

Eko Trace/Monitor 系统一体化设计

架构与依赖（以 Debug Session 为中心）

Debug Session 是监控/调试系统的核心抽象，与 Task 一一绑定（`sessionId === taskId`），负责聚合任务执行全链路的数据与快照，并通过 Callback 消息驱动向外广播事件（可被 WebSocket 服务消费），实现“数据即事件”的面板更新与可观测能力。

核心依赖关系：

- Debug Session \rightleftharpoons Task Context：双向绑定。Session 记录 `taskId`、工作流（`AgentTree`）、资源统计、事件序列；Context 提供对话与变量池的引用（或引用 id）。
- Callback Message Bus：所有拦截到的数据统一封装为 `message` 并投递到回调函数，属性 `message.type` 标示信息类型。
- 无侵入拦截（HOF Wrapper）：使用高阶函数包裹 Eko 的关键入口（`run/generate/execute`、Agent Prompt 构建、LLM 请求、Tool 执行等），执行前/后采集数据，同步写入 Session，并通过 Callback 立即广播。
- WebSocket Endpoint：对外提供基于 `sessionId` 的订阅通道，将 Callback 转化为前端可消费的事件流。
- 存储：内存为主、Postgres 为辅（JSONB）。Session 元数据与事件可批量持久化，支持历史检索；上下文快照用于 TimeMachine 重放。

采集目标（执行前读取、执行后确认）：

- Workflow 结构（`AgentTree`）、拓扑与并行/依赖关系。
- Agent 的 System/User Prompt。
- Agent 输出文本流（包含思考/文本片段、最终结果）。
- Tool 参数与结果（含错误信息）。
- 资源计量（tokens、耗时）、状态迁移（`planning/running/completed/error` 等）。

设计准则：

- 非侵入：HOF 包裹，保证原语义与类型不变。
- 消息驱动：统一格式的 `message` 与 `callback`，WS 仅做转发。
- 即时+持久：先同步写入 Session 与广播，再批量异步落库。
- 可重放：原子化节点粒度的上下文快照，支持任意节点重放。

概述

通过装饰式 Hook 无侵入获取执行数据，归档到按 Task 绑定的 Debug Session，并通过 WebSocket 对外持续推送动态更新，支持可选的 Postgres 持久化。

目标：

- 无侵入集成（Decorator/Wrapper Hook）
- 按任务维度聚合（Debug Session = Task ID 绑定）
- 实时可视化（WebSocket 事件流）
- 可选持久化（Postgres/SQLite，JSONB 记录原始与结构化数据）

- 最小实现与清晰使用方式，避免过度设计

架构总览

关键组件（围绕 Debug Session）

- Debug Session（会话）：以 `taskId` 为主键聚合所有事件、统计与元数据；内存缓存 + 可选持久化。
- TraceCollector（收集器）：包装 `StreamCallback`、LLM 调用、工具执行、执行链入口，统一产出 `MonitorEvent` 并路由到 Session。
- Prompt/Tool/Chain 装饰器：面向具体 Hook 点的轻量包装器，尽量少做逻辑，仅采集必要元数据。
- EventBroadcaster（WS）：推送 `session_update` 与 `monitor_event`，客户端基于 `taskId` 订阅。
- Storage（存储）：Memory/Redis 作为热数据缓存，Postgres（JSONB）作为历史归档，异步落库。

数据流（简化）

1. Eko.run/generate/execute（HOF 包裹）→ 读取 Workflow（AgentTree）→ 产出 `workflow_*` 事件。
2. Agent 执行（HOF 包裹）→ 捕获 System/User Prompt、LLM 请求/响应流、Agent 文本输出 → 产出 `prompt_*`、`llm_*`、`agent_*` 事件。
3. Tool 执行（HOF 包裹）→ 捕获参数与返回值/错误 → 产出 `tool_*` 事件。
4. 所有事件 → 构造 `message` 标注 `type` → 交由 `callback(message)` 处理。
5. TraceCollector ← Callback：写入 Debug Session（同步）→ EventBroadcaster（WS 推送）→ Storage（批量异步落库，Postgres）。

```

flowchart TD
    A["Eko.run/generate/execute  
HOF 包裹"] --> B["读取 Workflow/AgentTree"]
    B --> C["产出事件"]
    C --> D["message type=workflow"]
    A --> E["Agent 执行  
捕获 Prompt/LLM"]
    E --> F["产出事件"]
    F --> G["message type=prompt"]
    F --> H["message type=llm"]
    A --> I["Tool 执行  
捕获参数/结果"]
    I --> J["产出事件"]
    J --> K["message type=tool"]

    D --> L["callback(message)"]
    G --> L
    H --> L
    K --> L

    L --> M["TraceCollector  
写入 Debug Session"]
    M --> N["EventBroadcaster  
WS 推送 monitor_event"]
    N --> O["Storage  
批量异步落库(Postgres)"]
  
```

TimeMachine：重放与上下文快照

TimeMachine 提供“从任意 workflow 节点回溯并重放”的能力，核心是原子化节点前的上下文深拷贝快照与节点到快照的映射：`Pair<List<AgentNode>, Context>`。

快照策略

- 采集时机：每个原子化节点执行前（即将被哪些节点引用），记录 `nodeIds` 与对应的 Context 拷贝，存入 `snapshots`。

重放算法

输入：`sessionId, nodeId`

- 定位快照：根据 `nodeId` 从 `snapshots` 查找最近一次包含该 `nodeId` 的快照 `S`。
- 恢复 Context：将当前 Task Context 替换为 `S`。
- 执行重放节点：在恢复后的 Context 上执行 `nodeId` 对应的 Node 操作，得到新 Context `S'`。
- 级联更新映射：将重放节点的直接子节点映射到 `S'`；
- 清理分支：对重放节点下属所有树枝，从第二级子节点开始的 Node 和 Context 映射全部清除（等待按需重建）。

复杂度与注意：

- 快照频率与体积需受控（可采样/增量快照）；
- 恢复需与外部副作用隔离（文件/网络/时间），建议在工具层提供“虚拟模式”或幂等接口；
- 与并行执行兼容时，优先按拓扑分层进行局部清理，避免误删并发兄弟分支。

核心概念与数据模型（精简）

```
// Debug Session (按 taskId 聚合)
interface MonitorSession {
    id: string; // == taskId
    workflow?: Workflow;
    resources?: ResourceUsage;
    startTime: Date;
    endTime?: Date;
    metadata?: Record<string, any>;
    // TimeMachine: 上下文快照索引
    snapshots?: Array<{
        id: string; // 快照 id
        ts: Date; // 采集时间
        nodeIds: string[]; // 将被立即引用到的节点（前置/就绪节点）
        contextRef: string; // Context 引用（在存储中指向深拷贝对象）
    }>;
}

// 事件（最小字段集）
interface MonitorEvent<T = any> {
    id: string;
    sessionId: string; // == taskId
    timestamp: Date;
```

```

    type: MonitorEventType; // 'workflow_*' | 'agent_*' | 'tool_*' |
    'llm_*' | 'prompt_*'
    level?: 'trace' | 'debug' | 'info' | 'warn' | 'error';
    category?: 'execution' | 'prompt' | 'tool' | 'performance' | 'system';
    data: T; // 按类型承载具体负载
}

type MonitorEventType =
  | 'workflow_plan_start' | 'workflow_plan_complete'
  | 'workflow_execute_start' | 'workflow_execute_complete'
  | 'agent_start' | 'agent_complete' | 'agent_error'
  | 'prompt_build' | 'llm_request_start' | 'llm_request_complete'
  | 'tool_call_start' | 'tool_call_complete' | 'tool_call_error';

```

装饰式 Hook 设计（无侵入）

仅通过包装既有入口，不改变业务语义：

- 包装 Eko: `generate/execute/run` → 记录工作流与链级状态
- 包装 Callback: `StreamCallback.onMessage` → 统一事件拦截
- 包装 Agent: `buildSystemPrompt/buildUserPrompt` 与 LLM 调用入口
- 包装 Tool: `tool.execute` → 记录参数校验、执行与结果

高阶函数（HOF）拦截规范

所有拦截点使用同一模式：

```

type Hook<TArgs extends any[], TReturn> = (fn: (...args: TArgs) =>
TReturn) => (...args: TArgs) => TReturn;

// 统一的消息结构
interface CallbackMessageBase {
  taskId: string;
  type: string; // e.g. 'workflow', 'prompt', 'llm',
  'tool', 'agent', 'session'
  ts: number;
  payload: Record<string, unknown>;
}

type CallbackFn = (message: CallbackMessageBase) => void | Promise<void>;

// HOF 模板：执行前采集、执行后采集、错误采集（同步写 Session + 发送回调）
function wrap<TArgs extends any[], TReturn>(
  fn: (...args: TArgs) => TReturn,
  before: (args: TArgs) => void,
  after: (ret: TReturn, args: TArgs) => void,
  onError: (err: unknown, args: TArgs) => void
) {
  return (...args: TArgs) => {
    before(args);
    try {

```

```

    const r = fn(...args);
    if (r instanceof Promise) {
      return r.then((val) => {
        after(val, args);
        return val;
      }).catch((err) => {
        onError(err, args);
        throw err;
      }) as TReturn;
    }
    after(r, args);
    return r;
  } catch (e) {
    onError(e, args);
    throw e;
  }
} as (...args: TArgs) => TReturn;
}

```

统一收集器（核心拦截）

```

class TraceCollector {
  constructor(
    private storage: MonitorStorage,
    private broadcaster: EventBroadcaster
  ) {}

  // 统一包装现有回调
  interceptCallback(original?: StreamCallback): StreamCallback {
    return {
      onMessage: async (msg, agentCtx) => {
        await this.collect(msg, agentCtx); // 采集并路由
        await original?.onMessage?.(msg, agentCtx); // 保持原有行为
        this.broadcaster.broadcast(msg.taskId, { // 实时推送
          type: 'monitor_event',
          sessionId: msg.taskId,
          event: this.toEvent(msg, agentCtx)
        });
      }
    };
  }

  // 最小采集逻辑（示意）
  private async collect(msg: StreamCallbackMessage, agentCtx?: AgentContext) {
    await this.storage.ensureSession(msg.taskId, () => ({
      id: msg.taskId,
      status: 'running',
      currentPhase: 'execute',
      startTime: new Date()
    }));
  }
}

```

```

    await this.storage.appendEvent(this.toEvent(msg, agentCtx));
  }

  private toEvent(msg: StreamCallbackMessage, agentCtx?: AgentContext):
  MonitorEvent {
    return {
      id:
`${msg.taskId}_${Date.now()}_${Math.random().toString(36).slice(2,8)}`,
      sessionId: msg.taskId,
      timestamp: new Date(),
      type: this.mapType(msg.type),
      category: this.mapCategory(msg.type),
      level: 'info',
      data: { ...msg, agentId: agentCtx?.agent?.Id }
    };
  }

  private mapType(t: string): MonitorEventType {
    switch (t) {
      case 'workflow': return 'workflow_plan_complete';
      case 'agent_start': return 'agent_start';
      case 'tool_use': return 'tool_call_start';
      case 'tool_result': return 'tool_call_complete';
      case 'text':
      case 'thinking': return 'llm_request_complete';
      case 'finish': return 'workflow_execute_complete';
      default: return 'agent_complete';
    }
  }

  private mapCategory(t: string): MonitorEvent['category'] {
    if (t.startsWith('tool')) return 'tool';
    if (t === 'text' || t === 'thinking') return 'prompt';
    if (t === 'workflow') return 'execution';
    return 'execution';
  }
}

```

回调消息规范与 WS 映射

```

// callback 收到的 message (采集面向内部)
type Message =
  | { type: 'workflow'; taskId: string; ts: number; payload: { agentTree:
AgentTree; parallel?: boolean } }
  | { type: 'prompt'; taskId: string; ts: number; payload: { agentId:
string; role: 'system' | 'user'; text: string } }
  | { type: 'llm'; taskId: string; ts: number; payload: { agentId: string;
phase: 'start' | 'delta' | 'complete'; token?: string; usage?: any } }
  | { type: 'agent'; taskId: string; ts: number; payload: { agentId:
string; status: 'start' | 'complete' | 'error'; error?: string } }
  | { type: 'tool'; taskId: string; ts: number; payload: { agentId:

```

```

string; name: string; args?: any; result?: any; error?: string; phase:
'start' | 'complete' | 'error' } }
| { type: 'session'; taskId: string; ts: number; payload: { status:
'planning' | 'running' | 'paused' | 'completed' | 'error' | 'aborted' } } };

// WS 侧的事件 (面向外部)
type WsEvent =
| { type: 'session_update'; sessionId: string; data: { status?: string;
resources?: any; currentPhase?: string } }
| { type: 'monitor_event'; sessionId: string; event: MonitorEvent };

```

Agent/LLM 与 Tool 包装 (示例)

```

// Prompt/LLM Hook (仅示意, 不改变原有功能)
function wrapAgent(agent: Agent, collector: TraceCollector) {
  const buildSys = agent.buildSystemPrompt.bind(agent);
  agent.buildSystemPrompt = async (ctx, tools) => {
    const prompt = await buildSys(ctx, tools);
    collectorEvent('prompt_build', ctx, { kind: 'system', length:
prompt.length });
    return prompt;
  };

  const callLLM = agent.callLLM?.bind(agent);
  if (callLLM) {
    agent.callLLM = async (...args: any[]) => {
      collectorEvent('llm_request_start');
      const res = await callLLM(...args);
      collectorEvent('llm_request_complete', undefined, { usage: (res as
any)?.usage });
      return res;
    };
  }

  function collectorEvent(type: MonitorEventType, ctx?: AgentContext,
data?: any) {
    // 通过 TraceCollector 的公共入口写入事件 (略)
  }
}

// Tool Hook (示意)
function wrapTool(tool: Tool, collector: TraceCollector): Tool {
  const execute = tool.execute.bind(tool);
  return {
    ...tool,
    async execute(args, agentCtx, toolCall) {
      collectorEvent('tool_call_start', agentCtx, { tool: tool.name, args
});
      try {
        const ret = await execute(args, agentCtx, toolCall);
        collectorEvent('tool_call_complete', agentCtx, { tool: tool.name,

```

```

result: ret });
    return ret;
  } catch (err: any) {
    collectorEvent('tool_call_error', agentCtx, { tool: tool.name,
error: err?.message });
    throw err;
  }
}
} as Tool;

function collectorEvent(type: MonitorEventType, ctx?: AgentContext,
data?: any) {
  // 通过 TraceCollector 的公共入口写入事件（略）
}
}

```

与 Context 的引用策略（对话与变量池）

为避免重复存储与数据膨胀：

- 在消息中仅传 `contextRef`（如 `ctx.id` 或 `ctxPath`），Debug Session 中登记引用与统计。
- 通过公共 API 暴露只读查询：例如 `getConversation(contextRef, lastN)`、`getVariables(contextRef, keys)`。
- 需要快照时由 Storage 层执行 DeepCopy 并返回 `snapshotId`，写入 Session 的 `snapshots[*].contextRef`。

WebSocket 接口（实时推送）

服务端按 `taskId` 分组推送，消息类型尽量稳定：

```

// 订阅
// 客户端 → 服务器
{ type: 'subscribe', sessionId: '<taskId>' }

// 推送：会话更新
{ type: 'session_update', sessionId: '<taskId>', data: { status,
currentPhase, resources } }

// 推送：监控事件
{ type: 'monitor_event', sessionId: '<taskId>', event: MonitorEvent }

```

管理员可扩展：断线重连、心跳、限流与采样（生产环境）等。

最小使用方式

```

// 1) 初始化 Trace 系统
const traceSystem = new TraceSystem({
  enabled: true,

```



```
    realtime: { port: 8080 },
    storage: {
      type: 'postgres',
      connectionString: 'postgresql://user:pass@localhost:5432/eko_monitor'
    }
  });

await traceSystem.start(); // 启动存储与 WS 服务

// 2) 启用到 Eko 实例
const eko = new Eko(config);
const monitored = traceSystem.enable(eko); // 装饰式 Hook 自动生效

// 3) 正常使用 Eko, Trace 将自动采集并推送
const result = await monitored.run('分析这个文档');

// 4) 客户端订阅 (浏览器/Node)
const ws = new WebSocket('ws://localhost:8080');
ws.onopen = () => ws.send(JSON.stringify({ type: 'subscribe', sessionId:
result.taskId }));
ws.onmessage = (ev) => console.log('trace:', JSON.parse(ev.data));

// 5) (可选) 重放: 通过 TimeMachine API 触发
await traceSystem.timeMachine.replay({
  sessionId: result.taskId,
  nodeId: 'AgentNode_42'
});
```

小结

本 Trace/Monitor 方案以“装饰式 Hook + Debug Session + WS 实时推送 + Postgres 持久化”为主路径：

- 不侵入主流程，最小化实现复杂度；
- 以 `taskId` 聚合，洞察完整执行链路；
- 轻量实时可观测，支持后续 UI 与分析能力演进；
- 存储层 JSONB 结构灵活，兼顾查询与演进。