FlowCabal 架构设计文档 v5

日期：2026.02.22

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
3. **Python + Browser + WS 链路过长**：三进程协调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**：节点 prompt 的构成单元，通过引用传递输出
- **L0/L1/L2 层次化上下文**：索引 → 记忆文件 → 原稿的渐进加载
- **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

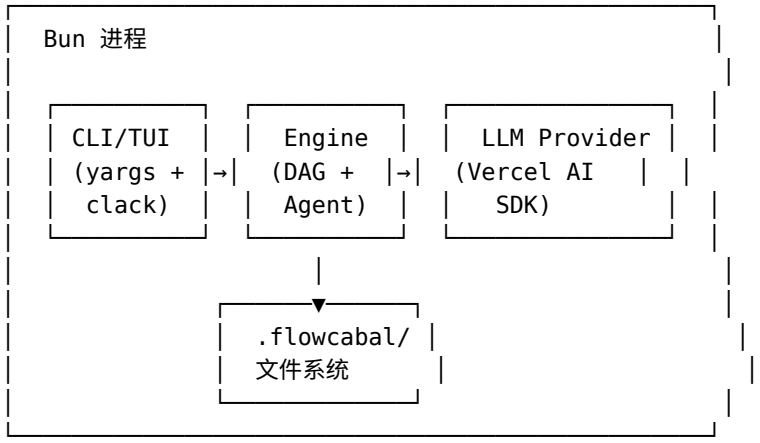
已废弃的旧概念

以下概念在设计演化中被证明不再需要：

- **冻结节点输出（freeze）**：v1-v3 中用于在交互式 UI 里锁定下游节点不被上游更新覆盖。v5 是 TUI/CLI + 单次 DAG 执行，不存在"上游重跑、下游要保护"的场景
- **环形节点**：迭代精修（生成→评估→重新生成）在 Agent 的推理内部完成，不需要 DAG 环。Kahn 算法天然拒绝环
- **规约/状态分层（prescriptive/descriptive）**：人和 Agent 共同读写所有记忆文件，不需要区分先验/后验
- **显式 Edge 类型**：从 TextBlock `kind: "ref"` 隐式推导即可
- **workflow 级 state.json**：被 per-node 增量缓存取代

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（Workflow 元数据层）
        schema.ts           # Zod schemas（运行时校验）
        dag/                # DAG 执行
          workflow.ts       # Kahn 拓扑排序
          executor.ts       # 增量构建执行器
          resolve.ts        # TextBlock 解析
        store/              # 文件系统存储
          paths.ts          # 路径定义（含完整路径树）
          store.ts          # CRUD
          index-gen.ts      # 生成 index.md (L0)
        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          # Agent 实现
          tools.ts          # Zod tool 定义
          prompts.ts        # 中文系统提示词
    cli/                    # TUI
      src/
        index.ts            # bin: flowcabal
        config.ts           # 配置加载
        commands/
```

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

核心概念：三层解耦

Workflow / Workspace / Project

三个概念彼此解耦：

概念	存储位置	职责
Workflow	data/workflows/	纯模板/蓝图。只描述节点结构和 prompt 组合方式（TextBlock[]），不含 LLM 配置。用于朋友间分享工作流
Project	memory/<project>/	小说项目。拥有独立的 Agent 记忆（角色、世界、文体、定稿章节）
Workspace	runner-cache/<workspace-id>/	Workflow 的一次实例化运行环境。绑定一个 project，存执行缓存。用户在其中反复调试。删除即释放全部缓存

- 同一个 Workflow 可以被多个 Workspace 实例化
- 同一个 Project 可以在不同 Workspace 中使用不同的 Workflow
- Workflow 是**分享**的单元，Workspace 是**工作**的单元

用户偏好

per-node LLM 覆盖等用户个性化配置存在 data/preferences/<workflow-id>.json，跨工作区生效。这解决了“每次新建 workspace 都要重配 LLM”的问题，同时不污染 workflow 模板的纯净性。

类型系统

TextBlock

节点 prompt 的构成单元，三种类型：

```
type TextBlock =
  | { kind: "literal"; content: string }
  | { kind: "ref"; nodeId: string }
  | { kind: "agent-inject"; hint: string };
```

- literal – 静态文本，用户直接编写
- ref – 引用上游节点输出，执行时动态替换
- agent-inject – Agent 注入点。hint 告诉 Agent 方向，Agent 读 L0 自主决定注入什么内容。一个节点可以有多个注入点，出现在 prompt 的不同位置

ref 同时定义了 DAG 的隐式连接关系—不需要显式 Edge 类型。

NodeDef

```
interface NodeDef {
  id: string;
  label: string;
  systemPrompt: TextBlock[];
  userPrompt: TextBlock[];
}
```

节点不存储 LLM 配置。LLM 选择在运行时由以下优先级决定：
1. `data/preferences/<workflow-id>.json` 中的 per-node 覆盖
2. `data/llm-configs.json` 中的 default 配置

Workflow

```
interface Workflow {
  id: string;
  name: string;
  nodes: NodeDef[];
}
```

纯模板。没有 edges（从 ref 隐式推导），没有 LLM 配置，没有运行状态。

LLM 配置

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

interface LlmConfig {
  name: string;
  provider: LlmProvider;
  baseURL?: string;
  apiKey: string;
  model: string;
}
```

用户在 `data/llm-configs.json` 中维护多套 LLM 配置，按 `name` 引用，其中一套为 `default`。`openai-compatible` 支持 DeepSeek 等第三方 API。

所有类型均有对应的手写 Zod schema（`schema.ts`），用于运行时校验。types.ts 与 schema.ts 各自维护，不用 `z.infer`。

记忆架构

设计哲学

约束查询是核心能力

记忆模块的底层能力是**约束查询**：给出新内容，判断它与已有章节有没有冲突、冲突在哪里。

关键洞察：**约束查询和上下文注入是同一个能力的两面。** Agent 拿节点 prompt + 上游输出作为锚点去 memory 做约束查询，查到的相关约束就是该注入的上下文。不存在独立的“检索阶段”和“生成阶段”。

Memory 是 manuscripts 的有损缓存

Memory 文件不是独立的知识库，而是 manuscripts（定稿章节）的**有损缓存**。类比 coding agent：代码库是 memory，grep 是检索。对于 FlowCabal：`manuscripts/` 是完整信息源，memory 文件是为了避免每次都加载全部原文的缓存层。

不用 RAG

RAG 的 embedding 召回基于语义相似性，但小说中语义相似的段落可能有几十上百种，召回无法区分哪个是当前需要的。Memory 的跳转链接是**因果关系驱动**的检索，比语义相似性更准确。

种子文件

init 时只创建有**真实约束力**的文件——即 Agent 无法从 manuscripts 自动衍生的东西：

文件	约束域	内容
index.md	导航	L0 索引，Agent 的导航入口。可含一句话主题内核
characters.md	角色一致性	生成性事实：背景因果→性格→动机→关系。写因果链，不写特征列表
world.md	设定一致性	世界硬规则、体系原理、边界。含类型设定约束（如“硬科幻：无魔法”）
voice.md	叙事一致性	POV、文体、句法模式。需要句法级正例和反例，不能只是抽象标签。含类型叙事约束
manuscripts/	全精度信息源	定稿章节。L2 层，通过跳转链接按需可达

为什么不是更多文件

以下文件在讨论中被明确排除：

- premise.md — 梗概是高层 outline（作者意图，约束力不可靠）；类型拆入 world + voice 各一行；主题内核太抽象，index.md 一句话够了
- outline.md — 纯作者意图，最善变，无约束力。作者意图是所有东西中约束能级最低的
- chronicle.md — 初始为空，是 Agent 从 manuscripts 按需衍生的缓存，不该是种子
- threads.md — 初始为空，已落笔伏笔的义务，Agent 在写作过程中自行创建

设计原则：init 时只播种 Agent 无法从 manuscripts 自动衍生的东西。其余文件在写作过程中由 Agent 自行创建，index.md 维护导航。

生成性事实 vs 派生断言

Memory 文件应写**生成性事实**（intensional），不写**派生断言**（extensional）。

- 好："奥托对卡莲之死的愧疚驱动他所有决策" — 一条生成规则，可以推导出无限多具体行为
- 坏："奥托不会放弃复活卡莲 / 奥托不会信任陌生人 / 奥托说话彬彬有礼" — 试图穷举无限集合

上下文加载

三级加载：

级别	内容	何时加载
L0	index.md — 导航入口	Agent 每次启动时
L1	各记忆文件（characters.md 等）	Agent 读 L0 后按需加载
L2	manuscripts/ 原文	通过跳转链接按需可达

文件间通过**稀疏跳转链接**构成有向图，Agent 按需导航。跳转链接可指向 manuscripts/，Agent 用 SubAgent 递归探索以管理上下文。

人和 Agent 共同读写

所有记忆文件对人和 Agent 完全开放，不区分 prescriptive/descriptive。用户可以直接用文本编辑器修改 memory 文件，Agent 也可以在写作过程中创建新文件或更新已有文件。

Runner-Core：增量构建引擎

执行模型

Runner-core 是一个**增量构建系统**，以 node 为粒度缓存：

```
遍历 DAG（Kahn 拓扑序）→ 对每个节点：
  1. 解析 TextBlock[]（literal 保留，ref 替换为上游输出）
  2. 计算 resolved prompt 的 hash（仅 literal + ref 部分）
  3. 与缓存的 prompt hash 比对
  4. 匹配 → 跳过执行，使用缓存输出
  5. 不匹配 → 调用 LLM，缓存新输出 + prompt hash
```

失效自动级联：节点 A 的输出变了 → 节点 B 引用了 A（ref）→ B 的 resolved prompt 变了 → hash 不匹配 → B 重跑 → 依此类推。

这个模型的动机是**用户调试工作流的实际行为**：改一个节点的 prompt → 跑一下看效果 → 再改 → 再跑。以 node 为粒度缓存意味着只有受影响的节点需要重跑。

缓存二维失效

结构性失效（自动）

节点的 literal + ref 部分解析后的 prompt hash 变化 → 必须重跑，自动级联下游。

上下文过期（预警）

agent-inject 的结果单独缓存（因为 Agent 查询比较贵），但引入了非确定性——同样的 prompt，Agent 两次注入的内容可能不同（memory 可能已更新）。

处理方式：当 project 的 memory 或 manuscripts 被修改时，所有尚存的工作区（workspace）收到预警：**“当前缓存的 agent-inject 项可能已经不再可靠”**。用户决定是否对特定节点重新触发 agent-inject。

per-node 缓存结构

```
{
  "promptHash": "abc123",
  "agentInjects": {
    "inject-point-id": {
      "content": "Agent 注入的上下文...",
      "timestamp": 1708600000
    }
  },
  "output": "节点的 LLM 输出..."
}
```

promptHash 只覆盖 literal + ref 部分。agent-inject 独立缓存、独立追踪时间戳。

Workspace 生命周期

Workspace 只有两个状态：**存在**或**不存在**。

· 创建：用户实例化一个 workflow + project 的组合

· 使用：反复调试，增量构建
· 删除：释放全部缓存，不可恢复

不存在“归档”或“关闭”状态。所有存在的 workspace 都是活跃的，都接收上下文过期预警。

.flowcabal/ 目录结构

```
.flowcabal/
├── data/                                # 持久化配置（跨项目、跨工作区）
│   ├── llm-configs.json                 # LLM 配置池（多套，按名引用，一套 default）
│   ├── workflows/                       # Workflow 模板（纯蓝图，用于分享）
│   │   └── <workflow-id>.json
│   └── preferences/                     # 用户对模板的个性化配置
│       └── <workflow-id>.json           # per-node LLM 覆盖等偏好，跨工作区生效
├── memory/                              # Agent 记忆（按项目隔离）
│   └── <project>/
│       ├── index.md                     # L0 导航
│       ├── characters.md                # 角色生成性事实
│       ├── world.md                     # 世界硬规则 + 类型设定约束
│       ├── voice.md                     # 文体约束 + 类型叙事约束
│       └── manuscripts/                 # L2 完整信息源（定稿章节）
└── runner-cache/                        # 工作区（按 workspace 隔离，删除即释放）
    └── <workspace-id>/
        ├── meta.json                    # { workflowId, projectId, createdAt }
        └── outputs/
            └── <node-id>.json           # { promptHash, agentInjects, output }
```

唯一一个状态目录，位于仓库根目录。免安装分发，不污染用户 home 空间。

路径定义见 `packages/engine/src/store/paths.ts`（文件顶部注释有同样的路径树）。

LLM 集成

Provider 工厂

```ts
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 等国产模型走这个路径。

生成接口

· `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`。

Agent 系统

单 Agent + Tool Calling

v4 的三个专门角色（Role A/B/C）简化为**单个 Agent + tool calling**。理由：

1. 上下文组装（原 Role A）= Agent 通过约束查询自主完成
2. 工作流构建（原 Role B）= 用户定义 workflow 模板
3. 事实检查（原 Role C）= 约束查询的自然副产物

当规模增大到需要专门角色时，再分拆。

agent-inject 机制

agent-inject 是 FlowCabal 的核心差异化能力。当 runner-core 遇到 `kind: "agent-inject"` 的 TextBlock 时：

1. 将节点 prompt + 上游输出（ref 已解析的部分）作为锚点
2. Agent 读 L0 索引，按需导航到 L1/L2
3. 执行约束查询：判断当前上下文与已有内容的关联和潜在冲突
4. 查询结果即为注入内容—约束查询和上下文注入是同一个动作

agent-inject 的结果被缓存在 workspace 中，与 prompt hash 独立追踪。

系统提示词

中文系统提示词，指导 Agent 读取记忆、执行约束查询、辅助创作。

架构决策记录

为什么不用 RAG

问题	分析
语义歧义	小说中语义相似的段落可能有几十上百种，embedding 召回无法区分哪个是当前需要的
因果盲区	伏笔和回收在语义空间里往往很远，余弦相似度捕捉不到因果链
事实幻觉	召回错误段落导致的不是普通幻觉，而是事实性幻觉—基于错误上下文的逻辑自洽输出
基础设施	不需要 embedding 模型、向量数据库、索引构建

Agent 驱动的导航（L0 索引 + 跳转链接）比语义相似性更准确，因为链接是因果关系驱动的。

为什么纯文件系统

维度	分析
可读性	Markdown 文件可直接阅读、编辑—人和 Agent 共同读写的前提
Git 友好	纯文本文件天然支持 diff、merge、history

足够	50 个小文件的 CRUD 不需要数据库
零依赖	不需要 SQLite、LevelDB 或任何外部存储

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

为什么增量构建而非全量执行

用户调试工作流的实际行为是：改一个节点 → 跑一下 → 再改 → 再跑。全量执行意味着每次修改都重跑所有节点（浪费 token + 时间）。以 node 为粒度的增量构建只重跑受影响的节点，缓存失效自动级联。

为什么 agent-inject 缓存需要预警而非自动失效

agent-inject 的 Agent 查询比较贵（需要多次 tool calling）。自动失效意味着每次 memory 变更都要重新执行所有 agent-inject——这在频繁编辑记忆时代价太高。预警机制让用户自主决定何时重新触发，平衡了**正确性**和**成本**。

实施路线

Phase 1：Headless Engine + TUI（当前）

- Monorepo 骨架（engine + cli）
- 类型系统 + Zod schema
- Memory 种子文件初始化
- DAG 拓扑排序 + 增量构建执行器
- Vercel AI SDK provider 工厂
- Agent + tool calling + 中文提示词
- CLI 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）
- Memory 浏览器（查看/编辑记忆文件）
- Workspace 管理（创建、切换、删除）
- 上下文过期预警 UI

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（如果单 Agent 不够用）
- 策展管线（自动一致性检查）
- 导入/导出（与其他写作工具互通）