

---

2 配置参数

```
pdf_folder = r"D:\pdf"          # PDF 所在文件夹
output_folder = r"D:\py-output"  # 输出 TXT 文件夹
ocr_lang = "chi_sim+eng"         # OCR 语言
max_workers = 4                  # 并发线程数
dpi = 300                        # OCR 渲染分辨率
poppler_path = r"D:\Poppler\poppler-25.07.0\Library\bin"
pytesseract.pytesseract.tesseract_cmd =
r"D:\tesseract-ocr\tesseract.exe"

os.makedirs(output_folder, exist_ok=True)
```

- `pdf_folder`：存放待处理 PDF 文件的目录。
- `output_folder`：提取文本输出目录。
- `ocr_lang`：OCR 使用语言包（简体中文 + 英文）。
- `max_workers`：并发线程数，通常设为 CPU 核心数。
- `dpi`：渲染 PDF 时的分辨率，300 dpi 常用于 OCR。
- `poppler_path`：Windows 下 PDF 渲染依赖 Poppler，需要提供 bin 路径。
- `tesseract_cmd`：指定 Tesseract 可执行文件路径。

-

`os.makedirs(..., exist_ok=True)`：确保输出目录存在，否则创建。

-

3　OCR 图像预处理函数

```
def preprocess_image(img: Image.Image) -> Image.Image:
    """
    OCR 预处理：
    1. 灰度化
    2. 放大 1.5 倍
    3. 自动对比度 + 二值化
    4. 可选去噪滤波
    """
    # 灰度
    gray = img.convert("L")
    # 放大 1.5 倍
    w, h = gray.size
    gray = gray.resize((int(w*1.5), int(h*1.5)), Image.BICUBIC)
    # 自动对比度
    gray = ImageOps.autocontrast(gray)
    # 二值化
    bw = gray.point(lambda x: 0 if x < 150 else 255, "1")
    # 轻度去噪
    bw = bw.filter(ImageFilter.MedianFilter(size=3))
    return bw
```

功能

1.

灰度化：把彩色图像转换为灰度，减少计算复杂度。

2.
3.

放大：放大 1.5 倍，提高小字体识别率。

4.
5.

自动对比度：增强字符与背景对比度。

6.
7.

二值化：黑白化图像，使文字更突出。

8.
9.

中值滤波去噪：消除孤立噪点，减少 OCR 错误。

10.
11.

返回处理后的 PIL `Image` 对象，供 `pytesseract` 使用。

12.

---

4 单个 PDF 处理函数

```
def process_pdf(pdf_path: str) -> str:
    filename = os.path.basename(pdf_path)
    txt_filename = os.path.splitext(filename)[0] + "_output.txt"
    txt_path = os.path.join(output_folder, txt_filename)
```

-

获取 PDF 文件名，生成对应的 TXT 文件名。

-
-

`txt_path` 是输出文件完整路径。

-

4.1 异常保护

```
try:
    out_lines = []
```

-

捕获整个 PDF 的处理异常，保证即使处理失败也返回错误信息。

4.2 打开 PDF & 遍历页

```
with pdfplumber.open(pdf_path) as pdf:
    num_pages = len(pdf.pages)
    for i in range(num_pages):
        page = pdf.pages[i]
        page_text = page.extract_text()
        if page_text and page_text.strip():
            out_lines.append(f"=== Page {i+1} (text layer)
===\n{page_text}\n")
```

逐页读取 PDF。

`page.extract_text()` 尝试提取文本层。

如果该页有可用文本，就直接加入输出列表，并标记 `text layer`。

避免 OCR 浪费时间处理已有文本页。

4.3 无文本页执行 OCR

```
        else:
            images = convert_from_path(
                pdf_path, first_page=i+1, last_page=i+1,
                dpi=dpi, poppler_path=poppler_path
            )
            img = images[0]
            try:
```

```
                img_proc = preprocess_image(img)
                ocr_text = pytesseract.image_to_string(
                    img_proc, lang=ocr_lang, config="--psm 3"
                )
                out_lines.append(f"=== Page {i+1} (OCR)
===\n{ocr_text}\n")
            finally:
                try: img.close()
                except: pass
                try: img_proc.close()
                except: pass
                del images
                gc.collect()
```

-

逐页渲染 PDF → 图像（避免一次性渲染整本 PDF 占用大量内存）。

-
-

调用 `preprocess_image()` 对图像进行灰度、放大、对比度增强、二值化和去噪。

-
-

OCR 使用 Tesseract：

-
    -

`--psm 3`：页面分割模式 3（自动检测整页文本块）。

    -
-

输出标记 `OCR`。

-
-

资源释放：

-
    -

`img.close()` 释放 PIL Image。

    -
    -

`img_proc.close()` 释放处理后的图像。

    -
    -

`del images + gc.collect()` 释放内存，避免大 PDF 内存泄漏。

    -

4.4 写入 TXT 文件

```
with open(txt_path, "w", encoding="utf-8") as f:
    f.writelines(out_lines)
return f"[完成] {filename} → {txt_filename}"
```

-

将 `out_lines` 写入 TXT 文件，保留页码和来源标记。

-
-

返回处理成功信息。

-

4.5 异常捕获

```
except Exception as e:
    return f"[错误] {filename} 处理失败: {e}"
```

-

捕获处理整个 PDF 可能出现的异常，返回错误信息，保证批量处理不中断。

-

5 主函数

```
def main():
    pdf_files = [
        os.path.join(pdf_folder, f)
        for f in os.listdir(pdf_folder)
        if f.lower().endswith(".pdf")
    ]
    if not pdf_files:
        print("❌ 没有找到 PDF 文件！")
        return
```

-

扫描 PDF 文件夹，收集所有 .pdf 文件。

-
-

如果没有 PDF 文件，打印提示并退出。

-

5.1 多线程处理

```
    results = []
    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        future_to_pdf = {executor.submit(process_pdf, pdf): pdf for pdf
in pdf_files}
        for future in tqdm(as_completed(future_to_pdf),
total=len(pdf_files), desc="处理进度"):
            results.append(future.result())
```

-

使用线程池同时处理多个 PDF，提高批量处理效率。

-
-

future_to_pdf 保存任务和 PDF 文件映射。

-

-

`tqdm(as_completed(...))` 显示处理进度条。

-
-

每完成一个任务，将结果（成功或错误信息）加入 `results`。

-

5.2 打印结果

```
print("\n===== 处理结果 =====")
for r in results:
    print(r)
```

-

打印所有 PDF 的处理结果，方便查看哪份成功、哪份失败。

-

---

6 脚本入口

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

-

Python 标准入口，防止被导入模块时自动执行。

-
-

直接运行脚本时调用 `main()`。

-

✓ 总结

1.

混合策略：优先使用 PDF 文本层，无文本页再 OCR。

2.
3.

逐页处理 & 内存优化：每次只渲染一页，处理后释放图像对象。

4.
5.

OCR 增强：灰度化、放大、自动对比度、二值化、去噪。

6.
7.

批量处理 + 多线程：同时处理多个 PDF，提高效率。

8.
9.

输出标识：每页都标记来源（文本层 / OCR）。

10.
11.

异常保护：单个 PDF 出错不会影响整个批量任务。

12.
13.

进度条：直观显示处理进度。

14.