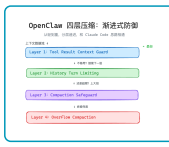
- 概览
  -
    最小代码实现
- Agent Loop
  - 流式响应
  - Try-Catch策略
    - 核心原则
      - 错误分类先于重试策略
      - 降级要分层，每层职责明确
      - 沉默故障比报错更危险
    - 可重试
      - 限流（429）、服务过载（529/503）、请求超时（408）、连接被重置（ECONNRESET）。
        - 服务端心跳 + 客户端看门狗（记录最后一次收到数据的时间，超过阈值就判定连接已失效）
    - 不可重试
      - 请求格式错误（400）、认证失败（401/403）、余额不足（402）。
    - 需要降级
      - 连续多次 529、流式连接反复断开。说明问题不是偶发的，单纯重试已经不够了，需要换策略。
        - 降级要分层，每层职责明确。偶发网络错误靠重试，协议层问题靠降级，服务级故障靠换模型或换 Provider。不要在同一层反复死磕
  - 循环保险策略
    - 工具死循环检测
      - 同一个工具 + 同一个参数被调用了 N 次。
      - 无进展轮询检测
      - 两个工具交替调用的 Ping-Pong 检测
      - 全局熔断器
    - Token 预算控制
      - 90% 时注入 nudge 消息「继续干活，别总结」
      - 检测递减回报「每次续写只说一点点新东西」，直接停止
    - 输出截断恢复
      - 提高上限
      - 注入恢复消息
      - 超过上限直接停止并把错误返回给客户端
  - 循环退出路径
    1. completed——正常完成
    2. max_turns——超过最大轮次
    3. aborted_streaming——用户在流式输出时中断
    4. aborted_tools——用户在工具执行时中断
    5. hook_stopped——钩子触发停止
    6. blocking_limit——上下文快满了，提前拦截
    7. prompt_too_long——上下文真的满了，但还有救
  - 模型调用与结果输出
    - Function Calling
      -
        Function Calling 的实现过程
    - MCP
      - 缺陷
        -
          Token 占用
        - 安全风险——Prompt Injection 的天然入口。一个恶意的 MCP Server 可以通过工具返回值来"操控"你的 Agent。
        - 复杂度。一个 MCP Server 需要：一个额外进程、一套配置、一条通信链路（stdio/SSE/HTTP）。如果涉及认证，还要处理 OAuth 流程
      - 管理
        - 三段式命名：mcp__<serverName>__<toolName>
        - 默认全部延迟加载
      - 适用场景
        - 有状态的 外部/远程 服务 连接/调用 。日常场景 Skill 优于 MCP
        - 需要交互式 UI（MCP Apps）
    - Skill
      -
        三层渐进式加载
        - Level 1：Frontmatter（永远加载，~100 token/skill）
        - Level 2：完整内容（按需加载）
        - Level 3：引用文件（再按需加载）
      -
        Skill vs MCP
      - 场景选择
        - 本地 Agent：Skills + CLI
        - 云端 Agent：Skills + MCP
    - Structured Output
      - 场景
        - 上下文摘要压缩
        - 生成式 UI
        - 信息提取
      - 具体实践
        - 用具体 enum 约束可选值而不用 type
        - 工具执行前进行错误校验
        - 清晰的错误反馈，给出具体改正
      - 采用标准
        - 推理阶段（Agent 在思考下一步该做什么）：不用 Structured Output，让模型自由生成文本
        - 动作阶段（Agent 决定调用哪个工具、传什么参数）：用 tool_use，约束输出格式
        - 输出阶段（需要固定格式的结果）：用 Structured Output

Tool System

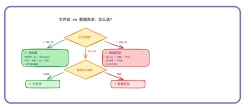
- Tools 设计
  - 描述要详细，至少 3-4 句话
  - 用命名空间前缀区分相似工具，可通过 response prefill 直接禁用不需要的工具
  - description 比 type 更重要
  -
    工具数量不宜过多
- Tools 执行流程
  - 管道校验
    -
  - 事件驱动
    - tool_execution_start → 记录开始时间、参数
      tool_execution_update → 中间状态更新
      tool_execution_end → 收集结果、触发后处理
  - 应用层的并发调度
    - parallel tool_use（原生）
      - 模型在一次响应中输出多个 tool_use 块，系统再把它们并行执行
    - Batch Tool （不依赖模型能力）
      - 模型只需要输出一个 batch 调用（这是一个普通的单工具调用），把多个子任务打包进参数里，并发的事交给应用层处理。
- Tools 数量过多优化策略
  - Deferred Tool Loading
    -
      只加载工具的描述，工具具体使用通过 ToolSearch 查找
  - Tool Profile
    - 根据场景动态加载列表
  - Atom Tools
    - 第一层：原子工具。固定的、最小化的 function calling 工具集。类似 file_write、browser_navigate、bash、search。
    - 第二层：CLI 工具。原子工具用不了的，让模型通过 bash 调用
    - 第三层：写脚本。bash 处理不了的，写自定义脚本实现
  -
- 权限拦截
  - 权限标准 Allow / Deny / Ask，可预配置多种权限模式
    -
  - 预制场景模式
    - 原则
      - 配置模式原则：Deny 规则的优先级高于 Allow 规则，Allow 规则的优先级高于模式默认行为
    - 例子
      -
        Claude Code 支持四种权限模式
        - Open'Code在 Claude Code 基础上的优化：
          1. 纠正反馈（Corrected Error）：用户拒绝时可以附带一段文字反馈。把"拒绝"从一个终止信号变成了一个学习信号——模型可以根据反馈调整后续行为。
          2. 拒绝级联：用户拒绝一个权限请求时，OpenCode 会自动拒绝同一 session 里所有正在排队的请求。反过来，用户选"总是允许"时，会自动检查排队中的请求，把匹配新规则的请求也一并放行。
      -
        OpenClaw 的五层规则过滤
        - 第一层：Profile Policy。预制多个场景工具集，不在当前 Profile 里的工具，连调用的机会都没有。
        - 第二层：Allow / Deny 白名单。为某些工具单独配置白名单，或者不同的模型提供商可以有不同的权限策略。
        - 第三层：Owner-only 工具。cron（定时任务）、gateway（网关控制）。这些工具能改变系统的全局行为，普通用户用不了
        - 第四层：Exec Approval。执行类工具（exec、shell、write、delete）需要显式审批。超时了会自动拒绝。
        - 第五层：Workspace 路径边界。所有文件操作被限制在 workspace root 之内。
  - 完整权限流程
    -
  - 权限系统设计原则
    - 操作分级是基础。把只读操作和写操作分开，把文件操作和 shell 命令分开，把可逆操作和不可逆操作分开。只读 + 可逆的操作默认放行
    - 规则要能记忆。提供 "总是允许这类操作"，下次同类操作应该自动放行，而不是再问一遍。或者直接通过 alwaysAllow 规则配置
    - 确认界面要有信息量。不是弹一个"是否允许？"就完了。好的确认界面应该告诉用户：这个操作要做什么、为什么需要审批、风险等级是什么。Claude Code 的权限提示会显示具体的命令内容和触发审批的原因（是因为规则还是 Hook）。
    - 尽可能自动化。大部分决策在规则层面自动完成，只有真正需要人判断的才弹出来。
  - 进阶方向：用 LLM 分类器做自动审批
    -
      用一个轻量 LLM 调用来做权限判断，否定优先原则，默认操作有问题
- Offload（卸载）
  - 把信息搬到外部文件系统（文件、数据库）
    - Manus 的"可恢复压缩 + 文件系统卸载"。
      Manus 给每个工具调用维护两个版本：完整版和紧凑版。完整版是原始工具输出，存在 sandbox 文件系统里；紧凑版只保留一个引用，比如文件路径。随着对话推进，老的工具结果从完整版切换到紧凑版。
- Reduce（压缩）
  - 就地缩小（compaction）或用摘要替换（summarization）
    -
      - Claude Summary 9 段结构——用户意图、技术概念、文件改动、错误修复、问题解决、用户消息、待办任务、当前工作、下一步
    - 
      - OpenClaw Summary 5 段结构：
        - Decisions——做了哪些决策
        - Open TODOs——还有哪些待办
        - Constraints/Rules——有什么约束和规则
        - Pending user asks——用户还有什么未回应的请求
        - Exact identifiers——关键标识符的精确值
      - Memory Flush：压缩之前先存档
        让 Agent 把当前会话中值得长期保存的信息写入记忆文件，然后再做压缩。

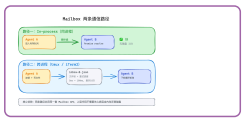
Agent 开发

- LLM模型
- Context Engineering
  - Retrieve（检索）
    - 从外部按需取回信息（RAG、文件读取）
      - JIT Context：记住索引，按需加载
    - RAG
      -
        RAG 流程
      -
        RAG 分块策略
      - 优化策略：
        - 混合检索——向量搜索理解语义，关键词搜索精确匹配，70/30 加权合并。别忘了分数归一化，否则合并没有意义。
        - Query 改写——HyDE 用假回答做检索，子问题分解拆复杂查询。本质是在用户的问题和文档的内容之间构建一道桥梁。
        - Reranker 精排——Bi-encoder 快速召回，Cross-encoder 精准排序。两阶段架构，不加 Reranker 的 RAG 在生产环境里基本不及格。
        - 多样性和时效性——MMR 避免结果高度重复，时间衰减让旧信息排后面。按需开启，别过度优化。
    - 编译知识库
      - schema（操作规范）
        告诉 LLM 这个 wiki 的结构约定、命名规范、工作流程。这一层是人和 LLM 共同维护的——人定规则，LLM 按规则执行。
      - 三个操作——Ingest（摄入新素材，触发多页面更新）、Query（在 wiki 上检索）、Lint（定期巡检矛盾和缺口）。
        -
      - 三级缓存架构
        - Layer 0：index.md
          常驻 system prompt，每个 wiki 页面一行，包含链接和一句话摘要，
        - Layer 1：wiki/（编译产物）
          LLM 生成和维护的结构化 Markdown 文件。这一层完全由 LLM "拥有"——它负责创建页面、更新内容、维护交叉引用、保持一致性。
          wiki 里面存的是经过整合的知识：实体页面（某个人、某个产品、某个概念的全貌）、对比分析（A 和 B 的区别）、综述页面（某个领域的全景）、时间线（事件的发展脉络）。LLM 在整合信息的时候可能会理解偏差、遗漏关键细节、甚至引入错误的关联。所以 wiki 页面不应该被当成"最终真相"——它更像是一个高质量的索引和摘要层。当你需要精确数据或者做重要决策的时候，还是要回溯到 raw/ 里的原始文档去核实。
        - Layer 2：raw/（原始素材）
          收集的所有原始材料——文章、论文、PDF、会议记录、截图、数据文件。这一层是只读的，LLM 可以读但不能改，原始文档仅在必要时回溯。
    - 最佳实践
      - **直接用编译知识库的场景：** 个人或小团队的知识体系，文档量不大（几十到几百篇），内容相对稳定，你需要知识持续积累和跨文档关联。比如个人研究笔记、团队内部 wiki、竞品分析库、课程知识体系。这种场景下编译知识库就够了，不需要再搭一套 RAG 管线。
      - **直接用 RAG 的场景：** 文档量大（几千几万篇），内容频繁变动，不可能也没必要全部"编译"。比如客服知识库、产品文档中心、法律法规库。这种场景下老老实实用 RAG，分块、embedding、向量搜索那一套。
  - Isolate（隔离）
    - 拆成多个独立的上下文（Multi-Agent）
  - Cache（缓存）
    - 复用已有的计算结果（KV Cache、Prompt Cache）
      - KV Cache：让前缀稳定，间接提高命中率（前缀稳定 = 命中）
      - Prompt Cache：不同模型厂商的显式标记（cache_control 标记或者显式创建缓存）和隐式缓存加
      - Context Collapse：可逆的上下文折叠，灵活性最高（应用层可逆压缩）
  - 最佳实践
    - Prompt
      - Chat Prompt
        - 追求"回答质量"——问一个问题，给一个好答案。
      - Agent Prompt
        - 追求"行为稳定性"——模型可能连续跑 50 轮，每一轮都要做出正确决策。
      -
        Agent 的 system prompt 不是"提示词"，而是一套行为控制系统。你不是在"提示"模型，你是在"编程"模型的行为模式
        - 怎么分模块？ Prompt Pipe 模式，每个 section 是一个独立函数，自己决定要不要出现
        - 怎么设计缓存？ 静态/动态分界线，不变的放前面做 cache prefix，变的放后面或走消息注入
          -
        - 怎么让用户自定义？ 配置分层加载，越具体优先级越高
        - 怎么对抗 Context Rot？ 控制入口，与其压缩不如一开始就少放东西
    - Offload/Reduce
      - 上下文管理基本顺序：
        1. 先管住入口——不必要的信息不要放进来（Cursor 的按需加载、Manus 的小工具集、Claude Code 的 Deferred Tool Loading）
        2. 再管住工具结果——这是最大的 token 消耗源（截断、compact、清除）
        3. 然后做对话压缩——Compaction / Summarization
        4. 需要的话拆 Agent——上下文隔离（目前是超纲内容，第六章会详细讲）
    - Cache/Cost
      - 最大化缓存命中率
      - 模型路由，不是所有步骤都需要最贵的模型：
        1. 按任务类型静态路由
        2. 按难度动态路由
    - Retrieve
      - RAG（预建索引、语义检索）、Agentic Search（运行时工具探索）、Context Offloading（主动卸载 + 按需恢复）。
    - Memory
      - CLAUDE.md 是静态上下文，Memory 是动态上下文。前者解决冷启动，所有人共享的基线规则。后者解决个性化，每个人独特的偏好和习惯。两者配合起来效果才最好。
  - 与其他模块关系
    -

- Memory
  - 原则：记忆系统只存那些"只存在于对话中、无法从其他地方获取"的信息，防止记太多噪音把有用信息淹没
    -
      类比计算机系统
  - 选择策略
    - 文件派：Claude Code 的 MEMORY.md
      Claude Code 的记忆系统是一个"索引 + 正文"的双层结构
      -
      - 第一层：MEMORY.md（索引文件）
        每一行是一条指针，指向一个具体的记忆文件：
        - [用户偏好](user_preferences.md) — 后端出身，前端不熟，偏好简洁代码
        - [项目规范](project_conventions.md) — pnpm，vitest，commit 用中文
        - [部署拉闸](deploy_freeze.md) — 2026-04-15 前不合并非关键 PR
        MEMORY.md 在每次会话开始的时候会被加载到 system prompt 里——所以它必须短
      - 第二层：独立记忆文件
        每个记忆是一个独立的 .md 文件，带 YAML frontmatter：
        ---
        name: 项目规范
        description: 包管理器、测试框架、commit 规范等项目约定
        type: feedback

        用 pnpm 不要用 npm。
        Why: 用户明确要求，团队统一规范。
        How to apply: 所有 install/add 命令用 pnpm。

        测试用 vitest 不要用 jest。
        Why: 项目已配好 vitest，jest 会引入额外依赖。
    - 数据库派：OpenClaw 的 SQLite + 向量
      -
      - 用 SQLite 加 sqlite-vec 加 FTS5 三管齐下，混合检索向量占七成、关键词占三成，配合时间衰减和 Memory Flush，精准可扩展，适合大规模场景。
    - 最佳实践
      - 
  - 记忆维护
    -
      记忆的五种失效模式

- Multi-Agent
  - 父子模式
    - 目的：拆上下文
      -
        子 Agent 结果回传
    -
      多 Agent 隔离思路
    -
      并发安全
    - 子 Agent 的任务结束需要主 Agent 审批
  - 团队模式
    - 1. 双向通信问题
      - 
      - 跨进程的 Agent 基于文件的消息系统：每个团队成员在磁盘上有一个 文件当收件箱，要给某个 Agent 发消息，就往它的收件箱文件里追加一条记录，Agent 在每轮 Agent Loop 的间隙检查自己的收件箱，有新消息就读取处理。
      - 同进程的 Agent 直接在内存里传递消息：发送方把消息放进一个内存队列，接收方那边有一个 Promise 在等着——消息一到，Promise 立刻 resolve，接收方马上就能处理。
    - 2. 状态共享
      - 共享任务列表，防止"抢活"。给每个团队起一个团队名字作为任务目录——不管 Agent 从哪种方式（in-process、tmux、iTerm2）启动，同一团队成员通过团队名字解析到同一个目录（~/.claude/teams/{team-name}/*），同时通过文件锁 + 重试的方式防止同时修改目录文件
    - 3. 权限代理
      -
      - 权限请求转发——Worker 把请求写到一个 pending/ 目录，Leader Agent 定期轮询这个目录，看到请求后呈现给用户。用户批准或拒绝后，结果写入 resolved/ 目录，Worker 读取结果继续（或放弃）执行，如果两者在同一进程则直接通过内存传递请求
    -
      总结：本质也是父子模式（Leader+Worker），只是Worker之间需要通讯
  -
  -
    行业其他方案：hub-and-spoke（中心化）、handoff（接力）、groupchat（广播）。选择哪种取决于你的场景：Agent 之间需要频繁来回协商用 hub-and-spoke；任务是线性流转的用 handoff；所有 Agent 需要看到全局信息用 groupchat。

Harness Engineering

- 模型本身两大问题：
  1. 上下文退化：模型不只是因为上下文太满而表现下降，它还会主动偷懒：感知到上下文快满的时候，开始跳过步骤、简化回答、不调用工具。
  2. 自我认知偏差：模型在评估自己产出的时候，会不自觉把问题合理化，夸大自己。
- Harness的本质
  - 第一层是"能力补偿"——补模型做不到的事。这一层会随着模型自身能力的提高会越来越薄
  - 第二层是"系统工程"——处理模型从定义上就做不了的事：
    1. 安全和权限
    2. 成本控制
    3. 外部系统交互
    4. 可观测性
    5. 失败恢复
- Harness 设计原则
  - 概率的归模型，确定的归工程
    - 那些"绝对不能出错"的地方，用确定性手段：权限靠白名单和规则匹配，循环检测靠哈希对比，多 Agent 并发写文件靠文件锁，崩溃恢复靠 append-only 的 JSONL 日志。
    - 模型负责把事做对，工程负责错了也别出大事。这两层职责一旦混了——比如让模型自己决定要不要放行一条危险命令——系统的安全下限就从"确定"滑向了"概率性事件"，在有些严肃的工程场景下这是不能接受的。
  - 每个组件都是一个假设，而假设会过期
    - 每次 Agent 犯错，分析根因，加一个最小组件防止再犯。 不要提前设计——你不知道 Agent 会在哪里犯错，等它犯了再补
    - 每次模型升级，重新测试每个组件，删掉不再需要的。
    - 每次模型升级，重新测试每个组件，删掉不再需要的。
    - 成功应该是不发声的，只有失败才需要发出声音。 比如一开始每次代码修改后都跑完整的测试套件，4000 个通过的测试结果直接灌进上下文——Agent 的上下文被无用信息淹没了，改成"通过了就沉默、失败了才输出"之后，效果就好了很多。
    - 如果 Agent 跑完你总是不满意，但它自己觉得很好——加 Evaluator。
    -

- Generator/Evaluator：最经典的 Harness 模式
  - GAN（生成对抗网络）：生成和判别必须分开。Generator 负责写代码、做功能，Evaluator 负责验收。
  -

    Evaluator 设计原则
    - 先对齐验收标准，再开始干活。 Anthropic 设计了一个叫 Sprint Contract 的机制——每个阶段开始前，Generator 和 Evaluator 先协商好"什么算做完了"。比如一个 Sprint 的 Contract 可能有 27 条验收标准，涵盖具体的功能点、交互行为、边界情况。验收标准写清楚了再动手，而不是做完了再讨论"这算不算合格"。这个思路其实跟软件工程里的 TDD（Test-Driven Development）很像——先写测试、再写代码。只不过这里的"测试"是 Evaluator 手里的验收标准。
    - Evaluator 不看推理过程，只看最终输出。防止 Evaluator 被 Generator 的推理"污染"，变成"理解 Generator 的意图"而不是"评判产出的质量"。
    - 用实际操作来验收，不是读代码。用 Playwright 像用户一样实际操作页面。根据操作结果来判断"体验好不好"，而不是根据代码。这个区别很重要，读代码你会不自觉地替 Generator 找借口（"虽然 CSS 有点乱但逻辑是对的"）。
- 可观测性
  - 可观测性工具
    -

      Gateway：钱花在哪了
      Langfuse：哪一步出了问题
  - 成本监测
    - 成本熔断器——给每个会话或每个任务设一个成本上限，超了就强制停止。
  - Cache 命中率
    - 如果 Agent 每次对话的 Cache 命中率都很低（比如低于 50%），说明你的上下文前缀不够稳定——可能是 system prompt 在变、工具列表在变、或者对话结构不够一致。
  - 死循环和重复工具调用
    - Agent 可能会陷入重复调用同一个工具、传同样参数的循环。可观测性系统应该能自动识别这种模式并告警——比如"同一个工具在最近 5 轮内被调用了 3 次以上、且参数完全相同"。这种告警比简单的错误率告警有价值得多，因为死循环不会产生"错误"——每次调用都"成功"了，只是在做无用功烧钱。
- 崩溃和恢复
  - Postgres 存所有数据（对话、工具调用、状态）
  - Redis 缓存热数据（活跃会话、速率限制、分布式锁）
  - 工具设计尽量幂等（崩溃重跑不出问题）。
- 沙箱隔离
  - 云端 Agent 用 microVM 服务（如 E2B）
  - 本地 Agent 用 OS 级权限控制
- 部署
  -

通信协议

- MCP & ACP
  -

  - 流程演示
    -

- A2A
  -

- 通信概览
  -

框架评估标准

1. **Loop**：循环谁控制？你能自定义退出条件吗？有没有保险丝？
2. **Tool**：工具怎么注册？运行时能动态加载吗？MCP 是原生支持还是 adapter 支持？
3. **Context**：有没有压缩机制？有没有 Cache 意识？上下文爆了框架帮你处理吗？
4. **Memory**：跨会话记忆是内置还是外挂？有没有向量检索？
5. **Multi-Agent**：多 Agent 怎么协作？上下文怎么隔离？有没有深度限制？
6. **Harness**：保险丝、权限控制、Generator/Evaluator 这些 Harness 零件，框架有没有内置？还是全要自己搭？

Presented with xmind