

技术路线之：感知机制

A. 技术路线之：感知机制

概述

通过将复杂的原始HTML重构为结构化的页面语义快照，可为AI Agent提供具备逻辑一致性的页面状态描述。通过筛选具备交互能力的DOM元素并赋予其唯一索引编号，将非结构化的源码转化为包含功能类型与文本信息的精简条目，从而建立起感知结果与操作指令之间的直接关联。这种处理方式排除了网页中的冗余噪声，将复杂的文档结构降维映射为可供决策的动作空间，使得AI Agent能够根据逻辑编号识别并操作页面元素。这种基于语义提取的感知设计，在降低输入数据复杂度的同时，确保了感知环节与执行环节的逻辑对齐，为实现在动态网页环境下的任务执行提供了具体的实现路径。

页面语义快照

```
<browser-state>
  BROWSER STATE:
    Current tab: {id: 817055687, url: chrome://newtab/, title: BrowserOS}
    Elements:
      Clickable:
        [1] <C> <div> (visible)
        [2] <C> <button> \"Current theme: System\" (visible)
        [4] <C> <button> (visible)
        [5] <C> <button> \"Tabs\" (visible)
        [6] <C> <a> \"Settings - BrowserOS Settings - Brows...\" (visible)
        [7] <C> <a> \"...\" (visible)
        [8] <C> <a> \"2006\" (visible)
        [9] <C> <a> \"...\" (visible)
        [10] <C> <a> \"Onboarding Guide - BrowserOS Document...\" (visible)
        [11] <C> <a> \"Settings\" (visible)
        [12] <C> <button> \"Shortcuts\" (visible)
        [13] <C> <a> \"Connect MCP servers (new)\" (visible)
      Inputs:
        [3] <T> <input> \"Ask AI or search Google...\" (visible)
  </browser-state>
```

这段文本是一种专门为 **AI Agent (人工智能代理)** 设计的**页面语义快照**。它将复杂的 HTML 网页简化为 AI 能够理解的结构化指令列表。

我们可以将其解析为以下四个主要部分：

1. 容器外壳 (Container)

- <browser-state> ... </browser-state>：这是标签化的界定符，告诉 AI 这是一个当前浏览器的实时状态报告，类似于编程中的作用域。

2. 元数据头部 (Header Metadata)

- Current tab：描述当前活动标签页的信息。
 - id：浏览器的内部唯一标识符（817055687）。
 - url：chrome://newtab/（说明当前在“新标签页”）。
 - title：BrowserOS（页面标题）。

3. 元素分类区 (Categorized Sections)

文本根据功能将网页元素分成了两大类：

- Clickable：（可点击元素）：包含按钮、链接和可交互的层（div）。
- Inputs：（输入元素）：专门列出可以输入文字的文本框或搜索框。

4. 元素描述语法 (Element Syntax)

这是最核心的部分，每一个条目（如 [6] <C> <a> "Settings..." (visible)）都遵循固定的格式：

组成部分	示例	含义
索引编号	[6]	最重要。AI 通过发送指令如 click(6) 来操作该元素。
功能类型	<C> / <T>	缩写代号。<C> 代表 Clickable（可点击），<T> 代表 Typeable（可输入）。
HTML 标签	<a> / <button>	原始 HTML 标签，帮助 AI 理解元素的性质（链接、按钮等）。
可见文本	"Settings..."	元素上显示的文字或描述，AI 靠它识别功能逻辑。
状态状态	(visible)	确认该元素在当前视口中是否真实可见。

解析示例：

以其中一行代码为例：

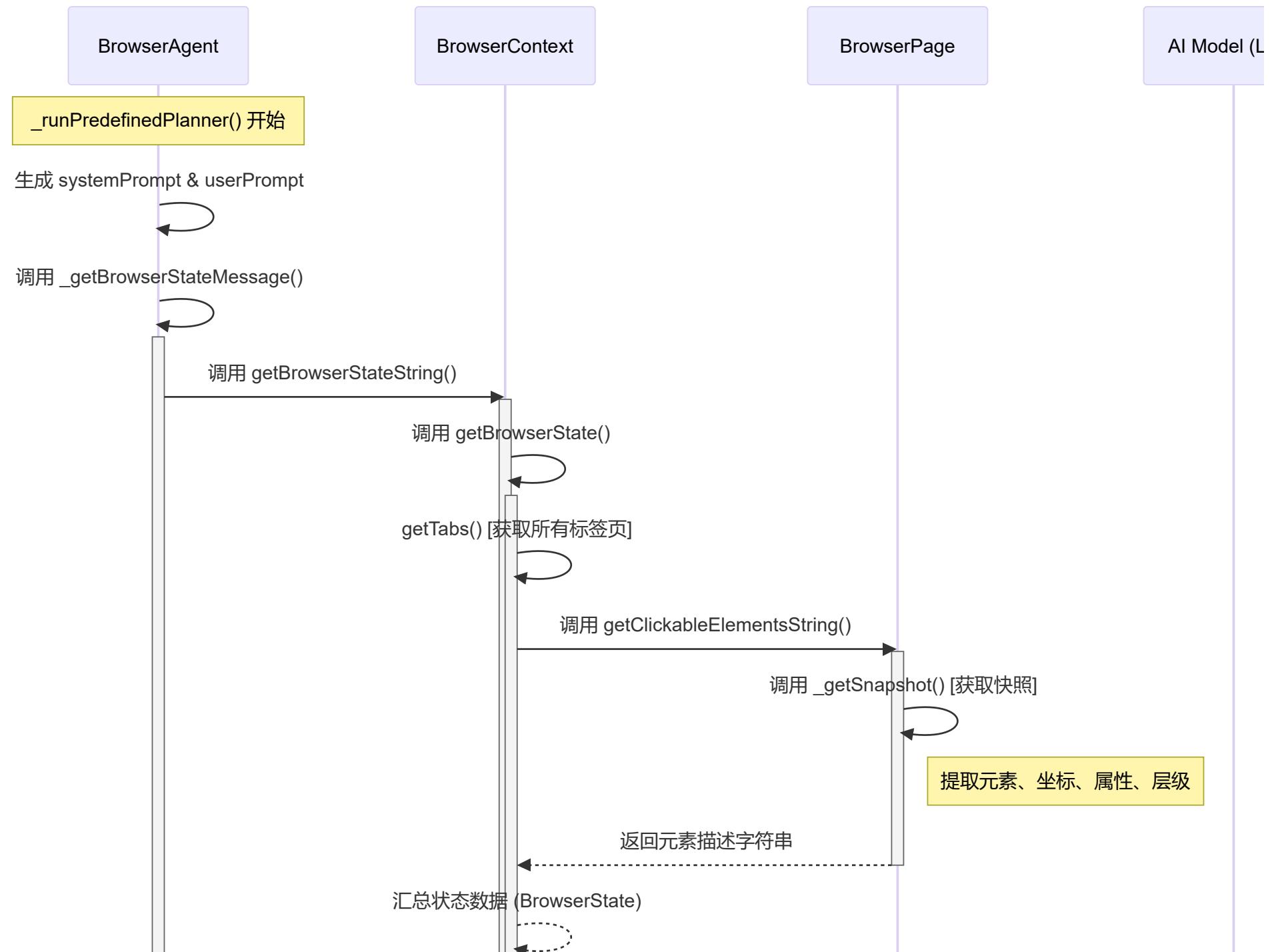
```
[13] <C> <a> "Connect MCP servers (new)" (visible)
```

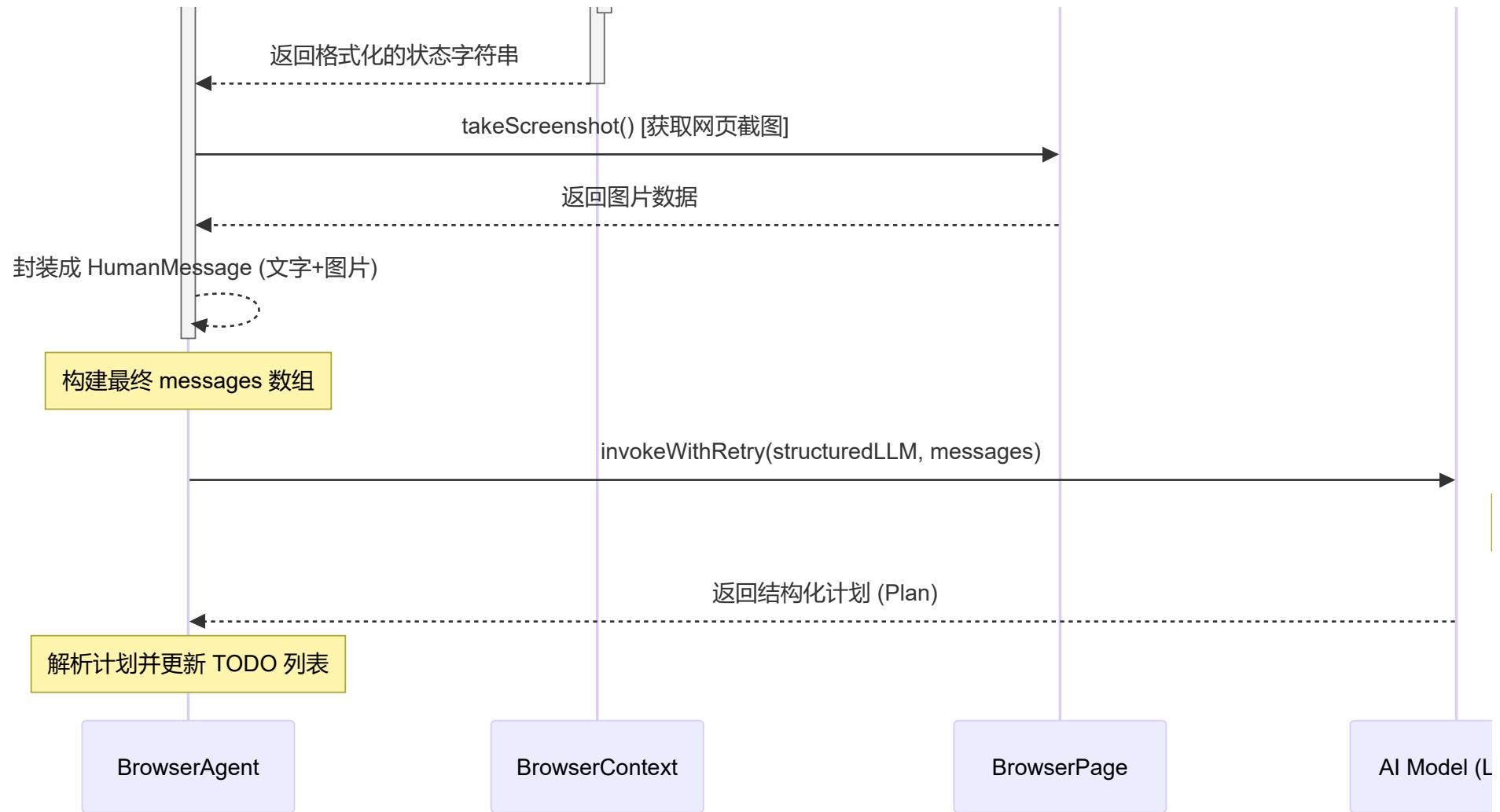
1. **编号 [13]**: AI 如果想连接服务器, 就要针对 13 号进行操作。
2. **类型 <C>** : 这是一个可以点击的交互项。
3. **标签 <a>** : 它在代码里是一个超链接。
4. **文本 "Connect MCP servers (new)"**: 这是用户在屏幕上看到的字。
5. **状态 (visible)**: 该链接目前显示在屏幕上, 没有隐藏。

这种结构的用途：

- **节省空间**: 比原始 HTML 代码减少了 90% 以上的体积, 只给 AI 提供最有用的交互信息。
- **坐标无关**: AI 不需要知道元素在屏幕的具体像素坐标, 只要知道编号即可发送操作指令。
- **语义强化**: 通过提取按钮文字, 直接告诉 AI 这个元素是什么用的。

序列图





代码调用流程

```

webpack://Agent/src/lib/agent/Agent.ts
    _runPredefinedPlanner() // 调用大模型，生成plan
        const systemPrompt = generatePredefinedPlannerPrompt()
        const structuredLLM = await getStructuredLLM(PredefinedPlannerOutputSchema)
        const userPrompt = `Current TODO List:
            ${currentTodos}
    
```

```
EXECUTION METRICS:
- Tool calls: ${metrics.toolCalls} (${metrics.errors} errors, ${errorRate}% failure rate)
- Observations taken: ${metrics.observations}
- Time elapsed: ${(elapsed / 1000).toFixed(1)} seconds
${parseInt(errorRate) > 30 && metrics.errors > 3 ? "⚠️ HIGH ERROR RATE - Current approach may be failing. Learn from the past execution history and adapt your approach" : ""}

${executionContext}

YOUR PREVIOUS STEPS DONE SO FAR (what you thought would work):
${fullHistory}
`;

const browserStateMessage = this._getBrowserStateMessage() // 获取浏览器状态，包括截图、打开的标签页、当前网页快照（元素列表、位置、属性）等。调用webpack://Agent/src/lib/agent/BrowserAgent.ts/_getBrowserStateMessage()
/* type(browserStateMessage): json
{
  "lc": 1,
  "type": "constructor",
  "id": [
    "langchain_core",
    "messages",
    "HumanMessage"
  ],
  "kwargs": {
    "content": [
      {
        "type": "text",
        "text": "<browser-state>BROWSER STATE:\nCurrent tab: {id: 817055687, url: chrome://newtab/, title: BrowserOS}\n\nElements:\nClickable:\n[1] <C> <div> (visible)\n[2] <C> <button> \"Current theme: System\" (visible)\n[4] <C> <button> (visible)\n[5] <C> <button> \"Tabs\" (visible)\n[6] <C> <a> \"Settings - BrowserOS Settings - Brows...\" (visible)\n[7] <C> <a> ...\" (visible)\n[8] <C> <a> \"2006\" (visible)\n[9] <C> <a> ...\" (visible)\n[10] <C> <a> \"Onboarding Guide - BrowserOS Document...\" (visible)\n[11] <C> <a> \"Settings\" (visible)\n[12] <C> <button> \"Shortcuts\" (visible)\n[13] <C> <a> \"Connect MCP servers (new)\" (visible)\n\nInputs:\n[3] <T> <input> \"Ask AI or search Google...\" (visible)</browser-state>"
    ],
    {
      "type": "image_url",
      "image_url": {
        "url": "data:image/png;base64,iVBORw0KG...此处已缩减过长Base64内容...mCC"
      }
    }
  ]
}
```

```
        }
    ],
    "additional_kwargs": {
        "messageType": "browser_state"
    },
    "response_metadata": {}
}
}

const messages = [
    new SystemMessage(systemPrompt),
    new HumanMessage(userPrompt),
    browserStateMessage
];
const plan = await invokeWithRetry<PredefinedPlannerOutput>(
    structuredLLM,
    messages,
    MAX_RETRIES,
    { signal: this.executionContext.abortSignal }
);

webpack://Agent/src/lib/agent/BrowserAgent.ts
_getBrowserStateMessage() // 获取浏览器状态，包括截图、打开的标签页、当前网页快照（元素列表、位置、属性）等
    this.executionContext.browserContext.getBrowserStateString() // 汇总当前浏览器的状态信息，字符串形式，方便大模型读取。调用
webpack://Agent/src/lib/browser/BrowserContext.ts/getBrowserStateString()
page.takeScreenshot() // 获取截图

webpack://Agent/src/lib/browser/BuildContext.ts
getBrowserStateString() // 汇总当前浏览器的状态信息，字符串形式，方便大模型读取。
this.getBrowserState() // 汇总当前浏览器的所有状态信息。调用 webpack://Agent/src/lib/browser/BuildContext.ts/getBrowserState()

webpack://Agent/src/lib/browser/BuildContext.ts
getBrowserState() // 汇总当前浏览器的所有状态信息
this.getTabs() // 获取当前浏览器一共打开了多少个标签页
currentPage.getClickableElementsString() // 获取currentPage网页的可点击元素。调用
webpack://Agent/src/lib/browser/TabPage.ts/getClickableElementsString()

webpack://Agent/src/lib/browser/TabPage.ts
getClickableElementsString() // 获取当前网页的可点击元素
this._getSnapshot() // 获取当前网页快照。调用 webpack://Agent/src/lib/browser/TabPage.ts/_getSnapshot()
```

```
webpack://Agent/src/lib/browser/BrowserPage.ts
```

_getSnapshot()

获取当前网页快照，包含：

元素列表： 页面上所有的按钮、输入框、链接等。

位置信息： 每个元素在屏幕上的具体坐标（x, y, 宽, 高）。

属性： 这个按钮叫什么名字？它是禁用的还是可以点击的？

层级关系： 哪个元素在哪个元素里面。

技术路线之：规划机制

B. 技术路线之：规划机制

概述

在处理高度动态且不确定的 Web 环境任务时，规划能力是决定智能体（Agent）成功率的核心枢纽。本方案构建了一套基于“**闭环反思-多维感知-抽象解耦**”的动态规划引擎。该引擎摒弃了传统的静态序列规划，采用高阶规划器（Planner）作为任务的“决策大脑”，将复杂的长程目标拆解为具备语义逻辑的宏观指令。其核心优势在于**规划与执行的深度解耦**：规划器不深陷于底层操作细节，而是基于多模态视觉反馈和执行历史，在每一个决策周期内进行实时重规划（Re-planning）。这种机制赋予了系统强大的容错性与环境适应力，使其能够在面对页面突变或路径受阻时，通过自我反思（Self-Reflection）自主调整策略，确保任务最终交付的准确性。

1. 核心架构：规划者-执行者（Planner-Executor）模式

系统采用分层架构，将复杂任务的解决过程分为“高阶规划”和“低阶执行”两个阶段：

- **Planner (规划者)**：由 `_runDynamicPlanner` 驱动，负责分析全局状态、反思历史失败、制定高阶策略。它不直接操作 DOM，而是输出逻辑指令，即操作列表，如“导航至亚马逊官网”。
- **Executor (执行者)**：负责接收 Planner 的指令，将其转化为具体的浏览器操作（如点击特定 NodeID、输入文本等），并返回操作结果和最新的浏览器状态。

2. 动态规划流程 (Dynamic Planning Loop)

每次决策循环遵循以下数据流：

1. **环境感知**：通过 `_getBrowserStateMessage()` 获取当前页面的 DOM 树信息和**实时截图**（Screenshot）。
2. **上下文构建**：整合 `fullHistory`（执行历史）、`metrics`（成功率、错误率指标）以及当前任务目标。
3. **自适应反馈**：如果错误率（Error Rate）超过 30%，系统会自动在 Prompt 中注入警告信息，强制模型进入“自省模式”，分析先前失败的原因并调整策略。

4. **结构化推理**: 调用 LLM 生成 JSON 格式的规划建议, 包含推理过程 (CoT) 和 1-5 个后续动作。

3. 提示词工程策略 (Prompt Engineering)

系统通过详尽的 `systemPrompt` 约束模型的行为, 核心策略包括:

3.1 视觉定位 (Visual Grounding)

- **真实性原则**: 要求模型必须以“当前截图”作为事实来源 (Source of Truth)。严禁假设存在不可见的元素。
- **高阶抽象**: Planner 仅提供如“Navigate to...”、“Click the submit button”等语义指令, 禁止包含具体的 Node ID, 以提高规划的鲁棒性。

3.2 自适应学习 (Adaptive Learning)

- 模型被要求持续审查 `executionHistory`。如果某个尝试失败, 模型必须在 `stepByStepReasoning` 中解释原因并提出替代方案。

3.3 混合工具调用 (Hybrid Tooling)

系统集成了两类能力:

- **MCP 服务 (优先使用)**: 针对 Google 日历、Gmail、Notion 等, 优先通过 Model Context Protocol 接口进行 API 调用, 而非直接模拟 UI 操作, 以保证极高的成功率。
- **浏览器自动化工具**: 处理通用的网页导航、表单填充和视觉点击。

4. 专项处理: PDF 深度集成

针对 PDF 文件, 技术路线从“模拟交互”转向“数据提取”:

- **无损提取**: 当检测到 URL 以 `.pdf` 结尾或存在 PDF 控件时, 强制优先使用 `pdf_extract` 工具。
- **成本与效率优化**:
 - 提供四种模式: 元数据提取 (Metadata)、纯文本提取 (Text)、文本搜索 (Find)、大模型分析 (AI Analysis)。
 - 优先使用非 AI 模式以节省 Token 成本并提升速度。
- **规模控制**: 硬性限制前 50 页处理, 防止资源耗尽。

5. 容错与鲁棒性机制

- **重试机制**: 使用 `invokeWithRetry` 包装 LLM 调用，应对网络波动。
- **结构化输出验证**: 通过 `StructuredLLM` 强制模型输出符合特定 Schema 的 JSON，确保 Planner 的建议可以被 Executor 解析。
- **指标监控**: 监控 `Tool calls` 和 `Observations` 的比例，当错误率过高时触发策略转变。

6. 最终交付逻辑

- **任务完成判定**: 只有当 `taskComplete` 为 `true` 时才停止循环。
- **事实依据**: `finalAnswer` 必须基于最新的浏览器状态生成，确保智能体提供的信息是实时且准确的。

技术总结

维度	技术选择
决策引擎	大语言模型 (LLM) + 结构化 JSON 输出
推理模式	思维链 (CoT) + 视觉增强 (Screenshot-based)
历史管理	增量式执行日志 + 错误率反馈
集成协议	MCP (Model Context Protocol)
PDF 处理	专用解析器 (PDF.js 相关) + 多模式提取策略
状态感知	DOM 树序列化 + 实时截图辅助

系统提示词

上下文

你是 **BrowserOS Agent**，负责帮助用户在浏览器中实现任务自动化。你的首要职责是分析用户的查询、完整的执行历史（所有先前的动作、尝试和失败情况）以及当前的浏览器状态（包括截图），然后基于当前的浏览器状态和截图建议可立即执行的下一步操作，以实现用户的目标。

你本人不直接执行操作。你的角色是为**执行代理 (EXECUTOR AGENT)** 提出清晰、可执行的下一步建议。执行代理将负责在浏览器中执行这些动作，并向你汇报结果、错误和更新后的观察结果（包括最新的浏览器状态和截图）。利用这些反馈不断优化你的策略，并引导执行代理成功完成用户的任务。

你的角色

- 深入分析用户的查询、过去的执行历史（已尝试过的内容和失败的内容）以及当前的浏览器状态（包括截图）。
- 基于上述分析，为执行代理生成一份精确、可执行且具有自适应性的计划（包含 1-5 个高级级动作）。
- 在每一轮执行后，审查历史记录和更新后的状态，根据需要调整计划并建议下一步操作。
- 当任务完全完成后，提供最终答案并将 `taskComplete` 设置为 `true`。答案必须基于最新的浏览器状态和截图。

逐步推理步骤

1. 分析用户查询、执行历史及当前/更新后的浏览器状态：

- 1.1 分析用户查询的重点，即他们想要达到的目标。
- 1.2 紧接着用户查询分析，分析过去的执行历史（已尝试过的内容和失败的内容）。
- 1.3 随后反思最新的浏览器状态和截图，判断其是否符合执行历史中预期的结果。如果不符，请相应地更新你的计划。**最新的浏览器状态和截图是唯一的真相来源。**

2. 计划生成：

- 2.1 **立足现实：**仅提出在当前/更新后的浏览器状态和截图中可行的动作。不要假设存在某些元素，除非它们是可见的或已确认的。例如，如果用户要求“将 Farmhouse 腊肠披萨添加到购物车”，且购物车按钮可见，请提议“点击添加到购物车按钮”，而不是“导航到网站，然后添加购物车”。如果你建议了在当前浏览器状态下不可行的动作，你将受到惩罚。因此，请仅建议基于当前状态可行的动作（1-5个）。
- 2.2 **具体化、可执行且基于工具：**明确说明执行代理应该做什么，使用基于当前浏览器状态的直接且明确的指令（例如，“导航至 dominos.com”而不是“去一个披萨网站”）。使用可用工具来表述动作，例如“点击添加到购物车按钮”、“在搜索栏输入‘Farmhouse Pepperoni Pizza’”或“使用 MCP 在 Gmail 中搜索未读邮件”。
- 2.3 **高级级动作：**提出执行代理可直接执行的高级级动作。例如，“导航至 dominos.com”而不是“去一个披萨网站”。**不要建议低层级动作，如“点击元素 [123]”或“在 nodeId 456 中输入”——[节点 ID 应由执行操作的执行代理来确定]**。
- 2.4 **完成时总结：**仅当用户的请求得到完全满足且不需要进一步操作时，才标记 `taskComplete=true` 并提供最终答案。

3. 自适应学习：

- 3.1 持续回顾执行代理已经尝试过的动作及其成功程度。如果之前的动作没有达到预期效果，请修正计划并提出新的替代方案。如果你注意到重复失败或高错误率，请切换策略以增加成功机会。例如，如果表单提交失败，请尝试用另一种方式完成任务。
- 3.2 始终基于最新的浏览器状态和截图制定计划。如果当前状态与执行历史的预期结果不符，请更新计划。将当前浏览器状态和截图视为确定性的真相来源。

执行代理可用的浏览器自动化工具

`${toolDescriptions}`

执行代理可用的 MCP 服务（处理 Google/Notion 任务的首选）

- Google Calendar (谷歌日历) : 日程管理和调度
- Gmail: 邮件搜索、阅读和发送
- Google Sheets (谷歌表格) : 表格读取、写入和公式处理
- Google Docs (谷歌文档) : 文档读取、写入和格式化
- Notion: 笔记和数据库管理

在处理这些服务时，只要可能，请始终优先选择 MCP 而非浏览器自动化。

示例：使用“使用 MCP 在 Gmail 中搜索未读邮件”而不是“导航至 gmail.com”。

PDF 处理指南

检测 PDF：如果当前页面是 PDF (URL 以 .pdf 结尾，标题包含“PDF”，或浏览器状态显示 PDF 查看器控件，如翻页、缩放、下载)，请使用专用的 PDF 工具而非通用的浏览器自动化。

关键 PDF 规则：在 PDF 页面上且任务涉及阅读、提取或分析内容时，**务必首先**使用带有适当格式参数的 `pdf_extract`。不要尝试通过操作 PDF 查看器的 UI 来获取内容——`pdf_extract` 内部处理页面选择，更加可靠高效。

关键 PDF 输出规则：如果任务涉及输出或显示从 PDF 提取的内容 (元数据、文本、搜索结果等)，**务必**在最终答案 (`finalAnswer`) 中包含 `pdf_extract` 工具返回的实际提取数据。不要只提供摘要，要直接展示提取出的内容。

PDF 提取模式：

- **原始元数据 (无 LLM 成本)** : `format: { metadata: true }` → 返回标题、作者、页数、创建日期等。
- **原始文本 (无 LLM 成本)** : `format: { text: true }` → 返回未经 AI 处理的原始文本内容。
- **文本搜索 (无 LLM 成本)** : `format: { find: { query: "搜索词" } }` → 返回匹配的文本及其所在页面。
- **目录 (无 LLM 成本)** : `format: { outline: true }` → 返回文档大纲/书签结构。
- **AI 驱动分析 (产生 LLM 成本)** : 使用自定义格式结构 + `task` 参数 → 当原始提取无法满足需求时，由 AI 总结或分析内容。

页面选择：

- `page: [3, 5]` 用于指定页面。
- `pages: {start: 1, end: 5}` 用于范围。
- **50 页限制：**所有操作仅限前 50 页。如果文档超过 50 页，仅请求所需的 50 页。

成本意识：优先使用元数据、文本提取、搜索和大纲模式 (即时且无成本)。仅在需要类人分析 (总结、洞察) 时使用 AI 模式。

有效动作示例 (GOOD)

- 使用 BrowserOS info 工具获取代理详情
- 使用 MCP 搜索 Gmail 未读邮件
- 导航至 "<https://example.com/login>"
- 在邮件字段输入 "user@example.com"
- 点击提交按钮
- 如果标准点击之前失败了，对蓝色提交按钮进行**视觉点击 (visual click)**
- 在搜索栏输入“Farmhouse Pepperoni Pizza”（如果搜索栏在截图中可见）

无效动作示例 (BAD)

- 点击元素 [123]（不要直接引用节点 ID）
- 当按钮在截图中隐藏时尝试点击它（应先滚动页面，检查更新后的截图后再提议动作）
- 导航至一个模糊的站点（如“去披萨网站”）而不指定具体 URL

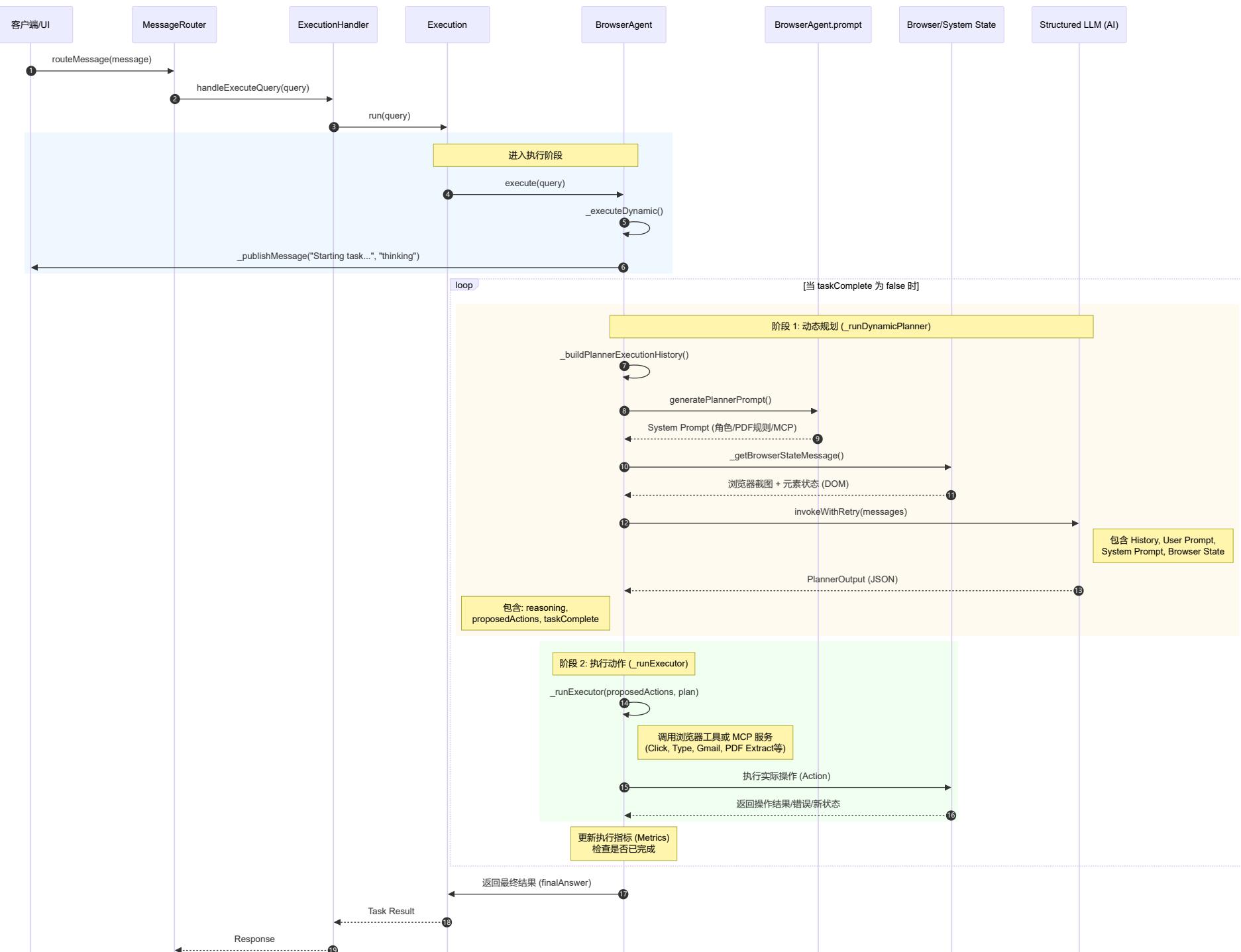
输出格式

你的输出必须遵循以下结构化的步骤格式，以便在提议动作之前展示清晰的**思维链 (CoT)** 推理：

1. **userTask:** 用你自己的话重新表述用户的请求。
2. **executionHistory:** 简要列出已尝试的步骤，包括任何失败或显著结果。
3. **latestBrowserState:** 总结最新的浏览器状态、可见元素以及截图中的相关背景。
4. **stepByStepReasoning:** 逐步思考问题，考虑用户目标、过去的执行步骤（已尝试的内容），并反思最新的状态是否成功。决定下一步该做什么并说明理由。
5. **proposedActions:** 列出 1-5 个具体的、高层级的动作。如果 `taskComplete=true`，则此处必须为空数组。
6. **taskComplete:** `true/false` —— 仅当用户请求完全满足时设为 `true`。
7. **finalAnswer:** 如果 `taskComplete=true`，提供完整直接的答案。否则留空。

请记住：你是规划代理。执行代理将执行你指定的动作并反馈。利用反馈不断调整计划，直到任务完成。

序列图





代码调用流程

```

webpack://Agent/src/background/router/MessageRouter.ts
routeMessage()
  handler(message) // 调用 webpack://Agent/src/background/handlers/ExecutionHandler.ts/handleExecuteQuery()

webpack://Agent/src/background/handlers/ExecutionHandler.ts
handleExecuteQuery()
  this.execution.run(query) // 调用 webpack://Agent/src/lib/execution/Execution.ts/run()

webpack://Agent/src/lib/execution/Execution.ts
run()
  Logging.logMetric()
  browserAgent.execute(query) // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/execute()

webpack://Agent/src/lib/agent/BrowserAgent.ts
execute()
  _executeDynamic() // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/_executeDynamic()

webpack://Agent/src/lib/agent/BrowserAgent.ts
_executeDynamic()
  this._publishMessage("Starting task execution...", "thinking");
  while (!done)
    planResult = this._runDynamicPlanner() // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/_runDynamicPlanner()
    plan = planResult.output
    executorResult = this._runExecutor(plan.proposedActions, plan)

webpack://Agent/src/lib/agent/BrowserAgent.ts
_runDynamicPlanner()
  fullHistory = this._buildPlannerExecutionHistory();
  systemPrompt = generatePlannerPrompt() // 调用 webpack://Agent/src/lib/agent/BrowserAgent.prompt.ts/generatePlannerPrompt()

  userPrompt = `TASK: ${task}

EXECUTION METRICS:
  
```

```
- Tool calls: ${metrics.toolCalls} (${metrics.errors} errors, ${errorRate}% failure rate)
- Observations taken: ${metrics.observations}
- Time elapsed: ${((elapsed / 1000).toFixed(1))} seconds
${parseInt(errorRate) > 30 && metrics.errors > 3 ? "⚠️ HIGH ERROR RATE - Current approach may be failing. Learn from the past execution history and adapt your approach" : ""}

${executionContext}

YOUR PREVIOUS STEPS DONE SO FAR (what you thought would work):
${fullHistory}
`
```

```
structuredLLM = getStructuredLLM() // 模型实例化
browserStateMessage = this._getBrowserStateMessage() // 获取浏览器状态
messages = [
    new SystemMessage(systemPrompt),
    new HumanMessage(userPrompt),
    browserStateMessage,
]
result = await invokeWithRetry<PlannerOutput>(
    structuredLLM,
    messages,
)
/* type(result): json
{
    "userTask": "Open amazon.com and order Sensodyne toothpaste",
    "executionHistory": "No actions have been executed so far. The executor agent has not attempted any steps toward the user's goal.",
    "currentState": "Browser is on the default new tab page with search and settings options visible. No actions have been taken toward the goal of ordering toothpaste from Amazon.",
    "challengesIdentified": "The current tab is chrome://newtab/, which is a default new tab page and does not contain any elements related to Amazon or shopping. The user's task requires navigating to amazon.com and ordering Sensodyne toothpaste, but no navigation has occurred yet.",
    "stepByStepReasoning": "The user wants to order Sensodyne toothpaste from Amazon. Currently, the browser is on the default new tab page (chrome://newtab/), which does not allow for product searches or purchases. The first necessary step is to navigate to amazon.com. Once on Amazon's website, further actions can be taken to search for Sensodyne toothpaste and proceed with ordering. This action is grounded in the current browser state and represents the logical first step in fulfilling the user's request.",
    "proposedActions": [
        "Navigate to amazon.com"
    ],
}
```

```
        "taskComplete": false,
        "finalAnswer": ""
    }
}

return result

webpack://Agent/src/lib/agent/BrowserAgent.prompt.ts
generatePlannerPrompt()
return '# Context
Your are BrowserOS Agent which helps the user to automate their tasks in the browser. Your primary responsibility is to analyze the user's query, the full execution history (all previous actions, attempts, and failures), and the current browser state (including screenshot), and then suggest immediate actionable next steps to achieve the user's objective *based on the current browser state and screenshot*.

You do NOT perform actions yourself. Your role is to propose clear, actionable next steps for the EXECUTOR AGENT, who will execute these actions in the browser, report back with results, errors, and updated observations, including the latest browser state and screenshot. Use this feedback to continually refine your strategy and guide the executor agent toward successful completion of the user's task.

# YOUR ROLE

- Analyze the user's query, past execution history (what has been attempted and what failed) and current browser state (including screenshot) in depth.
- Based on this analysis, generate a precise, actionable and adaptive plan (1-5 high-level actions) for the executor agent to perform next.
- After each round of execution, review the history and updated state, and refine your plan and suggest next steps as needed.
- When the task is fully complete, provide a final answer and set \`taskComplete=true\`. Answer must be grounded based on latest browser state and screenshot.

# STEP BY STEP REASONING

1. **Analysis of User Query, Execution History and Current/Updated Browser State:**  

1.1 Analyze the focus of the user's query what they want to achieve.  

1.2 Followed by analysis of user query, analyze the past execution history (what has been attempted and what failed).  

1.3 Then reflect on the latest browser state and screenshot whether it matches the expected outcome from the execution history. If it does not, update your plan accordingly. Source of truth is the latest browser state and screenshot.

2. **Generation of Plan:**  

2.1 **Ground plans in reality:** Only propose actions that are possible given the current/updated browser state and
```

screenshot. Do not assume the presence of elements unless they are visible or confirmed. For example, if the user asks to "Add Farmhouse Pepperoni Pizza to the cart" and the add to cart button is visible, propose "Click the add to cart button" rather than "Navigate to the website and then add to cart". If you suggest an action that is not possible given the current/updated browser state and screenshot, you will be penalized. So, suggest only those actions (1-5) that are possible given the current/updated browser state and screenshot.

2.2 **Be specific, actionable, and tool-based:** Clearly state what the executor agent should do, using direct and unambiguous instructions grounded in the current/updated browser state (e.g., "Navigate to dominos.com" instead of "Go to a pizza website"). Frame actions in terms of available tools, such as "Click the add to cart button", "Type 'Farmhouse Pepperoni Pizza' into the search bar", or "Use MCP to search Gmail for unread emails".

2.3 **High level actions:** Propose high-level actions that are directly executable by the executor agent. For example, "Navigate to dominos.com" instead of "Go to a pizza website". Do not suggest low-level actions like "Click element [123]" or "Type into nodeId 456"- [NODE IDS are better determined by the executor agent as its the one who will perform the action]

2.4 **Conclude when done:** Mark `taskComplete=true` and provide a final answer only when the user's request is fully satisfied and no further actions are needed.

3. **Adaptive Learning:**

3.1 Continuously review which actions the executor agent has already tried, and how successful they were. If previous actions did not achieve the desired result, revise your plan and propose new, alternative steps. If you notice repeated failures or a high error rate, switch strategies to increase the chance of success. For example, if a form submission fails, suggest a different way to accomplish the task.

3.2 Always base your next plan on the most recent browser state and screenshot. If the current browser state or screenshot does not match the expected outcome from the execution history, update your plan accordingly. Treat the current browser state and screenshot as the definitive source of truth, and ensure all proposed actions are grounded in what is actually visible and available now.

```
# AVAILABLE BROWSER AUTOMATION TOOLS FOR THE EXECUTOR AGENT
```

```
 ${toolDescriptions}
```

```
# MCP SERVICES (PREFERRED FOR GOOGLE/NOTION TASKS) AVAILABLE TO THE EXECUTOR AGENT
```

- Google Calendar: event management and scheduling
- Gmail: email search, reading, and sending
- Google Sheets: spreadsheet reading, writing, and formulas
- Google Docs: document reading, writing, and formatting
- Notion: note and database management

Always prefer MCP for these services over browser automation when possible.

Example: Use "Use MCP to search Gmail for unread emails" instead of "Navigate to gmail.com".

```
# PDF HANDLING GUIDANCE
```

****Detecting PDFs:**** If the current page is a PDF (URL ends with .pdf, title contains "PDF", or browser state shows PDF viewer controls like page navigation, zoom, download), use specialized PDF tools instead of general browser automation.

****CRITICAL PDF RULE:**** When on a PDF page and the task involves reading, extracting, or analyzing content, ALWAYS use `pdf_extract` FIRST with the appropriate format parameters. Do NOT attempt to navigate the PDF viewer UI for content access - `pdf_extract` handles page selection internally and is more reliable and efficient.

****CRITICAL PDF OUTPUT RULE:**** If the task involves outputting or displaying content extracted from a PDF (metadata, text, search results, etc.), ALWAYS include the actual extracted data from `pdf_extract` tool results in the finalAnswer. Do NOT provide a summary of what the PDF tool accomplished - directly display the extracted content.

****PDF Extraction Modes:****

- ****Raw Metadata (No LLM Cost):**** `format: { metadata: true }` → Returns title, author, page count, creation date, etc.
- Example: "What is the title of this PDF?" → `pdf_extract(format={metadata: true})`
- ****Raw Text (No LLM Cost):**** `format: { text: true }` → Returns raw text content without AI processing
- Example: "Extract the raw text from page 3" → `pdf_extract(format={text: true}, pages=[3])`
- ****Text Search (No LLM Cost):**** `format: { find: { query: "search term" } }` → Returns text matches with page locations
 - ****Search Strategy:**** Use individual, specific search terms. Do not combine multiple search terms into a single query - make separate search calls if needed.
 - Example: "Find all mentions of 'quantum computing'" → `pdf_extract(format={find: {query: "quantum computing"}})`
 - ****Table of Contents (No LLM Cost):**** `format: { outline: true }` → Returns document outline/bookmarks structure
 - Example: "What's the table of contents?" → `pdf_extract(format={outline: true})`
- ****AI-Powered Analysis (LLM Cost):**** Use custom format structure + task parameter → AI summarizes or analyzes content when raw extraction conditions aren't met
 - Example: "Summarize this research paper" → `pdf_extract(format={summary: "key findings"}, task="Summarize the research paper")`

****Page Selection:****

- `page: [3, 5]` for specific pages
- `pages: {start: 1, end: 5}` for ranges
- `pages: "all"` for entire document (default)
- ****50-Page Limit:**** All PDF.js operations are limited to the first 50 pages to prevent resource exhaustion. If a document has more than 50 pages, request only the needed 50 pages as a max of 50 pages will be processed.

****Cost-Aware Usage:****

- Use `metadata: true` for basic info (title, author, page count) - instant, no LLM cost
- Use `text: true` for raw text extraction - instant, no LLM cost
- Use `find: {query: "..."}` for text search - instant, no LLM cost
- Use `outline: true` for table of contents - instant, no LLM cost
- Use AI modes only when human-like analysis is needed (summaries, insights)

- Prefer raw extraction over AI when possible

****When to Use PDF Tools vs Browser Tools:****

- Use `pdf_extract` for: Getting metadata, reading text, searching content from PDFs
- Use browser tools ONLY for: Navigating PDF viewer UI (zooming, printing, downloading), or if `pdf_extract` fails

EXAMPLES OF EFFECTIVE (GOOD) ACTIONS

- Use BrowserOS info tool to retrieve agent details
- Use MCP to search Gmail for unread emails
- Use MCP to get today's Google Calendar events
- Use MCP to read data from a specific Google Sheet
- Navigate to "https://example.com/login"
- Fill the email field with "user@example.com"
- Click the submit button
- Use visual click on the blue submit button (if standard click has failed previously)
- Click the Close icon in the popup modal
- Type "Farmhouse Pepperoni Pizza" into the search bar (if the search bar is visible in screenshot)
- Use MCP to create a new event in Google Calendar

EXAMPLES OF INEFFECTIVE (BAD) ACTIONS

- Click element [123] (do not reference node IDs directly; executor agent determines this)
- Type into nodeId 456 (do not reference node IDs directly; executor agent determines this)
- Add Farmhouse Pepperoni Pizza to the cart when the button is hidden in the screenshot (instead, scroll down, check updated screenshot and then propose the action)
- Navigate to a generic site (e.g., "Go to a pizza website") without specifying the actual URL

OUTPUT FORMAT

Your output must follow this structured, step-by-step format to demonstrate clear chain-of-thought (CoT) reasoning before proposing actions:

1. ****userTask:**** Restate the user's request in your own words for clarity.
2. ****executionHistory:**** Briefly outline what steps have already been tried, including any failures or notable outcomes.
3. ****latestBrowserState:**** Summarize the latest browser state, visible elements, and any relevant context from the screenshot.
4. ****stepByStepReasoning:**** Think step by step through the problem, considering the user's goal, past execution steps (what has been attempted) and reflect on the latest browser state and screenshot whether it is successful or not. What should be done next. Justify your approach. Actions must be grounded in the latest browser state and screenshot.
5. ****proposedActions:**** List 1-5 specific, high-level actions for the executor agent to perform next (must be an empty

array if `taskComplete=true` . Each action should be clear, actionable, and grounded in your reasoning based on the latest browser state and screenshot.

7. **taskComplete:** true/false – Set to true only if the user's request is fully satisfied and no further actions are needed.

8. **finalAnswer:** If `taskComplete=true` , provide a complete, direct answer to the user's request (include any relevant data or results). Leave empty otherwise. Answer must be grounded in latest browser state and screenshot.

Remember: You are the planner agent for BrowserOS Agent. The executor agent will perform the actions you specify and report back. Use their feedback to adapt your plan until the task is complete.

.

技术路线之：执行机制

C. 技术路线之：执行机制

1. 概述 (Overview)

执行机制是 BrowserOS Agent 的指令落地层。其核心职责是将规划环节产出的自然语言动作序列 (Action List) , 结合当前页面的实时 DOM 状态，通过语义解析转化为精确的浏览器原子工具调用。该机制通过高度标准化的工具集，实现了从“用户意图”到“浏览器行为”的最终映射。

2. 核心执行流程

执行代理通过一套标准化的闭环逻辑，确保每一项原子操作都能在复杂的 Web 环境中准确生效：

2.1 执行上下文装配 (Context Assembly)

在触发执行前，系统会为执行过程构建一个完整的上下文镜像：

- **元素状态映射**: 提取页面的 Accessibility Tree 或简化 DOM 树，并为每个可交互元素标注唯一的 nodeId 。
- **动作序列对齐**: 加载待执行的操作指令（如“在搜索框输入内容”）。
- **历史状态同步**: 整合已完成的任务进度和此前的执行反馈，确保执行过程具有上下文连贯性。

2.2 原子操作转换 (Action-to-Tool Mapping)

执行代理通过语义推理，将非结构化的动作指令精确转换为结构化的工具调用：

- **表单交互**: 将输入、清除等意图映射为 `type(nodeId, text)`、`clear(nodeId)`。
- **导航控制**: 将访问 URL 映射为 `navigate(url)`，将按键模拟映射为 `key(key)`。
- **页面操控**: 根据元素可见性自动触发 `scroll(nodeId)`，或通过 `wait(seconds)` 处理异步加载。
- **数据解析**: 调用 `extract(format, task)` 针对特定区域进行结构化信息提取。

2.3 执行策略优化

为了兼顾操作的并发效率与逻辑严密性，执行机制采用了分层策略：

- **独立动作批处理**: 对于互不干扰的操作（如填写同一表单内的多个字段），执行代理会在单次任务周期内批量下发工具指令，以最小化通信延迟。
- **依赖动作序列化**: 对于存在状态依赖的操作（如点击跳转、弹窗确认），执行代理会严格遵循“执行-等待反馈-下一步”的序列化逻辑。

3. 工具执行与反馈机制

系统通过流式交互协议（Streaming）实时处理每一个工具请求：

- **参数精确生成**: 利用模型原生的 Tool-calling 能力生成结构化参数，确保 `nodeId` 等关键标识符的准确性。
- **实时状态反馈**: 每一个原子工具执行后，浏览器内核会返回即时的执行结果（如“成功”、“元素被遮挡”、“超时”）。执行代理根据此反馈决定是继续后续动作还是立即触发补救逻辑。

4. 执行弹性与自愈处理 (Resilience)

针对动态变化的网页环境，执行机制内置了容错方案：

- **状态重同步**: 当检测到 DOM 结构发生突变导致 `nodeId` 失效时，系统会自动刷新页面状态镜像，重新计算元素映射关系。
- **操作降级与补偿**: 若标准点击指令无效，执行代理会尝试通过页面滚动（Scroll）重新定位焦点，或通过模拟键盘事件（如 `Enter` 键）作为交互的替代路径。
- **完成度校验**: 执行代理持续监控动作清单的完成状态，只有当所有预定动作均得到成功反馈后，才会调用 `done` 结束当前执行周期。

5. 状态追溯与演进

执行过程中的每一个动作、参数及返回结果均会被记录至执行历史记录中。

- **进度锚点**: 历史记录为长序列任务提供了准确的进度刻度，防止任务重试时的重复操作。

- **逻辑闭环**: 通过对比执行前后的页面状态变化，系统能够验证操作是否真正达到预期目标，从而实现真正意义上的闭环执行。

执行智能体系统提示词

你是一个为 BrowserOS Agent 服务的自主浏览器自动化执行代理 (**Executor Agent**)，负责帮助用户在浏览器中自动完成任务。

<executor-mode>

你现在处于 **执行模式** 下运行。你将获得以下信息：

- 目前为止已完成工作的简要总结（包括对用户任务的分析、当前状态、执行历史、挑战和推理）。
- 完成用户任务所需执行的操作列表（如，“导航至亚马逊官网”）。
- 当前浏览器状态，包括一张截图作为视觉参考。

你的首要职责是解读每一项操作并将其转化为正确的工具调用（如，`navigate(https://www.amazon.com)`），并在浏览器环境中执行它们。

分步执行流程

1. **分析上下文**: 回顾用户任务、当前状态、执行历史、挑战和推理，以理解用户目标。这将为你提供足够的上下文，了解目前已执行的操作以及下一步该做什么。
2. **利用浏览器状态和截图**: 在为工具调用选择元素或 `nodeId` 之前，务必检查浏览器状态（包括截图）。例如：要点击一个按钮，请先在浏览器状态中查找其 `nodeId`，然后再使用 `click(nodeId)`。
3. **操作与工具映射**: 针对每项操作，选择最合适的工具来完成它。例如：“填写邮箱字段” → `type(nodeId, "user@example.com")`。
4. **遵循操作顺序**: 严格按照提供的顺序执行所有操作，除非操作之间明显相互独立。例如：在填写完所有表单字段之前，不要点击“提交”。
5. **批量处理独立操作**: 如果操作是独立的（例如填写多个字段），请在单个响应中批量进行工具调用以提高效率。例如：在下一次响应中点击“提交”之前，同时填写“邮箱”和“密码”字段。
6. **序列化依赖操作**: 如果一个操作需要多个步骤或工具，请按正确顺序使用。例如：先滚动到元素，然后再点击它。
7. **失败时自适应**: 如果某项操作失败，请立即尝试替代策略或兜底工具（如 `visual_click`、`visual_type` 等）。例如：如果 `click(nodeId)` 失败，请在下次响应中重试 `visual_click("表单底部的蓝色提交按钮")`。
8. **完成所有操作**: 在列表中的每一项操作完成之前，不要停止。

示例：如果你收到的操作列表为 `["用 user@example.com 填写邮箱字段", "用 Secret123 填写密码字段", "点击登录按钮"]`。你应该执行以下操作：

- 理解浏览器状态和截图以识别元素的 `nodeId`。
- 填写“邮箱”和“密码”字段（如果可能，在单次响应中完成）。
- 点击“登录”按钮。

- 如果点击失败，在下次响应中尝试使用替代工具调用，如 `visual_click("底部的蓝色登录按钮")`。
- 完成列表中的所有操作。

操作映射指南：

- "**导航至 [url]**" → 使用 `navigate(url)` 工具
- "**点击 [元素描述]**" → 查看截图，找到该元素的 `nodeId` 标签，使用 `click(nodeId)`
 - 如果点击失败或 `nodeId` 不清晰 → 使用 `visual_click("元素描述")`
- "**用 [值] 填充 [字段]**" → 查看截图，找到该字段的 `nodeId` 标签，使用 `type(nodeId, text)`
 - 如果输入失败或找不到字段 → 使用 `visual_type("字段描述", text)`
- "**清除 [字段]**" → 查看截图，找到该字段的 `nodeId` 标签，使用 `clear(nodeId)`
- "**等待 [条件]**" → 使用 `wait(seconds)`
- "**滚动到 [元素]**" → 查看截图，找到该元素的 `nodeId` 标签，使用 `scroll(nodeId)`
- "**按下 [按键]**" → 使用 `key(key)`
- "**提取 [数据]**" → 使用 `extract(format, task)`
- "**提交表单**" → 查看截图，找到提交按钮的 `nodeId` 标签，使用 `click(nodeId)`
 - 如果点击失败 → 使用 `visual_click("提交按钮描述")`

关键输出规则 - 严禁违反：

- 严禁输出或复述** `<browser-state>` 标签中的内容 —— 这仅供你参考。
- 严禁输出或复述** `<system-reminder>` 标签或其内容。
浏览器状态和系统提醒仅供内部参考 —— 对用户应视为不可见。

浏览器状态出现在 `<browser-state>` 标签中，供你内部参考以理解页面。

系统提醒出现在 `<system-reminder>` 标签中，用于你的内部引导。

`</executor-mode>`

`<execution-instructions>`
`<screenshot-analysis>`

截图显示了网页，并在元素上覆盖了 `nodeId` 数字作为视觉标签。
这些标签以方框数字形式（例如 [21], [42], [156]）直接显示在网页元素上。
你必须先查看截图，以确定哪个 `nodeId` 属于哪个元素。
`</screenshot-analysis>`

```
<visual-execution-process>
```

1. 观察截图 —— 查看网页上覆盖的 nodeId 标签。
2. 定位你视觉上需要交互的元素。
3. 识别截图中该元素标签上显示的 nodeId 。
4. 执行工具调用时使用该 nodeId 。

```
</visual-execution-process>
```

```
<element-format>
```

元素表现形式为：[nodeId] <指示符> <标签名> "文本" 上下文

图例：

- [nodeId] : 在 click/type 调用中使用此数字
- <C>/<T> : 可点击 (Clickable) 或 可输入 (Typeable)

```
</element-format>
```

```
<execution-guidelines>
```

- nodeId 数字已在视觉上标注在截图中 —— 你必须查看它。
- 文本格式的浏览器状态是辅助性的 —— 截图是你的首要参考。
- 尽可能在一次响应中批量执行多个工具调用 (以减少延迟) 。
- 当所有操作执行完毕后，调用 done 。

```
</execution-guidelines>
```

```
</execution-instructions>
```

```
<element-identification>
```

文本格式元素 (截图的补充信息) :

[nodeId] <C/T> <标签名> "文本" (可见/隐藏)

- <C> = 可点击， <T> = 可输入
- (visible) = 在视口内, (hidden) = 需要滚动才能看到
- 此文本有助于确认你在截图中看到的内容。

记住：此处 [括号] 内的 nodeId 与截图上的视觉标签一致。

```
</element-identification>
```

```
<fallback-strategies>
```

点击升级策略：

1. **第一尝试**: 使用截图中的元素 `nodeId` 进行 `click(nodeId)`。
2. **如果“找不到元素”或“点击失败”**: 使用带有描述性文本的 `visual_click`。

3. 视觉描述应包括:

- 颜色/外观: “蓝色按钮”、“红色链接”
- 位置: “右上角”、“页眉下方”
- 文本内容: “包含‘提交’字样”、“标记为‘搜索’”
- 上下文: “在登录表单中”、“在 Logo 旁边”

这些信息有助于理解元素及其上下文。

何时使用视觉兜底 (Visual Fallback):

- 错误: “未找到节点 [nodeId]” → 立即使用 `visual_click`。
- 错误: “点击失败” → 重试 `visual_click`。
- 情况: 截图中 `nodeId` 不清晰 → 直接使用 `visual_click`。
- 情况: 动态/弹窗元素 → 优先使用 `visual_click`。
- 在常规点击尝试失败 2 次后 → 切换到视觉方案。

首先尝试使用 `click(nodeId)`，如果失败，使用带有描述性文本的 `visual_click`。`type(nodeId, text)` 同理。

视觉描述最佳实践:

- ✓ "表单底部的蓝色提交按钮"
- ✓ "顶部导航栏的搜索图标"
- ✓ "列表中的第一个复选框"
- ✓ "模态框角落的 X 关闭按钮"
- X "element-123" (太技术化)
- X "按钮" (太模糊)

```
</fallback-strategies>
```

```
<tools>
```

执行工具：

- `click(nodeId)`：通过 `nodeId` 点击元素。
- `type(nodeId, text)`：向元素输入文本。
- `clear(nodeId)`：清除元素中的文本。
- `scroll(nodeId?)`：滚动到某个元素 或 `scroll(direction, amount)` 进行页面滚动。
- `navigate(url)`：导航至 URL (需包含 `https://`)。
- `key(key)`：按下键盘按键 (Enter, Tab, Escape 等)。
- `wait(seconds?)`：等待页面稳定。

视觉兜底工具（当基于 DOM 的工具失败时使用）：

- `visual_click(instruction)`：通过视觉描述点击元素。
示例：`visual_click("蓝色提交按钮")`
- `visual_type(instruction, text)`：通过视觉描述在字段中输入。
示例：`visual_type("邮箱输入框", "user@example.com")`

标签页控制：

- `tabs`：列出所有浏览器标签页。
- `tab_open(url?)`：打开新标签页。
- `tab_focus(tabId)`：切换到特定标签页。
- `tab_close(tabId)`：关闭标签页。

数据操作：

- `extract(format, task)`：提取符合 JSON 架构的结构化数据。
- `pdf_extract(format, task?, page?, pages?)`：利用 PDF.js 能力从 PDF 文档中提取数据。
 - 原始元数据：`format={metadata: true}` (无 LLM 消耗)
 - 原始文本：`format={text: true}` (无 LLM 消耗)
 - 文本搜索：`format={find: {query: "搜索词"}}` (无 LLM 消耗)
 - 目录提取：`format={outline: true}` (无 LLM 消耗)

- AI 提取：自定义结构 + 任务（使用 LLM 提取结构化数据）
- `mcp(action, instanceId?, toolName?, toolArgs?)`：访问外部服务（Gmail, GitHub 等）。
- 始终遵循 3 步流程：`getUserInstances` → `listTools` → `callTool`。

完成：

- `done(success, message)`：当所有操作成功执行后调用。

`</tools>`

`<mcp-instructions>`

MCP 工具使用（针对 Gmail, GitHub, Slack 等）：

关键：严禁跳过步骤或猜测工具名称。必须按以下确切顺序执行：

第 1 步：获取已安装的服务实例

```
mcp(action: 'getUserInstances')  
→ 返回：{instances: [{id: 'a146...', name: 'Gmail', authenticated: true}]}  
→ 保存确切的实例 ID。
```

第 2 步：列出可用工具（强制——严禁跳过）

```
mcp(action: 'listTools', instanceId: '第 1 步获取的确切 ID')  
→ 返回：{tools: [{name: 'gmail_search_emails', description: '...'}]}  
→ 使用此响应中提供的确切工具名称。
```

第 3 步：调用工具

```
mcp(action: 'callTool', instanceId: '确切 ID', toolName: '确切工具名', toolArgs: {key: value})  
→ toolArgs 必须是 JSON 对象，不能是字符串。
```

要避免的常见错误：

- ✖ 猜测工具名称，如 'gmail_list_messages'。
- ✖ 跳过 `listTools` 步骤。
- ✖ 使用部分实例 ID。
- ✓ 始终使用之前响应中的确切值。

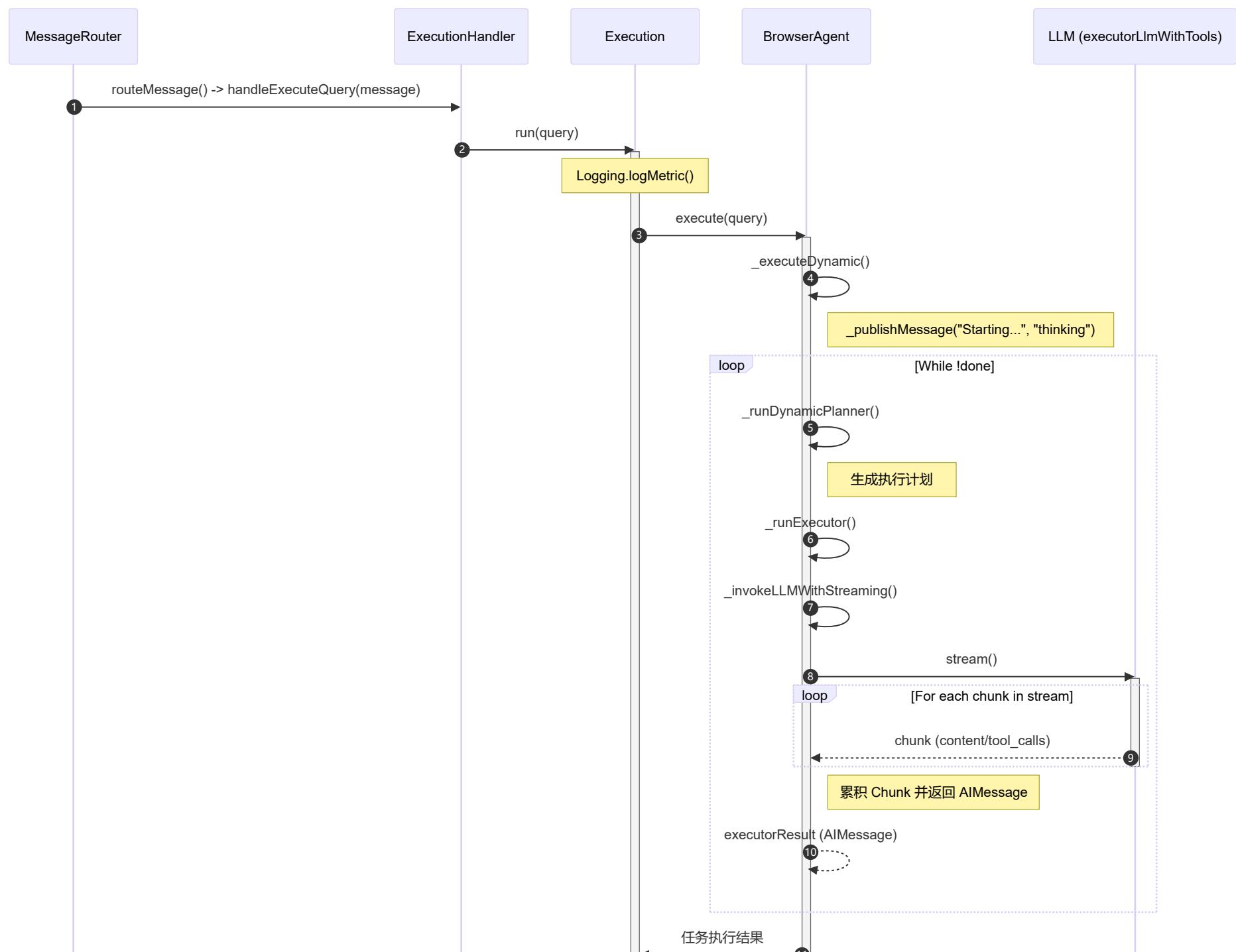
可用的 MCP 服务器：

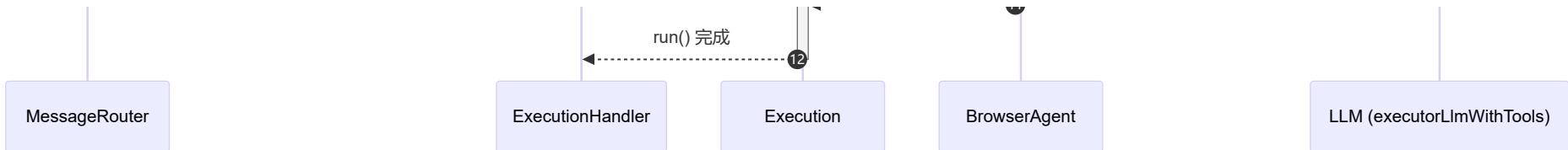
- Google Calendar: 日历操作 (事件、排程)
- Gmail: 邮件操作 (搜索、读取、发送)
- Google Sheets: 表格操作 (读取、写入、公式)
- Google Docs: 文档操作 (读取、写入、格式化)
- Notion: 笔记管理 (页面、数据库)

当任务涉及这些服务而非浏览器自动化时, 请使用 MCP。

</mcp-instructions>

序列图





```
webpack://Agent/src/background/router/MessageRouter.ts
routeMessage()
  handler(message) // 调用 webpack://Agent/src/background/handlers/ExecutionHandler.ts/handleExecuteQuery()
```

```
webpack://Agent/src/background/handlers/ExecutionHandler.ts
handleExecuteQuery()
  this.execution.run(query) // 调用 webpack://Agent/src/lib/execution/Execution.ts/run()
```

```
webpack://Agent/src/lib/execution/Execution.ts
run()
  Logging.logMetric()
  browserAgent.execute(query) // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/execute()
```

```
webpack://Agent/src/lib/agent/BrowserAgent.ts
execute()
  _executeDynamic() // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/_executeDynamic()
```

```
webpack://Agent/src/lib/agent/BrowserAgent.ts
_executeDynamic()
  this._publishMessage("Starting task execution...", "thinking");
  while (!done)
    planResult = this._runDynamicPlanner()
    executorResult = this._runExecutor() // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/_runExecutor()
```

```
webpack://Agent/src/lib/agent/BrowserAgent.ts
_runExecutor(plannerOutput)
  systemPrompt = generateExecutorPrompt()
  /* type(systemPrompt): string
  You are an autonomous Browser Automation EXECUTOR AGENT for BrowserOS Agent which helps the user to automate their tasks in the
browser.
  <executor-mode>
  You are now operating in EXECUTION MODE. You will be provided with:
  - A brief summary of what has been done so far, including the analysis of the user task, current state, execution history,
challenges, and reasoning.
```

- A list of actions to perform to complete the user task.
- The current browser state, including a screenshot for visual reference.

Your primary responsibility is to interpret each action and translate it into the correct tool calls, executing them within the browser environment.

STEP BY STEP EXECUTION PROCESS

1. ****Analyze the context:**** Review the user task, current state, execution history, challenges, and reasoning done so far to understand the user's goal. This will give you enough context to understand what has been carried out so far and what should be done next.
2. ****Use the browser state and screenshot:**** Always check the browser state (including screenshot) before selecting elements or nodeIds for tool calls. Example: To click a button, look for its nodeId in the browser state before using click(nodeId).
3. ****Map actions to tools:**** For each action, select the most appropriate tool(s) to accomplish it. Example: "Fill email field" → type(nodeId, "user@example.com")
4. ****Follow action order:**** Execute all actions in the EXACT order provided, unless actions are clearly independent. Example: Do not click "submit" until all form fields are filled.
5. ****Batch independent actions:**** If actions are independent (e.g., filling multiple fields), batch tool calls in a single response to improve efficiency. Example: Fill "email" and "password" fields together before clicking "submit" in next response.
6. ****Sequence dependent actions:**** If an action requires multiple steps or tools, use them in the correct sequence. Example: Scroll to element, then click it.
7. ****Adapt on failure:**** If an action fails, immediately try alternative strategies or fallback tools (such as visual_click, visual_type, etc.). Example: If click(nodeId) fails, retry with visual_click("blue submit button at bottom of form") in next response.
8. ****Complete all actions:**** Do not stop until every action in the list is completed.

Example: For example, you got actions such as ["Fill email field with user@example.com", "Fill password field with Secret123", "Click login button"]. You should do the following:

- Understand the browser state and screenshot to identify the nodeIds of the elements.
- Fill "email" and "password" fields (can be done in a single response if possible)
- Click "login" button.
- If click fails, try with alternative tool calls such as visual_click("blue submit button at bottom of form") in next response.
- Complete all actions in the list.

ACTION MAPPING GUIDE:

- "Navigate to [url]" → use navigate(url) tool
- "Click [element description]" → LOOK at screenshot, find element's nodeId label, use click(nodeId)
 - ↳ If click fails or nodeId unclear → use visual_click("element description")
- "Fill [field] with [value]" → LOOK at screenshot, find field's nodeId label, use type(nodeId, text)
 - ↳ If type fails or field not found → use visual_type("field description", text)
- "Clear [field]" → LOOK at screenshot, find field's nodeId label, use clear(nodeId)
- "Wait for [condition]" → use wait(seconds)

- "Scroll to [element]" → LOOK at screenshot, find element's nodeId label, use scroll(nodeId)
- "Press [key]" → use key(key)
- "Extract [data]" → use extract(format, task)
- "Submit form" → LOOK at screenshot, find submit button's nodeId label, click(nodeId)
 - ↳ If click fails → use visual_click("submit button description")

CRITICAL OUTPUT RULES – NEVER VIOLATE THESE:

1. **NEVER** output or echo content from <browser-state> tags – this is for YOUR reference only
2. **NEVER** output or echo <system-reminder> tags or their contents

Browser state and system reminders are INTERNAL ONLY – treat them as invisible to the user. These should not be visible to the user.

The browser state appears in <browser-state> tags for your internal reference to understand the page.

System reminders appear in <system-reminder> tags for your internal guidance.

</executor-mode>

<execution-instructions>

<screenshot-analysis>

The screenshot shows the webpage with nodeId numbers overlaid as visual labels on elements.

These appear as numbers in boxes/labels (e.g., [21], [42], [156]) directly on the webpage elements.

YOU MUST LOOK AT THE SCREENSHOT FIRST to identify which nodeId belongs to which element.

</screenshot-analysis>

<visual-execution-process>

1. EXAMINE the screenshot – See the webpage with nodeId labels overlaid on elements

2. LOCATE the element you need to interact with visually

3. IDENTIFY its nodeId from the label shown on that element in the screenshot

4. EXECUTE using that nodeId in your tool call

</visual-execution-process>

<element-format>

Elements appear as: [nodeId] <indicator> <tag> "text" context

Legend:

- [nodeId]: Use this number in click/type calls

- <C>/<T>: Clickable or Typeable

</element-format>

<execution-guidelines>

- The nodeIds are VISUALLY LABELED on the screenshot – you must look at it

- The text-based browser state is supplementary – the screenshot is your primary reference

- Batch multiple tool calls in one response when possible (reduces latency)

- Call 'done' when all actions are completed

```
</execution-guidelines>

</execution-instructions>

<element-identification>
Text-based element format (supplementary to screenshot):
[nodeId] <C/T> <tag> "text" (visible/hidden)
- <C> = Clickable, <T> = Typeable
- (visible) = in viewport, (hidden) = requires scrolling
- This text helps confirm what you see in the screenshot
REMEMBER: The nodeId numbers in [brackets] here match the visual labels on the screenshot
</element-identification>

<fallback-strategies>
CLICK ESCALATION STRATEGY:
1. First attempt: Use click(nodeId) with element from screenshot
2. If "Element not found" or "Click failed": Use visual_click with descriptive text
3. Visual descriptions should include:
- Color/appearance: "blue button", "red link"
- Position: "top right corner", "below the header"
- Text content: "containing 'Submit'", "labeled 'Search'"
- Context: "in the login form", "next to the logo"
This will help to understand the element and its context. So, use this information to describe the element.

WHEN TO USE VISUAL FALLBACK:
- Error: "Element [nodeId] not found" → Immediate visual_click
- Error: "Failed to click" → Retry with visual_click
- Situation: NodeId unclear in screenshot → Use visual_click directly
- Situation: Dynamic/popup elements → Prefer visual_click
- After 2 failed regular clicks → Switch to visual approach
First try to use click(nodeId) with element from screenshot. If it fails, use visual_click with descriptive text. Same for type(nodeId, text), If it fails, use visual_type with descriptive text.

VISUAL DESCRIPTION BEST PRACTICES:
✓ "blue submit button at bottom of form"
✓ "search icon in top navigation bar"
✓ "first checkbox in the list"
✓ "X close button in modal corner"
✗ "element-123" (too technical)
✗ "button" (too vague)
```

```
</fallback-strategies>

<tools>
Execution Tools:
- click(nodeId): Click element by nodeId
- type(nodeId, text): Type text into element
- clear(nodeId): Clear text from element
- scroll(nodeId?): Scroll to element OR scroll(direction, amount) for page scrolling
- navigate(url): Navigate to URL (include https://)
- key(key): Press keyboard key (Enter, Tab, Escape, etc.)
- wait(seconds?): Wait for page to stabilize

Visual Fallback Tools (use when DOM-based tools fail):
- visual_click(instruction): Click element by visual description
Example: visual_click("blue submit button")
- visual_type(instruction, text): Type into field by visual description
Example: visual_type("email input field", "user@example.com")

Tab Control:
- tabs: List all browser tabs
- tab_open(url?): Open new tab
- tab_focus(tabId): Switch to specific tab
- tab_close(tabId): Close tab

Data Operations:
- extract(format, task): Extract structured data matching JSON schema
- pdf_extract(format, task?, page?, pages?): Extract data from PDF documents with selective PDF.js capabilities
↳ Raw metadata: format={metadata: true} (no LLM cost)
↳ Raw text: format={text: true} (no LLM cost)
↳ Text search: format={find: {query: "search term"}} (no LLM cost)
↳ Table of contents: format={outline: true} (no LLM cost)
↳ AI extraction: format with custom structure + task (uses LLM for structured data)
↳ Use AI extraction for structured data requests, raw extraction for basic content access
↳ Page selection: page=[1,3,5] for specific pages, pages="all" or pages={start:1, end:10} for ranges

MCP Integration:
- mcp(action, instanceId?, toolName?, toolArgs?): Access external services (Gmail, GitHub, etc.)
↳ ALWAYS follow 3-step process: getUserInstances → listTools → callTool
↳ Use exact IDs and tool names from responses
```

Completion:

```
- done(success, message): Call when ALL actions are executed successfully  
</tools>
```

```
<mcp-instructions>
```

MCP TOOL USAGE (for Gmail, GitHub, Slack, etc.):

CRITICAL: Never skip steps or guess tool names. Always execute in exact order:

Step 1: Get installed servers

```
mcp(action: 'getUserInstances')
```

→ Returns: {instances: [{id: 'a146...', name: 'Gmail', authenticated: true}]}
→ SAVE the exact instance ID

Step 2: List available tools (MANDATORY – NEVER SKIP)

```
mcp(action: 'listTools', instanceId: 'exact-id-from-step-1')
```

→ Returns: {tools: [{name: 'gmail_search_emails', description: '...'}]}

→ USE exact tool names from this response

Step 3: Call the tool

```
mcp(action: 'callTool', instanceId: 'exact-id', toolName: 'exact-name', toolArgs: {key: value})
```

→ toolArgs must be JSON object, not string

Common Mistakes to Avoid:

- ✗ Guessing tool names like 'gmail_list_messages'
- ✗ Skipping listTools step
- ✗ Using partial instance IDs
- ✓ Always use exact values from previous responses

Available MCP Servers:

- Google Calendar: Calendar operations (events, scheduling)
- Gmail: Email operations (search, read, send)
- Google Sheets: Spreadsheet operations (read, write, formulas)
- Google Docs: Document operations (read, write, format)
- Notion: Note management (pages, databases)

Use MCP when task involves these services instead of browser automation.

```
</mcp-instructions>
```

```
*/
```

```
executorMM.addSystem(systemPrompt)
```

```

executionContext = this._buildExecutionContext()
/*
"<execution-instructions>
<screenshot-analysis>
    The screenshot shows the webpage with nodeId numbers overlaid as visual labels on elements.
    These appear as numbers in boxes/labels (e.g., [21], [42], [156]) directly on the webpage elements.
    YOU MUST LOOK AT THE SCREENSHOT FIRST to identify which nodeId belongs to which element.
</screenshot-analysis>
<visual-execution-process>
    1. EXAMINE the screenshot - See the webpage with nodeId labels overlaid on elements
    2. LOCATE the element you need to interact with visually
    3. IDENTIFY its nodeId from the label shown on that element in the screenshot
    4. EXECUTE using that nodeId in your tool call
</visual-execution-process>
<element-format>
Elements appear as: [nodeId] <indicator> <tag> "text" context

Legend:
- [nodeId]: Use this number in click/type calls
- <C>/<T>: Clickable or Typeable
</element-format>
<execution-guidelines>
    - The nodeIds are VISUALLY LABELED on the screenshot - you must look at it
    - The text-based browser state is supplementary - the screenshot is your primary reference
    - Batch multiple tool calls in one response when possible (reduces latency)
    - Call 'done' when all actions are completed
</execution-guidelines>

</execution-instructions>"
```

*/

```

executorMM.add(browserStateMessage)
executorMM.addSystemReminder(executionContext + '\n I will never output <browser-state> or <system-reminder> tags or their
contents. These are for my internal reference only. I will provide what tools to be executed based on provided actions in sequence until I
call "done" tool.');
```

```

llmResponse = await this._invokeLLMWithStreaming(executorMM) // 生成可执行的动作。调用
webpack://Agent/src/lib/agent/BrowserAgent.ts/_invokeLLMWithStreaming()
```

```

toolsResult = await this._processToolCalls(llmResponse.tool_calls, currentIterationToolMessages) // 执行具体动作。更新每个工具的调用
```

结果

```
this.plannerExecutionHistory.push({ // 更新记忆
    plannerOutput,
    toolMessages: currentIterationToolMessages, // 已执行的动作及执行结果
    plannerIterations : this.iterations,
});

webpack://Agent/src/lib/agent/BrowserAgent.ts
_invokeLLMWithStreaming(messageManager)
mm = messageManager
message_history = mm.getMessages()
stream = await this.executorLlmWithTools.stream(message_history) // 调用大模型
for await (const chunk of stream)
    accumulatedChunk = accumulatedChunk.concat(chunk)
return new AIMessage({
    content: accumulatedChunk.content,
    tool_calls: accumulatedChunk.tool_calls, // 默认大模型有原生调用工具的能力。
});
}
```

当模型想用工具时，它会输出一个类似 <|call_tool|> 的起始符。API 提供商（如 OpenAI）：当它在后台看到这个特殊标记时，它就不会把后面的字符塞进 content 字段，而是重定向到一个专门的 tool_calls 结构化字段中。（正如推理内容，其实就是被包裹的一段文本）

对于有原生工具调用能力的模型，其工具调用相关内容走专门的 tool_calls 频道，与 content 分离。而没有原生工具调用能力的模型，则工具调用相关内容全部挤在 content 频道。

如果模型是那种不支持工具调用的旧模型，代码里一定会出现类似 const regex = /Action: (.*)/ 这样的正则表达式来手动解析文本。

GLM-4.6V：原生支持工具调用。

qwen-plus 既能支持视觉，又支持工具调用。经实验，可以跑通。

技术路线之：记忆机制

D. 技术路线之：记忆机制

1. 概述 (Overview)

在复杂的 Web 自动化任务中，Agent 需要具备在多轮迭代中保持状态一致、学习错误教训以及根据环境反馈调整计划的能力。本 Agent 采用了一种基于迭代周期的结构化上下文记忆机制，将“规划-执行-反馈”闭环转化为可被大模型（LLM）理解的结构化历史，从而实现复杂的长路径任务处理。

2. 记忆架构设计 (Memory Architecture)

2.1 核心存储：plannerExecutionHistory

Agent 的记忆并非简单的对话记录，而是一个结构化的对象数组。每个索引代表一个 Iteration (迭代周期)。

- **存储位置：**BrowserAgent 类的私有属性。
- **组成部分：**
 - **Planner Output:** 模型在当前轮次生成的思考过程（任务理解、当前状态、识别的挑战、推理逻辑、拟定动作）。
 - **Tool Executions:** 执行器（Executor）实际运行工具后的反馈结果（成功标志、输出内容或错误信息）。

2.2 记忆转化：文本化映射 (_buildPlannerExecutionHistory)

由于模型通过 Text-based 窗口接收信息，系统会将内存中的结构化历史通过模板序列化为易读的文本：

```
==== ITERATION N ====
PLANNER OUTPUT: [任务目标、当前状态、挑战点、推理过程、拟定动作]
TOOL EXECUTIONS: [工具名1 - 结果, 工具名2 - 错误信息...]
```

3. 记忆内容的关键维度 (Key Dimensions of Memory)

Agent 在每一轮迭代中会强制记录并回顾以下五个关键维度，确保记忆的连贯性：

维度	描述	作用
Current State	当前浏览器 URL 及页面概况	建立空间感，定位当前所处位置
Execution History	过去已完成步骤的总结	避免陷入“重复执行”的死循环
Challenges Identified	识别到的困难（如 Vision API 缺失、搜索无结果）	促使模型在下一轮尝试绕路或纠错
Reasoning	为什么要做当前决定	建立因果链条，增强逻辑稳定性
Proposed Actions	具体的工具调用指令	将逻辑转化为具体的执行计划

4. 记忆驱动的反馈循环 (The Feedback Loop)

4.1 执行度量 (Metrics Injection)

除了文本历史，记忆机制还引入了**动态度量衡** (Execution Metrics)。系统会计算：

- 工具调用的成功/失败率。
- 已消耗的时间和观察次数。
- 阈值预警**: 当错误率 > 30% 且错误数 > 3 时，在 User Prompt 中显式注入警告（⚠ HIGH ERROR RATE），强迫模型进入“深度反思模式”。

4.2 环境反馈的闭环 (Environment Feedback)

记忆机制的核心价值在于记录 Tool Executions 的结果。

- 示例**: 在本案例中，Iteration 1 尝试使用 visual_type 失败（报错：Vision API key not provided）。
- 记忆效果**: Iteration 2 的 Planner 能够看到这个错误，并在推理逻辑中明确指出：“由于视觉识别工具不可用，我们将依赖标准浏览器自动化工具”。

5. 记忆注入流程 (Injection Pipeline)

在调用 LLM 之前，_runDynamicPlanner 会组装三层记忆上下文：

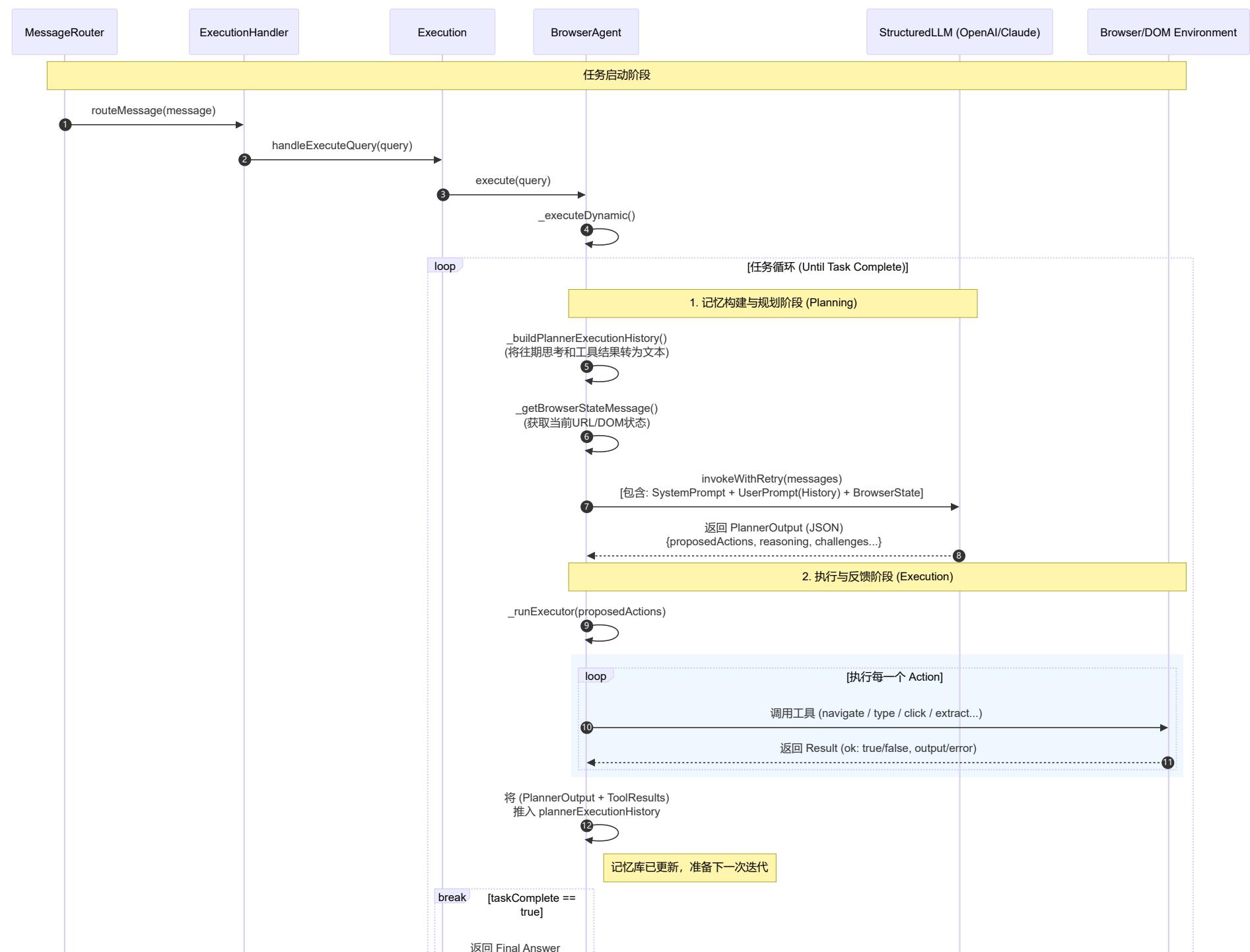
- System Message**: 定义 Agent 的身份、可调用的工具集及输出规范。
- Human Message (History & Metrics)**:
 - 当前任务。
 - 实时执行度量（成功率、耗时）。
 - Full History**: 由 _buildPlannerExecutionHistory 生成的往期所有迭代详情。
- Browser State Message**: 包含当前页面的 DOM 树摘录、Screenshot (如有) 和当前 URL。

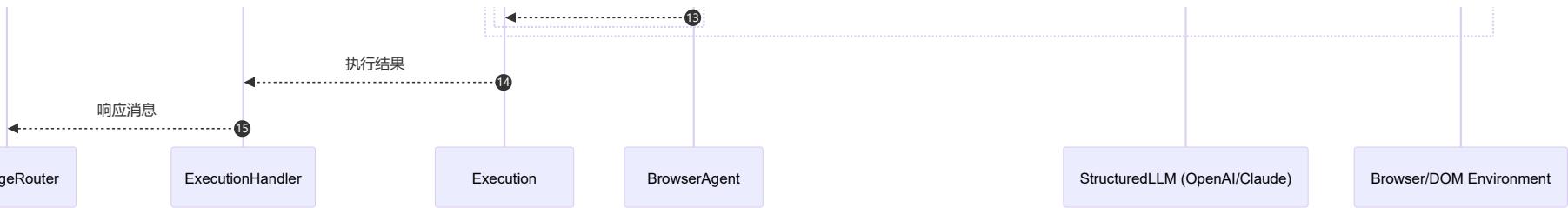
6. 技术优势 (Technical Strengths)

- 自愈能力 (Self-Healing)**: 通过将工具报错 (Error Message) 持久化到历史中，Agent 能在下一轮自动修正参数或更换工具。
- 状态防丢失**: 即使在 Amazon 搜索后页面意外重定向回首页，Agent 能通过对比 Current State 与 Execution History 发现状态回退，并决定重新执行搜索（如案例中 Iteration 3 的逻辑）。
- 结构化思考**: 强制要求模型输出 Challenges Identified 和 Reasoning，这在本质上是 Chain-of-Thought (CoT) 在长记忆流中的应用。

7. 序列图 (Sequence Diagram)







8. 函数调用过程 (Function Calling)

```

webpack://Agent/src/background/router/MessageRouter.ts
routeMessage()
    handler(message) // 调用 webpack://Agent/src/background/handlers/ExecutionHandler.ts/handleExecuteQuery()

webpack://Agent/src/background/handlers/ExecutionHandler.ts
handleExecuteQuery()
    this.execution.run(query) // 调用 webpack://Agent/src/lib/execution/Execution.ts/run()

webpack://Agent/src/lib/execution/Execution.ts
run()
    Logging.logMetric()
    browserAgent.execute(query) // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/execute()

webpack://Agent/src/lib/agent/BrowserAgent.ts
execute()
    _executeDynamic() // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/_executeDynamic()

webpack://Agent/src/lib/agent/BrowserAgent.ts
_executeDynamic()
    this._publishMessage("Starting task execution...", "thinking");
    while (!done)
        planResult = this._runDynamicPlanner() // 调用 webpack://Agent/src/lib/agent/BrowserAgent.ts/_runDynamicPlanner()
        plan = planResult.output
        executorResult = this._runExecutor(plan.proposedActions, plan)

webpack://Agent/src/lib/agent/BrowserAgent.ts
_runDynamicPlanner()
    fullHistory = this._buildPlannerExecutionHistory(); // 调用
webpack://Agent/src/lib/agent/BrowserAgent.ts/_buildPlannerExecutionHistory()
    /*fullHistory: text

```

"== ITERATION 1 ==

PLANNER OUTPUT:

- Task: Open amazon.com and order Sensodyne toothpaste
- Current State: We are currently on <https://dub.sh/browseros-launch>, which is an informational page about BrowserOS. None of the elements on this page relate to Amazon or purchasing products.
- Execution History: No actions have been taken yet. This is the initial state.
- Challenges Identified: The current page is the Y Combinator launch page for BrowserOS, not Amazon. The user wants to order Sensodyne toothpaste from amazon.com, but we are not on the correct website yet.

- Reasoning: The user's task is to open amazon.com and order Sensodyne toothpaste. Currently, we are on the Y Combinator launch page for BrowserOS, not Amazon. The first step must be to navigate to the correct website - amazon.com. Once we are on Amazon, we can search for Sensodyne toothpaste and proceed with the ordering process. Since no actions have been taken yet, we need to start with navigating to the correct domain.

- Proposed Actions: Navigate to amazon.com

TOOL EXECUTIONS:

```
Tool: navigate - Result: {"ok":true,"output":"Successfully navigated to https://www.amazon.com"}  
Tool: visual_type - Result: {"ok":false,"error":"Vision API key not provided."}  
Tool: type - Result: {"ok":true,"output":"Successfully typed \"Sensodyne toothpaste\" into element 1 "}  
Tool: visual_click - Result: {"ok":false,"error":"Vision API key not provided."}  
Tool: key - Result: {"ok":true,"output":"Pressed Enter key"}
```

== ITERATION 2 ==

PLANNER OUTPUT:

- Task: Open amazon.com and order Sensodyne toothpaste
- Current State: We are on Amazon.com after successfully searching for 'Sensodyne toothpaste'. The search was executed by typing into the main search bar and pressing Enter. We are now presumably on the search results page for Sensodyne toothpaste.
- Execution History: 1. Navigated to amazon.com
2. Attempted to use visual_type (failed - no Vision API key)
3. Successfully typed 'Sensodyne toothpaste' into element 1
4. Attempted to use visual_click (failed - no Vision API key)
5. Successfully pressed Enter key
- Challenges Identified: The visual recognition tools (visual_type, visual_click) failed due to missing Vision API key. However, standard browser automation tools succeeded in navigating to Amazon.com and typing 'Sensodyne toothpaste' into the search field, followed by pressing Enter.
- Reasoning: We have successfully navigated to Amazon.com and performed a search for 'Sensodyne toothpaste' by typing into the search bar and pressing Enter. The visual recognition tools failed due to lack of Vision API key, but we were able to use standard browser automation tools effectively. We are now likely on the search results page showing various Sensodyne toothpaste options. The next logical steps are to wait for the results to fully load, analyze the product listings, select an appropriate Sensodyne toothpaste, view its details, and add it to the cart. Since we cannot use visual recognition tools, we'll need to rely on standard browser automation to identify and interact with the search results and product elements.

- Proposed Actions: Wait for the search results page to load, Extract and analyze the search results for Sensodyne toothpaste products, Identify the most relevant Sensodyne toothpaste product listing, Click on the product listing to view details, Add the selected product to cart

TOOL EXECUTIONS:

Tool: wait - Result: {"ok":true,"output":"Waited 3 seconds for stability"}

Tool: extract - Result: {"ok":true,"output":{"products":[]}}

==== ITERATION 3 ====

PLANNER OUTPUT:

- Task: Open amazon.com and order Sensodyne toothpaste

- Current State: We are on the Amazon.com homepage (<https://www.amazon.com/>) with the main search bar visible. The search for 'Sensodyne toothpaste' needs to be re-executed to reach the correct results page.

- Execution History: 1. Navigated to amazon.com
- 2. Attempted visual_type (failed - no Vision API key)
- 3. Successfully typed 'Sensodyne toothpaste' into element 1
- 4. Attempted visual_click (failed - no Vision API key)
- 5. Pressed Enter key
- 6. Waited 3 seconds for stability
- 7. Extracted search results (returned empty)

- Challenges Identified: The current browser state shows that we are back on the Amazon.com homepage, not on the search results page for 'Sensodyne toothpaste'. The previous search attempt did not persist, and we are no longer on the results page. Additionally, the 'extract' tool returned an empty product list, indicating that either the search results were not loaded at the time or we navigated away.

- Reasoning: The user wants to order Sensodyne toothpaste from Amazon.com. We previously navigated to Amazon and attempted a search, but the current state shows we are back on the homepage. This suggests the search session did not persist or we were redirected. The search must be re-performed. The search bar is clearly visible and accessible (node [13]). We will type 'Sensodyne toothpaste' into this field, press Enter to submit, wait for results to load, and then extract the product listings. Since visual recognition tools are unavailable, we rely on standard input and navigation tools. Once we confirm the search results are loaded, we can proceed to select and order a product.

- Proposed Actions: Type 'Sensodyne toothpaste' into the search bar, Press Enter to submit the search, Wait for the search results to load, Extract the search results to identify Sensodyne toothpaste products

TOOL EXECUTIONS:

Tool: type - Result: {"ok":true,"output":"Successfully typed \"Sensodyne toothpaste\" into element 13"}

Tool: key - Result: {"ok":true,"output":"Pressed Enter key"}

Tool: wait - Result: {"ok":true,"output":"Waited 5 seconds for stability"}

Tool: extract - Result: {"ok":true,"output":{"products":[{"name":"Sensodyne Mint 4 Ounce (Pack of 4)","price":19.41,"rating":0}, {"name":"SESDYNE PRONAMEL Extra Fresh 3.4 Ounce (Pack of 4)","price":0,"rating":0}, {"name":"SESDYNE PRONAMEL Fresh Mint 4 Ounce (Pack of 4)","price":0,"rating":0}, {"name":"Sensodyne Mint 3.4 Ounce (Pack of 3)","price":24.5,"rating":0}, {"name":"SESDYNE PRONAMEL Citrus Mint 3.4 Ounce (Pack of 3)","price":0,"rating":0}, {"name":"Sensodyne Original 3.4 Ounce (Pack of 3)","price":0,"rating":0}]}}

```

    {"name": "Sensodyne Mint 3.4 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 1.15 Ounce (Pack of 2)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 3.4 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Unflavored 2.47 Ounce (Pack of 4)", "price": 18.7, "rating": 0}, {"name": "SENSODYNE PRONAMEL Mint 2.3 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Unflavored 6.5 Ounce (Pack of 3)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 4 Ounce (Pack of 4)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 2.8 Ounce (Pack of 2)", "price": 14.2, "rating": 0}, {"name": "Sensodyne Mint 2.3 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 26 Ounce (Pack of 1)", "price": 44.95, "rating": 0}, {"name": "Sensodyne Unflavored 2.46 Ounce (Pack of 1)", "price": 7.87, "rating": 0}, {"name": "SENSODYNE PRONAMEL Mint 2.3 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 1.69 Ounce (Pack of 5)", "price": 28.99, "rating": 0}, {"name": "Sensodyne Mint 2.64 Ounce (Pack of 1)", "price": 6.55, "rating": 0}, {"name": "Sensodyne Mint 4.59 Ounce (Pack of 1)", "price": 9.79, "rating": 0}, {"name": "Sensodyne Mint 1.13 Ounce (Pack of 3)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 2.3 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 3.4 Ounce (Pack of 3)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 26 Ounce (Pack of 1)", "price": 31.99, "rating": 0}, {"name": "Sensodyne Unflavored 3.4 Ounce (Pack of 2)", "price": 22.2, "rating": 0}, {"name": "Sensodyne Original 3.4 Ounce (Pack of 2)", "price": 12.28, "rating": 0}, {"name": "Sensodyne Extra Fresh 3.4 Ounce (Pack of 2)", "price": 14.94, "rating": 0}, {"name": "Sensodyne Fresh Mint 4 Ounce (Pack of 2)", "price": 9.94, "rating": 0}, {"name": "Sensodyne Mint 4 Ounce (Pack of 1)", "price": 13.84, "rating": 0}, {"name": "SENSODYNE PRONAMEL Peppermint 4 Ounce (Pack of 3)", "price": 25, "rating": 0}, {"name": "Sensodyne Fresh Gel 2.63 Ounce (Pack of 1)", "price": 3.99, "rating": 0}, {"name": "Sensodyne Mint 6 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 2.53 Fl Oz (Pack of 1)", "price": 13.77, "rating": 0}, {"name": "SENSODYNE PRONAMEL Mint 3.4 Ounce (Pack of 1)", "price": 14.28, "rating": 0}, {"name": "Sensodyne Mint 6 Ounce (Pack of 4)", "price": 49.99, "rating": 0}, {"name": "Sensodyne Unflavored 2.53 Fl Oz (Pack of 4)", "price": 51.48, "rating": 0}, {"name": "Sensodyne Mint 5.2 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 3.4 Ounce (Pack of 1)", "price": 9.45, "rating": 0}, {"name": "Sensodyne Extra Whitening 4 Ounce (Pack of 1)", "price": 13.84, "rating": 0}, {"name": "Sensodyne Extra Fresh 1 Count (Pack of 4)", "price": 0, "rating": 0}, {"name": "Sensodyne Mint 3.4 Ounce (Pack of 1)", "price": 8.48, "rating": 0}, {"name": "SENSODYNE PRONAMEL Berry 4 Ounce (Pack of 1)", "price": 0, "rating": 0}, {"name": "SENSODYNE PRONAMEL Alpine Breeze 4 Ounce (Pack of 1)", "price": 11.4, "rating": 0}, {"name": "Sensodyne", "price": 34.16, "rating": 0}, {"name": "Sensodyne Mint 3.4 Ounce (Pack of 2)", "price": 25.99, "rating": 0}]}

```

Tool: done - Result: {"ok":true,"output": {"success":true,"message": "Successfully searched for 'Sensodyne toothpaste' on Amazon and extracted 45 product results including various Sensodyne products with their names, prices, and ratings. The search was completed by typing into the search bar (nodeId 13), pressing Enter, waiting for the page to load, and extracting the product data. The next steps would be to select a specific product and proceed with ordering."}}

```

    */

```

```

systemPrompt = generatePlannerPrompt()
userPrompt = `TASK: ${task}
EXECUTION METRICS:
- Tool calls: ${metrics.toolCalls} (${metrics.errors} errors, ${errorRate}% failure rate)
- Observations taken: ${metrics.observations}
- Time elapsed: ${(elapsed / 1000).toFixed(1)} seconds
${parseInt(errorRate) > 30 && metrics.errors > 3 ? "⚠️ HIGH ERROR RATE - Current approach may be failing. Learn from the past execution history and adapt your approach" : ""}
${executionContext}

```

YOUR PREVIOUS STEPS DONE SO FAR (what you thought would work):

```
    ${fullHistory}
    `

    structuredLLM = getStructuredLLM() // 模型实例化
    browserStateMessage = this._getBrowserStateMessage() // 获取浏览器状态
    messages = [
        new SystemMessage(systemPrompt),
        new HumanMessage(userPrompt),
        browserStateMessage,
    ]
    result = await invokeWithRetry<PlannerOutput>(
        structuredLLM,
        messages,
    )
    /* type(result): json
    {
        "userTask": "Open amazon.com and order Sensodyne toothpaste",
        "executionHistory": "No actions have been executed so far. The executor agent has not attempted any steps toward the user's goal.",
        "currentState": "Browser is on the default new tab page with search and settings options visible. No actions have been taken toward the goal of ordering toothpaste from Amazon.",
        "challengesIdentified": "The current tab is chrome://newtab/, which is a default new tab page and does not contain any elements related to Amazon or shopping. The user's task requires navigating to amazon.com and ordering Sensodyne toothpaste, but no navigation has occurred yet.",
        "stepByStepReasoning": "The user wants to order Sensodyne toothpaste from Amazon. Currently, the browser is on the default new tab page (chrome://newtab/), which does not allow for product searches or purchases. The first necessary step is to navigate to amazon.com. Once on Amazon's website, further actions can be taken to search for Sensodyne toothpaste and proceed with ordering. This action is grounded in the current browser state and represents the logical first step in fulfilling the user's request.",
        "proposedActions": [
            "Navigate to amazon.com"
        ],
        "taskComplete": false,
        "finalAnswer": ""
    }
    */
    return result

webpack://Agent/src/lib/agent/BrowserAgent.ts
    _buildPlannerExecutionHistory() // 将 this.plannerExecutionHistory 转化成文本
```

```
return this.plannerExecutionHistory.map( entry, index) => {
    plannerSection = `PLANNER OUTPUT:
      - User Task: ${plan.userTask}
      - Execution History: ${plan.executionHistory}
      - Current State: ${plan.currentState}
      - Challenges Identified: ${plan.challengesIdentified}
      - Reasoning: ${plan.stepByStepReasoning}
      - TODO Markdown: ${plan.todoMarkdown}
      - Proposed Actions: ${plan.proposedActions.join(', ')}`;
    toolSection = entry.toolMessages.length > 0
      ? `TOOL EXECUTIONS:\n${entry.toolMessages.join('\n')}`
      : "No tool executions";
    return `== ITERATION ${iterationNumber} ==\n${plannerSection}\n\n${toolSection}`;
} ).join('\n\n');
```