

Vibe Handbook

Vibe Handbook

目录

Git & GitHub	5
什么是版本控制（以及为什么你需要它）	6
仓库、提交，以及历史	6
分支：无所畏惧地尝试	7
看清到底改了些什么（diff）	7
“糟糕”按钮：undo、revert、reset	8
远程与 GitHub：云端里的你的项目	8
拉取请求与议题	8
.gitignore 与绝不提交机密	9
你的第一个 repo，从头到尾	9
让 AI 来开车（而你仍掌舵）	9
回顾与练习	10
The Command Line	11
终端和 shell 究竟是什么	11
读懂提示符	12
每天都会用到的命令	12
运行一个程序和一个 dev 服务器	13
读懂输出和 exit 码	13
用 Ctrl-C 停掉一个进程	13
PATH 与“command not found”	13
安全地运行 AI 给你的命令	14
你会遇到的包管理器	14
回顾与练习	14
How the Web Works	16
两个角色：客户端与服务器	16
URL 实际上是什么	16
请求与响应的循环	17
HTTP 方法：动词们	17
状态码：成功了吗？	18
头部与 Cookie	18
HTTPS 与那把锁	18
从名字到服务器：DNS	18
这一切如何对应到你的应用	19
回顾与练习	19
API、JSON 与数据	20
API 究竟是什么	21
用大白话讲端点和 REST	21
JSON 是什么	21
你会遇到的数据类型	22

一个真实的请求和响应	22
API key 与认证 header	23
速率限制与账单暴雷陷阱	23
怎么读 API 文档	24
让 AI 去做调用——而你来检查形状	24
回顾与练习	24
一个应用的解剖	25
全局图景	26
frontend、backend、数据库、托管	26
HTML、CSS 和 JavaScript	26
包和依赖	27
环境变量与配置	27
一个项目的样子	28
一个请求是怎么流动的	29
回顾与练习	29
Vibe Coding 心态	30
Vibe coding 究竟是什么	31
心态的转变	31
循环：意图 □ 生成 □ 审查 □ 打磨	32
它的闪光之处	33
它失效的地方	33
常见的坑	33
品味是新的核心技能	34
现实的预期	34
回顾与练习	34
搭建你的 AI 工作环境	35
选择一个 AI 原生的编辑器或助手	36
把项目结构组织得让 AI 能导航	36
写一个上下文文件 (AGENTS.md)	37
让版本控制成为本能反应	38
让反馈循环保持紧凑	38
谨慎管理机密	39
一个健全的本地配置让 AI 的改动安全	39
实时诊断 (LSP)	39
扩展你的智能体	40
macOS vs Windows	40
TUI vs GUI	40
护具工程 (Harness Engineering)	41
成本安全：CLI vs API	42
回顾与练习	42
像工程师一样写提示词	43
在给出任务之前先给出上下文	43
指明输入、输出和边界情况	44
以小步前进	44
在写代码之前先要一份计划	45
在 diff 上迭代，而不是重写	45
提供示例	46
告诉 AI 不要做什么	46
像管理资源一样管理上下文	47
要避免的常见反模式	47
核心心态	47

回顾与练习	48
几分钟内从想法到规格	49
为什么要规格，而不只是凭感觉	49
第一步：厘清问题	50
第二步：定义范围与非目标	50
第三步：编写用户故事	51
第四步：起草一份轻量 PRD	51
第五步：勾勒数据模型	52
第六步：把规格拆成 <i>vibe</i> 大小的任务	53
让规格保持鲜活	53
一个常见陷阱：规定的是解决方案，而非问题	54
回报	54
回顾与练习	54
选择你的技术栈：语言一览	55
JavaScript / TypeScript	56
Python	56
Go	57
Rust	57
关于其他语言的简短一说	57
一览对比	58
异类选择的真实代价	58
按目标来选	59
回顾与练习	59
构建功能，一个 <i>vibe</i> 接一个 <i>vibe</i>	60
构建循环	61
“纵向切片”是什么意思	61
一个实战示例：“收藏”功能	62
不必看懂每一行也能读 diff	63
何时放手让 AI 跑，何时手动掌舵	64
让应用始终保持可运行	64
小步提交	64
当一个切片太大时	65
要点带走	65
回顾与练习	65
用 AI 调试	66
核心心态	66
一份调试清单	67
给 AI 喂正确的上下文	68
阅读你看不懂的错误	69
假设、日志和二分查找	69
根本原因与表象	70
当 AI 开始兜圈子	70
在保留修复之前先理解它	70
要点	70
回顾与练习	71
服务器基础设施	72
代码可以待的五个地方	72
几个行话术语，破译一下	73
取舍对照表	73
何时选哪个	74

AI 如何串联部署与配置	74
给单打独斗构建者的推荐默认路径	75
“托管 (managed)”到底意味着什么 (以及你仍然拥有什么)	76
一份快速的上线前清单	76
要点	77
回顾与练习	77
数据库基础设施	78
数据存储的三种形态	78
托管 vs 自托管	79
Schema 设计基础	79
迁移: 安全地更改 Schema	80
索引	81
缓存	82
队列	82
常见的初学者错误	82
指挥 AI 构建你的数据层	82
回顾与练习	83
防护栏、测试与审查	84
为什么指挥 AI 时你需要防护栏	84
测试: 你的安全网	85
类型检查与代码检查: 免费抓出的错误	86
限定 AI 改动的范围	86
带着批判地阅读 diff	87
审查关卡: 合并前的一份清单	87
CI 基础: 让检查自动化	87
默认安全的习惯	88
当防护栏尖叫时: 先分诊, 再修	88
回顾与练习	89
部署上线到生产环境	90
选一个托管平台并部署	90
自定义域名与 DNS	91
环境变量与密钥	92
接入支付	93
预发布环境与回滚	94
监控与错误追踪	94
性能与缓存	95
上线前清单	95
上线之后	95
回顾与练习	95
案例研究: 从 0 到上线	96
案例研究 1: 一个多语言落地页 + 支付页	97
案例研究 2: 一个带认证 + 数据库的小型 SaaS 工具	98
案例研究 3: 一个个人自动化小工具	100
案例研究 4: 一个不用你打理的内容站	101
这四个项目的共同点	103
回顾与练习	103
写给 Vibe Coder 的安全课	104
为什么 AI 生成的代码默认就是不安全的	105
注入: 当输入变成了代码	105
密钥与 API key: 密钥外泄陷阱	106

认证 vs 授权：那个谁都能调的接口	107
文件处理与上传	107
依赖风险：AI 凭空发明的那个包	108
安全审查关卡	108
自动化自满，安全篇	109
回顾与练习	109
智能体编排	110
三个层级	111
何时放手运行，何时手动操纵	111
给任务划定范围，让它无法乱跑	111
后台与并行智能体	112
多智能体：扇出、对手、综合	113
按规模审查输出	114
成本与失控的循环	114
诚实的限度	115
回顾与练习	115

Git & GitHub



如果在开始 vibe coding 之前你只学一个工具，那就学这个。Git 是让你敢于放手的工具。正因为有它，你才能对 AI 说”把整个文件重写一遍”还睡得着觉——因为你随时都能回到那个能正常运行的版本。

实话实说是这样的。当你指挥 AI 来构建软件时，它一定会弄坏东西——而且是经常。它会信心十足地重写一个文件，删掉本

来正常工作的部分，然后甩给你一团乱麻。一个糟糕的下午和损失一整周的区别，就在于你有没有一张安全网。Git 就是那张网。本章从零教起，不需要任何编程基础。

什么是版本控制（以及为什么你需要它）

想象一款有存档点的电子游戏。你通过一个棘手的关卡，存档，然后尝试一些冒险的操作。如果出了岔子，你读取上一个存档，而不是从头开始整个游戏。**版本控制就是你项目的存档点。**

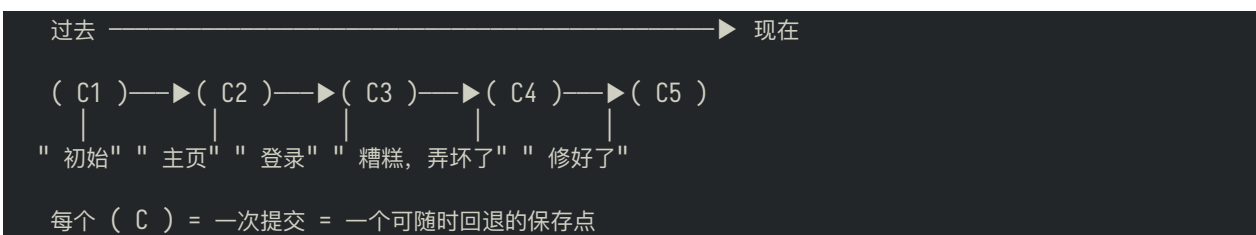
没有版本控制，你的项目只有一个状态：此刻它是什么样就是什么样。如果 AI 把代码弄坏了，那个好版本就没了。有了版本控制，你保留着每一个存档状态的完整时间线，可以跳回其中任意一个。

Git 是最常用的版本控制工具。它在你的电脑上运行，每当你要求时，就悄悄记录下项目的快照。它免费给你的几样东西：

- **一张安全网**——哪怕几小时后，也能撤销一次糟糕的改动。
- **一份历史**——精确看到什么变了、何时变的、为什么变。
- **底气**——尽情实验，因为没有任何东西会真正丢失。

最后这一点才是真正的奖赏。Vibe coding 在你快速行动、勇于尝试时效果最好。Git 让这种快速变得安全，而不是鲁莽。

把项目的历史想象成一排你随时可以回去的存档点：



仓库、提交，以及历史

三个你会不断听到的词。我们把它们讲清楚。

仓库 (repository, 简称“repo”) 就是一个 Git 在盯着的文件夹。你只需把一个普通文件夹变成 repo 一次，从那以后 Git 就会追踪里面的一切。

提交 (commit) 是一个存档点。它是某一时刻所有文件的快照，外加你写的一句简短说明，描述改了什么（“加上登录页面”“修好坏掉的注册按钮”）。你会不断地做提交——把每一个都想成一个你可能想回去的检查点。

历史是所有提交按顺序排成的列表。它就是时间线。你可以滚动浏览，读每次改动是什么，并穿越回任意一个时间点。

实际操作起来是这样的。不用背——大多数时候你会让 AI 替你运行——但跟着读一遍，让这些词不再神秘：

```
# 把当前文件夹变成一个 Git 仓库（只需做一次）
git init

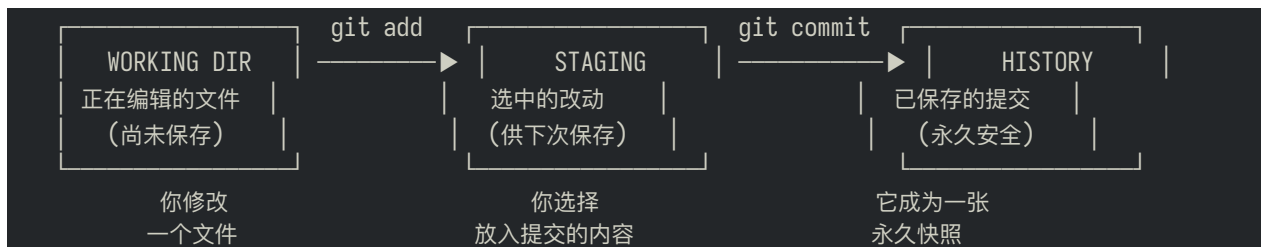
# 查看自上一个存档点以来哪些文件变了
git status

# 暂存你的改动，然后带一句说明保存为一次提交
git add .
git commit -m "Add the homepage layout"

# 查看你的历史——每一次提交，最新的在前
git log --oneline
```

`git add .` 是你在说“把我所有的改动都纳入下一个存档点”。`git commit` 才是真正的保存。`-m "..."` 是写给未来自己的便条。写一些一个月后你自己还能看懂的便条。

这两条命令把你的工作推过三个阶段。你编辑的文件，只有一路走到最右边，才算真正保存：



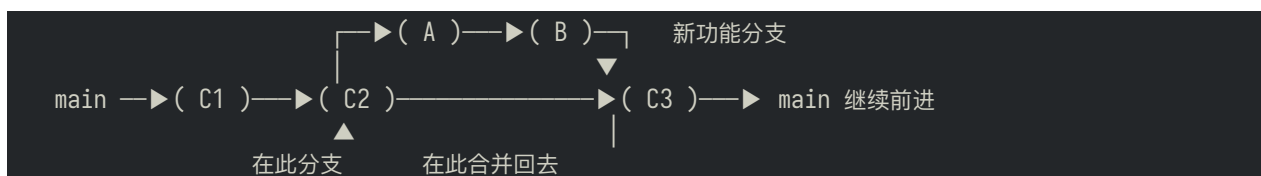
分支: 无所畏惧地尝试

分支 (branch) 是项目的一份平行副本，你可以在上面做实验。你的主要工作通常在一个叫 `main` 的分支上。当你想尝试点什么时——一个新功能，或一次冒险的 AI 重写——你新建一个分支，在那上面做，而你的 `main` 保持原封不动、安然无恙。

如果实验成功，你把它合并回 `main`。如果失败，你把分支丢掉，`main` 压根不知道发生过这回事。

这正是你该用来对待重大 AI 改动的方式。“把整个支付流程做出来”配得上拥有自己的分支。AI 拿出好东西，就合并；拿出一场灾难，就删掉分支，你什么都没损失。

画出来看，一个分支从 `main` 岔出去、自己生长，然后再合并回来：



```
# 新建一个分支并切换过去
git checkout -b new-feature

# ... 做你的工作，做几次提交 ...

# 切回你那安全的 main 分支
git checkout main

# 把分支上的好成果合并进 main
git merge new-feature
```

看清到底改了些什么 (diff)

在保存一次改动之前——尤其是在接受 AI 做的改动之前——你会想看清究竟有什么不同。这就是 **diff**。它一行一行地告诉你，什么被加上了（通常是绿色），什么被删掉了（通常是红色）。

这是你最重要的习惯之一。Vibe coding 的整套哲学就是理解你所交付的东西。你不必亲手敲代码，但你必须看一眼改了些什么并大致理解它。diff 就是你做这件事的地方。

```
# 查看你改了但还没提交的内容
git diff

# 查看已暂存、准备好提交的内容
git diff --staged
```

当 AI 说“搞定了，我更新了文件”，你的下一步就是看 diff。如果你看到它删掉了你需要的东西，你当下就能抓住——而不是在它交付之后。

“糟糕”按钮: *undo*、*revert*、*reset*

正是这一部分，把 Git 从一桩苦差变成了一种超能力。当出了岔子时——它一定会——这样把它倒回去。

- 丢弃未保存的改动——你（或 AI）编辑了一些文件、把事情弄得更糟，但你还没提交。你可以把一切都退回到上一个好提交。
- 撤销一次提交（*revert*）——你保存了一次改动，然后发现它很糟。git revert 会做一个新的提交来抵消那个糟糕的提交，让你的历史保持诚实。
- 重置到更早的点（*reset*）——你想把项目倒回到好几次提交之前的样子。

```
# 撤销所有未提交的改动，回到你的上一次提交
git restore .

# 安全地撤销某一次糟糕的提交（会创建一个新的“撤销”提交）
git revert <commit-id>

# 倒回到更早的提交（很强大——使用前请让 AI 解释清楚）
git reset --hard <commit-id>
```

提醒一句: git reset --hard 会永久丢弃工作。它有时是正确的工具，但是要小心对待的那一个。拿不准时，在运行前问问 AI”这条命令对我的文件到底会做什么？“。

远程与 *GitHub*: 云端里的你的项目

到目前为止的一切都发生在你自己的电脑上。*GitHub* 是一个把你的 repo 副本存到云端的网站。那份副本叫做远程 (remote)。

何必如此？三个理由：它是你笔记本报废时的备份，是你与别人（或 AI 工具和部署服务）共享代码的方式，也是协作发生的地方。

你把本地 repo 与远程关联一次，然后做 *push*（把你的提交送上 GitHub）和 *pull*（把 GitHub 上的提交取下来）。

把它想成你项目的两份副本——一份在笔记本上，一份在云端——靠 push 和 pull 保持同步：



```
# 把你的本地 repo 连接到一个 GitHub repo (只需一次)
git remote add origin https://github.com/yourname/yourproject.git

# 把你的提交送上 GitHub
git push origin main

# 取下 GitHub 上有、而你电脑上没有的提交
git pull origin main
```

把 push 想成“存到云端”，把 pull 想成“从云端取最新”。

拉取请求与议题

两个你很快会遇到的 GitHub 功能。

拉取请求 (pull request, PR) 是把一个分支合并进另一个分支的提议，并内置了一处空间，让你在改动并入之前先审查它们。哪怕独自工作，PR 也有用：它给你一块干净的屏幕，展示每一处改动，让你在合并进 main 之前审查 AI 的工作。这就是

diff 习惯的正式化。

议题 (issue) 是关于某件待办或待修之事——一个 bug、一个功能点子、一项 TODO——的追踪笔记。议题让你把待办清单贴在项目本身上，而不是散落在你脑子里。

.gitignore 与绝不提交机密

有些文件绝对不该进 Git。最要紧的几类：

- **机密**——API（应用程序编程接口）密钥、密码、数据库凭据。一旦你把机密提交到 GitHub，就当它已经泄露，哪怕你之后删掉它。历史会记得。
- **垃圾**——临时文件、下载下来的依赖（比如 node_modules）、构建产物。又大又没必要追踪。

你用一个叫 .gitignore 的文件来控制这件事。你在里面列出一些模式，Git 就当那些文件不存在。这对 vibe coder 至关重要，因为 AI 工具特别爱创建带着真实密钥的配置文件，而一不小心就会把它提交上去。

```
# 一个简单的 .gitignore 文件
.env
node_modules/
*.log
```

.env 那一行最要紧——机密通常就藏在那里。在你第一次提交之前就加上 .gitignore，并养成问 AI 的习惯：“这次提交里有没有什么是不该公开的？”

你的第一个 repo，从头到尾

这里把整个循环放在一处。新建一个文件夹，把它变成 repo，保存你的工作，再把它 push 到 GitHub：

```
# 1. 开一个新项目
mkdir my-first-project
cd my-first-project
git init

# 2. 在做任何事之前先保护好你的机密
echo ".env" > .gitignore

# 3. 做你的第一个存档点
git add .
git commit -m "Initial commit"

# 4. 连接到 GitHub 并把它推上去
git remote add origin https://github.com/yourname/my-first-project.git
git push -u origin main
```

就这样。你现在有了一个既有安全网、又有云端备份的项目。每次取得进展，就重复 add □ commit □ push 的节奏。

让 AI 来开车（而你仍掌舵）

这里有让人松一口气的部分：你完全不必背这些命令。你可以对 AI 说“用一句合适的说明把这个提交了”或者“为新功能建个分支并切过去”，它就会运行正确的命令。这也是一种完全不错的工作方式。

但陷阱在于——和本书其余部分是同一个陷阱——即便不是你亲手敲的，理解它做了什么的责任仍然在你身上。真正行得通的中间路线：

- 让 AI 来运行命令，但头几次请让它用大白话告诉你每条命令做什么。
- **diff 自己读**，任何一次提交之前都要读。这一条没得商量。这是你抓住 AI 删掉不该删之物的方式。
- **频繁地、小块地提交**。许多小存档点胜过一个巨大的，因为倒回时是精准的，而不是全有或全无。

- 干大事之前先建分支。在大刀阔斧重写之前让 AI 先建个分支，好让 main 保持安全。
- 当一条命令听起来有破坏性时（任何带 `reset --hard`、强制 `push` 或删除分支的），在批准之前先问“这到底会做什么？”。

你是技术负责人。AI 是你那快速、能干、偶尔鲁莽的队友。Git 就是那套让这位队友造不成真正损害的流程。把它学到足以自信指挥的程度，你就再也不会害怕那个“全部重写”的按钮了——因为你永远有一条回头的路。

回顾与练习

关键点

- Git 就是项目的存档点——一条你随时可以跳回去的完整时间线，正是它让你能够毫无顾虑地放手让 AI 去干。
- 核心节奏是 `add` → `commit` → `push`：暂存你的改动，用一条清晰的消息把它存为一次 `commit`，再推送到 GitHub 以备份和分享。
- 任何大动作之前先开分支，每次提交之前先读 *diff*——这是你抓住 AI 删掉不该删的东西的方法。
- `.gitignore` 把机密（如 `.env`）和垃圾（如 `node_modules/`）挡在历史之外。提交到 GitHub 的机密即便事后删除，也视为永久泄露。
- 你可以让 AI 来运行命令，但理解每条命令做了什么的责任仍在你身上——尤其是 `reset --hard` 这类破坏性命令。

动手试试

新建一个全新的文件夹，亲手把整个流程跑一遍：`git init`，创建一个包含 `.env` 的 `.gitignore`，添加一个简单的 `notes.txt` 文件，然后 `git add .` 和 `git commit -m "Initial commit"`。现在运行 `git log --oneline` 看看你的存档点，编辑这个文件，再运行 `git diff`，看 Git 把改了些什么精确地展示给你。你刚刚在一个绝不会出错的项目上，用过了本章所有的核心习惯。

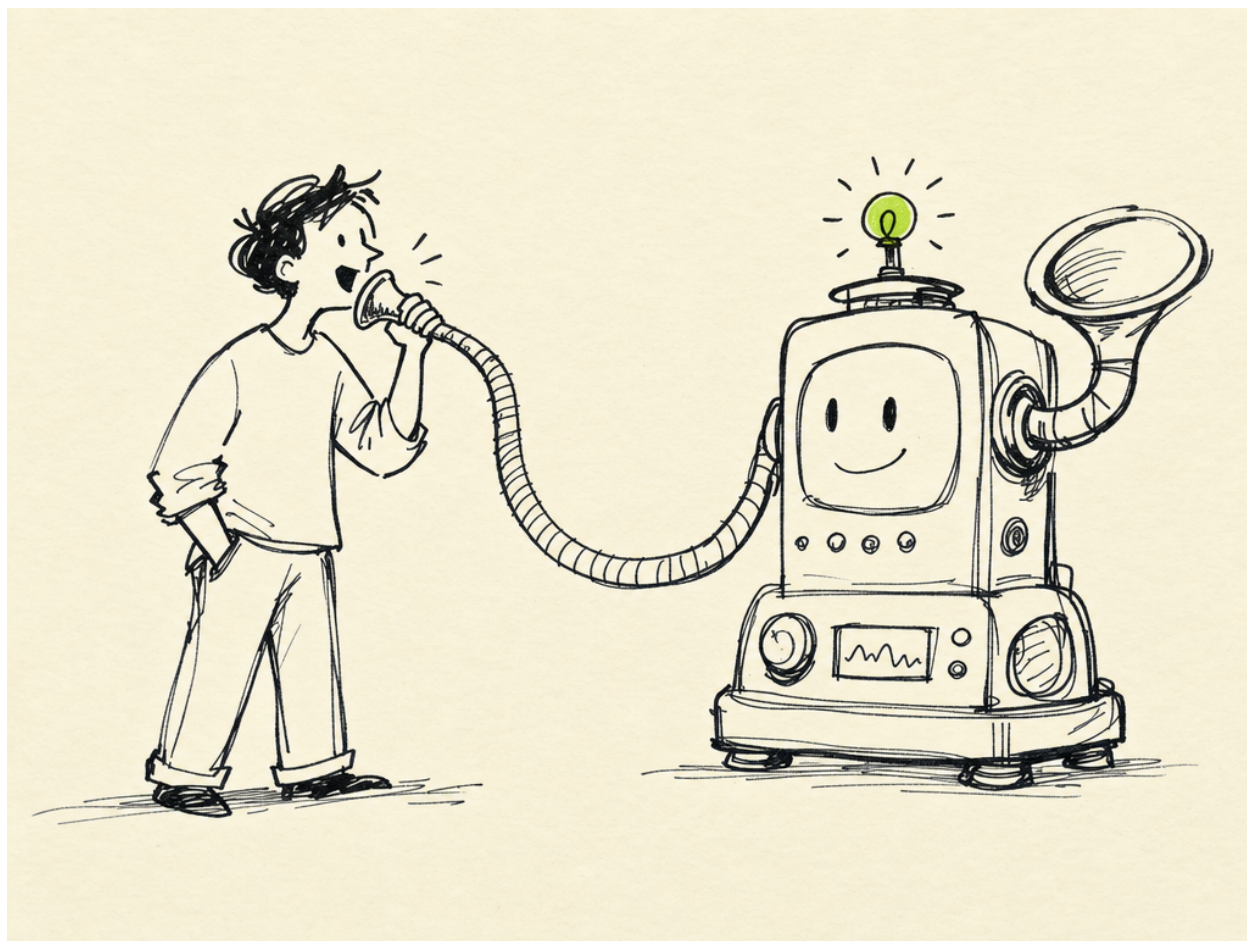
本章提示词

我是刚开始学习 Git 的新手，希望你替我运行命令，但在过程中教我。对于下面这个任务：

< 描述你想做的事 一例如" 保存我当前的工作" 或" 安全地尝试一个有风险的改动">

- 告诉我你将运行哪些 Git 命令，并用通俗的话说明每条命令的作用。
- 如果有破坏性操作（`reset --hard`、强制 `push`、删除分支），请在执行前清楚地警告我，并说明我会失去什么。
- 在任何提交之前，给我看 `diff` 并总结改动了什么。
- 如果改动中有不该公开的内容（秘密、密钥），提醒我。

The Command Line



第一次看着 AI 编程工具干活时，一个黑色窗口里满是向上滚动的文字，一个闪烁的光标等着你输入。那个窗口就是命令行。对许多没写过代码的人来说，这是整段旅程里最吓人的部分——它看起来就像电影里那样，仿佛一步走错就会抹掉你的整块硬盘。不会的。命令行只是告诉电脑该做什么的另一种方式：用文字，而不是点击。

本章要消除这份恐惧。你不需要背下一百个命令。你需要理解正在发生什么，认出少数几个每天都会用到的命令，并知道如何安全地运行 AI 递给你的命令。这些就足以让你不再害怕，开始真正高效地干活。

终端和 *shell* 究竟是什么

用普通应用时，你点击按钮，应用把你的点击翻译成给电脑的指令。命令行跳过了按钮。你输入一条指令，按下回车，电脑就去执行它。

有两个词你会经常听到：

- **终端 (*Terminal*)** ——你输入文字的那个窗口。它只是一个带历史记录文本框。（在 Mac 上叫“Terminal”或“iTerm”；在 Windows 上叫“PowerShell”或“Windows Terminal”；在 Linux 上通常就叫“Terminal”。）
- ***shell*** ——那个窗口里面读取你输入并运行它的程序。最常见的一个叫 ***bash***；Mac 用的是一个叫 ***zsh*** 的。就本书涉及的一切而言，两者的行为几乎完全相同。

每次你按下回车，循环就是这样的：**shell** 读取你输入的内容，找到并运行匹配的程序，再把那个程序产出的任何东西打印回窗口里。





AI 编程工具为什么住在这里？因为命令行是开发的通用控制台。安装工具、启动应用、运行测试、部署到互联网——所有这些都通过输入的命令发生。AI 智能体的工作方式就是输入与你会输入的相同命令，只是更快、没有拼写错误。当你看着它干活时，你看到的正是它在使用那扇你也将使用的门。

读懂提示符

在你输入任何东西之前，shell 会给你显示一个提示符 (**prompt**) ——一小段以某个符号结尾的文字，等待输入。它可能长这样：

```
mini@laptop ~/projects/my-app $
```

从左到右读，它告诉你：你的用户名 (mini)、你的电脑 (laptop)、你当前所在的位置 (~/projects/my-app 这个文件夹)，然后是表示“轮到你了”的 \$。~ 是你主文件夹的简写。结尾的符号通常是 \$ 或 %，有时是 >。别去输入那个符号——它只是 shell 在说它准备好了。

这比看起来更重要。命令行里的大多数困惑都来自不知道自己在哪里。提示符每一次都在告诉你。

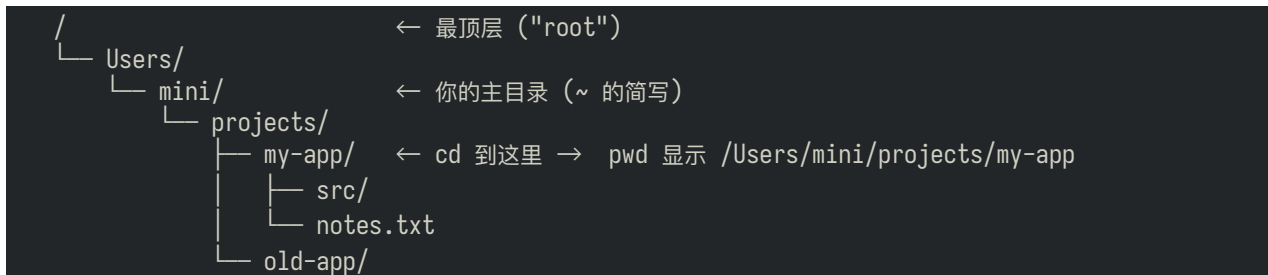
每天都会用到的命令

这里是你会不断看到的命令。把它们当句子来读：一个动词，有时后面跟着一个对象。

```
pwd          # "print working directory" ——我现在在哪里？
ls           # 列出这里的文件和文件夹
ls -la       # 列出所有内容，包括隐藏文件，并带详情
cd projects  # change directory ——进入 "projects" 文件夹
cd ..        # 向上一层文件夹 (".." 表示"上一层文件夹")
cd ~         # 回到你的主文件夹
mkdir my-app # 新建一个叫 "my-app" 的文件夹
cp notes.txt backup.txt # 复制一个文件（原件留着，做出一份副本）
mv old.txt new.txt      # 移动或重命名一个文件
rm draft.txt           # 删除一个文件 (remove)
```

几条能省去真实麻烦的提醒：

你的文件住在一棵层层嵌套的文件夹树里，而 cd 就是你在其中行走的方式。这里有一棵小树，pwd 显示了你 cd 进 my-app 后会站在哪里：



- cd 是你四处移动的方式。配上 pwd 和 ls，这就是你这辈子要做的事的 80%。你进入正确的文件夹，看一看，然后运行点什么。
- 路径可以是相对的，也可以是绝对的。cd projects 是相对于你所在位置的；cd /Users/mini/projects 是从最顶端算起的完整地址。

- **rm 是永久的。**命令行里没有回收站。你 rm 掉一个文件，它就没了——没有“你确定吗？”，也没有撤销。对 `rm -rf` 要格外小心，它会不问一声就删掉一个文件夹及其中的一切。如果你对一条 `rm` 命令会删掉什么哪怕有一点不确定，就别运行它。先请 AI 解释它。

运行一个程序和一个 dev 服务器

做软件意味着运行它。大多数时候你会输入一个程序的名字，后面跟上你想让它做的事。

```
node app.js      # 用 Node 运行一个 JavaScript 文件
python script.py # 运行一个 Python 文件
npm run dev      # 启动项目的开发服务器
```

最后那条很特别。**dev 服务器**是一个程序，它在你自己的电脑上运行你的应用，让你能在浏览器里通过 `http://localhost:3000` 这样的地址看到它。要理解的关键是：它会一直运行。终端看起来会“卡住”——它不再给你新的提示符，而是显示一连串的活动信息。那不是死机，那是在干活。只要那条命令在运行，你的应用就活着。打开浏览器，访问它打印出的地址，你就会看到你的应用。

干完之后，你把它停掉（下一节讲）。

读懂输出和 `exit` 码

命令运行完之后，通常会打印点什么。学会读懂那段输出是一项真本事，因为这正是你和 AI 弄清事情是否成功的方式。

- **正常的输出**只是信息——列出的文件、进度消息、服务器说它已启动。
- **错误**通常会直白地说出来：`Error:`、`command not found`、`permission denied`、`cannot find module`。冒号后面的词就是你的线索。把整段错误复制下来贴给你的 AI——这是出问题时你能递过去的最有用的一样东西。

在幕后，每条命令结束时都带着一个看不见的 **exit 码**：0 表示成功，其他任何值都表示有问题。你很少需要手动去查它，但你会听到有人提起。最近一次的可以这样看：

```
echo $?      # 打印上一条命令的 exit 码 (0 = 正常)
```

如果一条命令悄无声息地失败了而你不确定，`echo $?` 会告诉你它到底成没成功。

用 **Ctrl-C** 停掉一个进程

那个 dev 服务器会一直运行，直到你停掉它。通用的“停止”是 **Ctrl-C**——按住 Control 键再按 C。它向正在运行的任何东西发出一个礼貌的“现在请停下”信号，然后你就拿回了提示符。

- **Ctrl-C**——停掉这个终端里当前正在运行的程序。结束一个 dev 服务器，或取消一条耗时过长的命令时，这是你的首选。
- 关闭终端窗口也会停掉其中运行的一切，但 Ctrl-C 更干净利落。

记住 Ctrl-C，你就再也不会觉得被一个“卡住”的终端困住了。

PATH 与“`command not found`”

迟早你会输入一条命令，却得到 `command not found`。这几乎从不意味着你笨；它意味着电脑不知道那个程序住在哪里。

shell 通过查看一个叫 **PATH** 的文件夹清单来找程序。当你输入 `npm` 时，shell 会走遍 PATH 里的每个文件夹，找一个叫 `npm` 的程序。如果哪个文件夹里都没有，你就会得到 `command not found`。

你输入： `npm`



按顺序在 PATH 各目录中查找 "npm":
/usr/local/bin → 不在这里

```
/usr/bin    -> 不在此处
/opt/bin    -> 找到  ✓  -> 运行它
```

```
若所有目录都没有 -> "command not found"
```

```
echo $PATH      # 显示 shell 会搜索的文件夹清单
```

实际上 `command not found` 通常意味着两件事之一：那个工具还没装，或者它被装在了一个不在 `PATH` 上的地方。两者都很常见，也都能修。把错误贴给你的 AI，说出你刚才想运行什么——修 `PATH` 问题，正是 AI 擅长一步步带你做的那种繁琐的配置活儿。

安全地运行 AI 给你的命令

vibe coding 很大一部分就是 AI 说“运行这条命令”，递给你一行让你粘贴。这很正常，也没问题。但请养成一个会永远保护你的习惯：**运行之前先读**。

你不需要理解每个参数。你需要对这条命令做什么有个大致的感觉，尤其是开头的那个动词：

- `ls`、`pwd`、`cd`、`cat`、`echo`——无害。它们只是看看东西或四处移动。尽管运行。
- `npm install`、`pip install`——它们下载并配置工具。一般是安全的，但它们会改动你的项目。
- `rm`、`mv`、任何带 `sudo` 的、任何把下载直接灌进 `shell` 的 (`curl ... | bash`) ——慢下来。这些可能删掉东西，或以对你机器的完全权限运行代码。

如果一条命令看着吓人，或者你认不出那个动词，就问：「这条命令是做什么的，运行它安全吗？」一个好的 AI 会把它讲得明明白白。没有这道检查，绝不要粘贴来自某个随便网站的命令。命令行会丝毫不差地照吩咐执行——包括那些危险的事——所以先读这个小习惯，就是你的安全带。

你会遇到的包管理器

现代软件是由成千上万个预先写好的小零件搭起来的。**包管理器 (package manager)** 就是下载并整理这些零件的工具。你会不断遇到两个：

```
npm install      # 安装这个 JavaScript 项目需要的一切
npm install react # 添加一个叫 "react" 的特定包
pip install requests # 安装一个叫 "requests" 的 Python 包
```

- **npm** (Node 包管理器) 是 JavaScript 和 Node 项目的包管理器。`npm install` (常写成 `npm i`) 会读取一个列出项目所需内容的文件，并把它们全部下载下来。
- **pip** 为 Python 做同样的事。

第一次打开一个项目时常常会跑一次 `npm install`，之后就把它抛在脑后了。当 AI 添加一个需要新包的功能时，它会让你去安装它。现在你知道那是什么意思了：只是去取来一块 **building block**，没什么可怕的。

这就是命令行。不是黑客的武器——只是一个你怎么吩咐它就怎么照做的文本框。学会读懂提示符、认出十来个命令、用 `Ctrl-C` 停掉失控的进程、运行之前先读。其余的一切，你都可以问。

回顾与练习

关键点

- 命令行只是一个完全照你说的去做的文本框——用文字代替点击。AI 智能体的工作方式，就是敲下你本来也会敲的那些命令。
- 提示符每次都告诉你你现在在哪里；大部分困惑都源于不知道当前所在的文件夹。`pwd`、`ls`、`cd` 占了你日常操作的 80%。
- `rm` 是永久性的——没有回收站，没有撤销。遇到 `rm`、`sudo`，以及任何把下载内容直接灌进 `shell` 的命令 (`curl ... | bash`) 时，请放慢脚步。

- **开发服务器**会一直运行，并且故意看起来像“卡住”了——那正是它在工作。**Ctrl-C** 能停止任何正在运行的程序，把提示符还给你。
- **先读再运行**。你不必懂每个参数，只需对开头的动词有个大致感觉。拿不准时，问 AI”这条命令是做什么的，运行它安全吗？”

动手试试

打开终端，在不改动任何东西的前提下探索：运行 `pwd` 看看你在哪里，`ls` 看看这里有什么，用 `cd` 进入一个文件夹再 `ls` 一次，然后 `cd ..` 退回去。注意提示符每次是如何更新的。现在故意运行一条会失败的命令——敲 `lss`（一个拼写错误）并读一读 `command not found` 这条消息。你刚刚练习了两条最安全、最常用的命令，还学会了读错误信息，全程没碰任何一个文件。

本章提示词

我是命令行新手，对运行各种东西有点紧张。

这是你（或某个教程）让我运行的一条命令：

< 在此粘贴命令 >

- 请用通俗的语言一步一步解释这条命令做什么。
- 明确告诉我它是否安全，或者它是否可能删除文件、更改我的系统，或以完整权限运行代码。
- 如果有风险，请建议一种更安全的做法，或者我应该先检查什么。
- 告诉我运行它之前应该处在哪个文件夹里。

How the Web Works



在为网络构建任何东西之前，先了解网络实际上是什么会很有帮助。你不需要计算机科学学位，也不需要背诵任何东西。你只需要一个能用的心智模型——一幅在你脑中描绘出的画面：当你输入一个地址、一个页面随之出现时，到底发生了什么。

本章就来构建这幅画面。从登录表单到支付，一个用氛围编码（vibe coding）做出来的应用所做的一切，归根结底都是同一支简单舞蹈的某种版本：一台计算机发问，另一台计算机作答。一旦你能清晰地看到这支舞，本书的其余部分就不再像魔法，而开始像管道工程。

两个角色：客户端与服务器

网络上几乎所有东西，都是两个角色之间的一场对话。

- **客户端**是发问的那一方。多数时候那就是运行在你手机或笔记本上的网页浏览器——Chrome、Safari、Firefox。它就像走到柜台前的顾客。
- **服务器**是作答的那一方。它是坐落在某处数据中心里的一台计算机，整天运行着，等待着请求。它就像柜台后面的厨房。

客户端永远不做服务器的活，服务器也永远不做客户端的活。浏览器问：“把首页给我。”服务器答：“给你。”整个关系就是这样。服务器不是什么特殊的魔法盒子——它只是一台普通的计算机，其工作就是等待问题、送回答案。

还有一个你会不断听到的词：**request**（请求）（客户端发出的问题）与 **response**（响应）（服务器送回的答案）。把这两个词握在手边；它们是下面一切的脊梁。

URL 实际上是什么

URL（统一资源定位符）就是你所请求的地址。它看起来像一长串字符，但其实是几个带标签的部分缝接而成。把它拆开来看：

https://shop.example.com/products/shoes?size=10

https	scheme	— how to talk (use HTTPS, the secure way)
shop.example.com	host	— which server to talk to
/products/shoes	path	— which page or thing you want from it
?size=10	query	— extra details ("the size-10 ones")

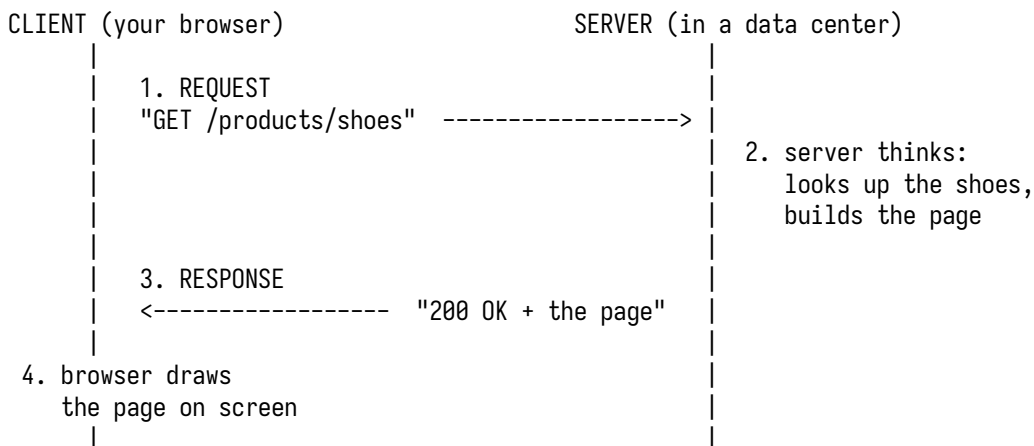
像读一个倒过来的邮政地址那样读它：

- **scheme** (https) 是协议——对话的规则。如今几乎总是 https。
- **host** (shop.example.com) 指明要联系哪一台服务器。
- **path** (/products/shoes) 指明你想要那台服务器上的哪样东西。
- 可选的 **query** (?size=10) 携带额外参数，比如筛选条件或搜索词。

当有人说“端点”或“路由”时，他们几乎总是指某台特定 **host** 上某个特定 **path**。那些吓人的词的全部含义也就是这些。

请求与响应的循环

这就是那支舞，从头到尾。你输入一个地址，幕后会在不到一秒内发生这样的事：



每一次都是四个步骤：

1. 客户端向某个 URL 发送一个 **request**。
2. 服务器**处理**它——也许它会查找些什么、运行一些逻辑、核实你是谁。
3. 服务器送回一个 **response**：一个状态（成功了吗？）外加一些内容（页面、图片或数据）。
4. 客户端拿这个响应做点什么——通常是把它画到你的屏幕上。

加载一个网页并不是一个请求——而是几十个。浏览器先请求页面，接着发现它还需要图片、字体和样式，于是为每一样都射出一个新的请求。同一支舞，许多遍，非常快。

HTTP 方法：动词们

每个请求都带着一个 **方法**——一个说明你想要哪种动作的动词。方法有好几个，但四个就涵盖了几乎一切，而且它们干净利落地对应到日常的观念上：

- **GET** ——“把这个给我。”读取、取回、查看。加载一个页面或一份列表。GET 永远不应该改变任何东西；它只是取回。
- **POST** ——“这里有个新东西。”创建。提交表单、注册、发表评论。
- **PUT** ——“更新这个。”修改某个已经存在的东西，比如编辑你的资料。（对于部分更新你还会听到 **PATCH**——同样的思路，范围更小。）
- **DELETE** ——“移除这个。”就是字面意思——删掉一条记录或一个文件。

别的都忘了也没关系，记住这个：**GET** 读取，**POST** 创建，**PUT** 更新，**DELETE** 移除。当你的应用出毛病时，知道发出的是哪个动词，往往是第一条线索。

状态码：成功了吗？

每个响应回来时都带着一个三位数的 **状态码**——服务器对这次请求结果的一眼判定。它们按首位数字分组：

- **2xx** —— **成功**。成功了。200 OK 是日常那句“喏，这是你要的东西”。
- **3xx** —— **重定向**。“这儿没有，去那边看看吧。”常见且无害；浏览器会自动跟过去。
- **4xx** —— **是你（客户端）出了错**。最有名的是 **404 Not Found**——你请求了一个不存在的 path。（还有 401/403——你没登录，或没被允许。）
- **5xx** —— **是服务器出了错**。**500 Internal Server Error** 意味着服务器在处理你的请求时，它自己的代码崩了或坏了。

4xx/5xx 的区分值得刻在脑子里，因为它告诉你该往哪儿看：

- **404** 意味着请求错了——通常是个糟糕的 URL 或打错的 path。看看到底在请求什么。
- **500** 意味着服务器倒下了。你的请求没问题；是后端代码坏了。去看服务器日志，而不是 URL。

当一个用氛围编码做出来的应用出了岔子，“这是 4xx 还是 5xx？”是最快能用来分诊的问题之一。

状态码				
2xx	—▶	成功	成功了（200 OK）	
3xx	—▶	重定向	" 去那边看看 "（自动跟随）	
4xx	—▶	客户端错误	你的请求有误	—▶ 检查 URL/path
5xx	—▶	服务器错误	后端崩溃了	—▶ 检查服务器日志

头部与 Cookie

与正文内容并排，每个请求和响应都带着 **头部 (headers)** ——一张张贴着标签、装着额外信息的小纸条。它们就像信封上写的字，跟里面的信是分开的。头部会说明诸如内容是什么格式 (Content-Type: text/html)，或者是谁在发问。

Cookie 是头部的一种具体而重要的用法。网络的记性很短：默认情况下，每个请求都是孤立的，服务器一答完就把你忘了。Cookie 修补了这一点。你登录之后，服务器递给你的浏览器一个小小的令牌——一个 **cookie**——而你的浏览器会在之后的每一个请求里自动把它带上。这就是网站如何跨页面“记住”你已登录的方式。没有 cookie，就没有记忆；你每点一下都得重新登录一次。

HTTPS 与那把锁

你见过地址栏里那把小锁。它意味着这条连接正在使用 **HTTPS**（超文本传输安全协议）——被一层叫做 **TLS** 的加密包裹起来的 HTTP（超文本传输协议）。

朴素的 HTTP 把一切都以可读的文本发送。坐在你和服务器之间的任何人——比如在公用 Wi-Fi 上——都能在你的密码经过时把它读走，就像邮路上人人可读的明信片。HTTPS 把这场对话封进一个只有你和服务器才能打开的信封里。那把锁确认了两件事：

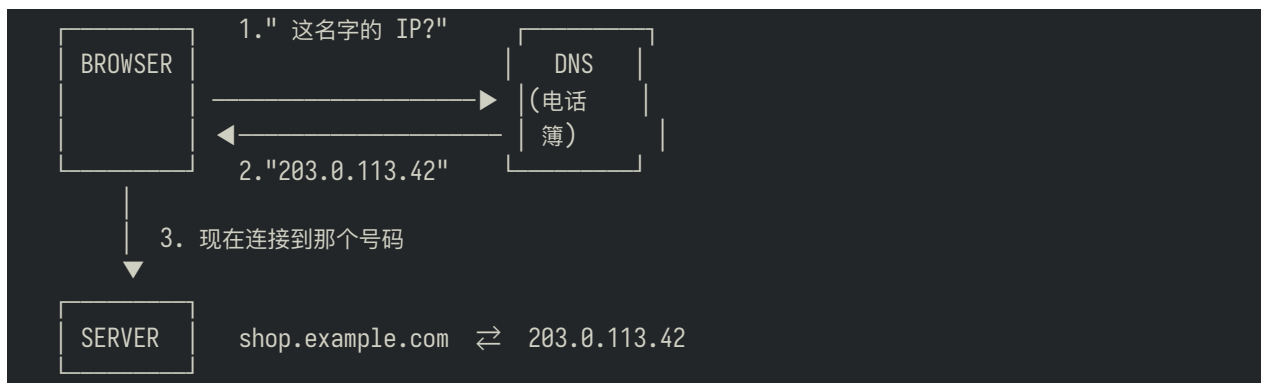
- **隐私**——中间的任何人都读不到你发送的内容。
- **身份**——你确实跟你以为的那台服务器对话，而不是冒名顶替者。

实用的规则很短：真正的应用使用 HTTPS，永远，毫无例外。任何用朴素 HTTP 处理登录或支付的东西，从设计上就是坏的。

从名字到服务器：DNS

还有一根没收尾的线头。你输入 shop.example.com，但计算机其实并不靠名字找到彼此——它们用像 203.0.113.42 这样的数字 **IP 地址**。**DNS** (Domain Name System, 域名系统) 是网络的电话簿：在你的浏览器能发出请求之前，它会悄

悄地请求 DNS 把那个友好的 域名 翻译成数字地址，然后连接过去。



你买下一个域名，把它的 DNS 记录指向运行你应用的那台服务器，从此这个名字就把人们引向你的机器。这是一个你几乎永远看不到的查询步骤，但一个好记的名字之所以能替代一串数字，正是因为它。

这一切如何对应到你的应用

现在，把这整幅画面铺到一个用氛围编码做出来的应用之上，因为你将要构建的每一块，都住在这支舞的某一侧：

- **前端**是运行在浏览器里的一切——用户看到并点击的按钮、表单和布局。它就是客户端。它主要负责发送请求、显示响应。
- **后端**是运行在服务器上的你的代码——决定如何处理一个请求、与数据库对话、并组装出响应的那套逻辑。它就是服务器。

把整个注册之旅放进一幅图里就是这样——本章每一个加粗的词都各就各位：



一次真实的交互会把这一切缝在一起。用户点击“注册”（前端）。浏览器带着表单数据，向一个像 `/api/signup` 这样的 path、通过 **HTTPS** 发送一个 **POST** 请求。你的后端收到它，创建账户，设置一个 **cookie** 让用户保持登录，然后送回一个 **200**。前端看到成功，于是展示一个欢迎页面。那里每一个加粗的词，都是你刚刚学过的东西——而那，就是你将要构建的几乎一切的整体形状。

你不必把这一切一次性全都装在脑子里。但当后面某一章说“请求打到了服务器，却以 500 返回”，或者“设置这个头部”，又或者“这个路由处理那个 POST”时，你现在就能确切地知道它指的是这支舞的哪一部分。那个心智模型，才是真正的先决条件——而你已经拥有它了。

回顾与练习

关键点

- 网络就是一支不断重复的简单舞蹈：**客户端**（通常是浏览器）发出**请求**，**服务器**返回**响应**。其余的一切都只是叠加在这之上的细节。
- **URL** 是带标签的几个部分拼接而成——scheme、host、path、query。方法就是动词：**GET** 读取，**POST** 创建，**PUT** 更新，**DELETE** 删除。
- **状态码**是服务器一眼给出的判定。4xx/5xx 之分告诉你该往哪儿看：**4xx** 意味着请求错了（URL 不对），**5xx** 意味着服务器自己的代码出了问题。

- **HTTPS** (小锁) 把对话封起来, 让中间任何人都读不到——真正的应用始终使用它。**Cookie** 是网站跨多个请求记住你已登录的方式。
- 你的**前端**是客户端 (在浏览器里运行), 你的**后端**是服务器 (运行你的逻辑并与数据库对话)。你构建的每个功能都处在这支舞蹈的某一侧。

动手试试

打开任意一个网站, 然后打开浏览器的开发者工具 (右键 □“检查”, 再切到“Network/网络”标签)。刷新页面。看着请求列表被填满——你会看到一个页面请求, 紧接着是为图片、字体和样式发出的几十个请求, 正如前文所述。点开其中一个, 看它的方法 (GET)、状态码 (但愿是 200) 和标头。你现在正实时看着请求/响应这支舞蹈上演, 而且具备了读懂它的词汇。

本章提示词

我是初学者, 正在学习网络是如何运作的。用我自己的应用作为例子:

< 描述你的应用, 或粘贴一个 URL 或你看到的错误 >

- 针对一个具体的操作 (比如加载页面或提交表单), 一步一步带我走一遍请求/响应的循环。
- 告诉我哪一部分是客户端 (前端), 哪一部分是服务器 (后端)。
- 如果我看到一个状态码或错误, 请解释它是 4xx (我的请求有误) 还是 5xx (服务器崩溃了), 以及该去哪里查看。
- 保持具体且对初学者友好 ——不假设任何已有知识。

API、JSON 与数据



你做的应用几乎不可能孤立存在。只要你想显示一张地图、扣一笔卡、发一封邮件，或者查一下天气，你的应用就得去和别人的应用对话。它说的这门语言叫 **API**（应用程序编程接口），而它们来回传递的话语，几乎总是写成一种叫 **JSON**（JavaScript 对象表示法）的格式。这一章讲的就是这两样东西——不是为了让你亲手去写这些管道，而是为了让你能读懂它、能检查 AI 搭出来的东西是否靠谱、能在它害你花冤枉钱之前就发现哪里出了问题。

这里没有任何东西需要你死记硬背。你需要的是认出这些“形状”。当 AI 说“我会调用 Stripe API 并解析 JSON 响应”时，你应该能大致想象出这是什么意思、以及哪里可能会出岔子。

API 究竟是什么

理解 API 最干净的方式，就是把它看成一个程序向另一个程序提供的一份请求菜单。

餐厅不会让你直接走进厨房自己做菜。它给你一份菜单：这是你能点的东西，这是点菜的方式，这是你会拿到的东西。你不需要知道厨房是怎么运作的，你只需要这份菜单。API 正是如此——一家公司说“这是你被允许向我们服务发出的请求，这是每个请求会返回的东西。”

当 AI 写出“调用天气 API”的代码时，它就是在照着那份菜单点单：把这个城市的天气预报给我。天气公司的服务器干活，然后把答案递回来。你的应用永远看不到他们的数据库或代码，它只拿到那道菜。



最重要的思维转变是：API 是一份**契约**。对方公司承诺，只要你按特定方式提问，就会得到特定形状的答案。你的工作——也是 AI 的工作——就是正确地提问、并正确地读懂答案。大多数 API 的 bug，都是某一方违背了这份契约。

用大白话讲端点和 REST

一个端点 (**endpoint**) 就是菜单上的一道菜。它就是一个做某件具体事情的 URL（统一资源定位符）。比如：

- <https://api.weather.com/forecast> —— 获取一份天气预报
- <https://api.weather.com/cities> —— 列出可用的城市

同一个厨房，不同的菜。每个端点就是一项能力。

你会接触到的大多数 API，都遵循一种宽松的风格，叫 **REST**（表述性状态转移）。你不需要学术定义，你只需要两个想法：

- **URL** 给一个东西命名（一个“资源”）——一个用户、一个订单、一份天气预报。
- **动词说明你想拿它做什么**。这些动词就是 HTTP（超文本传输协议）方法，和“网络是怎么运作的”那章里讲的是同一批：
 - GET —— 读取某个东西（取一份天气预报）。安全；什么都不改变。
 - POST —— 创建某个东西（下一个订单、注册一个用户）。
 - PUT / PATCH —— 更新某个已经存在的东西。
 - DELETE —— 删除某个东西。

所以 GET /orders/42 的意思是“读取 42 号订单”，而 DELETE /orders/42 的意思是“删掉它”。同一个名词，不同的动词，效果完全不同。当你读 API 文档或一段 diff 时，那两个词——**方法**和 **URL**——就告诉了你一个请求所做的大部分事情。

JSON 是什么

当你的应用向 API 提一个问题时，答案会以一种叫 **JSON** 格式的文本回来。它之所以成了默认选择，是因为它既能被人读懂，也容易被程序处理。你会不停地见到它，所以能读懂它很值得。

JSON 由**键**和**值**构成。键是一个带引号的标签；值是附在它上面的数据。下面是一个真实的例子，描述了一个用户：

```
{
  "id": 42,
  "name": "Ada Lovelace",
  "email": "ada@example.com",
  "isAdmin": false,
  "loginCount": 137,
  "tags": ["beta", "early-access"],
  "profile": {
    "city": "London",
    "avatarUrl": null
  }
}
```

把它当成一张有人填好的表格来读。"name" 是一个字段，"Ada Lovelace" 是他们填进去的内容。两种结构完成了所有的工作：

- 一个 **object** 用花括号 { } 包起来。它是一捆键/值对——就像一条记录。上面整个例子就是一个 object。
- 一个 **array** 用方括号 [] 包起来。它是一个有序的值列表。在上面，"tags" 装着一个由两个 string 组成的 array。

注意 "profile" 是一个 object 里面的 object。JSON 是可以嵌套的——一个值本身可以是一个 object 或一个 array，想嵌多深就嵌多深。这就是为什么一个响应可以一次性描述一个用户、他们的地址、以及他们最近的十个订单。

把同一个用户 object 画成一棵树，它的形状就一目了然——平坦的字段在上面，一个列表和一个嵌套的 object 从中分出枝杈：

```
{ } user (对象)
├── id ..... 42 (数字)
├── name ..... "Ada Lovelace" (字符串)
├── isAdmin ..... false (布尔值)
├── tags ..... [ ] (数组)
│   ├── "beta"
│   └── "early-access"
└── profile ..... { } (对象)
    ├── city ..... "London" (字符串)
    └── avatarUrl ..... null (null = 无值)
```

你会遇到的数据类型

JSON 里的每个值，都属于一小组类型中的一种。认识它们，能让你看出某个值的类型错了——这是一个常见的 bug 来源。

- **string** ——文本，总是带双引号："Ada Lovelace"、"ada@example.com"。
- **number** ——一个普通的数字，不带引号：42、3.14。(引号会改变含义："42" 是文本四二，不是数字。)
- **boolean** ——true 或 false，不带引号。用于像 "isAdmin" 这样的是/否标志。
- **null** ——一个刻意的“这里没有值”。"avatarUrl": null 表示这个用户没有头像——这和这个键根本不存在是两回事。
- **array** ——一个有序列表：["beta", "early-access"]。
- **object** ——一捆键/值对：那个 "profile" 块。

整套词汇就这些了。现实中最常见的错误是 string 与 number 的陷阱：代码期待的是数字 42，但 API 发来的是字符串 "42"，于是一次比较就悄无声息地失败了。当某个东西表现得很怪时，检查一个值的类型往往就是解药。

一个真实的请求和响应

下面是一次真实的 API 调用从头到尾的样子——你的应用发出去请求，以及它拿回来的 JSON：

```
GET /forecast?city=London HTTP/1.1
Host: api.weather.com
Authorization: Bearer sk_live_8f4b2c...
Accept: application/json
```

```
{
  "city": "London",
  "unit": "celsius",
  "current": {
    "temp": 14,
    "condition": "cloudy"
  },
  "forecast": [
    { "day": "Mon", "high": 16, "low": 9 },
    { "day": "Tue", "high": 18, "low": 11 }
  ]
}
```

那行 Authorization 就是下一个话题——也是最值得做对的那个。

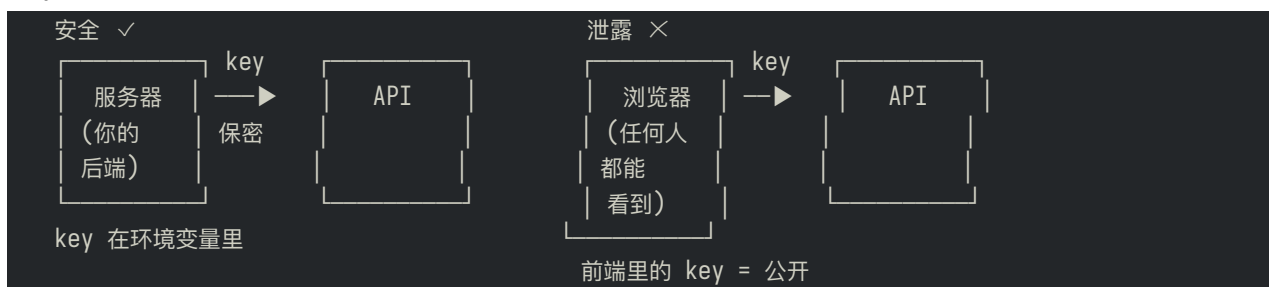
API key 与认证 header

大多数有用的 API 并不对全世界开放。公司需要知道是谁在提问，这样他们才能统计用量并向你收费。他们会给你一个 **API key**——一长串能标识你账户的密钥字符串。你把它附在每个请求上，通常放在一个叫 Authorization 的 **HTTP header** 里，就像上面那行 Bearer sk_live_...。

把 API key 完全当成密码来对待，因为它本来就是密码。任何拿到它的人都能以你的身份发请求——并刷爆你的账单或读走你的数据。这直接关联到安全那一章，而其中最重要的有一条规则：

- 绝不要把 **API key** 放进会发到浏览器里的代码。你前端里的任何东西都是公开的——用户可以打开开发者工具读到它。AI 被要求”从页面里调用 API”时，会乐呵呵地把你的密钥直接贴在那儿。拦住它。密钥属于服务器。
- 绝不要把 **key** 提交进 **git**。就算你之后删掉它，它也会永远活在历史记录里，而机器人会在几分钟内扫描公开仓库里的 key。
- 把 **key** 存在环境变量里，而不是代码本身里——这和安全那一章讲的是同一个模式。

key 必须从一个安全的地方出发，绝不能从浏览器出发。设想一下这两条路——一条安全，一条泄露：



如果你哪天看到一个真实的 key 出现在一段 diff 里、一个前端文件里、或一条聊天消息里，就把它当成已经泄露了：立刻轮换它（生成一个新的，停用旧的）。

速率限制与账单暴雷陷阱

还有两件菜单不会大声告诉你、但绝对会咬你一口的事。

速率限制 (Rate limits)。API 会限制你在一段时间窗口内能发多少请求——比如每分钟 100 次。超了你就会被拒绝，收到一个 429 Too Many Requests 错误，直到你慢下来。这个机制是为了防止某一个客户把服务压垮。陷阱在于：AI 写的代码

常常会在一个循环里不停地调 API、中间不停顿，瞬间就把限额冲爆。如果一个功能对单个条目能正常工作，但跑上一千个就崩了，速率限制就是头号嫌疑犯。

计量收费的 API。很多 API 按请求收费——每次几分之一美分。调几次不算什么，但当某个东西失控地循环时，就成了一笔真实的账单。经典的恐怖故事是：一个 bug 把一个失败的付费调用永远重试下去，或者一个公开的表单让陌生人能触发付费调用，于是你一觉醒来收到一张四位数的发票。保护好你自己：

- 在接入一个付费 API 之前先读定价。搞清楚每次调用的成本。
- 如果供应商的后台提供这功能，就设一个**消费上限或预算提醒**。大多数都提供。
- 对任何在循环里、或在每次页面加载时调用付费 API 的代码，都要心存警惕。

怎么读 API 文档

你会长期泡在 API 文档里，所以这里教你怎么不读全部就快速拿到你要的东西。对任何一个端点，去搜寻这五样东西：

- **方法和 URL**——GET /forecast。它做什么，两个词说清。
- **认证**——它需要什么 header 或 key。通常整个 API 共用一个章节。
- **参数**——你可以发的输入（这里是 city），哪些是必填的，以及它们的类型。
- **一个示例响应**——你会拿回的 JSON 形状。这是宝贝；把它复制下来。
- **错误**——出问题时会返回什么，以及状态码（401 未授权、404 未找到、429 触发速率限制）。

好的文档会包含一个可以直接复制粘贴的示例请求和响应。这两段代码片段，比几段文字告诉你的还要多——把它们抓过来，交给 AI。

让 AI 去做调用——而你来检查形状

这里是实际的分工。AI 真的很擅长写调用 API 的代码：它懂那些库、懂 header 的语法、懂错误处理。让它来。这些管道不用你亲手写。

但有一件事是你的，而 AI 很不擅长：**判断数据的形状到底对不对**。一个好的 workflow：

- **把真实的文档给 AI。**把文档里的示例请求和响应粘进你的提示里。AI 凭记忆猜一个 API 的形状，是头号 bug 来源——它会编出一些看起来很合理、但根本不存在的键。真实的例子能把它锚定在现实上。
- **让它先把原始响应给你看。**在解析之前，把 API 实际返回的 JSON 打印出来。“然后把它和代码所期待的对照一眼。它读取的那个键（`response.current.temp`）在响应里真的存在吗？一半的 API bug 都是键名对不上。
- **检查一下类型是否靠谱。**那是个 number 还是 string？一个本应总是存在的字段，会不会有时候是 null？你现在能读懂 JSON 了——用上它。
- **处理不顺利的那条路。**问一句：“如果这次调用失败了、超时了、或者撞上速率限制，会发生什么？”如果答案是“应用崩溃”，那是个 bug，不是特性。

你不需要写那个请求。你需要的是读懂响应、并问一句“这是我期待的形状吗？”。这一个习惯——拿数据去对照契约——就能在你的用户看到之前，抓住大部分集成 bug。

回顾与练习

关键点

- **API** 是一个程序向另一个程序提供的请求菜单，也是一份**契约**：按约定的方式提问，就会得到约定形状的回答。大多数 API 缺陷，都是某一方破坏了这份契约。
- **JSON** 是回答返回时的格式——由键和值构成，用对象 { } 和数组 [] 搭建，并且会嵌套。数据类型有 string、number、boolean、null、array、object。
- **字符串与数字的陷阱**（42 对 "42"）以及缺失与 null 之分，是最常见的形状缺陷。你现在能读 JSON 了——遇到行为怪异时，去检查值的类型。
- **API 密钥**就是密码。绝不要发送到浏览器，绝不要提交进 git，把它存在环境变量里。泄露的密钥应当立即轮换（rotate）。
- **留意速率限制（429）和按量计费**——AI 写的、毫不停顿地调用付费 API 的循环，正是你收到四位数账单的途径。先读定价，并设置预算提醒。

动手试试

在浏览器里找一个无需密钥的免费 API——比如在新标签页直接打开 <https://api.github.com/users/octocat>。你会看到一段原始 JSON 响应。像读表单一样读它：挑出对象 { }，如果有数组就找出来，并为三个不同的值说出它们的类型（哪些是字符串？数字？布尔值？有没有是 null 的？）。你刚刚做的，正是本章要求你对 AI 接入的每个 API 响应都做的那项检查。

本章提示词

我正在调用一个 API，作为初学者，我想自己检查数据的形状。

这是真实的文档（示例请求 + 示例响应）：

< 把 API 文档里的示例请求和 JSON 响应粘贴到这里 >

- 在编写解析代码之前，先给我看这次调用返回的原始 JSON，并指出那些键、它们的类型，以及任何可能为 null 或缺失的东西。
- 确认我的代码读取的键在那个响应里确实存在。
- 确保 API key 是从服务器上的环境变量读取的，绝不要硬编码或发送到浏览器。
- 告诉我如果这次调用失败、超时或触发速率限制会发生什么。

一个应用的解剖



当你让 AI “加一个登录页面”或者“把这个存到数据库里”时，它其实悄悄地在为五六个互相牵动的部件做决定。如果你不知道这些部件的存在，那些回答读起来就像外语一样——你也分不清哪个好答案、哪个是它自己都没搞清楚的话。

这一章是一张地图。它不会让你变成工程师，但它能让你指着一个现代应用的任何一块说出它是干什么的、它住在哪里。一旦

你有了这张地图，后面每一章都不再抽象：在你开始描述一个问题之前，你就会知道它属于哪一个方框。我们会说得很直白——不会甩出没有定义的术语，并且会用一个贯穿始终的例子（用户点击一个按钮）把每一层都走一遍。

全局图景

几乎你要做的每一个应用，都由四个互相对话的部分组成：

- **frontend**（前端）——用户看到并点击的东西。浏览器或手机应用里的那些界面、按钮和文字。
- **backend**（后端）——运行在你掌控的服务器上的逻辑。它来执行规则、干实际的活、并决定谁有权做什么。
- **数据库 (database)** ——信息存放的地方，这样到了明天它还在。账号、帖子、订单。
- **托管 (hosting)** ——在世界某个角落里真正运行以上这一切的那些电脑，好让别人能够访问到它。

它们是这样连起来的：



frontend 从来不会直接跟数据库对话。它总是要经过 backend，因为规则住在那里。这一个事实就解释了应用是怎么搭起来的很多道理——也解释了为什么“干脆让页面直接读数据库”几乎从来都不是答案。

frontend、backend、数据库、托管

我们再往深一层看，用我们的例子：一个用户点击了“保存资料”。

frontend。这是看得见的应用，跑在网页浏览器里，或者作为手机应用。它把表单画出来，注意到点击，并收集用户输入的内容。它在设计上就是不可信的：任何人都能打开它、检查它、改动它。所以它的工作是好看、响应灵敏，而不是去执行任何重要的东西。

backend。当按钮被点击时，frontend 把数据发给 backend——也就是跑在服务器上的代码。backend 检查用户是否已登录、校验数据（“这真的是一个邮箱地址吗？”）、套用业务规则，然后才把它保存下来。信任就住在这里，因为用户没法篡改它。

数据库。backend 把数据交给数据库去存储。它不是什么魔法盒子——你可以把它想成一组非常严格的电子表格（叫做表 / table），能够瞬间被检索、并且永远不会忘记。到了明天，backend 再把这份资料读出来重新显示。

托管。在 prod（生产）环境里，这一切都不会跑在你的笔记本上。它们跑在从托管服务商那里租来的电脑上，比如 Cloudflare、Vercel，或者某个云平台。托管就是让你的应用能够通过一个真实的网址被访问到，而不仅仅是在你自己的机器上。

一个有用的直觉：当某个东西坏了的时候，问问是哪一层。“按钮点了没反应”是 frontend。“它存错了东西”通常是 backend。“它把我的数据忘了”指向数据库。说出是哪一层，问题就解决了一半——同时也是写出一个 AI 能照着干的 prompt 的一半。

HTML、CSS 和 JavaScript

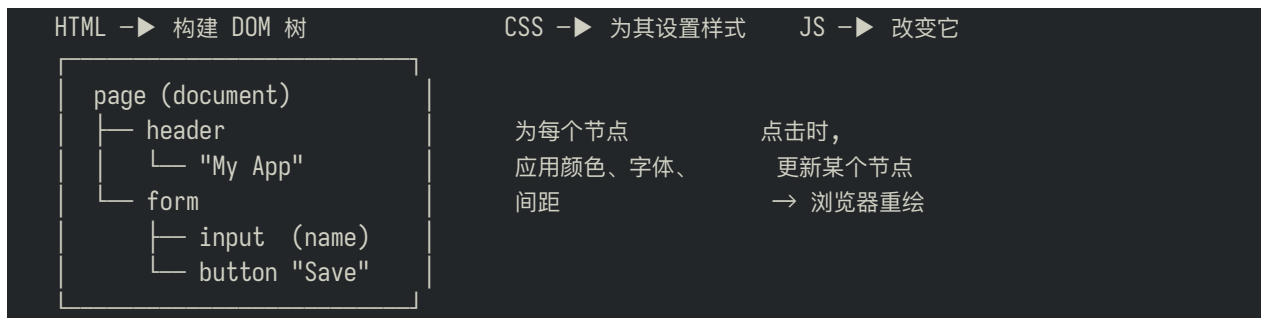
在浏览器里，frontend 恰好由三种语言搭起来。你不需要亲手去写它们，但你需要知道每一种各自管什么。

HTML（超文本标记语言）是结构——内容以及它的骨架。它说“这里是一个标题，这里是一个段落，这里是一个按钮”。它是页面的名词；HTML 单独存在时，是一份朴素、没有样式的文档，就像一个带标签的文本文件。浏览器会把你的 HTML 变成一棵存在内存里、活的对象树，叫做 **DOM**（Document Object Model，文档对象模型）；当代码“改变页面”时，它其实是在改变 DOM，然后浏览器重新绘制以保持一致。

CSS（层叠样式表）是样式——一切看起来是什么样。颜色、字体、间距、布局、靠左还是居中、小屏幕还是大屏幕。CSS 不会改变页面上有什么，只改变它怎么呈现。如果说 HTML 是名词，那 CSS 就是形容词。

JavaScript (JS) 是行为——你交互时发生了什么。它监听点击、把数据发给 backend、显示一个加载转圈、在答复回来时更新 DOM。HTML 和 CSS 是静态的；**JavaScript** 让一个页面做事情。它是动词。（同一种语言通常也用来跑 backend，这就是为什么你到处都会看到它。）

有一种方式可以想象这种分工：HTML 搭起一棵树（DOM），CSS 给它上色，而 JavaScript 伸进去改动它：



包和依赖

没有人会从零开始写一整个应用。那些又难又常见的问题——格式化日期、处理支付、渲染一个日历——别人早就解决好并打包好了。一个**包 (package)**就是一块可复用的代码，你把它拉进自己的项目，而不用自己去写。你项目的**依赖 (dependencies)**就是它所依赖的那一串包的清单。

在 JavaScript 的世界里，管这件事的工具是 **npm** (Node Package Manager, Node 包管理器)。有两个文件负责记账：

- **package.json** ——人类可读的清单，列出你的项目需要什么，外加运行它的命令。这个文件是由你（或者 AI）来编辑的。
- **锁文件 lockfile** (package-lock.json 或类似的文件) ——一份精确的、由机器生成的记录，记下每一个包及其确切的版本，好让这个应用在你的机器、队友的机器、以及服务器上都能构建得一模一样。这个文件你不用手动去改；交给工具去管理就好。

```
{
  "name": "my-app",
  "scripts": {
    "dev": "next dev",
    "build": "next build"
  },
  "dependencies": {
    "next": "16.0.0",
    "react": "19.0.0"
  }
}
```

为什么要在意这些？当 AI 说“我们加一个库来做这个吧”时，它就是在编辑 package.json，而你正在信任别人的代码——更少、更知名的依赖，比一堆来路不明的依赖要安全。还有，当一次构建莫名其妙地坏掉时，依赖版本对不上是常见的嫌疑犯，而“删掉再重新安装这些包”是一个实实在在、很常用的修法。

环境变量与配置

有些值是不能写进你的代码里的：密钥、数据库密码、某个服务的地址。还有一些值需要随着应用运行的地方而改变。你在笔记本上测试的代码（**dev** 环境）应该连到一个测试数据库；你的用户真正在用的那个线上应用（**prod**，即生产环境）必须连到真实的那个——而且用的是同一份代码。

答案就是环境变量 (*environment variables*, 常简称“*env vars*”): 一些有名字的值, 它们住在代码之外, 并在代码运行时被供给进去。代码说“把 DATABASE_URL 给我”, 然后由环境来决定要交出哪一个。

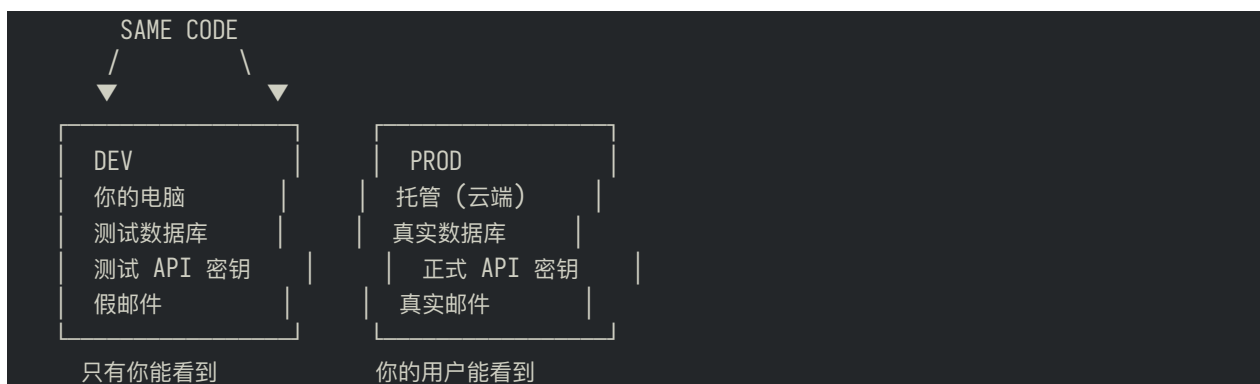
一个配置文件 (例如 .env) ——值住在这里, 而不是在代码里

```
DATABASE_URL=postgres://localhost/myapp_dev
STRIPE_SECRET_KEY=sk_test_51H...      # dev = 一个测试用的密钥
SEND_REAL_EMAILS=false                 # 测试时别给真实用户发邮件
```

有两条规则能帮你省掉实打实的痛苦:

- 密钥绝对不放进代码里。你代码里的任何东西都可能最终变成公开的 (尤其是在一个共享的代码仓库里), 而泄露一个支付密钥或数据库密码是一场货真价实的紧急事件。密钥应该放在 *env vars* 里, 并且不进入版本控制。
- **dev** 和 **prod** 是不一样的。重点在于同一份代码在两者里都表现正确, 因为是由环境——而不是代码——来决定用哪个数据库和哪些密钥。

同一份代码, 两个家——不同的只是那些 *env vars*:



一个项目的样子

打开一个典型的项目, 那一长串文件夹清单可能让人觉得像噪音。它不是——大多数项目都遵循一种认得出来的套路。下面是一个精简过的网页应用:

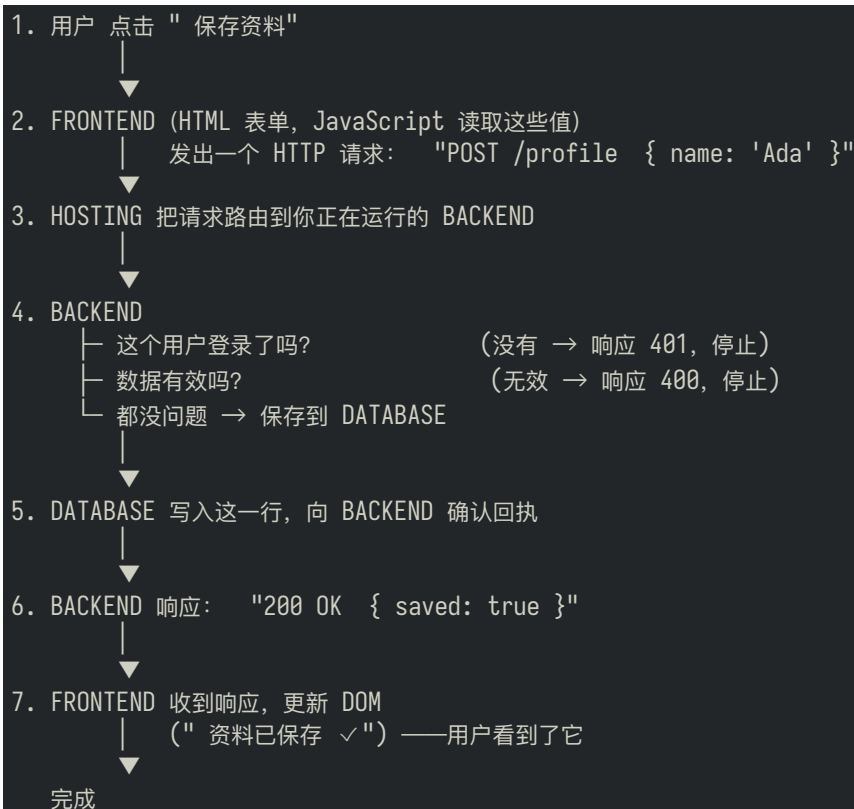
```
my-app/
├─ package.json          # 依赖 + 运行命令 (见上文)
├─ package-lock.json     # 锁文件 lockfile ——精确的版本
├─ .env                  # 本地的密钥/配置 ——不要共享
├─ .gitignore            # Git 应该忽略的文件 (比如 .env)
├─ README.md             # 这个项目是什么, 怎么运行它
├─ public/               # 原样提供的静态文件 (图片、图标)
├─ src/                  # 你真正的代码住在这里
│  └─ components/        # 可复用的 frontend 部件 (一个按钮、一张卡片)
│  └─ pages/ (or app/)   # 用户导航到的那些界面/路由
│  └─ lib/               # 共享的辅助代码和 backend 逻辑
│  └─ styles/            # CSS
```

你不用把这个背下来——只要认得出来就行, 这样当 AI 说“我会把这个加到 *src/lib* 里”时, 你大致知道那是哪里、为什么在那里。有几条起承重作用的约定:

- *src/* 装着你写的代码。最顶层那些杂乱的东西基本上都是配置。
- 根目录下的配置文件 (那个 *package.json*、那些点开头的文件) 配置的是工具, 而不是你应用的行为。
- *.gitignore* 列出那些不应该被存进版本控制的文件——你的 *.env* 就属于这里, 而这正是密钥得以不进入共享仓库的办法。

一个请求是怎么流动的

现在我们把它们串起来。我们来跟着用户点击“保存资料”，走过你刚刚认识的每一个部件。这是全书最有用的一个心智模型；如果你只内化一张图，就让它它是这张。



把每一层做了什么走一遍：**frontend** 收集了输入并把它发出去；**托管** 把请求送到了正确的地方；**backend** 执行了规则（登录、校验）并做了决定；**数据库** 把它变成永久的；然后响应一路流回来，好让 **frontend** 能更新屏幕。每一个应用都是这个循环的各种变体，每秒钟跑成千上万次。

注意那些检查全都发生在 **backend**，从不发生在 **frontend**——这就是第一节里那条规则的实际运用：**frontend** 是给用户的，**backend** 是给真相的。当你知道每一步发生在哪里时，你就能精确地描述一个功能（“保存之前在 **backend** 上校验邮箱”），而不是含糊地说（“把表单弄好”）——而精确，就是整场游戏的全部。

回顾与练习

关键点

- 每个应用都是四个部分：一个 **frontend**（用户看到的東西）、一个 **backend**（规则和逻辑）、一个 **数据库**（数据持久存放的地方）、以及 **托管**（这一切运行的地方）。**frontend** 从不直接碰数据库——它要经过 **backend**。
- 在浏览器里，**HTML** 是结构，**CSS** 是样式，**JavaScript** 是行为；代码通过改变 **DOM** 来改变页面。
- 你不用什么都自己写——包（由 **npm** 通过 `package.json` 和一个锁文件来管理）以 **依赖** 的形式提供了已经解决好的问题。
- **环境变量** 让密钥待在代码之外，并让同一份代码在 `dev` 和 `prod` 里都表现正确。
- 一个请求的流动是 **frontend** → **托管** → **backend** → **数据库** 再返回；说出一个问题住在哪一层，就解决了它的一半。

动手试试

挑一个你每天用的应用，大声地把一个动作沿着各层叙述一遍：“我点了发送（**frontend**）→ 它去到他们的服务器（**backend**）→ 服务器检查我是否被允许并保存了这条消息（**数据库**）→ 然后聊天界面更新了（**frontend**）。”然后猜猜下面每个问题住在哪一层：按钮看起来没对齐；应用说“你还没登录”；昨天的一条消息不见了。把这种“说出是哪一层”练熟，正是这本书剩下部分所建立的那项技能。

本章的 *prompt*

我在学应用是怎么拼到一起的。这是我想要的一个功能：

< 用一句话描述用户应该能够做什么 >

在写任何代码之前，请帮我按层拆解它：

- Frontend: 用户看到什么，以及什么会被发出去
- Backend: 必须跑哪些规则/检查，以及在哪儿跑
- Database: 需要存储或读取哪些数据
- 涉及到的任何 env vars 或配置（尤其是密钥）

然后用编号的步骤描述从点击到结果的请求流动。

请用大白话——我想在写代码之前先理解它的样子。

Vibe Coding 心态



Vibe coding 是个含义复杂的词。在某些人看来，它意味着敲下一行提示词，接受模型吐出的任何东西，然后一行都不读就直接上线。那个版本只是个玩具。它产出的演示，在真实用户一碰之下就会分崩离析。

本书讲的是另一个版本：像资深工程师指挥团队那样，指挥一个 AI 来构建生产级软件。你始终掌控构建什么以及为什么它是正确的。你把敲键盘的活儿委派出去。这就是那个转变，而它比听起来要大得多。

Vibe coding 究竟是什么

Vibe coding 就是用自然语言表达意图、让 **AI** 生成实现，与此同时你在一个紧凑的循环里掌舵、审查并打磨。

它不是：

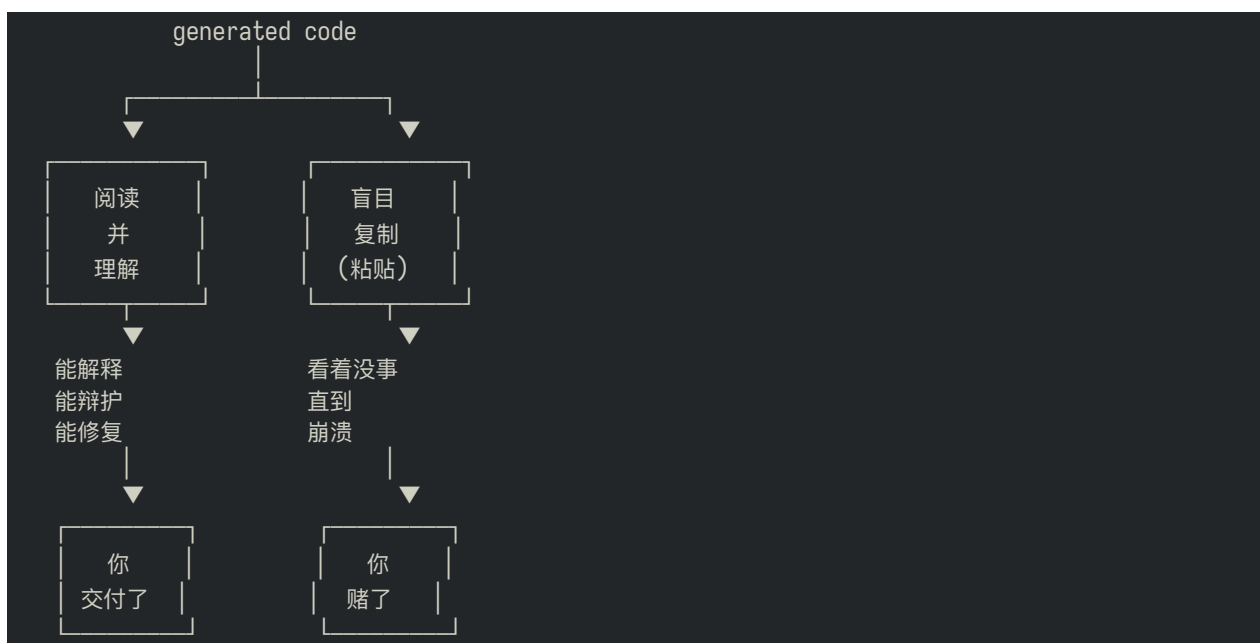
- 盲目接受你并不理解的生成代码。
- 用感觉代替工程判断。
- 一个能把一句话变成上线产品的魔法按钮。

它是：

- 把模型当作一个快速、博学、但略微过于自信的初级工程师。
- 把你的注意力花在架构、正确性和品味上，而不是语法上。
- 在过去要花几小时的事情上，用几分钟完成迭代。

最有效的心智模型是：你是技术负责人，AI 是你的团队。一个好的技术负责人不会亲手写每一行代码。他们决定方向、设定约束、审查 pull request（为获得批准而提交的一批改动，即拉取请求），并在坏点子上线前把它拦下来。你正在做的就是这件事——只不过是以前对话的速度。

这两条路从同样的生成代码出发，却通向截然不同的终点：



这个定义里有一个词挑着最重的担子：**理解**。整套做法都建立在一条规则之上——理解你所上线的东西。你不必亲手敲它，但你必须能解释它、能在代码评审里为它辩护、能在凌晨两点它把你呼起来时修好它。如果做不到，那你不是在做 vibe coding，你只是赌了一把然后运气好。这两者之间的鸿沟，正是整本书的主题。

心态的转变

如果你过去习惯了亲手敲下每一个字符，那么最难的部分不是学习工具，而是放开键盘。

你过去的瓶颈是生产——你能多快写出正确的代码。你新的瓶颈是规格说明与审查——你能多清晰地描述你想要什么，以及你能多好地判断返回的结果。

这改变了你优化的目标：

- **以前：**记住各种 API（应用程序编程接口——工具对外提供的命令菜单）、与语法搏斗、打字飞快。
- **现在：**精确描述意图、拆解问题、批判性地评估输出。

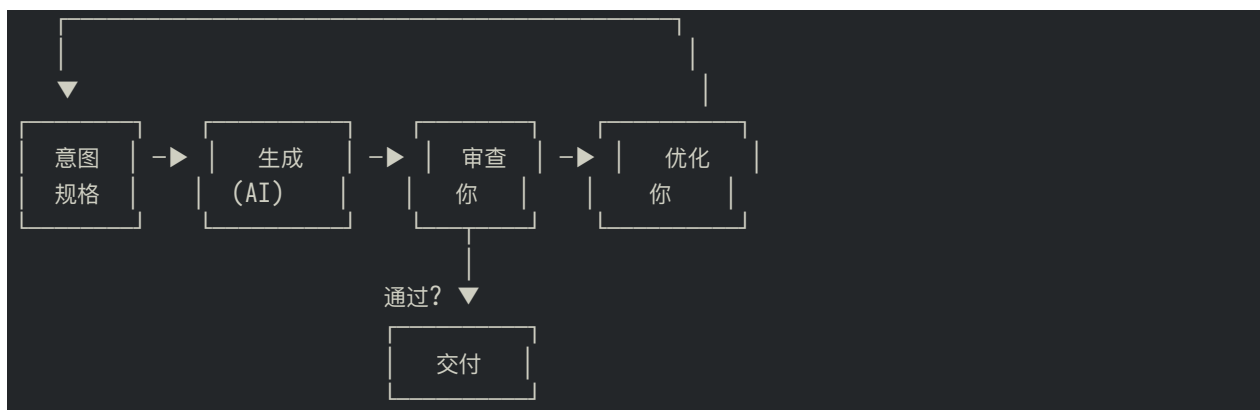
在这件事上如鱼得水的工程师并不是打字最快的人。他们是那些对”好”长什么样有强烈见解、并有纪律去拒绝达不到标准的输出的人。

还有一个更安静、常常让人措手不及的转变：你与不懂之间的关系。过去的本能是，动手之前先停下来把 API 学会。如今你往往能在一个陌生的技术栈上生成出第一版，然后通过回读它来学习。这是真正的超能力——也是个陷阱。读一段你本写不出来的代码，没问题。上线一段你无法评估的代码，不行。分界线在于你能否有把握地判断这个输出是对的。在”写得比你快”这件事上尽管依赖模型；在”懂得比你愿意去验证的更多”这件事上，永远别依赖它。

循环：意图 □ 生成 □ 审查 □ 打磨

本书中的一切都回归到一个循环。把它内化。

循环会一直转，直到输出达到你的标准——而审查是它每次都必须穿过的关口：



1. **意图**。陈述你想要什么以及关键的约束——输入、输出、边界情况、技术栈、风格。模糊的意图得到模糊的代码。
2. **生成**。让模型写实现。别去手把手管语法；描述行为。
3. **审查**。读返回的内容。它真的做到那件事了吗？它处理了空列表、null、失败的网络调用吗？它是能跑通的最简版本吗？
4. **打磨**。指出哪里不对，给出具体的修正，然后重新生成。重复直到它达到你的标准。

大多数新手会跳过第 3 步。他们生成，它能跑，他们就继续往下走。然后它在生产环境里、在他们从未描述过的那个情况上崩掉了。审查这一步，正是工程发生的地方。

下面是这个循环里一个好的初始提示词长什么样：

```
写一个 TypeScript 函数 `parseDuration(input: string): number`，  
把 "1h30m"、"45s"、"2d" 这样的字符串转换为总秒数。
```

要求：

- 支持单位：d（天），h（小时），m（分钟），s（秒）。
- 多个单位可组合："1h30m" = 5400。
- 对无效输入（空，未知单位，负数）抛出带清晰消息的 Error 予以拒绝。
- 不用外部库。包含覆盖边界情况的 5 个单元测试。

只展示函数和测试。

注意它的结构：一个精确的签名、明确的规则、点名列出的边界情况、陈述清楚的约束，以及对测试的要求。你不是在指望模型猜对——你是在消除那些会导致糟糕猜测的歧义。

打磨这一步和第一条提示词一样是门手艺。模糊的反馈得到模糊的修正。比较下面这两种修正：

差的："这是错的，修一下"

好的："parseDuration('1h30m') 返回 90 而不是 5400 ——你把数字相加了，却忽略了单位的倍率。在相加之前，把每个值乘以其

单位对应的秒数 (d=86400, h=3600, m=60, s=1)。
另外为组合情况 '2d3h' 加一个测试。"

好的那条点名了症状、原因和修法。你是在像评审同事的 pull request 那样评审——而你指得越精确，花的回合就越少。每一圈循环都应当收敛。如果你发现自己在原地打转，那是个信号：你的意图说明不足。停止重新生成，改去重写规格。

它的闪光之处

Vibe coding 在以下方面真正具有变革性：

- **全新项目的脚手架搭建。**搭起一个新应用、接好路由、配置好设置——这些活儿乏味但路数成熟。
- **样板代码和胶水代码。**CRUD 端点、表单校验、数据转换、两个 API 之间的适配器。
- **陌生领域。**一门你不太熟的语言或框架。模型懂那些惯用法；你提供判断。
- **重构与迁移。**把这些 promise 转成 async/await，“按职责把这个 400 行的文件拆开”。
- **用完即弃的探索。**反正都要重写的原型。速度比打磨更重要。

在这些领域，一个清晰的提示词加上一次认真的审查，每次都胜过亲手敲键盘。

贯穿其中的共同线索是，这些任务都**路数成熟且可验证**。模型见过上万个 CRUD 端点，所以它产出地道的版本；而你一眼就能看出结果对不对。当这两点同时成立——常见的模式、检查成本低廉——vibe coding 就近乎纯粹的收益。

它失效的地方

对它的边界要诚实，因为无人监督的 vibe coding 正是在这些地方把人坑惨：

- **深奥而新颖的逻辑。**微妙的算法、并发、任何“对了 95% “就等于”坏了”的东西。模型即便错了也很自信。
- **铺得很开的跨文件改动。**它在大型代码库里会跟丢线索，悄无声息地搞坏它看不见的东西。
- **安全和钱。**认证、支付、权限，任何一个看似合理的 bug 会带来真实后果的地方。审查这些时要像你的饭碗系于此一样，因为它可能真的是。
- **规格说明不足的问题。**如果你都不知道正确长什么样，AI 没法读你的心。规格说明是垃圾进，代码就是垃圾出。

失败的模式几乎总是一样的：代码看起来对，在顺利路径上能跑，却以一种你只要读了就能抓住的方式错了。解药就是审查这一步。它无法被跳过。

两个问题就能把任何任务放到这张地图上——模式是否常见，以及结果是否容易核查：

	易于验证	难以验证
常见模式	纯赚 (脚手架, 样板)	放慢, 要求测试
罕见 / 新颖	需读但可行	危险区 (新逻辑, 鉴权, 金钱)

注意上面那条规则的反面：这些都是罕见、微妙、或验证成本高的情形。模式不常见，所以模型在猜；或者 bug 在顺利路径上不可见，所以快速检查抓不到。这并不意味着在这里就要躲开 AI——它意味着你要放慢、把步子切小、要求测试，并把每一行都当作有敌意的来读。

常见的坑

有几个陷阱，几乎会在头一个月里把所有人都套进去：

- **接受第一个能跑的东西。**“它能跑”和“它正确”是两个不同的主张。前者只是说顺利路径没崩；后者是说你检查了边界。别把两者混为一谈。

- **巨型提示词。**一口气要整个功能，会给模型太多漂移的余地，也给你太多需要一次性审查的东西。拆解它。先上线最小的、可审查的一片，再在其上叠加。
- **任由上下文腐烂。**会话越长，模型就越会与早先的决定自相矛盾，并重新引入你已经修过的 bug。一旦出现这种苗头，与其和一条已经糊涂的线程缠斗，不如总结当前状态、另起一条新的。
- **把自信当成信号来信任。**模型的语气，在它对的时候和它在幻觉的时候是一模一样的。它的笃定什么都说明不了。你的验证才说明一切。
- **外包的不只是打字，连决策也外包了。**问“我在 X 上有哪些选项？”没问题。让那个答案在你的判断不在场的情况下替你选定架构，就是错误。

品味是新的核心技能

当任何人都能生成代码时，稀缺的技能就是知道哪些生成的代码是好的。

品味会告诉你：

- 这个抽象搞早了；去要那个更简单的版本。
- 这个能应付演示，但应付不了真实的输入。
- 这个技术上正确，但维护起来是场噩梦。
- 这个差不多对了——具体要改的就是这个地方。

你以老派的方式建立品味：读大量代码、把东西做出来上线、然后看它怎么坏。AI 不会取代那份经验——它让那份经验更有价值，因为如今你的判断才是瓶颈，也是区分高下的关键。有品味的人能快速交付稳固的软件。没有品味的人快速交付，然后淹没在 bug 里。

这里有个令人不适的推论：AI 抬高了所有人的下限，这意味着它抹平了懂语法的优势，磨利了有判断力的优势。那些靠它来跳过学习的初级工程师会触顶停滞，因为他们从未建立起那种能在输出错了时告诉他们的品味。而那些用它来交付更多、阅读更多的人则会复利增长。同样的工具，截然相反的结果——而其中的差别，完全取决于你是否对返回的东西保持投入。

现实的预期

现在就把这些预期立好，免得你日后失望：

- **你会读大量代码。**审查就是这份工作本身。如果你不想读代码，这并不是绕过它的捷径。
- **它不会一次搞定任何不平凡的东西。**预期要跑几轮循环。这很正常，不是失败。
- **责任仍在你身上。**当 AI 写的代码在生产环境崩掉时，“是模型写的”不是借口。是你把它上线的。
- **它让你更快，而不是免费。**这些工具非常了不起。它们不是魔法。判断始终是你的。

再直白地说一句本书不是什么：它不是一条让你上线自己都不理解的软件的路，也不是一夜暴富的捷径。任何把 **vibe coding** 当作“不学编程也能做出 App”来兜售的人，卖的都是那个玩具版本——碰到真实用户就分崩离析的那个。这门技能只会对那些把它当作叠加在工程判断之上的杠杆、而非判断的替代品的人复利增长。

Vibe coding 的承诺是真实的：你能构建得更多、更快、跨越比你独自一人更多的领域。但它奖励那个保持投入的工程师——那个去指挥、审查、打磨的人——并惩罚那个抽身离场、接受任何出现之物的人。

去当技术负责人，而不是旁观者。本书余下的内容，讲的就是如何做到。

回顾与练习

核心要点

- Vibe coding 是在你掌控做什么和为什么的前提下，指挥 AI 构建软件——而不是盲目交付你解释不了的代码。
- 把这一切串起来的规则是理解你交付的东西：你不必亲手敲它，但你必须能为它辩护、能修复它。
- 你的瓶颈从生产（敲得快）转移到了规格说明与审查（描述清楚、批判性地判断）。
- 活在循环里——意图 → 生成 → 审查 → 打磨——并且永远不要跳过审查这一步，真正的工程发生在那里。
- 模型的自信不是信号；你的验证才是。品味与判断力才是稀缺、且会复利累积的技能。

动手试试

挑一个你真正理解的小函数——比如“校验密码是否符合策略”或“把字节数格式化成人类可读的字符串”。为它写一个精确的提示词（签名、规则、边界情况、要求附测试），生成它，然后刻意执行审查这一步：亲手追踪至少一个边界情况，确认代码能处理。如果不能，就写一个精确的打磨提示词（症状 + 原因 + 修复），而不是“修一下”。目标是在你能完全评估的代码上，把整个循环从头到尾完整地感受一次。

本章提示词

作为我的结对编程伙伴。我会描述一个小而自包含的函数。在写任何代码之前，先复述需求并列出具你将处理的边界情况，好让我确认。然后写出该函数，以及覆盖这些边界情况的单元测试。

函数：< 一行描述 >

输入 / 输出：< 类型与形态 >

约束：< 技术栈，要用或要避免的库，性能限制 >

代码之后，指出你做出的、我应当核实的任何假设。

保持为满足需求的最简单版本。

搭建你的 AI 工作环境



Vibe coding 是一种合作。你带来意图和判断；AI 带来速度和不知疲倦的打字。但和任何合作一样，它在共享的环境、清晰的基本规则、以及一个允许犯错的安全之地中运作得最好。本章讲的就是在你写下第一个功能之前，先搭好那个环境。

一个好的工作环境做三件事：它给 AI 做出聪明改动所需的上下文，它让你能快速看到这些改动的结果，它让每一次改动都易

于撤销。把这些做对，AI 感觉就像一位资深合作者。做错了，它感觉就像一台老虎机。这个差别很少是因为模型——而是因为你把它放进了什么样的房间。一个能干的智能体，放进一个没有测试、没有上下文文件、没有版本控制的混乱仓库里，会手忙脚乱；一个平平无奇智能体，放进一个干净、仪表齐全的项目里，会让你刮目相看。配置才是那个乘数。

这些都不是做一次就忘的杂活。把你的工作环境当作一件需要持续调校的东西：每当 AI 以糟糕的方式让你吃惊时，问问是不是某块缺失的上下文、某个缓慢的反馈循环、或某个不安全的默认值放任了它——然后去修房间，而不只是修那一行代码。

选择一个 AI 原生的编辑器或助手

你的第一个决定是 AI 住在哪里。如今大致有三个类别，而你不必永远绑定。

- **AI 原生编辑器** (Cursor、Windsurf)：一个熟悉的代码编辑器，把 AI 深度接入其中。如果你想在 AI 协同工作时看到并触碰代码，这是最佳选择。
- **终端/CLI 智能体** (Claude Code、Codex CLI、Gemini CLI、Aider)：AI 作为一个命令行智能体运行，读取文件、编辑文件并运行命令。这里的 CLI（命令行界面）就是用输入命令代替点击按钮的纯文字控制台。最适合让 AI 跨多个文件更自主地工作。
- **IDE 插件** (GitHub Copilot、Continue、JetBrains AI)：在你已经钟爱的编辑器里附加上去的辅助。IDE（集成开发环境）就是大多数程序员写代码所用的一体化应用。

要交付真正的软件，挑一个能读取你的整个项目、编辑多个文件、并自行运行命令（测试、构建、linter）的工具。能运行并检查自己工作成果的能力，正是把一个有用的助手和一个自动补全玩具区分开来的东西。一个只能建议下一行的工具，会把所有验证都留给你手动做；一个能跑它刚改过的那个测试的工具，会自己把循环闭合。

在这个核心能力之外，定下来之前还有几样东西值得权衡：

- **上下文窗口与项目感知**——它能否真正加载相关文件，还是在一个任务做到一半时就忘了对话的开头？
- **权限模型**——它在运行命令和编辑文件之前会先问你，还是先动手？两种都有用武之地；要知道自己处在哪种模式。
- **成本与额度**——自主智能体能很快烧掉用量。在你放它跑一个小时之前，先弄清定价。

别在选择上纠结；本书中的技能在工具之间是通用的。多数有经验的 **vibe coder** 最后都会装两个——一个用于批量工作的快速终端智能体，一个用于审查的 GUI 编辑器（GUI 即图形用户界面，就是那种带窗口和按钮、靠点击操作的熟悉程序）——而本章收尾的 TUI vs GUI 一节，正会回到这对组合上。这里的 TUI（文本用户界面）则是它在终端里运行的纯文字版本。

把项目结构组织得让 AI 能导航

AI 模型靠读你的代码来推理。一个混乱、铺得很开、不一致的项目，对它们来说和对你来说一样难。你不需要完美的架构，但有几个习惯能立刻见效：

- **保持可预测的布局**。源码放一个地方，测试挨着源码或镜像源码的结构，配置放在根目录。
- **使用清晰、描述性的命名**。calculateMonthlyInvoiceTotal 胜过 calc2。AI 把名字当作线索。
- **偏好更小的文件**。对你们俩来说，一个 200 行的文件都比一个 2000 行的更容易正确地修改。
- **把相关的东西放在一起**。当一个功能的代码、样式和测试坐在一处时，AI 一次读取就能收集到上下文。
- **保留一个清晰的入口点**。应用启动的那个明显的单一位置，给了 AI 一根在梳理项目时可以顺着拉的线。

一个典型的、对 AI 友好的布局可能长这样：

```
my-app/
├── AGENTS.md           # 给 AI 的项目规则 & 上下文
├── README.md          # 项目简介、如何运行
├── .env.example        # 记录所需密钥（无真实值）
├── .gitignore          # 排除 .env、构建产物、node_modules
├── package.json        # 脚本：dev, test, lint, build
├── src/
│   ├── features/
│   │   └── invoices/   # 单个功能的代码 + 测试，放一起
│   └── lib/            # 共享辅助函数
```



```
|   └─ index.ts      # 入口点
|   └─ tests/        # 横切 / 集成测试
```

回报会复利式增长：你的项目越容易让 AI 导航，它的改动就越小、越精准。一个井井有条的仓库，让 AI 去碰三个相关文件，而不是在三十个文件之间瞎猜——这意味着更小的 diff、更少事故，以及你真能读得下去的审查。如果你发现 AI 在做大刀阔斧、漫无焦点的编辑，真正的罪魁祸首往往是结构，而不是提示词。

写一个上下文文件 (**AGENTS.md**)

你能做的杠杆率最高的一件事，就是写一个 AI 会自动读取的上下文文件。不同的工具寻找不同的名字——AGENTS.md、CLAUDE.md、.cursorrules，或者 .github/copilot-instructions.md——但理念是相同的：一份活的文档，告诉 AI 你的项目是如何运作的，以及你希望它如何行事。

没有它，AI 靠猜。有了它，AI 不再重新格式化你的代码，不再发明你不用的库，不再重新解释你早已做过的决定。把它想成给一位新队友的入职文档——这位队友很聪明，但在每次会话之间会完全失忆——他要变得高产所需的一切，都得放在他每次都会读的地方。

一个实用的起步文件：

```
# Project: Invoice Manager

## What this is
A small web app for freelancers to create and track invoices.

## Tech stack
- TypeScript + React (Vite)
- Tailwind for styling
- SQLite via Prisma
- Vitest for tests

## Commands
- Install: `npm install`
- Run dev server: `npm run dev`
- Run tests: `npm test`
- Lint: `npm run lint`

## Conventions
- Use functional components and hooks; no class components.
- Keep components small; one component per file.
- Write a test for every new function in `src/lib`.
- Never edit files in `src/generated/` — they are auto-generated.

## Don'ts
- Don't add new dependencies without flagging it first.
- Don't commit secrets; use environment variables.
```

让它保持简短，也保持最新。一个膨胀到一千行的上下文文件，就不再会被认真读了——无论是你还是模型——所以删起来要和加起来一样狠。当 AI 反复犯同一个错误的假设时，那就是你该加一行的信号；当一条规则不再符合现实时，在它开始主动误导智能体之前把它删掉。

有几个模式，把一个有效的上下文文件和一个被无视的区分开来：

- 写得具体而明确。”用我们的约定”毫无用处。“校验用 zod；我们不用 yup”才是一条 AI 真能照做的规则。
- 把”别做”说得和”要做”一样响亮。模型急于帮忙，这意味着除非你叫它别这么干，它会乐呵呵地加个依赖或重写一份配置。

- **指向示例。**照着 `src/features/invoices` 的风格来”给了 AI 一个可以照抄的具体范本，而不是一条要去揣摩的模糊指令。

把上下文文件当作一份记录，记下你本来每次都得重复一遍的每一条教训。它是这整章里最廉价、最持久的投资。

让版本控制成为本能反应

版本控制是你的安全网，而有了 AI 它变得不可或缺。AI 能在五秒内改十个文件——你需要一种干净的方式去检查、保留或丢弃那份工作。

- **频繁地、小块地提交。**在你要求一个大改动之前，先把能跑的提交了。一棵干净的工作树意味着你总能回到那里。
- **提交前读 *diff*。**这才是你真正的审查步骤。你不必理解每一行，但你应该理解改了什么、为什么改。
- **用分支来做有风险的工作。**开一个分支，让 AI 放手去干，只在你满意结果时才合并。如果不满意，删掉它——毫发无伤。
- **写诚实的提交信息。**未来的你和 AI 都会读历史来理解意图。

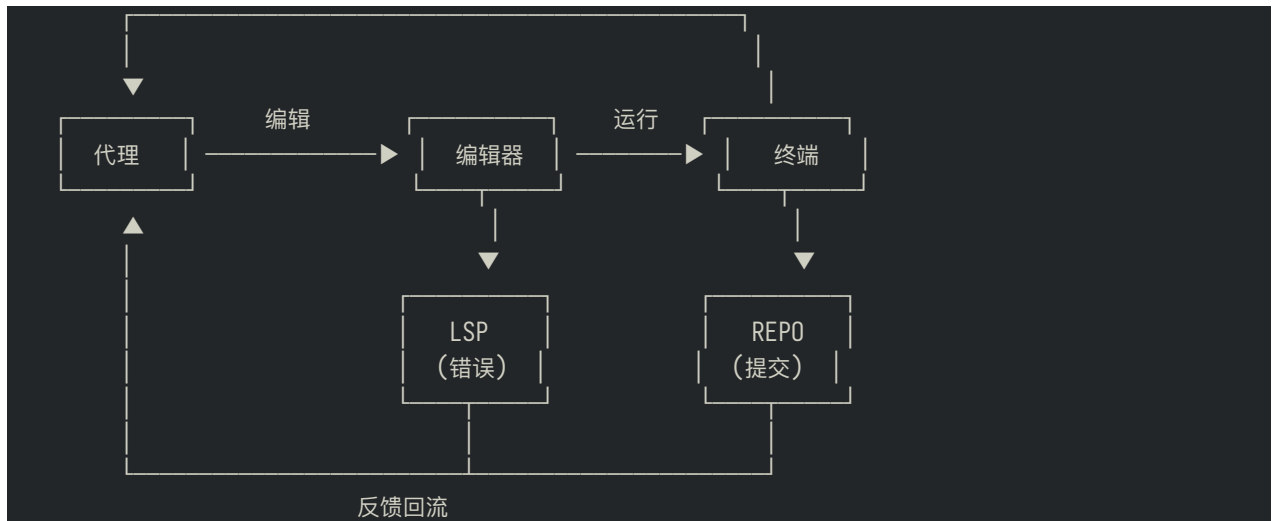
心态上的转变：提交不是官僚主义，它们是游戏里的存档点。在打 Boss 前先存档。再学会那两个让干净历史值得维护的逃生口——`git checkout .` (或 `git restore .`) 扔掉未提交的实验，`git revert` 撤销一个你已经做下的提交。知道自己永远能回到一个已知良好的状态，正是这一点让你能给 AI 一条长长的缰绳而不焦虑。

让反馈循环保持紧凑

你越快能看到一个改动是否有效，你和 AI 就越快收敛到正确的东西上。一个缓慢的循环——你要等上几分钟才知道有没有搞坏什么——会悄无声息地把每一个错误成倍放大。

- **让“跑起来”变成一条命令。**`npm run dev`、`make test`，无论是什么，把它记在你的上下文文件里，这样 AI 也能跑它。
- **依靠快速的测试。**哪怕只有少数几个单元测试，都能给 AI 一种验证自己工作的方法，而不用你盯着。
- **使用 *linter* 和格式化工具。**它们能瞬间抓住一整类错误，并把风格之争挡在你的审查之外。
- **把错误回传给 AI。**当出问题，把真实的错误信息贴回去。具体的反馈产出具体的修复；“它不工作”只会产出猜测。

AI 工作空间的各个部件构成一个相连的循环——智能体编辑仓库，编辑器和语言服务器报告问题，终端运行代码，而每一个结果都流回智能体：



这个想法最深的版本，是让 AI 自己把循环闭合。与其由你来运行命令、读取失败、再转述回去，不如给智能体权限，让它在每次改动后运行测试和类型检查器，并对自己看到的东西做出反应。循环于是变成：编辑、运行、读取失败、修复、再运行——全程没有你卡在中间。你的角色从信使转为导演：你设定目标和护栏，然后看着循环转起来。当 AI 能够编辑、运行、看到失败、并自行修复时，你就建好了一个能干真正活儿的循环，而你稳坐导演椅。

谨慎管理机密

AI 工具会读你的文件，有时还会把上下文分享给远程服务。这让泄露机密成为一个真实的风险。从第一天起就建立好习惯：

- 永远别把真实的密钥放进代码。使用环境变量和一个 `.env` 文件。
- 永远把你的 `.env` 加进 **gitignore**。提交一个带空占位符的 `.env.example`，这样 AI 不用看到值就知道需要什么。
- 把密钥的权限范围收窄，并轮换任何可能已被暴露的密钥。
- 在你的上下文文件里告诉 **AI** 永远不要硬编码或提交机密。

在你的 `AGENTS.md` 里放一条响亮的规则，比如 `Never read or print the contents of .env`，是一份廉价的保险。一个会拒绝包含疑似密钥的提交的 `pre-commit` 钩子或密钥扫描工具也是——泄露的代价是不对称的，所以在这里纵深防御很重要。一个被推到公开仓库的密钥，在它落地的那一刻就已经泄露了，哪怕你几秒后就删掉，因为机器人会实时抓取提交。拿不准时，就把暴露的密钥当成已经废了，去轮换它；轮换一个密钥只要一分钟，而给一个泄露的密钥善后可能要花上好几天。

一个健全的本地配置让 AI 的改动安全

用一个让错误既廉价又可恢复的环境把这一切串起来：

- 在分支上工作，而不是直接在 `main` 上。
- 在大改动之前保持工作树干净，好让 `diff` 有意义。
- 让测试和开发服务器随时能用一条命令跑起来。
- 把机密留在仓库之外，并让上下文文件保持最新。

有了这个地基，任何 AI 改动的最坏情况就是 `git checkout` . 加上一个全新的开始。正是那份安全，让你能快速行动、大胆指挥——知道 AI 所做的没有任何事是真正不可逆的。在下一章，我们将开始用这个工作环境来真正动手构建。

实时诊断 (LSP)

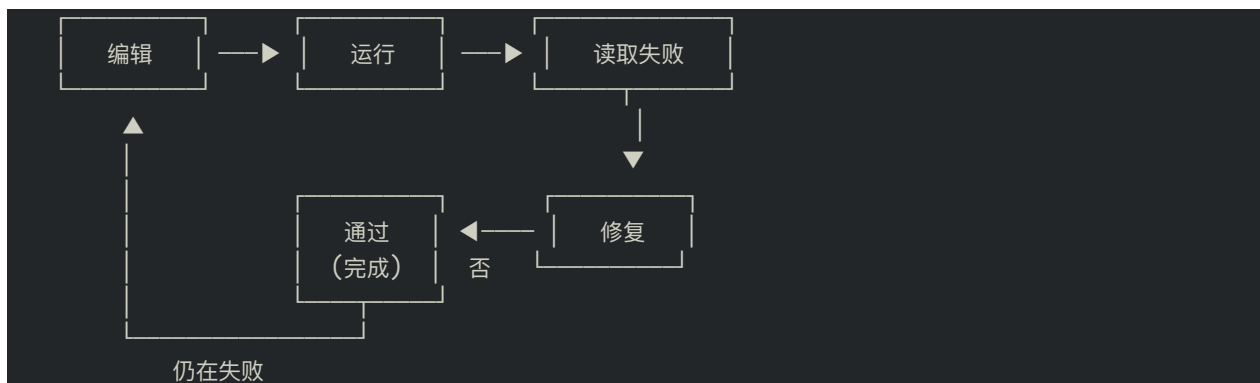
最紧凑的反馈循环不会等你去运行代码——它会在你刚敲下错误的那一刻就抓住它。这正是语言服务器协议 (**LSP**) 所提供的：你编辑器背后的实时“智能”，包括实时的错误、类型信息、跳转到定义和自动补全。语言服务器会在你的代码发生变化时读取它，并在任何东西运行之前就报告问题。

对 **vibe coding** 而言，这里的杠杆在于把这些诊断回传给 **AI**。没有它，循环很慢：AI 写代码，你运行它，它因为一个拼写错误或类型错误而崩溃，然后你手动把错误信息贴回去。把诊断接进来以后，智能体会在它写下那行代码的瞬间，看到和你一样的红色波浪线——并随手修掉类型或 `lint` 错误，根本不用等代码运行。

你可以用几种方式给智能体这双眼睛：

- 为你的技术栈运行语言服务器 (`tsserver`、`rust-analyzer`、`pyright` 等等) ——大多数 AI 原生编辑器会替你做好这件事。
- 让智能体运行一个类型检查或 `lint` 命令并读取输出，或者通过一个 MCP 服务器暴露诊断信息。
- 把一份干净的诊断报告当作“完成”的一部分，而不是事后才想起来的东西。

这样接好线，智能体就能自己跑一个自我纠正的循环——编辑、运行、读取失败、修复，再运行，直到诊断干净地返回：



这也是为什么一门带类型的语言，对 **vibe coding** 来说可以是一种安静的超能力：每一个类型注解都是一份小小的、由机器检查的规格，会在 AI 的错误运行起来之前就把它逮住。一个“运行—崩溃—粘贴”的循环，于是变成了一个安静的、自我纠正的循环。AI 少做无用功，收敛得更快。

扩展你的智能体

开箱即用，AI 助手是通用的。一旦你在它周围裹上一层工具，同一个基础模型就会变成一个好得多的协作者——把一个普通的聊天机器人，变成一套为你的项目调校过的**外壳 (harness)**。要知道的主要扩展点有：

- **技能 (Skills)** ——可复用的打包能力（指令加脚本），智能体按需加载，这样你的领域知识无需每次会话都重新解释一遍就能用上。
- **插件 (Plugins)** ——把技能、命令、钩子和连接器打包在一起、作为一个可安装单元发布的捆绑包。
- **MCP (模型上下文协议)** ——一个开放标准，通过一个通用接口把智能体连接到外部工具和数据（数据库、API（应用程序编程接口——其他软件对外提供的标准命令菜单）、你的文件系统）。
- **子智能体 (Subagents)** ——主智能体为专注的工作（搜索、审查、一段长长的支线任务）派生出来的专门助手智能体，每个都有自己的上下文窗口。
- **钩子 (Hooks)** ——在生命周期事件（一个命令运行之前、一个文件被编辑之后）自动触发的脚本，让你能够确定性地强制规则或注入上下文。
- **斜杠命令 (Slash commands)** ——你用 `/name` 调用的、有名字的短 workflow，把一个重复的多步请求收拢成一次按键。

你不需要在第一天就用上所有这些。先从朴素的开始，一旦你发现自己在重复同样的设置，就加上一个技能或一个斜杠命令。一条有用的经验法则：当你第三次敲下同一段多步指令时，把它变成一个斜杠命令；当你第一次希望智能体能够到一件它够不到的工具时，去找一个对应的 MCP 服务器。钩子尤其会在护栏那一章再次出现，作为一种强制执行你的规则的方式，而不是只盼着 AI 还记得。

macOS vs Windows

大多数 AI 工具——CLI 智能体、脚本、教程——都悄悄地假设你用的是 Unix shell。这左右了开发环境的实际选择。

- **macOS** 是 **Unix** 原生的。它自带一个真正的 POSIX shell，以及和开发界其余人一样的 git、ssh 和 curl。大多数智能体和脚本“开箱即用”。
- **Windows** 完全有能力胜任，但最顺滑的路子是 **WSL2** (Windows Subsystem for Linux)：Windows 里面一个真正的 Linux 环境。把你的 AI 编程放在 WSL2 里面做，而不是去跟原生 PowerShell 的怪癖较劲。
- **Linux** 是这个家族的第三位成员，就这些用途而言它的行为几乎和 macOS 一模一样——它就是 Unix，所以同样的智能体和脚本原封不动地跑。如果你已经在它上面，那你就齐活了。

几个让你的工作在它们之间保持可移植的坑：

- **路径**——`/home/you/project` 对比 `C:\Users\you\project`；反斜杠和盘符会让脚本崩掉。WSL2 绕开了其中大部分。
- **行尾**——Windows 用 CRLF，Unix 用 LF。不匹配会造成嘈杂的 diff 和坏掉的 shell 脚本；设置 `core.autocrlf` 和一个 `.gitattributes` 来统一归一化成 LF。
- **大小写敏感**——macOS 和 Windows 往往把 `File.ts` 和 `file.ts` 当作同一个；Linux 和 CI 不会，所以在本地能构建的代码可能在云上失败。在导入和文件名里对大小写保持一致。
- **权限**——Unix 的文件模式 (`chmod +x`) 没法干净地映射到 Windows 上。

无论你用哪个，都让 AI 把命令写得可移植——正斜杠、LF 行尾、一致的大小写——这样它的脚本在你的笔记本上和在和 CI 里跑起来都一样。当 AI 递给你一条假设了某个你没装的工具的单行命令时，最快的修法往往是在上下文文件里告诉你你的操作系统和 shell，好让它别再瞎猜。

TUI vs GUI

AI 编程工具有两种形态，而一个高产的 **vibe coder** 两种都会拿来用。**终端 / TUI 智能体** (Claude Code、Codex CLI、Aider) 跑在终端里：快、可脚本化、可组合，非常适合自动化以及让 AI 跨许多文件自主工作。**GUI / 编辑器工具** (Cursor、

Windsurf、IDE 扩展) 活在一个可视化编辑器里：易于发现，有内联 diff、点击接受，以及一个平缓的上手坡道——审查改动更容易。

	TUI / 终端	GUI / 编辑器
速度	非常快，开销低	较慢，UI 更多
可脚本化	可以——管道、串联、自动化	有限
可视化 diff	纯文本	丰富，内联
可发现性	低（要知道命令）	高（可点击）
最适合	自动化、自主、高级用户	审查、上手、可视化工作

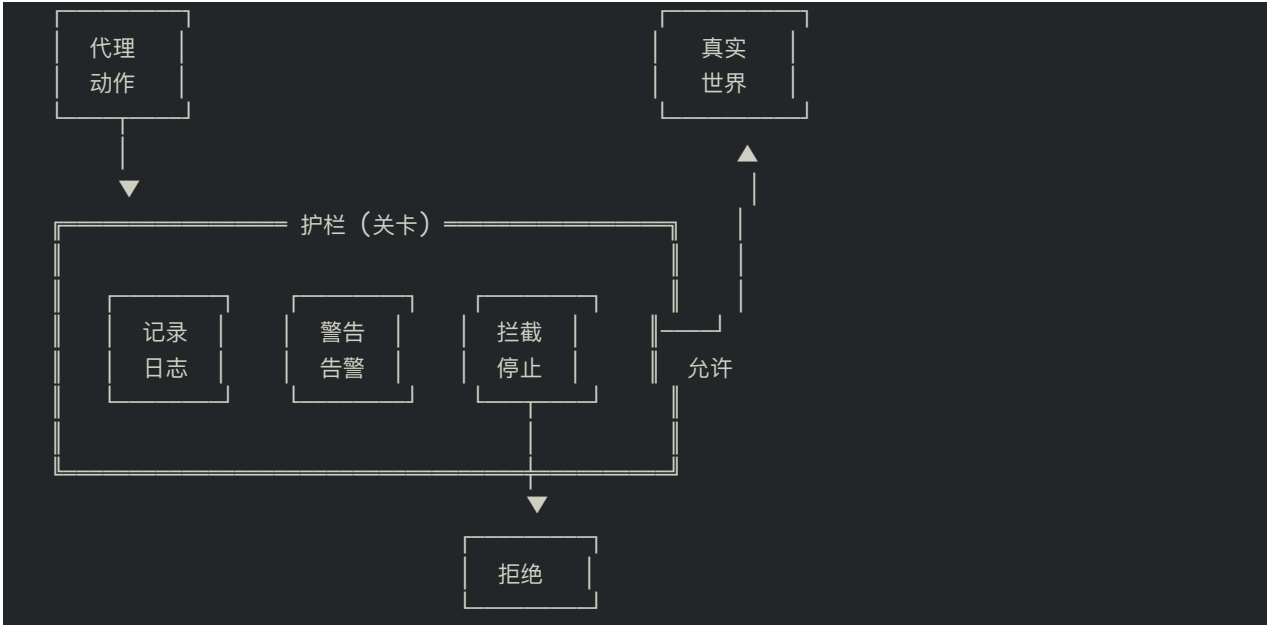
你不必永远只挑一个。组合才是关键所在：用一个快速的 TUI 智能体来推动大部分工作，然后切进一个 GUI 做可视化审查或细粒度的编辑。一个常见的工作流是，让一个终端智能体跨许多文件做出大范围的改动，然后打开 GUI 的左右对照 diff 去真正读一遍、批准它做的事——干活靠速度，核对靠清晰。挑一个用着舒服的当主力，并记住这本书里的技能在两者之间是通用的。

护具工程 (Harness Engineering)

习惯是你必须记得去做的事。在搭建工作空间时，下一个台阶是把这些检查变成 AI 无法跳过的东西。那就是护具 (harness)。

把护具想成一个立在智能体与真实世界之间的安检口。在每一个危险的步骤，它会做三件事之一：记录发生了什么、发出警告，或者干脆拦截这个操作。在命令运行之前、在文件被写入之前、在提示被提交时、在提交 (commit) 时、在推送 (push) 时——都有一道闸门。你可能会在凌晨两点忘了跑测试，但智能体绕不过它。

智能体采取的每一个危险操作都会先通过护具，每一道闸门都能在它触及真实世界之前记录它、警告你，或者拦截它：



为什么说它是必须的？氛围编程的根本赌注是”让 AI 全力冲刺，你来掌舵”。这只有在你眨眼的瞬间 AI 不可能冲下悬崖时才成立。护具把安全从警惕（读每一个 diff、记住每一条规则）转移到代码（自动强制执行的规则）。对任何要摆到真实用户面前的东西来说，这不是可有可无的打磨——它正是让自主变得安全的那个东西。

一个好的护具遵循几条值得了解的原则：

- 成功时安静，失败时响亮——一切正常时它什么都不说，所以一条警告才仍然有分量。
- 绝不自动修复——它提议、拦截或警告；它不会偷偷”纠正”智能体从而把问题藏起来。
- 添加新规则前，先退役休眠的规则——保持精简且可信，而不是变成一堆陈旧检查的堆积。
- 由配置驱动——一个引擎，按项目配置，能插入任何代码库。

- 只追加 (**append-only**) 的日志——一份记录智能体实际做了什么、且能识别篡改的账本。

你不必从零造这套东西。已经有现成的、即插即用的护具——`dancinlab/harness` 就是一个与项目无关的引擎，可以接到编码智能体的工具生命周期上。说句实话：护具并不能让 AI 变得正确，它只是让 AI 的错误变得便宜可抓——当你跑得很快时，这就是这场仗的大半。

成本安全：CLI vs API

一张账单就能毁掉一个项目，而它很少来自托管。它来自一个按量计费的 AI API。你为日常写代码而搭建好的工具——Claude Code、Cursor 之类——通常是按固定费率订阅的：多发一条 prompt 的成本实际上为零。可一旦你的应用去调用原始的 AI API 或 SDK (SDK 即软件开发工具包，是服务商提供的、预先打包好的代码库)，经济模型就翻转了：你是按 **token**、按请求计费的，没有上限。一个有 bug 的重试、一个无界限的批处理任务，或一个陷入循环的 agent，都能在一夜之间悄悄地把 \$0 的闲置账单变成一笔四位数的意外。这就是“bill shock”，也是单打独斗的构建者最常见的栽跟头方式。

在任何付费 API 上线之前，先给它套上笼子：

- 在提供商的控制台里设置一个硬性消费上限——一个真正的天花板，而不只是一封邮件提醒。
- 开发时优先用按固定费率的 **CLI**；把按量计费的 API 留给那个真正需要它的生产功能。
- 对你自己的调用做限流——限制并发数和每分钟请求数——并在头几天盯紧用量面板。
- 批量工作用便宜/小型的模型，并先在极小的输入上测试（一条记录，而不是整张表）。
- 绝不上线一个调用付费 API 的、不带节流的循环。给每个循环都设上界。

从一开始就把它定为规矩：每一个付费 API 都有一个硬性上限和一个节流器。

回顾与练习

核心要点

- 一个好的工作空间给 AI 上下文、快速的反馈循环和廉价的撤销——配置才是那个乘数，往往比模型本身更关键。
- 上下文文件 (AGENTS.md / CLAUDE.md) 是你写下的最高杠杆的东西：具体的规则、大声的禁止事项，以及指向示例的指针。
- 让版本控制成为本能——小步提交、读 diff、危险的活儿开分支——这样 AI 做的任何事都不会真正不可逆。
- 让循环保持紧凑，并让 agent 自己闭环：编辑、运行、读失败、修复。把诊断 (LSP、lint、类型检查) 反馈给 AI。
- 用硬性上限和节流器把每一个付费 API 关起来，把密钥留在仓库之外，并用 harness 让安全检查成为 AI 跳不过去的东西。

动手试试

打开你正在做的项目（或脚手架一个空的），亲手写一份 AGENTS.md 初稿：这个项目是什么、技术栈、运行 dev/test/lint 的确切命令、三条真实的约定，以及至少两条大声的禁止事项（例如“绝不读取或打印 .env”“不先打招呼不要加依赖”）。然后让你的 AI 工具做一个微不足道的改动，看它是否尊重这份文件。每当它做了你不想要的事，就加一行或把一行写得更锋利。你在调的是房间，而不只是代码。

本章提示词

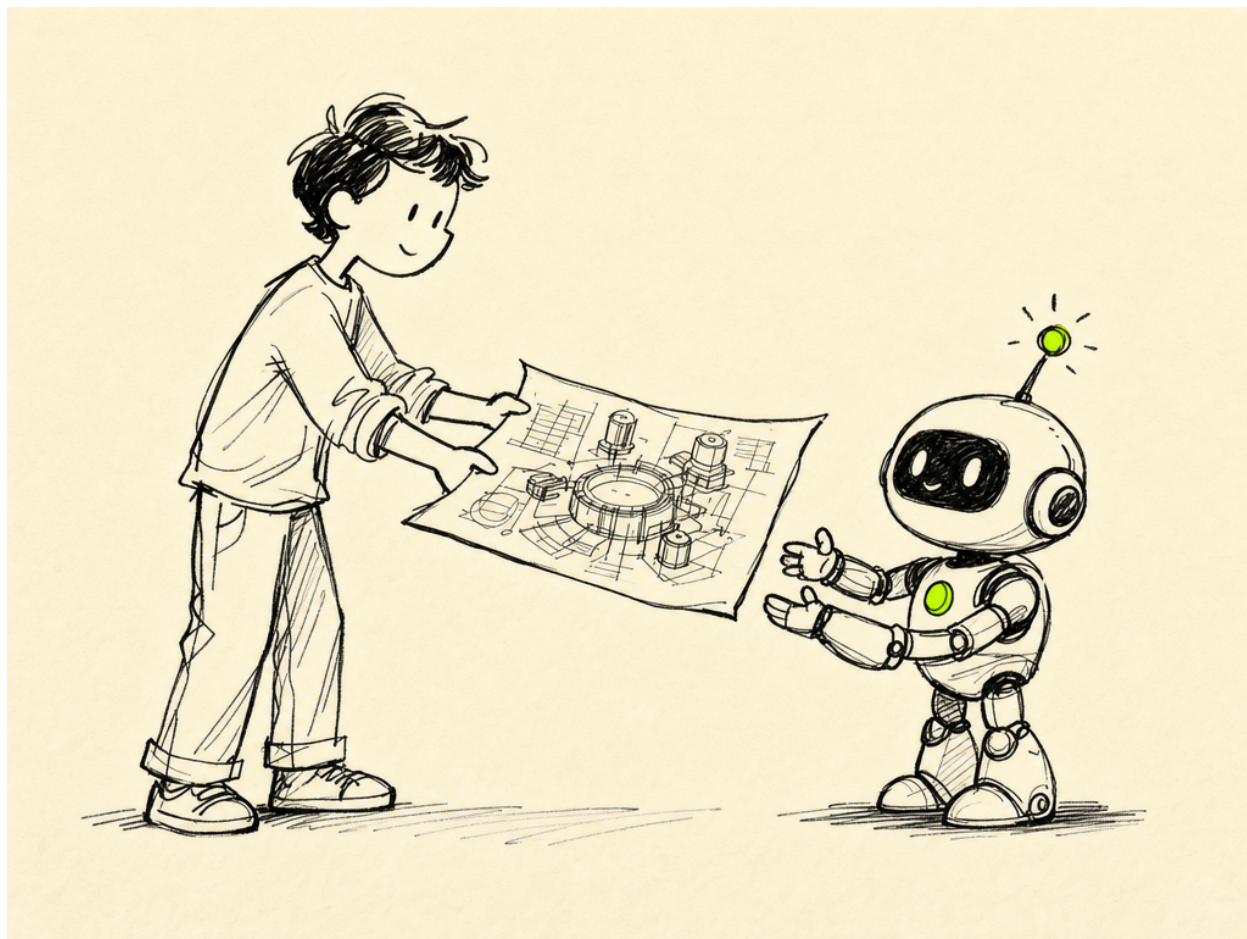
```
Read my project and help me write a concise AGENTS.md context file for it.
```

```
First, inspect the repo and tell me:
```

- the tech stack and versions you detect
- the commands for install / dev / test / lint / build
- the layout and where the entry point is

```
Then draft an AGENTS.md with these sections: What this is, Tech stack, Commands, Conventions, and Don'ts. Be specific and concrete (name the exact libraries and rules, not "follow our conventions"). State the don'ts as loudly as the do's, and point to one existing file as a style example. Keep it under ~40 lines — I'll prune from there.
```

像工程师一样写提示词



一次令人沮丧的 AI 会话和一次高产的会话之间的区别，很少在于模型。它在于提示词。大多数新手把 AI 当成搜索引擎——他们敲下一个模糊的愿望，然后听天由命。资深工程师把它当成一个犀利但死板的初级队友：一个又快又能干的人，但他会精确地构建出你描述的东西，包括你的错误在内。

本章讲的是弥合那道鸿沟。你不需要记住什么技巧，也不需要收藏一整个文件夹的“魔法”提示词模板。你需要像一个好工程师交代任务那样去沟通：带着上下文、约束，以及一个清晰的“完成”的定义。下面的一切都是这同一项技能的变奏。

在给出任务之前先给出上下文

AI 没法像你那样看到你的项目。它不知道你的框架版本、你的文件结构，或者你早已确立的惯例。当你跳过这些时，它就靠猜——而且猜得很糟。更糟的是，它自信地猜。所以在它撞上你真实的代码库之前，输出看起来都挺像样。

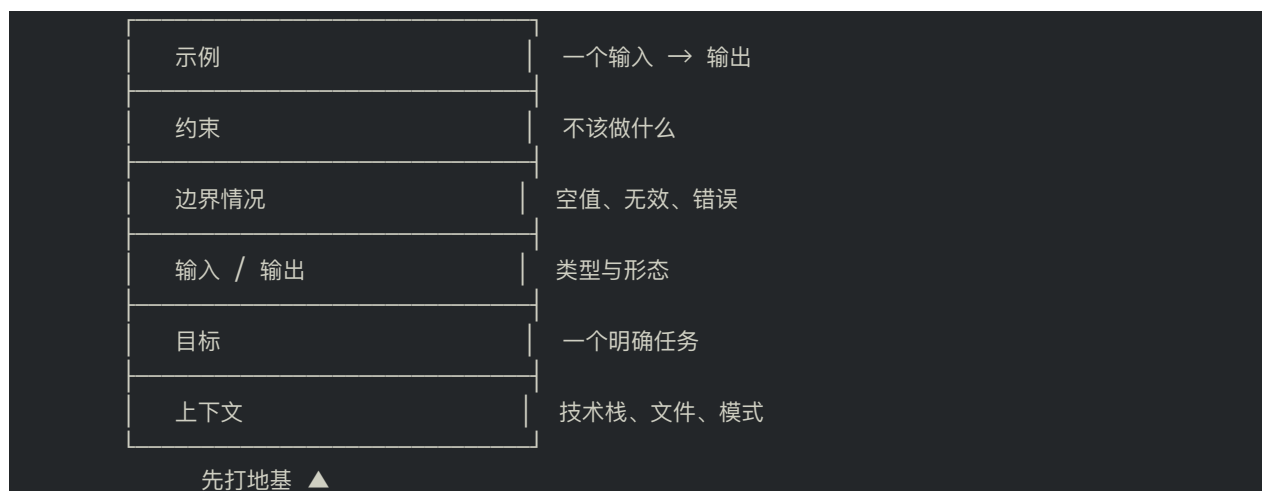
对比同一个请求的这两个提示词。

```
添加一个校验邮箱的函数。
```

```
这是一个 TypeScript 后端，使用 Express，并用 Zod 做校验。  
我们已经在 src/schemas/ 里用 Zod schema 校验输入。  
按那种风格添加一个邮箱校验 schema。邮箱必须是  
小写、最多 254 个字符，并拒绝来自 src/config.ts 里  
现有 BLOCKED_DOMAINS 列表的一次性域名。
```

第一个提示词产出泛泛的代码，它可能用了一个你不想要的正则，风格也和你的代码库不匹配。第二个产出的代码你可以直接粘贴进去。规则是：陈述技术栈、惯例，以及 **AI** 应该复用的现有部件。

一个好的提示词把这些部件自下而上按顺序堆叠——上下文在底层，真正的请求靠近顶层：



在你敲下回车之前，一份有用的心理清单：语言和框架（如果版本要紧就连版本一起），相关的文件或模块，项目里已有的模式，以及 AI 该偏好或该避开的库。不必每次都凑齐这四样——但如果答案会因其中之一而改变，就把它点出来。拿不准时，把新代码要适配的那个真实签名、类型或配置直接粘上去。几行现有代码教给 AI 的惯例，远胜过一整段描述。

指明输入、输出和边界情况

一个任务在它的边界被定义之前都不算被定义。资深工程师以“什么进去、什么出来、出错时会发生什么”的方式思考。把这些烤进提示词里。

```
写一个解析日期字符串的函数。
```

```
写一个函数 parseDate(input: string): Date | null。
```

输入：ISO 8601 字符串 ("2026-06-14"、"2026-06-14T10:30:00Z")。

输出：一个 Date 对象，若字符串无效则返回 null。

需要处理的边界情况：

- 空字符串或仅有空白 -> null
- 格式合法但日期不可能 (2026-02-30) -> null
- 末尾有垃圾字符 ("2026-06-14xyz") -> null

不要 throw。对所有无效输入都返回 null。

当你点名说出边界情况时，AI 就会处理它们。当你不说时，它写顺利路径，然后你在生产环境里发现那些缺口。同样的纪律也适用于纯函数之外：对一个端点，要指明成功响应以及错误响应（重复的邮箱返回什么状态码？请求体格式错误时呢？）。对一个 UI 组件，要指明空状态和加载状态，而不只是有数据的状态。大多数软件有意思的行为都住在它的边缘，所以你的提示词也应当在那里花笔墨。

如果你不确定边界情况有哪些，就先问 AI：“一个日期解析器应该处理哪些边界情况？”——然后把答案当作约束喂回去。这把模型从一个代码生成器变成了一个清单生成器，而审查一份清单，比调试你忘掉的那些情况要快。

以小步前进

最常见的单一错误，就是一次要求太多。“给我搭一个带认证、图表和设置页的用户仪表盘”会产出一堵你无法审查的代码墙，bug 散落在你没读过的文件里。

小步前进让你保持掌控：

- 每个提示词只交付一个函数、组件或端点。

- 在往下走之前先验证每一块都能用。
- 在确认能用的代码之上构建，而不是在一座未经验证的输出之塔上。

把整个认证系统搭出来，包括登录、注册、密码重置、会话，以及邮箱验证。

认证流程的第 1 步：只写注册端点。
POST /signup 接收 { email, password }。用 bcrypt 对密码做哈希，存储用户，返回 201 并带上用户 id。
假设 User 模型和 db 客户端已经存在（我会给你看它们的形态）。登录我们下一步再处理。

更小的提示词意味着更小的 diff，而更小的 diff 意味着你真的能读懂改了什么。这就是全部的游戏所在。还有第二份回报：当某处坏掉时，一个小步骤做二分定位易如反掌。如果你生成并验证了注册端点，然后是会话逻辑，再然后是密码重置，那么第三步之后冒出来的 bug 几乎肯定住在第三步里。一锅端式的生成丢掉了这个信号——bug 可能藏在你从没读过的四百行里的任何地方。

在写代码之前先要一份计划

对于任何不平凡的事，把思考和打字分开。先让 AI 勾勒出它的思路。这能在错误的假设变成 200 行错误代码之前就抓住它，而修一份计划要比修一个 pull request 便宜得多。

我想给我的 API 加上限流。在写任何代码之前，给我一份简短的计划：限流器该放在哪里、需要什么样的存储、达到上限时会发生什么，以及有哪些取舍。先别写代码——我会先批准方案。

当计划看起来对了，就说“好，实现第 1 步”。当它不对时，你花了三十秒纠正航向，而不是花一小时去解开一团乱码。先做计划，是最能把工程师提示词与一厢情愿式提示词区分开来的习惯。

计划阶段也是你抓住 AI 选了比所需更重的方案的地方——明明一个内存里的 map 就够，它却伸手去够 Redis；明明一行库调用就能解决，它却手搓一套。带着一个问题去读计划：这是能跑通的最简单做法吗？如果这个思路闻着像过度设计，就在一行代码落地之前用大白话顶回去（“这是个单服务器的业余项目——能不能跳过分布式存储？”）。

在 diff 上迭代，而不是重写

一旦代码存在了，就要抵住把整个东西重新要一遍的冲动。像“把它弄好一点”这种模糊的追问会扔掉能用的代码，并重新引入你早已修过的 bug。指向你想要的那个具体改动。

这不对，把整个东西重写一遍。

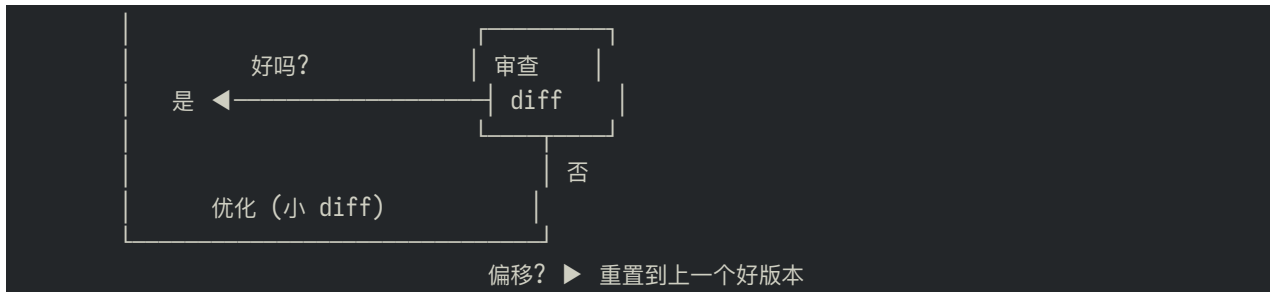
函数能用，但有两个问题：

1. 那行 `throw new Error` 的错误应当返回一个 `Result` 类型，而不是抛出——和上面 `validateUser` 函数里的模式保持一致。
2. 循环每次迭代都重新读 `items.length`；把它提到循环外。

只给我看这两处改动的 diff。

健康的循环是一个紧凑的回路：向模型提示，读取输出，然后用一个小而有针对性的 diff 来打磨——绝不整体重写——直到它对为止：





要一份 diff（而不是一次完整的重写）能让改动保持可审查，并保住那些已经能用的部分。把 AI 的输出当作一位同事的 pull request：对具体的行做评论，要求有针对性的编辑。当一个改动跑偏，代码随着每次回复离你想要的越来越远时，别再一味打补丁——回退到你信得过的上一个版本，再用一段更锋利的描述从那里重新提示。在一个坏掉的基础上一路向前迭代，只会把烂摊子越叠越大。

提供示例

一个具体的示例抵得上一段描述。如果你想要某种特定形状的输出，就把那个形状给它看。这是消除歧义最快的方法。

把数据格式化得漂亮些。

把这个用户数组转换成一个汇总对象。

输入：

```
[{ name: "Ann", role: "admin" }, { name: "Bo", role: "user" }]
```

期望输出：

```
{ total: 2, byRole: { admin: ["Ann"], user: ["Bo"] } }
```

写出转换函数，以及一个与该示例匹配的测试。

示例钉死了命名、结构和大小写——这些光靠文字会留得模糊不清。当你能做到时，把示例作为一个测试包含进去——这样“让这个测试通过”就成了一条没有歧义的指令。这个套路可以推广：一份样例 API（应用程序编程接口——一个程序向另一个程序请求数据的方式）响应钉死一个解析器，一对“改前/改后”钉死一次重构，一张截图或 ASCII 草图钉死一个布局。每当你发现自己正在写一段对某个形状的描述时，停下来，改为给它看那个形状的一个实例。

告诉 AI 不要做什么

约束不只关乎你想要的结果——它也关乎你想让 AI 避开的招数。不加约束地放任，模型就会朝着添加依赖、把你没让它碰的代码“热心地”重构、在你请求的边缘发明功能的方向漂移。几条否定式约束能让输出保持紧致。

给搜索输入框加一个防抖。

给搜索输入框加一个防抖。

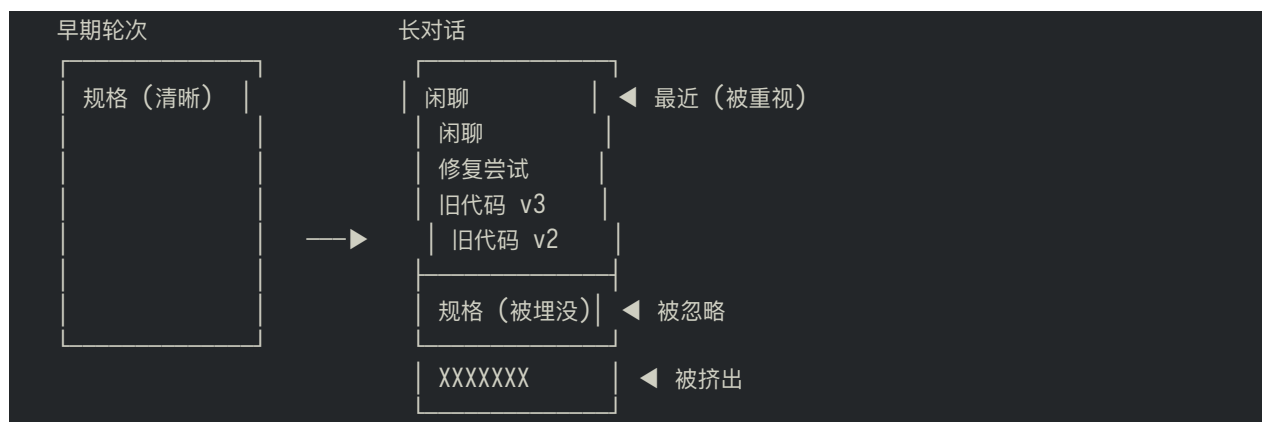
约束：

- 不要引入新依赖——我们已经有 lodash 了，用它。
- 只改 SearchBar.tsx。不要重构父组件。
- 保持现有的 prop 名称和类型不变。
- 300ms 延迟。

第二个提示词预先封死了第一个可能出错的十几条路。否定式约束在既有代码库里尤其值钱，因为在那里，一处没让它做的改动的代价，不只是审查它——而是合并冲突、被吓一跳的队友，以及本来好好的代码出现的回归。如果有一条你不想被越过的线，就明说出来；AI 不会从客气里推断你的边界。

像管理资源一样管理上下文

一段长对话不是免费的。随着线程变长，更早的指令会被埋没，模型会更倚重最近的消息，二十轮之前你陈述过的细节会悄悄不再要紧。把上下文窗口想象成一个固定大小的盒子：随着对话变长，它逐渐装满，而你最早的指令被推向模型最少留意的底部：



把上下文窗口当作一块你有责任保持整洁的工作台，而不是一份无限的记忆。

几个习惯能帮上忙：

- **转向时重述规格。**别假设十条消息前的某个要求还成立。把当前目标总结成一小段，让 AI 从那里开始工作。
- **新问题就重开。**当你从认证代码切到一个 CSS（层叠样式表——控制页面外观的语言）bug 时，开一段新对话。拖着不相关的上下文，只会让模型更差，而不是更好。
- **粘当前代码，而不是历史。**如果一个函数改过五次，AI 不需要那五个旧版本——它需要现在存在的那一个。把当前状态和下一处改动给它。
- **在对话之外留一份简短的事实来源。**对一个较长的任务，文件里的几行（“技术栈、关键决定、已完成、下一步”）能让你在几秒内把一段新对话重新播种，而不必从滚动记录里一点点重建。

目标和贯穿本章的那个一样：在每一刻，AI 都应当从一幅准确、紧凑的任务图景出发工作——而不是一份它得连蒙带猜才能穿过的、漫无边际的对话记录。

要避免的常见反模式

这些是悄无声息浪费你时间的提示词习惯。在你自己的会话里留意它们。

- **模糊的愿望。**“做一个好的登录页。”按谁的标准算好？把形容词换成具体项：哪些字段、哪种校验、哪个框架。
- **巨型提示词。**把一整个功能倒进一条消息里。把它拆成你能一次验证一个的步骤。
- **沉默的上下文。**假设 AI 记得或知道你的项目。重新陈述相关的文件、类型和惯例——尤其是在一次全新的对话里。
- **盲目接受。**不读就粘贴生成的代码。AI 即便错了也很自信；你仍然是那个署名在案的工程师。
- **没完没了的重试。**一遍遍跑“再试一次”，指望出现不同的输出。如果两次尝试都失败，那问题出在提示词上——加上下文或约束，别只是重摇骰子。
- **移动的靶子。**在对话中途改需求却不重新陈述它们。当你转向时，把当前的规格说明总结一下，这样 AI 才不是在一幅过时的图景上工作。
- **奉承循环。**把“你说得完全对！”当成修复正确的证据。同意不等于验证——自己跑一遍代码，亲自检查行为。

核心心态

这里的每一项技巧都归结为一个理念：**说出你本来得对一个无法读你心、但能力胜任的队友所要说的话。**上下文放在前面。清晰的输入和输出。小的、可验证的步骤。代码之前先有计划。有针对性的编辑胜过重写。在文字力不能及之处给出示例。对不许碰什么划下明确的界限。在每一刻都保有一幅整洁、最新的任务图景。

你不是在写魔法咒语。你是在写一份规格说明，简短而精确，然后审查结果。一贯地这么做，AI 就不再感觉像一台老虎机，而开始感觉像它本就能成为的那个快速、不知疲倦的工程师——而且同样重要的是，你始终是那个明白到底交付了什么的人。

回顾与练习

核心要点

- 给任务之前先给上下文：说明技术栈、约定、要复用的现有部件——或者直接贴上代码必须契合的真实签名。
- 定义边界：输入、输出、边界情况。软件有趣的行为都活在它的边缘，所以把你的字数花在那里。
- 用小而可验证的步骤推进，并在写代码前要一份计划；修计划远比修一个 `pull request` 便宜。
- 在有针对性的 `diff` 上迭代，而不是整体重写，并用具体的例子（最好做成测试）钉死那些文字留下的模糊之处。
- 把上下文窗口当作一种资源来管理，并说明不要做什么——明确的否定约束让输出保持紧凑。

动手试试

拿一个你平时可能直接打出来的模糊一行请求（“给列表加个搜索”），在发出任何东西之前先把它改写成工程师的提示词。加上上下文（技术栈、它要碰的文件）、输入/输出、两三个边界情况、一个期望结果的例子，以及一条否定约束（不要碰什么）。把改写后的版本发出去，和那个偷懒版本本会产出的东西对比。对三个真实请求都这么做，这种升级就会变成自动的。

本章提示词

Context: < 语言 + 框架 + 版本 >, 在 < 文件/模块 > 中工作。

We already < 可复用的相关现有模式或库 >。

Task: < 一件清晰要构建的事 >。

Inputs / outputs: < 类型与形态 >。

Edge cases to handle: < 列出要紧的那些 >。

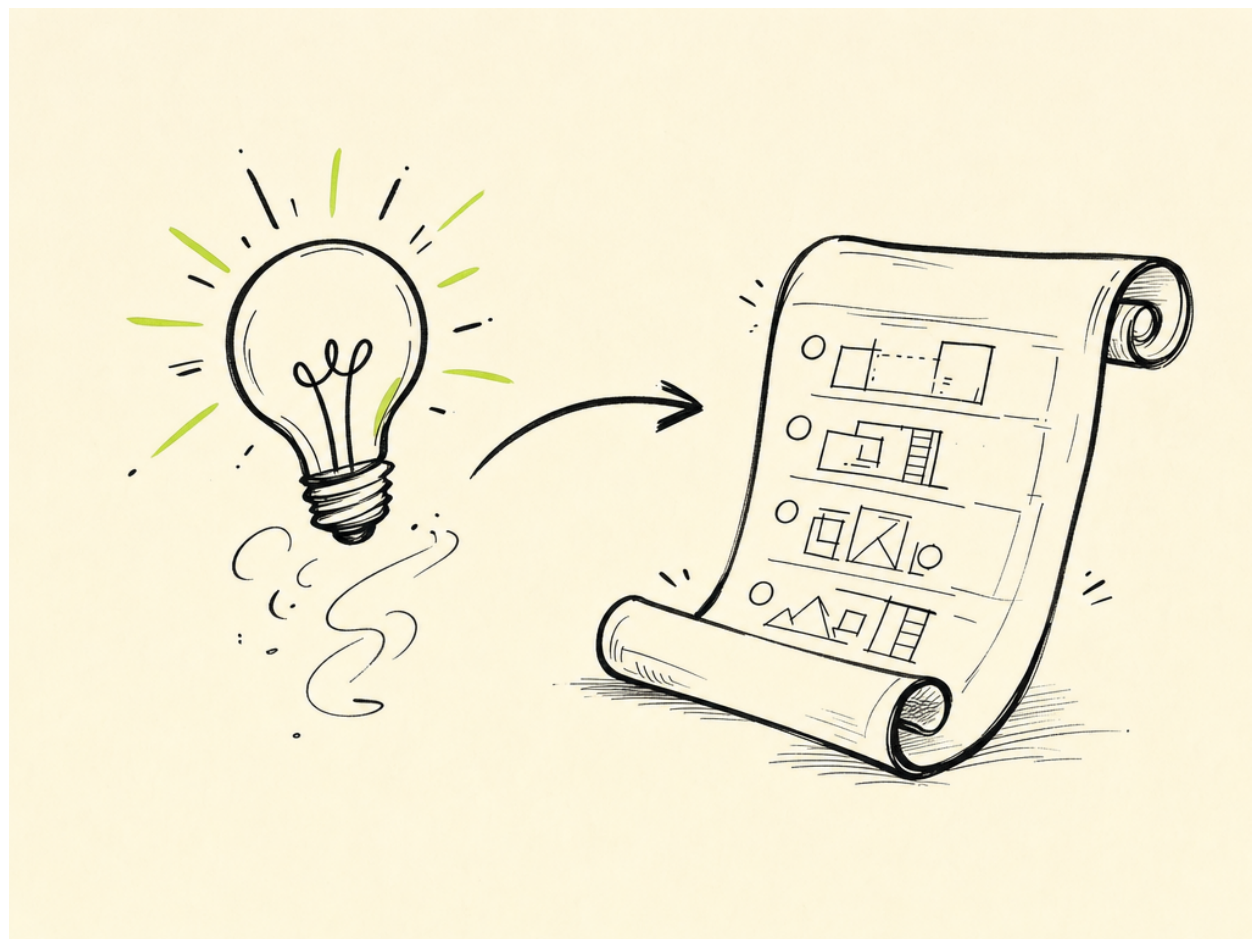
Example: < 一个具体输入 -> 期望输出 >。

Constraints:

- No new dependencies — use < 我们已经有的东西 >。
- Touch only < 这个文件 >; don't refactor anything else.
- Keep existing names and signatures unchanged.

Before coding, give me a 3-4 line plan and wait for my "go".

几分钟内从想法到规格



每个项目都始于一个模糊的念头：“我想要一个能帮我记录阅读的应用。”这是个不错的种子，但你无法直接据此动手开发——你的 AI 也不能。把一个含糊的想法交给 AI，产出的只会是含糊的软件：你没要的功能它加了，你要的部分它漏了，最后留给你一堆看不懂的代码。

解决办法是写一份**规格 (spec)**——一份简短、用大白话写成的文档，说明你要做什么，以及你不做什么。好消息是：有 AI 作为你的思考伙伴，你可以在几分钟而非几天内，把一句话的想法变成一份可落地开发的规格。本章就教你怎么做。

说清楚规格不是什么：它不是一份合同，不是一份你要在会议上据理力争的设计文档，也不是写一次就冻结起来的东西。它是写给你自己和你的 AI 的一份工作笔记。你会随着学到新东西而不断修改它。目标不是一份完美的文档——而是一份足够好的文档，让接下来一小时的开发比上一小时更清晰。

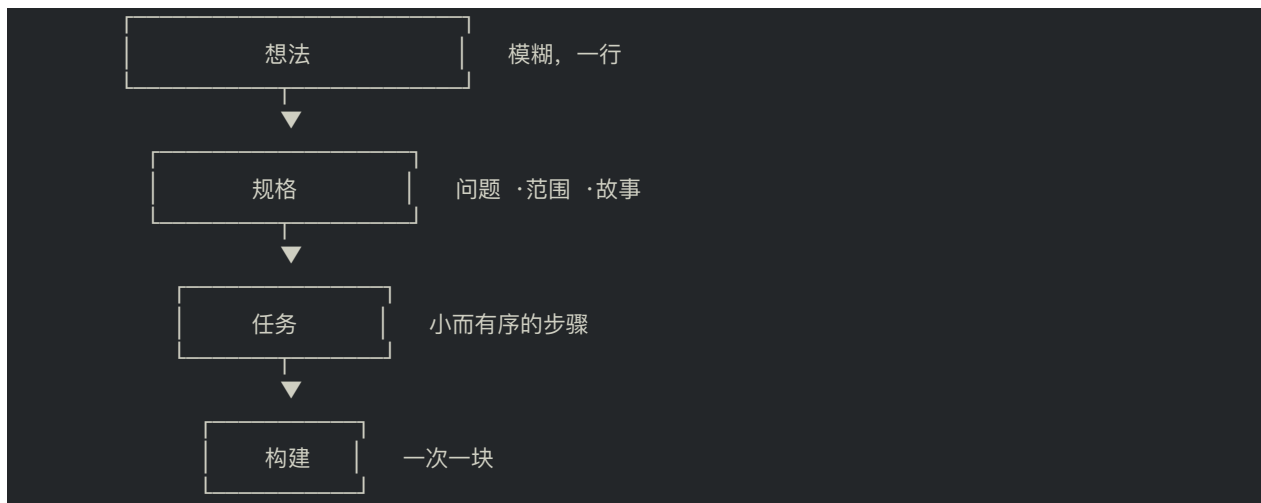
为什么要规格，而不只是凭感觉

Vibe coding 并不意味着跳过规划。它意味着把规划做得快而轻，然后指挥 AI 去执行。规格就是方向盘。没有它，AI 在每次提示时都得猜你的意图，而一个个小误解会层层累积成一团乱麻。

规格给你三样东西：

- **共享上下文**——你可以把它粘贴进任何 AI 会话，立刻重新确立你在做什么。
- **范围边界**——一条写下来的“做”与“不做”之间的界线，能遏制功能膨胀。
- **任务清单**——可逐个完成的、大小适中的开发块。

规格的运作像一个漏斗：它接住一个宽泛、模糊的想法，一步步收窄，变成 AI 真正能开发的小块。



它不需要很长。对大多数项目来说，一页就足够了。事实上，一份长到超过两页的规格，通常是个警示信号：要么你规划得太超前了，要么你在悄悄扩大范围。拿不准时，就删。被你删掉的部分很少会被想起，而留下来的部分会变得更锋利。

第一步：厘清问题

在谈功能之前，先点明问题以及遇到这个问题的人。忍住直接跳到解决方案的冲动。一份清晰的问题陈述，会让之后的每一个决定都保持诚实。

问问你自己（或者让 AI 来采访你）：

- 这是为谁做的？（要具体——“我自己”“独立开发者”“我的跑团”。）
- 在没有这个东西之前，他们今天正忍受着什么痛苦的事？
- 用一句话来说，成功是什么样子？

你大可以让 AI 来挖掘。试试这个提示：

我有一个粗略的想法：一个记录我在读的书的网页应用。
扮演产品思考伙伴。向我提出 5 个犀利的问题，
一次一个，以厘清问题、目标用户，以及
“完成”是什么样子。先别提议功能——只管追问
这个想法，直到它变得具体。

回答五个好问题，通常就足以把一个模糊的想法打磨成实在的东西。诀窍在于让 AI 反驳你，而不是对一切都点头同意。如果它问“你需要跨设备同步吗？”而你诚实的回答是“不需要，只在我的笔记本上用”，那你刚刚就删掉了一个后端、一套认证系统，以及好几周的工作量——而这一切都在写下一行代码之前。这正是关键所在：砍掉一个功能最便宜的地方是在对话里，而不是在代码库里。

留意这样一个时刻：当你的回答从“嗯，大概吧”变成“对，就是这样”。这种转变就是信号，说明你找到了真正的问题，可以往下走了。

第二步：定义范围与非目标

这是最有价值、也最常被跳过的一步。**非目标**——那些你刻意不做的事——能保护你不被一个总爱添加“贴心”附加功能的 AI 带偏。

对我们的图书追踪器来说，范围可能是：

- **范围之内**：添加一本书，标记为在读/已读完，查看列表，给读完的书评分。
- **范围之外（非目标）**：社交功能、推荐、从 Goodreads 导入、移动端应用、用户账户。

想象两栏：左边是你承诺要做的一切，右边是你刻意跳过的一切。它们之间的那条线，就是让 v1 保持小巧的边界。

范围内	非目标
添加书籍	社交功能
标记在读/完成	推荐
查看列表	导入 Goodreads
给读完的书评分	移动应用
	用户账户

现在做 ←————→ 暂不做

把非目标写下来有两重作用。它让你的第一个版本小到真能做完；同时，当 AI 跑偏时，它给了你一句可以直接粘贴的话：“记住，账户系统是 v1 的非目标——不要加登录。”

非目标不是“永远不做”——而是“现在不做”。把用户账户列为非目标，并不意味着你的应用永远不会有它，而是说它不属于这个版本，所以你今天不用为它付出代价。这种重新表述很重要：它让你能够对一个功能说不，却不会觉得自己在放弃一个好点子。你只是在排顺序。你非目标清单里的大多数条目，其实都是伪装起来的 v2 待办列表，这没什么不好——把它们写在某处，然后继续往前走。

第三步：编写用户故事

用户故事把范围转化为可测试的行为。格式很简单：

作为一名 **[用户]**，我想要 **[执行某动作]**，以便 **[获得某好处]**。

对图书追踪器来说：

- 作为一名读者，我想要按书名和作者添加一本书，以便我能开始追踪它。
- 作为一名读者，我想要把一本书标记为已读完，以便我能看到我的进度。
- 作为一名读者，我想要给读完的书打 1–5 星，以便我记得自己喜欢什么。

每个故事都是一个可独立开发并验证的单元。如果你想象不出怎样靠在应用里点点点来测试一个故事，那它就太含糊了——拆开它，或者重写它。

so that（以便）从句是人们最常丢掉的部分，而它恰恰是最重要的。“作为一名读者，我想要一个搜索框”告诉了 AI 要做什么，却没说为什么，于是它没法做出好的取舍。“作为一名读者，我想要通过输入书名的一部分在我的列表里找到一本书，以便我不必滚动翻过几十条记录”告诉了它真正要干的活——这下，一个简单的即输即筛或许就胜过一个重量级的搜索引擎。好处就是这份规格的规格：它把功能解释得足够清楚，让 AI（以及未来的你）能分辨出好的实现和错的实现。

第四步：起草一份轻量 PRD

PRD（产品需求文档）不过是把你的决定汇总到一处。它有一副可预测的骨架——每次都是那几个小节：

PRD	
— 问题	为何存在
— 用户	为谁而做
— 目标	做什么，一句话
— 范围内	v1 包含什么
— 非目标	v1 略去什么
— 用户故事	作为 __，我想 __，以便 __
— 数据模型	你存储的字段
— 完成标准	验收测试

保持它精简。下面是一份完整的入门级 PRD，你可以直接交给 AI：


```

# Book Tracker — Mini PRD

## Problem
I read a lot but lose track of what I've finished and what I thought of it.

## User
Me (a single user). No accounts needed for v1.

## Goal
A simple web app to add books, track reading status, and rate finished ones.

## In Scope (v1)
- Add a book (title, author)
- Set status: To Read / Reading / Finished
- List all books, filterable by status
- Rate finished books 1–5 stars

## Non-Goals (v1)
- User accounts / login
- Social features, sharing, recommendations
- Importing from other services
- Mobile / native apps

## Tech (keep it simple)
- Single-page web app
- Local browser storage (no backend for v1)

## Done When
I can add a book, change its status, rate it, and see my list
persist after a page refresh.

```

注意那行 **“Done When”** (完成的标志)。那是你的验收测试——一个判断你是否已交付 v1 的具体方式。永远要写上一条。一个好的“Done When”，是你能亲手做到的事：点这里、输入这个、刷新、看到那个。如果你的完成条件读起来像一种感觉（“等它打磨好了”“等它感觉对了”），那它永远完不成，因为没有什么可供你对照检查。把它写成一份你能交给陌生人去照做的脚本。

你不必手写这份 PRD。一个有用的做法是，把你前面几步的回答喂回给 AI，让它来拼出草稿，然后你来编辑。之所以是编辑而不是照单全收，是因为你必须理解并认同每一行——这份文档是你反复粘贴回去的事实来源，所以这里的一个错误会被复制进之后的每一次会话。

第五步：勾勒数据模型

大多数应用其实都是关于数据的：你存什么，以及各部分之间如何关联。快速勾勒一份数据模型，能让你日后免于混乱、前后不一致的代码。你不需要画数据库图——列出字段就够了。

对图书追踪器来说，主要只有一样东西——一个 Book：

```

Book
- id: 唯一字符串
- title: 字符串
- author: 字符串
- status: "to_read" | "reading" | "finished"
- rating: 数字 1–5（仅在读完时）
- addedAt: 日期

```

哪怕这么小的一份草图，也能让决定变得明确：评分只对读完的书存在，状态是三个固定值之一，而每本书都有一个 id。当你把这个交给 AI 时，它就不再猜测字段名，而会前后一致地构建。

之所以要趁早把这个钉死，是因为字段名会渗透到处处。一旦 AI 写出读取 `book.rating` 的代码，这个名字就会出现在表单、列表视图、存储层，以及你日后添加的任何功能里。如果你任由它漂移——此处叫 `rating`、那处叫 `stars`、第三处又叫 `score`——你就会得到一些追查起来很烦人的 bug。一份六行的数据模型，是防止这种事的最便宜的保险。

一段简短、可复制粘贴的规格块，对 AI 而言往往比散文更管用。下面是同一个模型，写成你真正会发出去的提示：

书籍请使用这个精确的数据结构。不要添加我没列出的字段，也不要重命名这些：

```
Book {
  id      —唯一字符串，创建时生成
  title   —字符串，必填
  author  —字符串，必填
  status  —取值之一: "to_read" | "reading" | "finished"
  rating  —整数 1-5, 仅当 status 为 "finished" 时存在
  addedAt — ISO 日期字符串，添加书籍时设置
}
```

如果你觉得缺了某个字段，添加前先问我。

最后那一行——*ask me before adding it*（要加之前先问我）——在实打实地起作用。它把 AI 从一个猜测者变成一个协作者，并让数据模型始终归你所有。

第六步：把规格拆成 *vibe* 大小的任务

最后一招是把你的规格切成 AI 能逐个执行的任务。一个 *vibe* 大小的任务，要小到你能在一分钟内审完结果，并判断它是否奏效。如果一个任务让 AI 改了好几个文件，而你又说不清它对不对，那它就太大了。

图书追踪器的好任务：

1. 搭好基本的页面布局，带一个页头和一个空的图书列表。
2. 做出“添加一本书”的表单（书名 + 作者），将其追加到内存中的列表里。
3. 加上状态（待读 / 在读 / 已读完），并用一个下拉框来切换它。
4. 把列表保存到浏览器本地存储，让它在刷新后依然存在。
5. 加一个只对已读完的书出现的评分控件。
6. 在列表上方加上状态筛选按钮。

每个任务都对应着一个用户故事或一行 PRD，建立在上一个之上，并以你能看见其运作的成果收尾。这就是 *vibe coding* 的节奏：小任务、跑起来、验证、下一个。

你甚至可以让 AI 替你生成这份拆解：

这是我的迷你 PRD 和数据模型 [粘贴在此]。
把它拆成 6-8 个小而有序的构建任务。每个任务都应
可独立测试，并建立在前一个之上。每个任务
保持在我点击应用就能验证的范围内。

让规格保持鲜活

规格不是一座你建起一次就供着观赏的纪念碑。它是一份活文档，而最常见的错误就是把它当成固定不变的。随着开发推进，你会学到一些规划阶段无法告诉你的事：某个功能比预想的更难，某个非目标原来不可或缺，某个用户故事其实暗藏着两个故事。当这种情况发生时，先改规格，再让 AI 去遵循新版本。规格在前，代码在后。

有几个习惯能让它保持诚实：

- **决定一变，立刻更新它。**一份嘴上说一套、而代码里做着另一套的规格，比没有规格更糟——它会主动误导下一次 AI 会话。
- **完成一个任务就划掉一个。**任务清单同时充当进度追踪器。看到八项里勾掉了六项，本身就是一种势头。
- **把意外挪进非目标或一份“以后再说”清单，而不是直接就把它们做了。**如果开发途中冒出一个诱人的功能，写下来，然后继续往前走。你以后可以再决定它是否配得上 v1 里的一席之地。

正是这种自律，把“我做了个东西”和“我做出了我本想做的那个东西”区分开来。规格是你让自己继续当项目作者、而不是沦为项目乘客的方式。

一个常见陷阱：规定的是解决方案，而非问题

有一种失败模式值得单独拎出来说，因为它实在太容易掉进去了。当你坐下来写规格时，诱惑在于去描述你脑中已经想好的那个解决方案——“一个带标签页的侧边栏”“一个弹出来的模态框”“一张 Postgres 表”。但一份满是解决方案的规格，会悄悄锁定一些你还没资格去做的决定。

对比这两行：

差： 添加一个左侧边栏，每个书籍状态有可折叠的分区。
好： 让我能按状态分组查看书籍，又不丢失完整列表。

第一句已经选定了一个侧边栏、可折叠的分区，以及一种布局——而这一切都在还没人确认它们是不是正确的选择之前。第二句陈述的是要完成的活，让 AI（以及你）有自由去找到能满足它的最简单方案，而那也许只是三个筛选按钮。描述你想要的结果；让实现尽可能长久地保持可商量。解决方案在规格里改起来很便宜，在代码里改起来却很昂贵。

回报

现在你有了一份问题陈述、范围与非目标、用户故事、一页 PRD、一个数据模型，还有一份有序的任务清单——一个真实项目的全部根基，全都在几分钟内、以 AI 为你的回声板起草完成。

这一切都不要求你会写代码。它要求的是你知道自己想要什么，并清晰地思考几分钟——剩下的思考，AI 会陪着你出声地完成。这才是真正的转变：你还没在写软件，你在写意图，而好的意图就是这场仗的大半。

把这份规格留在项目里的一个文件中（许多人用 SPEC.md 或 PRD.md）。它会成为你粘贴进全新 AI 会话、用来重新锚定上下文的那份文档，也会成为你逐个 vibe 大小任务往下做的那张清单。在下一章里，我们将拿起第一个任务，真正把它做出来。

回顾与练习

核心要点

- 规格是一份简短、平实语言的工作笔记——你在做什么、不做什么——而不是一份要冻结的合同；一页就够了。
- 在功能之前先厘清问题和拥有这个问题的人，并写下非目标（non-goals）：它们是抵挡 AI 添加“贴心”附加功能的最锋利防线。
- 带 so that 子句的用户故事，加上有具体“Done When”那一行的小型 PRD，把意图变成可测试、可构建的行为。
- 尽早钉死数据模型——字段名会渗漏到处都是——并把规格切成你能一次验证一个的 vibe 大小任务。
- 描述问题，而不是解法；结果在规格里改起来便宜，在代码里改起来昂贵。一边学一边让规格保持鲜活。

动手试试

拿一个你一直搁着的想法，在一次 AI 会话里把它变成一页规格。先让 AI 用五个尖锐的问题对你做访谈，然后——用你自己的话——写下问题、用户、三个用户故事（每个都带 so that）、一份明确的非目标清单、一个六行的数据模型，以及一行你能物理执行的“Done When”。存成 SPEC.md。好规格的检验标准：明天你把它粘进一个全新的对话，AI 就能确切知道该构建什么。

本章提示词

我有一个粗略的想法：< 一行描述 >。

扮演我的产品思考伙伴。首先，向我提出 5 个犀利的问题，一次一个，以厘清问题、目标用户，以及“完成”

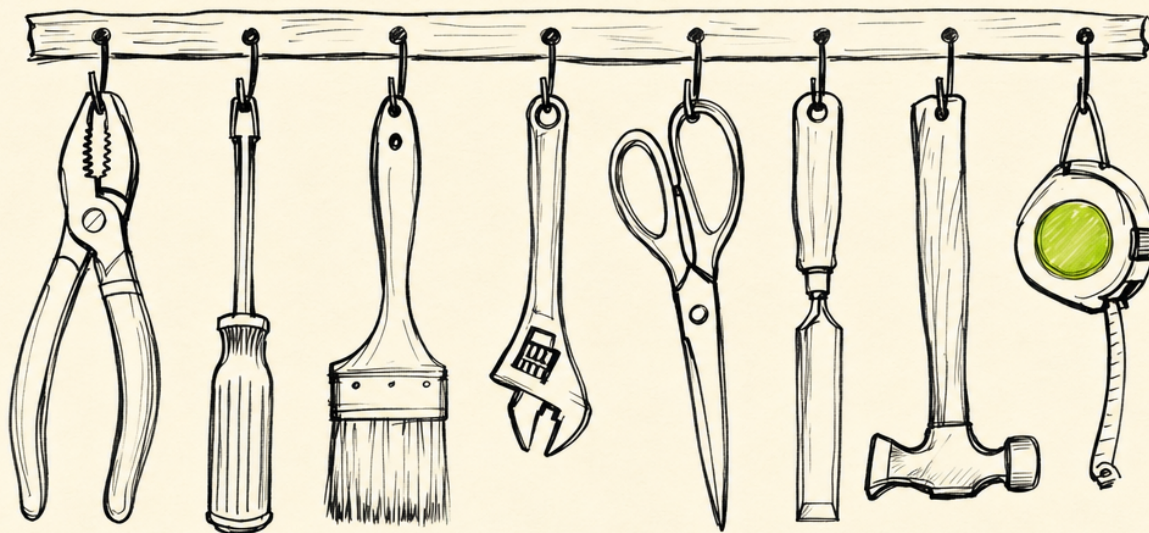
是什么样子。先别提议功能。

我回答后，按这些小节起草一页的规格：

- 问题 - 用户 - 目标
- 范围内（v1） - 非目标（v1）
- 用户故事（每条" 作为 __，我想 __，以便 __"）
- 数据模型（字段的简短列表）
- 完成标准（具体、可实际执行的验收测试）

保持精炼——一页以内。若有含糊之处，先问再假设。我会修改草稿，直到我认同每一行。

选择你的技术栈：语言一览



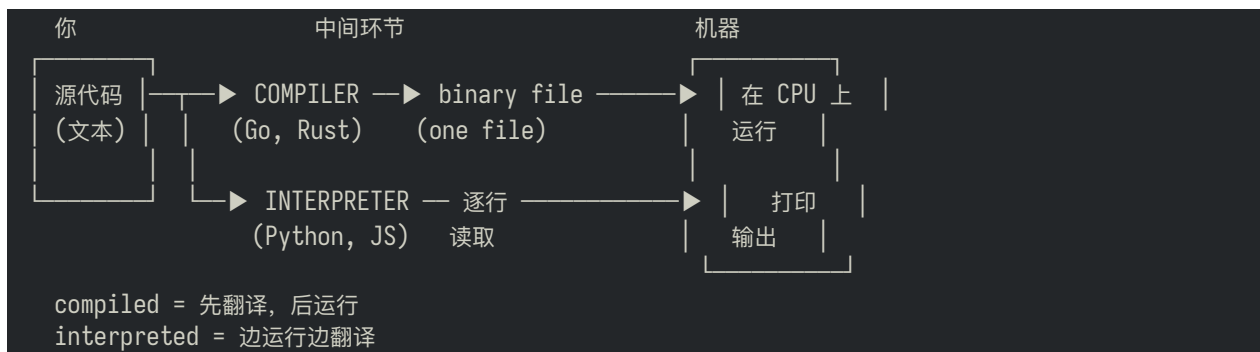
当你指挥 AI 来开发软件时，你挑选的语言依然很重要——非常重要。代码是 AI 写的，但后果由你承担：你交付有多快、部署有多容易、多久会撞上一堵墙。好消息是，你不需要精通这些语言中的任何一门。你只需要懂得足够多，能做出一个明智的选择，并能识别出 AI 何时正把你引向一条痛苦的路。

本章会逐一过一遍值得关注的语言，讲清每一门真正擅长什么、在哪里会咬人，以及——这对 vibe coding 至关重要——AI 助手对它的驾驭能力如何。最后我们会给出一份按目标选择的极简指南。

开始前先提一句：AI 助手对那些拥有海量公开代码的热门语言，要远远强于对小众语言的处理。这可不是个小差别。它往往是决定你的项目是否顺利的最大单一因素。流行度本身就是一项功能。当一门语言及其主要框架在公开仓库里出现过几百万

次时，AI 就见到了每一种常见模式、每一个典型错误、每一种地道的修法。如果没有，你就要为这份稀缺埋单——以失败的尝试和听起来煞有介事的胡话来偿付。

在看语言名字之前，先记住一幅图：你写的代码，从来不是机器真正运行的东西。中间有某个东西把你的文本变成运行中的软件——要么是编译器（提前把全部翻译成一个可运行的文件），要么是解释器（一行一行地读取并运行）。你选的语言决定了你走哪条路。



JavaScript / TypeScript

这是大多数 vibe coding 的默认选择，而且理由充分。它无处不在——浏览器、服务器 (Node.js)、边缘函数——而且它是整个 Web 前端的语言。TypeScript 是给 JavaScript 加装了一套类型系统，你几乎总应该优先选 TypeScript：类型能尽早抓住错误，更重要的是，它给 AI 设了护栏，让其产出更可靠。一个带类型的函数签名，就是一份助手能读懂的契约；没有它，AI 只能猜你的数据是什么形状，而猜测正是 bug 藏身之处。

- **擅长：**Web 应用（前端和后端）、任何用户在浏览器里点击的东西、快速原型、用一门语言搞定全栈项目、serverless/边缘部署。
- **痛点：**生态系统变化飞快且容易破坏既有代码；依赖泛滥是真实存在的（一个小应用可能拉进数百个包）；纯 JavaScript 的松散让微妙的 bug 得以溜过去——这又是该用 TypeScript 的一个理由。
- **典型用例：**SaaS（软件即服务——无需安装、在线租用的软件）仪表盘、落地页、电商、实时应用、浏览器扩展，以及任何你部署到 Vercel、Netlify 或 Cloudflare 的东西。
- **AI 驾驭能力：**出色。这是训练数据中代表性最强的语言。助手能自信地写 React、Next.js 和 Node，且很少卡壳。如果你没有强有力的理由另作选择，就从这里开始。

有一个值得了解的具体取舍：让 JavaScript 在原型设计上如此出色的那份速度，同样是让它不断翻新的速度。一篇十八个月前的教程可能已经在用一个被废弃的 API，而以过去快照训练出来的 AI——有时会递给你昨天的写法。当它写的东西跑不起来时，原因往往是库已经往前走了。修法通常是告诉它你用的版本，并让它去查当前文档。这是你为住在地球上最活跃的生态系统里所交的税，而对大多数项目而言，这税值得交。

Python

Python 读起来几乎像英语，这让它成为对初学者最友好的语言，也成为数据、脚本以及 AI 本身的通用语。如果你的项目涉及机器学习、数据分析、自动化或科学计算，答案通常就是 Python。

- **擅长：**AI/ML、数据处理、自动化脚本、后端 API（应用程序编程接口——一个程序与另一个程序对话时使用的门户）、胶水代码，以及任何已经有现成库的场景（对 Python 来说，通常都有）。
- **痛点：**纯数值运算很慢，除非你依赖底层用 C 写的库；虚拟环境和包版本带来的“依赖地狱”让新手困惑；不适合做浏览器前端。
- **典型用例：**爬虫、数据流水线、ML 实验、Discord/Telegram 机器人、内部工具、REST（表述性状态转移——构建这类 API 门户的一种常见风格）API (FastAPI、Django、Flask)。
- **AI 驾驭能力：**出色，与 JavaScript 不相上下。助手对 Python 及其主要库极为流利。对非 Web 项目而言是个很棒的默认选择。

Python 唯一会一再绊倒新手的，是环境，不是语言。一个脚本在 AI 那里某个文件夹里跑得好好的，换个文件夹就失败了，原因是那些包住在一个没被激活的虚拟环境里，或者两个项目想要同一个库的不同版本。这些都不是你代码的错，见

过一次之后也不难——但它是 Python 里最常见的”为什么就是跑不起来”时刻。从一开始就让 AI 设好一个虚拟环境和一份 requirements.txt (或用 uv、Poetry 这类工具)，大部分痛苦就消失了。

Go

Go 由 Google 设计，目标是简单、快速、易于部署。它编译成一个独立自包含的二进制文件，你可以直接丢到服务器上，无需安装任何运行时。它有意省去了花哨的特性，这让代码库以最好的方式保持”无聊”——而无聊对人类和 AI 来说都易于推理。

- **擅长：**快速的后端服务、命令行工具、网络编程、任何需要处理大量并发连接的东西、简单部署。
- **痛点：**啰嗦的错误处理（你会满眼都是 `if err != nil`）；不是前端语言；那种刻意为之的简单，在你想要富有表现力的捷径时会让人觉得受限。
- **典型用例：**API、微服务、CLI（命令行界面——通过输入命令来运行程序的纯文本方式）、基础设施工具、高流量后端。
- **AI 驾驭能力：**很好。Go 那套小而一致的特性集意味着大多数事情都有一种”正确做法”，助手也能很好地遵循它。训练数据比 JS/Python 少，但其简单性弥补了这一点。

对单干的开发者来说，Go 的杀手级特性正是那个单一二进制文件。没有”在服务器上装 Node 和这四十个包”的繁琐舞步，也没有你的机器和生产环境之间的运行时版本不匹配。你构建、复制一个文件、运行它。仅此一点就让部署的出错几率大幅降低——当你在 vibe coding、不想去调试基础设施时，这一点意义重大。啰嗦是它的代价：在 Python 里三行的代码，在 Go 里变成八行。但那些多出来的行是可预测的，而可预测恰恰是 AI 擅长应付的东西。

Rust

Rust 给你 C 级别的速度，外加一个拒绝让你写出整类内存 bug 的编译器。代价是一个出了名严格的编译器和一条陡峭的学习曲线。对 vibe coding 而言，Rust 是一把双刃剑：AI 能写它，但 Rust 的严格意味着编译器会更频繁地拒绝代码，从而带来更多来回修复的循环。

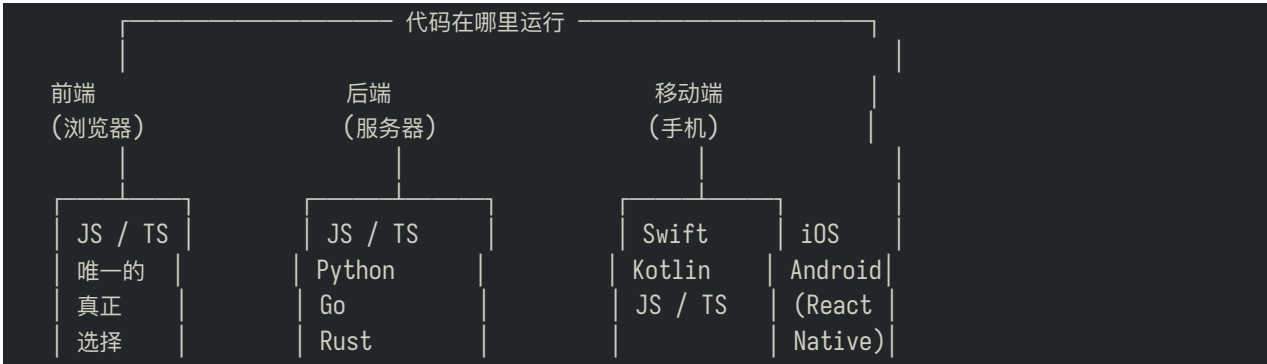
- **擅长：**性能关键的软件、系统编程、游戏引擎、WebAssembly、对速度有要求的命令行工具，以及任何把正确性与安全性放在首位的场景。
- **痛点：**学习曲线陡峭；编译速度慢；“借用检查器”让初学者抓狂；对大多数 Web 应用和脚本来说杀鸡用牛刀。
- **典型用例：**高性能后端、CLI 工具、WASM（WebAssembly）模块、近嵌入式系统、每一毫秒都要紧的基础设施。
- **AI 驾驭能力：**不错，但更颠簸。助手很懂 Rust，但严格的编译器会抛出需要反复迭代的错误。预期会比 Python 或 JS 出现更多”修错误、再试一次”的循环。当性能或安全性真正值得这份摩擦时再选 Rust——而不是把它当默认。

Rust 的严格有一个容易被忽略的微妙好处：编译器是一个免费而无情的审阅者。当 AI 写出马虎的 JavaScript，那个 bug 会被交付出去，在运行时咬到一个用户。当 AI 写出马虎的 Rust，编译器通常会在程序还没跑起来之前就抓住它。多出来的修复循环感觉像摩擦，但其中相当一部分，是编译器在替你拦下一个真实的 bug。在伸手去拿 Rust 之前要问的那个诚实问题是：你究竟是真的需要那份速度或安全，还是只是被告知你应该想要它们。对第一个项目来说，你几乎从来都不需要。

关于其他语言的简短一说

你会碰到这些，而且有时其中某一门正好对路：

想象一下每门语言主要住在哪里会有帮助。软件有层次——在浏览器里跑的（前端）、在服务器上跑的（后端）、在手机上跑的（移动端）——而大多数语言都有自己的主场：





- **Java / C#**: 大型企业的主力。成熟、稳定、工具链厚重, 是大型企业后端、Android (Java/Kotlin) 或 Windows/.NET 生态 (C#) 的理想之选。AI 对两者都驾驭得很好。作为单干的 **vibe coder**, 你很少需要它们, 除非你是在一个既有的企业技术栈里工作, 或者专门为 Android/Windows 开发。
- **Swift / Kotlin**: iOS (Swift) 和 Android (Kotlin) 应用的原生语言。如果你要把一个真正的移动应用交付到 App Store 或 Play Store, 你很可能会接触到它们。AI 的支持是扎实的, 但弱于 Web 语言, 而且移动开发还带着额外的摩擦 (设备、应用商店规则、构建工具链)。
- **PHP / Ruby**: Web 的老近卫军, 至今仍静静地驱动着其中很大一块。PHP 驱动着 WordPress, 因而撑起了所有网站里相当大的一个比例; Ruby on Rails 至今仍是一种快速、有主见地构建数据库支撑型应用的方式。两者都有充足的训练数据, 所以 AI 驾驭得很好——但如今新手很少会第一个去伸手拿它们, 你通常之所以选它们, 是因为你是在一个既有的代码库里工作, 而不是从头开始。
- **SQL (结构化查询语言)**: 它不是通用语言, 但一旦你的应用要存数据, 你就会遇到它。可以把它想成向数据库提问的语言——“把这周注册的所有用户给我列出来”。SQL 用来查询数据库。你不是用 SQL 代替其他语言——它和你选的任何东西并存。好消息是: AI 极擅长写 SQL, 所以你可以用大白话描述你想要什么数据, 让它产出查询。

一览对比

语言	最适合	需当心	AI 流利度
JavaScript/TypeScript	Web 应用、全栈、边缘	依赖泛滥、迭代飞快	出色
Python	AI/ML、数据、脚本、API	纯计算慢、环境配置	出色
Go	快速后端、CLI、服务	啰嗦、不适合前端	很好
Rust	性能、安全、WASM	曲线陡峭、修复循环多	不错 (更颠簸)
Java/C#	企业、Android、.NET	笨重、单干常属杀鸡用牛刀	很好
Swift/Kotlin	原生 iOS / Android 应用	移动摩擦、AI 较弱	不错
PHP/Ruby	WordPress、Rails、既有 Web 应用	默认设定过时、历史包袱	很好
SQL	与数据库对话	并存而非替代	出色

异类选择的真实代价

人很容易因为一门语言优雅、或快、或因为你敬重的某个人对它推崇备至, 就去选它。专就 **vibe coding** 而言, 这种本能可能悄悄地把一个项目拖沉。原因在于: 你花在和一门小众语言的工具链搏斗上的每一个钟头, 或者花在理顺那些因为模型见过的例子太少而微妙出错的 AI 产出上的每一个钟头, 都是你没拿去交付的钟头。异类选择很少会大声失败。它以一种缓慢的滴漏方式失败——多出来的修复循环、缺失的库、根本不存在的 **Stack Overflow** 答案。

具体地说, 这份惩罚出现在三个地方。第一, AI 不那么流利, 于是它犯更多错、也更不擅长修正——而不懂这门语言的你, 分不清好代码和坏代码。第二, 生态更薄, 于是你需要的那个库可能不存在、可能已被废弃、或者没有可供照抄的例子。第三, 当你卡住时, 整个网络更安静: 教程更少、被回答过的问题更少、撞上你这堵墙的人更少。在 Python 或 TypeScript 里, 这三样都不成其为问题。

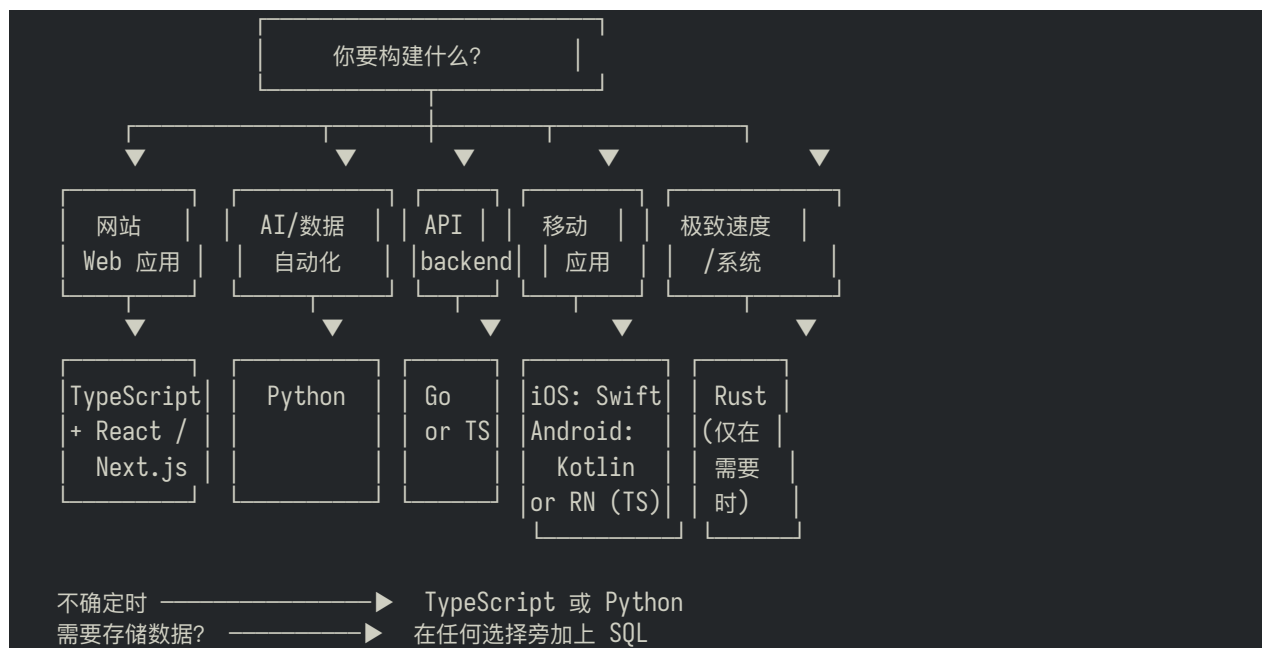
这些都不意味着异类语言不好。它意味着举证责任落在异类选择那一边。如果你有一个具体、明确的理由——你确实需要 Rust 的速度, 你被 App Store 逼着用 Swift, 你在扩展一个既有的 Go 服务——那就睁着眼睛去付这份代价。如果理由是”

它看着挺酷”，那就选热门的那个，去交付点东西。你随时可以日后重写，但你几乎从不会，因为热门的那个选择早已把活干完了。

按目标来选

别再纠结了。把你的目标对上一行，然后往前走——你随时可以改弦更张，而对于第一个项目来说，势头胜过优化。

把整个决定画成一棵树就是这样。从顶部开始，回答每个问题，顺着分支走到你的技术栈：



- 我想做一个网站或 **Web** 应用。□ TypeScript (配 React/Next.js)。显而易见、支持完善的选择。
- 我想做 **AI**、数据或自动化。□ Python。工具都在那儿。
- 我想要一个快速的 **API** 或后端服务。□ Go (简单又快) 或 TypeScript (如果它是某个 Web 技术栈的一部分)。Python 的 FastAPI 也运作得很好。
- 我在做一个移动应用。□ iOS 用 Swift, Android 用 Kotlin——或者考虑一种跨平台、基于 Web 的方案 (React Native), 从而留在 TypeScript 里。
- 我需要纯粹的速度或系统级的掌控。□ Rust, 但仅当你已确认自己确实需要它时。大多数人并不需要。
- 我在给一个已经存在的站点添东西。□ 用它本来就是用的那门语言。一个 WordPress 插件意味着 PHP; 一个老的 Rails 应用意味着 Ruby。顺着代码库来, 别跟它较劲。
- 我只想学点东西并快速交付。□ Python 或 TypeScript。两者都很宽容, 都是 AI 最熟悉的语言, 也都有无穷无尽的教程和示例。
- 我的应用需要存数据。□ 上面你选的任何东西, 外加用于数据库的 SQL。你不是在它们之间二选一。

那条元层面的教训是：拿不准时，选热门的那个。TypeScript 和 Python 覆盖了一个单干 vibe coder 这辈子会做的绝大多数东西，而且它们是你的 AI 助手处理得最可靠的语言。对 vibe coding 而言，“最好”的语言通常就是 AI 最熟悉的那一门——而那意味着热门的那一门。把那些异类的选择留到你具有具体理由的时候，反正无论哪种，语法都交给 AI 处理。

回顾与练习

要点回顾

- AI 对热门语言的驾驭远比小众语言流畅——热门本身就是一项功能，往往是决定项目是否顺利的最大单一因素。
- TypeScript 和 Python 是安全的默认选项：Web/全栈用 TypeScript, AI/数据/自动化用 Python, 两者的 AI 支持都很出色。
- Go (单一二进制后端)、Rust (速度/安全, 但修复循环更多) 以及其余语言各有狭窄的最佳适用面——只有在有具体理由时才去碰它们。

- SQL 不是二选一的选择；一旦你的应用要存数据，无论你选哪门语言，它都会与之并存。
- 让语言匹配你的目标，而不是匹配听起来唬人的东西——在第一个项目上，势头胜过优化。

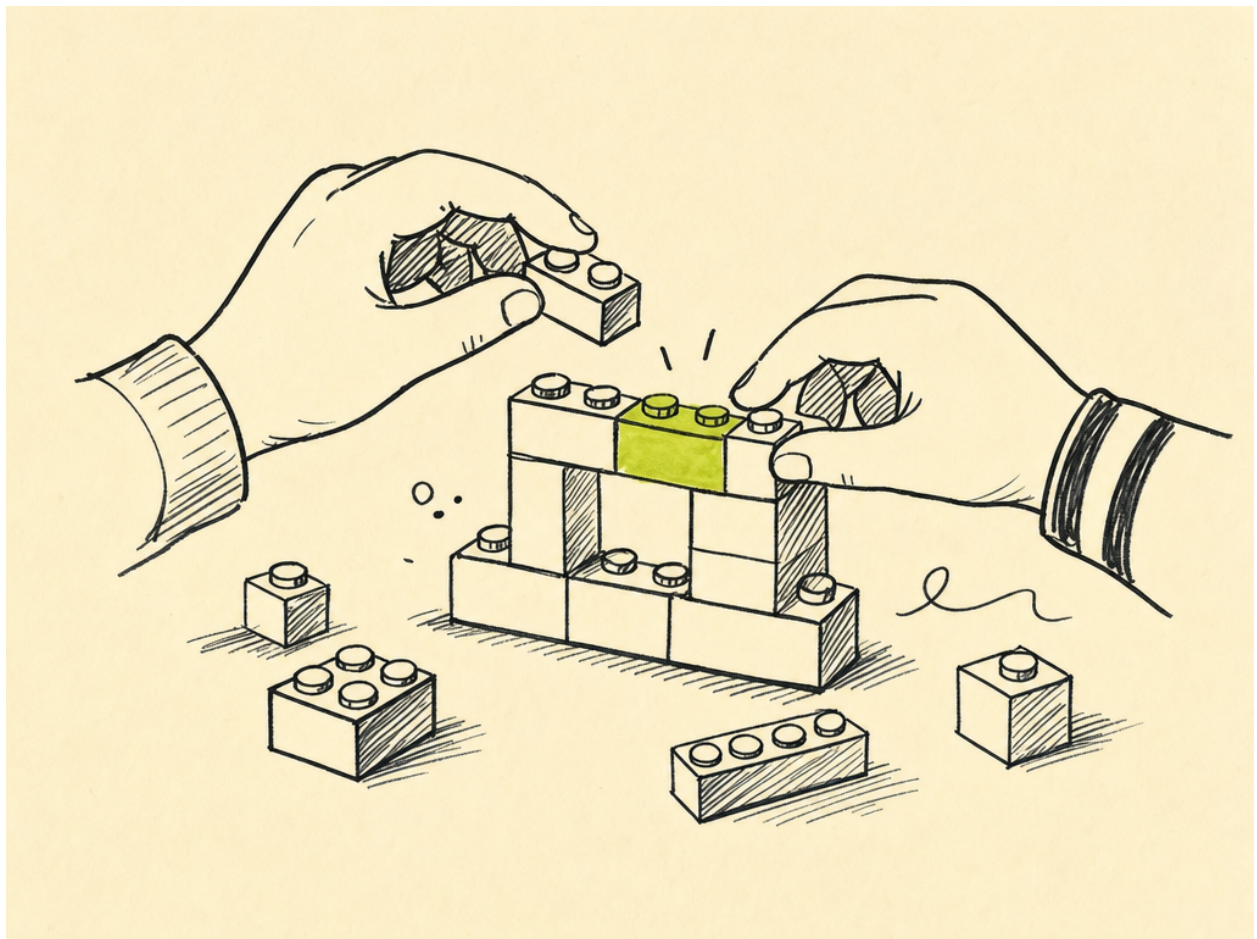
动手试试

挑一个你真正想做的点子，用一句话写下它的目标。把它对应到“按目标来选”列表中的某一行，然后让你的 AI 助手确认或质疑这个选择。让它用三个要点列出默认技术栈（语言、框架、部署到哪里），并请它指出在哪一种情形下另一门语言会更合适。在写下一行代码之前，你就握有了一个有依据的技术栈决定。

我想构建以下项目：[用一两句话描述它]。

为它的 *vibe coding* 推荐最佳的语言和框架，优化的目标是 AI 助手能多流畅地编写和修复代码，而不是追求极致性能。请给我：（1）语言和主要框架，（2）我会把它部署到哪里，（3）一个需要留意的现实陷阱，以及（4）一个换用其他语言反而会更适合的场景。请简短作答。

构建功能，一个 *vibe* 接一个 *vibe*



到现在，你已经能和你的 AI 对话，让它写出代码了。但一个真实的功能比单条提示要大。把能交付的人和会卡住的人区分开来的那项技能，是懂得如何把一个功能拆成小块，并驱动 AI 逐个把它们做完——在每一步都让应用保持可运行。

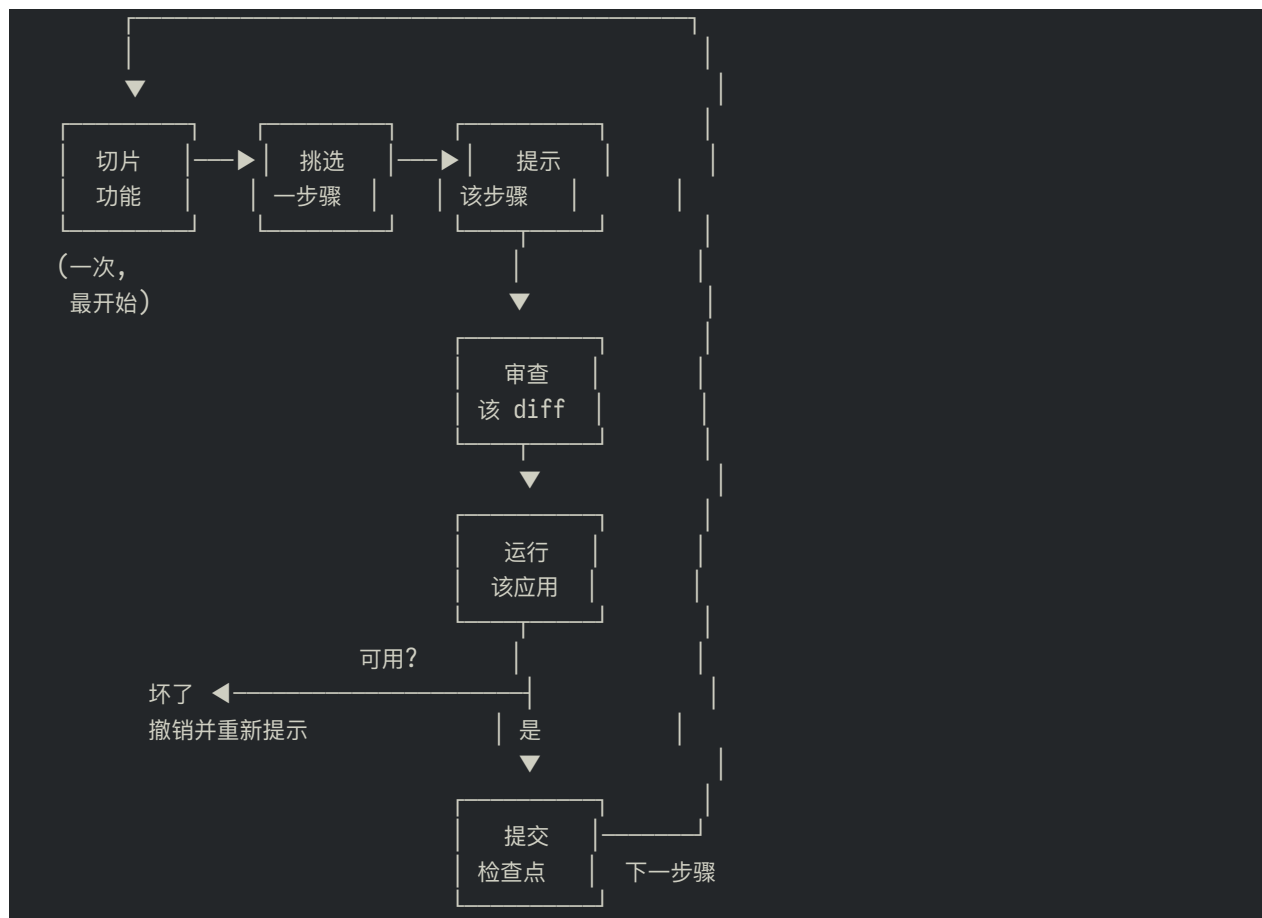
本章就是核心的构建循环。学会它一次，你就能在此后构建的每一个功能里反复复用。这正是一个稳步成长的项目，和一个在刚要变得有趣的那一刻就被自身重量压垮的项目之间的区别。

构建循环

下面就是你将一遍又一遍运行的循环。它故意做得很小。

1. 把功能切成小的纵向步骤。每一步都应是你能在几分钟内构建、运行、并看到它运作的东西。
2. 挑一个步骤。就一个。眼下其他都先不管。
3. 为那一个步骤写提示。对什么要改、什么要保持不变都要说具体。
4. 审查改动。读一读 AI 写了什么。你不需要看懂每一行，但要扫一眼有没有明显错的地方。
5. 运行应用。真的去点那个按钮、加载那个页面、请求那个端点。确认这一步奏效。
6. 提交。在版本控制里用一句简短的信息存一个检查点。
7. 重复第 2 步，换下一个步骤——直到功能完成。

把这个循环画成一个环就是下面这样——你绕着它一圈圈转，每圈一个小步骤，直到功能完成：



整个要点在于，你永远离一个可运行的应用不远。如果第 4 步产出了坏掉的东西，你只有一个小改动要撤销，而不是一团缠成结的乱麻。

跳过第 4 到第 6 步的人，会有大约二十分钟觉得自己更快。然后他们撞上一个找不到的 bug，因为他们应用上一个已知可用的版本，在一百个改动之前。这个循环之所以让人觉得慢，恰恰是因为它从不让你积累那种债务。

“纵向切片”是什么意思

一个纵向切片是功能里一道薄薄的、贯穿你整个应用的片段——一点 UI、一点逻辑、一点数据——而不是一次只构建一整层。

诱人却错误的做法（横向）：

- 先把整个数据库 schema 建好。
- 然后把所有后端逻辑写完。

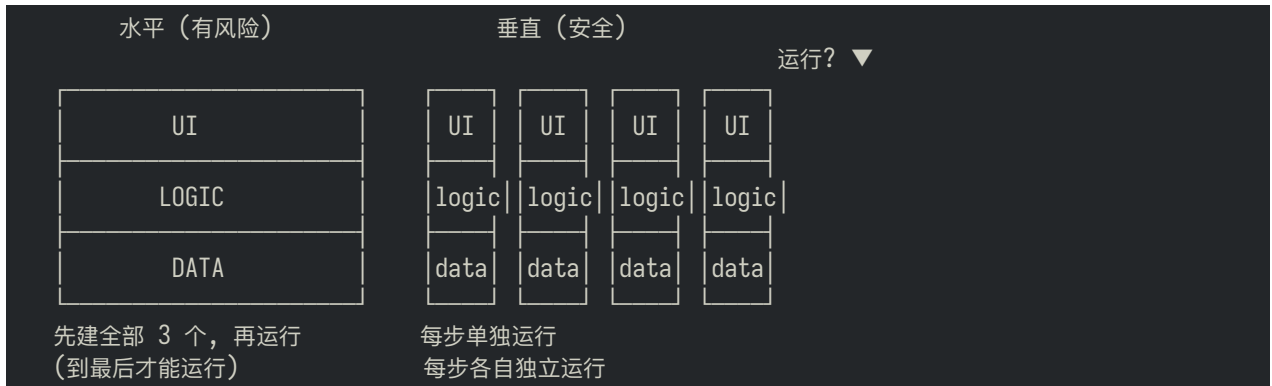
- 然后把整个 UI 做完。

三者全部做完之前，你什么都跑不起来，所以没法及早抓住问题。

更好的做法（纵向）：

- 第 1 步：一个按钮，用假数据做出这件事最简单的可能版本。
- 第 2 步：让它用上真实数据。
- 第 3 步：处理边界情况。

把两者并排放着看，差别最清楚。横向一次只造一整层，到最后才能运行；纵向每一步都造出一条贯穿所有层的、可运行的薄带：



每个切片都可运行。这正是让应用始终保持可运行的关键。

一个判断好切片的实用测试：**你能演示它吗？**如果你能指着屏幕说“看这个”，那它就是一个真正的纵向切片。如果唯一诚实的演示是“嗯，schema 现在搭好了”，那你构建的是一个横向层，而它对不对，你要到很久以后——通常是在最糟的时刻——才会知道。

一个实战示例：“收藏”功能

我们来端到端地做一个小功能：用户可以把列表里的条目标记为收藏，并看到自己收藏了多少个的计数。

切分它

我会这样把它拆成纵向步骤：

1. 在每个条目旁边加一个星标按钮（暂时什么都不做，只是显示出来）。
2. 点击星标切换它的亮/灭外观（仅状态，不保存）。
3. 显示一个反映这些切换的“3 favorites”计数器。
4. 保存收藏，让它在页面重新加载后依然存在。

注意每一步都可独立运行、可见。你在其中任何一步之后停下，都仍然拥有一个可运行的应用。

第 1 步：按钮

一条好的提示会点明文件、改动，以及不要碰什么：

在 `ItemCard` 组件 (`src/components/ItemCard.jsx`) 中，给每张卡片的右上角加一个星标按钮。它应该渲染一个轮廓星标图标。先别接上任何点击行为——只让它显示出来。保持卡片其余的布局不变。

运行应用。看到星标了吗？好。提交：

```
git add -A && git commit -m "Add star button to item cards"
```

第 2 步：切换状态

让星标按钮在被点击时在填充和轮廓之间切换。用本地组件状态来追踪它——先别保存到任何地方。填充的星标表示已收藏。

可以把状态（state）想成应用的短期记忆——此刻什么是真的，比如某颗星现在是否处于填充状态。现在你在同时处理状态和 **UI**，这很正常。（UI，即用户界面，就是应用中人能看到、能点击的那一部分。）AI 大概会加上类似这样的东西：

```
const [isFavorite, setIsFavorite] = useState(false);

<button onClick={() => setIsFavorite(!isFavorite)}>
  {isFavorite ? <StarFilled /> : <StarOutline />}
</button>
```

点几下星标。它们会切换吗？提交。

第 3 步：计数器

在条目列表顶部加一个“★ N favorites”计数器。它应该统计当前有多少个条目的星标被填充。如果需要，把收藏状态上提到列表组件，好让计数器能看到它。

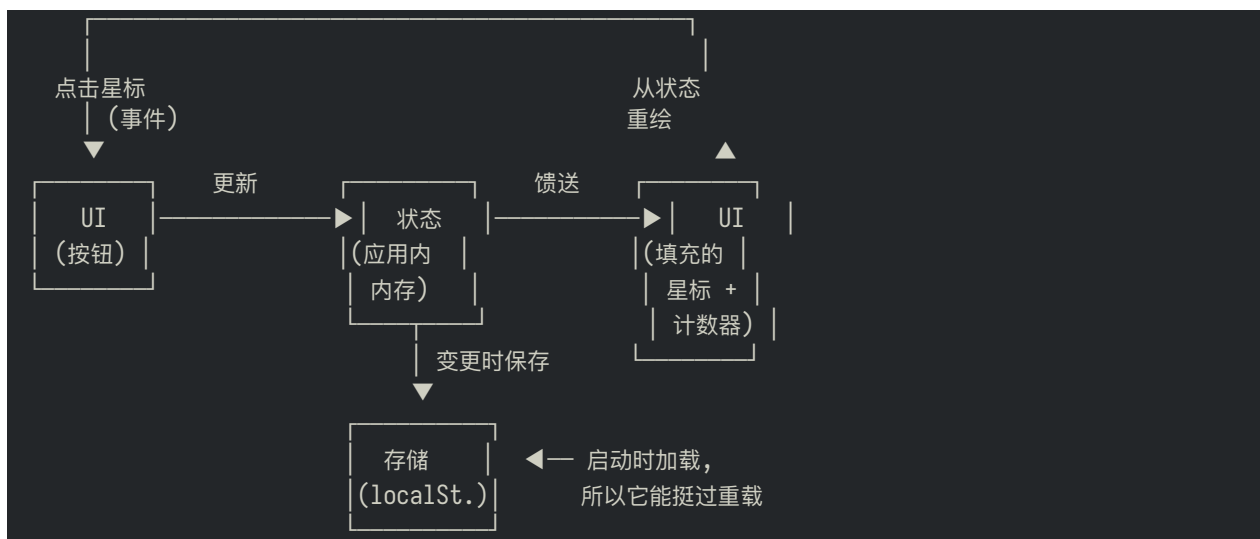
这一步可能要求 AI 重构状态所在之处——这在预料之中，而这恰恰是为什么你要把它作为独立的小步骤来做。运行它，切换几个星标，看着数字变化。提交。

第 4 步：让它持久化

把收藏持久化，让它们能挺过页面重载。现在先用 `localStorage`——每当收藏条目 ID 的列表变化时就保存它，并在启动时加载它。

（`localStorage` 就是浏览器在访客自己电脑上保留的一个小记事本，应用可以把东西记下来，重新加载之后再找回来。）重新加载页面。还是收藏状态吗？你刚刚交付了一个功能，一次一个可运行的切片。

来看看这个完成的功能里各部件是怎么互相对话的。一次点击向下流入状态，屏幕从状态重绘，而状态被镜像到存储里，于是它在重新加载后依然存活：



不必看懂每一行也能读 *diff*

循环的第 4 步是“审查改动”，也是人们最爱含糊带过的一步。要做出有用的审查，你并不需要能自己写出那些代码。你只需要会扫一眼一小组危险信号。

当 AI 把改动交给你时，看它最清爽的方式就是 diff——精确地列出新增和删除的行：

```
git diff
```

以 + 开头的行是新增的，以 - 开头的行是删除的。像个审查者那样去读，而不是像个编译器。你要找的是这类东西：

- 它有没有动了不该动的文件？你要的是计数器，为什么 auth.js 会出现在 diff 里？
- 它有没有删掉看起来重要的东西？一行你认不出的被删除的行（-）值得一问。
- 它有没有悄悄改了你没要求的行为？新的默认值、被改的条件、“贴心”的额外功能。
- 形态和你的诉求对得上吗？一句话的请求却产出了两百行，这值得再看一眼。

如果有什么看着不对，你不必自己去修——直接问就行：“你为什么改了 auth.js？我只要了计数器。”一半的时候，AI 会说“说得对”然后撤销它。光是这个习惯，就比你盯着自己没完全看懂的代码看再久，都能抓住更多的 bug。

何时放手让 AI 跑，何时手动掌舵

不是每一步都需要同样紧地握住方向盘。

在以下情况放手让 AI 自由跑：

- 步骤小且定义清晰（“加一个按钮”）。
- 这是一个 AI 显然见过上千次的常见模式。
- 出错的代价低且明显（你一运行应用就会看到）。

在以下情况手动掌舵——更小的提示、更细的审查：

- 改动触及你不容易恢复的数据（删除记录、迁移）。
- AI 反复在原地“修”同一个东西转圈。
- 行为很微妙，光扫一眼屏幕看不出来（金额计算、权限、鉴权）。
- 你已经试了两次，而它正在变得更糟，而非更好。

一条可靠的规则：**爆炸半径越大，步子迈得越小。**

让应用始终保持可运行

这是让其他一切成为可能的那门纪律：

- **每一步之后都运行。**不是每三步。是每一步。
- **只提交可运行的状态。**一次提交就是一个你能回到的存档点。如果某一步弄坏了某个你在一两分钟内修不好的东西，就把它扔掉（git restore . 或回退），用更清晰的诉求重新提示——别在一个坏掉的基础上再叠一个补丁。
- **每步只关注一件事。**如果你心痒想“顺手把那另一个东西也快速修一下”，忍住。先完成当前切片，提交，然后开始一个新步骤。
- **描述 bug，别去诊断它。**当某个东西坏了，把你看到的原原本本告诉 AI：“点了第二个星标之后，计数器显示的是 1，不是 2。”让它去找原因。

小步提交

小提交是你的安全网。每一个都应是一个单一、连贯的步骤：

```
git commit -m "Add star button to item cards"
git commit -m "Toggle favorite state on star click"
git commit -m "Add favorites counter to list header"
git commit -m "Persist favorites in localStorage"
```

如果第 4 步出了岔子，你前三步是安全的，你几乎什么都不会损失。又大又少的提交，正是人们最终害怕去碰自己代码的根源。

当一个切片太大时

有时你挑了”一个步骤”，结果它还是出岔子——AI 重写了应用的一半，diff 读都读不下来，一运行就同时蹦出三个新 bug。这几乎从不是运气不好。这是一个信号：切片太大了。

解法是把它切得更薄。如果”加上带持久化的收藏”炸了，那它就不是一个切片；它是整个功能披着一条单一提示的伪装。退回到你上一个可用的提交，把它拆开：

```
# 别一口气要求这个：
"加一个带计数器和 localStorage 持久化的收藏功能。"

# 改成要求这个，每个提交一件：
"给每个条目卡片加一个星标按钮。先别加行为。"
"让星标用本地状态切换填充/轮廓。"
"加一个反映已填充星标的计数器。"
"把收藏的 ID 持久化到 localStorage，并在启动时加载。"
```

一条好用的经验法则：如果你没法用一句简短的话说清一个步骤做什么，那它就是两个步骤。而如果一个步骤的 diff 长到一分钟内扫不完，那下一个步骤就该更小。切分是一项技能，你会在感受上一个切片哪里切得太厚的过程中，越来越擅长它。

要点带走

一个功能不是一条巨大的提示——它是一摞小的、可运行的切片。切分它，为一个步骤写提示，审查、运行、提交、重复。在简单的事情上放手让 AI 跑，在任何有风险的地方附近用力掌舵。在每个检查点都让应用保持可运行，你就永远只差一个小小的撤销，就能回到坚实的地面。

在下一章里，我们会看看当某一步没按计划进行时该怎么办——凭感觉调试。

回顾与练习

要点回顾

- 一个功能是一摞小的纵向切片，而不是一条巨大的提示——在继续之前，先构建、运行并亲眼看到每个切片能跑起来。
- 每次都跑构建循环：切片 □ 选一个步骤 □ 写提示 □ 审查 diff □ 运行应用 □ 提交 □ 重复。
- 像审查者而不是编译器那样读 diff——留意它本不该碰的文件、悄悄改变的行为，以及远大于需求的 diff。
- 在小的、廉价的、常见的步骤上放手让 AI 跑；在影响范围大的地方手动掌舵（更小的提示、更细的审查）。
- 只把可运行的状态小步提交——切片出岔子时，扔掉重写，而不是在坏掉的基础上再叠一个修复。

动手试试

挑一个你想加进项目的小功能，把它切成 3 到 5 个纵向步骤，每个写成一句短句。只构建第一个步骤：为它写提示，运行 git diff 略读改动，运行应用确认它能跑，然后用一行消息提交。眼下就停在这里。目标是感受一次干净的循环转动——一个可运行的切片、一份你真正读过的 diff、一个你能回到的提交。

```
我想加这个功能：[用一两句话描述这个功能]。
先别写任何代码。首先，把它切成 3 到 5 个垂直步骤，每一步都是我能自己
构建、运行并看到它独立工作的东西。把它们排好序，让应用在每一步之后
都还能用。每一步给我一句话的描述，并说出你会改的第一个文件。然后
等着——我会让你一次构建一步。
```

用 AI 调试



你构建的每个程序都会出问题。这不是你做错了的信号——这是软件的正常状态。Bug 不是技艺的失败；它们是真实的代码遇上真实的数据、真实的用户，以及没人想象得到的真实边缘情况时所发生的事。一个令人沮丧的下午和一个五分钟的修复之间的区别在于方法。当你用 **vibe coding** 来调试时，盯着堆栈跟踪直到看懂的人不是你——堆栈跟踪就是程序崩溃时打印出来的错误报告，我们稍后会拆开细说。你是导演：你复现问题，收集正确的证据，然后把一个紧凑、框定清晰的案例交给 AI，让它去做繁重的工作。本章将给你一个可重复的流程来做到这一点——无论 bug 是一行拼写错误，还是一个缠绕了三层深的竞态条件（**race condition**），这个流程都管用。（竞态条件是一种时序 bug：两件事以你没料到的顺序发生，就像两个人同时去抢最后一个座位。）

核心心态

初学者犯的最大错误，就是把“它不工作，修好它”粘贴到对话框里，然后祈祷。AI 看不到你的屏幕、你的数据，也不知道你期望发生什么。它只知道你告诉它的东西。当你什么都不给它时，它只能做它唯一能做的事：猜测——而一个碰巧错了的自信猜测，比根本没有答案更危险，因为你会照着它去行动。用 AI 调试主要是关于给它喂正确的上下文，这样它才能推理，而不是猜测。

一份好的 bug 报告——无论是给人还是给 AI——都会回答三个问题：

- 你做了什么？（触发它的步骤）
- 你期望发生什么？
- 实际发生了什么？（错误、错误的输出、崩溃）

如果你能回答这三个问题，你已经领先于大多数人了。这三点之所以重要，是因为一个 bug 按定义就是你所期望的和实际发生的之间的缺口。AI 找不到一个它看不到两侧的缺口。“它坏了”只描述了一侧；“我期望一个欢迎界面，结果得到一个空白页

面和这个错误”描述了两侧——于是缺口几乎会自报家门。

一份调试清单

按顺序逐项完成这些步骤。直接跳到”修好它”正是让 AI 兜圈子的原因。

1. **可靠地复现它。**找到每次都能让 bug 出现的确切步骤。一个你无法复现的 bug，是你无法确认自己已经修好的 bug——你将永远只是在猜测它是否消失了。如果它是间歇性出现的，去捕猎那个让它切换的变量：某个特定输入、登录还是登出的状态、缓慢的网络。
2. **阅读真正的错误信息。**不要略读。错误信息和堆栈跟踪的顶部通常会指出出问题的文件和行号。初学者常常因为某句话看起来吓人，就略过了屏幕上最实用的那句。
3. **收集上下文。**收集错误文本、堆栈跟踪、相关代码，以及一句话描述你期望的结果。
4. **把完整的画面交给 AI。**把以上所有内容粘贴在一条消息里。让它形成一个假设。
5. **形成假设，而不是修复方案。**在”修好这个”之前先问”什么可能导致这个？“。基于错误理论的错误修复只会浪费时间——有时还会在第一个 bug 之上叠加一个第二个 bug。
6. **添加日志来测试假设。**在修改代码之前，先确认代码实际在做什么。
7. **二分查找问题。**通过反复将可疑区域对半切分，缩小故障发生的位置。
8. **修复根本原因，而非表象。**确保修复针对的是它为什么出问题，而不仅仅是可见的故障。
9. **验证修复并检查回归。**重新运行复现步骤。然后确保附近的功能仍然正常工作——所谓回归，就是一个修复悄悄弄坏了原本好好的东西，就像补好一处漏水，旁边的管子却爆了。

这份清单其实就是一个你不断绕行、直到 bug 消失的循环。每个方框都把证据交给下一个，而一次失败的验证只会带着你学到的东西重新绕一圈：



▼
✓ 完成

◀———
(绕回去再来)

这个顺序并非随意。每一步都产生下一步所需要的证据。复现给你提供可读的东西；阅读给你提供可收集的上下文；上下文给 AI 提供可做假设的素材。跳过一步，你就是在给下一步喂一个窟窿。

给 AI 喂正确的上下文

这是你大部分杠杆所在之处。当你报告一个 bug 时，请包含：

- 逐字的错误信息。原样复制——不要转述。转述一个错误，就像在电话里描述一张照片：你会漏掉那个最关键的细节。
- 堆栈跟踪，尤其是顶部的几行。
- 相关代码——抛出错误的函数，以及它调用的任何东西。
- 你期望的结果与实际发生的结果对比。

下面是一个强力调试提示的真实样子：

我提交注册表单时报错了。下面是完整的输出：

```
TypeError: Cannot read properties of undefined (reading 'email')
    at validateUser (src/auth/validate.js:14:23)
    at handleSignup (src/routes/signup.js:31:10)
    at processTicksAndRejections (node:internal/process/task_queues:95:5)
```

复现步骤：填好表单，点击 "Sign up"。

期望：一条 "Welcome" 消息。

实际：页面变成空白，上面这个错误出现在控制台里。

这是 validate.js (第 14 行附近)：

[粘贴这个函数]

这是调用它的 handleSignup 函数：

[粘贴这个函数]

什么可能导致这个，以及在改动任何东西之前我该如何确认它？

注意最后一行。你是在请求一个假设和一个确认步骤，而不是立即修改。这能让 AI 保持诚实。

知道如何阅读你正在粘贴的那段跟踪，会很有帮助。堆栈跟踪是一份按最新到最旧排列的调用历史：最顶部那行是程序实际爆炸的地方，下面每一行都是调用了它上面那个函数的函数。所以在例子里，validateUser 是 email 从某个 undefined 的东西上被读取的地方，而 handleSignup 是调用 validateUser 的那一方。你自己代码下面的帧——比如 processTicksAndRejections——属于语言运行时；通常你可以忽略它们。从上往下读，直到碰到第一行指向你的文件，那几乎总是该开始查看的地方。

把同一段跟踪当作一摞调用来读，顺序就一目了然了。崩溃在最顶上，下面每一帧都是调用了它正上方那一帧的那一方：



向下读，直到第一行是你的文件

阅读你看不懂的错误

迟早你会遇到一个看起来像字母汤的错误——一个很长的类型名、一个全是框架内部的堆栈、一条用你从没见过的行话写的消息。别让它把你卡住。错误是数据，不是对你能力的判决。当消息本身晦涩难懂时，把它粘贴上去，让 AI 在修复之前先翻译：

我看不懂这个错误。在提出任何修复方案之前，请用通俗的语言解释：

它实际上在说什么，它指向哪一行，
以及通常什么东西会导致这个确切的错误。

[粘贴完整的错误和堆栈跟踪]

先要求用通俗语言解释含义，能做到两件事。它教会你这个模式，这样下次再看到这个错误时你自己就能认出来——而几个月下来，这正是你真正学会阅读自己堆栈的方式。它还迫使 AI 把修复建立在错误所说的内容之上，而不是跳到一个通用补丁。当心最常见的错误陷阱：一个指向第 14 行的错误，往往是在完全别的地方被引起的。跟踪里的那行是程序注意到问题的地方，并不总是问题开始的地方。

假设、日志和二分查找

一旦 AI 提出了一个原因，不要直接接受。用低成本的方式测试它。最便宜的测试通常是一行日志。

让 AI 添加临时日志：

“在我们改任何东西之前，添加一个 `console.log` (或 `print`)，在第 14 行之前显示 `user` 的值，这样我们就能看到它是不是真的为 `undefined`。”

运行它，把输出粘贴回来。现在你拥有的是证据，而不是一个理论。也许 `user` 确实是 `undefined`——这就确认了假设。也许它没问题，真正的元凶在上一层。无论哪种情况，你都向前推进了。调查时要慷慨地打日志：变量的值，以及一个标签，说明它是哪个变量、日志在哪里。一个打印出 `undefined` 的光秃秃的 `console.log(user)`，远不如 `console.log("user at validate.js:14 =", user)` 有用。还有，bug 修好后记得把临时日志拔出来——遗留的调试噪音本身就是一小摊烂摊子。

当 bug 藏在一个很长的流程中的某处，而你说不清在哪里时，使用二分查找。把可疑区域对半切分：在中点添加一条日志。

核心思路：每次检查都丢掉剩下一半，所以一个庞大的搜索空间会迅速塌缩。下面这个 bug 藏在第 1 步和第 8 步之间的某处——三次检查就把它逼了出来：

```
步骤： 1 2 3 4 5 6 7 8      bug 藏在这里某处
        ////////////////

检查 1 — 在 4 打日志 — OK —▶  bug 在 4 之后，弃掉 1-4
                               //////////////// (剩 5 6 7 8)

检查 2 — 在 6 打日志 — BAD -▶  bug 在 6 处/之前，弃掉 7-8
                               //////////////// (剩 5 6)

检查 3 — 在 5 打日志 — OK —▶  bug 在第 6 步    ✓ 找到

8 步 → 3 次检查。  1000 步 → 约 10 次检查。
```

那里的值看起来正确吗？那么 bug 在它之后——搜索那一半。那里的值已经看起来不对了吗？那么 bug 在它之前——搜索那一半。每一步都把搜索空间减半，所以即使是一千行的路径，也能在大约十次检查内缩小范围。明确地告诉 AI：“帮我二分排查——我应该在哪个中点打日志？”同样的技巧不仅对代码有效，对历史也有效：如果某个功能上周还能用、今天坏了，就对你的提交做二分查找，找到引入这个 bug 的那个确切改动。

根本原因与表象

调试中最诱人的陷阱，就是那种让错误消失却没修复原因的补丁。如果 `user` 是 `undefined`，你用 `if (user) { ... }` 来“修复”它，错误是停止了——但现在注册会悄无声息地什么都不做，这更糟糕：一次响亮的崩溃至少告诉你哪里出了错，而现在什么都不告诉你了。真正的问题是为什么 `user` 在这里是 `undefined`？也许数据库查询失败了。也许某个字段被重命名了。也许上游缺了一个 `await`。

当 AI 提出一个修复方案时，问它：

- “这是在修复原因，还是在掩盖表象？”
- “这个值出错的最初原因是什么？”
- “同样的根本原因会不会破坏其他东西？”

修复根本原因通常会让代码更简单或更清晰。修补表象通常会添加一个守卫、一个吞掉错误的 `try/catch`，或者一个掩盖真正问题的默认值。对第二种要保持警惕。有一个值得学会的迹象：如果一个修复添加的代码，其唯一职责是防止崩溃，而不是做正确的事，那你很可能是在修补表象。真正的修复倾向于把那个不可能的状态彻底移除，而不只是熬过它。

当 AI 开始兜圈子

有时 AI 提出一个修复方案，不起作用，它再提出另一个，那个也不起作用，于是你已经陷入三层“试试这个”的循环里了。这是一个信号，而不是死胡同。这样做：

- **停下来，重置上下文。**用一条全新的消息开始，总结你现在确定知道的一切（由日志确认的），而不是你尝试过的一切。一段塞满了失败尝试的长对话，会主动毒化 AI 的下一个回答——它会不断地对自己历史里的那些死胡同做模式匹配。
- **说明你已经排除了什么。**“我们确认了 `user` 是 `undefined`，数据库查询返回 `null`。查询字符串看起来是对的。所以问题在查询的上游。”这能阻止 AI 重新建议那些死胡同。
- **让它解释，而不是修复。**“一步一步带我走一遍，`user` 是如何从请求拿到它的值并到达第 14 行的。”强制它解释，往往能暴露出缺口。
- **添加更多日志，而不是更多猜测。**当理论用尽时，证据才是前进的方向。
- **缩小问题。**让 AI 帮你构建一个仍能复现 bug 的最小代码片段。通常，一旦噪音消失，bug 就会变得显而易见——而如果它在小版本里复现不出来，你就知道了原因藏在你删掉的部分中。

在保留修复之前先理解它

一个能用却不理解的修复，是一笔贷款，而不是一个解决方案——而当附近某个东西下次坏掉时，利息就到期了。因为 AI 替你打字，所以很容易接受一个绿色的对勾就继续往前，却从不知道代码为什么现在是绿的。抵制这种冲动。在你保留一个修复之前，让 AI 为它交代清楚：

- **“用一两句话解释为什么这能修好它。”**如果解释含糊——“这让它更健壮”“这处理了边缘情况”——就回怼回去。真正的解释会点明具体的原因，以及这个改动如何把它移除。
- **“那个 bug 究竟是什么，用因果来说？”**你应该能用自己的话把答案复述出来。如果不能，你就还没理解自己的程序。
- **“还有什么会触及这条代码路径、需要我重新测试？”**这把一个修复变成了一次回归检查。

这就是“理解你所交付之物”哲学应用在它最难的那种情况上。你不需要能从零写出这个修复——那是 AI 的活儿。但你确实需要把因果的故事攥在脑子里，因为它再次坏掉的凌晨两点，在场的人是你，而那时唯一帮得上忙的，就是知道这东西实际上是怎么运作的。

要点

用 AI 调试不是关于知道答案。它是关于运行一个有纪律的流程：复现、收集上下文、提出假设、用日志确认、二分查找，并修复根本原因——然后把这个修复理解到足以诚实地保留它。作为导演，你的工作是让调查始终扎根于证据，并识别出循环何时已经陷入停滞。做到这一点，大多数 bug 就不再是谜团，而只是构建真实软件过程中的又一个步骤。

回顾与练习

要点回顾

- bug 是你预期与实际发生之间的落差——把两边连同确切的复现步骤都给 AI，绝不要只说“它坏了”。
- 把错误一字不差地贴上去，并附上栈追踪的顶部；指向你自己文件的第一行几乎总是起点。
- 改动前先确认：加一行带标签的日志来验证假设，并通过二分查找（在代码或提交历史中）让搜索空间每一步减半。
- 修复根本原因，而不是表象——只能阻止崩溃的守卫或 `try/catch` 通常只是把真正的问题盖住。
- 当 AI 兜圈子时，把上下文重置为你确定知道的内容，让它去解释而不是去修，并补上证据而不是猜测。

动手试试

下次有东西坏掉时，在请求修复之前先写好 bug 报告。在一个全新的对话里给 AI 三样东西：复现步骤、你的预期，以及实际发生的情况——错误一字不差地复制，并包含栈追踪的顶部几帧。然后，不要说“修好它”，而是请它给出一个假设以及一个廉价的确认方式（通常是一行日志）。运行那个检查，把输出贴回去，然后才让 AI 提出修复方案。感受一下这条路比“它不工作，帮我”要直多少。

我有一个 bug。在提出任何修复之前，给我一个假设以及一个低成本的确证方式（比如一行日志）——先别改代码。

复现步骤：[你做了什么]

期望：[本该发生什么]

实际：[实际发生了什么]

完整的错误和堆栈跟踪：

[一字不差地粘贴]

相关代码：

[粘贴出错的函数，以及它调用的任何东西]

什么可能导致这个，以及在我们改动任何东西之前我该打什么日志来确认它？

服务器基础设施



你用 **vibe coding** 写出了某个能在自己笔记本上运行的东西。现在你想让其他人也能用它。那道鸿沟——“在我的机器上能跑”和“在互联网上能跑”之间的鸿沟——正是基础设施所关心的。好消息是：你不需要一份 **DevOps** 职业生涯就能跨越它。你需要的是一个能用的心智模型，以及愿意让 **AI** 来处理那些琐碎的配置。

本章揭开托管这片天地的神秘面纱，用平实的语言给你讲清各种取舍，并交给你一条默认路径，让一个单打独斗的构建者从原型走向生产，而不至于淹没在各种选择之中。

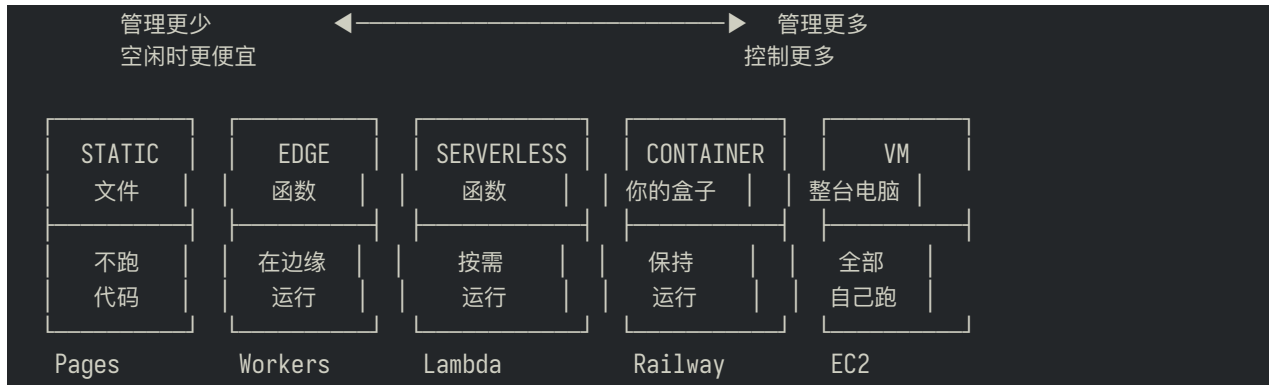
代码可以待的五个地方

地球上几乎每一个托管产品，都是这五个类别之一的变体。学会这些，那些营销页面就不再让人困惑了。

- **静态托管 (Static hosting)** ——把纯文件（HTML (HyperText Markup Language, 网页的骨架)、CSS (Cascading Style Sheets, 样式装扮)、JavaScript、图片）直接交付给浏览器。没有服务器“运行”你的代码；它只是把文件递过去。把它想象成互联网上的一个文件夹。因为没有什么会崩溃，也没有什么需要打补丁，所以这是你能放到互联网上的最可靠的东西。例子：GitHub Pages、Cloudflare Pages、Netlify、Vercel（静态模式）。
- **边缘函数 (Edge functions)** ——遍布全球、运行在物理上靠近你用户的服务器上的微小代码片段。它们几乎瞬间启动，非常适合轻量逻辑：身份验证检查、重定向、小型 API (Application Programming Interface, 应用程序编程接口——一个程序与另一个程序对话的门口) 调用。要注意的是：它们运行在一个功能精简的环境里，所以并不是每个库都能在那里工作。例子：Cloudflare Workers、Vercel Edge Functions、Deno Deploy。
- **无服务器函数 (Serverless functions)** ——按需运行、空闲时消失的代码。你按请求计费，而不是按小时。你写一个函数；平台负责管理机器。在单个请求需要运行好几分钟之前都很好用——大多数平台会在超时（通常 10 到 60 秒）后切断函数。例子：AWS Lambda、Google Cloud Functions、Vercel Functions。

- **容器 (Containers)** ——你的应用加上它的整个环境（正确的语言版本、库、系统工具）被打包进一个叫做容器的可移植盒子里。平台让那个盒子保持运行。更多控制权，稍多一点配置。定义那个盒子的那一个文件就是 Dockerfile，而 AI 写得很好。例子：Fly.io、Railway、Render、Google Cloud Run。
- **虚拟机 (VMs)** ——你在数据center里租用的一整台计算机。所有东西——操作系统、安全更新、Web 服务器，全部——都由你自己安装和管理。最大的控制权，最大的责任。例子：AWS EC2、DigitalOcean Droplets、Hetzner。

把这五个类别并排来看，越往下，你用更少的管理换取更多的控制：

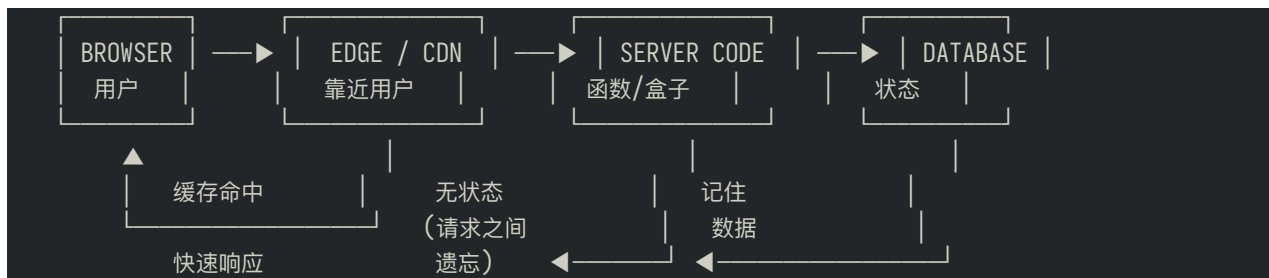


关于成本的一条粗略经验：平台替你管理得越多，没人用你应用时就越便宜，而当所有人都在用时它替你操的心也越多。

几个行话术语，破译一下

- **冷启动 (Cold start)** ——当代码有一段时间没运行过时，平台必须把它唤醒。那第一个请求会更慢（从几分之一秒到几秒）。边缘函数几乎没有这个问题；无服务器函数有时有；常驻运行的容器和 VM 则没有。对大多数应用来说它是看不见的；但对一个要求利落的结账流程来说，它可能很要紧。
- **扩展 (Scaling)** ——在处理更多用户时不至于崩溃。自动扩展意味着平台替你增加容量。手动扩展意味着你点击按钮，否则它就在负载下直接挂掉。“缩减到零 (scales to zero)”是你经常会看到的一个相关说法：当没人用你的应用时，它不花钱。
- **状态 (State)** ——需要持续保留的数据（一个数据库、上传的文件、一个用户会话）。静态层和无服务器层通常是“无状态的”——它们在请求之间忘记一切——所以状态存放在一个独立的数据库或存储服务里。这就是为什么几乎每个真实应用都是“你的托管加上某处的一个数据库”，而不是一个魔法盒子。
- **区域 (Region)** ——你的代码和数据在物理上所处于世界的哪个部分。它影响速度（越近越快），有时还影响法律（某些数据必须留在特定国家）。对第一个项目，选离你用户最近的区域，然后继续往前走。

下面是一个请求所走过的完整旅程，以及每一块各自待在哪里。浏览器与就近的边缘对话，边缘交付缓存的文件或唤醒你的代码，而代码在需要记住什么时就伸手去找数据库：



取舍对照表

托管类型	成本（空闲 □ 繁忙）	扩展	冷启动	复杂度	最适合
静态	免费 □ 非常便宜	自动，毫不费力	无	最低	营销网站、文档、博客、前端
边缘函数	免费额度 □ 便宜	自动，全球性	可忽略	低	身份验证、重定向、轻量 API、个性化
无服务器	空闲时 \$0 □ 随使用量扩展	自动	有时明显	低到中	API、webhook、突发/不可预测的流量
容器	运行期间付费	大致自动，可配置	无（如果保持温热）	中	完整后端、长时间运行的任务、自定义运行时
VM	全天候付费	手动	无	最高	特殊硬件、遗留软件、完全控制

规律是：随着你在表格中往下移动，你用简单性和低空闲成本换取控制权和灵活性。大多数 vibe coder 都应该努力待在靠近表格顶部的地方。

何时选哪个

- 只是一个网站或前端？静态托管。搞定。它免费、快速，而且几乎不可能弄坏。
- 需要一点后端逻辑——一个联系表单、一个保密的 **API key**、一次身份验证检查？边缘函数或无服务器函数。你待在慷慨的免费额度里，没人用时就缩减到零。
- 一个带数据库、后台任务，或期望有一个长时间运行服务器的框架的真实应用？一个像 Railway、Render 或 Fly.io 这样的容器平台。对大多数“真正的产品”项目来说，这是最佳甜蜜点。
- 你需要某个特定的 **GPU**、特殊的系统软件，或完全的控制权？一台 VM。只有当上面的某个选项确实做不了这件事时才走到这里，因为现在你就是系统管理员了。

一条有用的规则：选择能完成工作的最简单的选项，只有当你撞上一堵真实的墙时才往下移动一个层级——而不是一堵想象中的墙。

一堵“真实的墙”实际上长什么样？几个诚实的例子：你的构建产出了一个需要保持运行的服务器（一个维持着实时连接的聊天后端），所以静态行不通。或者一个函数老是撞上它的超时，因为一个请求要干真活（缩放视频、生成 PDF），于是你晋级到容器。或者你免费额度的请求上限被真实流量实实在在地耗尽了。注意这些的共同点：每一个都是一次测量，而不是一种直觉。“它将来可能会变大”不是一堵墙。

AI 如何串联部署与配置

这正是 vibe coding 大放异彩的地方。部署配置恰恰是那种精确、文档完善、充满样板代码的工作，AI 擅长此道。你来指挥；它来编写。

有效的提示听起来像这样：

- “为这个 Node 应用添加一个 Dockerfile，以及一个用来部署它的 Railway 配置。”
- “写一个 GitHub Actions 工作流，在每次推送到 main 时把 dist/ 文件夹部署到 Cloudflare Pages。”
- “我的无服务器函数遇到了冷启动超时——可能哪里出了问题，我该怎么修复配置？”

下面是 AI 会为你生成的那类文件——一个部署工作流，每当你推送代码时就自动发布一个静态网站：

```
# .github/workflows/deploy.yml
name: Deploy
on:
  push:
    branches: [main]
```

```
jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci && npm run build
      - name: Publish to Cloudflare Pages
        uses: cloudflare/wrangler-action@v3
        with:
          apiToken: ${ secrets.CF_API_TOKEN }
          command: pages deploy dist --project-name=my-app
```

你不需要背下这个语法。你需要认出它做了什么：在每次推送到 main 时，它检出你的代码，构建它，并发布结果。当出现问题时，你把错误粘贴回给 AI，让它修复配置。

很多构建者一开始干脆跳过 GitHub Actions 文件，直接在自己的终端里用一条命令部署。要在第一天就看到东西上线，这往往是最快的路。同样的 Cloudflare Pages 部署，从命令行看是这样：

```
# install the CLI once
npm install -g wrangler

# build your site, then push the output folder live
npm run build
wrangler pages deploy dist --project-name=my-app
```

第一次运行会让你通过浏览器登录；之后就是一条命令的事。取舍很简单：终端命令适合“现在就发”，GitHub Actions 文件适合“永远自动地发”。大多数项目从前者开始，等部署成了家常便饭再加上后者。看你处在哪个阶段，就让 AI 给你那个合适的。

几条能省下真切痛苦的指挥技巧：

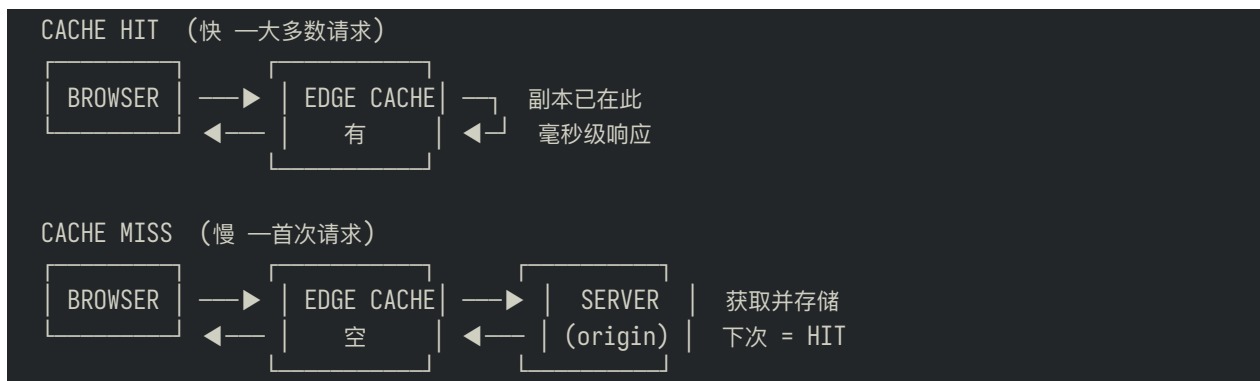
- 让 **AI** 先解释再行动。“在写配置之前，告诉我哪种托管类型适合这个项目，以及为什么。”这能及早抓住过度工程。
- 让密钥远离代码。API key 和密码要放进平台的环境变量 / 密钥存储中，绝不要放进你提交的文件里。让 AI 使用一个密钥引用，就像上面的 CF_API_TOKEN 那样。
- 尽早部署，频繁部署。在第一天就让一个“hello world”上线。一条从一开始就在运行的流水线，比一条在发布时才拼凑起来的流水线要好调试得多。
- 提前问清楚回滚的办法。“如果这次部署坏了，我怎么回到能用的那个版本？”好的平台会把你之前的部署放在一键之遥。在你需要之前就知道有撤销按钮，能把凌晨两点的恐慌变成一个耸肩。

给单打独斗构建者的推荐默认路径

这是一条已经对无数个单人项目奏效的路线。每一步都只在你真正需要时才采取。

1. **原型：静态 + 一个托管后端服务。**把你的前端放到静态托管上（Cloudflare Pages、Vercel 或 Netlify）。对于数据和身份验证，依靠一个像 Supabase 或 Firebase 这样的托管后端，这样你就跳过了自己运行数据库。成本：基本为零。
2. **早期生产：添加边缘函数或无服务器函数。**当你需要服务器端逻辑时——保密的 API 调用、webhook、自定义身份验证——在同一个平台上添加函数。仍然缩减到零，仍然便宜，仍然一次部署。
3. **真实产品：把后端迁移到容器平台。**当应用超出了函数的能力范围（长时间运行的进程、想要一个持久服务器的框架、后台任务）时，把它容器化，部署到 Railway、Render 或 Fly.io。让 AI 来写 Dockerfile 和平台配置。
4. **扩展或专门化：到那时才考虑 VM 或像 AWS 这样的云提供商。**这是复杂度（以及你的账单）跳升的地方。大多数单打独斗的构建者从不需要这一步，而这是把事情做对的标志——而不是一种局限。

CDN (Content Delivery Network, 内容分发网络) 坐在这条路径的前面, 会根据它是否已有副本而极大地改变速度。缓存 HIT 从边缘即刻应答; 缓存 MISS 则必须先一路回到你的服务器:



要避免的错误是从第 4 步开始, 仅仅因为那是“真正的工程师”据说会用的东西。事实恰恰相反: 从简单开始才是资深的做法。你随时可以晋级到更多的控制权, 但你很难找回那些与你本不需要的基础设施搏斗而损失掉的几个月。

“托管 (*managed*)”到底意味着什么 (以及你仍然拥有什么)

托管服务, 就是平台替你打理那些枯燥又危险的部分——服务器、安全补丁、扩展、备份。“无服务器”“托管数据库”“平台即服务 (PaaS)”都是同一笔交易的不同口味: 你把运维交出去, 自己继续构建。对一个单打独斗的构建者来说, 这不是偷懒, 而是杠杆。你没花在打 Linux 安全更新上的小时数, 就是你花在真正产品上的小时数。

但“托管”不是“魔法”。无论平台打理多少, 总有几件事仍然是你的责任:

- **你的数据。**平台让数据库保持运行; 但确保你有一份你真正测试过能恢复的备份, 是你的事。把“导出”按钮按一次, 确认那个文件不是空的。
- **你的密钥。**API key 住在平台的密钥存储里, 但轮换一个泄露的密钥, 以及永远不把密钥粘进公开仓库或截图, 是你的活。
- **你的账单。**自动扩展很美妙, 直到它把你的账单也自动扩展了。设好消费提醒和上限——下一节整节都在讲这个, 因为这是最让人疼的那种翻车。
- **你的理解。**当半夜里有东西坏了, “平台会处理”只能管到一定程度。你不必是专家, 但你应该能读懂日志, 并把故障向 AI 描述清楚。这就是“理解你所发布的東西”这一理念浓缩成的一句话。

健康的心态: 把运维外包出去, 但别把理解外包出去。让平台去跑机器; 你来当那个知道机器在干什么的人。

一份快速的上线前清单

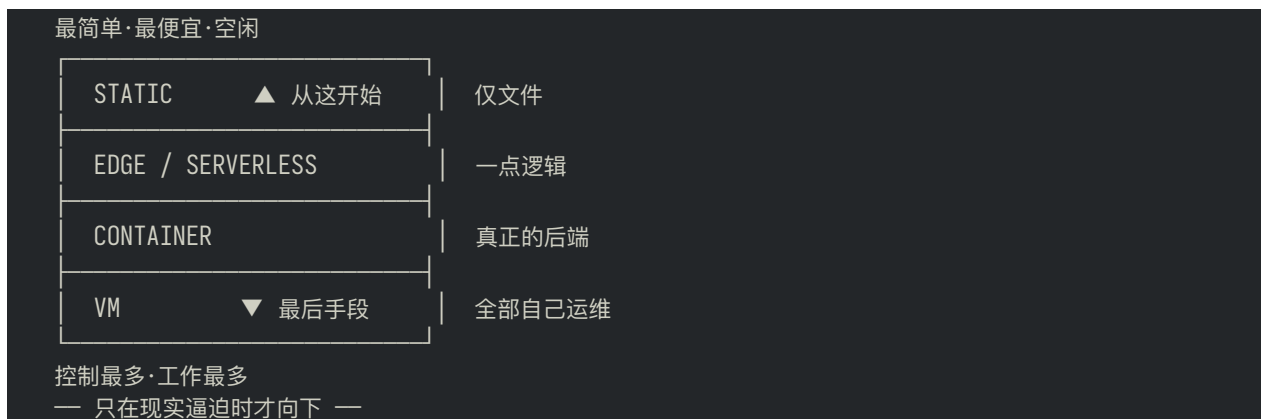
在你告诉别人去访问你的线上 URL (Uniform Resource Locator——网址) 之前, 先把这份清单走一遍。这里没有一项需要 DevOps 专长——大部分就是给 AI 的一条提示, 或者面板上的一次点击。

- **它确实能加载。**在一个全新的浏览器 (或你的手机) 里打开真正的 URL, 而不只是你本地的预览。那些“在我机器上能跑”、一上线就在公众面前 404 的发布, 多到能写进传说里。
- **密钥不在代码里。**推送之前, 在你的仓库里搜一下你自己的密钥。问 AI: “扫描这个项目里硬编码的 API key、密码或 token。”
- **消费上限已设好。**给每一个付费 API 设一个硬上限, 给你的托管账户设一个账单提醒。先设上限, 再庆祝。
- **HTTPS 已打开。**HTTPS (HyperText Transfer Protocol Secure) 是网页基础传输方式的加密版本——它就是给地址栏加上小锁的那个东西。现代平台会自动做这件事, 但确认一下那个小锁图标在那儿。浏览器会惩罚没加密的网站。
- **你知道怎么回滚。**趁现在什么都没着火, 先把“上一次部署”那个按钮找到。
- **有办法联系到你。**一个日志或错误上报工具, 哪怕是基础的, 好让问题的第一个信号不是一个愤怒的用户。让 AI 给你的技术栈接上错误日志。

这不是一次全面的生产环境审计——它是一份短清单, 专抓那些丢人、烧钱、又很容易避免的错误。每次发布都跑一遍, 直到它变成本能。

要点

基础设施是一架梯子，不是一次跳跃。静态在顶部，VM 在底部，而你的大部分工作都可以愉快地待在上面的横档上。



了解这五个类别，选择能完成工作的最简单的那个，让 AI 来写配置，并尽可能早地把东西上线发布。剩下的，只不过是现实——而且只有当现实——强迫你时，往下移动一档而已。

回顾与练习

关键点

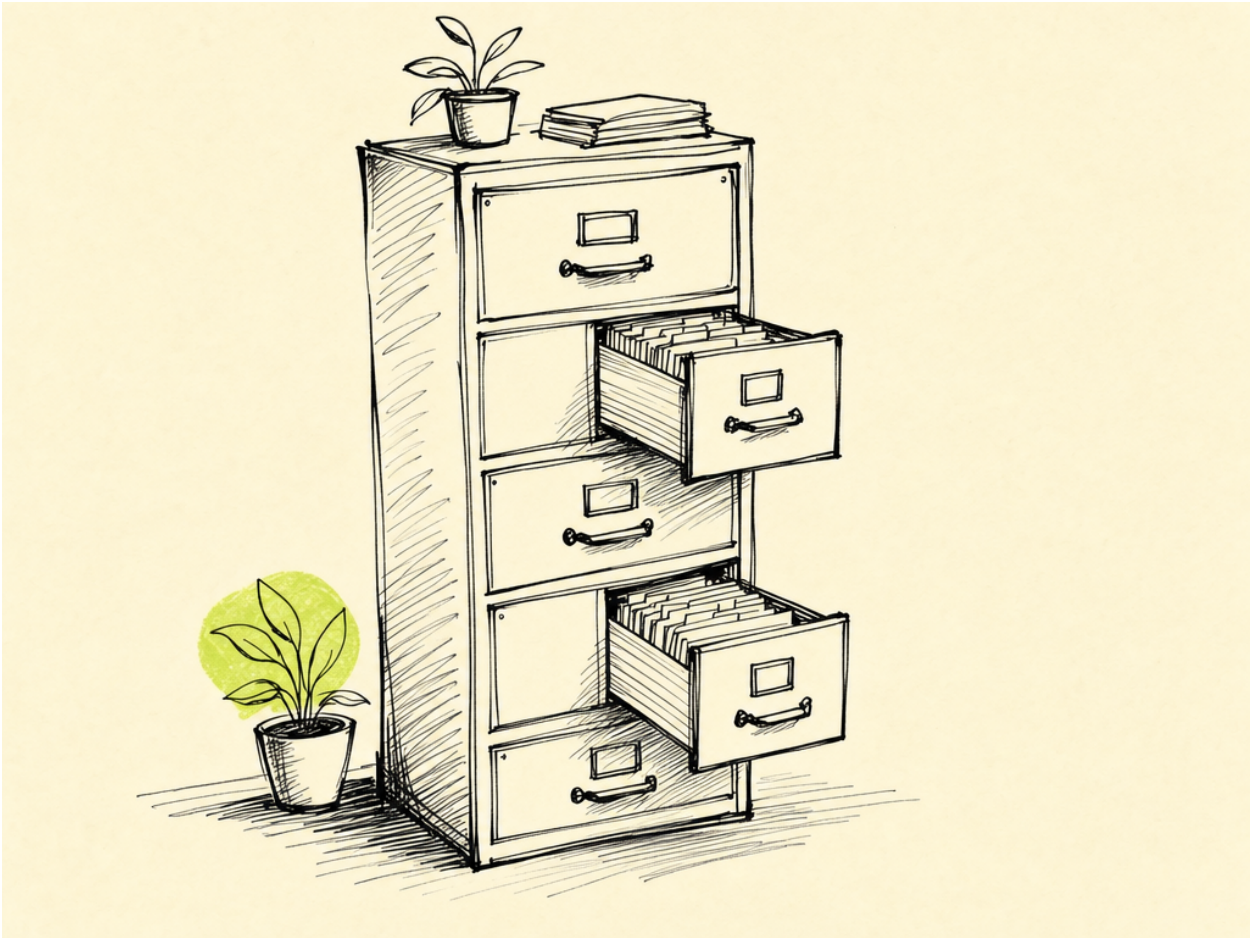
- 基础设施是一架五级的梯子—静态托管、Serverless、PaaS、托管容器，以及裸 VM —你应当选择能完成任务的最简单的那一级。
- 优先选择**托管服务**：把运维（服务器、补丁、扩缩容、备份）外包出去，把理解留在自己手里。
- 让 AI 来写部署配置和 CI（Continuous Integration，持续集成——每次推送代码时自动运行检查的机器人）工作流，但在信任之前先读一遍。
- “托管”不是“魔法”—你的数据、密钥、账单和理解，依然是你的责任。
- 在上线之前设好支出上限、确认 HTTPS，并知道如何回滚。

动手试试

拿一个你已有的小项目（哪怕只是一个 HTML 页面）部署到静态托管上。然后让 AI 生成一个每次 push 都重新部署的 GitHub Actions 工作流，逐行读懂生成的 YAML，并确认你能在托管商的控制台里找到“回滚到上一次部署”的按钮。

```
I have a static site in this repo. Walk me through deploying it to a free static host, then generate a GitHub Actions workflow that rebuilds and redeploys on every push to main. Explain each step of the YAML, and tell me where in the dashboard I roll back a bad deploy.
```

数据库基础设施



每一个真实的应用最终都需要记住一些东西——用户、订单、消息、设置。那份记忆存在于数据库里。对初学者来说，“数据库”可能感觉像一个由专家把守的黑盒子。它并不是。本章揭开数据层的神秘面纱，让你能做出好的选择，并指挥 AI 安全地完成繁重的工作。

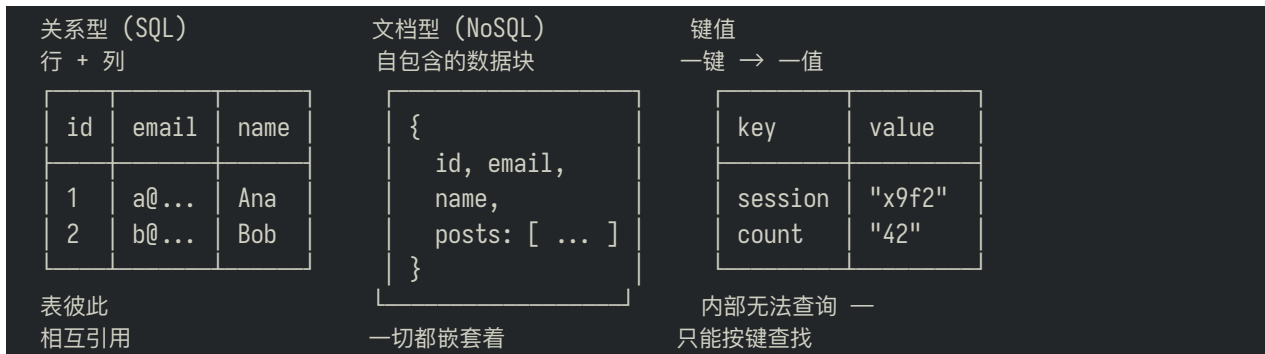
要早早内化的一点：代码是可替换的，数据通常不是。一个有 bug 的函数你可以花一个下午重写，但你上周二弄坏的某一行——或者你从未做过的备份——可能就永远没了。正是这种不对称，让数据层比你技术栈中几乎任何其他部分都更值得一点提前理解。

数据存储的三种形态

大多数数据库分为三个家族。你不需要去背产品——你需要认出哪种形态适合你的问题。

类型	例子	最适合	取舍
关系型 (SQL)	PostgreSQL, MySQL, SQLite	带关系的结构化数据；任何与金钱或准确性相关的关键场景	你必须预先设计 schema；把写入扩展到非常大的规模更难
文档型 / NoSQL	MongoDB, Firestore, DynamoDB	灵活或嵌套的数据、快速迭代、形态各异的记录	容易产生不一致的数据；关系和报表处理起来很别扭
键值型	Redis, Memcached, Cloudflare KV	缓存、会话、计数器、按已知键的快速查找	无法按内容查询；通常不是你的事实来源

这三种形态以确实不同的方式存储数据。把它们并排来看：



把关系型数据库想象成一组彼此知道如何相互引用的电子表格。你用来跟它对话的语言就是 SQL (Structured Query Language, 结构化查询语言)。对 90% 的项目来说一个好的默认选择：**从关系型数据库 (Postgres) 开始**。它成熟、可预测、强制结构化，并能处理真实应用不可避免会长出来的关系。当你的数据确实没有固定形态时，再去考虑文档存储；而键值存储则作为一个辅助层（缓存），而不是你的主存储。

要避免的陷阱是凭感觉选。“NoSQL（不使用 SQL 表的数据库——它存储灵活、自由形态的记录）扩展性更好”只对一小撮大多数应用永远碰不到的问题成立，而即使如此，代价也是放弃那些让数据可信赖的一致性保证。挑选与你的数据相匹配的形态，而不是在系统设计视频里听起来最唬人的那个。拿不准时，关系型是稳妥的选择——缓存或文档存储你以后随时可以加上，但解开一团乱麻的 NoSQL 数据模型却很痛苦。

托管 vs 自托管

你可以自己在一台服务器上运行数据库，也可以租用一个托管的 (Supabase、Neon、PlanetScale、RDS、Cloudflare D1)，由提供商来处理备份、打补丁和正常运行时间。

对于 vibe coding，**始终从托管开始**。自托管一个数据库意味着你现在要为备份、安全更新、磁盘空间，以及凌晨 3 点的恢复负责。那是真正的运维工作，会从构建中分散你的精力。托管服务花一点钱，却能节省巨大的时间和风险。只有当你有了具体的理由以及足以支撑的技能时，才晋级到自托管。

挑选托管提供商时，值得知道你实际在为什么付费，因为各家的营销页面看起来都差不多。有几点确实不同：

- **连接模型**。无服务器平台 (Cloudflare Workers、Vercel functions) 会不停地打开和关闭数据库连接，这可能耗尽经典的 Postgres 连接池。Neon 和 Supabase 这样的提供商正是为此提供了连接池器 (PgBouncer 或同类) —— 在无服务器代码里使用池化的连接字符串。
- **分支与备份**。Neon 和 PlanetScale 让你能像 Git 一样给数据库“开分支”，这样你可以在副本上测试迁移。确认时间点恢复 (point-in-time recovery) 已开启，并搞清楚它能回溯多久。
- **锁定 (lock-in)**。D1 和 DynamoDB 很出色，但会把你绑在一个云上。纯 Postgres (Supabase、Neon、RDS) 是可移植的——你的 schema 和查询能搬到任何 Postgres 主机上。对大多数初学者来说，可移植是更安全的长期选择。

Schema 设计基础

schema 是你数据的蓝图：存在哪些表、它们有哪些列，以及它们如何关联。好的 schema 设计大多是常识加上几个习惯。

- **每一种“东西”对应一张表**。用户、帖子、订单——各自拥有自己的表。
- **每一行都需要一个唯一的 ID**（一个主键）。
- **用外键链接表**，而不是到处复制数据。
- **选择诚实的类型**。金钱是 decimal，不是 float。日期是时间戳，不是文本。
- **不要重复**。如果一个用户的邮箱存在于五张表里，你最终会有五个不同的邮箱。

下面是一个博客的简单 schema，正是 AI 会为你生成的那种：

```
CREATE TABLE users (
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  email       TEXT NOT NULL UNIQUE,
```

```

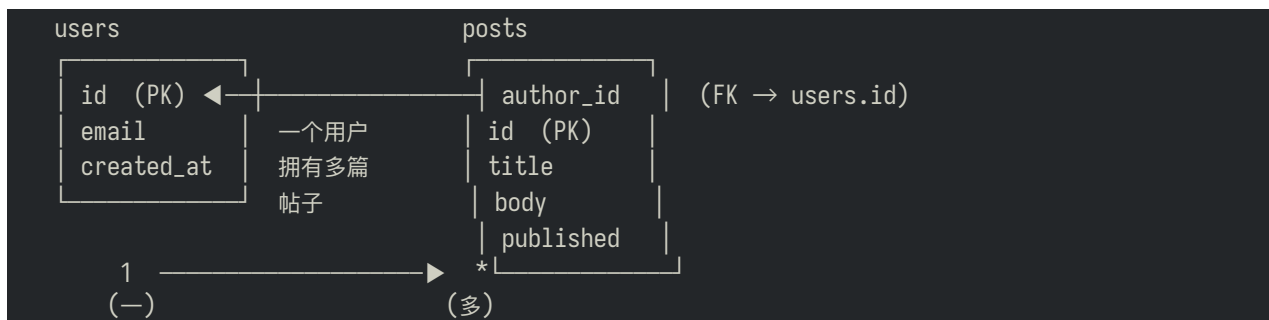
    created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE posts (
  id          UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  author_id   UUID NOT NULL REFERENCES users(id),
  title       TEXT NOT NULL,
  body        TEXT NOT NULL,
  published   BOOLEAN NOT NULL DEFAULT false,
  created_at  TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE INDEX idx_posts_author ON posts(author_id);

```

`author_id ... REFERENCES users(id)` 这一行就是外键。它把每篇帖子接回到恰好一个真实用户，于是数据库自身就会拒绝存储一篇没有有效作者的帖子：



注意 `REFERENCES`（一个把帖子绑定到真实用户的外键）以及 `NOT NULL` 和 `UNIQUE` 约束。这些是数据库为你强制执行的护栏，所以即使你的代码有 `bug`，坏数据也无法溜进来。

一个值得有意添加的约束，是当父记录消失时会发生什么。默认情况下，删除一个仍被帖子引用的用户会失败——而这往往正是你想要的。但你可以显式指定：`REFERENCES users(id) ON DELETE CASCADE` 会把帖子也一并删除，而 `ON DELETE RESTRICT` 会阻止该删除。有意去选，胜过在生产环境里以惨痛方式发现默认行为。同样的“让数据库保护你”的直觉也适用于 `CHECK` 约束（`CHECK (price >= 0)`）——它在源头拒绝荒谬的数据，而不是去信任每一条代码路径都会做校验。

迁移：安全地更改 *Schema*

你的 *schema* 会变化——你会添加一列、重命名一张表、加一个索引。把它想象成对一份共享文档的修订追踪：每一次更改都按顺序记录下来，于是每个人的副本都能用同样的方式更新到最新。一次**迁移 (migration)** 是一个版本化的、可重复的脚本，描述这样一次更改。迁移之所以重要，是因为你笔记本上的 *schema*、你队友笔记本上的 *schema*，以及生产环境里的 *schema* 必须保持完全一致。

让迁移保持安全的规则：

- 绝不要在生产环境里手动改数据库。写一次迁移。
- 迁移只往前走。每一次都是一个新文件；你不要去重写那些已经运行过的旧文件。
- 谨慎进行破坏性更改。删除一列或一张表会永久删除数据。先有一份备份。
- 追加式更改最安全。添加一个可空的列很少会破坏什么；重命名或移除一个列则可能破坏正在运行的代码。

迁移构成一条有序的链：每一次都是一个编号步骤，把数据库的每一份副本从一个已知状态推进到下一个，于是你的笔记本、队友的笔记本和生产环境最终都完全一致：





像 Prisma、Drizzle，或 Rails 迁移这样的工具会为你生成并跟踪这些。让它们去做。

一次迁移由两半组成：要应用的更改（up）和如何撤销它（down）。一个值得信赖的工具会把两者都生成出来，这样一次糟糕的部署就能干净地回滚：

```
-- up: 添加一个可选的 bio，可在线上安全地应用
ALTER TABLE users ADD COLUMN bio TEXT;

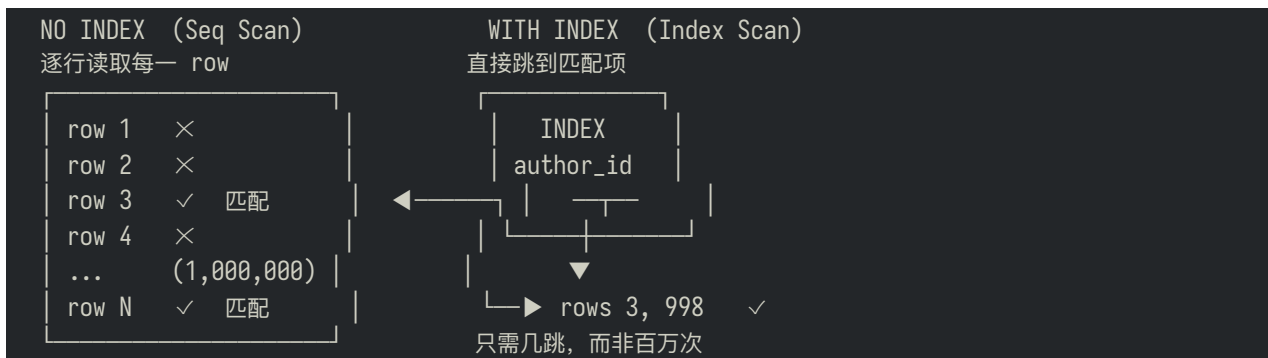
-- down: 把它反转回去
ALTER TABLE users DROP COLUMN bio;
```

真正危险的迁移是重命名和类型更改，因为旧代码和新 schema 会短暂地相互矛盾。专业的做法是把它们拆成追加式的步骤：添加新列、回填它、把代码切换到读取它，只有在之后的一次部署里、当没有任何东西再引用旧列时，才把旧列删掉。它感觉很慢，但这正是一次平滑发布与部署途中每个用户都看到 500 错误激增之间的区别。

索引

一个索引 (*index*) 就像一本书后面的索引：它让数据库直接跳到匹配的行，而不必扫描每一行。没有索引，查询会随着数据增长而变慢——在 100 行时还好，在 1,000,000 行时就痛苦了。

没有索引，数据库会逐行依次检查；有了索引，它就直接跳到匹配项——就像书的索引把你送到正确的页码，而不必从头读到尾：



实用的指导：

- 为你频繁筛选或连接的列建立索引（例如 `author_id`、`email`）。
- 主键会自动建立索引。
- 不要给所有东西都建索引——每个索引都会加速读取，但会拖慢写入并占用空间。
- 当你看到一个慢查询时再添加索引，而不是预先给每一列都加。

弄清楚某个索引到底有没有用的诚实办法，是去问数据库，而不是去猜。每个 SQL 引擎都有 EXPLAIN（在 Postgres 里是 EXPLAIN ANALYZE），它会展示它将采用的计划：

```
EXPLAIN ANALYZE
SELECT * FROM posts WHERE author_id = '...';
```

如果在一张大表上输出说 Seq Scan，那就是数据库在读取每一行——一个索引会有帮助的信号。加上一个之后，它应该切换为 Index Scan。这是一个很适合交给 AI 的任务：把慢查询和 EXPLAIN 的输出贴上去，问该加哪个索引以及为什么。你通过理解答案来保持掌控，而不是靠背索引理论。

缓存

一个**缓存 (cache)** 存储一个昂贵问题的答案，这样你就不必再问一遍。如果你的主页为每一个访客都运行同一个繁重的查询，那就把结果在 Redis 里缓存 60 秒，瞬间提供给用户。

缓存是一种性能优化，不是事实来源。经典的初学者陷阱是忘记**缓存失效 (cache invalidation)** ——缓存的副本变陈旧了，用户看到的是旧数据。经验法则：只有当你测量出一个真实的瓶颈时才缓存，并且始终对缓存值何时过期或刷新有一条清晰的规则。

有两种干净的策略，把它们搞混会引发大多数缓存 bug。**基于时间的过期 (TTL—Time To Live)**，就像盖在缓存值上的**保质期**说“这个值在 60 秒内有效，之后就扔掉”——简单，而且在略微陈旧的数据无害时（一个浏览计数、一个主页信息流）很合适。**基于事件的失效**说“在底层数据改变的那一刻就删除缓存值”——正确，但你必须在每一条写入路径上都记得这么做，而那恰恰是人们出岔子的地方。拿不准时，优先用短 TTL：一个最多陈旧一分钟的值，比一个因为你忘了一次失效而永远陈旧的值，是更小的 bug。

队列

一个**队列 (queue)** 让你能说“稍后再做这件事”，而不是“现在就做这件事”。当一个用户注册时，你不想让他们一边等着你发送欢迎邮件、调整他们的头像尺寸。把这些任务推到一个队列里 (RabbitMQ、BullMQ、Cloudflare Queues、SQS)；一个后台 worker 会把它们取走。用户得到一个即时的响应。

对任何缓慢的、可能失败并重试的，或者不需要在你响应用户之前完成的任务，都使用队列。

大多数初学者会撞上的坑是，一个入队的任务可能运行不止一次——一次网络抖动让 worker 重试，于是欢迎邮件就发了两遍。修复办法是让任务**幂等 (idempotent)**：把它们设计成运行同一个任务两次与运行一次效果相同（发送前先检查“我不是已经发过了？”）。也要预先决定，一个反复失败的任务该怎么办；成熟的队列会把它路由到一个死信队列，这样一条毒消息就不会把排在它后面的一切都堵死。

常见的初学者错误

- 因为 **NoSQL** 听起来很现代就去用它。大多数应用都有关系；SQL 处理它们更好。
- 把金钱存为浮点数。众所周知 $0.1 + 0.2$ 并不等于 0.3 。使用 decimal 类型。
- 没有备份。托管服务通常会自动备份——去确认它，别去假设它。
- 把密钥或巨大的文件放进数据库。文件属于对象存储 (S3、R2)；数据库存储那个链接。
- 让 **AI** 写把用户输入直接拼进去的原始查询。这会导致 SQL 注入。使用参数化查询。
- **N+1 查询**。一个对每个项都跑一次查询的循环（取 100 篇帖子，然后再分别做 100 次作者查找）会猛锤数据库。用一次带连接 (join) 的查询取回相关数据，或者把查找批量化。
- 从不做负载测试。一个在你笔记本的 50 行上瞬时的查询，在生产的 500 万行上可能会爬。在你信任性能之前，先灌入一份现实数量的假数据。

指挥 AI 构建你的数据层

AI 在 schema 设计和查询编写方面非常出色——前提是你给它约束去据此设计。不要说“做一个数据库”。描述领域、关系，以及你不可妥协的底线。

一个强力提示看起来像这样：

Design a PostgreSQL schema for a task-management app.

Entities and relationships:

- A user can own many projects.
- A project has many tasks.
- A task belongs to one project, has a status (todo / doing / done), and an optional due date.

Requirements:

- Use UUID primary keys and created_at timestamps.

- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL, plus a forward migration file.
- Explain any trade-offs you made.

然后用本章的清单来**审查输出**：诚实的类型、外键和约束齐备、合理的索引、没有重复的数据。当你请求查询时，明确要求**参数化查询**，这样用户输入就绝不会被拼接进 SQL。而对于任何删除或重命名某物的迁移，让 AI 把它点出来，这样你可以先备份。

数据库是你应用中最难撤销错误的部分——数据一旦损坏或丢失，就不总能回来。这正是为什么这些基本功值得了解：不是为了让你自己写出每一行，而是为了让你能自信地指挥 AI，并在那些危险的操作上线之前抓住它们。

回顾与练习

关键点

- 大多数应用只需要三种存储形态——关系型、文档型、键值型——中的一种，而关系型是安全的默认选择。
- 从托管数据库开始：数据和模式决策由你掌握，服务器的运行由平台负责。
- 好的模式设计意味着诚实的类型、外键与约束、合理的索引，以及没有重复的数据。
- 把每一次迁移都当作一道单向门：在 **drop** 或重命名任何东西之前先备份，并仔细审查 AI 写的迁移。
- 始终要求参数化查询，使用户输入永远不会被拼接进 SQL。

动手试试

挑一个小小的应用点子，先用平实的中文把它的实体和关系写出来。然后把这段描述交给 AI，请它给出一份模式加一个前向迁移——在运行任何东西之前，用本章的清单审查结果。

```
Design a relational schema for a personal bookmarks app.
```

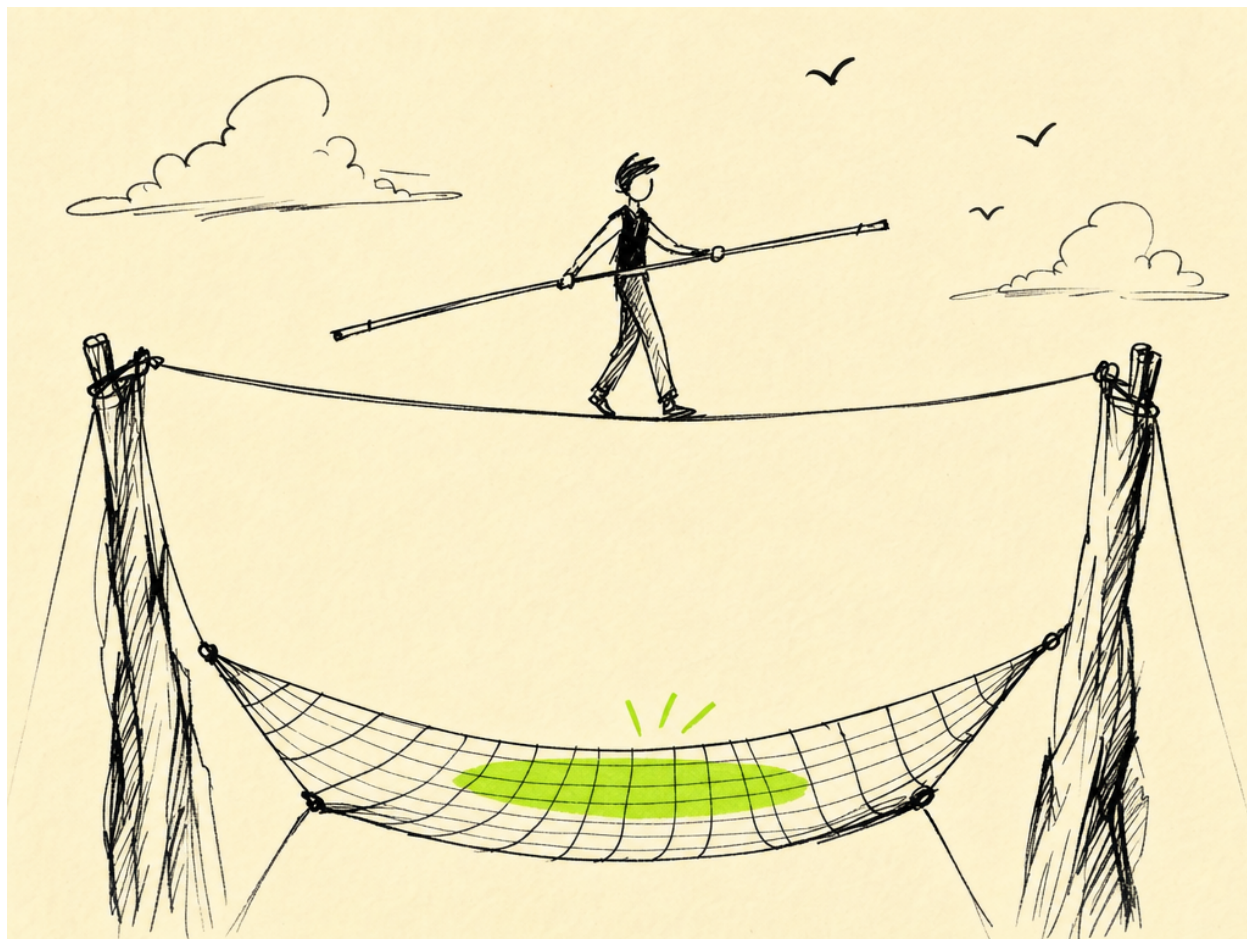
```
Entities and relationships:
```

- A user has many bookmarks.
- A bookmark belongs to one user, has a URL, a title, and many tags.
- A tag can apply to many bookmarks.

```
Requirements:
```

- Use UUID primary keys and created_at timestamps.
- Enforce relationships with foreign keys.
- Add indexes for the columns we'll filter on.
- Give me the schema as SQL plus a forward migration file.
- Flag any migration step that drops or renames so I can back up first.

防护栏、测试与审查



Vibe coding 让你跑得飞快。但危险在于，你也可能朝着错误的方向飞快前进——而且因为 AI 听起来信心十足，你可能直到某个东西在真实用户面前崩溃时才会察觉。防护栏就是让你在保持速度的同时不至于翻车的方法。

本章并不是要把你的项目变成一套官僚体系。它讲的是一小撮廉价、自动的检查，能在 AI（以及你自己）的错误抵达生产环境之前就把它抓出来。设置一次，它们就会永远在后台为你工作。本章的全部哲学就是一句话：永远不要发布你不理解的东西。防护栏的作用，就是把“理解它”变得足够廉价，让你每一次都真的会去做。

为什么指挥 AI 时你需要防护栏

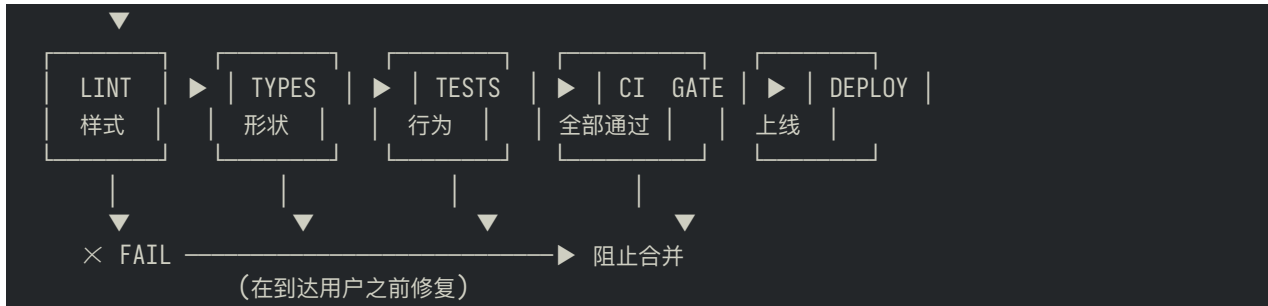
AI 并不知道你的应用本该做什么。它只是在向看起来合理的模式靠拢。这意味着它会乐此不疲地：

- 为了“修复报错”而删掉一个它不理解的检查
- 在你只要求它改一个文件时，却重构了三个文件
- 引入一个 happy path 不会暴露出来的隐蔽 bug
- 在它根本没运行过代码的情况下，自信地告诉你已经做完了

防护栏把“相信我”变成“拿出证据”。目标是构建一个让东西很难被意外搞坏的系统，这样你就能一直说“先试试看”，而不必屏着呼吸。

把这些防护栏想象成一条流水线，每一处改动在抵达真实用户之前都必须穿过它。每道关卡都廉价而自动，其中任何一道都能把一个坏改动当场拦下：

代码变更

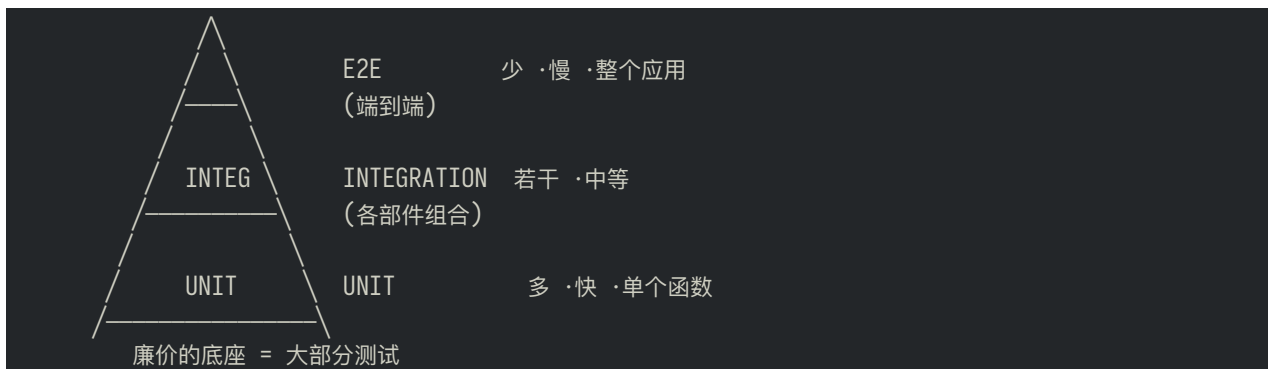


这件事在 AI 身上比在人类队友身上更重要，有一个更深层的原因。一个删掉校验检查的人类，通常知道自己在冒险，并且能感到那份分量。AI 什么也感觉不到。它会移除一个守卫、重写一个文件、然后报告成功——无论这个改动是绝妙还是灾难性的，都用同样欢快的语气。你读不出它的肢体语言，因为它根本没有肢体语言。防护栏就是它的肢体语言——它是关于这份工作是否真正稳妥，你能得到的唯一诚实信号。

测试：你的安全网

测试就是用来检查其它代码是否如你预期那样运行的代码。一旦写好，它会在几秒内运行完，并立即告诉你是否有东西回退 (regression) 了。

一套健康的测试套件长得像一座金字塔：底座是大量又小又快的单元测试，中间是较少的集成测试，顶端只有一小撮缓慢的端到端测试。你的大部分安全感，都廉价地来自底座：



你不需要 100% 的覆盖率。先从那些一旦坏掉就最伤的东西开始测：登录、支付、无法重新生成的数据、核心业务逻辑。AI 非常擅长写测试——往往比写功能还要强，因为预期行为是具体明确的。

下面这个 prompt 效果很好：

```

Write tests for the calculateDiscount function in src/pricing.js.
Cover: a normal case, the zero/empty case, the maximum-discount
boundary, and one invalid input. Use the existing test setup in
this repo (check how other *.test.js files are structured).
Run the tests and show me they pass before you finish.
  
```

最后那一行很关键。一定要让 AI 运行测试，而不只是写出来。一个从未被执行过的测试只是一个猜测。具体说明要覆盖哪些情况也很有价值，因为放任不管的话，AI 倾向于写出三种 happy path 的变体，然后就声称很周全了。真正能抓出 bug 的，是那些无聊的情况：空输入、边界值、本应抛出异常的东西。

它应当产出的东西的一个简单示例：

```

import { calculateDiscount } from "../pricing.js";
import { describe, it, expect } from "vitest";

describe("calculateDiscount", () => {
  it("applies a normal percentage discount", () => {
  
```

```

    expect(calculateDiscount(100, 0.2)).toBe(80);
  });

  it("returns the full price when discount is zero", () => {
    expect(calculateDiscount(100, 0)).toBe(100);
  });

  it("never discounts below zero", () => {
    expect(calculateDiscount(100, 1.5)).toBe(0);
  });

  it("rejects a negative price", () => {
    expect(() => calculateDiscount(-10, 0.2)).toThrow();
  });
});

```

当你之后改动某个功能时，先运行测试。如果它们变红了，你就在用户之前发现了破绽。当 AI 修复一个 bug 时，让它补一个针对旧行为会失败的测试——这样这个 bug 就不会悄无声息地卷土重来。这个习惯有个名字：回归测试 (regression test)。你遇到的每一个 bug，都是代码库付出代价换来的教训；回归测试就是你保住这个教训的方式。

一个提醒：测试本身也可能是错的。如果 AI 写了一个断言有 bug 的行为是正确的测试，它会永远通过，而你的应用会一直坏着。所以要读断言，而不只是看绿色对勾。一个通过的测试只意味着“代码做了测试所说的事”——你仍然得确认这个测试说的是对的事。

类型检查与代码检查：免费抓出的错误

有两种工具几乎不需要你费什么力气，就能抓出一整类错误：

- **类型检查** (TypeScript、mypy 等) 能抓出“你把一个 string 传给了期望 number 的地方”以及“这个值可能是 undefined”。AI 特别喜欢忽略那些被类型设计成不可能出现的边界情况。
- **代码检查 (Linting)** (ESLint、Ruff 等) 能抓出未使用的变量、不可达代码以及风格漂移。它能防止一个被 AI 反复改动的代码库慢慢糊成一团。

尽早把这些设置好，并告诉 AI 遵守它们：“运行类型检查器和 *linter*，并在你认为做完之前修复它们报出的所有问题。”把一次干净的类型检查当作“完成”定义的一部分。

不过，要警惕那些偷懒的逃生口。当类型检查器抱怨时，真正正确的修复是去处理它发现的那个情况——那个 undefined 值、那个错误的类型。而快速的修复，是用 `// @ts-ignore`、`any` 或者 `# type: ignore` 注释把抱怨压下去，AI 会不停地伸手去抓这些，因为它们能让红字消失。它们也恰恰在检查本想帮忙的那个位置，把检查变得毫无价值。如果你在 diff 里看到一条抑制 (suppression) 注释出现，把它当成一个问题，而不是一个解决方案：“这为什么是必要的？去修底层的类型，而不是忽略它。”

限定 AI 改动的范围

AI 造成损害的最大单一原因，就是放任它一次改太多东西。一个跨越九个文件、400 行的 diff 既无法审查，又很容易藏住 bug。

让改动保持小而集中：

- 一次只要求一个逻辑改动，而不是“既然你都改到这儿了，顺便也……”
- 明确告诉它：“只改 `src/auth.js`。别重构其它任何东西。”
- 经常提交可运行的状态，这样你总有一个干净的回滚点。
- 当它提议一个大重写时，让它把工作拆成你能一个一个审查的步骤。

范围小，才能让下一步——读 diff——真正变得可行。这里有一个复利效应：小的提交，其爆炸半径 (blast radius) 也小。如果一个范围紧凑的改动结果是错的，你回退一个提交，损失十分钟。如果一个铺得很开的改动是错的，你要么花一个小时去理清到底是哪部分搞坏了，要么扔掉一下午的工作。范围纪律是一份廉价的保险，在你知道自己会需要它之前就先买好。

带着批判地阅读 *diff*

diff 就是一份关于究竟改了什么的清单：红色行被删除，绿色行被新增。读 *diff* 是 *vibe coding* 中杠杆率最高的习惯，因为正是在这里你能抓出 AI 在做你没要求它做的事。

你不必理解每一行。扫一扫这些危险信号：

- 你没料到会变的文件。一个配置文件为什么被改动了？
- 被删掉的代码。它是不是为了“让测试通过”而删掉了一个校验检查、一个错误处理器，或一个测试？
- 密钥或硬编码的值。不该出现在代码里的 API (Application Programming Interface) key、密码或 URL (Uniform Resource Locator——网址)。
- 被禁用的检查。一个被跳过的测试、一段被注释掉的守卫、一个 `// @ts-ignore`，或者一个被放松的权限。
- 范围蔓延 (*Scope creep*)。离你要求的内容很远的改动。

如果有什么看起来不对劲，就问：“解释一下你为什么改了 X——我没要求改那个。”尤其要让 AI 为删除行为辩护。“它在导致报错”是一个去理解那个报错的理由，而不是删掉那个检查的理由。

一个养成习惯的实用做法：在你接受改动之前读 *diff*，绝不在之后。一旦代码被合并、应用看起来能跑，你就再也不会回去读它了——没有任何摩擦逼着你去读。审查的那一刻，必须就是决定的那一刻。工具在这里帮得上忙：`git diff --stat` 给你每个文件一行的摘要，让意料之外的文件立刻跳出来；在 *pull request* 页面上审查，会把整个改动呈现在一个可滚动的视图里，而不是散落在你编辑器各处。

审查关卡：合并前的一份清单

审查关卡 (review gate) 就是一份在改动上线前必须为真的简短清单。自己过一遍，或者让 AI 逐项确认：

- ☐ 我读了 *diff*，并理解改了什么、为什么改
- ☐ 改动被限定在我实际要求的范围内
- ☐ 测试通过（且新行为有对应的测试）
- ☐ 类型检查和 *linter* 都是干净的
- ☐ 没有遗留密钥、调试日志或被注释掉的代码
- ☐ 没有安全检查被悄悄移除
- ☐ 我自己试过这个改动，至少跑过 *happy path*

如果有哪一项你勾不上，那你就还没做完。

这份清单刻意保持简短，因为一个你会跳过的关卡，根本就不算关卡。尤其是当你跑得很快、改动“看起来没问题”时，会有一种直接放行的诱惑。对任何要面向用户的东西，请抵抗它。一个有用的小技巧，是让 AI 替你来填——把清单贴过去，然后说：“针对你刚做的改动，把这份审查关卡逐条走一遍。每一项，给出 *PASS* 或 *FAIL*，并附上一行证据。”AI 没运行过测试就没法诚实地写下“测试通过”，而强迫它逐行给出证据，会把一句含糊的“看着不错”变成一个你可以审计的东西。

CI 基础：让检查自动化

CI (持续集成, Continuous Integration) 是一个机器人，每次你推送代码时它都会运行你的那些检查，这样你就不会忘记去运行它们。大多数平台只需一个文件就能搞定。下面是一个最小化的 GitHub Actions 示例：

```
# .github/workflows/ci.yml
name: CI
on: [push, pull_request]

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
```

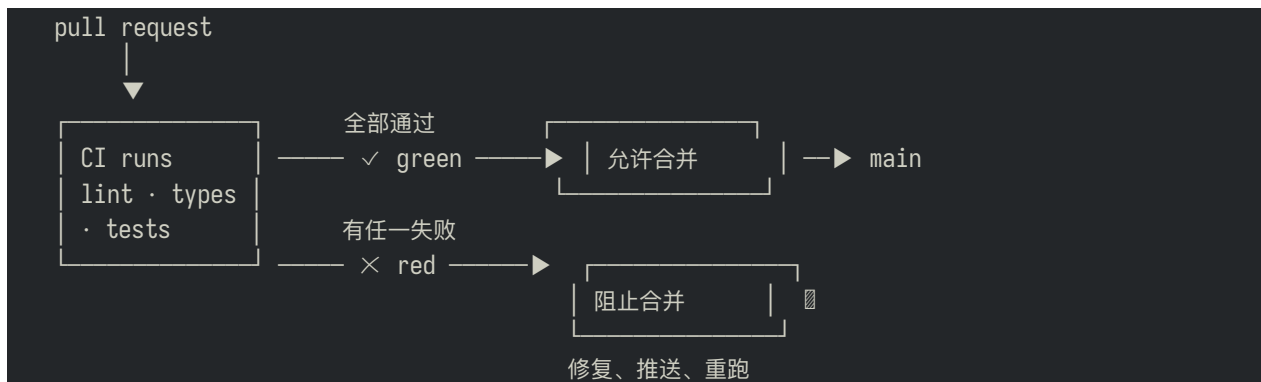
```

- run: npm ci
- run: npm run lint
- run: npm run typecheck
- run: npm test

```

现在每一次推送都会自动经过 lint、类型检查和测试。如果有任何一项失败，你会在代码能被合并之前看到一个红色标记。你甚至可以让 AI 为你的技术栈写好这个文件——“加一个 *GitHub Actions workflow*，在每个 *pull request* 上运行我们的 *lint*、类型检查和测试。”

一旦打开分支保护，红色的检查就不只是警告你——它会在构建重新变绿之前，从物理上锁住合并按钮：



CI 的重点并不是自动化本身，而是它把检查从你的记忆里挪到了流水线里——在那里，再疲惫的夜晚也跳不过去。等基本流程跑出绿灯之后，有两个值得做一次的升级。第一，把这些检查设为主分支上的必需状态（GitHub 上的“分支保护规则”），这样一个红色的构建会真正卡住合并按钮，而不只是显示一个你可以无视的警告。第二，当 CI 失败时，把失败日志直接贴给 AI：“这次运行的 CI 失败了，这是输出——诊断并修复它。”日志是一份精确的、机器生成的 bug 报告，而这恰恰是 AI 最擅长据以行动的那种输入。

默认安全的习惯

把这些汇聚成一种反射本能，让你能毫无畏惧地快速前进：

- 在分支上工作，而不是直接在生产环境上。分支是免费的；坏掉的生产环境可不是。
- 频繁提交小而可运行的状态。频繁的提交让回滚变得无足轻重。
- 永远让 AI 运行它写出来的东西——测试、应用、类型检查器。
- 给 AI 留一个终端，给你自己留一个，这样你能独立验证。
- 绝不让一个你没读过的改动上线。速度来自良好的防护栏，而不是来自跳过审查。

防护栏不会拖慢你——它们正是让你能心里不打鼓地说出“好，发布吧”的东西。设置一次，AI 就可以全力冲刺，而安全网会在它摔倒时接住它。

当防护栏尖叫时：先分诊，再修

防护栏只有在你好好回应时才有用，而失败模式是可预料的：构建变红了，你有点慌，然后把报错贴给 AI，附上“修一下”。有时这管用。但更多时候，它会造出一个修复——让红字消失，却没有处理真正出错的地方：AI 把测试削弱了、把类型放宽了，或者把那个失败的调用包进一个会吞掉问题的 try/catch 里。

在你放任它去“修”任何东西之前，花十秒做个分诊。防护栏的失败其实只有三类：

- 一个你刚刚引入的**真 bug**。检查正在尽职。去修代码，而不是修检查。
- 一个现在已经过时的测试。你是故意改了行为，旧的断言不再匹配了。把测试更新为断言新的正确行为——是经过深思的，而不是条件反射式的。
- 一个不稳定或本身就错的检查。真的很罕见。在这里要怀疑你自己；这正是人们用来倾倒真 bug 的那个桶，因为它是唯一一个能让你蒙混过关、继续往前走的桶。

那条能让 AI 保持诚实的指令，是逼它在改任何东西之前先做诊断：

"never discounts below zero" 这个测试失败了。

首先，告诉我它为什么失败——代码现在做了什么，对比测试期望什么。然后告诉我哪个才是对的：代码，还是测试。在我确认之前，什么都别改。

那一次停顿，就是”一个保护你的防护栏”和”一个被你悄悄训练 AI 去攻克”的防护栏”之间的区别。一个红色的检查是信息；你能做的最糟的事，就是在没懂它想告诉你什么之前，就花代价把它弄绿。

回顾与练习

关键点

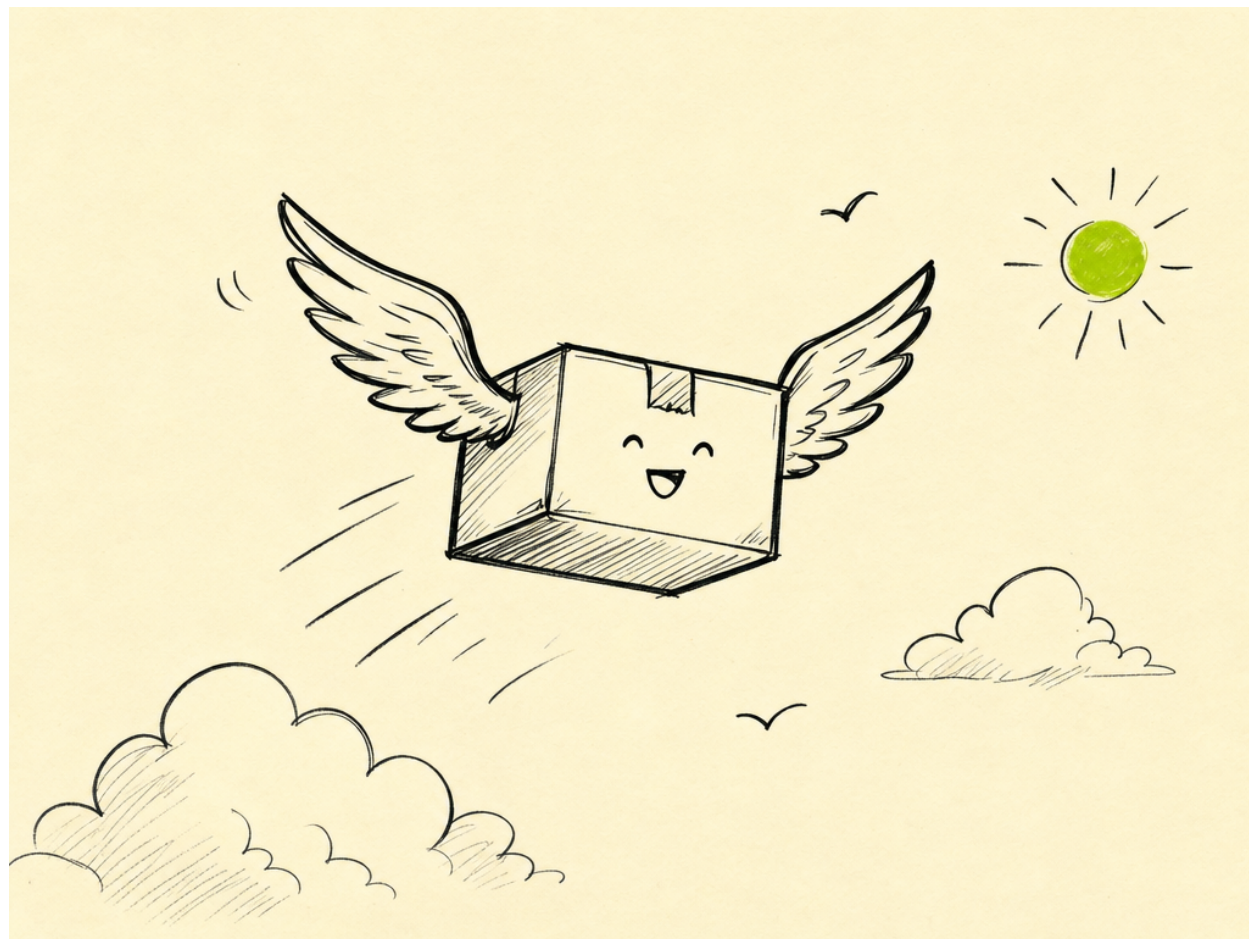
- 护栏让 AI 得以全速冲刺，同时让安全网接住跌落——它们让你更快，而非更慢。
- 测试、类型检查和 lint 会免费帮你抓住错误；把它们接入 CI，让检查自动运行。
- 把 AI 的改动范围限定得很窄，合并之前批判性地读每一处 diff。
- 红色的检查是信息——在让 AI “修复”它之前先分类（真正的 bug / 过时的测试 / 不稳定的检查）。
- 绝不上线你没读过的改动。

动手试试

在你的某个项目里，设置一个你还没有的自动检查——一个测试、一个 linter，或一个类型检查步骤——并把它加进 CI 工作流，让它在每次 push 时运行。然后故意弄坏一个小东西，确认在你修复之前检查会变红。

```
Add a CI workflow to this repo that runs the test suite, the linter,
and the type checker on every push and pull request. Generate the
config, explain each step, and tell me how to read the output when a
check fails so I can tell a real bug from an out-of-date test.
```

部署上线到生产环境



你一直在搭建。应用在你的机器上能跑，演示让一位朋友惊叹，想把它推向世界的冲动是真实的。这正是许多独立开发者卡住的时刻——不是因为代码坏了，而是因为“上线”感觉像是一门自带黑魔法的、完全不同的学问：DNS (Domain Name System, 域名系统，互联网的地址簿，把像 `yourdomain.com` 这样的名字翻译成服务器所在的那个号码)、服务器、密钥、支付 `webhook`。好消息是：这恰恰是 AI 最擅长的那种机械、有完善文档的工作。你的任务是指挥。它们的任务是敲命令。

但“指挥”这个词是承重的。这整本书的主题就是理解你所发布的東西。你不需要把那些标志参数背下来，但你确实需要知道每一步是为了什么——`webhook` 在保护你免于什么，为什么密钥绝不能进仓库，消费上限在给什么套笼子。当你理解了这件事的形状，你就能在 AI 那个自信满满的答案悄悄出错时察觉到，也能在第一次部署炸掉时把它救回来。在上线时被烫到的开发者，几乎总是那些粘贴了自己读不懂的命令的人。

本章会走完最后这一英里的全程，从一个能在本地运行的应用，到一个真实的、能让人们向你付款的 URL (Uniform Resource Locator, 统一资源定位符，人们在浏览器里键入的那个网址)。

选一个托管平台并部署

对大多数 `vibe coding` 出来的应用，你并不需要一台需要你时刻照看的服务器。现代平台拿到你的代码，几分钟内就能给你一个上线的 URL。常见的选择：

- 静态站点和前端：Cloudflare Pages、Vercel、Netlify、GitHub Pages。
- 全栈应用和 **API (Application Programming Interface, 应用程序编程接口, 一个程序用来与另一个程序对话的门户)**：Cloudflare Workers、Vercel、Render、Fly.io、Railway。
- 更重的后端：一个容器平台，或一台小型 VPS (Virtual Private Server, 虚拟专用服务器，相当于你在数据中心里租下的、一台永远开机的计算机的一块专属切片)。

正确的做法是告诉 AI 你搭了什么，让它来推荐。试试：“这是一个带 *Postgres* 数据库的 *Next.js* 应用。我想要一个最便宜、可靠、免费额度慷慨的托管平台。给我部署步骤和一个配置文件。”对那些真正约束选择的因素要说具体：你的框架、有没有数据库、是否需要后台任务或文件上传，以及你的预算。一个静态的营销页面，和一个带队列的全栈应用，住的地方截然不同。

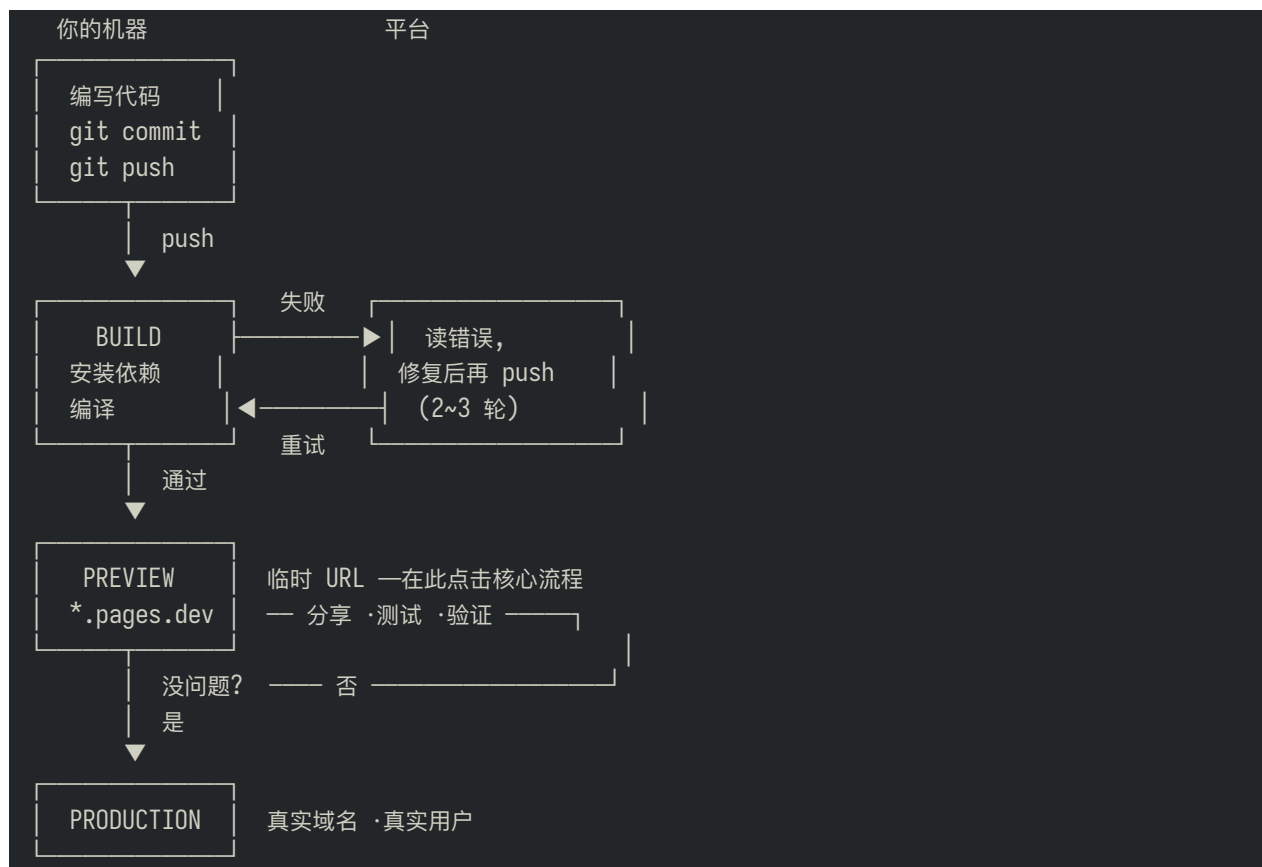
一旦选定了，部署通常就是几条命令：

```
# Example: Cloudflare Workers via Wrangler
npm install -g wrangler
wrangler login
wrangler deploy

# Example: Vercel
npm install -g vercel
vercel --prod
```

让 AI 生成平台配置（`wrangler.toml`、`vercel.json` 等），并解释第一次部署抛出的任何报错。第一次部署几乎总会抛出点什么——一个缺失的构建命令、一个错误的输出目录、一个 Node 版本不匹配、一个构建需要却找不到的环境变量（environment variable，从代码之外交给你应用的一项带名字的设置，就像放在门垫下、应用知道去哪里找的一把钥匙）。这些都不意味着你干了蠢事；它们意味着平台还需要你再告诉它一件事。把报错原样粘回去，让它给出修复方案，再问问为什么会发生，好让下一次少点意外。两三个回合，你就上线在一个临时 URL 上了。在庆祝之前，打开那个 URL，把核心流程点一遍——“部署成功了”和“应用能用”不是同一个论断。

每一次部署都走同一条流水线。预览那一步就是你的安全网：每一个变更都拥有自己专属的 URL，在它触达真实用户之前供你点击查看。



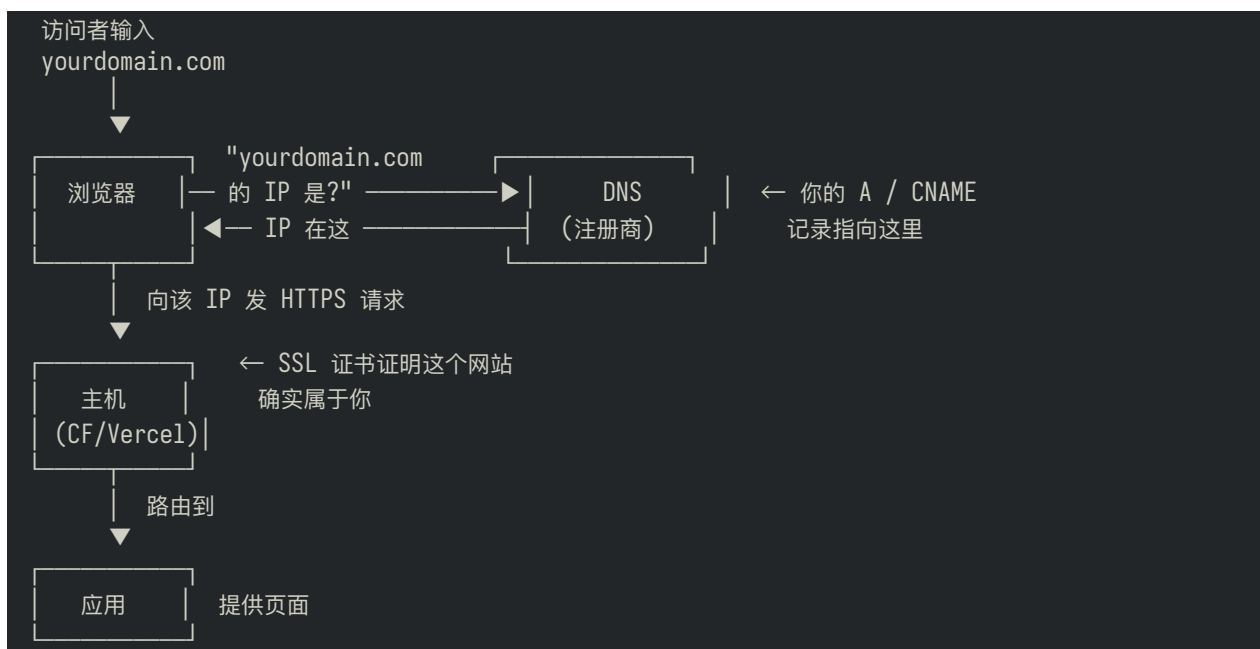
自定义域名与 DNS

一个 `*.workers.dev` 或 `*.vercel.app` 的 URL 拿来测试没问题，但你会想要属于自己的名字。流程如下：

1. 买下域名 (Namecheap、Cloudflare Registrar、Porkbun——挑一个)。
2. 把它的 DNS 指向你的托管平台。这通常意味着添加一条 CNAME 或 A 记录，或者彻底更换域名的 nameserver。
3. 在托管平台的控制面板里添加这个域名，让它自动签发一张 SSL 证书 (SSL certificate，证明这个站点确实属于你的数字身份证，它会点亮那把小锁，并开启 HTTPS——网络传输协议的安全加密版本)。

给你一个快速的心智模型，好让这些记录不再像魔法：一条 **A** 记录把一个名字直接指向一个 IP 地址；一条 **CNAME** 把一个名字指向另一个名字（当托管平台的 IP 可能在你不知情时变化时很有用）；而 **nameserver** 把整个域名交给一个服务商，让它替你管理每一条记录。大多数现代托管平台更偏好用 CNAME 或者整体切换 nameserver——它们想自己管理 SSL 证书和边缘路由。

下面是一个访客的请求所经过的整条链路，以及你设置的每一块分别住在其中的什么位置：



DNS 对新手来说是最令人困惑的一环，多半是因为术语和传播延迟。让 AI 来翻译。问它：“我的域名注册在 *Namecheap*，托管在 *Cloudflare Pages*。一步一步告诉我究竟要添加哪些记录、填什么值，以及如何验证它生效了。”然后用一次快速检查亲自验证结果，而不是仅仅刷新浏览器：

```

# 查询全世界实际看到的记录
dig yourdomain.com
nslookup yourdomain.com

# 确认证书已上线且未过期
curl -I https://yourdomain.com
  
```

DNS 改动的传播可能需要几分钟到一天不等的时间，而其中最大的骗子是你自己浏览器的缓存——在互联网其余部分早已更新之后，它还会乐呵呵地给你看一个过时的答案。如果它没有立即上线，那是正常的——在你以为出了问题之前，先用手机的蜂窝数据查一下，或者用一个在线的“DNS 传播”检测工具。

环境变量与密钥

你的代码需要 API key、数据库 URL 和各种 token。这些绝不能被提交到你的仓库里。密钥一旦进了 git 历史，实际上就等于永远公开了——在后来的提交里删掉那一行并不会把它从历史中抹去，而机器人会在几分钟内爬取新的公开仓库来寻找泄露的 key。两条规则能让你受用很久：

- 在本地把密钥放在一个 .env 文件里，并确保 .env 在你的 .gitignore 中。
- 在托管平台的控制面板里（或通过 CLI——Command-Line Interface，命令行界面，那个让你敲命令而非点击的文字提示窗口）把同样的值设置为生产环境的环境变量。

```
# .env (local only — never commit this)
DATABASE_URL=postgres://localhost:5432/app
STRIPE_SECRET_KEY=sk_test_xxx
SESSION_SECRET=change-me

# Set the production equivalents on the host
wrangler secret put STRIPE_SECRET_KEY
vercel env add STRIPE_SECRET_KEY production
```

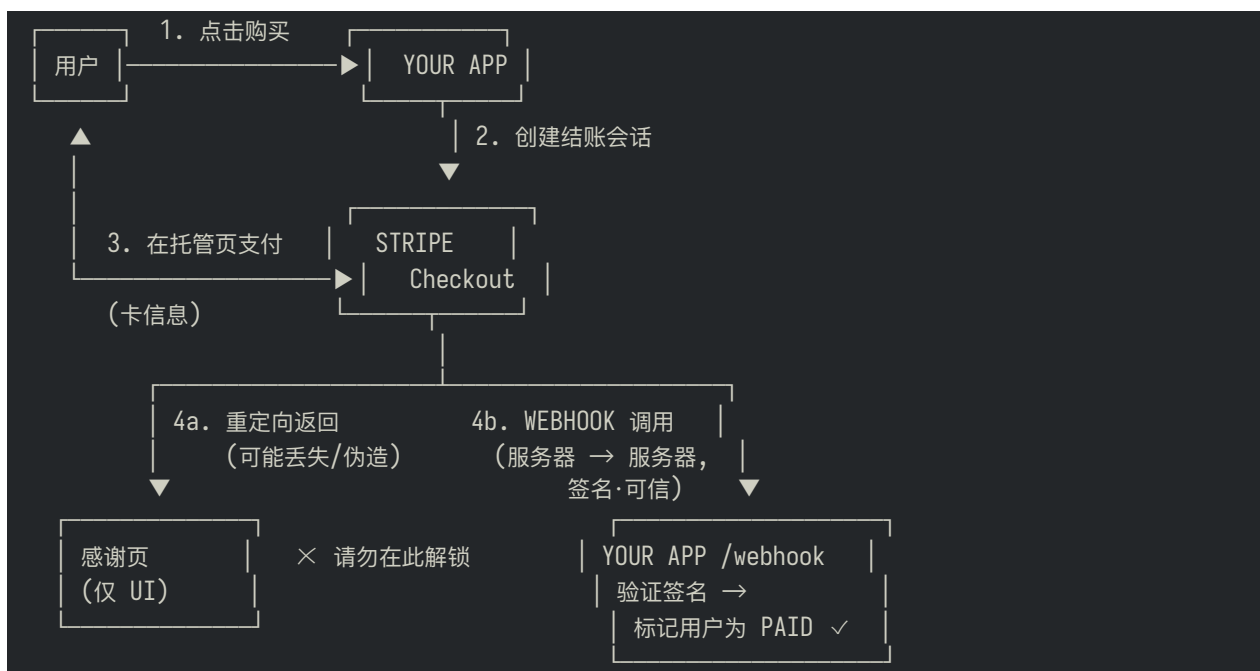
维护一个会被提交的 `.env.example`，列出每一个变量的名字（而非值），这样未来的你和 AI 都知道这个应用期望什么。注意上面那个 `sk_test_` 前缀——把你的测试 key 和正式 key 清楚地区分开，绝不要让一个正式 key 躺在一个你可能会粘进聊天的本地 `.env` 里。如果你曾经把一个密钥粘进了聊天，或者不小心提交了一个，就当它已经泄露，并立即轮换它：在服务商控制面板里生成一个新 key，在所有地方更新它，然后吊销旧的。轮换花你五分钟；一个泄露的数据库 URL 可能让你失去数据库里的一切。

接入支付

如果你想收钱，对独立开发者而言的标准路径是 Stripe（或者 Lemon Squeezy / Paddle，如果你想让它们替你处理销售税——作为登记在册的销售方（merchant of record），它们抽成更高，但让税务的头疼消失）。这些机制是重复且文档完善的，这让它们非常适合外包出去。你需要的几块拼图：

- 一个**结账 (checkout)** 流程——Stripe Checkout 是最快的；它直接给你一个托管好的支付页面，让你永远不必碰原始卡信息。
- 把 **webhook** 想象成支付公司在一笔交易刚一清算成功的那一刻就打电话通知你的应用，而不是让你的应用反复地追问“到底成没成？”。一个 **webhook** 正是如此：它是支付服务商会去调用的一个端点（**endpoint**，你服务器上的一个 URL），让你的应用知道一笔支付究竟何时真正成功了。这是新手会跳过的部分，然后他们纳闷为什么订单从来不会被履约。回到你站点的那个重定向可能被关闭、被拦截或被伪造；webhook 才是服务器对服务器的事实来源。绝不要只信任那个重定向；要信任 webhook。
- 一种**存储权益 (entitlements)** 的方式——在你的数据库里把用户标记为已付费。这样刷新一下或换一台设备，他们买的东西依然在。

关键的洞察在于你信任哪一根箭头。把用户带回你站点的那个重定向可能丢失或被伪造；真正解锁产品的，是服务器对服务器的那条 webhook 线：



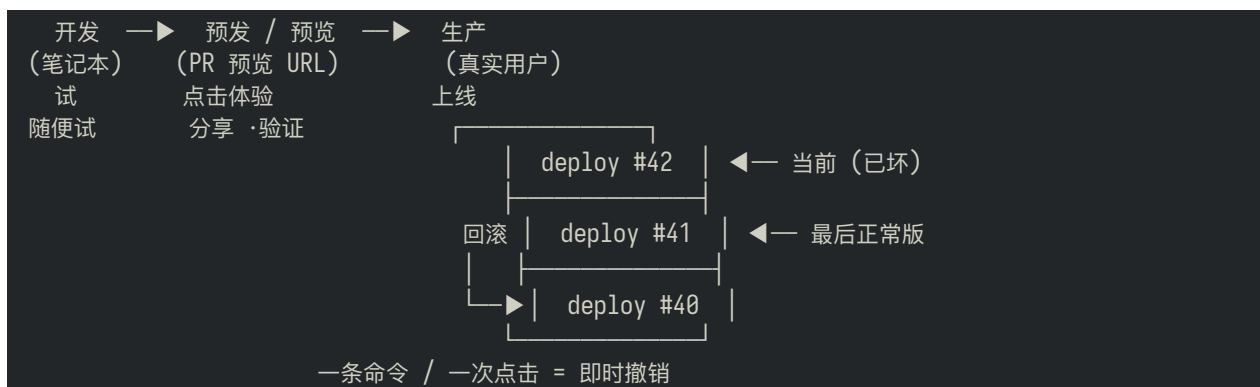
让 AI 把这三块都搭好，并解释 **webhook** 的签名验证，因为那是其中唯一对安全敏感的一步——没有它，任何找到你 **webhook URL** 的人都能伪造一个“支付成功”事件，把你的产品免费解锁。在切换到正式 **key** 之前，用 **Stripe** 的测试模式和它们的测试卡号（4242 4242 4242 4242 是最有名的那个）把一切都测一遍，并用 **Stripe CLI** 把 **webhook** 事件转发到你的本地机器，亲眼看着它们触发。不要跳过测试模式这场带妆彩排。第一笔流经你应用的真钱，应该是一笔你已经看过它成功运作十几次的交易。

预发布环境与回滚

有两个习惯，把一次能优雅恢复的上线，和一次变成午夜恐慌的上线区分开来：在变更触达真实用户之前先在安全的地方测试它，以及能够即刻撤销一次糟糕的部署。

一个**预发布（staging）**环境只不过是应用的第二份副本——同样的代码、独立的 **URL**、独立的（或一次性的）数据库——一个让你在生产之前先试一试的地方。大多数托管平台让这几乎免费：每一个 **pull request** 都自动获得它专属的**预览 URL**，于是你可以在合并之前点过一遍一个变更，并把它分享给某个人。对那些你没法轻易逆转的东西——比如一次数据库迁移——尤其要用它。

把这三个环境想象成一架变更要往上爬的梯子，而回滚就是那根能把生产环境直接落回上一个好梯级的绳子：



一次回滚是你对整个应用的撤销按钮。因为这些平台保留着此前的每一次部署，回到上一个好版本通常就是一条命令或一次面板点击：

```
# Cloudflare Pages: 重新部署一个已知良好的构建
wrangler pages deployment list
wrangler pages deployment rollback <DEPLOYMENT_ID>

# Vercel: 把一次之前的部署重新提升为生产环境
vercel rollback <DEPLOYMENT_URL>
```

在你需要它之前就知道这条命令，而不是在故障当中才去找。**vibe coding** 速度的全部意义在于发布很便宜——但只有当撤销发布同样便宜时，便宜的发布才让人感觉安全。

监控与错误追踪

一旦有人用你的应用，你就需要知道它何时坏了——最好在它们告诉你之前。这套轻量级的入门工具包：

- **错误追踪**：Sentry 会带着堆栈跟踪捕获并归类异常，并告诉你是哪个用户、哪一行触发了每一个错误。几行代码就能装好。
- **可用性监控**：一个免费的 **pinger** (UptimeRobot、Better Stack) 每隔几分钟敲一下你的站点，并在它宕机时给你发邮件。
- **日志**：搞清楚你的托管平台把它们放在哪 (wrangler tail、Vercel 控制面板等)，以便有什么看起来不对时你能去排查。

让 AI 加上 Sentry，并接一个基本的健康检查 (health-check) 端点——一个像 **/health** 这样的小路由，返回 **200 OK** 并确认数据库可达，让你的可用性监控可以去 **ping** 它。目标不是企业级的可观测性——它只是别从一条愤怒的推文里才得知出

了故障。

性能与缓存

慢吞吞的应用会流失用户，并在搜索中排名更靠后。你不需要在第一天就做微优化，但可以先把那些廉价的便宜拿到手：

- 想象把你的文件复制成很多份，分放在全世界各地的仓库里，于是每位访客都由离他最近的那一份来提供服务。那就是 **CDN**（Content Delivery Network，内容分发网络）——通过它来提供静态资源（大多数托管平台会自动这么做）。
- 缓存那些不常变化的昂贵响应和数据库查询。
- 压缩并懒加载图片；少发一点 JavaScript。
- 在 Chrome DevTools 里跑一遍 **Lighthouse**，把报告喂给 AI 来排出优先级修复。

问它：“这是我的 **Lighthouse** 报告。给我三个用最少力气换最大影响的改动。”让度量、而非凭空猜测，来驱动你优化什么——仅仅因为它感觉慢就花一整天去优化错误的东西，这真的很常见，而真正的代价其实是一条没建索引的查询，或一张你忘了压缩的 4 MB 首屏大图。

上线前清单

在你宣布任何事之前，过一遍这份清单。让 AI 帮你逐项验证，而不是凭记忆。

- ☐ 应用干净地部署，且生产环境 URL 能加载
- ☐ 自定义域名解析正常，且 HTTPS 有效
- ☐ 所有密钥都已在生产环境设置；没有一个被提交到仓库
- ☐ `.env` 已被 `gitignore`，且 `.env.example` 是最新的
- ☐ 支付已在测试模式下端到端测过（包括 `webhook`）
- ☐ 注册、登录以及核心 `happy path` 在生产环境都能用
- ☐ 错误追踪和可用性监控都已上线
- ☐ 你知道怎么回滚，而且已经测过一次
- ☐ 移动端布局已在一台真实手机上检查过
- ☐ 存在一个基本的隐私政策页和条款页
- ☐ 数据库备份已启用
- ☐ 有一条清晰的渠道让用户联系到你（邮箱或表单）

上线之后

发布是开始，而不是终点。第一个版本是一个假设；真实用户才是实验。盯着你的错误面板，读每一条消息，并快速发布小修复。要克制住一遇到摩擦就推倒重来的冲动——把尖锐的棱角打磨掉，在人们真正使用的东西上加倍投入，让其余的等一等。

这里最强大的习惯是一个紧凑的循环：注意到一个问题或一个想法，把它描述给 AI，发布改动，观察结果。因为机械性的工作被外包了出去，你可以用几小时而非几周来迭代。正是那份速度——而非任何单一功能——才是凭着 **vibe** 来构建的真正优势。你已经上线了。现在，继续前进。

回顾与练习

关键点

- 上线是一套 workflow，而不是一个瞬间：部署、观察、修补、重复——并且把回滚做成你已经测试过一次的东西。
- 在宣布任何东西之前先过一遍上线前检查清单，让 AI 逐项核验，而不要依赖你的记忆。
- 让测量来驱动优化——**Lighthouse** 报告永远胜过“感觉很慢”的直觉。
- 密钥放在生产配置里，绝不放进仓库；`.env` 加入 `gitignore`，`.env.example` 保持最新。
- 第一个版本是假设，真实用户才是实验，所以要用小而快的修复来迭代。

动手试试

挑一个你已部署的应用（或现在就部署一个很小的），逐行对照走一遍完整的上线前检查清单。对每一项你无法诚实勾选的，把失败的那一项贴给 AI，请它给出弥补差距的最小改动。最后点击线上站点的一个深层链接——而不只是首页——确认生产环境的

行为和开发环境一致。

本章提示词

```
Here is my deployed app and its stack: [describe app + hosting + services].  
Act as my launch reviewer. Walk this pre-launch checklist item by item  
and, for each, tell me how to verify it on the LIVE site (not in dev):  
- production URL loads over valid HTTPS on the custom domain  
- all secrets set in production; none committed to the repo  
- core happy path (sign-up, login, main action) works in production  
- payments tested end-to-end including the webhook (if any)  
- error tracking and uptime monitoring are live  
- I can roll back, and have tested it once  
- mobile layout works on a real phone  
- database backups are enabled  
For every item that fails, give me the SMALLEST change that fixes it.  
Do not assume anything passes — make me prove each one.
```

案例研究：从 0 到上线



理论是有用的，但没有什么比亲眼看着它发生更能教会你 **vibe coding**。本章会走过四个小而真实的项目，它们都是通过指挥 AI 从头到尾搭建并发布上线的。它们没有一个是玩具。每一个都有用户、一个域名，以及押在上面的金钱或时间。

对于每一个项目，你都会看到：想法、我们交给 AI 的 spec、我们选的技术栈、真正起作用的那些 prompt、差点让我们脱轨的障碍，以及它是如何上线的。把这些当作操作手册（runbook）来读，而不是精彩集锦。重点不在于这几个具体项目本身有多重要——而在于那些招式会重复。到第四个时，你会开始认出那个节奏：一段简短的 spec、一套托管的技术栈、靠证据驱动的调试，以及一次拿真实世界来检验的上线。

案例研究 1：一个多语言落地页 + 支付页

想法

一位朋友在卖一份售价 29 美元、面向移居韩国的外籍人士的 PDF 指南。她想要一个单页，全球加载都快，会说英语、韩语和中文，并且能在她完全不碰服务器的情况下收取银行卡支付。

Spec

在写任何代码之前，我们把 spec 控制在了一段话：

```
Build a single-page marketing site for a $29 digital product.
Three languages (en/ko/zh), auto-detected from the browser but
switchable with a toggle. One "Buy" button that opens Stripe
Checkout and redirects to a thank-you page on success. No backend
server I have to maintain. Must load fast from anywhere.
```

注意这段 spec 没有说什么：没有框架，没有设计系统，没有分析统计，没有邮件收集。这其中每一项都本可以是一个合理的功能，而每一项也都会拖慢首次上线。一段话 spec 的纪律，大半是把东西留在外面的纪律。

技术栈

“没有我要维护的后端”加上“全球都快”，直接指向了在边缘 CDN（Content Delivery Network，内容分发网络——把你的文件复制成多份存放在世界各地的数据中心里，于是每位访客都从就近的一份加载）上的静态托管，外加一个用于唯一需要密钥的事情——创建 Stripe session——的微型 serverless 函数。我们选择了在 Cloudflare Pages 上的一个静态站点，配上一个单独的 Pages Function。整个架构一句话就能装下——一个静态页调用一个函数、那个函数再跟 Stripe 对话——而正是这份简单，让它后来变得可调试。

整个系统就是三个盒子，而其中只有中间那个会接触到密钥：



关键的 prompt

我们从宽泛处起步，让 AI 提议结构：

```
Scaffold a static landing page (plain HTML/CSS/JS, no framework)
with a language toggle for en/ko/zh. Store the copy in a single
JS object keyed by language. Detect the default from
navigator.language. Keep it to one index.html plus one main.js.
```

然后是支付路径，密钥就住在这里：

```
Add a Cloudflare Pages Function at /api/checkout that creates a
Stripe Checkout session for a single $29 product and returns the
redirect URL. Read STRIPE_SECRET_KEY from the environment, never
```

```
hardcode it. The Buy button should POST to this function and then
window.location to the returned URL.
```

“never hardcode it”这一句配得上它的位置。任由它按默认行事，AI 会很乐意把一个占位密钥内联进去，好让示例“跑起来”，而占位密钥往往会变成一个真上线的真密钥。在 prompt 里大声说一次，比在评审里抓出来要便宜。

障碍

第一次线上测试失败了：Buy 按钮毫无反应，浏览器控制台显示一个 CORS (Cross-Origin Resource Sharing, 跨源资源共享——浏览器关于哪些站点被允许调用哪些站点的规则) 报错。我们没有去猜。我们把确切的报错粘了回去：

```
Clicking Buy logs: "Access to fetch at '/api/checkout' blocked
by CORS policy." The function and page are on the same domain.
Here's the function code: [pasted]. What's actually wrong?
```

AI 立刻发现了它：这个函数确实在以 JSON (JavaScript Object Notation, JavaScript 对象表示法——程序之间用来交换结构化数据的一种简单、易读的文本格式) 形式返回 Stripe URL，但 fetch 在函数的 OPTIONS 预检 (preflight) 被处理之前就发起了，因为我们误把静态页和这个函数部署到了两个不同的 Pages 项目里。真正的修复是部署拓扑，而不是代码。我们把函数移进了同一个项目的 /functions 目录，报错就消失了。

这个教训值得多停留一下：症状在代码里（一个来自 fetch 的 CORS 报错），但病因在基础设施里（两个项目而不是一个）。如果我们没有把配置粘上去，只让 AI“修这个 CORS 报错”，它就会硬塞上 Access-Control-Allow-Origin 头——一个能编译通过、把症状糊住、却让真 bug 继续活着的修复。把字面的报错连同它周围的上下文一起粘上去，并且如果第一个貌似合理的解释跟你的实际配置对不上，就别接受它。

上线

我们把正式的 Stripe key 作为加密的环境变量加进了 Pages 控制面板（绝不进仓库），把她域名的 DNS (Domain Name System, 域名系统，互联网的地址簿，把一个名字映射到一台服务器) 指向了 Pages，并用一张银行卡跑了一次真实的 29 美元测试购买，然后退了款。我们还特意走了一遍失败路径——一张会被拒的测试卡——以确认她不会被扣费，并且看到的是一条像样的消息而不是一片空白。一个下午就上线了。她当晚就做成了第一笔销售。

案例研究 2：一个带认证 + 数据库的小型 SaaS 工具

想法

一位自由职业设计师想要一个私人仪表盘，用来按客户记录可计费工时，并导出每月的 CSV (Comma-Separated Values, 逗号分隔值——一种纯文本的电子表格文件，任何电子表格应用都能打开)。没什么花哨的，但它需要账户（这样她的数据只属于她自己）和持久化存储。

Spec

认证和数据库抬高了赌注，所以 spec 在边界上变得更具体了：

```
A logged-in web app where a user can: sign up / log in with email,
create clients, log time entries (date, client, hours, note),
see a table of entries filtered by month, and download that month
as CSV. Each user only ever sees their own data. Mobile-friendly.
```

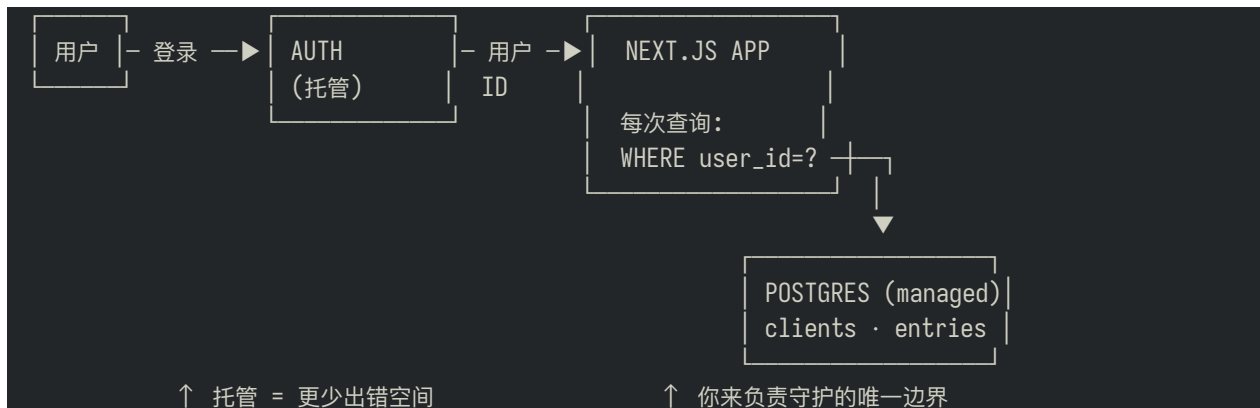
“Each user only ever sees their own data”看着像一行 UX 描述。它其实是把整个应用的安全模型压缩进了九个词里。把它写进 spec，意味着每次 AI 跑偏时我们都能指回这一句。

技术栈

对一个独立开发者来说，制胜的一手是选一个让认证和数据库都成为托管服务、而非你自己写的代码的技术栈。我们选了一个部署在 serverless 托管平台上的 Next.js 应用，配上一个托管的 Postgres 数据库，以及一个开箱即用的认证提供方，由

它替我们处理邮箱登录、会话和密码重置。能写错的代码越少，AI 能替我们写错的代码就越少。认证尤其是那种你几乎从不想跟 AI 一起手搓的类别：它的失败模式是静默的，爆炸半径是所有人的账户，而一个托管提供方早已被几百万次登录把边界情况捶打过。

托管的部件都在外圈；你自己掌管的那唯一一条规则在中间——每一条查询都按 `user_id` 过滤，于是一个用户永远读不到另一个用户的数据行：



关键的 *prompt*

我们让认证提供方自带的模板来挑大梁，然后指挥 AI 在它之上叠加业务逻辑：

```
We're using [auth provider]'s Next.js starter. Add a Postgres schema with two tables: clients (id, user_id, name) and time_entries (id, user_id, client_id, date, hours, note). Every query MUST filter by the logged-in user's id from the session. Generate the migration and the typed data-access functions.
```

“MUST filter by `user_id`”这一行是整个项目里最重要的一句话。我们在几乎每一个触及数据的 *prompt* 里都重复了这条约束，因为一个多用户应用里最可怕的单一 bug，就是一个用户看到了另一个用户的数据行。打字时这种重复感觉是冗余的；可正是这份冗余救了你，因为模型并不记得这条约束在一个个独立的 *prompt* 之间有多要紧。

至于导出：

```
Add a /api/export route that takes a month (YYYY-MM), pulls the logged-in user's time_entries for that month joined to client names, and streams a CSV download. Reject the request if there's no valid session.
```

障碍

在测试时，我们创建了两个账户，然后发现账户 B 能在一个下拉菜单里看到账户 A 的客户。这正是我们害怕的那个 bug。我们没有让 AI“修好它”，而是先让它把问题证明出来：

```
Account B is seeing Account A's clients. Show me every database query in the codebase that reads the clients table, and for each one tell me whether it filters by the session user_id. Don't fix anything yet — just audit.
```

这次审计浮出了一个查询——那个下拉菜单的加载器——它是在我们加上那条约束之前写的，于是溜了过去。我们让它补上了缺失的过滤条件，然后又要了一道守卫：

```
Add a single helper that every read goes through, which takes the session and injects the user_id filter, so no future query can forget it. Refactor the existing queries to use it.
```

这把一次性的修复变成了一个结构性的保证。接着我们让这个保证变得可测试，因为一道你没法验证的守卫不过是一种指望：

```
Write a test that creates two users, has each create a client,
then asserts that user A's session can never read user B's client
through any of the data-access functions.
```

教训：当 AI 引入一个安全 bug 时，别只修补这一个实例——要指挥它去消除这一整类错误，然后用一个一旦有人重新打开就会大声失败的测试，把这一类锁死。

上线

我们灌入了一个月的测试数据，导出了 CSV，在电子表格里打开它以确认数字和编码都正确，然后设置了一个强数据库密码，并把凭据从所有本地文件里轮换出去。我们部署到了 **serverless** 托管平台，在它的控制面板里加上了生产环境的环境变量，然后把 URL 给了她。她用一次真实的注册完成了自助入驻。整个构建只花了一个周末。

案例研究 3：一个个人自动化小工具

想法

一个纯粹利己的项目：一个脚本，每天早上抓取当天的日历事件和天气，格式化成一行摘要，发送到一个私人的 Telegram 聊天里。重点是别在喝咖啡之前还要打开三个应用。

Spec

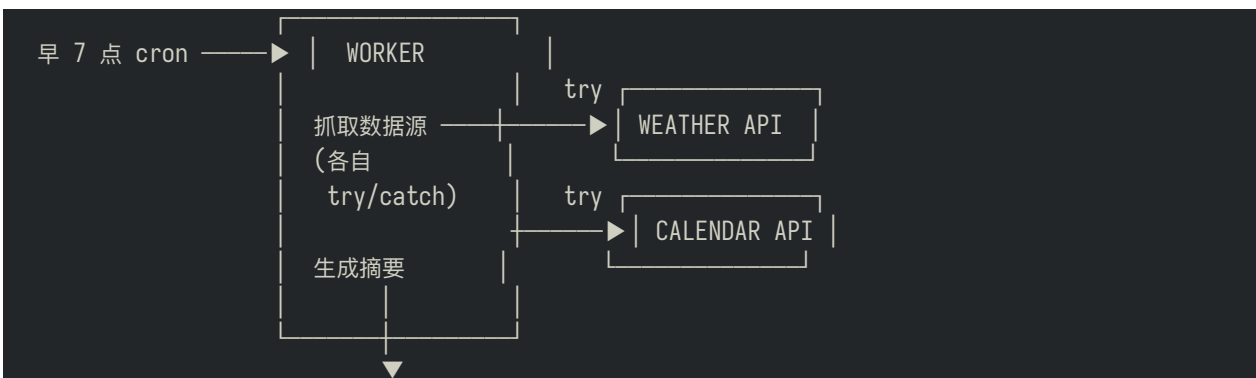
```
A scheduled job that runs at 7am my timezone. It reads today's
events from my calendar, fetches the forecast for my city, and
sends one Telegram message like: "3 events today, first at 9:30.
High 24C, rain after 4pm." If any source fails, still send what
you have and note what's missing.
```

最后那句话——优雅地降级而不是崩溃——是那种你必须明确提出来要求的行为。AI 不会自动假定它。不说出来，模型会去抓的默认就是“出错就抛异常”，而对一个早上 7 点的 cron 任务来说，那意味着一个静悄悄的早晨，以及对原因毫无头绪。

技术栈

一个单独的、定时触发的 **serverless** 函数（一个由 Cron 触发的 Worker），这样就没有一台需要保持运行的机器。整个东西就是一个文件加一份调度计划。对一个个人小工具来说，这比听上去更要紧：任何有一台需要照看的服务器的东西，你迟早会忘了续费，于是这个项目死于荒废，而不是死于失败。

一个触发器扇出到两个数据源，再汇聚回一条消息。包住每个数据源的 try/catch，让单个失败优雅降级，而不是杀掉整次运行：



TELEGRAM BOT

" 今天有 3 个日程...最高 24C"

只要有一个数据源失败 → 把已有内容发出去，并标注缺失的部分

关键的 *prompt*

```
Write a Cloudflare Worker triggered by cron at 7am Asia/Seoul.
It calls a weather API and a calendar API (I'll provide both
tokens as secrets), builds a one-line summary, and POSTs it to
the Telegram Bot API. Wrap each external call in try/catch so one
failure doesn't kill the message. Read every token from env.
```

当我们想要打磨措辞，又不想等到早上 7 点时：

```
Add a manual trigger: if the Worker is hit with a GET request and
a ?test=1 query param, run the same logic immediately and return
the message text in the response instead of sending it. Keep this
behind a secret token so only I can trigger it.
```

这个测试触发器把我们从残酷的反馈循环里救了出来——我们可以随时按需看到输出，而不必每天只看一次。这是个值得偷师的通用招式：每当你的代码按计划运行时，在你认命去等时钟之前，先给自己造一个能现在就按需跑它的法子。

障碍

Telegram 消息到了，但时间是错的：事件显示的是 UTC，而不是韩国时间。天气倒是没问题。我们把它隔离了出来：

```
Calendar times are 9 hours off — showing UTC, not Asia/Seoul.
The weather time ("rain after 4pm") is correct. Here's how I
format both: [pasted]. Fix only the calendar formatting and tell
me why the weather path was already right.
```

AI 解释说，天气 API (Application Programming Interface, 应用程序编程接口) 已经返回了本地化的字符串，而日历 API 返回的是我们直接原样打印出来的 UTC 时间戳。它在日历格式化那一步加上了一次时区转换。把修复范围限定在“只改日历路径”，阻止了它去重写那段本来就工作正常的天气代码——这是 AI 的修复弄坏本来好端端的东西的一种常见方式。那个 prompt 的后半句——“tell me why the weather path was already right”——一身二任：它给了我们真正的根因而不是一个猜测，并且逼着 AI 把这个差别讲清楚，好让它的修复不至于把两条路径意外“统一”成一条坏的。

上线

这里的“上线”只不过意味着部署 Worker、设置 cron 调度计划，以及把 API token 作为加密密钥加进去。我们点了几次？test=1 端点以确认消息读起来不错，然后就放着不管了。从那以后它每天早上都在运行——而那条优雅降级的话，在一个月后天气 API 出故障时证明了它的价值：消息照样到了，只是少了天气预报而已。

案例研究 4：一个不用你打理的内容站

想法

一位业余爱好者想把一个 Markdown 笔记文件夹——多年旅行积攒下来的餐厅点评——变成一个可搜索的公开站点，没有要登录的 CMS，也没有月度账单。加一条笔记，推送，搞定。

Spec

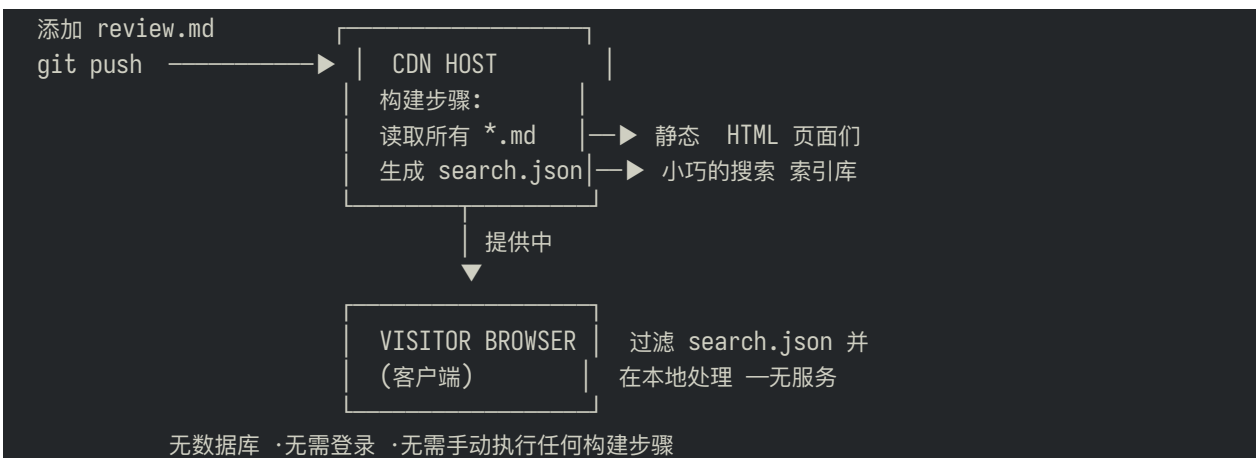
```
A static site generated from a folder of Markdown files, one file per review (title, city, rating, body in frontmatter). Build a homepage that lists reviews newest-first, a page per review, and client-side search by city or name. No database, no login, no build step I have to run by hand. Pushing a new .md file should publish it.
```

隐藏的需求在最后一句里。“No build step I have to run by hand”意味着这段 spec 其实讲的是工作流，而不只是产物——而工作流需求正是那种你不点名 AI 就会跳过的类别，因为它们不会在跑起来的应用里露面。

技术栈

一个在构建时读取 Markdown 的静态站点生成器，部署到一个带 Git 触发构建的 CDN 托管平台上。推送到仓库，托管方重新构建，CDN 来分发。搜索在构建时生成的一个小 JSON 索引上做客户端搜索，所以没有服务器、也没有查询成本——在几百条点评这个量级上这是个不错的取舍，而且是我们点明了的、而非撞上的一个有意选择。

整个工作流就是一根箭头：一次推送触发一次会发布的构建。除了访客自己的浏览器过滤索引之外，请求时刻没有任何东西在运行：



关键的 prompt

我们先把 AI 锚定在数据形状上，在任何 UI 之前：

```
Set up a static site that reads every .md file in /reviews. Each file's frontmatter has title, city, rating (1-5), date. Parse all of them at build time into a sorted list (newest first) and generate: a homepage listing them, and one page per review at /reviews/<slug>. Show me the data-loading code first, before any styling.
```

“before any styling”是一根有意为之的杠杆。一次性把整个东西要过来，你会得到一个连着你信不过的数据、却很漂亮的页面；先把数据层要过来，你就能在哪怕一个像素分散你注意之前，先验证好地基。然后是搜索，被框住以保持廉价：

```
Generate a search.json at build time with {title, city, slug} for every review. On the homepage, add a search box that filters the visible list client-side by matching city or title — no network calls, just filter the already-loaded index. Keep it under 50 lines of JS.
```

障碍

构建在本地通过了，但部署后的站点在每一个单独的点评页上都 404。首页能用；按点评分的页面不行。我们没去做推理，而是把症状和配置粘了上去：

```
Homepage works live, but /reviews/<slug> pages all 404 in
production while working in local dev. Here's my build output
directory listing and my host's deploy config: [pasted]. Why does
local dev serve these pages but production doesn't?
```

```
The AI traced it to the difference between dev-server routing (which
resolves paths dynamically) and static hosting (which serves only
files that physically exist). The build was generating the review
pages into the wrong output folder, so they never got uploaded —
dev had hidden the bug because its router faked the routes that
production couldn't.
```

我们把输出目录指向了托管方真正部署的那个，页面就出现了，并且我们在上线检查清单里加了一行：在线上站点点一个深层链接，而不只是首页。更广的教训是，“在 dev 里能用”和“部署后能用”是两个不同的主张，而静态导出恰恰就是这两者分道扬镳的地方——dev 服务器在路由上的宽容，是真正的基于文件的托管永远不会有有的。

上线

我们推送了一条真实的点评，看着托管方重新构建，然后在线上域名加载了通向那一条具体点评的深层链接——也就是 dev 撒过谎的那个案例。然后我们搜了一下它的城市，确认索引把它收进来了。加下一条点评是一次一行的提交，而这正是全部意义所在：上线不是一个时刻，而是一个不用我们就能继续运转的工作流。

这四个项目的共同点

- **Spec 先行，而且很短。**一段话，而不是一份文档。但它点明了那些重要的约束——语言、“只能看到自己的数据”、“先把你有的发出来”、“推送一个文件就该发布它”——包括那些永远不会在跑起来的应用里露面的工作流约束。
- **密钥从未碰过仓库。**每个项目都从环境变量或加密密钥里读取 key，这些都设在控制面板里，而不在代码中。我们把“never hardcode it”大声说了出来，因为默认恰恰相反。
- **技术栈是按“能写错的更少”来选的。**托管认证、托管数据库、静态托管、serverless 函数——每一个选择都拿掉了一类 AI 能替我们弄坏的代码，以及一类我们得去维护的东西。
- **障碍是靠喂给 AI 确切的证据来解决的——**字面的报错、实际的代码、部署配置、真实的症状——并且通过把修复范围限定得很窄，让能工作的代码保持能工作。有两次，症状在代码里，病因却在基础设施里。
- **上线意味着一次对现实的真实测试：**一次真实的购买（以及一次被拒的）、第二个用户账户、一次按需触发、线上站点的一个深层链接。而不是“它编译通过了”。

你不需要一个宏大的想法才能开始。你需要的是一个小想法、一段话的 spec，以及指挥而非敲键盘的意愿——以及在你称它做完之前，拿现实去检验它的那份耐心。

回顾与练习

关键点

- 规格先行，而且很短——一段话点明了真正重要的约束，包括那些在运行的应用里永远看不到的工作流约束。
- 密钥从未碰过仓库；每个项目都从环境变量或在仪表盘里设置的加密配置中读取密钥。
- 技术栈是为了少出错而选的一托管认证、托管数据库、静态托管、无服务器函数。
- 障碍是靠给 AI 喂确切证据来解决的一真实的报错、实际的代码、部署配置——并把修复范围收得很窄。
- 上线意味着对现实的真实测试，而不是“它编译通过了”：一笔真实购买、第二个账户、线上站点上的一个深层链接。

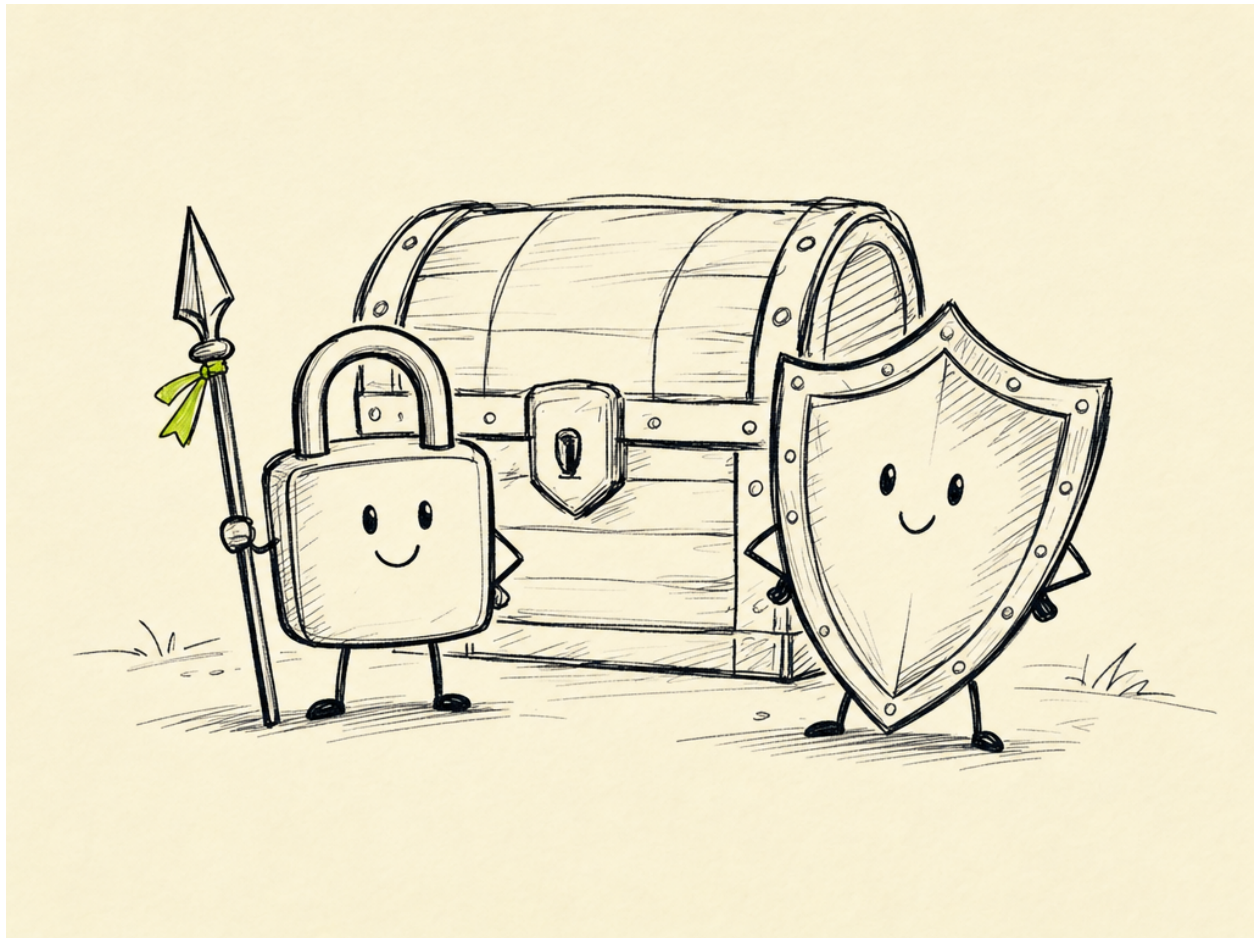
动手试试

挑一个你自己的点子，像这四个案例那样写出它的一段式规格：点明核心动作、每个用户能看到的数据，以及至少一个工作流约束（内容如何发布、谁能做什么）。然后列出你会依赖的托管构件，以减少会出问题的自定义代码。控制在一段话加一个简短的项目符号列表—如果超出了，就说明你规格定得过头了。

本章提示词

```
I want to build: [one-paragraph description of the app].  
Before any code, help me tighten this into a buildable spec.  
1. Restate the single core action a user performs.  
2. List the data each user can see — and what they must NOT see.  
3. Name the workflow constraints that won't show up in the UI  
   (e.g. "pushing a file should publish it", "only their own data").  
4. Recommend a stack that MINIMIZES custom code: managed auth,  
   managed database, static or serverless hosting where possible.  
5. Flag the one part most likely to go wrong, and how we'll test it  
   against reality (a real purchase, a second account, a deep link).  
Keep the spec to a paragraph plus bullets. Push back if it's too big.
```

写给 *Vibe Coder* 的安全课



安全是 **vibe coding** 里那种会悄悄反咬你一口的部分。一个功能坏了，问题立刻就暴露——按钮没反应，你随手就修了。可一个安全漏洞却几乎从不暴露，直到有一天有人发现了它，把你用户的数据扔到论坛上。到那时候，写出这个漏洞的 AI 早就不知道去哪了，而要面对一千个人、解释他们的密码为什么泄露的，是你。

这一章不是要吓唬你。你不需要变成渗透测试专家，也不用背下 OWASP Top 10，才能上线一个安全的小应用。你需要的是理解软件被攻破的那么几种常见方式，认出 AI 反复制造的那些模式，并且在线前加上一道审查关卡。这背后的理念和本书其余部分一样：**永远不要上线你自己不理解的东西**——而安全正是“不理解”代价最大的那个地方。

为什么 AI 生成的代码默认就是不安全的

AI 是在为你给它的提示词做优化，而你的提示词几乎总是说“让它能跑起来”，从不说“让它安全”。于是它产出的是通往一个能用功能的最短路径。而最短路径通常就是不安全的那条：用字符串拼接出来的查询、一个没有任何权限检查的接口 (endpoint, 端点——通往你应用、可被访问的那些门之一，也就是它用来应答请求的一个具体 URL)、一个直接内联粘贴进去的密钥——因为它学到的示例就是这么写的。

更糟的是，AI 是从整个互联网学来的——包括十年里那些本身就不安全的教程和废弃仓库。在它的训练数据里，不安全的代码比安全的代码更常见，因为安全的代码更难写、也更少有人发出来。当你让它写“一个登录表单”时，从统计上看你很可能拿到互联网上那个平均水平的登录表单，而互联网上那个平均水平的登录表单是有问题的。

AI 也完全没有威胁意识。一个人在做文件上传功能时会闪过一丝担忧——要是有人上传点恶心的东西怎么办？AI 什么都感觉不到。无论这段处理逻辑是滴水不漏还是门户大开，它都用同样的自信把它写出来。没有迟疑，没有那句“嗯，这部分让我有点不放心”。这份不放心，得你自己来提供。

要点不是“AI 代码很危险，别用它”。要点是“它能跑”和“它安全”是两个不同的问题，而 AI 只会回答第一个，除非你逼它回答第二个。

注入：当输入变成了代码

注入是最古老、至今仍最常见的严重漏洞。它发生在用户提供的数据被当成指令而不是数据来处理的时候。对大多数应用来说，有两种类型值得关注。

SQL injection (SQL 注入) ——SQL (Structured Query Language, 结构化查询语言) 是应用用来向数据库索要数据的语言——是用户输入混进了数据库查询。经典的脆弱写法：

```
// VULNERABLE: 用户的输入被直接粘进了查询里
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query(`SELECT * FROM users WHERE name = '${name}'`);
});
// 如果有人传入 name = '; DROP TABLE users; --
// 你的查询就变成了一条删表的命令。
```

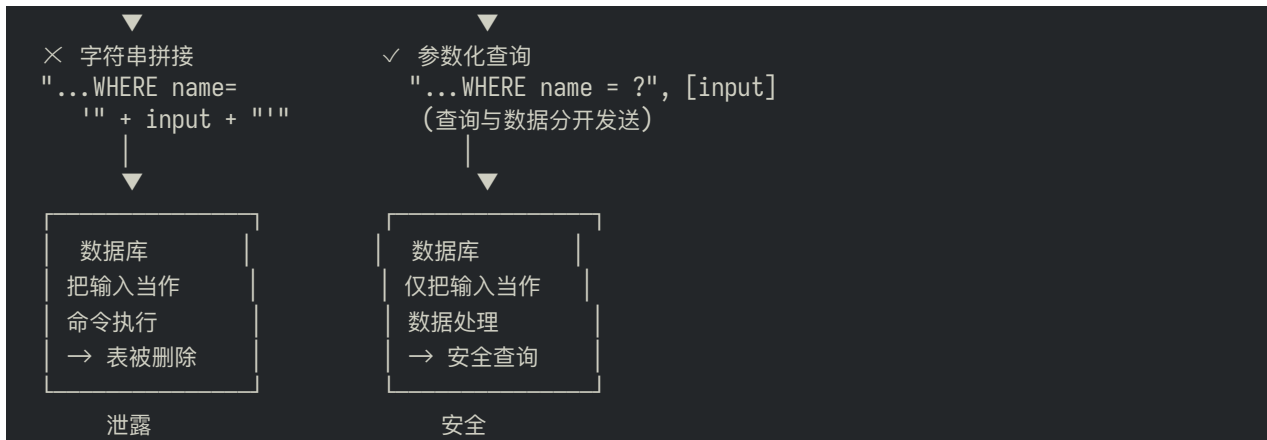
修复办法是用**参数化查询**（也叫预编译语句）。你把查询和数据分开发送，这样数据库就永远不会把其中一个错当成另一个：

```
// SAFE: 值作为参数传入，永远不作为代码
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query("SELECT * FROM users WHERE name = ?", [name]);
});
```

规则是：**永远不要用字符串拼接来构造查询**。如果你在 diff 里看到反引号或 + 把 SQL 拼在某个变量周围，停下来，要求改成参数化的版本。

两条路都把同一份用户输入送进数据库，但只有一条能让数据库不把那份输入误当成命令：

```
用户输入:      '; DROP TABLE users; --
```



XSS (cross-site scripting, 跨站脚本) 是同一个思路在浏览器里的版本。如果你拿用户输入、把它当作原始 HTML (HyperText Markup Language, 超文本标记语言, 告诉浏览器该显示什么的代码) 塞进页面, 攻击者就能注入一个 `<script>` 标签, 让它在其他用户的浏览器里运行——比如窃取他们的会话。修复办法是转义: 把用户内容当作文本来渲染, 而不是 HTML。像 React 这样的现代框架默认会转义, 这很棒——直到 AI 为了“让它能跑”伸手去用 `dangerouslySetInnerHTML` 或 `innerHTML` 这种逃生口。这些会绕过保护。在 diff 里看到任何这类调用, 都要当成需要质疑的东西。

每个注入漏洞背后统一的原则是: **让数据和代码保持分离**。每当用户输入要进入一个查询、一条命令、一个模板或 HTML 时, 总得有东西去转义它或参数化它。

密钥与 API key: 密钥外泄陷阱

密钥是任何能授予访问权限的东西: API (Application Programming Interface, 应用程序编程接口) key、数据库密码、支付服务商令牌、签名密钥。vibe coding 里最常见——也最昂贵——的错误, 就是泄露其中之一。

有两个陷阱不断地坑住人:

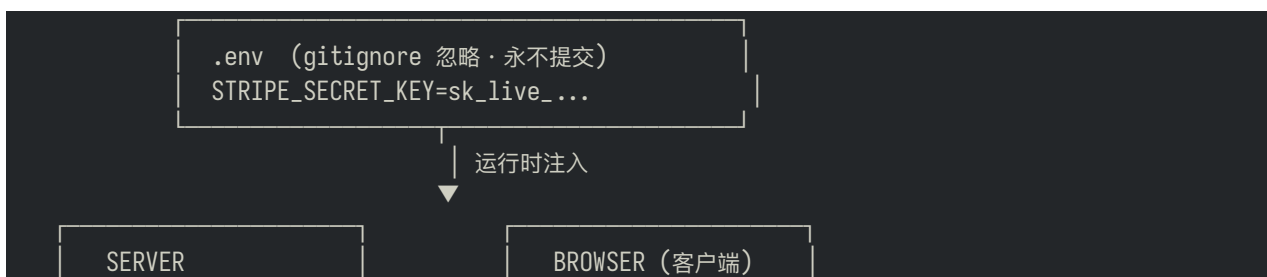
- **客户端代码里的密钥。**你前端里的任何东西 (在浏览器里运行的那段 JavaScript) 都是公开的。用户可以打开开发者工具直接读到它。AI 在被要求从 React 组件里调用一个 API 时, 会乐呵呵地把你的密钥直接粘在那儿——于是现在任何访问你网站的人都能把它复制走、把你的账单刷爆。密钥应该待在服务器上, 永远不要出现在会发送到浏览器的代码里。
- **仓库里的密钥。**硬编码在文件里的密钥会被提交进 git。就算你之后删掉它, 它也会永远留在 git 历史里, 而机器人会在你 push 后几分钟内就扫描公开仓库, 找的正是这种东西。

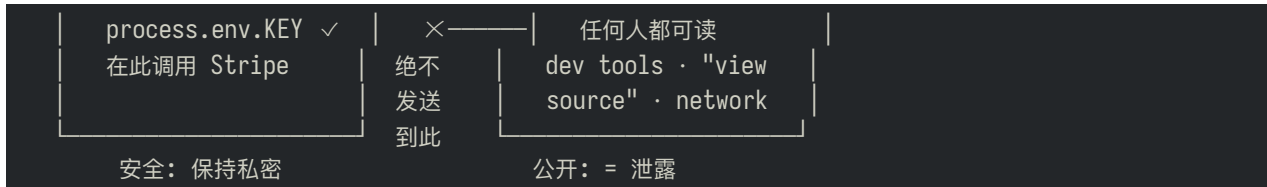
密钥正确的归宿是 **环境变量 (environment variable)** ——一个在运行时提供给应用的值, 完全不写进代码里:

```
// VULNERABLE: key 在源码里, 会被提交进 git
const stripe = new Stripe("sk_live_51H8xQ2eZvK...");

// SAFE: key 从环境中读取, 从不写在代码里
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
```

密钥只有一个安全的家——在服务器端, 从环境里读取。浏览器是公共场所: 任何被送到那里的东西, 任何打开开发者工具的人都能读到。





在你写下第一个密钥之前，就把 `.env`（以及 `.env.local` 等等）加进你的 `.gitignore`。如果某个密钥确实进了你的代码或历史，就把它当成已经烧掉了：轮换它（生成一个新的并吊销旧的）——删掉那一行是不够的，因为旧密钥依然有效，而且依然在外面流传着。

认证 vs 授权：那个谁都能调的接口

这两个词听起来很像，而二者的区别正是真正的数据泄露藏身之处。

- **认证 (Authentication)** 回答的是你是谁？——登录、证明身份。
- **授权 (Authorization)** 回答的是你被允许做什么？——这个已登录的用户是否可以对这份数据执行这个操作。

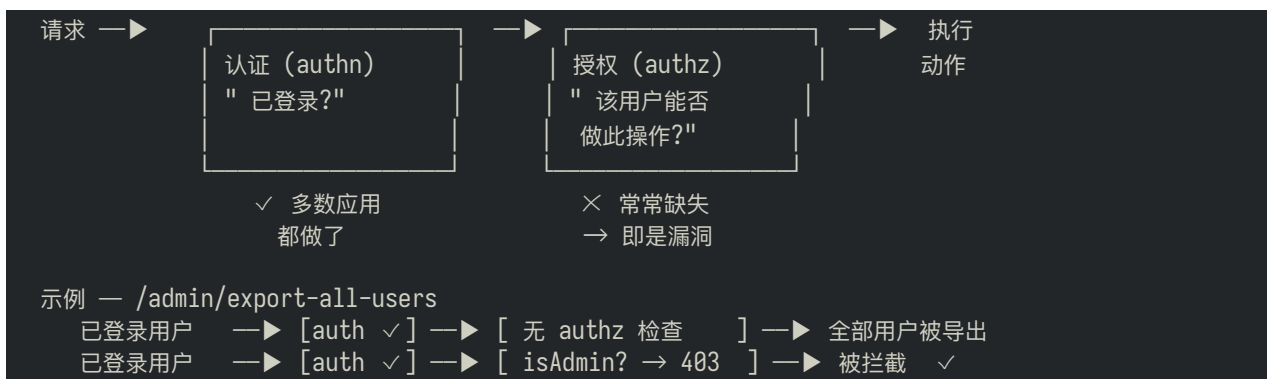
AI 在认证上还算不错；库帮它处理了大部分。授权才是它经常翻车的地方，因为授权是你应用规则所特有的，而 AI 并不知道这些规则。教科书级的灾难：

```
// VULNERABLE：检查了你是否登录，却没检查你是谁
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  res.json(db.getAllUsers()); // 任何已登录用户都能打这个接口
});
```

那个接口是“受保护的”——你必须登录。但任何已登录的用户，包括一个三十秒前才注册的，都能调用它、下载到每一个用户的数据。认证有了，授权没有。修复办法是在操作本身上检查权限：

```
// SAFE：确认这个用户确实是管理员
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  if (!req.user.isAdmin) return res.status(403).send("Forbidden");
  res.json(db.getAllUsers());
});
```

把一个请求必须通过的两道门想象出来。认证问“你登录了吗？”；授权问“是你被允许做这件事吗？”。当一个应用建了第一道门、却忘了第二道门时，入侵就发生了：



同样的陷阱以缩小版的形式到处出现：一个接口返回订单 #1234，却不检查这个订单是不是属于发起请求的那个用户。任何人都能改一改 URL 里的数字，读到别人的订单。规则一点也不光鲜，但绝对：**在每一个接触数据的接口上检查授权，并且不要信任来自客户端的 ID**。不要因为一个 URL 很隐蔽就假定它安全——“没人知道这玩意儿存在”不是一种安全控制手段。

文件处理与上传

让用户上传文件是一片悄无声息的雷区，而 AI 的初稿几乎从来不会堵上那些洞：

- **校验类型和大小。**没有限制，就会有人上传一个 10 GB 的文件或一个可执行程序。检查实际的内容，而不只是文件名后缀——后缀谁都能改。
- **永远不要信任文件名。**一个像 `../../etc/passwd` 这样的文件名，可能让一个天真的处理逻辑把文件写到目标文件夹之外（这叫 `path traversal`，路径穿越）。自己生成安全的文件名，而不是用对方给的那个。
- **不要从和你应用相同的源（*origin*）上提供上传的文件**，也不要让上传的文件被当作代码执行。一个用户提供的文件落进了你服务器会去运行的文件夹，那就是一条直通被接管的路。

这些你不必全都自己造——一个存储服务能帮你搞定大部分——但你得主动去要，因为 AI 不会主动给你。“加一个上传功能”给你的是理想路径。“加一个上传功能，要校验文件类型和大小、除了图片之外一律拒绝、并用生成的文件名来存储”给你的，才是更接近安全的东西。

依赖风险：AI 凭空发明的那个包

现代应用依赖着几十个第三方包，而 AI 推荐起它们来毫不手软。两个风险随之而来。

第一，**typosquatting**（仿冒抢注）和幻觉出来的包。攻击者会发布恶意的包，名字和真包只差一丁点（`requeusts` 而不是 `requests`），赌的就是你打错字。AI 有时会信心满满地导入一个根本不存在的包——而一个注意到这种习惯的攻击者，就能去注册那个一模一样的名字、里面塞上恶意代码。在安装 AI 推荐的任何东西之前，瞄一眼它：它真的存在吗，它有真实的下载量和仓库吗，名字的拼写是你预期的那样吗？

第二，**未经审查的依赖（总体而言）**。你加进来的每一个包，都是以你应用的全部权限在运行的代码。依赖越多，意味着 `bug` 和供应链攻击的暴露面也越大。优先选少数几个广为人知的库，而不是一长串名不见经传的；并在加一个包之前先问问“我们真的需要一个包来干这个吗，还是说这就十行代码？”

定期运行你所在生态的审计工具（`npm audit`、`pip-audit` 之类）——它会标出你已经装了的东西里那些已知的漏洞，而 AI 很擅长修复它报出来的问题。

安全审查关卡

下面这一个习惯，能把上面所有这些从一桩忧虑变成一道流程：在你上线之前，**让 AI 去攻击它自己的代码**。那个写出这个功能的模型，通常自己就能找出其中的漏洞——只是你不问，它就不找。把它从建造者翻转为对手：

你写了这个接口。现在扮演一个试图攻破它的攻击者。

列出一个恶意用户可以用来达成下列目的每一种方式：

- 读取或修改他们不该碰的数据（授权漏洞）
- 通过输入注入代码（SQL injection、XSS、命令注入）
- 滥用缺失的校验或速率限制

对每一种，给出能利用它的确切请求，然后给出修复办法。

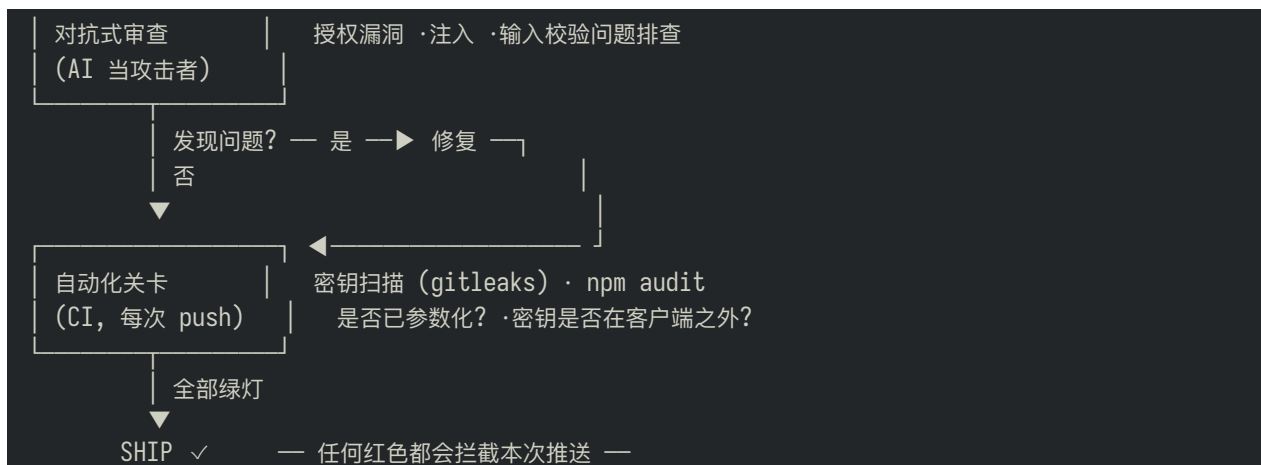
不要安慰我——假定这里就是有一个漏洞，把它找出来。

最后那一行很关键：放任不管的话，AI 倾向于说“看起来很安全！”。被要求假定缺陷存在时，它才会真的去找。把这一次对抗性审查，和一份你对任何面向用户的东西都跑一遍的简短上线前清单配套使用：

- ☐ 每个接口都检查了授权，而不只是用户已经登录
- ☐ 所有数据库查询都是参数化的——没有字符串拼出来的 SQL
- ☐ 渲染到页面上的用户输入都做了转义（没有原始 HTML 注入）
- ☐ 客户端代码里没有密钥，也没有任何密钥被提交进仓库
- ☐ `.env` 已被 `gitignore`；任何泄露过的密钥都已轮换
- ☐ 文件上传会校验类型和大小、并使用生成的文件名
- ☐ 新依赖都被亲眼核对过是否真实存在、口碑如何

把这道关卡想象成一个不通过就什么都发不出去的流程。AI 从建造者切换成攻击者的帽子，然后自动扫描兜住人工那一遍可能漏掉的东西：





并且在 push 之前跑一个**密钥扫描器**——像 gitLeaks 这样的工具（或者你平台内建的扫描）会在你的代码和历史里搜寻形似密钥的东西。它是一道一条命令就能就位的安全网，对应的正是这份清单上代价最昂贵的那个错误；你还可以让 AI 把它接进 CI（Continuous Integration，持续集成——那个在你每次推送代码时自动替你跑各项检查的机器人），这样每次 push 都会自动跑。

自动化自满，安全篇

最危险的时刻不是第一天——而是第九十天。第一天你很谨慎；你读每一份 diff，你跑那段攻击者提示词，你检查上传处理逻辑。然后一百次改动上线了，什么坏事都没发生，于是检查开始让人觉得像是走过场。你“就这一次”地在一个小改动上跳过了审查关卡。这个小改动加了一个没有授权检查的接口。有一阵子什么都没发生，因为从内部看，什么都没发生正是一个安全漏洞的样子。

这就是自动化自满对准了安全，而它在这里比在任何别的地方都更糟，因为安全失败是无声的、延迟的。一个坏掉的功能会在几分钟内惩罚你。一个坏掉的权限检查则会用表面上的成功来奖励你，一直奖励到泄露发生的那一刻。灾难的缺席并不是安全的证据——它正是一个有漏洞、只是还没被人发现的应用的正常状态。

防御之道，是把这道关卡做得足够便宜，让你不去跳过它；又足够自动，让你没法跳过它。把密钥扫描和 audit 放进 CI，这样它们不靠你记着也会运行；把那段攻击者提示词随手放在手边，用在任何接触 auth、数据或上传的东西上。守住一条线：一个处理用户数据或权限的改动，在有人——你，借助 AI 诚实的帮助——主动试图攻破它之前，不上线。vibe coding 里的速度来自好的关卡，而不是来自相信“安静”就意味着一切都好。

回顾与练习

关键点

- AI 生成的代码默认就是不安全的；假设漏洞存在并主动去找，而不要相信干净的第一印象。
- 大类反复出现：注入、外泄的密钥、缺失的授权、不安全的文件上传，以及凭空发明的依赖。
- 授权是按请求而非按登录——每个接口都必须检查这个用户是否可以执行这个操作，而不只是检查他是否已登录。
- 每次推送前都跑一次像 gitLeaks 这样的密钥扫描器，并轮换任何曾经泄露过的密钥；把扫描接入 CI，让你无法跳过。
- 第九十天比第一天更危险——把安全关卡做得便宜到你不会跳过、自动到你无法跳过。

动手试试

挑一个你自己涉及认证、用户数据或上传的功能，对它运行下面这条对抗式审查提示词。对 AI 报告的每个漏洞，索要利用它的确切请求和最小修复，应用修复后再次运行该提示词。最后对仓库和历史记录跑一遍 gitLeaks(或你平台内置的密钥扫描)收尾。

本章提示词

Here is the code for a feature that handles [auth / user data / uploads]:
[paste the code].

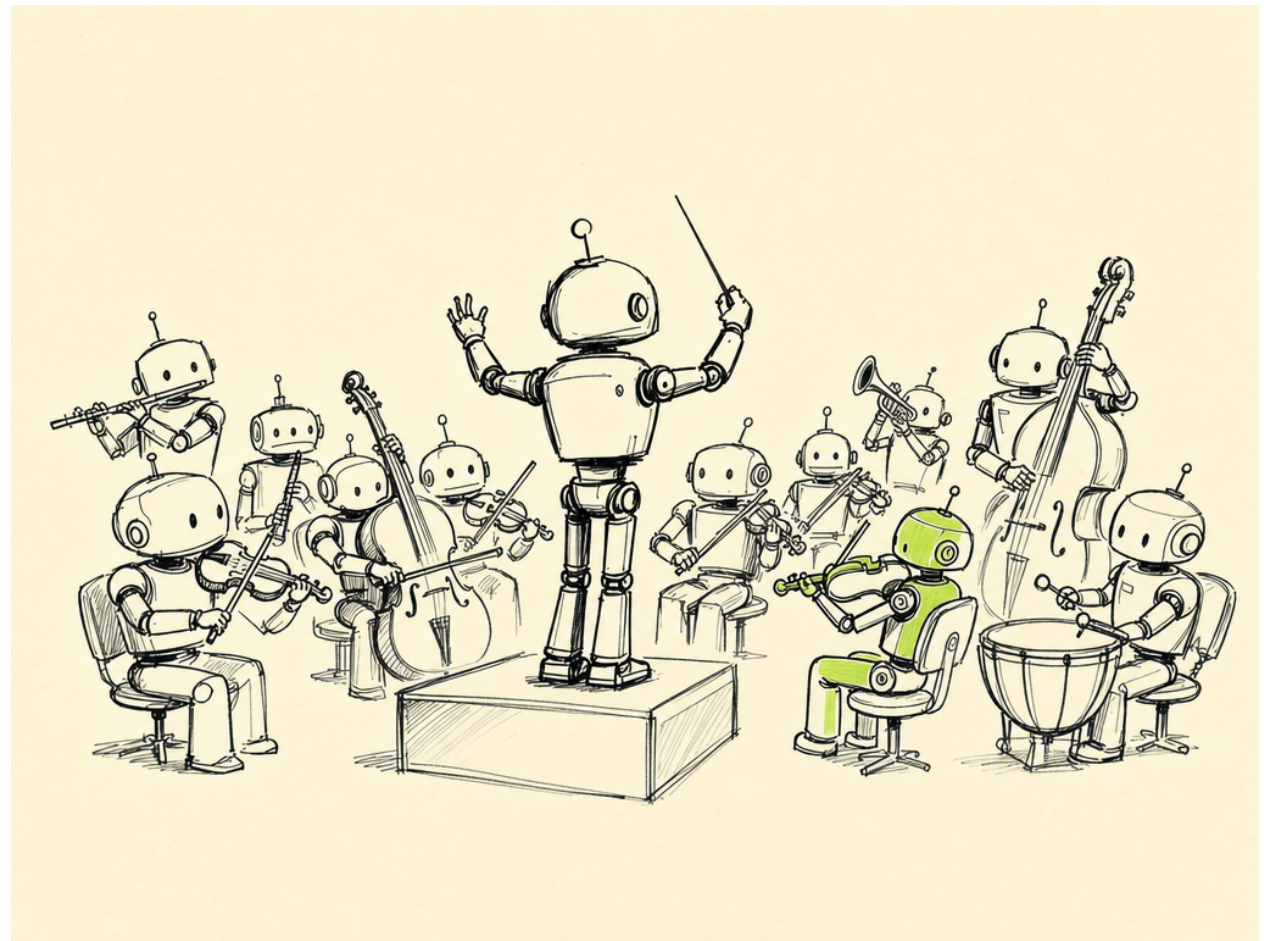
Act as an attacker reviewing this for security holes. Specifically check:

- broken authorization (can a user act on another user's data?)
- injection via input (SQL injection, XSS, command injection)
- exposed secrets or keys in code or responses
- unsafe file handling (type, size, path, generated names)
- missing validation or rate limits

For EACH issue, show the exact request that exploits it, then the fix.

Do NOT reassure me — assume there IS a vulnerability and find it.

智能体编排



本书的大部分内容里，AI 一直是你握在手中的工具：你问，它答，你检查结果。多数时候这仍是正确的心智模型。但工具长出了第二种模式——你交出的是一个目标而非一个步骤，AI 会工作几分钟（有时更久）才回到你这里。

本章讲的就是这种转变，以及当它发生时如何不丢掉主线。人们容易把「智能体」读成「现在我可以不再留意了」。正是这种读法会让人陷入麻烦。智能体是对你已经理解的工作的杠杆——而不是一张允许你发布自己已不再过目的代码的许可证。

三个层级

把 AI 辅助看成三个层级会有帮助，每一层都多交给你一点缰绳。

- **编辑器内补全。**你打字时，AI 提示下一行或下一块。你完全掌控；逐条接受或拒绝每个建议。工作单位是几行，而且它们就出现在光标正下方，所以默认你会全部审查。
- **仓库级助手。**你与一个能看到整个项目、一次编辑多个文件、运行命令并读取输出的助手对话。本书的大部分工作都在这里发生。工作单位是一个步骤或一个小功能，每一轮之后你审查 diff。
- **自主智能体。**你给出一个目标——「让测试套件通过」「把这个模块迁移到新 API (Application Programming Interface, 应用程序编程接口, 一个软件对外暴露、供其他软件使用的一组命令)」——然后智能体会规划、编辑、运行、读取错误，并跨越许多步骤反复尝试。你审查的是结果，而非每一个单独动作。工作单位是整个任务。

每一层都递给你更多的绳子：更大的工作单元、更少的连续操纵，以及更靠后的审查时点：

层级	工作单元	你审查	控制
补全 ▼	几行	每条建议	 紧
仓库助手 ▼	一步/功能	差异/回合	
自主 智能体	整个任务	那个结果	 松

更多缰绳 ▼ — 持续操控 → 放开，检查结果

要紧的跃迁是最后一个。前两层里你持续地操纵。在自主智能体这里，你放开它，然后检查结果。那是一种不同的技能，而本章余下的部分讲的就是它。

何时放手运行，何时手动操纵

是否让智能体无人值守地运行，归结为三个问题——把它们当成你离开之前要过一遍的清单。

- **爆炸半径。**如果出错，会坏掉多少？对整个代码库做一遍格式化，爆炸半径很小。数据库迁移或改动你的认证逻辑，则很大。半径越大，越要手动操纵。
- **可逆性。**能否廉价地撤销？在工作树干净的新建 git 分支上的工作是可逆的——git restore . 它就没了。删除数据、发送邮件、扣卡、推送到生产环境的工作则不是。绝不要让智能体无人值守地处理你拿不回来的东西。
- **可验证性。**你能判断结果是否正确吗？如果「完成」意味着一套绿色的测试或一个可见地正常工作的页面，你几秒内就能验证，智能体非常合适。如果「完成」取决于微妙的判断——金额计算、安全规则、竞态条件——你无法靠一次快速检查来设卡，那就贴近些。

把它们合起来的简单办法：**当工作可逆且结果廉价可验证时放手运行；当不是时则手动操纵。**把构建功能那章的规则带过来——爆炸半径越大，步骤越小——再加上它的孪生规则：爆炸半径越大，缰绳越短。

给任务划定范围，让它无法乱跑

自主智能体以一种特定的方式失败：它漂移。你要一件事，它「热心地」重构了另外三件，碰了你从未提及的文件，带着一份你读不懂的 diff 回来。修法不是更聪明的智能体——而是更紧的简报。

一个好的自主任务有四个部分：清晰的目标、明确的约束、具体的完成判据，以及智能体自己能运行的护栏。护栏是最重要的部分，因为它是智能体无法在你不察觉的情况下翻越的栅栏。

Goal: Make the failing tests in tests/checkout/ pass.

Constraints:

- Only edit files under src/checkout/. Do not touch src/auth/, the database schema, or anything in config/.
- Do not add new dependencies.

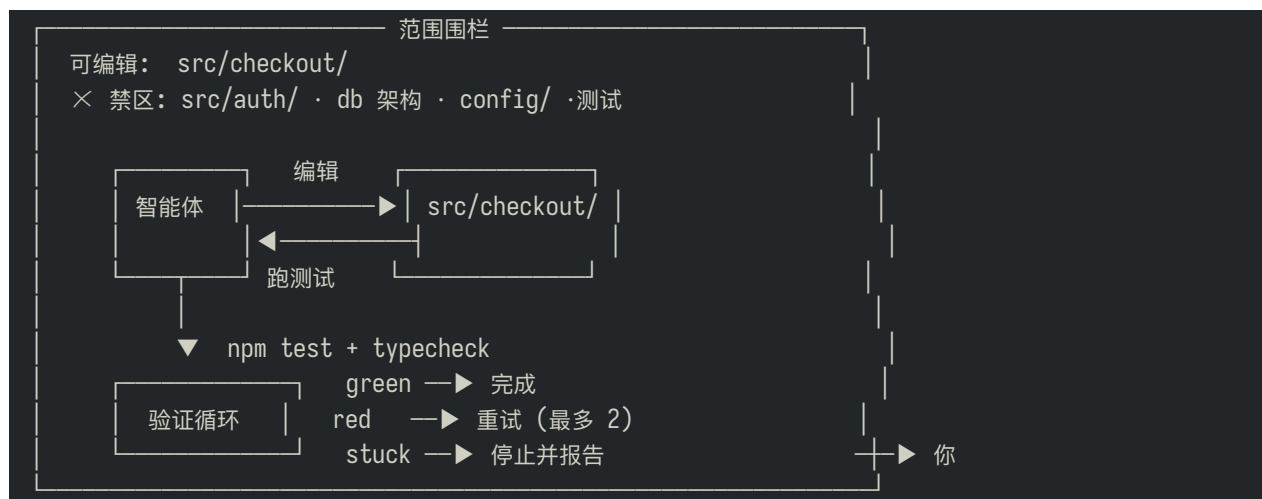
- Do not change the tests themselves — the tests define correct behavior.

Done when:

- ``npm test tests/checkout/`` is fully green.
- ``npm run typecheck`` passes.
- ``git diff --stat`` shows changes only under `src/checkout/`.

If you get stuck after two real attempts, stop and report what you tried and what's still failing. Don't paper over a failure to make the suite go green.

这份简报之所以管用，是因为它把智能体关进了盒子里：约束给它能碰的东西筑墙，完成判据是它能运行的一条命令，而验证循环让它保持诚实：



三样东西让这份简报难以让它跑偏。约束在智能体可以触碰的文件周围筑起一堵墙。完成判据是任何人都能运行的命令，而不是一种感觉。而最后那段给了它停下来发问而非乱扑腾的许可——正是这一点防止了那种慢镜头的灾难：智能体「修」了二十分钟，把代码树留得比它接手时还糟。

真正的栅栏是测试。能运行自己测试的智能体拥有一个紧凑而诚实的反馈回路，多半会留在其中；无法检查自己工作的智能体会信心十足地把坏掉的东西递给你。如果你从本章只带走一个习惯：给智能体一个验证自己的途径，并把通过它当作完成的定义。

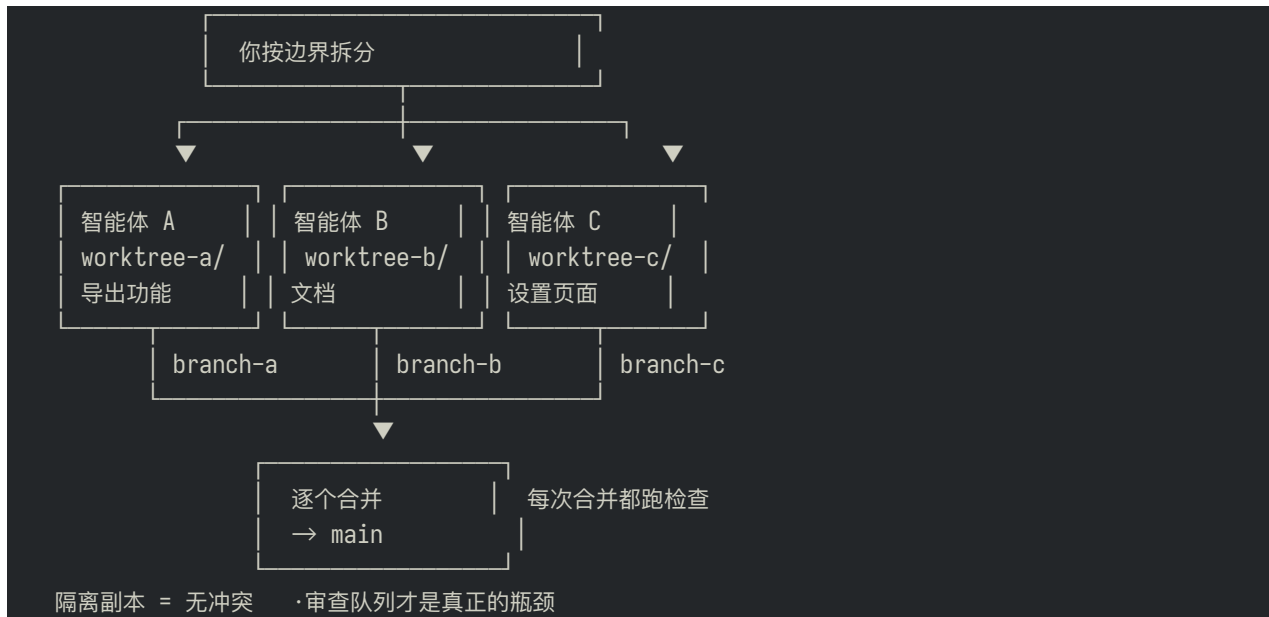
后台与并行智能体

一旦你在划定范围的任务上信任某个智能体，下一步就显而易见：一次运行好几个。现代工具让你能启动后台智能体——各自在自己的分支或工作树（*worktree*，项目的一份单独的工作副本，这样一个智能体的改动永远不会和另一个的相撞）上——这样在你做别的事时，三四个划定范围的任务都在推进。

这是真正的提速，但仅限于独立的工作。你拿来交换的代价是协调。两个智能体编辑同一个文件会产生冲突的 *diff*，手动合并它们可能比你省下的还贵。

- **按边界拆分，而非按行数。**给每个智能体一块与其他不重叠的部分——不同的模块、不同的功能、不同的文件。「智能体 A 做导出功能，智能体 B 写文档」很干净。「两个智能体都编辑 `app.js`」是预约了一场合并头疼。
- **隔离工作空间。**让每个智能体在自己的 *git* 分支或工作树上运行，使它们踩不到彼此未提交的改动。一次合并一个回来，每次合并时都运行你的检查。
- **诚实地控制数量。**你仍然必须审查每一个结果。四个智能体在跑，就是四份要读的 *diff* 和四套要信任的测试。超过一小撮之后，瓶颈就成了审查队列，而不是智能体。

每个智能体都在自己隔离的 *worktree* 里工作，于是它们的编辑永不碰撞；你一次合并回一个，并在每道关卡跑检查：



错误在于把并行当成免费。它不是——你把工作从编写挪到了审查与集成。这常常是笔很好的交易，但前提是你真的去审查。

多智能体：扇出、对手、综合

还有第二种「多个智能体」的口味，它运行在一个任务之内，而非跨越彼此独立的任务。不是让一个智能体做所有事，而是把角色拆开：

- **为求广度的扇出。**多个智能体从不同角度攻同一个问题——或探索一个大代码库的不同部分——然后回报。当你想要覆盖面时很有用：「找出我们调用这个已弃用函数的每一处」就很好并行。
- **对抗式审查者。**一个智能体写；第二个智能体唯一的工作是攻击第一个的输出——找出 bug、漏掉的边界情况、未处理的错误。一个被指向 diff、带着「假设这是错的，并查清为什么」指令的审查者，会逮住作者智能体把自己说服了的东西。
- **综合。**最后一遍收集这些发现，产出一个连贯的结果——可运行的代码，外加一段简短的说明：审查者标出了什么，以及它是如何被解决的。

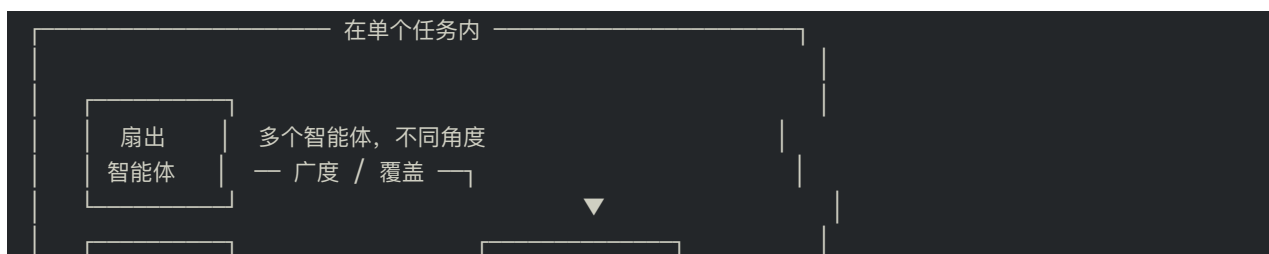
Reviewer agent brief:

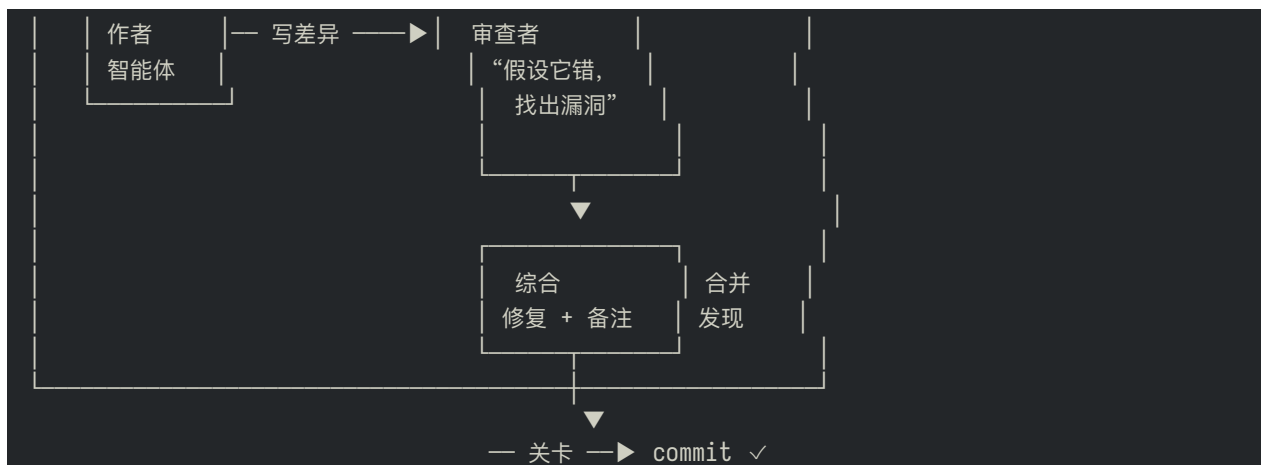
You are reviewing a diff written by another agent. Assume it contains at least one bug and find it. Specifically check:

- Does it actually satisfy the original goal, or just pass the tests?
- Edge cases: empty input, nulls, the largest realistic input.
- Anything deleted or changed that the task didn't ask for.

Report concrete problems with file and line. Do not rewrite the code — just report. If you find nothing after a genuine look, say so plainly.

三个角色构成一条末端带关卡的流水线：在对抗式审查者过完一遍、综合环节解决掉它的发现之前，什么都到不了提交：





对抗式审查者值得设置，即便你是没有一支智能体舰队的单枪匹马——因为一处改动的作者，无论人还是 AI，都是最不适合找出其缺陷的那个。用一段敌意的提示再过一遍，是廉价的保险。

按规模审查输出

这里有个诚实的问题：当智能体跨越多个任务写出数百行时，你无法像审查十行 diff 那样读完每一行。假装可以，会导致两种结果之一：盖橡皮图章（不看就批准）或瘫痪（什么都不信任）。两者都发布不出东西。

走通的办法是在几个层面设卡，而不是把一切都读完：

- **测试是底线。**相关测试不绿、类型检查不干净，任何智能体输出都不得合并。这不可商量，而且是自动的——这部分由机器来做。
- **对高风险部分做抽查。**你读不完全部，所以读爆炸半径高的部分：任何触及认证、金额、数据删除或外部调用的地方。其余的，按构建功能那章的危险信号略读——它不该碰的文件、被悄悄删掉的东西、一个小请求生出一份巨大 diff。
- **读 diff，而非整个代码库。**git diff --stat 几秒内就告诉你什么变了、变了多少。一个划定到一个模块的任务却在六个模块里显示改动，这是一个信号：在读任何一行之前，先更仔细地看。
- **运行它。**一套绿色的测试不是一个能用的应用。点一下那东西。最后的检查永远与其他每一章相同：它在你面前，是否真的做了你要它做的事？

划定范围在这里有双重回报。一个你紧紧划定范围的任务会产出一份你真能审查的 diff；一个你放开的任务会产出一份你不能审查的。审查问题大体在上游、在简报里就被解决了。

成本与失控的循环

自主智能体靠循环工作：尝试、检查、再尝试。那个循环正是全部要点所在——它同时也是那个会悄悄把账单拉高或永远转下去的东西。

这直接连到前面那个成本安全的观念：**定额订阅的工具以停止来失败；按量计费的付费 API 智能体以收费来失败。**一个订阅制的编程助手撞到上限时，它只是在窗口重置前不再多做。你针对一个付费 API 写的脚本卡在重试循环里时，它会一直调用——每次调用都花钱——直到你察觉或你的卡刷爆。

所以规则很直白：

- **绝不要针对付费 API 运行一个不加节流的智能体循环。**如果你写了自己的、会调用某个 LLM（Large Language Model，大语言模型——支撑着这些编程工具背后的那类 AI）API 的循环，它必须有一个硬上限——最大迭代次数、花费上限、超时。没有例外，没有「我会盯着的」。你不会盯。
- **给每个循环一个出口。**「N 次尝试后停下」「如果测试再失败两次就停下」「如果已经跑了十分钟就停下」。一个没有停止条件的智能体是 bug，不是功能。
- **盯住头几次运行。**在你信任一个智能体在后台运行之前，从头到尾盯几次前台运行。你是在检查它是否收敛——是越来越接近完成，而不是绕着同一个修法打转卡住。

- **优先选有护栏的工具。**一个带内置上限的托管助手，比你半夜写的一个赤裸的 API 循环更适合放着跑。订阅以关闭来失败是个功能；API 以敞开来失败是个隐患。

诚实的限度

智能体是本书中最强大的杠杆形式，正因如此，对它是什么、不是什么保持清醒的眼光才有价值。

它不是一种你可以停止理解的自主性。当你再也答不出「这段代码应该做什么，要是它错了我又怎么会知道」的那一刻，你就从使用杠杆越界到了指望运气。本章的每一层——划定范围、护栏、测试、抽查、成本上限——都是为了在让智能体承担远多得多的打字工作的同时，把你留在那条线上理解的一侧而存在。

技能的天花板与本书其余部分相同：你精确说出你想要什么以及认出你已得到它的能力。智能体把这个能力乘大。它不取代它。被烫伤的不是那些用了智能体的人——而是那些用智能体来停止思考的人。别做那种人。把工作交出去；把判断握在手里。

回顾与练习

关键点

- 智能体放大的是你精确表达需求、并识别是否如愿以偿的能力——而不是取代它。
- 界定范围有双重回报：范围收得紧的任务产出你真能审查的 diff，松散的任务产出你审不动的 diff。
- 审查 diff，而不是整个代码库—`git diff --stat` 几秒钟就能标出范围蔓延，而通过的测试套件永远不能替代亲自运行。
- 绝不要对付费 API 跑没有限流的智能体循环；给每个循环设硬上限——最大迭代次数、支出上限或超时。
- 在信任智能体后台运行之前，先在前台盯着头几次运行，确认它会收敛。

动手试试

挑一个小而边界清晰的任务，连同一个明确的停止条件（“尝试 3 次后停止”或“测试再失败两次就停止”）一起交给智能体。在前台从头到尾盯完整个运行，然后只用 `git diff --stat` 再加 `git diff` 审查这份 diff。如果改动触及的文件多于你界定的范围，那就是收紧任务说明并重跑的信号。

本章提示词

```
I'm giving you an autonomous task. Obey these guardrails exactly:
- SCOPE: only touch [list the files/modules]. If the fix needs more,
  STOP and tell me instead of expanding scope.
- GOAL: [one concrete, checkable outcome].
- STOP CONDITIONS: stop after 3 attempts, OR if tests fail twice in a
  row, OR if you've made changes outside the scope above.
- COST: do not call any paid API in a loop without a hard cap.
- REPORT: when you stop, show me `git diff --stat`, the test result,
  and one sentence on whether you converged or got stuck.
Begin, and narrate each attempt so I can watch you converge.
```