

FlowWrite 消息总线可行性分析

日期: 2026.01.19

本文档调研消息总线在 FlowWrite 中的可行性，并提出精简的实现方案。

目录

FlowWrite 消息总线可行性分析	1
背景与动机	2
当前架构的通信模式	2
现有架构的痛点	2
痛点一: Core 层与 UI 层断裂	2
痛点二: 跨组件通信困难	2
痛点三: 未来 core-runner 的集成	2
消息总线的设计目标	2
关键设计决策	2
决策一: 不需要状态共享层	2
分析	2
结论	3
决策二: 分离 metadata 与 running-state	3
设计原则	3
具体划分	3
好处	3
精简架构	3
架构图	3
数据流	4
典型执行流程	4
消息方向	4
EventBus 实现	4
类型定义	4
EventBus 实现	5
使用示例	5
实施计划	6
阶段一: EventBus 基础设施	6
任务	6
文件结构	6
阶段二: core 层重构	6
任务	6
阶段三: core-runner 实现	6
任务	6
阶段四: UI 集成	7
任务	7
风险与缓解	7
风险一: 内存泄漏	7
风险二: 消息顺序	7
风险三: 调试困难	7
结论	7

最终方案	7
核心收益	8
下一步	8

背景与动机

当前架构的通信模式

FlowWrite 当前使用以下通信模式：

1. **Props/Binding 通信**: 父子组件通过 `$props()` 和 `bind:` 传递数据
2. **Event Dispatch**: 子组件通过 `createEventDispatcher()` 向父组件发送事件
3. **状态提升**: FlowEditor 作为中心 hub, 管理所有节点、边、验证状态
4. **数据库持久化**: IndexedDB 仅用于存储, 不参与实时通信

现有架构的痛点

痛点一：Core 层与 UI 层断裂

UI 层的 `@xyflow/svelte` 节点模型与核心层的 Node 模型是分离的：

UI 层: { id, type, position, data: { label, status, model } }
Core 层: { id, name, apiConfig, output, state, position }

痛点二：跨组件通信困难

兄弟组件（如 FloatingBall 与 FlowEditor）无法直接通信，必须通过状态提升或数据库间接通道。

痛点三：未来 core-runner 的集成

core-runner 将与 UI 层有频繁的沟通，当前架构无法优雅支持执行状态的实时同步。

消息总线的设计目标

1. **UI → Core → UI 三层通信**: 最常见的模式，不会形成 `UI→core→UI→core` 的链式死循环
 2. **UI 层锁定机制**: 防止状态机混乱
 3. **FloatingBall 扩展**: 作为与本地存储沟通的桥梁
 4. **core-runner 集成**: 支持工作流执行时的状态同步
-

关键设计决策

决策一：不需要状态共享层

分析

状态共享层（如 Svelte Stores）在本项目中是多余的，原因如下：

1. **core-runner 的共享数据是易失的**: 虚拟文本块的输出内容、冻结状态等都是运行时状态，不需要持久化共享
2. **core-runner 只需与 db 交互**: 底层组件只有数据库，不存在多个消费者需要同步状态的场景
3. **IR 类比**: 中间表示 (IR) 只在有多种架构和多种系统时才有必要；单一架构单一系统下，IR 是冗余的

结论

消息总线足够，不需要额外的状态共享层。

决策二：分离 metadata 与 running-state

设计原则

- core/ → 只包含 metadata (静态定义)
core-runner/ → 包含 running-state (运行时状态)

具体划分

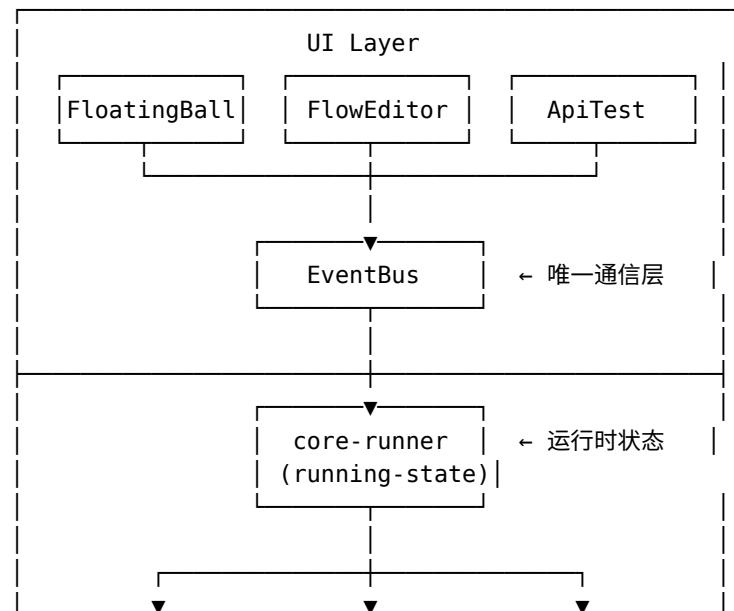
数据	归属	说明
Node.id, name, position	core/	元数据，持久化
ApiConfiguration	core/	元数据，持久化
TextBlock, VirtualTextBlock 定义	core/	元数据，持久化
running pending failure	core-runner/	运行时状态，易失
freeze: true false	core-runner/	运行时状态，易失
VirtualTextBlock.resolvedContent	core-runner/	运行时状态，易失

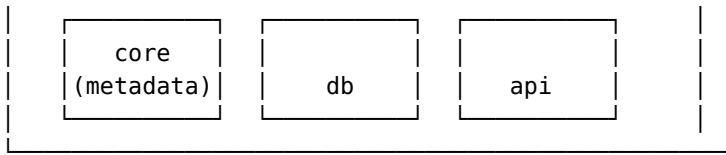
好处

- core/ 层保持简洁，只描述“是什么”
- core-runner/ 层负责“怎么运行”
- 保存工作流时只需序列化 core/ 层的 metadata
- 冻结的虚拟文本块可建议用户转换为普通文本块再保存

精简架构

架构图





数据流

典型执行流程

1. UI: 用户点击执行
FlowEditor → bus.emit('workflow:run', { workflowId })
2. core-runner: 接收并准备执行
bus.on('workflow:run') → 从 db 加载 workflow metadata
→ 初始化 running-state → bus.emit('ui:lock')
3. UI: 收到锁定
bus.on('ui:lock') → 禁用编辑
4. core-runner: 执行节点
对每个节点: bus.emit('node:state', { nodeId, state: 'running' })
调用 api → bus.emit('node:output', { nodeId, content })
5. core-runner: 完成
bus.emit('workflow:done') → bus.emit('ui:unlock')
6. UI: 解锁并更新显示

消息方向

UI → core-runner:
workflow:run, workflow:stop, node:freeze, node:unfreeze

core-runner → UI:
node:state, node:output, workflow:done, workflow:error
ui:lock, ui:unlock

FloatingBall ↔ db:
storage:sync, storage:export, storage:import

EventBus 实现

类型定义

```

// lib/bus/events.ts
import type { NodeId } from '$lib/core/node';

export interface BusEvents {
    // UI → core-runner
    'workflow:run': { workflowId: string };
    'workflow:stop': void;
    'node:freeze': { nodeId: NodeId };
    'node:unfreeze': { nodeId: NodeId };

    // core-runner → UI
    'node:state': { nodeId: NodeId; state: '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;

// FloatingBall 存储
'storage:sync': void;
'storage:export': { format: 'json' };
'storage:import': { data: string };
}

}

```

EventBus 实现

```

// lib/bus/eventbus.ts
import type { BusEvents } from './events';

type Callback<T> = (payload: T) => void;

class EventBus {
    private listeners = new Map<string, Set<Callback<unknown>>>();

    on<K extends keyof BusEvents>(
        event: K,
        callback: Callback<BusEvents[K]>
    ): () => void {
        if (!this.listeners.has(event)) {
            this.listeners.set(event, new Set());
        }
        this.listeners.get(event)!.add(callback as Callback<unknown>);

        // 返回 unsubscribe 函数
        return () => this.off(event, callback);
    }

    off<K extends keyof BusEvents>(
        event: K,
        callback: Callback<BusEvents[K]>
    ): void {
        this.listeners.get(event)?._delete(callback as Callback<unknown>);
    }

    emit<K extends keyof BusEvents>(event: K, payload: BusEvents[K]): void {
        this.listeners.get(event)?._forEach(cb => cb(payload));
    }
}

export const bus = new EventBus();

```

使用示例

```

// FlowEditor.svelte
import { bus } from '$lib/bus';
import { onMount, onDestroy } from 'svelte';

```

```

let isLocked = $state(false);
let nodeStates = $state<Map<string, string>>(new Map());
const unsubscribes: ((() => void)[] = []);

onMount(() => {
  unsubscribes.push(
    bus.on('ui:lock', () => { isLocked = true; }),
    bus.on('ui:unlock', () => { isLocked = false; }),
    bus.on('node:state', ({ nodeId, state }) => {
      nodeStates.set(nodeId, state);
      nodeStates = new Map(nodeStates); // trigger reactivity
    })
  );
});

onDestroy(() => unsubscribes.forEach(fn => fn()));

function handleRun() {
  bus.emit('workflow:run', { workflowId: currentWorkflowId });
}

```

实施计划

阶段一：EventBus 基础设施

任务

1. 创建 src/lib/bus/ 目录
2. 实现 events.ts (类型定义)
3. 实现 eventbus.ts (EventBus 类)
4. 导出 index.ts

文件结构

```

src/lib/bus/
├── index.ts      # export { bus } from './eventbus'
└── eventbus.ts   # EventBus 实现
  └── events.ts   # BusEvents 类型

```

阶段二：core 层重构

任务

1. 移除 core/ 中的运行时状态字段
2. Node 只保留 metadata: id, name, position, apiConfig
3. TextBlock 只保留定义: id, content, sourceNodeId
4. 确保 core/ 的所有类型都是可序列化的

阶段三：core-runner 实现

任务

1. 创建 src/lib/core-runner/ 目录
2. 实现 WorkflowRunner 类
 - 持有 running-state (节点状态、输出内容)
 - 监听 EventBus 命令

- 发送状态更新到 EventBus
3. 集成 api/client.ts 进行 LLM 调用

阶段四：UI 集成

任务

1. FlowEditor 订阅 EventBus 事件
 2. 实现 UI 锁定逻辑
 3. FloatingBall 集成存储功能
 4. 节点状态可视化
-

风险与缓解

风险一：内存泄漏

问题：订阅未清理导致内存泄漏。

缓解：

- on() 返回 unsubscribe 函数
- 组件 onDestroy 中统一清理
- 开发模式下监控订阅数量

风险二：消息顺序

问题：异步操作可能导致消息顺序混乱。

缓解：

- core-runner 内部使用队列处理命令
- 状态变更携带时间戳
- UI 锁定期间忽略用户操作

风险三：调试困难

问题：消息传递难以追踪。

缓解：

- 开发模式下 emit() 输出日志
 - 可选的消息历史记录
 - 未来可添加 devtools 插件
-

结论

最终方案

单一 EventBus，无状态共享层

- EventBus 作为唯一的跨组件通信机制
- core/ 只包含 metadata (可序列化)
- core-runner/ 持有 running-state (易失)
- 不引入 Svelte Stores 或第三方库

核心收益

1. 架构简洁：单一通信机制，无冗余层
2. 职责清晰：metadata vs running-state 分离
3. 类型安全：BusEvents 接口约束所有消息
4. 易于扩展：新功能只需定义新消息类型

下一步

确认方案后，按阶段实施：EventBus → core 重构 → core-runner → UI 集成