深入理解OpenClaw技术架构与实现原理（上）

原创 踏天 阿里云开发者 2026年3月19日 08:32 浙江

一、背景

最近OpenClaw如日中天，俨然已经是当下最热门并实用的个人助理。OpenClaw已经是我每日深度使用的效率工具，作为技术人，忍不住想系统性扒一下其技术架构与实现细节。当然了，本文也是通过与一堆Agent协作完成，包括OpenClaw、OpenCode、ClaudeCode、NotebookLLM、DeRisk等。

OpenClaw 在面向个人助手方向上，不仅仅体现在其灵活先进的智能体架构，还有其围绕个人助手方向的各种工具与生态的完整实现，是各类技术与工具的集大成者。 最让人惊讶的是，这些能力的基本全部通过AI-Coding实现，可以说彻底改变了软件开发的范式，而且清晰简洁的架构设计与表达，比传统人类编程的系统具有更高的标准，可以说是开启新的软件构建范式的开山之作，非常值得深入的研究。

由于OpenClaw涉及的技术点非常多，所以文章篇幅会显得很长。 这里建议大家按照感兴趣的模块，分模块阅读了解：

1.统一控制平面Gateway网关

2.Agentic Loop/Pi Loop

3.定时任务系统

4.工具系统

5.Channels

6.上下文管理

7.SubAgent子智能体

8.SandBox沙箱系统

9.记忆管理

10.Skills模块

11.Session管理

12.自进化机制

13.工作区与Agent路由

14.Nodes

15.安全策略

16.配置管理

二、OpenClaw总体架构

如下图所示为OpenClaw的技术架构图，其架构设计上是以**本地优先(Local-First)多端联动**为核心，建立一个高度灵活且可拓展的个人AI助手系统。其架构可以概括为一个以**Gateway(网关)**为核心的控制平面的分布式系统。

以下是对OpenClaw技术架构设计的详细解读:

1.核心控制平面: Gateway(网关)

Gateway是OpenClaw的心脏，充当系统的单一控制平面

- 功能职责: 负责管理会话(Sessions)、状态感知(Presence)、配置、定时任务(Cron)、网络钩子(Webhooks)以及控制界面(Control UI)和Canvas宿主
- 通信协议: 基于WebSocket(WS) 网络构建，为所有客户端、工具和事件提供统一的连接通道。
- 运行环境: 推荐在 Node ≥22 环境下运行，通常作为守护进程（Daemon）常驻后台。

2.智能体运行时: Pi Agent

Pi Agent是处理逻辑和生成回复的核心引擎:

- RPC模型: Pi Agent以RPC(远程过程调用)模式运行，支持工具流(Tool Streaming)和块流(Block Streaming)，确保响应的高效与实时性。
- 多智能体路由: 系统能够将来自不同频道、账户或同伴的输入路由到相互隔离的智能体(拥有独立的Workspace和会话)
- 会话模型: 提供main模式用户直接对话，并支持群组隔离、激活模式切换和队列管理。

3.连接生态: Channels(频道)

OpenClaw的一大特色是其极强的连接性，它将AI能力注入到用户已有的社交生态中。

- 多频道集成：原生支持包括 WhatsApp、Telegram、Slack、Discord、Google Chat、Signal、iMessage、Microsoft Teams、Matrix 以及 Zalo 等多种通讯平台
- 路由规则: 具备复杂的群组路由逻辑，包括提及门控（Mention gating）、回复标签处理以及针对不同频道的自动消息分块。

4.设备节点与伴侣应用: Nodes & Apps

通过将不同设备定义为“节点”，OpenClaw 实现了跨设备的硬件控制：

- 跨平台支持：包括 macOS 菜单栏应用、iOS 节点和 Android 节点。
- 硬件能力调用：通过 node.invoke 协议，智能体可以远程调用各节点上的硬件功能，如摄像头拍照/录码、屏幕录制、地理位置获取以及 macOS 特有的系统命令执行（system.run）。
- Voice Wake & Talk Mode：利用 ElevenLabs 等技术，在 macOS/iOS/Android 上提供始终在线的语音唤醒和连续对话能力。

5.工具与自动化：Tools & Skills

架构中集成了丰富的生产力工具：

- 浏览器控制：内置托管的 Chrome/Chromium 实例，支持快照、动作执行和文件上传。
- Live Canvas：基于 A2UI 构建的实时交互画布，允许智能体驱动视觉化的工作空间。
- 技能平台 (ClawHub)：提供技能注册表，支持捆绑技能、托管技能和工作区技能的自动搜索与安装。

6.安全与沙箱机制 (Security & Sandboxing)

由于 OpenClaw 会连接到真实的社交媒体和本地文件系统，安全性被置于重要位置：

- DM 配对策略：默认情况下，未知发送者必须通过配对码验证，bot 才会处理其消息，以防止不受信任的输入。
- Docker 沙箱：支持将 非主会话（如群组或外部频道）放入独立的 Docker 容器中运行，限制其对主机的访问权限，并对敏感工具（如浏览器、系统命令）进行黑白名单管理。

7.部署与远程访问

- 本地/远程灵活部署：Gateway 可以运行在本地或小型 Linux 实例上。
- 内网穿透：集成 Tailscale Serve/Funnel 或 SSH 隧道，使用户能够安全地从远程访问 Gateway 面板和 WebSocket 服务。

三、各系统模块详解

3.1 统一控制平面Gateway网关

3.1.1、核心定位

Gateway 是 OpenClaw 的**统一控制平面**，是一个 WebSocket 服务器，负责：

1.**消息路由** - 所有频道（Telegram、Discord、Slack 等）的消息路由

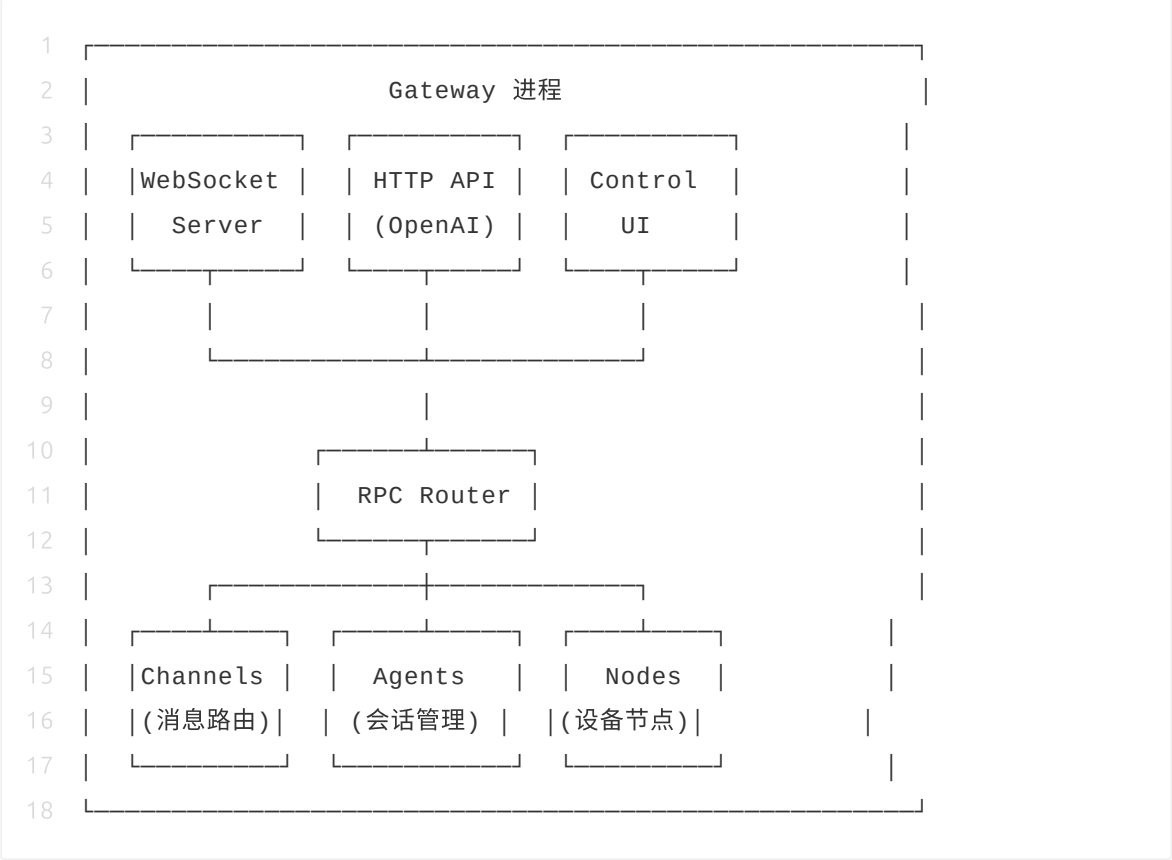
2.**会话管理** - Agent 会话的生命周期管理

3.**工具调用** - Agent 工具的执行协调

4.**节点通信** - iOS/Android 等移动节点的通信桥接

5.**HTTP API** - 提供 OpenAI 兼容的 REST API

3.1.2、架构模型



3.1.3、关键特性

特性	说明
单端口复用	WebSocket RPC + HTTP API + Control UI 共用一个端口（默认 18789）
协议版本化	客户端声明 minProtocol/maxProtocol，服务端拒绝不匹配的连接
角色分离	operator（控制面）和 node（能力节点）两种角色
作用域控制	细粒度的 scopes 控制（operator.read、operator.write、operator.admin 等）
设备认证	支持设备身份验证和配对机制
热重载	支持 hot/restart/hybrid 三种配置重载模式

3.1.4、协议机制

连接握手流程：

```
Gateway                              Client
   |                                   |
   |◀──── connect.challenge ──────────|  (可选：带 nonce 的挑战)
   |                                   |
   |──────── connect (req) ──────────▶|  携带 auth + role + scopes
   |                                   |
   |◀─────── hello-ok (res) ──────────|  返回 policy + 设备令牌
   |                                   |
   |◀──────── events ─────────────────|  持续推送状态变更
```

帧类型：

- Request: {type:"req", id, method, params}
- Response: {type:"res", id, ok, payload|error}
- Event: {type:"event", event, payload, seq?, stateVersion?}

3.1.5、认证模式

模式	使用场景
token	共享令牌认证（默认）
password	共享密码认证
trusted-proxy	反向代理认证（如 Pomerium）
device-token	设备身份认证（配对后自动获取）

安全强制：

- 非环回地址绑定必须启用认证
- 明文 ws:// 禁止连接非本机地址（CWE-319）

3.1.6、绑定模式

模式	地址	用途

loopback	127.0.0.1	默认，仅本机访问
lan	0.0.0.0	局域网访问
tailnet	Tailscale IP	Tailscale 网络
auto	自动选择	根据环境自动判断
custom	自定义地址	特定绑定需求

3.1.7、服务生命周期

macOS (launchd)：

```
openclaw gateway install   # 安装 LaunchAgent
openclaw gateway start     # 启动服务
openclaw gateway stop      # 停止服务
openclaw gateway restart   # 重启服务
```

Linux (systemd)：

```
openclaw gateway install
systemctl --user enable --now openclaw-gateway.service
```

3.1.8、配置热重载

模式	行为
off	不重载
hot	仅应用安全热更新
restart	需要重启时自动重启
hybrid	安全时热更新，必要时重启（默认）

3.1.9、关键配置项

```
{
  "gateway": {
    "port": 18789,
    "bind": "loopback",
    "mode": "local",
    "auth": {
      "mode": "token",
      "token": "your-token"
    },
    "tls": {
      "enabled": true,
      "certPath": "/path/to/cert.pem",
      "keyPath": "/path/to/key.pem"
    },
    "reload": {
      "mode": "hybrid",
      "debounceMs": 300
    }
  }
}
```

3.1.10、常用命令

```
# 启动网关
openclaw gateway --port 18789

# 查看状态
openclaw gateway status
openclaw gateway status --deep  # 深度检查

# 健康检查
openclaw gateway health
openclaw channels status --probe

# 发现局域网网关
openclaw gateway discover

# 查看日志
openclaw logs --follow
```

3.1.11、核心源码位置

模块	路径
CLI 入口	src/cli/gateway-cli/
客户端	src/gateway/client.ts
协议定义	src/gateway/protocol/
服务端 HTTP	src/gateway/server-http.ts
配置类型	src/config/types.gateway.ts

3.2 Agentic Loop / Pi Loop

如下图所示为OpenClaw的整个推理循环架构，也是构成整个系统执行的大脑思考核心。系统中所有的运行逻辑都由推理循环架构来控制，也就是AgenticLoop，OpenClaw的推理循环是一个**事件驱动**的架构：

1.**主循环** (run.ts) 负责错误处理、重试、profile轮换

2.**尝试层** (attempt.ts) 负责单次LLM调用的完整生命周期

3.**事件订阅** (subscribe.ts) 处理流式响应和工具调用

4.**工具循环** 由底层SDK自动管理，当模型返回tool_use时自动执行工具并继续调用。

下面是OpenClaw推理(inference/reasoning)循环实现的核心设计流程。

3.2.1 核心推理循环

主循环架构 (runEmbeddedPiAgent in run.ts:192)

```
runEmbeddedPiAgent()
  └── while (true) {  // 行538 - 主重试循环
        ├── 检查重试次数限制 (MAX_RUN_LOOP_ITERATIONS)
        ├── 调用 runEmbeddedAttempt()  // 单次推理尝试
        ├── 处理 context overflow → 自动压缩
        ├── 处理 auth failure → profile轮换
        ├── 处理 timeout → 重试或报错
        └── 成功则返回 payloads
      }
```

单次推理尝试 (runEmbeddedAttempt in run/attempt.ts:306)

```
runEmbeddedAttempt()
  ├── 1. 准备阶段
  │     ├── 创建 workspace 和 session
  │     ├── 解析 tools (createOpenClawCodingTools)
  │     ├── 构建 system prompt
  │     └── 创建 session manager
  │
  ├── 2. 会话初始化
  │     ├── createAgentSession()  // 行688
  │     ├── 设置 streamFn (LLM调用函数)
  │     └── 安装事件订阅器 subscribeEmbeddedPiSession()  // 行921
  │
  ├── 3. 执行推理
  │     ├── await activeSession.prompt(effectivePrompt)  // 行1180-1182
  │     │    └── 调用 LLM API(streamSimple/streamFn)
  │     │
  │     └── 事件流处理:
  │           ├── message_start/message_update/message_end  → handleMes
  │           ├── tool_execution_start/update/end          → handleTool
  │           └── agent_start/agent_end                    → handleAgent
  │
  └── 4. 返回结果
        ├── assistantTexts(生成的文本)
        ├── toolMetas(工具调用元数据)
        └── usage(token使用统计)
```

工具调用循环

工具调用由底层 SDK (@mariozechner/pi-coding-agent) 的 createAgentSession 自动管理。当模型返回 tool_use 时：

```
LLM Response(tool_use)
  └── SDK 自动执行:
        ├── handleToolExecutionStart()   // 记录工具开始
        │     └── emitAgentEvent({stream: "tool", data: {phase: "start"
        │
        ├── 执行工具函数
        │
        ├── handleToolExecutionUpdate()  // 流式更新
        │     └── emitAgentEvent({stream: "tool", data: {phase: "update
        │
        └── handleToolExecutionEnd()      // 工具完成
              ├── emitAgentEvent({stream: "tool", data: {phase: "result
              ├── 调用 after_tool_call hook
              └── SDK 自动将 tool_result 添加到消息历史
                    └── 继续调用 LLM(下一轮推理)
```

消息处理流程 (subscribeEmbeddedPiSession in pi-embedded-subscribe.ts:34)

```
事件分发 (createEmbeddedPiSessionEventHandler):
  ├── message_start    → handleMessageStart()
  │     └── 重置状态，准备新消息
  │
  ├── message_update   → handleMessageUpdate()
  │     ├── 处理 text_delta
  │     ├── 处理 thinking 块
  │     └── 调用 onPartialReply / onBlockReply
  │
  ├── message_end      → handleMessageEnd()
  │     ├── 提取最终文本
  │     ├── 处理 reasoning
  │     └── 推送最终回复
  │
  ├── tool_execution_* → handleToolExecution*()
  │     └── 跟踪工具状态，发送工具事件
  │
  └── agent_start/end  → handleAgentStart/End()
        └── 生命周期事件广播
```

3.2.2 关键调用链

```
用户消息
  ↓
runAgentTurnWithFallback() (agent-runner-execution.ts:72)
  ↓
runEmbeddedPiAgent() (pi-embedded-runner/run.ts:192)
  ↓ [while循环 - 重试]
runEmbeddedAttempt() (pi-embedded-runner/run/attempt.ts:306)
  ↓
createAgentSession() + activeSession.prompt()
  ↓ [LLM调用 + 工具循环]
subscribeEmbeddedPiSession() → 事件处理器
  ↓
onPartialReply / onBlockReply / onToolResult
  ↓
回复消息发送
```

3.2.3 LLM调用函数

实际的LLM API调用通过 streamFn 完成：

- 默认: streamSimple (来自 @mariozechner/pi-ai)
- Ollama: createOllamaStreamFn()
- 可通过 applyExtraParamsToAgent() 包装添加额外参数

3.3 定时任务系统

在OpenClaw中，定时任务是非常重要的一个基础设施，OpenClaw非常多的工作都是长任务，定时任务可以很好的满足这些长任务在后台单次或者周期性运行的诉求，同时与Heatbeat交互使得整个系统的交互更加拟人化，有些定时回复与交流，往往给人意想不到的拟人化体验。

3.3.1 、核心架构

```
┌──────────────────────────────────────────────────────────────────────┐
│                          CronService                                 │
│  (src/cron/service.ts)                                               │
├──────────────────────────────────────────────────────────────────────┤
│  ┌──────────────┐   ┌──────────────┐   ┌──────────────────────┐       │
│  │   Timer      │   │    Store     │   │   State              │       │
│  │  (timer.ts)  │   │  (store.ts)  │   │  (state.ts)          │       │
│  └──────┬───────┘   └──────┬───────┘   └──────────────────────┘       │
│         │                  │                                         │
│         ▼                  ▼                                         │
```

```
│  ┌──────────────────────────────────────────────────────┐  │
│  │              Jobs Collection(jobs.json)              │  │
│  └──────────────────────────────────────────────────────┘  │
└──────────────────────────────────────────────────────────────┘
```

3.3.2、调度类型

```
type CronSchedule =
  | { kind: "at"; at: string }           // 一次性任务，指定时间
  | { kind: "every"; everyMs: number; anchorMs?: number }  // 周期性任务
  | { kind: "cron"; expr: string; tz?: string; staggerMs?: number }  //
```

支持的

1.**at** - 一次性任务，执行后自动禁用

2.**every** - 固定间隔执行

3.**cron** - 标准cron表达式，支持时区和stagger

3.3.3、定时器机制

核心实现在 src/cron/service/timer.ts:

```
const MAX_TIMER_DELAY_MS = 60_000;  // 最大延迟60秒
const MIN_REFIRE_GAP_MS = 2_000;    // 最小重触发间隔2秒

// 定时器armed函数
export function armTimer(state: CronServiceState){
  const nextAt = nextWakeAtMs(state);  // 计算下次唤醒时间
  const delay = Math.max(nextAt - now, 0);
  const clampedDelay = Math.min(delay, MAX_TIMER_DELAY_MS);

  state.timer = setTimeout(() => {
    void onTimer(state).catch(...);
  }, clampedDelay);
}
```

关键特性：

- 定时器最大延迟60秒，防止时钟漂移

- 支持并发运行控制 (maxConcurrentRuns)
- 错误指数退避 (30s → 1min → 5min → 15min → 60min)
- 自动清理卡住的任务 (2小时超时)

3.3.4、任务持久化机制

存储位置：

- 默认路径: ~/.openclaw/cron/jobs.json
- 可通过配置 cron.store 自定义

存储格式：

```
type CronStoreFile = {
  version: 1;
  jobs: CronJob[];
};

type CronJob = {
  id: string;
  name: string;
  enabled: boolean;
  schedule: CronSchedule;
  sessionTarget: "main" | "isolated";
  payload: CronPayload;
  state: CronJobState;  // 运行时状态
  // ...
};
```

持久化流程：

1.原子写入（临时文件 + rename）

2.自动备份

3.支持热重载（文件修改时间检测）

运行日志：

- 路径: ~/.openclaw/cron/runs/<jobId>.jsonl

- 自动裁剪（默认2MB，保留2000行）

3.3.5、任务恢复机制

启动恢复流程 (src/cron/service/ops.ts):

```
export async function start(state: CronServiceState){
// 1. 加载存储
await ensureLoaded(state, { skipRecompute: true });

// 2. 清理卡住的任务
for (const job of jobs) {
if (job.state.runningAtMs) {
      job.state.runningAtMs = undefined;  // 清除过期标记
    }
  }

// 3. 运行错过的任务
await runMissedJobs(state);

// 4. 重新计算下次运行时间
  recomputeNextRuns(state);

// 5. 启动定时器
  armTimer(state);
}
```

3.3.6、任务执行类型

两种执行模式：

1.Main Session (sessionTarget: "main")

- 注入系统事件到主会话
- payload必须为 { kind: "systemEvent", text: string }

2.Isolated Agent (sessionTarget: "isolated")

- 独立agent会话执行
- payload必须为 { kind: "agentTurn", message: string, ... }
- 支持模型覆盖、thinking模式、超时设置

超时控制：

```
export async function executeJobCoreWithTimeout(state, job){
  const jobTimeoutMs = resolveCronJobTimeoutMs(job);
  return await Promise.race([
    executeJobCore(state, job, abortSignal),
    new Promise((_, reject) => {
      timeoutId = setTimeout(() => {
        abortController.abort();
        reject(new Error("cron: job execution timed out"));
      }, jobTimeoutMs);
    }),
  ]);
}
```

3.3.7、与Heartbeat的集成

定时任务通过Heartbeat机制唤醒agent：

```
// src/gateway/server-cron.ts
const cron = new CronService({
  enqueueSystemEvent: (text, opts) => {
    enqueueSystemEvent(text, { sessionKey, contextKey });
  },
  requestHeartbeatNow: (opts) => {
    requestHeartbeatNow({ reason, agentId, sessionKey });
  },
  runHeartbeatOnce: async (opts) => {
    return await runHeartbeatOnce({ cfg, reason, agentId, sessionKey })
  },
  // ...
});
```

Wake模式：

- next-heartbeat - 等待下次心跳执行
- now - 立即触发心跳

3.3.8、Webhook通知

支持任务完成后的Webhook回调：

```
if (webhookTarget && evt.summary) {
  await fetch(webhookTarget.url, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${webhookToken}`,
    },
    body: JSON.stringify(evt),
  });
}
```

3.3.9、CLI命令

```
openclaw cron status           # 查看调度器状态
openclaw cron list             # 列出任务
openclaw cron add              # 添加任务
openclaw cron edit             # 编辑任务
openclaw cron remove <id>      # 删除任务
openclaw cron run <id>         # 手动触发任务
```

3.3.10、关键设计特点

1.**单一定时器设计** - 只维护一个定时器，基于最近任务的nextRunAtMs

2.**文件持久化** - JSON存储，支持跨进程共享

3.**错误隔离** - 单个任务失败不影响其他任务

4.**自动恢复** - 启动时检测并运行错过的任务

5.**并发控制** - 可配置最大并发任务数

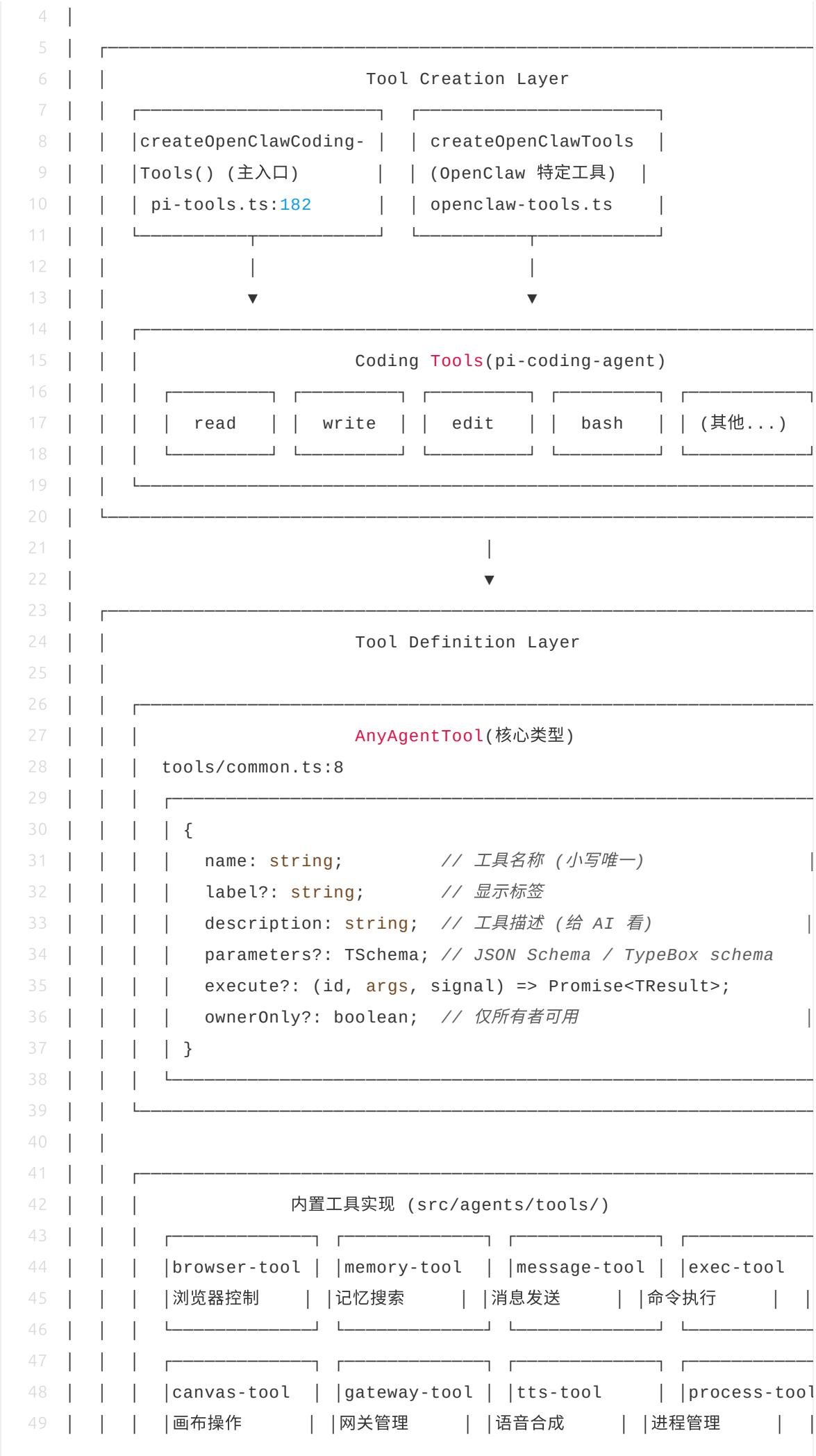
6.**进度追踪** - 完整的运行日志和状态跟踪

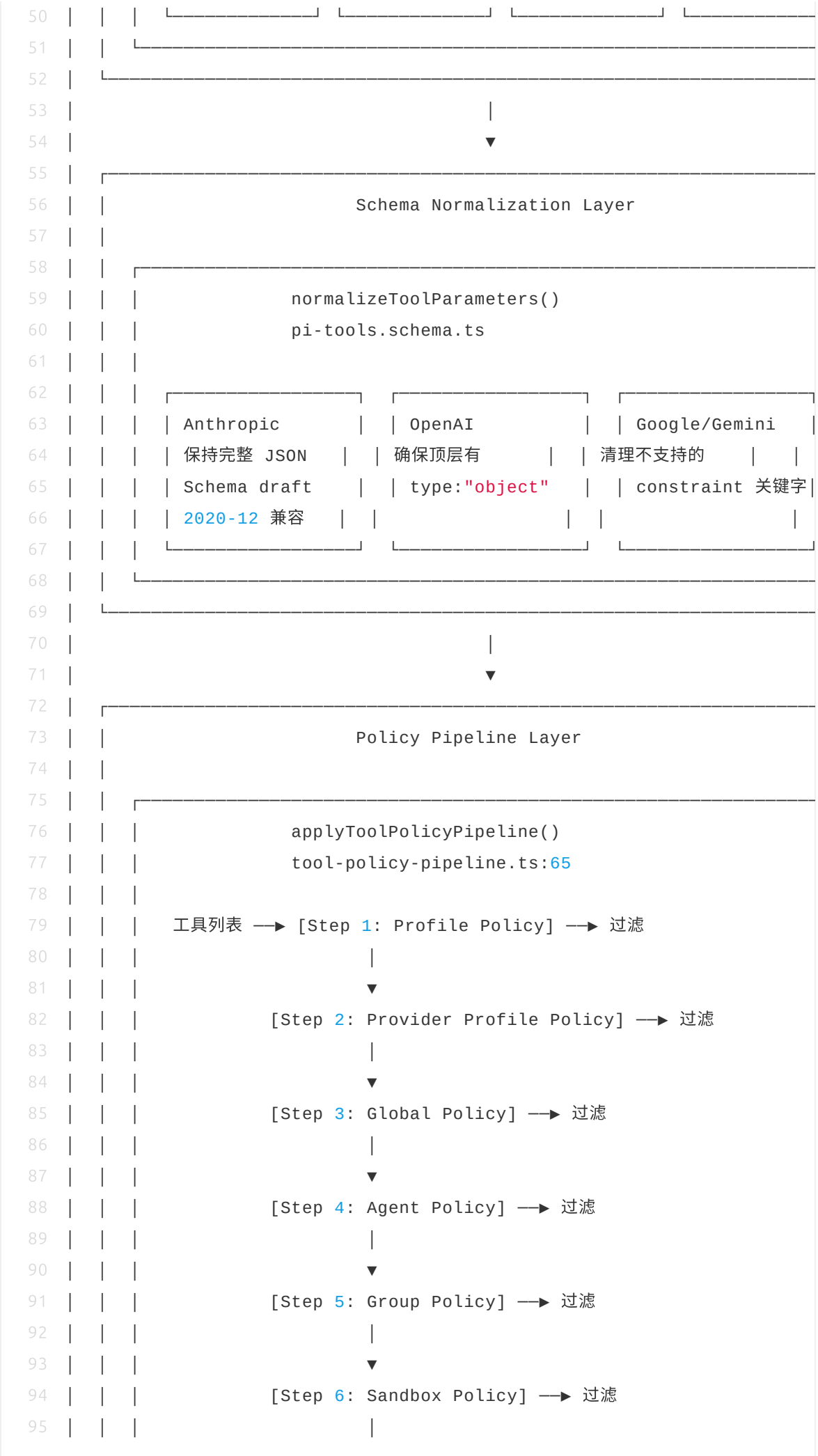
7.**Agent集成** - 可通过cron工具在agent中管理任务

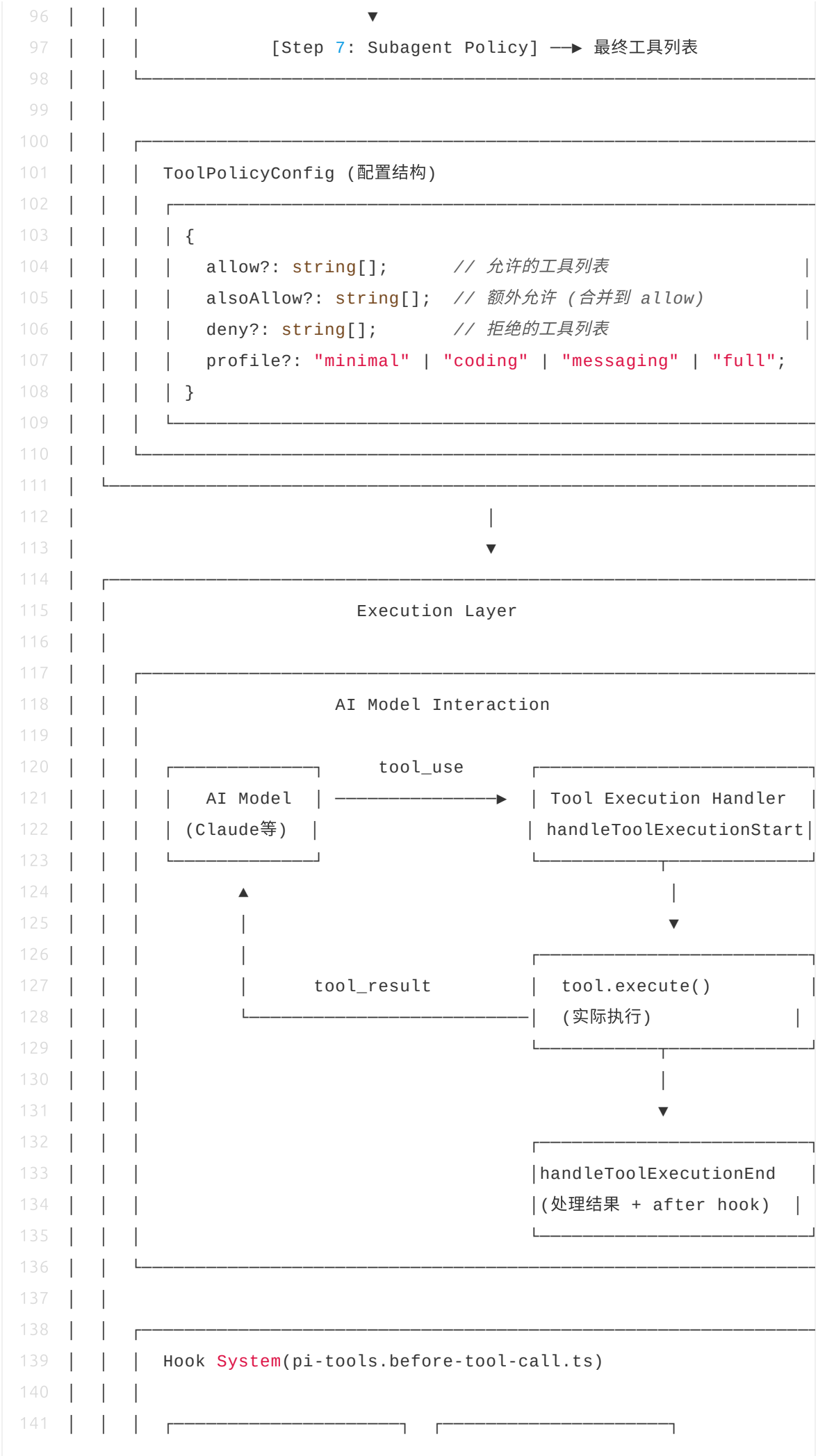
3.4 工具系统

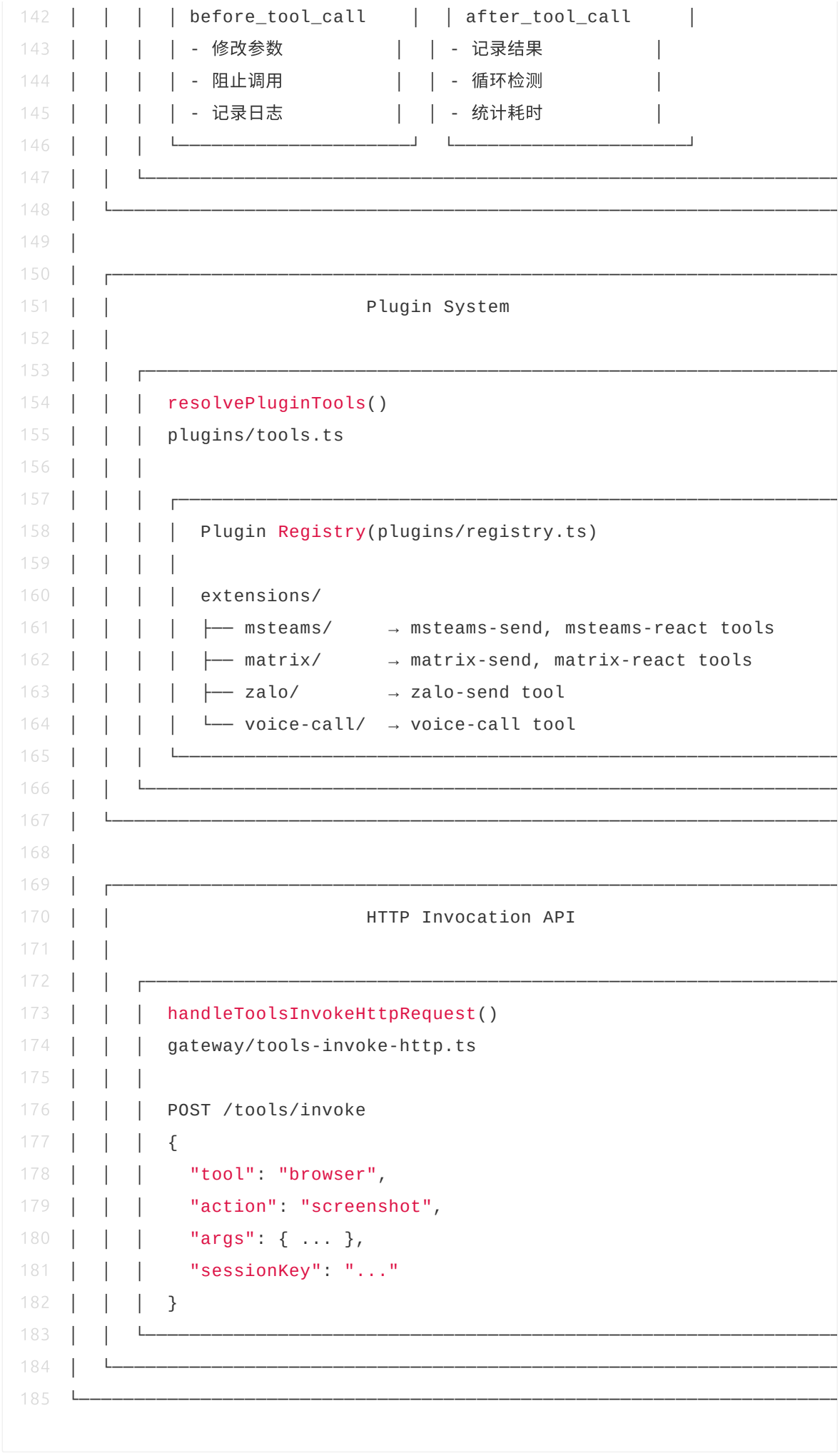
3.4.1 总体架构图

```
┌──────────────────────────────────────────────────────────────────────────┐
│                              OpenClaw Tool System                        │
├──────────────────────────────────────────────────────────────────────────┤
```


Tool Creation Layer
createOpenClawCoding-Tools() (主入口)
pi-tools.ts:182
createOpenClawTools
(OpenClaw 特定工具)
openclaw-tools.ts
Coding Tools(pi-coding-agent)
read
write
edit
bash
(其他...)
Tool Definition Layer
AnyAgentTool(核心类型)
tools/common.ts:8
{
name: string; // 工具名称 (小写唯一)
label?: string; // 显示标签
description: string; // 工具描述 (给 AI 看)
parameters?: TSchema; // JSON Schema / TypeBox schema
execute?: (id, args, signal) => Promise<TResult>;
ownerOnly?: boolean; // 仅所有者可用
}
内置工具实现 (src/agents/tools/)
browser-tool
浏览器控制
memory-tool
记忆搜索
message-tool
消息发送
exec-tool
命令执行
canvas-tool
画布操作
gateway-tool
网关管理
tts-tool
语音合成
process-tool
进程管理


Schema Normalization Layer
normalizeToolParameters()
pi-tools.schema.ts
Anthropic
保持完整 JSON Schema draft 2020-12 兼容
OpenAI
确保顶层有 type:"object"
Google/Gemini
清理不支持的 constraint 关键字
Policy Pipeline Layer
applyToolPolicyPipeline()
tool-policy-pipeline.ts:65
工具列表 ──▶ [Step 1: Profile Policy] ──▶ 过滤
[Step 2: Provider Profile Policy] ──▶ 过滤
[Step 3: Global Policy] ──▶ 过滤
[Step 4: Agent Policy] ──▶ 过滤
[Step 5: Group Policy] ──▶ 过滤
[Step 6: Sandbox Policy] ──▶ 过滤


[Step 7: Subagent Policy] ──▶ 最终工具列表
ToolPolicyConfig (配置结构)
{
allow?: string[]; // 允许的工具列表
alsoAllow?: string[]; // 额外允许（合并到 allow）
deny?: string[]; // 拒绝的工具列表
profile?: "minimal" | "coding" | "messaging" | "full";
}
Execution Layer
AI Model Interaction
AI Model
(Claude等)
tool_use
tool_result
Tool Execution Handler
handleToolExecutionStart
tool.execute()
(实际执行)
handleToolExecutionEnd
(处理结果 + after hook)
Hook System(pi-tools.before-tool-call.ts)


before_tool_call
- 修改参数
- 阻止调用
- 记录日志
after_tool_call
- 记录结果
- 循环检测
- 统计耗时
Plugin System
resolvePluginTools()
plugins/tools.ts
Plugin Registry(plugins/registry.ts)
extensions/
├── msteams/ → msteams-send, msteams-react tools
├── matrix/ → matrix-send, matrix-react tools
├── zalo/ → zalo-send tool
└── voice-call/ → voice-call tool
HTTP Invocation API
handleToolsInvokeHttpRequest()
gateway/tools-invoke-http.ts
POST /tools/invoke
{
"tool": "browser",
"action": "screenshot",
"args": { ... },
"sessionKey": "..."
}

3.4.2 核心组件详解

工具创建层

入口函数: createOpenClawCodingTools() (pi-tools.ts:182)

这是工具系统的主入口，负责：

- 解析工具策略配置
- 创建编码工具
- 创建 OpenClaw 特定工具
- 加载插件工具
- 应用策略过滤
- 规范化 Schema

```
// 工具创建流程
createOpenClawCodingTools() {
  1. resolveEffectiveToolPolicy() // 解析策略
  2. codingTools.flatMap()        // 处理基础编码工具
  3. createExecTool()             // 创建执行工具
  4. createOpenClawTools()        // 创建 OpenClaw 工具
  5. applyToolPolicyPipeline()    // 应用策略管道
  6. normalizeToolParameters()    // 规范化 Schema
  7. wrapToolWithBeforeToolCallHook() // 添加钩子
}
```

工具定义层

核心类型: AnyAgentTool (tools/common.ts:8)

```
type AnyAgentTool = AgentTool<any, unknown> & {
  ownerOnly?: boolean;  // 仅所有者可用标志
};
```

参数读取工具 (tools/common.ts):

- readStringParam() - 字符串参数
- readNumberParam() - 数字参数

- readStringArrayParam() - 字符串数组
- readReactionParams() - 表情反应参数

结果构造工具:

- jsonResult() - JSON 结果
- imageResult() - 图像结果

Schema 规范化层

函数: normalizeToolParameters() (pi-tools.schema.ts)

针对不同 AI 提供商的特殊处理：

提供商	处理方式
Anthropic	保持完整 JSON Schema draft 2020-12 兼容
OpenAI	确保顶层有 type: "object"
Google/Gemini	清理不支持的 format/约束关键字
所有	合并 anyOf/oneOf union schemas

策略管道层

函数: applyToolPolicyPipeline() (tool-policy-pipeline.ts:65)

管道步骤顺序（优先级从低到高）：

```
Profile Policy → Provider Profile → Global Policy → Agent Policy
    → Group Policy → Sandbox Policy → Subagent Policy
```

策略配置结构:

```
type ToolPolicyConfig = {
  allow?: string[];      // 白名单
  alsoAllow?: string[];  // 追加白名单
```

```
  deny?: string[];        // 黑名单
  profile?: "minimal" | "coding" | "messaging" | "full";
};
```

执行层

事件处理 (pi-embedded-subscribe.handlers.tools.ts):

- handleToolExecutionStart() - 记录开始时间，发出事件
- handleToolExecutionEnd() - 处理结果，运行 after hook

Hook 系统:

- before_tool_call - 可修改参数/阻止调用
- after_tool_call - 记录结果/循环检测

插件系统

工具解析: resolvePluginTools() (plugins/tools.ts)

插件工具注册：

```
type PluginToolRegistration = {
  pluginId: string;
  factory: OpenClawPluginToolFactory;
  names: string[];
  optional: boolean;
  source: string;
};
```

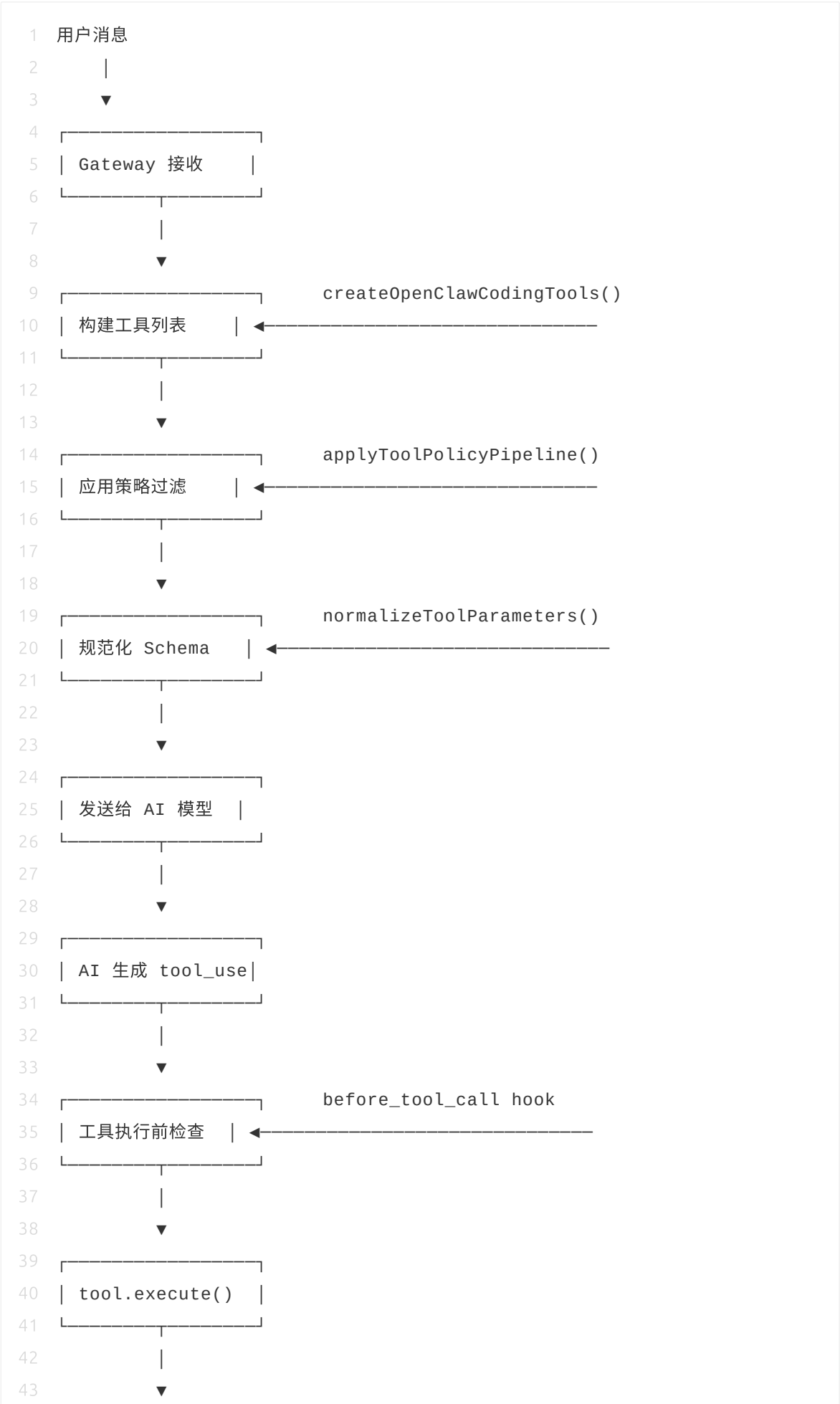
HTTP 调用 API

端点: POST /tools/invoke (gateway/tools-invoke-http.ts)

允许外部系统直接调用工具，支持：

- 认证验证
- 策略应用
- 结果返回

3.4.3 工具调用完整流程



```
┌──────────────────┐      after_tool_call hook
│ 工具执行后处理   │ ◄──────────────────────────────
└──────────┬───────┘
           │
           ▼
┌──────────────────┐
│ 返回 tool_result │
└──────────┬───────┘
           │
           ▼
┌──────────────────┐
│ AI 继续推理      │
└──────────────────┘
```

3.4.4 关键设计特点

1.**分层架构:** 创建 → 定义 → 规范化 → 策略 → 执行，职责清晰

2.**策略管道:** 多级策略叠加，支持精细控制

3.**Provider 适配:** 自动处理不同 AI 提供商的 Schema 差异

4.**插件扩展:** 插件工具与核心工具统一管理

5.**Hook 机制:** 支持工具调用前后拦截处理

6.**沙箱支持:** 隔离环境下的工具执行

3.5 Channels

Channels是OpenClaw进行社交生态连接最重要的设计，它将AI能力真正注入到了用户的社交与工作动线中。

3.5.1 核心架构

Channel抽象设计

```
┌──────────────────────────────────────────────────────────────────────┐
│                        ChannelPlugin 接口                            │
├──────────────────────────────────────────────────────────────────────┤
│  id: ChannelId                                                       │
│  meta: ChannelMeta (label, docsPath, aliases)                        │
│  capabilities: ChannelCapabilities (chatTypes, polls, threads...)    │
├──────────────────────────────────────────────────────────────────────┤
```

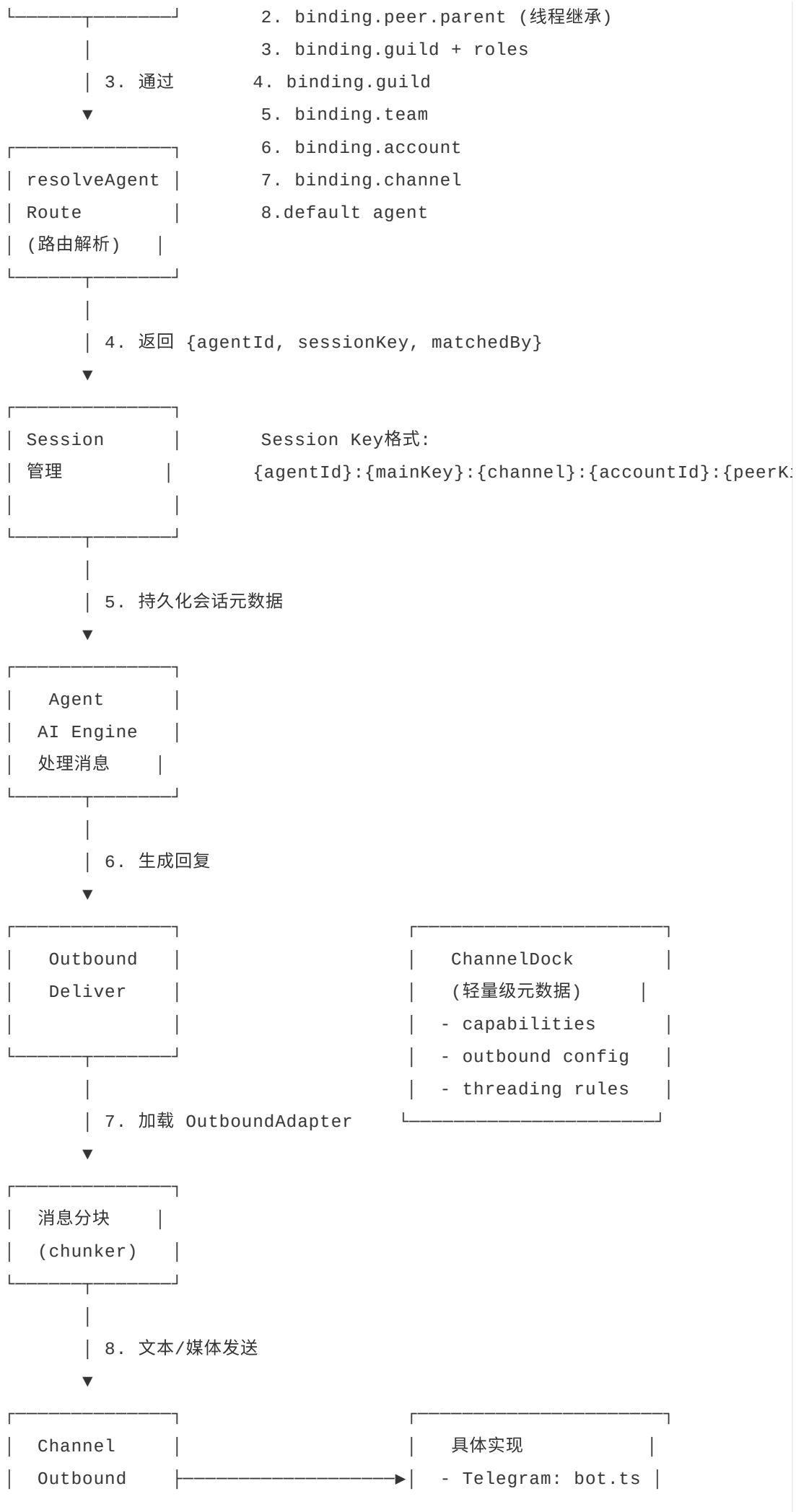
```
|                    12个独立适配器（职责分离）                              |
├──────────────────────────────────────────────────────────────────────────┤
| config       | 账户配置管理        | outbound    | 消息发送        |
| setup        | 账户设置流程        | status      | 状态探测        |
| gateway      | Gateway生命周期     | security    | 安全策略        |
| pairing      | 配对管理            | groups      | 群组管理        |
| threading    | 线程处理            | mentions    | 提及解析        |
| messaging    | 消息扩展            | directory   | 目录查询        |
| resolver     | 路由解析            | actions     | 消息动作        |
├──────────────────────────────────────────────────────────────────────────┤
| 可选: onboarding, auth, heartbeat, agentTools                           |
└──────────────────────────────────────────────────────────────────────────┘
```

消息流转完整架构

```
                              消息流向图
================================================================================

INBOUND（入站）                                                    OUTBOUND（出

┌──────────────┐
| 外部平台      |
| Telegram     |
| Discord      |
| Slack        |
| WhatsApp     |
| iMessage     |
| MSTeams...   |
└──────┬───────┘
       |
       | 1. Webhook/Gateway Event
       ▼
┌──────────────┐       ┌──────────────────────────────────────────────┐
| Channel      |       |              PLUGIN REGISTRY                 |
| Monitor      |──────►|  - channels: PluginChannelRegistration[]     |
| (接收层)     |       |  - plugins: PluginRecord[]                   |
└──────┬───────┘       |  - Lazy Loading + Caching                    |
       |               └──────────────────────────────────────────────┘
       | 2. 去重 & 预处理
       ▼
┌──────────────┐
| Allowlist    |       优先级（从高到低）:
| 验证         |       1. binding.peer (精确用户/群组)
```


2. binding.peer.parent (线程继承)
3. binding.guild + roles
3. 通过
4. binding.guild
5. binding.team
6. binding.account
resolveAgent
Route
(路由解析)
7. binding.channel
8.default agent
4. 返回 {agentId, sessionKey, matchedBy}
Session
管理
Session Key格式:
{agentId}:{mainKey}:{channel}:{accountId}:{peerK
5. 持久化会话元数据
Agent
AI Engine
处理消息
6. 生成回复
Outbound
Deliver
ChannelDock
(轻量级元数据)
- capabilities
- outbound config
- threading rules
7. 加载 OutboundAdapter
消息分块
(chunker)
8. 文本/媒体发送
Channel
Outbound
具体实现
- Telegram: bot.ts

```
| Adapter      |                    |   - Discord: send.ts |
└──────────────┘                    |   - Slack: send.ts   |
                                    |   - WhatsApp: web    |
                                    └──────────────────────┘
```

插件生命周期

```
┌──────────────────────────────────────────────────────────────────────┐
│                        Channel 生命周期                              │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  1. 注册阶段                                                         │
│     └─ 插件通过 registerChannel() 注册到 PluginRegistry              │
│                                                                      │
│  2. 初始化阶段                                                       │
│     ├─ 轻量加载: getChannelDock() → 仅元数据                         │
│     └─ 完整加载: getChannelPlugin() → 完整插件                       │
│                                                                      │
│  3. 配置阶段                                                         │
│     ├─ SetupAdapter: resolveAccountId(), applyAccountConfig()        │
│     └─ ConfigAdapter: listAccountIds(), resolveAccount()             │
│                                                                      │
│  4. 运行阶段                                                         │
│     ├─ Gateway启动: startAccount() [可选]                            │
│     ├─ 消息接收: inbound handlers                                    │
│     ├─ 路由解析: resolveAgentRoute()                                 │
│     └─ 消息发送: OutboundAdapter.sendText/sendMedia()                │
│                                                                      │
│  5. 监控阶段                                                         │
│     ├─ StatusAdapter: probeAccount(), auditAccount()                 │
│     └─ HeartbeatAdapter: checkReady()                                │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

核心适配器架构

```
┌──────────────────────────────────────────────────────────────────────┐
│                      ChannelPlugin 适配器矩阵                        │
├──────────────────┬───────────────────────────────────────────────────┤
│ 核心适配器       │ 职责                                              │
├──────────────────┼───────────────────────────────────────────────────┤
│ config           │ 账户配置: listAccountIds, resolveAccount,         │
│                  │ isEnabled, isConfigured, setAccountEnabled        │
```

```
├──────────────────┼──────────────────────────────────────────────────┤
| setup            | 账户设置: resolveAccountId, applyAccountConfig, |
|                  | validateAccountConfig                            |
├──────────────────┼──────────────────────────────────────────────────┤
| outbound         | 消息发送: sendText, sendMedia, sendPoll,         |
|                  | editText, deleteMessage, chunker                 |
├──────────────────┼──────────────────────────────────────────────────┤
| status           | 状态探测: probeAccount, auditAccount,            |
|                  | formatStatusSnapshot                             |
├──────────────────┼──────────────────────────────────────────────────┤
| gateway          | 生命周期: startAccount, stopAccount,             |
|                  | getRunningAccountIds, createInboundHandler       |
├──────────────────┼──────────────────────────────────────────────────┤
| security         | 安全策略: resolveDmPolicy, collectWarnings       |
├──────────────────┼──────────────────────────────────────────────────┤
| pairing          | 配对管理: resolvePairing, validatePairing        |
├──────────────────┼──────────────────────────────────────────────────┤
| 功能适配器       | 职责                                             |
├──────────────────┼──────────────────────────────────────────────────┤
| groups           | 群组: resolveRequireMention, resolveToolPolicy
├──────────────────┼──────────────────────────────────────────────────┤
| threading        | 线程: resolveReplyToMode, buildToolContext       |
├──────────────────┼──────────────────────────────────────────────────┤
| mentions         | 提及: resolveMentions, formatMention             |
├──────────────────┼──────────────────────────────────────────────────┤
| directory        | 目录: resolveUserDirectory, resolveGroupDirectory|
├──────────────────┼──────────────────────────────────────────────────┤
| resolver         | 路由: resolveAgentRoute (自定义路由逻辑)         |
├──────────────────┼──────────────────────────────────────────────────┤
| actions          | 动作: resolveMessageActions, handleAction        |
├──────────────────┼──────────────────────────────────────────────────┤
| messaging        | 消息扩展: resolveMessageMeta, formatMessage      |
└──────────────────┴──────────────────────────────────────────────────┘
```

目录结构映射

```
src/
├── channels/                    # Channel核心抽象
|   ├── plugins/                 # 插件系统
|   |   ├── types.plugin.ts      # ChannelPlugin接口定义
|   |   ├── types.adapters.ts    # 12个适配器接口
|   |   ├── types.core.ts        # Capabilities, Meta等
|   |   └── registry-loader.ts   # 加载器工厂
|   ├── dock.ts                  # 轻量级Dock (共享代码路径)
```

```
│   ├── registry.ts              # Channel ID规范化
│   ├── allow-from.ts            # Allowlist匹配
│   ├── channel-config.ts        # 配置匹配
│   └── session.ts               # 会话状态管理
│
├── routing/                     # 路由系统
│   ├── resolve-route.ts         # 路由解析（核心：291-443行）
│   ├── bindings.ts              # Agent绑定管理
│   └── session-key.ts           # Session Key构建
│
├── telegram/                    # Telegram实现
│   └── bot.ts                   # Bot创建，Update去重
├── discord/                     # Discord实现
│   ├── monitor.ts               # Gateway连接
│   ├── send.ts                  # 消息发送（Components V2）
│   └── ui.ts                    # UI容器
├── slack/                       # Slack实现
├── signal/                      # Signal实现
├── imessage/                    # iMessage实现
├── web/                         # WhatsApp Web实现
│   └── whatsapp-heartbeat.ts    # 心跳检测
│
├── infra/outbound/              # 出站消息基础设施
│   └── deliver.ts               # 消息发送流程
│
└── plugins/                     # 插件注册系统
    └── registry.ts              # PluginRegistry实现

extensions/                      # 扩展插件
├── msteams/                     # Microsoft Teams
│   └── src/channel.ts
├── matrix/                      # Matrix协议
│   └── src/channel.ts
├── zalo/                        # Zalo
└── voice-call/                  # 语音通话
```

3.5.2 关键设计要点

1.**分层抽象:** Application → Channel Abstraction → Implementation → Plugin Registry

2.**适配器模式:** 12个独立适配器，职责清晰分离

3.**性能优化:** Dock轻量加载、延迟加载、路由缓存、Update去重

4.**扩展性:** 新增Channel只需实现ChannelPlugin接口并注册

5.**安全隔离:** 每个channel独立的security、pairing、allowlist逻辑

3.6 上下文管理

3.6.1 核心概念

Context（上下文） = OpenClaw 在一次运行中发送给模型的**所有内容**，受模型的上下文窗口（token 限制）约束。如下为上下文构成要素，包括系统提示词汇、工具列表+描述、Skills列表(仅元数据)、工作区位置 + 时间 + 运行时数据。

注意：Context ≠ Memory。记忆可持久化到磁盘，Context 是模型当前窗口内的内容。

3.6.2 上下文窗口管理

上下文解析优先级：

1.**显式覆盖**contextTokensOverride → 直接使用

2.**配置参数**context1m: true (Anthropic 1M 模型) → 1,048,576 tokens

3.**模型注册表** → 从 models.json 或 provider catalog 发现

4.**配置文件覆盖** → models.providers.*.models[].contextWindow

5.**Fallback** → 使用传入的默认值

上下文窗口守卫：src/agents/context-window-guard.ts

```
CONTEXT_WINDOW_HARD_MIN_TOKENS = 16_000   // 低于此值阻断运行
CONTEXT_WINDOW_WARN_BELOW_TOKENS = 32_000 // 低于此值警告
```

三种检查结果：

- **shouldWarn:** 窗口 < 32K tokens（可能体验不佳）
- **shouldBlock:** 窗口 < 16K tokens（无法正常工作）
- **来源标记:** model | modelsConfig | agentContextTokens | default

3.6.3 上下文压缩

压缩机制

src/agents/compaction.ts

当会话接近或超过上下文窗口时，OpenClaw 自动触发压缩：

```
旧消息 ——→ LLM 总结 ——→ 紧凑摘要条目 ——→ 持久化到 JSONL
```

核心流程：

1.**Token 估算** (estimateMessagesTokens) - 计算当前消息总 token

2.**分块** (chunkMessagesByMaxTokens) - 按 token 限制分块

3.**摘要生成** (summarizeWithFallback) - 带重试的摘要生成

4.**历史裁剪** (pruneHistoryForContextShare) - 裁剪旧消息保持预算

自适应分块

```
BASE_CHUNK_RATIO = 0.4    // 基础分块比例
MIN_CHUNK_RATIO = 0.15    // 最小分块比例
```

```
SAFETY_MARGIN = 1.2        // 20% 缓冲补偿估算误差
```

当消息平均大小 > 上下文 10% 时，自动减小分块比例。

过大消息处理

```
isOversizedForSummary(msg, contextWindow)
// 单条消息 > 上下文 50% → 无法安全压缩
```

处理策略：

1.尝试完整压缩

2.失败 → 只压缩小消息，记录过大消息

3.最终回退 → 返回消息计数说明

3.6.4 上下文剪枝

与压缩的区别

特性	Compaction	Pruning
作用范围	整个历史	仅 toolResult 消息
持久化	✓ 写入 JSONL	✗ 仅内存
触发时机	接近窗口上限	每次请求前 (TTL 过期时)
内容变更	生成摘要	软修剪/硬清除

剪枝配置

src/agents/pi-extensions/context-pruning/settings.ts

```
DEFAULT_CONTEXT_PRUNING_SETTINGS = {
  mode: "cache-ttl",
  ttlMs: 5 * 60 * 1000,          // 5 分钟 TTL
  keepLastAssistants: 3,         // 保护最后 3 条助手消息
  softTrimRatio: 0.3,            // 上下文占用 > 30% 触发软修剪
```

```
    hardClearRatio: 0.5,          // 上下文占用 > 50% 触发硬清除
    minPrunableToolChars: 50_000,
    softTrim: {
      maxChars: 4_000,            // > 4K 字符触发软修剪
      headChars: 1_500,           // 保留头部 1500 字符
      tailChars: 1_500,           // 保留尾部 1500 字符
    },
    hardClear: {
      enabled: true,
      placeholder: "[Old tool result content cleared]",
    },
  }
```

剪枝执行流程

src/agents/pi-extensions/context-pruning/pruner.ts

```
1. 检查 TTL 是否过期
   ↓ 过期
2. 计算上下文占用比例
   ↓ 超过 softTrimRatio
3. 软修剪：对可修剪工具结果截取 head + tail
   ↓ 仍超过 hardClearRatio
4. 硬清除：替换为占位符
```

保护机制：

- 不修改用户/助手消息
- 跳过包含图片的 toolResult
- 保护 bootstrap 阶段消息（第一条用户消息之前）
- 保护最后 N 条助手消息之后的工具结果

3.6.5 工具结果上下文守卫

src/agents/pi-embedded-runner/tool-result-context-guard.ts

单条工具结果限制

```
SINGLE_TOOL_RESULT_CONTEXT_SHARE = 0.5  // 单条最多占上下文 50%
```

```
TOOL_RESULT_CHARS_PER_TOKEN_ESTIMATE = 2// 更保守的估算
```

执行逻辑

```
// 1. 每条工具结果限制
maxSingleToolResultChars = contextWindowTokens * 2 * 0.5

// 2. 总上下文预算 (75% headroom)
contextBudgetChars = contextWindowTokens * 4 * 0.75

// 3. 超预算时压缩最旧的工具结果
//    替换为 "[compacted: tool output removed to free context]"
```

3.6.6 运行时上下文注入

工作区文件注入

默认注入文件（如果存在）：

- AGENTS.md - 项目规则
- SOUL.md - 角色定义
- TOOLS.md - 工具指南
- IDENTITY.md - 身份信息
- USER.md - 用户偏好
- HEARTBEAT.md - 心跳状态
- BOOTSTRAP.md - 首次运行引导

截断配置：

```
{
  "agents": {
    "defaults": {
      "bootstrapMaxChars": 20000,       // 单文件上限
      "bootstrapTotalMaxChars": 150000  // 总上限
    }
  }
}
```

压缩后上下文刷新

src/auto-reply/reply/post-compaction-context.ts

压缩完成后，重新注入 AGENTS.md 中的关键章节：

- ## Session Startup
- ## Red Lines

目的：确保模型在压缩后仍遵循关键规则。

Sandbox 上下文

src/agents/sandbox/context.ts

为沙箱会话提供：

- 容器信息 (containerName, containerWorkdir)
- 工作区映射 (workspaceDir, agentWorkspaceDir)
- Docker 配置 (docker)
- 工具权限 (tools)
- 浏览器桥接 (browser, fsBridge)

3.6.7 检查与调试命令

命令	作用
/status	快速查看窗口占用率 + 会话设置
/context list	查看注入文件大小、工具 schema 大小
/context detail	详细分解各组件大小
/usage tokens	每次回复显示 token 使用量
/compact	手动触发压缩

3.6.8 关键配置汇总

```
{
  "agents": {
    "defaults": {
      // 上下文窗口
      "contextTokens": 200000,       // 硬性上限

      // Bootstrap 注入
      "bootstrapMaxChars": 20000,
      "bootstrapTotalMaxChars": 150000,

      // 压缩配置
      "compaction": {
        "mode": "auto",
        "targetTokens": 0.7          // 目标占用率
      },

      // 剪枝配置
      "contextPruning": {
        "mode": "cache-ttl",
        "ttl": "5m",
        "keepLastAssistants": 3,
        "softTrimRatio": 0.3,
        "hardClearRatio": 0.5
      }
    }
  },

  // 模型上下文窗口覆盖
  "models": {
    "providers": {
      "anthropic": {
        "models": [
          { "id": "claude-sonnet-4", "contextWindow": 200000 }
        ]
      }
    }
  }
}
```

3.7 SubAgent 架构详解

3.7.1 核心概念

SubAgent（子智能体）是从现有 Agent 运行中生成的后台独立运行实例。它们在独立的会话中执行任务，完成后将结果自动通告回请求者的聊天渠道。

关键特征

- **会话隔离：**每个 SubAgent 拥有独立的会话键 agent:<agentId>:subagent:<uuid>
- **后台执行：**非阻塞式运行，支持并行处理
- **结果通告：**完成时自动向父会话推送结果摘要
- **嵌套支持：**支持多层嵌套（最大5层深度，推荐2层）

3.7.2 架构组件

会话键系统

会话键格式与深度：

深度	会话键格式	角色	能否派生子智能体
0	agent:<id>:main	主智能体	总是可以
1	agent:<id>:subagent:<uuid>	子智能体（编排者）	仅当 maxSpawnDepth >= 2
2	agent:<id>:subagent:<uuid>:subagent:<uuid>	子子智能体（叶子工作者）	永远不能

深度计算逻辑（subagent-depth.ts:124-176）：

```
export function getSubagentDepthFromSessionStore(
  sessionKey: string | undefined | null,
  opts?: { cfg?: OpenClawConfig; store?: Record<string, SessionDepthEnt
): number {
  // 从会话键解析基础深度
  const fallbackDepth = getSubagentDepth(raw);

  // 读取会话存储中的 spawnDepth 字段
  const storedDepth = normalizeSpawnDepth(entry?.spawnDepth);

  // 或通过 spawnedBy 链递归计算父深度 + 1
  const parentDepth = depthFromStore(spawnedBy);
```

```
  return parentDepth + 1;
}
```

注册表

核心数据结构（subagent-registry.types.ts:6-35）：

```
export type SubagentRunRecord = {
  runId: string;                    // 运行标识符
  childSessionKey: string;          // 子会话键
  requesterSessionKey: string;      // 请求者会话键
  requesterOrigin?: DeliveryContext; // 请求者来源（渠道、账号等）
  task: string;                     // 任务描述
  cleanup: "delete" | "keep";       // 清理策略
  label?: string;                   // 显示标签
  model?: string;                   // 使用的模型
  runTimeoutSeconds?: number;       // 运行超时
  spawnMode?: SpawnSubagentMode;    // 运行模式
  createdAt: number;                // 创建时间
  startedAt?: number;               // 开始时间
  endedAt?: number;                 // 结束时间
  outcome?: SubagentRunOutcome;     // 运行结果
  suppressAnnounceReason?: "steer-restart" | "killed"; // 抑制通告原因
  endedReason?: SubagentLifecycleEndedReason; // 结束原因
};
```

核心职责：

1.**运行跟踪：**维护所有活跃和历史 SubAgent 运行记录

2.**生命周期监听：**通过 onAgentEvent 监听 lifecycle 事件（start/error/end）

3.**持久化：**运行记录持久化到磁盘，支持网关重启后恢复

4.**级联停止：**停止父运行时自动停止所有子运行

5.**孤儿检测：**恢复时检测并清理孤儿运行（缺失会话条目）

初始化流程（subagent-registry.ts:488-518）：

```
function restoreSubagentRunsOnce(){
  if (restoreAttempted) return;
```

```
  restoreAttempted = true;

  // 1. 从磁盘恢复运行记录
  const restoredCount = restoreSubagentRunsFromDisk({
    runs: subagentRuns,
    mergeOnly: true,
  });

  // 2. 协调孤儿运行
  if (reconcileOrphanedRestoredRuns()) {
    persistSubagentRuns();
  }

  // 3. 恢复未完成的工作
  for (const runId of subagentRuns.keys()) {
    resumeSubagentRun(runId);
  }
}
```

派生逻辑

核心流程（subagent-spawn.ts:166-550）：

```
┌─────────────────────────────────────────────────────────────┐
│ 1. 权限与深度检查                                            │
│    - 检查调用者深度 < maxSpawnDepth                          │
│    - 检查活跃子运行数 < maxChildrenPerAgent                  │
│    - 检查 agentId 允许列表                                   │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│ 2. 创建子会话                                                │
│    - 生成子会话键: agent:<id>:subagent:<uuid>                │
│    - 通过 sessions.patch 设置 spawnDepth                     │
│    - 设置模型和思考级别                                      │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│ 3. 线程绑定（可选）                                          │
│    - 调用 subagent_spawning 钩子准备线程绑定                 │
│    - 失败时回滚删除会话                                      │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│ 4. 启动子运行                                                │
```

```
|     - 构建子智能体系统提示                                                |
|     - 调用 gateway.agent() 启动运行                                       |
|     - 使用专属 lane: AGENT_LANE_SUBAGENT                                  |
└──────────────────────────────────────────────────────────────────────────┘
                                  ↓
┌──────────────────────────────────────────────────────────────────────────┐
| 5. 注册运行记录                                                           |
|     - 调用 registerSubagentRun() 注册到 registry                          |
|     - 开始等待完成                                                        |
|     - 触发 subagent_spawned 钩子                                          |
└──────────────────────────────────────────────────────────────────────────┘
```

系统提示构建（subagent-announce.ts:921-1025）：

```typescript
export function buildSubagentSystemPrompt(params: {
  requesterSessionKey?: string;
  childSessionKey: string;
  label?: string;
  task?: string;
  childDepth?: number;
  maxSpawnDepth?: number;
}) {
  const canSpawn = childDepth < maxSpawnDepth;

  const lines = [
    "# Subagent Context",
    `You are a **subagent** spawned by the ${parentLabel} for a specifi
    "",
    "## Your Role",
    `- You were created to handle: ${taskText}`,
    "- Complete this task. That's your entire purpose.",
    "",
    "## Rules",
    "1. **Stay focused** - Do your assigned task, nothing else",
    "2. **Complete the task** - Your final message will be automaticall
    "3. **Don't initiate** - No heartbeats, no proactive actions, no si
    "4. **Be ephemeral** - You may be terminated after task completion'
    "5. **Trust push-based completion** - Descendant results auto-annou
    "6. **Recover from compacted/truncated tool output** - Re-read with
    // ...
  ];

  if (canSpawn) {
    lines.push(
      "## Sub-Agent Spawning",
```

```
      "You CAN spawn your own sub-agents using `sessions_spawn`.",
      "Use the `subagents` tool to steer, kill, or check status.",
      "Your sub-agents will announce their results back to you automati
      // ...
    );
  } else if (childDepth >= 2) {
    lines.push(
      "## Sub-Agent Spawning",
      "You are a leaf worker and CANNOT spawn further sub-agents.",
      // ...
    );
  }
}
```

通告机制

核心流程（subagent-announce.ts:1053-1382）：

```
┌─────────────────────────────────────────────────────────────┐
│ 1. 等待运行结束                                              │
│    - 等待嵌入式运行完成（如果是嵌入式）                      │
│    - 调用 agent.wait 等待运行完成                            │
│    - 读取最新助手回复或工具结果                              │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│ 2. 构建通告消息                                              │
│    - 提取运行结果文本                                        │
│    - 计算运行统计（运行时间、token使用量、成本）             │
│    - 生成状态标签（成功/超时/失败）                          │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│ 3. 确定投递目标                                              │
│    - 检查线程绑定路由（bound 模式）                          │
│    - 调用 subagent_delivery_target 钩子（hook 模式）         │
│    - 回退到请求者来源（fallback 模式）                       │
└─────────────────────────────────────────────────────────────┘
                              ↓
┌─────────────────────────────────────────────────────────────┐
│ 4. 投递通告                                                  │
│    - 直接投递：调用 gateway.agent() 或 gateway.send()        │
│    - 队列投递：当请求者忙时入队等待                          │
│    - 嵌套处理：如果请求者是子智能体，向上冒泡                │
```

```
└──────────────────────────────────────────────────────────────┘
                              ↓
┌──────────────────────────────────────────────────────────────┐
| 5. 清理                                                      |
|    - 更新会话标签（如果有）                                  |
|    - 删除子会话（如果 cleanup: "delete"）                    |
|    - 触发 subagent_ended 钩子                                |
└──────────────────────────────────────────────────────────────┘
```

嵌套通告冒泡（subagent-announce.ts:1222-1253）：

```
// 如果请求者子智能体已结束，向上冒泡到其父
if (requesterIsSubagent && !isSubagentSessionRunActive(targetRequesterS
  // 检查父会话是否还存在
  const parentSessionEntry = loadSessionEntryByKey(targetRequesterSessi
  const parentSessionAlive = parentSessionEntry?.sessionId?.trim();

  if (!parentSessionAlive) {
    // 父会话已删除，回退到祖父
    const fallback = resolveRequesterForChildSession(targetRequesterSes
    if (fallback?.requesterSessionKey) {
      targetRequesterSessionKey = fallback.requesterSessionKey;
      targetRequesterOrigin = fallback.requesterOrigin;
      requesterDepth = getSubagentDepthFromSessionStore(targetRequester
      requesterIsSubagent = requesterDepth >= 1;
    }
  }
  // 如果父会话存活（只是没有活跃运行），继续向父注入
}
```

工具系统

工具定义（subagents-tool.ts:342-680）：

动作：

- list：列出活跃和最近的子运行
- kill <target>：停止指定的子运行（支持级联）
- steer <target> <message>：向运行中的子智能体发送指导消息

权限模型：

```
function resolveRequesterKey(params: {
  cfg: ReturnType<typeof loadConfig>;
  agentSessionKey?: string;
}): ResolvedRequesterKey {
  const callerSessionKey = resolveInternalSessionKey({ key: callerRaw,

  if (!isSubagentSessionKey(callerSessionKey)) {
    // 主智能体：查看自己的子运行
    return { requesterSessionKey: callerSessionKey, callerIsSubagent: f
  }

  const callerDepth = getSubagentDepthFromSessionStore(callerSessionKey
  const maxSpawnDepth = cfg.agents?.defaults?.subagents?.maxSpawnDepth

  if (callerDepth < maxSpawnDepth) {
    // 编排者子智能体：查看自己的子运行
    return { requesterSessionKey: callerSessionKey, callerIsSubagent: t
  }

  // 叶子子智能体：查看父的子运行（兄弟运行）
  const spawnedBy = callerEntry?.spawnedBy?.trim();
  return { requesterSessionKey: spawnedBy || callerSessionKey, callerIs
}
```

级联停止（subagents-tool.ts:276-318）：

```
async function cascadeKillChildren(params: {
  cfg: ReturnType<typeof loadConfig>;
  parentChildSessionKey: string;
  cache: Map<string, Record<string, SessionEntry>>;
}): Promise<{ killed: number; labels: string[] }> {
  const childRuns = listSubagentRunsForRequester(params.parentChildSess
  let killed = 0;
  const labels: string[] = [];

  for (const run of childRuns) {
    if (!run.endedAt) {
      const stopResult = await killSubagentRun({ cfg, entry: run, cache
      if (stopResult.killed) {
        killed += 1;
        labels.push(resolveSubagentLabel(run));
      }
    }

```

```
    // 递归停止孙运行
    const cascade = await cascadeKillChildren({
      cfg,
      parentChildSessionKey: run.childSessionKey,
      cache,
    });
    killed += cascade.killed;
    labels.push(...cascade.labels);
  }

  return { killed, labels };
}
```

3.7.3 配置与限制

核心配置项

```
{
  agents: {
    defaults: {
      subagents: {
        maxSpawnDepth: 2,           // 最大派生深度（1-5，默认1）
        maxChildrenPerAgent: 5,     // 每个会话最大活跃子运行数（1-20）
        maxConcurrent: 8,           // 全局并发上限（默认8）
        runTimeoutSeconds: 900,     // 默认超时（0=无超时）
        archiveAfterMinutes: 60,    // 自动归档时间（默认60分钟）
        model: "claude-3-haiku",    // 子智能体默认模型
        thinking: "medium",         // 默认思考级别
      },
    },
    list: [{
      agentId: "orchestrator",
      subagents: {
        allowAgents: ["*"],         // 允许派生任意 agentId
      },
    }],
  },
  tools: {
    subagents: {
      tools: {
        deny: ["gateway", "cron"],  // 工具黑名单
        allow: ["read", "exec"],    // 工具白名单
      },
    },
```

```
  },
}
```

工具策略

默认策略：

- 叶子子智能体：无会话工具（sessions_*）
- 编排者子智能体（深度1，当 maxSpawnDepth >= 2）：
  - 获得 sessions_spawn、subagents、sessions_list、sessions_history
  - 仍被拒绝：sessions_send、sessions_delete 等系统工具

工具过滤逻辑（docs:229-236）：

```
// 深度1编排者获得会话工具
if (isSubagentSessionKey(sessionKey) && depth === 1 && maxSpawnDepth >=
  allowedTools.push("sessions_spawn", "subagents", "sessions_list", "ses
}

// 深度2叶子工作者无会话工具
if (depth >= 2) {
  denySet.add("sessions_spawn");
}
```

3.7.4 插件钩子系统

钩子类型

钩子名称	触发时机	用途
subagent_spawning	派生前	准备线程绑定，验证权限
subagent_spawned	派生成功后	记录日志，更新UI状态
subagent_delivery_target	确定投递目标	自定义通告路由

subagent_ended	运行结束	清理资源，发送告别消息

Discord 扩展示例（extensions/discord/src/subagent-hooks.ts）：

```
// 注册钩子
export function registerDiscordSubagentHooks(){
  const hookRunner = getGlobalHookRunner();

  hookRunner.register("subagent_spawning", async (event, ctx) => {
    // 创建 Discord 线程并绑定到子智能体会话
    const thread = await createDiscordThread({
      channelId: event.requester.to,
      name: `Subagent: ${event.label || event.agentId}`,
    });

    return {
      status: "ok",
      threadBindingReady: true,
      threadId: thread.id,
    };
  });

  hookRunner.register("subagent_delivery_target", async (event, ctx) =>
    // 将通告路由到绑定的 Discord 线程
    const binding = getThreadBinding(event.childSessionKey);
    if (binding) {
      return {
        origin: {
          channel: "discord",
          to: binding.channelId,
          threadId: binding.threadId,
        },
      };
    }
    return { origin: event.requesterOrigin };
  });
}
```

3.7.5 并发与队列

Lane 系统（src/process/lanes.ts）

```
exportconstenumCommandLane {
```

```
  Main = "main",         // 主会话
  Cron = "cron",         // 定时任务
  Subagent = "subagent", // 子智能体
  Nested = "nested",     // 嵌套调用
}
```

并发控制：

- 子智能体使用专属 Subagent lane
- 全局并发上限由 maxConcurrent 控制（默认8）
- 每个会话的子运行数由 maxChildrenPerAgent 控制（默认5）

队列集成（subagent-announce.ts:645-702）

当请求者会话忙时，通告会被入队：

```
async function maybeQueueSubagentAnnounce(params): Promise<"steered" |
  const queueSettings = resolveQueueSettings({ cfg, channel, sessionEnt
  const isActive = isEmbeddedPiRunActive(sessionId);

  // 1. 尝试 steer 模式
  const shouldSteer = queueSettings.mode === "steer" || queueSettings.m
  if (shouldSteer) {
    const steered = queueEmbeddedPiMessage(sessionId, params.triggerMes
    if (steered) return"steered";
  }

  // 2. 尝试 followup/collect 模式
  const shouldFollowup =
    queueSettings.mode === "followup" ||
    queueSettings.mode === "collect" ||
    queueSettings.mode === "steer-backlog" ||
    queueSettings.mode === "interrupt";

  if (isActive && shouldFollowup) {
    enqueueAnnounce({
      key: buildAnnounceQueueKey(canonicalKey, origin),
      item: { announceId, prompt, sessionKey, origin },
      settings: queueSettings,
      send: sendAnnounce,
    });
    return"queued";
```

```
  }

  return"none";
}
```

3.7.6 认证与安全

认证继承（docs:196-204）

```
// 子智能体认证由 agentId 决定，而非会话类型
const authStore = loadAuthStore({ agentId: targetAgentId });

// 主智能体认证作为回退合并
const mainAuthStore = loadAuthStore({ agentId: requesterAgentId });
const mergedAuth = mergeAuthStores(authStore, mainAuthStore, {
  agentProfilesOverride: true, // 子智能体配置优先
});
```

允许列表（docs:122-128）

```
// 子智能体认证由 agentId 决定，而非会话类型
const authStore = loadAuthStore({ agentId: targetAgentId });
// 主智能体认证作为回退合并
const mainAuthStore = loadAuthStore({ agentId: requesterAgentId });
const mergedAuth = mergeAuthStores(authStore, mainAuthStore, {
  agentProfilesOverride: true, // 子智能体配置优先
});
```

3.7.7 典型场景

并行研究

```
用户："研究这三个主题并生成报告"
主智能体：
  - sessions_spawn(task: "研究主题A", label: "research-a")
  - sessions_spawn(task: "研究主题B", label: "research-b")
  - sessions_spawn(task: "研究主题C", label: "research-c")

[等待通告...]
research-a: ✅ 完成 - [结果摘要]
research-b: ✅ 完成 - [结果摘要]
research-c: ✅ 完成 - [结果摘要]
```

```
主智能体：综合结果生成最终报告
```

编排者模式（maxSpawnDepth=2）

```
用户："重构这个大型项目"
主智能体：
  - sessions_spawn(
      task："协调重构工作"，
      agentId："orchestrator"，
      label："refactor-coordinator"
    )

refactor-coordinator（深度1编排者）：
  - sessions_spawn(task："重构模块A"，label："worker-a")
  - sessions_spawn(task："重构模块B"，label："worker-b")
  - sessions_spawn(task："重构模块C"，label："worker-c")

[等待子运行通告...]
worker-a：✅ 完成
worker-b：✅ 完成
worker-c：✅ 完成

refactor-coordinator：综合结果，通知主智能体
主智能体：向用户报告完成
```

线程绑定会话

```
用户（在 Discord 线程中）："监控这个服务的性能"
主智能体：
  - sessions_spawn(
      task："启动性能监控"，
      thread：true，
      mode："session"
    )

[Discord 扩展创建专用线程]
[子智能体在专用线程中运行]

用户（在同一 Discord 线程）："当前状态如何？"
[消息路由到绑定的子智能体会话]
子智能体："当前 CPU 45%，内存 2.1GB..."

用户："/unfocus"
```

```
[解除线程绑定，后续消息路由回主智能体]
```

3.7.8 总结

SubAgent 架构的设计哲学：

1.**隔离与独立：**每个子智能体在独立会话中运行，拥有独立的上下文、token 配额和工具集

2.**推式通知：**结果自动通告，避免轮询开销和复杂性

3.**嵌套编排：**支持多层嵌套，实现复杂的编排模式

4.**资源可控：**通过深度限制、并发上限和工具策略控制资源消耗

5.**可扩展性：**通过插件钩子系统支持自定义行为（线程绑定、路由策略等）

关键设计决策：

- 使用会话键深度而非独立字段跟踪嵌套层级
- 通告机制而非返回值，支持异步和非阻塞语义
- 工具策略按深度区分，编排者获得管理工具而工作者专注任务
- 持久化注册表确保网关重启后不丢失运行状态

这套架构使 OpenClaw 能够高效处理并行任务、长时间运行作业和复杂的多智能体协作场景。

篇幅原因更多精彩内容在《深入理解OpenClaw技术架构与实现原理（下）》，请持续关注～

修改于2026年3月19日

深入理解OpenClaw技术架构与实现原理（下）

原创 踏天 阿里云开发者 2026年3月26日 08:32 浙江

阿里妹导读

本文是《 深入理解OpenClaw技术架构与实现原理（上）》的续篇，主要讲述从沙箱隔离到企业级智能体演进。

三、各系统模块讲解

3.8 SandBox 沙箱系统

Sandbox 是 OpenClaw 的 Docker 隔离层，用于在容器中执行 AI Agent 的工具操作，而非直接在主机上运行。

3.8.1 核心目的

- 限制工具执行（exec、read、write、edit 等）的安全边界
- 减少模型执行意外操作时的"爆炸半径"
- 提供可配置的隔离级别

3.8.2 关键文件结构

```
src/agents/sandbox/
├── types.ts           # 核心类型定义
├── config.ts          # 配置合并逻辑
├── context.ts         # 入口点 - 解析沙箱上下文
├── docker.ts          # Docker 容器管理
├── browser.ts         # 隔离浏览器容器
├── tool-policy.ts     # 工具允许/拒绝策略
├── validate-sandbox-security.ts  # 安全验证
├── fs-bridge.ts       # 文件系统操作桥接
└── prune.ts           # 容器自动清理
```

3.8.3 沙箱模式

模式	行为
"off"	不隔离，所有工具直接在主机运行
"non-main"	仅隔离非主会话（默认）
"all"	所有会话都隔离

3.8.4 容器作用域

作用域	容器数量
"session"	每个会话一个容器（默认）
"agent"	每个 Agent 一个容器
"shared"	所有会话共享一个容器

3.8.5 工作区访问权限

权限	挂载行为
"none"	完全隔离的工作区 ~/.openclaw/sandboxes

权限	挂载行为
"ro"	只读挂载 Agent 工作区到 /agent
"rw"	读写挂载到 /workspace

3.8.6 安全限制

禁止的绑定挂载：

- 系统路径：/etc、/proc、/sys、/dev、/root、/boot、/run
- Docker socket：/var/run/docker.sock
- 根文件系统：/

禁止的网络模式：

- host（绕过网络隔离）
- container:<id>（命名空间加入）

默认安全配置：

- 只读根文件系统 (readOnlyRoot: true)
- 无网络 (network: "none")
- 丢弃所有能力 (capDrop: ["ALL"])

3.8.7 工具策略层级

隔离时工具过滤顺序：

1. 全局工具策略
2. Agent 特定策略
3. **Sandbox 工具策略**（只能进一步限制）
4. 子 Agent 策略

默认允许的工具：exec、read、write、edit、apply_patch、image 等

默认禁止的工具：browser、canvas、nodes、cron、gateway 及所有消息通道

3.8.8 配置示例

```
{
  "agents": {
    "defaults": {
      "sandbox": {
        "mode": "non-main",
        "scope": "session",
        "workspaceAccess": "none",
        "docker": {
          "image": "openclaw-sandbox:bookworm-slim",
          "network": "none",
          "memory": "512m",
          "cpus": 1
        },
        "prune": {
          "idleHours": 24,
          "maxAgeDays": 7
        }
      }
    }
  }
}
```

3.8.9 CLI 命令

- openclaw sandbox list - 列出沙箱容器
- openclaw sandbox recreate - 强制重建容器
- openclaw sandbox explain - 调试当前配置

3.9 记忆管理

3.9.1 核心设计理念

OpenClaw 的记忆系统采用 "**文件即真相**" 的设计哲学：

- **存储介质：**纯 Markdown 文件（人类可读可编辑）
- **索引机制：**SQLite + 向量嵌入（机器可搜索）

- **工作模式：**文件优先，索引辅助

3.9.2 记忆文件布局

```
~/.openclaw/workspace/
├── MEMORY.md           # 长期记忆（精选、持久化）
└── memory/
    └── YYYY-MM-DD.md   # 每日记忆日志（append-only）
```

分层设计：

- **MEMORY.md：**长期记忆，存储决策、偏好、重要事实
- **memory/YYYY-MM-DD.md：**短期记忆，存储日常笔记、临时上下文

3.9.3 核心类型定义

```
type MemorySource = "memory" | "sessions";
type MemorySearchResult = {
  path: string;           // 文件路径
  startLine: number;      // 起始行号
  endLine: number;        // 结束行号
  score: number;          // 相关性得分
  snippet: string;        // 文本片段（~700 字符）
  source: MemorySource;   // 来源类型
  citation?: string;      // 引用标注
};
```

3.9.4 核心功能模块

记忆索引管理器 (MemoryIndexManager)

核心职责：

- 管理 SQLite 索引数据库
- 协调向量化搜索和全文搜索
- 监视文件变化并自动同步
- 处理 embedding 提供商

关键特性：

- 单例模式：通过缓存避免重复实例
- 混合搜索：向量 + BM25 关键词
- 自动同步：文件监视 + 定时刷新
- 会话集成：delta 更新机制

记忆搜索工具

- memory_search - 语义搜索
  - 基于向量嵌入的语义检索
  - 返回相关片段（路径 + 行号 + 得分）
  - 支持 MMR 去重和时间衰减
- memory_get - 定向读取
  - 按路径读取特定记忆文件
  - 支持行范围查询
  - 安全检查（防止路径穿越）

3.9.5 混合搜索机制

搜索流程

```
用户查询
    ↓
关键词提取（FTS）
向量嵌入
    ↓
并行搜索
    ├─ 向量搜索（语义相似）
    └─ BM25 搜索（关键词匹配）
    ↓
加权融合
    ↓
后处理
    ├─ 时间衰减
    └─ MMR 去重
```

```
    ↓
Top-K 结果
```

为什么需要混合搜索?

向量搜索优势：

- 理解语义："Mac Studio gateway host" ≈ "运行 gateway 的机器"
- 容错性强：拼写错误、同义词

向量搜索劣势：

- 对精确 token 弱：ID、代码符号、错误字符串

BM25 优势：

- 精确匹配：a828e60、memorySearch.query.hybrid
- 高信号 token

分数融合算法

```
finalScore = vectorWeight * vectorScore + textWeight * textScore
```

权重归一化为 1.0，默认 vectorWeight=0.7，textWeight=0.3

3.9.6 后处理机制

MMR（最大边际相关性）去重

目的：避免返回重复或高度相似的片段

算法：

```
score = λ × relevance - (1-λ) × max_similarity_to_selected
```

λ 参数：

- 1.0：纯相关性（不去重）
- 0.0：最大多样性（忽略相关性）

- 默认 0.7：平衡

```
查询："家庭网络设置"

无 MMR：
1. memory/2026-02-10.md (0.92) ← 路由器 + VLAN
2. memory/2026-02-08.md (0.89) ← 路由器 + VLAN（重复！）
3. memory/network.md (0.85)

有 MMR (λ=0.7)：
1. memory/2026-02-10.md (0.92) ← 路由器 + VLAN
2. memory/network.md (0.85) ← 参考文档（多样）
3. memory/2026-02-05.md (0.78) ← AdGuard DNS（多样）
```

时间衰减

目的：让近期记忆排名更高

公式：

```
decayedScore = score × e^(-λ × ageInDays)
```

半衰期：默认 30 天

- 今天：100%
- 7 天：84%
- 30 天：50%
- 90 天：12.5%

常青文件（不衰减）：

- MEMORY.md
- 非日期命名的文件（如 memory/projects.md）

3.9.7 Embedding 提供商

支持的提供商

1. OpenAI - text-embedding-3-small（默认）
2. Gemini - gemini-embedding-001
3. Voyage - voyage-4-large
4. Mistral - mistral-embed
5. Local - 本地模型

自动选择逻辑

```
if (local.modelPath 存在) return "local";
if (OpenAI key 可用) return "openai";
if (Gemini key 可用) return "gemini";
if (Voyage key 可用) return "voyage";
if (Mistral key 可用) return "mistral";
return disabled;
```

批量索引

- OpenAI Batch API：异步处理，折扣价格
- Gemini Batch：异步 embeddings 批量端点
- 并发控制：默认 2 个并发批处理任务

3.9.8 索引管理

SQLite 数据库结构

```
-- 文件表
CREATE TABLE files (
  path TEXT PRIMARY KEY,
  source TEXT,
  mtime INTEGER,
  hash TEXT
);
-- 分块表
CREATE TABLE chunks (
  id TEXT PRIMARY KEY,
  path TEXT,
  startLine INTEGER,
```

```
  endLine INTEGER,
  text TEXT,
  embedding BLOB,
  source TEXT
);
-- 向量表
CREATE VIRTUAL TABLE chunks_vec USING vec0(...);
-- 全文索引
CREATE VIRTUAL TABLE chunks_fts USING fts5(...);
-- Embedding 缓存
CREATE TABLE embedding_cache (
  hash TEXT PRIMARY KEY,
  embedding BLOB
);
```

索引更新策略

触发条件：

1. 会话启动时（sync.onSessionStart）

2. 搜索前（sync.onSearch）

3. 定时刷新（可配置间隔）

4. 文件变化（watcher，防抖 1.5s）

会话索引：

- 基于 delta 阈值：100KB 字节 或 50 条消息
- 异步更新，不阻塞搜索

3.9.9 自动记忆刷新

预压缩提示

触发时机：会话接近自动压缩时

机制：

```
tokenEstimate > contextWindow - reserveTokensFloor - softThresholdTokens
```

行为：

- 发送静默提示："Session nearing compaction. Store durable memories now."
- 模型可能回复（但通常 NO_REPLY）
- 每个压缩周期仅触发一次
- 只在可写工作空间执行

配置示例

```
{
  agents: {
    defaults: {
      compaction: {
        reserveTokensFloor: 20000,
        memoryFlush: {
          enabled: true,
          softThresholdTokens: 4000,
          systemPrompt: "Session nearing compaction...",
          prompt: "Write any lasting notes to memory/YYYY-MM-DD.md..."
        }
      }
    }
  }
}
```

3.9.10 最佳实践

何时写入记忆

- 长期记忆：决策、偏好、重要事实 → MEMORY.md
- 短期记忆：日常笔记、运行上下文 → memory/YYYY-MM-DD.md
- 用户明确要求："记住这个" → 立即写入

配置建议

小型语料库：

```
{
```

```
  "provider": "openai",
  "query": { "hybrid": { "enabled": false } }
}
```

大型语料库 + 每日笔记：

```
{
  "provider": "openai",
  "remote": { "batch": { "enabled": true } },
  "query": {
    "hybrid": {
      "enabled": true,
      "mmr": { "enabled": true, "lambda": 0.7 },
      "temporalDecay": { "enabled": true, "halfLifeDays": 30 }
    }
  }
}
```

完全本地：

```
{
  "provider": "local",
  "fallback": "none",
  "local": { "modelPath": "hf:ggml-org/embeddinggemma-300m-qat-q8_0-GGUF
}
```

3.9.11 性能优化

Embedding 缓存

```
{
  "cache": {
    "enabled": true,
    "maxEntries": 50000
  }
}
```

好处：

- 避免重复嵌入相同文本

- 加速增量更新
- 降低 API 成本

sqlite-vec 加速

```
{
  "store": {
    "vector": {
      "enabled": true,
      "extensionPath": "/path/to/sqlite-vec"
    }
  }
}
```

好处：

- 数据库内向量计算
- 避免加载所有 embedding 到 JS
- 查询速度提升显著

3.9.12 CLI 命令

```
# 查看状态
openclaw memory status
# 强制同步
openclaw memory sync --force
# 检查配置
openclaw config get agents.defaults.memorySearch
```

OpenClaw 的记忆管理系统是一个**生产级的混合记忆解决方案**，核心优势：

1. 简洁直观：Markdown 文件，人类可读可编辑

2. 强大灵活：语义搜索、混合检索、多种提供商

3. 自动化：自动索引、自动刷新、预压缩保存

4. 可扩展：插件架构、实验性后端

5. 隐私优先：本地存储、支持完全本地运行

这个设计体现了现代 AI Agent 记忆管理的最佳实践：**让文件成为真相，让索引成为加速器。**

3.10 Skills 模块详解

Skills 模块位于 src/agents/skills/，是 OpenClaw Agent 能力扩展的核心模块。

3.10.1 核心概念

Skill = 一个封装了特定能力的 Markdown 文件 (SKILL.md)，包含：

- YAML frontmatter（元数据）
- 使用指南/命令模板

3.10.2 目录结构

```
src/agents/skills/
├── types.ts            # 类型定义
├── config.ts           # 配置解析与过滤
├── workspace.ts        # 核心加载逻辑
├── frontmatter.ts      # SKILL.md 解析
├── filter.ts           # 技能过滤器
├── bundled-dir.ts      # 内置技能目录解析
├── bundled-context.ts  # 内置技能缓存
├── plugin-skills.ts    # 插件技能集成
├── refresh.ts          # 文件监听与版本刷新
├── env-overrides.ts    # 环境变量注入
├── serialize.ts        # 并发控制锁
├── tools-dir.ts        # 工具目录路径
└── skills-install.ts   # 技能安装器
```

3.10.3 Skill 加载优先级（从低到高）

```
// workspace.ts:369-388
extra → bundled → managed → agents-skills-personal → agents-skills-proje
```

来源	路径	说明
extra	config.skills.load.extraDirs	用户自定义目录

来源	路径	说明
bundled	skills/ (包内)	OpenClaw 内置技能
managed	~/.openclaw/skills	全局管理技能
agents-skills-personal	~/.agents/skills	个人 agents 目录
agents-skills-project	/.agents/skills	项目 agents 目录
workspace	/skills	项目技能 (最高优先)

3.10.4 核心类型

```
type SkillEntry = {
  skill: Skill;
  frontmatter: ParsedSkillFrontmatter;
  metadata?: OpenClawSkillMetadata;
  invocation?: SkillInvocationPolicy;
};
type OpenClawSkillMetadata = {
  always?: boolean;
  skillKey?: string;
  primaryEnv?: string;
  emoji?: string;
  os?: string[];
  requires?: {
    bins?: string[];
    env?: string[];
    config?: string[];
  };
  install?: SkillInstallSpec[];
};
type SkillSnapshot = {
  prompt: string;
  skills: Array<{ name, primaryEnv, requiredEnv }>;
  skillFilter?: string[];
  resolvedSkills?: Skill[];
  version?: number;
};
```

3.10.5 技能过滤逻辑

```
// config.ts:70-101
shouldIncludeSkill() 检查:
1. config.skills.entries[skillKey].enabled !== false
2. bundled allowlist 检查
3. 运行时资格评估:
   - OS 兼容性
   - 二进制依赖
   - 环境变量依赖
   - 配置路径检查
```

3.10.6 SKILL.md 结构示例

```
---
name: github
description: "GitHub operations via `gh` CLI..."
metadata:
  openclaw:
    emoji: "🐙"
    requires:
      bins: ["gh"]
    install:
      - id: brew
        kind: brew
        formula: gh
        bins: ["gh"]
---
# GitHub Skill
...使用说明...
```

3.10.7 关键导出 API

```
// skills.ts 导出
loadWorkspaceSkillEntries()      // 加载技能条目
buildWorkspaceSkillSnapshot()    // 构建快照（含 prompt）
buildWorkspaceSkillsPrompt()     // 生成 Agent prompt
filterWorkspaceSkillEntries()    // 过滤技能
buildWorkspaceSkillCommandSpecs() // 生成命令规格
syncSkillsToWorkspace()          // 同步到沙箱
applySkillEnvOverrides()         // 注入环境变量
```

3.10.8 安装支持类型

```
// types.ts:3-17
type SkillInstallSpec = {
  kind: "brew" | "node" | "go" | "uv" | "download";
  formula?: string;   // brew formula
  package?: string;   // npm/go/uv 包名
  module?: string;    // go module
  url?: string;       // download URL
  // ...
};
```

3.10.9 文件监听机制

```
// refresh.ts
ensureSkillsWatcher() // 监听 SKILL.md 变更
  → bumpSkillsSnapshotVersion() // 版本号递增
  → 事件通知 listeners
```

3.10.10 安全机制

1. 路径安全：resolveSandboxPath() 防止路径穿越

2. 环境变量安全：sanitizeSkillEnvOverrides() 过滤危险变量

3. 代码扫描：安装前 scanDirectoryWithSummary() 检查危险模式

3.11 Session 管理

3.11.1 整体架构设计

```
┌──────────────────────────────────────────────────────────────┐
│                        会话管理层级                          │
├──────────────────────────────────────────────────────────────┤
│  Session Key(身份标识)                                       │
│       ↓                                                      │
│  Session Entry(会话元数据)                                   │
│       ↓                                                      │
│  Session Store(持久化存储)                                   │
│       ↓                                                      │
│  Transcript File(对话历史)                                   │
└──────────────────────────────────────────────────────────────┘
```

3.11.2 Session Key 设计

1. 会话键格式规范

```
基础格式: agent:<agentId>:<rest>

示例:
- agent:main:main                    # 默认 agent 的主会话
- agent:ops:work                     # ops agent 的主会话
- agent:main:telegram:direct:user123 # Telegram DM 会话
- agent:main:discord:group:guild789  # Discord 群组会话
- agent:main:cron:daily-backup:run:uuid  # Cron 任务运行会话
- agent:main:subagent:child-session  # 子代理会话
```

2. 会话键解析（session-key-utils.ts:12-32）

```
function parseAgentSessionKey(sessionKey: string | undefined | null): Pa
  // 1. 规范化：小写、去空格
  const raw = (sessionKey ?? "").trim().toLowerCase();
  // 2. 验证最小结构 (agent:id:rest)
  const parts = raw.split(":").filter(Boolean);
  if (parts.length < 3 || parts[0] !== "agent") return null;
  // 3. 提取 agentId 和 rest
  return { agentId: parts[1], rest: parts.slice(2).join(":") };
}
```

3. 会话类型判断

类型	识别模式	示例
direct	包含 :direct: 或 :dm:	agent:main:telegram:direct:user123
group	包含 :group:	agent:main:discord:group:guild789
channel	包含 :channel:	agent:main:slack:channel:C123
cron	rest 以 cron: 开头	agent:main:cron:backup
subagent	包含 :subagent:	agent:main:subagent:child

类型	识别模式	示例
thread	包含 :thread: 或 :topic:	agent:main:discord:group:123:thread:456

3.11.3 Session Entry 数据结构

核心字段(**config/sessions/types.ts 相关)**

```
type SessionEntry = {
  // === 身份标识 ===
  sessionId: string;        // UUID，关联对话历史文件
  sessionFile?: string;     // 自定义会话文件路径

  // === 模型配置 ===
  model?: string;           // 当前运行时模型
  modelProvider?: string;   // 当前运行时提供商
  modelOverride?: string;   // 用户指定的模型覆盖
  providerOverride?: string;// 用户指定的提供商覆盖
  contextTokens?: number;   // 上下文窗口大小

  // === Token 统计 ===
  totalTokens?: number;     // 总 token 数
  inputTokens?: number;     // 输入 token
  outputTokens?: number;    // 输出 token
  totalTokensFresh?: boolean; // token 数据是否新鲜

  // === 投递路由 ===
  lastChannel?: string;     // 最后使用的通道
  lastTo?: string;          // 最后发送目标
  lastAccountId?: string;   // 最后账号 ID
  lastThreadId?: string;    // 最后线程 ID
  deliveryContext?: DeliveryContext; // 投递上下文

  // === 会话状态 ===
  updatedAt?: number;       // 最后更新时间戳
  systemSent?: boolean;     // 系统消息是否已发送
  abortedLastRun?: boolean; // 上次运行是否中断

  // === 行为配置 ===
  thinkingLevel?: string;   // 思考级别
  verboseLevel?: string;    // 详细级别
  reasoningLevel?: string;  // 推理级别
  sendPolicy?: string;      // 发送策略

```

```
  // === 群组/频道元数据 ===
  chatType?: "direct" | "group" | "channel";
  channel?: string;          // 来源通道
  subject?: string;          // 主题/名称
  groupId?: string;          // 群组 ID
  groupChannel?: string;     // 频道名称
  space?: string;            // 空间/工作区

  // === 显示 ===
  displayName?: string;      // 显示名称
  label?: string;            // 用户标签

  // === 线程/派生 ===
  forkedFromParent?: boolean; // 是否从父会话派生
  spawnedBy?: string;        // 创建来源

  // === 压缩状态 ===
  compactionCount?: number; // 压缩次数
  memoryFlushAt?: number;   // 内存刷新时间
};
```

3.11.4 会话生命周期管理

1. 会话初始化流程(auto-reply/reply/session.ts:165-579)

```
┌─────────────────────────────────────────────────────────────────┐
│                        会话初始化流程                            │
├─────────────────────────────────────────────────────────────────┤
│  1. 解析 Agent ID                                               │
│     resolveSessionAgentId({ sessionKey, config })               │
│                                                                 │
│  2. 加载会话存储                                                 │
│     loadSessionStore(storePath, { skipCache: true })            │
│                                                                 │
│  3. 检查重置触发器                                               │
│     匹配 /new, /reset 等命令                                     │
│                                                                 │
│  4. 评估会话新鲜度                                               │
│     evaluateSessionFreshness({ updatedAt, now, policy })        │
│                                                                 │
│  5. 处理线程派生 (可选)                                          │
│     forkSessionFromParent()                                     │
│                                                                 │
│  6. 持久化会话文件                                               │
│     resolveAndPersistSessionFile()                              │
```

```
21 |                                                                    |
22 |  7. 归档旧会话 (重置时)                                            |
23 |     archiveSessionTranscripts()                                    |
24 |                                                                    |
25 |  8. 触发插件钩子                                                   |
26 |     hookRunner.runSessionStart() / runSessionEnd()                 |
27 └────────────────────────────────────────────────────────────────────┘
```

2. 重置触发器机制 (auto-reply/reply/session.ts:249-273)

```
1  // 默认重置触发器
2  const DEFAULT_RESET_TRIGGERS = ["/new", "/reset"];
3
4  // 匹配逻辑
5  for (const trigger of resetTriggers) {
6    if (trimmedBodyLower === triggerLower) {
7      isNewSession = true;
8      bodyStripped = "";  // 清空消息体
9      resetTriggered = true;
10   }
11   // 支持带参数的重置: "/new 你好"
12   if (strippedForResetLower.startsWith(triggerPrefixLower)) {
13     isNewSession = true;
14     bodyStripped = strippedForReset.slice(trigger.length).trimStart();
15     resetTriggered = true;
16   }
17 }
```

3. 会话新鲜度策略 (config/sessions/reset.ts 相关)

```
1  type ResetPolicy = {
2    direct: number;   // DM 会话过期时间 (小时)
3    group: number;    // 群组会话过期时间
4    thread: number;   // 线程会话过期时间
5  };
6
7  // 默认策略
8  const DEFAULT_RESET_HOURS = { direct: 24, group: 4, thread: 24 };
9
10 // 评估函数
11 function evaluateSessionFreshness(params: {
12   updatedAt: number;
13   now: number;
14   policy: ResetPolicy;
```

```
}): { fresh: boolean; reason?: string };
```

4. 线程派生机制 (auto-reply/reply/session.ts:123-163)

```
// 当父会话 token 过多时跳过派生，避免上下文溢出
const DEFAULT_PARENT_FORK_MAX_TOKENS = 100_000;

function forkSessionFromParent(params: {
  parentEntry: SessionEntry;
  agentId: string;
  sessionsDir: string;
}): { sessionId: string; sessionFile: string } | null {
  // 1. 打开父会话管理器
  const manager = SessionManager.open(parentSessionFile);

  // 2. 创建分支会话
  const sessionFile = manager.createBranchedSession(leafId);

  // 3. 或创建新的派生会话文件
  const header = {
    type: "session",
    version: CURRENT_SESSION_VERSION,
    id: sessionId,
    parentSession: parentSessionFile,  // 链接到父会话
  };
}
```

3.11.5 会话存储机制

1. 存储结构(gateway/session-utils.ts:575-618)

```
~/.openclaw/
├── agents/
│   ├── main/sessions.json      # main agent 的会话存储
│   ├── ops/sessions.json       # ops agent 的会话存储
│   └── ...
└── sessions.json               # 全局存储（旧版兼容）
```

2. 多 Agent 存储合并(gateway/session-utils.ts:575-618)

```
function loadCombinedSessionStoreForGateway(cfg: OpenClawConfig): {
  storePath: string;
  store: Record<string, SessionEntry>;
```

```
} {
  // 模板路径: 支持每个 agent 独立存储
  if (isStorePathTemplate(storeConfig)) {
    for (const agentId of agentIds) {
      const storePath = resolveStorePath(storeConfig, { agentId });
      const store = loadSessionStore(storePath);
      // 合并到 combined，以 canonicalKey 为键
    }
  }
  // 单一文件: 所有 agent 共享存储
  else {
    // 直接加载，规范化键名
  }
}
```

3. 键名规范化与遗留键清理（gateway/session-utils.ts:237-260）

```
// 清理大小写不一致的遗留键
function pruneLegacyStoreKeys(params: {
  store: Record<string, unknown>;
  canonicalKey: string;
  candidates: Iterable<string>;
}) {
  // 1. 收集所有需要删除的键
  for (const candidate of candidates) {
    if (candidate !== canonicalKey) {
      keysToDelete.add(candidate);
    }
    // 2. 查找大小写变体
    for (const legacyKey of findStoreKeysIgnoreCase(store, candidate))
      if (legacyKey !== canonicalKey) {
        keysToDelete.add(legacyKey);
      }
    }
  }
  // 3. 批量删除
  for (const key of keysToDelete) {
    delete store[key];
  }
}
```

3.11.6 会话清理机制

Cron 会话清理器策略（cron/session-reaper.ts:57-156）

```
// 清理策略
const DEFAULT_RETENTION_MS = 24 * 3_600_000; // 24 小时
const MIN_SWEEP_INTERVAL_MS = 5 * 60_000;    // 最小清理间隔 5 分钟

async function sweepCronRunSessions(params: {
  cronConfig?: CronConfig;
  sessionStorePath: string;
}): Promise<ReaperResult> {
  // 1. 节流检查
  if (now - lastSweepAtMs < MIN_SWEEP_INTERVAL_MS) {
    return { swept: false, pruned: 0 };
  }

  // 2. 遍历存储，删除过期 cron 运行会话
  for (const key of Object.keys(store)) {
    if (isCronRunSessionKey(key) && entry.updatedAt < cutoff) {
      delete store[key];
      pruned++;
    }
  }

  // 3. 归档关联的对话文件
  archiveSessionTranscripts({ sessionId, reason: "deleted" });
}
```

3.11.7 会话投递路由

1. 投递信息持久化（channels/session.ts:21-58）

```
async function recordInboundSession(params: {
  storePath: string;
  sessionKey: string;
  ctx: MsgContext;
  updateLastRoute?: InboundLastRouteUpdate;
}): Promise<void> {
  // 1. 记录入站消息元数据
  await recordSessionMetaFromInbound({
    storePath,
    sessionKey: canonicalSessionKey,
    ctx,
    groupResolution,
  });

  // 2. 更新投递路由
```

```
  await updateLastRoute({
    storePath,
    sessionKey: targetSessionKey,
    deliveryContext: {
      channel: update.channel,
      to: update.to,
      accountId: update.accountId,
      threadId: update.threadId,
    },
  });
}
```

2. 投递路由解析（auto-reply/reply/session.ts:60-87）

```
function resolveLastChannelRaw(params: {
  originatingChannelRaw?: string;
  persistedLastChannel?: string;
  sessionKey?: string;
}): string | undefined {
  // 内部 webchat/系统消息不应覆盖已知的外部投递路由
  if (originatingChannel === INTERNAL_MESSAGE_CHANNEL) {
    // 优先使用已持久化的外部通道
    if (persistedChannel && isDeliverableMessageChannel(persistedChanne
      return persistedChannel;
    }
    // 回退到 sessionKey 中编码的通道提示
    if (sessionKeyChannelHint && isDeliverableMessageChannel(sessionKey
      return sessionKeyChannelHint;
    }
  }
}
```

3.11.8 关键设计决策

1. 大小写不敏感的键匹配

- 所有 sessionKey 在存储和查询时统一转为小写
- 遗留的大小写变体通过后台清理机制移除

2. 跳过缓存的会话加载

- loadSessionStore(storePath, { skipCache: true })

- 避免 Windows mtime 粒度问题导致的数据不一致

3. 线程会话的父会话派生

- 父会话 token 超过阈值时跳过派生，避免上下文溢出
- 派生会话记录 parentSession 链接，支持追溯

4. 重置时的行为继承

- /new 或 /reset 后保留用户设置的 thinkingLevel、verboseLevel 等
- 清空 token 统计和压缩计数，确保干净的会话状态

5. 会话归档机制

- 重置或删除会话时，对话文件移动到 .archived/ 目录
- 避免对话历史无限累积占用磁盘空间

3.12 自进化机制

OpenClaw 通过以下几个核心机制实现自我更新与进化：

3.12.1 可修改的核心文件

工作区包含可编辑的文件，agent 可在对话中修改：

- AGENTS.md - 操作指南和行为规则
- SOUL.md - 人设、语气和边界
- MEMORY.md - 长期记忆 (经验教训、重要决定)
- memory/YYYY-MM-DD.md - 短期日常记忆

3.12.2 动态系统提示

每次运行时动态构建系统提示，包括：

1. 注入的工作区文件内容
2. 可用的工具列表
3. Skills 列表 (可扩展)

4. 运行时环境信息

当文件被修改后，下次会话立即反映变化。

3.12.3 Skills 扩展系统

- 通过 ClawHub 发现和安装新 skills
- Skills 可以教 agent 使用新工具
- 支持 workspace、managed、bundled 三种位置
- 按需加载 SKILL.md 指令

3.12.4 自我修改能力

Agent 内置 read/write/edit 工具，可以：

1. 更新自己的行为 - 修改 AGENTS.md 添加新规则
2. 学习经验 - 将教训写入 MEMORY.md
3. 扩展能力 - 安装新的 skills
4. 调整人设 - 更新 SOUL.md 改变语气和边界

3.12.5 自我更新指令

系统提示包含 OpenClaw Self-Update 部分，agent 可以运行：

- config.apply - 应用配置变更
- update.run - 执行更新流程

3.12.6 进化循环

```
用户指令/反馈 → Agent 修改文件 → 下次会话加载新内容 → 行为改变 → 持续迭代
```

这种设计让 agent 能够像人类一样"学习"和"成长"，通过文件系统持久化进化成果。

3.13 工作区与 Agent 路由

3.13.1 工作区

工作区是代理的文件系统级隔离环境，包含代理的引导文件、记忆、身份和工具配置。

核心特性

1. 目录结构

- 默认路径：~/.openclaw/workspace（可通过 OPENCLAW_PROFILE 环境变量切换到 workspace-{profile}）
- 每个工作区包含以下引导文件：
- AGENTS.md - 代理行为指南
- SOUL.md - 代理核心人格
- TOOLS.md - 工具定义
- IDENTITY.md - 身份配置
- USER.md - 用户偏好
- HEARTBEAT.md - 心跳任务
- BOOTSTRAP.md - 初始引导
- MEMORY.md / memory.md - 持久记忆

2. 安全边界

- 所有文件读取通过 openBoundaryFile 进行边界检查
- 防止路径遍历攻击
- 文件大小限制：2MB
- 缓存机制：基于 inode/dev/size/mtime 的身份验证

3. 初始化流程

- 自动创建目录结构
- Git 仓库初始化

- 引导文件模板加载
- 入职状态跟踪（`.openclaw/workspace-state.json`）

4. 子代理过滤

- 子代理和定时任务会过滤掉非核心引导文件
- 只保留：AGENTS.md, TOOLS.md, SOUL.md, IDENTITY.md, USER.md

3.13.2 多代理路由策略

路由系统决定**哪个代理处理哪个消息**，是 OpenClaw 多代理架构的核心。

路由层次结构（从高到低优先级）

```
const tiers = [
  { matchedBy: "binding.peer", ... },           // 1. 精确对等匹配
  { matchedBy: "binding.peer.parent", ... },    // 2. 父对等匹配（线程）
  { matchedBy: "binding.guild+roles", ... },    // 3. 公会+角色匹配
  { matchedBy: "binding.guild", ... },          // 4. 公会匹配
  { matchedBy: "binding.team", ... },           // 5. 团队匹配
  { matchedBy: "binding.account", ... },        // 6. 账户匹配
  { matchedBy: "binding.channel", ... },        // 7. 频道匹配
  { matchedBy: "default", ... }                 // 8. 默认代理
];
```

路由匹配规则详解

1. 对等绑定（binding.peer）：

最高优先级

- 匹配特定聊天对象（用户/频道/群组）
- 示例：Discord 频道 c1 绑定到代理 chan

2. 父对等绑定（binding.peer.parent）：

用于线程继承

- 当消息来自线程但线程本身无绑定时，检查父频道绑定
- 确保 Discord 论坛主题能继承父频道路由

3. 公会 + 角色绑定（binding.guild+roles）：

- Discord 特有
- 匹配特定公会中的特定角色成员
- 需要同时满足 `guildId` 和 `roles` 约束

4. 公会绑定（binding.guild）：

- Discord 公会级匹配（无角色要求）
- 示例：公会 `g1` 的所有消息路由到代理 `guild`

5. 团队绑定（binding.team）：

- Slack 特有
- 匹配 Slack 工作区（团队）

6. 账户绑定（binding.account）：

- 匹配特定账户（非通配符 *）
- 示例：WhatsApp Business 账户 biz 绑定到代理 b

7. 频道绑定（binding.channel）：

- 跨账户通配符匹配（accountId: "*"）
- 匹配所有使用该频道的账户

8. 默认路由：

- 无任何绑定匹配时使用
- 路由到配置中的 `defaultAgentId`（默认为 `main`）

绑定配置示例

```
const bindings = [
  // 精确对等绑定
  {
    agentId: "support",
```

```
    match: {
      channel: "whatsapp",
      accountId: "biz",
      peer: { kind: "direct", id: "+1234567890" }
    }
  },
  // 公会绑定
  {
    agentId: "community",
    match: {
      channel: "discord",
      guildId: "123456789"
    }
  },
  // 角色绑定
  {
    agentId: "admin",
    match: {
      channel: "discord",
      guildId: "123456789",
      roles: ["admin", "moderator"]
    }
  }
];
```

3.13.3 会话键策略

会话键决定对话历史如何隔离和持久化。

DM 作用域类型

```
type DmScope =
  | "main"                    // 所有 DM 共享一个会话
  | "per-peer"                // 每个对等有独立会话
  | "per-channel-peer"        // 每个频道+对等组合独立
  | "per-account-channel-peer"; // 每个账户+频道+对等组合独立
```

会话键格式

```
agent:{agentId}:{mainKey}                              // 主会话
agent:{agentId}:direct:{peerId}                        // per-peer DM
agent:{agentId}:{channel}:direct:{peerId}              // per-channel-peer
agent:{agentId}:{channel}:{accountId}:direct:{peerId}  // per-account-ch
agent:{agentId}:{channel}:{peerKind}:{peerId}          // 群组/频道
```

```
agent:{agentId}:{channel}:{peerKind}:{peerId}:thread:{threadId}  // 线程
```

身份链接

- 允许跨平台合并会话。
- 示例：Telegram 用户 111111111 和 Discord 用户 222222222222222222 共享身份 alice。

```
identityLinks: {
  alice: ["telegram:111111111", "discord:2222222222222222"]
}
```

3.13.4 性能优化

1. 绑定缓存

- evaluatedBindingsCacheByCfg - WeakMap 缓存
- 按 channel + accountId 双键索引
- 最大 2000 个缓存条目

2. 文件缓存

- 基于 inode + dev + size + mtime 的缓存键
- 避免重复读取相同文件

3. 模板缓存

- 工作区模板预加载
- Promise 缓存防止重复编译

3.13.5 实际应用场景

场景 1：多租户 SaaS

```
binding 1: WhatsApp Business "company-a" -> agent "support-a"
binding 2: WhatsApp Business "company-b" -> agent "support-b"
```

场景 2：Discord 社区管理

```
binding 1: Guild "123" + Role "admin" -> agent "admin-bot"
binding 2: Guild "123" (no role) -> agent "community-bot"
binding 3: Channel "welcome" -> agent "greeter"
```

场景 3：Slack 企业部署

```
binding 1: Team "T-sales" -> agent "sales-assistant"
binding 2: Team "T-engineering" -> agent "tech-support"
```

场景 4：线程隔离

```
Parent Channel "general" -> agent "general-bot"
Thread in "general" inherits parent binding
```

3.13.6 关键设计决策

1. 确定性路由：相同输入总是产生相同输出，无随机性。

2. 显式优先级：层级清晰，避免隐式行为。

3. 安全边界：所有路径操作都经过验证和边界检查。

4. 性能优先：多级缓存，最小化 I/O。

5. 可扩展性：支持插件扩展路由逻辑。

这套系统实现了灵活的多代理路由，同时保证了安全性、性能和可维护性。

3.14 Nodes

Nodes 是 OpenClaw 的分布式设备/客户端管理架构，让 Gateway 可以远程控制和协调多个设备上的操作。

3.14.1 核心概念

Node = 一个可执行命令的远程客户端，例如：

- iOS/Android 手机
- macOS/Linux/Windows 主机

- 任何运行 openclaw node 的设备

3.14.2 核心数据结构

类型	位置	用途
NodeListNode	shared/node-list-types.ts	节点列表展示（含 caps/commands/permissions）
NodeSession	gateway/node-registry.ts	运行时 WebSocket 会话
NodePairingPairedNode	infra/node-pairing.ts	持久化配对记录（含认证 token）

3.14.3 配对流程

```
Node                                    Gateway
  |-- node.pair.request -------------->|  (发起请求)
  |                                     |  (加入 pending 列表)
  |                                     |
  |<-------- 广播 node.pair.requested --|  (通知管理员)
  |                                     |
  |          [管理员审批]               |
  |                                     |
  |<------- node.pair.approve ---------|  (生成 token)
  |                                     |
  |-- node.pair.verify --------------->|  (验证 token)
  |                                     |
  |<----------- 连接建立 --------------|  (认证成功)
```

3.14.4 Node Host 架构

src/node-host/ 中的代码运行在远程设备上：

```
┌─────────────────────────────────────────────────┐
│                  Node Host                      │
├─────────────────────────────────────────────────┤
│  runner.ts                                      │
│    - 启动 WebSocket 客户端                      │
│    - 连接 Gateway                               │
```

```
|    - 声明 caps/commands                          |
├──────────────────────────────────────────────────┤
|  invoke.ts                                       |
|    - 处理 system.run/which/browser.proxy         |
|    - 执行本地命令                                 |
|    - 管理执行审批                                 |
├──────────────────────────────────────────────────┤
|  config.ts                                       |
|    - 存储 nodeId/token/gateway 配置               |
|    - ~/.openclaw/node.json                       |
└──────────────────────────────────────────────────┘
```

3.14.5 命令系统

命令分类（gateway/node-command-policy.ts:56）

安全策略

1. 命令必须在 allowlist 中

2. 必须在节点声明的 commands 列表中

3. 敏感命令（camera.snap, screen.record, sms.send）需额外配置

3.14.6 唤醒机制

当 Node 离线时自动唤醒 (gateway/server-methods/nodes.ts:89):

```
1. maybeWakeNodeWithApns()
   ├── 检查 APNS 注册
   ├── 发送后台唤醒通知
   └── 等待重连 (3s)

2. 如果仍离线，重试一次
```

```
    └── 发送后台唤醒 + 等待 (12s)

3. 如果仍离线
    └── maybeSendNodeWakeNudge()
        └── 发送可见提醒："OpenClaw needs a quick reopen"
```

3.14.7 事件处理

Node 可向 Gateway 发送事件 (gateway/server-node-events.ts:247):

3.14.8 安全机制

1. 配对认证：token + 节点 ID 双重验证

2. 命令白名单：按平台限制可用命令

3. Exec 审批：敏感命令需用户确认

4. 环境隔离：sanitizeHostExecEnv 阻止 PATH 污染

5. 输出截断：200KB 上限防止内存溢出

3.14.9 典型调用流程

```
客户端 → Gateway.node.invoke({nodeId, command, params})
          │
          ├── 检查节点是否连接
          │     └── 未连接 → APNS 唤醒
          │
          ├── 命令策略检查
          │     └── node-command-policy.ts
          │
          ├── NodeRegistry.invoke()
          │     └── WebSocket 发送 node.invoke.request
```

```
            │
            └── Node Host 处理
                  ├── invoke.ts 路由到具体 handler
                  ├── 执行命令 (spawn + 安全检查)
                  └── 返回 node.invoke.result
```

3.15 安全策略

3.15.1 核心信任模型

核心原则：OpenClaw 采用"个人助手"模型，不是多租户共享总线。

```
单用户信任模型：
┌──────────────────────────────────────┐
│  一个可信操作者                       │
│  ├── 一个 Gateway                    │
│  │    └── 多个 Agent                 │
│  └── 多个会话（sessionKey 是路由控制，不是授权边界）
└──────────────────────────────────────┘
```

关键点：

- 通过 Gateway 认证的调用者 = 可信操作者
- sessionKey/session ID 只是路由控制，不是用户授权边界
- 同一 Gateway 上的操作者可以互相看到数据 → 预期行为
- 多用户场景：每个信任边界使用独立的 OS 用户/主机/Gateway

3.15.2 插件信任边界

插件以进程内方式加载，与 Gateway 同等权限：

```
插件 = 可信代码
├── 可以读取环境变量/文件
├── 可以执行主机命令
└── 与 Gateway 进程权限相同
```

安全报告必须证明边界绕过（如未认证加载、策略绕过），而不是"恶意插件执行了特权操作"。

3.15.3 执行沙箱默认行为

```
agents.defaults.sandbox.mode: off   # 默认关闭沙箱
tools.exec.host: sandbox            # 路由偏好，但沙箱未激活时在主机执行
```

含义：默认情况下，命令执行在主机上进行，因为操作者已受信任。

3.15.4 范围外事项（常见误报）

类型	说明
Prompt 注入	除非绕过策略/认证/沙箱边界
操作者预期的本地功能	如 TUI 本地 ! shell
已授权用户触发的本地操作	如 allowlisted 发送者运行 /export-session
恶意插件行为	安装/启用插件 = 授予信任
多租户隔离期望	同一 Gateway 不提供用户间隔离
ReDoS/DoS 需要可信配置输入	如自定义正则表达式
公网暴露	不建议但不是漏洞
暴露第三方凭证	除非能访问 OpenClaw 基础设施

3.15.5 部署假设

```
信任边界：
├── 主机 OS/管理员边界 = 信任边界
├── 能修改 ~/.openclaw 的人 = 可信操作者
└── 推荐模式：一用户一主机/VPS一Gateway
```

3.15.6 Web 接口安全

```
# 推荐：仅绑定回环地址
gateway.bind="loopback"  # 默认值
openclaw gateway run --bind loopback

```

```
# 远程访问方案：
# 1. SSH 隧道
# 2. Tailscale serve/funnel
# 3. 不要直接暴露到公网
```

3.15.7 运行时要求

- **Node.js ≥ 22.12.0**（包含安全补丁）
- Docker：非 root 用户、只读文件系统、最小权限

3.15.8 工具文件系统加固

```
# 推荐
tools.exec.applyPatch.workspaceOnly: true  # 限制写入到工作区

# 可选
tools.fs.workspaceOnly: true  # 限制所有文件操作到工作区
```

总结：OpenClaw 的安全模型基于"可信操作者"假设，核心是主机信任 + Gateway 认证，不提供多租户隔离。安全报告需要证明边界绕过，而非展示已授权/可信场景下的行为。

3.16 配置管理

3.16.1 架构概览

配置管理采用分层架构，核心模块位于 src/config/:

```
src/config/
├── io.ts              # 配置I/O核心引擎
├── paths.ts           # 路径解析策略
├── validation.ts      # 多层验证框架
├── zod-schema.ts      # Zod模式定义
├── defaults.ts        # 运行时默认值应用
├── includes.ts        # $include模块化支持
├── env-substitution.ts # 环境变量替换
└── types.ts           # TypeScript类型聚合
```

3.16.2 核心设计原则

配置文件格式与位置

JSON5 格式，支持注释和尾随逗号：

```
// 支持环境变量
{
  "models": {
    "providers": {
      "anthropic": {
        "apiKey": "${ANTHROPIC_API_KEY}"  // 运行时替换
      }
    }
  }
}
```

路径解析优先级（paths.ts:130-183）：

1. OPENCLAW_CONFIG_PATH 环境变量

2. OPENCLAW_STATE_DIR/openclaw.json

3. ~/.openclaw/openclaw.json (新规范)

4. ~/.clawdbot/clawdbot.json (历史兼容)

配置生命周期

```
加载 → 解析 → 合并 → 验证 → 应用默认值 → 缓存
```

关键实现（io.ts:682-818）：

```
function loadConfig(): OpenClawConfig {
  // 1. 读取原始JSON5文件
  const raw = deps.fs.readFileSync(configPath, "utf-8");
  const parsed = deps.json5.parse(raw);

  // 2. 解析$include指令
  const resolved = resolveConfigIncludesForRead(parsed, configPath, dep

  // 3. 环境变量替换
  const { resolvedConfigRaw } = resolveConfigForRead(resolved, deps.env

  // 4. 验证与默认值
  const validated = validateConfigObjectWithPlugins(resolvedConfigRaw);
  const cfg = applyModelDefaults(applyAgentDefaults(validated.config));

  // 5. 路径规范化
```

```
  normalizeConfigPaths(cfg);

  return cfg;
}
```

3.16.3 模块化配置 ($include)

设计目标

支持大型部署的配置拆分，实现关注点分离。

```
// openclaw.json
{
  "$include": [
    "./models/anthropic.json5",
    "./channels/telegram.json5"
  ],
  "gateway": {
    "mode": "remote"
  }
}
```

安全实现

路径遍历防护（includes.ts:198-222）：

```
// 拒绝访问配置目录外的文件
if (!isPathInside(this.rootDir, normalized)) {
  thrownew ConfigIncludeError(
    `Include path escapes config directory: ${includePath}`,
    includePath
  );
}

// 解析符号链接并重新验证
const real = fs.realpathSync(normalized);
if (!isPathInside(this.rootRealDir, real)) {
  thrownew ConfigIncludeError(
    `Include path resolves outside config directory (symlink)`,
    includePath
  );
}
```

深度限制：最大嵌套层级 10 层，防止循环引用

合并策略

深度合并算法（includes.ts:70-85）：

```
export function deepMerge(target: unknown, source: unknown): unknown {
  if (Array.isArray(target) && Array.isArray(source)) {
    return [...target, ...source]; // 数组连接
  }
  if (isPlainObject(target) && isPlainObject(source)) {
    const result = { ...target };
    for (const key of Object.keys(source)) {
      result[key] = key in result
        ? deepMerge(result[key], source[key]) // 递归合并
        : source[key];
    }
    return result;
  }
  return source; // 原始类型：后值覆盖前值
}
```

3.16.4 验证框架

多层验证架构

```
// 验证流水线
validateConfigObjectWithPlugins(raw)
  → validateConfigObjectRaw(raw)        // 1. 基础Zod验证
  → findLegacyConfigIssues(raw)         // 2. 历史配置检查
  → findDuplicateAgentDirs(config)      // 3. Agent目录冲突检测
  → validateIdentityAvatar(config)      // 4. Avatar路径安全验证
  → validatePluginSchemas(config)       // 5. 插件配置验证
```

Zod Schema 设计

严格模式 + 细粒度校验 (zod-schema.ts):

```
exportconst OpenClawSchema = z.object({
  // 元数据自动时间戳处理
  meta: z.object({
    lastTouchedAt: z.union([
      z.string(),
```

```
      z.number().transform((n, ctx) => {
        const d = new Date(n);
        if (Number.isNaN(d.getTime())) {
          ctx.addIssue({ code: z.ZodIssueCode.custom, message: "Invalid
          return z.NEVER;
        }
        return d.toISOString();
      })
    ]).optional()
  }).strict().optional(),

  // Gateway配置
  gateway: z.object({
    mode: z.union([z.literal("local"), z.literal("remote")]).optional()
    auth: z.object({
      mode: z.union([
        z.literal("none"),
        z.literal("token"),
        z.literal("password"),
        z.literal("trusted-proxy")
      ]).optional(),
      token: z.string().optional().register(sensitive) // 敏感值标记
    }).strict().optional()
  }).strict().optional(),

  // 跨字段验证
}).superRefine((cfg, ctx) => {
  // 验证broadcast中的agentId是否存在于agents.list
  const agentIds = new Set(cfg.agents?.list?.map(a => a.id) ?? []);
  for (const [peerId, ids] of Object.entries(cfg.broadcast ?? {})) {
    for (const agentId of ids) {
      if (!agentIds.has(agentId)) {
        ctx.addIssue({
          code: z.ZodIssueCode.custom,
          path: ["broadcast", peerId],
          message: `Unknown agent id "${agentId}"`
        });
      }
    }
  }
});
```

插件配置验证

动态Schema加载 (validation.ts:418-438):

```
for (const record of registry.plugins) {
  const pluginId = record.id;
  const entry = normalizedPlugins.entries[pluginId];

  if (record.configSchema) {
    const res = validateJsonSchemaValue({
      schema: record.configSchema,
      cacheKey: record.schemaCacheKey ?? pluginId,
      value: entry?.config ?? {}
    });

    if (!res.ok) {
      issues.push({
        path: `plugins.entries.${pluginId}.config`,
        message: `invalid config: ${res.errors.join(", ")}`
      });
    }
  }
}
```

3.16.5 环境变量处理

双阶段替换

读取时替换（io.ts:652-666）：

```
function resolveConfigForRead(resolvedIncludes, env){
  // 1. 应用config.env到process.env
  applyConfigEnvVars(resolvedIncludes, env);

  // 2. 替换${VAR}引用
  return {
    resolvedConfigRaw: resolveConfigEnvVars(resolvedIncludes, env),
    envSnapshotForRestore: { ...env } // 保存快照用于回写
  };
}
```

写入时恢复：

保持配置文件中的环境变量引用 (io.ts:1082-1109):

```
// 写入前恢复原始的${VAR}引用
cfgToWrite = restoreEnvVarRefs(
```

```
    cfgToWrite,
    parsedRes.parsed,      // 原始文件内容
    envForRestore          // 读取时的环境变量快照
);
```

3.16.6 运行时默认值

设计原则

不持久化默认值 - 只在加载时应用，写入时剥离。

```
// 加载时应用默认值
const cfg = applyModelDefaults(
  applyAgentDefaults(
    applySessionDefaults(validated.config)
  )
);

// 写入时不包含运行时默认值
const stampedOutputConfig = stampConfigVersion(outputConfig);
// 不调用 applyModelDefaults
```

关键默认值

Model 默认值（defaults.ts:213-347）：

- contextWindow: 200K tokens
- maxTokens: min(8192, contextWindow)
- cost: {input: 0, output: 0, cacheRead: 0, cacheWrite: 0}
- api: "anthropic-messages" (Anthropic provider)

Agent 默认值（defaults.ts:349-388）：

- maxConcurrent: 10
- subagents.maxConcurrent: 5

Session 默认值（defaults.ts:144-168）：

- mainKey: "main" (忽略用户配置)

- maintenance.mode: "warn"
- maintenance.pruneAfter: "30d"

3.16.7 缓存与性能优化

配置缓存

短期缓存策略（io.ts:1292-1374）：

```
const DEFAULT_CONFIG_CACHE_MS = 200; // 200ms TTL

export function loadConfig(): OpenClawConfig {
  // 检查运行时快照
  if (runtimeConfigSnapshot) {
    return runtimeConfigSnapshot;
  }

  // 检查缓存
  const cached = configCache;
  if (cached && cached.configPath === configPath && cached.expiresAt >
    return cached.config;
  }

  // 加载并更新缓存
  const config = io.loadConfig();
  configCache = {
    configPath,
    expiresAt: now + cacheMs,
    config
  };

  return config;
}
```

禁用缓存: OPENCLAW_DISABLE_CONFIG_CACHE=1

写入审计

安全审计日志 (io.ts:1178-1228):

```
const auditRecord = {
  ts: new Date().toISOString(),
```

```
  event: "config.write",
  result: "rename" | "copy-fallback" | "failed",
  configPath,
  previousHash,
  nextHash,
  previousBytes,
  nextBytes,
  changedPathCount,
  suspicious: [
    "size-drop",           // 大小骤降50%+
    "missing-meta-before-write",
    "gateway-mode-removed"
  ],
  pid, ppid, cwd, argv
};

await appendConfigWriteAuditRecord(deps, auditRecord);
```

3.16.8 错误处理与诊断

友好错误消息

DM策略错误提示 (io.ts:148-167):

```
function formatConfigValidationFailure(pathLabel, issueMessage){
  const match = issueMessage.match(OPEN_DM_POLICY_ALLOW_FROM_RE);
  if (!match) return `Config validation failed: ${pathLabel}: ${issueMe

  return [
    `Configuration mismatch: ${policyPath} is "open", but ${allowPath}
    "",
    "Fix with:",
    `  openclaw config set ${allowPath} '["*"]'`,
    "",
    "Or switch policy:",
    `  openclaw config set ${policyPath} "pairing"`
  ].join("\n");
}
```

历史配置检测

迁移警告 (legacy.ts):

```
export function findLegacyConfigIssues(raw: unknown): LegacyConfigIssue
  const issues: LegacyConfigIssue[] = [];

  // 检测已移除的配置项
  if (raw.routing?.allowFrom) {
    issues.push({
      path: "routing.allowFrom",
      message: "routing.allowFrom is removed. Use routing.channels.*.dm
    });
  }

  // 检测废弃的命名
  if (raw.gateway?.token) {
    issues.push({
      path: "gateway.token",
      message: 'Use "gateway.auth.token" instead.'
    });
  }

  return issues;
}
```

3.16.9 安全特性

原型污染防护

Blocked Keys（prototype-keys.ts）：

```
exportconst BLOCKED_OBJECT_KEYS = new Set([
  "__proto__",
  "constructor",
  "prototype"
]);

export function isBlockedObjectKey(key: string): boolean {
  return BLOCKED_OBJECT_KEYS.has(key);
}
```

敏感值处理

Zod 敏感值标记（zod-schema.sensitive.ts）：

```
exportconst sensitive = Symbol("sensitive");

export function redactSensitiveValues(config: unknown): unknown {
  // 递归遍历配置对象
  // 替换标记为sensitive的值为"[REDACTED]"
}
```

文件权限

安全默认值（io.ts:1236-1240）：

```
// 目录权限: 0o700 (仅所有者可访问)
await fs.promises.mkdir(dir, { recursive: true, mode: 0o700 });

// 文件权限: 0o600 (仅所有者可读写)
await fs.promises.writeFile(tmp, json, { encoding: "utf-8", mode: 0o600
```

3.16.10 扩展机制

插件配置 Schema

动态加载（plugins/config-schema.ts）

```
export interface PluginConfigSchema {
  id: string;
  kind: "channel" | "memory" | "tool" | "skill";
  configSchema?: JSONSchema;
  channels?: string[];
}
```

运行时覆盖

测试与诊断场景（runtime-overrides.ts）：

```
export function applyConfigOverrides(config: OpenClawConfig): OpenClawCo
  // 允许通过环境变量覆盖特定配置项
  // 例如: OPENCLAW_OVERRIDE_gateway_mode=local
  return config;
}
```

3.16.11 关键文件索引

文件	核心职责
io.ts:682-818	配置加载主流程
includes.ts:91-279	$include 解析器实现
validation.ts:173-453	多层验证框架
zod-schema.ts:131-813	完整 Schema 定义
defaults.ts:129-532	默认值应用逻辑
paths.ts:115-183	路径解析策略

四、总结展望

2025 年个人效率工具在编程、办公领域得到了极大的发展与提升，2026 年作为企业智能元年，可以预见会出来更多的企业级智能体，这些智能体将朝着更加分布式、安全可控且深度集成业务流程的方向演进。

4.1 架构方向演进

1. 控制平面与执行节点的解耦 (Decoupling Control & Execution)

企业架构将采用类似的“中枢管理 + 边缘执行”模式。总部部署中央控制平面负责权限审核、配置同步和任务路由，而执行节点则部署在员工终端、内部服务器或特定硬件设备上，确保数据在企业内网中闭环处理。

2. 多智能体协作网络 (Multi-Agent Networking)

企业内部将不再是单一的“万能助手”，而是由多个专业化智能体（如 HR 助手、财务助手、IT 运维助手）构成的网络。它们通过标准化的 RPC 协议或内部会话协议进行信息共享和任务接力，形成复杂的自动化流水线。

3. 软硬件深层权限管理 (Hardware & Permission Integration)

企业级 Agent 将深入集成到办公硬件（如会议室系统、扫码枪、PDA）。架构设计上将包含更精细的 TCC（透明度、同意和控制）策略，确保 Agent 在执行敏感操作（如调用公司摄像头或执行系统脚本）时受到严格的合规性审计。

4.2 应用方向演进

1. 全渠道业务渗透 (Omnichannel Business Integration)

企业 Agent 将无缝嵌入到所有内部协作工具中，打破“信息孤岛”。Agent 不仅能回答问题，还能直接在通讯软件中通过 Discord/Slack 动作或自动触发工作流来处理审批、请假或订单更新，实现“对话即办公”。

2. 动态可视化协同工作区 (Live Visual Collaboration)

在复杂的企业决策场景（如供应链管理或数据分析）中，Agent 将利用类似 Canvas 的技术提供动态可视化面板。团队成员可以与 Agent 在同一画布上进行交互式修改，Agent 实时生成数据图表或拓扑图，大幅提升决策效率。

3. 基于沙箱的安全生产环境 (Sandboxed Production Environment)

企业将普遍采用“隔离运行环境”。当 Agent 处理外部客户询价或不受信任的输入时，系统会自动将其放入临时的、受限的容器中执行，防止恶意脚本通过 Agent 渗透进企业核心数据库或内部网络。

4. 企业级“技能中心” (Enterprise Skill Hubs)

企业将建立私有的技能注册表。员工可以为自己的 Agent 订购经 IT 部门认证的“技能包”（如 ERP 插件、专业财务计算器等）。Agent 会根据当前任务上下文，自动从企业私有云拉取并挂载这些技能，实现功能的动态扩展。

未来的企业级智能体将借鉴 OpenClaw 的 Local-first（数据私有）和 Gateway/Node（控制与执行分离）理念，在保证数据所有权（Own your data）的前提下，通过多智能体协作和深度的软硬件集成，成为企业数字化转型的核心基础设施。

附录

- https://derisk.alipay.com/
- https://notebooklm.google.com/
- https://openclaw.ai/
- https://github.com/openclaw/openclaw
- https://github.com/anomalyco/opencode
- https://github.com/derisk-ai/OpenDerisk