

FlowWrite 是一款专为 AI 辅助文字创作而生的可视化工作流编辑器。本文档聚焦于产品核心功能设计，探索文本处理与创意写作的全新交互范式。

目录

产品愿景	1
缘起	1
理念	1
核心功能	2
workflow 功能模块的总体设计	2
交互方式	2
组件设计	3
文本块系统	3
基础文本块 (TextBlock)	3
虚拟文本块 (VirtualTextBlock)	3
文本块列表 (TextBlockList)	3
依赖解析	4
使用示例	4
节点系统 (Node)	5
节点结构	5
节点状态流转	5
LLM 配置	5
核心操作	5
工作流系统 (Workflow)	6
工作流结构	6
拓扑排序 (Kahn's Algorithm)	6
执行流程	6
状态传播	6
完整示例	7

产品愿景

FlowWrite 致力于成为 AI 时代文字创作者的得力助手，通过直观的可视化工作流，让复杂的 AI 赋能文本处理变得简单而优雅。

缘起

在 AI 写作实践中，创作者往往需要在多个工具间频繁切换，prompt 的调试、文本的迭代、创意的碰撞都充满了重复性的复制粘贴操作。FlowWrite 诞生于这样的痛点：我们渴望一个场景覆盖足够大的创作范式，让文字工作者能够专注于内容本身。

理念

AI 可以大幅提高写作的效率，但真正的创作往往需要反复打磨和人工干预。我们借鉴 ComfyUI 的节点化思想，为文本创作领域打造一个灵活、可定制的可视化工作流系统。我们希望以这样的方式，让文字创作者享受“人剑合一”的创作体验。

核心功能

- **节点化文本处理**：将复杂的 AI 流程操作拆解为（低耦合的）基于节点的工作流
- **实时依赖解析**：智能处理节点间的数据流转与依赖关系
- **创作友好设计**：为文字工作者的使用习惯深度优化

workflow 功能模块的总体设计

基于对交互方式的设想，我们发现具体组件的设计可以是低耦合而优雅的。

交互方式

一个节点代表一次 LLM 的 api 调用，也就是一次对话。一个节点的输入可以是多个，输出是唯一的，但可以作为多个节点的输入。那么一个节点的多个输入该如何被应用到这个节点的对话中？其实基于定制化的思路，基本上是这样形态：

[第一个输入]

请参照上面的要求，将以下三次 LLM 对话的输出总结成一个：

[第二个输入]

[第三个输入]

[第四个输入]

也就是说，某个节点的输出在还没有具体从一次 LLM 对话中产生的时候，就已经预订好了它在另一个节点的输入中所扮演的角色，这个设计催生了一些前期设想，如占位文本块，或是文本块引用的概念，前者使得一个节点可以不依赖于与其它节点的连接关系来定义，后者使得非节点输出（也就是不是从 AI 侧获取而是由用户自定义的文本）的文本块也可以共享状态（改一处其它地方也会改）

基于这类设计，工作流的运行主要是类似拓扑解析依赖的过程，当然后续可以加一些对有向环支持的花活。

注意到，前面提到的这两种设想似乎可以合并成一个虚拟文本块的概念。而且要加一个可以在工作流中暂时 freeze 一个虚拟文本块的功能，使得这个虚拟文本块在后续的使用中表现的就像一个普通文本块（这个可能会成为一个很常用的功能）。

不过普通文本块的引用功能还是禁掉吧，感觉没啥意义。

综合来看，状态设计不会很复杂。

组件设计

文本块系统

文本块是 FlowWrite 中最基础的数据单元，用于构建节点的输入内容。系统设计了三种核心抽象：

基础文本块 (TextBlock)

最简单的文本单元，包含静态的文本内容，不依赖任何外部状态。

```
interface TextBlock {  
  readonly type: 'text';  
  readonly id: TextBlockId;  
  content: string;  
}
```

特点：

- 内容由用户直接编辑
- 始终处于就绪状态
- 不参与依赖解析

虚拟文本块 (VirtualTextBlock)

动态文本单元，引用某个节点的输出。这是实现节点间数据流转的关键抽象。

```
interface VirtualTextBlock {  
  readonly type: 'virtual';  
  readonly id: TextBlockId;  
  readonly sourceNodeId: NodeId; // 引用的源节点  
  state: 'pending' | 'resolved' | 'error';  
  resolvedContent: string | null; // 解析后的内容  
  frozen: boolean; // 冻结状态  
  displayName?: string; // 占位显示名称  
}
```

状态流转：

- pending：源节点尚未执行，显示为占位符 [节点名称]
- resolved：源节点已产出内容，显示实际文本
- error：源节点执行失败，显示错误占位符

冻结功能：

- 当 frozen = true 时，虚拟文本块表现为普通文本块
- 冻结后不再响应源节点的更新
- 适用于需要“快照”某个中间结果的场景

文本块列表 (TextBlockList)

文本块的有序容器，代表一个节点的完整输入内容。

```
interface TextBlockList {  
  readonly id: string;  
  blocks: AnyTextBlock[]; // TextBlock | VirtualTextBlock  
}
```

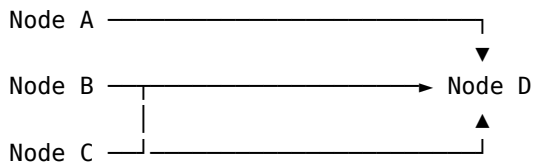
核心操作：

- getListContent()：拼接所有块的内容，生成最终 prompt

- `isListReady()`: 检查所有块是否就绪（可用于判断节点能否执行）
- `getDependencies()`: 获取所有未冻结的虚拟块依赖的节点 ID 列表
- `resolveNodeOutput()`: 当某节点产出内容时，更新所有引用该节点的虚拟块

依赖解析

基于文本块列表的 `getDependencies()` 方法，可以构建节点间的依赖图：



workflow 执行时，按拓扑顺序依次执行节点：

1. 收集所有节点的依赖关系
2. 找出无依赖的节点，优先执行
3. 节点执行完成后，通过 `resolveNodeOutput()` 更新下游节点的虚拟块
4. 重复步骤 2-3 直到所有节点执行完毕

使用示例

```
// 创建一个节点的输入内容
const inputList = createTextBlockList([
  createTextBlock('请参照以下要求：'),
  createVirtualTextBlock('node-requirements', '要求'),
  createTextBlock('\n\n将以下内容进行总结：\n'),
  createVirtualTextBlock('node-source', '源文本'),
]);

// 检查依赖
const deps = getDependencies(inputList);
// => ['node-requirements', 'node-source']

// 当 node-requirements 执行完成
const updated = resolveNodeOutput(inputList, 'node-requirements', '要求内容...');

// 检查是否就绪
isListReady(updated); // => false (node-source 仍为 pending)

// 当所有依赖都解析完成后
const final = resolveNodeOutput(updated, 'node-source', '源文本内容...');
isListReady(final); // => true

// 获取最终 prompt
getListContent(final);
// => '请参照以下要求：要求内容...\n\n将以下内容进行总结：\n源文本内容...'
```

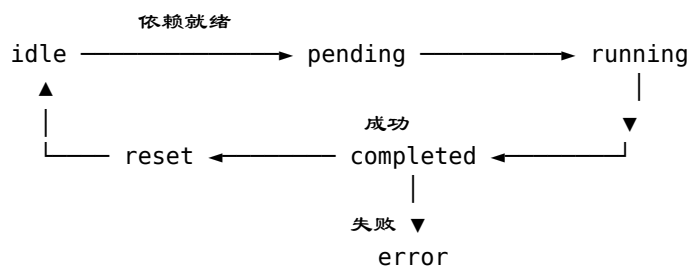
节点系统 (Node)

节点是工作流的核心执行单元，代表一次 LLM API 调用。

节点结构

```
interface Node {  
  readonly id: NodeId;  
  name: string; // 显示名称  
  input: TextBlockList; // 输入 prompt (文本块列表)  
  output: TextBlock | null; // LLM 输出 (执行完成后填充)  
  state: NodeState; // 执行状态  
  errorMessage?: string; // 错误信息  
  llmConfig: LLMConfig; // LLM 配置  
  position: { x: number; y: number }; // 画布位置  
}  
  
type NodeState = 'idle' | 'pending' | 'running' | 'completed' | 'error';
```

节点状态流转



- idle: 初始状态，等待依赖或用户触发
- pending: 所有依赖已就绪，排队等待执行
- running: 正在执行 LLM API 调用
- completed: 执行成功，output 已填充
- error: 执行失败，errorMessage 记录错误

LLM 配置

```
interface LLMConfig {  
  provider: string; // API 提供商 ('openai', 'anthropic', 'custom')  
  model: string; // 模型标识符  
  endpoint?: string; // 自定义 API 端点  
  temperature?: number; // 采样温度 (0-2)  
  maxTokens?: number; // 最大生成 token 数  
  systemPrompt?: string; // 系统提示词  
}
```

核心操作

- getNodeDependencies(node): 获取节点依赖的上游节点 ID 列表
- isNodeReady(node): 检查节点是否可执行（所有依赖已解析）
- getNodePrompt(node): 获取最终的 prompt 字符串
- getNodeOutput(node): 获取输出内容（未完成时返回空字符串）

工作流系统 (Workflow)

工作流管理节点集合，负责依赖解析和执行调度。

工作流结构

```
interface Workflow {
  readonly id: string;
  name: string;
  nodes: NodeMap;           // 节点集合 (Map<NodeId, Node>)
  state: WorkflowState;     // 执行状态
  executionOrder: NodeId[]; // 拓扑排序后的执行顺序
  currentIndex: number;     // 当前执行位置
}

type WorkflowState = 'idle' | 'running' | 'completed' | 'error';
```

拓扑排序 (Kahn's Algorithm)

工作流执行前，需要对节点进行拓扑排序以确定执行顺序：

```
function topologicalSort(nodes: NodeMap): TopologicalSortResult {
  // 1. 计算每个节点的入度 (依赖数量)
  // 2. 将入度为 0 的节点加入队列
  // 3. 依次处理队列中的节点:
  //    - 将其加入结果列表
  //    - 将其下游节点的入度减 1
  //    - 若下游节点入度变为 0, 加入队列
  // 4. 检测循环依赖 (未处理完所有节点)
}
```

错误处理：

- missing: 引用了不存在的节点
- cycle: 检测到循环依赖

执行流程

```
// 1. 准备工作流 (拓扑排序 + 状态初始化)
const prepared = prepareWorkflow(workflow);

// 2. 定义节点执行器 (实际的 LLM API 调用)
const executor: NodeExecutor = async (nodeId, prompt, node) => {
  const response = await callLLMAPI(node.llmConfig, prompt);
  return response.content;
};

// 3. 执行工作流
const result = await executeWorkflow(prepared, executor, (progress) => {
  console.log(`执行进度: ${progress.currentIndex}/${progress.executionOrder.length}`);
});
```

状态传播

当一个节点执行完成时，其输出会自动传播到所有依赖它的下游节点：

```
function propagateNodeOutput(nodes, completedNodeId, outputContent) {
  // 遍历所有节点
```

```
// 若节点的 input 中有虚拟块引用 completedNodeId  
// 则调用 resolveNodeOutput 更新该虚拟块  
}
```

完整示例

```
import {  
  createWorkflow,  
  createNode,  
  addNode,  
  updateNodeInput,  
  createTextBlockList,  
  createTextBlock,  
  createVirtualTextBlock,  
  executeWorkflow  
} from './lib/core';  
  
// 创建工作流  
let workflow = createWorkflow('文章润色工作流');  
  
// 创建节点 A: 生成大纲  
const nodeA = createNode('生成大纲', { x: 100, y: 100 });  
nodeA.input = createTextBlockList([  
  createTextBlock('请为以下主题生成一篇文章大纲: \n\n人工智能的未来发展')  
]);  
  
// 创建节点 B: 扩写第一部分  
const nodeB = createNode('扩写第一部分', { x: 100, y: 250 });  
nodeB.input = createTextBlockList([  
  createTextBlock('基于以下大纲, 扩写第一部分: \n\n'),  
  createVirtualTextBlock(nodeA.id, '大纲')  
]);  
  
// 创建节点 C: 扩写第二部分  
const nodeC = createNode('扩写第二部分', { x: 300, y: 250 });  
nodeC.input = createTextBlockList([  
  createTextBlock('基于以下大纲, 扩写第二部分: \n\n'),  
  createVirtualTextBlock(nodeA.id, '大纲')  
]);  
  
// 创建节点 D: 合并润色  
const nodeD = createNode('合并润色', { x: 200, y: 400 });  
nodeD.input = createTextBlockList([  
  createTextBlock('请将以下两部分内容合并并润色: \n\n第一部分: \n'),  
  createVirtualTextBlock(nodeB.id, '第一部分'),  
  createTextBlock('\n\n第二部分: \n'),  
  createVirtualTextBlock(nodeC.id, '第二部分')  
]);  
  
// 添加节点到工作流  
workflow = addNode(workflow, nodeA);  
workflow = addNode(workflow, nodeB);  
workflow = addNode(workflow, nodeC);
```

```
workflow = addNode(workflow, nodeD);
```

```
// 执行工作流
```

```
// 执行顺序将是: A → B, C (并行) → D
```

```
const result = await executeWorkflow(workflow, executor);
```

依赖图:

