技术文档: 论文导航者 (PaperPilot)

版本

版本	更新日期	摘要	产出角色
V1.0	2026.2.12	新建TDD	茶茶、Gemini

其他相关文档

PRD：PRD(生成式): 论文导航者 (PaperPilot) —— 大学生科研全流程辅助平台

整体技术架构 (Tech Stack)

- **前端框架**：`Next.js 15 (App Router)` + `TypeScript`。
- **UI 组件库**：`Tailwind CSS` + `Shadcn UI` + `Framer Motion`（用于茶茶动画和数字滚动）。
- **本地存储**：`Dexie.js` (IndexedDB 的极简封装)。
- **状态管理**：`Zustand`（轻量级，方便跨组件共享当前 PDF 状态）。
- **大模型接口**：`OpenAI SDK` (配置为 DeepSeek 终结点)。

重点技术实现方案

1. PDF 解析与渲染 (PDF Engine)

1.1 核心视图层架构 (The Stack)

为了实现波浪线和划词菜单，我们需要在一个 `div` 容器内堆叠三个层：

1. **Canvas Layer (底层)**：由 `pdf.js` 负责，将 PDF 渲染为图片，保证视觉还原。
2. **Text Layer (中层)**：由 `pdf.js` 渲染的透明文本层，由无数个透明 `<span>` 组成，负责浏览器的原生划词。
3. **Interaction Layer (顶层)**：我们自定义的 SVG 或 Canvas 画布，负责绘制 AI 引导波浪线。

1.2 核心操作技术方案

PDF 如何展示在界面中?

- **加载流**:
  a. 用户上传文件，生成 `Object URL`。
  b. `pdfjs.getDocument` 加载文件。
  c. **虚拟列表渲染 (Virtual List)**：为了保证长论文不卡顿，只渲染视口内及其上下 2 页的 `Canvas` 和 `TextLayer`。
- **缩放策略**:
  - 通过全局变量 `scale` 控制。
  - 缩放时，必须重新触发 `page.render()` 并同步更新 `TextLayer` 的 CSS 变换。

如何实现系统自动画“波浪线”？

AI 给出的只是段落文本，我们需要将其转化为屏幕上的物理坐标。

1. **文本定位 (Anchor Finding)**:
   - 利用 `page.getTextContent()` 获取页面所有字符的矩形数据（`transform` 矩阵）。
   - 在当前页匹配 AI 指定的关键词/句子。
2. **坐标转换**:
   - 将 PDF 的内部坐标（Points）转换为浏览器的像素坐标（Pixels）。
   - 公式：$DisplayX = PDFX \times scale$。
3. **SVG 绘制**:
   - 在 **Interaction Layer** 绘制 `<path>`。
   - **波浪线效果**：通过 `stroke-dasharray` 或自定义 SVG Path 形状（如 `M 0 5 Q 5 0 10 5 T 20 5`）实现。
   - **挂载 ID**：给每个波浪线 Path 绑定一个 `data-logic-id`，点击时通过该 ID 唤起右侧侧边栏对话。

如何实现划词并唤起悬浮菜单?

这是最考验细节的地方，我们需要利用浏览器的 `Selection API`。

1. **监听选择事件**:
   - 在 `TextLayer` 上监听 `onMouseUp` 事件。
2. **获取选区坐标**:

代码块

```
const selection = window.getSelection();
const range = selection.getRangeAt(0);
```

```
3   const rect = range.getBoundingClientRect(); // 获取选区在视口中的绝对位置
```

3. **菜单定位与显示**：
   - 创建一个状态 `menuPosition: { x, y, visible }`。
   - 计算位置：`y = rect.top - menuHeight - offset`，`x = rect.left + rect.width / 2`。
   - 使用 `framer-motion` 实现平滑的弹出效果。
4. **内容处理**：
   - 通过 `range.toString()` 获取选中的文本，传入 `useChat` Hook 或存入 `Zustand` 状态，等待用户点击菜单中的 [学术大白话] 或 [存入术语表]。

1.3 关键边缘情况处理 (Edge Cases)

- **页面滚动中**：划词菜单应随页面滚动而自动消失，或通过 `position: absolute` 相对于父容器定位，防止漂移。
- **多行划词**：`getBoundingClientRect` 会返回一个包含多行的整体矩形，如果需要精准贴合，应使用 `getClientRects()` 遍历每一行。
- **响应式缩放**：当用户点击“放大/缩小”时，必须调用 `Interaction Layer` 的重绘函数，重新计算波浪线的路径，否则波浪线会脱离原文。

---

2. 术语库全局高亮算法 (Terminology Matcher)

- **挑战**：对于 **Terminology Matcher（术语全局匹配器）** 来说，最大的技术挑战在于：**随着术语库（Glossary）的增大，如何在不阻塞主线程（保持 60fps 滚动性能）的前提下，在 PDF 复杂的 DOM 结构中精准找到并高亮这些词。**

核心匹配逻辑：Aho-Corasick 算法 (AC 自动机)

如果使用普通的正则表达式遍历（$O(n \times m)$），当术语有 500 个、PDF 文本有 1 万字时，匹配速度会产生肉眼可见的卡顿。

- **技术选型**：使用 **AC 自动机** 算法。
- **优势**：一次扫描（Linear Time）即可找出文本中所有匹配的术语。
- **工作流**：
  a. **构建期**：应用启动或术语库更新时，在本地构建一颗 `Trie Tree`（前缀树）并添加失败指针。

b. **匹配期**：将 PDF 页面的文本流输入 AC 自动机，输出所有命中的 `[start, end, term_id]`。

渲染策略：TextLayer 注入法

PDF.js 生成的 `TextLayer` 是由一个个绝对定位的 `<span>` 组成的。我们不能直接用 `innerHTML` 替换（会破坏 PDF.js 的定位索引）。

方案：基于 Range 的虚拟高亮

1. **文本重建**：将当前页所有 `<span>` 的 `textContent` 拼接成一个连续字符串，并记录每个 `<span>` 在字符串中的偏移量索引。
2. **索引映射**：
   - AC 自动机返回：术语 "SOTA" 出现在第 105-108 字符。
   - 查索引表得知：第 105-108 字符横跨了第 3 和第 4 个 `<span>`。
3. **DOM 渲染**：
   - 利用 `Selection` 和 `Range` API，针对这些跨度创建高亮。
   - 或者更简单的：在 `Interaction Layer` （SVG层）根据对应 `<span>` 的位置信息，绘制淡青色虚线框。
   - **推荐方案**：为匹配到的 `<span>` 添加一个 `data-term-id` 属性，并通过 CSS `::after` 或伪元素实现虚线效果。这样性能最好，且能原生响应 Hover 事件。

性能护城河：多线程与防抖 (Web Worker)

为了绝对不阻塞 UI 滚动，匹配逻辑必须搬离主线程。

- **Worker 协作流**：
  a. **主线程**：监听到页面进入视口（Intersection Observer）。
  b. **Worker 线程**：接收该页文本 + 术语树。
  c. **Worker 线程**：执行 AC 自动机匹配，返回命中列表 `[{word: 'SOTA', index: [105, 108]}]`。
  d. **主线程**：根据索引在 DOM 上打标记。

交互：Hover 预览与跳转

- **Hover 逻辑**：
  - 采用 **事件委托 (Event Delegation)**。在 PDF 容器上监听 `mouseover`。
  - 匹配 `e.target.dataset.termId`。
- **Pop-up 渲染**：
  - 从 Zustand Store 或 Dexie 获取该术语的“茶茶大白话”解释。

- 使用 `Floating UI` 库处理弹出位置，确保弹窗不超出屏幕边缘。

- **跳转溯源**：
  - 如果该术语是从另一篇论文 A 存入的，点击弹窗中的“查看来源”，侧边栏切换论文 A，并自动滚动到当时划词的页面。

异常处理与冲突

- **长短词冲突**：如果术语库同时有 "Transformer" 和 "Vision Transformer"，AC 自动机应支持**最长匹配原则**，避免嵌套高亮导致视觉混乱。
- **动态更新**：用户阅读时新存入一个术语，触发全局广播，所有当前已渲染的页面通过 `requestIdleCallback` 重新扫描。

---

3. DeepSeek 多轮对话与思考流处理

- 解决如何让AI流式输出、用户对话过长时，如何压缩上下文作为prompt传给AI等问题，后者是 AI 交互中最考验“工程审美”的部分。DeepSeek-V3.2 的 **Reasoning Content（思考流）** 如果处理得好，会像一个活生生的学长在思考；如果处理不好，就会导致页面跳动或 Token 成本失控。

DeepSeek 思考流与响应流的处理 (Streaming)

DeepSeek 的 API 返回的是典型的 `Server-Sent Events (SSE)`。特殊之处在于它比普通模型多了一个 `reasoning_content` 字段。

数据流解析逻辑

我们需要在前端实现一个流式解析器，实时分发“思考”和“回答”：

- **解析逻辑**：
  a. 监听 SSE 的每一帧。
  b. 如果帧内包含 `reasoning_content`，则追加到 `current_thought` 状态，UI 展现为“学长正在推导...”的折叠区域。
  c. 如果帧内包含 `content`，则追加到 `current_response` 状态，UI 展现为正常的对话气泡。
  d. **自动结束标志**：当 `finish_reason` 为 `stop` 时，关闭流并触发本地持久化（写入 IndexedDB）。

上下文压缩与 Prompt 策略 (Context Management)

随着对话深入，Context 会越来越长（尤其是包含 PDF 文本片段时）。为了节省 Token 并保持响应速度，必须采用 **“滑动窗口 + 语义压缩”** 策略。

1. 动态上下文构造模型

每一轮请求给 AI 的 Prompt 结构如下（优先级由高到低）：

1. **System Prompt**：核心人设与苏格拉底教学法规约（永远保留）。
2. **Current Context**：当前用户划选的原文片段（永远保留）。
3. **Recent History**：最近的 3-5 轮原始对话（保留细节）。
4. **Compressed Summary**：更早之前的对话摘要（压缩处理）。

2. 压缩执行逻辑 (The Summary Loop)

- **触发阈值**：当历史对话超过 10 轮或 Token 数达到 4000（可调）。
- **压缩动作**：
  a. 后台调用一个轻量级的 `gpt-4o-mini` 或 `DeepSeek-V3`（非思考模式），将前 6 轮对话压缩成一段 200 字以内的“对话简报”。
  b. **保留关键点**：摘要中必须包含：用户已弄懂的知识点、尚未解决的疑问。
  c. **更新内存**：在后续请求中，将这 6 轮对话从消息数组中剔除，替换为一条 `role: system` 的简报消息。

针对 PDF 的“精准引用”协议

为了让茶茶学长说话像“在看着 PDF 一样”，我们需要在 Prompt 中强制约束引用格式。

引用协议设计

在回复中，要求 AI 使用特定的 Markdown 格式来引用 PDF 内容：

- **格式**：`[[page_num, line_num]]`
- **前端渲染**：当 Markdown 渲染器解析到这个格式时，自动转换成一个**可点击的标签**（如：`第 5 页第 12 行`）。
- **交互逻辑**：用户点击标签 -> 左侧 PDF 预览区自动通过 `scrollTo` 滚动到对应坐标并闪烁高亮。

---

4. DeepSeek 对话输出显示

- 对话组件必须支持 LaTeX 和 Markdown 渲染。因为我们需要处理大量的 **LaTeX 公式（数学/物理论文核心）**、**Markdown 语法（解构看板核心）** 以及我们自定义的 **PDF 引用锚点**。

渲染引擎选型

为了保证性能和扩展性，我们采用 **React-Markdown** 生态链，因为它具有极强的插件化能力。

- **基础渲染**：`react-markdown`。
- **数学公式**：`remark-math` (解析) + `rehype-katex` (渲染)。

- **代码高亮**：`react-syntax-highlighter` (支持 Prism 或 hljs)。
- **安全性**：`rehype-raw` (慎用，需配合 `sanitize`)。

LaTeX 公式渲染逻辑 (Math Support)

学术论文中存在大量行内公式（如 $E=mc^2$）和块级公式。

- **匹配规则**：
  - 行内：`$ ... $`
  - 块级：`$$...$$`
- **性能优化**：
  - KaTeX 渲染比 MathJax 快得多。我们需要在全局引入 `katex.min.css`。
  - **流式预处理**：由于 DeepSeek 在流式输出时可能会先输出一个 `$`，导致公式解析器暂时失效。我们需要一个**正则缓冲区**，确保公式闭合后再进行 KaTeX 渲染，防止页面闪烁。

自定义渲染器 (Custom Components)

这是 PaperPilot 的特色功能。我们需要改写 `react-markdown` 的默认组件行为，插入我们的业务逻辑。

1. PDF 引用锚点渲染 (Citation Link)

- **逻辑**：识别文本中的 `[[page, line]]` 标记。
- **实现**：

代码块

```
const components = {
  text: ({ value }) => {
    const regex = /\[\[(\d+),\s*(\d+)\]\]/g;
    // 将文本替换为具有点击事件的 <Badge> 组件
    return value.split(regex).map(...);
  }
}
```

- **效果**：渲染为一个带图标的深蓝色小标签。点击后，调用 `usePDFStore.scrollTo(page, line)`。

2. 思考流展示区 (Thinking Box)

- **UI 表现**：对话气泡顶部一个浅灰色、带“逻辑脑”图标的折叠框。

- **状态联动**：
  - 当 `isThinking` 为真时，折叠框内部显示流式的 `reasoning_content`。
  - 当 `isThinking` 结束，折叠框默认收起，用户可手动展开查看学长的“心路历程”。

样式规范与交互设计 (UX Details)

1. Markdown 样式类 (Typography)

使用 `@tailwindcss/typography` 插件，但需要针对对话气泡进行微调：

- **表格 (Table)**：学术论文常有对比表，必须支持全宽展示、带边框、隔行变色。
- **列表 (List)**：区分有序列表和无序列表。
- **引用 (Blockquote)**：用于展示论文中的原话，左侧带粗边。

2. 流式自动滚动 (Auto-scroll)

- **逻辑**：当 AI 正在输出时，如果当前滚动条在底部，则随着内容增加自动向下滚动。
- **防止抖动**：如果用户手动向上滚动查看历史，则暂停自动滚动。

---

5. AI 预解析

等待期的交互增强 (The Parsing UI)

为了消解用户在预解析时的焦虑，前端需根据解析的**流式状态**同步反馈：

1. **0-3s (本地处理)**：UI 显示“茶茶正在帮你翻开书页...”。
2. **3-8s (AI 宏观扫描)**：UI 显示“茶茶正在概括论文大意...”。
3. **8-15s (逻辑埋点)**：UI 显示“茶茶正在寻找值得讨论的重点...”并逐步在 PDF 上渲染出第一批波浪线。

预解析的三大目标

- **结构化映射**：提取目录、标题、公式坐标，建立“坐标 -> 语义”的映射表。
- **语义提炼**：生成 Motivation（动机）、Method（方法）、Results（结果）、Gap（局限）的初步结论。
- **交互布点**：预判论文难点，生成 3-5 个苏格拉底式提问的“埋点”。

详细执行步骤 (Workflow)

第一阶段：本地预处理 (Client-side Extraction)

在请求 AI 之前，前端利用 `pdf.js` 进行特征提取，减轻 AI 的上下文压力。

- **文本流清洗**：提取全文文本，去除页码、页眉页脚等噪音。

- **目录解析**：通过字体大小和粗细判断 H1/H2 标题，生成 JSON 大纲。
- **图片/公式定位**：记录图中 Figure 和 Table 的具体页码位置，生成 `Assets_Map`。

第二阶段：AI 级联解析 (AI Multi-pass Processing)

由于 PDF 全文可能超过 5 万字，直接投喂会丢失细节。我们采用 **“快速扫描 + 深度采样”** 的策略。

Step 1: 宏观扫描 (Macro Scan)

- **输入**：Abstract + Introduction + Conclusion。
- **任务**：生成“一键解构”看板的初步草稿。
- **产出**：核心研究目标、主要贡献、结论。

Step 2: 难点嗅探 (Logic Probing)

- **输入**：Methodology 部分的核心段落 + 关键公式。
- **任务**：识别论文中最难懂、最具逻辑跨度的部分。
- **产出**：3-5 个 **[坐标 + 引导话术]**。
  - *示例*: `{ page: 5, rect: [x,y,w,h], query: "学弟，这个损失函数里加的这个惩罚项，你觉得是为了防止过拟合还是加速收敛？" }`

Step 3: 术语发现 (Glossary Mining)

- **任务**：比对当前用户的 `glossary` 表，发现文中出现的新生可能不懂的专有名词（如：Backpropagation, Attention Mask）。
- **产出**：建议存入术语表的词条。

提示词工程 (Prompt Orchestration)

预解析不是一次简单的总结，而是一次“深层语义扫描”。我们将使用 **Structured Prompting（结构化提示词）** 策略，强制要求 AI 产出符合前端数据结构的 JSON。

A. 预解析全局系统提示词 (Pre-parsing System Prompt)

> **Role**: 你是 PaperPilot 的智能中枢「茶茶学长」。你现在拥有一篇论文的全文文本（已清洗）。你的任务是充当“先遣队”，为用户建立阅读地图。
>
> **Task Objectives**:
>
> 1. **核心解构**：提取论文的四大支柱（Motivation, Method, Result, Gap）。
> 2. **逻辑锚点**：寻找 3-5 处论文中最具启发性、或逻辑转折最硬核的段落。
> 3. **知识发现**：识别出 5 个对新手有门槛的领域专有名词。
>
> **Output Requirement**: 必须且只能输出标准 JSON 格式，严禁任何解释性文字。
>
> **Reasoning Logic (思考路径)**:
>
> - 优先扫描 Introduction 寻找 Motivation。

- 扫描 Experiment 寻找 Results，并与 Abstract 里的结论对齐。
- 寻找诸如 "However", "In contrast", "We propose" 等关键词定位逻辑转折点。

B. 输入内容模版 (User Prompt Construction)

前端会将 PDF 提取出的信息按以下格式拼接后发送给 DeepSeek：

代码块

```
--- 论文元数据 ---
Title: {{title}}
Authors: {{authors}}

--- 结构化文本片段 ---
[Abstract]: {{abstract_text}}
[Introduction]: {{intro_chunks}}
[Methods]: {{method_chunks}}
[Conclusion]: {{conclusion_text}}

--- 任务指令 ---
请基于上述内容，完成预解析任务，并为每个“逻辑锚点”提供在原文中的精确字符串片段（用于前端定位）。
```

C. 预期输出数据结构 (Expected JSON Schema)

为了让前端（如 `Dexie.js`）能无缝解析，AI 返回的 JSON 必须严格遵循以下格式：

代码块

```
{
  "deconstruction": {  // 对应右侧“解构看板”数据
    "motivation": "作者认为现有的 [A技术] 在处理 [B场景] 时存在 [C缺陷]，因此提出此方案。",
    "method": "核心是引入了 [X机制]，通过 [Y算法] 实现了对 [Z变量] 的动态调整。",
    "result": "在 [Dataset] 上准确率提升了 15%，且计算开销降低了 20%。",
    "gap": "该方法在 [极端长文本] 场景下的表现尚未验证，且对 [硬件A] 有依赖。"
  },
  "logic_anchors": [ // 对应 PDF 上的“智能波浪线”
    {
      "anchor_text": "Specifically, we replace...", // 关键！这是原文的文本片段
      "page_hint": 4, // 缩小搜索范围，提高性能
      "chacha_comment": "学弟，注意看这里..." // 点击波浪线后，对话框的首句引导语
    }
  ],
  "suggested_glossary": [ // 对应术语预存建议
    { "term": "Ablation Study", "category": "General Academic" },
    { "term": "Transformer", "category": "Domain Specific" }
```

```
    ]
}
```

数据落地逻辑 (Data Hydration)

解析完成后，数据必须立即持久化到 IndexedDB，以保证“瞬时加载”体验。

IndexedDB `paper_analysis` 数据结构：

代码块

```
{
  paperId: "hash_id",
  deconstruction: {
    motivation: "...",
    method: "...",
    result: "...",
    gap: "..."
  },
  hints: [
    { page: 2, position: {x,y}, text: "...", chachaQuestion: "..." }
  ],
  status: "analyzed"
}
```

6. PDF 虚拟渲染

对于一个需要同时承载 **PDF渲染、TextLayer划词、SVG波浪线、术语高亮** 这四层复杂交互的项目来说，**虚拟渲染（Virtual Rendering / Windowing）** 不是“锦上添花”，而是“生存基石”。

如果一次性加载一篇 50 页的论文（每页包含大量 Canvas 像素和数千个 DOM 节点），内存占用会迅速飙升至 2GB 以上，导致低配设备浏览器直接崩溃。

以下是针对 PaperPilot 的 **PDF 虚拟渲染技术方案**。

核心原理：视口驱动的按需加载

虚拟渲染的核心是：**只渲染用户“看得见”的那几页，销毁“看不见”的 Canvas 以释放内存。**

- **渲染窗口（Buffer）**：设定当前视口页为 N，则实际渲染范围为 $[N-1, N, N+1]$。
- **占位容器（Placeholder）**：未渲染的页面使用一个与原页面比例一致的空白 `div` 撑开高度，确保滚动条的长度和位置始终准确。

技术实现路径

页面高度预估与占位

在 PDF 加载之初（`getDocument` 成功后），我们需要先获取每一页的原始比例（Viewbox）。

- **方案**：遍历 `pdf.getPage(i)` 获取宽和高，存入一个 `pageRects` 数组。
- **效果**：计算出全书总高度。滚动条会根据这个总高度渲染，用户拖动滚动条时，页面不会发生“跳动”。

滚动监听与动态装载 (The Intersection Observer)

我们不使用传统的 `onScroll` 监听（性能差），而是使用 `Intersection Observer` 监控每个页面的占位容器。

1. **进入视口**：当页面 N 的占位符进入视口 -> 触发 `pdf.getPage(N)` 渲染。
2. **渲染优先级**：
   - **Phase 1**: 渲染 `Canvas` （最快让用户看到内容）。
   - **Phase 2**: 延迟加载 `TextLayer` （用于划词和术语高亮）。
   - **Phase 3**: 延迟加载 `InteractionLayer` （用于画 AI 波浪线）。
3. **离开视口**：当页面离开视口超过 2 页距离 -> 销毁 Canvas 对象，将 DOM 节点恢复为简单的占位 `div` 。

内存管理 (Memory Management)

PDF.js 的渲染非常耗费显存。

- **策略**：显式调用 `canvas.width = 0; canvas.height = 0;` 来强制浏览器回收 Canvas 占用的内存。
- **清理内容**：同时清理该页在状态库（Zustand）中绑定的 DOM 引用，防止内存泄漏。

复杂性挑战：虚拟渲染下的“波浪线定位”

在虚拟渲染模式下，波浪线绘制面临一个悖论：**AI 给出的波浪线在第 10 页，但用户现在在第 1 页，第 10 页的 DOM 还没创建，怎么定位？**

解决方案：延迟补偿机制 (Lazy Compensation)

1. **数据挂起**：AI 预解析产生的 `logic_anchors` 存入全局 Store，标记状态为 `pending` 。
2. **滚动触发**：当用户滚动到第 10 页，虚拟渲染引擎创建该页的 `TextLayer` 。
3. **生命周期钩子**：在 `TextLayer.onRenderSuccess` 回调中，检查 Store ： “本页是否有待处理的波浪线？”。
4. **实时渲染**：如果有，此时再执行 `anchor_text` 匹配并绘制。

AI prompt

不同阶段给AI的prompt

详见 PRD: 论文导航者 (PaperPilot) —— 大学生科研全流程辅助平台

探索：AI是否有学习模式、甚至是学习模型？【二期】

- 在人工智能领域，**Mem0** 是一个被誉为“大模型记忆层”的开源项目。它解决了目前 AI 交互中的一个痛点：AI 往往会在对话结束后“忘记”你是谁，或者难以在多个应用之间同步你的偏好。
- 你说的关于学习模型的商业前景，在业内被称为 **ITS (Intelligent Tutoring Systems，智能导师系统)**。

发展阶段	特征	现状
L1: 问答机器人	你问它答，没有上下文感。	2023 年前的产品
L2: 伴学导师 (我们目前的 PaperPilot)	拥有 Prompt 约束，懂得引导，能感知 PDF。	**目前的 SOTA（尖端**
L3: 适应性导师 (Adaptive Learning)	**AI 拥有学习模型**。它能分析你的认知曲线，为你定制学习路径。	极少数创业公司（好
L4: 共同进化 AI	AI 和你一起读论文，它学习你的思维方式，最终成为你的“数字分身”。	实验室阶段

数据模型设计 (Database Schema)

LocalStorage

Key	类型	说明	为什么非存 LS 不可?
PP_HAS_GUIDED	boolean	**新手引导完成标志**。true 表示用户已完成 M1。	页面加载的第一时间就要判断：强行弹起 M1 教学框。如果存 I 现 M2 再跳 M1。
PP_LAST_PAPER_ID	string	hash_12345	实现“回到上次读到的那篇论文

IndexedDB 表结构 (Dexie 定义)

(1) `papers` 表

- **关键点**：我们把 `fileData` 直接存入 IndexedDB。这样用户刷新页面或断网时，不需要重新上传，实现“秒开”。

- **解构字段**：预解析完成后直接存入，不再重复请求 AI。

(2) `anchors` 表 (波浪线)

- **关键点**：存储 `anchorText` 而不是固定坐标。每次页面加载时，根据 `anchorText` 动态计算当前缩放比下的 `rects`。

(3) `messages` 表 (对话流)

- **关键点**： `reasoning` 字段单独存储。UI 渲染时可以根据这个字段是否存在来展示“思考折叠框”。
- **同步位**： `isMemorySynced` 标记该消息是否已经被 Mem0 提取成“长效记忆”。

(4) `glossary` 表 (术语库)

- **关键点**： `word` 作为主键，方便在全站 PDF 阅读时进行快速匹配（AC 自动机的数据源）。

(5) `userProfile` 表 (画像层)【未来启用】

- **关键点**：记录用户的“认知状态”。例如： `{ key: 'math_level', value: 'advanced' }`。这部分数据将作为长效记忆的本地副本。

代码块

```
// db.ts - Dexie 数据库定义
import Dexie, { type Table } from 'dexie';

export interface Paper {
  id: string;               // 文件 MD5 哈希作为唯一 ID
  title: string;
  fileData: ArrayBuffer;    // PDF 原始二进制数据
  status: 'reading' | 'archived'; // 阅读中或已存档
  progress: number;         // 阅读进度 (0-1)
  lastReadTime: number;     // 时间戳
  deconstruction: {         // M2 预解析的解构看板数据
    motivation: string;
    method: string;
    result: string;
    gap: string;
  };
}

export interface LogicAnchor {
  id: string;               // 锚点 ID
  paperId: string;          // 所属论文
  pageHint: number;         // 页码建议
  anchorText: string;       // 匹配的原文文本
  chachaComment: string;    // 茶茶的初始提问
  rects: any[];             // 前端计算出的坐标缓存（选填，缩放后需更新）
}
```

```ts

export interface Message {
  id?: number;              // 自增 ID
  paperId: string;
  role: 'user' | 'assistant';
  content: string;
  reasoning?: string;       // DeepSeek 的思考流内容
  timestamp: number;
  isMemorySynced: boolean;  // 是否已同步到长效记忆(Mem0)
}

export interface Glossary {
  word: string;             // 术语原文（主键）
  explanation: string;      // 茶茶大白话解释
  sourcePaperId: string;    // 来源论文 ID
  addedTime: number;
  masteryLevel: number;     // 掌握度 (0-1)，用于“学习模式”
}

export interface UserProfile {
  key: string;              // 'learning_style', 'academic_level' 等
  value: any;
}

// 定义 Dexie 实例
export class PaperPilotDB extends Dexie {
  papers!: Table<Paper>;
  anchors!: Table<LogicAnchor>;
  messages!: Table<Message>;
  glossary!: Table<Glossary>;
  userProfile!: Table<UserProfile>;

  constructor() {
    super('PaperPilotDB');
    this.version(1).stores({
      papers: 'id, title, status, lastReadTime', // 索引字段
      anchors: 'id, paperId',
      messages: '++id, paperId, timestamp, isMemorySynced',
      glossary: 'word, sourcePaperId, addedTime',
      userProfile: 'key'
    });
  }
}

export const db = new PaperPilotDB();
```

代码规范&可维护性

一、 宏观架构约束 (Macro Architecture)

> 1. **数据为王**：所有交互状态需在 Zustand 中同步，所有持久化数据存入 IndexedDB。
> 2. **性能至上**：PDF 渲染需支持虚拟列表，术语匹配需走 Web Worker。
> 3. **人设对齐**：AI 回复组件必须支持 LaTeX 和思考流折叠，且语气需符合‘茶茶学长’人设。
> 4. **代码整洁**：遵循分层架构，View 层不写业务逻辑，Service 层不操作 DOM。”

分层设计原则 (The 3-Tier Layering)

项目严格遵守 **“逻辑与视图分离”** 的三层架构，严禁在 UI 组件内直接书写复杂的算法或数据库查询。

- **View Layer (React Components)**: 仅负责 UI 渲染和响应用户事件。
- **Service Layer (Pure TS Classes)**: 核心逻辑层。包括 `AIService`、`DBService`、`PDFParser`。
- **Data Layer (Storage)**: 封装对 IndexedDB (Dexie) 和 LocalStorage 的访问。

PDF 三层夹心组件模型 (The Sandbox Model)

PDF 容器必须严格遵守 `Relative-Absolute` 布局，确保坐标系对齐：

1. **Bottom: Canvas Layer** (负责显示，`z-index: 1`)
2. **Middle: Text Layer** (负责划词，透明，`z-index: 2`)
3. **Top: Interaction Layer** (SVG/Canvas，负责波浪线，`z-index: 3`，`pointer-events: none`)

二、 重点微观代码规范 (Micro Coding Standards)

状态管理规范 (Zustand Slices)

禁止建立单一巨大的 Store。根据业务领域切分为 **Slices**：

- `usePaperStore`: 维护当前 PDF 的 `scale`，`rotation`，`currentPage`，`anchors`。
- `useChatStore`: 维护 `messages` 数组、`isThinking` 状态、`reasoningContent`。
- **约束**: 严禁在 Store 中存储非序列化的对象（如 PDF.js 的原生 Document 对象），只存储基础类型和简单 JSON。

异步与性能规范

- **Web Worker**: 凡是涉及全文匹配（AC 自动机）或大型文本清洗的操作，必须放入 `src/workers/` 下。
- **Virtualization**: PDF 列表渲染必须使用 `react-window` 或同类思想，离开视口的 `Canvas` 必须通过 `canvas.width = 0` 释放内存。
- **Effect Cleanup**: 所有的 `useEffect` 必须包含完整的 `cleanup` 函数（取消 API 请求、移除 EventListener），防止内存泄漏。

类型安全 (TypeScript)

- **No Any**: 严禁使用 `any`。所有 API 返回值、DB 表记录必须定义 `interface`。
- **Discriminated Unions**: 消息体推荐使用辨别联合类型：
- TypeScript

```
代码块
1   type Message = { role: 'user'; content: string } | { role: 'assistant';
    content: string; reasoning?: string };
```

三、 核心模块实现详述 (Core Module Specs)

PDF 渲染器逻辑 (PageRenderer.tsx)

AI 编写此组件时必须满足：

- **Props**: 接收 `pageNumber`, `scale`, `onTextLayerRendered`。
- **Logic**:
  a. 检测进入视口后再调用 `page.render()`。
  b. 渲染完成后，立即通知 `InteractionLayer` 执行波浪线比对。

划词菜单拦截逻辑

- 监听 `TextLayer` 的 `mouseup` 事件。
- 使用 `window.getSelection()` 获取选区，并计算 `rect = range.getBoundingClientRect()`。
- 若选区为空，立即销毁菜单；若不为空，计算菜单位置并显示。

四、 项目文件树规约 (Folder Structure)

```
src/
├── app/                    # Next.js App Router 路由
├── components/             # UI 视图
│   ├── m1/                 # 新手引导专属组件
│   ├── m2/                 # 核心实验室组件 (PDFView, ChatPanel, DeconBoard)
│   └── ui/                 # Shadcn UI 原子组件
├── services/               # 业务逻辑 (Class/Function)
│   ├── ai.service.ts       # 封装 DeepSeek 流式请求
│   ├── db.service.ts       # Dexie 初始化与 CRUD
│   └── search.worker.ts    # AC 自动机 Web Worker
├── store/                  # Zustand Stores
├── types/                  # .d.ts 定义
└── lib/                    # 工具函数 (PDF坐标转换, 格式化)
```
