

Vibe Handbook

Vibe Handbook

目 錄

Git & GitHub	5
什麼是版本控制（以及為什麼你需要它）	6
倉庫、提交，以及歷史	6
分支：無所畏懼地嘗試	7
看清到底改了什麼（diff）	7
“糟糕”按鈕：undo、revert、reset	8
遠端與 GitHub：雲端裡的你的專案	8
拉取請求與議題	8
.gitignore 與絕不提交機密	9
你的第一個 repo，從頭到尾	9
讓 AI 來開車（而你仍掌舵）	9
回顧與練習	10
The Command Line	11
終端和 shell 究竟是什麼	11
讀懂提示符	12
每天都會用到的命令	12
執行一個程式和一個 dev 伺服器	13
讀懂輸出和 exit 碼	13
用 Ctrl-C 停掉一個程序	13
PATH 與“command not found”	13
安全地執行 AI 給你的命令	14
你會遇到的包管理器	14
回顧與練習	14
How the Web Works	16
兩個角色：客戶端與伺服器	16
URL 實際上是什麼	17
請求與響應的迴圈	17
HTTP 方法：動詞們	17
狀態碼：成功了嗎？	18
頭部與 Cookie	18
HTTPS 與那把鎖	18
從名字到伺服器：DNS	19
這一切如何對應到你的應用	19
回顧與練習	19
API、JSON 與資料	21
API 究竟是什麼	21
用大白話講端點和 REST	22
JSON 是什麼	22
你會遇到的資料型別	23

一個真實的請求和響應	23
API key 與認證 header	24
速率限制與賬單暴雷陷阱	24
怎麼讀 API 文件	24
讓 AI 去做呼叫——而你來檢查形狀	25
回顧與練習	25
一個應用的解剖	26
全域性圖景	26
frontend、backend、資料庫、託管	27
HTML、CSS 和 JavaScript	27
包和依賴	28
環境變數與配置	28
一個專案的樣子	29
一個請求是什麼流動的	29
回顧與練習	30
Vibe Coding 心態	31
Vibe coding 究竟是什麼	31
心態的轉變	32
迴圈：意圖 □ 生成 □ 審查 □ 打磨	32
它的閃光之處	33
它失效的地方	34
常見的坑	34
品味是新的核心技能	34
現實的預期	35
回顧與練習	35
搭建你的 AI 工作環境	36
選擇一個 AI 原生的編輯器或助手	36
把專案結構組織得讓 AI 能導航	37
寫一個上下文檔案 (AGENTS.md)	37
讓版本控制成為本能反應	38
讓反饋迴圈保持緊湊	39
謹慎管理機密	39
一個健全的本地配置讓 AI 的改動安全	39
即時診斷 (LSP)	40
擴充套件你的智慧體	40
macOS vs Windows	41
TUI vs GUI	41
護具工程 (Harness Engineering)	41
成本安全：CLI vs API	42
回顧與練習	43
像工程師一樣寫提示詞	44
在給出任務之前先給出上下文	44
指明輸入、輸出和邊界情況	45
以小步前進	45
在寫程式碼之前先要一份計劃	46
在 diff 上迭代，而不是重寫	46
提供示例	47
告訴 AI 不要做什麼	47
像管理資源一樣管理上下文	48
要避免的常見反模式	48
核心心態	48

回顧與練習	49
幾分鐘內從想法到規格	50
為什麼要規格，而不只是憑感覺	50
第一步：釐清問題	51
第二步：定義範圍與非目標	51
第三步：編寫使用者故事	52
第四步：起草一份輕量 PRD	52
第五步：勾勒資料模型	53
第六步：把規格拆成 <i>vibe</i> 大小的任務	54
讓規格保持鮮活	54
一個常見陷阱：規定的是解決方案，而非問題	55
回報	55
回顧與練習	55
選擇你的技術棧：語言一覽	56
JavaScript / TypeScript	57
Python	57
Go	58
Rust	58
關於其他語言的簡短一說	58
一覽對比	59
異類選擇的真實代價	59
按目標來選	60
回顧與練習	60
構建功能，一個 <i>vibe</i> 接一個 <i>vibe</i>	61
構建迴圈	62
“縱向切片”是什麼意思	62
一個實戰示例：“收藏”功能	63
不必看懂每一行也能讀 diff	64
何時放手讓 AI 跑，何時手動掌舵	65
讓應用始終保持可執行	65
小步提交	65
當一個切片太大時	66
要點帶走	66
回顧與練習	66
用 AI 除錯	67
核心心態	67
一份除錯清單	68
給 AI 喂正確的上下文	69
閱讀你看不懂的錯誤	70
假設、日誌和二分查詢	70
根本原因與表象	71
當 AI 開始兜圈子	71
在保留修復之前先理解它	71
要點	71
回顧與練習	72
伺服器基礎設施	73
程式碼可以待的五個地方	73
幾個行話術語，破譯一下	74
取捨對照表	74
何時選哪個	75

AI 如何串聯部署與配置	75
給單打獨鬥构建者的推薦預設路徑	76
“託管 (managed)”到底意味著什麼 (以及你仍然擁有什麼)	77
一份快速的上線前清單	77
要點	78
回顧與練習	78
資料庫基礎設施	79
資料儲存的三種形態	79
託管 vs 自託管	80
Schema 設計基礎	80
遷移：安全地更改 Schema	81
索引	82
快取	83
佇列	83
常見的初學者錯誤	83
指揮 AI 构建你的資料層	83
回顧與練習	84
防護欄、測試與審查	85
為什麼指揮 AI 時你需要防護欄	85
測試：你的安全網	86
型別檢查與程式碼檢查：免費抓出的錯誤	87
限定 AI 改動的範圍	87
帶著批判地閱讀 diff	88
審查關卡：合併前的一份清單	88
CI 基礎：讓檢查自動化	88
預設安全的習慣	89
當防護欄尖叫時：先分診，再修	89
回顧與練習	90
部署上線到生產環境	91
選一個託管平臺並部署	91
自定義域名與 DNS	93
環境變數與金鑰	93
接入支付	94
預釋出環境與回滾	95
監控與錯誤追蹤	95
效能與快取	96
上線前清單	96
上線之後	96
回顧與練習	96
案例研究：從 0 到上線	98
案例研究 1：一個多語言落地頁 + 支付頁	98
案例研究 2：一個帶認證 + 資料庫的小型 SaaS 工具	100
案例研究 3：一個個人自動化小工具	102
案例研究 4：一個不用你打理的內容站	103
這四個專案的共同點	105
回顧與練習	105
寫給 Vibe Coder 的安全課	106
為什麼 AI 生成的程式碼預設就是不安全的	106
注入：當輸入變成了程式碼	107
金鑰與 API key：金鑰外洩陷阱	108

認證 vs 授權：那個誰都能調的介面	108
檔案處理與上傳	109
依賴風險：AI 憑空發明的那個包	109
安全審查關卡	110
自動化自滿，安全篇	111
回顧與練習	111
智慧體編排	112
三個層級	112
何時放手執行，何時手動操縱	113
給任務劃定範圍，讓它無法亂跑	113
後臺與並行智慧體	114
多智慧體：扇出、對手、綜合	115
按規模審查輸出	116
成本與失控的迴圈	116
誠實的限度	116
回顧與練習	117

Git & GitHub



如果在開始 vibe coding 之前你只學一個工具，那就學這個。Git 是讓你敢於放手的工具。正因為有它，你才能對 AI 說”把整個檔案重寫一遍”還睡得著覺——因為你隨時都能回到那個能正常執行的版本。

實話實說是這樣的。當你指揮 AI 來構建軟體時，它一定會弄壞東西——而且是經常。它會信心十足地重寫一個檔案，刪掉本

來正常工作的部分，然後甩給你一團亂麻。一個糟糕的下午和損失一整週的區別，就在於你有沒有一張安全網。Git 就是那張網。本章從零教起，不需要任何程式設計基礎。

什麼是版本控制（以及為什麼你需要它）

想像一款有存檔點的電子遊戲。你通過一個棘手的關卡，存檔，然後嘗試一些冒險的操作。如果出了岔子，你讀取上一個存檔，而不是從頭開始整個遊戲。**版本控制就是你專案的存檔點。**

沒有版本控制，你的專案只有一個狀態：此刻它是什麼樣就是什麼樣。如果 AI 把程式碼弄壞了，那個好版本就沒了。有了版本控制，你保留著每一個存檔狀態的完整時間線，可以跳回其中任意一個。

Git 是最常用的版本控制工具。它在你的電腦上執行，每當你要求時，就悄悄記錄下專案的快照。它免費給你的幾樣東西：

- **一張安全網**——哪怕幾小時後，也能撤銷一次糟糕的改動。
- **一份歷史**——精確看到什麼變了、何時變的、為什麼變。
- **底氣**——盡情實驗，因為沒有任何東西會真正丟失。

最後這一點才是真正的獎賞。Vibe coding 在你快速行動、勇於嘗試時效果最好。Git 讓這種快速變得安全，而不是魯莽。

把專案的歷史想像成一排你隨時可以回去的存檔點：



倉庫、提交，以及歷史

三個你會不斷聽到的詞。我們把它們講清楚。

倉庫 (repository, 簡稱“repo”) 就是一個 Git 在盯著資料夾。你只需把一個普通資料夾變成 repo 一次，從那以後 Git 就會追蹤裡面的一切。

提交 (commit) 是一個存檔點。它是某一時刻所有檔案的快照，外加你寫的一句簡短說明，描述改了什麼（“加上登入頁面”“修好壞掉的註冊按鈕”）。你會不斷地做提交——把每一個都想成一個你可能想回去的檢查點。

歷史是所有提交按順序排成的列表。它就是時間線。你可以滾動瀏覽，讀每次改動是什麼，並穿越回任意一個時間點。

實際操作起來是這樣的。不用背——大多數時候你會讓 AI 替你執行——但跟著讀一遍，讓這些詞不再神秘：

```
# 把當前資料夾變成一個 Git 倉庫（只需做一次）
git init

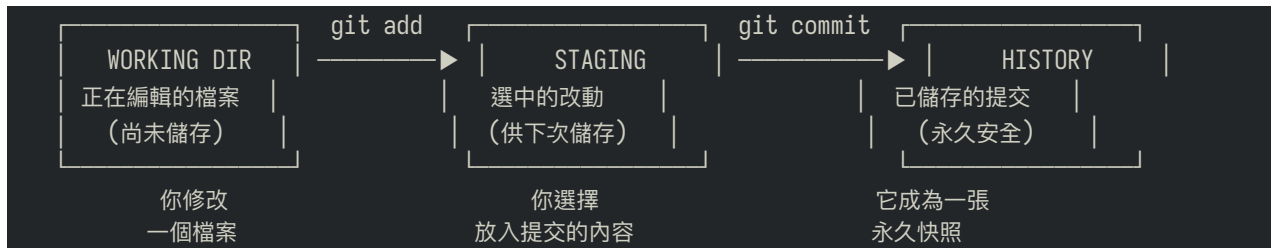
# 檢視自上一個存檔點以來哪些檔案變了
git status

# 暫存你的改動，然後帶一句說明儲存為一次提交
git add .
git commit -m "Add the homepage layout"

# 檢視你的歷史——每一次提交，最新的在前
git log --online
```

git add . 是你在說“把我所有的改動都納入下一個存檔點”。git commit 才是真正的儲存。-m "... " 是寫給未來自己的便條。寫一些一個月後你自己還能看懂的便條。

這兩條命令把你的工作推過三個階段。你編輯的檔案，只有一路走到最右邊，才算真正儲存：



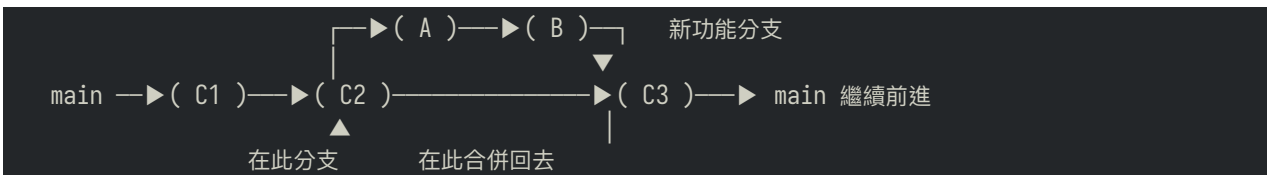
分支：無所畏懼地嘗試

分支（branch）是專案的一份平行副本，你可以在上面做實驗。你的主要工作通常在一個叫 `main` 的分支上。當你想嘗試點什麼時——一個新功能，或一次冒險的 AI 重寫——你新建一個分支，在那上面做，而你的 `main` 保持原封不動、安然無恙。

如果實驗成功，你把它合併回 `main`。如果失敗，你把分支丟掉，`main` 壓根不知道發生過這回事。

這正是你該用來對待重大 AI 改動的方式。“把整個支付流程做出來”配得上擁有自己的分支。AI 拿出好東西，就合併；拿出一場災難，就刪掉分支，你什麼都沒損失。

畫出來看，一個分支從 `main` 岔出去、自己生長，然後再合併回來：



```
# 新建一個分支並切換過去
git checkout -b new-feature

# ... 做你的工作，做幾次提交 ...

# 切回你那安全的 main 分支
git checkout main

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

看清到底改了什麼（diff）

在儲存一次改動之前——尤其是在接受 AI 做的改動之前——你會想看清究竟有什麼不同。這就是 **diff**。它一行一行地告訴你，什麼被加上了（通常是綠色），什麼被刪掉了（通常是紅色）。

這是你最重要的習慣之一。Vibe coding 的整套哲學就是理解你所交付的東西。你不必親手敲程式碼，但你必須看一眼改了什麼並大致理解它。diff 就是你做這件事的地方。

```
# 檢視你改了但還沒提交的内容
git diff

# 檢視已暫存、準備好提交的内容
git diff --staged
```

當 AI 說“搞定了，我更新了檔案”，你的下一步就是看 diff。如果你看到它刪掉了你需要的東西，你當下就能抓住——而不是在它交付之後。

“糟糕”按鈕: *undo*、*revert*、*reset*

正是這一部分，把 Git 從一樁苦差變成了一種超能力。當出了岔子時——它一定會——這樣把它倒回去。

- 丟棄未儲存的改動——你（或 AI）編輯了一些檔案、把事情弄得更糟，但你還沒提交。你可以把一切都退回到上一個好提交。
- 撤銷一次提交（*revert*）——你儲存了一次改動，然後發現它很糟。`git revert` 會做一個新的提交來抵消那個糟糕的提交，讓你的歷史保持誠實。
- 重置到更早的點（*reset*）——你想把專案倒回到好幾次提交之前的樣子。

```
# 撤銷所有未提交的改動，回到你的上一次提交
git restore .

# 安全地撤銷某一次糟糕的提交（會建立一個新的“撤銷”提交）
git revert <commit-id>

# 倒回到更早的提交（很強大——使用前請讓 AI 解釋清楚）
git reset --hard <commit-id>
```

提醒一句: `git reset --hard` 會永久丟棄工作。它有時是正確的工具，但是要小心對待的那一個。拿不準時，在執行前問問 AI”這條命令對我的檔案到底會做什麼?”。

遠端與 *GitHub*: 雲端裡的你的專案

到目前為止的一切都發生在你自己的電腦上。*GitHub* 是一個把你的 repo 副本存到雲端的網站。那份副本叫做遠端 (remote)。

何必如此？三個理由：它是你筆記本報廢時的備份，是你與別人（或 AI 工具和部署服務）共享程式碼的方式，也是協作發生的地方。

你把本地 repo 與遠端關聯一次，然後做 *push*（把你的提交送上 GitHub）和 *pull*（把 GitHub 上的提交取下來）。

把它想成你專案的兩份副本——一份在筆記本上，一份在雲端——靠 *push* 和 *pull* 保持同步：



```
# 把你的本地 repo 連線到一個 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、而你電腦上沒有的提交
git pull origin main
```

把 *push* 想成”存到雲端”，把 *pull* 想成”從雲端取最新”。

拉取請求與議題

兩個你很快會遇到的 GitHub 功能。

拉取請求 (pull request, PR) 是把一個分支合併進另一個分支的提議，並內建了一處空間，讓你在改動併入之前先審查它們。哪怕獨自工作，PR 也有用：它給你一塊乾淨的螢幕，展示每一處改動，讓你在合併進 main 之前審查 AI 的工作。這就

是 diff 習慣的正式化。

議題 (issue) 是關於某件待辦或待修之事——一個 bug、一個功能點子、一項 TODO——的追蹤筆記。議題讓你把待辦清單貼在專案本身上，而不是散落在你腦子裡。

.gitignore 與絕不提交機密

有些檔案絕對不該進 Git。最要緊的幾類：

- **機密**——API（應用程式程式設計介面）金鑰、密碼、資料庫憑據。一旦你把機密提交到 GitHub，就當它已經洩露，哪怕你之後刪掉它。歷史會記得。
- **垃圾**——臨時檔案、下載下來的依賴（比如 node_modules）、構建產物。又大又沒必要追蹤。

你用一個叫 .gitignore 的檔案來控制這件事。你在裡面列出一些模式，Git 就當那些檔案不存在。這對 vibe coder 至關重要，因為 AI 工具特別愛建立帶著真實金鑰的配置檔案，而一不小心就會把它提交上去。

```
# 一個簡單的 .gitignore 檔案
.env
node_modules/
*.log
```

.env 那一行最要緊——機密通常就藏在那裡。在你第一次提交之前就加上 .gitignore，並養成問 AI 的習慣：“這次提交裡有沒有什麼是不該公開的？”。

你的第一個 repo，從頭到尾

這裡把整個迴圈放在一處。新建一個資料夾，把它變成 repo，儲存你的工作，再把它 push 到 GitHub：

```
# 1. 開一個新專案
mkdir my-first-project
cd my-first-project
git init

# 2. 在做任何事之前先保護好你的機密
echo ".env" > .gitignore

# 3. 做你的第一個存檔點
git add .
git commit -m "Initial commit"

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

就這樣。你現在有了一個既有安全網、又有雲端備份的專案。每次取得進展，就重複 add □ commit □ push 的節奏。

讓 AI 來開車（而你仍掌舵）

這裡有讓人鬆一口氣的部分：你完全不必背這些命令。你可以對 AI 說“用一句合適的說明把這個提交了”或者“為新功能建個分支並切過去”，它就會執行正確的命令。這也是一種完全不錯的工作方式。

但陷阱在於——和本書其餘部分是同一個陷阱——即便不是你親手敲的，理解它做了什麼的責任仍然在你身上。真正行得通的中間路線：

- 讓 AI 來執行命令，但頭幾次請讓它用大白話告訴你每條命令做什麼。
- diff 自己讀，任何一次提交之前都要讀。這一條沒得商量。這是你抓住 AI 刪掉不該刪之物的方式。
- 頻繁地、小塊地提交。許多小存檔點勝過一個巨大的，因為倒回時是精準的，而不是全有或全無。

- 幹大事之前先建分支。在大刀闊斧重寫之前讓 AI 先建個分支，好讓 main 保持安全。
- 當一條命令聽起來有破壞性時（任何帶 `reset --hard`、強制 `push` 或刪除分支的），在批准之前先問”這到底會做什麼？“。

你是技術負責人。AI 是你那快速、能幹、偶爾魯莽的隊友。Git 就是那套讓這位隊友造不成真正損害的流程。把它學到足以自信指揮的程度，你就再也不會害怕那個”全部重寫”的按鈕了——因為你永遠有一條回頭的路。

回顧與練習

關鍵要點

- Git 就是專案的存檔點——一條你隨時可以跳回去的完整時間線，正是它讓你能夠毫無顧慮地放手讓 AI 去幹。
- 核心節奏是 `add` → `commit` → `push`：暫存你的改動，用一條清晰的訊息把它存為一次 `commit`，再推送到 GitHub 以備份和分享。
- 任何大動作之前先開分支，每次提交之前先讀 *diff*——這是你抓住 AI 刪掉不該刪的東西的方法。
- `.gitignore` 把機密（如 `.env`）和垃圾（如 `node_modules/`）擋在歷史之外。提交到 GitHub 的機密即便事後刪除，也視為永久洩露。
- 你可以讓 AI 來執行命令，但理解每條命令做了什麼的責任仍在你身上——尤其是 `reset --hard` 這類破壞性命令。

動手試試

新建一個全新的資料夾，親手把整個流程跑一遍：`git init`，建立一個包含 `.env` 的 `.gitignore`，新增一個簡單的 `notes.txt` 檔案，然後 `git add .` 和 `git commit -m "Initial commit"`。現在執行 `git log --oneline` 看看你的存檔點，編輯這個檔案，再執行 `git diff`，看 Git 把改了什麼精確地展示給你。你剛剛在一個絕不會出錯的專案上，用過了本章所有的核心習慣。

本章提示詞

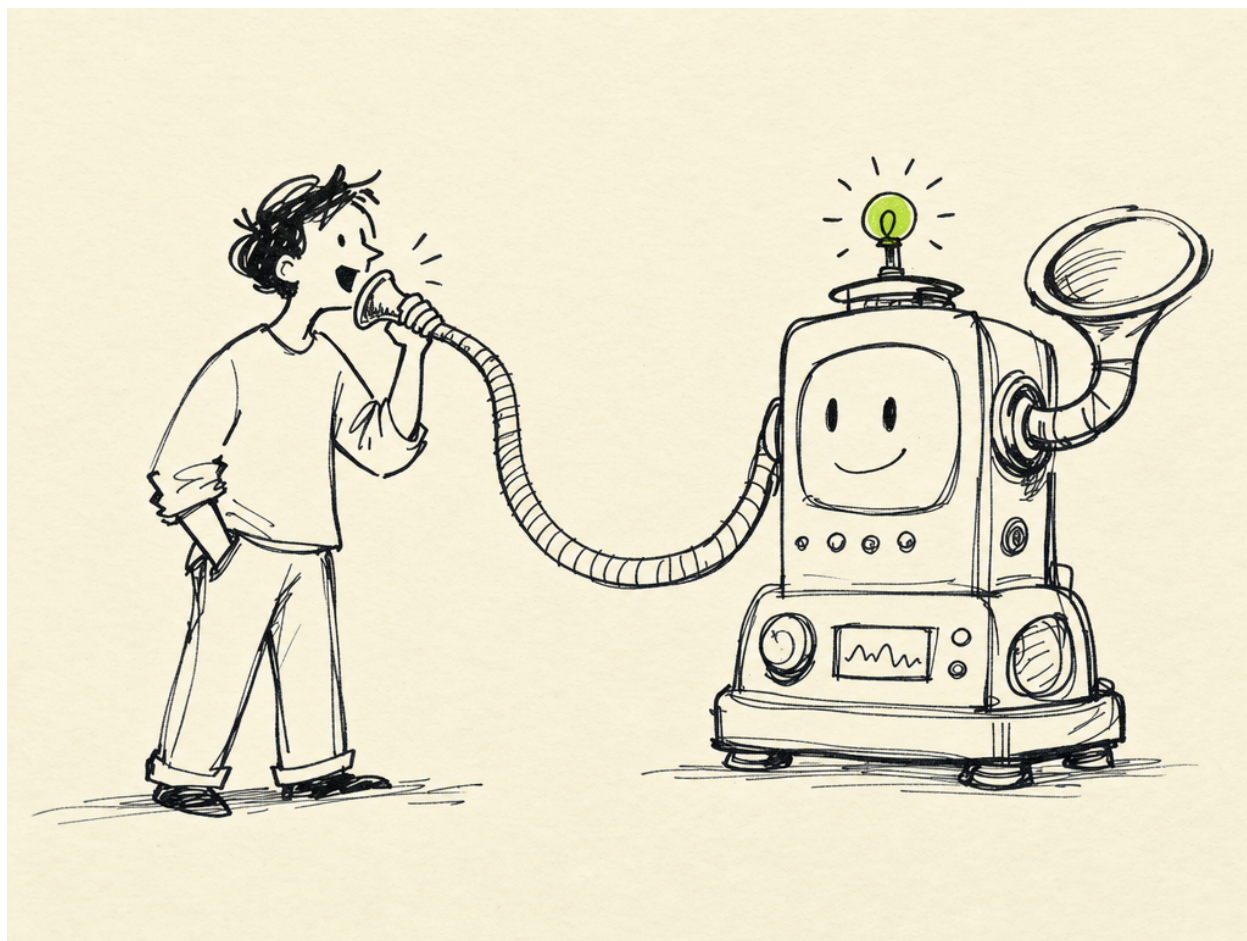
我是剛開始學習 Git 的新手，希望你替我執行命令，

但在過程中教我。對於下面這個任務：

< 描述你想做的事 —例如" 儲存我當前的工作" 或" 安全地嘗試一個有風險的改動">

- 告訴我你將執行哪些 Git 命令，並用通俗的話說明每條命令的作用。
- 如果有破壞性操作（`reset --hard`、強制 `push`、刪除分支），請在執行前清楚地警告我，並說明我會失去什麼。
- 在任何提交之前，給我看 `diff` 並總結改動了什麼。
- 如果改動中有不該公開的內容（秘密、金鑰），提醒我。

The Command Line



第一次看著 AI 程式設計工具幹活時，一個黑色窗口裡滿是向上滾動的文字，一個閃爍的游標等著你輸入。那個視窗就是命令列。對許多沒寫過程式碼的人來說，這是整段旅程裡最嚇人的部分——它看起來就像電影裡那樣，彷彿一步走錯就會抹掉你的整塊硬碟。不會的。命令列只是告訴電腦該做什麼的另一種方式：用文字，而不是點選。

本章要消除這份恐懼。你不需要背下一百個命令。你需要理解正在發生什麼，認出少數幾個每天都會用到的命令，並知道如何安全地執行 AI 遞給你的命令。這些就足以讓你不再害怕，開始真正高效地幹活。

終端和 *shell* 究竟是什麼

用普通應用時，你點選按鈕，應用把你的點選翻譯成給電腦的指令。命令列跳過了按鈕。你輸入一條指令，按下回車，電腦就去執行它。

有兩個詞你會經常聽到：

- **終端 (Terminal)** ——你輸入文字的那個視窗。它只是一個帶歷史記錄的文本框。(在 Mac 上叫“Terminal”或“iTerm”；在 Windows 上叫“PowerShell”或“Windows Terminal”；在 Linux 上通常就叫“Terminal”)
- **shell** ——那個視窗裡面讀取你輸入並執行它的程式。最常見的一個叫 **bash**；Mac 用的是一個叫 **zsh** 的。就本書涉及的一切而言，兩者的行為幾乎完全相同。

每次你按下回車，迴圈就是這樣的：shell 讀取你輸入的內容，找到並執行匹配的程式，再把那個程式產出的任何東西列印回窗口裡。





AI 程式設計工具為什麼住在這裡？因為命令列是開發的通用控制台。安裝工具、啟動應用、執行測試、部署到網際網路——所有這些都通過輸入的命令發生。AI 智慧體的工作方式就是輸入與你會輸入的相同命令，只是更快、沒有拼寫錯誤。當你看著它幹活時，你看到的正是它在使用那扇你也將使用的門。

讀懂提示符

在你輸入任何東西之前，shell 會給你顯示一個提示符 (**prompt**) ——一小段以某個符號結尾的文字，等待輸入。它可能長這樣：

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

從左到右讀，它告訴你：你的使用者名稱 (mini)、你的電腦 (laptop)、你當前所在的位置 (~/projects/my-app 這個資料夾)，然後是表示“輪到你了”的 \$。~ 是你主資料夾的簡寫。結尾的符號通常是 \$ 或 %，有時是 >。別去輸入那個符號——它只是 shell 在說它準備好了。

這比看起來更重要。命令列裡的大多數困惑都來自不知道自己在哪裡。提示符每一次都在告訴你。

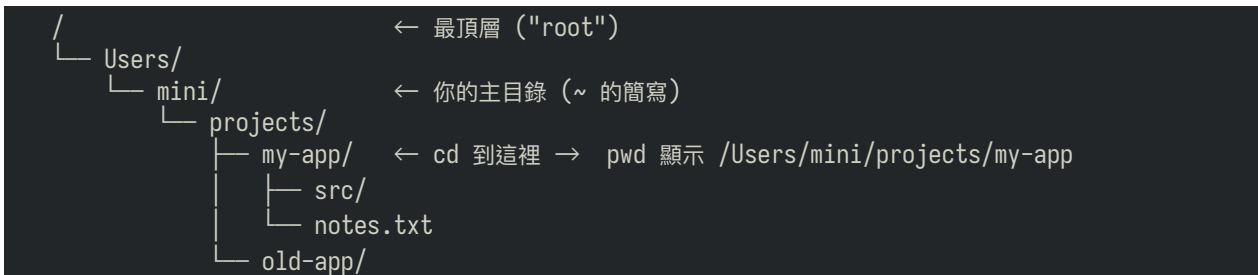
每天都會用到的命令

這裡是你會不斷看到的命令。把它們當句子來讀：一個動詞，有時後面跟著一個物件。

```
pwd                # "print working directory" ——我現在在哪裡？
ls                 # 列出這裡的檔案和資料夾
ls -la            # 列出所有內容，包括隱藏檔案，並帶詳情
cd projects       # change directory ——進入 "projects" 資料夾
cd ..             # 向上一層資料夾 (".." 表示" 上一層資料夾")
cd ~              # 回到你的主資料夾
mkdir my-app      # 新建一個叫 "my-app" 的資料夾
cp notes.txt backup.txt # 複製一個檔案 (原件留著，做出一份副本)
mv old.txt new.txt  # 移動或重新命名一個檔案
rm draft.txt      # 刪除一個檔案 (remove)
```

幾條能省去真實麻煩的提醒：

你的檔案住在一棵層層巢狀的資料夾樹裡，而 cd 就是你在其中行走的方式。這裡有一棵小樹，pwd 顯示了你 cd 進 my-app 後會站在哪裡：



- cd 是你四處移動的方式。配上 pwd 和 ls，這就是你這輩子要做的事的 80%。你進入正確的資料夾，看一看，然後執行點什麼。
- 路徑可以是相對的，也可以是絕對的。cd projects 是相對於你所在位置的；cd /Users/mini/projects 是從最頂端算起的完整地址。

- **rm 是永久的。**命令列裡沒有回收站。你 rm 掉一個檔案，它就沒了——沒有“你確定嗎？”，也沒有撤銷。對 `rm -rf` 要格外小心，它會不問一聲就刪掉一個資料夾及其中的一切。如果你對一條 `rm` 命令會刪掉什麼哪怕有一點不確定，就別執行它。先請 AI 解釋它。

執行一個程式和一個 **dev** 伺服器

做軟體意味著執行它。大多數時候你會輸入一個程式的名字，後面跟上你想讓它做的事。

```
node app.js      # 用 Node 執行一個 JavaScript 檔案
python script.py # 執行一個 Python 檔案
npm run dev      # 啟動專案的開發伺服器
```

最後那條很特別。**dev 伺服器**是一個程式，它在你自己的電腦上執行你的應用，讓你能在瀏覽器裡通過 `http://localhost:3000` 這樣的地址看到它。要理解的關鍵是：它會**一直執行**。終端看起來會“卡住”——它不再給你新的提示符，而是顯示一連串的活動資訊。那不是宕機，那是在幹活。只要那條命令在執行，你的應用就活著。開啟瀏覽器，訪問它打印出的地址，你就會看到你的應用。

幹完之後，你把它停掉（下一節講）。

讀懂輸出和 **exit** 碼

命令執行完之後，通常會列印點什麼。學會讀懂那段輸出是一項真本事，因為這正是你和 AI 弄清事情是否成功的方式。

- **正常的輸出**只是資訊——列出的檔案、進度訊息、伺服器說它已啟動。
- **錯誤**通常會直白地說出來：`Error:`、`command not found`、`permission denied`、`cannot find module`。冒號後面的詞就是你的線索。把整段錯誤複製下來貼給你的 AI——這是出問題時你能遞過去的最有用的一樣東西。

在幕後，每條命令結束時都帶著一個看不見的 **exit 碼**：`0` 表示成功，其他任何值都表示有問題。你很少需要手動去查它，但你會聽到有人提起。最近一次的可以這樣看：

```
echo $?          # 列印上一條命令的 exit 碼 (0 = 正常)
```

如果一條命令悄無聲息地失敗了而你不确定，`echo $?` 會告訴你它到底成沒成功。

用 **Ctrl-C** 停掉一個程序

那個 **dev 伺服器**會一直執行，直到你停掉它。通用的“停止”是 **Ctrl-C**——按住 Control 鍵再按 C。它向正在執行的任何東西發出一個禮貌的“現在請停下”訊號，然後你就拿回了提示符。

- **Ctrl-C**——停掉這個終端裡當前正在執行的程式。結束一個 **dev 伺服器**，或取消一條耗時過長的命令時，這是你的首選。
- **關閉終端視窗**也會停掉其中執行的一切，但 **Ctrl-C** 更乾淨利落。

記住 **Ctrl-C**，你就再也不會覺得被一個“卡住”的終端困住了。

PATH 與“**command not found**”

遲早你會輸入一條命令，卻得到 `command not found`。這幾乎從不意味著你笨；它意味著電腦不知道那個程式住在哪裡。

shell 通過檢視一個叫 **PATH** 的資料夾清單來找程式。當你輸入 `npm` 時，shell 會走遍 **PATH** 裡的每個資料夾，找一個叫 `npm` 的程式。如果哪個資料夾裡都沒有，你就會得到 `command not found`。

你輸入： `npm`



按順序在 **PATH** 各目錄中查詢 "npm":
`/usr/local/bin` → 不在這裡

```
/usr/bin    -▶ 不在這裡  
/opt/bin    -▶ 找到 ✓ -▶ 執行它
```

```
若所有目錄都沒有 -▶ "command not found"
```

```
echo $PATH      # 顯示 shell 會搜尋的資料夾清單
```

實際上 `command not found` 通常意味著兩件事之一：那個工具還沒裝，或者它被裝在了一個不在 `PATH` 上的地方。兩者都很常見，也都能修。把錯誤貼給你的 AI，說出你剛才想執行什麼——修 `PATH` 問題，正是 AI 擅長一步步帶你做的那種繁瑣的配置活兒。

安全地執行 AI 給你的命令

vibe coding 很大一部分就是 AI 說“執行這條命令”，遞給你一行讓你貼上。這很正常，也沒問題。但請養成一個會永遠保護你的習慣：**執行之前先讀**。

你不需要理解每個引數。你需要對這條命令做什麼有個大致的感覺，尤其是開頭的那個動詞：

- `ls`、`pwd`、`cd`、`cat`、`echo`——無害。它們只是看看東西或四處移動。儘管執行。
- `npm install`、`pip install`——它們下載並配置工具。一般是安全的，但它們會改動你的專案。
- `rm`、`mv`、任何帶 `sudo` 的、任何把下載直接灌進 `shell` 的 (`curl ... | bash`) ——慢下來。這些可能刪掉東西，或以對你機器的完全許可權執行程式碼。

如果一條命令看著嚇人，或者你認不出那個動詞，就問：「這條命令是做什麼的，執行它安全嗎？」一個好的 AI 會把它講得明明白白。沒有這道檢查，絕不要貼上來自某個隨便網站的命令。命令列會絲毫不差地照吩咐執行——包括那些危險的事——所以先讀這個小習慣，就是你的安全帶。

你會遇到的包管理器

現代軟體是由成千上萬個預先寫好的小零件搭起來的。**包管理器 (package manager)** 就是下載並整理這些零件的工具。你會不斷遇到兩個：

```
npm install      # 安裝這個 JavaScript 專案需要的一切  
npm install react # 新增一個叫 "react" 的特定包  
pip install requests # 安裝一個叫 "requests" 的 Python 包
```

- **npm** (Node 包管理器) 是 JavaScript 和 Node 專案的包管理器。`npm install` (常寫成 `npm i`) 會讀取一個列出專案所需內容的檔案，並把它們全部下載下來。
- **pip** 為 Python 做同樣的事。

第一次開啟一個專案時常常會跑一次 `npm install`，之後就把它拋在腦後了。當 AI 新增一個需要新包的功能時，它會讓你去安裝它。現在你知道那是什麼意思了：只是去取來一塊 **building block**，沒什麼可怕的。

這就是命令列。不是駭客的武器——只是一個你怎麼吩咐它就怎麼照做的文本框。學會讀懂提示符、認出十來個命令、用 `Ctrl-C` 停掉失控的程序、執行之前先讀。其餘的一切，你都可以問。

回顧與練習

關鍵要點

- 命令列只是一個完全照你說的去做的文本框——用文字代替點選。AI 智慧體的工作方式，就是敲下你本來也會敲的那些命令。
- 提示符每次都告訴你你現在在哪裡；大部分困惑都源於不知道當前所在的資料夾。`pwd`、`ls`、`cd` 佔了你日常操作的 80%。
- `rm` 是永久性的——沒有回收站，沒有撤銷。遇到 `rm`、`sudo`，以及任何把下載內容直接灌進 `shell` 的命令 (`curl ... | bash`) 時，請放慢腳步。

- **開發伺服器**會一直執行，並且故意看起來像”卡住”了——那正是它在工作。**Ctrl-C** 能停止任何正在執行的程式，把提示符還給你。
- **先讀再執行**。你不必懂每個引數，只需對開頭的動詞有個大致感覺。拿不準時，問 AI”這條命令是做什麼的，執行它安全嗎？”

動手試試

開啟終端，在不改動任何東西的前提下探索：執行 `pwd` 看看你在哪裡，`ls` 看看這裡有什麼，用 `cd` 進入一個資料夾再 `ls` 一次，然後 `cd ..` 退回去。注意提示符每次是如何更新的。現在故意執行一條會失敗的命令——敲 `lss`（一個拼寫錯誤）並讀一讀 `command not found` 這條訊息。你剛剛練習了兩條最安全、最常用的命令，還學會了讀錯誤資訊，全程沒碰任何一個檔案。

本章提示詞

我是命令列新手，對執行各種東西有點緊張。

這是你（或某個教程）讓我執行的一條命令：

< 在此貼上命令 >

- 請用通俗的語言一步一步解釋這條命令做什麼。
- 明確告訴我它是否安全，或者它是否可能刪除檔案、更改我的系統，或以完整許可權執行程式碼。
- 如果有風險，請建議一種更安全的做法，或者我應該先檢查什麼。
- 告訴我執行它之前應該處在哪個資料夾裡。

How the Web Works



在為網路構建任何東西之前，先了解網路實際上是什麼會很有幫助。你不需要電腦科學學位，也不需要背誦任何東西。你只需要一個能用的心智模型——一幅在你腦中描繪出的畫面：當你輸入一個地址、一個頁面隨之出現時，到底發生了什麼。

本章就來構建這幅畫面。從登入表單到支付，一個用氛圍編碼（vibe coding）做出來的應用所做的一切，歸根結底都是同一支簡單舞蹈的某種版本：一臺計算機發問，另一臺計算機作答。一旦你能清晰地看到這支舞，本書的其餘部分就不再像魔法，而開始像管道工程。

兩個角色：客戶端與伺服器

網路上幾乎所有東西，都是兩個角色之間的一場對話。

- **客戶端**是發問的那一方。多數時候那就是執行在你手機或筆記本上的網頁瀏覽器——Chrome、Safari、Firefox。它就像走到櫃檯前的顧客。
- **伺服器**是作答的那一方。它是坐落在某處資料中心裡的一臺計算機，整天執行著，等待著請求。它就像櫃檯後面的廚房。

客戶端永遠不做伺服器的活，伺服器也永遠不做客戶端的活。瀏覽器問：“把首頁給我。”伺服器答：“給你。”整個關係就是這樣。伺服器不是什麼特殊的魔法盒子——它只是一臺普通的計算機，其工作就是等待問題、送回答案。

還有一個你會不斷聽到的詞：**request**（請求）（客戶端發出的問題）與 **response**（響應）（伺服器送回的答案）。把這兩個詞握在手邊；它們是下面一切的脊梁。

URL 實際上是什麼

URL（統一資源定位符）就是你所請求的地址。它看起來像一長串字元，但其實是幾個帶標籤的部分鏈接而成。把它拆開來看：

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

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

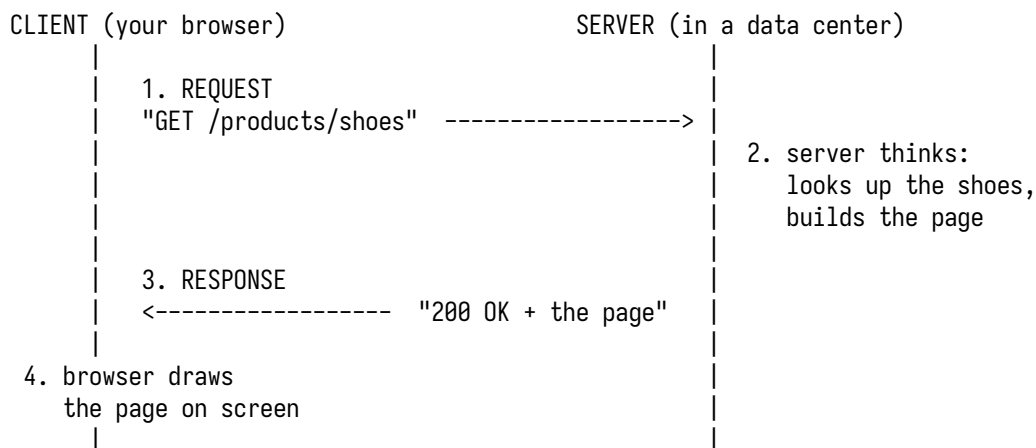
像讀一個倒過來的郵政地址那樣讀它：

- **scheme** (`https`) 是協議——對話的規則。如今幾乎總是 `https`。
- **host** (`shop.example.com`) 指明要聯絡哪一臺伺服器。
- **path** (`/products/shoes`) 指明你想要那臺伺服器上的哪樣東西。
- 可選的 **query** (`?size=10`) 攜帶額外引數，比如篩選條件或搜尋詞。

當有人說“端點”或“路由”時，他們幾乎總是指某臺特定 `host` 上某個特定 `path`。那些嚇人的詞的全部含義也就是這些。

請求與響應的迴圈

這就是那支舞，從頭到尾。你輸入一個地址，幕後會在不到一秒內發生這樣的事：



每一次都是四個步驟：

1. 客戶端向某個 URL 傳送一個 **request**。
2. 伺服器處理它——也許它會查詢些什麼、執行一些邏輯、核實你是誰。
3. 伺服器送回一個 **response**：一個狀態（成功了嗎？）外加一些內容（頁面、圖片或資料）。
4. 客戶端拿這個響應做點什麼——通常是把它畫到你的螢幕上。

載入一個網頁並不是一個請求——而是幾十個。瀏覽器先請求頁面，接著發現它還需要圖片、字型和樣式，於是為每一樣都射出一個新的請求。同一支舞，許多遍，非常快。

HTTP 方法：動詞們

每個請求都帶著一個 **方法**——一個說明你想要哪種動作的動詞。方法有好幾個，但四個就涵蓋了幾乎一切，而且它們乾淨利落落地對應到日常的觀念上：

- **GET** ——“把這個給我。”讀取、取回、檢視。載入一個頁面或一份列表。GET 永遠不應該改變任何東西；它只是取回。
- **POST** ——“這裡有個新東西。”建立。提交表單、註冊、發表評論。

- **PUT** ——“更新這個。”修改某個已經存在的東西，比如編輯你的資料。（對於部分更新你還會聽到 **PATCH**——同樣的思路，範圍更小。）
- **DELETE** ——“移除這個。”就是字面意思——刪掉一條記錄或一個檔案。

別的都忘了也沒關係，記住這個：**GET** 讀取，**POST** 建立，**PUT** 更新，**DELETE** 移除。當你的應用出毛病時，知道發出的是哪個動詞，往往是第一條線索。

狀態碼：成功了嗎？

每個響應回來時都帶著一個三位數的 **狀態碼**——伺服器對這次請求結果的一眼判定。它們按首位數字分組：

- **2xx** ——**成功**。成功了。200 OK 是日常那句“喏，這是你想要的東西”。
- **3xx** ——**重定向**。“這兒沒有，去那邊看吧。”常見且無害；瀏覽器會自動跟過去。
- **4xx** ——**是你（客戶端）出了錯**。最有名的是 **404 Not Found**——你請求了一個不存在的 path。（還有 401/403——你沒登入，或沒被允許。）
- **5xx** ——**是伺服器出了錯**。**500 Internal Server Error** 意味著伺服器在處理你的請求時，它自己的程式碼崩了或壞了。

4xx/5xx 的區分值得刻在腦子裡，因為它告訴你該往哪兒看：

- **404** 意味著請求錯了——通常是個糟糕的 URL 或打錯的 path。看看到底在請求什麼。
- **500** 意味著伺服器倒下了。你的請求沒問題；是後端程式碼壞了。去看伺服器日誌，而不是 URL。

當一個用氛圍編碼做出來的應用出了岔子，“這是 4xx 還是 5xx？”是最快能用來分診的問題之一。

狀態碼				
2xx	→	成功	成功了（200 OK）	
3xx	→	重定向	" 去那邊看看"（自動跟隨）	
4xx	→	客戶端錯誤	你的請求有誤	→ 檢查 URL/path
5xx	→	伺服器錯誤	後端崩潰了	→ 檢查伺服器日誌

頭部與 Cookie

與正文內容並排，每個請求和響應都帶著 **頭部 (headers)** ——一張張貼著標籤、裝著額外資訊的小紙條。它們就像信封上寫的字，跟裡面的信是分開的。頭部會說明諸如內容是什麼格式 (Content-Type: text/html)，或者是誰在發問。

Cookie 是頭部的一種具體而重要的用法。網路的記性很短：預設情況下，每個請求都是孤立的，伺服器一答完就把你忘了。Cookie 修補了這一點。你登入之後，伺服器遞給你的瀏覽器一個小小的令牌——一個 **cookie**——而你的瀏覽器會在之後的每一個請求裡自動把它帶上。這就是網站如何跨頁面“記住”你已登入的方式。沒有 **cookie**，就沒有記憶；你每點一下都得重新登入一次。

HTTPS 與那把鎖

你見過位址列裡那把小鎖。它意味著這條連線正在使用 **HTTPS**（超文本傳輸安全協議）——被一層叫做 **TLS** 的加密包裹起來的 HTTP（超文本傳輸協議）。

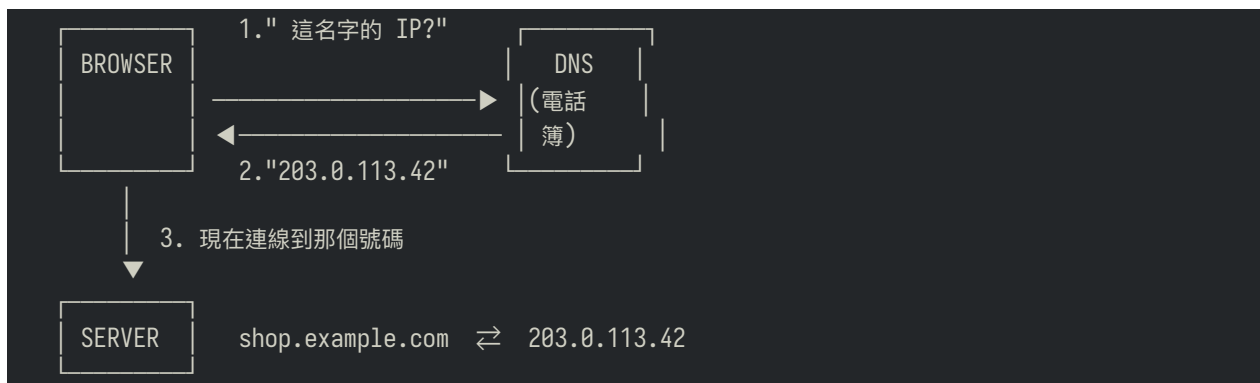
樸素的 HTTP 把一切都以可讀的文本傳送。坐在你和伺服器之間的任何人——比如在公用 Wi-Fi 上——都能在你的密碼經過時把它讀走，就像郵路上人人可讀的明信片。HTTPS 把這場對話封進一個只有你和伺服器才能開啟的信封裡。那把鎖確認了兩件事：

- **隱私**——中間的任何人都讀不到你傳送的内容。
- **身份**——你確實在跟你以為的那臺伺服器對話，而不是冒名頂替者。

實用的規則很短：真正的應用使用 HTTPS，永遠，毫無例外。任何用樸素 HTTP 處理登入或支付的東西，從設計上就是壞的。

從名字到伺服器：**DNS**

還有一根沒收尾的線頭。你輸入 `shop.example.com`，但計算機其實並不靠名字找到彼此——它們用像 `203.0.113.42` 這樣的數字 **IP 地址**。**DNS** (Domain Name System，域名系統) 是網路的電話簿：在你的瀏覽器能發出請求之前，它會悄悄地請求 DNS 把那個友好的 **域名** 翻譯成數字地址，然後連線過去。



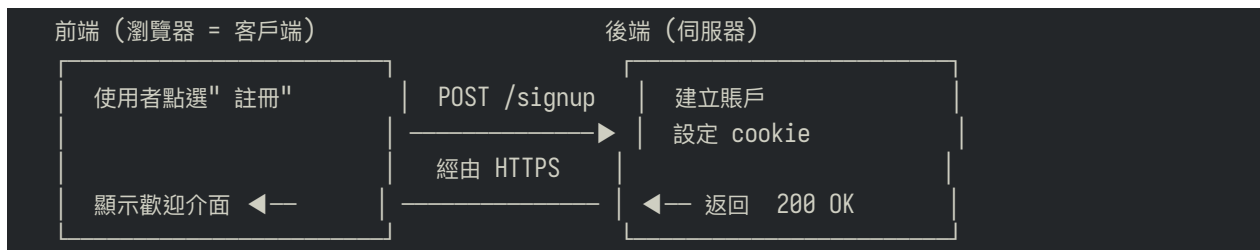
你買下一個域名，把它的 DNS 記錄指向執行你應用的那臺伺服器，從此這個名字就把人們引向你的機器。這是一個你幾乎永遠看不到的查詢步驟，但一個好記的名字之所以能替代一串數字，正是因為它。

這一切如何對應到你的應用

現在，把這整幅畫面鋪到一個用氛圍編碼做出來的應用之上，因為你將要構建的每一塊，都住在這支舞的某一側：

- **前端**是執行在瀏覽器裡的一切——使用者看到並點選的按鈕、表單和佈局。它就是客戶端。它主要負責傳送請求、顯示響應。
- **後端**是執行在伺服器上的你的程式碼——決定如何處理一個請求、與資料庫對話、並組裝出響應的那套邏輯。它就是伺服器。

把整個註冊之旅放進一幅圖裡就是這樣——本章每一個加粗的詞都各就各位：



一次真實的互動會把這一切縫在一起。使用者點選“註冊”（前端）。瀏覽器帶著表單資料，向一個像 `/api/signup` 這樣的 path、通過 **HTTPS** 傳送一個 **POST** 請求。你的後端收到它，建立帳戶，設定一個 **cookie** 讓使用者保持登入，然後送回一個 **200**。前端看到成功，於是展示一個歡迎頁面。那裡每一個加粗的詞，都是你剛剛學過的東西——而那，就是你將要構建的幾乎一切的整體形狀。

你不必把這一切一次性全都裝在腦子裡。但當後面某一章說“請求打到了伺服器，卻以 **500** 返回”，或者“設定這個頭部”，又或者“這個路由處理那個 **POST**”時，你現在就能確切地知道它指的是這支舞的哪一部分。那個心智模型，才是真正的先決條件——而你已經擁有它了。

回顧與練習

關鍵要點

- 網路就是一支不斷重複的簡單舞蹈：**客戶端**（通常是瀏覽器）發出**請求**，**伺服器**返回**響應**。其餘的一切都只是疊加在這之上的細節。
- **URL** 是帶標籤的幾個部分拼接而成——scheme、host、path、query。方法就是動詞：**GET** 讀取，**POST** 建立，**PUT** 更新，**DELETE** 刪除。
- **狀態碼**是伺服器一眼給出的判定。4xx/5xx 之分告訴你該往哪兒看：**4xx** 意味著請求錯了（URL 不對），**5xx** 意味著伺服器自己的程式碼出了問題。
- **HTTPS**（小鎖）把對話封起來，讓中間任何人都讀不到——真正的應用始終使用它。**Cookie** 是網站跨多個請求記住你已登入的方式。
- 你的**前端**是客戶端（在瀏覽器裡執行），你的**後端**是伺服器（執行你的邏輯並與資料庫對話）。你構建的每個功能都處在這支舞蹈的某一側。

動手試試

開啟任意一個網站，然後開啟瀏覽器的開發者工具（右鍵 □“檢查”，再切到”Network/網路”標籤）。重新整理頁面。看著請求列表被填滿——你會看到一個頁面請求，緊接著是為圖片、字型和樣式發出的幾十個請求，正如前文所述。點開其中一個，看它的方法（GET）、狀態碼（但願是 200）和標頭。你現在正即時看著請求/響應這支舞蹈上演，而且具備了讀懂它的詞彙。

本章提示詞

我是初學者，正在學習網路是如何運作的。用我自己的應用作為例子：

< 描述你的應用，或貼上一個 URL 或你看到的錯誤 >

- 針對一個具體的操作（比如載入頁面或提交表單），一步一步帶我走一遍請求/響應的迴圈。
- 告訴我哪一部分是客戶端（前端），哪一部分是伺服器（後端）。
- 如果我看到一個狀態碼或錯誤，請解釋它是 4xx（我的請求有誤）還是 5xx（伺服器崩潰了），以及該去哪裡檢視。
- 保持具體且對初學者友好 ——不假設任何已有知識。

API、JSON 與資料



你做的應用幾乎不可能孤立存在。只要你想顯示一張地圖、扣一筆卡、發一封郵件，或者查一下天氣，你的應用就得去和別人的應用對話。它說的這門語言叫 **API**（應用程式程式設計介面），而它們來回傳遞的話語，幾乎總是寫成一種叫 **JSON**（JavaScript 物件表示法）的格式。這一章講的就是這兩樣東西——不是為了讓你親手去寫這些管道，而是為了讓你能讀懂它、能檢查 AI 搭出來的東西是否靠譜、能在它害你花冤枉錢之前就發現哪裡出了問題。

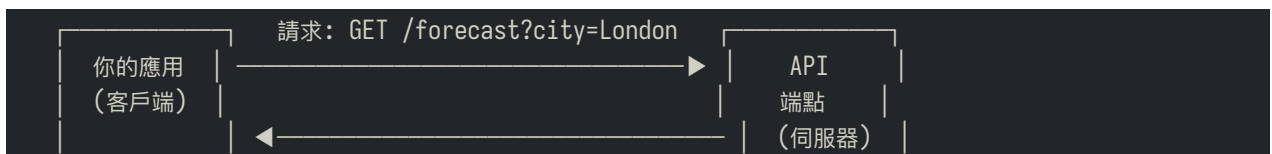
這裡沒有任何東西需要你死記硬背。你需要的是認出這些”形狀”。當 AI 說”我會呼叫 Stripe API 並解析 JSON 響應”時，你應該能大致想像出這是什麼意思、以及哪裡可能會出岔子。

API 究竟是什麼

理解 API 最乾淨的方式，就是把它看成一個程式向另一個程式提供的一份請求選單。

餐廳不會讓你直接走進廚房自己做菜。它給你一份選單：這是你能點的東西，這是點菜的方式，這是你會拿到的東西。你不需要知道廚房是怎麼運作的，你只需要這份選單。API 正是如此——家公司說”這是你被允許向我們服務發出的請求，這是每個請求會返回的東西。”

當 AI 寫出”呼叫天氣 API”的程式碼時，它就是在照著那份選單點單：把這個城市的天氣預報給我。天氣公司的伺服器幹活，然後把答案遞回來。你的應用永遠看不到他們的資料庫或程式碼，它只拿到那道菜。



照著選單
點餐

響應: { "temp": 14, ... } (JSON)

幹活,
上菜

最重要的思維轉變是：API 是一份契約。對方公司承諾，只要你按特定方式提問，就會得到特定形狀的答案。你的工作——也是 AI 的工作——就是正確地提問、並正確地讀懂答案。大多數 API 的 bug，都是某一方違背了這份契約。

用大白話講端點和 *REST*

一個端點 (*endpoint*) 就是選單上的一道菜。它就是一個做某件具體事情的 URL (統一資源定位符)。比如：

- `https://api.weather.com/forecast` ——獲取一份天氣預報
- `https://api.weather.com/cities` ——列出可用的城市

同一個廚房，不同的菜。每個端點就是一項能力。

你會接觸到的大多數 API，都遵循一種寬鬆的風格，叫 *REST* (表述性狀態轉移)。你不需要學術定義，你只需要兩個想法：

- **URL** 給一個東西命名 (一個“資源”) ——一個使用者、一個訂單、一份天氣預報。
- 動詞說明你想拿它做什麼。這些動詞就是 HTTP (超文本傳輸協議) 方法，和“網路是怎麼運作的”那章裡講的是同一批：
 - GET ——讀取某個東西 (取一份天氣預報)。安全；什麼都不改變。
 - POST ——建立某個東西 (下一個訂單、註冊一個使用者)。
 - PUT / PATCH ——更新某個已經存在的東西。
 - DELETE ——刪除某個東西。

所以 GET `/orders/42` 的意思是“讀取 42 號訂單”，而 DELETE `/orders/42` 的意思是“刪掉它”。同一個名詞，不同的動詞，效果完全不同。當你讀 API 文件或一段 diff 時，那兩個詞——方法和 **URL**——就告訴了你一個請求所做的大部分事情。

JSON 是什麼

當你的應用向 API 提一個問題時，答案會以一種叫 **JSON** 格式的文本回來。它之所以成了預設選擇，是因為它既能被人讀懂，也容易被程式處理。你會不停地見到它，所以能讀懂它很值得。

JSON 由鍵和值構成。鍵是一個帶引號的標籤；值是附在它上面的資料。下面是一個真實的例子，描述了一個使用者：

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

把它當成一張有人填好的表格來讀。“name”是一個欄位，“Ada Lovelace”是他們填進去的內容。兩種結構完成了所有的工作：

- 一個 **object** 用花括號 { } 包起來。它是一捆鍵/值對——就像一條記錄。上面整個例子就是一個 object。
- 一個 **array** 用方括號 [] 包起來。它是一個有序的值列表。在上面，“tags”裝著一個由兩個 string 組成的 array。

注意 “profile” 是一個 object 裡面的 object。JSON 是可以巢狀的——一個值本身可以是一個 object 或一個 array，想嵌多深就嵌多深。這就是為什麼一個響應可以一次性描述一個使用者、他們的地址、以及他們最近的十個訂單。

把同一個使用者 `object` 畫成一棵樹，它的形狀就一目瞭然——平坦的欄位在上面，一個列表和一個巢狀的 `object` 從中分出枝杈：

```
{ } user (物件)
├── id ..... 42 (數字)
├── name ..... "Ada Lovelace" (字串)
├── isAdmin ..... false (布林值)
├── tags ..... [ ] (陣列)
│   ├── "beta"
│   └── "early-access"
└── profile ..... { } (物件)
    ├── city ..... "London" (字串)
    └── avatarUrl ..... null (null = 無值)
```

你會遇到的資料型別

JSON 裡的每個值，都屬於一小組型別中的一種。認識它們，能讓你看出某個值的型別錯了——這是一個常見的 bug 來源。

- **string** —— 文本，總是帶雙引號：`"Ada Lovelace"`、`"ada@example.com"`。
- **number** —— 一個普通的數字，不帶引號：`42`、`3.14`。(引號會改變含義：`"42"` 是文本四二，不是數字)
- **boolean** —— `true` 或 `false`，不帶引號。用於像 `"isAdmin"` 這樣的是/否標誌。
- **null** —— 一個刻意的「這裡沒有值」。`"avatarUrl": null` 表示這個使用者沒有頭像——這和這個鍵根本不存在是兩回事。
- **array** —— 一個有序列表：`["beta", "early-access"]`。
- **object** —— 一捆鍵/值對：那個 `"profile"` 塊。

整套詞彙就這些了。現實中最常見的錯誤是 `string` 與 `number` 的陷阱：程式碼期待的是數字 `42`，但 API 發來的是字串 `"42"`，於是一次比較就悄無聲息地失敗了。當某個東西表現得很怪時，檢查一個值的型別往往就是解藥。

一個真實的請求和響應

下面是一次真實的 API 呼叫從頭到尾的樣子——你的應用發出去的請求，以及它拿回來的 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` 就是下一個話題——也是最值得做對的那個。

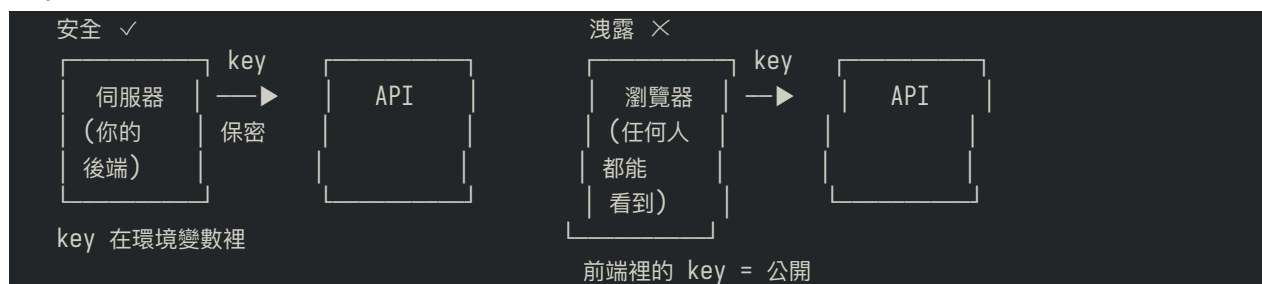
API key 與認證 header

大多數有用的 API 並不對全世界開放。公司需要知道是誰在提問，這樣他們才能統計用量並向你收費。他們會給你一個 **API key**——一長串能標識你賬戶的金鑰字串。你把它附在每個請求上，通常放在一個叫 Authorization 的 **HTTP header** 裡，就像上面那行 `Bearer sk_live_...`。

把 API key 完全當成密碼來對待，因為它本來就是密碼。任何拿到它的人都能以你的身份發請求——並刷爆你的賬單或讀走你的資料。這直接關聯到安全那一章，而其中最重要的有一條規則：

- 絕不要把 **API key** 放進會發到瀏覽器裡的程式碼。你前端裡的任何東西都是公開的——使用者可以開啟開發者工具讀到它。AI 被要求“從頁面裡呼叫 API”時，會樂呵呵地把你的金鑰直接貼在那兒。攔住它。金鑰屬於伺服器。
- 絕不要把 **key** 提交進 **git**。就算你之後刪掉它，它也會永遠活在歷史記錄裡，而機器人會在幾分鐘內掃描公開倉庫裡的 key。
- 把 **key** 存在環境變數裡，而不是程式碼本身裡——這和安全那一章講的是同一個模式。

key 必須從一個安全的地方出發，絕不能從瀏覽器出發。設想一下這兩條路——一條安全，一條洩露：



如果你哪天看到一個真實的 key 出現在一段 diff 裡、一個前端檔案裡、或一條聊天訊息裡，就把它當成已經洩露了：立刻輪換它（生成一個新的，停用舊的）。

速率限制與賬單暴雷陷阱

還有兩件選單不會大聲告訴你、但絕對會咬你一口的。

速率限制 (Rate limits)。API 會限制你在一段時間視窗內能發多少請求——比如每分鐘 100 次。超了你就會被拒絕，收到一個 429 Too Many Requests 錯誤，直到你慢下來。這個機制是為了防止某一個客戶把服務壓垮。陷阱在於：AI 寫的程式碼常常會在一個迴圈裡不停地調 API、中間不停頓，瞬間就把限額衝爆。如果一個功能對單個條目能正常工作，但跑上一千個就崩了，速率限制就是頭號嫌疑犯。

計量收費的 API。很多 API 按請求收費——每次幾分之一美分。調幾次不算什麼，但當某個東西失控地迴圈時，就成了一筆真實的賬單。經典的恐怖故事是：一個 bug 把一個失敗的付費呼叫永遠重試下去，或者一個公開的表單讓陌生人能觸發付費呼叫，於是你一覺醒來收到一張四位數的發票。保護好你自己：

- 在接入一個付費 API 之前先讀定價。搞清楚每次呼叫的成本。
- 如果供應商的後臺提供這功能，就設一個**消費上限或預算提醒**。大多數都提供。
- 對任何在迴圈裡、或在每次頁面載入時呼叫付費 API 的程式碼，都要心存警惕。

怎麼讀 API 文件

你會長期泡在 API 文件裡，所以這裡教你怎麼不讀全部就快速拿到你要的東西。對任何一個端點，去搜尋這五樣東西：

- **方法和 URL**——GET /forecast。它做什麼，兩個詞說清。
- **認證**——它需要什麼 header 或 key。通常整個 API 共用一個章節。
- **引數**——你可以發的輸入（這裡是 city），哪些是必填的，以及它們的型別。
- **一個示例響應**——你會拿回的 JSON 形狀。這是寶貝；把它複製下來。
- **錯誤**——出問題時會返回什麼，以及狀態碼（401 未授權、404 未找到、429 觸發速率限制）。

好的文件會包含一個可以直接複製貼上的示例請求和響應。這兩段程式碼片段，比幾段文字告訴你的還要多——把它們抓過來，交給 AI。

讓 AI 去做呼叫——而你來檢查形狀

這裡是實際的分工。AI 真的很擅長寫呼叫 API 的程式碼：它懂那些庫、懂 header 的語法、懂錯誤處理。讓它來。這些管道不用你親手寫。

但有一件事是你的，而 AI 很不擅長：判斷資料的形狀到底對不對。一個好的工作流：

- 把真實的文件給 AI。把文件裡的示例請求和響應粘進你的提示裡。AI 憑記憶猜一個 API 的形狀，是頭號 bug 來源——它會編出一些看起來很合理、但根本不存在的鍵。真實的例子能把它錨定在現實上。
- 讓它先把原始響應給你看看。”在解析之前，把 API 實際返回的 JSON 打印出來。“然後把它和程式碼所期待的對照一眼。它讀取的那個鍵（`response.current.temp`）在響應裡真的存在嗎？一半的 API bug 都是鍵名對不上。
- 檢查一下型別是否靠譜。那是個 number 還是 string？一個本應總是存在的欄位，會不會有時候是 null？你現在能讀懂 JSON 了——用上它。
- 處理不順利的那條路。問一句：“如果這次呼叫失敗了、超時了、或者撞上速率限制，會發生什麼？”如果答案是“應用崩潰”，那是個 bug，不是特性。

你不需要寫那個請求。你需要的是讀懂響應、並問一句“這是我期待的形狀嗎？”。這一個習慣——拿資料去對照契約——就能在你的使用者看到之前，抓住大部分整合 bug。

回顧與練習

關鍵要點

- **API** 是一個程式向另一個程式提供的請求選單，也是一份契約：按約定的方式提問，就會得到約定形狀的回答。大多數 API 缺陷，都是某一方破壞了這份契約。
- **JSON** 是回答返回時的格式——由鍵和值構成，用物件 `{ }` 和陣列 `[ ]` 搭建，並且會巢狀。資料型別有 string、number、boolean、null、array、object。
- 字串與數字的陷阱（42 對 "42"）以及缺失與 null 之分，是最常見的形狀缺陷。你現在能讀 JSON 了——遇到行為怪異時，去檢查值的型別。
- **API 金鑰** 就是密碼。絕不要傳送到瀏覽器，絕不要提交進 git，把它存在環境變數裡。洩露的金鑰應當立即輪換（rotate）。
- 留意速率限制（429）和按量計費——AI 寫的、毫不停頓地呼叫付費 API 的迴圈，正是你收到四位數賬單的途徑。先讀定價，並設定預算提醒。

動手試試

在瀏覽器裡找一個無需金鑰的免費 API——比如在新標籤頁直接開啟 <https://api.github.com/users/octocat>。你會看到一段原始 JSON 響應。像讀表單一樣讀它：挑出物件 `{ }`，如果有陣列就找出來，併為三個不同的值說出它們的型別（哪些是字串？數字？布林值？有沒有是 null 的？）。你剛剛做的，正是本章要求你對 AI 接入的每個 API 響應都做的那項檢查。

本章提示詞

我正在呼叫一個 API，作為初學者，我想自己檢查資料的形狀。

這是真實的文件（示例請求 + 示例響應）：

< 把 API 文件裡的示例請求和 JSON 響應貼上到這裡 >

- 在編寫解析程式碼之前，先給我看這次呼叫返回的原始 JSON，並指出那些鍵、它們的型別，以及任何可能為 null 或缺失的東西。
- 確認我的程式碼讀取的鍵在那個響應裡確實存在。
- 確保 API key 是從伺服器上的環境變數讀取的，絕不要硬編碼或傳送到瀏覽器。
- 告訴我如果這次呼叫失敗、超時或觸發速率限制會發生什麼。

一個應用的解剖



當你讓 AI “加一個登入頁面”或者“把這個存到資料庫裡”時，它其實悄悄地在為五六個互相牽動的部件做決定。如果你不知道這些部件的存在，那些回答讀起來就像外語一樣——你也分不清哪個是好答案、哪個是它自己都沒搞清楚的話。

這一章是一張地圖。它不會讓你變成工程師，但它能讓你指著一個現代應用的任何一塊說出它是幹什麼的、它住在哪裡。一旦你有了這張地圖，後面每一章都不再抽象：在你開始描述一個問題之前，你就會知道它屬於哪一個方框。我們會說得很直白——不會甩出沒有定義的術語，並且會用一個貫穿始終的例子（使用者點選一個按鈕）把每一層都走一遍。

全域性圖景

幾乎你要做的每一個應用，都由四個互相對話的部分組成：

- **frontend**（前端）——使用者看到並點選的東西。瀏覽器或手機應用裡的那些介面、按鈕和文字。
- **backend**（後端）——執行在你掌控的伺服器上的邏輯。它來執行規則、幹實際的活、並決定誰有權做什麼。
- 資料庫 (**database**) ——資訊存放的地方，這樣到了明天它還在。賬號、帖子、訂單。
- 託管 (**hosting**) ——在世界某個角落裡真正執行以上這一切的那些電腦，好讓別人能夠訪問到它。

它們是這樣連起來的：



你看到的東西

規則 + 邏輯

資料存放的地方

這一切都執行在 HOSTING 上

frontend 從來不會直接跟資料庫對話。它總是要經過 backend，因為規則住在那裡。這一個事實就解釋了應用是怎麼搭起來的很多道理——也解釋了為什麼“乾脆讓頁面直接讀資料庫”幾乎從來都不是答案。

frontend、backend、資料庫、託管

我們再往深一層看，用我們的例子：一個使用者點選了“儲存資料”。

frontend。這是看得見的應用，跑在網頁瀏覽器裡，或者作為手機應用。它把表單畫出來，注意到點選，並收集使用者輸入的內容。它在設計上就是不可信的：任何人都能開啟它、檢查它、改動它。所以它的工作是好看、響應靈敏，而不是去執行任何重要的東西。

backend。當按鈕被點選時，frontend 把資料發給 backend——也就是跑在伺服器上的程式碼。backend 檢查使用者是否已登入、校驗資料（“這真的是一個郵箱地址嗎？”）、套用業務規則，然後才把它儲存下來。信任就住在這裡，因為使用者沒法篡改它。

資料庫。backend 把資料交給資料庫去儲存。它不是什麼魔法盒子——你可以把它想成一組非常嚴格的電子表格（叫做表 / table），能夠瞬間被檢索、並且永遠不會忘記。到了明天，backend 再把這份資料讀出來重新顯示。

託管。在 prod（生產）環境裡，這一切都不會跑在你的筆記本上。它們跑在從託管服務商那裡租來的電腦上，比如 Cloudflare、Vercel，或者某個雲平臺。託管就是讓你的應用能夠通過一個真實的網址被訪問到，而不僅僅是在你自己的機器上。

一個有用的直覺：當某個東西壞了的時候，問問是哪一層。“按鈕點了沒反應”是 frontend。“它存錯了東西”通常是 backend。“它把我的資料忘了”指向資料庫。說出是哪一層，問題就解決了一半——同時也是寫出一個 AI 能照著乾的 prompt 的一半。

HTML、CSS 和 JavaScript

在瀏覽器裡，frontend 恰好由三種語言搭起來。你不需要親手去寫它們，但你需要知道每一種各自管什麼。

HTML（超文本標記語言）是結構——內容以及它的骨架。它說“這裡是一個標題，這裡是一個段落，這裡是一個按鈕”。它是頁面的名詞；HTML 單獨存在時，是一份樸素、沒有樣式的文件，就像一個帶標籤的文本檔案。瀏覽器會把你的 HTML 變成一棵存在記憶體裡、活的物件樹，叫做 **DOM**（Document Object Model，文件物件模型）；當代碼“改變頁面”時，它其實是在改變 DOM，然後瀏覽器重新繪製以保持一致。

CSS（層疊樣式表）是樣式——一切看起來是什麼樣。顏色、字型、間距、佈局、靠左還是居中、小螢幕還是大螢幕。CSS 不會改變頁面上有什麼，只改變它怎麼呈現。如果說 HTML 是名詞，那 CSS 就是形容詞。

JavaScript (JS) 是行為——你互動時發生了什麼。它監聽點選、把資料發給 backend、顯示一個載入轉圈、在答覆回來時更新 DOM。HTML 和 CSS 是靜態的；**JavaScript** 讓一個頁面做事情。它是動詞。（同一種語言通常也用來跑 backend，這就是為什麼你到處都會看到它。）

有一種方式可以想像這種分工：HTML 搭起一棵樹（DOM），CSS 給它上色，而 JavaScript 伸進去改動它：

HTML → 構建 DOM 樹

CSS → 為其設定樣式

JS → 改變它

```
page (document)
├── header
│   └── "My App"
└── form
    ├── input (name)
    └── button "Save"
```

為每個節點
應用顏色、字型、
間距

點選時，
更新某個節點
→ 瀏覽器重繪

包和依賴

沒有人會從零開始寫一整個應用。那些又難又常見的問題——格式化日期、處理支付、渲染一個日曆——別人早就解決好並打包好了。一個包 (**package**) 就是一塊可複用的程式碼，你把它拉進自己的專案，而不用自己去寫。你專案的**依賴 (dependencies)** 就是它所依賴的那一串包的清單。

在 JavaScript 的世界裡，管這件事的工具是 **npm** (Node Package Manager, Node 包管理器)。有兩個檔案負責記賬：

- **package.json** ——人類可讀的清單，列出你的專案需要什麼，外加執行它的命令。這個檔案是由你（或者 AI）來編輯的。
- 鎖檔案 **lockfile** (**package-lock.json** 或類似的檔案) ——一份精確的、由機器生成的記錄，記下每一個包及其確切的版本，好讓這個應用在你的機器、隊友的機器、以及伺服器上都構建得一模一樣。這個檔案你不用手動去改；交給工具去管理就好。

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

為什麼要在意這些？當 AI 說“我們加一個庫來做這個吧”時，它就是在編輯 **package.json**，而你正在信任別人的程式碼——更少、更知名的依賴，比一堆來路不明的依賴要安全。還有，當一次構建莫名其妙地壞掉時，依賴版本對不上是常見的嫌疑犯，而“刪掉再重新安裝這些包”是一個實實在在、很常用的修法。

環境變數與配置

有些值是不能寫進你的程式碼裡的：金鑰、資料庫密碼、某個服務的地址。還有一些值需要隨著應用執行的地方而改變。你在筆記本上測試的程式碼 (**dev** 環境) 應該連到一個測試資料庫；你的使用者真正正在用的那個線上應用 (**prod**，即生產環境) 必須連到真實的那個——而且用的是同一份程式碼。

答案就是**環境變數 (environment variables, 常簡稱“env vars”)**：一些有名字的值，它們住在程式碼之外，並在程式碼執行時被供給進去。程式碼說“把 DATABASE_URL 給我”，然後由環境來決定要交出哪一個。

```
# 一個配置檔案 (例如 .env) ——值住在這裡，而不是在程式碼裡

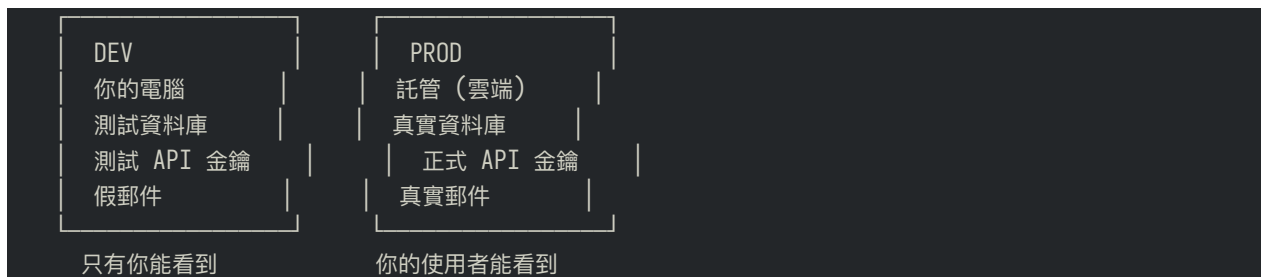
DATABASE_URL=postgres://localhost/myapp_dev
STRIPE_SECRET_KEY=sk_test_51H...      # dev = 一個測試用的金鑰
SEND_REAL_EMAILS=false                 # 測試時別給真實使用者發郵件
```

有兩條規則能幫你省掉實打實的痛苦：

- **金鑰絕對不放進程式碼裡**。你程式碼裡的任何東西都可能最終變成公開的（尤其是在一個共享的程式碼倉庫裡），而洩露一個支付金鑰或資料庫密碼是一場貨真價實的緊急事件。金鑰應該放在 **env vars** 裡，並且不進入版本控制。
- **dev 和 prod 是不一樣的**。重點在於同一份程式碼在兩者裡都表現正確，因為是由環境——而不是程式碼——來決定用哪個資料庫和哪些金鑰。

同一份程式碼，兩個家——不同的只是那些 **env vars**：





一個專案的樣子

開啟一個典型的專案，那一長串資料夾清單可能讓人覺得像噪音。它不是——大多數專案都遵循一種認得出來的套路。下面是一個精簡過的網頁應用：

```
my-app/
├─ package.json           # 依賴 + 執行命令 (見上文)
├─ package-lock.json      # 鎖檔案 lockfile ——精確的版本
├─ .env                   # 本地的金鑰/配置 ——不要共享
├─ .gitignore             # Git 應該忽略的檔案 (比如 .env)
├─ README.md              # 這個專案是什麼，怎麼執行它
├─ public/                # 原樣提供的靜態檔案 (圖片、圖示)
├─ src/                   # 你真正的程式碼住在這裡
│  └─ components/         # 可複用的 frontend 部件 (一個按鈕、一張卡片)
│  └─ pages/ (or app/)    # 使用者導航到的那些介面/路由
│  └─ lib/                # 共享的輔助程式碼和 backend 邏輯
│  └─ styles/             # CSS
```

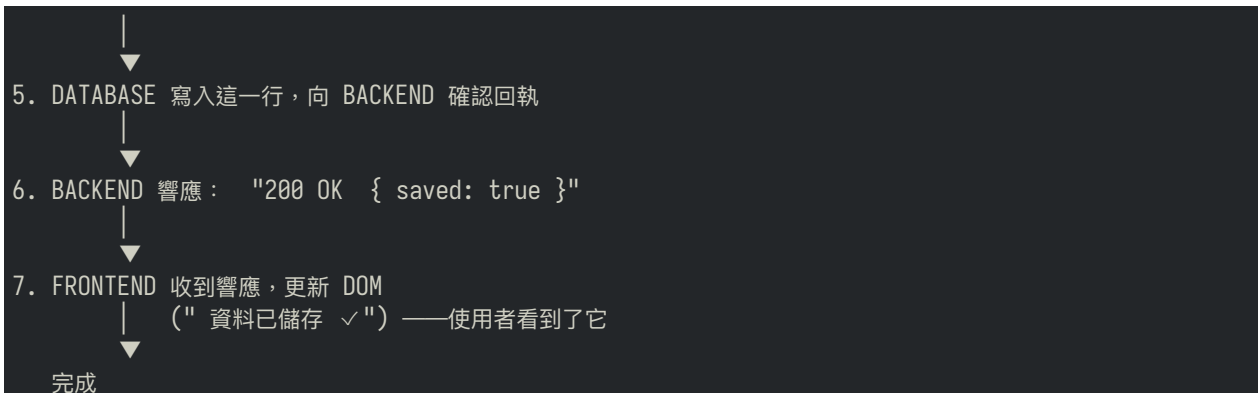
你不用把這個背下來——只要認得出來就行，這樣當 AI 說“我會把這個加到 src/lib 裡”時，你大致知道那是哪裡、為什麼在那裡。有幾條起承重作用的約定：

- src/ 裝著你寫的程式碼。最頂層那些雜亂的東西基本上都是配置。
- 根目錄下的配置檔案（那個 package.json、那些點開頭的檔案）配置的是工具，而不是你應用的行為。
- .gitignore 列出那些不應該被存進版本控制的檔案——你的 .env 就屬於這裡，而這正是金鑰得以不進入共享倉庫的辦法。

一個請求是怎麼流動的

現在我們把它們串起來。我們來跟著使用者點選“儲存資料”，走過你剛剛認識的每一個部件。這是全書最有用的一個心智模型；如果你只內化一張圖，就讓它是這張。

```
1. 使用者 點選 " 儲存資料"
   ↓
2. FRONTEND (HTML 表單, JavaScript 讀取這些值)
   | 發出一個 HTTP 請求: "POST /profile { name: 'Ada' }"
   ↓
3. HOSTING 把請求路由到你正在執行的 BACKEND
   ↓
4. BACKEND
   └─ 這個使用者登入了嗎? (沒有 → 響應 401, 停止)
   └─ 資料有效嗎? (無效 → 響應 400, 停止)
   └─ 都沒問題 → 儲存到 DATABASE
```



把每一層做了什麼走一遍：**frontend** 收集了輸入並把它發出去；**託管** 把請求送到了正確的地方；**backend** 執行了規則（登入、校驗）並做了決定；**資料庫** 把它變成永久的；然後響應一路流回來，好讓 **frontend** 能更新螢幕。每一個應用都是這個迴圈的各種變體，每秒鐘跑成千上萬次。

注意那些檢查全都發生在 **backend**，從不發生在 **frontend**——這就是第一節裡那條規則的實際運用：**frontend** 是給使用者的，**backend** 是給真相的。當你知道每一步發生在哪裡時，你就能精確地描述一個功能（“儲存之前在 **backend** 上校驗郵箱”），而不是含糊地說（“把表單弄好”）——而精確，就是整場遊戲的全部。

回顧與練習

關鍵要點

- 每個應用都是四個部分：一個 **frontend**（使用者看到的東西）、一個 **backend**（規則和邏輯）、一個**資料庫**（資料持久存放的地方）、以及**託管**（這一切執行的地方）。**frontend** 從不直接碰資料庫——它要經過 **backend**。
- 在瀏覽器裡，**HTML** 是結構，**CSS** 是樣式，**JavaScript** 是行為；程式碼通過改變 **DOM** 來改變頁面。
- 你不用什麼都自己寫——**包**（由 **npm** 通過 `package.json` 和一個鎖檔案來管理）以**依賴**的形式提供了已經解決好的問題。
- **環境變數**讓金鑰待在程式碼之外，並讓同一份程式碼在 `dev` 和 `prod` 裡都表現正確。
- 一個請求的流動是 **frontend** → **託管** → **backend** → **資料庫**再返回；說出一個問題住在哪一層，就解決了它的一半。

動手試試

挑一個你每天用的應用，大聲地把一個動作沿著各層敘述一遍：“我點了傳送（**frontend**）→ 它去到他們的伺服器（**backend**）→ 伺服器檢查我是否被允許並儲存了這條訊息（**資料庫**）→ 然後聊天介面更新了（**frontend**）。”然後猜猜下面每個問題住在哪一層：按鈕看起來沒對齊；應用說“你還沒登入”；昨天的一條訊息不見了。把這種“說出是哪一層”練熟，正是這本書剩下部分所建立的那項技能。

本章的 **prompt**

我在學應用是怎麼拼到一起的。這是我想要的一個功能：
< 用一句話描述使用者應該能夠做什麼 >

在寫任何程式碼之前，請幫我按層拆解它：

- Frontend：使用者看到什麼，以及什麼會被髮出去
- Backend：必須跑哪些規則/檢查，以及在這裡跑
- Database：需要儲存或讀取哪些資料
- 涉及到的任何 `env vars` 或配置（尤其是金鑰）

然後用編號的步驟描述從點選到結果的請求流動。

請用大白話——我想在寫程式碼之前先理解它的樣子。

Vibe Coding 心態



Vibe coding 是個含義複雜的詞。在某些人看來，它意味著敲下一行提示詞，接受模型吐出的任何東西，然後一行都不讀就直接上線。那個版本只是個玩具。它產出的演示，在真實使用者一碰之下就會分崩離析。

本書講的是另一個版本：像資深工程師指揮團隊那樣，指揮一個 AI 來構建生產級軟體。你始終掌控構建什麼以及為什麼它是正確的。你把敲鍵盤的活兒委派出去。這就是那個轉變，而它比聽起來要大得多。

Vibe coding 究竟是什麼

Vibe coding 就是用自然語言表達意圖、讓 **AI** 生成實現，與此同時你在一個緊湊的迴圈裡掌舵、審查並打磨。

它不是：

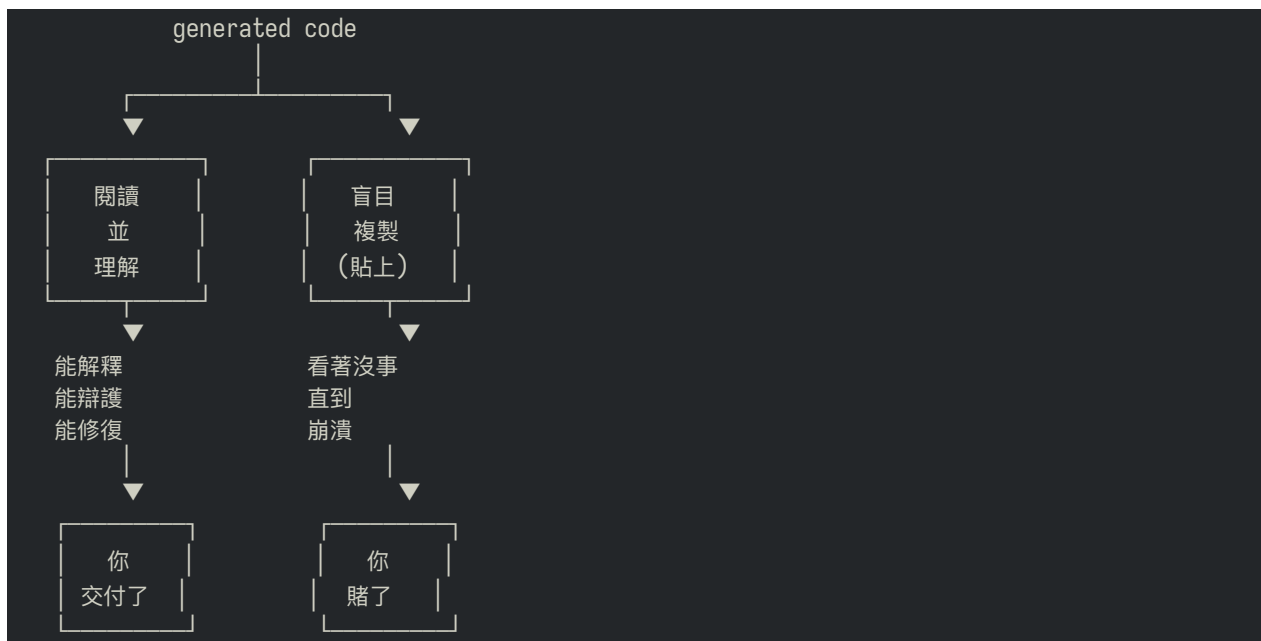
- 盲目接受你並不理解的生成程式碼。
- 用感覺代替工程判斷。
- 一個能把一句話變成上線產品的魔法按鈕。

它是：

- 把模型當作一個快速、博學、但略微過於自信的初級工程師。
- 把你的注意力花在架構、正確性和品味上，而不是語法上。
- 在過去要花幾小時的事情上，用幾分鐘完成迭代。

最有效的心智模型是：你是技術負責人，AI 是你的團隊。一個好的技術負責人不會親手寫每一行程式碼。他們決定方向、設定約束、審查 pull request（為獲得批准而提交的一批改動，即拉取請求），並在壞點子上線前把它攔下來。你正在做的就是這件事——只不過是以對話的速度。

這兩條路從同樣的生成程式碼出發，卻通向截然不同的終點：



這個定義裡有一個詞挑著最重的擔子：**理解**。整套做法都建立在一條規則之上——理解你所上線的東西。你不必親手敲它，但你必須能解釋它、能在程式碼評審裡為它辯護、能在凌晨兩點它把你呼起來時修好它。如果做不到，那你不是在做 **vibe coding**，你只是賭了一把然後運氣好。這兩者之間的鴻溝，正是整本書的主題。

心態的轉變

如果你過去習慣了親手敲下每一個字元，那麼最難的部分不是學習工具，而是放開鍵盤。

你過去的瓶頸是生產——你能多快寫出正確的程式碼。你新的瓶頸是規格說明與審查——你能多清晰地描述你想要什麼，以及你能多好地判斷返回的結果。

這改變了你最佳化的目標：

- **以前：**記住各種 API（應用程式程式設計介面——工具對外提供的命令選單）、與語法搏鬥、打字飛快。
- **現在：**精確描述意圖、拆解問題、批判性地評估輸出。

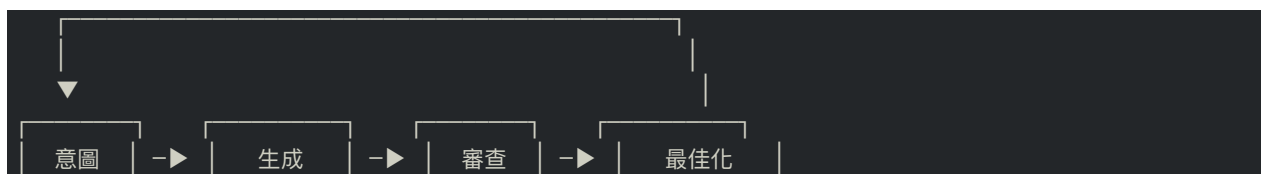
在這件事上如魚得水的工程師並不是打字最快的人。他們是那些對”好”長什麼樣有強烈見解、並有紀律去拒絕達不到標準的輸出的人。

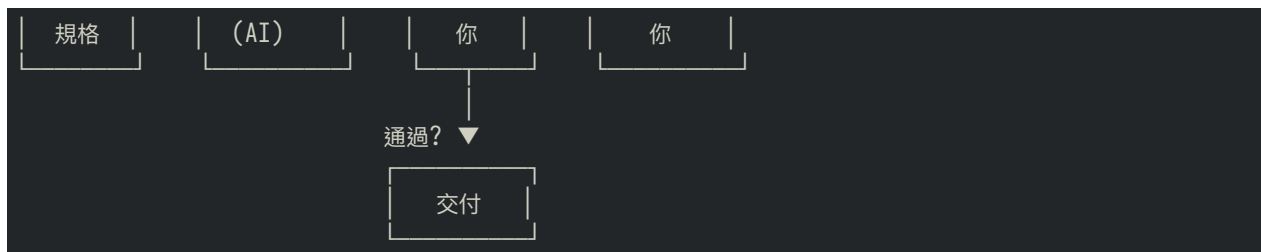
還有一個更安靜、常常讓人措手不及的轉變：你與不懂之間的關係。過去的本能是，動手之前先停下來把 API 學會。如今你往往能在一個陌生的技術棧上生成出第一版，然後通過回讀它來學習。這是真正的超能力——也是個陷阱。讀一段你本寫不出來的程式碼，沒問題。上線一段你無法評估的程式碼，不行。分界線在於你能否有把握地判斷這個輸出是對的。在”寫得比你快”這件事上儘管依賴模型；在”懂得比你願意去驗證的更多”這件事上，永遠別依賴它。

迴圈：意圖 □ 生成 □ 審查 □ 打磨

本書中的一切都回歸到一個迴圈。把它內化。

迴圈會一直轉，直到輸出達到你的標準——而審查是它每次都必須穿過的關口：





1. **意圖**。陳述你想要什麼以及關鍵的約束——輸入、輸出、邊界情況、技術棧、風格。模糊的意圖得到模糊的程式碼。
2. **生成**。讓模型寫實現。別去手把手管語法；描述行為。
3. **審查**。讀返回的內容。它真的做到那件事了嗎？它處理了空列表、null、失敗的網路呼叫嗎？它是能跑通的最簡版本嗎？
4. **打磨**。指出哪裡不對，給出具體的修正，然後重新生成。重複直到它達到你的標準。

大多數新手會跳過第 3 步。他們生成，它能跑，他們就繼續往下走。然後它在生產環境裡、在他們從未描述過的那個情況下崩掉了。審查這一步，正是工程發生的地方。

下面是這個迴圈裡一個好的初始提示詞長什麼樣：

```
寫一個 TypeScript 函式 `parseDuration(input: string): number`，  
把 "1h30m"、"45s"、"2d" 這樣的字串轉換為總秒數。
```

要求：

- 支援單位：d (天)，h (小時)，m (分鐘)，s (秒)。
- 多個單位可組合："1h30m" = 5400。
- 對無效輸入（空，未知單位，負數）丟擲帶清晰訊息的 Error 予以拒絕。
- 不用外部庫。包含覆蓋邊界情況的 5 個單元測試。

只展示函式和測試。

注意它的結構：一個精確的簽名、明確的規則、點名列出的邊界情況、陳述清楚的約束，以及對測試的要求。你不是在指望模型猜對——你是在消除那些會導致糟糕猜測的歧義。

打磨這一步和第一條提示詞一樣是門手藝。模糊的反饋得到模糊的修正。比較下面這兩種修正：

差的："這是錯的，修一下"

好的："parseDuration('1h30m') 返回 90 而不是 5400 ——你把數字
相加了，卻忽略了單位的倍率。在相加之前，把每個值乘以其
單位對應的秒數（d=86400，h=3600，m=60，s=1）。
另外為組合情況 '2d3h' 加一個測試。"

好的那條點明了症狀、原因和修法。你是在像評審同事的 pull request 那樣評審——而你指得越精確，花的回合就越少。每一圈迴圈都應當收斂。如果你發現自己在原地打轉，那是個訊號：你的意圖說明不足。停止重新生成，改去重寫規格。

它的閃光之處

Vibe coding 在以下方面真正具有變革性：

- **全新專案的腳手架搭建**。搭起一個新應用、接好路由、配置好設定——這些活兒乏味但路數成熟。
- **樣板程式碼和膠水程式碼**。CRUD 端點、表單校驗、資料轉換、兩個 API 之間的介面卡。
- **陌生領域**。一門你不太熟的語言或框架。模型懂那些慣用法；你提供判斷。
- **重構與遷移**。把這些 promise 轉成 async/await，按職責把這個 400 行的檔案拆開。
- **用完即棄的探索**。反正都要重寫的原型。速度比打磨更重要。

在這些領域，一個清晰的提示詞加上一次認真的審查，每次都勝過親手敲鍵盤。

貫穿其中的共同線索是，這些任務都**路數成熟且可驗證**。模型見過上萬個 CRUD 端點，所以它產出地道的版本；而你一眼就能看出結果對不對。當這兩點同時成立——常見的模式、檢查成本低廉——vibe coding 就近乎純粹的收益。

它失效的地方

對它的邊界要誠實，因為無人監督的 vibe coding 正是在這些地方把人坑慘：

- **深奧而新穎的邏輯**。微妙的演算法、併發、任何”對了 95% “就等於”壞了”的東西。模型即便錯了也很自信。
- **鋪得很開的跨檔案改動**。它在大型程式碼庫裡會丟去線索，悄無聲息地搞壞它看不見的東西。
- **安全和錢**。認證、支付、許可權，任何一個看似合理的 bug 會帶來真實後果的地方。審查這些時要像你的飯碗繫於此一樣，因為它可能真的是。
- **規格說明不足的問題**。如果你都不知道正確長什麼樣，AI 沒法讀你的心。規格說明是垃圾進，程式碼就是垃圾出。

失敗的模式幾乎總是一樣的：程式碼看起來對，在順利路徑上能跑，卻以一種你只要讀了就能抓住的方式錯了。解藥就是審查這一步。它無法被跳過。

兩個問題就能把任何任務放到這張地圖上——模式是否常見，以及結果是否容易核查：

	易於驗證	難以驗證
常見 模式	純賺 (腳手架, 樣板)	放慢, 要求測試
罕見 / 新穎	需讀但可行	危險區 (新邏輯, 鑑權, 金錢)

注意上面那條規則的反面：這些都是罕見、微妙、或驗證成本高的情形。模式不常見，所以模型在猜；或者 bug 在順利路徑上不可見，所以快速檢查抓不到。這並不意味著在這裡就要躲開 AI——它意味著你要放慢、把步子切小、要求測試，並把每一行都當作有敵意的來讀。

常見的坑

有幾個陷阱，幾乎會在頭一個月裡把所有人都套進去：

- **接受第一個能跑的東西**。”它能跑”和”它正確”是兩個不同的主張。前者只是說順利路徑沒崩；後者是說你檢查了邊界。別把兩者混為一談。
- **巨型提示詞**。一口氣要整個功能，會給模型太多漂移的餘地，也給你太多需要一次性審查的東西。拆解它。先上線最小的、可審查的一片，再在其上疊加。
- **任由上下文腐爛**。會話越長，模型就越會與早先的決定自相矛盾，並重新引入你已經修過的 bug。一旦出現這種苗頭，與其和一條已經糊塗的執行緒纏鬥，不如總結當前狀態、另起一條新的。
- **把自信當成訊號來信任**。模型的語氣，在它對的時候和它在幻覺的時候是一模一樣的。它的篤定什麼都說明不了。你的驗證才說明一切。
- **外包的不只是打字，連決策也外包了**。問”我在 X 上有哪些選項？”“沒問題。讓那個答案在你的判斷不在場的情況下替你選定架構，就是錯誤。

品味是新的核心技能

當任何人都能生成程式碼時，稀缺的技能就是知道哪些生成的程式碼是好的。

品味會告訴你：

- 這個抽象搞早了；去要那個更簡單的版本。
- 這個能應付演示，但應付不了真實的輸入。

- 這個技術上正確，但維護起來是場噩夢。
- 這個差不多對了——具體要改的就是這個地方。

你以老派的方式建立品味：讀大量程式碼、把東西做出來上線、然後看它怎麼壞。AI 不會取代那份經驗——它讓那份經驗更有價值，因為如今你的判斷才是瓶頸，也是區分高下的關鍵。有品味的人能快速交付穩固的軟體。沒有品味的人快速交付，然後淹沒在 bug 裡。

這裡有個令人不適的推論：AI 抬高了所有人的下限，這意味著它抹平了懂語法的優勢，磨利了有判斷力的優勢。那些靠它來跳過學習的初級工程師會觸頂停滯，因為他們從未建立起那種能在輸出錯了時告訴他們的品味。而那些用它來交付更多、閱讀更多的人則會複利增長。同樣的工具，截然相反的結果——而其中的差別，完全取決於你是否對返回的東西保持投入。

現實的預期

現在就把這些預期立好，免得你日後失望：

- 你會讀大量程式碼。審查就是這份工作本身。如果你不想讀程式碼，這並不是繞過它的捷徑。
- 它不會一次搞定任何不平凡的東西。預期要跑幾輪迴圈。這很正常，不是失敗。
- 責任仍在你身上。當 AI 寫的程式碼在生產環境崩掉時，“是模型寫的”不是藉口。是你把它上線的。
- 它讓你更快，而不是免費。這些工具非常了不起。它們不是魔法。判斷始終是你的。

再直白地說一句本書不是什麼：它不是一條讓你上線自己都不理解的軟體的路，也不是一夜暴富的捷徑。任何把 **vibe coding** 當作“不學程式設計也能做出 App”來兜售的人，賣的都是那個玩具版本——一碰到真實使用者就分崩離析的那個。這門技能只會對那些把它當作疊加在工程判斷之上的槓桿、而非判斷的替代品的人複利增長。

Vibe coding 的承諾是真實的：你能構建得更多、更快、跨越比你獨自一人更多的領域。但它獎勵那個保持投入的工程師——那個去指揮、審查、打磨的人——並懲罰那個抽身離場、接受任何出現之物的人。

去當技術負責人，而不是旁觀者。本書餘下的內容，講的就是如何做到。

回顧與練習

核心要點

- **Vibe coding** 是在你掌控做什麼和為什麼的前提下，指揮 AI 構建軟體——而不是盲目交付你解釋不了的程式碼。
- 把這一切串起來的規則是理解你交付的東西：你不必親手敲它，但你必須能為它辯護、能修復它。
- 你的瓶頸從生產（敲得快）轉移到了規格說明與審查（描述清楚、批判性地判斷）。
- 活在迴圈裡——意圖 □ 生成 □ 審查 □ 打磨——並且永遠不要跳過審查這一步，真正的工程發生在那裡。
- 模型的自信不是訊號；你的驗證才是。品味與判斷力才是稀缺、且會複利累積的技能。

動手試試

挑一個你真正理解的小函式——比如“校驗密碼是否符合策略”或“把位元組數格式化成人類可讀的字串”。為它寫一個精確的提示詞（簽名、規則、邊界情況、要求附測試），生成它，然後刻意執行審查這一步：親手追蹤至少一個邊界情況，確認程式碼能處理。如果不能，就寫一個精確的打磨提示詞（症狀 + 原因 + 修復），而不是“修一下”。目標是在你能完全評估的程式碼上，把整個迴圈從頭到尾完整地感受一次。

本章提示詞

作為我的結對程式設計夥伴。我會描述一個小而自包含的函式。在寫任何程式碼之前，先複述需求並列出你將處理的邊界情況，好讓我確認。然後寫出該函式，以及覆蓋這些邊界情況的單元測試。

函式：< 一行描述 >

輸入 / 輸出：< 型別與形態 >

約束：< 技術棧，要用或要避免的庫，效能限制 >

程式碼之後，指出你做出的、我應當核實的任何假設。
保持為滿足需求的最簡單版本。

搭建你的 AI 工作環境



Vibe coding 是一種合作。你帶來意圖和判斷；AI 帶來速度和不知疲倦的打字。但和任何合作一樣，它在共享的環境、清晰的基本規則、以及一個允許犯錯的安全之地中運作得最好。本章講的就是在你寫下第一個功能之前，先搭好那個環境。

一個好的工作環境做三件事：它給 AI 做出聰明改動所需的上下文，它讓你快速看到這些改動的結果，它讓每一次改動都易於撤銷。把這些做對，AI 感覺就像一位資深合作者。做錯了，它感覺就像一臺老虎機。這個差別很少是因為模型——而是因為你把它放進了什麼樣的房間。一個能幹的智慧體，放進一個沒有測試、沒有上下文檔案、沒有版本控制的混亂倉庫裡，會手忙腳亂；一個平平無奇的智慧體，放進一個乾淨、儀表齊全的專案裡，會讓你刮目相看。配置才是那個乘數。

這些都不是做一次就忘的雜活。把你的工作環境當作一件需要持續調校的東西：每當 AI 以糟糕的方式讓你吃驚時，問問是不是某塊缺失的上下文、某個緩慢的反饋迴圈、或某個不安全的預設值放任了它——然後去修房間，而不只是修那一行程式碼。

選擇一個 AI 原生的編輯器或助手

你的第一個決定是 AI 住在哪裡。如今大致有三個類別，而你不必永遠繫結。

- **AI 原生編輯器** (Cursor、Windsurf)：一個熟悉的程式碼編輯器，把 AI 深度接入其中。如果你想在 AI 協同工作時看到並觸碰程式碼，這是最佳選擇。
- **終端/CLI 智慧體** (Claude Code、Codex CLI、Gemini CLI、Aider)：AI 作為一個命令列智慧體執行，讀取檔案、編輯檔案並執行命令。這裡的 CLI (命令列介面) 就是用輸入命令代替點選按鈕的純文字控制台。最適合讓 AI 跨多個檔案更自主地工作。
- **IDE 外掛** (GitHub Copilot、Continue、JetBrains AI)：在你已經鍾愛的編輯器裡附加上去的輔助。IDE (整合開發環境) 就是大多數程式設計師寫程式碼所用的一體化應用。

要交付真正的軟體，挑一個能讀取你的整個專案、編輯多個檔案、並自行執行命令（測試、構建、linter）的工具。能執行並檢查自己工作成果的能力，正是把一個有用的助手和一個自動補全玩具區分開來的東西。一個只能建議下一行的工具，會把所有驗證都留給你手動做；一個能跑它剛改過的那個測試的工具，會自己把迴圈閉合。

在這個核心能力之外，定下來之前還有幾樣東西值得權衡：

- **上下文視窗與專案感知**——它能否真正載入相關檔案，還是在一個任務做到一半時就忘了對話的開頭？
- **許可權模型**——它在執行命令和編輯檔案之前會先問你，還是先動手？兩種都有用武之地；要知道自己處在哪種模式。
- **成本與額度**——自主智慧體能很快燒掉用量。在你放它跑一個小時之前，先弄清定價。

別在選擇上糾結；本書中的技能在工具之間是通用的。多數有經驗的 **vibe coder** 最後都會裝兩個——一個用於批次工作的快速終端智慧體，一個用於審查的 GUI 編輯器（GUI 即圖形使用者介面，就是那種帶視窗和按鈕、靠點選操作的熟悉程式）——而本章收尾的 TUI vs GUI 一節，正會回到這對組合上。這裡的 TUI（文本使用者介面）則是它在終端裡執行的純文字版本。

把專案結構組織得讓 AI 能導航

AI 模型靠讀你的程式碼來推理。一個混亂、鋪得很開、不一致的專案，對它們來說和對你來說一樣難。你不需要完美的架構，但有幾個習慣能立刻見效：

- **保持可預測的佈局**。原始碼放一個地方，測試挨著原始碼或映象原始碼的結構，配置放在根目錄。
- **使用清晰、描述性的命名**。calculateMonthlyInvoiceTotal 勝過 calc2。AI 把名字當作線索。
- **偏好更小的檔案**。對你們倆來說，一個 200 行的檔案都比一個 2000 行的更容易正確地修改。
- **把相關的東西放在一起**。當一個功能的程式碼、樣式和測試坐在一處時，AI 一次讀取就能收集到上下文。
- **保留一個清晰的入口點**。應用啟動的那個明顯的單一位置，給了 AI 一根在梳理專案時可以順著拉的線。

一個典型的、對 AI 友好的佈局可能長這樣：

```
my-app/
├── AGENTS.md           # 給 AI 的專案規則 & 上下文
├── README.md           # 專案簡介、如何執行
├── .env.example         # 記錄所需金鑰（無真實值）
├── .gitignore          # 排除 .env、構建產物、node_modules
├── package.json         # 指令碼: dev, test, lint, build
├── src/
│   ├── features/
│   │   └── invoices/    # 單個功能的程式碼 + 測試，放一起
│   ├── lib/             # 共享輔助函式
│   └── index.ts          # 入口點
└── tests/               # 橫切 / 整合測試
```

回報會複利式增長：你的專案越容易讓 AI 導航，它的改動就越小、越精準。一個井井有條的倉庫，讓 AI 去碰三個相關檔案，而不是在三十個檔案之間瞎猜——這意味著更小的 diff、更少事故，以及你真能讀得下去的審查。如果你發現 AI 在做大刀闊斧、漫無焦點的編輯，真正的罪魁禍首往往是結構，而不是提示詞。

寫一個上下文檔案（AGENTS.md）

你能做的槓桿率最高的一件事，就是寫一個 AI 會自動讀取的上下文檔案。不同的工具尋找不同的名字——AGENTS.md、CLAUDE.md、.cursorrules，或者 .github/copilot-instructions.md——但理念是相同的：一份活的文件，告訴 AI 你的專案是如何運作的，以及你希望它如何行事。

沒有它，AI 靠猜。有了它，AI 不再重新格式化你的程式碼，不再發明你不用的庫，不再重新解釋你早已做過的決定。把它想成給一位新隊友的人職文件——這位隊友很聰明，但在每次會話之間會完全失憶——他要變得高產所需的一切，都得放在他每次都會讀的地方。

一個實用的起步檔案：

```
# 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.
```

讓它保持簡短，也保持最新。一個膨脹到一千行的上下文檔案，就不會被認真讀了——無論是你還是模型——所以刪起來要和加起來一樣狠。當 AI 反覆犯同一個錯誤的假設時，那就是你該加一行的訊號；當一條規則不再符合現即時，在它開始主動誤導智慧體之前把它刪掉。

有幾個模式，把一個有效的上下文檔案和一個被無視的區分開來：

- 寫得具體而明確。”用我們的約定”毫無用處。“校驗用 zod；我們不用 yup”才是一條 AI 真能照做的規則。
- 把”別做”說得和”要做”一樣響亮。模型急於幫忙，這意味著除非你叫它別這麼幹，它會樂呵呵地加個依賴或重寫一份配置。
- 指向示例。”照著 src/features/invoices 的風格來”給了 AI 一個可以照抄的具體範本，而不是一條要去揣摩的模糊指令。

把上下文檔案當作一份記錄，記下你本來每次都得重複一遍的每一條教訓。它是這整章裡最廉價、最持久的投資。

讓版本控制成為本能反應

版本控制是你的安全網，而有了 AI 它變得不可或缺。AI 能在五秒內改十個檔案——你需要一種乾淨的方式去檢查、保留或丟棄那份工作。

- 頻繁地、小塊地提交。在你要求一個大改動之前，先把能跑的提交了。一棵乾淨的工作樹意味著你總能回到那裡。
- 提交前讀 *diff*。這才是你真正的審查步驟。你不必理解每一行，但你應該理解改了什麼、為什麼改。
- 用分支來做有風險的工作。開一個分支，讓 AI 放手去幹，只在你滿意結果時才合併。如果不滿意，刪掉它——毫髮無傷。
- 寫誠實的提交資訊。未來的你和 AI 都會讀歷史來理解意圖。

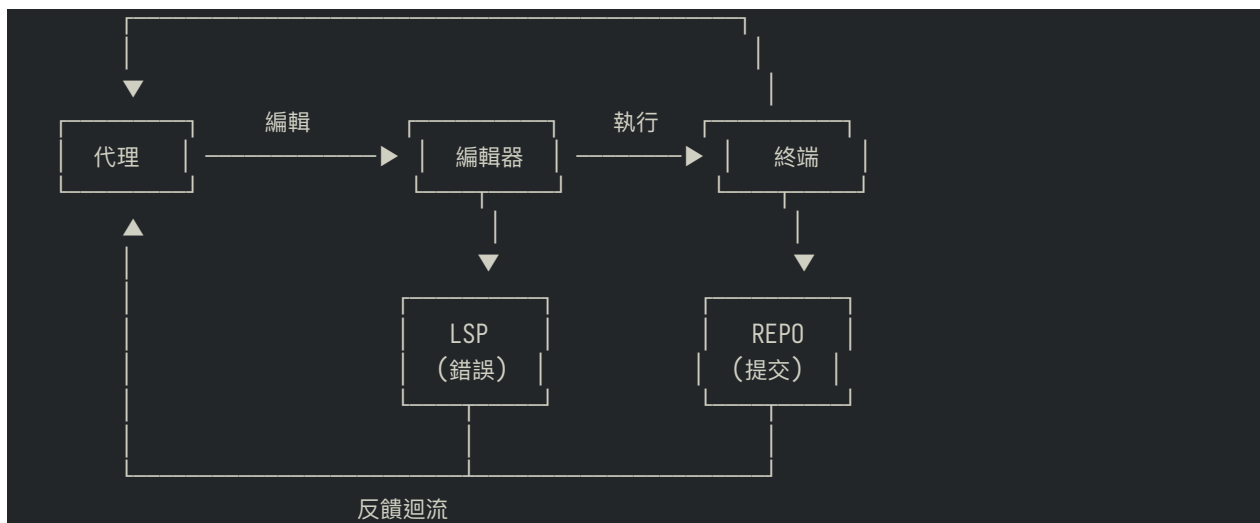
心態上的轉變：提交不是官僚主義，它們是遊戲裡的存檔點。在打 Boss 前先存檔。再學會那兩個讓乾淨歷史值得維護的逃生口——git checkout . (或 git restore .) 扔掉未提交的實驗，git revert 撤銷一個你已經做下的提交。知道自己永遠能回到一個已知良好的狀態，正是這一點讓你能給 AI 一條長長的韁繩而不焦慮。

讓反饋迴圈保持緊湊

你越快能看到一個改動是否有效，你和 AI 就越快收斂到正確的東西上。一個緩慢的迴圈——你要等上幾分鐘才知道有沒有搞壞什麼——會悄無聲息地把每一個錯誤成倍放大。

- 讓“跑起來”變成一條命令。`npm run dev`、`make test`，無論是什麼，把它記在你的上下文檔案裡，這樣 AI 也能跑它。
- 依靠快速的測試。哪怕只有少數幾個單元測試，都能給 AI 一種驗證自己工作的方法，而不用你盯著。
- 使用 *linter* 和格式化工具。它們能瞬間抓住一整類錯誤，並把風格之爭擋在你的審查之外。
- 把錯誤回傳給 AI。當出問題時，把真實的錯誤資訊貼回去。具體的反饋產出具體的修復；“它不工作”只會產出猜測。

AI 工作空間的各個部件構成一個相連的迴圈——智慧體編輯倉庫，編輯器和語言伺服器報告問題，終端執行程式碼，而每一個結果都流回智慧體：



這個想法最深的版本，是讓 AI 自己把迴圈閉合。與其由你來執行命令、讀取失敗、再轉述回去，不如給智慧體許可權，讓它在每次改動後執行測試和型別檢查器，並對自己看到的東西做出反應。迴圈於是變成：編輯、執行、讀取失敗、修復、再執行——全程沒有你卡在中間。你的角色從信使轉為導演：你設定目標和護欄，然後看著迴圈轉起來。當 AI 能夠編輯、執行、看到失敗、並自行修復時，你就建好了一個能幹真正活兒的迴圈，而你穩坐導演椅。

謹慎管理機密

AI 工具會讀你的檔案，有時還會把上下文分享給遠端服務。這讓洩露機密成為一個真實的風險。從第一天起就建立好習慣：

- 永遠別把真實的金鑰放進程式碼。使用環境變數和一個 `.env` 檔案。
- 永遠把你的 `.env` 加進 *gitignore*。提交一個帶空佔位符的 `.env.example`，這樣 AI 不用看到值就知道需要什麼。
- 把金鑰的許可權範圍收窄，並輪換任何可能已被暴露的金鑰。
- 在你的上下文檔案裡告訴 AI 永遠不要硬編碼或提交機密。

在你的 `AGENTS.md` 裡放一條響亮的規則，比如 `Never read or print the contents of .env`，是一份廉價的保險。一個會拒絕包含疑似金鑰的提交的 `pre-commit` 鉤子或金鑰掃描工具也是——洩露的代價是不對稱的，所以在這裡縱深防禦很重要。一個被推到公開倉庫的金鑰，在它落地的那一刻就已經洩露了，哪怕你幾秒後就刪掉，因為機器人會即時抓取提交。拿不準時，就把暴露的金鑰當成已經廢了，去輪換它；輪換一個金鑰只要一分鐘，而給一個洩露的金鑰善後可能要花上好幾天。

一個健全的本地配置讓 AI 的改動安全

用一個讓錯誤既廉價又可恢復的環境把這一切串起來：

- 在分支上工作，而不是直接在 `main` 上。
- 在大改動之前保持工作樹乾淨，好讓 `diff` 有意義。

- 讓測試和開發伺服器隨時能用一條命令跑起來。
- 把機密留在倉庫之外，並讓上下文檔案保持最新。

有了這個地基，任何 AI 改動的最壞情況就是 `git checkout`，加上一個全新的開始。正是那份安全，讓你能快速行動、大膽指揮——知道 AI 所做的沒有任何事是真正不可逆的。在下一章，我們將開始用這個工作環境來真正動手構建。

即時診斷 (LSP)

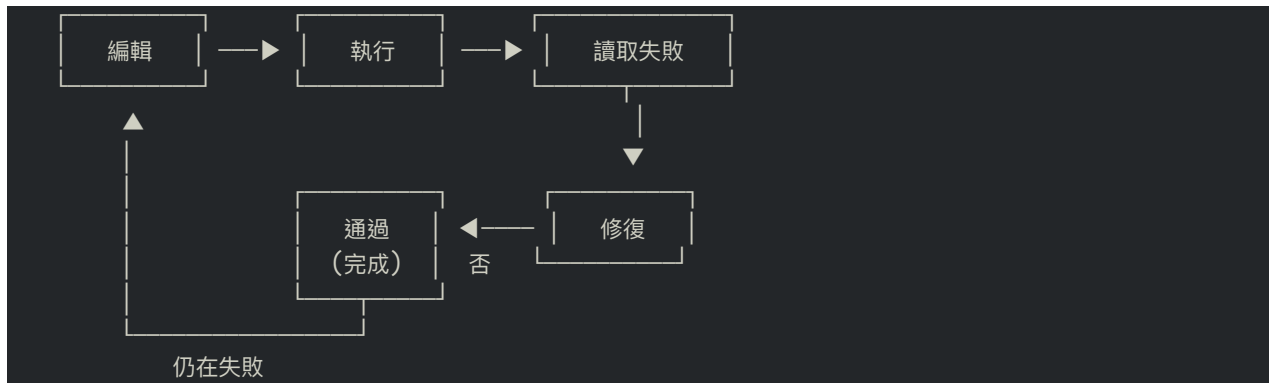
最緊湊的反饋迴圈不會等你去執行程式碼——它會在你剛敲下錯誤的那一刻就抓住它。這正是語言伺服器協議 (LSP) 所提供的：你編輯器背後的即時“智慧”，包括即時的錯誤、型別資訊、跳轉到定義和自動補全。語言伺服器會在你的程式碼發生變化時讀取它，並在任何東西執行之前就報告問題。

對 *vibe coding* 而言，這裡的槓桿在於把這些診斷回傳給 AI。沒有它，迴圈很慢：AI 寫程式碼，你執行它，它因為一個拼寫錯誤或型別錯誤而崩潰，然後你手動把錯誤資訊貼回去。把診斷接進來以後，智慧體會在它寫下那行程式碼的瞬間，看到和你一樣的紅色波浪線——並隨手修掉型別或 lint 錯誤，根本不用等程式碼執行。

你可以用幾種方式給智慧體這雙眼睛：

- 為你的技術棧執行語言伺服器 (tsserver、rust-analyzer、pyright 等等) ——大多數 AI 原生編輯器會替你做好這件事。
- 讓智慧體執行一個型別檢查或 lint 命令並讀取輸出，或者通過一個 MCP 伺服器暴露診斷資訊。
- 把一份乾淨的診斷報告當作“完成”的一部分，而不是事後才想起來的東西。

這樣接好線，智慧體就能自己跑一個自我糾正的迴圈——編輯、執行、讀取失敗、修復，再執行，直到診斷乾淨地返回：



這也是為什麼一門帶型別的语言，對 *vibe coding* 來說可以是一種安靜的超能力：每一個型別註解都是一份小小的、由機器檢查的規格，會在 AI 的錯誤執行起來之前就把它逮住。一個“執行—崩潰—貼上”的迴圈，於是變成了一個安靜的、自我糾正的迴圈。AI 少做無用功，收斂得更快。

擴充套件你的智慧體

開箱即用時，AI 助手是通用的。一旦你在它周圍裹上一層工具，同一個基礎模型就會變成一個好得多的協作者——把一個普通的聊天機器人，變成一套為你的專案調校過的外殼 (harness)。要知道的主要擴充套件點有：

- **技能 (Skills)** ——可複用的打包能力（指令加指令碼），智慧體按需載入，這樣你的領域知識無需每次會話都重新解釋一遍就能用上。
- **外掛 (Plugins)** ——把技能、命令、鉤子和聯結器打包在一起、作為一個可安裝單元釋出的捆綁包。
- **MCP (模型上下文協議)** ——一個開放標準，通過一個通用介面把智慧體連線到外部工具和資料（資料庫、API（應用程式程式設計介面——其他軟體對外提供的標準命令選單）、你的檔案系統）。
- **子智慧體 (Subagents)** ——主智慧體為專注的工作（搜尋、審查、一段長長的支線任務）派生出來的專門助手智慧體，每個都有自己的上下文視窗。
- **鉤子 (Hooks)** ——在生命週期事件（一個命令執行之前、一個檔案被編輯之後）自動觸發的指令碼，讓你能夠確定性地強制規則或注入上下文。
- **斜槓命令 (Slash commands)** ——你用 `/name` 呼叫的、有名字的短工作流，把一個重複的多步請求收攏成一次敲鍵。

你不需要在第一天就用上所有這些。先從樸素的開始，一旦你發現自己在重複同樣的設定，就加上一個技能或一個斜槓命令。一條有用的經驗法則：當你第三次敲下同一段多步指令時，把它變成一個斜槓命令；當你第一次希望智慧體能夠到一件它夠不到的工具時，去找一個對應的 MCP 伺服器。鉤子尤其會在護欄那一章再次出現，作為一種強制執行你的規則的方式，而不是隻盼著 AI 還記得。

macOS vs Windows

大多數 AI 工具——CLI 智慧體、指令碼、教程——都悄悄地假設你用的是 Unix shell。這左右了開發環境的實際選擇。

- **macOS** 是 **Unix** 原生的。它自帶一個真正的 POSIX shell，以及和開發界其餘人一樣的 git、ssh 和 curl。大多數智慧體和指令碼“開箱即用”。
- **Windows** 完全有能力勝任，但最順滑的路子是 **WSL2** (Windows Subsystem for Linux)：Windows 裡面一個真正的 Linux 環境。把你的 AI 程式設計放在 WSL2 裡面做，而不是去跟原生 PowerShell 的怪癖較勁。
- **Linux** 是這個家族的第三位成員，就這些用途而言它的行為幾乎和 macOS 一模一樣——它就是 Unix，所以同樣的智慧體和指令碼原封不動地跑。如果你已經在它上面，那你就齊活了。

幾個讓你的工作在它們之間保持可移植的坑：

- **路徑**——/home/you/project 對比 C:\Users\you\project；反斜槓和磁碟機代號會讓指令碼崩掉。WSL2 繞開了其中大部分。
- **行尾**——Windows 用 CRLF，Unix 用 LF。不匹配會造成嘈雜的 diff 和壞掉的 shell 指令碼；設定 core.autocrlf 和一個 .gitattributes 來統一歸一化成 LF。
- **大小寫敏感**——macOS 和 Windows 往往把 File.ts 和 file.ts 當作同一個；Linux 和 CI 不會，所以在本地能構建的程式碼可能在雲上失敗。在匯入和檔名裡對大小寫保持一致。
- **許可權**——Unix 的檔案模式 (chmod +x) 沒法乾淨地對映到 Windows 上。

無論你用哪個，都讓 AI 把命令寫得可移植——正斜槓、LF 行尾、一致的大小寫——這樣它的指令碼在你的筆記本上和 CI 裡跑起來都一樣。當 AI 遞給你一條假設了某個你沒裝的工具的單行命令時，最快的修法往往是在上下文檔案裡告訴它你的作業系統和 shell，好讓它別再瞎猜。

TUI vs GUI

AI 程式設計工具有兩種形態，而一個高產的 vibe coder 兩種都會拿來用。**終端 / TUI 智慧體** (Claude Code、Codex CLI、Aider) 跑在終端裡：快、可指令碼化、可組合，非常適合自動化以及讓 AI 跨許多檔案自主工作。**GUI / 編輯器工具** (Cursor、Windsurf、IDE 擴充套件) 活在一個視覺化編輯器裡：易於發現，有內聯 diff、點選接受，以及一個平緩的上手坡道——審查改動更容易。

	TUI / 終端	GUI / 編輯器
速度	非常快，開銷低	較慢，UI 更多
可指令碼化	可以——管道、串聯、自動化	有限
視覺化 diff	純文本	豐富，內聯
可發現性	低（要知道命令）	高（可點選）
最適合	自動化、自主、高階使用者	審查、上手、視覺化工作

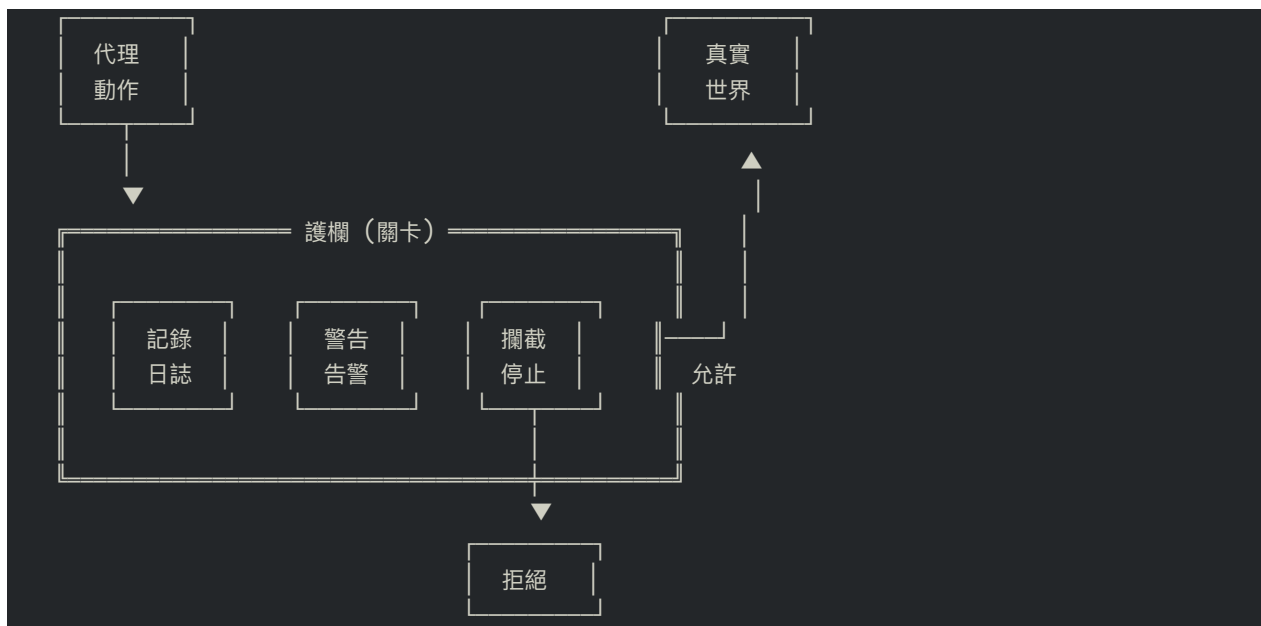
你不必永遠只挑一個。組合才是關鍵所在：用一個快速的 TUI 智慧體來推動大部分工作，然後切進一個 GUI 做視覺化審查或細粒度的編輯。一個常見的工作流是，讓一個終端智慧體跨許多檔案做出大範圍的改動，然後開啟 GUI 的左右對照 diff 去真正讀一遍、批准它做的事——幹活靠速度，核對靠清晰。挑一個用著舒服的當主力，並記住這本書裡的技能在兩者之間是通用的。

護具工程 (Harness Engineering)

習慣是你必須記得去做的事。在搭建工作空間時，下一個臺階是把這些檢查變成 AI 無法跳過的東西。那就是護具 (harness)。

把護具想成一個立在智慧體與真實世界之間的安檢口。在每一個危險的步驟，它會做三件事之一：記錄發生了什麼、發出警告，或者乾脆攔截這個操作。在命令執行之前、在檔案被寫入之前、在提示被提交時、在提交（commit）時、在推送（push）時——都有一道閘門。你可能會在凌晨兩點忘了跑測試，但智慧體繞不過它。

智慧體採取的每一個危險操作都會先通過護具，每一道閘門都能在它觸及真實世界之前記錄它、警告你，或者攔截它：



為什麼說它是必須的？氛圍程式設計的根本賭注是“讓 AI 全力衝刺，你來掌舵”。這隻有在你眨眼的瞬間 AI 不可能衝下懸崖時才成立。護具把安全從警惕（讀每一個 diff、記住每一條規則）轉移到程式碼（自動強制執行的規則）。對任何要擺到真實使用者面前的東西來說，這不是可有可無的打磨——它正是讓自主變得安全的那個東西。

一個好的護具遵循幾條值得了解的原則：

- 成功時安靜，失敗時響亮——一切正常時它什麼都不說，所以一條警告才仍然有分量。
- 絕不自動修復——它提議、攔截或警告；它不會偷偷“糾正”智慧體從而把問題藏起來。
- 新增新規則前，先退役休眠的規則——保持精簡且可信，而不是變成一堆陳舊檢查的堆積。
- 由配置驅動——一個引擎，按專案配置，能插入任何程式碼庫。
- 只追加（*append-only*）的日誌——一份記錄智慧體實際做了什麼、且能識別篡改的賬本。

你不必從零造這套東西。已經有現成的、即插即用的護具——[dancinlab/harness](#) 就是一個與專案無關的引擎，可以接到編碼智慧體的工具生命週期上。說句實話：護具並不能讓 AI 變得正確，它只是讓 AI 的錯誤變得便宜可抓——當你跑得很快時，這就是這場仗的大半。

成本安全：CLI vs API

一張賬單就能毀掉一個專案，而它很少來自託管。它來自一個按量計費的 AI API。你為日常寫程式碼而搭建好的工具——Claude Code、Cursor 之類——通常是按固定費率訂閱的：多發一條 prompt 的成本實際上為零。可一旦你的應用去呼叫原始的 AI API 或 SDK（SDK 即軟體開發工具包，是服務商提供的、預先打包好的程式碼庫），經濟模型就翻轉了：你是按 *token*、按請求計費的，沒有上限。一個有 bug 的重試、一個無界限的批處理任務，或一個陷入迴圈的 agent，都能在一夜之間悄悄地把 \$0 的閒置賬單變成一筆四位數的意外。這就是“bill shock”，也是單打獨鬥的構建者最常見的栽跟頭方式。

在任何付費 API 上線之前，先給它套上籠子：

- 在提供商的控制台裡設定一個硬性消費上限——一個真正的天花板，而不只是一封郵件提醒。
- 開發時優先用按固定費率的 **CLI**；把按量計費的 API 留給那個真正需要它的生產功能。
- 對你自己的呼叫做限流——限制併發數和每分鐘請求數——並在頭幾天盯緊用量面板。
- 批次工作用便宜/小型的模型，並先在極小的輸入上測試（一條記錄，而不是整張表）。

- 絕不上線一個呼叫付費 **API** 的、不帶節流的迴圈。給每個迴圈都設上界。

從一開始就把它定為規矩：每一個付費 **API** 都有一個硬性上限和一個節流器。

回顧與練習

核心要點

- 一個好的工作空間給 AI 上下文、快速的反饋迴圈和廉價的撤銷——配置才是那個乘數，往往比模型本身更關鍵。
- 上下文檔案 (AGENTS.md / CLAUDE.md) 是你寫下的最高槓杆的東西：具體的規則、大聲的禁止事項，以及指向示例的指標。
- 讓版本控制成為本能——小步提交、讀 diff、危險的活兒開分支——這樣 AI 做的任何事都不會真正不可逆。
- 讓迴圈保持緊湊，並讓 agent 自己閉環：編輯、執行、讀失敗、修復。把診斷 (LSP、lint、型別檢查) 反饋給 AI。
- 用硬性上限和節流器把每一個付費 API 關起來，把金鑰留在倉庫之外，並用 harness 讓安全檢查成為 AI 跳不過去的東西。

動手試試

開啟你正在做的專案（或腳手架一個空的），親手寫一份 AGENTS.md 初稿：這個專案是什麼、技術棧、執行 dev/test/lint 的確切命令、三條真實的約定，以及至少兩條大聲的禁止事項（例如“絕不讀取或列印 .env”“不先打招呼不要加依賴”）。然後讓你的 AI 工具做一個微不足道的改動，看它是否尊重這份檔案。每當它做了你不想要的事，就加一行或把一行寫得更鋒利。你在調的是房間，而不只是程式碼。

本章提示詞

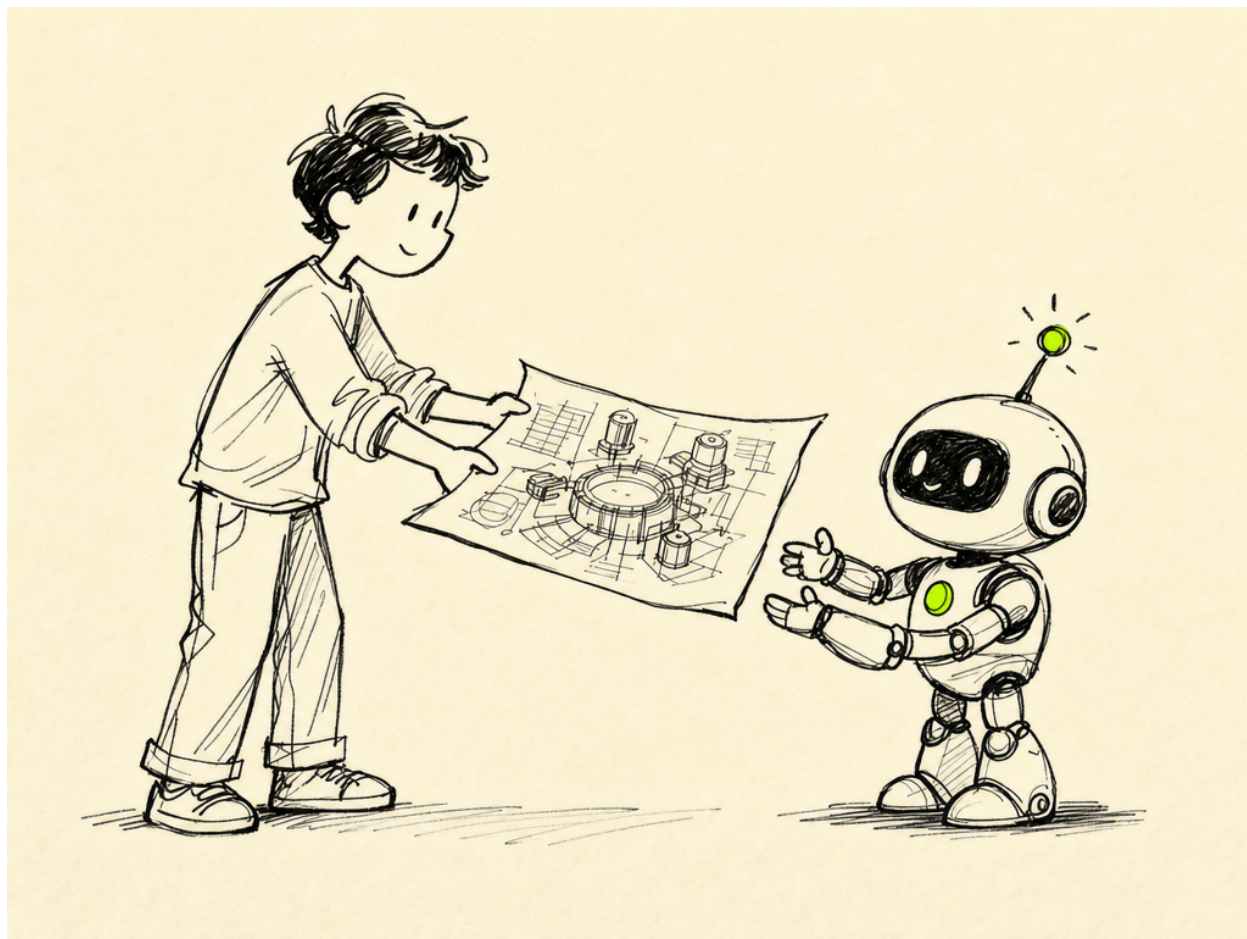
```
Read my project and help me write a concise AGENTS.md context file for it.
```

```
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.
```

像工程師一樣寫提示詞



一次令人沮喪的 AI 會話和一次高產的會話之間的區別，很少在於模型。它在於提示詞。大多數新手把 AI 當成搜尋引擎——他們敲下一個模糊的願望，然後聽天由命。資深工程師把它當成一個犀利但死板的初級隊友：一個又快又能幹的人，但他會精確地構建出你描述的東西，包括你的錯誤在內。

本章講的是彌合那道鴻溝。你不需要記住什麼技巧，也不需要收藏一整個資料夾的「魔法」提示詞模板。你需要像一個好工程師交代任務那樣去溝通：帶著上下文、約束，以及一個清晰的「完成」的定義。下面的一切都是這同一項技能的變奏。

在給出任務之前先給出上下文

AI 沒法像你那樣看到你的專案。它不知道你的框架版本、你的檔案結構，或者你早已確立的慣例。當你跳過這些時，它就靠猜——而且猜得很糟。更糟的是，它自信地猜。所以在它撞上你真實的程式碼庫之前，輸出看起來都挺像樣。

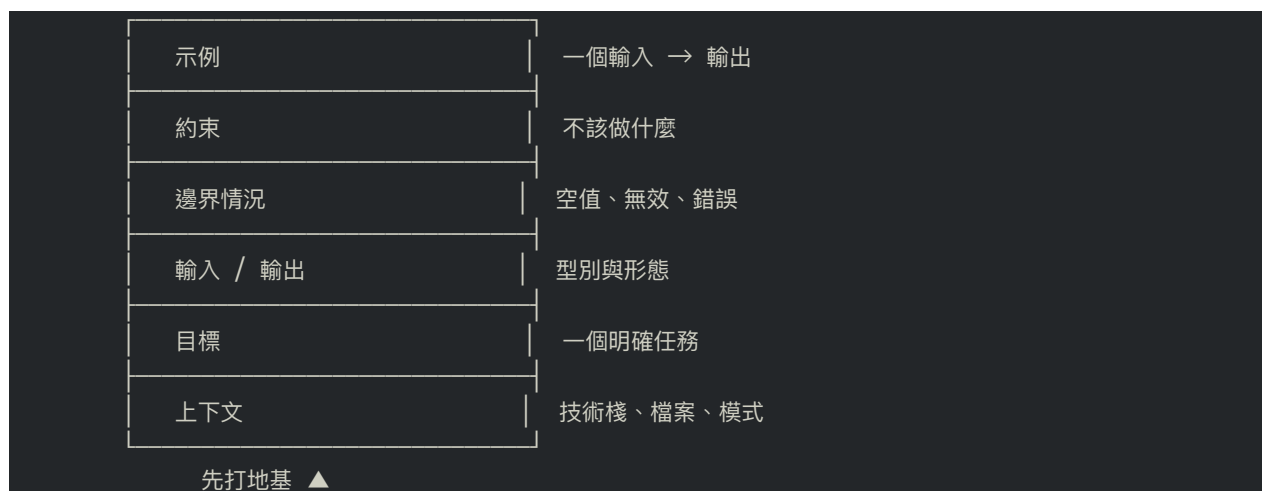
對比同一個請求的這兩個提示詞。

新增一個校驗郵箱的函式。

這是一個 TypeScript 後端，使用 Express，並用 Zod 做校驗。
我們已經在 `src/schemas/` 裡用 Zod schema 校驗輸入。
按那種風格新增一個郵箱校驗 schema。郵箱必須是
小寫、最多 254 個字元，並拒絕來自 `src/config.ts` 裡
現有 `BLOCKED_DOMAINS` 列表的一次性域名。

第一個提示詞產出泛泛的程式碼，它可能用了一個你不想要的正則，風格也和你的程式碼庫不匹配。第二個產出的程式碼你可以直接貼上進去。規則是：陳述技術棧、慣例，以及 AI 應該複用的現有部件。

一個好的提示詞把這些部件自下而上按順序堆疊——上下文在底層，真正的請求靠近頂層：



在你敲下回車之前，一份有用的心理清單：語言和框架（如果版本要緊就連版本一起），相關的檔案或模組，專案裡已有的模式，以及 AI 該偏好或該避開的庫。不必每次都湊齊這四樣——但如果答案會因其中之一而改變，就把它點出來。拿不準時，把新程式碼要適配的那個真實簽名、型別或配置直接粘上去。幾行現有程式碼教給 AI 的慣例，遠勝過一整段描述。

指明輸入、輸出和邊界情況

一個任務在它的邊界被定義之前都不算被定義。資深工程師以“什麼進去、什麼出來、出錯時會發生什麼”的方式思考。把這些烤進提示詞裡。

寫一個解析日期字串的函式。

寫一個函式 `parseDate(input: string): Date | null`。

輸入：ISO 8601 字串 ("2026-06-14"、"2026-06-14T10:30:00Z")。

輸出：一個 `Date` 物件，若字串無效則返回 `null`。

需要處理的邊界情況：

- 空字串或僅有空白 -> `null`
- 格式合法但日期不可能 (2026-02-30) -> `null`
- 末尾有垃圾字元 ("2026-06-14xyz") -> `null`

不要 `throw`。對所有無效輸入都返回 `null`。

當你點名說出邊界情況時，AI 就會處理它們。當你不說時，它寫順利路徑，然後你在生產環境裡發現那些缺口。同樣的紀律也適用於純函式之外：對一個端點，要指明成功響應以及錯誤響應（重複的郵箱返回什麼狀態碼？請求體格式錯誤時呢？）。對一個 UI 元件，要指明空狀態和載入狀態，而不只是有資料的狀態。大多數軟體有意思的行為都住在它的邊緣，所以你的提示詞也應當在那裡花筆墨。

如果你不確定邊界情況有哪些，就先問 AI：“一個日期解析器應該處理哪些邊界情況？”——然後把答案當作約束喂回去。這把模型從一個程式碼生成器變成了一個清單生成器，而審查一份清單，比除錯你忘掉的那些情況要快。

以小步前進

最常見的單一錯誤，就是一次要求太多。“給我搭一個帶認證、圖表和設定頁的使用者儀表盤”會產出一堵你無法審查的程式碼牆，bug 散落在你沒讀過的檔案裡。

小步前進讓你保持掌控：

- 每個提示詞只交付一個函式、元件或端點。

- 在往下走之前先驗證每一塊都能用。
- 在確認能用的程式碼之上構建，而不是在一座未經驗證的輸出之塔上。

把整個認證系統搭出來，包括登入、註冊、密碼重置、會話，以及郵箱驗證。

認證流程的第 1 步：只寫註冊端點。
POST /signup 接收 { email, password }。用 bcrypt 對密碼做雜湊，儲存使用者，返回 201 並帶上使用者 id。
假設 User 模型和 db 客戶端已經存在（我會給你看看它們的形態）。登入我們下一步再處理。

更小的提示詞意味著更小的 diff，而更小的 diff 意味著你真的能讀懂改了什麼。這就是全部的遊戲所在。還有第二份回報：當某處壞掉時，一個小步驟做二分定位易如反掌。如果你生成並驗證了註冊端點，然後是會話邏輯，再然後是密碼重置，那麼第三步之後冒出來的 bug 幾乎肯定住在第三步裡。一鍋端式的生成丟掉了這個訊號——bug 可能藏在你從沒讀過的四百行裡的任何地方。

在寫程式碼之前先要一份計劃

對於任何不平凡的事，把思考和打字分開。先讓 AI 勾勒出它的思路。這能在錯誤的假設變成 200 行錯誤程式碼之前就抓住它，而修一份計劃要比修一個 pull request 便宜得多。

我想給我的 API 加上限流。在寫任何程式碼之前，給我一一份簡短的計劃：限流器該放在哪裡、需要什麼樣的儲存、達到上限時會發生什麼，以及有哪些取舍。先別寫程式碼——我會先批准方案。

當計劃看起來對了，就說”好，實現第 1 步”。當它不對時，你花了三十秒糾正航向，而不是花一小時去解開一團亂碼。先做計劃，是最能把工程師式提示詞與一廂情願式提示詞區分開來的習慣。

計劃階段也是你抓住 AI 選了比所需更重的方案的地方——明明一個記憶體裡的 map 就夠，它卻伸手去夠 Redis；明明一行庫呼叫就能解決，它卻手搓一套。帶著一個問題去讀計劃：這是能跑通的最簡單做法嗎？如果這個思路聞著像過度設計，就在一行程式碼落地之前用大白話頂回去（“這是個單伺服器業餘專案——能不能跳過分散式儲存？”）。

在 diff 上迭代，而不是重寫

一旦程式碼存在了，就要抵住把整個東西重新要一遍的衝動。像”把它弄好一點”這種模糊的追問會扔掉能用的程式碼，並重新引入你早已修過的 bug。指向你想要的那個具體改動。

這不對，把整個東西重寫一遍。

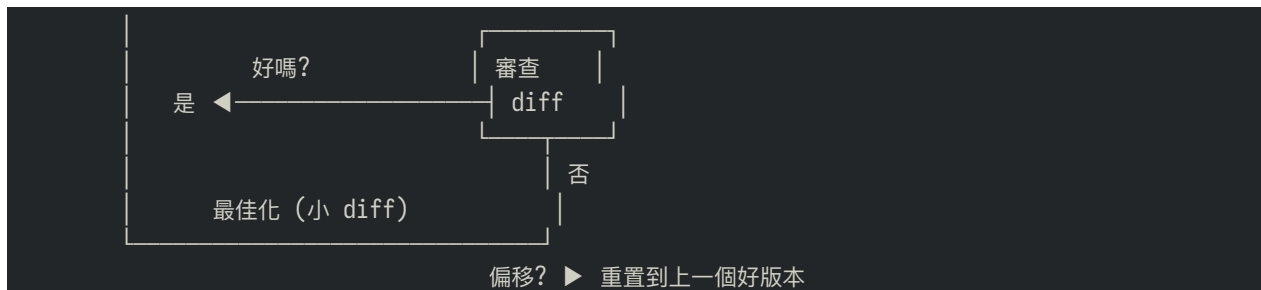
函式能用，但有兩個問題：

1. 那行 `throw new Error` 的錯誤應當返回一個 Result 型別，而不是丟擲——和上面 validateUser 函數里的模式保持一致。
2. 迴圈每次迭代都重新讀 `items.length`；把它提到迴圈外。

只給我看這兩處改動的 diff。

健康的迴圈是一個緊湊的迴路：向模型提示，讀取輸出，然後用一個小而有針對性的 diff 來打磨——絕不整體重寫——直到它對為止：





要一份 diff（而不是一次完整的重寫）能讓改動保持可審查，並保住那些已經能用的部分。把 AI 的輸出當作一位同事的 pull request：對具體的行做評論，要求有針對性的編輯。當一個改動跑偏，程式碼隨著每次回覆離你想要的越來越遠時，別再一味打補丁——回退到你信得過的上一個版本，再用一段更鋒利的描述從那裡重新提示。在一個壞掉的基礎上一路向前迭代，只會把爛攤子越疊越大。

提供示例

一個具體的示例抵得上一段描述。如果你想要某種特定形狀的輸出，就把那個形狀給它看。這是消除歧義最快的方法。

把資料格式化得漂亮些。

把這個使用者陣列轉換成一個彙總物件。

輸入：

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

期望輸出：

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

寫出轉換函式，以及一個與該示例匹配的測試。

示例釘死了命名、結構和大小寫——這些光靠文字會留得模糊不清。當你能做到時，把示例作為一個測試包含進去——這樣”讓這個測試通過”就成了一條沒有歧義的指令。這個套路可以推廣：一份樣例 API（應用程式程式設計介面——一個程式向另一個程式請求資料的方式）響應釘死一個解析器，一對”改前/改後”釘死一次重構，一張截圖或 ASCII 草圖釘死一個佈局。每當你發現自己正在寫一段對某個形狀的描述時，停下來，改為給它看那個形狀的一個例項。

告訴 AI 不要做什麼

約束不只關乎你想要的結果——它也關乎你想讓 AI 避開的招數。不加約束地放任，模型就會朝著新增依賴、把你沒讓它碰的程式碼”熱心地”重構、在你請求的邊緣發明功能的方向漂移。幾條否定式約束能讓輸出保持緊緻。

給搜尋輸入框加一個防抖。

給搜尋輸入框加一個防抖。

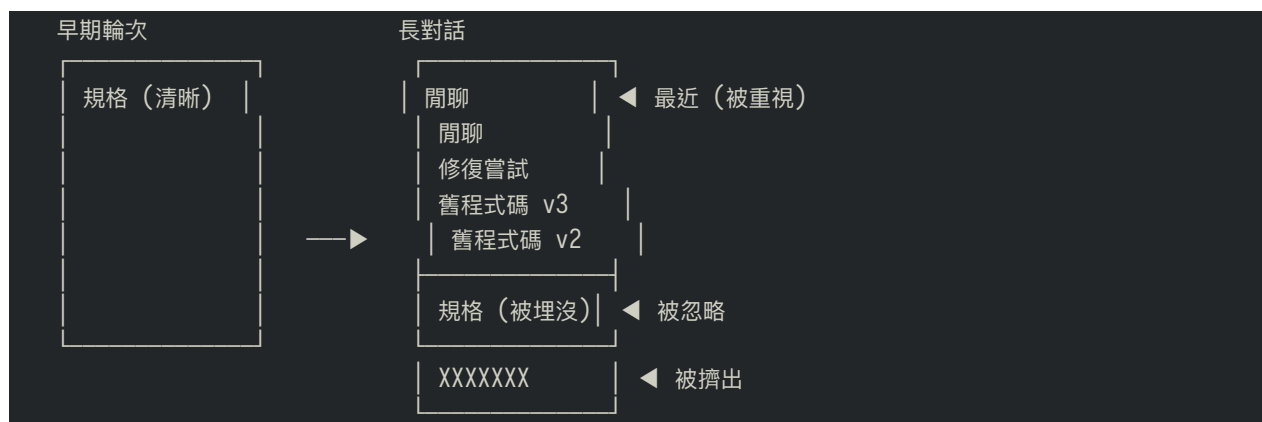
約束：

- 不要引入新依賴——我們已經有 lodash 了，用它。
- 只改 SearchBar.tsx。不要重構父元件。
- 保持現有的 prop 名稱和型別不變。
- 300ms 延遲。

第二個提示詞預先封死了第一個可能出錯的十幾條路。否定式約束在既有程式碼庫裡尤其值錢，因為在那裡，一處沒讓它做的改動的代價，不只是審查它——而是合併衝突、被嚇一跳的隊友，以及本來好好的程式碼出現的迴歸。如果有一條你不想被越過的線，就明說出來；AI 不會從客氣裡推斷你的邊界。

像管理資源一樣管理上下文

一段長對話不是免費的。隨著執行緒變長，更早的指令會被埋沒，模型會更倚重最近的訊息，二十輪之前你陳述過的細節會悄悄不再要緊。把上下文視窗想像成一個固定大小的盒子：隨著對話變長，它逐漸裝滿，而你最早的指令被推向模型最少留意的底部：



把上下文視窗當作一塊你有責任保持整潔的工作臺，而不是一份無限的記憶。

幾個習慣能幫上忙：

- **轉向時重述規格。**別假設十條訊息前的某個要求還成立。把當前目標總結成一小段，讓 AI 從那裡開始工作。
- **新問題就重開。**當你從認證程式碼切到一個 CSS（層疊樣式表——控制頁面外觀的語言）bug 時，開一段新對話。拖著不相關的上下文，只會讓模型更差，而不是更好。
- **粘當前程式碼，而不是歷史。**如果一個函式改過五次，AI 不需要那五個舊版本——它需要現在存在的那一個。把當前狀態和下一處改動給它。
- **在對話之外留一份簡短的事實來源。**對一個較長的任務，檔案裡的幾行（“技術棧、關鍵決定、已完成、下一步”）能讓你幾秒內把一段新對話重新播種，而不必從滾動記錄裡一點點重建。

目標和貫穿本章的那個一樣：在每一刻，AI 都應當從一幅準確、緊湊的任務圖景出發工作——而不是一份它得連蒙帶猜才能穿過的、漫無邊際的對話記錄。

要避免的常見反模式

這些是悄無聲息浪費你時間的提示詞習慣。在你自己的會話裡留意它們。

- **模糊的願望。**“做一個好的登入頁。”按誰的標準算好？把形容詞換成具體項：哪些欄位、哪種校驗、哪個框架。
- **巨型提示詞。**把一整個功能倒進一條訊息裡。把它拆成你能一次驗證一個的步驟。
- **沉默的上下文。**假設 AI 記得或知道你的專案。重新陳述相關的檔案、型別和慣例——尤其是在一次全新的對話裡。
- **盲目接受。**不讀就貼上生成的程式碼。AI 即便錯了也很自信；你仍然是那個署名在案的工程師。
- **沒完沒了的重試。**一遍遍跑“再試一次”，指望出現不同的輸出。如果兩次嘗試都失敗，那問題出在提示詞上——加上下文或約束，別隻是重搖骰子。
- **移動的靶子。**在對話中途改需求卻不重新陳述它們。當你轉向時，把當前的規格說明總結一下，這樣 AI 才不是在一幅過時的圖景上工作。
- **奉承迴圈。**把“你說得完全對！”當成修復正確的證據。同意不等於驗證——自己跑一遍程式碼，親自檢查行為。

核心心態

這裡的每一項技巧都歸結為一個理念：說出你本來得對一個無法讀你心、但能力勝任的隊友所要說的話。上下文放在前面。清晰的輸入和輸出。小的、可驗證的步驟。程式碼之前先有計劃。有針對性的編輯勝過重寫。在文字力不能及之處給出示例。對不許碰什麼劃下明確的界限。在每一刻都保有一幅整潔、最新的任務圖景。

你不是在寫魔法咒語。你是在寫一份規格說明，簡短而精確，然後審查結果。一貫地這麼做，AI 就不再感覺像一臺老虎機，而開始感覺像它本就能成為的那個快速、不知疲倦的工程師——而且同樣重要的是，你始終是那個明白到底交付了什麼的人。

回顧與練習

核心要點

- 給任務之前先給上下文：說明技術棧、約定、要複用的現有部件——或者直接貼上程式碼必須契合的真實簽名。
- 定義邊界：輸入、輸出、邊界情況。軟體有趣的行為都活在它的邊緣，所以把你的字數花在那裡。
- 用小而可驗證的步驟推進，並在寫程式碼前要一份計劃；修計劃遠比修一個 `pull request` 便宜。
- 在有針對性的 `diff` 上迭代，而不是整體重寫，並用具體的例子（最好做成測試）釘死那些文字留下的模糊之處。
- 把上下文視窗當作一種資源來管理，並說明不要做什麼——明確的否定約束讓輸出保持緊湊。

動手試試

拿一個你平時可能直接打出來的模糊一行請求（“給列表加個搜尋”），在發出任何東西之前先把它改寫成工程師的提示詞。加上上下文（技術棧、它要碰的檔案）、輸入/輸出、兩三個邊界情況、一個期望結果的例子，以及一條否定約束（不要碰什麼）。把改寫後的版本發出去，和那個偷懶版本會產出的東西對比。對三個真實請求都這麼做，這種升級就會變成自動的。

本章提示詞

Context: < 語言 + 框架 + 版本 >，在 < 檔案/模組 > 中工作。
We already < 可複用的相關現有模式或庫 >。

Task: < 一件清晰要構建的事 >。

Inputs / outputs: < 型別與形態 >。

Edge cases to handle: < 列出要緊的那些 >。

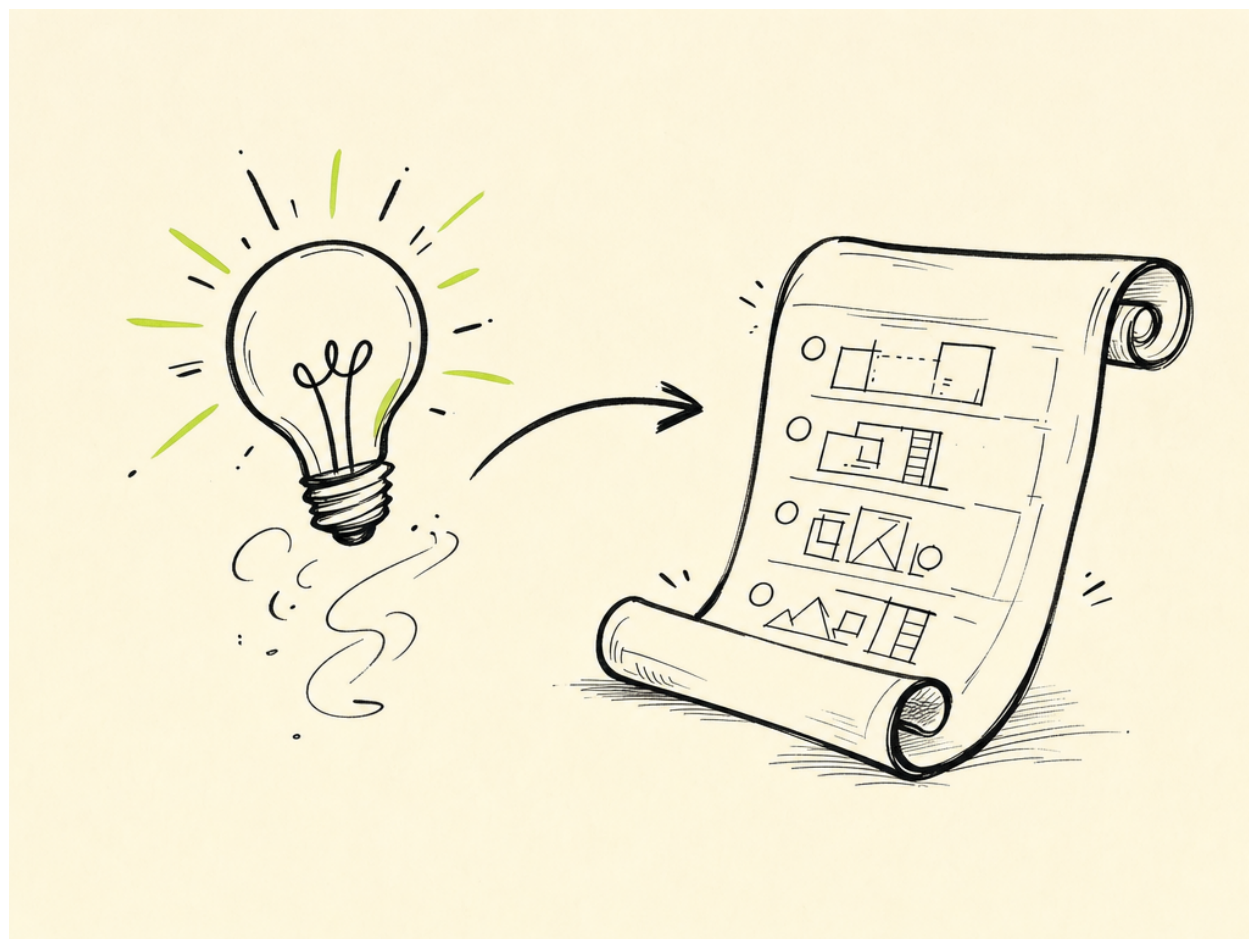
Example: < 一個具體輸入 -> 期望輸出 >。

Constraints:

- No new dependencies — use < 我們已經有的東西 >。
- Touch only < 這個檔案 >; 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".

幾分鐘內從想法到規格



每個專案都始於一個模糊的念頭：“我想要一個能幫我記錄閱讀的應用。”這是個不錯的種子，但你無法直接據此動手開發——你的 AI 也不能。把一個含糊的想法交給 AI，產出的只會是含糊的軟體：你沒要的功能它加了，你要的部分它漏了，最後留給你一堆看不懂的程式碼。

解決辦法是寫一份**規格 (spec)**——一份簡短、用大白話寫成的文件，說明你要做什麼，以及你不做什麼。好消息是：有 AI 作為你的思考夥伴，你可以在幾分鐘而非幾天內，把一句話的想法變成一份可落地開發的規格。本章就教你怎麼做。

說清楚規格不是什麼：它不是一份合同，不是一份你要在會議上據理力爭的設計文件，也不是寫一次就凍結起來的東西。它是寫給你自己和你的 AI 的一份工作筆記。你會隨著學到新東西而不斷修改它。目標不是一份完美的文件——而是一份足夠好的文件，讓接下來一小時的開發比上一小時更清晰。

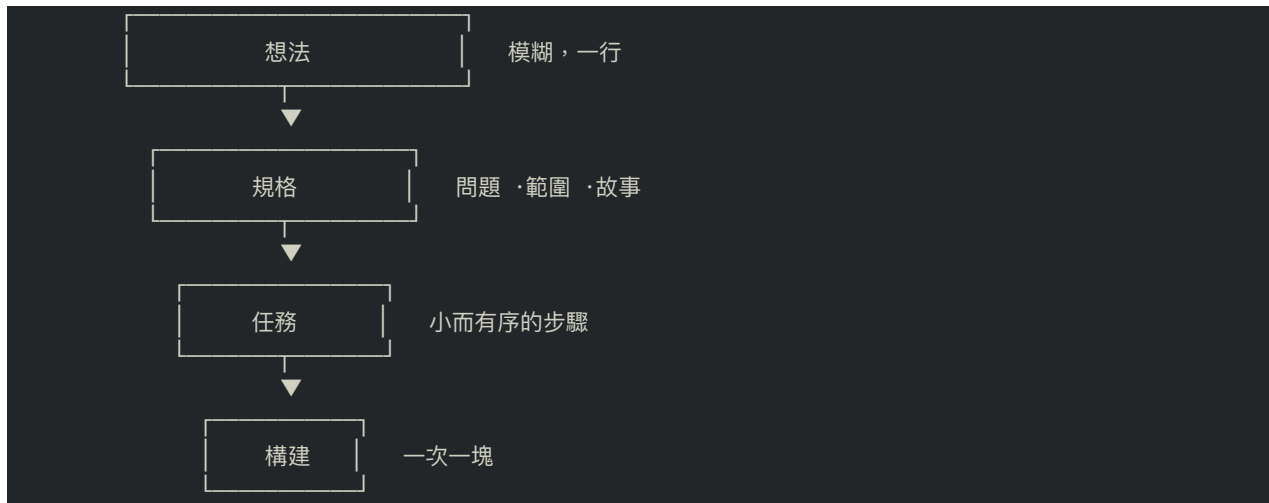
為什麼要規格，而不只是憑感覺

Vibe coding 並不意味著跳過規劃。它意味著把規劃做得快而輕，然後指揮 AI 去執行。規格就是方向盤。沒有它，AI 在每次提示時都得猜你的意圖，而一個個小誤解會層層累積成一團亂麻。

規格給你三樣東西：

- **共享上下文**——你可以把它貼上進任何 AI 會話，立刻重新確立你在做什麼。
- **範圍邊界**——一條寫下來的“做”與“不做”之間的界線，能遏制功能膨脹。
- **任務清單**——可逐個完成的、大小適中的開發塊。

規格的運作像一個漏斗：它接住一個寬泛、模糊的想法，一步步收窄，變成 AI 真正能開發的小塊。



它不需要很長。對大多數專案來說，一頁就足夠了。事實上，一份長到超過兩頁的規格，通常是個警示訊號：要麼你規劃得太超前了，要麼你在悄悄擴大範圍。拿不準時，就刪。被你刪掉的部分很少會被想起，而留下來的部分會變得更鋒利。

第一步：釐清問題

在談功能之前，先點明問題以及遇到這個問題的人。忍住直接跳到解決方案的衝動。一份清晰的問題陳述，會讓之後的每一個決定都保持誠實。

問問你自己（或者讓 AI 來採訪你）：

- 這是為誰做的？（要具體——“我自己”“獨立開發者”“我的跑團”）
- 在沒有這個東西之前，他們今天正忍受著什麼痛苦的事？
- 用一句話說，成功是什麼樣子？

你大可以讓 AI 來挖掘。試試這個提示：

我有一個粗略的想法：一個記錄我在讀的書的網頁應用。
扮演產品思考夥伴。向我提出 5 個犀利的問題，
一次一個，以釐清問題、目標使用者，以及
“完成”是什麼樣子。先別提議功能——只管追問
這個想法，直到它變得具體。

回答五個好問題，通常就足以把一個模糊的想法打磨成實在的東西。訣竅在於讓 AI 反駁你，而不是對一切都點頭同意。如果它問“你需要跨裝置同步嗎？”而你誠實的回答是“不需要，只在我的筆記本上用”，那你剛剛就刪掉了一個後端、一套認證系統，以及好幾周的工作量——而這一切都在寫下一行程式碼之前。這正是關鍵所在：砍掉一個功能最便宜的地方是在對話裡，而不是在程式碼庫裡。

留意這樣一個時刻：當你的回答從“嗯，大概吧”變成“對，就是這樣”。這種轉變就是訊號，說明你找到了真正的問題，可以往下走了。

第二步：定義範圍與非目標

這是最有價值、也最常被跳過的一步。**非目標**——那些你刻意不做的事——能保護你不被一個總愛新增“貼心”附加功能的 AI 帶偏。

對我們的圖書追蹤器來說，範圍可能是：

- **範圍之內**：新增一本書，標記為在讀/已讀完，檢視列表，給讀完的書評分。
- **範圍之外（非目標）**：社交功能、推薦、從 Goodreads 匯入、移動端應用、使用者賬戶。

想像兩欄：左邊是你承諾要做的一切，右邊是你刻意跳過的一切。它們之間的那條線，就是讓 v1 保持小巧的邊界。

範圍內	非目標
新增書籍	社交功能
標記在讀/完成	推薦
檢視列表	匯入 Goodreads
給讀完的書評分	移動應用
	使用者帳戶
現在做 ←————→ 暫不做	

把非目標寫下來有兩重作用。它讓你的第一個版本小到真能做完；同時，當 AI 跑偏時，它給了你一句可以直接貼上的話：“記住，帳戶系統是 v1 的非目標——不要加登入。”

非目標不是“永遠不做”——而是“現在不做”。把使用者帳戶列為非目標，並不意味著你的應用永遠不會有它，而是說它不屬於這個版本，所以你今天不用為它付出代價。這種重新表述很重要：它讓你對一個功能說不，卻不會覺得自己在放棄一個好點子。你只是在排順序。你非目標清單裡的大多數條目，其實都是偽裝起來的 v2 待辦列表，這沒什麼不好——把它們寫在某處，然後繼續往前走。

第三步：編寫使用者故事

使用者故事把範圍轉化為可測試的行為。格式很簡單：

作為一名 **[使用者]**，我想要 **[執行某動作]**，以便 **[獲得某好處]**。

對圖書追蹤器來說：

- 作為一名讀者，我想要按書名和作者新增一本書，以便我能開始追蹤它。
- 作為一名讀者，我想要把一本書標記為已讀完，以便我能看到我的進度。
- 作為一名讀者，我想要給讀完的書打 1–5 星，以便我記得自己喜歡什麼。

每個故事都是一個可獨立開發並驗證的單元。如果你想像不出怎樣靠在應用裡點點點來測試一個故事，那它就太含糊了——拆開它，或者重寫它。

so that（以便）從句是人們最常丟掉的部分，而它恰恰是最重要的。“作為一名讀者，我想要一個搜尋框”告訴了 AI 要做什麼，卻沒說為什麼，於是它沒法做出好的取捨。“作為一名讀者，我想要通過輸入書名的一部分在我的列表裡找到一本書，以便我不必滾動翻過幾十條記錄”告訴了它真正要乾的活——這下，一個簡單的即輸即篩或許就勝過一個重量級的搜尋引擎。好處就是這份規格的規格：它把功能解釋得足夠清楚，讓 AI（以及未來的你）能分辨出好的實現和錯的實現。

第四步：起草一份輕量 PRD

PRD（產品需求文件）不過是把你的決定彙總到一處。它有一副可預測的骨架——每次都是那幾個小節：

PRD	
問題	為何存在
使用者	為誰而做
目標	做什麼，一句話
範圍內	v1 包含什麼
非目標	v1 略去什麼
使用者故事	作為 __，我想 __，以便 __
資料模型	你儲存的欄位
完成標準	驗收測試

保持它精簡。下面是一份完整的入門級 PRD，你可以直接交給 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”**（完成的標誌）。那是你的驗收測試——一個判斷你是否已交付 v1 的具體方式。永遠要寫上一條。一個好的“Done When”，是你能親手做到的事：點這裡、輸入這個、重新整理、看到那個。如果你的完成條件讀起來像一種感覺（“等它打磨好了”“等它感覺對了”），那它永遠完不成，因為沒有什麼可供你對照檢查。把它寫成一份你能交給陌生人去照做的指令碼。

你不必手寫這份 PRD。一個有用的做法是，把你前面幾步的回答喂回給 AI，讓它來拼出草稿，然後你來編輯。之所以是編輯而不是照單全收，是因為你必須理解並認同每一行——這份文件是你會反覆貼上回去的事實來源，所以這裡的一個錯誤會被複制進之後的每一次會話。

第五步：勾勒資料模型

大多數應用其實都是關於資料的：你存什麼，以及各部分之間如何關聯。快速勾勒一份資料模型，能让你日後免於混亂、前後不一致的程式碼。你不需要畫資料庫圖——列出欄位就夠了。

對圖書追蹤器來說，主要只有一樣東西——一個 Book：

```

Book
- id: 唯一字串
- title: 字串
- author: 字串
- status: "to_read" | "reading" | "finished"
- rating: 數字 1–5（僅在讀完時）
- addedAt: 日期

```

哪怕這麼小的一份草圖，也能讓決定變得明確：評分只對讀完的書存在，狀態是三個固定值之一，而每本書都有一個 id。當你把这个交給 AI 時，它就不再猜測欄位名，而會前後一致地構建。

之所以要趁早把这个釘死，是因為欄位名會滲透到處處。一旦 AI 寫出讀取 `book.rating` 的程式碼，這個名字就會出現在表單、列表檢視、儲存層，以及你日後新增的任何功能裡。如果你任由它漂移——這處叫 `rating`、那處叫 `stars`、第三處又叫 `score`——你就會得到一些追查起來很煩人的 bug。一份六行的資料模型，是防止這種事的最便宜的保險。

一段簡短、可複製貼上的規格塊，對 AI 而言往往比散文更管用。下面是同一個模型，寫成你真正會發出去的提示：

書籍請使用這個精確的資料結構。不要新增我沒列出的欄位，也不要重新命名這些：

```
Book {
  id      —唯一字串，建立時生成
  title   —字串，必填
  author  —字串，必填
  status  —取值之一: "to_read" | "reading" | "finished"
  rating  —整數 1–5，僅當 status 為 "finished" 時存在
  addedAt — ISO 日期字串，新增書籍時設定
}
```

如果你覺得缺了某個欄位，新增前先問我。

最後那一行——*ask me before adding it*（要加之前先問我）——在實打實地起作用。它把 AI 從一個猜測者變成一個協作者，並讓資料模型始終歸你所有。

第六步：把規格拆成 *vibe* 大小的任務

最後一招是把你的規格切成 AI 能逐個執行的任務。一個 *vibe* 大小的任務，要小到你能在一分鐘內審完結果，並判斷它是否奏效。如果一個任務讓 AI 改了好幾個檔案，而你又說不清它對不對，那它就太大了。

圖書追蹤器的好任務：

1. 搭好基本的頁面佈局，帶一個頁頭和一個空的圖書列表。
2. 做出“新增一本書”的表單（書名 + 作者），將其追加到記憶體中的列表裡。
3. 加上狀態（待讀 / 在讀 / 已讀完），並用一個下拉框來切換它。
4. 把列表儲存到瀏覽器本地儲存，讓它在重新整理後依然存在。
5. 加一個只對已讀完的書出現的評分控制元件。
6. 在列表上方加上狀態篩選按鈕。

每個任務都對應著一個使用者故事或一行 PRD，建立在上一個之上，並以你能看見其運作的成果收尾。這就是 *vibe coding* 的節奏：小任務、跑起來、驗證、下一個。

你甚至可以讓 AI 替你生成這份拆解：

這是我的迷你 PRD 和資料模型 [貼上在此]。
把它拆成 6–8 個小而有序的構建任務。每個任務都應
可獨立測試，並建立在前一個之上。每個任務
保持在我點選應用就能驗證的範圍內。

讓規格保持鮮活

規格不是一座你建起一次就供著觀賞的紀念碑。它是一份活文件，而最常見的錯誤就是把它當成固定不變的。隨著開發推進，你會學到一些規劃階段無法告訴你的事：某個功能比預想的更難，某個非目標原來不可或缺，某個使用者故事其實暗藏著兩個故事。當這種情況發生時，先改規格，再讓 AI 去遵循新版本。規格在前，程式碼在後。

有幾個習慣能讓它保持誠實：

- **決定一變，立刻更新它。**一份嘴上說一套、而程式碼裡做著另一套的規格，比沒有規格更糟——它會主動誤導下一次 AI 會話。
- **完成一個任務就劃掉一個。**任務清單同時充當進度追蹤器。看到八項裡勾掉了六項，本身就是一種勢頭。
- **把意外挪進非目標或一份“以後再說”清單，而不是直接就把它們做了。**如果開發途中冒出一個誘人的功能，寫下來，然後繼續往前走。你以後可以再決定它是否配得上 v1 裡的一席之地。

正是這種自律，把“我做了個東西”和“我做出了我本想做的那個東西”區分開來。規格是你讓自己繼續當專案作者、而不是淪為專案乘客的方式。

一個常見陷阱：規定的是解決方案，而非問題

有一種失敗模式值得單獨拎出來說，因為它實在太容易掉進去了。當你坐下來寫規格時，誘惑在於去描述你腦中已經想好的那個解決方案——“一個帶標籤頁的側邊欄”“一個彈出來的模態框”“一張 Postgres 表”。但一份滿是解決方案的規格，會悄悄鎖定一些你還沒資格去做的決定。

對比這兩行：

差： 新增一個左側邊欄，每個書籍狀態有可摺疊的分割槽。
好： 讓我能按狀態分組檢視書籍，又不丟失完整列表。

第一句已經選定了一個側邊欄、可摺疊的分割槽，以及一種佈局——而這一切都在還沒人確認它們是不是正確的選擇之前。第二句陳述的是要完成的活，讓 AI（以及你）有自由去找到能滿足它的最簡單方案，而那也許只是三個篩選按鈕。描述你想要的結果；讓實現儘可能長久地保持可商量。解決方案在規格裡改起來很便宜，在程式碼裡改起來卻很昂貴。

回報

現在你有了一份問題陳述、範圍與非目標、使用者故事、一頁 PRD、一個數據模型，還有一份有序的任務清單——一個真實專案的全部根基，全都在幾分鐘內、以 AI 為你的回聲板起草完成。

這一切都不要求你會寫程式碼。它要求的是你知道自己想要什麼，並清晰地思考幾分鐘——剩下的思考，AI 會陪著你出聲地完成。這才是真正的轉變：你還沒在寫軟體，你在寫意圖，而好的意圖就是這場仗的大半。

把這份規格留在專案裡的一個檔案中（許多人用 SPEC.md 或 PRD.md）。它會成為你貼上進全新 AI 會話、用來重新錨定上下文的那份文件，也會成為你逐個 vibe 大小任務往下做的那張清單。在下一章裡，我們將拿起第一個任務，真正把它做出來。

回顧與練習

核心要點

- 規格是一份簡短、平實語言的工作筆記——你在做什麼、不做什麼——而不是一份要凍結的合同；一頁就夠了。
- 在功能之前先釐清問題和擁有這個問題的人，並寫下非目標（non-goals）：它們是抵擋 AI 新增“貼心”附加功能的最鋒利防線。
- 帶 so that 子句的使用者故事，加上有具體“Done When”那一行的小型 PRD，把意圖變成可測試、可構建的行為。
- 儘早釘死資料模型——欄位名會滲漏到處都是——並把規格切成你能一次驗證一個的 vibe 大小任務。
- 描述問題，而不是解法；結果在規格裡改起來便宜，在程式碼裡改起來昂貴。一邊學一邊讓規格保持鮮活。

動手試試

拿一個你一直擱著的想法，在一次 AI 會話裡把它變成一頁規格。先讓 AI 用五個尖銳的問題對你做訪談，然後——用你自己的話——寫下問題、使用者、三個使用者故事（每個都帶 so that）、一份明確的非目標清單、一個六行的資料模型，以及一行你能物理執行的“Done When”。存成 SPEC.md。好規格的檢驗標準：明天你把它粘進一個全新的對話，AI 就能確切知道該構建什麼。

本章提示詞

我有一個粗略的想法：< 一行描述 >。

扮演我的產品思考夥伴。首先，向我提出 5 個犀利的問題，

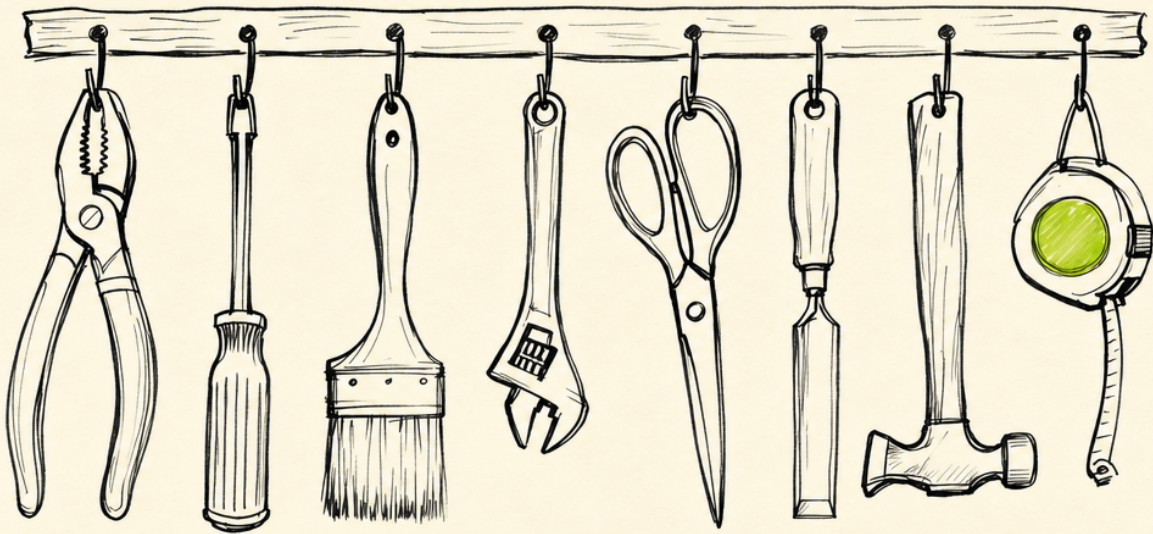
一次一個，以釐清問題、目標使用者，以及"完成"是什麼樣子。先別提議功能。

我回答後，按這些小節起草一頁的規格：

- 問題 - 使用者 - 目標
- 範圍內 (v1) - 非目標 (v1)
- 使用者故事 (每條"作為 __，我想 __，以便 __")
- 資料模型 (欄位的簡短列表)
- 完成標準 (具體、可實際執行的驗收測試)

保持精煉——一頁以內。若有含糊之處，先問再假設。我會修改草稿，直到我認同每一行。

選擇你的技術棧：語言一覽



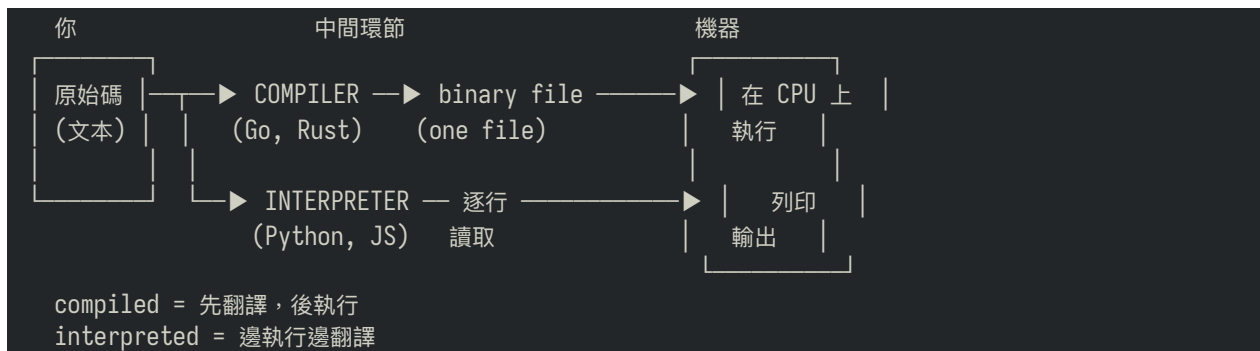
當你指揮 AI 來開發軟體時，你挑選的語言依然很重要——非常重要。程式碼是 AI 寫的，但後果由你承擔：你交付有多快、部署有多容易、多久會撞上一堵牆。好消息是，你不需要精通這些語言中的任何一門。你只需要懂得足夠多，能做出一個明智的選擇，並能識別出 AI 何時正把你引向一條痛苦的路。

本章會逐一過一遍值得關注的語言，講清每一門真正擅長什麼、在哪裡會咬人，以及——這對 vibe coding 至關重要——AI 助手對它的駕馭能力如何。最後我們會給出一份按目標選擇的極簡指南。

開始前先提一句：AI 助手對那些擁有海量公開程式碼的熱門語言，要遠遠強於對小眾語言的處理。這可不是個小差別。它往

往是決定你的專案是否順利的最大單一因素。流行度本身就是一項功能。當一門語言及其主要框架在公開倉庫裡出現過幾百萬次時，AI 就見過了每一種常見模式、每一個典型錯誤、每一種地道的修法。如果沒有，你就要為這份稀缺埋單——以失敗的嘗試和聽起來煞有介事的胡話來償付。

在看語言名字之前，先記住一幅圖：你寫的程式碼，從來不是機器真正執行的東西。中間有某個東西把你的文本變成執行中的軟體——要麼是編譯器（提前把全部翻譯成一個可執行的檔案），要麼是直譯器（一行一行地讀取並執行）。你選的語言決定了你走哪條路。



JavaScript / TypeScript

這是大多數 vibe coding 的預設選擇，而且理由充分。它無處不在——瀏覽器、伺服器（Node.js）、邊緣函式——而且它是整個 Web 前端的語言。TypeScript 是給 JavaScript 加裝了一套型別系統，你幾乎總應該優先選 TypeScript：型別能儘早抓住錯誤，更重要的是，它給 AI 設了護欄，讓其產出更可靠。一個帶型別的函式簽名，就是一份助手能讀懂的契約；沒有它，AI 只能猜你的資料是什麼形狀，而猜測正是 bug 藏身之處。

- **擅長：** Web 應用（前端和後端）、任何使用者在瀏覽器裡點選的東西、快速原型、用一門語言搞定全棧專案、serverless/邊緣部署。
- **痛點：** 生態系統變化飛快且容易破壞既有程式碼；依賴氾濫是真實存在的（一個小應用可能拉進數百個包）；純 JavaScript 的鬆散讓微妙的 bug 得以溜過去——這又是該用 TypeScript 的一個理由。
- **典型用例：** SaaS（軟體即服務——無需安裝、線上租用的軟體）儀表盤、落地頁、電商、即時應用、瀏覽器擴充套件，以及任何你部署到 Vercel、Netlify 或 Cloudflare 的東西。
- **AI 駕馭能力：** 出色。這是訓練資料中代表性最強的語言。助手能自信地寫 React、Next.js 和 Node，且很少卡殼。如果你沒有強有力的理由另作選擇，就從這裡開始。

有一個值得了解的具體取捨：讓 JavaScript 在原型設計上如此出色的那份速度，同樣是讓它不斷翻新的速度。一篇十八個月前的教程可能已經在用一個被廢棄的 API，而以過去快照訓練出來的 AI——有時會遞給你昨天的寫法。當它寫的東西跑不起來時，原因往往是庫已經往前走了。修法通常是告訴你你用的版本，並讓它去查當前文件。這是你為住在地球上最活躍的生態系統裡所交的稅，而對大多數專案而言，這稅值得交。

Python

Python 讀起來幾乎像英語，這讓它成為對初學者最友好的語言，也成為資料、指令碼以及 AI 本身的通用語。如果你的專案涉及機器學習、資料分析、自動化或科學計算，答案通常就是 Python。

- **擅長：** AI/ML、資料處理、自動化指令碼、後端 API（應用程式程式設計介面——一個程式與另一個程式對話時使用的門戶）、膠水程式碼，以及任何已經有現成庫的場景（對 Python 來說，通常都有）。
- **痛點：** 純數值運算很慢，除非你依賴底層用 C 寫的庫；虛擬環境和包版本帶來的“依賴地獄”讓新手困惑；不適合做瀏覽器前端。
- **典型用例：** 爬蟲、資料流水線、ML 實驗、Discord/Telegram 機器人、內部工具、REST（表述性狀態轉移——構建這類 API 門戶的一種常見風格）API（FastAPI、Django、Flask）。
- **AI 駕馭能力：** 出色，與 JavaScript 不相上下。助手對 Python 及其主要庫極為流利。對非 Web 專案而言是個很棒的預設選擇。

Python 唯一會一再絆倒新手的，是環境，不是語言。一個指令碼在 AI 那裡某個資料夾裡跑得好好的，換個資料夾就失敗了，原因是那些包住在一個沒被啟用的虛擬環境裡，或者兩個專案想要同一個庫的不同版本。這些都不是你程式碼的錯，

見過一次之後也不難——但它是 Python 裡最常見的”為什麼就是跑不起來”時刻。從一開始就讓 AI 設好一個虛擬環境和一份 requirements.txt (或用 uv、Poetry 這類工具)，大部分痛苦就消失了。

Go

Go 由 Google 設計，目標是簡單、快速、易於部署。它編譯成一個獨立自包含的二進位制檔案，你可以直接丟到伺服器上，無需安裝任何執行時。它有意省去了花哨的特性，這讓程式碼庫以最好的方式保持”無聊”——而無聊對人類和 AI 來說都易於推理。

- **擅長**：快速的後端服務、命令列工具、網路程式設計、任何需要處理大量併發連線的東西、簡單部署。
- **痛點**：囉嗦的錯誤處理 (你會滿眼都是 `if err != nil`)；不是前端語言；那種刻意為之的簡單，在你想要富有表現力的捷徑時會讓人覺得受限。
- **典型用例**：API、微服務、CLI (命令列介面——通過輸入命令來執行程式的純文本方式)、基礎設施工具、高流量後端。
- **AI 駕馭能力**：很好。Go 那套小而一致的特性集意味著大多數事情都有一種”正確做法”，助手也能很好地遵循它。訓練資料比 JS/Python 少，但其簡單性彌補了這一點。

對單幹的開發者來說，Go 的殺手級特性正是那個單一二進位制檔案。沒有”在伺服器上裝 Node 和這四十個包”的繁瑣舞步，也沒有你的機器和生產環境之間的執行時版本不匹配。你構建、複製一個檔案、執行它。僅此一點就讓部署的出錯機率大幅降低——當你在 vibe coding、不想去除錯基礎設施時，這一點意義重大。囉嗦是它的代價：在 Python 裡三行的程式碼，在 Go 裡變成八行。但那些多出來的行是可預測的，而可預測恰恰是 AI 擅長應付的東西。

Rust

Rust 給你 C 級別的速度，外加一個拒絕讓你寫出整類記憶體 bug 的編譯器。代價是一個出了名嚴格的編譯器和一條陡峭的學習曲線。對 vibe coding 而言，Rust 是一把雙刃劍：AI 能寫它，但 Rust 的嚴格意味著編譯器會更頻繁地拒絕程式碼，從而帶來更多來回修復的迴圈。

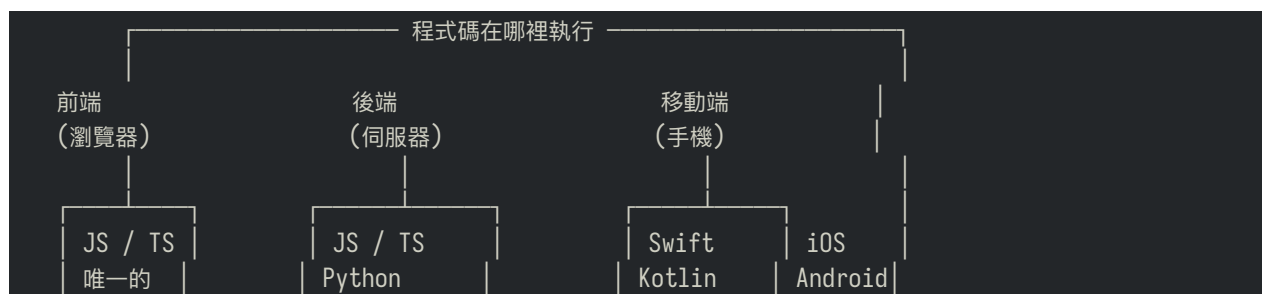
- **擅長**：效能關鍵的軟體、系統程式設計、遊戲引擎、WebAssembly、對速度有要求的命令列工具，以及任何把正確性與安全性放在首位的場景。
- **痛點**：學習曲線陡峭；編譯速度慢；“借用檢查器”讓初學者抓狂；對大多數 Web 應用和指令碼來說殺雞用牛刀。
- **典型用例**：高效能後端、CLI 工具、WASM (WebAssembly) 模組、近嵌入式系統、每一毫秒都要緊的基礎設施。
- **AI 駕馭能力**：不錯，但更顛簸。助手很懂 Rust，但嚴格的編譯器會丟擲需要反覆迭代的錯誤。預期會比 Python 或 JS 出現更多”修錯誤、再試一次”的迴圈。當效能或安全性真正值得這份摩擦時再選 Rust——而不是把它當預設。

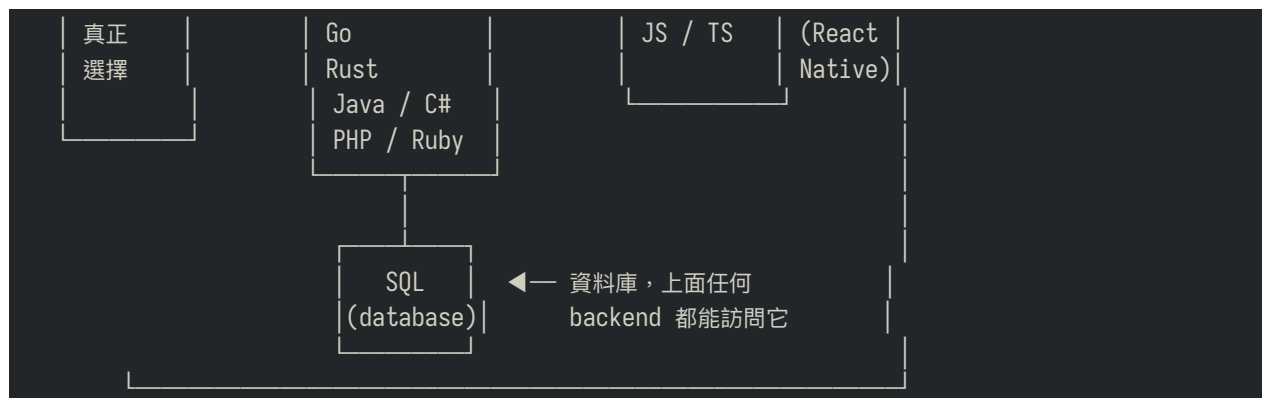
Rust 的嚴格有一個容易被忽略的微妙好處：編譯器是一個免費而無情的審閱者。當 AI 寫出馬虎的 JavaScript，那個 bug 會被交付出去，在執行時咬到一個使用者。當 AI 寫出馬虎的 Rust，編譯器通常會在程式還沒跑起來之前就抓住它。多出來的修復迴圈感覺像摩擦，但其中相當一部分，是編譯器在替你攔下一個真實的 bug。在伸手去拿 Rust 之前要問的那個誠實問題是：你究竟是真的需要那份速度或安全，還是隻是被告知你應該想要它們。對第一個專案來說，你幾乎從來都不需要。

關於其他語言的簡短一說

你會碰到這些，而且有時其中某一門正好對路：

想像一下每門語言主要住在哪裡會有幫助。軟體有層次——在瀏覽器裡跑的 (前端)、在伺服器上跑的 (後端)、在手機上跑的 (移動端) ——而大多數語言都有自己的主場：





- **Java / C#**：大型企業的主力。成熟、穩定、工具鏈厚重，是大型企業後端、Android (Java/Kotlin) 或 Windows/.NET 生態 (C#) 的理想之選。AI 對兩者都駕馭得很好。作為單幹的 *vibe coder*，你很少需要它們，除非你是在一個既有的企業技術棧裡工作，或者專門為 Android/Windows 開發。
- **Swift / Kotlin**：iOS (Swift) 和 Android (Kotlin) 應用的原生語言。如果你要把一個真正的移動應用交付到 App Store 或 Play Store，你很可能會接觸到它們。AI 的支援是紮實的，但弱於 Web 語言，而且移動開發還帶著額外的摩擦（裝置、應用商店規則、構建工具鏈）。
- **PHP / Ruby**：Web 的老近衛軍，至今仍靜靜地驅動著其中很大一塊。PHP 驅動著 WordPress，因而撐起了所有網站裡相當大的一個比例；Ruby on Rails 至今仍是一種快速、有主見地構建資料庫支撐型應用的方式。兩者都有充足的訓練資料，所以 AI 駕馭得很好——但如今新手很少會第一個去伸手拿它們，你通常之所以選它們，是因為你在一個既有的程式碼庫裡工作，而不是從頭開始。
- **SQL (結構化查詢語言)**：它不是通用語言，但一旦你的應用要存資料，你就會遇到它。可以把它想成向資料庫提問的語言——“把這周註冊的所有使用者給我列出來”。SQL 用來查詢資料庫。你不是用 SQL 代替其他語言——它和你選的任何東西並存。好消息是：AI 極擅長寫 SQL，所以你可以用大白話描述你想要什麼資料，讓它產出查詢。

一覽對比

語言	最適合	需當心	AI 流利度
JavaScript/TypeScript	Web 應用、全棧、邊緣	依賴氾濫、迭代飛快	出色
Python	AI/ML、資料、指令碼、API	純計算慢、環境配置	出色
Go	快速後端、CLI、服務	囉嗦、不適合前端	很好
Rust	效能、安全、WASM	曲線陡峭、修復迴圈多	不錯（更顛簸）
Java/C#	企業、Android、.NET	笨重、單幹常屬殺雞用牛刀	很好
Swift/Kotlin	原生 iOS / Android 應用	移動摩擦、AI 較弱	不錯
PHP/Ruby	WordPress、Rails、既有 Web 應用	預設設定過時、歷史包袱	很好
SQL	與資料庫對話	並存而非替代	出色

異類選擇的真實代價

人很容易因為一門語言優雅、或快、或因為你敬重的某個人對它推崇備至，就去選它。專就 *vibe coding* 而言，這種本能可能悄悄地把一個專案拖沉。原因在於：你花在和一門小眾語言的工具鏈搏鬥上的每一個鐘頭，或者花在理順那些因為模型見過例子太少而微妙出錯的 AI 產出上的每一個鐘頭，都是你沒拿去交付的鐘頭。異類選擇很少會大聲失敗。它以一種緩慢的滴漏方式失敗——多出來的修復迴圈、缺失的庫、根本不存在的 Stack Overflow 答案。

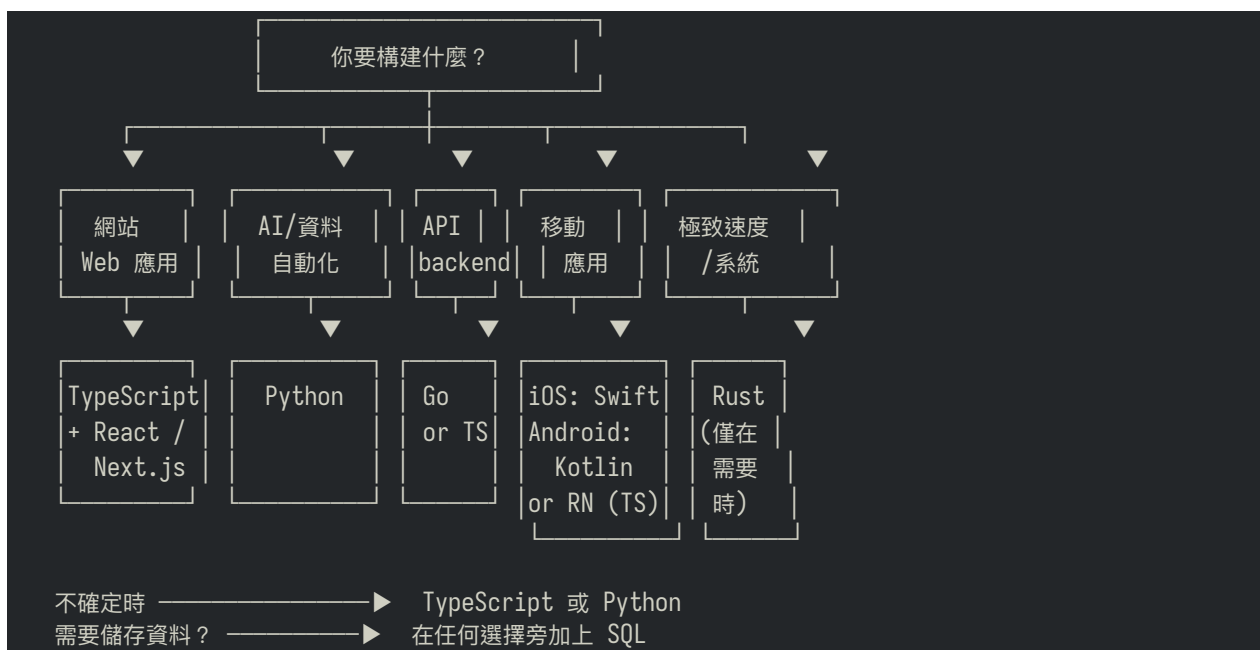
具體地說，這份懲罰出現在三個地方。第一，AI 不那麼流利，於是它犯更多錯、也更不擅長修正——而不懂這門語言的你，分不清好程式碼和壞程式碼。第二，生態更薄，於是你需要的那個庫可能不存在、可能已被廢棄、或者沒有可供照抄的例子。第三，當你卡住時，整個網路更安靜：教程更少、被回答過的問題更少、撞上你這堵牆的人更少。在 Python 或 TypeScript 裡，這三樣都不成其為問題。

這些都不意味著異類語言不好。它意味著舉證責任落在異類選擇那一邊。如果你有一個具體、明確的理由——你確實需要 Rust 的速度，你被 App Store 逼著用 Swift，你在擴充套件一個既有的 Go 服務——那就睜著眼睛去付這份代價。如果理由是“它看著挺酷”，那就選熱門的那個，去交付點東西。你隨時可以日後重寫，但你幾乎從不會，因為熱門的那個選擇早已把活幹完了。

按目標來選

別再糾結了。把你的目標對上一行，然後往前走——你隨時可以改弦更張，而對於第一個專案來說，勢頭勝過最佳化。

把整個決定畫成一棵樹就是這樣。從頂部開始，回答每個問題，順著分支走到你的技術棧：



- 我想做一個網站或 **Web** 應用。□ TypeScript (配 React/Next.js)。顯而易見、支援完善的選擇。
- 我想做 **AI**、資料或自動化。□ Python。工具都在那兒。
- 我想要一個快速的 **API** 或後端服務。□ Go (簡單又快) 或 TypeScript (如果它是某個 Web 技術棧的一部分)。Python 的 FastAPI 也運作得很好。
- 我在做一個移動應用。□ iOS 用 Swift，Android 用 Kotlin——或者考慮一種跨平臺、基於 Web 的方案 (React Native)，從而留在 TypeScript 裡。
- 我需要純粹的速度或系統級的掌控。□ Rust，但僅當你已確認自己確實需要它時。大多數人並不需要。
- 我在給一個已經存在的站點添東西。□ 用它本來就是用的那門語言。一個 WordPress 外掛意味著 PHP；一個老的 Rails 應用意味著 Ruby。順著程式碼庫來，別跟它較勁。
- 我只想學點東西並快速交付。□ Python 或 TypeScript。兩者都很寬容，都是 AI 最熟悉的語言，也都有無窮無盡的教程和示例。
- 我的應用需要存資料。□ 上面你選的任何東西，外加用於資料庫的 SQL。你不是在它們之間二選一。

那條元層面的教訓是：拿不準時，選熱門的那個。TypeScript 和 Python 覆蓋了一個單幹 **vibe coder** 這輩子會做的絕大多數東西，而且它們是你的 AI 助手處理得最可靠的語言。對 **vibe coding** 而言，“最好”的語言通常就是 AI 最熟悉的那一門——而那意味著熱門的那一門。把那些異類的選擇留到你具有具體理由的時候，反正無論哪種，語法都交給 AI 處理。

回顧與練習

要點回顧

- AI 對熱門語言的駕馭遠比小眾語言流暢——熱門本身就是一項功能，往往是決定專案是否順利的最大單一因素。
- TypeScript 和 Python 是安全的預設選項：Web/全棧用 TypeScript，AI/資料/自動化用 Python，兩者的 AI 支援都很出色。

- Go（單一二進位制後端）、Rust（速度/安全，但修復迴圈更多）以及其餘語言各有狹窄的最佳適用面——只有在有具體理由時才去碰它們。
- SQL 不是二選一的選擇；一旦你的應用要存資料，無論你選哪門語言，它都會與之並存。
- 讓語言匹配你的目標，而不是匹配聽起來唬人的東西——在第一個專案上，勢頭勝過最佳化。

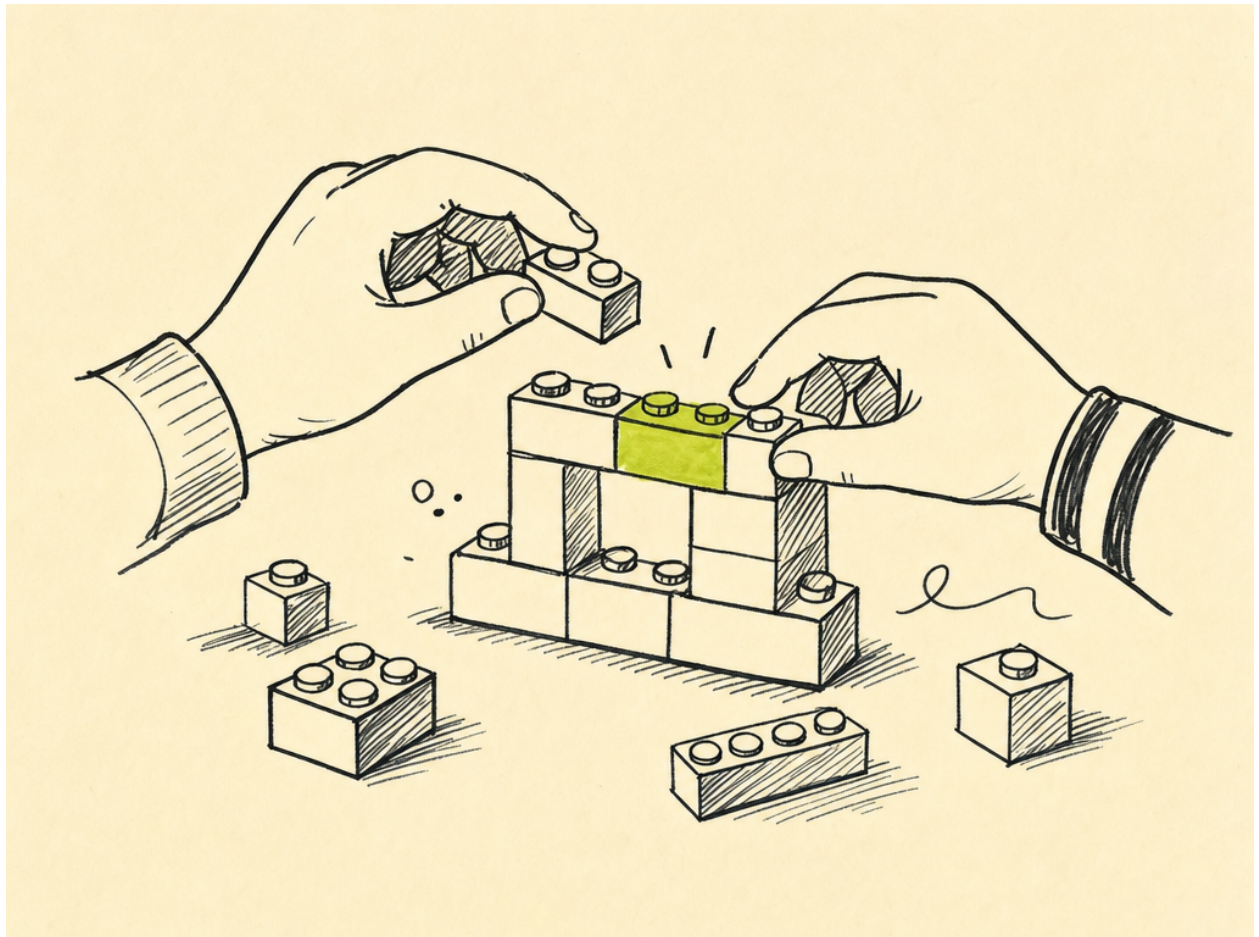
動手試試

挑一個你真正想做的專案點子，用一句話寫下它的目標。把它對應到“按目標來選”列表中的某一行，然後讓你的 AI 助手確認或質疑這個選擇。讓它用三個要點列出預設技術棧（語言、框架、部署到哪裡），並請它指出在哪一種情形下另一門語言會更合適。在寫下一行程式碼之前，你就握有了一個有依據的技術棧決定。

我想構建以下專案：[用一兩句話描述它]。

為它的 *vibe coding* 推薦最佳的語言和框架，最佳化的目標是 AI 助手能多流暢地編寫和修復程式碼，而不是追求極致效能。請給我：（1）語言和主要框架，（2）我會把它部署到哪裡，（3）一個需要留意的現實陷阱，以及（4）一個換用其他語言反而會更合適的場景。請簡短作答。

構建功能，一個 *vibe* 接一個 *vibe*



到現在，你已經能和你的 AI 對話，讓它寫出程式碼了。但一個真實的功能比單條提示要大。把能交付的人和會卡住的人區分開來的那項技能，是懂得如何把一個功能拆成小塊，並驅動 AI 逐個把它們做完——在每一步都讓應用保持可執行。

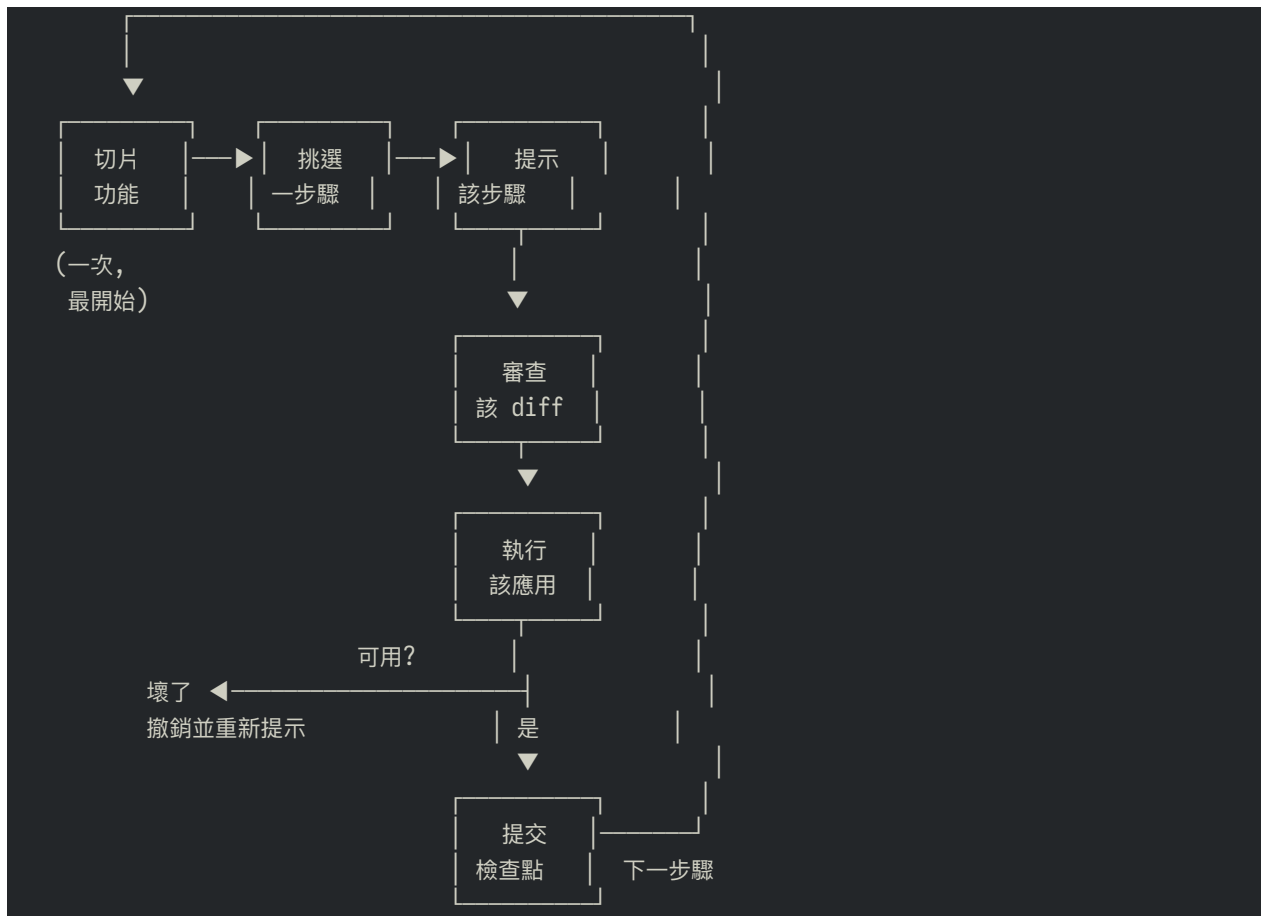
本章就是核心的構建迴圈。學會它一次，你就能在此後構建的每一個功能裡反覆複用。這正是一個穩步成長的專案，和一個在剛要變得有趣的那一刻就被自身重量壓垮的專案之間的區別。

構建迴圈

下面就是你將一遍又一遍執行的迴圈。它故意做得很小。

1. 把功能切成小的縱向步驟。每一步都應是你能在幾分鐘內構建、執行、並看到它運作的東西。
2. 挑一個步驟。就一個。眼下其他都先不管。
3. 為那一個步驟寫提示。對什麼要改、什麼要保持不變都要說具體。
4. 審查改動。讀一讀 AI 寫了什麼。你不需要看懂每一行，但要掃一眼有沒有明顯錯的地方。
5. 執行應用。真的去點那個按鈕、載入那個頁面、請求那個端點。確認這一步奏效。
6. 提交。在版本控制裡用一句簡短的資訊存一個檢查點。
7. 重複第 2 步，換下一個步驟——直到功能完成。

把這個迴圈畫成一個環就是下面這樣——你繞著它一圈圈轉，每圈一個小步驟，直到功能完成：



整個要點在於，你永遠離一個可執行的應用不遠。如果第 4 步產出了壞掉的東西，你只有一個小改動要撤銷，而不是一團纏成結的亂麻。

跳過第 4 到第 6 步的人，會有大約二十分鐘覺得自己更快。然後他們撞上一個找不到的 bug，因為他們應用上一個已知可用的版本，在一百個改動之前。這個迴圈之所以讓人覺得慢，恰恰是因為它從不讓你積累那種債務。

“縱向切片”是什麼意思

一個縱向切片是功能裡一道薄薄的、貫穿你整個應用的片段——一點 UI、一點邏輯、一點資料——而不是一次只構建一整層。

誘人卻錯誤的做法（橫向）：

- 先把整個資料庫 schema 建好。

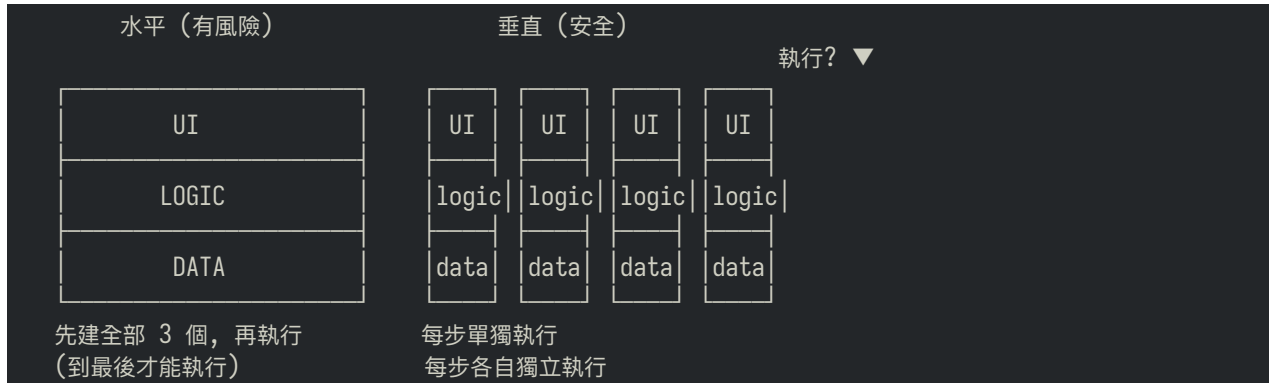
- 然後把所有後端邏輯寫完。
- 然後把整個 UI 做完。

三者全部做完之前，你什麼都跑不起來，所以沒法及早抓住問題。

更好的做法（縱向）：

- 第 1 步：一個按鈕，用假資料做出這件事最簡單的可能版本。
- 第 2 步：讓它用上真實資料。
- 第 3 步：處理邊界情況。

把兩者並排放著看，差別最清楚。橫向一次只造一整層，到最後才能執行；縱向每一步都造出一條貫穿所有層的、可執行的薄帶：



每個切片都可執行。這正是讓應用始終保持可執行的關鍵。

一個判斷好切片的實用測試：**你能演示它嗎？**如果你能指著螢幕說“看這個”，那它就是一個真正的縱向切片。如果唯一誠實的演示是“嗯，schema 現在搭好了”，那你構建的是一個橫向層，而它對不對，你要到很久以後——通常是在最糟的時刻——才會知道。

一個實戰示例：“收藏”功能

我們來端到端地做一個小功能：使用者可以把列表裡的條目標記為收藏，並看到自己收藏了多少個的計數。

切分它

我會這樣把它拆成縱向步驟：

1. 在每個條目旁邊加一個星標按鈕（暫時什麼都不做，只是顯示出來）。
2. 點選星標切換它的亮/滅外觀（僅狀態，不儲存）。
3. 顯示一個反映這些切換的“3 favorites”計數器。
4. 儲存收藏，讓它在頁面重新載入後依然存在。

注意每一步都可獨立執行、可見。你在其中任何一步之後停下，都仍然擁有一個可執行的應用。

第 1 步：按鈕

一條好的提示會點明檔案、改動，以及不要碰什麼：

在 `ItemCard` 元件 (`src/components/ItemCard.jsx`) 中，給每張卡片的右上角加一個星標按鈕。它應該渲染一個輪廓星標圖示。先別接上任何點選行為——只讓它顯示出來。保持卡片其餘的佈局不變。

執行應用。看到星標了嗎？好。提交：

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

第 2 步：切換狀態

讓星標按鈕在被點選時在填充和輪廓之間切換。用本地元件狀態來追蹤它——先別儲存到任何地方。填充的星標表示已收藏。

可以把狀態 (state) 想成應用的短期記憶——此刻什麼是真的，比如某顆星星現在是否處於填充狀態。現在你在同時處理狀態和 **UI**，這很正常。(UI，即使用者介面，就是應用中人能看到、能點選的那一部分。) AI 大概會加上類似這樣的東西：

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

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

點幾下星標。它們會切換嗎？提交。

第 3 步：計數器

在條目列表頂部加一個 "★ N favorites" 計數器。它應該統計當前有多少個條目的星標被填充。如果需要，把收藏狀態上提到列表元件，好讓計數器能看到它。

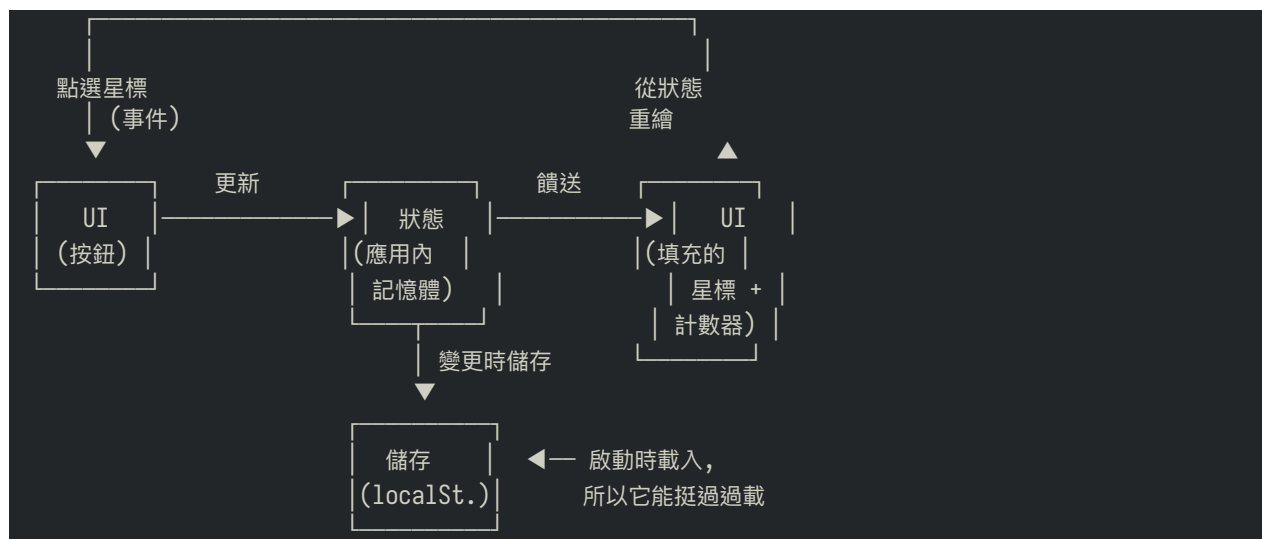
這一步可能要求 AI 重構狀態所在之處——這在預料之中，而這恰恰是為什麼你要把它作為獨立的小步驟來做。執行它，切換幾個星標，看著數字變化。提交。

第 4 步：讓它持久化

把收藏持久化，讓它們能挺過頁面過載。現在先用 `localStorage`——每當收藏條目 ID 的列表變化時就儲存它，並在啟動時載入它。

(`localStorage` 就是瀏覽器在訪客自己電腦上保留的一個小記事本，應用可以把東西記下來，重新載入之後再找回來。) 重新載入頁面。還是收藏狀態嗎？你剛剛交付了一個功能，一次一個可執行的切片。

來看看這個完成的功能裡各部件是怎麼互相對話的。一次點擊向下流入狀態，螢幕從狀態重繪，而狀態被映象到儲存裡，於是它在重新載入後依然存活：



不必看懂每一行也能讀 *diff*

迴圈的第 4 步是“審查改動”，也是人們最愛含糊帶過的一步。要做出有用的審查，你並不需要能自己寫出那些程式碼。你只需要會掃一眼一小組危險訊號。

當 AI 把改動交給你時，看它最清爽的方式就是 diff——精確地列出新增和刪除的行：

```
git diff
```

以 + 開頭的行是新增的，以 - 開頭的行是刪除的。像個審查者那樣去讀，而不是像個編譯器。你要找的是這類東西：

- 它有沒有動了不該動的檔案？你要的是計數器，為什麼 auth.js 會出現在 diff 裡？
- 它有沒有刪掉看起來重要的東西？一行你認不出的被刪除的行 (-) 值得一問。
- 它有沒有悄悄改了你沒要求的行為？新的預設值、被改的條件、“貼心”的額外功能。
- 形態和你的訴求對得上嗎？一句話的請求卻產出了兩百行，這值得再看一眼。

如果有什麼看著不對，你不必自己去修——直接問就行：“你為什麼改了 auth.js？我只要了計數器。”一半的時候，AI 會說“說得對”然後撤銷它。光是這個習慣，就比你盯著自己沒完全看懂的程式碼看再久，都能抓住更多的 bug。

何時放手讓 AI 跑，何時手動掌舵

不是每一步都需要同樣緊地握住方向盤。

在以下情況放手讓 AI 自由跑：

- 步驟小且定義清晰（“加一個按鈕”）。
- 這是一個 AI 顯然見過上千次的常見模式。
- 出錯的代價低且明顯（你一執行應用就會看到）。

在以下情況手動掌舵——更小的提示、更細的審查：

- 改動觸及你不容易恢復的資料（刪除記錄、遷移）。
- AI 反覆在原地”修”同一個東西轉圈。
- 行為很微妙，光掃一眼螢幕看不出來（金額計算、許可權、鑑權）。
- 你已經試了兩次，而它正在變得更糟，而非更好。

一條可靠的規則：爆炸半徑越大，步子邁得越小。

讓應用始終保持可執行

這是讓其他一切成為可能的那門紀律：

- 每一步之後都執行。不是每三步。是每一步。
- 只提交可執行的狀態。一次提交就是一個你能回到的存檔點。如果某一步弄壞了某個你在一兩分鐘內修不好的東西，就把它扔掉（git restore . 或回退），用更清晰的訴求重新提示——別在一個壞掉的基礎上再疊一個補丁。
- 每步只關注一件事。如果你心癢想”順手把那另一個東西也快速修一下”，忍住。先完成當前切片，提交，然後開始一個新步驟。
- 描述 bug，別去診斷它。當某個東西壞了，把你看到的原原本本告訴 AI：“點了第二個星標之後，計數器顯示的是 1，不是 2。”讓它去找原因。

小步提交

小提交是你的安全網。每一個都應是一個單一、連貫的步驟：

```
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，你幾乎什麼都不會損失。又大又少的提交，正是人們最終害怕去碰自己程式碼的根源。

當一個切片太大時

有時你挑了“一個步驟”，結果它還是出岔子——AI 重寫了應用的一半，diff 讀都讀不下來，一執行就同時蹦出三個新 bug。這幾乎從不是運氣不好。這是一個訊號：切片太大了。

解法是把它切得更薄。如果“加上帶持久化的收藏”炸了，那它就不是一個切片；它是整個功能披著一條單一提示的偽裝。退回到你上一個可用的提交，把它拆開：

```
# 別一口氣要求這個：
"加一個帶計數器和 localStorage 持久化的收藏功能。"

# 改成要求這個，每個提交一件：
"給每個條目卡片加一個星標按鈕。先別加行為。"
"讓星標用本地狀態切換填充/輪廓。"
"加一個反映已填充星標的計數器。"
"把收藏的 ID 持久化到 localStorage，並在啟動時載入。"
```

一條好用的經驗法則：如果你沒法用一句簡短的話說清一個步驟做什麼，那它就是兩個步驟。而如果一個步驟的 diff 長到一分鐘內掃不完，那下一個步驟就該更小。切分是一項技能，你會在感受上一個切片哪裡切得太厚的過程中，越來越擅長它。

要點帶走

一個功能不是一條巨大的提示——它是一摞小的、可執行的切片。切分它，為一個步驟寫提示，審查、執行、提交、重複。在簡單的事情上放手讓 AI 跑，在任何有風險的地方附近用力掌舵。在每個檢查點都讓應用保持可執行，你就永遠只差一個小小的撤銷，就能回到堅實的地面。

在下一章裡，我們會看看當某一步沒按計劃進行時該怎麼辦——憑感覺除錯。

回顧與練習

要點回顧

- 一個功能是一摞小的縱向切片，而不是一條巨大的提示——在繼續之前，先構建、執行並親眼看到每個切片能跑起來。
- 每次都跑構建迴圈：切片 □ 選一個步驟 □ 寫提示 □ 審查 diff □ 執行應用 □ 提交 □ 重複。
- 像審查者而不是編譯器那樣讀 diff——留意它本不該碰的檔案、悄悄改變的行為，以及遠大於需求的 diff。
- 在小的、廉價的、常見的步驟上放手讓 AI 跑；在影響範圍大的地方手動掌舵（更小的提示、更細的審查）。
- 只把可執行的狀態小步提交——切片出岔子時，扔掉重寫，而不是在壞掉的基礎上再疊一個修復。

動手試試

挑一個你想加進專案的小功能，把它切成 3 到 5 個縱向步驟，每個寫成一句短句。只構建第一個步驟：為它寫提示，執行 git diff 略讀改動，執行應用確認它能跑，然後用一行訊息提交。眼下就停在這裡。目標是感受一次乾淨的迴圈轉動——一個可執行的切片、一份你真正讀過的 diff、一個你能回到的提交。

```
我想加這個功能：[用一兩句話描述這個功能]。
先別寫任何程式碼。首先，把它切成 3 到 5 個垂直步驟，每一步都是我能自己
構建、執行並看到它獨立工作的東西。把它們排好序，讓應用在每一步之後
都還能用。每一步給我一句話的描述，並說出你會改的第一個檔案。然後
等著——我會讓你一次構建一步。
```

用 AI 除錯



你構建的每個程式都會出問題。這不是你做錯了的訊號——這是軟體的正常狀態。Bug 不是技藝的失敗；它們是真實的程式碼遇上真實的資料、真實的使用者，以及沒人想像得到的真實邊緣情況時所發生的事。一個令人沮喪的下午和一個五分鐘的修復之間的區別在於方法。當你用 **vibe coding** 來除錯時，盯著堆疊跟蹤直到看懂的人不是你——堆疊跟蹤就是程式崩潰時打印出來的錯誤報告，我們稍後會拆開細說。你是導演：你復現問題，收集正確的證據，然後把一個緊湊、框定清晰的案例交給 AI，讓它去做繁重的工作。本章將給你一個可重複的流程來做到這一點——無論 bug 是一行拼寫錯誤，還是一個纏繞了三層深的競態條件（**race condition**），這個流程都管用。（競態條件是一種時序 bug：兩件事以你沒料到的順序發生，就像兩個人同時去搶最後一個座位。）

核心心態

初學者犯的最大錯誤，就是把“它不工作，修好它”貼上到對話方塊裡，然後祈禱。AI 看不到你的螢幕、你的資料，也不知道你期望發生什麼。它只知道你告訴它的東西。當你什麼都不給它時，它只能做它唯一能做的事：猜測——而一個碰巧錯了的自信猜測，比根本沒有答案更危險，因為你會照著它去行動。用 AI 除錯主要是關於給它喂正確的上下文，這樣它才能推理，而不是猜測。

一份好的 bug 報告——無論是給人還是給 AI——都會回答三個問題：

- 你做了什麼？（觸發它的步驟）
- 你期望發生什麼？
- 實際發生了什麼？（錯誤、錯誤的輸出、崩潰）

如果你能回答這三個問題，你已經領先於大多數人了。這三點之所以重要，是因為一個 bug 按定義就是你所期望的和實際發生的之間的缺口。AI 找不到一個它看不到兩側的缺口。“它壞了”只描述了一側；“我期望一個歡迎介面，結果得到一個空白頁

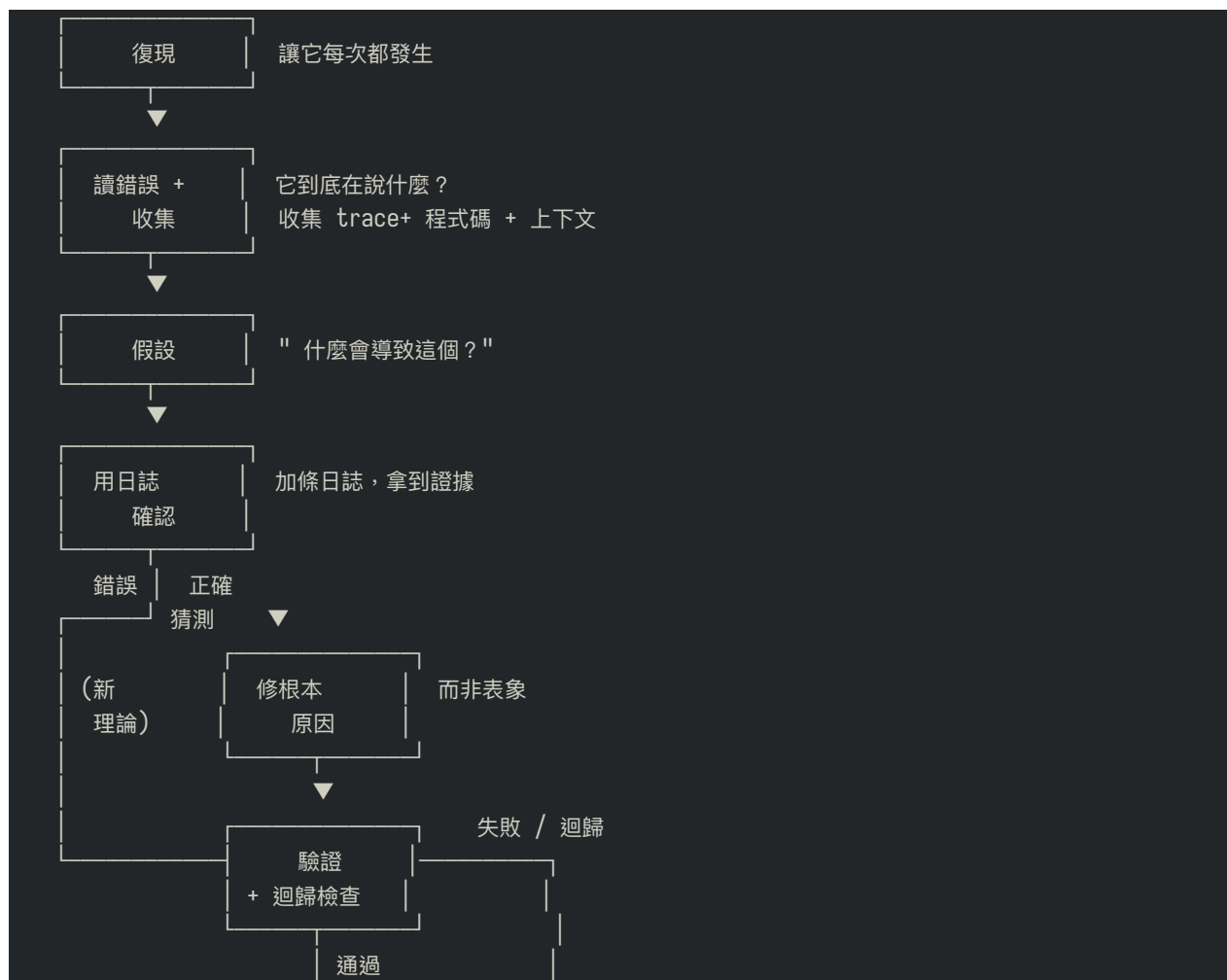
面和這個錯誤”描述了兩側——於是缺口幾乎會自報家門。

一份除錯清單

按順序逐項完成這些步驟。直接跳到”修好它”正是讓 AI 兜圈子的原因。

1. **可靠地復現它。**找到每次都能讓 bug 出現的確切步驟。一個你無法復現的 bug，是你無法確認自己已經修好的 bug——你將永遠只是在猜測它是否消失了。如果它是間歇性出現的，去捕獵那個讓它切換的變數：某個特定輸入、登入還是登出的狀態、緩慢的網路。
2. **閱讀真正的錯誤資訊。**不要略讀。錯誤資訊和堆疊跟蹤的頂部通常會指出問題的檔案和行號。初學者常常因為某句話看起來嚇人，就略過了螢幕上最有用那句。
3. **收集上下文。**收集錯誤文本、堆疊跟蹤、相關程式碼，以及一句話描述你期望的結果。
4. **把完整的畫面交給 AI。**把以上所有內容貼上在一條訊息裡。讓它形成一個假設。
5. **形成假設，而不是修復方案。**在”修好這個”之前先問”什麼可能導致這個？“。基於錯誤理論的錯誤修復只會浪費時間——有時還會在第一個 bug 之上疊加一個第二個 bug。
6. **新增日誌來測試假設。**在修改程式碼之前，先確認程式碼實際在做什麼。
7. **二分查詢問題。**通過反覆將可疑區域對半切分，縮小故障發生的位置。
8. **修復根本原因，而非表象。**確保修復針對的是它為什麼出問題，而不僅僅是可見的故障。
9. **驗證修復並檢查回歸。**重新運行復現步驟。然後確保附近的功能仍然正常工作——所謂迴歸，就是一個修復悄悄弄壞了原本好好的東西，就像補好一處漏水，旁邊的管子卻爆了。

這份清單其實就是一個你不斷繞行、直到 bug 消失的迴圈。每個方框都把證據交給下一個，而一次失敗的驗證只會帶著你學到的東西重新繞一圈：



▼
✓ 完成

◀———▶
(繞回去再來)

這個順序並非隨意。每一步都產生下一步所需要的證據。復現給你提供可讀的東西；閱讀給你提供可收集的上下文；上下文給 AI 提供可做假設的素材。跳過一步，你就是在給下一步喂一個窟窿。

給 AI 喂正確的上下文

這是你大部分槓桿所在之處。當你報告一個 bug 時，請包含：

- 逐字的錯誤資訊。原樣複製——不要轉述。轉述一個錯誤，就像在電話裡描述一張照片：你會漏掉那個最關鍵的細節。
- 堆疊跟蹤，尤其是頂部的幾行。
- 相關程式碼——丟擲錯誤的函式，以及它呼叫的任何東西。
- 你期望的結果與實際發生的結果對比。

下面是一個強力除錯提示的真實樣子：

我提交登錄檔單時報錯了。下面是完整的輸出：

```
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)
```

復現步驟：填好表單，點選 "Sign up"。

期望：一條 "Welcome" 訊息。

實際：頁面變成空白，上面這個錯誤出現在控制台裡。

這是 validate.js (第 14 行附近)：

[貼上這個函式]

這是呼叫它的 handleSignup 函式：

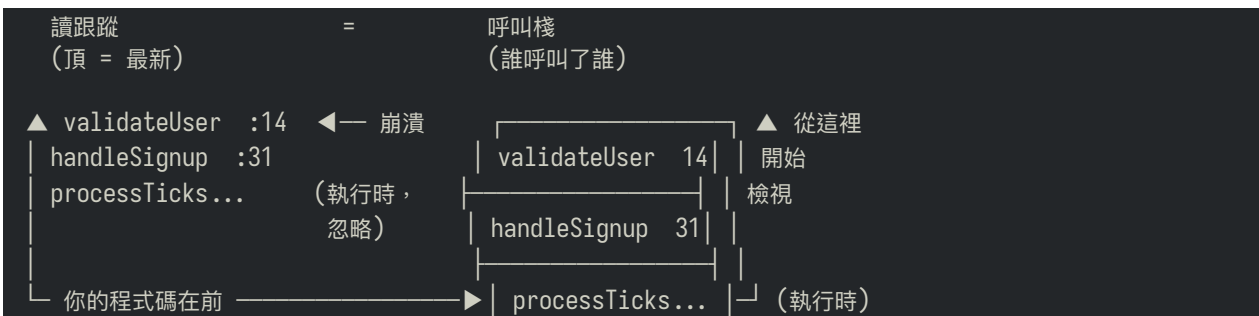
[貼上這個函式]

什麼可能導致這個，以及在改動任何東西之前我該如何確認它？

注意最後一行。你是在請求一個假設和一個確認步驟，而不是立即修改。這能讓 AI 保持誠實。

知道如何閱讀你正在貼上的那段跟蹤，會很有幫助。堆疊跟蹤是一份按最新到最舊排列的呼叫歷史：最頂部那行是程式實際爆炸的地方，下面每一行都是呼叫了它上面那個函式的函式。所以在例子裡，validateUser 是 email 從某個 undefined 的東西上被讀取的地方，而 handleSignup 是呼叫 validateUser 的那一方。你自己程式碼下面的幀——比如 processTicksAndRejections——屬於語言執行時；通常你可以忽略它們。從上往下讀，直到碰到第一行指向你的檔案，那幾乎總是該開始檢視的地方。

把同一段跟蹤當作一摞呼叫來讀，順序就一目瞭然了。崩潰在最頂上，下面每一幀都是呼叫了它正上方那一幀的那一方：



向下讀，直到第一行是你的檔案

閱讀你看不懂的錯誤

遲早你會遇到一個看起來像字母湯的錯誤——一個很長的型別名、一個全是框架內部的堆疊、一條用你從沒見過的行話寫的訊息。別讓它把你卡住。錯誤是資料，不是對你能力的判決。當訊息本身晦澀難懂時，把它貼上上去，讓 AI 在修復之前先翻譯：

我看不懂這個錯誤。在提出任何修復方案之前，請用通俗的語言解釋：
它實際上在說什麼，它指向哪一行，
以及通常什麼東西會導致這個確切的錯誤。

[貼上完整的錯誤和堆疊跟蹤]

先要求用通俗語言解釋含義，能做到兩件事。它教會你這個模式，這樣下次再看到這個錯誤時你自己就能認出來——而幾個月下來，這正是你真正學會閱讀自己堆疊的方式。它還迫使 AI 把修復建立在錯誤所說的內容之上，而不是跳到一个通用補丁。當心最常見的錯誤陷阱：一個指向第 14 行的錯誤，往往是在完全別的地方被引起的。跟蹤裡的那行是程式注意到問題的地方，並不總是問題開始的地方。

假設、日誌和二分查詢

一旦 AI 提出了一個原因，不要直接接受。用低成本的方式測試它。最便宜的測試通常是一行日誌。

讓 AI 新增臨時日誌：

“在我們改任何東西之前，新增一個 `console.log` (或 `print`)，在第 14 行之前顯示 `user` 的值，這樣我們就能看到它是不是真的為 `undefined`。”

執行它，把輸出貼上回來。現在你擁有的是證據，而不是一個理論。也許 `user` 確實是 `undefined`——這就確認了假設。也許它沒問題，真正的元兇在上一層。無論哪種情況，你都向前推進了。調查時要慷慨地打日誌：變數的值，以及一個標籤，說明它是哪個變數、日誌在哪裡。一個打印出 `undefined` 的光禿禿的 `console.log(user)`，遠不如 `console.log("user at validate.js:14 =", user)` 有用。還有，bug 修好後記得把臨時日誌拔出來——遺留的除錯噪音本身就是一小攤爛攤子。

當 bug 藏在一個很長的流程中的某處，而你說不清在哪裡時，使用二分查詢。把可疑區域對半切分：在中點新增一條日誌。

核心思路：每次檢查都丟掉剩下一半，所以一個龐大的搜尋空間會迅速塌縮。下面這個 bug 藏在第 1 步和第 8 步之間的某處——三次檢查就把它逼了出來：

```
步驟:  1  2  3  4  5  6  7  8      bug 藏在這裡某處
        ///////////////////////////////////////////////////

檢查 1 — 在 4 打日誌 — OK —▶  bug 在 4 之後，棄掉 1-4
                               /////////////// (剩 5 6 7 8)

檢查 2 — 在 6 打日誌 — BAD —▶  bug 在 6 處/之前，棄掉 7-8
                               /////////////// (剩 5 6)

檢查 3 — 在 5 打日誌 — OK —▶  bug 在第 6 步    ✓ 找到

8 步 → 3 次檢查。  1000 步 → 約 10 次檢查。
```

那裡的值看起來正確嗎？那麼 bug 在它之後——搜尋那一半。那裡的值已經看起來不對了嗎？那麼 bug 在它之前——搜尋那一半。每一步都把搜尋空間減半，所以即使是一千行的路徑，也能在大約十次檢查內縮小範圍。明確地告訴 AI：“幫我二分排查——我應該在哪個中點打日誌？”同樣的技巧不僅對程式碼有效，對歷史也有效：如果某個功能上週還能用、今天壞了，就對你的提交做二分查詢，找到引入這個 bug 的那個確切改動。

根本原因與表象

除錯中最誘人的陷阱，就是那種讓錯誤消失卻沒修復原因的補丁。如果 `user` 是 `undefined`，你用 `if (user) { ... }` 來“修復”它，錯誤是停止了——但現在註冊會悄無聲息地什麼都不做，這更糟糕：一次響亮的崩潰至少告訴了你哪裡出了錯，而現在什麼都不告訴你了。真正的問題是為什麼 `user` 在這裡是 `undefined`？也許資料庫查詢失敗了。也許某個欄位被重新命名了。也許上游缺了一個 `await`。

當 AI 提出一個修復方案時，問它：

- “這是在修復原因，還是在掩蓋表象？”
- “這個值出錯的最初原因是什麼？”
- “同樣的根本原因會不會破壞其他東西？”

修復根本原因通常會讓程式碼更簡單或更清晰。修補表象通常會新增一個守衛、一個吞掉錯誤的 `try/catch`，或者一個掩蓋真正問題的預設值。對第二種要保持警惕。有一個值得學會的跡象：如果一個修復新增的程式碼，其唯一職責是防止崩潰，而不是做正確的事，那你很可能是在修補表象。真正的修復傾向於把那個不可能的狀態徹底移除，而不只是熬過它。

當 AI 開始兜圈子

有時 AI 提出一個修復方案，不起作用，它再提出另一個，那個也不起作用，於是你已經陷入三層“試試這個”的迴圈裡了。這是一個訊號，而不是死衚衕。這樣做：

- **停下來，重置上下文。**用一條全新的訊息開始，總結你現在確定知道的一切（由日誌確認的），而不是你嘗試過的一切。一段塞滿了失敗嘗試的長對話，會主動毒化 AI 的下一個回答——它會不斷地對自己歷史裡的那些死衕衕做模式匹配。
- **說明你已經排除了什麼。**“我們確認了 `user` 是 `undefined`，資料庫查詢返回 `null`。查詢字串看起來是對的。所以問題在查詢的上游。”這能阻止 AI 重新建議那些死衕衕。
- **讓它解釋，而不是修復。**“一步一步帶我走一遍，`user` 是如何從請求拿到它的值併到達第 14 行的。”強制它解釋，往往能暴露出缺口。
- **新增更多日誌，而不是更多猜測。**當理論用盡時，證據才是前進的方向。
- **縮小問題。**讓 AI 幫你構建一個仍能復現 bug 的最小程式碼片段。通常，一旦噪音消失，bug 就會變得顯而易見——而如果它在小版本里復現不出來，你就知道了原因藏在你刪掉的部分中。

在保留修復之前先理解它

一個能用卻你不理解的修復，是一筆貸款，而不是一個解決方案——而當附近某個東西下次壞掉時，利息就到期了。因為 AI 替你打字，所以很容易接受一個綠色的對勾就繼續往前，卻從不知道程式碼為什麼現在是綠的。抵制這種衝動。在你保留一個修復之前，讓 AI 為它交代清楚：

- “用一兩句話解釋為什麼這能修好它。”如果解釋含糊——“這讓它更健壯”“這處理了邊緣情況”——就回懟回去。真正的解釋會點明具體的原因，以及這個改動如何把它移除。
- “那個 **bug** 究竟是什麼，用因果來說？”你應該能用自己的話把答案複述出來。如果不能，你就還沒理解自己的程式。
- “還有什麼會觸及這條程式碼路徑、需要我重新測試？”這把一個修復變成了一次迴歸檢查。

這就是“理解你所交付之物”哲學應用在它最難的那種情況上。你不需要能從零寫出這個修復——那是 AI 的活兒。但你確實需要把因果的故事攥在腦子裡，因為當它再次壞掉的凌晨兩點，在場的人是你，而那時唯一幫得上忙的，就是知道這東西實際上是怎麼運作的。

要點

用 AI 除錯不是關於知道答案。它是關於執行一個有紀律的流程：復現、收集上下文、提出假設、用日誌確認、二分查詢，並修復根本原因——然後把這個修復理解到足以誠實地保留它。作為導演，你的工作是讓調查始終紮根於證據，並識別出迴圈何時已經陷入停滯。做到這一點，大多數 bug 就不再是謎團，而只是構建真實軟體過程中的又一個步驟。

回顧與練習

要點回顧

- bug 是你預期與實際發生之間的落差——把兩邊連同確切的復現步驟都給 AI，絕不要只說“它壞了”。
- 把錯誤一字不差地貼上去，並附上棧追蹤的頂部；指向你自己檔案的第一行幾乎總是起點。
- 改動前先確認：加一行帶標籤的日誌來驗證假設，並通過二分查詢（在程式碼或提交歷史中）讓搜尋空間每一步減半。
- 修復根本原因，而不是表象——只能阻止崩潰的守衛或 try/catch 通常只是把真正的問題蓋住。
- 當 AI 兜圈子時，把上下文重置為你確定知道的內容，讓它去解釋而不是去修，並補上證據而不是猜測。

動手試試

下次有東西壞掉時，在請求修復之前先寫好 bug 報告。在一個全新的對話裡給 AI 三樣東西：復現步驟、你的預期，以及實際發生的情況——錯誤一字不差地複製，並包含棧追蹤的頂部幾幀。然後，不要說“修好它”，而是請它給出一個假設以及一個廉價的確認方式（通常是一行日誌）。執行那個檢查，把輸出貼回去，然後才讓 AI 提出修復方案。感受一下這條路比“它不工作，幫我”要直多少。

我有一個 bug。在提出任何修復之前，給我一個假設以及一個低成本的確認方式（比如一行日誌）——先別改程式碼。

復現步驟：[你做了什麼]

期望：[本該發生什麼]

實際：[實際發生了什麼]

完整的錯誤和堆疊跟蹤：

[一字不差地貼上]

相關程式碼：

[粘貼出錯的函式，以及它呼叫的任何東西]

什麼可能導致這個，以及在我們改動任何東西之前我該打什麼日誌來確認它？

伺服器基礎設施



你用 **vibe coding** 寫出了某個能在自己筆記本上執行的東西。現在你想讓其他人也能用它。那道鴻溝——“在我的機器上能跑”和“在網際網路上能跑”之間的鴻溝——正是基礎設施所關心的。好消息是：你不需要一份 DevOps 職業生涯就能跨越它。你需要的是一個能用的心智模型，以及願意讓 AI 來處理那些瑣碎的配置。

本章揭開託管這片天地的神秘面紗，用平實的語言給你講清各種取舍，並交給你一條預設路徑，讓一個單打獨鬥的構建者從原型走向生產，而不至於淹沒在各種選擇之中。

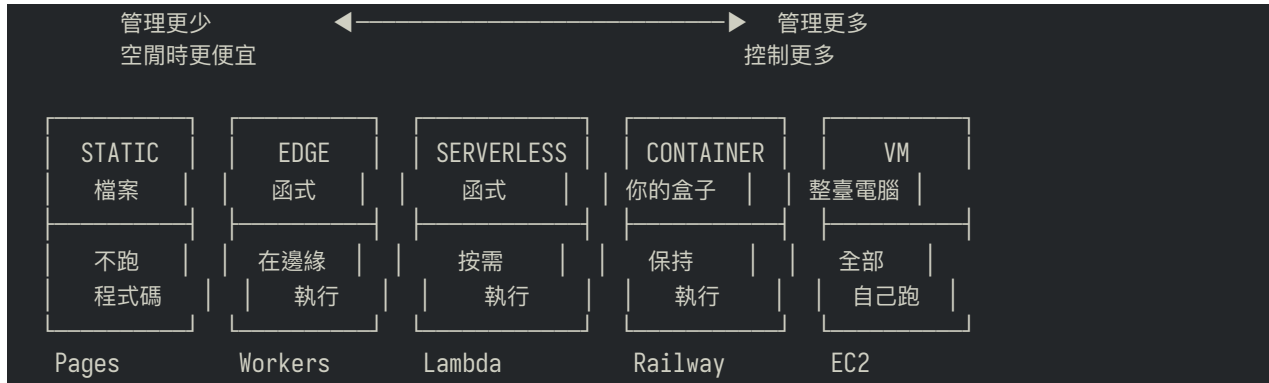
程式碼可以待的五個地方

地球上幾乎每一個託管產品，都是這五個類別之一的變體。學會這些，那些營銷頁面就不再讓人困惑了。

- **靜態託管 (Static hosting)** —— 把純檔案 (HTML (HyperText Markup Language, 網頁的骨架)、CSS (Cascading Style Sheets, 樣式裝扮)、JavaScript、圖片) 直接交付給瀏覽器。沒有伺服器執行你的程式碼；它只是把檔案遞過去。把它想像成網際網路上的一個資料夾。因為沒有什麼會崩潰，也沒有什麼需要打補丁，所以這是你能放到網際網路上的最可靠的東西。例子：GitHub Pages、Cloudflare Pages、Netlify、Vercel (靜態模式)。
- **邊緣函式 (Edge functions)** —— 遍佈全球、執行在物理上靠近你使用者的伺服器上的微小程序碼片段。它們幾乎瞬間啟動，非常適合輕量邏輯：身份驗證檢查、重定向、小型 API (Application Programming Interface, 應用程式設計介面)——一個程式與另一個程式對話的門口) 呼叫。要注意的是：它們執行在一個功能精簡的環境裡，所以並不是每個庫都能在那裡工作。例子：Cloudflare Workers、Vercel Edge Functions、Deno Deploy。
- **無伺服器函式 (Serverless functions)** —— 按需執行、空間時消失的程式碼。你按請求計費，而不是按小時。你寫一個函式；平臺負責管理機器。在單個請求需要執行好幾分鐘之前都很好用——大多數平臺會在超時 (通常 10 到 60 秒) 後切斷函式。例子：AWS Lambda、Google Cloud Functions、Vercel Functions。

- **容器 (Containers)** ——你的應用加上它的整個環境（正確的語言版本、庫、系統工具）被打包進一個叫做容器的可移植盒子裡。平臺讓那個盒子保持執行。更多控制權，稍多一點配置。定義那個盒子的那一個檔案就是 Dockerfile，而 AI 寫得很好。例子：Fly.io、Railway、Render、Google Cloud Run。
- **虛擬機器 (VMs)** ——你在資料中心裡租用的一整臺計算機。所有東西——作業系統、安全更新、Web 伺服器，全部——都由你自己安裝和管理。最大的控制權，最大的責任。例子：AWS EC2、DigitalOcean Droplets、Hetzner。

把這五個類別並排來看，越往下，你用更少的管理換取更多的控制：

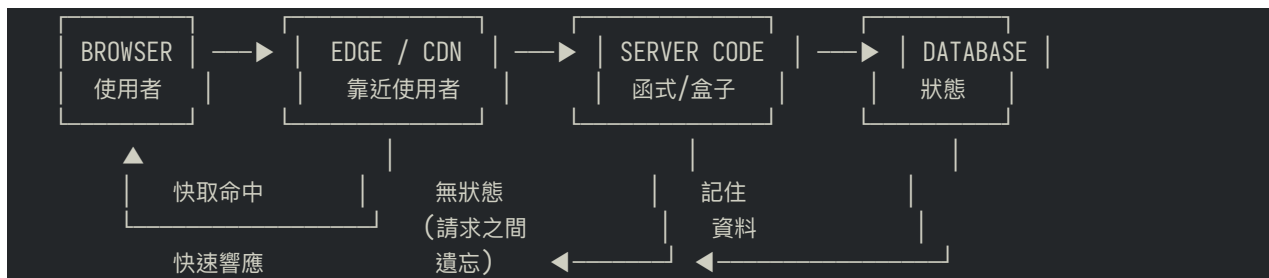


關於成本的一條粗略經驗：平臺替你管理得越多，沒人用你應用時就越便宜，而當所有人都在用時它替你操的心也越多。

幾個行話術語，破譯一下

- **冷啟動 (Cold start)** ——當代碼有一段時間沒執行過時，平臺必須把它喚醒。那第一個請求會更慢（從幾分之一秒到幾秒）。邊緣函式幾乎沒有這個問題；無伺服器函式有時有；常駐執行的容器和 VM 則沒有。對大多數應用來說它是看不見的；但對一個要求利落的結賬流程來說，它可能很要緊。
- **擴充套件 (Scaling)** ——在處理更多使用者時不至於崩潰。自動擴充套件意味著平臺替你增加容量。手動擴充套件意味著你點選按鈕，否則它就在負載下直接掛掉。“縮減到零 (scales to zero)”是你會經常看到的一個相關說法：當沒人用你的應用時，它不花錢。
- **狀態 (State)** ——需要持續保留的資料（一個數據庫、上傳的檔案、一個使用者會話）。靜態層和無伺服器層通常是“無狀態的”——它們在請求之間忘記一切——所以狀態存放在一個獨立的資料庫或儲存服務裡。這就是為什麼幾乎每個真實應用都是“你的託管加上某處的一個數據庫”，而不是一個魔法盒子。
- **區域 (Region)** ——你的程式碼和資料在物理上所處的世界的哪個部分。它影響速度（越近越快），有時還影響法律（某些資料必須留在特定國家）。對第一個專案，選離你使用者最近的區域，然後繼續往前走。

下面是一個請求所走過的完整旅程，以及每一塊各自待在哪裡。瀏覽器與就近的邊緣對話，邊緣交付快取的檔案或喚醒你的程式碼，而程式碼在需要記住什麼時就伸手去找資料庫：



取捨對照表

託管型別	成本（空間 □ 繁忙）	擴充套件	冷啟動	複雜度	最適合
靜態	免費 □ 非常便宜	自動，毫不費力	無	最低	營銷網站、文件、部落格、前端
邊緣函式	免費額度 □ 便宜	自動，全球性	可忽略	低	身份驗證、重定向、輕量 API、個性化
無伺服器	空閒時 \$0 □ 隨使用量擴充套件	自動	有時明顯	低到中	API、webhook、突發/不可預測的流量
容器	執行期間付費	大致自動，可配置	無（如果保持溫熱）	中	完整後端、長時間執行的任務、自定義執行時
VM	全天候付費	手動	無	最高	特殊硬體、遺留軟體、完全控制

規律是：隨著你在表格中往下移動，你用簡單性和低空間成本換取控制權和靈活性。大多數 vibe coder 都應該努力待在靠近表格頂部的地方。

何時選哪個

- 只是一個網站或前端？靜態託管。搞定。它免費、快速，而且幾乎不可能弄壞。
- 需要一點後端邏輯——一個聯絡表單、一個保密的 **API key**、一次身份驗證檢查？邊緣函式或無伺服器函式。你待在慷慨的免費額度裡，沒人用時就縮減到零。
- 一個帶資料庫、後臺任務，或期望有一個長時間執行伺服器的框架的真實應用？一個像 Railway、Render 或 Fly.io 這樣的容器平臺。對大多數“真正的產品”專案來說，這是最佳甜蜜點。
- 你需要某個特定的 **GPU**、特殊的系統軟體，或完全的控制權？一臺 VM。只有當上面的某個選項確實做不了這件事時才走到這裡，因為現在你就是系統管理員了。

一條有用的規則：選擇能完成工作的最簡單的選項，只有當你撞上一堵真實的牆時才往下移動一個層級——而不是一堵想像中的牆。

一堵“真實的牆”實際上長什麼樣？幾個誠實的例子：你的構建產出了一個需要保持執行的伺服器（一個維持著即時連線的聊天後端），所以靜態行不通。或者一個函式老是撞上它的超時，因為一個請求要幹真活（縮放影片、生成 PDF），於是你晉級到容器。或者你免費額度的請求上限被真實流量實實在在地耗盡了。注意這些的共同點：每一個都是一次測量，而不是一種直覺。“它將來可能會變大”不是一堵牆。

AI 如何串聯部署與配置

這正是 vibe coding 大放異彩的地方。部署配置恰恰是那種精確、文件完善、充滿樣板程式碼的工作，AI 擅長此道。你來指揮；它來編寫。

有效的提示聽起來像這樣：

- “為這個 Node 應用新增一個 Dockerfile，以及一個用來部署它的 Railway 配置。”
- “寫一個 GitHub Actions 工作流，在每次推送到 main 時把 dist/ 資料夾部署到 Cloudflare Pages。”
- “我的無伺服器函式遇到了冷啟動超時——可能哪裡出了問題，我該怎麼修復配置？”

下面是 AI 會為你生成的那類檔案——一個部署工作流，每當你推送程式碼時就自動釋出一個靜態網站：

```
# .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

```

你不需要背下這個語法。你需要認出它做了什麼：在每次推送到 main 時，它檢出你的程式碼，構建它，併發布結果。當出問題時，你把錯誤貼上回給 AI，讓它修復配置。

很多構建者一開始乾脆跳過 GitHub Actions 檔案，直接在自己的終端裡用一條命令部署。要在第一天就看到東西上線，這往往是最快的路。同樣的 Cloudflare Pages 部署，從命令列看是這樣：

```

# 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

```

第一次執行會讓你通過瀏覽器登入；之後就是一條命令的事。取捨很簡單：終端命令適合“現在就發”，GitHub Actions 檔案適合“永遠自動地發”。大多數專案從前者開始，等部署成了家常便飯再加上後者。看你處在哪個階段，就讓 AI 給你那個合適的。

幾條能省下真切痛苦的指揮技巧：

- 讓 AI 先解釋再行動。“在寫配置之前，告訴我哪種託管型別適合這個專案，以及為什麼。”這能及早抓住過度工程。
- 讓金鑰遠離程式碼。API key 和密碼要放進平臺的環境變數 / 金鑰儲存中，絕不要放進你提交的檔案裡。讓 AI 使用一個金鑰引用，就像上面的 CF_API_TOKEN 那樣。
- 儘早部署，頻繁部署。在第一天就讓一個“hello world”上線。一條從一開始就在執行的流水線，比一條在釋出時才拼湊起來的流水線要好除錯得多。
- 提前問清楚回滾的辦法。“如果這次部署壞了，我怎麼回到能用的那個版本？”好的平臺會把你之前的部署放在一鍵之遙。在你需要之前就知道有撤銷按鈕，能把凌晨兩點的恐慌變成一個聳肩。

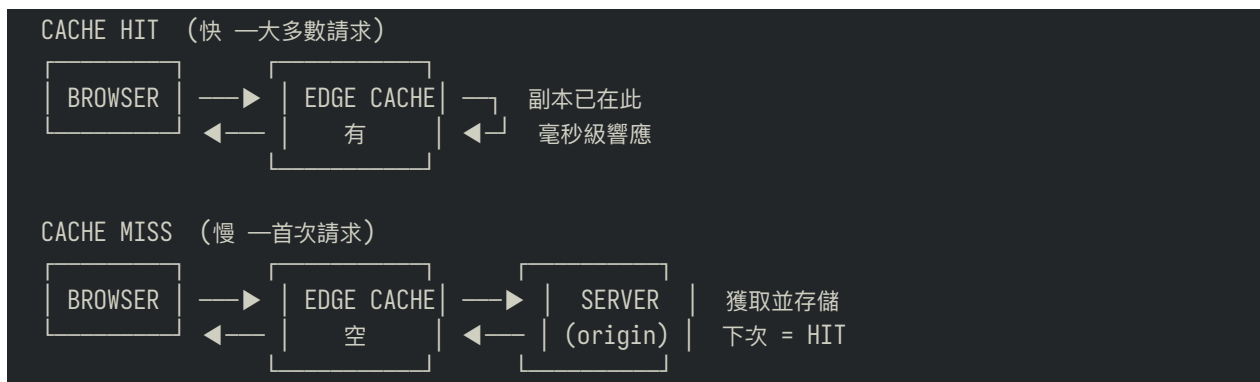
給單打獨鬥構建者的推薦預設路徑

這是一條已經對無數個單人專案奏效的路線。每一步都只在你真正需要時才採取。

1. **原型：靜態 + 一個託管後端服務。**把你的前端放到靜態託管上（Cloudflare Pages、Vercel 或 Netlify）。對於資料和身份驗證，依靠一個像 Supabase 或 Firebase 這樣的託管後端，這樣你就跳過了自己執行資料庫。成本：基本為零。
2. **早期生產：新增邊緣函式或無伺服器函式。**當你需要同伺服器端邏輯時——保密的 API 呼叫、webhook、自定義身份驗證——在同一個平臺上新增函式。仍然縮減到零，仍然便宜，仍然一次部署。
3. **真實產品：把後端遷移到容器平臺。**當應用超出了函式的能力範圍（長時間執行的程序、想要一個持久伺服器的框架、後臺任務）時，把它容器化，部署到 Railway、Render 或 Fly.io。讓 AI 來寫 Dockerfile 和平臺配置。

4. 擴充套件或專門化：到那時才考慮 **VM** 或像 **AWS** 這樣的雲提供商。這是複雜度（以及你的賬單）跳升的地方。大多數單打獨鬥的構建者從不需要這一步，而這是把事情做對的標誌——而不是一種侷限。

CDN（Content Delivery Network，內容分發網路）坐在這條路徑的前面，會根據它是否已有副本而極大地改變速度。快取 HIT 從邊緣即刻應答；快取 MISS 則必須先一路回到你的伺服器：



要避免的錯誤是從第 4 步開始，僅僅因為那是“真正的工程師”據說會用的東西。事實恰恰相反：從簡單開始才是資深的做法。你隨時可以晉級到更多的控制權，但你很難找回那些與你本不需要的基礎設施搏鬥而損失掉的幾個月。

“託管 (managed)”到底意味著什麼（以及你仍然擁有什麼）

託管服務，就是平臺替你打理那些枯燥又危險的部分——伺服器、安全補丁、擴充套件、備份。“無伺服器”“託管資料庫”“平臺即服務 (PaaS)”都是同一筆交易的不同口味：你把運維交出去，自己繼續構建。對一個單打獨鬥的構建者來說，這不是偷懶，而是槓桿。你沒花在打 Linux 安全更新上的小時數，就是你花在真正產品上的小時數。

但“託管”不是“魔法”。無論平臺打理多少，總有幾件事仍然是你的責任：

- **你的資料**。平臺讓資料庫保持執行；但確保你有一份你真正測試過能恢復的備份，是你的事。把“匯出”按鈕按一次，確認那個檔案不是空的。
- **你的金鑰**。API key 住在平臺的金鑰儲存裡，但輪換一個洩露的金鑰，以及永遠不把金鑰粘進公開倉庫或截圖，是你的活。
- **你的賬單**。自動擴充套件很美妙，直到它把你的賬單也自動擴充套件了。設好消費提醒和上限——下一節整節都在講這個，因為這是最讓人疼的那種翻車。
- **你的理解**。當半夜裡有東西壞了，“平臺會處理”只能管到一定程度。你不必是專家，但你應該能讀懂日誌，並把故障向 AI 描述清楚。這就是“理解你所釋出的東西”這一理念濃縮成的一句話。

健康的心態：把運維外包出去，但別把理解外包出去。讓平臺去跑機器；你來當那個知道機器在幹什麼的人。

一份快速的上線前清單

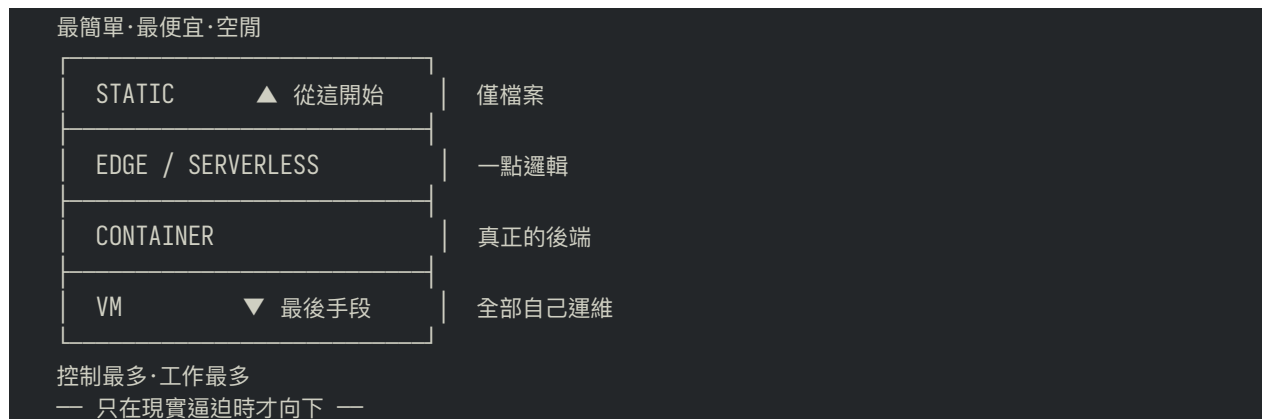
在你告訴別人去訪問你的線上 URL（Uniform Resource Locator——網址）之前，先把這份清單走一遍。這裡沒有一項需要 DevOps 專長——大部分就是給 AI 的一條提示，或者面板上的一次點選。

- **它確實能載入**。在一個全新的瀏覽器（或你的手機）裡開啟真正的 URL，而不只是你本地的預覽。那些“在我機器上能跑”、一上線就在公眾面前 404 的釋出，多到能寫進傳說裡。
- **金鑰不在程式碼裡**。推送之前，在你的倉庫裡搜一下你自己的金鑰。問 AI：“掃描這個專案裡硬編碼的 API key、密碼或 token。”
- **消費上限已設好**。給每一個付費 API 設一個硬上限，給你的託管賬戶設一個賬單提醒。先設上限，再慶祝。
- **HTTPS 已開啟**。HTTPS (HyperText Transfer Protocol Secure) 是網頁基礎傳輸方式的加密版本——它就是給位址列加上小鎖的那個東西。現代平臺會自動做這件事，但確認一下那個小鎖圖示在那兒。瀏覽器會懲罰沒加密的網站。
- **你知道怎麼回滾**。趁現在什麼都沒著火，先把“上一次部署”那個按鈕找到。
- **有辦法聯絡到你**。一個日誌或錯誤上報工具，哪怕是基礎的，好讓問題的第一個訊號不是一個憤怒的使用者。讓 AI 給你的技術棧接上錯誤日誌。

這不是一次全面的生產環境審計——它是一份短清單，專抓那些丟人、燒錢、又很容易避免的錯誤。每次釋出都跑一遍，直到它變成本能。

要點

基礎設施是一架梯子，不是一次跳躍。靜態在頂部，VM 在底部，而你的大部分工作都可以愉快地待在上面的橫檔上。



瞭解這五個類別，選擇能完成工作的最簡單的那個，讓 AI 來寫配置，並儘可能早地把東西上線釋出。剩下的，只不過是在現實——而且只有當現實——強迫你時，往下移動一檔而已。

回顧與練習

關鍵要點

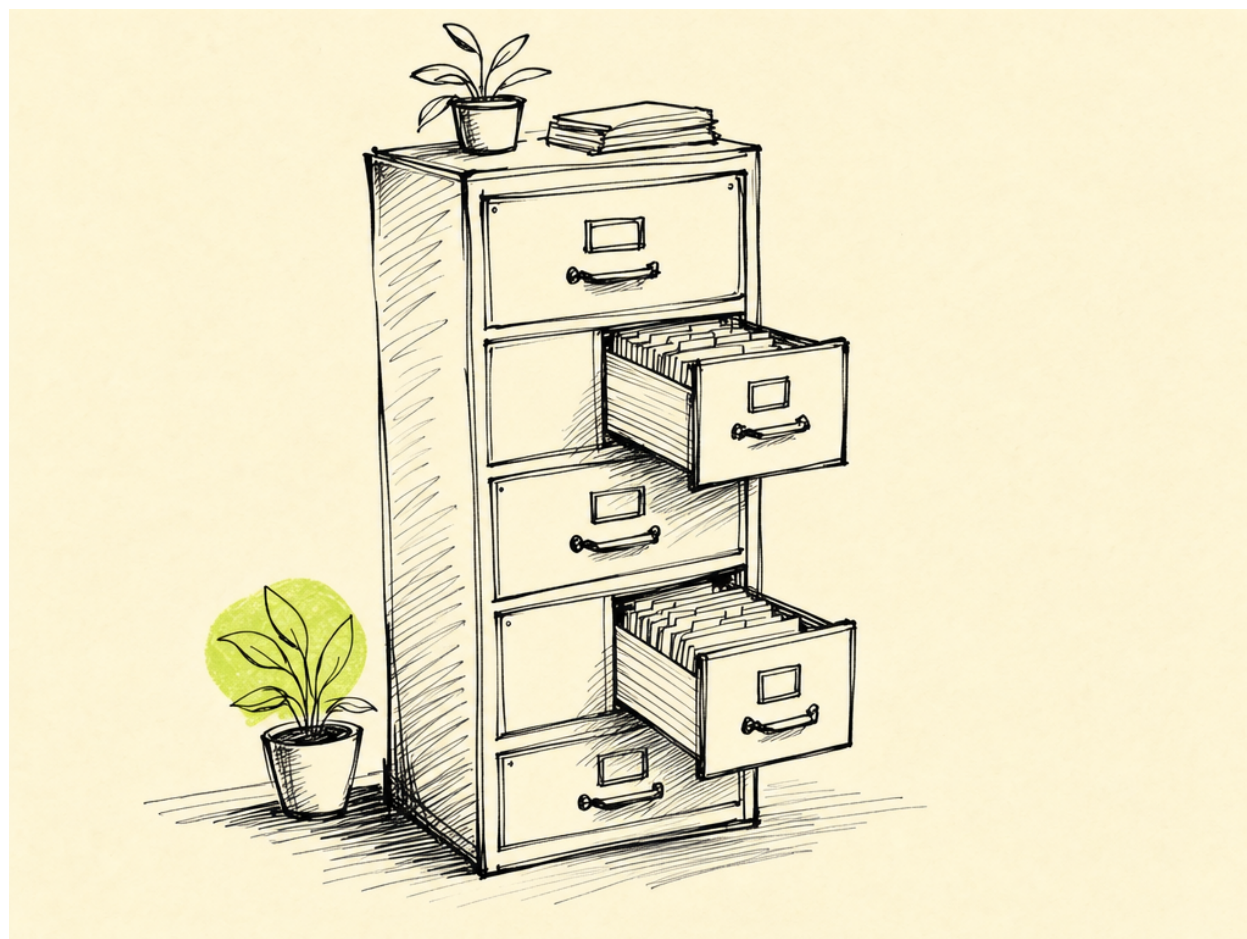
- 基礎設施是一架五級的梯子——靜態託管、Serverless、PaaS、託管容器，以及裸 VM——你應當選擇能完成任務的最簡單的那一級。
- 優先選擇託管服務：把運維（伺服器、補丁、擴縮容、備份）外包出去，把理解留在自己手裡。
- 讓 AI 來寫部署配置和 CI（Continuous Integration，持續整合——每次推送程式碼時自動執行檢查的機器人）工作流，但在信任之前先讀一遍。
- “託管”不是“魔法”——你的資料、金鑰、賬單和理解，依然是你的責任。
- 在上線之前設好支出上限、確認 HTTPS，並知道如何回滾。

動手試試

拿一個你已有的小專案（哪怕只是一個 HTML 頁面）部署到靜態託管上。然後讓 AI 生成一個每次 push 都重新部署的 GitHub Actions 工作流，逐行讀懂生成的 YAML，並確認你能在託管商的控制台裡找到“回滾到上一次部署”的按鈕。

```
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.
```

資料庫基礎設施



每一個真實的應用最終都需要記住一些東西——使用者、訂單、訊息、設定。那份記憶存在於資料庫裡。對初學者來說，“資料庫”可能感覺像一個由專家把守的黑盒子。它並不是。本章揭開資料庫的神秘面紗，讓你能做出好的選擇，並指揮 AI 安全地完成繁重的工作。

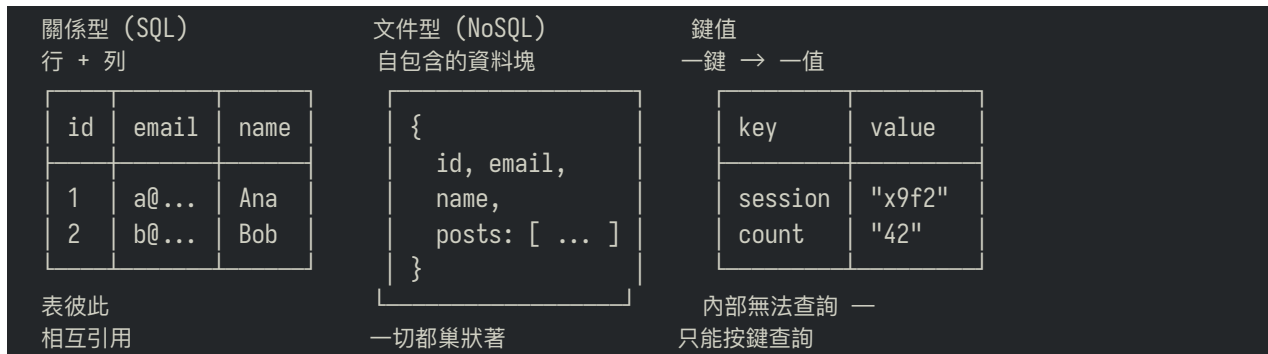
要早早內化的一點：程式碼是可替換的，資料通常不是。一個有 bug 的函式你可以花一個下午重寫，但你上週二弄壞的某一列——或者你從未做過的備份——可能就永遠沒了。正是這種不對稱，讓資料庫比你技術棧中幾乎任何其他部分都更值得一點點提前的理解。

資料儲存的三種形態

大多數資料庫分為三個家族。你不需要去背產品——你需要認出哪種形態適合你的問題。

型別	例子	最適合	取捨
關係型 (SQL)	PostgreSQL, MySQL, SQLite	帶關係的結構化資料；任何與金錢或準確性相關的關鍵場景	你必須預先設計 schema；把寫入擴充套件到非常大的規模更難
文件型 / NoSQL	MongoDB, Firestore, DynamoDB	靈活或巢狀的資料、快速迭代、形態各異的記錄	容易產生不一致的資料；關係和報表處理起來很彆扭
鍵值型	Redis, Memcached, Cloudflare KV	快取、會話、計數器、按已知鍵的快速查詢	無法按內容查詢；通常不是你的事實來源

這三種形態以確實不同的方式儲存資料。把它們並排來看：



把關係型資料庫想像成一組彼此知道如何相互引用的電子表格。你用來跟它對話的語言就是 SQL (Structured Query Language, 結構化查詢語言)。對 90% 的專案來說一個好的預設選擇：從關係型資料庫 (**Postgres**) 開始。它成熟、可預測、強制結構化，並能處理真實應用不可避免會長出來的關係。當你的資料確實沒有固定形態時，再去考慮文件儲存；而鍵值儲存則作為一個輔助層 (快取)，而不是你的主儲存。

要避免的陷阱是憑感覺選。“NoSQL (不使用 SQL 表的資料庫——它儲存靈活、自由形態的記錄) 擴充套件性更好”只對一小撮大多數應用永遠碰不到的問題成立，而即便如此，代價也是放棄那些讓資料可信賴的一致性保證。挑選與你的資料相匹配的形態，而不是在系統設計影片裡聽起來最唬人的那個。拿不準時，關係型是穩妥的選擇——快取或文件儲存你以後隨時可以加上，但解開一團亂麻的 NoSQL 資料模型卻很痛苦。

託管 vs 自託管

你可以自己在一臺伺服器上執行資料庫，也可以租用一個託管的 (Supabase、Neon、PlanetScale、RDS、Cloudflare D1)，由提供商來處理備份、打補丁和正常執行時間。

對於 *vibe coding*，始終從託管開始。自託管一個數據庫意味著你現在要為備份、安全更新、磁碟空間，以及凌晨 3 點的恢復負責。那是真正的運維工作，會從構建中分散你的精力。託管服務花一點錢，卻能節省巨大的時間和風險。只有當你有了具體的理由以及足以支撐的技能時，才晉級到自託管。

挑選託管提供商時，值得知道你實際在為什麼付費，因為各家的營銷頁面看起來都差不多。有幾點確實不同：

- **連線模型**。無伺服器平臺 (Cloudflare Workers、Vercel functions) 會不停地開啟和關閉資料庫連線，這可能耗盡經典的 Postgres 連線池。Neon 和 Supabase 這樣的提供商正是為此提供了連線池器 (PgBouncer 或同類)——在無伺服器程式碼裡使用池化的連線字串。
- **分支與備份**。Neon 和 PlanetScale 讓你能像 Git 一樣給資料庫“開分支”，這樣你可以在副本上測試遷移。確認時間點恢復 (point-in-time recovery) 已開啟，並搞清楚它能回溯多久。
- **鎖定 (lock-in)**。D1 和 DynamoDB 很出色，但會把你綁在一個雲上。純 Postgres (Supabase、Neon、RDS) 是可移植的——你的 schema 和查詢能搬到任何 Postgres 主機上。對大多數初學者來說，可移植是更安全的長期選擇。

Schema 設計基礎

schema 是你資料的藍圖：存在哪些表、它們有哪些列，以及它們如何關聯。好的 schema 設計大多是常識加上幾個習慣。

- 每一種“東西”對應一張表。使用者、帖子、訂單——各自擁有自己的表。
- 每一行都需要一個唯一的 **ID** (一個主鍵)。
- 用外部索引鍵連結表，而不是到處複製資料。
- 選擇誠實的型別。金錢是 decimal，不是 float。日期是時間戳，不是文本。
- 不要重複。如果一個使用者的郵箱存在於五張表裡，你最終會有五個不同的郵箱。

下面是一個部落格的簡單 schema，正是 AI 會為你生成的那種：

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



```

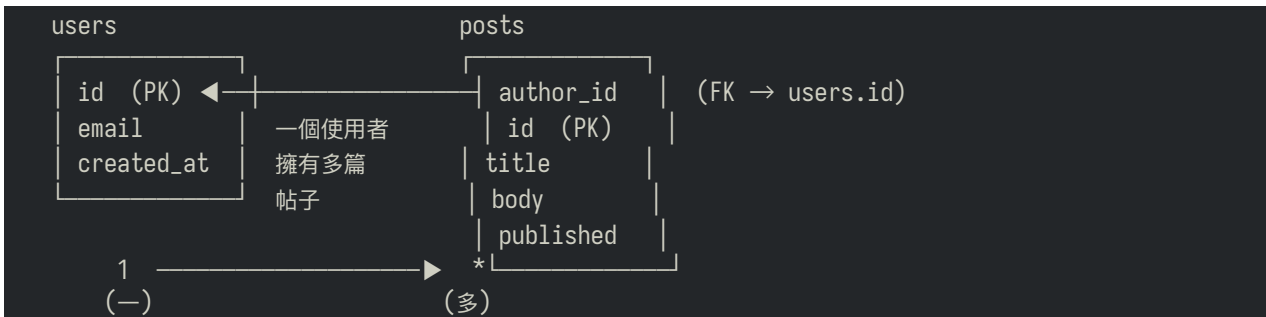
    created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

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)` 這一行就是外部索引鍵。它把每篇帖子接回到恰好一個真實使用者，於是資料庫自身就會拒絕儲存一篇沒有有效作者的帖子：



注意 `REFERENCES`（一個把帖子繫結到真實使用者的外部索引鍵）以及 `NOT NULL` 和 `UNIQUE` 約束。這些是資料庫為你強制執行的護欄，所以即使你的程式碼有 `bug`，壞資料也無法溜進來。

一個值得有意新增的約束，是當父記錄消失時會發生什麼。預設情況下，刪除一個仍被帖子引用的使用者會失敗——而這往往正是你想要的。但你可以顯式指定：`REFERENCES users(id) ON DELETE CASCADE` 會把帖子也一併刪除，而 `ON DELETE RESTRICT` 會阻止該刪除。有意去選，勝過在生產環境裡以慘痛方式發現預設行為。同樣的“讓資料庫保護你”的直覺也適用於 `CHECK` 約束（`CHECK (price >= 0)`）——它在源頭拒絕荒謬的資料，而不是去信任每一條程式碼路徑都會做校驗。

遷移：安全地更改 *Schema*

你的 `schema` 會變化——你會新增一列、重新命名一張表、加一個索引。把它想像成對一份共享文件的修訂追蹤：每一次更改都按順序記錄下來，於是每個人的副本都能用同樣的方式更新到最新。一次**遷移** (*migration*) 是一個版本化的、可重複的指令碼，描述這樣一次更改。遷移之所以重要，是因為你筆記本上的 `schema`、你隊友筆記本上的 `schema`，以及生產環境裡的 `schema` 必須保持完全一致。

讓遷移保持安全的規則：

- 絕不要在生產環境裡手動改資料庫。寫一次遷移。
- 遷移只往前走。每一次都是一個新檔案；你不要去重寫那些已經執行過的舊檔案。
- 謹慎進行破壞性更改。刪除一列或一張表會永久刪除資料。先有一份備份。
- 追加式更改最安全。新增一個可空的列很少會破壞什麼；重新命名或移除一個列則可能破壞正在執行的程式碼。

遷移構成一條有序的鏈：每一次都是一個編號步驟，把資料庫的每一份副本從一個已知狀態推進到下一個，於是你的筆記本、隊友的筆記本和生產環境最終都完全一致：





像 Prisma、Drizzle，或 Rails 遷移這樣的工具會為你生成並跟蹤這些。讓它們去做。

一次遷移由兩半組成：要應用的更改（up）和如何撤銷它（down）。一個值得信賴的工具會把兩者都生成出來，這樣一次糟糕的部署就能幹淨地回滾：

```
-- up: 新增一個可選的 bio，可在線上安全地應用
ALTER TABLE users ADD COLUMN bio TEXT;

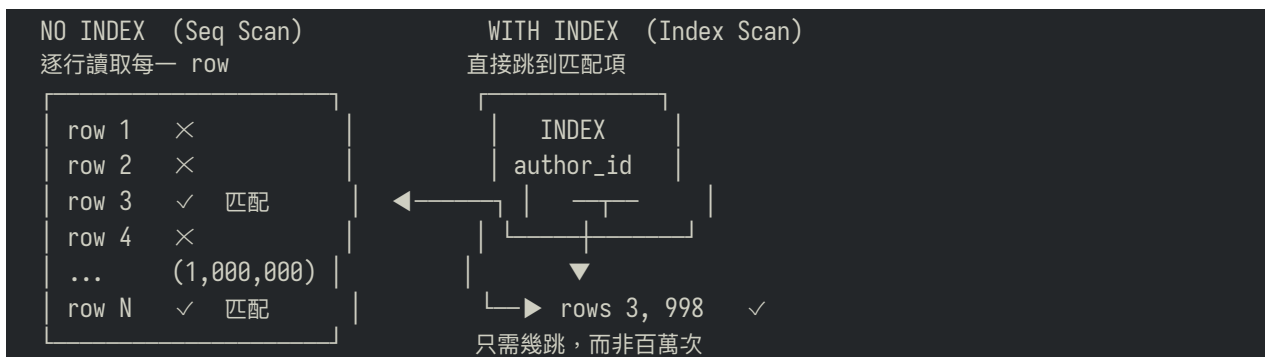
-- down: 把它反轉回去
ALTER TABLE users DROP COLUMN bio;
```

真正危險的遷移是重新命名和型別更改，因為舊程式碼和新 schema 會短暫地相互矛盾。專業的做法是把它們拆成追加式的步驟：新增新列、回填它、把程式碼切換到讀取它，只有在之後的一次部署裡、當沒有任何東西再引用舊列時，才把舊列刪掉。它感覺很慢，但這正是一次平滑釋出與部署途中每個使用者都看到 500 錯誤激增之間的區別。

索引

一個索引（**index**）就像一本書後面的索引：它讓資料庫直接跳到匹配的行，而不必掃描每一行。沒有索引，查詢會隨著資料增長而變慢——在 100 行時還好，在 1,000,000 行時就痛苦了。

沒有索引，資料庫會逐行依次檢查；有了索引，它就直接跳到匹配項——就像書的索引把你送到正確的頁碼，而不必從頭讀到尾：



實用的指導：

- 為你頻繁篩選或連線的列建立索引（例如 `author_id`、`email`）。
- 主鍵會自動建立索引。
- 不要給所有東西都建索引——每個索引都會加速讀取，但會拖慢寫入並佔用空間。
- 當你看到一個慢查詢時再新增索引，而不是預先給每一列都加。

弄清楚某個索引到底有沒有用的誠實辦法，是去問資料庫，而不是去猜。每個 SQL 引擎都有 EXPLAIN（在 Postgres 裡是 EXPLAIN ANALYZE），它會展示它將採用的計劃：

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

如果在一大張表上輸出說 Seq Scan，那就是資料庫在讀取每一行——一個索引會有幫助的訊號。加上一個之後，它應該切換為 Index Scan。這是一個很適合交給 AI 的任務：把慢查詢和 EXPLAIN 的輸出貼上去，問該加哪個索引以及為什麼。你通過理解答案來保持掌控，而不是靠背索引理論。

快取

一個快取 (**cache**) 儲存一個昂貴問題的答案，這樣你就不必再問一遍。如果你的主頁為每一個訪客都運行同一個繁重的查詢，那就把結果在 Redis 裡快取 60 秒，瞬間提供給使用者。

快取是一種效能最佳化，不是事實來源。經典的初學者陷阱是忘記快取失效 (**cache invalidation**) ——快取的副本變陳舊了，使用者看到的是舊資料。經驗法則：只有當你測量出一個真實的瓶頸時才快取，並且始終對快取值何時過期或重新整理有一條清晰的規則。

有兩種乾淨的策略，把它們搞混會引發大多數快取 bug。基於時間的過期 (**TTL**——**Time To Live**，就像蓋在快取值上的保質期) 說“這個值在 60 秒內有效，之後就扔掉”——簡單，而且在略微陳舊的資料無害時（一個瀏覽計數、一個主頁資訊流）很合適。基於事件的失效說“在底層資料改變的那一刻就刪除快取值”——正確，但你必須在每一條寫入路徑上都記得這麼做，而那恰恰是人們出岔子的地方。拿不準時，優先用短 TTL：一個最多陳舊一分鐘的值，比一個因為你忘了一次失效而永遠陳舊的值，是更小的 bug。

佇列

一個佇列 (**queue**) 讓你說“稍後再做這件事”，而不是“現在就做這件事”。當一個使用者註冊時，你不想讓他們一邊等著你傳送歡迎郵件、調整他們的頭像尺寸。把這些任務推到一個佇列裡 (RabbitMQ、BullMQ、Cloudflare Queues、SQS)；一個後臺 worker 會把它們取走。使用者得到一個即時的響應。

對任何緩慢的、可能失敗並重試的，或者不需要在你響應使用者之前完成的任務，都使用佇列。

大多數初學者會撞上的坑是，一個入隊的任務可能執行不止一次——一次網路抖動讓 worker 重試，於是歡迎郵件就發了兩遍。修復辦法是讓任務冪等 (**idempotent**)：把它們設計成運行同一個任務兩次與執行一次效果相同（傳送前先檢查“我不是已經發過了？”）。也要預先決定，一個反覆失敗的任務該怎麼辦；成熟的佇列會把它路由到一個死信佇列，這樣一條毒訊息就不會把排在它後面的一切都堵死。

常見的初學者錯誤

- 因為 **NoSQL** 聽起來很現代就去用它。大多數應用都有關係；SQL 處理它們更好。
- 把金錢存為浮點數。眾所周知 $0.1 + 0.2$ 並不等於 0.3 。使用 **decimal** 型別。
- 沒有備份。託管服務通常會自動備份——去確認它，別去假設它。
- 把金鑰或巨大的檔案放進資料庫。檔案屬於物件儲存 (S3、R2)；資料庫儲存那個連結。
- 讓 **AI** 寫把使用者輸入直接拼進去的原始查詢。這會導致 SQL 注入。使用引數化查詢。
- **N+1** 查詢。一個對每個項都跑一次查詢的迴圈（取 100 篇帖子，然後再分別做 100 次作者查詢）會猛錘資料庫。用一次帶連線 (join) 的查詢取回相關資料，或者把查詢批次化。
- 從不做負載測試。一個在你筆記本的 50 行上瞬時的查詢，在生產的 500 萬行上可能會爬。在你信任效能之前，先灌入一份現實數量的假資料。

指揮 **AI** 構建你的資料層

AI 在 schema 設計和查詢編寫方面非常出色——前提是你給它約束去據此設計。不要說“做一個數據庫”。描述領域、關係，以及你不可妥協的底線。

一個強力提示看起來像這樣：

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.

然後用本章的清單來**審查輸出**：誠實的型別、外部索引鍵和約束齊備、合理的索引、沒有重複的資料。當你請求查詢時，明確要求**引數化查詢**，這樣使用者輸入就絕不會被拼接進 SQL。而對於任何刪除或重新命名某物的遷移，讓 AI 把它點出來，這樣你可以先備份。

資料庫是你應用中最難撤銷錯誤的部分——資料一旦損壞或丟失，就不總能回來。這正是為什麼這些基本功值得了解：不是為了讓你自己寫出每一行，而是為了讓你能自信地指揮 AI，並在那些危險的操作上線之前抓住它們。

回顧與練習

關鍵要點

- 大多數應用只需要三種儲存形態——關係型、文件型、鍵值型——中的一種，而關係型是安全的預設選擇。
- 從託管資料庫開始：資料和模式決策由你掌握，伺服器的執行由平臺負責。
- 好的模式設計意味著誠實的型別、外部索引鍵與約束、合理的索引，以及沒有重複的資料。
- 把每一次遷移都當作一道單向門：在 **drop** 或重新命名任何東西之前先備份，並仔細審查 AI 寫的遷移。
- 始終要求引數化查詢，使使用者輸入永遠不會被拼接進 SQL。

動手試試

挑一個小小的應用點子，先用平實的中文把它的實體和關係寫出來。然後把這段描述交給 AI，請它給出一份模式加一個前向遷移——在執行任何東西之前，用本章的清單審查結果。

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

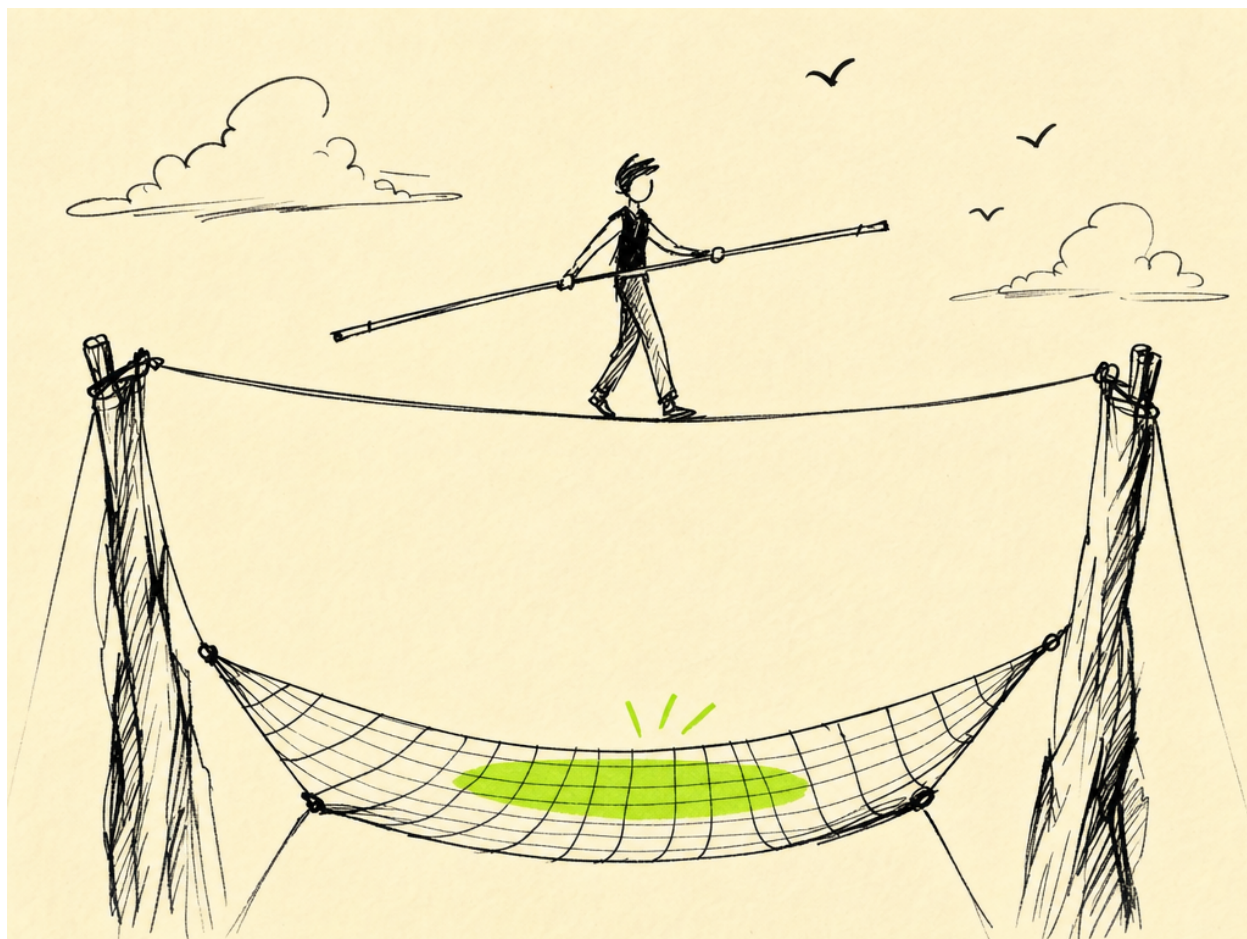
```
Entities and relationships:
```

- 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.

防護欄、測試與審查



Vibe coding 讓你跑得飛快。但危險在於，你也可能朝著錯誤的方向飛快前進——而且因為 AI 聽起來信心十足，你可能直到某個東西在真實使用者面前崩潰時才會察覺。防護欄就是讓你在保持速度的同時不至於翻車的方法。

本章並不是要把你的專案變成一套官僚體系。它講的是一小撮廉價、自動的檢查，能在 AI（以及你自己）的錯誤抵達生產環境之前就把它們抓出來。設定一次，它們就會永遠在後臺為你工作。本章的全部哲學就是一句話：永遠不要釋出你不理解的東西。防護欄的作用，就是把“理解它”變得足夠廉價，讓你每一次都真的會去做。

為什麼指揮 AI 時你需要防護欄

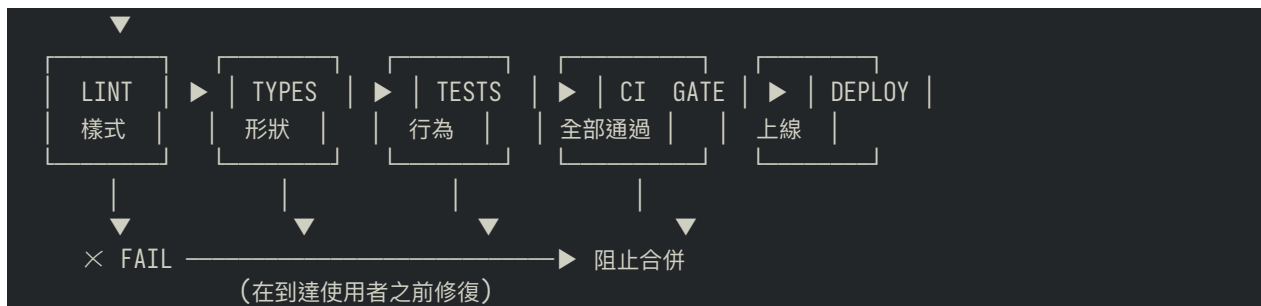
AI 並不知道你的應用本該做什麼。它只是在向看起來合理的模式靠攏。這意味著它會樂此不疲地：

- 為了“修復報錯”而刪掉一個它不理解的檢查
- 在你只要求它改一個檔案時，卻重構了三個檔案
- 引入一個 happy path 不會暴露出來的隱蔽 bug
- 在它根本沒執行過程式碼的情況下，自信地告訴你已經做完了

防護欄把“相信我”變成“拿出證據”。目標是構建一個讓東西很難被意外搞壞的系統，這樣你就能一直說“先試試看”，而不必屏著呼吸。

把這些防護欄想像成一條流水線，每一處改動在抵達真實使用者之前都必須穿過它。每道關卡都廉價而自動，其中任何一道都能把一個壞改動當場攔下：

程式碼變更

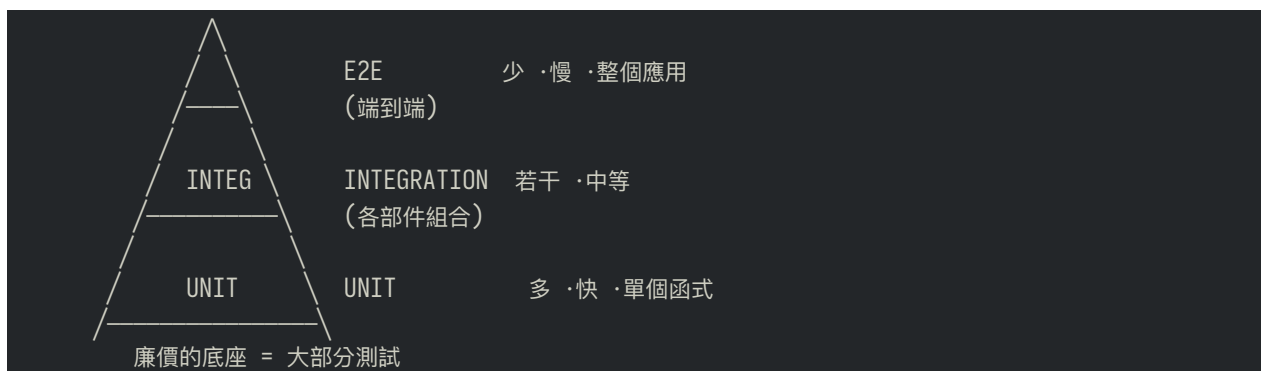


這件事在 AI 身上比在人類隊友身上更重要，有一個更深層的原因。一個刪掉校驗檢查的人類，通常知道自己在冒險，並且能感到那份分量。AI 什麼也感覺不到。它會移除一個守衛、重寫一個檔案、然後報告成功——無論這個改動是絕妙還是災難性的，都用同樣歡快的語氣。你讀不出它的肢體語言，因為它根本沒有肢體語言。防護欄就是它的肢體語言——它是關於這份工作是否真正穩妥，你能得到的唯一誠實訊號。

測試：你的安全網

測試就是用來檢查其它程式碼是否如你預期那樣執行的程式碼。一旦寫好，它會在幾秒內執行完，並立即告訴你是否東西回退 (regression) 了。

一套健康的測試套件長得像一座金字塔：底座是大量又小又快的單元測試，中間是較少的整合測試，頂端只有一小撮緩慢的端到端測試。你的大部分安全感，都廉價地來自底座：



你不需要 100% 的覆蓋率。先從那些一旦壞掉就最傷的東西開始測：登入、支付、無法重新生成的資料、核心業務邏輯。AI 非常擅長寫測試——往往比寫功能還要強，因為預期行為是具體明確的。

下面這個 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.
```

最後那一行很關鍵。一定要讓 AI 執行測試，而不只是寫出來。一個從未被執行過的測試只是一個猜測。具體說明要覆蓋哪些情況也很有價值，因為放任不管的話，AI 傾向於寫出三種 happy path 的變體，然後就聲稱很周全了。真正能抓出 bug 的，是那些無聊的情況：空輸入、邊界值、本應丟擲異常的東西。

它應當產出的東西的一個簡單示例：

```
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();
  });
});

```

當你之後改動某個功能時，先執行測試。如果它們變紅了，你就在使用者之前發現了破綻。當 AI 修復一個 bug 時，讓它補一個針對舊行為會失敗的測試——這樣這個 bug 就不會悄無聲息地捲土重來。這個習慣有個名字：迴歸測試 (regression test)。你遇到的每一個 bug，都是程式碼庫付出代價換來的教訓；迴歸測試就是你保住這個教訓的方式。

一個提醒：測試本身也可能是錯的。如果 AI 寫了一個斷言有 *bug* 的行為是正確的測試，它會永遠通過，而你的應用會一直壞著。所以要讀斷言，而不只是看綠色對勾。一個通過的測試只意味著“程式碼做了測試所說的事”——你仍然得確認這個測試說的是對的事。

型別檢查與程式碼檢查：免費抓出的錯誤

有兩種工具幾乎不需要你費什麼力氣，就能抓出一整類錯誤：

- **型別檢查** (TypeScript、mypy 等) 能抓出“你把一個 string 傳給了期望 number 的地方”以及“這個值可能是 undefined”。AI 特別喜歡忽略那些被型別設計成不可能出現的邊界情況。
- **程式碼檢查 (Linting)** (ESLint、Ruff 等) 能抓出未使用的變數、不可達程式碼以及風格漂移。它能防止一個被 AI 反覆改動的程式碼庫慢慢糊成一團。

儘早把這些設定好，並告訴 AI 遵守它們：“執行型別檢查器和 *linter*，並在你認為做完之前修復它們報出的所有問題。”把一次乾淨的型別檢查當作“完成”定義的一部分。

不過，要警惕那些偷懶的逃生口。當型別檢查器抱怨時，真正正確的修復是去處理它發現的那個情況——那個 undefined 值、那個錯誤的型別。而快速的修復，是用 `// @ts-ignore`、`any` 或者 `# type: ignore` 註釋把抱怨壓下去，AI 會不停地伸手去抓這些，因為它們能讓紅字消失。它們也恰恰在檢查本想幫忙的那個位置，把檢查變得毫無價值。如果你在 diff 裡看到一條抑制 (suppression) 註釋出現，把它當成一個問題，而不是一個解決方案：“這為什麼是必要的？去修底層的型別，而不是忽略它。”

限定 AI 改動的範圍

AI 造成損害的最大單一原因，就是放任它一次改太多東西。一個跨越九個檔案、400 行的 diff 既無法審查，又很容易藏住 bug。

讓改動保持小而集中：

- 一次只要求一個邏輯改動，而不是“既然你都改到這兒了，順便也……”
- 明確告訴它：“只改 `src/auth.js`。別重構其它任何東西。”
- 經常提交可執行的狀態，這樣你總有一個乾淨的回滾點。
- 當它提議一個大重寫時，讓它把工作拆成你能一個一個審查的步驟。

範圍小，才能讓下一步——讀 diff——真正變得可行。這裡有一個複利效應：小的提交，其爆炸半徑 (blast radius) 也小。如果一個範圍緊湊的改動結果是錯的，你回退一個提交，損失十分鐘。如果一個鋪得很開的改動是錯的，你要麼花一個小時去理清到底是哪部分搞壞了，要麼扔掉一下午的工作。範圍紀律是一份廉價的保險，在你知道自己會需要它之前就先買好。

帶著批判地閱讀 *diff*

diff 就是一份關於究竟改了什麼的清單：紅色行被刪除，綠色行被新增。讀 *diff* 是 *vibe coding* 中槓桿率最高的習慣，因為正是在這裡你能抓出 AI 在做你沒要求它做的事。

你不必理解每一行。掃一掃這些危險訊號：

- 你沒料到會變的檔案。一個配置檔案為什麼被改動了？
- 被刪掉的程式碼。它是不是為了“讓測試通過”而刪掉了一個校驗檢查、一個錯誤處理器，或一個測試？
- 金鑰或硬編碼的值。不該出現在程式碼裡的 API (Application Programming Interface) key、密碼或 URL (Uniform Resource Locator——網址)。
- 被停用的檢查。一個被跳過的測試、一段被註釋掉的守衛、一個 `// @ts-ignore`，或者一個被放鬆的許可權。
- 範圍蔓延 (*Scope creep*)。離你要求的內容很遠的改動。

如果有什麼看起來不對勁，就問：“解釋一下你為什麼改了 X——我沒要求改那個。”尤其要讓 AI 為刪除行為辯護。“它在導致報錯”是一個去理解那個報錯的理由，而不是刪掉那個檢查的理由。

一個養成習慣的實用做法：在你接受改動之前讀 *diff*，絕不在之後。一旦程式碼被合併、應用看起來能跑，你就再也不會回去讀它了——沒有任何摩擦逼著你去讀。審查的那一刻，必須就是決定的那一刻。工具在這裡幫得上忙：`git diff --stat` 給你每個檔案一行的摘要，讓意料之外的檔案立刻跳出來；在 *pull request* 頁面上審查，會把整個改動呈現在一個可滾動的視圖裡，而不是散落在你編輯器各處。

審查關卡：合併前的一份清單

審查關卡 (review gate) 就是一份在改動上線前必須為真的簡短清單。自己過一遍，或者讓 AI 逐項確認：

- ☐ 我讀了 *diff*，並理解改了什麼、為什麼改
- ☐ 改動被限定在我實際要求的範圍內
- ☐ 測試通過（且新行為有對應的測試）
- ☐ 型別檢查和 linter 都是乾淨的
- ☐ 沒有遺留金鑰、除錯日誌或被註釋掉的程式碼
- ☐ 沒有安全檢查被悄悄移除
- ☐ 我自己試過這個改動，至少跑過 happy path

如果有哪一項你勾不上，那你就還沒做完。

這份清單刻意保持簡短，因為一個你會跳過的關卡，根本就不算關卡。尤其是當你跑得很快、改動“看起來沒問題”時，會有一種直接放行的誘惑。對任何要面向使用者的東西，請抵抗它。一個有用的小技巧，是讓 AI 替你來填——把清單貼過去，然後說：“針對你剛做的改動，把這份審查關卡逐條走一遍。每一項，給出 *PASS* 或 *FAIL*，並附上一行證據。”AI 沒執行過測試就沒法誠實地寫下“測試通過”，而強迫它逐行給出證據，會把一句含糊的“看著不錯”變成一個你可以審計的東西。

CI 基礎：讓檢查自動化

CI (持續整合, Continuous Integration) 是一個機器人，每次你推送程式碼時它都會執行你的那些檢查，這樣你就不會忘記去執行它們。大多數平臺只需一個檔案就能搞定。下面是一個最小化的 GitHub Actions 示例：

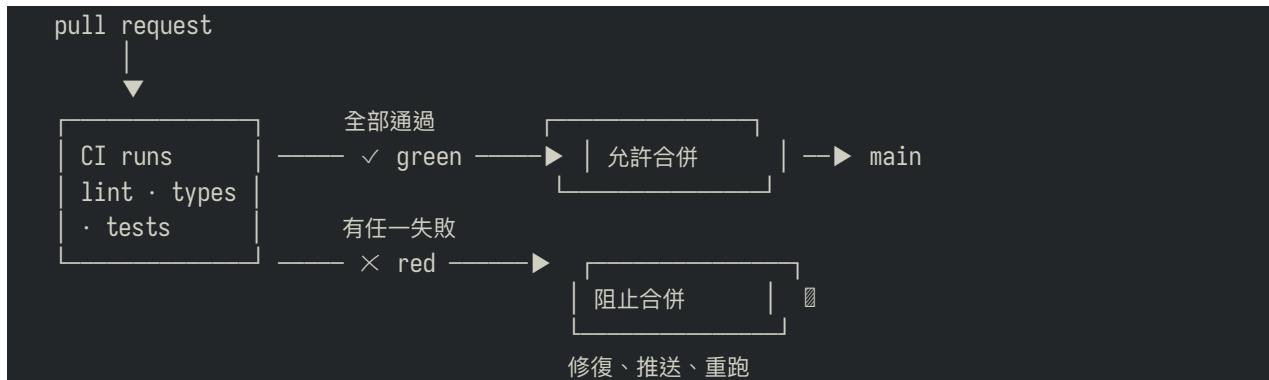
```
# .github/workflows/ci.yml
name: CI
on: [push, pull_request]

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
```

現在每一次推送都會自動經過 lint、型別檢查和測試。如果有任何一項失敗，你會在程式碼能被合併之前看到一個紅色標記。你甚至可以讓 AI 為你的技術棧寫好這個檔案——“加一個 *GitHub Actions workflow*，在每個 *pull request* 上執行我們的 *lint*、型別檢查和測試。”

一旦開啟分支保護，紅色的檢查就不只是警告你——它會在構建重新變綠之前，從物理上鎖住合併按鈕：



CI 的重點並不是自動化本身，而是它把檢查從你的記憶裡挪到了流水線裡——在那裡，再疲憊的夜晚也跳不過去。等基本流程跑出綠燈之後，有兩個值得做一次的升級。第一，把這些檢查設為主分支上的必需狀態（GitHub 上的“分支保護規則”），這樣一個紅色的構建會真正卡住合併按鈕，而不只是顯示一個你可以無視的警告。第二，當 CI 失敗時，把失敗日誌直接貼給 AI：“這次執行的 *CI* 失敗了，這是輸出——診斷並修復它。”日誌是一份精確的、機器生成的 bug 報告，而這恰恰是 AI 最擅長據以行動的那種輸入。

預設安全的習慣

把這些匯聚成一種反射本能，讓你能毫無畏懼地快速前進：

- 在分支上工作，而不是直接在生產環境上。分支是免費的；壞掉的生產環境可不是。
- 頻繁提交小而可執行的狀態。頻繁的提交讓回滾變得無足輕重。
- 永遠讓 *AI* 執行它寫出來的東西——測試、應用、型別檢查器。
- 給 *AI* 留一個終端，給你自己留一個，這樣你能獨立驗證。
- 絕不讓一個你沒讀過的改動上線。速度來自良好的防護欄，而不是來自跳過審查。

防護欄不會拖慢你——它們正是讓你能心裡不打鼓地說出“好，釋出吧”的東西。設定一次，AI 就可以全力衝刺，而安全網會在它摔倒時接住它。

當防護欄尖叫時：先分診，再修

防護欄只有在你好好回應時才有用，而失敗模式是可預料的：構建變紅了，你有點慌，然後把報錯貼給 AI，附上“修一下”。有時這管用。但更多時候，它會造出一個修復——讓紅字消失，卻沒有處理真正出錯的地方：AI 把測試削弱了、把型別放寬了，或者把那個失敗的呼叫包進一個會吞掉問題的 *try/catch* 裡。

在你放任它去“修”任何東西之前，花十秒做個分診。防護欄的失敗其實只有三類：

- 一個你剛剛引入的**真 bug**。檢查正在盡職。去修程式碼，而不是修檢查。
- 一個現在已經過時的測試。你是故意改了行為，舊的斷言不再匹配了。把測試更新為斷言新的正確行為——是經過深思的，而不是條件反射式的。
- 一個不穩定或本身就錯的檢查。真的很罕見。在這裡要懷疑你自己；這正是人們用來傾倒真 bug 的那個桶，因為它是唯一一個能讓你矇混過關、繼續往前走的桶。

那條能讓 AI 保持誠實的指令，是逼它在改任何東西之前先做診斷：

"never discounts below zero" 這個測試失敗了。

首先，告訴我它為什麼失敗——程式碼現在做了什麼，對比測試期望什麼。然後告訴我哪個才是對的：程式碼，還是測試。在我確認之前，什麼都別改。

那一次停頓，就是”一個保護你的防護欄”和”一個被你悄悄訓練 AI 去攻克的防護欄”之間的區別。一個紅色的檢查是資訊；你能做的最糟的事，就是在沒讀懂它想告訴你什麼之前，就花代價把它弄綠。

回顧與練習

關鍵要點

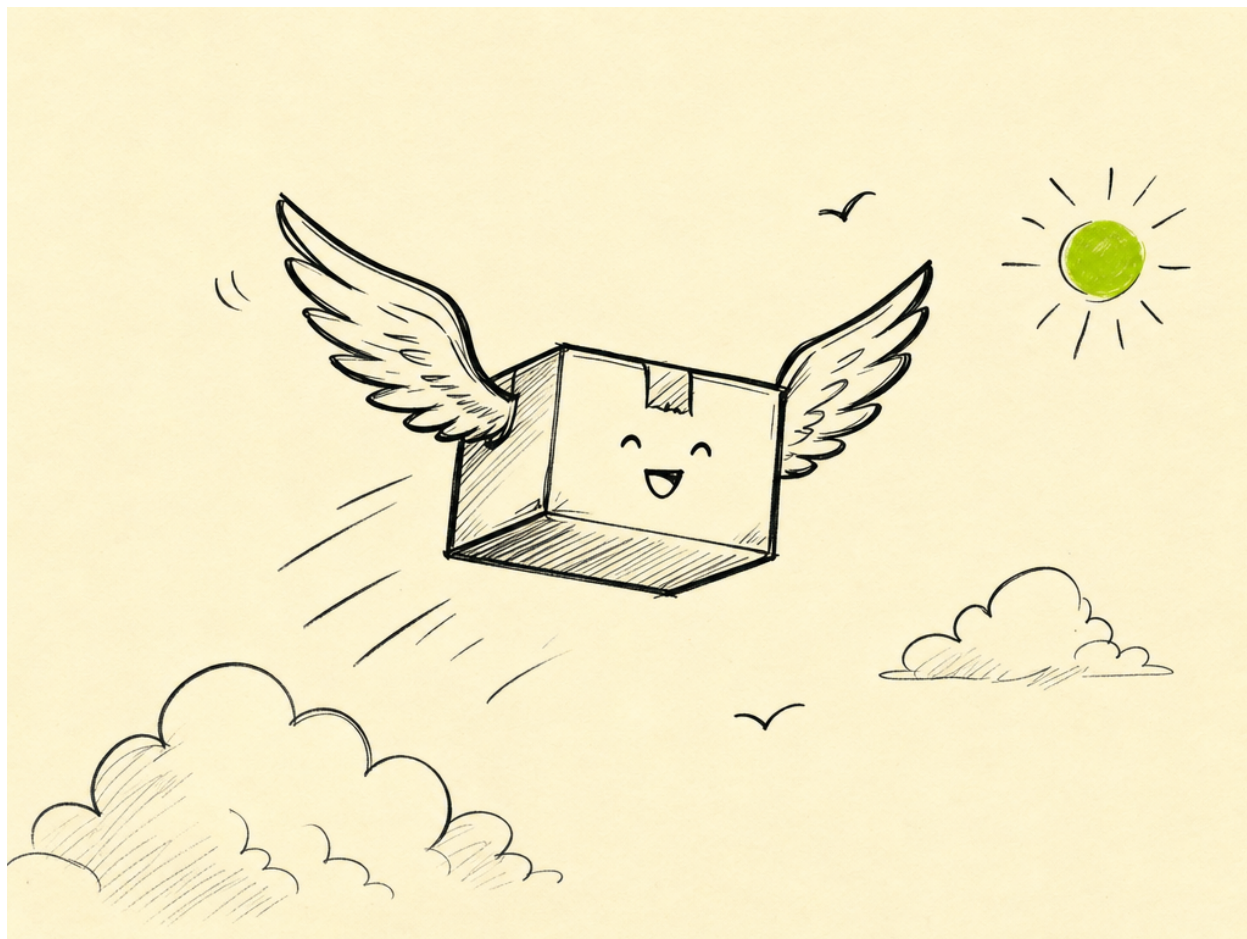
- 護欄讓 AI 得以全速衝刺，同時讓安全網接住跌落——它們讓你更快，而非更慢。
- 測試、型別檢查和 lint 會免費幫你抓住錯誤；把它們接入 CI，讓檢查自動執行。
- 把 AI 的改動範圍限定得很窄，合併之前批判性地讀每一處 diff。
- 紅色的檢查是資訊——在讓 AI “修復”它之前先分類（真正的 bug / 過時的測試 / 不穩定的檢查）。
- 絕不上線你沒讀過的改動。

動手試試

在你的某個專案裡，設定一個你還沒有的自動檢查——一個測試、一個 linter，或一個型別檢查步驟——並把它加進 CI 工作流，讓它在每次 push 時執行。然後故意弄壞一個小東西，確認在你修復之前檢查會變紅。

```
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.
```


部署上線到生產環境



你一直在搭建。應用在你的機器上能跑，演示讓一位朋友驚歎，想把它推向世界的衝動是真實的。這正是許多獨立開發者卡住的時刻——不是因為程式碼壞了，而是因為“上線”感覺像是一門自帶黑魔法的、完全不同的學問：DNS (Domain Name System，域名系統，網際網路的地址簿，把像 `yourdomain.com` 這樣的名字翻譯成伺服器所在的那個號碼)、伺服器、金鑰、支付 `webhook`。好消息是：這恰恰是 AI 最擅長的那種機械、有完善文件的工作。你的任務是指揮。它們的任務是敲命令。

但“指揮”這個詞是承重的。這整本書的主題就是理解你所釋出的東西。你不需要把那些標誌引數背下來，但你確實需要知道每一步是為了什麼——`webhook` 在保護你免於什麼，為什麼金鑰絕不能進倉庫，消費上限在給什麼套籠子。當你理解了這件事的形狀，你就能在 AI 那個自信滿滿的答案悄悄出錯時察覺到，也能在第一次部署炸掉時把它救回來。在上線時被燙到的開發者，幾乎總是那些貼上了自己讀不懂的命令的人。

本章會走完最後這一英里的全程，從一個能在本地執行的應用，到一個真實的、能讓人們向你付款的 URL (Uniform Resource Locator，統一資源定位符，人們在瀏覽器裡鍵入的那個網址)。

選一個託管平臺並部署

對大多數 `vibe coding` 出來的應用，你並不需要一臺需要你時刻照看的伺服器。現代平臺拿到你的程式碼，幾分鐘內就能給你一個上線的 URL。常見的選擇：

- 靜態站點和前端：Cloudflare Pages、Vercel、Netlify、GitHub Pages。
- 全棧應用和 **API** (**Application Programming Interface**，應用程式程式設計介面，一個程式用來與另一個程式對話的門戶)：Cloudflare Workers、Vercel、Render、Fly.io、Railway。

- **更重的後端**：一個容器平臺，或一臺小型 VPS (Virtual Private Server，虛擬專用伺服器，相當於你在資料中心裡租下的、一臺永遠開機的計算機的一塊專屬切片)。

正確的做法是告訴 AI 你搭了什麼，讓它來推薦。試試：“這是一個帶 *Postgres* 資料庫的 *Next.js* 應用。我想要一個最便宜、可靠、免費額度慷慨的託管平臺。給我部署步驟和一個配置檔案。”對那些真正約束選擇的因素要說具體：你的框架、有沒有資料庫、是否需要後臺任務或檔案上傳，以及你的預算。一個靜態的營銷頁面，和一個帶佇列的全棧應用，住的地方截然不同。

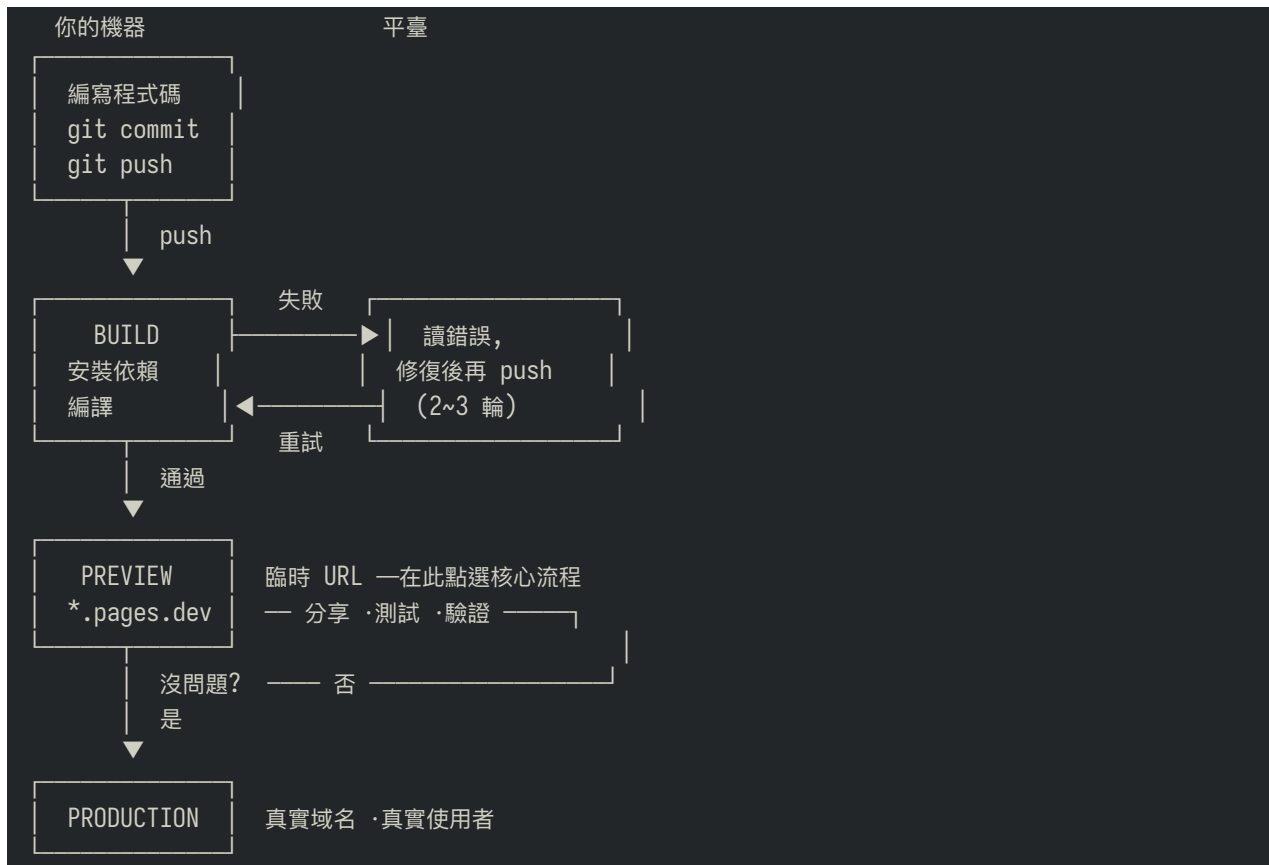
一旦選定了，部署通常就是幾條命令：

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

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

讓 AI 生成平臺配置 (wrangler.toml、vercel.json 等)，並解釋第一次部署丟擲的任何報錯。第一次部署幾乎總會丟擲點什麼——一個缺失的構建命令、一個錯誤的輸出目錄、一個 Node 版本不匹配、一個構建需要卻找不到的環境變數 (environment variable，從程式碼之外交給你應用的一項帶名字的設定，就像放在門墊下、應用知道去哪裡找的一把鑰匙)。這些都不意味著你幹了蠢事；它們意味著平臺還需要你再告訴它一件事。把報錯原樣粘回去，讓它給出修復方案，再問問為什麼會發生，好讓下一次少點意外。兩三個回合，你就上線在一個臨時 URL 上了。在慶祝之前，開啟那個 URL，把核心流程點一遍——“部署成功了”和“應用能用”不是同一個論斷。

每一次部署都走同一條流水線。預覽那一步就是你的安全網：每一個變更都擁有自己專屬的 URL，在它觸達真實使用者之前供你點選檢視。



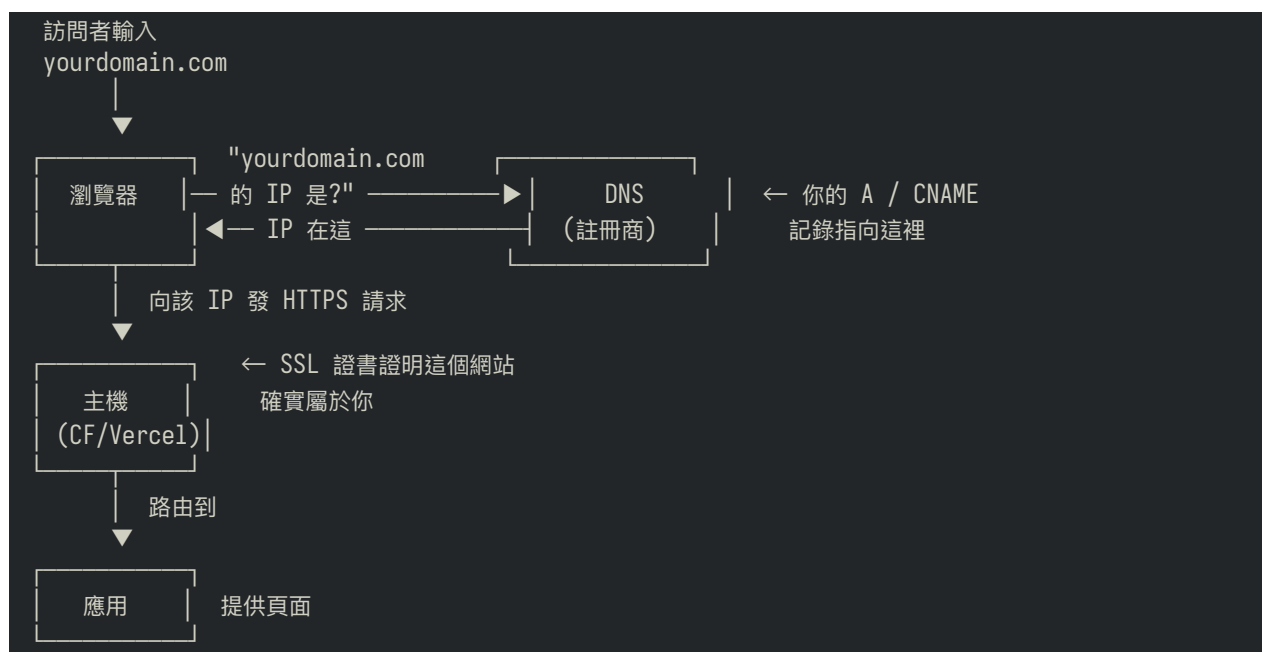
自定義域名與 DNS

一個 `*.workers.dev` 或 `*.vercel.app` 的 URL 拿來測試沒問題，但你會想要屬於自己的名字。流程如下：

1. 買下域名（Namecheap、Cloudflare Registrar、Porkbun——挑一個）。
2. 把它的 DNS 指向你的託管平臺。這通常意味著新增一條 CNAME 或 A 記錄，或者徹底更換域名的 nameserver。
3. 在託管平臺的控制面板裡新增這個域名，讓它自動簽發一張 SSL 證書（SSL certificate，證明這個站點確實屬於你的數字身份證，它會點亮那把小鎖，並開啟 HTTPS——網路傳輸協議的安全加密版本）。

給你一個快速的心智模型，好讓這些記錄不再像魔法：一條 **A** 記錄把一個名字直接指向一個 IP 地址；一條 **CNAME** 把一個名字指向另一個名字（當託管平臺的 IP 可能在你不知情時變化時很有用）；而 **nameserver** 把整個域名交給一個服務商，讓它替你管理每一條記錄。大多數現代託管平臺更偏好用 CNAME 或者整體切換 nameserver——它們想自己管理 SSL 證書和邊緣路由。

下面是一個訪客的請求所經過的整條鏈路，以及你設定的每一塊分別住在其中的什麼位置：



DNS 對新手來說是最令人困惑的一環，多半是因為術語和傳播延遲。讓 AI 來翻譯。問它：“我的域名註冊在 *Namecheap*，託管在 *Cloudflare Pages*。一步一步告訴我究竟要新增哪些記錄、填什麼值，以及如何驗證它生效了。”然後用一次快速檢查親自驗證結果，而不是僅僅重新整理瀏覽器：

```
# 查詢全世界實際看到的記錄
dig yourdomain.com
nslookup yourdomain.com

# 確認證書已上線且未過期
curl -I https://yourdomain.com
```

DNS 改動的傳播可能需要幾分鐘到一天不等的時間，而其中最大的騙子是你自己瀏覽器的快取——在網際網路其餘部分早已更新之後，它還會樂呵呵地給你一個過時的答案。如果它沒有立即上線，那是正常的——在你以為出了問題之前，先用手機的蜂窩資料查一下，或者用一個線上的“DNS 傳播”檢測工具。

環境變數與金鑰

你的程式碼需要 API key、資料庫 URL 和各種 token。這些絕不能被提交到你的倉庫裡。金鑰一旦進了 git 歷史，實際上就等於永遠公開了——在後來的提交裡刪掉那一行並不會把它從歷史中抹去，而機器人會在幾分鐘內爬取新的公開倉庫來尋找洩露的 key。兩條規則能讓你受用很久：

- 在本地把金鑰放在一個 `.env` 檔案裡，並確保 `.env` 在你的 `.gitignore` 中。
- 在託管平臺的控制面板裡（或通過 CLI——Command-Line Interface，命令列介面，那個讓你敲命令而非點選的文字提示視窗）把同樣的值設定為生產環境的環境變數。

```
# .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
```

維護一個會被提交的 `.env.example`，列出每一個變數的名字（而非值），這樣未來的你和 AI 都知道這個應用期望什麼。注意上面那個 `sk_test_` 字首——把你的測試 key 和正式 key 清楚地區分開，絕不要讓一個正式 key 躺在一個你可能會粘進聊天的本地 `.env` 裡。如果你曾經把一個金鑰粘進了聊天，或者不小心提交了一個，就當它已經洩露，並立即輪換它：在服務商控制面板裡生成一個新 key，在所有地方更新它，然後吊銷舊的。輪換花你五分鐘；一個洩露的資料庫 URL 可能讓你失去資料庫裡的一切。

接入支付

如果你想收錢，對獨立開發者而言的標準路徑是 Stripe（或者 Lemon Squeezy / Paddle，如果你想讓它們替你處理銷售稅——作為登記在冊的銷售方（merchant of record），它們抽成更高，但讓稅務的頭疼消失）。這些機制是重複且文件完善的，這讓它們非常適合外包出去。你需要的幾塊拼圖：

- 一個**結賬（checkout）**流程——Stripe Checkout 是最快的；它直接給你一個託管好的支付頁面，讓你永遠不必碰原始卡資訊。
- 把 **webhook** 想像成支付公司在每一筆交易剛一清算成功的那一刻就打電話通知你的應用，而不是讓你的應用反覆地追問“到底成沒成？”。一個 **webhook** 正是如此：它是支付服務商會去呼叫的一個端點（endpoint，你伺服器上的一個 URL），讓你的應用知道一筆支付究竟何時真正成功了。這是新手會跳過的部分，然後他們納悶為什麼訂單從來不會被履約。回到你站點的那個重定向可能被關閉、被攔截或被偽造；webhook 才是伺服器對伺服器的事實來源。絕不要只信任那個重定向；要信任 webhook。
- 一種**儲存權益（entitlements）**的方式——在你的資料庫裡把使用者標記為已付費。這樣重新整理一下或換一臺裝置，他們買的東西依然在。

關鍵的洞察在於你信任哪一根箭頭。把使用者帶回你站點的那個重定向可能丟失或被偽造；真正解鎖產品的，是伺服器對伺服器的那條 webhook 線：



(僅 UI)

驗證簽名 →

標記使用者為 PAID ✓

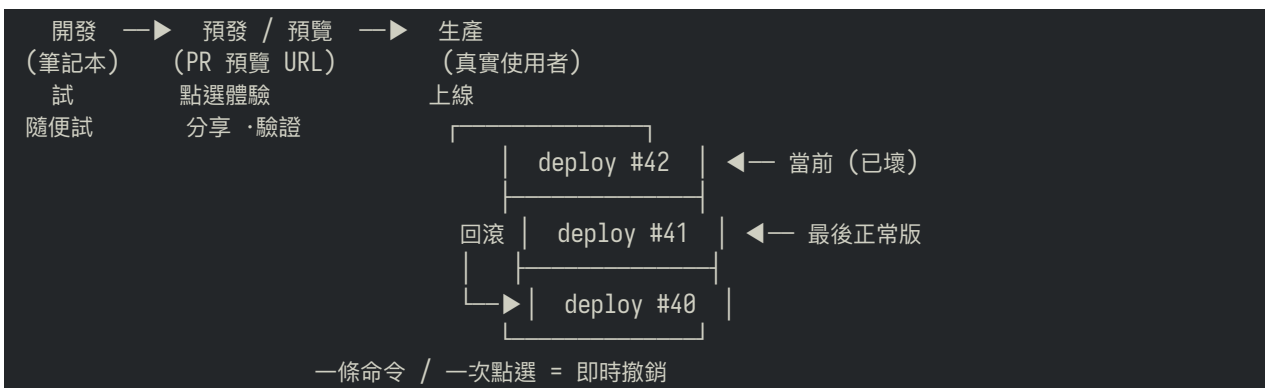
讓 AI 把這三塊都搭好，並解釋 webhook 的簽名驗證，因為那是其中唯一對安全敏感的一步——沒有它，任何找到你 webhook URL 的人都能偽造一個“支付成功”事件，把你的產品免費解鎖。在切換到正式 key 之前，用 Stripe 的測試模式和它們的測試卡號（4242 4242 4242 4242 是最有名的那個）把一切都測一遍，並用 Stripe CLI 把 webhook 事件轉發到你的本地機器，親眼看著它們觸發。不要跳過測試模式這場帶妝彩排。第一筆流經你應用的真錢，應該是一筆你已經看過它成功運作十幾次的交易。

預釋出環境與回滾

有兩個習慣，把一次能優雅恢復的上線，和一次變成午夜恐慌的上線區分開來：在變更觸達真實使用者之前先在安全的地方測試它，以及能夠即刻撤銷一次糟糕的部署。

一個**預釋出 (staging)** 環境只不過是你應用的第二份副本——同樣的程式碼、獨立的 URL、獨立的（或一次性的）資料庫——一個讓你在生產之前先試一試的地方。大多數託管平臺讓這幾乎免費：每一個 pull request 都自動獲得它專屬的**預覽 URL**，於是你可以在合併之前點過一遍一個變更，並把它分享給某個人。對那些你沒法輕易逆轉的東西——比如一次資料庫遷移——尤其要用它。

把這三個環境想像成一架變更要往上爬的梯子，而回滾就是那根能把生產環境直接落回上一個好梯級的繩子：



一次回滾是你對整個應用的撤銷按鈕。因為這些平臺保留著此前的每一次部署，回到上一個好版本通常就是一條命令或一次面板點選：

```
# Cloudflare Pages: 重新部署一個已知良好的構建
wrangler pages deployment list
wrangler pages deployment rollback <DEPLOYMENT_ID>

# Vercel: 把一次之前的部署重新提升為生產環境
vercel rollback <DEPLOYMENT_URL>
```

在你需要它之前就知道這條命令，而不是在故障當中才去找。vibe coding 速度的全部意義在於釋出很便宜——但只有當撤銷釋出同樣便宜時，便宜的釋出才讓人感覺安全。

監控與錯誤追蹤

一旦有人用你的應用，你就需要知道它何時壞了——最好在他們告訴你之前。這套輕量級的入門工具包：

- **錯誤追蹤**：Sentry 會帶著堆疊跟蹤捕獲並歸類異常，並告訴你是哪個使用者、哪一行觸發了每一個錯誤。幾行程式碼就能裝好。
- **可用性監控**：一個免費的 pinger (UptimeRobot、Better Stack) 每隔幾分鐘敲一下你的站點，並在它宕機時給你發郵件。

- **日誌**：搞清楚你的託管平臺把它們放在哪（`wrangler tail`、Vercel 控制面板等），以便有什麼看起來不對時你能去排查。

讓 AI 加上 Sentry，並接一個基本的健康檢查（`health-check`）端點——一個像 `/health` 這樣的小路由，返回 200 OK 並確認資料庫可達，讓你的可用性監控可以去 ping 它。目標不是企業級的可觀測性——它只是別從一條憤怒的推文裡才得知出了故障。

效能與快取

慢吞吞的應用會流失使用者，並在搜尋中排名更靠後。你不需要在第一天就微微最佳化，但可以先把那些廉價的便宜拿到手：

- 想像把你的檔案複製成很多份，分放在全世界各地的倉庫裡，於是每位訪客都由離他最近的那一份來提供服務。那就是 **CDN**（Content Delivery Network，內容分發網路）——通過它來提供靜態資源（大多數託管平臺會自動這麼做）。
- **快取**那些不常變化的昂貴響應和資料庫查詢。
- 壓縮並懶載入圖片；少發一點 JavaScript。
- 在 Chrome DevTools 裡跑一遍 **Lighthouse**，把報告餵給 AI 來排出優先順序修復。

問它：“這是我的 **Lighthouse** 報告。給我三個用最少力氣換最大影響的改動。”讓度量、而非憑空猜測，來驅動你最佳化什麼——僅僅因為它感覺慢就花一整天去最佳化錯誤的東西，這真的很常見，而真正的代價其實是一條沒建索引的查詢，或一張你忘了壓縮的 4 MB 全屏大圖。

上線前清單

在你宣佈任何事之前，過一遍這份清單。讓 AI 幫你逐項驗證，而不是憑記憶。

- ☐ 應用乾淨地部署，且生產環境 URL 能載入
- ☐ 自定義域名解析正常，且 HTTPS 有效
- ☐ 所有金鑰都已在生產環境設定；沒有一個被提交到倉庫
- ☐ `.env` 已被 `gitignore`，且 `.env.example` 是最新的
- ☐ 支付已在測試模式下端到端測過（包括 `webhook`）
- ☐ 註冊、登入以及核心 `happy path` 在生產環境都能用
- ☐ 錯誤追蹤和可用性監控都已上線
- ☐ 你知道怎麼回滾，而且已經測過一次
- ☐ 移動端佈局已在一臺真實手機上檢查過
- ☐ 存在一個基本的隱私政策頁和條款頁
- ☐ 資料庫備份已啟用
- ☐ 有一條清晰的渠道讓使用者聯絡到你（郵箱或表單）

上線之後

釋出是開始，而不是終點。第一個版本是一個假設；真實使用者才是實驗。盯著你的錯誤面板，讀每一條訊息，並快速釋出一小修復。要剋制住一遇到摩擦就推倒重來的衝動——把尖銳的稜角打磨掉，在人們真正使用的東西上加倍投入，讓其餘的等一等。

這裡最強大的習慣是一個緊湊的迴圈：注意到一個問題或一個想法，把它描述給 AI，釋出改動，觀察結果。因為機械性的工作被外包了出去，你可以用幾小時而非幾周來迭代。正是那份速度——而非任何單一功能——才是憑著 **vibe** 來構建的真正優勢。你已經上線了。現在，繼續前進。

回顧與練習

關鍵要點

- 上線是一套工作流，而不是一個瞬間：部署、觀察、修補、重複——並且把回滾做成你已經測試過一次的東西。
- 在宣佈任何東西之前先過一遍上線前檢查清單，讓 AI 逐項核驗，而不要依賴你的記憶。
- 讓測量來驅動最佳化——**Lighthouse** 報告永遠勝過“感覺很慢”的直覺。

- 金鑰放在生產配置裡，絕不放進倉庫；`.env` 加入 `gitignore`，`.env.example` 保持最新。
- 第一個版本是假設，真實使用者才是實驗，所以要用小而快的修復來迭代。

動手試試

挑一個你已部署的應用 (或現在就部署一個很小的)，逐行對照走一遍完整的上線前檢查清單。對每一項你無法誠實勾選的，把失敗的那一項貼給 AI，請它給出彌補差距的最小改動。最後點選線上站點的一個深層連結一而不只是首頁—確認生產環境的行為和開發環境一致。

本章提示詞

```
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：從 0 到上線



理論是有用的，但沒有什麼比親眼看著它發生更能教會你 **vibe coding**。本章會走過四個小而真實的專案，它們都是通過指揮 AI 從頭到尾搭建併發布上線的。它們沒有一個是玩具。每一個都有使用者、一個域名，以及押在上面的金錢或時間。

對於每一個專案，你都會看到：想法、我們交給 AI 的 **spec**、我們選的技術棧、真正起作用的那些 **prompt**、差點讓我們脫軌的障礙，以及它是如何上線的。把這些當作操作手冊（**runbook**）來讀，而不是精彩集錦。重點不在於這幾個具體專案本身有多重要——而在於那些招式會重複。到第四個時，你會開始認出那個節奏：一段簡短的 **spec**、一套託管的技術棧、靠證據驅動的除錯，以及一次拿真實世界來檢驗的上線。

案例研究 1：一個多語言落地頁 + 支付頁

想法

一位朋友在賣一份售價 29 美元、面向移居韓國的外籍人士的 PDF 指南。她想要一個單頁，全球載入都快，會說英語、韓語和中文，並且能在她完全不碰伺服器的情況下收取銀行卡支付。

Spec

在寫任何程式碼之前，我們把 **spec** 控制在了一段話：

```
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.
```

注意這段 spec 沒有說什麼：沒有框架，沒有設計系統，沒有分析統計，沒有郵件收集。這其中每一項都本可以是一個合理的功能，而每一項也都會拖慢首次上線。一段話 spec 的紀律，大半是把東西留在外面的紀律。

技術棧

“沒有我要維護的後端”加上“全球都快”，直接指向了在邊緣 CDN（Content Delivery Network，內容分發網路——把你的檔案複製成多份存放在世界各地的資料中心裡，於是每位訪客都從就近的一份載入）上的靜態託管，外加一個用於唯一需要金鑰的事情——建立 Stripe session——的微型 serverless 函式。我們選擇了在 Cloudflare Pages 上的一個靜態站點，配上一個單獨的 Pages Function。整個架構一句話就能裝下——一個靜態頁呼叫一個函式、那個函式再跟 Stripe 對話——而正是這份簡單，讓它後來變得可除錯。

整個系統就是三個盒子，而其中只有中間那個會接觸到金鑰：



關鍵的 prompt

我們從寬泛處起步，讓 AI 提議結構：

```
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.
```

然後是支付路徑，金鑰就住在這裡：

```
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”這一句配得上它的位置。任由它按預設行事，AI 會很樂意把一個佔位金鑰內聯進去，好讓示例“跑起來”，而佔位金鑰往往會變成一個真上線的真金鑰。在 prompt 裡大聲說一次，比在評審裡抓出來要便宜。

障礙

第一次線上測試失敗了：Buy 按鈕毫無反應，瀏覽器控制台顯示一個 CORS（Cross-Origin Resource Sharing，跨源資源共享——瀏覽器關於哪些站點被允許呼叫哪些站點的規則）報錯。我們沒有去猜。我們把確切的報錯粘了回去：

```
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 立刻發現了它：這個函式確實在以 JSON（JavaScript Object Notation，JavaScript 物件表示法——程式之間用來交換結構化資料的一種簡單、易讀的文本格式）形式返回 Stripe URL，但 fetch 在函式的 OPTIONS 預檢（preflight）被處理之前就發起了，因為我們誤把靜態頁和這個函式部署到了兩個不同的 Pages 專案裡。真正的修復是部署拓撲，而不是程式碼。我們把函式移進了同一個專案的 /functions 目錄，報錯就消失了。

這個教訓值得多停留一下：症狀在程式碼裡（一個來自 fetch 的 CORS 報錯），但病因在基礎設施裡（兩個專案而不是一個）。如果我們沒有把配置粘上去，只讓 AI”修這個 CORS 報錯”，它就會硬塞上 Access-Control-Allow-Origin 頭——一個能編譯通過、把症狀糊住、卻讓真 bug 繼續活著的修復。把字面的報錯連同它周圍的上下文一起粘上去，並且如果第一個貌似合理的解釋跟你的實際配置對不上，就別接受它。

上線

我們把正式的 Stripe key 作為加密的環境變數加進了 Pages 控制面板（絕不進倉庫），把她域名的 DNS (Domain Name System, 域名系統，網際網路的地址簿，把一個名字對映到一臺伺服器) 指向了 Pages，並用一張銀行卡跑了一次真實的 29 美元測試購買，然後退了款。我們還特意走了一遍失敗路徑——一張會被拒的測試卡——以確認她不會被扣費，並且看到的是一條像樣的訊息而不是一片空白。一個下午就上線了。她當晚就做成了第一筆銷售。

案例研究 2：一個帶認證 + 資料庫的小型 SaaS 工具

想法

一位自由職業設計師想要一個私人儀表盤，用來按客戶記錄可計費工時，並匯出每月的 CSV (Comma-Separated Values, 逗號分隔值——一種純文本的電子表格檔案，任何電子表格應用都能開啟)。沒什麼花哨的，但它需要賬戶（這樣她的資料只屬於她自己）和持久化儲存。

Spec

認證和資料庫抬高了賭注，所以 spec 在邊界上變得更具體了：

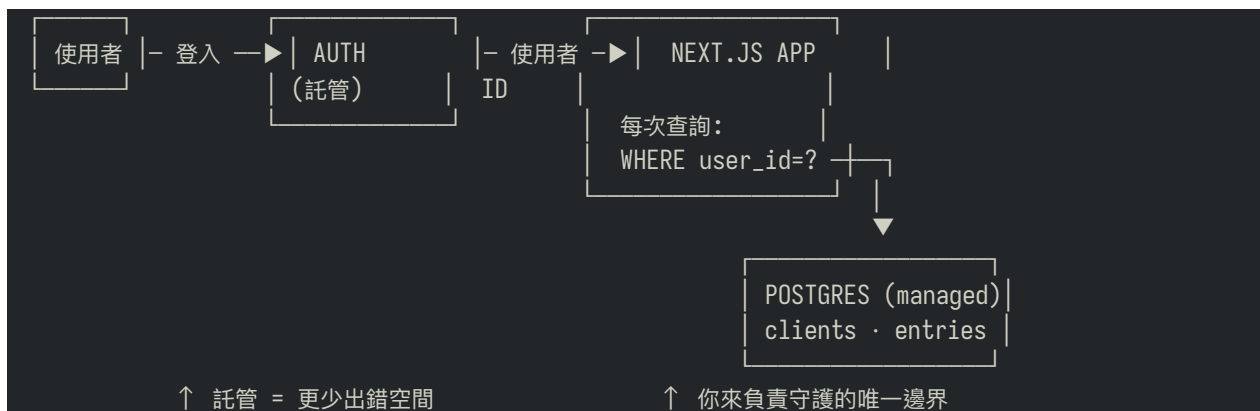
```
A logged-in web app where a user can: sign up / log in with email,
create clients, log time entries (date, client, hours, note),
see a table of entries filtered by month, and download that month
as CSV. Each user only ever sees their own data. Mobile-friendly.
```

“Each user only ever sees their own data”看著像一行 UX 描述。它其實是把整個應用的安全模型壓縮進了九個詞裡。把它寫進 spec，意味著每次 AI 跑偏時我們都能指回這一句。

技術棧

對一個獨立開發者來說，制勝的一手是選一個讓認證和資料庫都成為託管服務、而非你自己寫的程式碼的技術棧。我們選了一個部署在 serverless 託管平臺上的 Next.js 應用，配上一個託管的 Postgres 資料庫，以及一個開箱即用的認證提供方，由它替我們處理郵箱登入、會話和密碼重置。能寫錯的程式碼越少，AI 能替我們寫錯的程式碼就越少。認證尤其是那種你幾乎從不想跟 AI 一起手搓的類別：它的失敗模式是靜默的，爆炸半徑是所有人的賬戶，而一個託管提供方早已被幾百萬次登入把邊界情況捶打過。

託管的部件都在外圍；你自己掌管的那唯一一條規則在中間——每一條查詢都按 user_id 過濾，於是一個使用者永遠讀不到另一個使用者的資料行：



關鍵的 *prompt*

我們讓認證提供方自帶的模板來挑大樑，然後指揮 AI 在它之上疊加業務邏輯：

```
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”這一行是整個專案裡最重要的一句話。我們在幾乎每一個觸及資料的 *prompt* 裡都重複了這條約束，因為一個多使用者應用裡最可怕的單一 bug，就是一個使用者看到了另一個使用者的資料行。打字時這種重複感覺是冗餘的；可正是這份冗餘救了你，因為模型並不記得這條約束在一個個獨立的 *prompt* 之間有多要緊。

至於匯出：

```
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.
```

障礙

在測試時，我們建立了兩個帳戶，然後發現帳戶 B 能在一個下拉選單裡看到帳戶 A 的客戶。這正是我們害怕的那個 bug。我們沒有讓 AI“修好它”，而是先讓它把問題證明出來：

```
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.
```

這次審計浮出了一個查詢——那個下拉選單的載入器——它是在我們加上那條約束之前寫的，於是溜了過去。我們讓它補上了缺失的過濾條件，然後又要了一道守衛：

```
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.
```

這把一次性的修復變成了一個結構性的保證。接著我們讓這個保證變得可測試，因為一道你沒法驗證的守衛不過是一種指望：

```
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.
```

教訓：當 AI 引入一個安全 bug 時，別隻修補這一個例項——要指揮它去消除這一整類錯誤，然後用一個一旦有人重新開啟就會大聲失敗的測試，把這一類鎖死。

上線

我們灌入了一個月的測試資料，匯出了 CSV，在電子表格裡開啟它以確認數字和編碼都正確，然後設定了一個強資料庫密碼，並把憑據從所有本地檔案裡輪換出去。我們部署到了 *serverless* 託管平臺，在它的控制面板里加上了生產環境的環境變數，然後把 URL 給了她。她用一次真實的註冊完成了自助入駐。整個構建只花了一個週末。

案例研究 3：一個個人自動化小工具

想法

一個純粹利己的專案：一個指令碼，每天早上抓取當天的日曆事件和天氣，格式化成一行摘要，傳送到一個私人的 Telegram 聊天裡。重點是別在喝咖啡之前還要開啟三個應用。

Spec

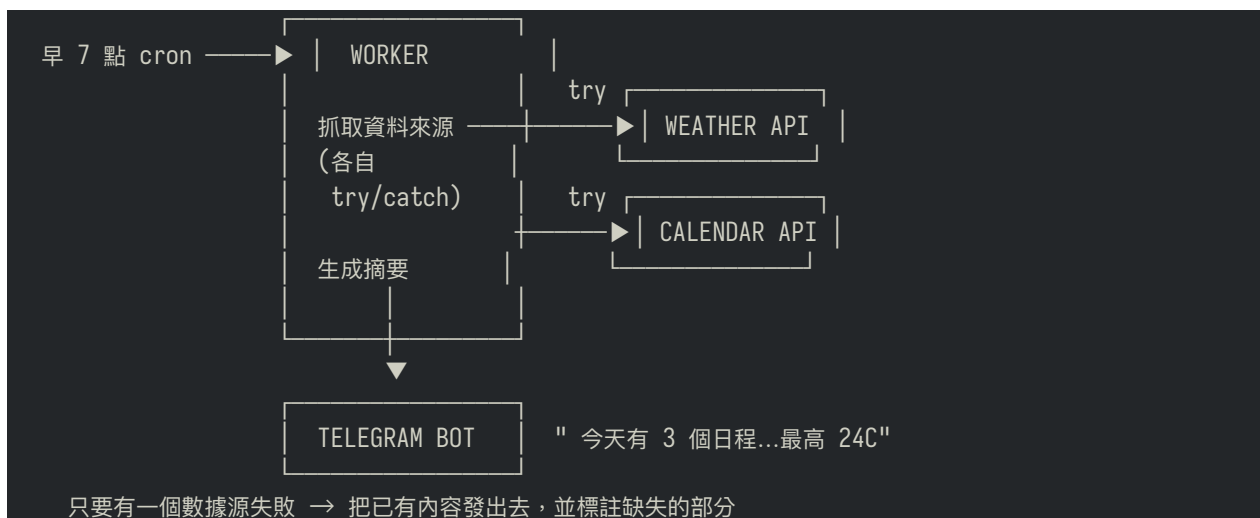
```
A scheduled job that runs at 7am my timezone. It reads today's events from my calendar, fetches the forecast for my city, and sends one Telegram message like: "3 events today, first at 9:30. High 24C, rain after 4pm." If any source fails, still send what you have and note what's missing.
```

最後那句話——優雅地降級而不是崩潰——是那種你必須明確提出來要求的行為。AI 不會自動假定它。不說出來，模型會去抓的預設就是「出錯就拋異常」，而對一個早上 7 點的 cron 任務來說，那意味著一個靜悄悄的早晨，以及對原因毫無頭緒。

技術棧

一個單獨的、定時觸發的 serverless 函式（一個由 Cron 觸發的 Worker），這樣就沒有一臺需要保持執行的機器。整個東西就是一個檔案加一份排程計劃。對一個個人小工具來說，這比聽上去更要緊：任何有一臺需要照看的伺服器的東西，你遲早會忘了續費，於是這個專案死於荒廢，而不是死於失敗。

一個觸發器扇出到兩個資料來源，再匯聚回一條訊息。包住每個資料來源的 try/catch，讓單個失敗優雅降級，而不是殺掉整次執行：



關鍵的 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.
```

當我們想要打磨措辭，又不想等到早上 7 點時：

```
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.
```

這個測試觸發器把我們從一個殘酷的反饋迴圈裡救了出來——我們可以隨時按需看到輸出，而不必每天只看一次。這是個值得偷師的通用招式：每當你的程式碼按計劃執行時，在你認命去等時鐘之前，先給自己造一個能現在就按需跑它的法子。

障礙

Telegram 訊息到了，但時間是錯的：事件顯示的是 UTC，而不是韓國時間。天氣倒是沒問題。我們把它隔離了出來：

```
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 解釋說，天氣 API (Application Programming Interface，應用程式程式設計介面) 已經返回了本地化的字串，而日曆 API 返回的是我們直接原樣打印出來的 UTC 時間戳。它在日曆格式化那一步加上了一次時區轉換。把修復範圍限定在“只改日曆路徑”，阻止了它去重寫那段本來就工作正常的天氣程式碼——這是 AI 的修復弄壞本來好端端的東西的一種常見方式。那個 prompt 的後半句——“tell me why the weather path was already right”——一身二任：它給了我們真正的根因而不是一個猜測，並且逼著 AI 把這個差別講清楚，好讓它的修復不至於把兩條路徑意外“統一”成一條壞的。

上線

這裡的“上線”只不過意味著部署 Worker、設定 cron 排程計劃，以及把 API token 作為加密金鑰加進去。我們點了幾次？test=1 端點以確認訊息讀起來不錯，然後就放著不管了。從那以後它每天早上都在執行——而那條優雅降級的話，在一個月後天氣 API 出故障時證明了它的價值：訊息照樣到了，只是少了天氣預報而已。

案例研究 4：一個不用你打理的內容站

想法

一位業餘愛好者想把一個 Markdown 筆記資料夾——多年旅行積攢下來的餐廳點評——變成一個可搜尋的公開站點，沒有要登入的 CMS，也沒有月度賬單。加一條筆記，推送，搞定。

Spec

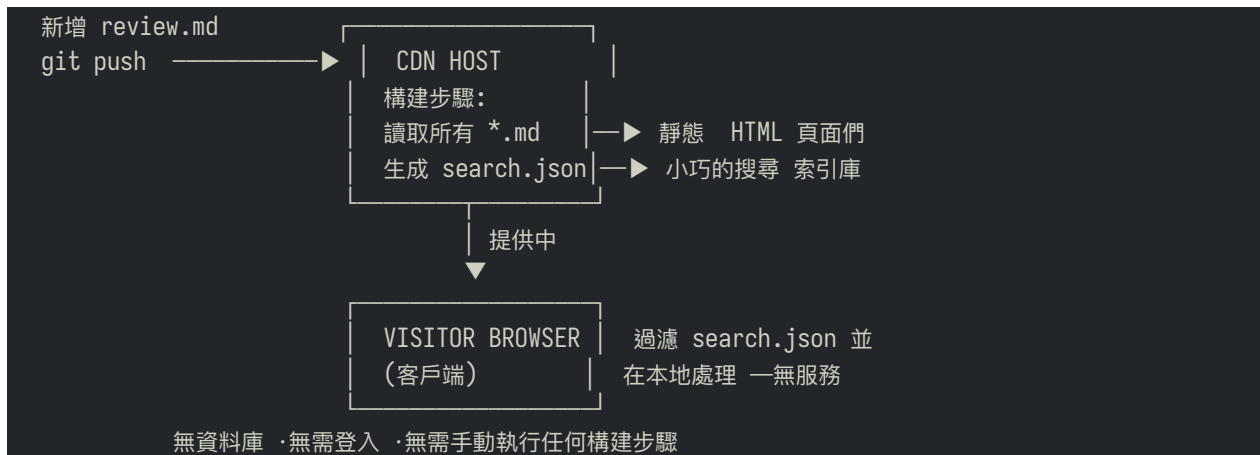
```
A static site generated from a folder of Markdown files, one file  
per review (title, city, rating, body in frontmatter). Build a  
homepage that lists reviews newest-first, a page per review, and  
client-side search by city or name. No database, no login, no  
build step I have to run by hand.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

隱藏的需求在最後一句裡。“No build step I have to run by hand”意味著這段 spec 其實講的是工作流，而不只是產物——而工作流需求正是那種你不點名 AI 就會跳過的類別，因為它們不會在跑起來的應用裡露面。

技術棧

一個在構建時讀取 Markdown 的靜態站點生成器，部署到一個帶 Git 觸發構建的 CDN 託管平臺上。推送到倉庫，託管方重新構建，CDN 來分發。搜尋在構建時生成的一個小 JSON 索引上做客戶端搜尋，所以沒有伺服器、也沒有查詢成本——在幾百條點評這個量級上這是個不錯的取捨，而且是我們點明瞭的、而非撞上的一個有意選擇。

整個工作流就是一根箭頭：一次推送觸發一次會發布的構建。除了訪客自己的瀏覽器過濾索引之外，請求時刻沒有任何東西在執行：



關鍵的 *prompt*

我們先把 AI 錨定在資料形狀上，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”是一根有意為之的槓桿。一次性把整個東西要過來，你會得到一個連著你信不過的資料、卻很漂亮的頁面；先把資料層要過來，你就能在哪怕一個畫素分散你注意之前，先驗證好地基。然後是搜尋，被框住以保持廉價：

```
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.
```

障礙

構建在本地通過了，但部署後的站點在每一個單獨的點評頁上都 404。首頁能用；按點評分的頁面不行。我們沒去做推理，而是把症狀和配置粘了上去：

```
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.
```

我們把輸出目錄指向了託管方真正部署的那個，頁面就出現了，並且我們在上線檢查清單里加了一行：在線上站點點一個深

層連結，而不只是首頁。更廣的教訓是，“在 dev 裡能用”和“部署後能用”是兩個不同的主張，而靜態匯出恰恰就是這兩者分道揚鑣的地方——dev 伺服器在路由上的寬容，是真正的基於檔案的託管永遠不會有的。

上線

我們推送了一條真實的點評，看著託管方重新構建，然後在線上域名載入了通向那一條具體點評的深層連結——也就是 dev 撒過謊的那個案例。然後我們搜了一下它的城市，確認索引把它收進來了。加下一條點評是一次一行的提交，而這正是全部意義所在：上線不是一個時刻，而是一個不用我們就能繼續運轉的工作流。

這四個專案的共同點

- **Spec 先行，而且很短。**一段話，而不是一份文件。但它點明瞭那些重要的約束——語言、“只能看到自己的資料”、“先把你有的發出來”、“推送一個檔案就該釋出它”——包括那些永遠不會在跑起來的應用裡露面的工作流約束。
- **金鑰從未碰過倉庫。**每個專案都從環境變數或加密金鑰裡讀取 key，這些都設在控制面板裡，而不在程式碼中。我們把“never hardcode it”大聲說了出來，因為預設恰恰相反。
- **技術棧是按“能寫錯的更少”來選的。**託管認證、託管資料庫、靜態託管、serverless 函式——每一個選擇都拿掉了一類 AI 能替我們弄壞的程式碼，以及一類我們得去維護的東西。
- **障礙是靠餵給 AI 確切的證據來解決的——**字面的報錯、實際的程式碼、部署配置、真實的症狀——並且通過把修復範圍限定得很窄，讓能工作的程式碼保持能工作。有兩次，症狀在程式碼裡，病因卻在基礎設施裡。
- **上線意味著一次對現實的真實測試：**一次真實的購買（以及一次被拒的）、第二個使用者帳戶、一次按需觸發、線上站點的一個深層連結。而不是“它編譯通過了”。

你不需要一個宏大的想法才能開始。你需要的是一個小想法、一段話的 spec，以及指揮而非敲鍵盤的意願——以及在你稱它做完之前，拿現實去檢驗它的那份耐心。

回顧與練習

關鍵要點

- 規格先行，而且很短——一段話點明瞭真正重要的約束，包括那些在執行的應用裡永遠看不到工作流約束。
- 金鑰從未碰過倉庫；每個專案都從環境變數或在儀表盤裡設定的加密配置中讀取金鑰。
- 技術棧是為了少出錯而選的一託管認證、託管資料庫、靜態託管、無伺服器函式。
- 障礙是靠給 AI 喂確切證據來解決的一真實的報錯、實際的程式碼、部署配置——並把修復範圍收得很窄。
- 上線意味著對現實的真實測試，而不是“它編譯通過了”：一筆真實購買、第二個帳戶、線上站點上的一個深層連結。

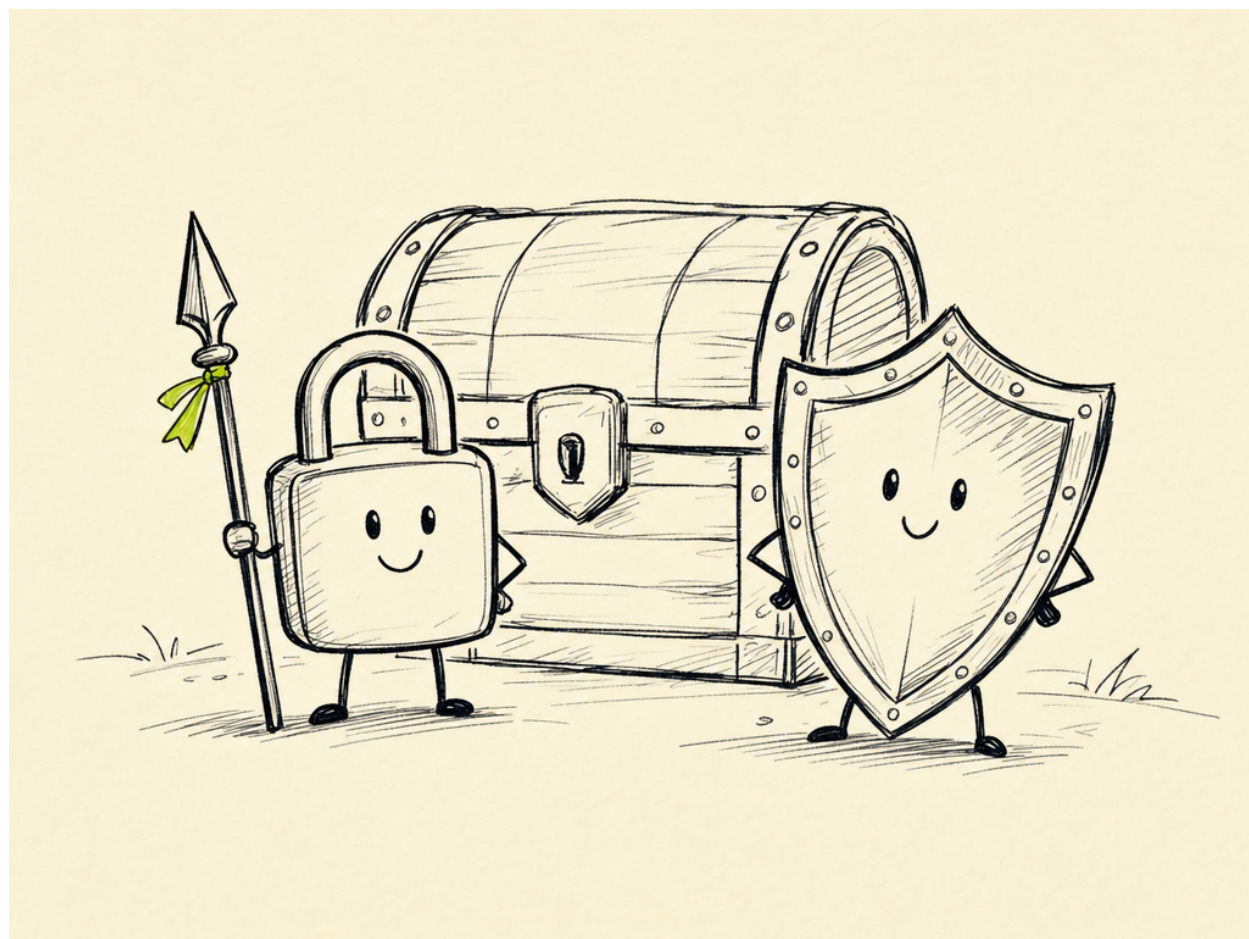
動手試試

挑一個你自己的點子，像這四個案例那樣寫出它的一段式規格：點明核心動作、每個使用者能看到的資料，以及至少一個工作流約束（內容如何釋出、誰能做什麼）。然後列出你會依賴的託管構件，以減少會出問題的自定義程式碼。控制在一段話加一個簡短的專案符號列表——如果超出了，就說明你規格定得過頭了。

本章提示詞

```
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.
```


寫給 *Vibe Coder* 的安全課



安全是 *vibe coding* 裡那種會悄悄反咬你一口的部分。一個功能壞了，問題立刻就暴露——按鈕沒反應，你隨手就修了。可一個安全漏洞卻幾乎從不暴露，直到有一天有人發現了它，把你使用者的資料扔到論壇上。到那時候，寫出這個漏洞的 AI 早就不知道去哪了，而要面對一千個人、解釋他們的密碼為什麼洩露的，是你。

這一章不是要嚇唬你。你不需要變成滲透測試專家，也不用背下 OWASP Top 10，才能上線一個安全的小應用。你需要的是理解軟體被攻破的那麼幾種常見方式，認出 AI 反複製造的那些模式，並且在上線前加上一道審查關卡。這背後的理念和本書其餘部分一樣：**永遠不要上線你自己不理解的東西**——而安全正是“不理解”代價最大的那個地方。

為什麼 *AI* 生成的程式碼預設就是不安全的

AI 是在為你給它的提示詞做最佳化，而你的提示詞幾乎總是說“讓它能跑起來”，從不說“讓它安全”。於是它產出的是通往一個能用功能的最短路徑。而最短路徑通常就是不安全的那條：用字串拼接出來的查詢、一個沒有任何許可權檢查的介面 (endpoint, 端點——通往你應用、可被訪問的那些門之一，也就是它用來應答請求的一個具體 URL)、一個直接內聯貼上進去的金鑰——因為它學到的示例就是這麼寫的。

更糟的是，AI 是從整個網際網路學來的——包括十年裡那些本身就不安全的教程和廢棄倉庫。在它的訓練資料裡，不安全的程式碼比安全的程式碼更常見，因為安全的程式碼更難寫、也更少有人發出來。當你讓它寫“一個登入表單”時，從統計上看你很可能拿到網際網路上那個平均水平的登入表單，而網際網路上那個平均水平的登入表單是有問題的。

AI 也完全沒有威脅意識。一個人在做檔案上傳功能時會閃過一絲擔憂——要是有人上傳點噁心的東西怎麼辦？AI 什麼都感覺不到。無論這段處理邏輯是滴水不漏還是門戶大開，它都用同樣的自信把它寫出來。沒有遲疑，沒有那句“嗯，這部分讓我有點不放心”。這份不放心，得你自己來提供。

要點不是“AI 程式碼很危險，別用它”。要點是“它能跑”和“它安全”是兩個不同的問題，而 AI 只會回答第一個，除非你逼它回答第二個。

注入：當輸入變成了程式碼

注入是最古老、至今仍最常見的嚴重漏洞。它發生在使用者提供的資料被當成指令而不是資料來處理的時候。對大多數應用來說，有兩種型別值得關注。

SQL injection (SQL 注入) ——SQL (Structured Query Language, 結構化查詢語言) 是應用用來向資料庫索要資料的語言——是使用者輸入混進了資料庫查詢。經典的脆弱寫法：

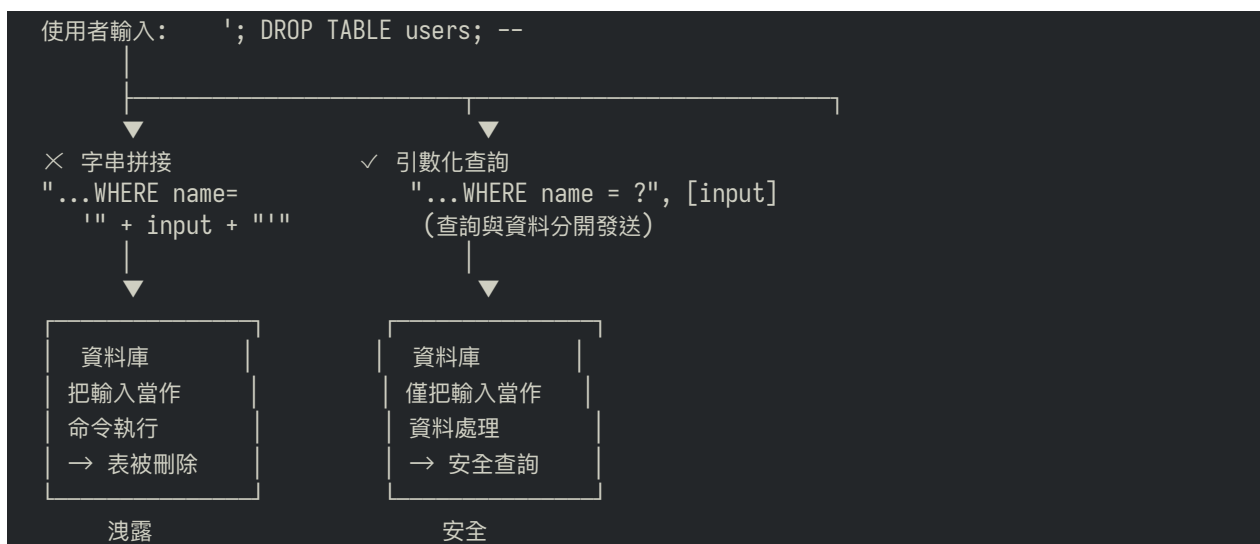
```
// VULNERABLE: 使用者的輸入被直接粘進了查詢裡
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query(`SELECT * FROM users WHERE name = '${name}'`);
});
// 如果有人傳入 name = '; DROP TABLE users; --
// 你的查詢就變成了一條刪表的命令。
```

修復辦法是用**引數化查詢**（也叫預編譯語句）。你把查詢和資料分開發送，這樣資料庫就永遠不會把其中一個錯當成另一個：

```
// SAFE: 值作為引數傳入，永遠不作為程式碼
app.get("/user", (req, res) => {
  const name = req.query.name;
  db.query("SELECT * FROM users WHERE name = ?", [name]);
});
```

規則是：**永遠不要用字串拼接來構造查詢**。如果你在 diff 裡看到反引號或 + 把 SQL 拼在某個變數周圍，停下來，要求改成引數化的版本。

兩條路都把同一份使用者輸入送進資料庫，但只有一條能讓資料庫不把那份輸入誤當成命令：



XSS (cross-site scripting, 跨站指令碼) 是同一個思路在瀏覽器裡的版本。如果你拿使用者輸入、把它當作原始 HTML (HyperText Markup Language, 超文本標記語言, 告訴瀏覽器該顯示什麼的程式碼) 塞進頁面，攻擊者就能注入一個 `<script>` 標籤，讓它在你其他使用者的瀏覽器裡執行——比如竊取他們的會話。修復辦法是轉義：把使用者內容當作文本來渲染，而不是 HTML。像 React 這樣的現代框架預設會轉義，這很棒——直到 AI 為了“讓它能跑”伸手去用 `dangerouslySetInnerHTML` 或 `innerHTML` 這種逃生口。這些會繞過保護。在 diff 裡看到任何這類呼叫，都要當成需要質疑的東西。

每個注入漏洞背後統一的原則是：讓資料和程式碼保持分離。每當使用者輸入要進入一個查詢、一條命令、一個模板或 HTML 時，總得有東西去轉義它或引數化它。

金鑰與 *API key*：金鑰外洩陷阱

金鑰是任何能授予訪問許可權的東西：API (Application Programming Interface，應用程式程式設計介面) key、資料庫密碼、支付服務商令牌、簽名金鑰。vibe coding 裡最常見——也最昂貴——的錯誤，就是洩露其中之一。

有兩個陷阱不斷地坑住人：

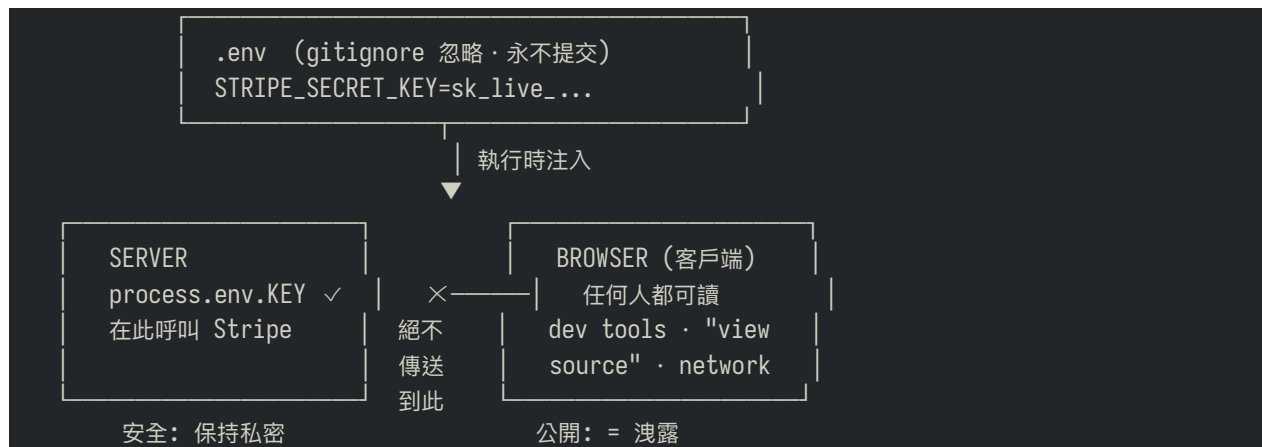
- **客戶端程式碼裡的金鑰**。你前端裡的任何東西（在瀏覽器裡執行的那段 JavaScript）都是公開的。使用者可以開啟開發者工具直接讀到它。AI 在被要求從 React 元件裡呼叫一個 API 時，會樂呵呵地把你的金鑰直接粘在那兒——於是現在任何訪問你網站的人都能把它複製走、把你的賬單刷爆。金鑰應該待在伺服器上，永遠不要出現在會發送到瀏覽器的程式碼裡。
- **倉庫裡的金鑰**。硬編碼在檔案裡的金鑰會被提交進 git。就算你之後刪掉它，它也會永遠留在 git 歷史裡，而機器人會在你 push 後幾分鐘內就掃描公開倉庫，找的正是這種東西。

金鑰正確的歸宿是環境變數 (environment variable) ——一個在執行時提供給應用的值，完全不寫進程式碼裡：

```
// VULNERABLE: key 在原始碼裡，會被提交進 git
const stripe = new Stripe("sk_live_51H8xQ2eZvK...");

// SAFE: key 從環境中讀取，從不寫在程式碼裡
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY);
```

金鑰只有一個安全的家——在伺服器端，從環境裡讀取。瀏覽器是公共場所：任何被送到那裡的東西，任何開啟開發者工具的人都能讀到。



在你寫下第一個金鑰之前，就把 .env (以及 .env.local 等等) 加進你的 .gitignore。如果某個金鑰確實進了你的程式碼或歷史，就把它當成已經燒掉了：輪換它 (生成一個新的並吊銷舊的) ——刪掉那一行是不夠的，因為舊金鑰依然有效，而且依然在外面流傳著。

認證 vs 授權：那個誰都能調的介面

這兩個詞聽起來很像，而二者的區別正是真正的資料洩露藏身之處。

- **認證 (Authentication)** 回答的是你是誰？——登入、證明身份。
- **授權 (Authorization)** 回答的是你被允許做什麼？——這個已登入的使用者是否可以對這份資料執行這個操作。

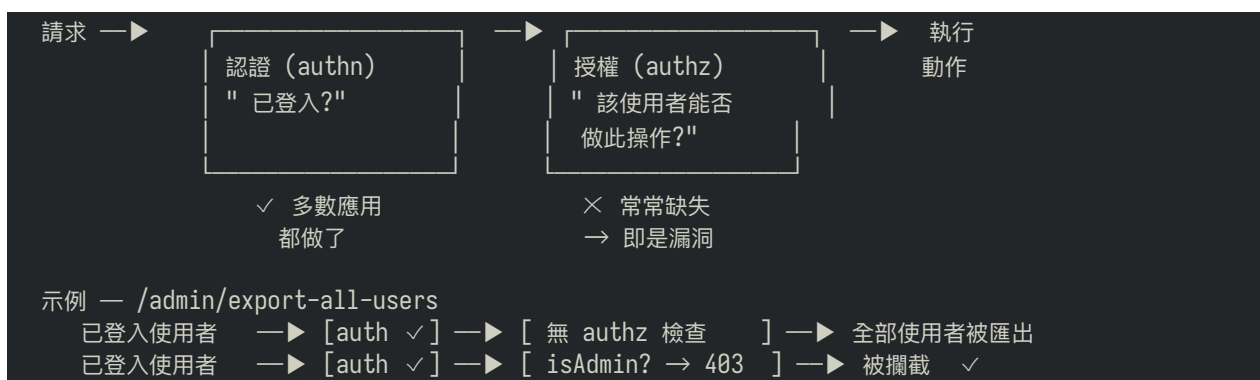
AI 在認證上還算不錯；庫幫它處理了大部分。授權才是它經常翻車的地方，因為授權是你應用規則所特有的，而 AI 並不知道這些規則。教科書級的災難：

```
// VULNERABLE: 檢查了你是否登入，卻沒檢查你是誰
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  res.json(db.getAllUsers()); // 任何已登入使用者都能打這個介面
});
```

那個介面是“受保護的”——你必須登入。但任何已登入的使用者，包括一個三十秒前才註冊的，都能呼叫它、下載到每一個使用者的資料。認證有了，授權沒有。修復辦法是在操作本身上檢查許可權：

```
// SAFE: 確認這個使用者確實是管理員
app.get("/admin/export-all-users", requireLogin, (req, res) => {
  if (!req.user.isAdmin) return res.status(403).send("Forbidden");
  res.json(db.getAllUsers());
});
```

把一個請求必須通過的兩道門想像出來。認證問“你登入了嗎？”；授權問“你是被允許做這件事嗎？”。當一個應用建了第一道門、卻忘了第二道門時，入侵就發生了：



同樣的陷阱以縮小版的形式到處出現：一個介面返回訂單 #1234，卻不檢查這個訂單是不是屬於發起請求的那個使用者。任何人都能改一改 URL 裡的數字，讀到別人的訂單。規則一點也不光鮮，但絕對：**在每一個接觸資料的介面上檢查授權，並且不要信任來自客戶端的 ID**。不要因為一個 URL 很隱蔽就假定它安全——“沒人知道這玩意兒存在”不是一種安全控制手段。

檔案處理與上傳

讓使用者上傳檔案是一片悄無聲息的雷區，而 AI 的初稿幾乎從來不會堵上那些洞：

- 校驗型態和大小。沒有限制，就會有人上傳一個 10 GB 的檔案或一個可執行程式。檢查實際的內容，而不只是檔名字尾——字尾誰都能改。
- 永遠不要信任檔名。一個像 ../../etc/passwd 這樣的檔名，可能讓一個天真的處理邏輯把檔案寫到目標資料夾之外（這叫 path traversal，路徑穿越）。自己生成安全的檔名，而不是用對方給的那個。
- 不要從和你應用相同的源（*origin*）上提供上傳的檔案，也不要讓上傳的檔案被當作程式碼執行。一個使用者提供的檔案落進了伺服器會去執行的資料夾，那就是一條直通被接管的路。

這些你不必全都自己造——一個儲存服務能幫你搞定大部分——但你得主動去要，因為 AI 不會主動給你。“加一個上傳功能”給你的是理想路徑。“加一個上傳功能，要校驗檔案型態和大小、除了圖片之外一律拒絕、並用生成的檔名來儲存”給你的，才是更接近安全的東西。

依賴風險：AI 憑空發明的那個包

現代應用依賴著幾十個第三方包，而 AI 推薦起它們來毫不手軟。兩個風險隨之而來。

第一，**typosquatting**（仿冒搶注）和幻覺出來的包。攻擊者會發布惡意的包，名字和真包只差一丁點（requests 而不是 request），賭的就是你打錯字。AI 有時會信心滿滿地匯入一個根本不存在的包——而一個注意到這種習慣的攻擊者，就能

去註冊那個一模一樣的名字、裡面塞上惡意程式碼。在安裝 AI 推薦的任何東西之前，瞄一眼它：它真的存在嗎，它有真實的下載量和倉庫嗎，名字的拼寫是你預期的那樣嗎？

第二，未經審查的依賴（總體而言）。你加進來的每一個包，都是以你應用的全部許可權在執行的程式碼。依賴越多，意味著 bug 和供應鏈攻擊的暴露面也越大。優先選少數幾個廣為人知的庫，而不是一長串名不見經傳的；並在加一個包之前先問問“我們真的需要一個包來幹這個嗎，還是說這就十行程式碼？”

定期執行你所在生態的審計工具（npm audit、pip-audit 之類）——它會標出你已經裝了的東西里那些已知的漏洞，而 AI 很擅長修復它報出來的問題。

安全審查關卡

下面這一個習慣，能把上面所有這些從一樁憂慮變成一道流程：在你上線之前，讓 **AI** 去攻擊它自己的程式碼。那個寫出這個功能的模型，通常自己就能找出其中的漏洞——只是你不問，它就不找。把它從建造者翻轉成對手：

你寫了這個介面。現在扮演一個試圖攻破它的攻擊者。

列出一個惡意使用者可以用來達成下列目的的每一種方式：

- 讀取或修改他們不該碰的資料（授權漏洞）
- 通過輸入注入程式碼（SQL injection、XSS、命令注入）
- 濫用缺失的校驗或速率限制

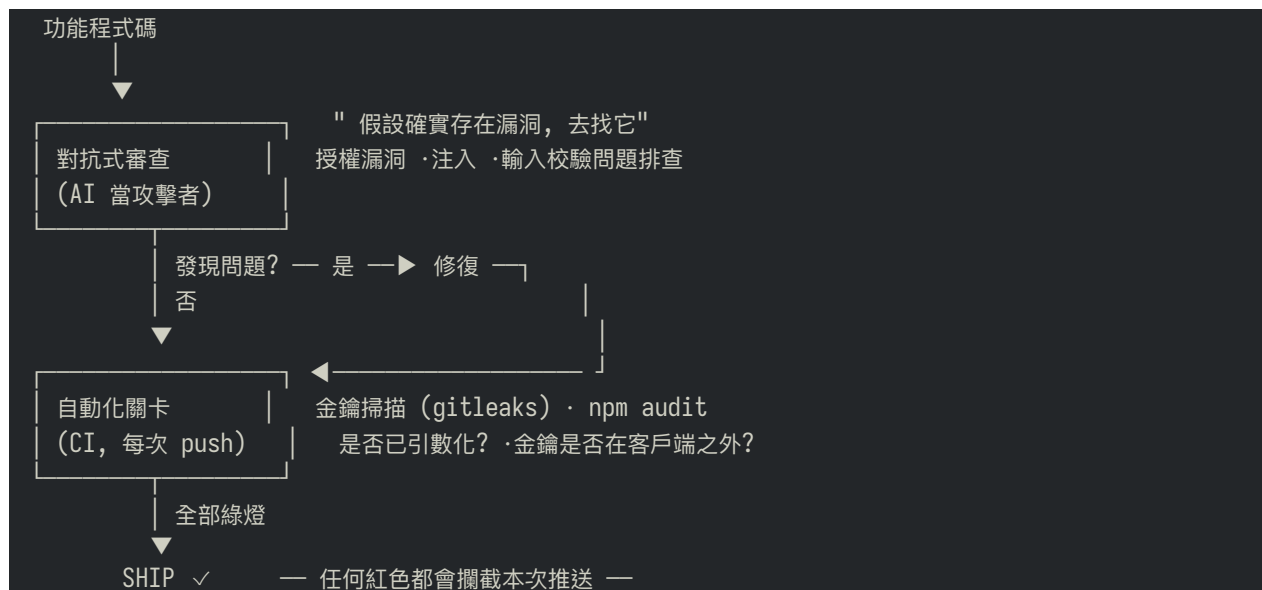
對每一種，給出能利用它的確切請求，然後給出修復辦法。

不要安慰我——假定這裡就是有一個漏洞，把它找出來。

最後那一行很關鍵：放任不管的話，AI 傾向於說“看起來很安全！”。被要求假定缺陷存在時，它才會真的去找。把這一次對抗性審查，和一份你對任何面向使用者的東西都跑一遍的簡短上線前清單配套使用：

- ☐ 每個介面都檢查了授權，而不只是使用者已經登入
- ☐ 所有資料庫查詢都是引數化的——沒有字串拼出來的 SQL
- ☐ 渲染到頁面上的使用者輸入都做了轉義（沒有原始 HTML 注入）
- ☐ 客戶端程式碼裡沒有金鑰，也沒有任何金鑰被提交進倉庫
- ☐ .env 已被 gitignore；任何洩露過的金鑰都已輪換
- ☐ 檔案上傳會校驗型別和大小、並使用生成的檔名
- ☐ 新依賴都被親眼核對過是否真實存在、口碑如何

把這道關卡想像成一個不通過就什麼都發不出去的流程。AI 從建造者切換成攻擊者的帽子，然後自動掃描兜住人工那一遍可能漏掉的東西：



並且在 push 之前跑一個金鑰掃描器——像 gitleaks 這樣的工具（或者你平臺內建的掃描）會在你的程式碼和歷史裡搜尋

形似金鑰的東西。它是一道一條命令就能就位的安全網，對應的正是這份清單上代價最昂貴的那個錯誤；你還可以讓 AI 把它接進 CI (Continuous Integration，持續整合——那個在你每次推送程式碼時自動替你跑各項檢查的機器人)，這樣每次 push 都會自動跑。

自動化自滿，安全篇

最危險的時刻不是第一天——而是第九十天。第一天你很謹慎；你讀每一份 diff，你跑那段攻擊者提示詞，你檢查上傳處理邏輯。然後一百次改動上線了，什麼壞事都沒發生，於是檢查開始讓人覺得像是走過場。你“就這一次”地在一個小改動上跳過了審查關卡。這個小改動加了一個沒有授權檢查的介面。有一陣子什麼都沒發生，因為從內部看，什麼都沒發生正是一個安全漏洞的樣子。

這就是自動化自滿對準了安全，而它在這裡比在任何別的地方都更糟，因為安全失敗是無聲的、延遲的。一個壞掉的功能會在幾分鐘內懲罰你。一個壞掉的許可權檢查則會用表面上的成功來獎勵你，一直獎勵到洩露發生的那一刻。災難的缺席並不是安全的證據——它正是一個有漏洞、只是還沒被人發現的應用的正常狀態。

防禦之道，是把這道關卡做得足夠便宜，讓你不去跳過它；又足夠自動，讓你沒法跳過它。把金鑰掃描和 audit 放進 CI，這樣它們不靠你記著也會執行；把那段攻擊者提示詞隨手放在手邊，用在任何接觸 auth、資料或上傳的東西上。守住一條線：一個處理使用者資料或許可權的改動，在有人——你，藉助 AI 誠實的幫助——主動試圖攻破它之前，不上線。vibe coding 裡的速度來自好的關卡，而不是來自相信“安靜”就意味著一切都好。

回顧與練習

關鍵要點

- AI 生成的程式碼預設就是不安全的；假設漏洞存在並主動去找，而不要相信乾淨的第一印象。
- 大類反覆出現：注入、外洩的金鑰、缺失的授權、不安全的檔案上傳，以及憑空發明的依賴。
- 授權是按請求而非按登入——每個介面都必須檢查這個使用者是否可以執行這個操作，而不只是檢查他是否已登入。
- 每次推送前都跑一次像 gitleaks 這樣的金鑰掃描器，並輪換任何曾經洩露過的金鑰；把掃描接入 CI，讓你無法跳過。
- 第九十天比第一天更危險——把安全關卡做得便宜到你不會跳過、自動到你無法跳過。

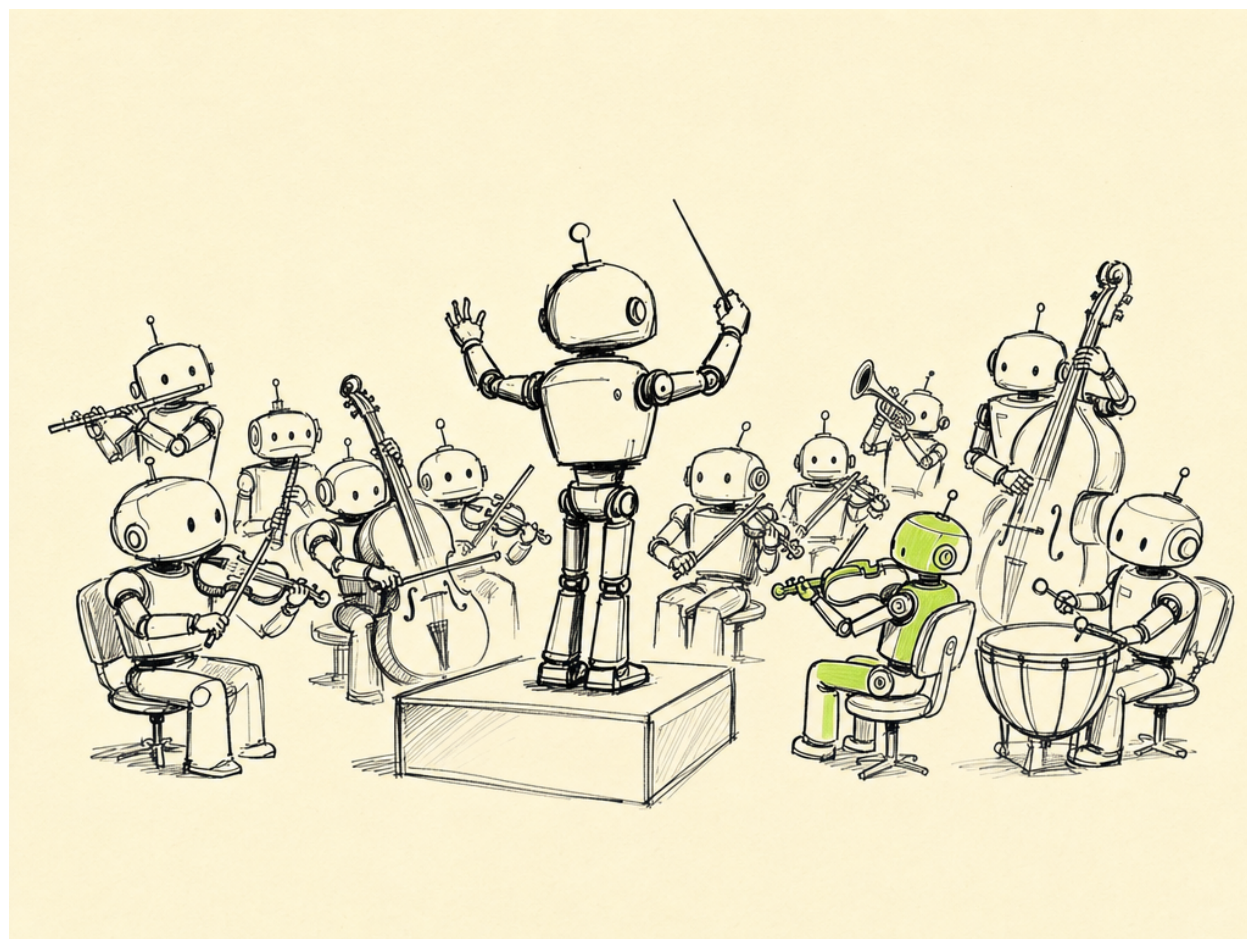
動手試試

挑一個你自己涉及認證、使用者資料或上傳的功能，對它執行下面這條對抗式審查提示詞。對 AI 報告的每個漏洞，索要利用它的確切請求和最小修復，應用修復後再次執行該提示詞。最後對倉庫和歷史記錄跑一遍 gitleaks(或你平臺內建的金鑰掃描) 收尾。

本章提示詞

```
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.
```

智慧體編排



本書的大部分內容裡，AI 一直是你握在手中的工具：你問，它答，你檢查結果。多數時候這仍是正確的心智模型。但工具長出了第二種模式——你交出的是一個目標而非一個步驟，AI 會工作幾分鐘（有時更久）才回到你這裡。

本章講的就是這種轉變，以及當它發生時如何不丟掉主線。人們容易把「智慧體」讀成「現在我可以不再留意了」。正是這種讀法會讓人陷入麻煩。智慧體是對你已經理解的工作的槓桿——而不是一張允許你釋出自己已不再過目的程式碼的許可證。

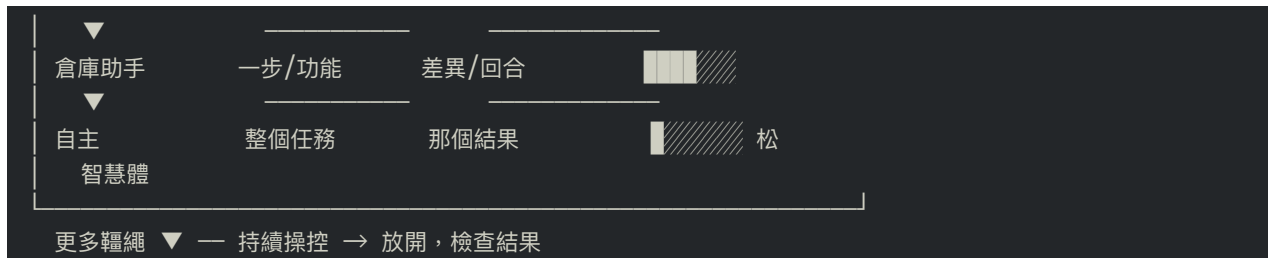
三個層級

把 AI 輔助看成三個層級會有幫助，每一層都多交給你一點韁繩。

- **編輯器內補全。**你打字時，AI 提示下一行或下一塊。你完全掌控；逐條接受或拒絕每個建議。工作單位是幾行，而且它們就出現在游標正下方，所以預設你會全部審查。
- **倉庫級助手。**你與一個能看到整個專案、一次編輯多個檔案、執行命令並讀取輸出的助手對話。本書的大部分工作都在這裡發生。工作單位是一個步驟或一個小功能，每一輪之後你審查 diff。
- **自主智慧體。**你給出一個目標——「讓測試套件通過」「把這個模組遷移到新 API (Application Programming Interface，應用程式程式設計介面，一個軟體對外暴露、供其他軟體使用的一組命令)」——然後智慧體會規劃、編輯、執行、讀取錯誤，並跨越許多步驟反覆嘗試。你審查的是結果，而非每一個單獨動作。工作單位是整個任務。

每一層都遞給你更多的繩子：更大的工作單元、更少的連續操縱，以及更靠後的審查時點：

層級	工作單元	你審查	控制
補全	幾行	每條建議	緊



要緊的躍遷是最後一個。前兩層裡你持續地操縱。在自主智慧體這裡，你放開它，然後檢查結果。那是一種不同的技能，而本章餘下的部分講的就是它。

何時放手執行，何時手動操縱

是否讓智慧體無人值守地執行，歸結為三個問題——把它們當成你離開之前要過一遍的清單。

- **爆炸半徑**。如果出錯，會壞掉多少？對整個程式碼庫做一遍格式化，爆炸半徑很小。資料庫遷移或改動你的認證邏輯，則很大。半徑越大，越要手動操縱。
- **可逆性**。能否廉價地撤銷？在工作樹幹淨的新建 `git` 分支上的工作是可逆的——`git restore`。它就沒了。刪除資料、傳送郵件、扣卡、推送到生產環境的工作則不是。絕不要讓智慧體無人值守地處理你拿不回來的東西。
- **可驗證性**。你能判斷結果是否正確嗎？如果「完成」意味著一套綠色的測試或一個可見地正常工作的頁面，你幾秒內就能驗證，智慧體非常合適。如果「完成」取決於微妙的判斷——金額計算、安全規則、競態條件——你無法靠一次快速檢查來設卡，那就貼近些。

把它們合起來的簡單辦法：**當工作可逆且結果廉價可驗證時放手執行；當不是時則手動操縱**。把構建功能那章的規則帶過來——爆炸半徑越大，步驟越小——再加上它的孿生規則：爆炸半徑越大，韁繩越短。

給任務劃定範圍，讓它無法亂跑

自主智慧體以一種特定的方式失敗：它漂移。你要一件事，它「熱心地」重構了另外三件，碰了你從未提及的檔案，帶著一份你讀不懂的 `diff` 回來。修法不是更聰明的智慧體——而是更緊的簡報。

一個好的自主任務有四個部分：清晰的**目標**、明確的**約束**、具體的**完成判據**，以及智慧體自己能執行的**護欄**。護欄是最重要的部分，因為它是智慧體無法在你不察覺的情況下翻越的柵欄。

Goal: Make the failing tests in `tests/checkout/` pass.

Constraints:

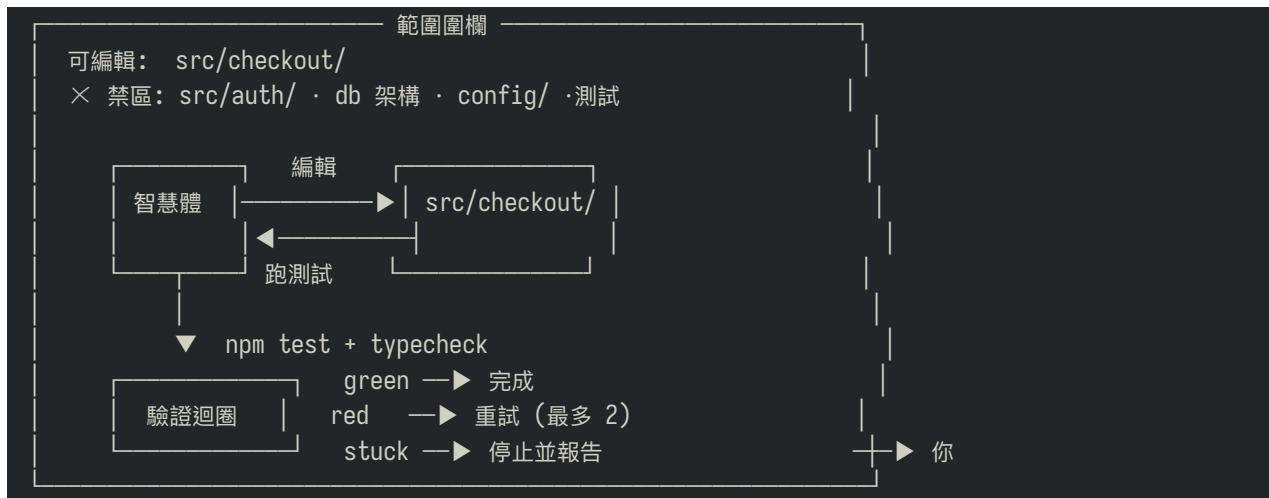
- Only edit files under `src/checkout/`. Do not touch `src/auth/`, the database schema, or anything in `config/`.
- Do not add new dependencies.
- Do not change the tests themselves — the tests define correct behavior.

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.

這份簡報之所以管用，是因為它把智慧體關進了盒子裡：約束給它能碰的東西築牆，完成判據是它能執行的一條命令，而驗證迴圈讓它保持誠實：



三樣東西讓這份簡報難以讓它跑偏。約束在智慧體可以觸碰的檔案周圍築起一堵牆。完成判據是任何人都能執行的命令，而不是一種感覺。而最後那段給了它停下來發問而非亂撲騰的許可——正是這一點防止了那種慢鏡頭的災難：智慧體「修」了二十分鐘，把程式碼樹留得比它接手時還糟。

真正的柵欄是測試。能執行自己測試的智慧體擁有一個緊湊而誠實的反饋迴路，多半會留在其中；無法檢查自己工作的智慧體會信心十足地把壞掉的東西遞給你。如果你從本章只帶走一個習慣：給智慧體一個驗證自己的途徑，並把通過它當作完成的定義。

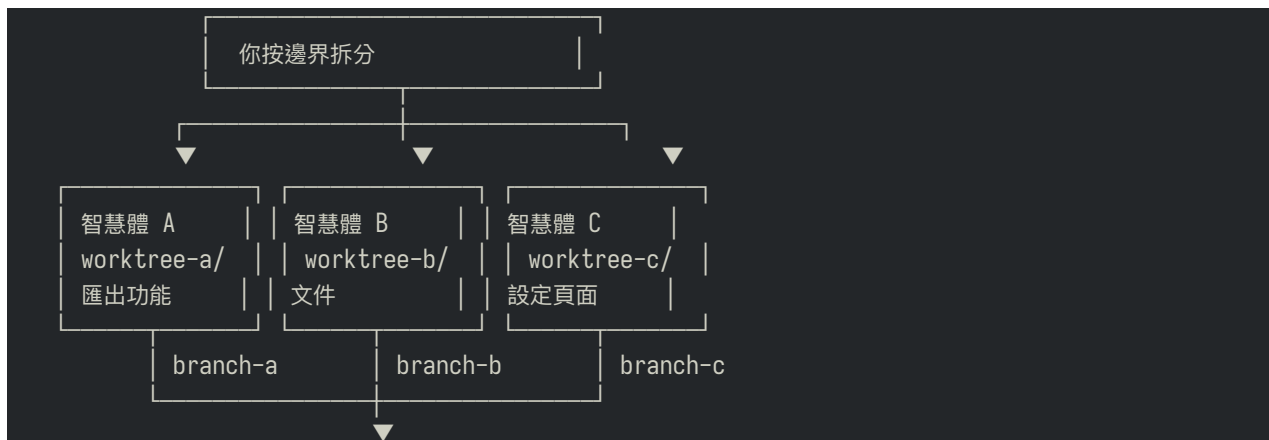
後臺與並行智慧體

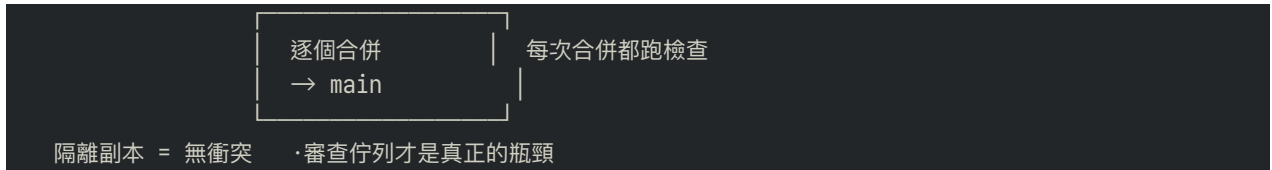
一旦你在劃定範圍的任務上信任某個智慧體，下一步就顯而易見：一次執行好幾個。現代工具讓你啟動後臺智慧體——各自在自己的分支或工作樹（worktree，專案的一份單獨的工作副本，這樣一個智慧體的改動永遠不會和另一個的相撞）上——這樣在你做別的事時，三四個劃定範圍的任務都在推進。

這是真正的提速，但僅限於獨立的工作。你拿來交換的代價是協調。兩個智慧體編輯同一個檔案會產生衝突的 diff，手動合併它們可能比你省下的還貴。

- **按邊界拆分，而非按行數。**給每個智慧體一塊與其他不重疊的部分——不同的模組、不同的功能、不同的檔案。「智慧體 A 做匯出功能，智慧體 B 寫文件」很乾淨。「兩個智慧體都編輯 app.js」是預約了一場合並頭疼。
- **隔離工作空間。**讓每個智慧體在自己的 git 分支或工作樹上執行，使它們踩不到彼此未提交的改動。一次合併一個回來，每次合併時都執行你的檢查。
- **誠實地控制數量。**你仍然必須審查每一個結果。四個智慧體在跑，就是四份要讀的 diff 和四套要信任的測試。超過一小撮之後，瓶頸就成了審查佇列，而不是智慧體。

每個智慧體都在自己隔離的 worktree 裡工作，於是它們的編輯永不碰撞；你一次合併回一個，並在每道關卡跑檢查：





錯誤在於把並行當成免費。它不是——你把工作從編寫挪到了審查與整合。這常常是筆很好的交易，但前提是你真的去審查。

多智慧體：扇出、對手、綜合

還有第二種「多個智慧體」的口味，它執行在一個任務之內，而非跨越彼此獨立的任務。不是讓一個智慧體做所有事，而是把角色拆開：

- **為求廣度的扇出**。多個智慧體從不同角度攻同一個問題——或探索一個大程式碼庫的不同部分——然後回報。當你想要覆蓋面時很有用：「找出我們呼叫這個已棄用函式的每一處」就很好並行。
- **對抗式審查者**。一個智慧體寫；第二個智慧體唯一的工作是攻擊第一個的輸出——找出 bug、漏掉的邊界情況、未處理的錯誤。一個被指向 diff、帶著「假設這是錯的，並查清為什麼」指令的審查者，會逮住作者智慧體把自己說服了的東西。
- **綜合**。最後一遍收集這些發現，產出一個連貫的結果——可執行的程式碼，外加一段簡短的說明：審查者標出了什麼，以及它是如何被解決的。

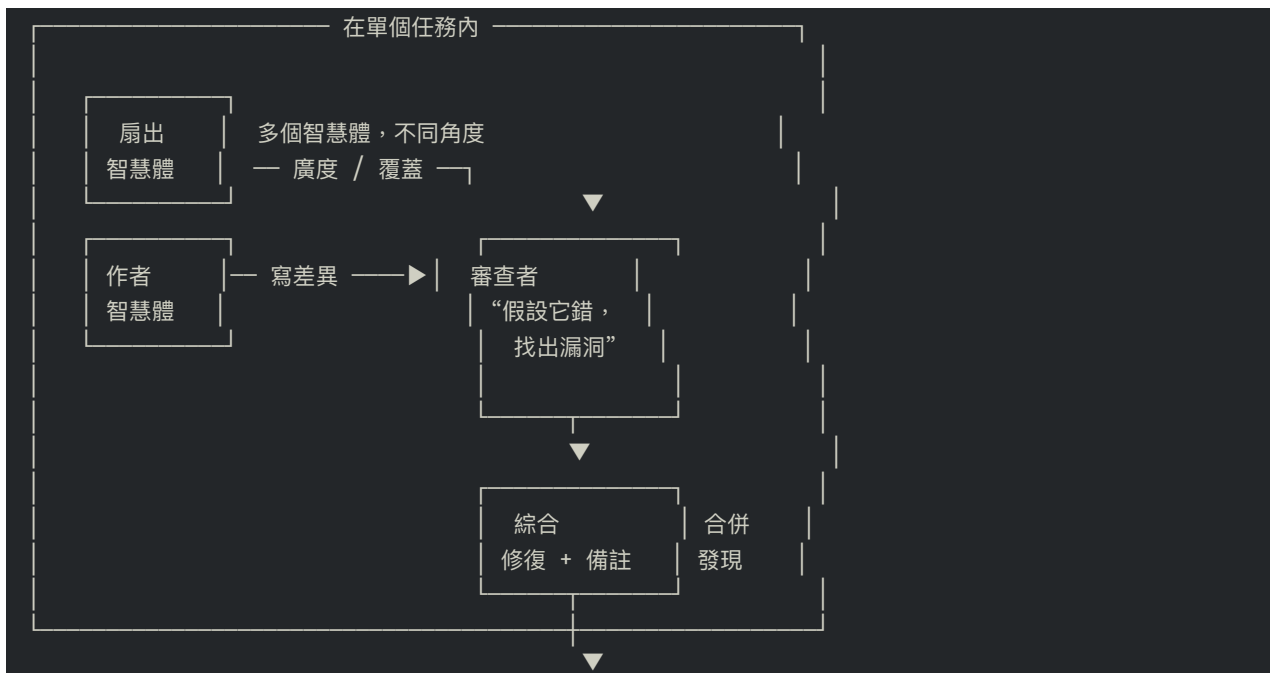
Reviewer agent brief:

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.

三個角色構成一條末端帶關卡的流水線：在對抗式審查者過完一遍、綜合環節解決掉它的發現之前，什麼都到不了提交：



對抗式審查者值得設定，即便你是沒有一支智慧體艦隊的單槍匹馬——因為一處改動的作者，無論人還是 AI，都是最不適找出其缺陷的那個。用一段敵意的提示再過一遍，是廉價的保險。

按規模審查輸出

這裡有個誠實的問題：當智慧體跨越多個任務寫出數百行時，你無法像審查十行 diff 那樣讀完每一行。假裝可以，會導致兩種結果之一：蓋橡皮圖章（不看就批准）或癱瘓（什麼都不信任）。兩者都發布不出東西。

走通的辦法是在幾個層面設卡，而不是把一切都讀完：

- **測試是底線**。相關測試不綠、型別檢查不乾淨，任何智慧體輸出都不得合併。這不可商量，而且是自動的——這部分由機器來做。
- **對高風險部分做抽查**。你讀不完全部，所以讀爆炸半徑高的部分：任何觸及認證、金額、資料刪除或外部呼叫的地方。其餘的，按構建功能那章的危險訊號略讀——它不該碰的檔案、被悄悄刪掉的東西、一個小請求生出一份巨大 diff。
- **讀 diff，而非整個程式碼庫**。git diff --stat 幾秒內就告訴你什麼變了、變了多少。一個劃定到一個模組的任務卻在六個模組裡顯示改動，這是一個訊號：在讀任何一行之前，先更仔細地看。
- **執行它**。一套綠色的測試不是一個能用的應用。點一下那東西。最後的檢查永遠與其他每一章相同：它在你面前，是否真的做了你要它做的事？

劃定範圍在這裡有雙重回報。一個你緊緊劃定範圍的任務會產出一份你真能審查的 diff；一個你放開的任務會產出一份你不能審查的。審查問題大體在上游、在簡報裡就被解決了。

成本與失控的迴圈

自主智慧體靠迴圈工作：嘗試、檢查、再嘗試。那個迴圈正是全部要點所在——它同時也是那個會悄悄把賬單拉高或永遠轉下去的東西。

這直接連到前面那個成本安全的觀念：定額訂閱的工具以停止來失敗；按量計費的付費 **API** 智慧體以收費來失敗。一個訂閱制的程式設計助手撞到上限時，它只是在視窗重置前不再多做。你針對一個付費 API 寫的指令碼卡在重試迴圈裡時，它會一直呼叫——每次呼叫都花錢——直到你察覺或你的卡刷爆。

所以規則很直白：

- **絕不要針對付費 API 執行一個不加節流的智慧體迴圈**。如果你寫了自己的、會呼叫某個 LLM（Large Language Model，大語言模型——支撐著這些程式設計工具背後的那類 AI）API 的迴圈，它必須有一個硬上限——最大迭代次數、花費上限、超時。沒有例外，沒有「我會盯著」。你不會盯。
- **給每個迴圈一個出口**。「N 次嘗試後停下」「如果測試再失敗兩次就停下」「如果已經跑了十分鐘就停下」。一個沒有停止條件的智慧體是 bug，不是功能。
- **盯住頭幾次執行**。在你信任一個智慧體在後臺執行之前，從頭到尾盯幾次前臺執行。你是在檢查它是否收斂——是越來越接近完成，而不是繞著同一個修法打轉卡住。
- **優先還有護欄的工具**。一個帶內建上限的託管助手，比你半夜寫的一個赤裸的 API 迴圈更適合放著跑。訂閱以關閉來失敗是個功能；API 以敞開來失敗是個隱患。

誠實的限度

智慧體是本書中最強大的槓桿形式，正因如此，對它是什麼、不是什麼保持清醒的眼光才有價值。

它不是一種你可以停止理解的自主性。當你再也答不出「這段程式碼應該做什麼，要是它錯了我又怎麼會知道」的那一刻，你就從使用槓桿越界到了指望運氣。本章的每一層——劃定範圍、護欄、測試、抽查、成本上限——都是為了在讓智慧體承擔遠多得多的打字工作的同時，把你留在那條線上理解的一側而存在。

技能的天花板與本書其餘部分相同：你精確說出你想要什麼以及認出你已得到它的能力。智慧體把這個能力乘大。它不取代它。被燙傷的不是那些用了智慧體的人——而是那些用智慧體來停止思考的人。別做那種人。把工作交出去；把判斷握在手裡。

回顧與練習

關鍵要點

- 智慧體放大的是你精確表達需求、並識別是否如願以償的能力—而不是取代它。
- 界定範圍有雙重回報：範圍收得緊的任務產出你真能審查的 diff，鬆散的任務產出你審不動的 diff。
- 審查 diff，而不是整個程式碼庫—`git diff --stat` 幾秒鐘就能標出範圍蔓延，而通過的測試套件永遠不能替代親自執行。
- 絕不要對付費 API 跑沒有限流的智慧體迴圈；給每個迴圈設硬上限—最大迭代次數、支出上限或超時。
- 在信任智慧體後臺執行之前，先在前臺盯著頭幾次執行，確認它會收斂。

動手試試

挑一個小而邊界清晰的任務，連同一個明確的停止條件（“嘗試 3 次後停止”或“測試再失敗兩次就停止”）一起交給智慧體。在前臺從頭到尾盯完整個執行，然後只用 `git diff --stat` 再加 `git diff` 審查這份 diff。如果改動觸及的檔案多於你界定的範圍，那就是收緊任務說明並重跑的訊號。

本章提示詞

```
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.
```