FlowCabal 参考设计分析

哪些设计值得采纳、为什么、以及价值如何迁移

2026 年 2 月 17 日

1 概述

本文分析了三个参考项目中可迁移至 FlowCabal 的设计模式：

项目	领域	相关度
OpenViking	AI 智能体上下文管理——虚拟文件系统、分层信息、层次化检索	直接相关
OpenClaw 记忆系统	AI 智能体持久记忆——时间压缩、蒸馏、有界存储	结构相关
Trellis	AI 开发工作流编排——规范注入、质量门禁、多智能体钩子	类比相关

FlowCabal 是一个面向 AI 辅助长篇写作的可视化工作流编辑器。它的核心挑战是：*AI* 智能体如何为一部 *8* 万字小说的第 N 章组装正确的上下文？ 三个参考项目各自解决相关但不同的问题。本文提取最根本的设计，按其对 FlowCabal 核心挑战的直接程度排列优先级。

设计分为三个层级：

- **第一层**：直接解决 FlowCabal 核心问题的设计。完整采纳。
- **第二层**：显著改善子系统的设计。适当改造后采纳。
- **第三层**：值得内化为原则的模式。在自然契合处应用。

2 第一层：完整采纳

2.1 三级信息模型（OpenViking：L0 / L1 / L2）

2.1.1 原始价值

OpenViking 将每个资源存储为三个粒度层级：

层级	名称	内容
L0	摘要	一句话描述（< 50 token）
L1	概览	结构化摘要：章节标题、要点（200–500 token）
L2	完整内容	原始全文

这解决了 RAG 系统中的一个根本问题：检索完整文档消耗大量 token，而仅检索向量嵌入则丢失结构化上下文。通过实体化三个层级，OpenViking 让智能体能够在 *L0* 广泛扫描、在 *L1* 缩小范围、在 *L2* 深入阅读——一种模仿人类浏览图书馆的渐进加载策略。

三个层级在资源入库后异步生成。Observer 组件监视新资源并在后台队列中触发基于 VLM 的摘要生成。这意味着入库很快（只存储 L2），语义层是懒加载构建的。

2.1.2 迁移至 FlowCabal

这是对 FlowCabal 价值最高的设计。Role A（上下文智能体）必须决定一部 200 章手稿中哪些部分与当前节点相关。没有分层信息，Role A 只能：

- 读取全部内容（不可能：炸掉上下文窗口），或
- 仅依赖向量相似度（不充分：遗漏结构关系）。

有了 L0/L1/L2，Role A 的检索变为：

```
1. 扫描全部章节 L0（每章 < 50 token，~200 章 = ~10K token）
2. 读取有前景章节的 L1（~5 章 x 400 token = 2K token）
3. 深读最相关的 1--2 章的 L2
4. 始终包含：实体文件、风格指南、世界规则（这些很小）
```

总上下文预算： 15–20K token，而非 500K+ 的完整手稿。

变化之处：在 OpenViking 中，L0/L1/L2 由通用 VLM 生成。在 FlowCabal 中，摘要应由 Agent LLM 生成，且当某章的策展输出更新时应重新生成。摘要不是一次性的入库步骤——它是策展管线的一部分。

具体映射：

OpenViking 虚拟路径	FlowCabal 应用
`/manuscript/chapter-N/content.md`	L2：第 N 章的策展输出
`/manuscript/chapter-N/summary-paragraph.md`	L1：段落摘要
`/manuscript/chapter-N/summary-sentence.md`	L0：一句话摘要
`/manuscript/chapter-N/entity-changes.json`	第 N 章后的实体状态变化
`/entities/characters/*.md`	累积的角色档案

/summaries/arc-*.md	多章节叙事弧线摘要
/meta/style-guide.md	写作风格约束

2.2 层次化检索与意图分析（OpenViking）

2.2.1 原始价值

OpenViking 不使用扁平向量搜索。其 HierarchicalRetriever 结合了两种策略：

1. **全局向量搜索**：在整个存储中查找语义相似的资源。
2. **递归目录遍历**：给定目标目录 URI，遍历树结构，使用 L0/L1 摘要决定深入探索哪些子树。

在检索之前，IntentAnalyzer 对查询进行分类，确定需要什么类型的上下文——而不仅仅是什么文本相似。这是“查找关于龙的段落”（相似性搜索）和“查找写主角与反派对话场景所需的上下文”（意图驱动检索）之间的区别。

两阶段架构（向量召回 + 重排序）进一步将快速候选检索与慢但精确的相关性评分分离。

2.2.2 迁移至 FlowCabal

Role A 的上下文检索是 FlowCabal 性能最关键的操作。每个节点执行都以 Role A 决定注入什么上下文开始。层次化检索模式直接映射：

```
# Role A 的检索管线
def get_context(node_id, node_config, project):
    # 步骤 1：意图分析
    intent = agent_llm.analyze_intent(node_config, project.meta)
    # 例如："对话场景，第 12 章，角色：[主角，反派]"

    # 步骤 2：确定性包含（始终相关）
    ctx = [project.meta.style_guide, project.meta.outline]

    # 步骤 3：实体查找（结构化，非语义）
    for char in intent.characters:
        ctx.append(project.entities.get(char))

    # 步骤 4：层次化手稿搜索
    #   扫描全部章节 L0 → 阅读有前景的 L1 → 深读最佳 L2
    ctx.extend(hierarchical_retrieve(intent, project.manuscript))

    # 步骤 5：弧线摘要提供更广的叙事上下文
    ctx.extend(project.summaries.get_relevant_arcs(intent))

    return ctx
```

变化之处：OpenViking 的检索是通用的（任何文档类型）。FlowCabal 的检索是领域特定的（小说手稿）。这是一个优势：Role A 可以利用关于“写第 N 章需要什么”的结构化知识（前一章、相关角色弧线、世界规则），而非仅依赖语义相似度。因为领域受限，意图分析变得更简单、更可靠。

2.3 策展持久化：并非所有信息都值得存储（OpenClaw + FlowCabal v3）

2.3.1 原始价值（OpenClaw）

OpenClaw 的 4D 验证测试要求每个候选记忆条目同时满足：

1. **错误预防**：没有这条信息，智能体会犯一个具体的、可识别的错误。
2. **广泛适用**：该信息适用于许多未来会话，而非仅一个任务。

3. **自包含**：该条目无需额外上下文即可理解。
4. **非重复**：该信息尚未存在于存储中。

加上一个**反向检查**："没有这条信息会发生什么具体错误？"如果答不上来，拒绝该条目。

这不仅仅是一个过滤器——它是一种持久化哲学。系统默认所有信息都是临时的，只将最有价值的部分提升到长期存储。

2.3.2 迁移至 FlowCabal

FlowCabal 的 v3 设计中已有这个原则：只有用户批准的输出才进入 SQLite 并触发 OpenViking 索引。但策展的机制可以借鉴 OpenClaw 的验证框架来丰富。

当用户选择 `output:persist` 时，系统不应盲目存储。Role C（监控者）或专门的策展步骤应该：

1. 验证输出是否足够自包含，可作为未来上下文使用。
2. 检查与现有策展内容的矛盾（例如，角色的眼睛颜色在章节间发生了非作者意图的变化）。
3. 立即生成 L0/L1 摘要。
4. 更新实体变更记录。

4D 测试适配 FlowCabal：

维度	**FlowCabal 解读**
错误预防	这段内容能否防止未来章节中的连续性错误?
广泛适用	这段内容是否会被多个未来节点引用?
自包含	这段内容能否脱离完整执行上下文被理解?
非重复	这段内容是否在已策展内容之外增加了新的叙事信息?

变化之处：OpenClaw 验证的是智能体关于用户的记忆。FlowCabal 验证的是创意写作输出。验证标准从"智能体会犯什么错误？"转变为"未来生成会产生什么连续性错误？"根本原则——激进的过滤保持信号质量——是相同的。

2.4 规范注入优于记忆（Trellis）

2.4.1 原始价值

Trellis 的核心洞察：与其希望 AI 记住对话早期的指令，不如在每次智能体调用时重新注入所有相关规范。三个钩子实现了这一点：

- **会话启动钩子**：注入全局项目上下文。
- **子智能体上下文钩子**：拦截每次智能体调度，注入任务特定的规范。
- **质量控制钩子**：在允许智能体停止前强制执行标准。

每次智能体调用都变得自包含——无论对话历史如何，它在 prompt 中拥有所需的一切。这消除了上下文漂移——随着对话增长，指令遵循能力逐渐退化的现象。

2.4.2 迁移至 FlowCabal

FlowCabal 的三层 prompt 组装在结构上与 Trellis 的注入模式相同：

Trellis	**FlowCabal**	**机制**

会话启动钩子	第 1 层：用户的 TextBlockList	执行开始时加载的静态规范
子智能体上下文钩子	第 2 层：Role A 上下文注入	每节点动态注入的上下文
质量控制钩子	Role C 评估	执行后质量门禁

需要内化的关键原则：**智能体上下文注入必须是临时的**。FlowCabal 的设计已经规定“Agent 不修改持久化的 metadata——上下文注入是临时的”。这正是 Trellis 的模式：每次注入新鲜内容，永不修改源数据。

实践含义： 实现 Role A 时，上下文注入应该是一个纯函数：`(node_config, project_state) -> additional_context`。它读取项目状态但永不写入。唯一的写入通过策展管线（用户批准的输出）或 Role B（工作流拓扑变更，同样需要用户批准）发生。

这还意味着**在相同项目状态下重新运行节点应产生相同的上下文注入**。上下文组装对其输入是确定性的，即使 LLM 输出是随机的。

3 第二层：改造后采纳

3.1 虚拟文件系统作为知识组织（OpenViking）

3.1.1 原始价值

OpenViking 将所有上下文组织为具有确定性路径的虚拟文件系统。每个资源都有一个 `viking://` URI。文件系统范式提供了：

- **可预测的寻址**：智能体可以程序化地构造路径（如 `viking://resources/characters/{name}`）。
- **层次化组织**：目录将相关资源分组，支持基于树的检索。
- **统一接口**：`ls`、`read`、`stat`、`find`——智能体天然理解的操作。

这不仅仅是命名约定。目录结构编码了领域知识，表达信息之间的关联方式。`/meta`、`/entities`、`/manuscript`、`/summaries` 的层次结构本身就是关于知识类别的设计决策。

3.1.2 迁移至 FlowCabal

FlowCabal 的 OpenViking 集成已经定义了项目结构：

```
/project
  /meta           -- 大纲、风格指南、世界规则
  /entities       -- 角色、地点、情节线
  /manuscript     -- 章节内容 + 摘要
  /summaries      -- 弧线和全作品摘要
```

需要保留的价值是确定性路径约定。Role A 应能直接构造路径如 `/entities/characters/protagonist` 而无需搜索。这使上下文组装部分基于规则（始终包含 `/meta/style-guide.md`），部分基于搜索（通过 L0 扫描查找相关章节）。

变化之处： OpenViking 使用 AGFS（基于 Go 的文件系统服务器）作为存储后端。FlowCabal 直接使用 SQLite。虚拟路径是 SQLite 表中的逻辑键，而非实际的文件系统路径。这简化了部署（无需单独进程），但失去了 AGFS 的原生文件系统语义。这个取舍是合适的——FlowCabal 是单用户本地应用，不是多租户服务器。

3.2 项目知识的时间压缩（OpenClaw）

3.2.1 原始价值

OpenClaw 的时间压缩管线：每日日志 → 每周摘要 → 长期 MEMORY.md。每一层都更加精炼。7 天保留窗口保持近期数据的完整细节，而较旧的数据被压缩为分类摘要（决策 / 发现 / 偏好 / 任务）。

这解决了无界增长问题：没有压缩，知识存储随时间线性增长，最终消耗整个上下文窗口。压缩为长期知识维持了固定的预算。

3.2.2 迁移至 FlowCabal

FlowCabal 的手稿如果从时间维度看，已经有类似的结构：

OpenClaw	**FlowCabal 对应**	**机制**
每日日志	每章完整内容（L2）	完整文本，始终可用
每周摘要	每章摘要（L1）	用于扫描的压缩形式
MEMORY.md	弧线摘要 + 实体档案	最高层级的项目知识

可迁移的原则是**有界上下文预算**。为节点组装上下文时，Role A 应在 token 预算内操作：

```
总预算：~20K token
  /meta（始终包含）：    ~2K token
  实体文件（按需）：      ~3K token
  章节 L0 扫描：          ~3K token（全部章节）
  章节 L1 阅读：          ~4K token（前 5-8 章）
  章节 L2 深读：          ~6K token（前 1-2 章）
  弧线摘要：              ~2K token
```

如果手稿增长到 500 章，L0 扫描增长到 15K token，这太多了。此时系统应切换到**先扫描弧线级别的 L0**，再在选定弧线内进行章节级别扫描。这是时间压缩原则在叙事结构上的应用。

变化之处： OpenClaw 沿时间轴压缩（旧数据被压缩）。FlowCabal 沿叙事距离轴压缩（远离当前章节的内容被更多压缩）。机制不同但原则相同：有界预算下的渐进摘要。

3.3 质量门禁作为结构性强制（Trellis：Ralph 循环）

3.3.1 原始价值

Trellis 的 Ralph 循环拦截 Check 智能体的停止事件，要求在允许完成前提供验证证明。它支持程序化检查（运行 lint 命令）和标记型检查（要求输出中包含特定完成标记）。硬性上限 5 次迭代防止无限循环。

关键洞察：质量强制应该是结构性的（无法绕过）而非建议性的（可能被遗忘的指令）。

3.3.2 迁移至 FlowCabal

Role C（监控者）评估输出质量并决定：批准 / 重试 / 交给人类。Ralph 循环模式直接映射：

```
MAX_RETRIES = 3

for attempt in range(MAX_RETRIES):
    output = user_llm.generate(prompt)
    evaluation = role_c.evaluate(output, node_config, project_state)

    if evaluation.decision == 'approve':
        ws.push('node:completed', output, evaluation)
        break
    elif evaluation.decision == 'retry':
        prompt = adjust_prompt(prompt, evaluation.feedback)
        continue
    elif evaluation.decision == 'flag-human':
        ws.push('node:needs-human', output, evaluation)
        human_response = await ws.receive('human:decision')
        handle_human_decision(human_response)
        break
else:
    # 重试次数耗尽
    ws.push('node:needs-human', output, {'reason': 'max retries exceeded'})
```

变化之处： Trellis 的质量检查关于代码正确性（lint、类型检查）。FlowCabal 的质量检查关于叙事质量（连续性、角色声音、情节连贯性）。检查必然更加主观，依赖 Agent LLM 的判断而非确定性工具。结构性模式（带硬性上限的重试循环 + 人类兜底）直接迁移。

3.4 实体关系追踪系统（OpenViking）

3.4.1 原始价值

OpenViking 支持资源之间的显式关系链接。关系存储为目录内的 `.relations.json` 文件，在检索时作为 `RelatedContext` 浮现。这使智能体能够发现主题或结构相关但可能不共享文本相似性的资源。

3.4.2 迁移至 FlowCabal

实体追踪是长篇小说的关键关注点。角色、地点和情节线形成关系网络。关系系统直接映射：

```
/entities/characters/protagonist  --关联-->  /entities/locations/castle
/entities/characters/protagonist  --冲突-->  /entities/characters/antagonist
/manuscript/chapter-05            --引入-->  /entities/characters/mentor
/entities/plot-threads/revenge    --涉及-->  /entities/characters/antagonist
```

当 Role A 为某章组装上下文时，可以沿关系链接追溯："这一章涉及主角 → 沿链接查找主角的相关地点和冲突角色 → 包含那些实体文件。"

变化之处： OpenViking 的关系通过 `client.link()` 手动创建。在 FlowCabal 中，关系应在策展期间由 Role A 自动维护。当一章被策展时，Role A 解析实体变更并更新关系图。这对应 OpenViking 项目结构中的 `entity-changes.json` 文件。

4 第三层：内化为原则

4.1 幂等操作（OpenClaw）

OpenClaw 的 Session ID 前缀匹配确保没有会话被捕获两次，即使在 cron 重试或时钟偏移下也是如此。原则：**每个自动化管线步骤都应可安全重复运行**。

对 FlowCabal 来说，这适用于策展管线。策展同一输出两次应该是空操作。为已摘要的章节重新生成摘要应产生相同结果或原地更新。`output:persist` 处理程序应在插入前检查现有条目。

4.2 解析与语义分离（OpenViking）

OpenViking 有一条严格规则：解析器永远不调用 LLM。解析（将文档转换为结构化内容）和语义处理（生成摘要、嵌入）完全解耦，异步运行。

对 FlowCabal 来说，这意味着策展管线应分离：
1. 存储（即时）：将策展输出保存到 SQLite。
2. 语义处理（异步）：生成 L0/L1 摘要、更新实体记录、重建嵌入。

用户应获得即时确认（`output:persisted`），无需等待摘要完成。摘要可在后台任务中生成，在下一个执行周期可用。

4.3 溯源追踪（OpenClaw + OpenViking）

两个系统都维护完整的溯源链。OpenClaw：MEMORY.md 条目 → 每周摘要 → 每日日志 → 会话。OpenViking：搜索结果 → 资源 URI → 原始内容。

FlowCabal 的 v3 设计已在 `node:completed` 消息中包含 `contextSources: string[]`。应扩展为不仅追踪使用了哪些资源，还追踪 Role A 为什么选择它们：

```
contextSources: {
  uri: string;           // 如 "/entities/characters/protagonist"
  reason: string;        // 如 "角色出现在场景中"
  level: 'L0' | 'L1' | 'L2';  // 读取了哪个层级
}[]
```

这使用户能够理解和调试上下文组装过程："为什么 AI 认为这一章是关于 X 的？"

4.4 双写可靠性（OpenClaw）

OpenClaw 通过活跃智能体（实时）和 cron 任务（安全网）两种方式捕获重要信息。原则：关键数据应有两条捕获路径。

对 FlowCabal 来说，这意味着运行时缓存（Python 内存）和策展存储（SQLite）应有明确的生命周期语义。如果应用在执行期间崩溃，运行时缓存会丢失——但这是可接受的，因为只有用户批准的输出才应持久化。然而，工作流状态（哪些节点已完成、它们的输出）应在长时间执行期间定期检查点写入 SQLite，提供崩溃恢复能力。

4.5 单一事实来源与派生数据（Trellis）

Trellis 的 `AI_TOOLS` 注册表将平台数据定义一次，其他一切由其派生。结合编译时断言，这防止了不完整的添加。

对 FlowCabal 来说，这适用于 `core/` 类型定义。`NodeDefinition`、`WorkflowDefinition` 和 `TextBlockList` 是工作流结构的单一事实来源。Python 后端应精确镜像这些类型（通过代码生成或共享模式定义），WebSocket 协议应从它们派生。浏览器和后端类型定义之间的任何分歧都会导致隐微的 bug。

5 总结：优先级映射

优先级	设计	来源	FlowCabal 目标
1	三级信息模型（L0/L1/L2）	OpenViking	OpenViking 适配器、策展管线
2	层次化检索 + 意图分析	OpenViking	Role A 上下文组装
3	策展持久化哲学	OpenClaw	output:persist 管线、Role C
4	规范注入（临时性）	Trellis	三层 prompt 组装
5	虚拟文件系统组织	OpenViking	SQLite 支持的 OpenViking 路径
6	有界上下文预算	OpenClaw	Role A token 预算管理
7	带重试循环的质量门禁	Trellis	Role C 重试 + 人类兜底
8	实体关系追踪	OpenViking	自动关系维护
9	幂等策展	OpenClaw	output:persist 去重
10	异步语义处理	OpenViking	后台 L0/L1 生成
11	上下文来源溯源	两者	扩展的 contextSources 元数据
12	跨语言类型单一事实来源	Trellis	TS/Python 类型同步

5.1 共同的根本洞察

三个参考项目汇聚于一个原则：**激进的、有原则的过滤是可持续 AI 知识管理的关键**。OpenClaw 的 4D 验证、OpenViking 的三级渐进加载、Trellis 的规范注入都服务于同一目的——确保 AI 在正确的时间以正确的量接收正确的信息，而不是一次性接收所有信息。

对 FlowCabal 来说，这表现为：只有策展输出进入知识存储，只有相关上下文进入 prompt，每个资源只加载适当的详细程度（L0/L1/L2）。工作流 DAG 提供结构骨架；Agent 系统提供高效导航该结构的智能。