FlowWrite 架构前瞻与建议

日期：2026.01.19

本文档基于对现有代码的分析，独立思考 FlowWrite 未来可能面临的架构挑战，并提出具体建议。

目录

当前架构评估

优势

1. **Core 层设计精良**：TextBlock、VirtualTextBlock、ApiConfiguration 的抽象清晰
2. **不可变更新模式**：所有函数返回新对象，支持 undo/redo
3. **依赖追踪机制**：VirtualTextBlock 的 sourceNodeId 自然形成 DAG
4. **拓扑排序正确**：Kahn 算法实现无误

需要关注的设计张力

张力一：@xyflow 的节点模型

@xyflow/svelte 有自己的节点数据结构：

```
// \@xyflow 的 Node
{
  id: string;
  type: string;
  position: { x: number; y: number };
  data: Record<string, unknown>;  // 任意数据
}
```

而 core/node.ts 定义的 Node：

```
// FlowWrite 的 Node
{
  id: NodeId;
  name: string;
  apiConfig: ApiConfiguration;
  output: TextBlock | null;
  state: NodeState;
  position: { x: number; y: number };
}
```

问题：两者不兼容。消息总线解决通信问题，但不解决数据模型问题。

建议：不要试图统一两者。接受 @xyflow 是 UI 层的事实，让它管理视觉状态（位置、选中、拖拽）。Core Node 作为业务模型，通过 ID 关联。FlowEditor 持有两份数据：flowNodes（@xyflow）和 coreWorkflow（core），由 EventBus 事件驱动同步。

张力二：metadata vs running-state 的当前混杂

core/node.ts 的 Node 混合了两类数据：

```
interface Node {
  // metadata（应持久化）
  id, name, position, apiConfig

  // running-state（应在 core-runner）
  output, state, errorMessage
}
```

问题：序列化时需要手动排除 running-state 字段。

建议：重构为两个独立类型：

```
// core/node.ts - 纯 metadata
interface NodeDefinition {
  readonly id: NodeId;
  name: string;
  position: { x: number; y: number };
  apiConfig: ApiConfiguration;
}

// core-runner/state.ts - 纯 running-state
interface NodeRuntimeState {
  state: 'idle' | 'pending' | 'running' | 'completed' | 'error';
  output: string | null;
  errorMessage?: string;
}

type RuntimeStateMap = Map<NodeId, NodeRuntimeState>;
```

这样 `core/` 可以直接序列化，无需过滤。

未被充分考虑的问题

问题一：流式输出的处理

当前 `executeNode` 等待 LLM 完整响应后才更新状态。但现代 LLM API 都支持流式输出，用户期望看到逐字生成。

当前行为

```
// workflow.ts
const output = await executor(currentNodeId, currentNode);  // 等待完成
nodes.set(currentNodeId, setNodeCompleted(currentNode, output));
```

流式输出的挑战

1. **下游节点何时可以开始？** 流式输出意味着输出是增量的
2. **VirtualTextBlock 如何处理？** 当前 `resolveVirtualTextBlock` 假设内容是完整的
3. **UI 如何显示？** 需要逐字更新节点输出

建议方案

引入三层状态：

```
type NodeRuntimeState =
  | { state: 'idle' }
  | { state: 'pending' }
  | { state: 'running'; partialOutput: string }  // 新增：流式中
  | { state: 'completed'; output: string }
  | { state: 'error'; message: string };
```

EventBus 消息：

```
'node:output': {
  nodeId: NodeId;
  content: string;
  streaming: boolean;  // true = 增量更新
  done: boolean;       // true = 流式结束
}
```

关键决策：下游节点是否应该在上游流式完成前开始？

- **保守策略**：等待上游完成。简单可靠。
- **激进策略**：上游流式中即可开始。复杂但响应快。

建议先实现保守策略，流式输出仅用于 UI 显示，不影响执行逻辑。

问题二：并行执行

当前实现是严格串行：

```
// workflow.ts
while (current.state === 'running') {
  current = await executeNode(current, executor);  // 一个接一个
}
```

但拓扑排序的结果中，同一"层"的节点（入度同时变为 0）可以并行执行。

建议

修改 `executeWorkflow` 支持并行：

```
async function executeWorkflowParallel(
  workflow: Workflow,
  executor: NodeExecutor
): Promise<Workflow> {
  // 按层分组
  const layers = groupByTopologicalLayers(workflow.nodes);

  for (const layer of layers) {
    // 同一层的节点并行执行
    const results = await Promise.all(
      layer.map(nodeId => executor(nodeId, workflow.nodes.get(nodeId)!))
    );
    // 更新状态，传播输出...
  }
}
```

注意：并行执行会增加 API 调用的并发数，需考虑 rate limiting。

问题三：VirtualTextBlock 的 freeze 语义

当前 freeze 的语义：

```
freezeVirtualTextBlock(block)  // 锁定当前内容，不再更新
```

潜在问题：

1. 用户保存工作流时，frozen VirtualTextBlock 包含 `resolvedContent`，这是 running-state
2. 如果用户重新执行，frozen 块不会更新，可能导致困惑

建议

freeze 应该有两种模式：

```
type FreezeMode =
  | 'session'   // 本次会话冻结，关闭后恢复
  | 'permanent' // 永久冻结，转换为 TextBlock
```

保存工作流时：

- `session` 模式：不保存 resolvedContent

· permanent 模式：转换为普通 TextBlock 后保存

问题四：错误恢复与重试

当前错误处理：

```
setNodeError(node, errorMessage)  // 设置错误状态
// 整个 workflow 变为 error 状态
return { ...workflow, state: 'error' };
```

问题：一个节点失败，整个工作流停止。用户无法：

· 重试单个节点
· 跳过失败节点继续执行
· 手动填入输出

建议

1. **节点级重试**：

```
bus.emit('node:retry', { nodeId })
// core-runner 重新执行该节点，不影响其他节点
```

2. **手动填充输出**：

```
bus.emit('node:set-output', { nodeId, content: '手动输入的内容' })
// 允许用户绕过 LLM 调用
```

3. **跳过并继续**：

```
bus.emit('node:skip', { nodeId })
// 标记为 skipped，下游节点使用空输出或默认值
```

问题五：大型工作流的性能

内存

· 每个节点的输出可能很大（LLM 响应数千 token）
· 100 个节点 × 4KB/输出 = 400KB running-state

建议：考虑 LRU 缓存策略，只保留最近 N 个节点的完整输出，其余只保留摘要或丢弃。

渲染

· @xyflow 在节点数量多时可能卡顿
· 节点内容更新触发重渲染

建议：

· 使用 $state.raw 避免深度响应式
· 节点输出内容使用虚拟滚动或截断显示
· 考虑 Web Worker 处理拓扑排序和验证

未来功能预判

功能一：条件执行

“ComfyUI for text” 迟早需要条件分支：

```
[Input] → [LLM: 分类] → [如果是问题] → [LLM: 回答]
                      → [如果是命令] → [LLM: 执行]
```

建议

不要现在实现，但预留扩展点：

```
// 未来可能的节点类型
type NodeType =
  | 'llm'        // 当前唯一类型
  | 'condition'  // 条件分支
  | 'merge'      // 合并分支
  | 'loop'       // 循环
  | 'transform'  // 文本变换
```

当前架构的 `NodeDefinition` 应该是 discriminated union 的基础：

```
interface BaseNodeDefinition {
  id: NodeId;
  name: string;
  position: { x: number; y: number };
}

interface LLMNodeDefinition extends BaseNodeDefinition {
  type: 'llm';
  apiConfig: ApiConfiguration;
}

// 未来
interface ConditionNodeDefinition extends BaseNodeDefinition {
  type: 'condition';
  condition: string;  // 表达式
}

type NodeDefinition = LLMNodeDefinition | ConditionNodeDefinition | ...;
```

功能二：节点模板与预设

用户会想要：

- 保存常用的 API 配置为模板
- 保存整个子工作流为组件
- 社区分享工作流

建议

数据库 schema 预留：

```
// db schema
interface Template {
  id: string;
  name: string;
  type: 'node' | 'subworkflow';
  data: NodeDefinition | WorkflowDefinition;
}
```

功能三：多 Provider 支持

当前 `ApiConnection` 假设 OpenAI 兼容 API：

```
interface ApiConnection {
  endpoint: string;
  apiKey: string;
```

```
  model: string;
}
```

未来可能需要：
- Anthropic（不同的 API 格式）
- Google Gemini
- 本地 Ollama
- Azure OpenAI（不同的认证）

建议

```
type ApiConnection =
  | { type: 'openai-compatible'; endpoint: string; apiKey: string; model: string }
  | { type: 'anthropic'; apiKey: string; model: string }
  | { type: 'azure'; endpoint: string; apiKey: string; deployment: string }
  | { type: 'ollama'; endpoint: string; model: string };
```

`api/client.ts` 相应地需要 adapter 模式。

重构路线图建议

阶段 0：基础设施（当前）

1. 实现 EventBus
2. 不改动 `core/` 的类型定义

阶段 1：core 层净化

1. 将 `Node` 拆分为 `NodeDefinition`（metadata）
2. 将 running-state 移至 `core-runner/`
3. 确保 `core/` 的所有类型可直接 `JSON.stringify`

阶段 2：core-runner 实现

1. 实现 `WorkflowRunner` 类
2. 集成流式输出
3. 实现节点级重试

阶段 3：@xyflow 桥接

1. FlowEditor 持有双数据源（flowNodes + coreWorkflow）
2. EventBus 驱动同步
3. 节点编辑器组件

阶段 4：高级功能

1. 并行执行
2. 条件节点（如果需求明确）
3. 模板系统

风险警示

警示一：过度设计

FlowWrite 仍处于早期阶段。不要为了“可能需要”而提前实现：

· 条件执行
· 循环节点
· 插件系统

先让基础功能稳定运行，再根据实际使用反馈扩展。

警示二：@xyflow 的锁定

@xyflow 是功能丰富的库，但也是技术债务来源：
· 版本升级可能破坏兼容性
· 自定义节点组件与其深度绑定
· 如果未来需要换库，成本很高

建议：将 @xyflow 相关代码隔离在 `src/lib/flow/` 目录，不让 core 层依赖它。

警示三：IndexedDB 的局限

IndexedDB 适合本地持久化，但：
· 无法跨设备同步
· 浏览器可能清理数据
· 大文件性能差

如果未来需要云同步，需要后端服务。现阶段可以添加导出/导入功能作为备份手段。

总结

核心建议

1. **接受 @xyflow 与 core 的双数据模型**，不强求统一
2. **立即分离 metadata 与 running-state**，这是正确的架构方向
3. **流式输出优先用于 UI 显示**，不改变执行语义
4. **预留扩展点但不提前实现**，避免过度设计

优先级排序

1. EventBus 基础设施
2. NodeDefinition / NodeRuntimeState 分离
3. core-runner 实现
4. 流式输出 UI
5. 节点级重试
6. 并行执行（可选）

不建议现在做的事

· 条件/循环节点
· 插件系统
· 多 Provider 支持（除非有明确需求）
· 云同步