本文档基于 design_doc.typ 中的产品设计，梳理当前已实现的组件，分析设计与实现之间的差距，并规划后续的集成工作。

目录

实现状态总览

核心数据层（src/lib/core/）

核心数据层已完整实现设计文档中的所有抽象，代码质量较高。

模块	状态	说明
textblock.ts	✅ 完成	TextBlock、VirtualTextBlock、TextBlockList 完全符合设计
apiconfig.ts	✅ 完成	ApiConfiguration、ApiConnection、ApiParameters 完全符合设计
node.ts	✅ 完成	Node、NodeState、NodeMap 及状态转换函数完全符合设计
workflow.ts	✅ 完成	Workflow、拓扑排序、执行引擎完全符合设计

API 客户端层（src/lib/api/）

API 客户端层提供了 OpenAI 兼容的 LLM 调用能力。

模块	状态	说明
types.ts	✅ 完成	ChatCompletionRequest、ChatCompletionResponse、StreamChunk 等类型
client.ts	✅ 完成	OpenAICompatibleClient 支持流式和非流式调用

数据持久化层（src/lib/db/）

数据持久化层使用 Dexie.js 实现 IndexedDB 存储。

模块	状态	说明
index.ts	✅ 完成	数据库定义、序列化/反序列化、类型验证
workflows.ts	✅ 完成	工作流 CRUD 操作

settings.ts	✅ 完成	设置 CRUD 操作

UI 组件层（src/lib/）

UI 组件层使用 SvelteFlow 构建节点编辑器，但**与核心数据层存在断层**。

节点组件（nodes/）

组件	状态	说明
LLMNode.svelte	⚠ 独立	LLM 节点 UI，使用 SvelteFlow 的 data 结构
TextNode.svelte	⚠ 独立	文本节点 UI
InputNode.svelte	⚠ 独立	输入节点 UI
OutputNode.svelte	⚠ 独立	输出节点 UI

边组件（edges/）

组件	状态	说明
LabeledEdge.svelte	✅ 完成	带标签的边
CustomEdge.svelte	✅ 完成	自定义样式边

通用组件（components/）

组件	状态	说明
FlowCanvas.svelte	✅ 完成	SvelteFlow 画布封装
ContextMenu.svelte	✅ 完成	右键菜单
Toolbar.svelte	✅ 完成	工具栏
NodeSidebar.svelte	✅ 完成	节点面板侧边栏

工具函数（utils/）

模块	状态	说明
validation.ts	✅ 完成	工作流验证
computing.ts	✅ 完成	UI 层拓扑排序
layout.ts	✅ 完成	Dagre 图布局

核心问题：数据模型断层

问题描述

当前实现存在一个关键的架构问题：**UI 层的节点模型与核心数据层的节点模型是分离的**。

核心层的 Node（设计文档定义）

```
interface Node {
  readonly id: NodeId;
  name: string;
  apiConfig: ApiConfiguration;  // 包含 TextBlockList 的完整配置
  output: TextBlock | null;
  state: NodeState;
  position: { x: number; y: number };
}
```

核心层的 Node 包含：
- 完整的 API 配置（连接、参数、系统提示词、用户提示词）
- 提示词是 TextBlockList，支持虚拟文本块引用
- 执行状态和输出

UI 层的 Node（SvelteFlow）

```
// FlowEditor.svelte 中的节点定义
{
  id: 'llm-1',
  type: 'llm',
  position: { x: 400, y: 200 },
  data: {
    label: 'Generate Content',
    provider: 'OpenAI',
    model: 'gpt-4',
    status: 'idle'
  }
}
```

UI 层的节点是 SvelteFlow 的通用结构：
- data 是一个扁平的键值对象
- 没有 TextBlockList、ApiConfiguration 等核心抽象
- 节点类型是字符串（'llm', 'text', 'input', 'output'）

断层的影响

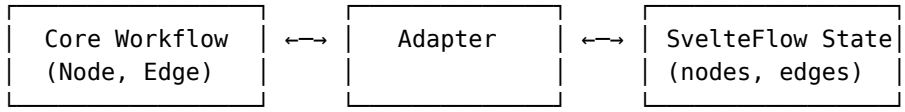
1. **无法使用虚拟文本块**：UI 无法表达“引用另一个节点输出”的概念
2. **API 配置未集成**：节点的 LLM 调用参数无处存放
3. **执行引擎未连接**：workflow.ts 中的执行逻辑无法作用于 UI 节点
4. **持久化不完整**：保存/加载时无法保留完整的工作流状态

集成方案

方案一：适配器模式（推荐）

保持 SvelteFlow 的 UI 结构，通过适配器在两种数据模型间转换。

数据流



实现要点

```
// 核心 Node → SvelteFlow Node
function coreNodeToFlowNode(node: CoreNode): FlowNode {
  return {
    id: node.id,
    type: 'llm',  // 根据配置判断
    position: node.position,
    data: {
      label: node.name,
      status: node.state,
      model: node.apiConfig.connection.model,
      // 保留核心 Node 引用或 ID
      coreNodeId: node.id
    }
  };
}

// SvelteFlow Node → 核心 Node 更新
function syncFlowNodeToCore(flowNode: FlowNode, coreNode: CoreNode): CoreNode {
  return {
    ...coreNode,
    name: flowNode.data.label,
    position: flowNode.position
  };
}
```

优点

- 不破坏现有 UI 代码
- 核心逻辑与 UI 逻辑解耦
- 渐进式迁移

缺点

- 需要维护两套数据结构的同步
- 适配器代码可能变复杂

方案二：统一数据源

将核心 Node 直接作为 SvelteFlow 节点的 data。

实现要点

```
// 直接使用核心 Node 作为 data
const flowNodes: FlowNode[] = Array.from(workflow.nodes.values()).map(node => ({
  id: node.id,
  type: determineNodeType(node),
  position: node.position,
  data: node  // 整个核心 Node 作为 data
}));
```

优点
- 单一数据源，无需同步
- 核心功能直接可用

缺点
- 需要重写所有节点组件
- SvelteFlow 的某些功能可能不兼容

方案三：混合模式
UI 层维护视觉状态，核心层维护业务状态，通过事件同步。

实现要点

```
// UI 事件触发核心更新
function onNodeDrag(flowNode: FlowNode) {
  workflow = updateNode(workflow, flowNode.id, n => ({
    ...n,
    position: flowNode.position
  }));
}

// 核心状态变化触发 UI 更新
$effect(() => {
  for (const [id, node] of workflow.nodes) {
    const flowNode = nodes.find(n => n.id === id);
    if (flowNode) {
      flowNode.data.status = node.state;
    }
  }
});
```

实施计划

阶段一：建立桥接
目标：在不改动现有代码的前提下，建立核心层与 UI 层的连接。

任务清单

1. 创建 `src/lib/bridge/` 模块
2. 实现 `WorkflowAdapter` 类
   - `fromWorkflow(workflow: Workflow): { nodes: FlowNode[], edges: FlowEdge[] }`
   - `toWorkflow(nodes: FlowNode[], edges: FlowEdge[]): Workflow`
3. 在 `FlowEditor.svelte` 中引入适配器
4. 验证双向同步

阶段二：集成节点编辑
目标：让用户能够编辑节点的完整配置（API 连接、参数、提示词）。

任务清单

1. 创建 `NodeEditor.svelte` 组件

   - 编辑 ApiConnection（endpoint、apiKey、model）
   - 编辑 ApiParameters（temperature、maxTokens 等）
   - 编辑 systemPrompt 和 userPrompt（TextBlockList）
2. 创建 TextBlockListEditor.svelte 组件
   - 添加/删除文本块
   - 创建虚拟文本块（选择源节点）
   - 冻结/解冻虚拟块
3. 在 NodeSidebar 或弹窗中集成编辑器

阶段三：集成执行引擎

目标：实现工作流的实际执行。

任务清单

1. 在 FlowEditor 中引入 executeWorkflow
2. 实现 NodeExecutor
   - 使用 OpenAICompatibleClient
   - 处理流式响应
   - 更新节点状态
3. 在 UI 上显示执行进度
   - 节点状态变化动画
   - 流式输出显示
4. 错误处理与重试

阶段四：持久化集成

目标：保存和加载完整的工作流状态。

任务清单

1. 在 FlowEditor 中集成 saveWorkflow / loadWorkflow
2. 实现工作流列表页面
3. 自动保存功能
4. 导入/导出功能

附录：现有组件详解

TextBlock 系统（textblock.ts）

已实现的类型

- TextBlock：静态文本块
- VirtualTextBlock：虚拟文本块，引用节点输出
- TextBlockList：文本块列表容器

已实现的操作

```
// 创建
createTextBlock(content: string): TextBlock
createVirtualTextBlock(sourceNodeId: NodeId, displayName?: string): VirtualTextBlock
createTextBlockList(initialBlocks?: AnyTextBlock[]): TextBlockList
```

```
// 虚拟块操作
resolveVirtualTextBlock(block, content): VirtualTextBlock  // 解析
freezeVirtualTextBlock(block): VirtualTextBlock            // 冻结
unfreezeVirtualTextBlock(block): VirtualTextBlock          // 解冻
resetVirtualTextBlock(block): VirtualTextBlock             // 重置

// 列表操作
appendBlock(list, block): TextBlockList
insertBlock(list, index, block): TextBlockList
removeBlock(list, blockId): TextBlockList
updateBlock(list, blockId, updater): TextBlockList

// 查询
getBlockContent(block): string      // 获取显示内容
isBlockReady(block): boolean        // 检查是否就绪
getListContent(list): string        // 获取拼接内容
isListReady(list): boolean          // 检查列表是否就绪
getDependencies(list): NodeId[]     // 获取依赖节点
```

ApiConfiguration 系统 (apiconfig.ts)

已实现的类型

- ApiConnection: 连接设置 (endpoint、apiKey、model)
- ApiParameters: 请求参数 (temperature、maxTokens 等)
- ApiConfiguration: 完整配置 (connection + parameters + prompts)

已实现的操作

```
createApiConfiguration(): ApiConfiguration
getApiConfigDependencies(config): NodeId[]
isApiConfigReady(config): boolean
getSystemPromptContent(config): string
getUserPromptContent(config): string
resolveApiConfigOutput(config, nodeId, content): ApiConfiguration
updateApiConnection(config, connection): ApiConfiguration
updateApiParameters(config, parameters): ApiConfiguration
updateSystemPrompt(config, systemPrompt): ApiConfiguration
updateUserPrompt(config, userPrompt): ApiConfiguration
```

Node 系统 (node.ts)

已实现的类型

- NodeState: ‘idle’ | ‘pending’ | ‘running’ | ‘completed’ | ‘error’
- Node: 完整节点定义
- NodeMap: Map<NodeId, Node>

已实现的操作

```
createNode(name, position?, apiConfig?): Node
getNodeDependencies(node): NodeId[]
isNodeReady(node): boolean
getNodePrompt(node): { system: string; user: string }
```

```
getNodeOutput(node): string

// 状态转换
setNodePending(node): Node
setNodeRunning(node): Node
setNodeCompleted(node, outputContent): Node
setNodeError(node, errorMessage): Node
resetNode(node): Node

// 集合操作
propagateNodeOutput(nodes, completedNodeId, outputContent): NodeMap
```

Workflow 系统 (workflow.ts)

已实现的类型

- WorkflowState: 'idle' | 'running' | 'completed' | 'error'
- Workflow: 完整工作流定义
- DependencyError: 依赖错误
- TopologicalSortResult: 拓扑排序结果

已实现的操作

```
createWorkflow(name, nodes?): Workflow
topologicalSort(nodes): TopologicalSortResult
prepareWorkflow(workflow): Workflow | DependencyError
executeNode(workflow, executor): Promise<Workflow>
executeWorkflow(workflow, executor, onProgress?): Promise<Workflow>

// 修改
addNode(workflow, node): Workflow
removeNode(workflow, nodeId): Workflow
updateNode(workflow, nodeId, updater): Workflow
resetWorkflow(workflow): Workflow

// 查询
getNode(workflow, nodeId): Node | undefined
getNodes(workflow): Node[]
```

API Client (client.ts)

已实现的类

```
class OpenAICompatibleClient {
  constructor(config: ClientConfig)
  chatCompletion(request): Promise<ChatCompletionResponse>
  chatCompletionStream(request): AsyncGenerator<StreamChunk>
}
```

已实现的辅助函数

```
buildRequestFromConfig(config: ApiConfiguration): ChatCompletionRequest
buildChatRequest(messages, model, options?): ChatCompletionRequest
extractContent(response): string
extractUsage(response): UsageInfo
```

```
extractStreamContent(chunk): string
handleStream(stream, options): Promise<string>
```