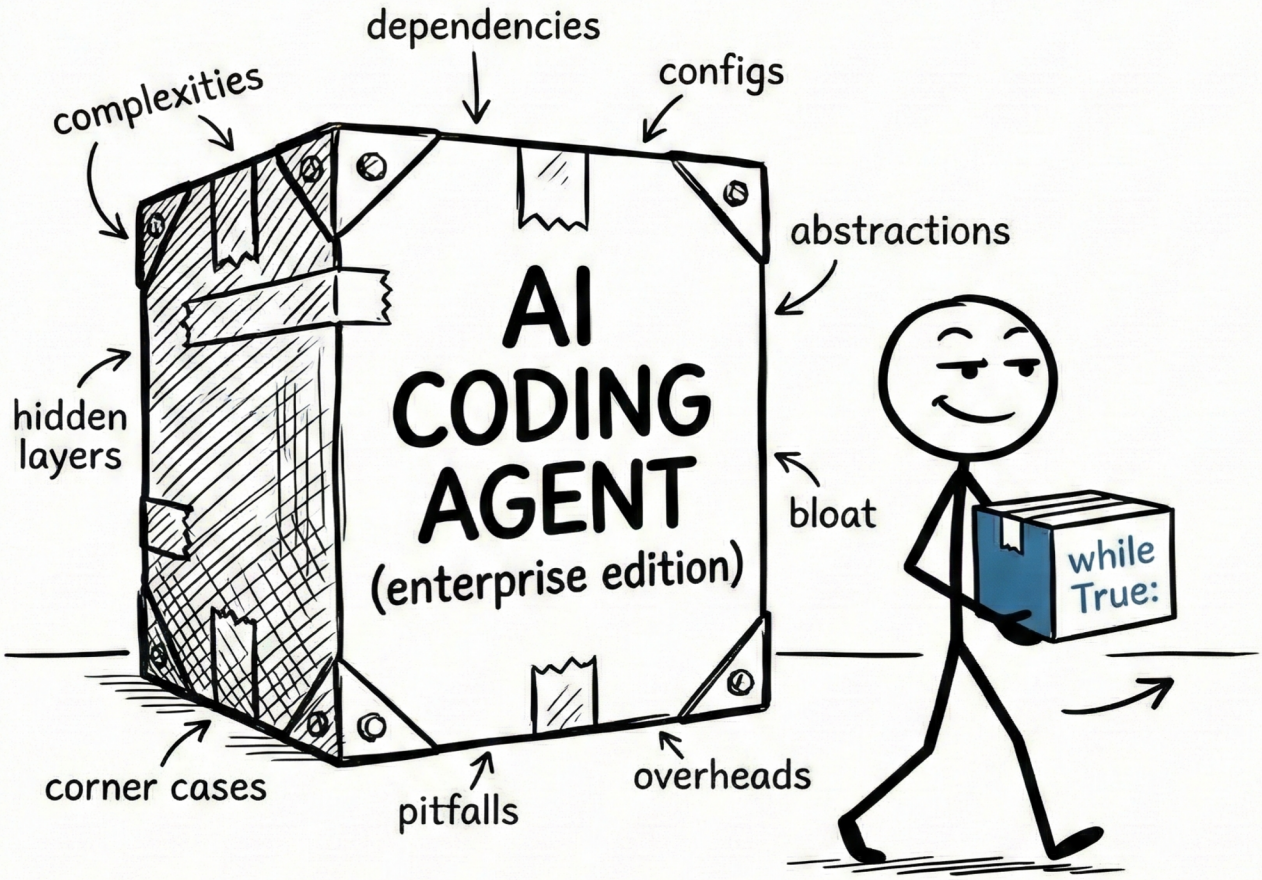
Build Your Own
CODING AGENT

The Zero-Magic Guide to AI Agents in Pure Python



Owen Ou

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

Owen Ou 和 TranslateAI

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

此版本发布于 2026-02-20

在 Twitter 上分享这本书!

请在 Twitter 上帮助作者 Owen Ou 和 TranslateAI 宣传!

对这本书建议的推文是:

刚用 HTTP 调用和命令行 shell 从零开始构建了一个代码智能体。原来所谓的"AI 魔法" 不过就是一个 while 循环加上一个 API 调用而已。

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 **1**
这本书适合谁 1
你将构建什么 1
测试方法 1
代码示例 2
本书使用的约定 2

第一部分：大脑 **4**

第一章：零魔法宣言 **5**
代理到底是什么？ 5
我们要构建什么 6
项目设置 6
AgentStop 异常 8
Agent 类 8
通过测试定义成功 9
主循环 11
运行程序 12
总结 12

第二章：原始请求 **13**
获取 API 密钥 13
密钥保管（.env） 13
请求的解析 14
代码 14
运行程序 16
故障排除 17
清理工作 18
总结 18

第三章：无限循环 **19**

记忆的假象 19
测试问题 19
响应类型 20
模拟大脑模式 21
定义成功标准 21
Claude 类 24
Agent 类（更新版） 25
主循环（已更新） 27
验证测试是否通过 28
测试内存 28
上下文窗口问题 29
总结 30

第 4 章：通用适配器 **31**
适配器模式 31
HTTP 弹性处理 31
大脑接口 31
FakeBrain（已更新） 31
Claude 大脑（重构版） 31
DeepSeek 大脑 31
BRAINS 注册表 32
Agent 类（更新版） 32
多智能模型支持的测试 32
主循环（已更新） 32
设置 DeepSeek 32
试一试 32
“我们只是移动了代码位置” 32
总结 33

第二部分：实践之手 **34**

第 5 章：工具协议 **35**
工具实际如何工作 35
定义工具接口 35
ReadFile 工具 35
写入文件工具 35
工具辅助函数 35
更新 Thought 类 35
更新 Claude 类 36

带工具的智能代理类 36
主循环 36
测试一下 36
总结 36

第 6 章：记事本（内存） 37
“零魔法 “的内存 37
内存类 37
ToolContext 类 37
SaveMemory 工具 37
更新 Claude 类 37
构建系统提示 37
更新 Agent 类 38
主循环（已更新） 38
测试持久性 38
总结 38

第 7 章：安全防护机制（计划模式） 39
概念 39
先写测试 39
扩展 ToolContext 39
受保护的 WriteFile 工具 39
代理类（已更新） 39
主循环（更新版） 39
测试框架 40
“计划 “的心理学 40
总结 40

第 8 章：上下文管道（映射与搜索） 41
ListFiles 工具 41
搜索代码库工具 41
更新工具列表 41
“深入探索 “测试 41
等等，这是 RAG 吗？ 41
总结 41

第 9 章：现实检验（运行代码） 42
反馈循环 42
先有测试 42
命令执行工具 42

交互式陷阱 42
自我修复演示 42
TDD 工作流程 42
精准修改 43
闭环 43
上下文压缩 43
安全注意事项 43
总结 43

第三部分：探索前沿 **44**

第十章：转向本地（本地模型） **45**
权衡 45
安装 Ollama 45
Ollama 大脑类 45
使用 Ollama 运行 45
“无限循环 “实验 45
实际差异 45
混合工作流程 46
模型选择 46
Ollama 故障排除 46
总结 46

第 11 章：扩展功能（网页搜索） **47**
第 1 步：元提示 47
第 2 步：手术过程 47
步骤 3：参考实现 47
步骤 4：测试 47
自我修改 47
总结 47

第 12 章：终章（构建一个游戏） **48**
第 1 步：准备工作 48
第 2 步：设计师（规划模式） 49
第 3 步：构建器（执行模式） 49
第 4 步：现实检查 50
第 5 步：转折点（功能蔓延） 51
可能出现的问题 51
总结 52
结语 52

致谢 54

前言

这本书适合谁

你是一位对 AI 炒作持怀疑态度的软件工程师。

你看过演示。你试过各种框架。你亲眼目睹过你的 LangChain 应用因为产生幻觉而删除了生产数据库。于是你想：“一定有更好的方法。”

确实有。这本书是为那些想要真正理解 AI 代理运行时发生什么的开发者而写的。不是营销图表。不是 “推理引擎 “抽象概念。而是实际的 HTTP 请求。实际的`while` 循环。

如果你会读 Python，而且之前构建过网络应用或 CLI 工具，你就具备了所需的一切条件。

你将构建什么

Nanocode 是一个在终端中运行的编码代理。在本书结束时，它将能够：

- 读写你代码库中的文件
- 执行 shell 命令
- 使用纯 Python 搜索代码
- 在会话之间保持上下文
- 在执行危险操作前请求许可
- 在网上搜索文档和答案

你将从零开始构建它，仅使用`requests`、`subprocess` 和`pytest`。不用 LangChain。不用向量数据库。不用 “编排框架 “。只用可以用`print()` 调试的 Python 代码。

唯一的例外是：第 11 章添加了用于网络搜索的`ddgs`——这是唯一一个轻量级依赖。

测试方法

本书采用 “同步测试 “方法。对于每个功能：

1. 我们介绍概念——为什么需要这个功能
2. 我们首先展示测试，这样你就知道成功的标准是什么
3. 我们实现代码以通过测试
4. 我们用 pytest 验证

这种方法教授了测试驱动开发的思维方式，而无需在书中完整展示红-绿-重构的仪式。它还解决了一个关键问题：你不能通过实际调用 LLM 来测试基于 LLM 的应用。API 调用很慢、很贵，而且结果不确定。

从第 3 章开始，我们将介绍`FakeBrain` 模式——一个能返回可预测响应的测试替身。这让你可以运行整个测试套件，而无需进行一次 API 调用或花费一分钱。

代码示例

本书遵循 “代码优先 “方法。每章都建立在前一章的基础上，每个代码示例都是从可运行文件中提取的。

获取代码：

- **GitHub：** 从 GitHub 克隆或下载。[1]
- **Leanpub：** 完整的源代码也包含在你购买后可下载的资源中。

代码按章节组织（`ch01/`、`ch02/`等）。每个文件夹包含：

- `nanocode.py` — 该章节完整的、可运行的代理代码
- `test_nanocode.py` — 无需 API 调用即可验证代码的测试

你可以复制任何章节文件夹并从那里开始。运行`pytest` 来验证你的代码是否符合预期行为。

本书使用的约定

在本书中，你会看到三种标注：

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

> ** 开发提示：** 适用于本项目之外的架构智慧。这些是你可以在自己的工作中重用的模式。

** 警告：** 安全风险。请注意这些内容——忽视它们可能导致文件被删除或 API 密钥泄露。

** 旁注：** 深入探讨和延伸内容。有用的背景知识，但首次阅读时可以跳过而不会影响主线理解。

代码讲解遵循一致的模式：首先是上下文（为什么需要这段代码），然后是代码本身，最后是重要部分的逐行解释。

现在开始编写代码吧。

第一部分：大脑

在这些开篇章节中，你将构建一个智能代理的框架：一个通过原始 HTTP 与 Claude 对话的`while` 循环。到第 4 章时，你将通过适配器模式支持多个大语言模型提供商——并理解 “AI 智能代理 “不过就是一个循环、一个大脑和一系列消息的组合。

第一章：零魔法宣言

如果你在过去两年中尝试构建人工智能应用，你很可能已经感受到了 “框架疲劳 “。

你安装了一个流行的库。你导入了一个 `ReasoningEngine`。你调用 `.run()`。对于 “Hello World“示例，它像魔法一样工作。但当你试图做一些实际的事情时——比如在不删除导入语句的情况下编辑 Python 文件中的特定行——它就崩溃了。

而且因为你使用了框架，你无法修复它。你不得不深入挖掘抽象类、工厂模式和 “链 “的层层结构，试图找出导致幻觉的那个提示。

我们在这里不会这样做。

这本书是对 “魔法 “的反叛。我们将采用 “零魔法 “方法：用纯 Python 构建一个生产级的编码代理，称为 **Nanocode**。不用 LangChain，不用 AutoGPT，不用 Pydantic。

为什么？因为自主代理并不是魔法。它只是一个 `while` 循环。

代理到底是什么？

撇开风险投资营销不谈，一个 “代理 “就像一个**恒温器**。

恒温器读取温度（输入），将其与目标温度比较（决策），然后打开加热器（动作）。之后等待并重复。就这么简单。人工智能代理做的事情也一样，只是用文本替代了温度。

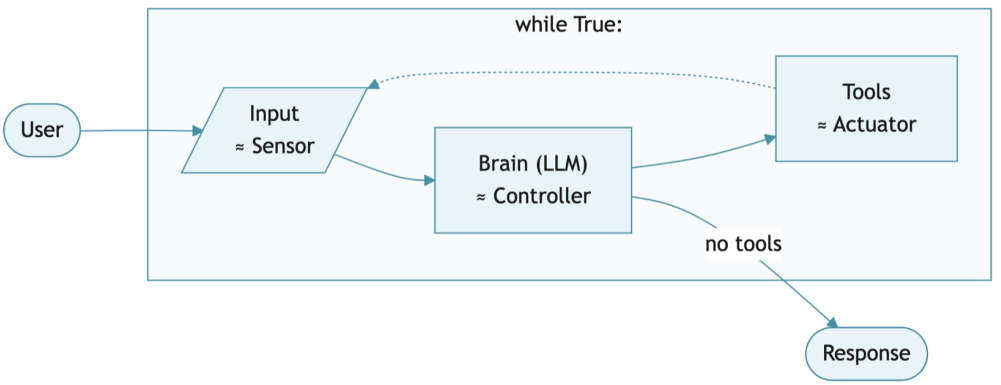


图 1. 代理循环：用户输入流经一个 while 循环，其中输入（传感器）流向大脑/LLM（控制器），后者触发工具（执行器），循环返回到输入，直到大脑输出响应。

更具体地说，一个代理有四个部分。大脑就是 LLM——一个无状态函数，你发送文本它返回文本。它调用工具——像读取文件和运行命令这样的函数——来与外部世界交互。所有这些都位于一个循环（一个 `while True`）中，循环会一直持续到任务完成，同时内存——就是一个 Python 列表——会在此过程中累积对话历史。（当程序结束时列表就会消失；我们会在第 6 章添加持久存储。）

如果你会写 `while` 循环，你就能构建一个代理。

通过从零开始构建，你将拥有框架用户所没有的东西：控制权。当我们的代理陷入循环时，你会准确知道是哪行代码导致的。当 API 账单变得太高时，你会清楚地看到代币在哪里泄漏。

我们要构建什么

Nanocode 是一个在终端中运行的命令行工具。你可以像和同事交谈一样与它对话。它能读取你的文件。它能运行你的命令。它能编辑你的代码。

到本书结束时，你将把它连接到 Claude Sonnet 4.6（或 DeepSeek，或通过 Ollama 的本地模型）。你将赋予它双手——读取文件、写入文件和运行 shell 命令的工具——以及搜索代码库的双眼。你还将构建一个安全防护装置，这样它就不会意外执行 `rm -rf /`。

项目设置

> ** 开发提示：** 对人工智能项目最大的威胁是 “依赖腐烂 “。人工智能库发展很快且经常破坏兼容性。通过仅使用 `requests` 和 `subprocess`，这个代理很可能在 5 年后仍然能运行。

1. 初始化项目

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

2. 创建虚拟环境

切勿全局安装 AI 工具。这会与系统包发生冲突。

```
# 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` 文件中加载 API 密钥。
- `pytest` — 用于在不进行 API 调用的情况下测试代码。

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 key 推送到 GitHub，机器人会在几分钟内抓取它并掏空你的账户。

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。使用异常比在代码中分散使用 `break` 语句要好。

** 背景：** 异常不仅仅用于错误处理；它们也是一种控制流机制。当用户输入 `/q` 时，我们抛出 `AgentStop`。主循环捕获它并实现清理退出。

代码：

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

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

这段代码位于 `nanocode.py` 的顶部。这是一个标记异常——没有逻辑，仅作为一个信号。

Agent 类

现在来看核心抽象：`Agent` 类。它将状态和逻辑集中在一处，这使得测试变得容易。

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

代码：

```
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)"
```

演练：

- ** 第 13-14 行：** 暂时为空的构造函数。我们将在后面的章节中添加`brain`和`tools`。
- ** 第 18-19 行：** 检查退出命令`/q`。抛出`AgentStop`而不是返回特殊值。
- ** 第 21-22 行：** 跳过空输入。返回空字符串（无需显示输出）。
- ** 第 24 行：** 回显输入内容。这是一个占位符——稍后，我们会将其发送到大脑。

通过测试定义成功

在编写主循环之前，我们需要测试。

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
```

一切正常。我们的代理程序正确处理了基本用例。

** 旁注：** 为什么选择 pytest？它能发现以test_开头的函数并运行它们。不需要样板代码，不需要类。测试代码就是普通的 Python 代码——没有任何魔法。

主循环

现在来看连接代理程序和终端的简单 I/O 包装器：

```python
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()` 并打印输出内容。
- **第 41-43 行：** 捕获 `AgentStop`（来自 `/q`）或 `KeyboardInterrupt`（来自 Ctrl+C）并正常退出。

附注： Python 的 `input()` 每次读取一行内容。本书中的所有提示都是单行的。这样可以保持代码简单——而在生产环境中的代理会使用更丰富的输入方式，如 readline 或完整的文本用户界面。

请注意这种分离：`Agent.handle_input()` 包含了所有的逻辑。`main()` 只是输入/输出的粘合代码。这使得代理可以在不需要模拟标准输入/输出的情况下进行测试。

运行程序

```
python nanocode.py
```

您应该看到：

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

 hello
You said: hello
(Agent not yet connected)

 /q

Exiting...
```

这就是框架。接下来是引擎部分。

总结

这就是框架：一个`Agent` 类、一个`handle_input()` 方法和一个`while True` 循环。目前它还没有任何实际功能——但我们接下来构建的所有内容都将插入这个骨架中。测试能确保我们在继续开发时不会破坏已有的功能。

第二章：原始请求

大多数教程会告诉你运行 `pip install anthropic`。但我们不会这样做。

SDK 隐藏了真相。它们增加了抽象层，使 “Hello World“变得容易，却让 “Error 400“的调试变成噩梦。当你学习 SDK 时，你只是在学习 SDK。但当你学习原始 HTTP 调用时，你学习的是所有 SDK 底层的协议。

我们将只使用 `requests` 库向 Claude 发送消息。Claude 是 Anthropic 的旗舰 LLM——是最擅长编程任务的模型之一。

获取 API 密钥

要与 Claude 对话，你需要一个 API 密钥。这是一串字符，就像你的信用卡一样重要。

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

警告：要像对待密码一样保护这个密钥。任何拥有它的人都可以消费你的账户余额。

密钥保管（.env）

我们需要一个安全的地方来存储这个密钥。我们决不能直接将密钥放在代码中。

在项目根目录创建一个名为 `.env` 的文件：

```
1 touch .env
```

打开并粘贴您的密钥：

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

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

我们在第 1 章安装 python-dotenv 就是为了这个目的——它读取 .env 并将值加载到 os.environ 中。

请求的解析

要与 LLM 对话，我们需要发送 HTTP POST 请求到：

```
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）、请求体中的配置信息（使用哪个模型，多少令牌），以及消息本身。

请求头

Anthropic 需要三个请求头：

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

负载

“Messages API” 需要一个消息字典列表：

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

每条消息都有一个 role（可以是 "user" 或 "assistant"）和 content（文本内容）。

代码

创建一个名为 test_api.py 的文件。这是一个 “冒烟测试 “，用于证明我们的连接是否正常工作。我们稍后会删除它。

上下文：我们正在编写线性过程式代码。不使用函数，不使用类。我们想要直接看到底层实现。

代码：

```python
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()` 定位并加载 `.env` 文件中的变量到 `os.environ`。
- **第 8 行：** 我们获取 API 密钥。切勿将其硬编码。
- **第 11-13 行：** 基础健全性检查。如果没有这个检查，缺失的密钥会在请求头字典中导致令人困惑的 `NoneType` 错误。
- **第 21 行：** `anthropic-version` 请求头是必需的。如果省略，API 会拒绝请求。
- **第 27 行：** `claude-sonnet-4-6` 指定我们要使用的模型。
- **第 28 行：** `max_tokens` 是必需的。它限制响应长度并防止成本失控。
- **第 36 行：** 我们发送请求时设置 2 分钟超时。这里没有使用 try/except——如果网络断开，就让 Python 崩溃。你需要看到具体在哪里失败。
- **第 41-47 行：** 检查状态码。200 表示成功（格式化打印 JSON）。其他任何状态码都会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
  }
}
```

故障排除

错误	原因	解决方法
401 Unauthorized	API 密钥错误	检查 `.env` 是否正确加载。打印 `os.environ.get("ANTHROPIC_API_KEY")` 进行验证。
400 Bad Request	JSON 格式错误	是否忘记设置 `max_tokens`？`messages` 是否为列表？
429 Rate Limit	请求过多或额度用尽	等待一段时间，或为账户充值。

错误	原因	解决方法

清理工作

我们已经证实可以与 brain 进行通信。删除 `test_api.py`——我们不再需要它了。

补充说明：这个 `test_api.py` 是一次性代码——仅用于进行一次性的冒烟测试。真正的自动化测试（使用 `FakeBrain` 和 pytest）将在第 3 章介绍。在验证 API 连接正常工作后，你应该始终删除这个文件。

开发提示：每次 API 调用都会产生费用。输入令牌（你发送的内容）费用较低。输出令牌（模型生成的内容）的费用大约是输入的 5 倍。请保持你的提示简洁。

补充说明：要监控你的支出，请查看 Anthropic Console 中的使用量标签页。截至 2026 年初，使用 Claude Sonnet 进行 20-30 次交互的典型编程会话费用在 0.10-0.50 美元之间。响应 JSON 底部的 `usage` 字段显示确切的令牌计数——你可以通过程序记录这些数据来追踪成本。

总结

这就是原始的 API 调用：包括请求头、JSON 负载和响应解析。在你和底层之间没有任何抽象。当出现问题时（这种情况一定会发生），你可以准确知道是哪一层出了问题，因为只有一层。

一个问题是：Claude 会完全失忆。每个请求都是一个全新的开始。我们将通过在每次交互时重放整个对话历史来模拟记忆功能。

第三章：无限循环

我们遇到了一个问题。

运行第二章的脚本两次。说 “我的名字是爱丽丝。“Claude 说你好。再次运行并询问 “我的名字是什么？“Claude 说 “我不知道。”

这是因为大语言模型是无状态的。它们会完全失忆。每个请求对它们来说都像是第一次见面。

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。

记忆的假象

大语言模型中的 “记忆 “不是硬盘。它是一个日志文件。

当你与 ChatGPT 聊天时，它并不 “记得 “你 5 分钟前说过什么。在后台，代码会在每个新消息中将**整个对话历史**发送回模型。

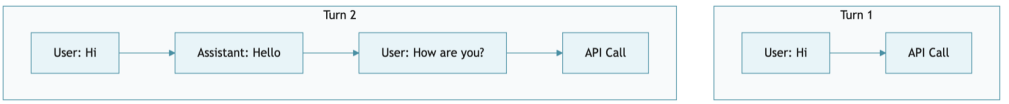


图 2. 上下文累积：第 1 轮只向 API 发送 “User: Hi“。第 2 轮向 API 发送完整历史记录—“User: Hi”、“Assistant: Hello”、“User: How are you?”

模型每次都会看到完整的对话记录。这就是诀窍所在。

我们将手动实现这个上下文循环。但首先，我们需要让我们的代码可测试。

测试问题

这是一个残酷的事实：你不能通过实际调用大语言模型来测试基于大语言模型的应用程序。

API 调用速度慢（每次 2-10 秒），成本高（每次调用都需要真金白银），而且非确定性（每次可能得到不同的响应）。想象一下运行一个花费 5 美元且需要 20 分钟的测试套件。你永远不会去运行它。

解决方案是依赖注入。我们不在智能体内部硬编码 API 调用，而是传入一个 “大脑 “对象。在生产环境中，这个大脑就是 Claude。在测试中，这个大脑是一个返回可预测响应的假对象。

在编写更多生产代码之前，我们将建立这个模式。

响应类型

在构建大脑之前，我们需要定义它返回什么。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。

上下文： Claude 可以在单个响应中返回文本、工具调用，或两者都返回。我们需要数据类来表示这些可能性。

代码：

```
class ToolCall:
    """A tool invocation request from the brain."""

    def __init__(self, id, name, args):
        self.id = id
        self.name = name
        self.args = args  # dict
```

`ToolCall` 表示大脑请求我们执行一个工具。`id` 是用于跟踪的唯一标识符（当我们向 Claude 报告结果时需要它）。`name` 是要运行的工具名称。`args` 是一个参数字典。

我们暂时不会使用 `ToolCall`——因为大脑还不能调用工具——但我们现在定义它是因为它是 `Thought` 响应类型的一部分。当我们添加工具后，Claude 想要读取文件或执行命令时就会返回这些内容。

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

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

Thought 是大脑思考后返回的结果。它可能包含文本、工具调用，或者两者都有，也可能两者都没有。这种抽象设计将使我们之后能够将 Claude 替换为 DeepSeek，而无需更改任何其他代码。

模拟大脑模式

现在我们可以构建一个用于测试的模拟大脑。

背景：我们需要一个大脑模型，它能返回可预测的响应，追踪被调用的次数，并记录接收到的对话内容。

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")
```

这段代码应该放在 test_nanocode.py 中，而不是生产代码中。注意 FakeBrain 与我们真实的大脑具有相同的接口——一个接收对话并返回 Thought 的 think() 方法。

附注：这种模式——在测试中用可预测的伪对象替换真实依赖——被称为依赖注入。Martin Fowler 的文章 “Mocks Aren’t Stubs“[1]解释了其中的变体（伪对象、存根、模拟对象、间谍对象）。对于 LLM 测试，通常只需要一个带有预设响应的简单伪对象就足够了。

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

定义成功标准

在编写生产代码之前，让我们先定义成功的标准。这些测试将指导我们的实现。

测试 1：大脑返回响应

```
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` 传递给 Agent。这就是依赖注入的实际应用。

测试 2：对话会累积

```
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

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

每次交互后，对话都应该同时包含用户消息和助手回复。

测试 3：正确的消息结构

```
def test_conversation_contains_correct_roles():
    """Verify conversation has correct role alternation."""
    brain = FakeBrain(responses=[Thought(text="AI response")])
    agent = Agent(brain=brain)

    agent.handle_input("User message")

    assert agent.conversation[0]["role"] == "user"
    assert agent.conversation[0]["content"] == "User message"
    assert agent.conversation[1]["role"] == "assistant"
    assert agent.conversation[1]["content"] == "AI response"
```

消息必须具有 Claude 期望的确切格式：`{"role": "user", "content": "..."}`。

测试 4：Brain 接收对话

```
def test_brain_receives_conversation():
    """Verify brain.think is called with the conversation list."""
    brain = FakeBrain()
    agent = Agent(brain=brain)

    agent.handle_input("Test message")

    assert brain.last_conversation is not None
    assert len(brain.last_conversation) == 1
    assert brain.last_conversation[0]["content"] == "Test message"
```

大脑必须接收完整的对话，而不仅仅是当前的消息。

现在运行这些测试——它们应该都会失败：

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

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

好，现在让我们让它们通过测试。

Claude 类

现在到了核心部分。

** 背景：** 我们需要一个封装 Claude API 的类。它应该处理身份验证、发送对话历史，并将响应解析为一个 `Thought`。

代码：

```python
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"
        }
        payload = {
            "model": self.model,
            "max_tokens": 4096,
            "messages": conversation
        }

        print("(Claude is thinking...)")
```

```
59          response = requests.post(self.url, headers=headers, json=payload,
            ↪  timeout=120)
60          response.raise_for_status()
61          return self._parse_response(response.json()["content"])
```

详解：

- ** 第 39-41 行：** 加载 API 密钥，如果密钥缺失则快速失败。
- ** 第 43-44 行：** 存储配置。我们稍后会使配置模型变得可配置。
- **第 46 行：**`think()` 方法是大脑的接口——与 `FakeBrain` 相同。
- ** 第 52-55 行：** 负载包含 `"messages": conversation`——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 61 行：** 将 Claude 的复杂响应格式解析为我们的简单 `Thought`。

现在是响应解析器：

```
63      def _parse_response(self, content):
64          """Convert Claude's response format to Thought."""
65          text_parts = []
66          tool_calls = []
67
68          for block in content:
69              if block["type"] == "text":
70                  text_parts.append(block["text"])
71              elif block["type"] == "tool_use":
72                  tool_calls.append(ToolCall(
73                      id=block["id"],
74                      name=block["name"],
75                      args=block["input"]
76                  ))
77
78          return Thought(
79              text="\n".join(text_parts) if text_parts else None,
80              tool_calls=tool_calls
81          )
```

Claude 的 API 会返回一个 “内容块 “列表。每个块都有一个 `type` —— 可能是 `"text"` 或 `"tool_use"`。我们将所有文本块收集到一个字符串中，并将 tool_use 块转换为 `ToolCall` 对象。

Agent 类（更新版）

现在我们更新第一章中的 Agent，使其能够接受大脑组件并维护对话历史。

代码：

```
86  class Agent:
87      """A coding agent with conversation memory."""
88
89      def __init__(self, brain):
90          self.brain = brain
91          self.conversation = []
92
93      def handle_input(self, user_input):
94          """Handle user input. Returns output string, raises AgentStop to
            ↪ quit."""
95          if user_input.strip() == "/q":
96              raise AgentStop()
97
98          if not user_input.strip():
99              return ""
100
101         self.conversation.append({"role": "user", "content": user_input})
102
103         try:
104             thought = self.brain.think(self.conversation)
105             text = thought.text or ""
106             self.conversation.append({"role": "assistant", "content": text})
107             return text
108         except Exception as e:
109             self.conversation.pop()  # Remove failed user message
110             return f"Error: {e}"
```

演练：

- ** 第 89-91 行：** 通过依赖注入接收智能核心。初始化一个空的对话列表。
- ** 第 101 行：** 在调用智能核心之前将用户消息添加到历史记录中。
- ** 第 103-107 行：** 调用智能核心，提取文本，将响应添加到历史记录中。

- ** 第 108-110 行：** 如果 API 调用失败，删除我们刚刚添加的用户消息。这样可以保持对话状态的有效性。

注意第 101 行：我们在调用智能核心之前添加用户消息。智能核心需要看到包含当前消息在内的完整对话。

> ** 开发提示：** 当出现问题时，你的首选调试工具是 `print(self.conversation)`。这可以准确显示智能核心所看到的内容。检查原始列表时，格式错误的消息、缺失的角色或截断的内容都会变得很明显。

主循环（已更新）

主循环现在只是一个简单的输入/输出包装器：

```
115 def main():
116     brain = Claude()
117     agent = Agent(brain)
118     print("⚡ Nanocode v0.2 (Conversation Memory)")
119     print("Type '/q' to quit.\n")
120 
121     while True:
122         try:
123             user_input = input("  ")
124             output = agent.handle_input(user_input)
125             if output:
126                 print(f"\n{output}\n")
127 
128         except (AgentStop, KeyboardInterrupt):
129             print("\nExiting...")
130             break
131 
132 
133 if __name__ == "__main__":
134     main()
```

所有的逻辑都在 Agent 类中。循环只是读取输入，调用 handle_input()，并打印结果。这种分离使得代理变得可测试——我们可以直接测试 Agent.handle_input()，而不需要模拟 input() 或 print()。

验证测试是否通过

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
```

测试全部通过。这些测试验证了我们的实现，而没有发起一次 API 调用。

测试内存

现在用真实的大脑来测试：

```
python nanocode.py
```

来试试这段对话：

```
 I am building a Python agent.
(Claude is thinking...)

That sounds exciting! Building a Python agent is a great project...

 What language am I using?
(Claude is thinking...)

You are using Python.
```

对话列表正在发挥其作用。

上下文窗口问题

你可能在想：“我可以一直这样运行下去吗？”

不能。

每次循环迭代，`messages` 列表都会增长：

轮次	大约令牌数
1	50
10	5,000
100	50,000

最终，你会达到上下文限制——Claude Sonnet 是 20 万个令牌，DeepSeek 是 12.8 万个，某些本地模型低至 4 千个。超过限制后，API 会返回`400 Bad Request`。我们在第 108 行的错误处理会捕获这个错误并报告，这样代理就不会悄无声息地崩溃。但是对话实际上已经卡住了——由于历史记录仍然太长，之后的每条消息都会失败。

目前，重启代理可以清除历史记录并让你重新开始。在第 9 章构建反馈循环时，我们会添加适当的上下文压缩——从 API 响应中跟踪令牌使用情况，并在溢出之前自动总结旧消息。那时对话确实会膨胀，而且这个解决方案将发挥其价值。

> **开发提示：**注意到`print("(Claude is thinking...")`了吗？网络调用需要 2-10 秒。始终要立即反馈 Enter 键已生效，否则用户会以为程序冻结而按 Ctrl+C。

总结

Claude 现在能记住了——或者更确切地说，我们让它认为它记住了。对话列表在每一轮都会增长，而`FakeBrain`让我们可以在不花一分钱的情况下测试整个系统。

这两种模式将贯穿本书的其余部分。我们构建的每个大脑（Claude、DeepSeek、Ollama）都将实现相同的`think()`接口，而`FakeBrain`将测试它们所有。

还有一个遗留问题：我们的代码是硬编码到 Anthropic 的 API 的。如果我们想添加 DeepSeek 或本地模型，我们就必须复制大量代码。

第 4 章：通用适配器

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

适配器模式

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

HTTP 弹性处理

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

大脑接口

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

FakeBrain（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

Claude 大脑（重构版）

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

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

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

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

多智能模型支持的测试

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

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

设置 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

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

“我们只是移动了代码位置”

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

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

第二部分：实践之手

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

第 5 章：工具协议

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

工具实际如何工作

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

定义工具接口

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

ReadFile 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

写入文件工具

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

工具辅助函数

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

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

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

带工具的智能代理类

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

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

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

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

第 6 章：记事本（内存）

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

“零魔法 “的内存

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

内存类

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

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

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

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

构建系统提示

更新 Agent 类

主循环（已更新）

测试持久性

总结

第 7 章：安全防护机制（计划模式）

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

概念

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

先写测试

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

扩展 ToolContext

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

受保护的 WriteFile 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

代理类（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

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

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

“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

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

第 8 章：上下文管道（映射与搜索）

ListFiles 工具

搜索代码库工具

更新工具列表

“深入探索 “测试

等等，这是 RAG 吗？

总结

第 9 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

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

先有测试

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

命令执行工具

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

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

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

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

精准修改

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

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

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

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

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

第三部分：探索前沿

第十章：转向本地（本地模型）

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

权衡

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

安装 Ollama

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

Ollama 大脑类

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

使用 Ollama 运行

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

“无限循环 “实验

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

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

混合工作流程

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

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

Ollama 故障排除

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

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

第 11 章：扩展功能（网页搜索）

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

第 1 步：元提示

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

第 2 步：手术过程

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

步骤 3：参考实现

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

步骤 4：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

自我修改

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

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

第 12 章：终章（构建一个游戏）

Nanocode 真的能构建些什么吗？

“零代码挑战”：使用 Python 和 Pygame 构建一个经典的贪吃蛇游戏。规则很简单——你不允许编写任何 Python 代码。你只能用英语与代理对话。

** 附注：** 这个演示涉及许多 API 调用。如果你遇到速率限制（HTTP 429），代理会自动重试。对于较长的会话，考虑使用 Ollama 的本地模型来完全避免限制。

第 1 步：准备工作

从你的项目根目录（包含`nanocode.py` 和`.env` 的目录）开始，创建一个游戏的工作目录：

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

如果你正在使用本书的代码仓库，请从 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：在 Windows 上，这应该可以直接使用。在 macOS 上，你可能需要先安装 SDL 库：`brew install sdl2 sdl2_image sdl2_mixer sdl2_ttf`。在 Linux 上，安装 SDL 开发包：`sudo apt install libsdl2-dev libsdl2-image-dev libsdl2-mixer-dev libsdl2-ttf-dev`。

第 2 步：设计师（规划模式）

启动代理程序。我们从规划模式开始，因为在开始砌砖之前，我们需要一份蓝图。

```
python nanocode.py
```

提示语：

```
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.
```

代理将使用 `write_file` 来创建 `PLAN.md`。阅读它。它应该在单个文件中概述蛇类、食物类和游戏循环。

如果看起来没问题，就批准它。

第 3 步：构建器（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.
```

查看终端：

```
  Writing snake.py
```

代理程序正在基于存储在 PLAN.md 中的上下文生成代码。

第 4 步：现实检查

在运行游戏之前，需要增加超时时间。打开 nanocode.py 并在 RunCommand.execute() 中将 timeout=30 改为 timeout=300——默认的 30 秒不足以完成一局游戏。（这是零代码挑战的唯一例外。）

提示：

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

代理执行 run_command。一个窗口弹出。你开始玩贪吃蛇游戏。

> ** 开发提示：** 游戏窗口会阻塞代理。当你运行 python snake.py 时，代理会等待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。

** 如果游戏崩溃：** 大语言模型的输出是非确定性的。你的代理在第一次尝试时可能会产生 bug。如果游戏出现类似 AttributeError: 'Snake' object has no attribute 'draw' 这样的错误而崩溃，不要自己修复它。让代理看到 stderr 的输出。

提示：

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

代理程序将读取回溯信息，使用 read_file 找到程序错误，使用 edit_file 进行修复，然后重新运行。

> ** 开发提示：** 这就是第 9 章中提到的反馈循环在实践中的应用。代理程序编写代码、运行程序、读取错误并进行修复。你只需要观察整个过程。

第 5 步：转折点（功能蔓延）

游戏可以运行，但看起来很粗糙。蛇只是由绿色方块组成。让我们来测试一下代理程序重构代码的能力。

提示语：

```
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.
```

代理应该：

1. 使用 `search_web` 查找调色板
2. 使用 `read_file` 理解当前的渲染逻辑
3. 使用 `edit_file` 注入新功能
4. 运行游戏进行验证

可能出现的问题

你的结果会与我的不同——LLM 具有非确定性。但以下是通常会发生的情况，以及需要注意的事项。

首次运行时的常见失败：

- `ModuleNotFoundError: No module named 'pygame'` — 代理忘记了你需要安装它，或者它在不同的环境中运行脚本。告诉它先运行 `pip install pygame`。
- 在调用点上出现代理定义但拼写错误的方法时会产生 `AttributeError`。一旦代理看到回溯信息就会快速修复这些问题。
- 碰撞检测中的差一错误。蛇会穿过墙壁或过早死亡。这些问题需要 2-3 次编辑-运行-修复的迭代。

典型会话流程：

在我的测试中，代理通常在 2-4 次迭代后就能得到一个可运行（但不够美观）的游戏。第一次编写会产生崩溃的代码。第二或第三次修复后能让它运行起来。功能扩展步骤（颜色变化、速度调整）会再增加 3-5 次迭代，因为代理需要阅读自己的代码，进行精确的编辑，并验证每个更改。

到最后，对话通常会深入到 15-20 轮。如果你使用 Claude，要注意压缩触发器——在第 12-15 轮左右，当令牌数接近 75% 阈值时，你会看到 “(正在压缩对话...)“。压缩后，代理会失去一些早期回合的细节，但仍能继续工作。这正是第 9 章中的系统发挥作用的地方。

代理的困难之处：

Pygame 的坐标系统和事件循环比较棘手。代理有时会写出能正确渲染但无法正确处理键盘输入的代码，或者以错误的顺序绘制蛇导致蛇头出现在身体后面。这些是人类一眼就能发现但代理看不到的错误——它没有视觉反馈，只有标准输出和标准错误。如果游戏运行时没有错误但看起来不对，你需要描述视觉问题：“蛇的渲染是反的——蛇头应该在前面。”

关键不在于第一次就要完美。重要的是代理能够收敛——写代码、运行、读取错误、修复、重复——使用我们在十一章中构建的每一个工具。

总结

计划、实现、崩溃、调试、修复、再运行——这体现了每一章的价值。

那么接下来会如何发展？

结语

整个项目大约有 750 行 Python 代码。没有框架。`nanocode.py` 是你的了。你可以用它做任何事：

- 在测试通过后自动提交的 Git 集成
- 基于截图的前端工作调试（Claude 可以读取图像）
- 通过 Whisper 进行语音输入，这样你可以说话而不是打字
- 用于连接团队已在使用的外部服务的 MCP

像 Claude Code、Cursor 和 Copilot 这样的生产级代理能做的更多——流式响应、并行工具执行、tree-sitter 解析、沙盒执行环境、跨越数千文件的多文件上下文窗口。从

750 行到 750,000 行的差距是真实存在的。但架构是相同的：一个大脑、一个循环、工具、内存和安全防护机制。现在你知道幕后是什么了。

模型会继续改进。而防护机制——循环、工具、安全检查——那部分是工程。那部分不会消失。

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