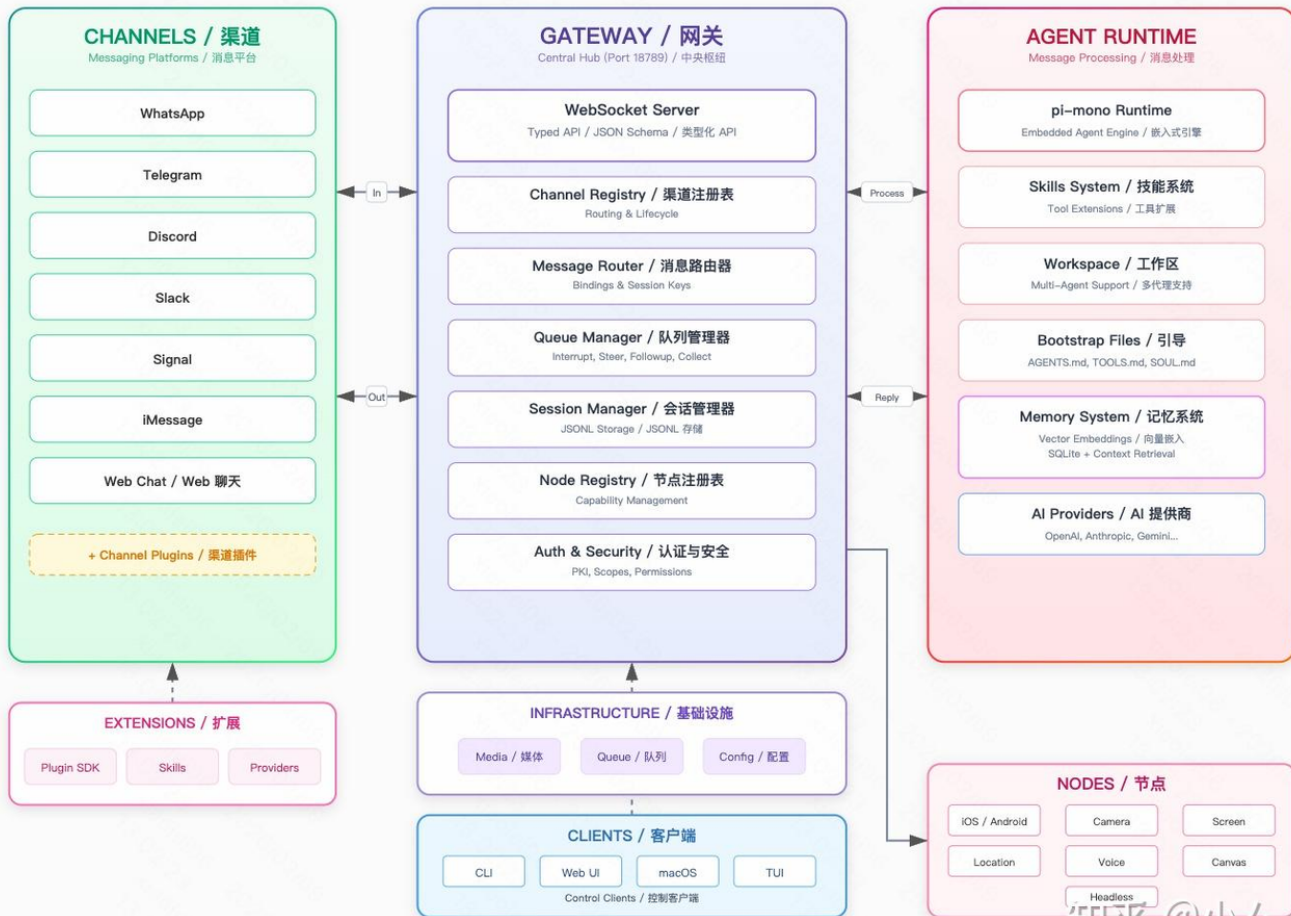


openclaw架构分享

OpenClaw 是一个通过Gateway，连接即时通讯平台（Channel, 如 Telegram、[Discord](#)、Slack 等）与本地AI Agent，能24×7 运行的个人助手。它不仅仅是一个简单的消息转发器，而是一个具备完整**会话管理**、**并发控制**、**记忆检索**以及**丰富工具支持**的复杂 Agent 运行时环境。

OpenClaw Architecture / OpenClaw 架构

Multi-Platform AI Agent Gateway / 多平台 AI 代理网关

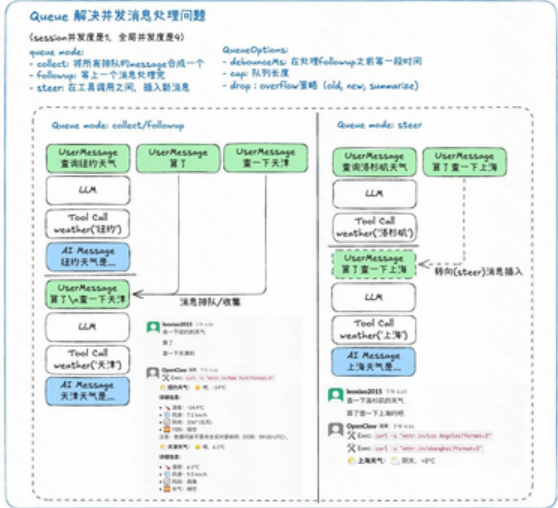
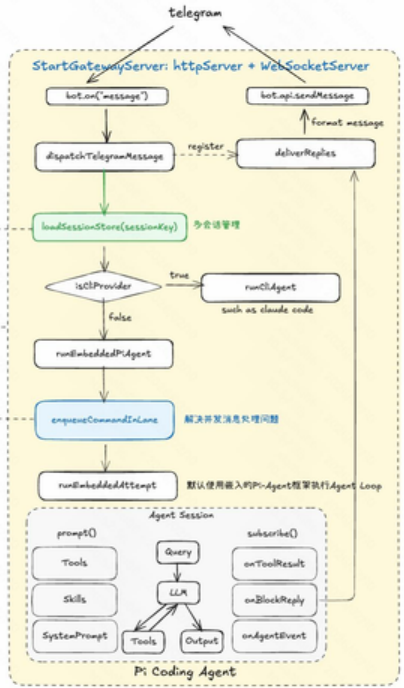
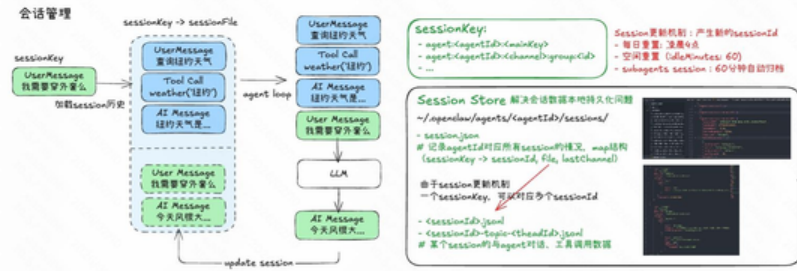


Legend / 图例: Channel Gateway Agent

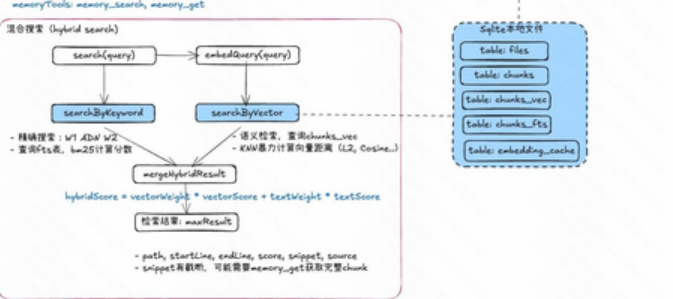
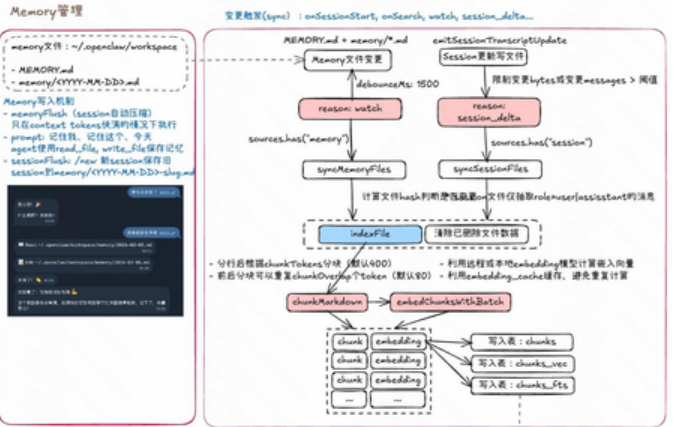
Message Flow: Channel → Gateway → Router → Agent → Response → Gateway → Channel

知乎 @小公

OpenClaw 核心运行流程

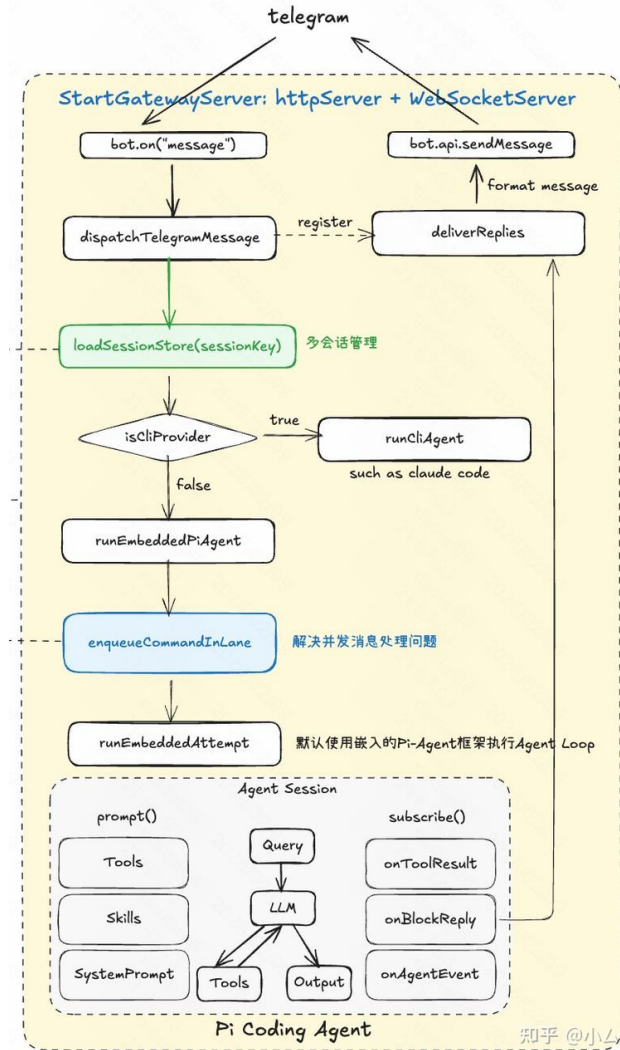


- ### Tools & Skills
- Tools**
- coding tools
 - exec
 - process
 - sandbox
 - apply_patch
 - web_search
 - web_fetch
 - browser
 - cron
 - openclaw tools: gateway, session, channel, tools, plugin, tools
 - ...
- Tools Policy 任务工具策略**
- profile
 - global
 - agent
 - subagent
 - sandbox
 - ...
- Skills: openclaw/skills**
- kind
 - gitlab
 - weather
 - coding-agent
 - reporter
 - ...



以telegram为例, 类似飞书和微信的channels

知乎 @小公



1.网关Gateway

startGatewayServer 持久运行的控制平面

心跳（维护连接）、Cron（定时清理）、Hook（事件扩展）、Session管理、认证、消息队列...

2.消息接入与分发

createTelegramMessageProcessor

其中核心分发函数 dispatchTelegramMessage 负责处理来自 Telegram 的消息
注册回复消息的回调函数 deliverReplies

3.Session Key机制

由于OpenClaw支持非常多的Channel账号、以及私聊、群组、Thread等各种会话形式。需要一种命名方式来唯一识别不同的会话。

4.Agent loop

OpenClaw支持调用已有的CLI Agent，比如Claude Code等；但默认情况下也嵌入了基于Pi-Agent框架 执行Agent运行时。

消息进入AgentSession后，通过一种ReAct范式执行Agent Loop（包括工具调用），通过注册reply回调（监听message_end事件）将AI回复的消息通过机器人发送给Telegram。

Gateway的重要机制-heartbeat

OpenClaw 的心跳机制（Heartbeat）是其最独特的功能之一，它将 Agent 从**被动响应式聊天机器人**转变为**主动自主监控器**。心跳本质上是一个定期执行的 Agent 轮次，在没有用户输入的情况下自动运行。

工作机制

调度器 (startHeartbeatRunner)

读取配置中的心跳间隔（默认 30m）

管理多个代理的心跳时间表

使用 setTimeout 定时触发

支持静默时段（23:00-08:00等）

事件类型

Cron 事件：预定提醒触发

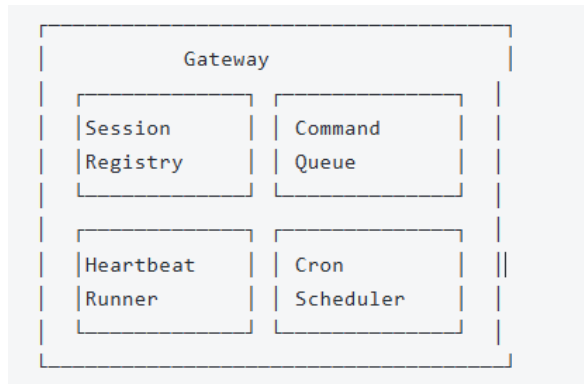
Exec 完成：异步命令完成通知

定期检查：纯粹的周期性心跳

单次运行 (runHeartbeatOnce)

// 简化流程

1. 检查是否在活动时段内
2. 读取 HEARTBEAT.md 文件内容
3. 检查待处理事件（cron/exec完成）
4. 如果无内容 → 回复 HEARTBEAT_OK
5. 如果有内容 → 生成用户提醒
6. 更新最后运行时间



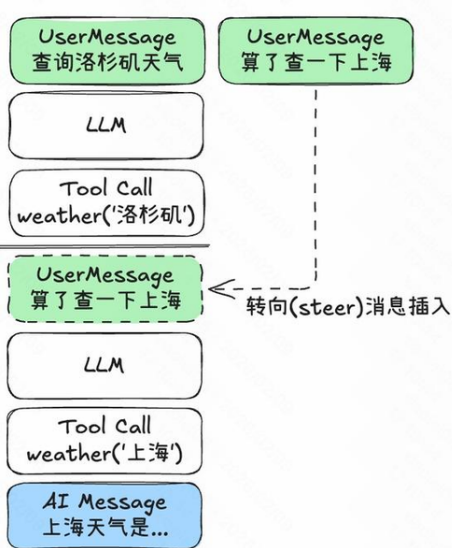
核心实现位于： /usr/lib/node_modules/openclaw/dist/health-BxAqqNt.js

队列与并发控制

Queue mode: collect/followup



Queue mode: steer



collect - 收集模式（默认）：将所有排队的消息合并成单个后续回复

steer - 转向模式：立即注入到当前agent回合中

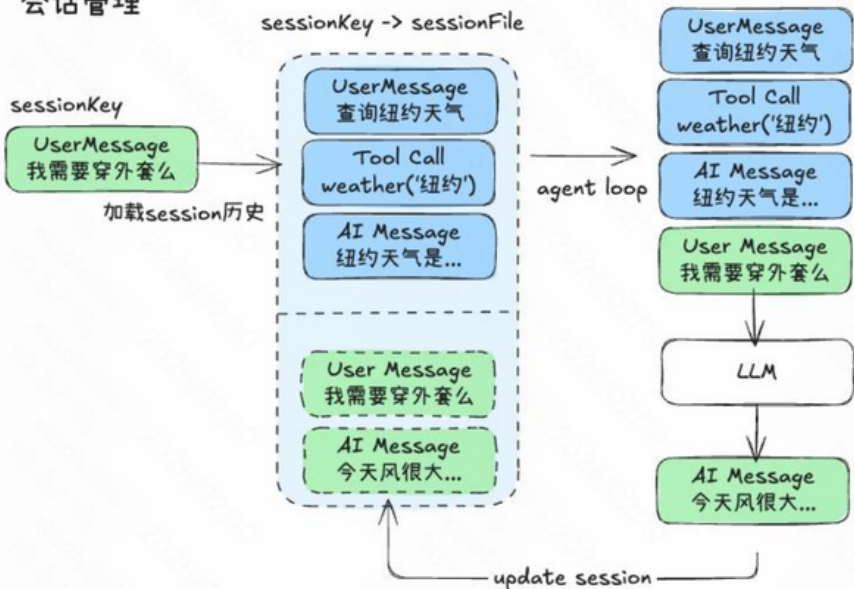
followup - 跟进模式：当前运行结束后，为下一个agent回合排队

steer-backlog - 转向+积压模式：现在转向当前回合，然后保留消息用于后续回合

OpenClaw 使用两层并发控制：

会话级别：同一会话内的消息串行处理，避免状态混乱
全局级别：默认并发度为 4，允许最多 4 个会话同时处理

会话管理



sessionKey:

- agent:<agentId>:<mainKey>
- agent:<agentId>:<channel>:group:<id>
- ...

- Session更新机制: 产生新的sessionId
- 每日重置: 凌晨4点
 - 空闲重置 (idleMinutes: 60)
 - subagents session: 60分钟自动归档

Session Store 解决会话数据本地持久化问题

- ~/openclaw/agents/<agentId>/sessions/
- session.json
- # 记录agentId对应所有session的情况, map结构 (sessionKey -> sessionId, file, lastChannel)

由于session更新机制
一个sessionKey, 可以对应多个sessionId

- <sessionId>.jsonl
- <sessionId>-topic-<theadId>.jsonl
- # 某个session的与agent对话、工具调用数据



```
sessions/
├─ sessions.json # 会话索引 (sessionKey -> 元数据映射)
├─ <sessionId>/ # 单个会话目录
│   ├─ meta.json # 会话元数据 (ID、渠道、令牌计数、状态等)
│   ├─ messages.jsonl # 消息历史 (JSON Lines 格式, 追加写入)
│   └─ context.json # 上下文快照 (压缩后的缓存, 用于快速加载)
```

私聊会话隔离策略

通过 dmScope配置, 控制私聊会话的隔离粒度:

- main (默认): 所有用户的私聊共享同一个主会话, 历史完全混合。
- per-peer: 按发送者隔离, 每个用户拥有独立的会话历史。
- per-channel-peer: 按渠道 + 发送者隔离, 同一用户在不同渠道 (如 Telegram 和 Slack) 拥有不同的会话。
- per-account-channel-peer: 按账户 + 渠道 + 发送者隔离, 用于多账户管理场景。

记忆系统

OpenClaw 拥有一个完善的记忆系统，通过对记忆相关的Markdown文件的实时索引和混合检索来实现长期记忆

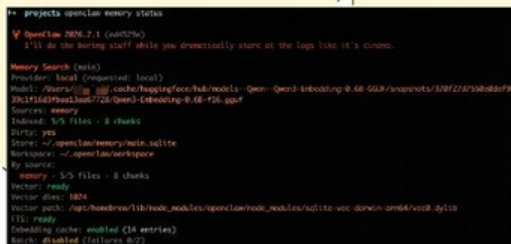
Memory

memory文件: ~/openclaw/workspace

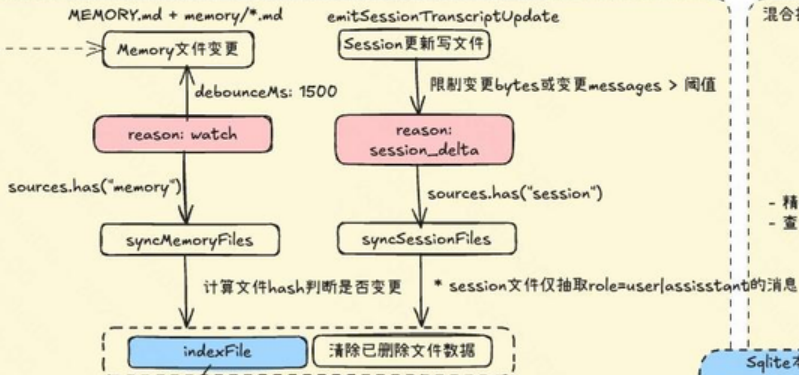
- MEMORY.md
- memory/<YYYY-MM-DD>.md

Memory写入机制

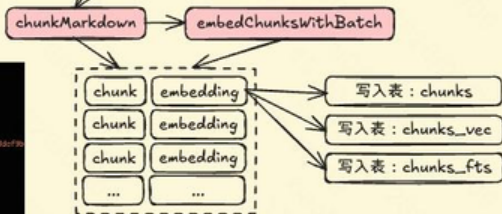
- memoryFlush (session自动压缩)
- 只在context tokens快满的情况下执行
- prompt: 记住我、记住这个、今天agent使用read_file, write_file保存记忆
- sessionFlush: /new 新session保存旧session到memory/<YYYY-MM-DD>-slug.md



变更触发(sync) : onSessionStart, onSearch, watch, session_delta...

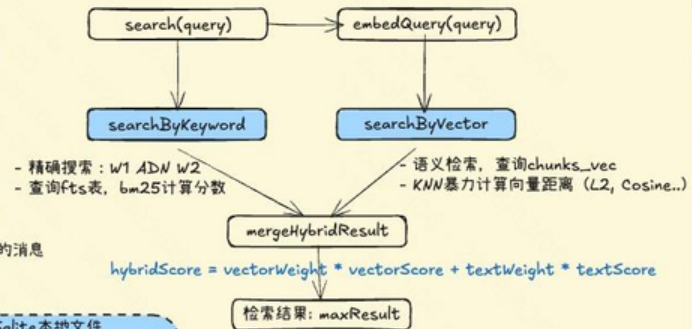


- 分行后根据chunkTokens分块 (默认400)
- 利用远程或本地embedding模型计算嵌入向量
- 前后分块可以重复chunkOverlap个token (默认80)
- 利用embedding_cache缓存, 避免重复计算



memoryTools: memory_search, memory_get

混合搜索 (hybrid search)

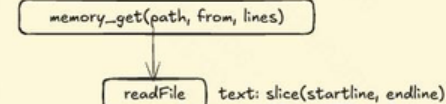


- 精确搜索: W1 A/DN W2
- 查询fts表, bm25计算分数
- 语义检索, 查询chunks_vec
- KNN暴力计算向量距离 (L2, Cosine..)

SQLite本地文件

- table: files
- table: chunks
- table: chunks_vec
- table: chunks_fts
- table: embedding_cache

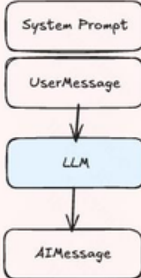
获取完整记忆文本



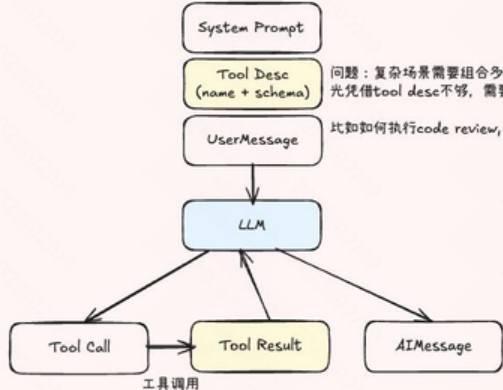
另一个重要机制skill

Skills的演化

无工具调用



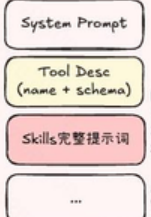
工具调用: ReAct



问题: 复杂场景需要组合多个工具
光凭借tool desc不够, 需要额外的提示词
比如如何执行code review, 如何做架构设计

总结: Claude Skills就是将AI如何学习各种复杂技能的提示词加载标准化
依赖: 文件系统读取工具
好处:
- 按需加载: 节约token
- 标准化: 写一个skill, 所有支持skill的agent都能使用 (比如codex, gemini-ch)
洞察: Skill这种加载形式并没有提升agent能力, Skill提示词内容才是关键
因此需要我们挖掘适合任务的好的Skills, 扩展agent能力边界。

通过skills增强能力 (本质就是补充提示词)

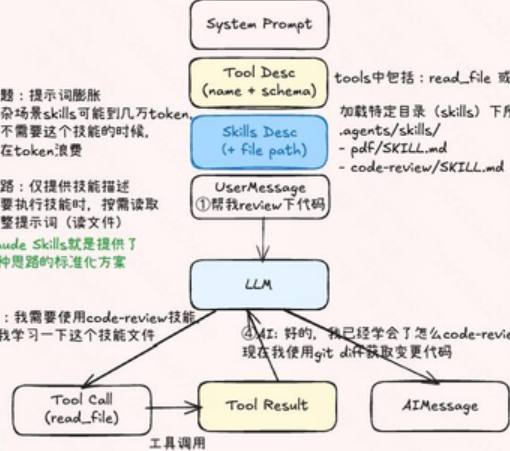


问题: 提示词膨胀
复杂场景skills可能到几万token,
在不需要这个技能的时候,
存在token浪费

思路: 仅提供技能描述
需要执行技能时, 按需读取
完整提示词 (读文件)
Claude Skills就是提供了
这种思路的标准化方案

②AI: 我需要使用code-review技能
先让我学习一下这个技能文件

渐进式披露: 利用文件系统+读写文件工具



tools中包括: read_file 或 bash tools能读文件

加载特定目录 (skills) 下所有SKILL.md文件的头部信息
- .agents/skills/
- pdf/SKILL.md
- code-review/SKILL.md

①帮我review下代码

③AI: 好的, 我已经学会了怎么code-review,
现在我用git diff获取变更代码

③read_file(.agents/skills/code-review/SKILL.md)

```
-- name: code-review
description: 在需要review代码的时候使用code-review技能
--
# 执行流程
1. 使用git diff获取commit变更代码
2. 分析变更代码是否存在漏洞、不合理的代码设计
3. ...
# 参考样例
1. 阅读 ./reference.md 文件学习历史Code review结果
2. ...
... (更多技能所需的提示词)
```

知乎 @小么