API 参考

本文档涵盖 `@table-git/memory-engine` 暴露的主要类型、类与函数。示例均以 TypeScript 撰写。

模块总览

```
import {
  FlowBuilder,
  NodeRuntime,
  NodeRegistry,
  registerBuiltinNodes,
  TableGitAdapter,
  FormatterRegistry,
  ComposedTagParser,
  EventBus
} from '@table-git/memory-engine';
```

核心 API 分五层：类型定义、解析&格式化、节点与运行时、适配器、扩展。

类型系统（`src/core/types.ts`）

ConversationTurn

```
interface ConversationTurn {
  id: string;
  role: 'user' | 'assistant' | 'system' | 'tool';
  content: string;
  timestamp?: string;
  metadata?: Record<string, unknown>;
}
```

描述单轮对话，供标签解析器消费。

TagInstruction

```
interface TagInstruction {
  tag: string;
  action: string;
  target?: Record<string, unknown>;
  payload?: Record<string, unknown>;
  confidence?: number;
  raw?: string;
}
```

标签解析结果，`ApplyChanges` 节点会将其转换为 `TableChangeCommand`。

FlowContext

```ts
interface FlowContext {
  conversation: ConversationTurn[];
  sheetId?: string;
  variables?: Record<string, unknown>;
  services: Record<string, unknown>;
}
```

在节点之间共享的上下文：

- `conversation`：对话数组
- `sheetId`：当前操作的表 ID
- `variables`：节点存储的临时变量（如 `instructions`、`lastSnapshot`）
- `services`：外部依赖，如缓存、Logger 等

NodeResult

```ts
interface NodeResult {
  outputs?: Record<string, unknown>;
  warnings?: string[];
  events?: Array<{ event: string; payload?: unknown }>;
}
```

节点执行可返回：

- `outputs`：写入状态机的结果（会以节点 ID 为 key 存入 `state`）
- `warnings`：运行警告
- `events`：额外事件，会通过全局 EventBus 下发

适配器（`src/core/table-adapter.ts`）

TableAdapter 接口

```ts
interface TableAdapter {
  load(sheetId: string): Promise<TableSnapshot>;
  applyChanges(
    sheetId: string,
    changes: TableChangeCommand[],
    options?: ApplyChangesOptions
  ): Promise<TableSnapshot>;
```

```
  snapshot(sheetId: string, options?: SnapshotOptions): Promise<TableSnapshot>;
}
```

- `load`：读取最新快照（通常包含 staged 变更）
- `applyChanges`：执行一组命令，返回更新后的快照
- `snapshot`：生成只读快照，可定制格式与元数据

TableChangeCommand

```
interface TableChangeCommand {
  type: 'set' | 'delete' | 'insertRow' | 'removeRow' | 'insertColumn' |
'removeColumn';
  sheetId?: string;
  payload: TableCellPayload | Record<string, unknown>;
  tags?: string[];
}
```

内置节点会生成上述命令，适配器负责真正持久化。

TableSnapshot

```
interface TableSnapshot {
  sheetId: string;
  rows: Array<Array<unknown>>;
  columns?: string[];
  metadata?: Record<string, unknown>;
  revision?: string;
  generatedAt: string;
}
```

标签解析（`src/core/tag-parser.ts`）

TagParserPlugin

```
interface TagParserPlugin {
  name: string;
  match(turn: ConversationTurn): boolean;
  parse(turn: ConversationTurn, context: TagParserContext): TagInstruction[] |
Promise<TagInstruction[]>;
}
```

- `match`：是否处理该消息
- `parse`：返回 0+ 条指令

ComposedTagParser

```
const parser = new ComposedTagParser();
parser.register(plugin);
const instructions = await parser.parse({ conversation, variables });
```

- `register`：注册插件（按顺序执行）
- `clear`：清空插件列表

格式化器（`src/core/formatter.ts`）

FormatterRegistry

```
runtime.getFormatters().register({
  name: 'summary',
  factory: ({ snapshot }) => snapshot.rows[0]?.join(',') ?? ''
});

const output = await runtime.getFormatters().format('summary', context);
```

- `register(entry)`：注册格式化器
- `has(name)` / `list()`：检查与列出
- `format(name, context)`：执行指定格式化器

`FormatterContext` 提供 `snapshot`、`instructions`、`variables`、`mode` 等数据。

节点系统

基类接口（`src/nodes/base-node.ts`）

```
interface RuntimeNode {
  readonly type: string;
  getSchema(): NodeSchema;
  validate?(config: Record<string, unknown>): void;
  execute(context: NodeExecutionContext): Promise<NodeResult | void>;
}
```

- `getSchema`：描述节点 label、分类、输入输出、配置结构
- `validate`：配置参数校验（可选）
- `execute`：核心逻辑，返回 `NodeResult`

NodeExecutionContext

```ts
interface NodeExecutionContext {
  graph: GraphDefinition;
  node: NodeInstance;
  flowContext: FlowContext;
  services: NodeServices;
  state: Map<string, unknown>;
}
```

`services` 包含适配器、解析器、格式化器、事件总线以及自定义依赖。

NodeRegistry

```ts
const registry = new NodeRegistry();
registry.register(customNode);
registry.get('ApplyChanges');
registry.list();
registry.clear();
```

通常由 `NodeRuntime` 内部维护。

FlowBuilder（`src/runtime/flow-builder.ts`）

```ts
const flow = FlowBuilder.create('id', 'label', 'description')
  .useNode('LoadTable', { sheetId: 'main' })
  .useNode('ParseTags')
  .useNode('ApplyChanges', { dryRun: true })
  .useNode('FormatPrompt', { formatter: 'prompt' })
  .build();
```

常用方法：

- `useNode(type, config?, options?)`：添加节点
  - `options.id`：自定义节点 ID
  - `options.connectFrom`：指定前驱节点列表（默认连接上一个节点）
  - `options.metadata`：写入元数据（在 `GraphDefinition.metadata` 中可见）
- `link(fromId, toId)`：手动连接两个节点
- `metadata(nodeId, payload)`：补充节点元数据
- `build()`：输出 `GraphDefinition`

NodeRuntime（`src/runtime/node-runtime.ts`）

```
const runtime = new NodeRuntime({
  adapter,
  parser,
  formatters,
  eventBus,
  logger: (msg, extras) => console.debug(msg, extras)
});

runtime.register(customNode);
runtime.registerDefaults();

const { context, state } = await runtime.run(graph, flowContext, {
  services: { log: console.log }
});
```

构造参数

- `adapter`：实现 `TableAdapter` 的实例（必需）
- `parser`：自定义 `TagParser`（默认 `ComposedTagParser`）
- `formatters`：格式化器注册表（默认 `FormatterRegistry`）
- `eventBus`：事件总线（默认 `EventBus`）
- `registry`：节点注册表，默认创建新实例
- `logger`：日志函数
- `services`：额外服务注入（会合并到每个节点的 `NodeServices`）

运行结果

- `context`：运行后的 `FlowContext`（包含变量、最新快照等）
- `state`：Map 结构，记录每个节点的 `outputs`

辅助方法

- `getRegistry()` / `getParser()` / `getFormatters()` / `getEventBus()`
- `register(node)`：注册自定义节点
- `registerDefaults()`：注册内置节点、格式化器和标签解析插件

执行顺序

`NodeRuntime` 会根据节点拓扑排序执行，若图中存在循环或断开的节点会抛出错误。

事件系统（`src/core/event-bus.ts`）

```
const bus = runtime.getEventBus();

bus.on('afterNode', payload => {
  console.log(payload.nodeType, 'done');
});
```

```
await bus.emit('afterNode', { nodeId: 'format-prompt-1', nodeType:
'FormatPrompt' });
```

事件枚举：beforeLoad、afterLoad、beforeNode、afterNode、beforeApply、afterApply、conflict、error。

MemoryWorkflowEngine（src/runtime/memory-workflow-engine.ts）

```
const runtime = new NodeRuntime({ adapter, parser, formatters, eventBus });
const engine = new MemoryWorkflowEngine(runtime);

engine.register('ai:reply', {
  graph: () => FlowBuilder.create('ai-reply', 'AI 回复')
    .useNode('LoadTable')
    .useNode('ParseTags')
    .useNode('ApplyChanges')
    .build()
});

await engine.dispatch({ id: 'evt-1', type: 'ai:reply', conversation });
```

- register(eventType, resolver)：为事件类型注册流程，resolver.graph 可直接传 GraphDefinition 或延迟生成的函数
- unregister(eventType) / has(eventType)：移除或检测流程
- dispatch(event)：执行对应流程，若未匹配会回退尝试 '*' 通配流程

EventFlowResolver

```
interface EventFlowResolver {
  graph: GraphDefinition | (() => GraphDefinition);
  createContext?: (event: MemoryEvent) => FlowContext;
  createRunOptions?: (event: MemoryEvent) => RunOptions | undefined;
  afterRun?: (result: DispatchResult) => void | Promise<void>;
}
```

- createContext：自定义 FlowContext，未提供时会注入 variables.event 保存事件元信息
- createRunOptions：生成传给 runtime.run 的额外配置，如服务注入或调试开关
- afterRun：流程结束后的钩子，可异步执行链路告警、指标上报

MemoryEvent 与 DispatchResult

```ts
interface MemoryEvent {
  id: string;
  type: MemoryEventType;
  timestamp?: string;
  actor?: string;
  sheetId?: string;
  conversation?: ConversationTurn[];
  payload?: Record<string, unknown>;
  services?: Record<string, unknown>;
  context?: Record<string, unknown>;
}

interface DispatchResult {
  event: MemoryEvent;
  flowContext: FlowContext;
  state: Map<string, unknown>;
}
```

`MemoryEventType` 预定义了 `table:init`、`ai:reply`、`user:message`，同时允许扩展任意字符串。`dispatch` 会返回最新的 `FlowContext` 与节点状态 Map，便于调试或用于后续服务。

registerDefaultEventFlows（`src/runtime/default-event-flows.ts`）

```ts
const engine = new MemoryWorkflowEngine(runtime);
registerDefaultEventFlows(engine);
```

该辅助方法注册三类常用事件：

- `table:init`：加载表格并生成首个提示
- `ai:reply`：加载、解析标签、写入更改并重新格式化提示
- `user:message`：在 dry-run 模式下尝试应用指令，生成新的提示

可在此基础上继续使用 `engine.register('*', resolver)` 作为兜底流程。

TableGitAdapter（`src/adapters/table-git-adapter.ts`）

构造函数

```ts
const adapter = new TableGitAdapter({
  tableGit,           // 默认 new TableGit()
  defaultSheetId: 'memory',
  defaultAuthor: 'Bot',
  defaultEmail: 'bot@example.com',
  defaultCommitMessage: cmds => `Apply ${cmds.length} changes`,
  autoInit: true,
  initBranch: 'main',
```

```
  initSheetName: 'memory'
});
```

- 若 `autoInit !== false` 且未传入 `tableGit`，会自动 `init`
- `applyChanges` 默认在非 dry-run 时触发一次 `tableGit.commit`
- 支持 `insertRow/Column`、`removeRow/Column`、`set/delete` 等命令

常见用法

```
await adapter.applyChanges('memory', [
  { type: 'set', payload: { row: 0, column: 0, value: 'Alice' } }
]);

const snapshot = await adapter.snapshot('memory', { includeMetadata: true });
```

扩展策略

注册自定义节点

```
runtime.register({
  type: 'CustomNode',
  getSchema: () => ({ type: 'CustomNode', label: '自定义节点' }),
  async execute({ flowContext }) {
    flowContext.variables = {
      ...(flowContext.variables ?? {}),
      custom: 'value'
    };
  }
});
```

覆盖内置节点

```
runtime.getRegistry().replace(myEnhancedApplyChangesNode);
```

扩展服务

```
await runtime.run(flow, context, {
  services: {
    metrics: (name: string, value: number) => report(name, value)
  }
});
```

节点内可通过 `services.metrics?.('apply', commands.length)` 使用。

异常处理

- 运行时若未提供 `TableAdapter` 会抛出 `TableAdapter is required` 错误
- 图中存在环或断开节点时，将抛出 `Graph contains a cycle or disconnected node`
- 解析器未注册格式化器时，`FormatPrompt` 会抛出 `Formatter '<name>' is not registered`

建议：

- 在 `runtime.run` 外层包裹 try/catch 并订阅 `error` 事件
- 在 `ApplyChanges` 前可添加自定义节点执行数据校验

版本兼容

当前包处于 `0.x` 阶段，API 仍可能调整。建议通过 `registerDefaults` 获得内置能力，并在自定义扩展时关注变更日志。