从 Claude Code 到 Koder：我为什么要自己写一个 Coding Agent

原创 Feisky Feisky 2026年1月5日 21:18 上海

这两年 AI 编程助手火得一塌糊涂。GitHub Copilot、Cursor、Codex、Claude Code……工具一个比一个强，用起来确实爽。只需要用自然语言描述需求，AI 就能帮你写代码、改 Bug、跑测试，甚至排查之前让你绞尽脑汁的线上问题。

与此同时，Anthropic、OpenAI 这些前沿公司也在持续分享他们构建 Agent 的实践经验。比如虽然 Anthropic 家的模型在国内禁用了，但他们的 Engineering Blog 简直是个宝藏，持续分享很多关于 AI Agent 构建的实践干货，每次读完都有意犹未尽的感觉。

作为一个工程师，光看这些文章还不过瘾，忍不住就想动手实现一下。看到工具设计的最佳实践，就想自己写几个试试效果。这种“看完就想写代码”的冲动，大概是职业病吧。

但用得越多，我越有一个困惑。

这些 Agent 到底是怎么工作的？不是说“LLM + 工具 + 循环 ”这种笼统的回答。我想知道的是更具体的东西——它怎么决定什么时候读文件、什么时候执行命令？工具调用是怎么串起来的？内存怎么管理？上下文工程是怎么串起来的？

说白了，**会用 AI Agent 和理解 AI Agent 背后的详细原理，是两回事**。

那就自己造一个吧

想明白这件事之后，我决定自己动手造一个 Coding Agent。不是为了重复造轮子，也不是觉得 Claude Code 不够好。纯粹是想搞清楚这个黑盒里面到底装了什么。

Agent 听起来很玄乎，但拆开来看，核心就是 LLM + 工具 + 循环。LLM 思考，决定用什么工具，执行工具，拿到结果，继续思考……如此往复。这个循环里藏着很多细节——工具怎么定义？结果怎么反馈给 LLM？什么时候该停下来？怎么处理错误？这些问题，只有自己实现一遍才能真正搞懂。

还有一个原因是想有个试验田。看到一篇论文说“某某方法能提升 Agent 表现”，想试试？用 Claude Code 你没法定制化它的内部工作流程。但如果你有自己的 Agent，分分钟就能加上去跑一下看效果。

自己的 Agent 就是自己的试验田，想种什么种什么，想怎么折腾怎么折腾。

于是有了 Koder

基于这些想法，我写了 Koder——一个实验性质的终端 AI 编程助手。

先说清楚定位。这是一个学习/研究型项目。它不是要取代 Claude Code 或 Cursor，而是提供一个可以研究、可以魔改、可以学习的开源实现，并且使用了我一直都熟悉的 Python 开发。底层基于 OpenAI Agents SDK 构建，省去了自己造轮子实现 Agent 循环的麻烦，可以专注在工具设计和上下文工程上。

功能上，该有的基本都有。通过 LiteLLM 支持 OpenAI、Anthropic、Google、GitHub Copilot 等 100 多个模型提供商，也可以用 Claude Pro/Max、ChatGPT Plus、Google Gemini、Antigravity 这些订阅服务，不用单独买 API。

基本的工具系统也已经有了，除了内置的文件读写、Shell 命令、Todo 管理等常用工具之外，Koder 也支持 MCP、Skills、AGENTS.md、会话管理（用 SQLite 存储）、Token 和费用跟踪等常用的功能。

安装也很简单，推荐用 uv，速度快不说，依赖也完全隔离，不会污染其他 Python 项目：

```
uv tool install koder
```

当然，pip 安装也是支持的：

```
pip install koder
```

上手试试

安装完之后，设置一下 API Key 和模型就能用了：

```
export OPENAI_API_KEY="your-api-key"
export KODER_MODEL="gpt-5.2"
koder
```

Koder 会自动检测你的模型来决定用哪个 Provider。如果你有 Claude Max 或 ChatGPT Plus 订阅，也可以用 OAuth 登录：

```
koder auth login claude      # Claude Pro/Max
koder auth login chatgpt     # ChatGPT Plus
koder auth login google      # Google Gemini
koder auth login antigravity # Google Antigravity
```

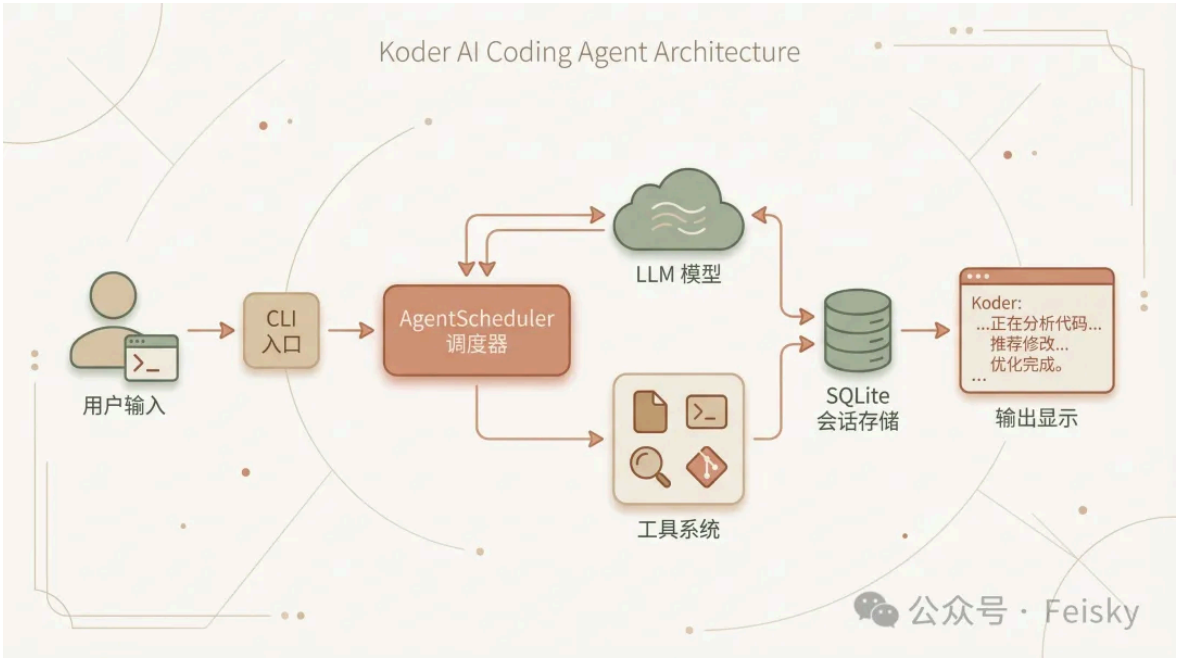
```
# 登录后设置模型开始使用
export KODER_MODEL="claude/claude-opus-4-5-20251101"
koder

export KODER_MODEL="chatgpt/gpt-5.2"
koder
```

基本使用方式和其他命令行 AI Agent 差不多。直接运行 `koder` 进入交互模式，或者 `koder "你的问题"` 单次提问。想指定会话名的话加个 `-s` 参数，起个自己的会话名字。所有会话都会存到 `~/.koder/` 中，下次可以继续聊。

除了环境变量，Koder 也支持通过修改 `~/.koder/config.yaml` 来设置模型、参数以及 MCP。这些功能的文档在项目 README 里都有，这里就不展开了。

拆开看看里面有什么

写这个项目的过程中，我对 Agent 架构的理解深了不少。先看整体流程，再聊几个有意思的设计。



Koder 工作原理

敲下回车之后发生了什么

用户敲下 `koder "帮我写个XXX函数"` 之后，发生了什么？

CLI 入口 `cli.py` 先解析参数，加载当前目录的 `AGENTS.md` 作为项目上下文。这个设计应该是所有 AI Agent 的标配了——每个项目可以有自己的指令文件，告诉 Agent 这个项目的技术栈、代码规范、常用命令。

接下来是关键的懒加载设计。`AgentScheduler` 初始化时并不创建 Agent，而是等到真正收到第一条消息时才调用 `_ensure_agent_initialized()`。为什么这么做？因为创建 Agent 要加载工具、连接 MCP 服务器、初始化模型客户端，挺重的。如果用户只是想跑个 `koder config show` 查看配置，没必要把整个 Agent 都拉起来。

调度器怎么指挥全局

`AgentScheduler` 是整个系统的指挥中心。它维护了一个信号量 `asyncio.Semaphore(10)` 来控制并发，防止同时跑太多请求把 API 打爆。

流式输出的处理是我花时间最多的地方。用户看到的是文字一个字一个字蹦出来，背后其实有不少门道。

问题出在哪？Agent 的输出是交错的。它可能先说一句话，然后调用工具，工具还可以多个一起调用，工具返回结果后继续说话，中间还可能穿插更多工具调用。如果不做处理，用户看到的就是一团乱麻。

`StreamingDisplayManager` 用了一个简单但有效的方案。它维护一个 `pending_tool_calls` 计数器，Agent 每发起一次工具调用就加一，工具返回结果就减一。关键在于：只有计数器为零的时候，文字才会立即显示。工具调用期间产生的文字会先攒着，等工具执行完再一起输出。

还有个细节是怎么把工具结果匹配回对应的调用。`active_tool_calls` 字典用 `call_id` 做 key，工具结果返回时根据 id 找到对应的调用记录。如果 id 匹配不上（有些模型不返回 id），就退化成先进先出的队列模式。

还有个小细节是 ESC 键取消。在 Unix 终端下，通过 `select.select()` 做非阻塞键盘检测，用户按 ESC 可以立即中断当前执行。这个功能实现起来不难，但对用户体验提升很大。在 Agent 跑偏的时候，能快速止损。

工具怎么注册和管理

工具用装饰器注册，类似这样：

```
@function_tool
def read_file(model: FileReadModel) -> str:
    # 实现
```

`get_all_tools()` 函数收集所有工具，然后给每个工具动态挂上 `skill_restriction_guardrail`。这个守卫会检查当前激活的 Skill 是否允许使用某个工具。比如你定义了一个只能读写文件的 Skill，它就没法调用 Shell 命令。

这种设计的好处是工具定义和权限控制解耦。写工具的时候不用操心权限，守卫统一处理。

聊太多了怎么办

`EnhancedSQLiteSession` 继承了 OpenAI Agents SDK 的 `SQLiteSession`，加了一个关键功能——token 感知的上下文压缩。

每次添加消息时，它会用 tiktoken 估算当前对话的 token 数。超过阈值（默认 50k）就触发压缩：保留所有用户消息（代表意图），把 Assistant 的回复做摘要。这样既不会丢失关键信息，又能控制 context 长度。

当然这个实现还是比较简陋的，很多 Context Engineering 的实践和想法暂时还没有加进来。

API 抽风了怎么办

LLM API 偶尔抽风是常态，适当加些错误处理和重试是必要的。`RetryingLitellmModel` 封装了 backoff 库的指数退避重试：

```
@backoff.on_exception(
    backoff.expo,
    (ServiceUnavailable, RateLimit, APIConnection, Timeout),
    max_tries=3,
    jitter=backoff.full_jitter,
)
async def get_response(...):
```

遇到限流、超时、服务不可用，自动重试最多 3 次，每次间隔指数增长加随机抖动。这个在高峰期特别有用，很多时候第一次失败、第二次就过了。

写在最后

对工程师来说，最好的学习方式从来都不是看文档、看论文，而是动手写代码。

Agent 这个领域也一样。概念可以很快理解，“LLM + 工具 + 循环”一句话就能说清楚。但真正的门道都藏在细节里：工具怎么设计才好用？上下文怎么管理才高效？用户体验怎么打磨才顺手？这些东西，只有自己踩过坑才能真正明白。

自己写过一遍之后，再看 Claude Code 或 Cursor 的设计，感觉完全不一样了。很多以前觉得理所当然的功能，现在知道背后藏着多少细节。

Koder 的代码完全开源，欢迎来看看、试试、提 PR。项目地址在 GitHub https://github.com/feiskyer/koder。

在 AI Agent 大潮中，不要只做一个旁观者。以身入局，才能真正理解这场变革。

好了，今天就聊到这儿。如果你也在探索 AI 工具和云原生技术，欢迎关注 Feisky 公众号，我会定期分享实践中的发现和踩坑经验。

Feisky

K8S | AI Infra | AI Agent 开源爱好者 | 极客时间专栏作者 AI实践分享 | 让AI成为...

122篇原创内容

公众号