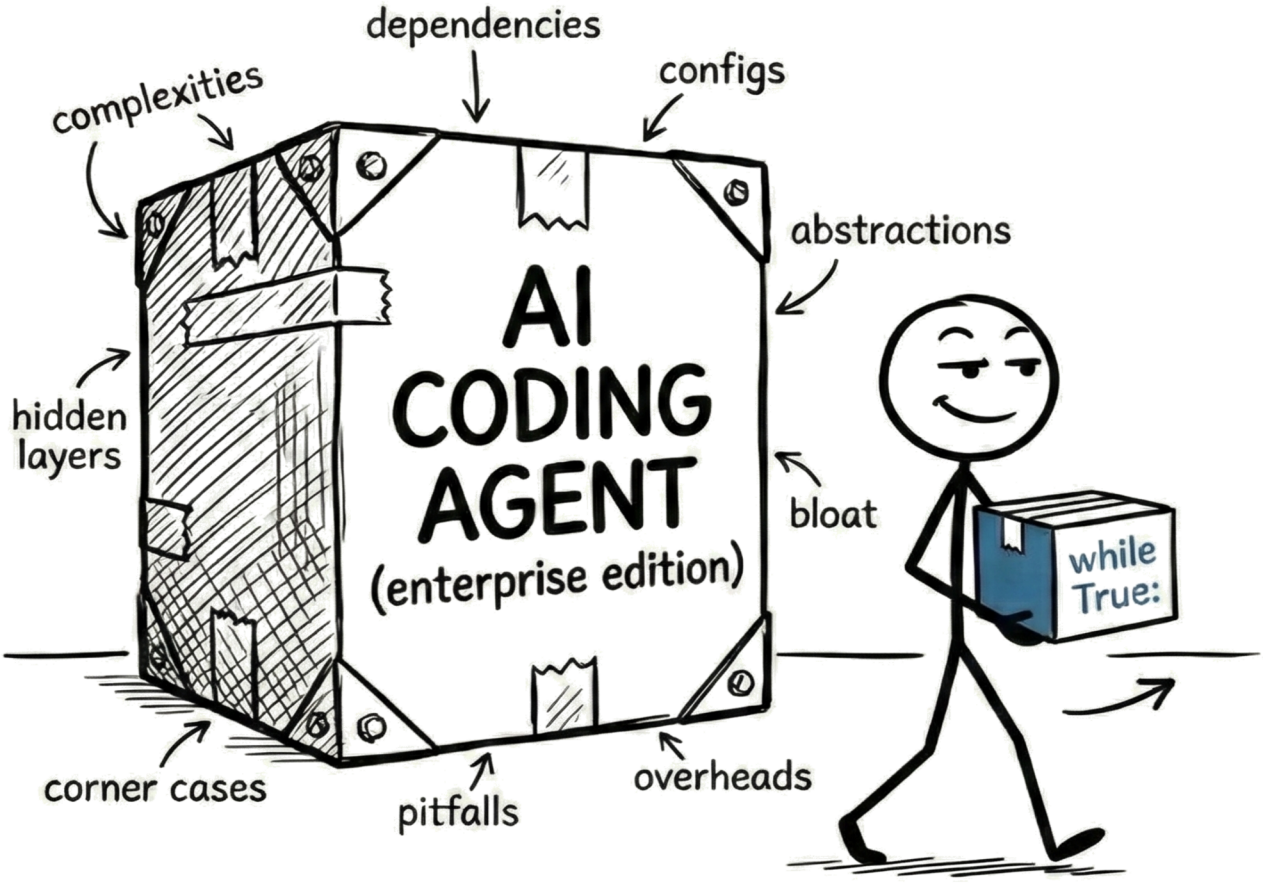
Build Your Own
CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



J. Owen

简体中文版

构建你自己的编程智能助手 (简体中文版)

零魔法：纯 Python 实现人工智能代理指南

J. Owen

这本书的网址是
https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

此版本发布于 2026-04-19

在 Twitter 上分享这本书!

请在 Twitter 上帮助作者 J. Owen 宣传!

对这本书建议的推文是:

Just built a coding agent from scratch with nothing but HTTP calls and a shell. Turns out "AI magic" is just a while loop and an API call.

对这本书建议的 hashtag 是 #buildyourowncodingagent.

若想知道其他人对这本书的看法，可以点击此链接搜索 Twitter 上的 hashtag:

#buildyourowncodingagent

谨以此书献给我的妻子、儿子、父母和祖父母——是你们教会我人生没有魔法，唯有努力。我所建造的一切，都因你们而存在。

Contents

前言

本书适合谁

你是一名对 AI 炒作持怀疑态度的软件工程师。

你尝试过各种框架——然后眼睁睁地看着自己的 LangChain 应用胡乱生成指令，把生产数据库给删了。你心想：“一定有更好的办法。”

确实有。本书面向那些想真正理解 AI 智能体运行原理的开发者。不是营销图表。不是什么 “推理引擎 “抽象层。而是实际的 HTTP 请求。实际的`while` 循环。

只要你能读懂 Python，并且有过构建 Web 应用或 CLI 工具的经历，你就已经具备了阅读本书所需的一切。

你将构建什么

Nanocode 是一个在终端中运行的编程智能体。读完本书后，它将能够：

- 读写代码库中的文件
- 执行 shell 命令
- 使用纯 Python 搜索代码
- 跨会话保留上下文
- 通过安全模式将规划与执行分离
- 在网络上搜索文档和答案

你将使用 `requests`、`python-dotenv` 和 `pytest` 从零开始构建它；shell 访问能力来自 Python 标准库中的 `subprocess` 模块。没有 LangChain，没有向量数据库，没有 “编排框架 “，只有你可以用 `print()` 调试的 Python 代码。

唯一的例外：第 11 章新增了 `ddgs` 用于网络搜索——这是本书唯一新增的一个轻量级外部依赖。

测试方法

本书采用 “测试并行 “的方式。对于大多数功能：

1. 我们先介绍概念——为什么需要这个功能
2. 先展示测试，让你清楚成功的标准是什么
3. 再编写代码，让测试通过
4. 最后用 pytest 验证

这种方式传授了测试驱动开发的思维方式，同时省去了书面呈现完整红-绿-重构仪式的繁琐。它还解决了一个关键问题：你无法通过真正调用大语言模型来测试由其驱动的应用。API 调用速度慢、成本高，而且结果具有不确定性。

从第 3 章开始，我们引入 `FakeBrain` 模式——一种能返回可预测响应的测试替身（test double）。这样你就可以在不发起任何 API 调用、不花费一分钱的情况下，运行完整的测试套件。

代码示例

本书遵循 “代码优先 “的方式。每章在前一章的基础上递进，所有代码示例均从可运行的文件中提取。

获取代码：

- **GitHub：**从 GitHub 克隆或下载。[1]
- **Leanpub：**完整源代码也包含在购买后可下载的资源包中。

代码按章节组织（`ch01/`、`ch03/`、`ch04/` 等），另有一个 `appendix/` 文件夹。大多数文件夹包含：

- `nanocode.py` — 该章节完整的、可运行的智能体代码
- `test_nanocode.py` — 无需 API 调用即可验证代码正确性的测试

有两个例外：`ch02/` 只包含一个独立的 `test_api.py` 脚本（仅使用一次后即弃用），`ch12/` 的智能体快照在功能上与 `ch11/` 相同，并附有收官项目成果。

你可以复制任意章节文件夹，从那里继续。运行 `pytest` 来验证你的代码是否符合预期行为。

[1] https://github.com/optimalone/build-your-own-coding-agent

本书使用的约定

全书中，你会看到三种标注框：

> **开发建议：**适用范围超越本项目的架构心得。这些是你可以在自己工作中复用的模式。

警告：安全或风险提示。请务必留意——忽视它们可能导致文件被删除或 API 密钥泄露。

旁注：深度探讨与延伸内容。有用的背景知识，但初次阅读时跳过也不影响理解主线。

代码讲解遵循统一的结构：先是背景（为什么需要这段代码），然后是代码本身，最后逐行解释重要部分。

是时候写代码了。

第一部分：大脑

第一部分将搭建一个智能体的骨架：一个通过原始 HTTP 与 Claude 通信的 `while` 循环。到第 4 章时，你将通过适配器模式支持多个 LLM 提供商——并理解 “AI 智能体 “只是一个循环、一个大脑，以及一个消息列表。

第一章：零魔法宣言

如果你在过去两年里尝试过构建 AI 应用，你很可能已经体会到了 “框架疲劳 “。

你安装了一个流行的库，导入了一个 `ReasoningEngine`，调用了 `.run()`。在 “Hello World“示例里，它像魔法一样运行。但当你试图做一些真正有意义的事情时——比如在不删除导入语句的情况下编辑 Python 文件中的某一特定行——它就出问题了。

而因为你用了框架，你也无从修复它。你只能在一层又一层的抽象类、工厂模式和 “链 “中层层翻找，试图找到那个导致幻觉的提示词。

我们不打算这样做。

这本书是对 “魔法 “的一次反叛。我们将采用 “零魔法 “方式：用纯 Python 构建一个生产级编程智能体，名为 **Nanocode**。不用 LangChain，不用 AutoGPT，不用 Pydantic。

为什么？因为自主智能体并不是什么魔法，它不过是一个 `while` 循环。

智能体到底是什么？

剥去风险投资的营销包装，“智能体 “不过就是一个**恒温器**。

恒温器读取温度（输入），与目标值比较（决策），然后打开加热器（动作）。接着等待，再重复。就这样。AI 智能体做的是同样的事，只不过用文本代替了温度。

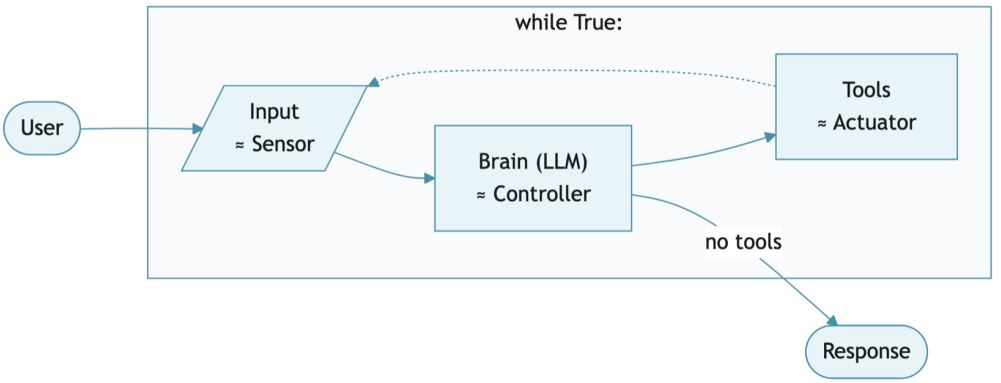


图 1. 智能体循环：用户输入流经一个 while 循环，输入（传感器）送入大脑/LLM（控制器），触发工具（执行器），再循环回输入，直到大脑输出响应。

更具体地说，一个智能体由四个部分组成。大脑是大语言模型（LLM）——一个无状态函数，你发送文本，它返回文本。它调用工具——比如 “读取文件 “和 “运行命令 “这样的函数——来与外部世界交互。这一切都运行在一个循环（`while True`）中，不断循环直到任务完成，而记忆——只是一个 Python 列表——则在此过程中不断积累对话历史。（程序结束时列表就消失了；我们将在第 6 章添加持久化存储。）

只要你会写 `while` 循环，你就能构建一个智能体。

通过从零开始构建，你将拥有框架用户所没有的东西：掌控力。当我们的智能体陷入死循环时，你能精确知道是哪一行代码造成的。当 API 账单过高时，你能清楚地看到 token 是在哪里泄漏的。

我们在构建什么

Nanocode 是一个在终端中运行的命令行工具。你像和同事交谈一样与它对话。它读取你的文件，运行你的命令，编辑你的代码。

到本书末尾，你将把它接入 Claude Sonnet 4.6（或 DeepSeek，或通过 Ollama 运行的本地模型）。你会给它一双手——读取文件、写入文件、运行 Shell 命令的工具——以及一双眼睛来搜索你的代码库。你还会为它构建一套安全防护机制，让它不会不小心执行 `rm -rf /`。

项目搭建

> **开发提示：** AI 项目最大的威胁是 “依赖腐烂 “。AI 库更新迭代极快，随时可能破坏现有功能。通过坚持只使用 `requests` 和 `subprocess`，这个智能体在 5 年后大概率仍然可以正常运行。

1. 初始化项目

```
mkdir nanocode
cd nanocode
git init
```

2. 创建虚拟环境

永远不要全局安装 AI 工具。它们会与系统包产生冲突。

```
# Mac/Linux
python3 -m venv venv
source venv/bin/activate

# Windows
python -m venv venv
venv\Scripts\activate
```

3. 安装依赖项

我们只需要三个库：

- `requests` — 用于与 LLM API 通信。
- `python-dotenv` — 用于从 `.env` 文件加载 API 密钥。
- `pytest` — 用于在不调用 API 的情况下测试我们的代码。

创建 `requirements.txt`：

```
requests
python-dotenv
pytest
```

安装：

```
pip install -r requirements.txt
```

4. 保护你的密钥

警告：如果你将 API 密钥推送到 GitHub，机器人会在几分钟内抓取并耗尽你的账户余额。

创建 `.gitignore`：

```
.env
__pycache__/
venv/
.DS_Store
.nanocode/
```

AgentStop 异常

在编写事件循环之前，我们需要一个干净的退出机制。使用异常比在代码各处散落 `break` 语句更简洁。

背景：异常不仅仅用于错误处理，它也是一种控制流机制。当用户输入 `/q` 时，我们抛出 `AgentStop`。主循环捕获该异常后，便会干净退出。

代码：

```python
# --- Exceptions ---

class AgentStop(Exception):
    """Raised when the agent should stop processing."""
    pass
```

这段代码放在 `nanocode.py` 的顶部。它是一个标记异常——没有任何逻辑，只是一个信号。

Agent 类

现在来看核心抽象：`Agent` 类。它将状态和逻辑集中在同一个地方，这使得测试变得非常方便。

背景：我们可以把所有逻辑都放在 `main()` 里。但那样的话，我们就必须模拟 `input()` 和 `print()` 才能对其进行测试。通过将逻辑提取到 `Agent.handle_input()` 中，我们可以直接对其进行测试。

代码如下：

```
10  class Agent:
11      """A coding agent that processes user input."""
12
13      def __init__(self):
14          pass
15
16      def handle_input(self, user_input):
17          """Handle user input. Returns output string, raises AgentStop to
            ↪ quit."""
18          if user_input.strip() == "/q":
19              raise AgentStop()
20
21          if not user_input.strip():
22              return ""
23
24          return f"You said: {user_input}\n(Agent not yet connected)"
```

操作说明：

- **第 13-14 行：**暂时留空的构造函数。我们将在后续章节中添加 `brain` 和 `tools`。
- **第 18-19 行：** `/q` 命令会抛出 `AgentStop`，而不是返回一个特殊值——由调用方决定如何处理退出。
- **第 24 行：**回显输入内容。这只是一个占位符——稍后我们会将其发送给 Brain。

通过测试定义成功标准

在编写主循环之前，我们需要编写测试。

创建 `test_nanocode.py`：

```python
import pytest
from nanocode import Agent, AgentStop


def test_handle_input_returns_string():
    """Verify handle_input returns a string for normal input."""
    agent = Agent()
    result = agent.handle_input("hello")
    assert isinstance(result, str)
    assert "hello" in result


def test_empty_input_returns_empty_string():
    """Verify empty/whitespace input returns empty string."""
    agent = Agent()
    assert agent.handle_input("") == ""
    assert agent.handle_input("   ") == ""
    assert agent.handle_input("\n") == ""


def test_quit_command_raises_agent_stop():
    """Verify /q raises AgentStop exception."""
    agent = Agent()
    with pytest.raises(AgentStop):
        agent.handle_input("/q")


def test_quit_command_with_whitespace():
    """Verify /q works with surrounding whitespace."""
    agent = Agent()
    with pytest.raises(AgentStop):
        agent.handle_input("  /q  ")
```

运行测试：

```
pytest test_nanocode.py -v
```

```
test_nanocode.py::test_handle_input_returns_string PASSED
test_nanocode.py::test_empty_input_returns_empty_string PASSED
test_nanocode.py::test_quit_command_raises_agent_stop PASSED
test_nanocode.py::test_quit_command_with_whitespace PASSED
```

全部通过。我们的智能体能正确处理基本情况。

旁注：为什么选择 pytest？它会自动发现以 test_ 开头的函数并运行它们。无需样板代码，无需类。测试代码本身就是纯 Python——毫无魔法。

主循环

现在来看将智能体与终端连接起来的轻量 I/O 封装层：

```
def main():
    agent = Agent()
    print("⚡ Nanocode v0.1 initialized.")
    print("Type '/q' to quit.")

    while True:
        try:
            user_input = input("\n❯ ")
            output = agent.handle_input(user_input)
            if output:
                print(output)

        except (AgentStop, KeyboardInterrupt):
            print("\nExiting...")
            break


if __name__ == "__main__":
    main()
```

详解：

- **第 30–32 行：**创建代理并打印启动消息。
- **第 36 行：** `input()` 阻塞并等待用户输入内容。
- **第 37–39 行：**调用 `handle_input()` 并打印相应输出。
- **第 41–43 行：**捕获 `AgentStop`（来自 `/q`）或 `KeyboardInterrupt`（来自 Ctrl+C），并跳出循环。

旁注： Python 的 `input()` 每次读取一行。本书中的所有提示均为单行。这样可以保持代码简洁——生产环境中的代理会使用更丰富的输入方式，例如 readline 或完整的 TUI。

注意这里的关注点分离：`Agent.handle_input()` 包含所有逻辑，而 `main()` 只是 I/O 粘合层。这使得代理无需模拟 stdin/stdout 即可进行测试。

运行程序

```
python nanocode.py
```

你应该会看到：

```
⚡ Nanocode v0.1 initialized.
Type '/q' to quit.

 hello
You said: hello
(Agent not yet connected)

 /q

Exiting...
```

这就是底盘。接下来是引擎。

小结

这就是我们的底盘：一个 `Agent` 类、一个 `handle_input()` 方法、一个 `while True` 循环。它目前还做不了任何有用的事——但我们后续构建的所有内容都会接入这个骨架。测试确保我们在推进过程中，不会破坏已经正常运行的部分。

第二章：原始请求

大多数教程都会让你执行 `pip install anthropic`。我们不打算这么做。

SDK 掩盖了真相。它们增加了一层又一层的抽象，让 “Hello World“变得轻而易举，却让调试 “Error 400“成为噩梦。学会了 SDK，你只是学会了 SDK 本身。学会了原始 HTTP 请求，你才真正掌握了每一个 SDK 背后的底层协议。

我们将只使用 `requests` 库，直接向 Claude 发送消息。Claude 是 Anthropic 的旗舰大语言模型，在编程任务方面是能力最强的模型之一。

获取 API Key

要与 Claude 通信，你需要一个 API Key。这是一串很长的字符，作用类似于密码，与你的计费账户绑定。

1. 前往 Anthropic 控制台。[1]
2. 注册并添加支付方式（最低充值 $5）。
3. 创建一个新的 API Key，并将其命名为 `nanocode`。
4. 复制该 Key（它以 `sk-ant-...` 开头）。

警告：请像对待密码一样保管好这个 Key。任何持有它的人都可以花掉你的钱。

保险库（.env）

我们需要一个安全的地方来存储这个 Key。切勿将 Key 直接写入代码。

在你的项目根目录下创建一个名为 `.env` 的文件：

[1] https://console.anthropic.com/settings/keys

```
touch .env
```

打开它并粘贴你的密钥：

```
ANTHROPIC_API_KEY=sk-ant-api03-...
```

我们在第 1 章安装 `python-dotenv` 正是为了这个目的——它会读取 `.env` 文件并将其中的值加载到 `os.environ` 中。

请求的结构

要与 LLM 通信，我们需要向以下地址发送一个 HTTP POST 请求：

```
https://api.anthropic.com/v1/messages
```

这个请求需要三个要素：请求头中的身份验证信息（你的 API 密钥）、请求体中的配置参数（使用哪个模型、多少个 token），以及消息本身。

请求头

Anthropic 要求提供三个请求头：

请求头	值	用途
`x-api-key`	你的私有密钥	身份验证
`anthropic-version`	`2023-06-01`	API 版本
`content-type`	`application/json`	格式

载荷

“Messages API” 期望接收一个由消息字典组成的列表：

```
"messages": [
    {"role": "user", "content": "Hello, world!"}
]
```

每条消息都有一个 role（值为 "user" 或 "assistant"）和 content（文本）。

代码

创建一个名为 test_api.py 的文件。这是一个 “冒烟测试 “，用来验证我们的连接是否正常。之后我们会将其删除。

背景说明：我们编写的是线性的、过程式代码。没有函数，没有类。我们想直接看到最原始的底层实现。

代码：

```
import os
import requests
import json
from dotenv import load_dotenv

# 1. Load the vault
load_dotenv()
api_key = os.getenv("ANTHROPIC_API_KEY")

# Basic check so we don't crash with a confusing "NoneType" error later
if not api_key:
    print("Error: ANTHROPIC_API_KEY not found in .env")
    exit(1)

# 2. Define the target
url = "https://api.anthropic.com/v1/messages"

# 3. Authenticate
headers = {
    "x-api-key": api_key,
    "anthropic-version": "2023-06-01",
    "content-type": "application/json"
```

```
}

# 4. Construct the payload
payload = {
    "model": "claude-sonnet-4-6",
    "max_tokens": 4096,
    "messages": [
        {"role": "user", "content": "Hello, are you ready to code?"}
    ]
}

# 5. Fire! (No safety net)
print("📡 Sending request to Claude...")
response = requests.post(url, headers=headers, json=payload, timeout=120)

# 6. Inspect the raw result
print(f"Status: {response.status_code}")

if response.status_code == 200:
    # Success: Print the beautiful JSON
    print("Response:")
    print(json.dumps(response.json(), indent=2))
else:
    # Failure: Print the ugly raw text so we can debug
    print("Error:", response.text)
```

代码详解：

- **第 7 行：** `load_dotenv()` 会找到 `.env` 文件，并将其中的变量加载到 `os.environ` 中。
- **第 8 行：** 我们在这里获取 API 密钥。切记不要将其硬编码到代码里。
- **第 11-13 行：** 基本的有效性检查。如果没有这一步，缺失的密钥会在 headers 字典中引发令人困惑的 `NoneType` 错误。
- **第 21 行：** `anthropic-version` 请求头是必填项。若省略，API 将拒绝你的请求。
- **第 27 行：** `claude-sonnet-4-6` 指定了我们要使用的模型。
- **第 28 行：** `max_tokens` 是必填项。它用于限制响应长度，防止费用失控。
- **第 36 行：** 我们以 2 分钟的超时时间发送请求，没有使用 try/except——如果网络断开，就让 Python 崩溃。你需要看到它在哪里出错了。

- **第 41-47 行：**检查状态码。200 表示成功（以美化格式输出 JSON）。其他任何状态码都意味着我们打印原始错误文本以便调试。

运行程序

```
python test_api.py
```

如果一切正常，你应该会看到：

```
Status: 200
Response:
{
  "id": "msg_01...",
  "type": "message",
  "role": "assistant",
  "content": [
    {
      "type": "text",
      "text": "Hello! Yes, I'm ready to code..."
    }
  ],
  "stop_reason": "end_turn",
  "usage": {
    "input_tokens": 15,
    "output_tokens": 81
  }
}
```

故障排查

错误	原因	解决方法
401 Unauthorized	API 密钥错误	检查 `.env` 是否已加载。打印 `os.environ.get("ANTHROPIC_API_KEY")` 进行验证。
400 Bad Request	JSON 格式错误	是否忘记了 `max_tokens`？`messages` 是列表吗？
429 Rate Limit	请求过多或额度不足	稍等片刻，或向账户充值。

清理工作

我们已经证明可以和这个 “大脑 “对话了。删除 `test_api.py`——我们不再需要它了。

旁注：这个 `test_api.py` 是一次性代码——只用一次的冒烟测试。真正的自动化测试（使用 `FakeBrain` 和 pytest）会在第 3 章介绍。验证 API 连接正常后，你应该删除这个文件。

开发小贴士：每次 API 调用都要花钱。输入令牌（你发送的内容）很便宜，而输出令牌（模型生成的内容）则贵约 5 倍。保持提示词简洁。

旁注：要监控你的消费情况，可以查看 Anthropic Console 中的 “使用情况 “选项卡。截至 2026 年初，使用 Claude Sonnet 进行一次包含 20-30 轮对话的典型编程会话，费用约为 $0.10-$0.50。响应 JSON 底部的 `usage` 字段会显示精确的令牌数量——你可以记录这些数据，以便通过编程方式追踪费用。

小结

这就是一次原始的 API 调用：请求头、JSON 有效载荷、响应解析。你和底层网络之间没有任何抽象层。当出现问题时（这是迟早的事），你能准确知道是哪一层出了故障——因为本来就只有一层。

有一个问题：Claude 完全没有记忆。每次请求都是一片空白。我们将通过在每一轮对话中重放完整的对话历史，来模拟出记忆的效果。

第三章：无限循环

我们遇到了一个问题。

想象一下，把第二章的脚本运行两次。你说 “我叫 Alice。“Claude 说了声你好。再次运行，问 “我叫什么名字？“Claude 说 “我不知道。”

这是因为 LLM 是无状态的。它们完全没有记忆。每一次请求，对它来说都是第一次见面。

要构建一个智能体，我们需要通过创建人工记忆来解决这个问题。

记忆的幻觉

LLM 中的 “记忆 “不是硬盘，而是一份日志文件。

当你和 ChatGPT 聊天时，它并不真正 “记得 “你五分钟前说过什么。在幕后，代码会在每条新消息发出时，把**完整的对话历史**一并发送给模型。

图 2. 上下文累积：第一轮只向 API 发送 “User: Hi“。第二轮则将完整历史——“User: Hi”、“Assistant: Hello”、“User: How are you?”——全部发送给 API。

模型每次都能看到完整的对话记录。这就是其中的诀窍。

让我们手动实现这个上下文循环。但在此之前，我们需要让代码具备可测试性。

测试的难题

有一个残酷的现实：你不能通过真正调用 LLM 来测试一个由 LLM 驱动的应用程序。

API 调用很慢（每次需要 2 到 10 秒），很贵（每次调用都需要花费真实的金钱），而且具有不确定性（每次可能得到不同的响应）。想象一下，运行一套测试套件要花 5 美元、耗时 20 分钟——你根本不会想去运行它。

解决方案是依赖注入。我们不把 API 调用硬编码在智能体内部，而是传入一个 “大脑 “对象。在生产环境中，大脑是 Claude；在测试中，大脑是一个能返回可预测响应的伪对象。

我们现在就建立这个模式，在编写更多生产代码之前。

响应类型

在构建大脑之前，我们需要定义它返回的内容。Claude 的 API 会返回包含多个内容块的复杂 JSON。我们需要简单的 Python 对象来处理这些数据。

背景： Claude 可以在单个响应中返回文本、工具调用，或两者兼而有之。我们需要普通的数据对象来表示这些可能性。（我们有意跳过了 `@dataclass`——这些类已经足够简单，装饰器虽能省去几行代码，却会掩盖 `__init__` 实际的执行内容。）

代码如下：

```
17  class ToolCall:
18      """A tool invocation request from the brain."""
19
20      def __init__(self, id, name, args):
21          self.id = id
22          self.name = name
23          self.args = args  # dict
```

`ToolCall` 表示大脑请求我们执行某个工具。`id` 是用于追踪的唯一标识符（当我们向 Claude 汇报结果时，Claude 需要用到它）。`name` 是要运行的工具名称。`args` 是一个参数字典。

我们暂时不会用到 `ToolCall`——大脑目前还无法调用工具——但我们现在就定义它，因为它是 `Thought` 响应类型的一部分。当我们添加工具后，Claude 在想要读取文件或执行命令时，就会返回这些。

```
class Thought:
    """Standardized response from any Brain."""

    def __init__(self, text=None, tool_calls=None, thinking=None):
        self.text = text  # str or None
        self.tool_calls = tool_calls or []  # list of ToolCall
        self.thinking = thinking  # str or None
```

Thought 是大脑思考后返回的结果。它可能包含文本、工具调用、两者兼有，或者什么都没有。thinking 字段记录了模型的推理摘要——当我们在下面构建 Claude 类时，会看到它的来源。这一抽象设计将让我们在不修改任何其他代码的情况下，把 Claude 替换为 DeepSeek。

FakeBrain 模式

现在我们可以构建一个用于测试的假大脑。

背景：我们需要一个能返回可预测的响应、追踪被调用的次数，并记录收到的对话内容的大脑。

代码：

```
class FakeBrain:
    """Fake brain for testing - returns predictable responses."""

    def __init__(self, responses=None):
        self.responses = responses or [Thought(text="Fake response")]
        self.call_count = 0
        self.last_conversation = None

    def think(self, conversation):
        self.last_conversation = list(conversation)  # Store a copy
        if self.call_count < len(self.responses):
            response = self.responses[self.call_count]
            self.call_count += 1
            return response
        return Thought(text="No more responses")
```

这段代码放在 `test_nanocode.py` 中，而不是生产代码里。注意，`FakeBrain` 与我们真实的 `brain` 具有相同的接口——一个接受对话并返回 `Thought` 的 `think()` 方法。

旁注：这种模式——用可预测的伪对象替换真实依赖以便测试——称为测试替身。Martin Fowler 的文章 “Mocks Aren’t Stubs”[1] 解释了各种变体（fakes、stubs、mocks、spies）。对于 LLM 测试来说，一个带有预置响应的简单伪对象通常就足够了。

定义成功

在编写生产代码之前，让我们先定义何为成功。这些测试将指导我们的实现。

测试 1：`brain` 返回响应

```
def test_handle_input_returns_brain_response():
    """Verify handle_input returns the brain's response text."""
    brain = FakeBrain(responses=[Thought(text="Hello from brain!")])
    agent = Agent(brain=brain)
    result = agent.handle_input("hi")
    assert result == "Hello from brain!"
```

注意，我们将 `brain=brain` 传入 Agent。这就是依赖注入的实际体现。

测试 2：对话内容逐步累积

```
def test_conversation_accumulates():
    """Verify conversation list grows with each interaction."""
    brain = FakeBrain(responses=[
        Thought(text="Response 1"),
        Thought(text="Response 2")
    ])
    agent = Agent(brain=brain)

    agent.handle_input("First message")
    assert len(agent.conversation) == 2  # user + assistant
```

[1] https://martinfowler.com/articles/mocksArentStubs.html

```
11
12      agent.handle_input("Second message")
13      assert len(agent.conversation) == 4  # 2 users + 2 assistants
```

每次交互后，对话中应同时包含用户消息和助手回复。

测试 3：正确的消息结构

```
1   def test_conversation_contains_correct_roles():
2       """Verify conversation has correct role alternation."""
3       brain = FakeBrain(responses=[Thought(text="AI response")])
4       agent = Agent(brain=brain)
5
6       agent.handle_input("User message")
7
8       assert agent.conversation[0]["role"] == "user"
9       assert agent.conversation[0]["content"] == "User message"
10      assert agent.conversation[1]["role"] == "assistant"
11      assert agent.conversation[1]["content"] == "AI response"
```

消息必须采用 Claude 所要求的确切格式：`{"role": "user", "content": "..."}`。

测试 4：Brain 接收对话

```
1   def test_brain_receives_conversation():
2       """Verify brain.think is called with the conversation list."""
3       brain = FakeBrain()
4       agent = Agent(brain=brain)
5
6       agent.handle_input("Test message")
7
8       assert brain.last_conversation is not None
9       assert len(brain.last_conversation) == 1
10      assert brain.last_conversation[0]["content"] == "Test message"
```

大脑必须接收完整的对话内容，而不仅仅是当前消息。

现在运行这些测试——它们应该全部失败：

```
pytest test_nanocode.py -v
```

```
FAILED test_nanocode.py::test_handle_input_returns_brain_response
FAILED test_nanocode.py::test_conversation_accumulates
...
```

很好。现在让我们让它们通过。

Claude 类

现在来看真正的核心部分。

背景说明：我们需要一个封装 Claude API 的类。它应负责处理身份验证、发送对话历史，并将响应解析为一个 `Thought`。我们还启用了扩展思考——这是一项让模型在回答之前先进行内部草稿式思考的功能。可以把它想象成模型在开口说话之前，先在一张草稿纸上自言自语。这会消耗额外的 tokens，但质量提升十分显著，尤其是在第 5 章添加工具之后——届时模型需要推理应该调用哪个工具以及调用原因。

代码：

```
class Claude:
    """Claude API - the brain of our agent."""

    def __init__(self):
        self.api_key = os.getenv("ANTHROPIC_API_KEY")
        if not self.api_key:
            raise ValueError("ANTHROPIC_API_KEY not found in .env")
        self.model = "claude-sonnet-4-6"
        self.url = "https://api.anthropic.com/v1/messages"

    def think(self, conversation):
        headers = {
            "x-api-key": self.api_key,
            "anthropic-version": "2023-06-01",
            "content-type": "application/json"
        }
```

```
53          payload = {
54              "model": self.model,
55              "max_tokens": 16000,
56              "thinking": {
57                  "type": "enabled",
58                  "budget_tokens": 10000
59              },
60              "messages": conversation
61          }
62
63          response = requests.post(self.url, headers=headers, json=payload,
            ↪  timeout=120)
64          response.raise_for_status()
65          return self._parse_response(response.json()["content"])
```

代码解析：

- **第 41-43 行：** 加载 API 密钥，如果缺失则快速失败。
- **第 44-45 行：** 存储配置。稍后我们会使模型可配置。
- **第 47 行：** `think()` 方法是大脑的接口——与 `FakeBrain` 相同。
- **第 55-59 行：** 我们启用了扩展思考——模型在响应之前会生成推理摘要，从而提升处理复杂任务时的质量，但代价是消耗更多 token。`budget_tokens` 限制了模型可用于推理的 token 数量（此处为 10,000）——这些 token 和输出 token 一样计入费用。在我们的设置中，`max_tokens` 涵盖了包括思考和响应在内的全部输出，因此 Anthropic 要求它必须超过 `budget_tokens`。当思考 token 上限为 10,000、`max_tokens` 为 16,000 时，响应本身最多可使用 6,000 个 token。
- **第 60 行：** 请求体中包含 `"messages": conversation`——即完整的历史记录，而不仅仅是当前消息。这就是上下文循环。
- **第 65 行：** 将 Claude 的复杂响应格式解析为我们简单的 `Thought`。

接下来是响应解析器：

```
    def _parse_response(self, content):
        """Convert Claude's response format to Thought."""
        text_parts = []
        tool_calls = []
        thinking = None

        for block in content:
            if block["type"] == "thinking":
                thinking = block["thinking"]
            elif block["type"] == "text":
                text_parts.append(block["text"])
            elif block["type"] == "tool_use":
                tool_calls.append(ToolCall(
                    id=block["id"],
                    name=block["name"],
                    args=block["input"]
                ))

        return Thought(
            text="\n".join(text_parts) if text_parts else None,
            tool_calls=tool_calls,
            thinking=thinking
        )
```

Claude 的 API 会返回一个 “内容块 “列表。每个块都有一个 type 字段，其值为 "thinking"、"text" 或 "tool_use"。thinking 块最先到达，其中包含模型推理过程的摘要——我们将其存储在 Thought 上，供调用者展示。text 块会成为响应内容，而 tool_use 块则会转换为 ToolCall 对象。解析器本身不打印任何内容，它只是将原始 JSON 转换为整洁的 Thought。

Agent 类（更新版）

现在，我们将第 1 章中的 Agent 进行更新，使其能够接受一个大脑并维护对话历史。

代码：

```python
class Agent:
    """A coding agent with conversation memory."""

    def __init__(self, brain):
        self.brain = brain
        self.conversation = []

    def handle_input(self, user_input):
        """Handle user input. Returns output string, raises AgentStop to
        ↪ quit."""
        if user_input.strip() == "/q":
            raise AgentStop()

        if not user_input.strip():
            return ""

        self.conversation.append({"role": "user", "content": user_input})

        try:
            thought = self.brain.think(self.conversation)
            if thought.thinking:
                lines = thought.thinking.strip().split("\n")[:5]
                for i, line in enumerate(lines):
                    prefix = "  💭 " if i == 0 else "     "
                    print(f"\033[2m{prefix}{line}\033[0m")
            text = thought.text or ""
            self.conversation.append({"role": "assistant", "content": text})
            return text
        except Exception as e:
            self.conversation.pop()  # Remove failed user message
            return f"Error: {e}"
```

代码解析：

- **第 97-99 行：**通过依赖注入接收一个 brain。初始化一个空的对话列表。
- **第 109 行：**在调用 brain 之前，将用户消息追加到历史记录中。
- **第 112-120 行：**调用 brain，以暗淡文本显示最多五行思考内容（`\033[2m` 是用于暗淡显示的 ANSI 转义码，`\033[0m` 用于重置），提取响应内容，并将其追加到历史记录中。

- **第 121-123 行：**如果 API 调用失败，则移除刚刚添加的用户消息。这样可以确保对话保持在有效状态。

请注意第 109 行：我们在调用 brain 之前就添加了用户消息。brain 需要看到包含当前消息在内的完整对话内容。

> **开发小贴士：**当出现问题时，你的第一个调试工具就是 `print(self.conversation)`。它能精确地显示 brain 所看到的内容。当你检查原始列表时，格式错误的消息、缺失的角色或被截断的内容都会一目了然。

主循环（已更新）

主循环现在只是一个精简的 I/O 封装层：

```
128 def main():
129     brain = Claude()
130     agent = Agent(brain)
131     print("⚡ Nanocode v0.2 (Conversation Memory)")
132     print("Type '/q' to quit.\n")
133 
134     while True:
135         try:
136             user_input = input("❯ ")
137             output = agent.handle_input(user_input)
138             if output:
139                 print(f"\n{output}\n")
140 
141         except (AgentStop, KeyboardInterrupt):
142             print("\nExiting...")
143             break
144 
145 
146 if __name__ == "__main__":
147     main()
```

所有逻辑都在 Agent 类中。循环只负责读取输入、调用 handle_input()，并打印结果。这种分离使得 agent 具有可测试性——我们可以直接测试 Agent.handle_input()，而无需模拟 input() 或 print()。

验证测试通过

再次运行测试：

```
pytest test_nanocode.py -v
```

```
test_nanocode.py::test_handle_input_returns_brain_response PASSED
test_nanocode.py::test_conversation_accumulates PASSED
test_nanocode.py::test_conversation_contains_correct_roles PASSED
test_nanocode.py::test_brain_receives_conversation PASSED
```

全绿。这些测试在不发起任何 API 调用的情况下验证了我们的实现。

测试记忆

现在用真实的大脑来测试：

```
python nanocode.py
```

试试这段对话：

```
 I am building a Python agent.
  💭 The user is telling me about their project. They want to build
     a Python agent. I should respond helpfully and ask what kind
     of agent they're building.

That sounds exciting! What kind of agent are you building?

 What language am I using?
  💭 The user previously said they are building a Python agent.
     The answer is Python.

You are using Python.
```

对话列表正在发挥它的作用。

上下文窗口问题

你可能会想：“我能让它一直运行下去吗？”

不能。

每次循环迭代，`messages` 列表都会增长：

轮次	近似 Token 数
1	50
10	5,000
100	50,000

最终，你会触及上下文限制——Claude Sonnet 为 200k token，DeepSeek 为 128k，某些本地模型甚至低至 4k。一旦超出，API 就会返回 `400 Bad Request`。我们在第 121 行的错误处理会捕获这个错误并报告，所以智能体不会悄无声息地崩溃。但对话实际上已经卡死了——由于历史记录依然过长，后续每条消息都会失败。

目前，重启智能体可以清空历史记录，让你重新回到正轨。我们将在第 9 章构建反馈循环时，加入真正的上下文压缩机制——即追踪 API 响应中的 token 用量，并在消息溢出之前自动对旧消息进行摘要。那才是对话真正容易爆掉的地方，也是这个修复方案真正派上用场的时候。

> **开发提示：**网络调用需要 2 到 10 秒。第 113–117 行的思考显示功能能让用户立刻获得视觉反馈，知道智能体正在工作中。没有它，用户只会盯着一个毫无动静的提示符，然后伸手去按 Ctrl+C。

本章小结

Claude 现在 “记得 “了——或者说，我们让它误以为自己记得。对话列表随着每一轮交互不断增长，而 `FakeBrain` 让我们无需花费一分钱就能测试整个流程。

这两种模式将贯穿本书其余部分。我们构建的每一个大脑（Claude、DeepSeek、Ollama）都将实现相同的 `think()` 接口，而 `FakeBrain` 将对它们全部进行测试。

还有一个遗留问题：我们的代码是硬编码到 Anthropic API 的。如果想添加 DeepSeek 或本地模型，就不得不重复大量代码。

第四章：通用适配器

适配器模式

HTTP 弹性

Brain 接口

FakeBrain（已更新）

Claude 大脑（重构版）

DeepSeek 大脑

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

BRAINS 注册表

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

Agent 类（更新版）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

多大脑支持的测试

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

主循环（已更新）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

配置 DeepSeek

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

试一试

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

“我们不过是把代码挪了挪”

小结

第二部分：双手

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

第五章：工具协议

工具究竟是如何运作的

定义工具接口

ReadFile 工具

WriteFile 工具

工具辅助函数

更新 Thought 类

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

更新 Claude 类

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

带工具的 Agent 类

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

主循环

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

测试一下

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

总结

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

第六章：便签本（记忆）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

“零魔法 “记忆

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

Memory 类

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

ToolContext 类

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

SaveMemory 工具

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

更新 Claude 类

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

编写系统提示

更新 Agent 类

主循环（已更新）

测试持久性

总结

第七章：安全装置（计划模式）

核心概念

测试先行

WritePlan 工具

一份列表，两种视图

告诉大脑当前处于哪种模式

Agent 类（已更新）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

主循环（更新版）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

测试框架

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

“计划 “的心理学

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

小结

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

第 8 章：上下文管道（映射与搜索）

ListFiles 工具

SearchCodebase 工具

更新工具列表

“放大查看 “测试

等等，这是 RAG 吗？

小结

第九章：现实检验（运行代码）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

反馈循环

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

测试先行

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

RunCommand 工具

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

交互式陷阱

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

自我修复演示

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

TDD 工作流程

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

外科手术式编辑

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

闭环

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

加固循环

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

上下文压缩

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

安全注意事项

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

总结

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

第三部分：前沿

第十章：离线运行（本地模型）

权衡取舍

安装 Ollama

Ollama 大脑类

使用 Ollama 运行

“无限循环 “实验

实际差异

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

混合工作流

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

模型选择

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

Ollama 故障排查

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

本章小结

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

第 11 章：扩展功能（网络搜索）

步骤 1：元提示词

第二步：手术

第三步：参考实现

第 4 步：测试

自我修改

小结

第 12 章：压轴章节（构建一个游戏）

Nanocode 真的能构建出东西吗？

“零代码挑战”：用 Python 和 Pygame 构建一个经典的贪吃蛇游戏。规则只有一条——你不能编写哪怕一行 Python 代码，只能用英语与智能体对话。

旁注：本演示涉及大量 API 调用。如果触及速率限制（HTTP 429），智能体会自动重试。对于较长的会话，可以考虑通过 Ollama 使用本地模型，从而完全避免此类限制。

第一步：准备工作

在项目根目录（即包含 `nanocode.py` 和 `.env` 的目录）下，为游戏创建一个工作目录：

旁注：我们在子目录中进行操作，这样智能体的 `list_files` 只会看到游戏相关文件，而不会看到其自身的源代码，同时也能以全新的记忆状态开始工作。

```
mkdir -p snake_game
cp nanocode.py snake_game/
cp .env snake_game/
cd snake_game
```

如果你使用的是本书的代码仓库，请改为从 `ch11` 复制：

```
mkdir -p snake_game
cp resources/code/ch11/nanocode.py snake_game/
cp .env snake_game/
cd snake_game
```

安装 Pygame：

```
pip install pygame
```

旁注：上述 Shell 命令适用于 macOS/Linux。在 Windows 上，请改用 `mkdir snake_game`、`copy nanocode.py snake_game\` 和 `copy .env snake_game\`。如果在 macOS 上执行 `pip install pygame` 失败，可能需要先运行 `brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf`。在 Linux 上，请安装 SDL 开发包：`sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev`。在 Windows 上，`pip install pygame` 已捆绑所有依赖。

第二步：架构师（计划模式）

启动智能体。我们从计划模式开始，因为我们希望在动工之前先画好蓝图。

```
python nanocode.py
```

提示词：

```
Build a classic Snake game using Pygame. Include a score counter and Game
↪  Over screen with a restart option. Put ALL code in ONE file: snake.py.
↪  Write the plan in PLAN.md.
```

智能体将使用 `write_plan` 来创建 `PLAN.md`。请阅读该文件。它应该概述 Snake 类、Food 类以及游戏循环——全部放在一个文件中。

如果计划看起来合理，请继续进行第 3 步。

第 3 步：构建者（执行模式）

切换到执行模式：

```
/mode act
```

提示词：

```
Implement the plan in snake.py. All code in one file.
```

注意终端：

```
  Writing snake.py
```

智能体正在根据它存储在 PLAN.md 中的上下文生成代码。

第四步：现实检验

在运行游戏之前，先调高超时时间。默认的 30 秒根本不够用来实际玩一局游戏——Pygame 的主循环会一直阻塞，直到你关闭窗口。在启动前设置好环境变量：

```
export NANOCODE_TIMEOUT=300
```

提示词：

```
Run the game with: python snake.py
```

代理执行 run_command。一个窗口弹出。你玩贪吃蛇游戏。

> **开发提示：**游戏窗口会阻塞代理。当你运行 python snake.py 时，代理会等到你关闭游戏窗口后才继续执行。这是正常现象——Pygame 的主循环会占用终端。

如果程序崩溃：LLM 是非确定性的。你的代理可能在第一次尝试时就出现 bug。如果游戏崩溃并出现类似 AttributeError: 'Snake' object has no attribute 'draw' 的错误，不要自己动手修复。让代理查看 stderr 输出。

提示词：

```
The game crashed. Read the error and fix it.
```

智能体将读取 traceback，使用 `read_file` 找到 bug，使用 `edit_file` 打上补丁，然后再次运行。

第五步：转向（需求蔓延）

游戏可以运行了，但很难看。蛇只是一堆绿色方块。让我们来压力测试一下智能体的重构能力。

提示词：

```
The game looks boring. Make the snake change color as it eats food, increase
↪ speed every 5 points, and search the web for 'cool retro game color
↪ palettes' to apply.
```

智能体应该：

1. 使用 `search_web` 查找配色方案
2. 使用 `read_file` 了解当前的渲染逻辑
3. 使用 `edit_file` 注入新功能
4. 运行游戏进行验证

会出什么问题

你的结果会和我的不同——大语言模型是非确定性的。但以下是通常会发生的情况，以及需要注意的地方。

第一次运行时的常见失败：

- `ModuleNotFoundError: No module named 'pygame'` —— 智能体忘记了需要先安装它，或者在不同的环境中运行了脚本。告诉它先运行 `pip install pygame`。
- 智能体定义了某个方法，但在调用处拼写错误，导致 `AttributeError`。一旦看到错误追踪信息，智能体会很快修复这类问题。
- 碰撞检测中的差一错误。蛇穿墙而过，或者提前一个像素就死掉了。这类问题需要 2-3 次 “编辑-运行-修复 “的迭代。

典型的会话流程：

在我的测试中，智能体通常在 2-4 次迭代内就能得到一个可运行（但不太好看）的游戏。第一次编写的代码会崩溃，第二或第三次修复后游戏就能跑起来了。功能扩展步骤（修改颜色、速度递增）又会增加 3-5 次迭代，因为智能体需要读取自己的代码、进行精细修改，并验证每一处改动。

到最后，对话已经深入到 15-20 轮。如果你使用的是 Claude，注意压缩触发的时机——大约在第 12-15 轮，当 token 数量接近 75% 阈值时，你会看到 "(Compacting conversation...)"的提示。压缩之后，智能体会丢失一些早期轮次的细节，但仍会继续工作。这正是第 9 章那套系统在证明其价值。

智能体容易卡壳的地方：

Pygame 的坐标系和事件循环比较棘手。智能体有时会写出渲染正常但键盘输入处理不当的代码，或者绘制蛇的顺序有误，导致蛇头出现在身体后面。这类 bug 人类一眼就能发现，但智能体看不到——它没有视觉反馈，只有 stdout 和 stderr。如果游戏运行时没有报错但显示有问题，你需要描述视觉上的 bug，比如："蛇渲染反了——蛇头应该在前面。"

关键不在于第一次就做到完美，而在于智能体能够收敛——编写、运行、读取错误、修复、再重复——贯穿十一章所构建的每一件工具都在此派上用场。

在你交付最终的 `snake.py` 之前，先读一遍。智能体写的代码能跑，但人类仍然应该审查那些测试循环无法发现的问题：硬编码的魔法数字、遗漏的边界情况（如果窗口被调整大小怎么办?），以及代码结构是否是你愿意长期维护的样子。智能体是快速的起草者，而不是最终的审查者。

结语

计划、实现、崩溃、调试、修复、再运行——每一章的价值都在此体现。

那么，接下来该去哪里?

后记

整个项目大约 750 行 Python 代码，没有任何框架。`nanocode.py` 是你的了，想怎么用就怎么用：

- Git 集成，在测试通过后自动提交
- 基于截图的调试，用于前端工作（Claude 能读取图片）

- 通过 Whisper 实现语音输入，让你可以说话代替打字
- 用 MCP 连接团队已在使用的外部服务
- 派生对话、并行处理子任务的子智能体

Claude Code、Cursor、Copilot 这类生产级智能体能做的比这更多：流式响应（参见附录 A）、并行工具执行、tree-sitter 解析、沙盒执行环境、跨越数千个文件的多文件上下文窗口。750 行和 750,000 行之间的差距是真实存在的。但架构是一样的：一个大脑、一个循环、工具、记忆，以及一套安全约束机制。现在你知道幕后是什么了。

模型会持续进步。而那套保障机制——循环、工具、安全检查——那部分是工程问题，它不会消失。

附录 A：流式响应

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

流式传输的工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

具体实现

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

与第 11 章相比的变化

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

权衡取舍

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

运行代码

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26

致谢

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 https://leanpub.com/build-your-own-coding-agent-zh-Hans-224caa26