1、系统概述

1.1、系统简介

在我们日常学习和看论文的过程中，经常需要复制/翻译 PDF 中的内容用以询问 AI，在阅读比较长的文献时也会需要从文章中快速定位相关知识点。传统的解决方案是通过 OCR 获取纯文字，同时搜索也基于以上的纯文字，这样就难免出现低效率和不准确的情况，针对以上的情况，我们设计了学生文档助手这个程序。

我们的学生文档助手是一个前后端分离的项目，其主要分成前端和后端两个系统。其中，前端又可分为处理结果展示模块、嵌入结果导出、设置三个子模块；后端又可分为 PDF 转 Markdown 模块、数据库模块、设置相关模块和 RESTfulAPI 服务器模块。

1.2、术语表

定义系统或产品中涉及的重要术语，为读者在阅读文档时提供必要的参考信息。

序号	术语或缩略语	说明性定义
1	OCR	Optical Character Recognition,光学字符识别，将图像中的文字转换为可编辑的文本
2	Embedding	文本向量化表示，将文本转换为多维数字向量，用于计算文本相似度
3	FastAPI	Python 的现代 Web 框架，用于构建高性能的 API 接口
4	Vue3	渐进式 JavaScript 框架，用于构建用户界面
5	RESTful API	一种基于 HTTP 的 API 设计风格，通过标准 HTTP 方法提供服务
6	Markdown	一种轻量级标记语言，用于创建格式化文本
7	BLOB	Binary Large Object，用于在数据库中存储二进制数据
8	SQLite	轻量级关系型数据库，适用于嵌入式和本地应用
9	Multipart	HTTP 请求的一种形式，用于上传文件等二进制数据
10	Parallel Processing	并行处理，同时处理多个任务以提高效率
11	Vector Search	向量搜索，基于向量相似度的检索方法

1.3、系统运行环境

硬件平台：

CPU：12th Gen Intel® Core(TM) i7-12700H (20) @ 4.70 GHz

GPU：NVIDIA GeForce RTX 3050 Laptop GPU (2048) @ 2.10GHz

操作系统：Windows® 11 / Fedora Linux 42 (KDE Plasma Desktop Edition) x86_64

数据库系统：SQLite3

编程平台：VScode / Trae (with Extensions)

网络协议：HTTP

1.4、开发环境

工程工具：VScode v1.1.00.0 / Trae v1.0.14321 (with Extensions)

开发语言：前端：Vue3；后端：Python3.12

前端打包工具：Vite v4

后端运行时：Uvicorn ASGI Server (Python 3.12 Conda Environment)

接口测试工具：Reqable v2.33.7

2、数据结构说明

2.1、常量

系统运行所需的关键配置参数，来自源代码中实际定义的常量。

```
# 处理参数
MINERU_MIN_BATCH_INFERENCE_SIZE = \
int(os.environ.get('MINERU_MIN_BATCH_INFERENCE_SIZE', 200))

# 目录结构
os.makedirs("temp", exist_ok=True)
os.makedirs("images", exist_ok=True)
os.makedirs("pdf", exist_ok=True)
os.makedirs("source_docs", exist_ok=True)
```

2.2、变量

系统运行时的核心组件实例。

```
# FastAPI 应用实例和数据库实例
app = FastAPI()
vector_db = VectorDB()

# 跨域中间件配置
app.add_middleware(
    CORSMiddleware,
    allow_headers=["*"],
    allow_origins=["*"],
    allow_methods=["*"],
    allow_credentials=True
)
```

2.3、数据结构

2.3.1、PDF 处理结果结构

PDF 文档处理后的输出数据结构。从 parallel_pdf_extract.py 中可以看到，使用 ParallelBatchAnalyze 类处理 PDF 页面，并返回 InferenceResult 对象。每个页面的处理结果包含页码、尺寸和版面分析数据。

代码实现在 parallel_pdf_extract.py 中：

```
 def process_pdf(pdf_path: str, use_parallel: bool = True, num_workers:
int = 3):
    """处理PDF文件"""
    ds = PymuDocDataset(pdf_bytes)
    if use_parallel:
```

```
        infer_result = parallel_doc_analyze(
            ds,
            ocr=True,
            table_enable=False,
            formula_enable=False,
            num_workers=num_workers
        )
    else:
        infer_result = doc_analyze(
            ds,
            ocr=True,
            table_enable=False,
            formula_enable=False
        )
    return infer_result
```

2.3.2、向量数据库结构

向量数据库使用 SQLite 存储文档内容和向量数据。从 embedding.py 中的 VectorDB 类可以看到具体实现：

```
class VectorDB:
    def __init__(self, db_name='vector_db.db'):
        self.conn = sqlite3.connect(db_name)
        self.cursor = self.conn.cursor()
        self.model = SentenceTransformer('all-MiniLM-L6-v2')
        self._init_db()

    def _init_db(self):
        self.cursor.execute('''
            CREATE TABLE IF NOT EXISTS documents (
                id INTEGER PRIMARY KEY,
                filename TEXT NOT NULL,
                content TEXT NOT NULL,
                vector BLOB
            )
        ''')
```

2.3.3、API 响应格式

来自 server.py 中的实际 API 响应格式：

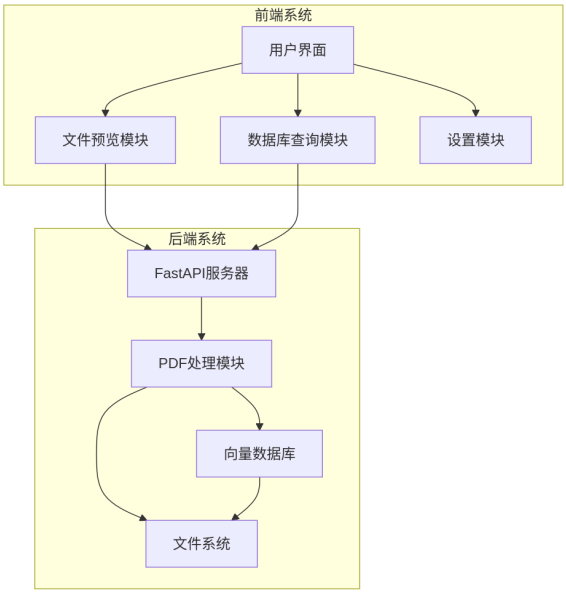
```
# PDF 上传响应
{
    "status": "success",
    "filename": filename,
    "message": "PDF successfully processed and stored"
}
```

```
# 搜索响应
{
    "status": "success",
    "results": results
}

# 文件列表响应
{
    "status": "success",
    "files": files
}

# 文件内容响应
{
    "status": "success",
    "filename": filename,
    "content": content
}
```

3、模块设计

3.1、软件结构

系统采用前后端分离的架构设计，主要分为以下模块：



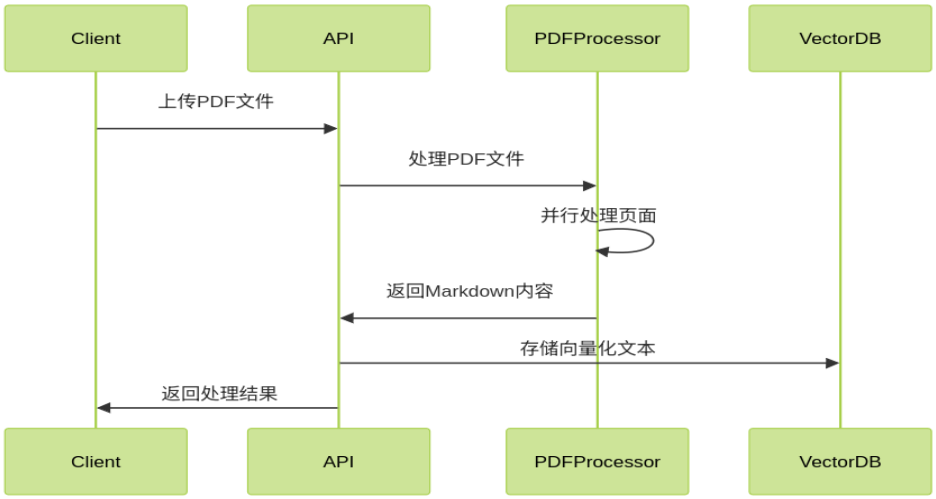
3.2、功能设计说明

系统采用模块化设计，各模块职责明确，通过 REST API 进行通信：

1. 前端系统：
    - 文件预览模块：处理 PDF 上传和预览
    - 数据库查询模块：向量检索和内容展示
    - 设置模块：系统配置管理
2. 后端系统：
    - FastAPI 服务器：提供 RESTful API 接口
    - PDF 处理模块：PDF 解析和 Markdown 转换
    - 向量数据库：文本向量化存储和检索
    - 文件系统：管理上传的文件和生成的资源

3.3、PDF 处理模块

3.3.1、设计图



3.3.2、功能描述

PDF 处理模块负责将 PDF 文档转换为 Markdown 格式，并支持图片提取和并行处理。

3.3.3、输入数据

PDF 文件（通过 multipart/form-data 上传）

处理参数：

- use_parallel: 是否使用并行处理
- num_workers: 并行处理的工作线程数

3.3.4、输出数据

Markdown 格式的文本内容

提取的图片文件

处理状态和错误信息

3.3.5、数据设计

给临时文件存储：

- temp/：临时 PDF 文件
- images/：提取的图片文件
- pdf/：已处理的 PDF 文件备份

3.3.6、算法和流程

1. 文件上传与验证
2. 并行页面处理：
   - 页面分批
   - OCR 文本识别
   - 图片提取
3. Markdown 转换
4. 向量化存储

3.3.7、函数说明

```
# server/src/parallel_pdf_extract.py
def process_pdf(pdf_path: str, use_parallel: bool = True, num_workers: int =
3):
    """
    处理 PDF 文件
    参数:
```

```
        pdf_path: PDF 文件路径
        use_parallel: 是否使用并行处理
        num_workers: 工作线程数
    返回:
        InferenceResult 对象
    """
```

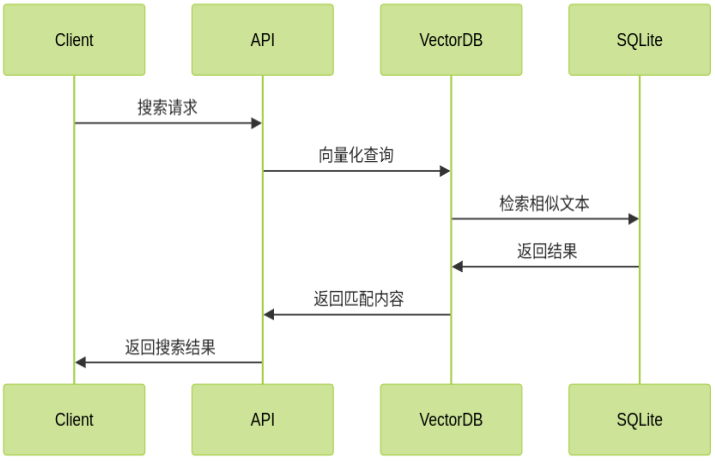
3.3.8、全局数据结构与该模块的关系

访问 VectorDB 类进行向量存储

使用临时文件系统存储处理结果

3.4、向量数据库模块

3.4.1、设计图



3.4.2、功能描述

向量数据库模块负责文本的向量化存储和相似度检索。

3.4.3、输入数据

文本内容

文档标识信息

查询参数

3.4.4、输出数据

向量化后的文本表示

相似度搜索结果

检索到的文本片段

3.4.5、数据设计

使用 SQLite 数据库存储文档内容和向量数据。

3.4.6、算法和流程

1. 文本分块
2. 向量化转换
3. 数据库存储
4. 相似度计算与检索

3.4.7、函数说明

```
class VectorDB:
```

```
    def add_markdown(self, text: str, filename: str):
        """添加并向量化 Markdown 文档"""

    def global_search(self, query: str, top_k: int = 5):
        """全局相似性搜索"""
```

3.4.8、全局数据结构与该模块的关系

维护 SQLite 数据库连接

管理向量化模型实例

4、接口设计

4.1、用户接口

系统提供 Web 图形用户界面，主要包含以下功能区域：

1. PDF 文件上传和预览区
2. Markdown 预览和复制区
3. 向量检索交互区
4. 系统设置面板

4.2、外部接口

系统通过 RESTful API 提供以下主要接口

4.2.1、PDF 处理接口

```
POST /upload_pdf
Content-Type: multipart/form-data

请求参数:
- file: PDF 文件

返回数据:
{
    "status": "success|error",
    "filename": "文件名",
    "message": "处理结果信息"
}
```

4.2.1、向量检索接口

```
POST /search
Content-Type: application/json

请求参数:
{
    "text": "搜索文本",
    "top_k": 10  // 可选，默认返回前 10 条结果
}
```

返回数据：

```
{
    "status": "success|error",
    "results": ["匹配文本1", "匹配文本2", ...]
}
```

4.2.1、文件管理接口

从 server.py 中实际实现的 API 路由：

```python
@app.get("/files")
async def get_files():
    try:
        files = vector_db.get_all_filenames()
        return JSONResponse(content={
            "status": "success",
            "files": files
        })
    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )

@app.get("/file/content/{filename}")
async def get_file_content(filename: str):
    try:
        content = vector_db.get_markdown_text(filename)
        if content is None:
            return JSONResponse(
                status_code=404,
                content={"status": "error", "message": "File not found"}
            )
        return JSONResponse(content={
            "status": "success",
            "filename": filename,
            "content": content
        })
    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )

@app.get("/file/pdf/{filename}")
async def get_pdf_file(filename: str):
```

```
    try:
        pdf_path = f"pdf/{filename}"
        return FileResponse(
            path=pdf_path,
            media_type="application/pdf"
        )
    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )

@app.get("/images/{filename}")
async def get_image(filename: str):
    try:
        image_path = f"images/{filename}"
        return FileResponse(
            path=image_path,
            media_type="image/jpeg"
        )
    except Exception as e:
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )
```

4.2.1、数据导出接口

```
GET /export
返回：ZIP 文件，包含所有 PDF、Markdown 和图片文件
```

4.3、内部接口

4.3.1、模块间通信

以下展示了从 server.py 中提取的核心模块间通信代码：

1. PDF 处理模块 → 向量数据库模块

```
# 处理 PDF 并存储到向量数据库
result = process_pdf(temp_path, use_parallel=True, num_workers=10)
markdown_content = result.pipe_txt_mode(dr).get_markdown("output")
true_markdown = process_images(markdown_content)
vector_db.add_markdown(true_markdown, filename.replace(".pdf", ""))
```

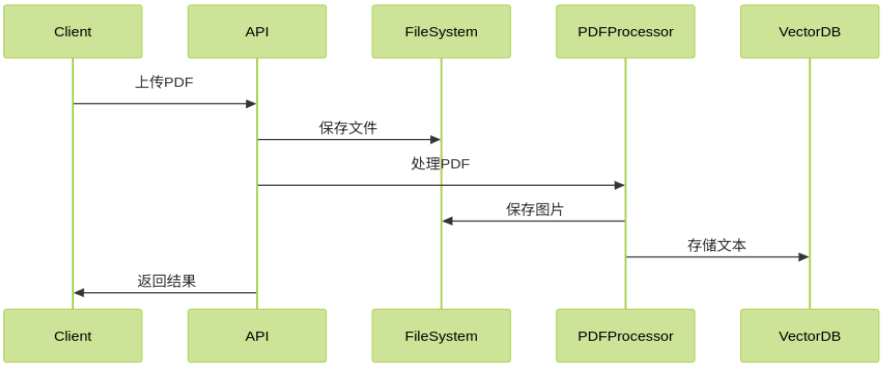
2. 向量数据库模块 → 文件系统

```
def get_markdown_text(self, filename: str):
    """获取原始 Markdown 内容"""
    file_path = f'source_docs/{filename}.md'
    if not os.path.exists(file_path):
        return None
    with open(file_path, 'r', encoding='utf-8') as f:
```

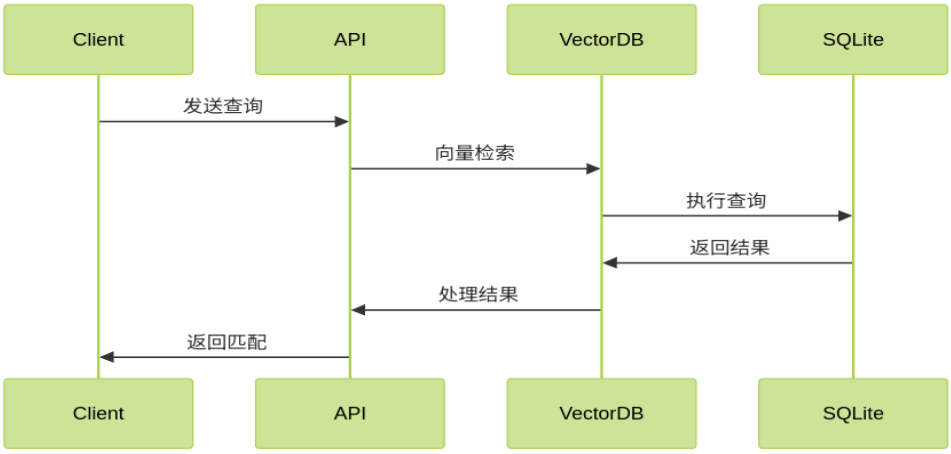
```
7           return f.read()
```

4.3.2、数据流

1. PDF 上传流程



2. 检索流程



5、数据库设计

系统使用 SQLite3 作为数据库管理系统，主要用于存储文档内容和向量数据。

5.1、数据库表设计

主要是存储文档内容和向量数据的主表

```
1  CREATE TABLE documents (
2      id INTEGER PRIMARY KEY,   -- 自增主键
3      filename TEXT NOT NULL,   -- 文档名称
4      content TEXT NOT NULL,    -- 文本内容片段
```

```
    vector BLOB              -- 向量化后的内容数据
)
```

字段说明：

- id: 自增主键，用于唯一标识每条记录
- filename: 文档名称，用于关联原始文件
- content: 文本内容片段，存储分割后的文本段落
- vector: BLOB 类型，存储向量化后的数据

5.1、文件系统结构

除数据库外，系统还维护以下目录结构：

```
├── temp/              # 临时文件目录
├── images/            # 提取的图片文件
├── pdf/              # 原始 PDF 文件
└── source_docs/      # Markdown 文档存储
```

6、系统出错处理

6.1、出错信息

6.1.1、API 错误响应

错误代码	错误类型	说明	处理方式
404	文件未找到	请求的文件不存在	检查文件名，确认文件是否已上传
500	服务器错误	PDF 处理或数据库操作失败	查看服务器日志，检查系统资源

6.1.1、前端错误提示

错误类型	提示信息	处理建议
PDF 加载失败	“PDF 加载失败，请重试”	检查网络连接和文件格式
内容复制失败	“复制失败，请重试”	检查浏览器剪贴板权限
数据导出失败	“无法下载文件”	检查存储空间/文件访问权限

6.1、错误处理机制

1. 服务器端错误处理，来自 server.py 中的实际实现：

```python
@app.post("/upload_pdf")
async def upload_pdf(file: UploadFile = File(...)):
    try:
        # 处理 PDF 文件
        temp_path = f"temp/{str(uuid.uuid4())}.pdf"
        with open(temp_path, "wb") as buffer:
            shutil.copyfileobj(file.file, buffer)
        # ...处理逻辑...
    except Exception as e:
```

```
        return JSONResponse(
            status_code=500,
            content={"status": "error", "message": str(e)}
        )
```

2. 前端错误处理，来自 FileView.vue 中的实际实现：

```
const handlePdfError = () => {
  ElMessage.error('PDF 加载失败，请重试')
  pdfUrl.value = ''
}

const copyToClipboard = async () => {
  try {
    await navigator.clipboard.writeText(markdownContent.value)
    ElMessage.success('已复制到剪贴板')
  } catch (error) {
    console.error('复制失败:', error)
    ElMessage.error('复制失败，请重试')
  }
}
```

6.2、补救措施

1. 文件处理错误：
- 清理临时文件
- 提供重试机制
2. 数据库错误：
- 事务回滚
- 定期备份

7、其他设计

7.1、性能优化

7.1.1、并行处理机制

系统采用多线程并行处理机制来提高 PDF 处理速度。从 parallel_pdf_extract.py 中可以看到具体实现：

```
class ParallelBatchAnalyze:
    def __init__(self, num_workers=3):
        self.num_workers = num_workers
        self.result_queue = queue.PriorityQueue()
        self.lock = threading.Lock()
        self.model_manager = ModelSingleton()

    def process_batches(self, batches: List[List[Tuple]], ocr: bool,
```

```
show_log: bool,
 9                                    layout_model=None, formula_enable=None,
table_enable=None) -> List:
10          """使用线程池并行处理批次"""
11                                                                          with
concurrent.futures.ThreadPoolExecutor(max_workers=self.num_workers)        as
executor:
12              futures = []
13              for batch_id, batch in enumerate(batches):
14                  future = executor.submit(
15                      self.worker,
16                      batch_id,
17                      batch,
18                      ocr,
19                      show_log,
20                      layout_model,
21                      formula_enable,
22                      table_enable
23                  )
24                  futures.append(future)
```

关键优化点：

1. 使用 ThreadPoolExecutor 进行并行处理
2. 可配置的工作线程数（num_workers）
3. 使用优先级队列保证结果顺序
4. 线程锁保证并发安全

7.1.2、并行处理机制

系统在向量数据库模块中选择了 all-MiniLM-L6-v2 作为文本向量化模型，这个模型具有以下优势：

1. 选择 384 维向量表示，相比其他模型（如 768 维或更高）大幅降低存储和计算开销
2. 保持较好的语义表示能力，适合文本相似度检索
3. 批量处理文本分块，提高向量化效率

7.2、安全设计

系统采用异常处理机制来保证运行时的稳定性。对于文件操作、数据库访问等关键操作，都实现了异常捕获和错误响应处理，确保在出现异常情况时能够返回合适的错误信息并进行必要的资源清理。