FlowCabal 架构设计文档 v5

日期：2026.02.20

推倒重来。抛弃 Python 后端、OpenViking、浏览器 UI、SQLite。纯 TypeScript + Bun，文件系统存储，TUI 优先。

目录

设计背景

从 v4 到 v5：为什么推倒重来

v4 设计是“正确但过重”的架构。核心问题：

1. **OpenViking 强制 embedding**：向量搜索在约束空间 50-100KB 的场景下是杀鸡用牛刀。Agent 读一个索引文件就能决定加载什么，比余弦相似度更准确—Agent 理解叙事上下文
2. **多角度侧写与约束切分重合**：v4 设计了 5 种侧写类型（角色、情节线、世界状态、主题、文风），但主题是人类决策、文风是流动的，不该被固化为侧写。剩下的三个角度（角色、时间线、世界观）用纯文件就能管理
3. **Python + Browser + WS 链路过长**：三进程协调（浏览器 → WebSocket → Python + AGFS 子进程）增加了大量偶发复杂度。单进程 TypeScript 可以消除所有 IPC
4. **三角色 Agent 过早优化**：Role A/B/C 的分离在没有工作的基础执行引擎之前是空中楼阁

v5 的策略：**先让最小可用版本跑起来**，再逐步添加复杂度。

保留的核心洞察

从 v4（及更早版本）保留的设计决策：

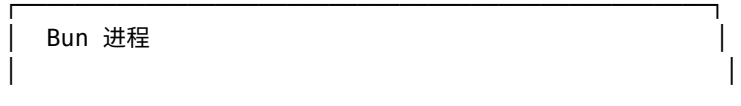
- **DAG 工作流 + 拓扑排序执行**：核心抽象不变
- **TextBlock = literal | ref**：节点间通过引用传递输出
- **L0/L1/L2 层次化上下文**：索引 → 摘要 → 全文的渐进加载
- **规约 vs 状态**：两类信息的区分（prescriptive vs descriptive）
- **人类定义 what，AI 负责 how**：创作哲学不变
- **有界上下文预算**：无论手稿多长，上下文加载有预算上限

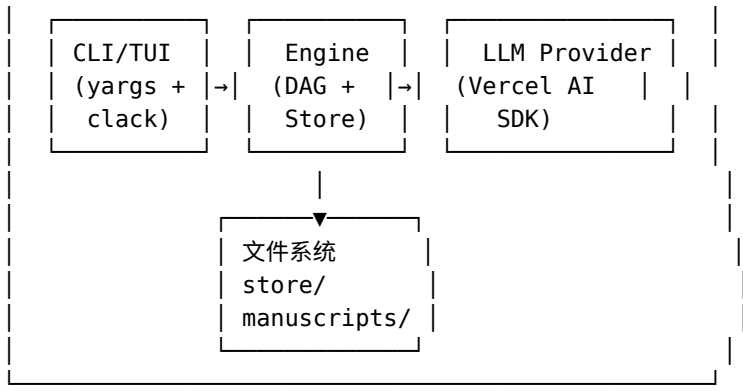
新的参考架构

来源	采纳的设计	在 v5 中的体现
OpenCode	项目骨架	TypeScript + Bun monorepo，CLI 架构，配置管理
OpenClaw	记忆设计	三角度切片 + 两层信息分类，文件系统 store
Vercel AI SDK	LLM 抽象	统一的 provider/model 接口，流式输出，tool calling

整体架构

单进程 TypeScript





没有数据库、没有 WebSocket、没有子进程。一个 Bun 进程，读写文件系统。

Monorepo 结构

```
flowcabal/
  package.json              # Bun workspace root
  packages/
    engine/                 # 核心无头引擎（零 UI 依赖）
      src/
        types.ts            # 领域类型
        schema.ts           # Zod schemas
        dag/                # Workflow DAG
          workflow.ts       # Workflow 容器 + Kahn 拓扑排序
          executor.ts       # 顺序执行（流式）
          resolve.ts        # TextBlock ref → 上游输出
        store/              # 文件系统记忆
          store.ts          # CRUD
          index-gen.ts      # 生成 index.md (L0)
          paths.ts          # 路径工具
        context/            # 上下文组装
          assembler.ts      # L0 + 按需 L1
          budget.ts         # token 估算
        llm/                # LLM 集成
          provider.ts       # Vercel AI SDK provider 工厂
          generate.ts       # generateText + streamText
        agent/              # Agent
          agent.ts          # runAgent + conversationalAgent
          tools.ts          # 5 个 Zod tool
          prompts.ts        # 中文系统提示词
    cli/                    # TUI
      src/
        index.ts            # bin: flowcabal
        config.ts           # 加载 flowcabal.json
        commands/
          init.ts           # flowcabal init
          add-chapter.ts    # flowcabal add-chapter <file>
          status.ts         # flowcabal status
          generate.ts       # flowcabal generate（对话式 REPL）
          store.ts          # flowcabal store ls|read|write|index
```

Engine 包零 UI 依赖，可以被任何前端（TUI、Web、Electron）使用。

与 v4 架构对比

维度	v4	v5
语言	Python 后端 + TypeScript 前端	纯 TypeScript + Bun
进程	Browser + WS + Python + AGFS	单进程
存储	SQLite + OpenViking (AGFS + LevelDB)	纯文件系统
检索	OpenViking 向量搜索 + HierarchicalRetriever	Agent 驱动的 L0 索引 + 按需加载
UI	Svelte 浏览器 UI	TUI (Phase 1) → Web (Phase 2)
Agent	三角色 (A/B/C)	单 Agent + tool calling
LLM 集成	直接 HTTP 调用	Vercel AI SDK
配置	SQLite + 内存	`flowcabal.json`
代码量	3000+ 行 (预估)	1200 行

类型系统

核心类型

```
// 节点 prompt 的构成单元
type TextBlock =
  | { kind: "literal"; content: string }   // 字面量文本
  | { kind: "ref"; nodeId: string };       // 引用上游节点输出

// DAG 节点定义
interface NodeDef {
  id: string;
  label: string;
  role: "user_llm" | "agent_llm";          // 使用哪套 LLM
  systemPrompt: TextBlock[];
  userPrompt: TextBlock[];
  parameters?: { temperature?: number; maxTokens?: number };
}

// DAG 边
interface Edge { id: string; source: string; target: string }

// 工作流 = 节点 + 边
interface Workflow {
  id: string; name: string;
  nodes: NodeDef[]; edges: Edge[];
}
```

`TextBlock` 是 v4 以来一直保留的核心抽象。`kind: "ref"` 实现节点间数据流——下游节点的 prompt 可以引用上游节点的输出，在执行时动态替换。

配置

```
type LlmProvider = "openai" | "anthropic" | "google" | "openai-compatible";

interface LlmConfig {
  provider: LlmProvider;
  baseURL?: string;        // openai-compatible 需要（如 DeepSeek）
  apiKey: string;
  model: string;
}

interface ProjectConfig {
  name: string;
  rootDir: string;
  userLlm: LlmConfig;      // 创作用
  agentLlm: LlmConfig;     // 元推理用（分析、提取、对话）
}
```

两套 LLM 配置从 v4 保留。创作用高质量模型，元推理用快速/廉价模型。`openai-compatible` provider 支持 DeepSeek 等第三方 API。

所有类型均有对应的 Zod schema，用于运行时校验 `flowcabal.json`。

记忆架构

设计哲学

v4 的五种侧写类型（角色、情节线、世界状态、主题、文风）在实践中存在问题：

- **主题**是人类的创作决策，不应被 LLM 提取固化——固化后反而会限制创作方向
- **文风**是流动的，尤其在长篇中可能随情节阶段变化——固化为侧写会引导 LLM 趋向平均化

保留三个角度，因为它们是**可客观提取**的事实性信息：

角度	路径	内容
角色	`characters/`	姓名、身份、外貌、性格、语癖、能力、当前状态
时间线	`timeline/`	按时间顺序的事件记录，每条简短
世界观	`world-rules/`	地名、组织、体系、规则——合并成设定清单

另有 `plot/`（大纲）作为规约的一部分。

两类信息

类型	含义	来源	Store 位置
规约（prescriptive）	生成前就存在的约束	人维护，LLM 辅助起草	`constraints/`
状态（descriptive）	从已定稿章节提取的事实	Agent 从手稿提取	`state/`

这个区分来自 OpenClaw 的“策展持久化哲学”。规约是**先验**的——作者在写作前建立的世界规则和角色设定。状态是**后验**的——从已完成的章节中提取的事件和角色变化。

两者对 LLM 的约束力不同：规约是硬约束（“这个世界没有枪械”），状态是软约束（“截至第 5 章，主角在城堡中”—但下一章可以离开）。

Store 目录结构

```
<project>/
  flowcabal.json
  manuscripts/              # 手稿文件
  store/
    constraints/            # 规约（先验）
      characters/           # 角色卡
      world-rules/          # 设定清单
      plot/                 # 大纲
    state/                  # 状态（后验）
      timeline/             # 每章事件线
      character-status/     # 当前快照
    index.md                # L0 自动生成
```

纯 Markdown 文件。没有数据库，没有二进制格式。用户可以直接用文本编辑器查看和修改。

上下文加载策略

核心假设：约束空间很小。一部百万字小说的全部角色卡 + 世界规则 + 大纲 + 时间线，大约 50-100KB（ 15-30K token）。这个规模完全可以直接加载，不需要向量搜索。

三级加载：

级别	内容	大小	何时加载
L0	index.md—每条一行摘要	2-3K token	始终注入 system prompt
L1	单个条目的完整内容	按需	Agent 读 L0 后决定加载哪些
L2	手稿原文	按需	Agent 需要引用具体段落时

```
Agent 视角：

1. 收到 system prompt，其中包含 L0 索引：
   - [constraints/characters/mirael.md] 制图师，女性，固执
   - [constraints/characters/davan.md] 制图师公会成员，mirael的同事
   - [state/timeline/ch7.md] 第七章事件线
   - ...

2. Agent 根据当前任务决定需要哪些完整文件：
   → read_store("constraints/characters/mirael.md")
   → read_store("state/timeline/ch7.md")

3. 有了完整上下文后执行任务
```

为什么这比向量搜索好：

- **Agent 理解叙事上下文**：写第 8 章时，Agent 知道需要第 7 章的时间线和出场角色，这不是余弦相似度能捕捉的
- **零基础设施**：不需要 embedding 模型、向量数据库、索引构建
- **可调试**：index.md 是纯文本，用户可以直接阅读理解 Agent 看到了什么
- **足够快**：文件系统读取 50 个小文件是微秒级操作

index.md 格式

```
# Store Index

- [constraints/characters/mirael.md] 制图师，主角
- [constraints/characters/davan.md] 制图师公会成员
- [constraints/world-rules/pale-reach.md] 苍白之境——正在向南推进的神秘冰川
- [constraints/plot/outline.md] 全篇大纲
- [state/timeline/ch7.md] 第七章：Mirael 到达 Thornwall
- [state/character-status/mirael.md] 在 Thornwall，准备前往苍白之境
```

由 `index-gen.ts` 自动生成：取每个 Markdown 文件的第一行（去掉 `#` 前缀）作为摘要。Agent 每次写入 `store` 后应调用 `update_index` 刷新。

DAG 执行引擎

工作流模型

工作流是有向无环图（DAG）。节点是 LLM 调用，边是数据依赖。

```
[Node A: 分析章节] ──→ [Node B: 生成大纲] ──→ [Node C: 写正文]
```

Node B 的 userPrompt 可以包含 `{ kind: "ref", nodeId: "A" }`，执行时自动替换为 Node A 的输出。

拓扑排序 + 顺序执行

1. Kahn 算法拓扑排序 → 得到合法执行顺序
2. 按序执行每个节点：
   a. 解析 TextBlock（literal 保留，ref 替换为上游输出）
   b. 根据 node.role 选择 LLM 配置（userLlm 或 agentLlm）
   c. 调用 LLM（支持流式输出）
   d. 存储输出，供下游节点引用
3. 返回所有节点输出

循环检测内置于 Kahn 排序——如果排序结果节点数少于总节点数，说明存在环，直接报错。

流式执行

Executor 支持回调：

- `onNodeStart(node)` — 节点开始执行
- `onNodeStream(node, chunk)` — 流式输出片段
- `onNodeEnd(node, output)` — 节点执行完成

TUI 通过这些回调实时显示生成过程。

LLM 集成

Provider 工厂

```
// provider.ts — 根据配置创建 Vercel AI SDK provider
function getProvider(config: LlmConfig) {
  switch (config.provider) {
    case "openai":        return createOpenAI({ apiKey });
    case "openai-compatible": return createOpenAI({ apiKey, baseURL });
    case "anthropic":     return createAnthropic({ apiKey });
```

```
    case "google":        return createGoogleGenerativeAI({ apiKey });
  }
}
```

`openai-compatible` 复用 `@ai-sdk/openai`，只是覆盖 `baseURL`。DeepSeek、Moonshot、零一万物等国产模型都走这个路径。

生成接口

两个函数：

- `generate()` — 非流式，返回完整文本。用于 Agent 的 tool calling 循环
- `streamGenerate()` — 流式，通过回调逐块输出。用于创作生成

底层都是 Vercel AI SDK 的 `generateText` / `streamText`，统一处理了 provider 差异。

Agent 系统

从三角色到单 Agent

v4 设计了三个专门角色：

- Role A: 上下文组装
- Role B: 工作流构建
- Role C: 事实检查

v5 简化为**单个 Agent + tool calling**。理由：

1. 上下文组装（原 Role A）现在就是 Agent 调用 `read_store` 工具——不需要单独的角色
2. 工作流构建（原 Role B）在 TUI 阶段不需要——工作流由用户定义
3. 事实检查（原 Role C）可以作为 Agent 的指令之一，不需要独立角色

当规模增大到需要专门角色时，再分拆。符合“先让它跑起来”的原则。

工具集

Agent 拥有 5 个工具，通过 Vercel AI SDK 的 tool calling 机制调用：

工具	参数	功能
`list_store`	无	列出 store 中所有条目路径
`read_store`	`path`	读取指定条目的完整内容
`write_store`	`path, content`	写入或更新条目（Markdown）
`read_manuscript`	`filename`	读取手稿文件
`update_index`	无	重新生成 `index.md`（L0 索引）

工具定义使用 Zod schema，自动生成 JSON Schema 供 LLM 理解参数格式。

两种 Agent 模式

单次 Agent（runAgent）

用于自动化任务，如分析章节：

```
用户: flowcabal add-chapter chapter7.md
  → Agent 收到章节内容
  → list_store() 查看现有记忆
  → 分析章节，提取角色/事件/设定
  → write_store() 写入多个条目
  → update_index() 刷新索引
  → 返回分析报告
```

Agent 最多执行 20 步（tool calling 循环），防止无限执行。

对话 Agent（conversationalAgent）

用于交互式创作（`flowcabal generate`）：

```
用户: "帮我构思第八章的开头"
  → Agent 读取 store 中的角色和时间线
  → 结合上下文生成建议
  → 用户继续对话，逐步精炼
```

对话 Agent 维护消息历史，支持流式输出。

系统提示词

三套中文系统提示词：

- **分析模式**：指导 Agent 从章节中提取角色信息、世界观设定、情节大纲、时间线、角色状态
- **生成模式**：指导 Agent 读取 store 辅助创作，保持角色一致性和世界观连续性
- **对话模式**：通用创作助手，支持讨论和修改 store

CLI（TUI）

命令一览

命令	参数	功能
`flowcabal init`	`[name]`	交互式创建项目（选择 LLM、输入 API key）
`flowcabal add-chapter`	`<file>`	添加章节到 manuscripts/ + Agent 自动分析
`flowcabal status`	无	显示手稿数、store 条目数、index.md 内容
`flowcabal generate`	无	对话式创作 REPL
`flowcabal store`	`ls|read|write|index [path]`	直接管理 store 条目

交互设计

TUI 使用 `@clack/prompts` 提供美观的终端 UI（spinner、选择框、文本输入）。核心操作是命令式的，创作环节（`generate`）是对话式的。

```
$ flowcabal init mynovel
◆  FlowCabal 项目初始化
|
```

```
◇  选择 LLM 提供商
│  OpenAI Compatible (DeepSeek 等)
│
◇  API Key
│  sk-***
│
◇  模型名称
│  deepseek-chat
│
◆  项目创建完成
│
│  下一步:
│  cd mynovel
│  flowcabal add-chapter <file>
│  flowcabal status
│  flowcabal generate
│
└  开始创作吧!
```

架构决策记录

为什么抛弃 OpenViking

问题	分析
强制 embedding	约束空间 50-100KB，向量搜索是 O(n) 的，而 Agent 读索引是 O(1) 的决策
多角度切分重合	OpenViking 的多级信息模型和 FlowCabal 的侧写系统功能重叠
依赖重量	OpenViking 引入 AGFS 子进程 + LevelDB + embedding 模型—三层间接依赖
调试困难	向量检索是黑盒，Agent 驱动的文件读取完全可追踪

为什么纯文件系统而非 SQLite

维度	分析
简单性	Markdown 文件可直接阅读、编辑、版本控制
Git 友好	纯文本文件天然支持 diff、merge、history
足够	50-100 个小文件的 CRUD 不需要数据库
可升级	如果未来需要，可以在文件系统之上加索引层

SQLite 的优势（事务、查询）在当前规模下不需要。如果 store 增长到数千个文件（不太可能——一部小说的约束空间是有限的），再考虑引入。

为什么 Vercel AI SDK

维度	分析

统一接口	一套代码支持 OpenAI / Anthropic / Google / 兼容 API
Tool calling	内置 Zod schema → JSON Schema 转换，自动处理多步 tool calling 循环
流式输出	`streamText` 提供开箱即用的 AsyncIterable
TypeScript 原生	类型安全，与项目技术栈一致

为什么 TUI 优先

Phase 1 是 headless engine + TUI。理由：

1. **引擎优先**：先确保核心逻辑（DAG 执行、store 管理、Agent tool calling）正确工作
2. **零前端依赖**：不需要 Svelte、React、打包器、浏览器——开发速度快
3. **对话式交互天然适合终端**：`flowcabal generate` 就是一个 REPL
4. **引擎可复用**：TUI 和未来的 Web UI 共享同一个 engine 包

Phase 2（Web UI / 可视化 DAG 编辑器）在引擎稳定后再开发。

与 v4 设计的关系

模块	**v4**	**v5**
语言	Python + TypeScript	纯 TypeScript + Bun
存储	SQLite + OpenViking	纯文件系统（Markdown）
检索	OpenViking 向量搜索	Agent 驱动的 L0 索引
侧写	5 种多角度侧写	3 角度切片（角色/时间线/世界观）
Agent	三角色（A/B/C）	单 Agent + 5 工具
UI	Svelte 浏览器	TUI（Phase 1）
通信	WebSocket	直接函数调用
配置	SQLite + 内存	`flowcabal.json`
LLM	直接 HTTP	Vercel AI SDK
TextBlock	保留	保留
DAG 执行	保留（Python）	保留（TypeScript）
L0/L1/L2	OpenViking 自动生成	index-gen.ts + Agent 按需加载
规约/状态	保留	保留
Prompt 组装	三层模型	resolveBlocks（literal + ref）
策展管线	两级模型 + 异步侧写	手动 store 管理 + Agent 分析

递归/进化	概念设计	推迟
代码量	3000+ 行（预估）	1200 行

v4 的核心抽象（TextBlock、DAG、L0/L1/L2、规约/状态）全部保留。变化的是实现方式—从分布式多进程架构变为单进程文件系统架构。

实施路线

Phase 1：Headless Engine + TUI（当前）

已完成。 1200 行 TypeScript，27 个文件。

- Monorepo 骨架（engine + cli）
- 类型系统 + Zod schema
- 文件系统 store CRUD + L0 索引生成
- DAG 拓扑排序 + 执行器（流式）
- Vercel AI SDK provider 工厂
- Agent（单次 + 对话）+ 5 工具 + 中文提示词
- CLI 5 个命令

Phase 2：可视化 DAG 编辑器（未来）

在 engine 稳定后开发 Web 前端：

- 可视化工作流编辑（拖拽节点、连线）
- 实时执行可视化（节点状态、流式输出）
- Store 浏览器（查看/编辑记忆条目）
- 工作流模板（常用创作模式的预设 DAG）

Engine 包作为 Web 前端的后端，通过直接 import 或 API 层连接。

Phase 3：高级能力（未来）

视实际使用情况决定是否需要：

- 多 Agent 角色分拆（回到 Role A/B/C，如果单 Agent 不够用）
- 递归调用 + 进化式迭代（v4 设计的高级工作流组件）
- 策展管线（自动一致性检查、异步侧写更新）
- 导入/导出（与其他写作工具互通）