第 1 章：为什么写 Nimo

我做项目管理工作，日常离不开 TAPD 和 SVN——查需求状态、翻任务列表、看迭代进度、填工时、查提交日志。这些操作本身不复杂，但**频繁且琐碎**。

典型的场景：领导问"这个迭代的 bug 修得怎么样了"，我的操作流程是：打开 TAPD → 找到项目 → 进入迭代 → 筛选 bug 列表 → 看状态分布 → 截图发群里。整个过程 5-10 分钟，一周发生好几次。

那时候 Function Calling 已经是 LLM API 的标配能力。我突然想到：能不能做一个 CLI 工具，我用自然语言说一句"查下这个迭代的 bug 情况"，它自己调 TAPD API 去查、自己整理成报告？

就有了 Nimo。

1.1 Nimo 能做什么

Nimo 是一个命令行 AI Agent。启动后看到一个简洁的欢迎画面：

```
╭──────────────────────────────────────────────────────────╮
│    ╭──╮   Nimo v1.0.0                                   │
│    │· ·│   deepseek-v4-flash · Nimo Agent               │
│    ╰──╯                                                  │
│      /help 帮助  ·  /chain 调用链  ·  /clear 清除        │
╰──────────────────────────────────────────────────────────╯
```

然后用自然语言和它对话：

```
> 帮我看看有哪些项目
╭─ Nimo ──────────────────────────────── tapd query ─╮
  共有 13 个项目/产品/组织：
  | 序号 | 名称                | ID        | 类型 |
  | 1    | 敏捷研发管理        | 22674041  | 产品 |
  | 2    | SKY Android 维护    | 34308359  | 项目 |
  ...
╰──────────────────────── P:11.5k C:602 ──────────────╯

> 列出敏捷研发管理的所有活跃迭代
╭─ Nimo ──────────────────────────── tapd query x 2 ─╮
  项目 22674041 共有 3 个活跃迭代：
  | 迭代名称         | 起止日期               |
  | 2024-Q4 Sprint   | 2024-10-01 ~ 2024-12-31 |
  ...
╰──────────────────────── P:13.2k C:480 ──────────────╯

> 给任务 1001 填 4 小时工时，备注"需求评审"
✅ 已填写工时：任务 1001（4h）

> 看看今天的 SVN 提交记录
╭─ Nimo ──────────────────────────────── svn op ──────╮
  今天共 7 次提交：
  | 版本   | 作者    | 时间   | 说明               |
  | r4521  | 张三    | 09:15  | fix: 登录页样式修复 |
```

```
  ...
╰──────────────────────── P:5.1k C:320 ─────────────────╯
```

它就是我要的东西——一句自然语言，自动完成原本需要几次手动操作的事情。

1.2 Nimo 的架构全景

Nimo 的内部架构是这样的：

```
用户输入 → Agent 核心循环
              ├─ 理解意图 → 调 LLM（DeepSeek）
              ├─ 选择工具 → ToolRegistry 自动分发
              ├─ 执行操作 → ExecutionEngine 确定性执行
              ├─ 记住上下文 → ConversationHistory 滑动窗口
              └─ 扩展能力 → Skill 系统（即插即用的外部能力包）
```

这个架构不是一天设计出来的，而是经过反复迭代——从最初 40 行代码的 Hello World，到后来加入记忆管理、执行引擎、Skill 系统、定时任务。每次加新能力，都会暴露之前设计的不足。

这本书就是带你回顾这个过程。我们不会直接拆解 Nimo 的源码，而是**从零构建一个叫 MiniAgent 的教学项目**，每章解决一个问题，最终得到一个架构完整的 Agent。

1.3 Agent 的三要素

虽然 Nimo 做的事情很具体（TAPD + SVN），但它的核心机制适用于所有 Agent。

Agent 由三部分组成：

- **LLM 是大脑**：理解你的自然语言，判断"你想做什么"，选择用什么工具
- **Tool 是手脚**：执行具体操作——调 TAPD API、跑 SVN 命令、读文件
- **Agent Loop 是神经系统**：大脑做出决定 → 手脚执行 → 看结果 → 再思考 → 再动手，直到任务完成

传统程序是"预设执行流程"——你写好了第 1 步干什么、第 2 步干什么，它照着做。Agent 是"理解意图后自主选择行动"——你告诉它目标，它自己决定走哪条路。

举个例子。用户说"查一下敏捷研发管理项目这个迭代的 bug 情况"。

Agent 内部的执行过程可能是：

```
第 1 轮：LLM 判断"需要先知道项目 ID"
  → 调 workspace_list() 拿到 22674041
第 2 轮：LLM 判断"需要知道有哪些迭代"
  → 调 iteration_list(workspace_id=22674041)
第 3 轮：LLM 判断"选当前迭代，查 bug"
  → 调 bug_list(workspace_id=22674041, iteration_id=...)
第 4 轮：LLM 分析结果，生成总结
  → "本迭代共 23 个 bug，已修复 18 个，5 个待处理。其中 2 个高优先级..."
```

这个流程不是写死的——LLM 在第 1 轮看到项目列表后，自己决定下一步查迭代还是直接查任务。这就是 Agent 的智能所在。

1.4 Function Calling：让 LLM 能"动手"

Agent 能工作的基石是 **Function Calling**。

2023 年，OpenAI 在 API 中加入了 Function Calling 能力。在此之前，LLM 只能输出文本。你想让它做点什么？自己解析它的输出，自己判断要不要执行。

Function Calling 改变了这个模式。你告诉 LLM 有哪些工具可以用，它返回的不是"我觉得应该查一下项目"，而是**结构化的函数调用指令**：

```
{
  "name": "workspace_list",
  "arguments": {}
}
```

你的程序收到这个指令，执行 `workspace_list()`，把结果发回给 LLM。LLM 再基于结果决定下一步。

关键一句话：**LLM 不执行函数，它只是告诉你"我想调这个函数，参数是这些"。** 真正执行的是你的代码。这个"大脑"和"手脚"的分离，是整个 Agent 架构的核心。

1.5 为什么不用框架

市面上有 LangChain、AutoGPT、CrewAI 等框架，封装了 Agent 循环、工具管理、记忆系统。我没用它们，原因是：

- **调试成本**：框架内部的抽象层太多。出了问题追进去，是 `AgentAction → AgentFinish → AgentStep` 的嵌套调用。自己写的 50 行循环，一眼看到底。
- **框架迭代快**：去年用 LangChain 写的 Agent，今年 API 已经 breaking change 了。原生 Python + OpenAI SDK 的 API 稳定得多。
- **学到的是框架用法，不是 Agent 原理**：用框架能 30 分钟搭一个 demo，但你不理解它为什么 work。用原生代码写，你能拆开每一个决策——为什么这里用 `asyncio.gather`、为什么循环检测是 3 次而不是 2 次。

这本书只用 Python 标准库 + OpenAI Python SDK。写完你不会成为"XX 框架专家"，但你真正理解 Agent 的底层运作方式。以后再用任何框架，你能一眼看穿它的设计意图和局限。

1.6 主线项目：MiniAgent

这本书的项目驱动方式是：**每章解决一个问题，在上一章的基础上增加一块能力。**

我们构建的主线项目叫 **MiniAgent**，它是一个通用的 CLI Agent（不绑定 TAPD/SVN），架构直接来自 Nimo 的实战经验。

章节	新增能力	MiniAgent 能做什么
第 2 章	Hello World	调用一个工具，跑通最小闭环
第 3 章	工具注册系统	用装饰器注册工具，自动发现新工具
第 4 章	核心循环	多轮对话，并行执行，循环检测
第 5 章	记忆管理	记住上下文，超出窗口时智能摘要

章节	新增能力	MiniAgent 能做什么
第 6 章	配置系统	YAML 配置 + 环境变量 + 用户画像
第 7 章	执行引擎	编排与执行分离，批量操作模式
第 8 章	意图工具	LLM 用结构化参数而非 CLI 字符串
第 9 章	Skill 系统	可安装的外部能力包，即插即用
第 10 章	定时任务	Cron 定时触发，后台自动执行
第 11 章	动态执行	LLM 写 Python 代码处理数据
第 12 章	CLI 打磨	美观的终端界面，进度动画
第 13 章	流式响应	SSE 流式输出，增量 tool_call 处理
第 14 章	测试策略	Mock LLM，测试隔离，覆盖率
第 15 章	错误韧性	重试、降级、上下文超限恢复
第 16 章	安全加固	注入防护，路径遍历，沙箱隔离

1.7 每章的"Nimo 实战笔记"

这本书的独特之处在于：**每个设计决策背后，都有一个真实的迭代故事。**

每章末尾有一个"Nimo 实战笔记"小节。它不是泛泛的"最佳实践"，而是：

- Nimo 的 git 提交历史中真实的 bug 和修复
- 教学代码和 Nimo 真实实现的差异
- "如果重来一次我会换种写法"的反思

你在书中读到的循环检测机制、profile 重构、Skill 系统的循环导入解决——这些都不是凭空编造的，而是我从凌晨调试的终端里带出来的。

1.8 这本书不做什么

- **不讲 LLM 基础**：你应该已经用过 OpenAI API 或 DeepSeek API，知道怎么调 chat/completions
- **不讲 Prompt Engineering 大全**：prompt 只在具体场景中出现（system prompt 设计、工具选择引导）
- **不教 Web 开发**：MiniAgent 是 CLI 应用
- **不做性能 benchmark**：关注架构设计和工程实践

1.9 你需要准备什么

- Python 3.10+，能写 async/await
- `pip install openai`（兼容 DeepSeek）
- 一个 LLM API Key（DeepSeek 或 OpenAI 任选）
- 愿意动手跟着写代码

每章配套教学代码下载：github.com/FuZhengCN/nimo-agent → `book/code/` 目录。Nimo 完整源码也在同一仓库。

下一章，我们写第一个 40 行的 Agent，让 LLM 能调用工具。

第 2 章：Hello World Agent

上一章我们讲了 Agent 是什么。这章我们写一个能跑起来的。

目标很简单：用户说一句话，Agent 根据需要调用工具，然后回复。整个过程不超过 80 行代码。

2.1 最简架构

搞清楚我们要做什么。整个流程只有 4 个步骤：

```
用户输入 → LLM 分析 → （如果需要）执行工具 → LLM 基于结果回复
```

拆开来看：

1. 用户的自然语言作为 message 发给 LLM
2. LLM 判断：这个问题我能直接回答，还是需要调用工具？
3. 如果需要工具，LLM 返回一个 tool_call（工具名 + 参数），我们执行它
4. 把工具执行结果发回给 LLM，让它生成最终回复

代码分三个文件：

- `llm_client.py`：封装 OpenAI SDK 调用
- `agent.py`：Agent 核心逻辑（最简的循环）
- `main.py`：命令行入口

2.2 封装 LLM 调用

先把和 LLM 通信的部分封好。OpenAI SDK 的调用很简单，但我们需要一个自己的一层——不是为了过度设计，是为了后面加重试、加超时、加错误处理的时候不改 Agent 的代码。

```python
# llm_client.py
from openai import AsyncOpenAI


class LLMClient:
    def __init__(self, api_key: str, base_url: str, model: str):
        self.client = AsyncOpenAI(api_key=api_key, base_url=base_url)
        self.model = model

    async def chat(self, messages: list[dict], tools: list[dict] | None = None) -> dict:
        kwargs = {"model": self.model, "messages": messages}
        if tools:
            kwargs["tools"] = tools
        response = await self.client.chat.completions.create(**kwargs)
```

```
        return response.model_dump()
```

注意 `tools` 是可选的。Agent 有时候需要 LLM 不带工具直接回复（比如轮数耗尽后的降级总结），所以 `tools=None` 时就不传。

用 `model_dump()` 把 SDK 的 ChatCompletion 对象转成普通 dict，避免后续代码依赖 SDK 内部类型。这样以后换模型、换 SDK，只要改这一个文件。

2.3 注册你的第一个工具

Function Calling 的核心是告诉 LLM"你能用什么工具"。每个工具是一个 JSON Schema，描述名字、用途、参数：

```
# agent.py
TOOLS = [
    {
        "type": "function",
        "function": {
            "name": "get_current_time",
            "description": "获取当前日期和时间",
            "parameters": {
                "type": "object",
                "properties": {},
                "required": [],
            },
        },
    }
]
```

这个工具没有参数——用户问"几点了"，LLM 就知道该调 `get_current_time`。

有了工具定义，还需要实际能执行的函数。用个 dict 做名字到函数的映射：

```
def get_current_time() -> dict:
    from datetime import datetime
    now = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    return {"success": True, "data": now}


TOOL_MAP = {"get_current_time": get_current_time}
```

返回 `{"success": True, "data": ...}` 这种统一格式是好习惯。后面的章节会把它抽象成 `ToolResult`，但现在一个 dict 就够了。

2.4 最小 Agent 循环

现在把所有东西串起来：

```
import json
from llm_client import LLMClient
```

```python
class Agent:
    def __init__(self, api_key: str, base_url: str, model: str):
        self.llm = LLMClient(api_key, base_url, model)

    async def run(self, user_input: str) -> str:
        messages = [{"role": "user", "content": user_input}]

        # 第一轮：让 LLM 决定是否需要工具
        response = await self.llm.chat(messages, tools=TOOLS)
        choice = response["choices"][0]
        message = choice["message"]

        # 没有 tool_calls → LLM 直接回答了，返回
        if not message.get("tool_calls"):
            return message.get("content", "")

        # 有 tool_calls → 执行工具，结果加入消息
        messages.append(message)
        for tc in message["tool_calls"]:
            name = tc["function"]["name"]
            args = json.loads(tc["function"]["arguments"])
            func = TOOL_MAP.get(name)
            if func:
                result = func(**args)
            else:
                result = {"success": False, "error": f"未知工具: {name}"}
            messages.append({
                "role": "tool",
                "tool_call_id": tc["id"],
                "content": json.dumps(result, ensure_ascii=False),
            })

        # 第二轮：LLM 基于工具结果生成回复
        response2 = await self.llm.chat(messages)
        return response2["choices"][0]["message"].get("content", "")
```

这就是整个 Agent 的核心逻辑。不到 40 行，但它已经是一个真正的 Agent 了。

一步一步走：

1. **构建消息**：`[{"role": "user", "content": "几点了？"}]`
2. **第一次 LLM 调用（带 tools）**：LLM 看到 `get_current_time` 工具，判断用户想知道时间，返回 `tool_calls: [{"function": {"name": "get_current_time", "arguments": "{}"}}]`
3. **执行工具**：`TOOL_MAP["get_current_time"]()` → `{"success": True, "data": "2026-07-24 15:30:00"}`
4. **第二次 LLM 调用（不带 tools）**：LLM 把工具结果翻译成自然语言 → "现在是 2026 年 7 月 24 日 15:30。"

如果用户说"你好"，LLM 可能觉得不需要工具，第一步就直接返回文本了。这种情况下 `tool_calls` 为空，`run()` 直接返回 LLM 的文本回复，不走工具执行流程。

2.5 命令行入口

最后加一个简单的 REPL：

```
# main.py
import asyncio
import os
from agent import Agent


async def main():
    api_key = os.getenv("LLM_API_KEY", "sk-your-key")
    base_url = os.getenv("LLM_BASE_URL", "https://api.deepseek.com")
    model = os.getenv("LLM_MODEL", "deepseek-chat")

    agent = Agent(api_key, base_url, model)
    print("MiniAgent 已启动。输入问题，输入 /exit 退出。\n")

    while True:
        try:
            user_input = input("> ")
        except (EOFError, KeyboardInterrupt):
            print("\n再见！")
            break
        if user_input.strip() == "/exit":
            print("再见！")
            break
        if not user_input.strip():
            continue
        response = await agent.run(user_input)
        print(f"\n{response}\n")


if __name__ == "__main__":
    asyncio.run(main())
```

配置通过环境变量传入，不写配置文件。这一章的重点是 Agent 循环，不是配置系统。

运行它：

```
export LLM_API_KEY="sk-your-deepseek-key"
python main.py
```

试试这几个问题：

```
> 你好
你好！有什么我可以帮你的吗？

> 现在几点了？
现在是 2026 年 7 月 24 日 15:30:00。

> 今天星期几？
今天是星期五。
```

第二个问题触发了 `get_current_time` 工具调用，第三个问题 LLM 基于上下文（刚获取的时间）直接推理出了答案——没有再次调工具。这就是 Agent 的智能之处：它知道什么时候需要动手，什么时候已经够了。

2.6 为什么这 40 行已经比很多"框架"强

别小看这 40 行代码。它已经包含了 Agent 的三个核心要素：

1. **工具注册**：`TOOLS` 列表告诉 LLM "我能做什么"
2. **工具分发**：`TOOL_MAP` 把 LLM 的意图映射到真实执行
3. **多轮对话**：工具结果回传 → LLM 再处理 → 最终回复

很多"Agent 框架"做的事就是把这个循环包装在十几层抽象里。你现在看到的这 40 行，就是它们的内核。

当然，这个版本有很多局限：

- 只支持一轮工具调用（不能连续多轮调不同工具）
- 没有对话记忆（每次都是新对话，不记得上一句说了什么）
- 没有错误处理（API 挂了直接崩）
- 没有并行执行

这些正是后面章节要解决的问题。但核心骨架已经在了，后面的每一章只是在这个骨架上长肉。

2.7 Nimo 实战笔记

Nimo 的真实实现比这个教学版本多了不少东西。说几个最关键的差异：

LLM 调用的重试机制。Nimo 的 `LLMClient.chat()` 有 4 次尝试（1 次初始 + 3 次重试），每次间隔指数退避（1s、2s、4s）。只对 `RateLimitError`、`APITimeoutError`、`InternalServerError` 重试——这些是"临时性问题，等一下可能就好了"。`BadRequestError`（参数错误）和 `AuthenticationError`（密钥错误）不重试，因为重试一万次也不会好。

这个设计来自真实踩坑：DeepSeek API 在高峰期偶尔返回 503，不加重试的话用户体验很差。但重试不能无脑加——4 种临时异常是从 OpenAI SDK 的异常层次中手动挑出来的。详细讨论在第 15 章。

agent.run() 的复杂度。教学版的 `run()` 只有一次 LLM 调用 + 一次可能的工具执行。Nimo 的真实版本是 `for round in 1..max_tool_rounds` 的多轮循环，每轮可以并行执行多个工具，有循环检测（连续 3 次相同调用自动终止），有上下文超限自动恢复。这些我们会在第 4 章展开。

system prompt 的加载。教学版没写 system prompt。Nimo 从 `nimo/prompts/system.md` 读取行为准则，并追加"今天是 XXXX 年 X 月 X 日 星期 X"——因为 LLM 不知道当前日期。这个细节在测试时很重要：mock 的 system prompt 里日期是固定的，否则测试结果每次都不一样。

完整的代码在本章教学目录 `book/code/chapter-02/` 下。你可以直接跑起来，也可以对照 Nimo 源码看差异。

Nimo 源码对照

本章概念	Nimo 实现
LLM API 封装与重试	llm/client.py
Agent 入口与输入循环	main.py
最小可运行 Agent	agent.py:36-94（`__init__` 初始化部分）

下一章，我们让工具注册变得优雅——不用手动维护 `TOOLS` 和 `TOOL_MAP` 两个数据结构，一个装饰器搞定一切。

试读结束

以上为《从零构建 AI Agent》前两章内容。全书共 16 章 + 附录，涵盖工具注册、核心循环、记忆管理、执行引擎、Skill 插件系统、流式响应、安全加固等完整主题。

每章末尾的实战笔记是本书最有价值的部分——来自 Nimo 项目真实 git 提交历史的 bug、设计失误和凌晨调试的教训。

购买完整版：[Leanpub 链接]
教学代码：github.com/FuZhengCN/nimo-agent → book/code/