

FlowWrite 消息总线可行性分析

日期: 2026.01.19

本文档基于 `handwrite.typ` 中的思考, 调研消息总线在 FlowWrite 中的可行性, 分析现有架构痛点, 并提出具体实现方案。

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
- 方案对比 3
 - 方案一: Svelte Stores 3
 - 实现示例 3
 - 优点 3
 - 缺点 3
 - 适用场景 3
 - 方案二: 自定义 EventBus 3
 - 实现示例 3
 - 类型安全版本 4
 - 优点 5
 - 缺点 5
 - 适用场景 5
 - 方案三: 使用第三方库 5
 - 选项 A: mitt (200B) 5
 - 选项 B: nanostores (300B) 5
 - 选项 C: RxJS (大型) 6
 - 对比表格 6
- 推荐方案: 混合架构 6
 - 架构设计 6
 - 消息流设计 7
 - UI → Core → UI 典型流程 7
 - UI 锁定机制 7
 - FloatingBall 存储桥接 8
- 实现计划 8
 - 阶段一: 基础设施 (1-2 天) 8
 - 任务 8
 - 文件结构 8
 - 阶段二: Core Runner 集成 (2-3 天) 9
 - 任务 9
 - 阶段三: 桥接层实现 (2-3 天) 9
 - 任务 9

阶段四：优化与测试（1-2 天）	9
任务	9
风险与缓解	9
风险一：状态同步复杂度	9
风险二：内存泄漏	9
风险三：性能瓶颈	9
结论	10
核心收益	10
技术选型	10
下一步	10

背景与动机

当前架构的通信模式

通过代码分析，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

`plan_doc.typ` 中已明确指出，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 }
```

两者之间缺乏桥接机制，导致：

- 无法使用 `VirtualTextBlock` 的依赖追踪
- 执行引擎无法作用于 UI 节点
- 状态变化无法自动同步

痛点二：跨组件通信困难

当前架构下，兄弟组件（如 `FloatingBall` 与 `FlowEditor`）无法直接通信，必须通过：

1. 状态提升到共同父组件（`App.svelte`）
2. 或使用数据库作为间接通道

痛点三：未来 `core-runner` 的集成

`handwrite.typ` 提到 `core-runner` 将与 UI 层有频繁通信，当前架构无法优雅支持：

- 执行状态的实时同步
- 流式输出的 UI 更新
- 错误处理与恢复

消息总线的设计目标

基于 `handwrite.typ` 的分析，消息总线需要支持：

1. **UI → Core → UI 三层通信**：这是最常见的模式，不会形成闭环

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
-

方案对比

方案一：Svelte Stores

Svelte 5 兼容的 writable/readable stores，已内置于框架。

实现示例

```
// stores/workflow.ts
import { writable, derived } from 'svelte/store';
import type { Workflow } from '$lib/core/workflow';

export const workflowStore = writable<Workflow | null>(null);
export const executionState = writable<'idle' | 'running' | 'completed'>('idle');

// 派生状态
export const isExecuting = derived(executionState, $state => $state === 'running');
```

优点

- 零依赖，Svelte 原生支持
- 与 Svelte 5 Runes 兼容（通过 \$ 前缀访问）
- 自动订阅/取消订阅
- 简单的 pub/sub 模式

缺点

- 缺乏消息类型区分（只有值变化，没有消息语义）
- 不支持消息队列或优先级
- 难以实现复杂的消息路由

适用场景

适合简单的状态共享，不适合复杂的消息传递。

方案二：自定义 EventBus

基于浏览器 CustomEvent 或简单的 pub/sub 实现。

实现示例

```
// lib/bus/eventbus.ts
type EventCallback<T = unknown> = (payload: T) => void;

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

  on<T>(event: string, callback: EventCallback<T>): () => void {
    if (!this.listeners.has(event)) {
      this.listeners.set(event, new Set());
    }
    this.listeners.get(event)!.add(callback as EventCallback);
    return () => this.off(event, callback);
  }
}
```

```

off<T>(event: string, callback: EventCallback<T>): void {
    this.listeners.get(event)?.delete(callback as EventCallback);
}

emit<T>(event: string, payload: T): void {
    this.listeners.get(event)?.forEach(cb => cb(payload));
}

}

export const bus = new EventBus();

```

类型安全版本

```

// lib/bus/typed-eventbus.ts
import type { NodeId, Node } from '$lib/core/node';
import type { Workflow } from '$lib/core/workflow';

// 消息类型定义
interface BusEvents {
    // UI → Core
    'node:execute': { nodeId: NodeId };
    'workflow:run': { workflowId: string };
    'workflow:stop': void;

    // Core → UI
    'node:state-changed': { nodeId: NodeId; state: Node['state'] };
    'node:output-ready': { nodeId: NodeId; content: string };
    'workflow:progress': { current: number; total: number };
    'workflow:completed': { workflow: Workflow };
    'workflow:error': { error: string; nodeId?: NodeId };

    // FloatingBall 相关
    'storage:sync-request': void;
    'storage:sync-complete': { success: boolean };

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

class TypedEventBus {
    private listeners = new Map<string, Set<Function>>();

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

    off<K extends keyof BusEvents>({
        event: K,

```

```

    callback: (payload: BusEvents[K]) => void
  ): void {
    this.listeners.get(event)?.delete(callback);
  }

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

export const bus = new TypedEventBus();

```

优点

- 完全自定义，符合项目需求
- 类型安全
- 零依赖
- 支持任意消息语义

缺点

- 需要自行实现
- 缺乏高级功能（消息队列、中间件、持久化）
- 调试困难（没有 devtools）

适用场景

适合 FlowWrite 的规模，能满足 UI↔Core 通信需求。

方案三：使用第三方库

选项 A: mitt (200B)

极简的 EventEmitter，功能类似方案二。

```

import mitt from 'mitt';

type Events = {
  'node:execute': { nodeId: string };
  'node:complete': { nodeId: string; output: string };
};

const emitter = mitt<Events>();
emitter.on('node:execute', ({ nodeId }) => { /* ... */ });
emitter.emit('node:execute', { nodeId: 'node-1' });

```

选项 B: nanostores (300B)

轻量级状态管理，支持 computed 和 actions。

```

import { atom, computed, action } from 'nanostores';

const $workflow = atom<Workflow | null>(null);
const $isRunning = computed($workflow, w => w?.state === 'running');

const runWorkflow = action($workflow, 'run', (store) => {
  const workflow = store.get();
  // 执行逻辑
});

```

选项 C: RxJS (大型)

功能强大但体积大，适合复杂的响应式需求。

```
import { Subject, filter, map } from 'rxjs';

interface Message { type: string; payload: unknown }
const bus$ = new Subject<Message>();

// 订阅特定消息
bus$.pipe(
  filter(m => m.type === 'node:execute'),
  map(m => m.payload as { nodeId: string })
).subscribe(({ nodeId }) => { /* ... */ });
```

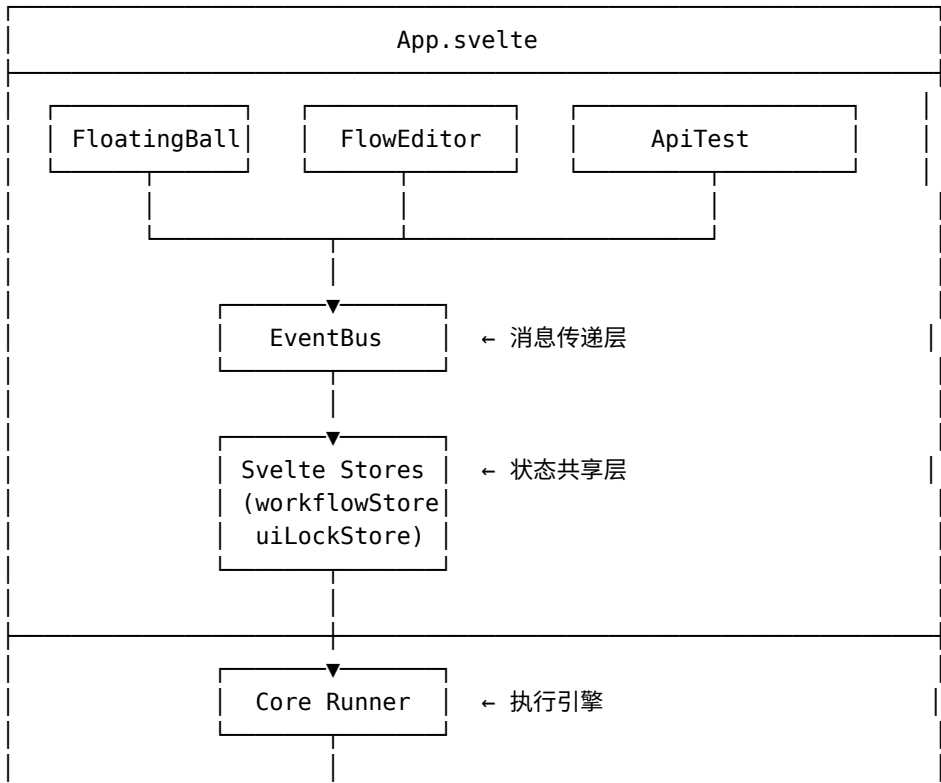
对比表格

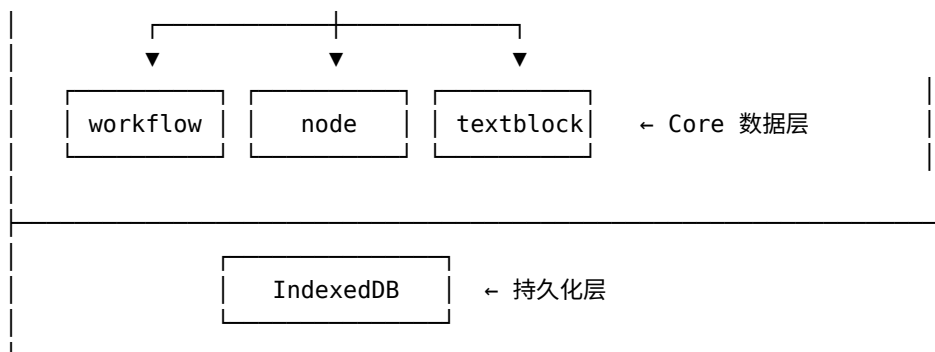
方案	体积	类型安全	学习成本	功能
自定义 EventBus	50 行	✅ 完全控制	低	基础
mitt	200B	✅ 泛型	低	基础
nanostores	300B	✅ 良好	中	状态+计算
RxJS	30KB	✅ 完整	高	强大

推荐方案：混合架构

基于 FlowWrite 的实际需求，推荐采用 自定义 TypedEventBus + Svelte Stores 的混合方案。

架构设计





消息流设计

UI → Core → UI 典型流程

1. 用户点击 "执行工作流"
FlowEditor → bus.emit('workflow:run', { workflowId })
2. Core Runner 接收并开始执行
bus.on('workflow:run') → prepareWorkflow() →
bus.emit('ui:lock', { reason: 'executing' })
3. UI 收到锁定消息
bus.on('ui:lock') → 禁用编辑功能
4. 节点依次执行
executeNode() → bus.emit('node:state-changed', { nodeId, state: 'running' })
LLM API 调用 → bus.emit('node:output-ready', { nodeId, content })
5. 执行完成
bus.emit('workflow:completed', { workflow })
bus.emit('ui:unlock')
6. UI 解锁, 更新显示
bus.on('workflow:completed') → 更新节点状态显示
bus.on('ui:unlock') → 恢复编辑功能

UI 锁定机制

```

// stores/uiLock.ts
import { writable, derived } from 'svelte/store';

interface LockState {
  locked: boolean;
  reason: string | null;
  lockedAt: number | null;
}

export const uiLockStore = writable<LockState>({
  locked: false,
  reason: null,
  lockedAt: null
});

export const isUILocked = derived(uiLockStore, $lock => $lock.locked);

// 在 EventBus 监听器中更新
  
```

```

bus.on('ui:lock', ({ reason }) => {
  uiLockStore.set({ locked: true, reason, lockedAt: Date.now() });
});

bus.on('ui:unlock', () => {
  uiLockStore.set({ locked: false, reason: null, lockedAt: null });
});

```

FloatingBall 存储桥接

```

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

let unsubscribes: (() => void)[] = [];

onMount(() => {
  // 监听同步请求
  unsubscribes.push(
    bus.on('storage:sync-request', async () => {
      try {
        await syncToIndexedDB();
        bus.emit('storage:sync-complete', { success: true });
      } catch (e) {
        bus.emit('storage:sync-complete', { success: false });
      }
    })
  );
});

onDestroy(() => {
  unsubscribes.forEach(unsub => unsub());
});

```

实现计划

阶段一：基础设施 (1-2 天)

任务

1. 创建 src/lib/bus/ 目录
2. 实现 TypedEventBus 类
3. 定义消息类型接口 BusEvents
4. 创建 src/lib/stores/ 目录
5. 实现 workflowStore 和 uiLockStore
6. 添加单元测试

文件结构

```

src/lib/
├── bus/
│   ├── index.ts           # 导出 bus 实例
│   ├── eventbus.ts        # TypedEventBus 实现
│   └── events.ts          # BusEvents 类型定义
├── stores/
└── index.ts               # 导出所有 stores

```

		workflow.ts	# workflow状态
		ui.ts	# UI 锁定状态

阶段二：Core Runner 集成 (2-3 天)

任务

- 创建 src/lib/runner/ 目录
- 实现 WorkflowRunner 类，使用 EventBus 发送状态
- 将 workflow.ts 中的 executeWorkflow 包装为 Runner
- 在 FlowEditor 中订阅执行事件
- 实现 UI 锁定逻辑

阶段三：桥接层实现 (2-3 天)

任务

- 实现 Core Node ↔ SvelteFlow Node 的双向转换
- 在 FlowEditor 中使用 workflowStore 作为数据源
- 实现节点编辑时的自动同步
- 实现 FloatingBall 的存储桥接功能

阶段四：优化与测试 (1-2 天)

任务

- 添加消息日志 (开发模式)
- 实现错误边界处理
- 性能测试与优化
- 集成测试

风险与缓解

风险一：状态同步复杂度

问题：Core 层与 UI 层的双向同步可能导致状态不一致。

缓解：

- 明确单向数据流：UI 操作 → EventBus → Core 更新 → Store 更新 → UI 响应
- 所有状态变更通过 EventBus，便于追踪和调试
- 添加状态快照和回滚机制

风险二：内存泄漏

问题：订阅未正确清理可能导致内存泄漏。

缓解：

- on() 方法返回 unsubscribe 函数
- 在组件 onDestroy 中清理所有订阅
- 使用 Svelte 的自动订阅语法 \$store

风险三：性能瓶颈

问题：频繁的消息传递可能影响性能。

缓解：

- 合并相近的状态更新 (debounce)
 - 流式输出使用 `requestAnimationFrame` 限流
 - 监控消息频率，必要时添加队列
-

结论

消息总线在 FlowWrite 中是 **可行且推荐** 的。

核心收益

1. **解耦 Core 与 UI**: 通过 EventBus 实现松耦合通信
2. **状态可追踪**: 所有状态变更通过消息，便于调试
3. **扩展性**: 新功能 (如 FloatingBall 扩展) 可轻松接入
4. **锁定机制**: 防止执行期间的状态混乱

技术选型

- **消息传递**: 自定义 TypedEventBus (50 行, 类型安全)
- **状态共享**: Svelte Stores (零依赖, 框架原生)
- **无需引入**: mitt、RxJS 等第三方库 (项目规模不需要)

下一步

1. 确认方案后, 创建 `src/lib/bus/` 模块
2. 定义完整的 BusEvents 消息类型
3. 实现 Core Runner 并集成到 FlowEditor