FlowWrite 产品设计文档 v2

日期：2026.01.19

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。本文档是基于 v1 设计文档的重构版本，核心变化是 **分离 metadata 与 running-state**，并引入 **EventBus 通信机制**。

目录

产品愿景

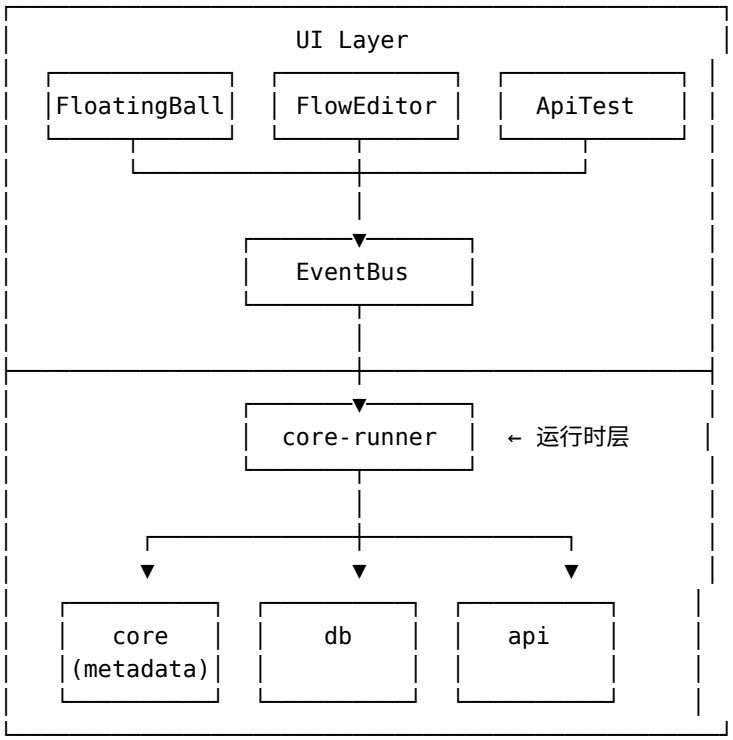
FlowWrite 致力于成为 AI 时代文字创作者的得力助手，通过直观的可视化工作流，让复杂的 AI 赋能文本处理变得简单而优雅。

核心功能

- **节点化文本处理**：将复杂的 AI 流程操作拆解为基于节点的工作流
- **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

架构总览

分层架构



核心设计原则

原则一：metadata 与 running-state 分离

层	数据类型	特性
core/	metadata	静态定义、可序列化、持久化
core-runner/	running-state	运行时状态、易失、不持久化

metadata 示例：节点 ID、名称、位置、API 配置、文本块定义

running-state 示例：执行状态（running/pending/error）、已解析的输出内容、冻结状态

原则二：EventBus 作为唯一跨层通信机制

- UI 层不直接调用 core-runner 的方法
- 所有跨层通信通过 EventBus 消息
- 消息流最多三层：UI → core-runner → UI，不会形成链式循环

原则三：UI 层锁定机制

- 工作流执行期间，UI 层进入锁定状态
- 锁定期间禁止节点编辑、删除、连接等操作
- 防止 running-state 与 UI 操作产生冲突

Core 层设计（metadata）

Core 层只包含静态定义，所有类型都可直接 JSON.stringify。

文本块系统

TextBlock（基础文本块）

```
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

VirtualTextBlockDef（虚拟文本块定义）

```
interface VirtualTextBlockDef {
  readonly type: 'virtual';
  readonly id: TextBlockId;
  readonly sourceNodeId: NodeId;  // 引用的源节点
  displayName?: string;           // 占位显示名称
}
```

关键变化：v1 中的 state、resolvedContent、frozen 字段移至 core-runner 层。

VirtualTextBlockDef 只描述“这个文本块引用哪个节点”，不包含运行时状态。

TextBlockList

```
type AnyTextBlockDef = TextBlock | VirtualTextBlockDef;

interface TextBlockList {
  readonly id: string;
  blocks: AnyTextBlockDef[];
}
```

核心操作：

- `getDependencies(list)`：获取所有虚拟块依赖的节点 ID 列表
- `appendBlock(list, block)`：追加文本块
- `removeBlock(list, blockId)`：移除文本块

API 配置系统

ApiConnection

```
interface ApiConnection {
  endpoint: string;  // OpenAI 兼容的 API 端点
  apiKey: string;    // API 密钥
  model: string;     // 模型标识符
}
```

ApiParameters

```
interface ApiParameters {
  temperature: number;
  maxTokens: number;
  topP: number;
  presencePenalty: number;
  frequencyPenalty: number;
  stopSequences: string[];
  streaming: boolean;
}
```

ApiConfiguration

```
interface ApiConfiguration {
  connection: ApiConnection;
  parameters: ApiParameters;
  systemPrompt: TextBlockList;  // 可包含虚拟文本块
  userPrompt: TextBlockList;    // 可包含虚拟文本块
}
```

节点系统

NodeDefinition（节点定义）

```
interface NodeDefinition {
  readonly id: NodeId;
  name: string;
  position: { x: number; y: number };
  apiConfig: ApiConfiguration;
}
```

关键变化：v1 中的 `state`、`output`、`errorMessage` 字段移至 `core-runner` 层。

`NodeDefinition` 只描述“这个节点是什么”，不包含执行状态。

核心操作

```
// 创建节点
function createNodeDef(name: string, position?: Position): NodeDefinition;

// 获取依赖
function getNodeDependencies(node: NodeDefinition): NodeId[];

// 更新 API 配置
function updateNodeApiConfig(
  node: NodeDefinition,
  updater: (config: ApiConfiguration) => ApiConfiguration
): NodeDefinition;
```

工作流系统

WorkflowDefinition（工作流定义）

```
interface WorkflowDefinition {
  readonly id: string;
  name: string;
  nodes: Map<NodeId, NodeDefinition>;
}
```

关键变化：移除了 `state`、`executionOrder`、`currentIndex` 字段，这些属于运行时状态。

核心操作

```
// 创建工作流
function createWorkflowDef(name: string): WorkflowDefinition;

// 节点操作
function addNode(workflow: WorkflowDefinition, node: NodeDefinition):
WorkflowDefinition;
function removeNode(workflow: WorkflowDefinition, nodeId: NodeId):
WorkflowDefinition;
function updateNode(
  workflow: WorkflowDefinition,
  nodeId: NodeId,
  updater: (node: NodeDefinition) => NodeDefinition
): WorkflowDefinition;

// 拓扑排序（纯函数，不修改状态）
function topologicalSort(workflow: WorkflowDefinition): TopologicalSortResult;
```

Core-Runner 层设计（running-state）

Core-runner 层管理工作流的执行状态，所有数据都是易失的。

运行时状态类型

VirtualBlockState（虚拟块运行时状态）

```
interface VirtualBlockState {
  state: 'pending' | 'resolved' | 'error';
  resolvedContent: string | null;
  frozen: boolean;
}
```

```
// 按 blockId 索引
type VirtualBlockStateMap = Map<TextBlockId, VirtualBlockState>;
```

NodeRuntimeState（节点运行时状态）

```
interface NodeRuntimeState {
  state: 'idle' | 'pending' | 'running' | 'completed' | 'error';
  output: string | null;
  errorMessage?: string;
}

// 按 nodeId 索引
type NodeStateMap = Map<NodeId, NodeRuntimeState>;
```

WorkflowRuntimeState（工作流运行时状态）

```
interface WorkflowRuntimeState {
  state: 'idle' | 'running' | 'completed' | 'error';
  executionOrder: NodeId[];
  currentIndex: number;
  nodeStates: NodeStateMap;
  virtualBlockStates: VirtualBlockStateMap;
}
```

WorkflowRunner 类

```
class WorkflowRunner {
  private workflow: WorkflowDefinition;
  private runtimeState: WorkflowRuntimeState;
  private apiClient: OpenAICompatibleClient;

  constructor(workflow: WorkflowDefinition) {
    this.workflow = workflow;
    this.runtimeState = this.initRuntimeState();
  }

  // 初始化运行时状态
  private initRuntimeState(): WorkflowRuntimeState {
    const nodeStates = new Map<NodeId, NodeRuntimeState>();
    for (const [id] of this.workflow.nodes) {
      nodeStates.set(id, { state: 'idle', output: null });
    }
    return {
      state: 'idle',
      executionOrder: [],
      currentIndex: 0,
      nodeStates,
      virtualBlockStates: new Map()
    };
  }

  // 执行工作流
  async run(): Promise<void> {
    bus.emit('ui:lock', { reason: 'executing' });

    const sortResult = topologicalSort(this.workflow);
    if (sortResult.error) {
      bus.emit('workflow:error', { error: sortResult.error.message });
      bus.emit('ui:unlock');
```

```
      return;
    }

    this.runtimeState.executionOrder = sortResult.order;
    this.runtimeState.state = 'running';

    for (const nodeId of this.runtimeState.executionOrder) {
      await this.executeNode(nodeId);
      if (this.runtimeState.state === 'error') break;
    }

    this.runtimeState.state = this.runtimeState.state === 'error' ? 'error' :
'completed';
    bus.emit('workflow:done', { workflowId: this.workflow.id });
    bus.emit('ui:unlock');
  }

  // 执行单个节点
  private async executeNode(nodeId: NodeId): Promise<void> {
    const nodeDef = this.workflow.nodes.get(nodeId)!;
    const nodeState = this.runtimeState.nodeStates.get(nodeId)!;

    nodeState.state = 'running';
    bus.emit('node:state', { nodeId, state: 'running' });

    try {
      const prompt = this.buildPrompt(nodeDef);
      const response = await this.apiClient.chatCompletion(
        buildRequestFromConfig(nodeDef.apiConfig, prompt)
      );
      const output = extractContent(response);

      nodeState.state = 'completed';
      nodeState.output = output;

      this.propagateOutput(nodeId, output);
      bus.emit('node:output', { nodeId, content: output, streaming: false });
      bus.emit('node:state', { nodeId, state: 'completed' });

    } catch (error) {
      nodeState.state = 'error';
      nodeState.errorMessage = error.message;
      this.runtimeState.state = 'error';
      bus.emit('node:state', { nodeId, state: 'error' });
      bus.emit('workflow:error', { error: error.message, nodeId });
    }
  }

  // 构建 prompt（解析虚拟块）
  private buildPrompt(node: NodeDefinition): { system: string; user: string } {
    return {
      system: this.resolveTextBlockList(node.apiConfig.systemPrompt),
      user: this.resolveTextBlockList(node.apiConfig.userPrompt)
    };
  }
```

```
  // 解析 TextBlockList
  private resolveTextBlockList(list: TextBlockList): string {
    return list.blocks.map(block => {
      if (block.type === 'text') {
        return block.content;
      }
      // virtual block
      const state = this.runtimeState.virtualBlockStates.get(block.id);
      if (state?.frozen && state.resolvedContent) {
        return state.resolvedContent;
      }
      if (state?.state === 'resolved' && state.resolvedContent) {
        return state.resolvedContent;
      }
      return `[${block.displayName ?? block.sourceNodeId}]`;
    }).join('');
  }

  // 传播输出到下游虚拟块
  private propagateOutput(nodeId: NodeId, content: string): void {
    for (const [, nodeDef] of this.workflow.nodes) {
      this.propagateToList(nodeDef.apiConfig.systemPrompt, nodeId, content);
      this.propagateToList(nodeDef.apiConfig.userPrompt, nodeId, content);
    }
  }

  private propagateToList(list: TextBlockList, nodeId: NodeId, content: string): void
{
    for (const block of list.blocks) {
      if (block.type === 'virtual' && block.sourceNodeId === nodeId) {
        const existing = this.runtimeState.virtualBlockStates.get(block.id);
        if (existing?.frozen) continue;  // 冻结的块不更新

        this.runtimeState.virtualBlockStates.set(block.id, {
          state: 'resolved',
          resolvedContent: content,
          frozen: false
        });
      }
    }
  }

  // 冻结虚拟块
  freezeBlock(blockId: TextBlockId): void {
    const state = this.runtimeState.virtualBlockStates.get(blockId);
    if (state && state.state === 'resolved') {
      state.frozen = true;
    }
  }

  // 解冻虚拟块
  unfreezeBlock(blockId: TextBlockId): void {
    const state = this.runtimeState.virtualBlockStates.get(blockId);
    if (state) {
      state.frozen = false;
    }
```

```
  }
}
```

EventBus 设计

消息类型

```typescript
interface BusEvents {
  // UI → core-runner
  'workflow:run': { workflowId: string };
  'workflow:stop': void;
  'node:freeze': { blockId: TextBlockId };
  'node:unfreeze': { blockId: TextBlockId };

  // core-runner → UI
  'node:state': {
    nodeId: NodeId;
    state: 'idle' | 'pending' | 'running' | 'completed' | 'error';
  };
  'node:output': {
    nodeId: NodeId;
    content: string;
    streaming: boolean;
  };
  'workflow:done': { workflowId: string };
  'workflow:error': { error: string; nodeId?: NodeId };

  // UI 锁定
  'ui:lock': { reason: string };
  'ui:unlock': void;
}
```

消息流示例

```
用户点击"执行"按钮
    │
    ▼
FlowEditor: bus.emit('workflow:run', { workflowId })
    │
    ▼
WorkflowRunner 接收消息，开始执行
    │
    ├─► bus.emit('ui:lock', { reason: 'executing' })
    │       │
    │       ▼
    │   FlowEditor: 禁用编辑功能
    │
    ├─► bus.emit('node:state', { nodeId: 'A', state: 'running' })
    │       │
    │       ▼
    │   FlowEditor: 更新节点 A 的 UI 状态
    │
    ├─► (LLM API 调用)
    │
    ├─► bus.emit('node:output', { nodeId: 'A', content: '...', streaming: false })
    │       │
```

```
    │       ▼
    │   FlowEditor: 显示节点 A 的输出
    │
    ├─► bus.emit('node:state', { nodeId: 'A', state: 'completed' })
    │
    ├─► (继续执行节点 B, C, D...)
    │
    ├─► bus.emit('workflow:done', { workflowId })
    │
    └─► bus.emit('ui:unlock')
            │
            ▼
        FlowEditor: 恢复编辑功能
```

数据持久化

存储内容

只有 metadata 需要持久化，running-state 不存储。

```typescript
interface WorkflowRecord {
  id: string;
  name: string;
  data: string;  // JSON.stringify(WorkflowDefinition)
  createdAt: number;
  updatedAt: number;
}
```

序列化

```typescript
function serializeWorkflow(workflow: WorkflowDefinition): string {
  return JSON.stringify({
    ...workflow,
    nodes: Array.from(workflow.nodes.entries())
  });
}

function deserializeWorkflow(data: string): WorkflowDefinition {
  const parsed = JSON.parse(data);
  return {
    ...parsed,
    nodes: new Map(parsed.nodes)
  };
}
```

冻结块的持久化处理

当用户保存工作流时，如果存在已冻结的虚拟块：

1. 提示用户："检测到冻结的虚拟块，是否转换为普通文本块？"
2. 如果用户同意，将 `VirtualTextBlockDef` 转换为 `TextBlock`，内容为 `resolvedContent`
3. 如果用户拒绝，保存原始的 `VirtualTextBlockDef`（下次加载时内容将丢失）

```typescript
function convertFrozenBlocksToText(
  workflow: WorkflowDefinition,
  frozenStates: Map<TextBlockId, VirtualBlockState>
): WorkflowDefinition {
```

```
  // 遍历所有节点的所有 TextBlockList
  // 将已冻结的 VirtualTextBlockDef 转换为 TextBlock
}
```

完整示例

定义工作流

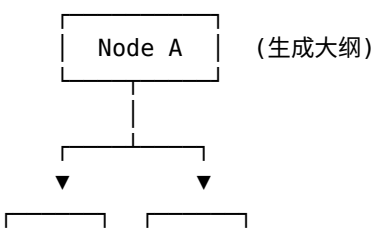
```
import {
  createWorkflowDef,
  createNodeDef,
  addNode,
  createTextBlockList,
  createTextBlock,
  createVirtualBlockDef
} from '$lib/core';

// 创建工作流
let workflow = createWorkflowDef('文章润色工作流');

// 节点 A: 生成大纲
const nodeA = createNodeDef('生成大纲', { x: 100, y: 100 });
nodeA.apiConfig = {
  ...nodeA.apiConfig,
  systemPrompt: createTextBlockList([
    createTextBlock('你是一个专业的文章大纲生成助手。')
  ]),
  userPrompt: createTextBlockList([
    createTextBlock('请为以下主题生成一个文章大纲：\n\n人工智能的未来发展')
  ])
};

// 节点 B: 扩写第一部分（依赖 A）
const nodeB = createNodeDef('扩写第一部分', { x: 100, y: 250 });
nodeB.apiConfig = {
  ...nodeB.apiConfig,
  userPrompt: createTextBlockList([
    createTextBlock('基于以下大纲，扩写第一部分：\n\n'),
    createVirtualBlockDef(nodeA.id, '大纲')
  ])
};

// 节点 C: 扩写第二部分（依赖 A）
const nodeC = createNodeDef('扩写第二部分', { x: 300, y: 250 });
nodeC.apiConfig = {
  ...nodeC.apiConfig,
  userPrompt: createTextBlockList([
    createTextBlock('基于以下大纲，扩写第二部分：\n\n'),
    createVirtualBlockDef(nodeA.id, '大纲')
  ])
};

// 节点 D: 合并润色（依赖 B 和 C）
const nodeD = createNodeDef('合并润色', { x: 200, y: 400 });
nodeD.apiConfig = {
  ...nodeD.apiConfig,
```

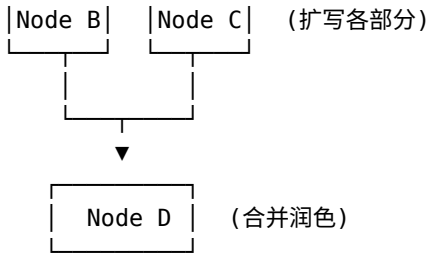
```
  userPrompt: createTextBlockList([
    createTextBlock('请将以下两部分内容合并并润色：\n\n第一部分：\n'),
    createVirtualBlockDef(nodeB.id, '第一部分'),
    createTextBlock('\n\n第二部分：\n'),
    createVirtualBlockDef(nodeC.id, '第二部分')
  ])
};

// 添加节点
workflow = addNode(workflow, nodeA);
workflow = addNode(workflow, nodeB);
workflow = addNode(workflow, nodeC);
workflow = addNode(workflow, nodeD);

// 保存到数据库
await saveWorkflow(workflow);
```

执行工作流

```
import { bus } from '$lib/bus';
import { WorkflowRunner } from '$lib/core-runner';

// UI 层监听事件
const unsubscribes = [
  bus.on('ui:lock', () => { isLocked = true; }),
  bus.on('ui:unlock', () => { isLocked = false; }),
  bus.on('node:state', ({ nodeId, state }) => {
    updateNodeUI(nodeId, state);
  }),
  bus.on('node:output', ({ nodeId, content }) => {
    displayNodeOutput(nodeId, content);
  }),
  bus.on('workflow:done', () => {
    showNotification('工作流执行完成');
  }),
  bus.on('workflow:error', ({ error }) => {
    showError(error);
  })
];

// 加载工作流
const workflow = await loadWorkflow(workflowId);

// 创建 Runner 并执行
const runner = new WorkflowRunner(workflow);

// 触发执行（通过 EventBus）
bus.emit('workflow:run', { workflowId: workflow.id });
```

依赖图





拓扑排序结果：`[A, B, C, D]` 或 `[A, C, B, D]`

执行时 B 和 C 可并行（未来优化方向）。

与 v1 的对比

方面	v1	v2
Node 类型	包含 state/output	只有 metadata
VirtualTextBlock	包含 state/resolvedContent/frozen	只有 sourceNodeId
Workflow	包含 executionOrder/currentIndex	只有 nodes
通信机制	无	EventBus
运行时状态	混在 core/	独立的 core-runner/
持久化	需要过滤字段	直接序列化
UI 锁定	无	有

迁移影响

1. `core/node.ts` → 重命名为 `NodeDefinition`，移除 state/output
2. `core/textblock.ts` → `VirtualTextBlock` 重命名为 `VirtualTextBlockDef`，移除运行时字段
3. `core/workflow.ts` → 重命名为 `WorkflowDefinition`，移除运行时字段，`executeWorkflow` 移至 `core-runner/`
4. 新增 `core-runner/` 目录
5. 新增 `bus/` 目录

未来扩展

流式输出

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

UI 层根据 `streaming` 和 `done` 决定如何显示。

节点级重试

```
'node:retry': { nodeId: NodeId }
```

WorkflowRunner 重新执行指定节点。

并行执行

将拓扑排序结果按“层”分组，同一层的节点使用 `Promise.all` 并行执行。

条件节点

```
interface ConditionNodeDefinition {
  type: 'condition';
  id: NodeId;
  name: string;
  condition: string;  // 表达式
  trueBranch: NodeId;
  falseBranch: NodeId;
}
```

当前不实现，预留扩展点。