Contents

OpenCode 通用 Agent 平台设计方案

将 OpenCode 从编码助手抽象为通用 Agent 平台，融合 Swarm 蜂群智能、Google Agents 白皮书核心理念、Aime 动态规划架构与 oh-my-opencode 实战经验

目录

---

1. 设计目标

1.1 核心目标

将 OpenCode 从一个专注于编码的 AI 助手，转变为一个通用 Agent 平台，支持：

场景	说明
需求分析	分析用户故事、提取功能点、识别边界条件
需求文档	撰写 PRD、技术设计文档、API 文档
BUG 修复	问题定位、根因分析、修复方案、验证测试
特性开发	需求理解 → 设计 → 实现 → 测试 → 文档
代码审查	代码质量、安全性、性能、最佳实践
项目管理	任务分解、进度跟踪、风险评估
知识管理	文档生成、知识库构建、问答系统

1.2 设计原则

设计原则

1. 可扩展性 - 新 Agent/Tool 可热插拔，无需修改核心代码
2. 可组合性 - Agent 可组合、Tool 可复用、Workflow 可嵌套
3. 可配置性 - 通过配置而非代码定义行为
4. 可观测性 - 执行过程可追踪、可回放、可调试
5. 安全性 - 细粒度权限控制、沙箱执行、审计日志
6. 动态性 - Swarm 蜂群模式，Agent 可动态创建/销毁/扩容
7. 去中心化 - 扁平协作，人类可介入任意层级

1.3 Swarm 蜂群核心哲学

借鉴 Swarm-IDE 的设计思想，我们引入以下核心哲学：

Swarm 蜂群核心哲学

极简原语 (Minimal Primitives)
系统只依赖少量通信原语即可表达多 Agent 行为
核心是 create + send，复杂协作由此组合而来

液态拓扑 (Liquid Topology)
拓扑不预设、在运行中自演化
遇到复杂任务时由 Agent 主动"雇佣"下属

扁平协作 (Flat Collaboration)
人类可以像聊天一样介入任意层级
使复杂拓扑可观察、可调试、可介入

统一视角 (Unified Perspective)
用户和 Agent 完全等价，用户只是一种特殊的 Agent
所有视角可切换，消除人机交互的边界感

概念简化：没有 nodes 和 edges 的复杂抽象，只需把系统理解为"很多个人"：
每个人都能生孩子、也能和任意一个人说话。
只要有这两种能力，就能实现任意结构。

1.4 Google Agents 白皮书核心理念

基于 Google "Agents"白皮书（2024），整合以下核心概念：

1.4.1 Agent 认知架构

Agent 认知架构 (Cognitive Architecture)

Agent = Model + Orchestration + Tools

Model (大脑核心)	Orchestration (编排层)	Tools (能力扩展)
• 推理能力 • 规划能力 • 决策能力 • 学习能力	• ReAct • CoT • ToT • 多Agent协作	• Extensions • Functions • Data Stores • Code Exec

核心洞察:
- Agent 是扩展了模型能力边界的自主系统
- 模型是 Agent 的"大脑", 但不是全部
- 编排层决定 Agent 如何推理、规划、行动
- 工具让 Agent 能够感知和影响外部世界

1.4.2 Grounding（知识接地）

Grounding 知识接地机制

问题: 模型的知识有时效性限制和幻觉风险
方案: 通过 Grounding 连接实时外部数据源

Grounding Sources

Google Search	实时网络信息、新闻、事实核查
Data Stores	企业私有数据、向量数据库、知识图谱
Code Repos	代码库、文档、API 定义
APIs	第三方服务、实时数据接口
Databases	结构化数据查询、业务数据

实现模式:

1. RAG (Retrieval Augmented Generation) - 检索增强生成
2. 动态上下文注入 - 运行时获取最新信息
3. 知识图谱关联 - 结构化知识推理

1.4.3 工具类型体系

Agent 工具类型体系

Extensions（扩展）

- API 调用能力：连接外部服务（天气、地图、支付等）
- 标准化接口：OpenAPI 规范定义
- Agent 侧执行：模型决策后由 Agent 环境执行

Functions（函数）

- 客户端执行：模型生成参数，客户端执行逻辑
- 灵活性高：可访问客户端本地资源
- 适用场景：UI 操作、本地文件、敏感数据处理

Data Stores（数据存储）

- 向量数据库：语义相似性搜索
- 知识库索引：结构化/非结构化文档检索
- 动态更新：支持实时数据同步

Code Execution（代码执行）

- 沙箱环境：安全执行生成的代码
- 多语言支持：Python、JavaScript、Shell 等
- 结果反馈：执行结果返回给模型进行下一步推理

1.4.4 编排模式（Orchestration Patterns）

编排模式 (Orchestration Patterns)

ReAct (Reasoning + Acting)

```
Thought    Action    Observation    Thought    ...
"我需要..."  执行工具       获取结果        "根据结果..."
```

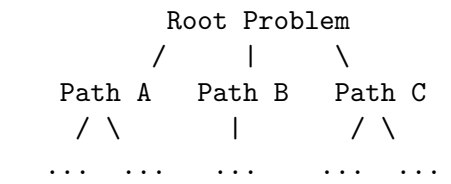
特点：交替进行推理和行动，动态调整策略

Chain-of-Thought (CoT) 思维链

```
Step 1    Step 2    Step 3    ...    Final Answer
分解问题      逐步推理      中间结论            最终答案
```

特点：显式化推理过程，提高复杂问题解决能力

Tree-of-Thought (ToT) 思维树



特点：探索多条推理路径，回溯择优，适合创造性任务

Self-Refine 自我迭代

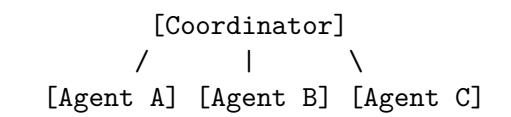
```
Generate    Critique    Refine    Critique    ...    Final
 初始方案       自我评审       改进方案
```

特点：迭代优化输出质量，无需外部反馈

1.4.5 多 Agent 协作模式

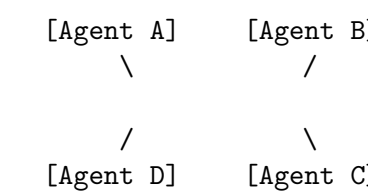
多 Agent 协作模式 (Multi-Agent Patterns)

1. 层级模式 (Hierarchical)



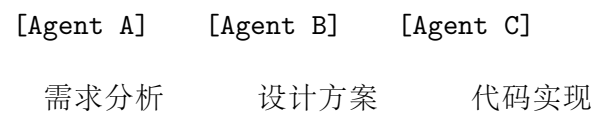
- 协调者分解任务分配给子 Agent
- 子 Agent 执行并汇报结果
- 适合：复杂任务分解、工作流管理

2. 对等模式 (Peer-to-Peer)



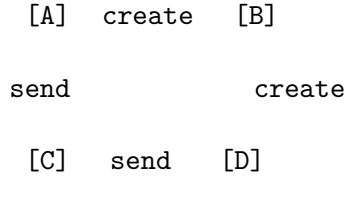
- Agent 之间直接通信协作
- 无中心协调者
- 适合：头脑风暴、多角度分析

3. 顺序模式 (Sequential/Pipeline)



- 上一个 Agent 输出作为下一个输入
- 流水线式处理
- 适合：阶段性工作流

4. 动态模式 (Dynamic/Swarm)



- Agent 可动态创建子 Agent
- 拓扑在运行时自演化
- 适合：复杂度不确定的任务

1.4.6 Agentic 系统分级（5 个级别） 基于 Google Agents 白皮书，Agentic 系统可按能力分为 5 个级别：

Agentic 系统分级 (5 Levels)

Level 0: 核心推理系统 (Core Reasoning System)

- 纯 LM 独立运行，基于预训练知识响应
- 无工具、无记忆、无实时交互
- 优势：广泛的训练知识，能解释概念和规划
- 劣势：完全缺乏实时意识，对训练数据外的事件"盲目"
- 示例：解释棒球规则 | 查询昨晚比赛结果

Level 1: 互联的问题解决者 (Connected Problem Solver)

- 推理引擎 + 外部工具连接
- 能力不再局限于静态的预训练知识
- 可调用 API、搜索引擎、数据库进行 RAG
- 示例：
  任务："昨晚洋基队比赛比分？"
  思考 → 识别为实时数据需求
  行动 → 调用 Google Search API
  观察 → "洋基队 5-3 获胜"
  输出 → 综合为最终答案

Level 2: 战略性问题解决者 (Strategic Problem Solver)

- 从执行简单任务转向战略性规划复杂目标
- 关键技能：上下文工程 (Context Engineering)
- 能主动选择、打包、管理每一步最相关的信息
- 示例：
  任务："在我办公室和客户办公室之间找个好的咖啡店"
  步骤1：思考→需要找中点 | 行动→调用地图工具 | 观察→米尔布雷
  步骤2：思考→搜索咖啡店 | 行动→调用places API(4星+)
  步骤3：思考→综合呈现 | 输出→推荐列表

Level 3：协作式多 Agent 系统 (Collaborative Multi-Agent)

- 从单一"超级 Agent"转向"专家团队"
- Agent 将其他 Agent 视为工具
- 模仿人类组织的劳动分工
- 示例：
  任务："发布新产品'Solaris'耳机"
  项目经理 Agent：
    → 分配 MarketResearchAgent："分析竞品定价"
    → 分配 MarketingAgent："起草新闻稿"
    → 分配 WebDevAgent："生成产品页面 HTML"
    → 聚合结果，完成发布

Level 4：自进化系统 (Self-Evolving System)

- 从委托执行到自主创造和适应
- 能识别自身能力差距，动态创建新工具/Agent
- 从使用固定资源转变为主动扩展资源
- 示例：
  任务："监控'Solaris'社交媒体情绪"
  思考(元推理) → "我缺乏这种能力"
  行动(自主创造) → 调用 AgentCreator 工具
    "构建一个新 Agent 用于社交媒体情感分析"
  观察 → 新的 SentimentAnalysisAgent 被即时创建
  结果 → 系统能力被动态扩展

核心洞察：每个级别都建立在上一级别的能力之上
选择哪个级别取决于任务复杂性、风险容忍度和资源约束

1.4.7 Agentic 5 步问题解决流程

Agentic 5 步问题解决流程

1. 获取任务　　用户请求 / 自动触发器
Get Task　　"为团队组织会议差旅"

2. 扫描场景　　感知环境，收集上下文
Scan Scene　　- 用户请求说了什么?
- 短期记忆中有什么?
- 工具能访问什么?

3. 思考分析　　核心"思考"循环，由推理模型驱动
Think　　对比任务(1)和场景(2)，制定计划
"要预订差旅，首先需要团队名单..."

4. 采取行动　　执行计划的第一个具体步骤
Act　　选择并调用工具: get_team_roster()
这是 Agent 对世界采取的行动

5. 观察迭代　　观察行动结果
Observe　　结果: 返回5人名单
添加到上下文/记忆

返回步骤 3，直到任务完成

实际示例: 客户支持 Agent

用户问: "我的订单 #12345 在哪里? "

思考分析（制定策略）:
"用户想要配送状态。我需要一个多步骤计划:
1. 识别: 在内部数据库找到订单，获取详细信息

2. 追踪：提取追踪号，查询承运商 API 获取实时状态
3. 报告：整合信息，生成清晰回复"

执行循环：
[行动] find_order("12345") [观察] 订单记录，追踪号="ZYX987"
[行动] get_shipping_status("ZYX987") [观察] "正在派送中"
[行动] 生成回复 "您的订单 #12345 正在派送中！"

1.4.8 记忆系统

Agent 记忆系统 (Memory System)

Memory Architecture

短期记忆
(Short-term)

- 当前会话上下文
- 最近N轮对话
- 工作中间状态
- Token 窗口管理

长期记忆
(Long-term)

- 历史交互摘要
- 用户偏好画像
- 学习到的模式
- 向量化存储

工作记忆
(Working)

- 当前任务状态
- 推理中间结果
- 待验证假设

情景记忆
(Episodic)

- 具体事件记录
- 成功/失败案例
- 问题解决路径

语义记忆
(Semantic)

- 领域知识
- 事实与概念
- 知识图谱

程序记忆
(Procedural)

- 技能/技巧
- 标准操作流程
- 最佳实践模板

记忆管理策略：
- 遗忘机制：基于时间衰减 + 重要性加权
- 整合机制：周期性将短期记忆压缩为长期摘要
- 检索机制：语义相似性 + 时间相关性 + 任务相关性

1.4.9 Agent 生命周期

状态转换触发条件：
- Create → Init: Agent 定义加载完成
- Init → Run: 资源准备就绪，收到任务
- Run → Pause: 等待用户确认/外部依赖
- Pause → Run: 依赖满足/用户确认
- Run → Terminate: 任务完成/超时/错误

1.4.10 Agent Ops 运维规范　基于 Google Agents 白皮书，Agent Ops 是管理 Agentic 系统不可预测性的结构化方法：

Agent Ops 运维规范

Agent Ops 是 DevOps 和 MLOps 的自然演进，专为 AI Agent 独特挑战设计
将不可预测性从负债转变为可管理、可测量、可靠的特性

核心挑战：从确定性软件到概率性 Agent

传统软件测试：assert output == expected
Agent 测试：响应是概率性的，无法简单断言
解决方案：用 LM 评估"质量"——响应是否完成任务、语气是否适当

关键绩效指标 (KPIs) - 像 A/B 实验一样检测成功

业务指标：
- 目标完成率 (Goal Completion Rate)
- 用户满意度得分 (User Satisfaction Score)
- 任务延迟 (Task Latency)
- 每次交互运营成本 (Cost per Interaction)
- 收入/转化率/客户保留率影响

技术指标：
- 工具调用成功率
- 推理链完整性
- 上下文利用效率
- 幻觉率 (Hallucination Rate)
- 错误恢复率

可观测性三支柱

1. 追踪 (Tracing)
   - 完整推理轨迹：模型内部独白 → 工具选择 → 参数生成 → 结果观察
   - 跨 Agent 调用链追踪
   - 每个决策点的上下文快照

2. 日志 (Logging)
   - 结构化日志：思考过程、行动、观察分离记录
   - 异常检测：偏离预期执行路径的情况
   - 审计日志：敏感操作记录

3. 指标 (Metrics)
   - 实时仪表盘：任务状态、成功率、延迟分布
   - 告警规则：异常阈值触发
   - 趋势分析：长期性能变化

持续评估与 CI/CD

Build	Test	Evaluate	Deploy
构建	测试	评估	部署
提示+工具	单元测试	LM 评估	金丝雀发布
配置更新	集成测试	业务指标	A/B 测试
	对抗测试	人工审查	渐进式上线

模型升级策略：
- AI 领域快速进化，"一劳永逸"心态不可持续
- 投资灵活的操作框架，持续根据业务指标评估新模型
- 降低升级风险，无需彻底的架构改革

1.4.11 多 Agent 设计模式

多 Agent 设计模式 (Design Patterns)

1. 协调器模式 (Coordinator Pattern)

Coordinator
"经理 Agent"

Researcher Writer Programmer
研究员 作者 程序员

适用场景：动态或非线性任务
工作方式：协调者分析请求 → 分割任务 → 路由到专家 → 聚合响应

2. 顺序模式 (Sequential/Pipeline Pattern)

Agent A Agent B Agent C Agent D
需求分析 架构设计 代码实现 测试验证

上一个输出 = 下一个输入

适用场景：线性工作流，数字装配线

3. 迭代优化模式 (Iterative Refinement Pattern)

Generator Critic
生成器 批评家

创建内容 反馈意见 评估质量

Final Output 达到质量标准时输出

适用场景：需要高质量输出的创作任务

4. 人在环路模式 (Human-in-the-Loop / HITL Pattern)


```
ask_for_confirmation()
ask_for_date_input()
```

适用场景：高风险任务，需要人类审批的关键决策
实现方式：短信、邮件、数据库任务队列

5. 专家团队模式 (Expert Team Pattern)

不是构建一个"超级 Agent"，而是构建一个"专家团队"
模仿人类组织的劳动分工

优势：
- 每个 Agent 更简单、更专注
- 更容易构建、测试和维护
- 动态或长期运行的业务流程理想选择

模型路由策略：
- 前沿模型(如 Gemini 2.5 Pro)用于初步规划和复杂推理
- 轻量模型(如 Gemini 2.5 Flash)用于分类、总结等简单任务
- 优化性能和成本的关键策略

1.4.12 核心概念总结

Google Agents 白皮书核心概念总结

Agent 本质定义：

```
Agent = 模型 + 工具 + 编排层 + 运行时服务
      = LM 在一个带有工具的循环中以完成一个目标
```

Agent 本质上是一个致力于"上下文窗口策展艺术"的系统
不懈的循环：组装上下文 → 提示模型 → 观察结果 → 重新组装上下文

开发者角色转变：

传统开发者 = "砌砖工"
精确定义每一个逻辑步骤

Agent 开发者 = "导演"
设置场景（指导性说明和提示）
选择演员（工具和 API）
提供背景（数据）
引导自主"演员"交付预期表现

系统提示 = Agent 的"宪法"：
- 定义身份和人设
- 提供约束条件和期望输出模式
- 设置交战规则和语气
- 明确何时/为何应使用工具
- 包含示例场景（少样本学习）

核心洞察：
- LM 最大优势（灵活性）也是最大难题
- "无所不能"使其很难可靠地做好一件特定的事
- 全面的评估和评测通常比初始提示的影响更重要
- Agent 是语言模型在软件中变得实用的自然进化产物

1.5 Oh-My-OpenCode 实战哲学

整合 oh-my-opencode 项目的核心设计理念和实战经验

1.5.1 核心哲学：规划与执行分离

Oh-My-OpenCode 核心哲学：规划与执行分离

传统 AI Agent 的问题：
- 规划和执行混在一起
- 上下文污染和目标漂移
- 产出低质量代码（AI slop）

Oh-My-OpenCode 的解决方案：

规划阶段 (Planning Phase)

用户请求

咨询

Prometheus
规划者
(只读模式)

Metis
咨询顾问
(需求分析)

高精度模式

Momus
审核官

无情审核，直到计划完美

.sisyphus/plans/{name}.md
包含所有 TODO 的单一计划文件

/start-work

执行阶段 (Execution Phase)

boulder.json 状态管理，支持跨会话恢复

Atlas
编排者
(Extended
Thinking)

Oracle (GPT-5.2) 设计、调试
Librarian 文档查阅、开源检索
Explore (Haiku) 极速代码 Grep
Frontend Engineer 前端开发
Multimodal Looker PDF/图像分析

核心原则：

Prometheus 永远只读：只能创建/修改 .sisyphus/ 目录下的 Markdown
单计划原则：无论任务多大，所有 TODO 都在一个 .md 文件中
主动委派：Atlas 不亲自动手，而是委派给专业 Agent
Boulder 恢复：会话中断后可自动恢复进度

1.5.2 Category + Skill 路由系统　oh-my-opencode 的智能任务路由机制：

Category + Skill 路由系统

用户请求

Category 分类器

coding	search	frontend	docs	planning
代码开发	信息检索	前端开发	文档查阅	战略规划

Sisyphus	Oracle	Frontend	Librarian	Prometheus
(Opus 4.5)	(GPT-5.2)	(Gemini 3)	(GLM-4.7)	(Opus 4.5)

Skill 技能注入

Skills = 动态注入的专业知识片段

SKILL.md 定义：

```yaml
---
name: react-best-practices
triggers:
  - "*.tsx"
  - "*.jsx"
```

```
category: frontend
---
# React 最佳实践
- 使用函数组件和 Hooks
- 避免 prop drilling，使用 Context
- ...
```

触发方式：
- 文件名匹配（*.tsx, *.py, etc.）
- 关键词匹配（refactor, debug, test, etc.）
- Category 自动关联

Bypass 机制（简单任务跳过规划）：

- 请求长度 < 100 字符 → 直接执行
- 单文件操作 → 直接执行
- 有明确 Category 匹配 → 直接路由

1.5.3 专业 Agent 矩阵　oh-my-opencode 的 10 个专业化 Agent：

Oh-My-OpenCode Agent 矩阵

主力 Agent

Sisyphus (Claude Opus 4.5, temp=0.1)

- 主编排者，"像你一样编码"
- SF Bay Area 工程师人设
- 全工具访问权限
- Fallback: kimi-k2.5 → glm-4.7 → gpt-5.2-codex

Atlas (Claude Sonnet 4.5, Extended Thinking 32k)

- 计划执行器，持有 TODO 列表
- 不直接写代码，主动委派给专业 Agent
- 永不停止直到任务完成

规划 Agent

Prometheus (Claude Opus 4.5, 只读)

- 战略规划师，Interview/Consultant 模式
- 只能写 .sisyphus/ 目录下的 Markdown
- 永不写代码，专注"怎么做"

Metis (Claude Opus 4.5, temp=0.3)

- 预规划分析，Gap 检测
- 识别隐藏意图，防止过度工程

Momus (GPT-5.2, temp=0.1)

- 无情的计划审核官
- 拒绝并要求修改直到完美

专业 Agent

Oracle (GPT-5.2)
- 战略顾问、调试专家
- 工具限制：无 write, edit, task, delegate_task

Librarian (GLM-4.7)
- 文档查阅、GitHub 搜索、开源实现检索
- 使用 Context7 MCP 获取官方文档

Explore (Claude Haiku 4.5)
- 极速上下文感知 Grep
- 代码库快速探索

Frontend Engineer (Gemini 3 Pro)
- 前端 UI/UX 开发
- 专攻 React/Vue/CSS

Multimodal Looker (Gemini 3 Flash)
- PDF/图像分析
- 只允许 read 工具

Sisyphus-Junior (Claude Sonnet 4.5)

- 委派任务执行器
- Category 派生，无法再委派

关键设计：

温度控制：代码 Agent 一律 temp 0.1
工具限制：按角色精确控制可用工具
思考预算：关键 Agent 使用 Extended Thinking (32k)
模型回退：主模型不可用时自动切换备选
永不信任：NEVER trust "I'm done" - 必须验证输出

1.5.4 重规划触发机制 基于 oh-my-opencode 的 AIME 动态规划设计：

重规划触发机制

触发条件：

事件触发

- onTaskFailed: { maxRetries: 2 }
  任务失败超过阈值 → 触发重规划
- onNewEvidence: { significanceLevel: "high" }
  发现重要新信息 → 触发重规划
- onDependencyBlocked: true
  依赖任务阻塞 → 触发重规划

阈值触发

- progressStalled: { minutes: 5 }
  5 分钟无进展 → 触发重规划
- tokenBudgetLow: { remainingPercent: 0.2 }
  Token 预算剩余 < 20% → 触发策略调整

频率控制（防止死循环）：

```
FrequencyControl:
  maxReplansPerPlan: 3        # 单计划最大重规划次数
  cooldownSeconds: 60         # 两次重规划最小间隔
  escalationThreshold: 3      # 超过此阈值请求人工干预
```

评估决策流程：

执行状态

检查失败任务数量
failedCount >= 2?

YES

检查高重要性证据
criticalEvidence?

添加原因：
"N tasks failed"

YES

检查阻塞任务
blockedTasks > 0?

添加原因：
"Critical evid"

返回：{ shouldReplan: reasons.length > 0,
reasons: [...], priority: calculated }

1.5.5 UltraWork 魔法词

UltraWork 魔法词

用法：在提示中包含 "ultrawork" 或 "ulw"

效果：

- 并行 Agent 执行
- 后台任务自动运行
- 深度探索模式
- 不懈执行直到完成

适用场景：
- 复杂任务，懒得解释上下文
- 让 Agent 自己搞定一切

决策流程：

是快速修复或简单任务?
YES → 直接提示
NO → 解释完整上下文很麻烦?
YES → 输入 "ulw"，让 Agent 自己搞定
NO → 需要精确可验证的执行?
YES → 使用 @plan + /start-work
NO → 用 "ulw"

---

2. 架构概览

2.1 分层架构

客户端层 (Client Layer)

CLI	TUI	Web	IDE	API
Terminal	@opentui	React	VS Code	REST

网关层 (Gateway Layer)

Hono HTTP Server + WebSocket

Session API | Agent API | Tool API | Workflow API | Swarm API

编排层 (Orchestration Layer)

Session Manager | Workflow Engine | Swarm Coordinator | State Machine

Event Bus | Agent Router | Message Queue

核心层 (Core Layer)

Agent Registry

Code Agents | Analyst Agents | Writer Agents | QA Agents | Dynamic Agents

Tool Registry

File Tools | Code Tools | Doc Tools | Swarm Tools | Data Tools

Swarm Runtime

Agent Factory | Message Broker | Topology Graph | Wake Manager | Runner Pool

基础设施层（Infrastructure Layer）

Provider	MCP	Storage	Auth	Plugin
（AI模型）	（扩展）	（SQLite）	（认证）	（插件）

2.2 核心概念模型

```
// 核心实体关系
interface AgentPlatform {
  // 注册表
  agents: AgentRegistry        // Agent 注册表
  tools: ToolRegistry          // Tool 注册表
  workflows: WorkflowRegistry // Workflow 注册表
  templates: TemplateRegistry // 场景模板注册表

  // Swarm 蜂群运行时
  swarm: SwarmRuntime          // 蜂群运行时
  topology: TopologyGraph      // 动态拓扑图
  messageHub: MessageHub       // 消息中心

  // 知识接地系统（Google Agents）
  grounding: GroundingSystem  // 知识接地
  memory: MemorySystem         // 记忆系统
  orchestrator: Orchestrator  // 编排引擎

  // 运行时
  sessions: SessionManager     // 会话管理
  executor: WorkflowExecutor  // 工作流执行器
  scheduler: TaskScheduler     // 任务调度器

  // 基础设施
  providers: ProviderRegistry // AI 模型提供商
  plugins: PluginManager       // 插件管理器
  config: ConfigManager        // 配置管理器
}

// Swarm 蜂群核心接口
interface SwarmRuntime {
  // 动态创建 Agent
  createAgent(spec: DynamicAgentSpec): Promise<AgentInstance>

  // 发送消息到任意 Agent
```

```
  sendMessage(to: AgentID, message: SwarmMessage): Promise<void>

  // 唤醒 Agent
  wakeAgent(agentId: AgentID): Promise<void>

  // 获取 Agent 拓扑
  getTopology(): TopologyGraph

  // 销毁 Agent
  destroyAgent(agentId: AgentID): Promise<void>
}

// Grounding 知识接地接口（Google Agents）
interface GroundingSystem {
  // 数据源管理
  registerSource(source: GroundingSource): Promise<void>

  // 知识检索
  retrieve(query: string, options?: RetrievalOptions): Promise<GroundingResult[]>

  // RAG 增强生成
  augment(prompt: string, context: GroundingResult[]): string
}

interface GroundingSource {
  id: string
  type: 'web_search' | 'data_store' | 'code_repo' | 'api' | 'database'
  config: Record<string, unknown>

  // 检索方法
  search(query: string): Promise<GroundingResult[]>
}

// Memory 记忆系统接口（Google Agents）
interface MemorySystem {
  // 短期记忆
  shortTerm: ShortTermMemory

  // 长期记忆
  longTerm: LongTermMemory

  // 工作记忆
  working: WorkingMemory

  // 情景记忆
  episodic: EpisodicMemory

  // 记忆管理
```

```
  consolidate(): Promise<void>  // 短期 → 长期整合
  forget(criteria: ForgetCriteria): Promise<void>  // 遗忘
}

interface ShortTermMemory {
  // 当前会话上下文
  getContext(sessionId: string): Promise<Message[]>

  // 添加到上下文
  append(sessionId: string, message: Message): Promise<void>

  // 窗口管理（Token 限制）
  truncate(sessionId: string, maxTokens: number): Promise<void>
}

interface LongTermMemory {
  // 存储记忆
  store(memory: MemoryEntry): Promise<void>

  // 语义检索
  recall(query: string, options?: RecallOptions): Promise<MemoryEntry[]>

  // 用户偏好
  getUserPreferences(userId: string): Promise<UserPreferences>
}

interface WorkingMemory {
  // 当前任务状态
  getTaskState(taskId: string): Promise<TaskState>

  // 推理中间结果
  setIntermediateResult(key: string, value: unknown): void
  getIntermediateResult(key: string): unknown
}

interface EpisodicMemory {
  // 记录事件
  recordEpisode(episode: Episode): Promise<void>

  // 检索相似经历
  findSimilarEpisodes(context: string): Promise<Episode[]>

  // 成功/失败案例
  getSuccessPatterns(taskType: string): Promise<Episode[]>
}

// Orchestrator 编排引擎（Google Agents）
interface Orchestrator {
```

```
  // 编排模式
  patterns: {
    react: ReActOrchestrator        // ReAct 模式
    cot: ChainOfThoughtOrchestrator // 思维链模式
    tot: TreeOfThoughtOrchestrator  // 思维树模式
    selfRefine: SelfRefineOrchestrator // 自我迭代模式
  }

  // 执行编排
  execute(pattern: OrchestrationPattern, task: Task): Promise<Result>
}

interface ReActOrchestrator {
  // Thought → Action → Observation 循环
  step(thought: string): Promise<{ action: Action; observation: string }>

  // 最大迭代次数
  maxIterations: number
}

interface ChainOfThoughtOrchestrator {
  // 生成推理链
  generateChain(problem: string): Promise<ThoughtStep[]>

  // 执行推理链
  executeChain(steps: ThoughtStep[]): Promise<string>
}

interface TreeOfThoughtOrchestrator {
  // 生成思维树
  generateTree(problem: string, depth: number): Promise<ThoughtTree>

  // 搜索最优路径
  search(tree: ThoughtTree, strategy: 'bfs' | 'dfs'): Promise<ThoughtPath>

  // 评估路径
  evaluate(path: ThoughtPath): Promise<number>
}
```

---

3. 核心抽象层设计

3.1 Domain 领域抽象

将不同场景抽象为 Domain（领域），每个 Domain 包含特定的 Agents、Tools、Workflows：

```
// domain/types.ts
export interface Domain {
```

```
  id: string                          // 领域唯一标识
  name: string                        // 显示名称
  description: string                 // 描述
  icon?: string                       // 图标

  // 领域资源
  agents: AgentDefinition[]           // 该领域的 Agents
  tools: ToolDefinition[]             // 该领域的 Tools
  workflows: WorkflowDefinition[]     // 该领域的 Workflows
  skills: SkillDefinition[]           // 该领域的 Skills

  // 领域配置
  defaultAgent?: string               // 默认 Agent
  permissions?: PermissionRuleset     // 权限规则
  settings?: Record<string, unknown>  // 领域设置
}

// 预置领域
export const BUILTIN_DOMAINS = {
  coding: "coding",             // 编码开发
  analysis: "analysis",         // 需求分析
  documentation: "documentation", // 文档撰写
  testing: "testing",           // 测试质量
  operations: "operations",     // 运维部署
  management: "management",     // 项目管理
} as const
```

3.2 Capability 能力抽象

定义通用能力接口，供 Agents 和 Tools 实现：

```
// capability/types.ts
export interface Capability {
  id: string
  name: string
  category: CapabilityCategory

  // 能力检查
  check(): Promise<CapabilityStatus>

  // 依赖声明
  dependencies?: string[]

  // 配置 schema
  configSchema?: z.ZodType
}

export type CapabilityCategory =
  | "file"          // 文件操作
```

```
  | "code"          // 代码分析
  | "document"      // 文档处理
  | "api"           // API 调用
  | "data"          // 数据处理
  | "search"        // 搜索查询
  | "reasoning"     // 推理规划
  | "communication" // 通信协作
```

3.3 Context 上下文抽象

统一的执行上下文，携带所有必要信息：

```
// context/types.ts
export interface ExecutionContext {
  // 基础信息
  sessionID: string
  domain: string
  workspace: WorkspaceInfo

  // 用户信息
  user?: UserInfo
  permissions: PermissionSet

  // 执行状态
  state: ExecutionState
  history: Message[]
  artifacts: Artifact[]

  // 工具访问
  tools: ToolAccessor
  agents: AgentAccessor

  // 事件发布
  emit(event: Event): Promise<void>

  // 权限请求
  ask(request: PermissionRequest): Promise<void>

  // 进度报告
  progress(info: ProgressInfo): void

  // Grounding 知识接地
  grounding: GroundingAccessor

  // Memory 记忆访问
  memory: MemoryAccessor
}
```

3.4 Grounding 知识接地系统

基于 Google Agents 白皮书的 Grounding 机制，实现知识增强：

```typescript
// grounding/types.ts

/**
 * Grounding 知识接地系统
 * 解决模型知识时效性和幻觉问题
 */
export interface GroundingService {
  // 数据源
  sources: Map<string, GroundingSource>

  // 注册数据源
  register(source: GroundingSource): void

  // 检索知识
  retrieve(query: RetrievalQuery): Promise<GroundingResult[]>

  // RAG 增强
  augment(prompt: string, results: GroundingResult[]): AugmentedPrompt
}

/**
 * Grounding 数据源类型
 */
export type GroundingSourceType =
  | "web_search"      // 网络搜索（实时信息）
  | "vector_store"    // 向量数据库（语义检索）
  | "code_repo"       // 代码仓库（代码检索）
  | "api_endpoint"    // API 端点（实时数据）
  | "database"        // 数据库（结构化查询）
  | "knowledge_graph" // 知识图谱（关系推理）
  | "document_index" // 文档索引（全文检索）

/**
 * Grounding 数据源定义
 */
export interface GroundingSource {
  id: string
  type: GroundingSourceType
  name: string
  description: string

  // 数据源配置
  config: GroundingSourceConfig

  // 检索实现
```

```
  search(query: string, options?: SearchOptions): Promise<GroundingResult[]>

  // 健康检查
  healthCheck(): Promise<HealthStatus>
}

/**
 * 各类型数据源配置
 */
export type GroundingSourceConfig =
  | WebSearchConfig
  | VectorStoreConfig
  | CodeRepoConfig
  | ApiEndpointConfig
  | DatabaseConfig
  | KnowledgeGraphConfig

// 向量存储配置
interface VectorStoreConfig {
  type: "vector_store"
  provider: "pinecone" | "weaviate" | "chroma" | "milvus" | "qdrant"
  collection: string
  embeddingModel: string
  dimensions: number
  topK: number
  minScore: number
}

// 代码仓库配置
interface CodeRepoConfig {
  type: "code_repo"
  provider: "local" | "github" | "gitlab"
  path?: string
  repo?: string
  branch?: string
  includePatterns: string[]
  excludePatterns: string[]
}

// 知识图谱配置
interface KnowledgeGraphConfig {
  type: "knowledge_graph"
  provider: "neo4j" | "dgraph" | "amazon_neptune"
  endpoint: string
  schema: GraphSchema
}

/**
```

```
 * 检索查询
 */
export interface RetrievalQuery {
  text: string                    // 查询文本
  sources?: string[]              // 指定数据源
  filters?: RetrievalFilter[]     // 过滤条件
  limit?: number                  // 结果数量
  minRelevance?: number           // 最小相关度
  rerank?: boolean                // 是否重排序
}

/**
 * 检索结果
 */
export interface GroundingResult {
  source: string                  // 数据源 ID
  content: string                 // 内容
  metadata: Record<string, unknown>
  relevance: number               // 相关度分数
  citation?: Citation             // 引用信息
}

/**
 * RAG 增强 Prompt
 */
export interface AugmentedPrompt {
  original: string                // 原始 Prompt
  context: string                 // 检索上下文
  augmented: string               // 增强后的 Prompt
  citations: Citation[]           // 引用列表
}
```

3.4.1 Grounding 使用示例

```
// 使用 Grounding 增强回答质量
async function answerWithGrounding(
  question: string,
  grounding: GroundingService,
  model: LanguageModel
): Promise<GroundedAnswer> {
  // 1. 检索相关知识
  const results = await grounding.retrieve({
    text: question,
    sources: ["code_repo", "vector_store", "web_search"],
    limit: 10,
    minRelevance: 0.7,
    rerank: true
  })
```

```
  // 2. 构建增强 Prompt
  const augmented = grounding.augment(question, results)

  // 3. 生成回答
  const response = await model.generate({
    prompt: augmented.augmented,
    systemPrompt: `你是一个专业助手，基于提供的上下文回答问题。
                   务必在回答中引用信息来源。
                   如果上下文不足以回答，请明确说明。`
  })

  return {
    answer: response.text,
    citations: augmented.citations,
    confidence: calculateConfidence(results)
  }
}
```

3.5 Memory 记忆系统

基于 Google Agents 白皮书的记忆架构，实现多层次记忆管理：

```
// memory/types.ts

/**
 * 记忆系统完整接口
 */
export interface MemoryService {
  // 记忆层
  shortTerm: ShortTermMemoryStore   // 短期记忆
  longTerm: LongTermMemoryStore     // 长期记忆
  working: WorkingMemoryStore       // 工作记忆
  episodic: EpisodicMemoryStore     // 情景记忆
  semantic: SemanticMemoryStore     // 语义记忆
  procedural: ProceduralMemoryStore // 程序记忆

  // 记忆管理
  manager: MemoryManager
}

/**
 * 短期记忆存储
 * 管理当前会话的上下文窗口
 */
export interface ShortTermMemoryStore {
  // 获取会话上下文
  getContext(sessionId: string): Promise<ConversationContext>
```

```typescript
  // 追加消息
  append(sessionId: string, message: Message): Promise<void>

  // Token 窗口管理
  getTokenCount(sessionId: string): Promise<number>
  truncate(sessionId: string, maxTokens: number): Promise<void>

  // 滑动窗口
  getWindow(sessionId: string, windowSize: number): Promise<Message[]>

  // 摘要压缩
  summarize(sessionId: string): Promise<string>
}

/**
 * 长期记忆存储
 * 持久化的用户知识和偏好
 */
export interface LongTermMemoryStore {
  // 存储记忆
  store(entry: LongTermMemoryEntry): Promise<void>

  // 语义检索
  recall(query: string, options?: RecallOptions): Promise<LongTermMemoryEntry[]>

  // 用户画像
  getUserProfile(userId: string): Promise<UserProfile>
  updateUserProfile(userId: string, updates: Partial<UserProfile>): Promise<void>

  // 偏好学习
  learnPreference(userId: string, preference: Preference): Promise<void>
  getPreferences(userId: string): Promise<Preference[]>
}

export interface LongTermMemoryEntry {
  id: string
  userId: string
  content: string
  embedding: number[]           // 向量嵌入
  type: 'fact' | 'preference' | 'skill' | 'episode'
  importance: number            // 重要性权重
  accessCount: number           // 访问次数
  lastAccess: Date
  createdAt: Date
  metadata: Record<string, unknown>
}

/**
```

```typescript
 * 工作记忆存储
 * 当前任务的临时状态
 */
export interface WorkingMemoryStore {
  // 任务状态
  getTaskState(taskId: string): Promise<TaskState>
  setTaskState(taskId: string, state: TaskState): Promise<void>

  // 推理中间结果
  setVariable(taskId: string, key: string, value: unknown): void
  getVariable(taskId: string, key: string): unknown

  // 假设管理
  addHypothesis(taskId: string, hypothesis: Hypothesis): void
  getHypotheses(taskId: string): Hypothesis[]
  validateHypothesis(taskId: string, hypothesisId: string, result: boolean): void

  // 注意力焦点
  setFocus(taskId: string, focus: AttentionFocus): void
  getFocus(taskId: string): AttentionFocus
}

/**
 * 情景记忆存储
 * 具体事件和经历的记录
 */
export interface EpisodicMemoryStore {
  // 记录事件
  recordEpisode(episode: Episode): Promise<void>

  // 检索相似经历
  findSimilar(context: string, limit?: number): Promise<Episode[]>

  // 成功模式
  getSuccessPatterns(taskType: string): Promise<Episode[]>

  // 失败案例
  getFailureCases(taskType: string): Promise<Episode[]>

  // 时间线查询
  getTimeline(userId: string, range: DateRange): Promise<Episode[]>
}

export interface Episode {
  id: string
  userId: string
  sessionId: string
  taskType: string
```

```
  context: string                // 情境描述
  actions: Action[]              // 采取的行动
  outcome: 'success' | 'failure' | 'partial'
  lessons: string[]              // 经验教训
  embedding: number[]            // 向量嵌入
  timestamp: Date
}

/**
 * 语义记忆存储
 * 领域知识和概念
 */
export interface SemanticMemoryStore {
  // 知识存储
  storeKnowledge(knowledge: KnowledgeEntry): Promise<void>

  // 知识检索
  queryKnowledge(query: string): Promise<KnowledgeEntry[]>

  // 概念关联
  getRelatedConcepts(concept: string): Promise<ConceptNode[]>

  // 知识图谱查询
  graphQuery(query: GraphQuery): Promise<GraphResult>
}

/**
 * 程序记忆存储
 * 技能和操作流程
 */
export interface ProceduralMemoryStore {
  // 注册技能
  registerSkill(skill: SkillProcedure): Promise<void>

  // 获取技能
  getSkill(skillId: string): Promise<SkillProcedure>

  // 匹配最佳流程
  matchProcedure(taskDescription: string): Promise<SkillProcedure[]>

  // 更新技能（学习）
  updateSkill(skillId: string, updates: SkillUpdate): Promise<void>
}

export interface SkillProcedure {
  id: string
  name: string
  description: string
```

```
  triggers: string[]              // 触发条件
  steps: ProcedureStep[]          // 操作步骤
  successRate: number             // 成功率
  usageCount: number              // 使用次数
  lastUsed: Date
}

/**
 * 记忆管理器
 * 负责记忆的生命周期管理
 */
export interface MemoryManager {
  // 记忆整合（短期 → 长期）
  consolidate(sessionId: string): Promise<ConsolidationResult>

  // 记忆遗忘
  forget(criteria: ForgetCriteria): Promise<ForgetResult>

  // 记忆优化
  optimize(): Promise<OptimizationResult>

  // 重要性评估
  evaluateImportance(entry: MemoryEntry): number

  // 记忆检索策略
  setRetrievalStrategy(strategy: RetrievalStrategy): void
}

export interface ForgetCriteria {
  // 时间衰减
  olderThan?: Duration

  // 重要性阈值
  importanceBelow?: number

  // 访问频率
  accessCountBelow?: number

  // 类型过滤
  types?: MemoryType[]
}

export interface RetrievalStrategy {
  // 权重配置
  weights: {
    semanticSimilarity: number    // 语义相似度权重
    temporalRecency: number       // 时间近度权重
    accessFrequency: number       // 访问频率权重
```

```
    importance: number              // 重要性权重
    taskRelevance: number           // 任务相关性权重
  }

  // 检索参数
  topK: number
  minScore: number
  diversify: boolean                // 结果多样化
}
```

3.5.1 Memory 使用示例

```
// 带记忆的 Agent 对话
async function conversationWithMemory(
  userId: string,
  sessionId: string,
  message: string,
  memory: MemoryService,
  model: LanguageModel
): Promise<Response> {
  // 1. 获取短期记忆（当前会话）
  const recentContext = await memory.shortTerm.getWindow(sessionId, 10)

  // 2. 检索长期记忆（相关知识）
  const relevantMemories = await memory.longTerm.recall(message, {
    userId,
    limit: 5,
    minRelevance: 0.6
  })

  // 3. 获取用户偏好
  const preferences = await memory.longTerm.getPreferences(userId)

  // 4. 检索相似经历
  const similarEpisodes = await memory.episodic.findSimilar(message, 3)

  // 5. 构建上下文
  const context = buildContextWithMemory({
    recentContext,
    relevantMemories,
    preferences,
    similarEpisodes
  })

  // 6. 生成回答
  const response = await model.generate({
    systemPrompt: context.systemPrompt,
    messages: context.messages
```

```
  })

  // 7. 更新记忆
  await memory.shortTerm.append(sessionId, {
    role: 'user',
    content: message
  })
  await memory.shortTerm.append(sessionId, {
    role: 'assistant',
    content: response.text
  })

  // 8. 学习偏好（如果有新发现）
  if (response.detectedPreferences) {
    for (const pref of response.detectedPreferences) {
      await memory.longTerm.learnPreference(userId, pref)
    }
  }

  return response
}
```

---

4. Agent 注册系统

4.1 Agent 定义规范

```
// agent/types.ts
export interface AgentDefinition {
  // 基础信息
  id: string                          // 唯一标识（如 "requirement-analyst"）
  name: string                        // 显示名称
  description: string                 // 描述
  version: string                     // 版本号

  // 分类信息
  domain: string                      // 所属领域
  category: AgentCategory             // 类别
  tags: string[]                      // 标签

  // 能力配置
  capabilities: string[]              // 支持的能力
  tools: ToolBinding[]                // 工具绑定
  skills: string[]                    // 技能引用

  // 运行配置
  mode: "primary" | "subagent" | "all"
  model?: ModelSpec                   // 指定模型
```

```
  temperature?: number
  maxSteps?: number

  // Prompt 模板
  systemPrompt: string | PromptTemplate

  // 权限规则
  permissions: PermissionRuleset

  // 元数据
  metadata?: Record<string, unknown>
}

// Agent 类别
export type AgentCategory =
  | "orchestrator"   // 编排者 - 负责任务分解和协调
  | "specialist"     // 专家 - 特定领域深度能力
  | "advisor"        // 顾问 - 提供建议和审查
  | "executor"       // 执行者 - 具体任务执行
  | "utility"        // 工具型 - 辅助功能

// 工具绑定
export interface ToolBinding {
  tool: string                          // 工具 ID
  alias?: string                        // 别名
  config?: Record<string, unknown>      // 工具配置
  required?: boolean                    // 是否必需
}
```

4.2 Agent 注册表

```
// agent/registry.ts
export class AgentRegistry {
  private agents = new Map<string, AgentDefinition>()
  private factories = new Map<string, AgentFactory>()

  // 注册 Agent
  register(definition: AgentDefinition): void {
    this.validate(definition)
    this.agents.set(definition.id, definition)
    Bus.publish(AgentEvent.Registered, { agent: definition.id })
  }

  // 批量注册（从配置）
  registerFromConfig(config: AgentConfig[]): void {
    for (const cfg of config) {
      this.register(this.parseConfig(cfg))
    }
```

```
  }

  // 从目录扫描注册
  async scanAndRegister(dirs: string[]): Promise<void> {
    for (const dir of dirs) {
      const files = await glob(`${dir}/**/*.agent.{json,yaml,md}`)
      for (const file of files) {
        const def = await this.loadAgentFile(file)
        this.register(def)
      }
    }
  }

  // 获取 Agent
  get(id: string): AgentDefinition | undefined {
    return this.agents.get(id)
  }

  // 按领域查询
  byDomain(domain: string): AgentDefinition[] {
    return [...this.agents.values()].filter(a => a.domain === domain)
  }

  // 按能力查询
  byCapability(capability: string): AgentDefinition[] {
    return [...this.agents.values()]
      .filter(a => a.capabilities.includes(capability))
  }

  // 实例化 Agent
  async instantiate(id: string, ctx: ExecutionContext): Promise<Agent> {
    const def = this.get(id)
    if (!def) throw new Error(`Agent not found: ${id}`)

    const factory = this.factories.get(def.category) ?? defaultFactory
    return factory.create(def, ctx)
  }
}
```

4.3 Agent 定义方式

方式一：JSON/YAML 配置

```yaml
# agents/requirement-analyst.agent.yaml
id: requirement-analyst
name: 需求分析师
description: 专注于用户需求分析、用例提取、边界条件识别
version: "1.0.0"
```

```yaml
domain: analysis
category: specialist
tags: [requirement, analysis, use-case]

capabilities:
  - document.read
  - document.write
  - reasoning.analysis

tools:
  - tool: read
  - tool: write
  - tool: search
  - tool: ask_user
    alias: clarify
    config:
      defaultPrefix: " 关于需求，我想确认："

model:
  provider: anthropic
  model: claude-sonnet-4-20250514

systemPrompt: |
  你是一位资深的需求分析师，擅长：

  ## 核心能力
  - 从用户描述中提取结构化需求
  - 识别功能性需求和非功能性需求
  - 发现隐含需求和边界条件
  - 评估需求的完整性和一致性

  ## 工作流程
  1. 理解用户的业务背景
  2. 提取核心用例（Use Cases）
  3. 分析每个用例的前置条件、后置条件、异常流程
  4. 输出结构化的需求文档

permissions:
  read: allow
  edit: ask
  bash: deny
```

方式二：Markdown 配置

```yaml
<!-- agents/requirement-analyst.agent.md -->
---
id: requirement-analyst
name: 需求分析师
```

```
domain: analysis
category: specialist
capabilities:
  - document.read
  - document.write
  - reasoning.analysis
tools:
  - read
  - write
  - ask_user
---

# 需求分析师

你是一位资深的需求分析师，擅长从模糊的用户描述中提取清晰的需求。

## 分析框架

使用以下框架进行需求分析：

### 1. 用户故事 (User Stories)
- As a [角色], I want [功能], so that [价值]

### 2. 验收标准 (Acceptance Criteria)
- Given [前置条件], When [操作], Then [预期结果]

### 3. 边界条件
- 输入边界
- 并发场景
- 异常处理
```

方式三：TypeScript 代码

```typescript
// agents/requirement-analyst.agent.ts
import { defineAgent } from "@opencode-ai/agent-sdk"

export default defineAgent({
  id: "requirement-analyst",
  name: " 需求分析师",
  domain: "analysis",
  category: "specialist",

  capabilities: ["document.read", "document.write", "reasoning.analysis"],

  tools: [
    { tool: "read" },
    { tool: "write" },
    { tool: "ask_user", alias: "clarify" },
```

```
  ],

  // 动态 Prompt 构建
  systemPrompt: async (ctx) => {
    const projectInfo = await ctx.workspace.getProjectInfo()
    return `
      你是 ${projectInfo.name} 项目的需求分析师。

      ## 项目背景
      ${projectInfo.description}

      ## 已有需求文档
      ${await listRequirementDocs(ctx)}

      ## 你的任务
      分析用户需求，输出结构化文档。
    `
  },

  // 生命周期钩子
  hooks: {
    beforeExecute: async (ctx, input) => {
      // 预处理
    },
    afterExecute: async (ctx, output) => {
      // 后处理，如保存到知识库
    },
  },
})
```

---

5. Tool 注册系统

5.1 Tool 定义规范

```
// tool/types.ts
export interface ToolDefinition {
  // 基础信息
  id: string                          // 唯一标识
  name: string                        // 显示名称
  description: string                 // 描述（给 LLM 看）
  version: string                     // 版本

  // 分类信息
  domain?: string                     // 所属领域（可选）
  category: ToolCategory              // 工具类别
  tags: string[]                      // 标签
```

```
  // 参数定义
  parameters: z.ZodType                // Zod schema

  // 输出定义
  output?: z.ZodType                   // 输出 schema

  // 执行函数
  execute: ToolExecutor

  // 权限要求
  requiredPermissions?: string[]

  // UI 配置
  ui?: ToolUIConfig

  // 元数据
  metadata?: Record<string, unknown>
}

// 工具类别
export type ToolCategory =
  | "file"           // 文件操作
  | "code"           // 代码分析
  | "document"       // 文档处理
  | "search"         // 搜索查询
  | "api"            // API 调用
  | "data"           // 数据处理
  | "shell"          // 命令执行
  | "communication" // 通信协作
  | "utility"        // 通用工具

// 执行器类型
type ToolExecutor = (
  params: unknown,
  ctx: ToolContext
) => Promise<ToolResult>

// 工具上下文
interface ToolContext {
  sessionID: string
  messageID: string
  callID: string
  agent: string
  abort: AbortSignal

  // 权限请求
  ask(request: PermissionRequest): Promise<void>
```

```
  // 元数据设置
  metadata(data: Record<string, unknown>): void

  // 进度报告
  progress(info: ProgressInfo): void
}

// 工具结果
interface ToolResult {
  title: string
  output: string
  metadata?: Record<string, unknown>
  attachments?: Attachment[]
}
```

5.2 Tool 注册表

```
// tool/registry.ts
export class ToolRegistry {
  private tools = new Map<string, ToolDefinition>()
  private mcpTools = new Map<string, MCPToolAdapter>()

  // 注册工具
  register(definition: ToolDefinition): void {
    this.validate(definition)
    this.tools.set(definition.id, definition)
    Bus.publish(ToolEvent.Registered, { tool: definition.id })
  }

  // 从 MCP 服务器加载
  async loadFromMCP(server: MCPServer): Promise<void> {
    const tools = await server.listTools()
    for (const tool of tools) {
      const adapter = new MCPToolAdapter(tool, server)
      this.mcpTools.set(adapter.id, adapter)
    }
  }

  // 扫描目录注册
  async scanAndRegister(dirs: string[]): Promise<void> {
    for (const dir of dirs) {
      const files = await glob(`${dir}/**/*.tool.{ts,js}`)
      for (const file of files) {
        const module = await import(file)
        if (module.default) {
          this.register(module.default)
        }
      }
```

```
    }
  }

  // 获取工具
  get(id: string): ToolDefinition | undefined {
    return this.tools.get(id) ?? this.mcpTools.get(id)
  }

  // 按类别查询
  byCategory(category: ToolCategory): ToolDefinition[] {
    return [...this.tools.values()].filter(t => t.category === category)
  }

  // 获取 Agent 可用工具
  forAgent(agentDef: AgentDefinition): ToolDefinition[] {
    const toolIds = agentDef.tools.map(t => typeof t === "string" ? t : t.tool)
    return toolIds.map(id => this.get(id)).filter(Boolean)
  }

  // 转换为 AI SDK 格式
  toAISDKTools(ids: string[]): Record<string, CoreTool> {
    const result: Record<string, CoreTool> = {}
    for (const id of ids) {
      const def = this.get(id)
      if (def) {
        result[id] = this.convertToCoreTool(def)
      }
    }
    return result
  }
}
```

5.3 通用工具示例

文档生成工具

```
// tools/document/generate-doc.tool.ts
import { defineTool } from "@opencode-ai/tool-sdk"

export default defineTool({
  id: "generate_document",
  name: " 生成文档",
  description: " 根据模板和数据生成结构化文档",
  category: "document",

  parameters: z.object({
    template: z.enum([
      "prd",           // 产品需求文档
      "tech-design",   // 技术设计文档
```

```
      "api-doc",        // API 文档
      "user-guide",     // 用户指南
      "test-plan",      // 测试计划
      "release-note",   // 发布说明
    ]).describe(" 文档模板类型"),

    title: z.string().describe(" 文档标题"),
    content: z.record(z.string(), z.any()).describe(" 文档内容数据"),
    outputPath: z.string().optional().describe(" 输出路径"),
    format: z.enum(["markdown", "html", "pdf"]).default("markdown"),
  }),

  async execute(params, ctx) {
    const template = await loadTemplate(params.template)
    const rendered = await renderTemplate(template, params.content)

    if (params.outputPath) {
      await ctx.ask({
        permission: "edit",
        patterns: [params.outputPath],
        metadata: { template: params.template },
      })
      await writeFile(params.outputPath, rendered)
    }

    return {
      title: `Generated ${params.template}: ${params.title}`,
      output: rendered,
      metadata: {
        template: params.template,
        outputPath: params.outputPath,
      },
    }
  },
})
```

API 调用工具

```
// tools/api/http-request.tool.ts
export default defineTool({
  id: "http_request",
  name: "HTTP 请求",
  description: " 发送 HTTP 请求到外部 API",
  category: "api",

  parameters: z.object({
    url: z.string().url().describe(" 请求 URL"),
    method: z.enum(["GET", "POST", "PUT", "DELETE", "PATCH"]).default("GET"),
```

```
    headers: z.record(z.string()).optional(),
    body: z.any().optional(),
    timeout: z.number().default(30000),
  }),

  requiredPermissions: ["network"],

  async execute(params, ctx) {
    await ctx.ask({
      permission: "network",
      patterns: [new URL(params.url).hostname],
      metadata: { url: params.url, method: params.method },
    })

    const response = await fetch(params.url, {
      method: params.method,
      headers: params.headers,
      body: params.body ? JSON.stringify(params.body) : undefined,
      signal: AbortSignal.timeout(params.timeout),
    })

    const data = await response.text()

    return {
      title: `${params.method} ${params.url}`,
      output: data,
      metadata: {
        status: response.status,
        headers: Object.fromEntries(response.headers),
      },
    }
  },
})
```

数据分析工具

```
// tools/data/analyze-data.tool.ts
export default defineTool({
  id: "analyze_data",
  name: " 数据分析",
  description: " 对结构化数据进行统计分析",
  category: "data",

  parameters: z.object({
    source: z.union([
      z.object({ type: z.literal("file"), path: z.string() }),
      z.object({ type: z.literal("inline"), data: z.array(z.any()) }),
    ]).describe(" 数据源"),
```

```
    analysis: z.array(z.enum([
      "summary",      // 基本统计
      "distribution", // 分布分析
      "correlation",  // 相关性分析
      "trend",        // 趋势分析
      "anomaly",      // 异常检测
    ])).describe(" 分析类型"),

    groupBy: z.string().optional(),
    filters: z.record(z.any()).optional(),
  }),

  async execute(params, ctx) {
    const data = await loadData(params.source)
    const results: Record<string, unknown> = {}

    for (const type of params.analysis) {
      results[type] = await runAnalysis(type, data, params)
    }

    return {
      title: "Data Analysis Results",
      output: formatAnalysisResults(results),
      metadata: { rowCount: data.length, analysisTypes: params.analysis },
    }
  },
})
```

---

6. Workflow 编排系统

6.1 Workflow 定义规范

```
// workflow/types.ts
export interface WorkflowDefinition {
  id: string
  name: string
  description: string
  version: string

  // 触发条件
  triggers?: WorkflowTrigger[]

  // 输入参数
  inputs: z.ZodType

  // 输出定义
```

```
  outputs?: z.ZodType

  // 步骤定义
  steps: WorkflowStep[]

  // 错误处理
  onError?: ErrorHandler

  // 超时设置
  timeout?: number
}

// 工作流步骤
export type WorkflowStep =
  | AgentStep      // 调用 Agent
  | ToolStep       // 调用 Tool
  | ConditionStep  // 条件分支
  | ParallelStep   // 并行执行
  | LoopStep       // 循环执行
  | SubWorkflowStep // 子工作流
  | HumanStep      // 人工介入

// Agent 步骤
export interface AgentStep {
  type: "agent"
  id: string
  agent: string                          // Agent ID
  input: string | InputTemplate          // 输入模板
  output?: string                        // 输出变量名
  config?: Partial<AgentDefinition>      // 临时配置覆盖
}

// 工具步骤
export interface ToolStep {
  type: "tool"
  id: string
  tool: string                           // Tool ID
  params: Record<string, unknown>        // 参数
  output?: string                        // 输出变量名
}

// 条件分支
export interface ConditionStep {
  type: "condition"
  id: string
  condition: string                      // 表达式
  then: WorkflowStep[]
  else?: WorkflowStep[]
```

```
}

// 并行执行
export interface ParallelStep {
  type: "parallel"
  id: string
  branches: WorkflowStep[][]
  mergeStrategy?: "all" | "any" | "race"
}

// 人工介入
export interface HumanStep {
  type: "human"
  id: string
  prompt: string                    // 提示用户的问题
  options?: string[]                // 可选选项
  timeout?: number                  // 超时时间
  output?: string                   // 输出变量名
}
```

6.2 Workflow 示例

需求到开发完整流程

```yaml
# workflows/requirement-to-dev.workflow.yaml
id: requirement-to-dev
name: 需求到开发完整流程
description: 从需求分析到代码实现的完整工作流
version: "1.0.0"

inputs:
  type: object
  properties:
    requirement:
      type: string
      description: 原始需求描述
    priority:
      type: string
      enum: [high, medium, low]
      default: medium

steps:
  # 1. 需求分析
  - type: agent
    id: analyze
    agent: requirement-analyst
    input: |
      请分析以下需求:
```

```
    {{ inputs.requirement }}

    输出结构化的需求文档。
  output: requirementDoc

# 2. 人工确认
- type: human
  id: confirm_requirement
  prompt: |
    需求分析已完成，请确认：

    {{ steps.analyze.output }}

    是否继续进行技术设计？
  options: [继续, 修改需求, 取消]
  output: confirmation

# 3. 条件判断
- type: condition
  id: check_confirmation
  condition: "steps.confirm_requirement.output == '继续'"
  then:
    # 4. 技术设计
    - type: agent
      id: design
      agent: tech-architect
      input: |
        基于以下需求文档，进行技术设计：

        {{ steps.analyze.output }}

        输出技术设计文档，包括：
        - 系统架构
        - 数据模型
        - API 设计
        - 实现方案
      output: designDoc

    # 5. 代码实现（并行）
    - type: parallel
      id: implement
      branches:
        # 分支 1: 后端实现
        - - type: agent
            id: backend
            agent: build
            input: |
              根据技术设计文档实现后端代码：
```

```
          {{ steps.design.output }}
        output: backendCode

    # 分支 2: 前端实现
    - - type: agent
        id: frontend
        agent: build
        input: |
          根据技术设计文档实现前端代码:

          {{ steps.design.output }}
        output: frontendCode

  # 6. 代码审查
  - type: agent
    id: review
    agent: code-reviewer
    input: |
      请审查以下代码变更:

      ## 后端代码
      {{ steps.implement.backend.output }}

      ## 前端代码
      {{ steps.implement.frontend.output }}
    output: reviewResult

  # 7. 生成测试
  - type: agent
    id: test
    agent: qa-engineer
    input: |
      根据需求和实现生成测试用例:

      ## 需求
      {{ steps.analyze.output }}

      ## 代码变更
      {{ steps.implement.backend.output }}
    output: testCases

else:
  - type: tool
    id: cancel
    tool: notify
    params:
      message: " 工作流已取消"
```

```yaml
outputs:
  requirementDoc: "{{ steps.analyze.output }}"
  designDoc: "{{ steps.design.output }}"
  reviewResult: "{{ steps.review.output }}"
  testCases: "{{ steps.test.output }}"
```

6.3 Workflow 执行引擎

```typescript
// workflow/engine.ts
export class WorkflowEngine {
  private registry: WorkflowRegistry
  private agentRegistry: AgentRegistry
  private toolRegistry: ToolRegistry

  async execute(
    workflowId: string,
    inputs: unknown,
    ctx: ExecutionContext
  ): Promise<WorkflowResult> {
    const workflow = this.registry.get(workflowId)
    if (!workflow) throw new Error(`Workflow not found: ${workflowId}`)

    // 验证输入
    const validatedInputs = workflow.inputs.parse(inputs)

    // 创建执行状态
    const state = new WorkflowState(workflow, validatedInputs)

    // 执行步骤
    for (const step of workflow.steps) {
      if (ctx.abort.aborted) {
        state.status = "cancelled"
        break
      }

      try {
        await this.executeStep(step, state, ctx)
      } catch (error) {
        if (workflow.onError) {
          await workflow.onError(error, state, ctx)
        } else {
          throw error
        }
      }
    }

    return state.toResult()
```

```
}

private async executeStep(
  step: WorkflowStep,
  state: WorkflowState,
  ctx: ExecutionContext
): Promise<void> {
  switch (step.type) {
    case "agent":
      return this.executeAgentStep(step, state, ctx)
    case "tool":
      return this.executeToolStep(step, state, ctx)
    case "condition":
      return this.executeConditionStep(step, state, ctx)
    case "parallel":
      return this.executeParallelStep(step, state, ctx)
    case "human":
      return this.executeHumanStep(step, state, ctx)
    case "subworkflow":
      return this.executeSubWorkflowStep(step, state, ctx)
  }
}

private async executeAgentStep(
  step: AgentStep,
  state: WorkflowState,
  ctx: ExecutionContext
): Promise<void> {
  const agent = await this.agentRegistry.instantiate(step.agent, ctx)
  const input = this.renderTemplate(step.input, state)

  const result = await agent.run(input)

  if (step.output) {
    state.setVariable(step.output, result)
  }
  state.recordStep(step.id, result)
}

private async executeParallelStep(
  step: ParallelStep,
  state: WorkflowState,
  ctx: ExecutionContext
): Promise<void> {
  const promises = step.branches.map(branch =>
    this.executeBranch(branch, state.fork(), ctx)
  )
```

```
    const results = await Promise.all(promises)
    state.mergeBranches(results, step.mergeStrategy)
  }
}
```

6.5 核心动态规划器 (Dynamic Planner)

参考 Aime 论文 (arXiv:2507.11988) 和 Kimi K2.5 Agent Swarm 设计 核心能力：领域分析 → 动态组装 Agent → 实时反馈调整 → 并行编排执行

6.5.1 设计动机与问题分析　传统的 Plan-and-Execute 框架存在三大核心缺陷：

传统 Plan-and-Execute 框架的缺陷

刚性计划执行 (Rigid Plan Execution)

- 计划生成后固定不变，Planner 在执行期间空闲
- 无法适应执行过程中的实时反馈和意外结果
- 执行偏离计划时缺乏自适应能力

静态 Agent 能力 (Static Agent Capabilities)

- Agent 预定义角色和工具集固定不变
- 无法处理需要新技能的未预见任务
- 限制了系统的扩展性和鲁棒性

低效通信 (Inefficient Communication)

- Agent 间任务交接容易丢失上下文
- 缺乏集中状态管理，Agent 对全局进度视图不完整
- 导致重复工作和协调失败

我们的解决方案：动态规划器 (Dynamic Planner) 系统，实现：- 动态计划调整：基于实时反馈持续优化策略 - 按需 Agent 组装：根据任务需求动态创建专业化 Agent - 集中进度管理：全局状态感知，消除信息孤岛

6.5.2 动态规划器架构总览

动态规划器 (Dynamic Planner) 架构

用户请求

1. Domain Analyzer 领域分析器

意图识别	领域分类	复杂度评估
Intent Detect	Domain Classify	Complexity Est

领域分析结果: { domains, skills, complexity }

2. Dynamic Planner 动态规划器

任务分解	策略生成	依赖分析
Task Decompose	Strategy Gen	Dependency Map

输出: 全局任务列表 L = { subtask1, subtask2, ..., subtaskN }
每个 subtask 包含: 目标、完成标准、所需技能、预估时间

3. Actor Factory Agent 工厂

Persona 生成	Toolkit 组装	Knowledge 注入
角色人设	工具包选择	知识库加载

动态创建: At = { LLM, Toolkit, Prompt, Memory }

Dynamic Actor	Dynamic Actor	Dynamic Actor
(并行执行)	(并行执行)	(并行执行)
ReAct 循环	ReAct 循环	ReAct 循环

4. Progress Management 进度管理模块

状态追踪	实时同步	完成验证
Status Track	Real-time Sync	Completion Val

全局进度列表：[ ] Task1 | [x] Task2 进行中 | [ ] Task3 待开始

反馈循环：实时状态 → Dynamic Planner

持续迭代直到任务完成

6.5.3 领域分析器 (Domain Analyzer) 领域分析器负责分析用户请求，识别涉及的领域、所需技能和复杂度等级：

```
// planner/domain-analyzer.ts

/**
 * 领域分析器
 * 第一步：分析任务涉及的领域和所需能力
 */
export interface DomainAnalyzer {
  /**
   * 分析用户请求
   */
  analyze(request: UserRequest): Promise<DomainAnalysisResult>
}

/**
 * 领域分析结果
 */
export interface DomainAnalysisResult {
  // 识别的意图
  intent: {
    primary: string           // 主要意图
    secondary?: string[]      // 次要意图
    confidence: number        // 置信度 0-1
  }

  // 涉及的领域
  domains: DomainMatch[]

  // 所需技能
  requiredSkills: SkillRequirement[]

  // 所需工具
  requiredTools: ToolRequirement[]

  // 复杂度评估
  complexity: ComplexityAssessment
```

```
  // 建议的执行模式
  suggestedMode: ExecutionMode
}

/**
 * 领域匹配
 */
export interface DomainMatch {
  domain: string              // 领域 ID (coding, analysis, documentation, etc.)
  relevance: number           // 相关度 0-1
  subdomains?: string[]       // 子领域
}

/**
 * 技能需求
 */
export interface SkillRequirement {
  skillId: string             // 技能 ID
  importance: "critical" | "important" | "optional"
  reason: string              // 需要该技能的原因
}

/**
 * 复杂度评估
 */
export interface ComplexityAssessment {
  level: 0 | 1 | 2 | 3 | 4    // 对应 Agentic 5 级分级
  factors: ComplexityFactor[]
  estimatedSteps: number      // 预估步骤数
  estimatedDuration: string   // 预估耗时
  parallelizable: boolean     // 是否可并行
  parallelDegree?: number     // 建议并行度
}

/**
 * 执行模式
 */
export type ExecutionMode =
  | "single_agent"            // 单 Agent 模式 (Level 0-1)
  | "sequential"              // 顺序编排 (Level 2)
  | "parallel"                // 并行编排 (Level 2-3)
  | "swarm"                   // 蜂群模式 (Level 3-4)
  | "agent_swarm"             // Agent Swarm 并行蜂群 (Level 4, Kimi K2.5 模式)

/**
 * 领域分析器实现
 */
```

```
export class LLMDomainAnalyzer implements DomainAnalyzer {
  private model: LanguageModel
  private domainRegistry: DomainRegistry
  private skillRegistry: SkillRegistry

  async analyze(request: UserRequest): Promise<DomainAnalysisResult> {
    // 1. 构建分析 Prompt
    const analysisPrompt = this.buildAnalysisPrompt(request)

    // 2. 调用 LLM 进行领域分析
    const rawAnalysis = await this.model.generate({
      messages: [
        { role: "system", content: this.getSystemPrompt() },
        { role: "user", content: analysisPrompt }
      ],
      responseFormat: { type: "json_object" }
    })

    // 3. 解析并验证结果
    const parsed = DomainAnalysisSchema.parse(JSON.parse(rawAnalysis))

    // 4. 与注册表匹配，验证可用性
    const validated = await this.validateWithRegistries(parsed)

    return validated
  }

  private getSystemPrompt(): string {
    return `你是一个专业的任务分析专家。分析用户请求，输出结构化的领域分析结果。

分析维度：
1. 意图识别：用户想要完成什么
2. 领域分类：涉及哪些专业领域 (coding/analysis/documentation/testing/operations/management)
3. 技能需求：需要哪些具体技能
4. 复杂度评估：
   - Level 0: 纯推理问答，无需工具
   - Level 1: 需要外部工具/数据
   - Level 2: 多步骤战略性规划
   - Level 3: 需要多 Agent 协作
   - Level 4: 需要动态创建新能力
5. 并行性分析：任务是否可拆分并行执行

可用领域列表：
${JSON.stringify(this.domainRegistry.list(), null, 2)}

可用技能列表：
${JSON.stringify(this.skillRegistry.list(), null, 2)}
`
```

```
  }
}
```

6.5.4 动态规划器核心 (Dynamic Planner Core)　动态规划器是整个系统的核心，负责任务分解和持续策略调整：

```
// planner/dynamic-planner.ts

/**
 * 动态规划器
 * 核心创新：持续根据实时反馈调整策略
 *
 * 形式化定义（来自 Aime 论文）：
 * (L_{t+1}, g_{t+1}) = LLM_planner(P_planner, (G, L_t, H_t))
 *
 * 其中：
 * - G：总体目标
 * - L_t：当前任务列表
 * - H_t：历史执行结果
 * - L_{t+1}：更新后的任务列表（战略输出）
 * - g_{t+1}：下一个执行动作（战术输出）
 */
export interface DynamicPlanner {
  /**
   * 初始规划：分解目标为子任务
   */
  plan(goal: Goal, analysis: DomainAnalysisResult): Promise<TaskList>

  /**
   * 动态调整：根据反馈更新计划
   */
  replan(
    goal: Goal,
    currentTaskList: TaskList,
    executionHistory: ExecutionHistory
  ): Promise<ReplanResult>

  /**
   * 获取下一个执行动作
   */
  getNextAction(taskList: TaskList): Promise<NextAction>
}

/**
 * 任务列表（全局状态）
 */
export interface TaskList {
  id: string
```

```
  goal: Goal
  tasks: Task[]
  status: "planning" | "executing" | "completed" | "failed"
  createdAt: Date
  updatedAt: Date
}

/**
 * 任务定义
 */
export interface Task {
  id: string
  parentId?: string           // 父任务 ID（支持层级）

  // 任务描述
  title: string
  description: string

  // 完成标准（关键创新：显式定义）
  completionCriteria: string[]

  // 所需能力
  requiredSkills: string[]
  requiredTools: string[]

  // 依赖关系
  dependencies: string[]      // 依赖的其他任务 ID

  // 状态
  status: TaskStatus

  // 执行结果
  result?: TaskResult

  // 分配的 Actor
  assignedActorId?: string

  // 优先级和预估
  priority: "critical" | "high" | "medium" | "low"
  estimatedDuration?: number  // 分钟
}

export type TaskStatus =
  | "pending"       // 待开始
  | "ready"         // 依赖已满足，可执行
  | "assigned"      // 已分配 Actor
  | "in_progress"   // 执行中
  | "completed"     // 已完成
```

```
  | "failed"         // 失败
  | "blocked"        // 被阻塞

/**
 * 重规划结果
 */
export interface ReplanResult {
  // 更新后的任务列表（战略调整）
  updatedTaskList: TaskList

  // 变更说明
  changes: TaskChange[]

  // 下一个动作（战术决策）
  nextAction: NextAction

  // 规划说明
  reasoning: string
}

/**
 * 下一个动作
 */
export type NextAction =
  | { type: "dispatch"; taskId: string }          // 派发任务
  | { type: "wait"; reason: string }              // 等待
  | { type: "human_input"; prompt: string }       // 请求人工输入
  | { type: "complete"; summary: string }         // 完成
  | { type: "fail"; error: string }               // 失败

/**
 * 动态规划器实现
 */
export class LLMDynamicPlanner implements DynamicPlanner {
  private model: LanguageModel
  private progressManager: ProgressManager

  async plan(goal: Goal, analysis: DomainAnalysisResult): Promise<TaskList> {
    const prompt = this.buildPlanningPrompt(goal, analysis)

    const response = await this.model.generate({
      messages: [
        { role: "system", content: this.getPlanningSystemPrompt() },
        { role: "user", content: prompt }
      ],
      responseFormat: { type: "json_object" }
    })
```

```
  const tasks = TaskListSchema.parse(JSON.parse(response))

  // 初始化进度管理
  await this.progressManager.initialize(tasks)

  return tasks
}

async replan(
  goal: Goal,
  currentTaskList: TaskList,
  executionHistory: ExecutionHistory
): Promise<ReplanResult> {
  // 核心：双输出设计
  // 1. 战略层：更新全局任务列表
  // 2. 战术层：决定下一个具体动作

  const prompt = this.buildReplanPrompt(goal, currentTaskList, executionHistory)

  const response = await this.model.generate({
    messages: [
      { role: "system", content: this.getReplanSystemPrompt() },
      { role: "user", content: prompt }
    ],
    responseFormat: { type: "json_object" }
  })

  const result = ReplanResultSchema.parse(JSON.parse(response))

  // 同步更新进度管理器
  await this.progressManager.update(result.updatedTaskList)

  return result
}

async getNextAction(taskList: TaskList): Promise<NextAction> {
  // 找到所有"就绪"状态的任务
  const readyTasks = taskList.tasks.filter(t =>
    t.status === "ready" ||
    (t.status === "pending" && this.areDependenciesMet(t, taskList))
  )

  if (readyTasks.length === 0) {
    // 检查是否有正在进行的任务
    const inProgress = taskList.tasks.filter(t => t.status === "in_progress")
    if (inProgress.length > 0) {
      return { type: "wait", reason: "等待执行中的任务完成" }
    }
```

```
      // 检查是否全部完成
      const allCompleted = taskList.tasks.every(t => t.status === "completed")
      if (allCompleted) {
        return { type: "complete", summary: " 所有任务已完成" }
      }

      // 否则可能是失败或阻塞
      return { type: "fail", error: " 任务执行受阻" }
    }

    // 按优先级选择下一个任务
    const nextTask = this.selectHighestPriority(readyTasks)
    return { type: "dispatch", taskId: nextTask.id }
  }

  private getPlanningSystemPrompt(): string {
    return `你是一个专业的任务规划专家。将用户目标分解为可执行的子任务。

规划原则:
1. 每个子任务应该足够具体，可以被单独执行
2. 明确定义每个任务的完成标准
3. 识别任务之间的依赖关系
4. 评估哪些任务可以并行执行
5. 为每个任务指定所需技能和工具

输出格式: JSON 结构的任务列表，包含:
- tasks: 任务数组
- parallelGroups: 可并行的任务组
- criticalPath: 关键路径任务
`
  }
}
```

6.5.5 Actor 工厂 (Actor Factory)　Actor 工厂负责按需创建专业化的 Actor，这是动态 Agent 组装的核心:

```
// planner/actor-factory.ts

/**
 * Actor 工厂
 * 核心能力: 按需创建专业化 Actor（自助捏合 Agent）
 *
 * 形式化定义 (来自 Aime 论文):
 * A_t = F_factory(g_t) where A_t = { LLM_t, T_t, P_t, M_t }
 */
export interface ActorFactory {
  /**
```

```
    * 创建专业化 Actor
    */
  createActor(spec: ActorSpec): Promise<DynamicActor>

  /**
   * 获取可用的组件
   */
  getAvailableComponents(): Promise<ActorComponents>
}

/**
 * Actor 规格（描述要创建什么样的 Actor）
 */
export interface ActorSpec {
  // 任务信息
  task: Task

  // 领域分析结果（来自 Domain Analyzer）
  domainAnalysis: DomainAnalysisResult

  // 上下文信息
  context: ExecutionContext
}

/**
 * Actor 组件库
 */
export interface ActorComponents {
  // 可用的角色人设
  personas: PersonaTemplate[]

  // 可用的工具包
  toolBundles: ToolBundle[]

  // 可用的知识库
  knowledgeBases: KnowledgeBase[]

  // 可用的技能
  skills: SkillDefinition[]
}

/**
 * 角色人设模板
 */
export interface PersonaTemplate {
  id: string
  name: string
  description: string
```

```
  expertise: string[]

  // System Prompt 模板
  systemPromptTemplate: string

  // 适用场景
  applicableDomains: string[]

  // 推荐工具
  recommendedTools: string[]
}

/**
 * 工具包（Bundle）
 * 关键设计：将工具组织为功能包，而非扁平列表
 */
export interface ToolBundle {
  id: string
  name: string
  description: string
  category: string

  // 包含的工具
  tools: ToolDefinition[]

  // 适用场景
  useCases: string[]
}

/**
 * 动态 Actor（运行时实例）
 */
export interface DynamicActor {
  id: string

  // 组装的组件
  persona: PersonaTemplate
  toolkit: ToolDefinition[]
  knowledgeBase: KnowledgeBase[]

  // 生成的 System Prompt
  systemPrompt: string

  // 执行任务
  execute(task: Task): Promise<TaskResult>

  // 报告进度
  reportProgress(status: ProgressUpdate): Promise<void>
```

```
}

/**
 * Actor 工厂实现
 */
export class LLMActorFactory implements ActorFactory {
  private model: LanguageModel
  private personaRegistry: PersonaRegistry
  private toolBundleRegistry: ToolBundleRegistry
  private knowledgeBaseRegistry: KnowledgeBaseRegistry
  private skillRegistry: SkillRegistry

  async createActor(spec: ActorSpec): Promise<DynamicActor> {
    // 1. 选择合适的 Persona
    const persona = await this.selectPersona(spec)

    // 2. 组装工具包
    const toolkit = await this.assembleToolkit(spec)

    // 3. 加载相关知识库
    const knowledge = await this.loadKnowledge(spec)

    // 4. 生成定制化 System Prompt
    const systemPrompt = await this.generateSystemPrompt(
      persona,
      toolkit,
      knowledge,
      spec.task,
      spec.context
    )

    // 5. 创建 Actor 实例
    return new DynamicActorImpl({
      id: generateId(),
      persona,
      toolkit,
      knowledgeBase: knowledge,
      systemPrompt,
      model: this.model,
      progressManager: spec.context.progressManager
    })
  }

  /**
   * 选择 Persona
   * 基于任务需求选择最合适的角色人设
   */
  private async selectPersona(spec: ActorSpec): Promise<PersonaTemplate> {
```

```
  const { task, domainAnalysis } = spec

  // 获取候选 Personas
  const candidates = await this.personaRegistry.findByDomains(
    domainAnalysis.domains.map(d => d.domain)
  )

  if (candidates.length === 0) {
    // 动态生成 Persona
    return this.generatePersona(spec)
  }

  // 用 LLM 选择最合适的
  const selection = await this.model.generate({
    messages: [{
      role: "user",
      content: `任务: ${task.title}\n描述: ${task.description}\n所需技能: ${task.requiredSkill
    }]
  })

  return candidates.find(p => p.id === selection.trim()) || candidates[0]
}

/**
 * 组装工具包
 * Bundle 模式: 选择功能包而非单个工具
 */
private async assembleToolkit(spec: ActorSpec): Promise<ToolDefinition[]> {
  const { task, domainAnalysis } = spec

  // 1. 根据所需工具获取 Bundles
  const requiredBundles = new Set<string>()

  for (const toolReq of domainAnalysis.requiredTools) {
    const bundle = await this.toolBundleRegistry.findByTool(toolReq.toolId)
    if (bundle) {
      requiredBundles.add(bundle.id)
    }
  }

  // 2. 根据领域补充推荐 Bundles
  for (const domain of domainAnalysis.domains) {
    const domainBundles = await this.toolBundleRegistry.findByDomain(domain.domain)
    domainBundles.forEach(b => requiredBundles.add(b.id))
  }

  // 3. 组装最终工具列表
  const tools: ToolDefinition[] = []
```

```
    for (const bundleId of requiredBundles) {
      const bundle = await this.toolBundleRegistry.get(bundleId)
      tools.push(...bundle.tools)
    }

    // 4. 添加通用系统工具
    tools.push(this.getProgressUpdateTool())

    return tools
  }

  /**
   * 生成 System Prompt
   * 组合多个组件生成定制化提示
   */
  private async generateSystemPrompt(
    persona: PersonaTemplate,
    toolkit: ToolDefinition[],
    knowledge: KnowledgeBase[],
    task: Task,
    context: ExecutionContext
  ): Promise<string> {
    // 公式 (来自 Aime):
    // P_t = Compose( _t, desc(T_t),  _t,  , Γ)

    const components = {
      // _t: Persona
      persona: this.renderPersona(persona, task),

      // desc(T_t): Tool descriptions
      toolDescriptions: this.renderToolDescriptions(toolkit),

      // _t: Knowledge
      knowledge: await this.renderKnowledge(knowledge, task),

      //  : Environment
      environment: this.renderEnvironment(context),

      // Γ: Format requirements
      format: this.renderFormatRequirements(task)
    }

    return `${components.persona}

## 可用工具

${components.toolDescriptions}
```

```
## 相关知识

${components.knowledge}

## 环境信息

${components.environment}

## 输出要求

${components.format}

## 当前任务

** 标题 **: ${task.title}
** 描述 **: ${task.description}
** 完成标准 **:
${task.completionCriteria.map(c => `- ${c}`).join('\n')}

请开始执行任务，使用 ReAct 模式：思考 → 行动 → 观察 → 思考 → ...
在执行过程中，适时调用 update_progress 工具报告进度。
`
  }

  /**
   * 获取进度更新工具
   * 特殊系统工具，用于 Actor 向 Planner 报告进度
   */
  private getProgressUpdateTool(): ToolDefinition {
    return {
      name: "update_progress",
      description: " 向系统报告当前任务进度，包括重要里程碑或遇到的障碍",
      parameters: z.object({
        status: z.enum(["in_progress", "milestone", "blocked", "completed", "failed"]),
        message: z.string().describe(" 进度说明"),
        percentage: z.number().min(0).max(100).optional(),
        artifacts: z.array(z.object({
          type: z.string(),
          path: z.string()
        })).optional()
      }),
      execute: async (params, ctx) => {
        await ctx.progressManager.reportProgress(ctx.taskId, params)
        return { success: true }
      }
    }
  }
}
```

6.5.6 进度管理模块 (Progress Management) 进度管理模块是系统协调的核心，提供全局状态感知：

```typescript
// planner/progress-manager.ts

/**
 * 进度管理模块
 * 核心职责：维护全局任务状态，作为系统"单一事实源"
 */
export interface ProgressManager {
  /**
   * 初始化任务列表
   */
  initialize(taskList: TaskList): Promise<void>

  /**
   * 更新任务列表
   */
  update(taskList: TaskList): Promise<void>

  /**
   * 接收进度报告
   */
  reportProgress(taskId: string, update: ProgressUpdate): Promise<void>

  /**
   * 获取当前进度
   */
  getProgress(): Promise<ProgressSnapshot>

  /**
   * 订阅进度变更
   */
  subscribe(listener: ProgressListener): Unsubscribe
}

/**
 * 进度快照
 */
export interface ProgressSnapshot {
  taskList: TaskList

  // 统计信息
  stats: {
    total: number
    completed: number
    inProgress: number
    failed: number
    blocked: number
```

```
    pending: number
  }

  // 完成百分比
  overallProgress: number

  // 关键路径状态
  criticalPathStatus: "on_track" | "delayed" | "blocked"

  // 预估剩余时间
  estimatedTimeRemaining?: number
}

/**
 * 进度列表（Markdown 格式，人机可读）
 *
 * 示例:
 * - Objective 1: Perform Initial Research
 *   - [x] Sub-objective 1.1: Research top attractions
 *   - [x] Sub-objective 1.2: Investigate transportation options
 * - Objective 2: Finalize Itinerary and Budget
 *   - [ ] Sub-objective 2.1: Research hotel accommodations
 *   - [ ] Sub-objective 2.2: Calculate total estimated budget
 */
export class ProgressListRenderer {
  render(taskList: TaskList): string {
    const lines: string[] = []

    const renderTask = (task: Task, indent: number) => {
      const prefix = "  ".repeat(indent)
      const status = task.status === "completed" ? "[x]" : "[ ]"
      const statusEmoji = this.getStatusEmoji(task.status)

      lines.push(`${prefix}- ${status} ${statusEmoji} ${task.title}`)

      // 如果有子任务，递归渲染
      const children = taskList.tasks.filter(t => t.parentId === task.id)
      for (const child of children) {
        renderTask(child, indent + 1)
      }
    }

    // 渲染顶级任务
    const topLevelTasks = taskList.tasks.filter(t => !t.parentId)
    for (const task of topLevelTasks) {
      renderTask(task, 0)
    }
```

```
    return lines.join('\n')
  }

  private getStatusEmoji(status: TaskStatus): string {
    const map: Record<TaskStatus, string> = {
      pending: " ",
      ready: " ",
      assigned: " ",
      in_progress: " ",
      completed: " ",
      failed: " ",
      blocked: " "
    }
    return map[status]
  }
}

/**
 * 进度管理器实现
 */
export class ProgressManagerImpl implements ProgressManager {
  private taskList: TaskList
  private listeners: Set<ProgressListener> = new Set()
  private eventEmitter: EventEmitter

  async initialize(taskList: TaskList): Promise<void> {
    this.taskList = taskList
    this.notifyListeners()
  }

  async update(taskList: TaskList): Promise<void> {
    // 计算差异
    const changes = this.computeChanges(this.taskList, taskList)

    this.taskList = taskList

    // 持久化
    await this.persist()

    // 通知变更
    this.notifyListeners(changes)
  }

  async reportProgress(taskId: string, update: ProgressUpdate): Promise<void> {
    const task = this.taskList.tasks.find(t => t.id === taskId)
    if (!task) return

    // 更新任务状态
```

```
    if (update.status === "completed") {
      task.status = "completed"
      task.result = {
        success: true,
        output: update.message,
        artifacts: update.artifacts
      }
    } else if (update.status === "failed") {
      task.status = "failed"
      task.result = {
        success: false,
        error: update.message
      }
    } else if (update.status === "blocked") {
      task.status = "blocked"
    } else {
      task.status = "in_progress"
    }

    // 发送实时更新事件
    this.eventEmitter.emit("progress_update", {
      taskId,
      update,
      snapshot: await this.getProgress()
    })

    this.notifyListeners()
  }

  async getProgress(): Promise<ProgressSnapshot> {
    const tasks = this.taskList.tasks

    const stats = {
      total: tasks.length,
      completed: tasks.filter(t => t.status === "completed").length,
      inProgress: tasks.filter(t => t.status === "in_progress").length,
      failed: tasks.filter(t => t.status === "failed").length,
      blocked: tasks.filter(t => t.status === "blocked").length,
      pending: tasks.filter(t => t.status === "pending" || t.status === "ready").length
    }

    return {
      taskList: this.taskList,
      stats,
      overallProgress: Math.round((stats.completed / stats.total) * 100),
      criticalPathStatus: this.analyzeCriticalPath(),
      estimatedTimeRemaining: this.estimateRemainingTime()
    }
```

```
  }
}
```

6.5.7 并行调度 (Parallel Orchestration，借鉴 Kimi K2.5)　借鉴 Kimi K2.5 Agent Swarm 的 PARL (Parallel-Agent Reinforcement Learning) 思想：

```typescript
// planner/parallel-orchestrator.ts

/**
 * 并行编排器
 * 借鉴 Kimi K2.5 Agent Swarm 的设计：
 * - 可协调多达 100 个子 Agent
 * - 执行多达 1500 步的并行工作流
 * - 无需预定义角色或手工工作流
 */
export interface ParallelOrchestrator {
  /**
   * 分析任务的并行潜力
   */
  analyzeParallelism(taskList: TaskList): Promise<ParallelismAnalysis>

  /**
   * 执行并行调度
   */
  executeParallel(
    tasks: Task[],
    factory: ActorFactory,
    context: ExecutionContext
  ): Promise<ParallelExecutionResult>
}

/**
 * 并行度分析结果
 */
export interface ParallelismAnalysis {
  // 可并行的任务组
  parallelGroups: TaskGroup[]

  // 最大并行度
  maxParallelism: number

  // 关键步骤数（延迟指标）
  criticalSteps: number

  // 预估时间节省
  estimatedTimeSaving: number
}
```

```typescript
/**
 * 关键步骤计算
 * 借鉴 Kimi K2.5 的 Critical Steps 指标:
 *
 * CriticalSteps = Σ(S_main(t) + max_i S_sub,i(t))
 *
 * S_main(t): 主编排器开销
 * max_i S_sub,i(t): 每阶段最慢子 Agent
 */
export function calculateCriticalSteps(
  orchestratorSteps: number[],
  subAgentSteps: number[][]
): number {
  let total = 0

  for (let t = 0; t < orchestratorSteps.length; t++) {
    const mainSteps = orchestratorSteps[t]
    const maxSubSteps = subAgentSteps[t]
      ? Math.max(...subAgentSteps[t])
      : 0

    total += mainSteps + maxSubSteps
  }

  return total
}

/**
 * PARL 奖励函数
 * 借鉴 Kimi K2.5 的奖励设计:
 *
 * r_PARL(x, y) = 1 · r_parallel + 2 · r_finish + r_perf(x, y)
 *
 * r_parallel: 鼓励并行实例化（防止串行坍塌）
 * r_finish: 鼓励子任务完成（防止虚假并行）
 * r_perf: 任务级结果质量
 */
export interface PARLReward {
  // 并行奖励: 鼓励并行化
  parallelReward: number

  // 完成奖励: 鼓励子任务成功完成
  finishReward: number

  // 性能奖励: 任务质量
  performanceReward: number

  // 退火系数（训练过程中递减）
```

```
  lambda1: number
  lambda2: number

  // 总奖励
  total(): number
}

/**
 * 并行编排器实现
 */
export class ParallelOrchestratorImpl implements ParallelOrchestrator {
  private actorPool: Map<string, DynamicActor> = new Map()
  private maxConcurrency: number

  constructor(options: { maxConcurrency?: number } = {}) {
    this.maxConcurrency = options.maxConcurrency || 100  // Kimi K2.5 最大 100 子 Agent
  }

  async analyzeParallelism(taskList: TaskList): Promise<ParallelismAnalysis> {
    const tasks = taskList.tasks

    // 构建依赖图
    const dependencyGraph = this.buildDependencyGraph(tasks)

    // 拓扑排序，识别可并行任务
    const levels = this.topologicalLevels(dependencyGraph)

    // 分析每层的并行度
    const parallelGroups = levels.map((levelTasks, levelIndex) => ({
      level: levelIndex,
      tasks: levelTasks,
      parallelism: levelTasks.length
    }))

    const maxParallelism = Math.max(...parallelGroups.map(g => g.parallelism))

    // 计算关键步骤
    const criticalSteps = this.estimateCriticalSteps(parallelGroups)

    // 对比串行执行估算时间节省
    const serialSteps = tasks.reduce((sum, t) => sum + (t.estimatedDuration || 1), 0)
    const estimatedTimeSaving = Math.max(0, serialSteps - criticalSteps)

    return {
      parallelGroups,
      maxParallelism,
      criticalSteps,
      estimatedTimeSaving
```

```
  }
}

async executeParallel(
  tasks: Task[],
  factory: ActorFactory,
  context: ExecutionContext
): Promise<ParallelExecutionResult> {
  // 限制并行度
  const concurrency = Math.min(tasks.length, this.maxConcurrency)

  // 创建 Actor 实例
  const actors = await Promise.all(
    tasks.slice(0, concurrency).map(task =>
      factory.createActor({
        task,
        domainAnalysis: context.domainAnalysis,
        context
      })
    )
  )

  // 记录开始时间
  const startTime = Date.now()

  // 并行执行
  const results = await Promise.allSettled(
    actors.map((actor, index) => actor.execute(tasks[index]))
  )

  // 收集结果
  const executionResults: TaskResult[] = results.map((result, index) => {
    if (result.status === "fulfilled") {
      return result.value
    } else {
      return {
        success: false,
        error: result.reason.message
      }
    }
  })

  // 计算实际关键步骤
  const actualDuration = Date.now() - startTime

  return {
    results: executionResults,
    parallelism: concurrency,
```

```
    duration: actualDuration,
    successRate: executionResults.filter(r => r.success).length / executionResults.length
  }
}

/**
 * 构建依赖图
 */
private buildDependencyGraph(tasks: Task[]): Map<string, string[]> {
  const graph = new Map<string, string[]>()

  for (const task of tasks) {
    graph.set(task.id, task.dependencies || [])
  }

  return graph
}

/**
 * 拓扑分层
 * 同一层的任务可以并行执行
 */
private topologicalLevels(graph: Map<string, string[]>): Task[][] {
  const levels: Task[][] = []
  const visited = new Set<string>()
  const taskMap = new Map<string, Task>()

  // 计算每个节点的入度
  const inDegree = new Map<string, number>()
  for (const [id, deps] of graph) {
    if (!inDegree.has(id)) inDegree.set(id, 0)
    for (const dep of deps) {
      inDegree.set(dep, (inDegree.get(dep) || 0) + 1)
    }
  }

  // BFS 分层
  let currentLevel = Array.from(graph.keys()).filter(id =>
    (inDegree.get(id) || 0) === 0
  )

  while (currentLevel.length > 0) {
    levels.push(currentLevel.map(id => taskMap.get(id)!))

    const nextLevel: string[] = []
    for (const id of currentLevel) {
      visited.add(id)
      const deps = graph.get(id) || []
```

```
        for (const dep of deps) {
          const newDegree = (inDegree.get(dep) || 0) - 1
          inDegree.set(dep, newDegree)
          if (newDegree === 0 && !visited.has(dep)) {
            nextLevel.push(dep)
          }
        }
      }

      currentLevel = nextLevel
    }

    return levels
  }
}
```

6.5.8 完整工作流程

动态规划器完整工作流程

Step 1: 用户请求

"帮我开发一个用户管理模块，包括注册、登录、权限控制"

Step 2: 领域分析 (Domain Analyzer)

```
{
  intent: { primary: "feature_development", confidence: 0.95 },
  domains: [
    { domain: "coding", relevance: 0.9 },
    { domain: "documentation", relevance: 0.6 },
    { domain: "testing", relevance: 0.7 }
  ],
  requiredSkills: ["backend_dev", "api_design", "auth_impl"],
  complexity: { level: 3, parallelizable: true, parallelDegree: 3 },
  suggestedMode: "parallel"
}
```

Step 3: 任务分解 (Dynamic Planner)

```
TaskList = {
  tasks: [
```

```
    { id: "T1", title: "API 接口设计", dependencies: [] },
    { id: "T2", title: "用户注册功能", dependencies: ["T1"] },
    { id: "T3", title: "用户登录功能", dependencies: ["T1"] },
    { id: "T4", title: "权限控制模块", dependencies: ["T1"] },
    { id: "T5", title: "单元测试编写", dependencies: ["T2","T3","T4"] },
    { id: "T6", title: "API 文档生成", dependencies: ["T1"] }
  ]
}
```

Step 4: 并行分析

```
Level 0: [T1]                    ← 串行，前置依赖
Level 1: [T2, T3, T4, T6]        ← 并行度 4
Level 2: [T5]                    ← 串行，等待前置

Critical Steps = 3（关键路径长度）
Serial Steps = 6（串行执行长度）
Time Saving  50%
```

Step 5: Actor 动态创建 (Actor Factory)

```
T1 → Actor_Architect { persona: "API设计师", tools: [read_file, write] }
T2 → Actor_Backend   { persona: "后端开发", tools: [code, test, debug] }
T3 → Actor_Backend   { persona: "后端开发", tools: [code, test, debug] }
T4 → Actor_Security  { persona: "安全工程师", tools: [code, analyze] }
T5 → Actor_QA        { persona: "测试工程师", tools: [test, coverage] }
T6 → Actor_DocWriter { persona: "技术文档", tools: [read, write_doc] }
```

Step 6: 并行执行 + 实时反馈

```
 T1执行       完成

 T2执行    T3执行    T4执行    T6执行    ← 并行

     进度报告   进度报告   进度报告
```

```
        Progress Manager 实时更新
  - [ ] T1
  - [ ] T2   75%
  - [ ] T3   60%
  - [ ] T4   40%
  - [ ] T5
  - [x] T6
```

Step 7: 动态重规划（如需要）

如果 T4 失败:
Planner 重新评估 → 添加 T4'（修复权限问题）→ 更新依赖 → 继续执行

Step 8: 完成汇总

所有任务完成 → 生成总结报告 → 输出给用户

6.5.9 自助 Agent 捏合接口　提供用户级别的 Agent 自定义能力:

```typescript
// planner/agent-composer.ts

/**
 * Agent 捏合器
 * 允许用户自定义组装 Agent
 */
export interface AgentComposer {
  /**
   * 交互式创建 Agent
   */
  createCustomAgent(spec: CustomAgentSpec): Promise<AgentDefinition>

  /**
   * 从模板创建
   */
  createFromTemplate(templateId: string, overrides?: Partial<CustomAgentSpec>): Promise<AgentDe

  /**
   * 验证 Agent 配置
```

```
   */
  validate(spec: CustomAgentSpec): Promise<ValidationResult>
}

/**
 * 自定义 Agent 规格
 */
export interface CustomAgentSpec {
  // 基本信息
  name: string
  description: string

  // 角色设定
  persona?: {
    role: string              // 角色名称
    expertise: string[]       // 专业领域
    personality?: string      // 性格特点
  }

  // System Prompt（可选，否则自动生成）
  systemPrompt?: string

  // 技能选择
  skills?: string[]

  // 工具选择
  tools?: string[]

  // 工具包选择（推荐方式）
  toolBundles?: string[]

  // 知识库
  knowledgeBases?: string[]

  // 高级配置
  advanced?: {
    // 模型选择
    model?: string
    temperature?: number
    maxTokens?: number

    // 执行模式
    orchestrationPattern?: "react" | "cot" | "tot" | "self_refine"

    // 权限
    permissions?: PermissionSet

    // 记忆配置
```

```
    memoryConfig?: {
      shortTermSize?: number
      longTermEnabled?: boolean
      episodicEnabled?: boolean
    }
  }
}

/**
 * Agent 捏合器实现
 */
export class AgentComposerImpl implements AgentComposer {
  private personaGenerator: PersonaGenerator
  private promptBuilder: PromptBuilder
  private toolRegistry: ToolRegistry
  private skillRegistry: SkillRegistry

  async createCustomAgent(spec: CustomAgentSpec): Promise<AgentDefinition> {
    // 1. 验证规格
    const validation = await this.validate(spec)
    if (!validation.valid) {
      throw new ValidationError(validation.errors)
    }

    // 2. 生成/使用 Persona
    const persona = spec.persona
      ? await this.personaGenerator.fromSpec(spec.persona)
      : await this.personaGenerator.generate(spec)

    // 3. 生成 System Prompt
    const systemPrompt = spec.systemPrompt
      || await this.promptBuilder.build({
          persona,
          skills: spec.skills,
          tools: await this.resolveTools(spec)
        })

    // 4. 组装工具
    const tools = await this.resolveTools(spec)

    // 5. 创建 Agent 定义
    const agentDef: AgentDefinition = {
      id: generateId(),
      name: spec.name,
      description: spec.description,
      systemPrompt,

      skills: spec.skills || [],
```

```
    tools: tools.map(t => t.name),

    model: spec.advanced?.model || "default",
    temperature: spec.advanced?.temperature || 0.7,
    maxTokens: spec.advanced?.maxTokens || 4096,

    orchestrationPattern: spec.advanced?.orchestrationPattern || "react",

    knowledgeBases: spec.knowledgeBases || [],

    permissions: spec.advanced?.permissions || defaultPermissions(),

    metadata: {
      createdBy: "agent_composer",
      createdAt: new Date().toISOString(),
      version: "1.0.0"
    }
  }

  // 6. 持久化
  await this.saveAgentDefinition(agentDef)

  return agentDef
}

private async resolveTools(spec: CustomAgentSpec): Promise<ToolDefinition[]> {
  const tools: ToolDefinition[] = []

  // 从工具包解析
  if (spec.toolBundles) {
    for (const bundleId of spec.toolBundles) {
      const bundle = await this.toolRegistry.getBundle(bundleId)
      tools.push(...bundle.tools)
    }
  }

  // 单独添加的工具
  if (spec.tools) {
    for (const toolId of spec.tools) {
      const tool = await this.toolRegistry.get(toolId)
      if (tool && !tools.find(t => t.name === tool.name)) {
        tools.push(tool)
      }
    }
  }

  return tools
}
```

```
}
```

6.5.10 Web 配置界面示例

```yaml
# 动态规划器 Web 配置示例
# 可通过 Web UI 配置以下内容

# 1. 预置领域配置
domains:
  - id: fullstack_dev
    name: " 全栈开发"
    description: " 前后端一体化开发"
    skills:
      - backend_dev
      - frontend_dev
      - api_design
      - database_design
    toolBundles:
      - code_editing
      - file_system
      - git_operations
      - testing
    defaultPersona: fullstack_engineer

# 2. 预置 Agent 模板
agentTemplates:
  - id: api_designer
    name: "API 设计师"
    persona:
      role: "Senior API Architect"
      expertise: ["RESTful API", "GraphQL", "OpenAPI"]
    toolBundles: [code_editing, documentation]
    orchestrationPattern: cot

  - id: security_engineer
    name: " 安全工程师"
    persona:
      role: "Security Engineer"
      expertise: ["Authentication", "Authorization", "OWASP"]
    toolBundles: [code_editing, security_analysis]
    orchestrationPattern: react

# 3. 工具包配置
toolBundles:
  - id: code_editing
    name: " 代码编辑"
    tools: [read_file, write_file, search_code, edit_code]
```

```yaml
  - id: testing
    name: " 测试工具"
    tools: [run_test, coverage_report, debug]

  - id: security_analysis
    name: " 安全分析"
    tools: [dependency_scan, code_audit, vulnerability_check]

# 4. 执行策略配置
executionPolicy:
  maxParallelism: 10              # 最大并行度
  defaultTimeout: 300000          # 默认超时（ms）
  enableDynamicReplan: true       # 启用动态重规划
  humanApprovalRequired:          # 需要人工确认的操作
    - file_delete
    - external_api_call
    - production_deploy
```

---

7. Swarm 蜂群动态扩容系统

借鉴 Swarm-IDE 的设计思想，实现 Agent 的动态创建、自组织协作和弹性扩缩容

7.1 Swarm 核心架构

Message Queue (per Agent)

Topology Graph

Human 人类也是特殊的 Agent

Agent (1-1) Agent (1-2) Agent (1-3)

Agent (1-1-1) Agent (1-1-2) Agent (1-1-3)

拓扑在运行时动态演化，Agent 按需"雇佣"下属

7.2 核心数据模型

```typescript
// swarm/types.ts

// Swarm Workspace - 一个独立的蜂群空间
export interface SwarmWorkspace {
  id: string
  name: string
  createdAt: Date

  // 人类 Agent（特殊）
  humanAgentId: string

  // 初始 Agent
  initialAgentId: string

  // 默认群组
  defaultGroupId: string
```

```
}

// Agent 实例（运行时）
export interface AgentInstance {
  id: string
  workspaceId: string

  // 角色信息
  role: string                          // 角色名称（如 "coder", "reviewer"）
  roleIndex: string                     // 层级索引（如 "1-1-2"）

  // 父子关系
  parentId?: string                     // 父 Agent ID
  childIds: string[]                    // 子 Agent IDs

  // LLM 历史（持久化）
  llmHistory: LLMMessage[]              // 单 Agent 全局记忆

  // 运行状态
  status: AgentStatus                   // idle | running | waiting | terminated

  // 元数据
  createdAt: Date
  lastActiveAt: Date
}

// Agent 状态
export type AgentStatus =
  | "idle"        // 空闲，等待消息
  | "running"     // 正在执行 LLM 推理
  | "waiting"     // 等待工具执行
  | "terminated"  // 已终止

// LLM 消息格式
export interface LLMMessage {
  role: "system" | "user" | "assistant" | "tool"
  content: string
  toolCalls?: ToolCall[]
  toolCallId?: string
  timestamp: Date
}

// 群组（IM 系统）
export interface SwarmGroup {
  id: string
  workspaceId: string
  name?: string                         // 群名，P2P 可为空
  memberIds: string[]                   // 成员 Agent IDs
```

```
  createdAt: Date
}

// 群成员关系
export interface GroupMember {
  groupId: string
  agentId: string
  lastReadMessageId?: string          // 最后读到的消息 ID
  joinedAt: Date
}

// 消息
export interface SwarmMessage {
  id: string                          // UUID v7 (可排序)
  workspaceId: string
  groupId: string
  senderId: string                    // 发送者 Agent ID

  contentType: "text" | "image" | "file" | "tool_result"
  content: string

  sendTime: Date
}
```

7.3 Agent 动态创建

```
// swarm/factory.ts
export class AgentFactory {
  private registry: AgentRegistry
  private pool: RunnerPool

  /**
   * 动态创建 Agent
   * 核心原语之一: create
   */
  async createAgent(spec: DynamicAgentSpec): Promise<AgentInstance> {
    // 1. 验证创建权限
    await this.validateCreationPermission(spec)

    // 2. 生成角色索引
    const roleIndex = await this.generateRoleIndex(spec.parentId, spec.role)

    // 3. 创建 Agent 实例
    const agent: AgentInstance = {
      id: generateId(),
      workspaceId: spec.workspaceId,
      role: spec.role,
      roleIndex,
```

```
    parentId: spec.parentId,
    childIds: [],
    llmHistory: [],
    status: "idle",
    createdAt: new Date(),
    lastActiveAt: new Date(),
  }

  // 4. 初始化 System Prompt
  const systemPrompt = await this.buildSystemPrompt(agent, spec)
  agent.llmHistory.push({
    role: "system",
    content: systemPrompt,
    timestamp: new Date(),
  })

  // 5. 持久化
  await Storage.agents.insert(agent)

  // 6. 更新父 Agent 的 childIds
  if (spec.parentId) {
    await Storage.agents.update(spec.parentId, {
      childIds: sql`array_append(childIds, ${agent.id})`,
    })
  }

  // 7. 创建 P2P 群组（与 Human）
  await this.createP2PGroup(spec.workspaceId, agent.id)

  // 8. 启动 Runner
  await this.pool.startRunner(agent)

  // 9. 发布事件
  Bus.publish(SwarmEvent.AgentCreated, { agent })

  return agent
}

/**
 * 生成层级索引
 * 例如: parent = "1-1", role = "coder" → "1-1-1"
 */
private async generateRoleIndex(
  parentId: string | undefined,
  role: string
): Promise<string> {
  if (!parentId) {
    // 根 Agent，从 1 开始
```

```
      const count = await Storage.agents.countRootAgents()
      return String(count + 1)
    }

    const parent = await Storage.agents.get(parentId)
    const siblingCount = parent.childIds.length
    return `${parent.roleIndex}-${siblingCount + 1}`
  }

  /**
   * 构建 System Prompt
   * 包含拓扑信息、协作规则
   */
  private async buildSystemPrompt(
    agent: AgentInstance,
    spec: DynamicAgentSpec
  ): Promise<string> {
    const topology = await this.getTopologyInfo(agent)

    return `
你是一个 ${spec.role}，在蜂群协作系统中工作。

## 你的身份
- 角色: ${spec.role}
- 索引: ${agent.roleIndex}
- 父级: ${spec.parentId ? `Agent ${spec.parentId}` : " 无（根级）"}

## 蜂群拓扑
${topology}

## 协作规则
1. 你可以使用 create_agent 工具创建子 Agent 来分担任务
2. 你可以使用 send_message 工具与任意 Agent 通信
3. 当任务不清晰时，向父级或同级 Agent 请求澄清
4. 完成任务后，向发起者报告结果

## 可用工具
${spec.tools?.map(t => `- ${t}`).join('\n') || '继承自父级'}

${spec.additionalInstructions || ''}
`.trim()
  }
}

// 动态 Agent 规格
export interface DynamicAgentSpec {
  workspaceId: string
  role: string                          // 角色名称
```

```
  parentId?: string                    // 父 Agent

  // 可选配置
  tools?: string[]                     // 工具列表
  model?: ModelSpec                    // 指定模型
  additionalInstructions?: string      // 额外指令
}
```

7.4 消息系统

```
// swarm/messaging.ts
export class MessageHub {
  private queues: Map<string, MessageQueue> = new Map()

  /**
   * 发送消息
   * 核心原语之一：send
   */
  async sendMessage(params: SendMessageParams): Promise<void> {
    // 1. 创建消息
    const message: SwarmMessage = {
      id: generateULID(),
      workspaceId: params.workspaceId,
      groupId: params.groupId,
      senderId: params.senderId,
      contentType: params.contentType || "text",
      content: params.content,
      sendTime: new Date(),
    }

    // 2. 持久化
    await Storage.messages.insert(message)

    // 3. 通知群成员
    const members = await Storage.groupMembers.byGroup(params.groupId)
    for (const member of members) {
      if (member.agentId !== params.senderId) {
        // 唤醒目标 Agent
        await this.wakeAgent(member.agentId, message)
      }
    }

    // 4. 发布 UI 事件
    Bus.publish(UIEvent.MessageCreated, { message })
  }

  /**
   * 发送私信
```

```
 * 自动定位/创建 P2P 群组
 */
async sendDirectMessage(params: DirectMessageParams): Promise<void> {
  // 查找或创建 P2P 群组
  let group = await Storage.groups.findP2P(params.senderId, params.toAgentId)

  if (!group) {
    group = await Storage.groups.create({
      workspaceId: params.workspaceId,
      memberIds: [params.senderId, params.toAgentId],
      name: null,
    })
  }

  // 发送消息
  await this.sendMessage({
    ...params,
    groupId: group.id,
  })
}

/**
 * 唤醒 Agent
 */
async wakeAgent(agentId: string, message: SwarmMessage): Promise<void> {
  const queue = this.getOrCreateQueue(agentId)
  queue.push(message)

  // 通知 Runner
  Bus.publish(SwarmEvent.AgentWake, { agentId, messageId: message.id })
}

/**
 * 获取未读消息
 */
async getUnreadMessages(agentId: string): Promise<UnreadBatch[]> {
  // 获取该 Agent 所在的所有群组
  const memberships = await Storage.groupMembers.byAgent(agentId)

  const batches: UnreadBatch[] = []
  for (const membership of memberships) {
    const messages = await Storage.messages.after(
      membership.groupId,
      membership.lastReadMessageId
    )

    if (messages.length > 0) {
      batches.push({
```

```
          groupId: membership.groupId,
          messages,
        })

        // 更新已读位置
        const lastMessage = messages[messages.length - 1]
        await Storage.groupMembers.updateLastRead(
          membership.groupId,
          agentId,
          lastMessage.id
        )
      }
    }

    return batches
  }
}

interface UnreadBatch {
  groupId: string
  messages: SwarmMessage[]
}
```

7.5 Agent Runner 生命周期

```
// swarm/runner.ts
export class AgentRunner {
  private agent: AgentInstance
  private messageHub: MessageHub
  private llm: LLMClient
  private aborted = false

  /**
   * Agent 主循环
   * 每个 Agent 独立运行，互不阻塞
   */
  async run(): Promise<void> {
    while (!this.aborted) {
      // 1. 检查未读消息
      const unread = await this.messageHub.getUnreadMessages(this.agent.id)

      if (unread.length === 0) {
        // 2. 无消息，进入等待
        await this.waitForWake()
        continue
      }

      // 3. 处理未读消息
```

```
    await this.processUnreadMessages(unread)
  }
}

/**
 * 处理未读消息
 */
private async processUnreadMessages(batches: UnreadBatch[]): Promise<void> {
  // 将消息加入 LLM 历史
  for (const batch of batches) {
    const groupInfo = await Storage.groups.get(batch.groupId)

    for (const msg of batch.messages) {
      const sender = await Storage.agents.get(msg.senderId)
      this.agent.llmHistory.push({
        role: "user",
        content: `[来自 ${sender.role}(${sender.roleIndex}) 在群组 ${groupInfo.name || 'P2P'}
        timestamp: msg.sendTime,
      })
    }
  }

  // 进入 LLM 推理循环
  await this.llmLoop()
}

/**
 * LLM 推理循环
 */
private async llmLoop(): Promise<void> {
  this.agent.status = "running"

  while (true) {
    // 1. 调用 LLM
    const response = await this.llm.chat({
      messages: this.agent.llmHistory,
      tools: await this.getAvailableTools(),
      stream: true,
    })

    // 2. 流式输出
    for await (const chunk of response.stream) {
      Bus.publish(UIEvent.AgentLLMChunk, {
        agentId: this.agent.id,
        chunk
      })
    }
```

```
    // 3. 判断结果
    const result = await response.finalMessage()

    this.agent.llmHistory.push({
      role: "assistant",
      content: result.content,
      toolCalls: result.toolCalls,
      timestamp: new Date(),
    })

    // 4. 处理工具调用
    if (result.toolCalls && result.toolCalls.length > 0) {
      await this.executeToolCalls(result.toolCalls)
      continue // 继续循环处理工具结果
    }

    // 5. 推理完成
    if (result.finishReason === "stop") {
      break
    }
  }

  // 6. 持久化 LLM 历史
  await Storage.agents.updateLLMHistory(
    this.agent.id,
    this.agent.llmHistory
  )

  this.agent.status = "idle"
  Bus.publish(UIEvent.AgentLLMDone, { agentId: this.agent.id })
}

/**
 * 执行工具调用
 */
private async executeToolCalls(toolCalls: ToolCall[]): Promise<void> {
  for (const call of toolCalls) {
    this.agent.status = "waiting"

    Bus.publish(UIEvent.ToolCallStart, {
      agentId: this.agent.id,
      toolCall: call
    })

    const result = await this.executeTool(call)

    this.agent.llmHistory.push({
      role: "tool",
```

```
        content: JSON.stringify(result),
        toolCallId: call.id,
        timestamp: new Date(),
      })

      Bus.publish(UIEvent.ToolCallDone, {
        agentId: this.agent.id,
        toolCall: call,
        result
      })
    }
  }

  /**
   * 等待唤醒
   */
  private async waitForWake(): Promise<void> {
    this.agent.status = "idle"

    return new Promise((resolve) => {
      const unsub = Bus.subscribe(SwarmEvent.AgentWake, (event) => {
        if (event.agentId === this.agent.id) {
          unsub()
          resolve()
        }
      })
    })
  }
}
```

7.6 Swarm 工具集

```
// swarm/tools.ts

/**
 * 创建子 Agent 工具
 */
export const createAgentTool = defineTool({
  id: "create_agent",
  name: " 创建子 Agent",
  description: " 创建一个子 Agent 来协助完成任务",
  category: "swarm",

  parameters: z.object({
    role: z.string().describe(" 子 Agent 的角色名称，如 'coder', 'reviewer', 'analyst'"),
    task: z.string().describe(" 分配给子 Agent 的任务描述"),
    tools: z.array(z.string()).optional().describe(" 子 Agent 可用的工具列表"),
  }),
```

```
  async execute(params, ctx) {
    const factory = ctx.services.get(AgentFactory)

    const agent = await factory.createAgent({
      workspaceId: ctx.workspaceId,
      role: params.role,
      parentId: ctx.agentId,
      tools: params.tools,
      additionalInstructions: `你的任务是: ${params.task}`,
    })

    // 自动发送任务消息
    await ctx.services.get(MessageHub).sendDirectMessage({
      workspaceId: ctx.workspaceId,
      senderId: ctx.agentId,
      toAgentId: agent.id,
      content: params.task,
    })

    return {
      title: `创建了子 Agent: ${agent.role}(${agent.roleIndex})`,
      output: `子 Agent ID: ${agent.id}\n角色: ${params.role}\n任务已发送`,
      metadata: { agentId: agent.id, roleIndex: agent.roleIndex },
    }
  },
})

/**
 * 发送消息工具
 */
export const sendMessageTool = defineTool({
  id: "send_message",
  name: " 发送消息",
  description: " 向另一个 Agent 发送消息",
  category: "swarm",

  parameters: z.object({
    toAgentId: z.string().optional().describe(" 目标 Agent ID"),
    toRole: z.string().optional().describe(" 目标 Agent 角色（如果不知道 ID）"),
    content: z.string().describe(" 消息内容"),
  }),

  async execute(params, ctx) {
    const hub = ctx.services.get(MessageHub)

    let targetId = params.toAgentId
```

```
    // 如果提供的是角色名，查找对应的 Agent
    if (!targetId && params.toRole) {
      const agent = await Storage.agents.findByRole(
        ctx.workspaceId,
        params.toRole
      )
      if (!agent) {
        throw new Error(`未找到角色为 "${params.toRole}" 的 Agent`)
      }
      targetId = agent.id
    }

    if (!targetId) {
      throw new Error(" 必须提供 toAgentId 或 toRole")
    }

    await hub.sendDirectMessage({
      workspaceId: ctx.workspaceId,
      senderId: ctx.agentId,
      toAgentId: targetId,
      content: params.content,
    })

    const target = await Storage.agents.get(targetId)

    return {
      title: `消息已发送给 ${target.role}(${target.roleIndex})`,
      output: `消息内容: ${params.content.substring(0, 100)}...`,
      metadata: { targetId, targetRole: target.role },
    }
  },
})

/**
 * 列出群组工具
 */
export const listGroupsTool = defineTool({
  id: "list_groups",
  name: " 列出群组",
  description: " 列出当前 Agent 所在的所有群组",
  category: "swarm",

  parameters: z.object({}),

  async execute(params, ctx) {
    const memberships = await Storage.groupMembers.byAgent(ctx.agentId)
    const groups = await Promise.all(
      memberships.map(m => Storage.groups.get(m.groupId))
```

```
    )

    const formatted = groups.map(g => ({
      id: g.id,
      name: g.name || "(P2P)",
      memberCount: g.memberIds.length,
    }))

    return {
      title: `找到 ${groups.length} 个群组`,
      output: JSON.stringify(formatted, null, 2),
      metadata: { count: groups.length },
    }
  },
})

/**
 * 查询拓扑工具
 */
export const getTopologyTool = defineTool({
  id: "get_topology",
  name: " 查询拓扑",
  description: " 查询当前 Agent 在蜂群中的位置和相关 Agent",
  category: "swarm",

  parameters: z.object({
    scope: z.enum(["ancestors", "children", "siblings", "all"]).default("all"),
  }),

  async execute(params, ctx) {
    const topology = ctx.services.get(TopologyManager)
    const info = await topology.getInfo(ctx.agentId, params.scope)

    return {
      title: " 拓扑信息",
      output: formatTopologyTree(info),
      metadata: info,
    }
  },
})
```

7.7 动态扩缩容策略

```
// swarm/scaling.ts
export class SwarmScaler {
  /**
   * 自动扩容策略
   */
```

```
  async evaluateAndScale(workspaceId: string): Promise<void> {
    const metrics = await this.collectMetrics(workspaceId)

    // 策略 1: 基于队列长度
    if (metrics.avgQueueLength > QUEUE_THRESHOLD) {
      await this.scaleUp(workspaceId, metrics)
    }

    // 策略 2: 基于响应时间
    if (metrics.avgResponseTime > RESPONSE_THRESHOLD) {
      await this.scaleUp(workspaceId, metrics)
    }

    // 策略 3: 基于空闲率
    if (metrics.idleRatio > IDLE_THRESHOLD) {
      await this.scaleDown(workspaceId, metrics)
    }
  }

  /**
   * 扩容: 建议 Agent 创建更多子 Agent
   */
  private async scaleUp(
    workspaceId: string,
    metrics: SwarmMetrics
  ): Promise<void> {
    // 找到负载最高的 Agent
    const busiest = await this.findBusiestAgent(workspaceId)

    // 发送扩容建议
    await this.messageHub.sendMessage({
      workspaceId,
      senderId: "system",
      groupId: await this.getSystemGroupId(busiest.id),
      content: `
[系统建议] 检测到任务积压，建议创建子 Agent 分担工作。

当前状态:
- 消息队列长度: ${metrics.avgQueueLength}
- 平均响应时间: ${metrics.avgResponseTime}ms

你可以使用 create_agent 工具创建专门的子 Agent 来处理特定类型的任务。
      `.trim(),
    })
  }

  /**
   * 缩容: 终止空闲的 Agent
```

```
   */
  private async scaleDown(
    workspaceId: string,
    metrics: SwarmMetrics
  ): Promise<void> {
    // 找到长期空闲的子 Agent
    const idleAgents = await this.findIdleAgents(
      workspaceId,
      IDLE_DURATION_THRESHOLD
    )

    for (const agent of idleAgents) {
      // 只终止子 Agent，保留根 Agent
      if (agent.parentId) {
        await this.terminateAgent(agent.id)
      }
    }
  }

  /**
   * 终止 Agent
   */
  async terminateAgent(agentId: string): Promise<void> {
    // 1. 停止 Runner
    await this.runnerPool.stopRunner(agentId)

    // 2. 通知父 Agent
    const agent = await Storage.agents.get(agentId)
    if (agent.parentId) {
      await this.messageHub.sendDirectMessage({
        workspaceId: agent.workspaceId,
        senderId: "system",
        toAgentId: agent.parentId,
        content: `[系统通知] 子 Agent ${agent.role}(${agent.roleIndex}) 因长期空闲已被终止。`,
      })

      // 更新父 Agent 的 childIds
      await Storage.agents.update(agent.parentId, {
        childIds: sql`array_remove(childIds, ${agentId})`,
      })
    }

    // 3. 更新状态
    await Storage.agents.update(agentId, { status: "terminated" })

    // 4. 发布事件
    Bus.publish(SwarmEvent.AgentTerminated, { agentId })
  }
```

```
}
```

7.8 UI 事件流

```
// swarm/ui-events.ts

/**
 * UI 事件类型
 * 通过 SSE 推送到前端
 */
export enum UIEvent {
  // Agent 事件
  AgentCreated = "ui.agent.created",
  AgentTerminated = "ui.agent.terminated",

  // 消息事件
  MessageCreated = "ui.message.created",

  // LLM 事件
  AgentLLMStart = "ui.agent.llm.start",
  AgentLLMChunk = "ui.agent.llm.chunk",
  AgentLLMDone = "ui.agent.llm.done",

  // 工具事件
  ToolCallStart = "ui.agent.tool_call.start",
  ToolCallDone = "ui.agent.tool_call.done",

  // 群组事件
  GroupCreated = "ui.group.created",

  // 拓扑事件
  TopologyChanged = "ui.topology.changed",
}

/**
 * SSE 路由
 */
export const swarmEventRoutes = new OpenAPIHono()
  .get("/events", async (c) => {
    const workspaceId = c.req.query("workspaceId")

    return streamSSE(c, async (stream) => {
      const unsub = Bus.subscribe("ui.*", async (event) => {
        if (event.workspaceId === workspaceId) {
          await stream.writeSSE({
            event: event.type,
            data: JSON.stringify(event.payload),
          })
```

```
        }
      })

      stream.onAbort(() => unsub())
    })
  })
```

7.9 前端 Swarm 可视化

```
// components/SwarmGraph.tsx
import ReactFlow, {
  Controls,
  Background,
  MiniMap,
  useNodesState,
  useEdgesState,
} from "reactflow"

export function SwarmGraph({ workspaceId }: { workspaceId: string }) {
  const [nodes, setNodes, onNodesChange] = useNodesState([])
  const [edges, setEdges, onEdgesChange] = useEdgesState([])
  const [activeMessages, setActiveMessages] = useState<MessageFlow[]>([])

  // 订阅拓扑变化
  useEffect(() => {
    const eventSource = new EventSource(
      `/api/swarm/events?workspaceId=${workspaceId}`
    )

    eventSource.addEventListener("ui.topology.changed", (e) => {
      const topology = JSON.parse(e.data)
      const { nodes, edges } = topologyToFlow(topology)
      setNodes(nodes)
      setEdges(edges)
    })

    eventSource.addEventListener("ui.message.created", (e) => {
      const msg = JSON.parse(e.data)
      // 添加消息动画
      setActiveMessages(prev => [...prev, {
        id: msg.id,
        from: msg.senderId,
        to: msg.targetId,
      }])

      // 3 秒后移除动画
      setTimeout(() => {
        setActiveMessages(prev => prev.filter(m => m.id !== msg.id))
```

```
      }, 3000)
    })

    return () => eventSource.close()
  }, [workspaceId])

  return (
    <div className="swarm-graph h-full">
      <ReactFlow
        nodes={nodes}
        edges={edges}
        onNodesChange={onNodesChange}
        onEdgesChange={onEdgesChange}
        nodeTypes={{
          agent: AgentNode,
          human: HumanNode,
        }}
        edgeTypes={{
          message: AnimatedMessageEdge,
        }}
        fitView
      >
        <Controls />
        <MiniMap />
        <Background />

        {/* 消息流动动画层 */}
        <MessageAnimationLayer messages={activeMessages} />
      </ReactFlow>
    </div>
  )
}

// Agent 节点组件
function AgentNode({ data }: { data: AgentNodeData }) {
  return (
    <div className={cn(
      "agent-node px-4 py-2 rounded-lg border-2",
      data.status === "running" && "border-green-500 animate-pulse",
      data.status === "idle" && "border-gray-300",
      data.status === "waiting" && "border-yellow-500",
    )}>
      <div className="font-bold">{data.role}</div>
      <div className="text-xs text-gray-500">{data.roleIndex}</div>
      {data.status === "running" && (
        <div className="text-xs text-green-600"> 推理中...</div>
      )}
    </div>
```

```
  )
}
```

---

8. Web 配置管理系统

7.1 架构设计

7.2 API 路由设计

```
// server/routes/admin.ts
import { OpenAPIHono } from "@hono/zod-openapi"

export const adminRoutes = new OpenAPIHono()

// ============ Agent 管理 ============
  .get("/agents",
    // 获取所有 Agent 列表
```

```
    async (c) => {
      const agents = AgentRegistry.list()
      return c.json({ agents })
    }
  )

  .get("/agents/:id",
    // 获取单个 Agent 详情
    async (c) => {
      const agent = AgentRegistry.get(c.req.param("id"))
      return c.json({ agent })
    }
  )

  .post("/agents",
    validator("json", AgentDefinitionSchema),
    // 注册新 Agent
    async (c) => {
      const definition = c.req.valid("json")
      AgentRegistry.register(definition)
      return c.json({ success: true, id: definition.id })
    }
  )

  .put("/agents/:id",
    validator("json", AgentDefinitionSchema.partial()),
    // 更新 Agent
    async (c) => {
      const id = c.req.param("id")
      const updates = c.req.valid("json")
      AgentRegistry.update(id, updates)
      return c.json({ success: true })
    }
  )

  .delete("/agents/:id",
    // 删除 Agent
    async (c) => {
      AgentRegistry.unregister(c.req.param("id"))
      return c.json({ success: true })
    }
  )

// ============ Tool 管理 ============
  .get("/tools", async (c) => { ... })
  .post("/tools", async (c) => { ... })
  .put("/tools/:id", async (c) => { ... })
  .delete("/tools/:id", async (c) => { ... })
```

```
// ============ Workflow 管理 ============
  .get("/workflows", async (c) => { ... })
  .post("/workflows", async (c) => { ... })
  .post("/workflows/:id/execute", async (c) => { ... })
  .get("/workflows/:id/executions", async (c) => { ... })

// ============ 配置管理 ============
  .get("/config", async (c) => { ... })
  .put("/config", async (c) => { ... })
  .post("/config/import", async (c) => { ... })
  .get("/config/export", async (c) => { ... })

// ============ 监控 ============
  .get("/events", async (c) => {
    // SSE 事件流
    return streamSSE(c, async (stream) => {
      const unsub = Bus.subscribeAll(async (event) => {
        await stream.writeSSE({ data: JSON.stringify(event) })
      })
      stream.onAbort(() => unsub())
    })
  })

  .get("/metrics", async (c) => {
    // 指标数据
    return c.json({
      sessions: SessionManager.metrics(),
      agents: AgentRegistry.metrics(),
      tools: ToolRegistry.metrics(),
    })
  })
```

7.3 前端组件设计

Agent Builder 组件

```
// components/AgentBuilder.tsx
import { useState } from "react"
import { MonacoEditor } from "@monaco-editor/react"

export function AgentBuilder() {
  const [definition, setDefinition] = useState<AgentDefinition>(defaultAgent)
  const [mode, setMode] = useState<"visual" | "code">("visual")

  return (
    <div className="agent-builder">
      <header>
        <h1>Agent Builder</h1>
```

```
        <div className="mode-switch">
          <button onClick={() => setMode("visual")}> 可视化 </button>
          <button onClick={() => setMode("code")}> 代码 </button>
        </div>
      </header>

      {mode === "visual" ? (
        <VisualEditor
          definition={definition}
          onChange={setDefinition}
        />
      ) : (
        <MonacoEditor
          language="yaml"
          value={toYAML(definition)}
          onChange={(v) => setDefinition(parseYAML(v))}
        />
      )}

      <aside className="preview">
        <h2> 预览 </h2>
        <AgentPreview definition={definition} />
      </aside>

      <footer>
        <button onClick={() => testAgent(definition)}> 测试 </button>
        <button onClick={() => saveAgent(definition)}> 保存 </button>
        <button onClick={() => deployAgent(definition)}> 部署 </button>
      </footer>
    </div>
  )
}

function VisualEditor({ definition, onChange }) {
  return (
    <div className="visual-editor">
      <section className="basic-info">
        <TextField label="ID" value={definition.id} />
        <TextField label=" 名称" value={definition.name} />
        <TextArea label=" 描述" value={definition.description} />
        <Select label=" 领域" options={domains} value={definition.domain} />
        <Select label=" 类别" options={categories} value={definition.category} />
      </section>

      <section className="capabilities">
        <h3> 能力配置 </h3>
        <CapabilitySelector selected={definition.capabilities} />
      </section>
```

```
      <section className="tools">
        <h3> 工具绑定 </h3>
        <ToolBindingEditor bindings={definition.tools} />
      </section>

      <section className="prompt">
        <h3>System Prompt</h3>
        <PromptEditor value={definition.systemPrompt} />
      </section>

      <section className="permissions">
        <h3> 权限规则 </h3>
        <PermissionEditor rules={definition.permissions} />
      </section>
    </div>
  )
}
```

Workflow Builder 组件

```
// components/WorkflowBuilder.tsx
import ReactFlow, { Controls, Background } from "reactflow"

export function WorkflowBuilder() {
  const [workflow, setWorkflow] = useState<WorkflowDefinition>(defaultWorkflow)
  const [nodes, setNodes] = useState<Node[]>([])
  const [edges, setEdges] = useState<Edge[]>([])

  // 从 workflow 生成 ReactFlow 节点
  useEffect(() => {
    const { nodes, edges } = workflowToFlow(workflow)
    setNodes(nodes)
    setEdges(edges)
  }, [workflow])

  return (
    <div className="workflow-builder">
      <aside className="toolbox">
        <h2> 步骤类型 </h2>
        <DraggableStep type="agent" label="Agent 调用" />
        <DraggableStep type="tool" label=" 工具调用" />
        <DraggableStep type="condition" label=" 条件分支" />
        <DraggableStep type="parallel" label=" 并行执行" />
        <DraggableStep type="human" label=" 人工介入" />
        <DraggableStep type="subworkflow" label=" 子工作流" />
      </aside>
```

```
      <main className="canvas">
        <ReactFlow
          nodes={nodes}
          edges={edges}
          onNodesChange={onNodesChange}
          onEdgesChange={onEdgesChange}
          onConnect={onConnect}
          nodeTypes={customNodeTypes}
        >
          <Controls />
          <Background />
        </ReactFlow>
      </main>

      <aside className="properties">
        <h2> 属性面板 </h2>
        {selectedNode && (
          <StepPropertyEditor
            step={selectedNode.data}
            onChange={updateStep}
          />
        )}
      </aside>

      <footer>
        <button onClick={() => validateWorkflow(workflow)}> 验证 </button>
        <button onClick={() => runWorkflow(workflow)}> 运行 </button>
        <button onClick={() => saveWorkflow(workflow)}> 保存 </button>
      </footer>
    </div>
  )
}
```

---

9. 场景模板系统

8.1 模板定义

```
// template/types.ts
export interface ScenarioTemplate {
  id: string
  name: string
  description: string
  icon: string

  // 模板分类
  category: TemplateCategory
  tags: string[]
```

```
  // 模板内容
  domain: string                    // 使用的领域
  agents: string[]                  // 预置 Agents
  tools: string[]                   // 预置 Tools
  workflows: string[]               // 预置 Workflows

  // 配置预设
  config: Partial<Config>

  // 初始化向导
  wizard?: TemplateWizard

  // 示例数据
  examples?: TemplateExample[]
}

export type TemplateCategory =
  | "development"     // 开发相关
  | "documentation"   // 文档相关
  | "analysis"        // 分析相关
  | "testing"         // 测试相关
  | "operations"      // 运维相关
  | "management"      // 管理相关
  | "custom"          // 自定义
```

8.2 预置模板

需求分析模板

```yaml
# templates/requirement-analysis.template.yaml
id: requirement-analysis
name: 需求分析工作站
description: 完整的需求分析工具集，支持从原始需求到结构化文档
icon:
category: analysis

domain: analysis

agents:
  - requirement-analyst      # 需求分析师
  - domain-expert           # 领域专家
  - stakeholder-simulator   # 利益相关者模拟

tools:
  - read
  - write
  - search
  - ask_user
```

```yaml
  - generate_document
  - create_diagram

workflows:
  - requirement-extraction  # 需求提取
  - use-case-analysis       # 用例分析
  - requirement-review      # 需求评审

config:
  defaultAgent: requirement-analyst
  permissions:
    read: allow
    edit:
      "*.md": allow
      "docs/**": allow

wizard:
  steps:
    - id: project-info
      title: 项目信息
      fields:
        - name: projectName
          label: 项目名称
          type: text
          required: true
        - name: projectType
          label: 项目类型
          type: select
          options: [Web 应用, 移动应用, 后端服务, SDK/库, 其他]
        - name: stakeholders
          label: 利益相关者
          type: multiselect
          options: [产品经理, 开发团队, QA 团队, 运维团队, 客户]

    - id: requirement-source
      title: 需求来源
      fields:
        - name: sourceType
          label: 来源类型
          type: select
          options: [用户访谈, 竞品分析, 产品愿景, 技术迁移, 其他]
        - name: initialDocs
          label: 初始文档
          type: file
          multiple: true

examples:
  - title: 电商订单系统
```

```
    description: 分析电商订单管理系统的需求
    input: |
      我们需要一个订单管理系统，支持用户下单、支付、物流跟踪、售后处理。
      预计日订单量 10 万单。
```

BUG 修复模板

```yaml
# templates/bug-fix.template.yaml
id: bug-fix
name: BUG 修复工作站
description: 从问题定位到验证的完整 BUG 修复流程
icon:
category: development

domain: coding

agents:
  - bug-investigator        # BUG 调查员
  - root-cause-analyst      # 根因分析师
  - fix-implementer         # 修复实施者
  - regression-tester       # 回归测试者

tools:
  - read
  - write
  - edit
  - bash
  - grep
  - lsp_goto_definition
  - lsp_find_references
  - git_log
  - git_blame

workflows:
  - bug-investigation       # BUG 调查
  - root-cause-analysis     # 根因分析
  - fix-implementation      # 修复实施
  - regression-test         # 回归测试

config:
  defaultAgent: bug-investigator

wizard:
  steps:
    - id: bug-info
      title: BUG 信息
      fields:
        - name: bugId
```

```
      label: BUG ID
      type: text
    - name: description
      label: 问题描述
      type: textarea
      required: true
    - name: reproduction
      label: 复现步骤
      type: textarea
    - name: expectedBehavior
      label: 期望行为
      type: textarea
    - name: actualBehavior
      label: 实际行为
      type: textarea
    - name: environment
      label: 环境信息
      type: textarea
```

特性开发模板

```yaml
# templates/feature-development.template.yaml
id: feature-development
name: 特性开发工作站
description: 从需求到上线的完整特性开发流程
icon:
category: development

domain: coding

agents:
  - requirement-analyst
  - tech-architect
  - build
  - code-reviewer
  - qa-engineer
  - docs-writer

workflows:
  - requirement-to-dev     # 需求到开发
  - code-review            # 代码审查
  - test-coverage          # 测试覆盖
  - documentation          # 文档生成

config:
  defaultAgent: build
```

10. 权限与安全

9.1 权限模型扩展

```typescript
// permission/types.ts
export interface PermissionSystem {
  // 权限类别
  categories: {
    file: FilePermissions       // 文件操作
    code: CodePermissions       // 代码操作
    network: NetworkPermissions // 网络访问
    shell: ShellPermissions     // 命令执行
    data: DataPermissions       // 数据访问
    external: ExternalPermissions // 外部服务
  }

  // 权限规则
  rules: PermissionRule[]

  // 审计日志
  audit: AuditLog
}

// 文件权限
interface FilePermissions {
  read: PermissionAction | PatternPermission
  write: PermissionAction | PatternPermission
  delete: PermissionAction | PatternPermission
  execute: PermissionAction | PatternPermission
}

// 网络权限（新增）
interface NetworkPermissions {
  http: PermissionAction | DomainPermission
  websocket: PermissionAction | DomainPermission
  dns: PermissionAction
}

// 数据权限（新增）
interface DataPermissions {
  database: PermissionAction | TablePermission
  api: PermissionAction | EndpointPermission
  secrets: PermissionAction
}
```

9.2 沙箱执行

```typescript
// sandbox/executor.ts
export class SandboxExecutor {
```

```
  private container?: Container

  async execute(
    tool: ToolDefinition,
    params: unknown,
    ctx: ExecutionContext
  ): Promise<ToolResult> {
    // 检查是否需要沙箱
    if (this.requiresSandbox(tool)) {
      return this.executeInSandbox(tool, params, ctx)
    }

    return tool.execute(params, ctx)
  }

  private async executeInSandbox(
    tool: ToolDefinition,
    params: unknown,
    ctx: ExecutionContext
  ): Promise<ToolResult> {
    // 创建隔离环境
    const sandbox = await this.createSandbox({
      permissions: tool.requiredPermissions,
      resources: {
        cpu: "0.5",
        memory: "512m",
        timeout: 60000,
      },
      network: this.getNetworkPolicy(tool),
      filesystem: this.getFilesystemPolicy(tool),
    })

    try {
      return await sandbox.run(tool.execute, params, ctx)
    } finally {
      await sandbox.destroy()
    }
  }
}
```

9.3 审计日志

```
// audit/logger.ts
export class AuditLogger {
  async log(entry: AuditEntry): Promise<void> {
    await Storage.audit.insert({
      ...entry,
      timestamp: Date.now(),
```

```
      traceId: getCurrentTrace(),
    })

    // 实时事件
    Bus.publish(AuditEvent.Logged, entry)
  }
}

interface AuditEntry {
  type: AuditType
  actor: {
    type: "user" | "agent" | "system"
    id: string
    name: string
  }
  action: string
  resource: {
    type: string
    id: string
    path?: string
  }
  outcome: "success" | "failure" | "denied"
  details?: Record<string, unknown>
}
```

---

11. 实现路线图

11.1 阶段划分

实现路线图

```
Phase 1: 核心抽象 (4 周)
  Domain 领域抽象
  Agent 注册系统重构
  Tool 注册系统重构
  配置系统扩展

Phase 2: Workflow 引擎 (4 周)
  Workflow 定义规范
  步骤执行器实现
  状态机管理
  错误处理与恢复

Phase 3: Swarm 蜂群系统 (5 周)  新增
  Agent 动态创建/销毁
```

消息系统 (create + send 原语)
Runner Pool 运行时
拓扑管理与可视化
动态扩缩容策略

Phase 4: Web 管理系统 (6 周)
Admin API 设计与实现
Agent Builder UI
Tool Builder UI
Workflow Builder UI
Swarm Graph 可视化 新增
监控 Dashboard

Phase 5: 场景模板 (4 周)
模板系统实现
预置模板开发
需求分析模板
BUG 修复模板
特性开发模板
文档撰写模板
模板市场 MVP

Phase 6: 安全与优化 (4 周)
权限系统扩展
沙箱执行环境
审计日志
性能优化

11.2 里程碑

里程碑	时间	交付物
M1	Week 4	核心抽象层，可通过配置注册自定义 Agent/Tool
M2	Week 8	Workflow 引擎，支持 YAML 定义工作流
M3	Week 13	Swarm 蜂群系统，支持动态创建/协作/扩容 ★
M4	Week 19	Web 管理界面 MVP + Swarm 可视化
M5	Week 23	4 个场景模板可用
M6	Week 27	完整安全体系，生产就绪

---

12. API 设计

12.1 Agent SDK

```
// @opencode-ai/agent-sdk
import { defineAgent, AgentContext } from "@opencode-ai/agent-sdk"
```

```typescript
// 定义 Agent
export const myAgent = defineAgent({
  id: "my-agent",
  name: "My Agent",
  domain: "custom",
  category: "specialist",

  // 能力声明
  capabilities: ["document.read", "document.write"],

  // 工具绑定
  tools: ["read", "write", "search"],

  // System Prompt
  systemPrompt: `You are a specialized agent for...`,

  // 生命周期钩子
  hooks: {
    beforeExecute: async (ctx: AgentContext, input: string) => {
      // 预处理
    },
    afterExecute: async (ctx: AgentContext, output: string) => {
      // 后处理
    },
  },
})
```

12.2 Tool SDK

```typescript
// @opencode-ai/tool-sdk
import { defineTool, ToolContext, z } from "@opencode-ai/tool-sdk"

// 定义 Tool
export const myTool = defineTool({
  id: "my-tool",
  name: "My Tool",
  category: "utility",

  // 参数定义
  parameters: z.object({
    input: z.string().describe("Input value"),
    options: z.object({
      format: z.enum(["json", "yaml", "text"]).default("json"),
    }).optional(),
  }),

  // 执行函数
```

```
  async execute(params, ctx: ToolContext) {
    // 请求权限
    await ctx.ask({
      permission: "custom",
      patterns: [params.input],
      metadata: {},
    })

    // 执行逻辑
    const result = await processInput(params.input, params.options)

    return {
      title: "Processed input",
      output: result,
      metadata: { format: params.options?.format },
    }
  },
})
```

12.3 Workflow SDK

```
// @opencode-ai/workflow-sdk
import { defineWorkflow, step, condition, parallel } from "@opencode-ai/workflow-sdk"

// 定义 Workflow
export const myWorkflow = defineWorkflow({
  id: "my-workflow",
  name: "My Workflow",

  inputs: z.object({
    data: z.string(),
  }),

  steps: [
    // Agent 步骤
    step.agent({
      id: "analyze",
      agent: "analyst",
      input: "Analyze: {{ inputs.data }}",
      output: "analysis",
    }),

    // 条件分支
    condition({
      id: "check",
      if: "steps.analyze.output.includes('critical')",
      then: [
        step.agent({
```

```
          id: "escalate",
          agent: "escalation-handler",
          input: "Handle critical issue: {{ steps.analyze.output }}",
        }),
      ],
      else: [
        step.tool({
          id: "log",
          tool: "log_event",
          params: { message: "{{ steps.analyze.output }}" },
        }),
      ],
    }),

    // 并行执行
    parallel({
      id: "notify",
      branches: [
        [step.tool({ id: "email", tool: "send_email", params: {} })],
        [step.tool({ id: "slack", tool: "send_slack", params: {} })],
      ],
    }),
  ],
})
```

12.4 Swarm SDK

```
// @opencode-ai/swarm-sdk
import { SwarmClient, defineSwarmAgent } from "@opencode-ai/swarm-sdk"

// 创建 Swarm 客户端
const swarm = new SwarmClient({
  workspaceId: "ws-123",
})

// 动态创建 Agent
const coder = await swarm.createAgent({
  role: "coder",
  tools: ["read", "write", "edit", "bash"],
  task: " 实现用户登录功能",
})

// 发送消息
await swarm.sendMessage({
  to: coder.id,
  content: " 请先分析 src/auth 目录的代码结构",
})
```

```
// 监听事件
swarm.on("agent.created", (agent) => {
  console.log(`新 Agent 创建: ${agent.role}(${agent.roleIndex})`)
})

swarm.on("message.created", (msg) => {
  console.log(`[${msg.senderRole}] ${msg.content}`)
})

swarm.on("topology.changed", (topology) => {
  renderTopologyGraph(topology)
})

// 定义自组织 Agent
export const autoScalingAgent = defineSwarmAgent({
  role: "task-coordinator",

  // 自动扩容规则
  scaling: {
    // 当消息队列超过阈值时创建子 Agent
    onHighLoad: async (ctx) => {
      const worker = await ctx.createAgent({
        role: "worker",
        task: " 处理积压任务",
      })
      await ctx.delegateTo(worker, ctx.pendingTasks)
    },

    // 子 Agent 空闲时自动回收
    idleTimeout: 300000, // 5 分钟
  },

  // 协作规则
  collaboration: {
    // 遇到不熟悉的任务时咨询专家
    onUnknownTask: async (ctx, task) => {
      const expert = await ctx.findAgent({
        capability: task.requiredCapability
      })
      if (expert) {
        return ctx.askFor(expert, task)
      } else {
        return ctx.createAgent({
          role: `${task.requiredCapability}-expert`,
          task: task.description,
        })
      }
    },
```

```
  },
})
```

---

13. 示例场景

13.1 需求分析场景

```
// 使用需求分析模板
const session = await platform.createSession({
  template: "requirement-analysis",
  inputs: {
    projectName: " 电商订单系统",
    projectType: "Web 应用",
  },
})

// 发送需求描述
await session.prompt(`
  我们需要开发一个订单管理系统，具体需求如下：

  1. 用户可以在线下单购买商品
  2. 支持多种支付方式（支付宝、微信、银行卡）
  3. 订单状态实时更新
  4. 支持物流跟踪
  5. 完善的售后流程（退款、退货、换货）

  预计日订单量 10 万单，需要考虑高并发场景。
`)

// Agent 自动执行需求分析流程
// 输出：结构化的需求文档、用例图、边界条件分析
```

13.2 BUG 修复场景

```
// 使用 BUG 修复模板
const session = await platform.createSession({
  template: "bug-fix",
  inputs: {
    bugId: "BUG-12345",
    description: " 用户登录后页面白屏",
    reproduction: `
      1. 打开登录页面
      2. 输入用户名密码
      3. 点击登录按钮
      4. 页面变白，无任何内容
    `,
    environment: "Chrome 120, macOS 14",
```

```
  },
})

// Agent 自动执行调查流程
// 1. 分析日志和错误堆栈
// 2. 定位问题代码
// 3. 分析根因
// 4. 提出修复方案
// 5. 实施修复
// 6. 验证修复
```

13.3 PRD 撰写场景

```
// 使用文档模板
const session = await platform.createSession({
  template: "prd-writing",
  inputs: {
    productName: " 智能客服系统",
    targetUsers: [" 客服人员", " 终端用户"],
  },
})

await session.prompt(`
  请帮我撰写智能客服系统的 PRD，包括：
  - 产品背景和目标
  - 用户画像
  - 核心功能列表
  - 功能详细设计
  - 非功能性需求
  - 里程碑计划
`)

// Agent 自动生成完整 PRD 文档
```

13.4 Swarm 蜂群协作场景

```
// 使用 Swarm 模式处理复杂任务
const swarm = await platform.createSwarm({
  workspaceId: "project-alpha",
})

// 人类发起任务
await swarm.sendMessage({
  content: `
    我们需要重构整个认证模块：
    1. 分析现有代码结构
    2. 设计新的认证架构
    3. 实现 JWT + OAuth2 支持
```

```
    4. 编写单元测试
    5. 更新 API 文档
  `,
})

// 主 Agent 自动分解任务，动态创建子 Agent
//
// 运行时拓扑演化过程:
//
//            Human
//
//
//
//          Manager      主 Agent 分析任务
//           (1)
//
//
//
//
//
//
//  Anal    Arch    Coder        Test
//  (1-1)   (1-2)   (1-3)        (1-4)
//
//
//
//
//
//           JWT     OAuth    API      Coder 进一步分配
//           1-3-1   1-3-2    1-3-3
//
//
// Agent 间消息流:
// 1. Manager → Analyst: " 分析 src/auth 目录"
// 2. Analyst → Manager: " 分析完成，发现 3 个主要模块..."
// 3. Manager → Architect: " 基于分析结果设计新架构"
// 4. Architect → Manager: " 新架构设计完成..."
// 5. Manager → Coder: " 按照设计实现代码"
// 6. Coder 创建 3 个子 Agent 并行实现不同模块
// 7. 子 Agent 完成后向 Coder 汇报
// 8. Coder → Manager: " 代码实现完成"
// 9. Manager → Test: " 编写测试用例"
// ...

// 人类可随时介入任意 Agent
await swarm.chatWith("1-3-2", `
  OAuth2 的 scope 设计需要考虑哪些场景?
`)
```

```
// 查看实时拓扑
const topology = await swarm.getTopology()
console.log(topology.toASCII())

// 查看某个 Agent 的 LLM 历史（不再是黑箱）
const history = await swarm.getAgentHistory("1-3")
console.log(history.messages)
```

13.5 自动扩缩容场景

```
// 高负载场景：Agent 自动扩容
const orchestrator = await swarm.getAgent("manager")

// 模拟高并发请求
for (let i = 0; i < 100; i++) {
  await swarm.sendMessage({
    to: orchestrator.id,
    content: `处理任务 #${i}: ${taskDescriptions[i]}`,
  })
}

// Swarm 系统检测到队列积压，自动建议扩容
// Manager Agent 收到系统消息后，主动创建更多 Worker：
//
// [System] 检测到任务积压，建议创建子 Agent 分担工作
// [Manager] 好的，我将创建 5 个 Worker 来并行处理任务
//
// → create_agent(role: "worker-1", task: " 处理任务 #0-#19")
// → create_agent(role: "worker-2", task: " 处理任务 #20-#39")
// → create_agent(role: "worker-3", task: " 处理任务 #40-#59")
// → create_agent(role: "worker-4", task: " 处理任务 #60-#79")
// → create_agent(role: "worker-5", task: " 处理任务 #80-#99")

// 任务完成后，空闲的 Worker 自动回收
// [System] Worker-3 空闲超过 5 分钟，已自动终止
// [System] Worker-5 空闲超过 5 分钟，已自动终止
```

总结

本设计方案将 OpenCode 从编码助手扩展为通用 Agent 平台，融合 Swarm 蜂群智能思想，核心改进包括：

1. 领域抽象 - 支持多种业务领域（编码、分析、文档、测试等）
2. 注册系统 - Agent 和 Tool 可动态注册、热插拔
3. 工作流引擎 - 支持复杂业务流程编排
4. Swarm 蜂群系统 - 动态创建、自组织协作、弹性扩缩容 ★
5. Web 管理 - 可视化配置和监控界面

6. 场景模板 - 开箱即用的业务场景支持
7. 安全体系 - 细粒度权限控制和审计

Swarm 蜂群核心价值

Swarm 蜂群核心价值

极简原语
只需 create + send 两个原语即可构建任意复杂的多 Agent 系统

液态拓扑
Agent 按需"雇佣"，拓扑在运行时自演化，无需预设固定结构

扁平协作
人类可随时介入任意层级的 Agent，像微信聊天一样自然

弹性扩缩容
根据负载自动扩容（创建更多子 Agent），空闲时自动缩容

可观测性
实时可视化拓扑图、消息流、LLM 历史，Agent 不再是黑箱

通过这些改进，OpenCode 将成为一个可扩展、可配置、可组合、可自组织的通用 AI Agent 平台。