AI GOAL GRID COURSE — HANDOUT

AI 互動行銷頁實戰
目標九宮格 · 上課講義

填一張目標九宮格,AI 補完、生成專屬桌布、自然留下 email —
一頁做完整個行銷漏斗。含全部章節講義、prompt 範本、常見坑與驗收清單。

成品對照:https://yazelin.github.io/ai-goal-grid-course/app/

2026-06 · 「AI 互動行銷頁實作課」進階續作

目錄

AI 互動行銷頁實戰 — 目標九宮格 — 課程總覽

> 前課教零件,這門課把零件組成一個真的產品:一條會自己收名單的 AI 行銷漏斗。

這次我們要做什麼

一個叫「目標九宮格 Goal Grid」的完整產品:訪客在九宮格中央寫下核心目標,AI 補滿八個子目標、再展開成 64 個具體行動;接著挑一種風格,AI 生成桌布背景、前端把九宮格文字疊上去,變成一張專屬桌布直接下載;下載旁邊放一個誘因明確的 email 表單 —— 「留下 email,立即拿 30 天行動追蹤模板」,送出當場給連結。

成品已上線,先玩再上課:https://yazelin.github.io/ai-goal-grid-course/app/

這不只是一個工具頁,而是一條完整的 AI 行銷漏斗:

```
免費價值(AI 幫你填九宮格 + 專屬桌布)
    ↓ 不設牆,先給
帶走成品(PNG 直接下載)
    ↓ 旁邊放誘因
留下 email(換 30 天行動追蹤模板,當場兌現)
    ↓
你的名單(D1 資料庫 + 後台 + CSV 匯出)
```

順帶一提:這個方法本身的故事 —— 源自原田メソッド、大谷翔平高一填過的那張表 —— 以及為什麼產品「不能」叫大谷九宮格,是第 01 章的教材。那是一堂行銷人必修的命名安檢課。

跟前課的差別

	前課(AI 互動行銷頁實作課)	本課
教什麼	零件:landing、視覺、特效、遊戲、部署、Worker、名單	把零件組成一個真產品
demo 形式	每章獨立小品	漸進建造同一個 app,每章結束你手上的東西都更完整
後端	一個功能一個 Worker	一個 Worker 三個職責(AI 補格 / 生圖代理 / 名單)
終點	一頁會動的活動頁	一條會自己收名單的漏斗,整套可以換皮成你的產業

完課後你拿到什麼

1. 一個上線的「目標九宮格」:訪客玩得動、AI 叫得動、名單收得到,整套部署在你自己的帳號上、用你自己的額度。
2. 一套可換皮的漏斗模板:把「九宮格 + 桌布」換成你產業的免費價值(測驗、診斷、產生器),骨架完全相同。
3. 真產品等級的功課:長任務等待 UX、防刷額度、降級設計 —— 這些是「做一頁」跟「做一個產品」之間的差距。
4. 每章一份可重現的 AI prompt 範本:就算不抄本課的程式碼,也能叫 AI 重做一次。

章節地圖

章	主題	對應 demo
00	課程總覽(本章)+ 前課銜接(00b)	—
00c	金鑰申請指南:OpenAI / Groq key 申請、用 ChatGPT 介面手動生圖	—
01	產品定位與命名:IP/商標研究實錄	—
02	九宮格 UI 與資料模型(CSS Grid + 「null = 留給 AI」)	`demos/01-grid-ui/`
03	AI 補格:Worker + LLM JSON 輸出 + prompt 設計	`demos/02-ai-fill/`
04	圖片上傳與前端縮圖(FileReader / canvas / base64)	—
05	AI 桌布背景:長任務 job + 輪詢 + 等待 UX	`demos/03-wallpaper/`
06	canvas 文字合成(文字=資料、背景=藝術;改字成本對照)	`demos/03-wallpaper/`
07	AI 編輯迴圈:edit 端點 + 參考圖 + 可攜帶的 prompt	`demos/03-wallpaper/`
08	防刷與額度:Turnstile + 四層防線	`demos/04-quota/`
09	Email 收集與 admin 後台(D1 + CSV)	`demos/05-email/`
10	進階 9×9 + 發佈與 SEO/AEO	`app/`

demo 跟前課的「獨立小品」不同:`demos/01-grid-ui/` 是第 02 章結束時你手上的東西,`demos/02-ai-fill/` 在它上面加 AI,一路疊到 `app/` 的完整版。每個 demo 都是單檔、雙擊能開。

需要準備什麼

這門課假設你已經會前課的四件事:把資料夾部署成 GitHub Pages、知道 Cloudflare Worker 在幹嘛、會用【】分節的 prompt 範本跟 AI 要東西、看過 demo 09 的 email 收集模式。任何一項不確定,先花十分鐘看銜接章 `00b-recap.md`,每一項都附前課傳送門。

帳號(都免費):

1. **GitHub**:前課就有的,Pages 部署用。
2. **Cloudflare**:Worker、D1 資料庫、Turnstile 人機驗證都在這(第 03 章起會用到)。
3. **Groq**:免費 AI API key,AI 補格用(到第 03 章再申請即可)。
4. **AI 聊天工具**:ChatGPT / Claude / Gemini 任一,你的「工讀生」。

生圖(第 05 到 07 章)不用先準備:課程示範站有共用額度(防刷限量,每人每天生圖 3 次),也可以走「複製 prompt → 自己貼到 ChatGPT/Gemini 生圖 → 把圖匯回來」的零後端路線 —— 這條降級路線本身就是第 07 章的教材。等到你要自己部署整套時,生圖是三層架構:主路接 OpenAI Images API(需要一把按張計費的 OpenAI API key)、備援接自架生圖服務(沒設 OpenAI 金鑰才走)、最後是上面那條零後端降級。金鑰怎麼申請、費用怎麼估,見銜接章 00c(金鑰申請指南)。

課程的四個原則

前三個沿用前課,第四個是本課新增:

1. **零 build**:全部是純 HTML/CSS/JS,雙擊能開,不安裝任何開發工具。
2. **免費 tier**:Pages、Worker、D1、Turnstile、Groq 全部從免費額度起步。
3. **prompt 是核心技能**:每章附「給 AI 的 prompt 範本」,你帶走的不是程式碼,是跟 AI 要東西的方法。
4. **降級路線是設計的一部分**:AI 服務全掛的時候,訪客還是要能走完核心體驗(風格漸層照樣出桌布、生圖 prompt 可以外帶)。真產品跟 demo 的差別,就在這種地方。

前課銜接 — 四個會一直用到的零件

> 本課假設你上過「AI 互動行銷頁實作課」(https://yazelin.github.io/ai-marketing-pages-course/)。這頁用半頁複習四個會一直用到的核心,每段附前課傳送門;四段都覺得「嗯,我知道」,就直接開工。

1. GitHub Pages:把資料夾變成網址

repo 裡放 `index.html` → Settings → Pages → Branch 選 `main` 、 `/ (root)` → 一兩分鐘後拿到 `https://你的帳號.github.io/repo名/` 。改版 = 重新 commit,稍等就生效。本課的成品 app 與所有 demo 都是這樣上線的。

複習:前課模組 5「上線:部署決策樹 + GitHub Pages」 https://yazelin.github.io/ai-marketing-pages-course/course/05-deployment.md

2. Cloudflare Worker:幫頁面藏 key 的免費代理

API key 等於信用卡,放進網頁任何人按 F12 都看得到。解法是中間加一個 Worker:網頁呼叫 Worker、Worker 拿著藏在 Secret 裡的 key 呼叫 AI、把答案傳回來。

```
訪客瀏覽器 ──> Cloudflare Worker(key 藏在 Secret)──> AI API
 GitHub Pages          免費 10 萬次/天
```

前課一個 Worker 做一件事;本課升級成「一個 Worker 三個職責」: `/fill` AI 補格、 `/image` 生圖代理、 `/signup` 收名單,從第 03 章起逐章蓋出來。

複習:前課模組 6「頁面即時呼叫 AI」 https://yazelin.github.io/ai-marketing-pages-course/course/06-live-ai-on-page.md

3. AI prompt 範本的用法

本課每章都有「給 AI 的 prompt 範本」,用法跟前課完全一樣:用【】分節把 brief 寫清楚(要什麼、資料長怎樣、風格、驗收標準)→ 整段貼給 ChatGPT / Claude → 存檔開啟、逐項驗收 → 修改靠對話下指令,一次只改一兩件事。你是總監,AI 是工讀生,驗收清單就是你的武器。

複習:前課模組 1「第一個活動頁」與模組 8「Prompt 兵法」 https://yazelin.github.io/ai-marketing-pages-course/course/01-first-landing-page.md https://yazelin.github.io/ai-marketing-pages-course/course/08-prompt-playbook.md

4. demo 09 的 email 收集模式

前課最後做過一條名單收集線:報名頁 POST email → Worker 驗證(email 格式、honeypot 隱形欄位、每 IP 限流)→ 寫進 D1 資料庫(UNIQUE 去重)→ admin 後台用 Authorization 標頭帶 token 看名單、匯出 CSV。本課第 09 章直接沿用這個模式,再加上 Turnstile 與「留 email 當場拿模板」的誘因設計。

複習:前課模組 9「收 email 名單」 https://yazelin.github.io/ai-marketing-pages-course/course/09-email-list.md

30 秒自我檢查

- 我能在十分鐘內把一個資料夾變成公開網址(不行 → 模組 5)
- 我說得出「為什麼 API key 不能放進網頁」(說不出 → 模組 6)
- 我貼過至少一次【】分節的 prompt,並照清單驗收過成品(沒有 → 模組 1、8)
- 我知道 honeypot 跟 admin token 各在防什麼(不知道 → 模組 9)

四項都打勾,就從第 01 章開始 —— 那章不寫程式,講一個行銷人必修的真實命名故事。

金鑰申請與自備額度 — 兩把鑰匙,和一條不用鑰匙的路

> 本課的 AI 功能用到兩種金鑰:Groq(文字補格,免費)與 OpenAI(桌布背景,計量付費)。這一頁一次講清楚:去哪申請、申請完貼到哪裡、各要花多少錢——以及完全不想申請 API 的人,怎麼用手上現成的 ChatGPT 訂閱走完全程。不用一開始就全辦齊:第 03 章才用到 Groq、第 05 章才碰生圖,到時候再回來查也行。

1. Groq API key(文字補格用,免費)

它負責什麼:第 03 章的「AI 補滿空格」與第 10 章的 9×9 展開——Worker 的 `/fill` 端點拿這把 key 呼叫 Groq 上的 LLM。

申請步驟(不用綁卡):

1. 開 https://console.groq.com/ ,用 Google 或 GitHub 帳號註冊登入。
2. 左側選單點 **API Keys** → **Create API Key**,取個名字(例如 `goal-grid`)。
3. 建立後立刻複製——key 只在這一刻完整顯示一次,長相是 `gsk_` 開頭的一長串。

免費額度:Groq 免費方案不收錢、不用信用卡,限制的是「請求頻率」(每分鐘/每日的次數上限,依模型而異)。本課的用量——補格一次只是一個請求——個人練習與小流量示範站綽綽有餘;真撞到上限,等幾分鐘就恢復。

申請完貼到哪裡?看你的角色:

角色	貼到哪	效果
站長(自部署 Worker)	在 `worker-deploy/` 目錄執行 `npx wrangler secret put GROQ_API_KEY`,把 key 貼進去	key 藏在 Worker Secret,訪客用你的額度,key 永不露出
學員自己玩	app 頁尾「進階設定:自帶金鑰(BYO)」的 **Groq API Key** 欄位	只存你瀏覽器的 localStorage,補格改為前端直連 Groq、走你自己的額度

兩條路的差別第 03 章講過:BYO 直連只能自己玩(key 在前端,按 F12 看得到),要給別人用,key 必須進 Worker Secret——這是底線,不是建議。

2. OpenAI API key(AI 桌布背景用,計量付費)

它負責什麼:第 05-07 章的 AI 桌布背景。Worker 的 `/image` 端點主路走 OpenAI Images API(`gpt-image-1`),需要這把 key。

先講清楚一件最常被搞混的事:**API 跟 ChatGPT Plus 訂閱是兩回事**。Plus 的月費買的是「聊天介面的使用權」;API 是另一個錢包,按實際用量計費——就算你訂了 Plus,API 餘額仍然是零,要另外綁卡或儲值。反過來也成立:只用 API 不需要訂 Plus。

申請步驟:

1. 開 https://platform.openai.com/ 註冊(跟 chatgpt.com 是同一個帳號,但這裡是開發者後台)。
2. **Settings → Billing**:綁信用卡或先儲值一小筆(例如 5 美元)。沒做這步,key 建得出來也不能用。
3. **API keys → Create new secret key**,複製 `sk-` 開頭的 key——同樣只完整顯示一次。

要花多少錢:`gpt-image-1` 按生成量計費,一張 medium 品質的直式桌布(1024x1536)約 **0.06-0.07 美元**;本課 Worker 預設每 IP 每日 3 張,小流量示範站通常是每月幾美元的等級。品質檔位由 Worker 環境變數 `OPENAI_IMAGE_QUALITY` 控制(預設 medium,low 較省、high 較貴)。即便金額不大,仍強烈建議到 **Billing → Limits** 設用量上限(例如每月 10 美元),超過自動停——這是所有「綁了卡的服務」的基本安全帶。

貼到哪裡:只有一個地方——

```
cd worker-deploy
npx wrangler secret put OPENAI_API_KEY
```

注意:頁尾的 BYO 進階設定**沒有** OpenAI key 的欄位,這是刻意的——`sk-` key 連著你的信用卡,不該出現在任何網頁欄位裡。學員想用自己的 OpenAI 額度生圖,正路是照課程自部署一套 Worker(key 放進自己的 Secret),再到進階設定把「Worker 網址」指向它。

3. 不想申請 API?用 ChatGPT 介面手動生圖

完全不申請任何 key,也能拿到 AI 桌布——這就是第 07 章教的「prompt 是可攜帶的資產」降級路線,在 app 的步驟四就能走:

1. 展開「**服務忙線或想自己生?**」面板,點「**複製生圖 prompt**」——app 把組好的完整 prompt(風格詞、構圖約束、長寬比提示都在裡面)放進剪貼簿。
2. 貼到 https://chatgpt.com (或 Gemini)送出,等它生成。
3. 有上傳參考圖的話,記得自己把圖一併附給 ChatGPT——這條路不經過任何後端,參考圖本來就在你手上。
4. 生成後下載圖片,回到 app 點「**匯入背景圖**」放回來,後面的文字合成、下載 PNG 完全相同。

成本:用你現有的 ChatGPT 訂閱額度(免費帳號也能生圖,次數較少)。免 API key、免綁卡、站方後端全掛也走得通——對「只想拿到自己那張桌布」的使用者,這條常常反而是最快的路。

4. 三層架構一覽

Worker 的 `/image` 端點按這個順序決定走哪條路:有 `OPENAI_API_KEY` 就走 OpenAI;沒設但有 `CODEX_IMAGE_KEY` 就走自架的 codex-image-service;兩把都沒有,回 503 `not_configured`,前端把「複製 prompt 自己生」列為主要出路。

層	走哪裡	誰適合	成本	穩定度
主路	OpenAI Images API(gpt-image-1,同步 90 秒)	要正式給訪客用的站長	約 0.06-0.07 美元/張(medium 直式),計量付費	高:官方雲端服務
備援	自架 codex-image-service(https://github.com/yazelin/codex-image-service)	玩家級自架:家裡有機器、想燒 ChatGPT 訂閱額度而不是 API 帳單的人	沒有 API 帳單(走 ChatGPT 訂閱額度),但機器要自己養	中:homelab 等級,斷電斷網就停
降級	prompt 外帶(複製 → ChatGPT/Gemini 手動生 → 匯回)	任何人,包括一把 key 都沒有的訪客	用使用者自己現有的訂閱	不依賴本站任何後端,永遠走得通(代價是手動)

設計觀:三層不是疊床架屋,而是讓每一種「壞掉」都有出路——站長還沒填 key、自架服務睡著了、今天額度用完了,訪客都不會被一句「請稍後再試」送客。三層在程式裡的長相,分別在第 05 章(主路與備援的呼叫)、第 07 章(prompt 外帶)、第 08 章(額度與錯誤文案)。

常見坑

1. **key 貼錯地方**:`gsk_`(Groq)兩個地方都能貼——站長進 Worker Secret、自己玩貼頁尾 BYO;`sk-`(OpenAI)只能進 Worker Secret,前端進階設定刻意沒有它的欄位。分辨法只要記一句:**會連到信用卡的 key,永遠不進瀏覽器**。
2. **OpenAI 沒綁卡就呼叫**:免費註冊的帳號沒有 API 額度,key 建得出來,一呼叫就是 429(insufficient_quota)。先到 Billing 綁卡或儲值,再測一次。
3. **key 外洩了**:截圖貼社群、commit 進 git、貼進聊天記錄,都算外洩。處理只有一招:立刻回對應後台 revoke(刪除)那把 key、重發一把新的,Worker 端 `wrangler secret put` 重新貼一次。兩家的 key 都可以無痛重發,猶豫才是最貴的。
4. **`secret put` 吃進多餘字元**:複製貼上時 key 尾端多帶一個換行或空白,Worker 就拿著「錯的 key」去呼叫、得到 401。這是本專案部署時真實踩過的雷——貼完多看一眼,懷疑時直接重 put 一次最快。

第 01 章 — 產品定位與命名:IP 與商標研究實錄

學完能做什麼

- 在幫產品、活動或工具命名之前,自己跑一次「命名安檢」:通稱、商標、人名三關。
- 分得清「描述性引用」與「品牌使用」的界線 —— 知道哪些寫法安全、哪些會踩雷。
- 用本章的 prompt 範本,叫 AI 幫你的下一個產品做命名安檢初篩。

核心觀念:大家都這樣叫,不代表你能拿來用

本課的產品,做的是「大谷翔平高中時填的那張目標表」的數位版。動手寫第一行程式之前,先碰上一個行銷問題:這個東西到底叫什麼?網路上的叫法五花八門 —— 「大谷九宮格」「曼陀羅九宮格」「マンダラチャート」「目標達成表」…… 我們替每一個叫法做了身家調查,結論是:最順口、搜尋量最大的幾個名字,全部不能用。

這章把那次真實的研究原樣攤開。命名是行銷人最常踩雷、卻最少被教要安檢的環節:名字取錯,輕則上線後被迫改名、累積的 SEO 與口碑歸零,重則收到律師函。而安檢本身只要半天。

步驟:這次安檢怎麼跑

1. 列出所有叫法,先全部當嫌疑犯

別急著挑順口的。把候選名與網路通稱全部列成清單:大谷九宮格、曼陀羅九宮格、マンダラチャート、マンダラート、目標達成表、Open Window 64……心態是:每個名字背後,都可能站著一個經營它多年的權利人。

2. 追血統:這個方法是誰的?

查證後,那張表的正確淵源是:

- 大谷那張表,日本媒體通稱「**目標達成シート**」(目標達成表)。
- 它的方法血統來自原田隆史的「**原田メソッド**」,工具正式名稱叫 **Open Window 64**(OW64)—— 花卷東高校監督佐佐木洋把這套方法引入球隊,大谷在高一時填寫了那張著名的表。
- 更上游的「把想法放進 3×3 格子發散」這個格式,在日本另有兩條各自獨立經營的品牌:マンダラチャート與マンダラート(下一步查它們)。

先弄清血統,才知道該把功勞還給誰、該避開誰、頁尾聲明該寫什麼。

3. 查商標:通稱可能是別人的註冊商標

三個查證結果:

名稱	權利人	對我們的意義
マンダラチャート®	松村寧雄(Clover 經営研究所)2006 年註冊,2022 年起由一般社団法人マンダラチャート協会管理	個人、學校、非營利內部使用免費;**營利使用需協會授權**。本產品是課程商品的一部分,屬營利情境 —— 不能用
マンダラート	今泉浩晃(ヒロ・アートディレクションズ)長期經營的品牌	他人品牌,不進產品名
Open Window 64	原田隆史(原田教育研究所)的招牌工具名,商業色彩極強	他人品牌,不進產品名

最容易踩的雷在第一列:「マンダラチャート」(以及它的中文轉寫「曼陀羅九宮格」)聽起來像通稱,其實是有人註冊、有協會管理的商標。「大家都這樣叫」不是授權。

4. 查人名:比商標更危險的是肖像權

「大谷九宮格」直覺最強、搜尋量最大 —— 也最不能用。把真實人名放進產品名,踩的是肖像權與姓名權,而 MLB 球員的品牌保護出了名地積極。

但書店裡到處是《大谷翔平也在用的……》這類書名,為什麼可以?因為那是**描述性使用**:內容在陳述一個事實(他用過這個方法),而不是把人名當成自己產品的品牌。界線整理成一張表:

寫法	性質	安全嗎
產品名叫「大谷九宮格」	把人名當品牌使用	不行
產品名叫「マンダラチャート產生器」	把他人註冊商標當品牌使用	不行(營利情境)
內文介紹「大谷翔平高一時填過的目標達成表」並附出處	描述性引用	可以
頁尾標註方法源自原田メソッド、商標歸屬協會	澄清聲明	可以,而且應該

5. 定名,並把該說的話寫進頁尾

安檢跑完,定案:中文「**目標九宮格**」、英文「**Goal Grid**」—— 描述產品做什麼(用九宮格管理目標),不借任何人的品牌與名字。然後在頁尾放三行聲明(成品 app 的頁尾原文):

> 本工具使用的九宮格目標法,源自原田隆史的「原田メソッド」(Open Window 64);大谷翔平高中時期填寫的目標達成表,是這個方法最著名的案例。
>
> 「マンダラチャート」は一般社団法人マンダラチャート協会の登録商標です。
>
> 本站為課程「AI 互動行銷頁實戰 — 目標九宮格」的成品範例,與上述協會及相關團體均無關聯。

三行各司其職:把功勞還給方法的源頭(這也是內容信任感的來源)、澄清商標歸屬、劃清與權利人的關係。注意:聲明是配套,不是免死金牌 —— 核心始終是「不把別人的商標和人名當自己的品牌用」。

給 AI 的 prompt 範本:幫你的產品做命名安檢

```
你是品牌命名顧問,幫我為一個產品做「命名安檢」。請保守判斷:拿不準的一律標成風險。

【產品】(一句話描述產品 + 使用情境;務必寫明是否營利,例如:放進付費課程、帶廣告、導購)
【候選名】1. ○○○ 2. ○○○ 3. ○○○
【方法或概念來源】這個產品用到的方法/格式/玩法來自:○○○
(請幫我追它的發明人、正式名稱、相關品牌與商標,包括外語原文)
【要查的面向】
1. 每個候選名是否撞到既有商標或知名產品(中文、日文、英文寫法都查)
2. 名稱是否含真實人名,或會讓人聯想到特定人物(肖像權/姓名權風險)
3. 這個方法「大家慣用的叫法」是否其實是某人的註冊商標
4. 若有風險:屬於「品牌使用」(不能用),還是改成「描述性引用」就能解決
【輸出】
- 每個候選名:風險等級(高/中/低)+ 原因 + 建議
- 給 3 個無風險的替代命名方向(描述功能、不借他人品牌)
- 一段可放頁尾的「方法淵源與商標歸屬」聲明草稿
```

三件事要記住:AI 的回答是**線索清單,不是法律意見**;商標最終以官方資料庫為準(台灣查經濟部智慧財產局的商標檢索系統、日本查 J-PlatPat、美國查 USPTO);AI 可能漏查也可能講錯,重大命名換另一個 AI 再問一次交叉比對,真要商用又拿不準,問律師。

常見坑

1. **把網路通稱當安全名**:愈順口、愈多人這樣叫的名字,愈可能是某人經營多年的品牌(本章的「マンダラチャート」就是)。
2. **用「我是佛系免費」自我安慰**:授權條件看的是你的情境。工具本身免費,但放在課程裡、帶廣告、導流購買 —— 都算營利使用。
3. **只查中文**:方法來自日本就查日文原文,要面向國際就查英文。商標是分地區、分語言註冊的。
4. **低估人名風險**:商標還有資料庫可查,肖像權、姓名權沒有資料庫 —— 真實人名一律不進產品名;內容引用時附出處、講事實、不暗示代言。
5. **AI 取的名字不安檢**:AI 很會生名字,也很會「生出既有品牌」。AI 給的候選名,照樣全部過一次安檢。
6. **以為寫了聲明就沒事**:聲明處理的是描述性引用的禮貌與清晰;如果名字本身侵權,聲明救不了。

對照成品

本章沒有程式碼,成品就是上線頁面上的兩個寫法:

- 打開成品 app(https://yazelin.github.io/ai-goal-grid-course/app/),拉到頁尾看三行聲明 —— 淵源、商標歸屬、無關聯,各司其職。

- 同一頁步驟一的「著名案例」區塊:引用大谷的表時,註明內容整理自日本媒體報導、格位為示意排列並非原表版面 —— 這就是描述性引用的標準姿勢。

第 02 章 — 九宮格 UI 與資料模型

學完能做什麼

- 做出一個 3×3 九宮格填寫頁:中央是核心目標、周圍八格是子目標,每格都能打字。
- 設計一個讓 AI 之後接得上手的資料模型:「null = 留給 AI;有字 = 使用者的,不准動」。
- 讓填寫內容即時存進 localStorage:關頁、重新整理都不會丟。

核心觀念:畫面是畫面,資料是資料

這章開始動手蓋產品,但第一課不是「怎麼畫九宮格」,而是「資料跟畫面分開」。畫面是九個 textarea;資料是一個長度 9 的陣列,活在 JS 變數裡、備份在 localStorage。畫面隨時可以砍掉重畫,資料才是本體 —— 之後每一章(AI 補格、展開 81 格、桌布合成)都是對這個陣列做事,不是對畫面做事。

資料模型裡還藏著整個產品最重要的一條約定:每一格的值要嘛是 `null`,要嘛是 `{text, source}`。`null` 不是「沒資料」,是一個明確的指令 ——「這格留給 AI」。`source` 記錄這格是 `'user'` 還是 `'ai'` 填的;下一章 AI 補格的鐵律「使用者寫的一字不改」,就是靠這個欄位才執行得了。用行銷的話說:「尊重使用者輸入」這個產品承諾,直接寫進了資料結構。

步驟

1. 用 CSS Grid 排出 3×3

九宮格不需要任何畫布或框架,CSS Grid 一行就排好:

```
.grid3 { display: grid; grid-template-columns: repeat(3, 1fr); gap: 10px; }
.cell {
  border: 1px solid #2c3a52;
  border-radius: 12px;
  background: rgba(8, 12, 20, 0.62);
}
.cell-core { border-color: #c9a44e; }   /* 中央格用金色強調 */
.cell textarea {
  display: block; width: 100%; min-height: 96px;
  background: transparent; border: 0; outline: none; resize: none;
  color: #e8ecf4; font-size: 14px; line-height: 1.55; text-align: center;
  padding: 1.45rem 0.5rem 0.6rem;
}
```

`repeat(3, 1fr)` 的意思是「三欄、每欄等寬」。手機上不用寫任何 media query,3×3 天生就是窄螢幕友善的版型。HTML 端只需要一個空容器:

```
<div class="grid3" id="grid3"></div>
```

2. 定義資料模型:null = 留給 AI

```
const STATE_KEY = 'goal-grid-state-v1';

function defaultState() {
  return {
    version: 1,
    // 長度 9 的陣列,index 4 = 中央核心目標
    // 每格:null(留給 AI)或 {text, source:'user'|'ai'}
    cells: Array(9).fill(null),
    updatedAt: new Date().toISOString(),
  };
}
```

兩個設計眼:`version: 1` 讓你之後改格式時,認得出舊資料、不會被它弄壞;`index 4 = 核心目標` 是因為九格由左到右、由上到下編號 0 到 8,正中央剛好是 4。

3. 讀寫 localStorage

```
function loadState() {
  try {
    const raw = JSON.parse(localStorage.getItem(STATE_KEY));
    if (raw && raw.version === 1 && Array.isArray(raw.cells)) return raw;
  } catch { /* 資料壞了就當沒存過 */ }
  return defaultState();
}

function saveState(state) {
  localStorage.setItem(STATE_KEY, JSON.stringify(state));
}

function setCell(state, idx, text, source = 'user') {
  const t = String(text ?? '').trim();
  state.cells[idx] = t ? { text: t, source } : null;   // 清空 = 還給 AI
  state.updatedAt = new Date().toISOString();
  saveState(state);
}
```

`setCell` 是唯一的寫入口:trim 之後是空字串,就把該格寫回 `null` ——「把字刪光」等於「這格還給 AI」。所有寫入都走同一個函式,存檔就不會漏。

4. 畫出九宮格,用 dataset.idx 對位

```
const state = loadState();
const grid = document.getElementById('grid3');

for (let i = 0; i < 9; i++) {
  const cell = document.createElement('div');
  cell.className = i === 4 ? 'cell cell-core' : 'cell';

  const ta = document.createElement('textarea');
  ta.dataset.idx = String(i);                 // 畫面第 i 格 ⟷ state.cells[i]
  ta.placeholder = i === 4 ? '我的核心目標' : '留給 AI';
  ta.value = state.cells[i] ? state.cells[i].text : '';
  ta.addEventListener('input', () => {
    setCell(state, Number(ta.dataset.idx), ta.value);   // 即打即存
  });

  cell.appendChild(ta);
  grid.appendChild(cell);
}
```

`dataset.idx` 是這章的關鍵小技巧:每個 textarea 自己「記得」自己是第幾格,事件發生時用它寫回陣列的正確位置。不要用 DOM 順序去猜對應關係 —— 之後版面一改(插了標籤、換了排序),用順序猜的程式碼就全錯位,而 `dataset.idx` 不動如山。

注意 placeholder 的文案:「留給 AI」。空格在這個產品裡不是未完成,是一種合法狀態 —— UI 文案跟資料模型講同一種語言。

5. 驗收

- 填三格(含中央),重新整理 → 內容都在。
- 把某一格的字全部刪掉,重新整理 → 該格回到 placeholder「留給 AI」。
- F12 → Application → Local Storage → 找到 `goal-grid-state-v1`,值是一串 JSON,手填的格子 `source` 是 `"user"`。
- 視窗縮到手機寬度,3×3 沒有破版、沒有橫向捲軸。

給 AI 的 prompt 範本

```
做一個單檔 HTML 的「目標九宮格」填寫頁(所有 CSS/JS 內嵌在同一個檔案):

【版面】3×3 九宮格,用 CSS Grid(repeat(3,1fr)、gap 10px);每格一個 textarea;
中央格是「核心目標」,邊框用金色強調;其餘八格是子目標
【資料模型】一個長度 9 的陣列 cells,index 4 = 核心目標;每格的值:
null 代表「留給 AI」,有字則存 {text, source:'user'}(之後 AI 填的會用 source:'ai',
所以結構要先留好)
【儲存】每次輸入即存 localStorage(key 'goal-grid-state-v1',JSON 格式、含 version:1);
頁面載入時讀回;JSON 解析失敗或版本不對,就用全新空白狀態,不准報錯
【對位】每個 textarea 用 dataset.idx 記住自己是第幾格,事件裡用它寫回陣列;
清空格子時要把該格寫回 null(trim 後為空就算清空),不是存空字串
【風格】深夜藍底 #0e1420、金色 #c9a44e 點綴,繁體中文,手機優先;
空格的 placeholder 文字是「留給 AI」,中央格是「我的核心目標」
【驗收】1. 填三格 → 重新整理 → 內容都在
2. 把一格字刪光 → 重新整理 → 該格回到 placeholder
3. 開發者工具看 localStorage,手填格子的 source 是 "user"
4. 視窗縮到 375px 寬不破版
```

常見坑

1. **dataset 永遠是字串**:`ta.dataset.idx` 拿到的是 `"4"` 不是 `4`。當索引用要先 `Number()`,直接拿去 `=== 4` 比較會永遠 false。
2. **把字串直接存進陣列**:`cells[i] = "減重五公斤"` 看起來能動,但丟掉了 `source` —— 下一章 AI 補格就分不出哪些是使用者手填的,「一字不改」的承諾沒辦法落實。包成 `{text, source}` 是在替未來的功能留接口。
3. **JSON.parse 沒包 try/catch**:localStorage 裡可能躺著舊版格式或壞掉的字串,一 parse 就丟例外、整頁 JS 停擺。讀回來的資料永遠當「來路不明」處理(完整版 `app/js/state.js` 的 `normalizeState` 甚至逐格重新驗證)。
4. **清空格子沒有歸 null**:沒做 trim、直接存 `{text:""}` 或一串空白,AI 會以為這格已被使用者佔用而跳過它。`setCell` 裡「trim 後為空就寫回 null」那行,就是在防這個。
5. **用 innerHTML 塞使用者輸入**:重繪畫面時用 `innerHTML` 把使用者文字拼進 HTML,等於讓任何輸入(包括之後 AI 的輸出)注入任意標籤。一律用 `textarea.value` 或 `textContent`。
6. **忘了中央格的特殊性**:index 4 是核心目標,跑「八個子目標」的迴圈時要跳過它。完整版 `app/js/grid.js` 用 `subIndexOf(pos)` 做這個換算(0–3 不變、5–8 減一),之後的配色與 81 格展開都靠它。

對照成品

- `demos/01-grid-ui/index.html` —— 本章成品,單檔、雙擊能開;照步驟做完(或叫 AI 重現)後,你手上應該就是這個東西。

- 完整版的同一套邏輯在 `app/js/state.js`(資料模型 + localStorage,外加 IndexedDB helpers)與 `app/js/grid.js`(3×3 之外還有 9×9 檢視、AI 徽章格)—— 讀起來會發現骨架跟本章一模一樣,只是格子變多了。
- 線上版: https://yazelin.github.io/ai-goal-grid-course/app/ 的步驟二,就是這章的 UI 加上完整視覺。

第 3 章 — AI 補格:Worker + LLM JSON 輸出 + prompt 設計

學完能做什麼

- 在九宮格頁面加一顆「AI 補滿空格」按鈕:留空的格子交給 AI,自己填的格子一字不動。
- 看懂並改寫一份「會回 JSON 的 prompt」——這是所有「AI 結果要進程式」的場景的共通技能。
- 知道為什麼 AI 失敗時要「退還額度」,以及這件事在程式裡長什麼樣子。

核心觀念:AI 補格不是 AI 代寫

第 2 章的資料模型已經埋好伏筆:每個格子要嘛是 `null`(留給 AI),要嘛是 `{text, source}`(使用者寫的)。這一章的所有設計都從這個約定長出來——**`null` 是授權,有字是主權**。AI 只能填 `null` 的格子;使用者寫過的字,連 AI 想「順手潤飾」都不行。這不只是禮貌,是產品立場:目標是使用者的,AI 是教練不是代筆。

第二個觀念是「跟 AI 約格式」。前課模組 6 的聊天小幫手,AI 愛怎麼回都行,反正人在讀;這一章不一樣——AI 的回答要直接塞回九個格子,**程式在讀**。所以我們要求 AI 只准輸出 JSON,而且程式收到後還要逐項驗收:長度對不對、有沒有偷改使用者的字。對 AI 的態度跟對廠商一樣:合約寫清楚,驗收不能省。

步驟

1. 複習:key 為什麼還是不能放前端

跟前課模組 6 完全同一個理由:網頁程式碼任何人按 F12 都看得到,Groq API key 放前端等於把信用卡貼在店門口。所以中間加一個 Cloudflare Worker,key 藏在 Worker 的 Secret 裡:

```
訪客瀏覽器 ──POST /fill──> Cloudflare Worker(GROQ_API_KEY 藏在這)──> Groq API
  GitHub Pages                順便做:每日額度、失敗退款、結果驗收
```

差別在於:這次 Worker 不只是「代打電話」,它還負責驗證輸入、扣額度、驗收 AI 的輸出。一個 Worker、多個職責,後面幾章會繼續往裡面加東西。

2. 跟 AI 約好只回 JSON

呼叫 Groq 時帶 `response_format: { type: "json_object" }`,模型就會被強制只輸出 JSON(這是 OpenAI 相容 API 的標準參數)。Worker 端的呼叫長這樣(`worker-deploy/src/lib/groq.js`):

```
const res = await fetch("https://api.groq.com/openai/v1/chat/completions", {
  method: "POST",
  headers: {
    Authorization: `Bearer ${env.GROQ_API_KEY}`,
    "Content-Type": "application/json",
  },
  body: JSON.stringify({
    model: "openai/gpt-oss-120b",
    messages: [{ role: "user", content: prompt }],
    response_format: { type: "json_object" }, // 跟 AI 約好:只准回 JSON
    max_tokens: 600,
    temperature: 0.8,
  }),
});
```

即使如此,程式仍要「防禦性解析」——模型偶爾會把 JSON 包在 ```` ```json ```` 程式碼框裡。先直接 `JSON.parse`,失敗再去框裡撈一次:

```
export function parseJSONLoose(text) {
  const s = String(text ?? "");
  try { return JSON.parse(s); } catch { /* 繼續 */ }
  const m = s.match(/```json\s*([\s\S]*?)```/i) || s.match(/```\s*([\s\S]*?)```/);
  if (m) {
    try { return JSON.parse(m[1]); } catch { /* 落空 */ }
  }
  return null;
}
```

3. 補格 prompt 逐段拆解

成品的 prompt 在 `worker-deploy/src/lib/prompts.js`,`corePrompt(cells)` 接一個長度 9 的陣列(空格是 `null`),組出這樣一段話:

```
export function corePrompt(cells) {
  const goal = cells[4] ? `核心目標:「${cells[4]}」` : "核心目標也留空,請先推斷一個具體可衡量的核心目標";
  const filled = cells.map((c, i) => c && i !== 4 ? `位置${i}:「${c}」` : null).filter(Boolean).join("\n") || "(其餘全空)";
  return `你是目標設定教練,使用「九宮格目標法」(源自原田メソッド,即大谷翔平用過的目標達成表的方法)。
使用者的 3×3 九宮格:中央(位置4)是核心目標,周圍 8 格(位置0-3、5-8)是支撐核心目標的子目標。
${goal}
已填的格子:
${filled}
請補滿所有空格。規則:
1. 子目標要具體、彼此不重複、共同支撐核心目標;盡量涵蓋「心・技・體・生活」四個面向(原田メソッド慣例)。
2. 每格 2-8 個字,繁體中文(台灣用語),不用標點結尾,不用 emoji。
3. 已填的格子原樣保留,一字不改。
只輸出 JSON:{"cells":["位置0文字","位置1文字",...,"位置8文字"]},長度必為 9。`;
}
```

一段一段看它在做什麼:

1. **角色與方法論**(第一行):給 AI 一個立場——「目標設定教練」,並把方法淵源(原田メソッド、大谷案例)寫進去。這不是裝飾:模型知道這套方法的結構,補出來的子目標才會「支撐核心目標」而不是八個不相干的願望。
2. **格局說明**(第二行):模型「看」不到九宮格,它只看得到文字,所以位置對應要講到死——中央是位置 4、周圍是位置 0-3 和 5-8。所有「AI 結果要對回版面」的場景都要做這件事。
3. **動態組裝**(`${goal}` 和 `${filled}`):把使用者已填的內容連同位置編號餵回去。注意第一行的三元判斷——連核心目標都留空時,不是報錯,而是請 AI「先推斷一個具體可衡量的核心目標」。輸入不完整時優雅補位,而不是把問題丟回給使用者。
4. **規則 1 是品質規則**:具體、不重複、共同支撐,以及「心・技・體・生活」四面向——這是原田メソッド的慣例,寫進去之後 AI 補的格子明顯比較不會八格全是「努力練習」這種同義反覆。
5. **規則 2 是版面規則**:每格 2-8 個字、不用標點結尾。這些字最後要塞進桌布上的小格子,字數是版面問題,不是文采問題——在 prompt 層先控制,比事後裁切體面。
6. **規則 3 加最後一行是合約**:已填的一字不改、只輸出 JSON、長度必為 9。下一步你會看到,程式端對這三條每一條都有對應的驗收。

進階模式的 `expandPrompt`(把一個子目標展開成 8 個具體行動)套路相同,差異有兩個值得學:行動的品質標準改成「可每日/每週檢核」,而且直接舉大谷的實例——「大谷在『運』底下寫『撿垃圾』『打招呼』這種日常小事」。**給 AI 一個具體範例,比三行形容詞都有效**。

4. Worker 端:先扣後跑、失敗退款、驗收輸出

`worker-deploy/src/index.js` 的 `handleFill` 把整個流程串起來(節錄重點):

```
// 先扣後跑:額度先 -1 再呼叫 AI(防止同時送兩發繞過上限)
const quota = await takeQuota(env.DB, { kind: "fill", ip, limit: 12, day });
if (!quota.ok) {
  return json({ error: "quota_exceeded", resetAt: taipeiResetAt() }, 429, cors);
}

let out = null;
try { out = await chatJSON(env, prompt, 600); } catch { out = null; }

// 驗收:必須是長度 9 的字串陣列,否則退款 + 回報 llm_failed
const arr = Array.isArray(out?.cells) ? out.cells : null;
if (!arr || arr.length !== 9 || arr.some((x) => typeof x !== "string" || !x.trim())) {
  await refundQuota(env.DB, { kind: "fill", ip, day }); // 確定失敗 → 把額度還給使用者
  return json({ error: "llm_failed" }, 502, cors);
}
const cells = arr.map((x) => x.trim());
normCells.forEach((c, i) => { if (c) cells[i] = c; }); // 已填格被 AI 改了?用原值蓋回去
return json({ cells }, 200, cors);
```

三件事值得停下來看:

- **先扣後跑**:額度先扣再呼叫 AI。如果反過來「先跑再扣」,有人同時送十發請求,每一發檢查額度時都還沒被扣,十發全過——上限形同虛設。
- **失敗退款**:AI 掛了、回了垃圾,都不是使用者的錯,額度退還(`refundQuota`),前端文案才能理直氣壯寫「失敗不會重複扣額度」。先扣是防濫用,退款是對人公平,兩件事不衝突。
- **雙保險**:prompt 規則 3 已經叫 AI「已填的一字不改」,程式還是再蓋一次原值。prompt 是請求,程式是保證——使用者的字不能靠 AI 的自覺來保護。

5. 模型選型:一個真實的雷

這個專案用 `openai/gpt-oss-120b` 而不是前課模組 6 的 `llama-3.3-70b-versatile`,是踩過雷的選擇:聊天場景 llama-3.3-70b 在 Groq 上好好的,但一旦要求**結構化輸出**(tool-call、嚴格 JSON),它在 Groq 上常常直接被退單或格式跑掉;換 `openai/gpt-oss-120b` 就穩了。

通則:「會聊天」和「會交格式正確的資料」是兩種能力,選模型要照你的場景測,不要照排行榜。換模型的成本很低——上面程式碼裡的一個字串而已,值得多試幾顆。

6. 接上前端

前端呼叫只是一個普通的 `fetch`(`app/js/api.js` 的 `fillCore`),拿回 `cells` 之後,只把原本是 `null` 的格子寫入 state、標記 `source: 'ai'`,UI 上給 AI 填的格子一個淡金邊框和「AI」徽章——讓使用者一眼分清哪些字是自己的、哪些是 AI 的,並且每一格都可以改寫或重抽。

給 AI 的 prompt 範本

下面這段貼給 ChatGPT/Claude,可以重現本章的單檔練習版(自帶 Groq key 直連,適合自己玩;正式上線請照本章 Worker 架構把 key 藏起來):

```
請幫我做一個單檔網頁(所有 CSS/JS 內嵌在同一個 HTML),功能是「AI 目標九宮格」:

【版面】3×3 九宮格,每格一個 textarea;中央格是核心目標(外觀強調);
空格的 placeholder 寫「留給 AI」。下方一顆「AI 補滿空格」按鈕和一個訊息區。
深夜藍底(#0e1420)配金色強調(#c9a44e),繁體中文,不用 emoji。

【資料模型】用一個長度 9 的陣列存格子,index 4 = 核心目標;
空格存 null,有字的格子存 {text, source:'user'|'ai'}。
每次輸入即存 localStorage,重新整理不丟。

【AI 呼叫】頁面最下方有一個輸入框讓我貼自己的 Groq API key(只存 localStorage)。
按「AI 補滿空格」時 POST https://api.groq.com/openai/v1/chat/completions ,
model 用 "openai/gpt-oss-120b",帶 response_format: {type:"json_object"},
temperature 0.8,max_tokens 600。

【prompt】程式組 prompt 時原樣使用下面這段(${} 處帶入資料):
你是目標設定教練,使用「九宮格目標法」(源自原田メソッド,即大谷翔平用過的目標達成表的方法)。
使用者的 3×3 九宮格:中央(位置4)是核心目標,周圍 8 格(位置0-3、5-8)是支撐核心目標的子目標。
${核心目標,留空時改寫成:核心目標也留空,請先推斷一個具體可衡量的核心目標}
已填的格子:
${每個已填格一行:位置i:「文字」;全空時寫(其餘全空)}
請補滿所有空格。規則:
1. 子目標要具體、彼此不重複、共同支撐核心目標;盡量涵蓋「心・技・體・生活」四個面向(原田メソッド慣例)。
2. 每格 2-8 個字,繁體中文(台灣用語),不用標點結尾,不用 emoji。
3. 已填的格子原樣保留,一字不改。
只輸出 JSON:{"cells":["位置0文字","位置1文字",...,"位置8文字"]},長度必為 9。

【驗收規則】收到回應後:先 JSON.parse,失敗再從 ```json 程式碼框裡撈;
cells 必須是長度 9 的字串陣列,否則顯示「這次沒有生成成功,請重試」;
只把原本是 null 的格子寫入(來源標 'ai',格子加淡金邊框和 AI 徽章),
原本有字的格子一律保留原值不覆蓋。
九格全空時按鈕要擋下並提示「至少先填一格」;九格全滿時提示不需要補。
```

常見坑

1. **key 貼進前端就上線**:練習版的「貼 key 直連」只能自己玩。要給別人用,key 必須照本章架構搬進 Worker Secret——這是底線,不是建議。
2. **相信 AI 會乖乖回 JSON**:`response_format` 大幅降低翻車率,但不是零。`parseJSONLoose` 加長度驗證那兩層不能省,省了就是某天頁面莫名其妙顯示 `undefined`。
3. **AI「好心」改了使用者的字**:prompt 叫它別改,它偶爾還是改。程式端用原值蓋回去那一行(`normCells.forEach`)才是真正的保護。
4. **失敗也扣額度**:忘了退款,使用者遇到 AI 故障還被扣次數,客訴文案都不知道怎麼寫。記住口訣:先扣後跑、確定失敗就退。
5. **拿聊天模型做結構化輸出**:llama-3.3-70b 在 Groq 上跑 tool-call/JSON 會被退是實戰雷。換場景要重測模型,別沿用上一個專案的選擇。

6. **prompt 沒寫字數限制**:AI 給你一格三十個字的「子目標」,桌布版面直接爆炸。版面約束要寫進 prompt(每格 2-8 個字),別等渲染時才處理。

對照成品

- `demos/02-ai-fill/` — 本章的單檔練習版:九宮格 + 貼自己的 Groq key 直連補格,不依賴 Worker,雙擊就能玩。
- `app/` — 完整版:同一套 prompt 放在 Worker(`worker-deploy/src/lib/prompts.js`),走 `/fill` 端點,key 藏在 Secret,加上額度與退款;前端在 `app/js/api.js` 的 `fillCore` / `expandSub`。先玩 demo 理解流程,再對照完整版看「上線版多了哪些防護」。

第 4 章 — 圖片上傳與前端縮圖:在瀏覽器裡先把圖變小

學完能做什麼

- 讓使用者從手機/電腦選照片當 AI 的參考圖,選完先在瀏覽器裡縮小,再送出。
- 把圖片轉成 API 要的 base64 格式,並知道那串 `data:image/...` 前綴為什麼要拿掉。
- 把圖片存進 IndexedDB,重新整理頁面照片還在——而且知道為什麼不能用 localStorage 存。

核心觀念:參考圖是給 AI 看的,不是要印海報

使用者上傳的照片是要給 AI 當風格與人物參考(第 7 章的 AI 編輯迴圈會用到),AI 看一張長邊 1280 像素的圖,跟看一張 4000×3000 的原圖,理解到的構圖、人物、氛圍幾乎沒有差別。但檔案大小差了十倍以上——傳原圖等於付費請 AI 看你自己都看不出差別的細節,還順便把上傳時間拉長十倍、把伺服器的請求大小上限撞爆。

所以原則是:**在瀏覽器裡先縮,再上傳**。現代瀏覽器本身就是一台夠用的影像處理器(讀檔、解碼、縮放、轉格式、轉編碼都有內建 API),這章就是把這條管線一步步接起來:選檔 → 縮圖 → 轉 base64 → 存起來。

步驟

1. 選檔:一個 input 就夠

```
<input id="refs-input" type="file" accept="image/*" multiple>
```

`accept="image/*"` 讓手機直接開相簿,`multiple` 允許一次選多張。本專案限制最多 3 張(上游生圖服務的約定),超過就把 input 停用。

2. 縮圖:解碼 → 畫到 canvas → 輸出 JPEG

核心工具是 `app/js/wallpaper.js` 裡的 `resizeImageBlob`,整段如下:

```
// 縮圖:長邊縮到 maxEdge,輸出 JPEG blob
export async function resizeImageBlob(blob, maxEdge = 1280, quality = 0.85) {
  const bitmap = await createImageBitmap(blob);   // 1. 解碼成點陣圖
  const scale = Math.min(1, maxEdge / Math.max(bitmap.width, bitmap.height));
  const w = Math.max(1, Math.round(bitmap.width * scale));
  const h = Math.max(1, Math.round(bitmap.height * scale));
  const c = document.createElement('canvas');     // 2. 開一張縮小後尺寸的畫布
  c.width = w;
  c.height = h;
  c.getContext('2d').drawImage(bitmap, 0, 0, w, h); // 3. 把圖畫上去 = 縮放
  bitmap.close();
  return new Promise((resolve, reject) => {
    c.toBlob((out) => {                           // 4. 畫布輸出成 JPEG 檔
      if (out) resolve(out);
      else reject(new Error('縮圖失敗'));
    }, 'image/jpeg', quality);
  });
}
```

逐行白話:

- `createImageBitmap(blob)`:把圖片檔解碼成記憶體裡的點陣圖,什麼常見格式進來都吃。
- `Math.min(1, ...)`:只縮小、不放大——圖本來就比 1280 小就原尺寸通過。
- `drawImage(...)` 畫到一張比較小的 canvas 上,縮放就完成了,瀏覽器自帶平滑演算法。
- `toBlob('image/jpeg', 0.85)`:輸出成 85% 品質的 JPEG。參考圖不需要無損,JPEG 比 PNG 小很多。

一張 8MB 的手機原圖,經過這條管線通常剩 150-400KB,肉眼看內容沒差。

3. 轉 base64,並把前綴拿掉

生圖 API 的約定是收「純 base64 字串」(欄位 `reference_images_base64`)。
`FileReader.readAsDataURL` 讀出來的其實是「data URL」,長這樣:

```
...
└──────── 前綴 ──────────┘└── 真正的 base64 ──
```

所以轉完要把逗號前面那段切掉(`app/js/wallpaper.js`):

```
// Blob → base64(去掉 data: 前綴,符合 upstream reference_images_base64 約定)
export function blobToBase64(blob) {
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = () => resolve(String(reader.result).split(',')[1] || '');
    reader.onerror = () => reject(reader.error);
    reader.readAsDataURL(blob);
  });
}
```

要不要去前綴沒有對錯,純看你呼叫的 API 怎麼約——文件說收 data URL 就留著,說收 raw base64 就切掉。撞到 400 錯誤時,這是第一個該檢查的地方。

4. 為什麼一定要前端先縮:算給你看

假設使用者選了 3 張手機原圖,每張 6MB:

- base64 編碼會讓體積再膨脹約 33%:6MB × 1.33 ≒ 8MB
- 3 張 = 24MB,整包 JSON 超過伺服器 nginx 的 20MB 請求上限(`client_max_body_size`),整個請求直接被擋,連 AI 的面都沒見到
- 就算過了,Worker 端還有一道驗證:每張 base64 超過 4MB 回 400(`worker-deploy/src/index.js` 的 `MAX_REF_B64`)

縮到長邊 1280 之後:每張約 300KB,base64 後約 400KB,3 張加起來 1.2MB——上傳幾秒內完成,離所有上限都遠。**體積問題在最靠近源頭的地方解決最便宜**,這裡的源頭就是使用者的瀏覽器。

5. 存進 IndexedDB:重新整理照片還在

文字狀態存 localStorage(第 2 章),但 localStorage 只能存字串、總量約 5-10MB,拿來存圖片很快就爆。瀏覽器專門存大塊資料的是 **IndexedDB**,可以直接存 Blob(二進位圖檔),空間以 GB 計。

開庫的樣板程式(`app/js/state.js`,寫一次之後到處用):

```
function openDb() {
  return new Promise((resolve, reject) => {
    const req = indexedDB.open('goal-grid', 1);
    req.onupgradeneeded = () => {
      // 兩個櫃子:'refs' 放參考圖、'wallpapers' 放生成的背景
      for (const name of ['refs', 'wallpapers']) {
        if (!req.result.objectStoreNames.contains(name)) {
          req.result.createObjectStore(name, { keyPath: 'id', autoIncrement: true });
        }
      }
    };
    req.onsuccess = () => resolve(req.result);
    req.onerror = () => reject(req.error);
  });
}
```

包成三個好用的小函式之後(完整版在 `app/js/state.js` 的 `idbPut` / `idbAll` / `idbDelete`),上傳流程就是把所有零件串起來(`app/js/main.js`):

```
refsInput.addEventListener('change', async () => {
  const files = [...refsInput.files].slice(0, 3 - refRecords.length); // 最多 3 張
  refsInput.value = '';
  for (const file of files) {
    try {
      const blob = await resizeImageBlob(file, 1280); // 長邊 1280,控制 body 體積
      await idbPut('refs', { blob, meta: { name: file.name }, createdAt: new Date().toISOString()
});
    } catch (err) {
      showMsg(bgMsgEl, '這張圖讀不進來,請換一張(支援一般圖片格式)。');
    }
  }
  loadRefs(); // 重新從 IndexedDB 讀出來畫縮圖列表
});
```

注意 `try/catch`:少數格式(例如某些 HEIC)瀏覽器解不開,`createImageBitmap` 會丟錯——接住、給一句人話、讓使用者換一張,不要讓整個頁面停擺。

6. 顯示縮圖:借用記憶體網址

從 IndexedDB 讀出來的 Blob 要顯示成 `<img>`,用 `URL.createObjectURL` 生一個臨時網址,圖載入完就釋放,不佔記憶體:

```
const img = document.createElement('img');
img.src = URL.createObjectURL(r.blob);
img.addEventListener('load', () => URL.revokeObjectURL(img.src), { once: true });
```

到這裡,參考圖管線完工:選檔 → 縮圖 → 存 IndexedDB → 顯示縮圖;要送 AI 時再 `blobToBase64` 轉格式。送出去的部分屬於第 5、7 章。

給 AI 的 prompt 範本

在第 3 章成品的基礎上加上傳功能,貼這段:

```
請在我現有的九宮格網頁(單一 HTML 檔)加一個「參考圖上傳區」:

【選檔】一個 input type="file" accept="image/*" multiple,最多收 3 張;
已滿 3 張時停用選檔並提示。

【縮圖】每張選進來的圖先在前端縮小:用 createImageBitmap 解碼,
長邊超過 1280 就等比例縮到 1280(不放大),畫到 canvas 後以
canvas.toBlob 輸出 image/jpeg、品質 0.85。解碼失敗的檔案要 try/catch
接住,顯示「這張圖讀不進來,請換一張」,不能讓頁面壞掉。

【儲存】縮好的 JPEG Blob 存進 IndexedDB:資料庫名 'goal-grid',
object store 名 'refs',keyPath 'id' 自動編號,
每筆 {id, blob, meta:{name}, createdAt}。頁面載入時把 'refs' 全部讀出來。

【顯示】上傳區下方顯示縮圖列表:每張用 URL.createObjectURL(blob) 當
img src,載入完成後 revokeObjectURL;每張縮圖旁有「移除」按鈕,
按了從 IndexedDB 刪除並重畫列表。

【轉換工具】另外給我一個 blobToBase64(blob) 函式:用 FileReader
readAsDataURL 讀出後,把 "data:...;base64," 前綴切掉只留純 base64
(之後要送給生圖 API 用,先寫好備用)。

【驗收】重新整理頁面,已上傳的縮圖還在;選一張 5MB 以上的手機照片,
存進去的 Blob 要明顯小於 1MB(可在 console 印 blob.size 確認)。
風格沿用現有頁面(深夜藍 #0e1420 + 金 #c9a44e),繁體中文,不用 emoji。
```

常見坑

1. **原圖直接 base64 上傳**:三張手機照片就能撞破 20MB 請求上限,而且 base64 還會放大 33%。先縮再傳,沒有例外。
2. **忘記去掉 data URL 前綴**:API 約定收純 base64 時,帶著 `` 前綴送過去就是 400。`split(',')[1]` 一行的事。
3. **iPhone 的 HEIC 檔解不開**:部分瀏覽器的 `createImageBitmap` 不支援 HEIC,會直接丟錯。一定要 `try/catch` 包住並給出「請換一張」的人話提示(iPhone 使用者可以從相簿「分享為 JPEG」或截圖代替)。
4. **拿 localStorage 存圖**:它只能存字串(就得存 base64,又膨脹 33%),配額約 5-10MB,存兩三張就滿,然後整個 app 的狀態保存跟著一起壞。圖片一律 IndexedDB。

5. **createObjectURL 沒釋放**:每呼叫一次就佔一塊記憶體,列表重畫幾十次後頁面越來越肥。圖片 `load` 之後就 `revokeObjectURL`。
6. **放大小圖**:縮圖函式忘了 `Math.min(1, ...)`,800px 的小圖被放大到 1280,糊了還變大。只縮不放。

對照成品

`app/` 的步驟四就是本章管線的完整版:選檔與縮圖列表在 `app/js/main.js`(搜尋 `refsInput`),縮圖與 base64 工具在 `app/js/wallpaper.js`(`resizeImageBlob`、`blobToBase64`),IndexedDB 三件組在 `app/js/state.js`(`idbPut`/`idbAll`/`idbDelete`)。這些參考圖在第 5 章送給 AI 生背景、在第 7 章的編輯迴圈再次出場。

第 5 章 — AI 桌布背景:長任務的正確姿勢(job + 輪詢 + 等待 UX)

學完能做什麼

- 接一個要跑 1-3 分鐘的 AI 生圖任務,而且手機收訊閃一下、使用者重新整理頁面,結果都不會丟。
- 把「同步等待」改造成「掛號-叫號」:送單拿單號,之後每幾秒問一次好了沒。
- 看懂上線版的三層生圖架構:主路 OpenAI 同步、備援自架 job + 輪詢、降級 prompt 外帶——以及為什麼「夠快夠穩」的主路可以同步。
- 設計一條誠實的進度條與階段文案,讓使用者願意等三分鐘。

核心觀念:快任務和慢任務是兩種工程

到上一章為止,所有 API 呼叫都在幾秒內回來,HTTP 的預設假設——「問了馬上答」——都成立。AI 生圖打破了這個假設:一張桌布背景快則幾十秒、慢則 70-180 秒。讓一條 HTTP 連線開著等三分鐘,等於在賭三件事同時不出錯:中間的代理不逾時(Cloudflare Worker 的子請求,社群實測約 90-100 秒就可能被切)、使用者的手機網路三分鐘不抖一下、使用者三分鐘內不切換頁面。在行動網路上,這個賭局幾乎必輸——而且最慘的是,連線斷掉時圖其實已經快生好了,只是你再也拿不回來。

正解是把一次「長對話」拆成多次「短對話」:送出任務立刻拿到一個單號(job id),連線馬上結束;之後每隔幾秒拿單號去問「好了嗎」。每次請求都在一秒內完成,逾時風險整類消失,而且單號可以存起來——重新整理頁面、甚至換個瀏覽器分頁,拿著單號就能接著等。這個模式叫**非同步 job + 輪詢(polling)**,所有「AI 慢任務」(生圖、生影片、批次處理)都是這一套。

那為什麼上線版的主路是同步的?

先說一個會讓你疑惑的事實:上線版 Worker 的生圖主路其實是同步的。完整的生圖路徑是三層:

1. **主路 = OpenAI Images API(gpt-image-1)**:官方付費 API,夠快也夠穩,多數圖在 90 秒內回得來。Worker 同步呼叫、用 `AbortSignal.timeout(90000)` 在 90 秒截斷(正好卡在代理層被切線的臨界值之前),拿到圖直接回 `{mode:'sync', images}`。
2. **備援 = 自架 codex-image-service**:Worker 沒設 `OPENAI_API_KEY` 時才走。自架服務一張 70-180 秒、實質併發只有一到二張,「同步硬等」的三重賭局全部成立——所以這條路必須用 job + 輪詢。
3. **降級 = prompt 外帶**:兩條後端路都不通時,複製生圖 prompt 自己去 ChatGPT/Gemini 生、再匯回來,零後端依賴(第 07 章的教材)。

「同步」不是打臉前面講的原則,是有條件的例外:上游夠快夠穩、又有明確截斷時,同步是最簡單正確的做法;上游慢或不穩,job + 輪詢才是正解。本章接下來教的就是第二層——它既是這個產品備援路徑的實作,也是所有 AI 慢任務的通用骨架,換掉哪個上游都用得上。

步驟

0. 主路:OpenAI Images API,同步 90 秒截斷

`worker-deploy/src/lib/openaiimage.js` 把官方 API 包成一個函式:文生圖打 `/v1/images/generations`,帶參考圖時改打 `/v1/images/edits`(multipart,把 base64 還原成 PNG 附件),品質由 `OPENAI_IMAGE_QUALITY` 環境變數控制(預設 medium)。`handleImage` 的主路分支(節錄,有簡化):

```
// 主路:OpenAI Images API(同步,90s 截斷)
if (env.OPENAI_API_KEY) {
  const oa = await generateOpenAI(env, { prompt, size: payload.size, refsBase64: refs });
  if (oa.timeout) return fail({ error: "sync_fallback_timeout" }, 504); // 退額度 + 解鎖
  if (oa.status === 200) {
    await releaseLock(env.DB, ip);            // 成功:解鎖但不退款
    return json({ mode: "sync", images: oa.images }, 200, cors);
  }
  return fail({ error: "upstream_failed" }, 502);
}
// 沒設 OPENAI_API_KEY → 往下走自架備援(本章其餘步驟的主角)
```

注意它沒有「OpenAI 失敗就改打自架備援」——失敗就是退款、回錯誤,讓前端引導使用者重試或走降級。兩條路用金鑰擇一決定,行為單純可預測(debug 的時候你會感謝這個決定)。

1. 備援的前置工程:先讓自架上游支援 job

備援層的生圖後端是 codex-image-service(自架服務)。它原本只有同步端點 `POST /v1/images/generate`——連線開著等 70-180 秒,而且它自己的文件就承認:504 之後那次的結果拿不回來。

所以這個專案動工前,第一件事不是寫前端,而是去上游 repo 開了一個 PR(codex-image-service PR #5),加上兩個 job 端點:

```
POST /v1/images/jobs        → 202 {id, status:"queued"}   (參數跟原本的 generate 一樣)
GET  /v1/images/jobs/{id}   → {status: queued|running|succeeded|failed,
                               images: [{url, expires_at}], error}
```

既有的同步端點原樣保留(向下相容),只是多了一條「掛號」路徑。這裡有一個值得帶走的觀念:**當你依賴的服務缺一個能力,選項不是只有「忍耐」或「換一家」,還有「去把它修好」**——尤其當它是開源專案或自家服務的時候。

2. Worker 送單:鎖、扣額度、拿單號

`worker-deploy/src/index.js` 的 `handleImage` 是送單端。鎖與扣額度是主路、備援共用的(進來先做,走哪條路都一樣);`submitJob` 之後是備援限定——步驟 0 的主路分支沒攔截(`OPENAI_API_KEY` 未設)才會

走到。流程節錄:

```
// 同一個 IP 同時只能有一張在生(in-flight lock),搶不到鎖回 409
if (!(await acquireLock(env.DB, ip, "pending"))) {
  return json({ error: "in_flight" }, 409, cors);
}
// 先扣後跑(同第 3 章):每日生圖額度先 -1
const quota = await takeQuota(env.DB, { kind: "img", ip, limit: 3, day });
if (!quota.ok) {
  await releaseLock(env.DB, ip);
  return json({ error: "quota_exceeded", resetAt: taipeiResetAt() }, 429, cors);
}

const result = await submitJob(env, payload);   // POST 上游 /v1/images/jobs
if (result.mode === "job" && result.data?.id) {
  const jobId = result.data.id;
  await putJob(env.DB, jobId, { ip, day });     // 記下單號 + 建立時間(逾時判定用)
  await updateLock(env.DB, ip, jobId);
  return json({ jobId, mode: "job" }, 202, cors); // 202 = 已受理,還沒做完
}
```

`submitJob`(`worker-deploy/src/lib/imagejobs.js`)還藏了一手:先打 job 端點,如果上游回 404(代表 job API 還沒部署),自動退回同步端點、用 `AbortSignal.timeout(85000)` 在 85 秒截斷。這讓 Worker 和上游可以**分開部署、互不卡死**——協定升級永遠要留向下相容的路。

3. Worker 查單:終局處理與 660 秒自動退款

前端拿單號來問進度時,Worker 轉發給上游,並且在「終局」做該做的事(`handleImageStatus`):

```
const st = res.data?.status;
if (st === "queued" || st === "running") {
  // job 建立超過 660 秒還沒好 → 主動判逾時:退額度 + 解鎖
  if (jobTimedOut(job)) {
    await refundJob(env.DB, jobId);
    return json({ status: "failed", error: "timeout", refunded: true }, 200, cors);
  }
  return json({ status: st }, 200, cors);        // 還在做,前端繼續等
}
if (st === "succeeded") {
  await releaseLock(env.DB, job.ip);             // 成功:解鎖(額度照扣)
  return json({ status: "succeeded", images: res.data.images || [] }, 200, cors);
}
await refundJob(env.DB, jobId);                  // 失敗:退額度 + 解鎖
return json({ status: "failed", error: String(res.data?.error || "failed") }, 200, cors);
```

兩個設計重點:

- **660 秒逾時自動退款**:上游卡死不能變成使用者的損失。退款動作是「冪等」的——`refundJob` 內部先檢查這筆 job 退過沒,退過就跳過,所以前端就算連問十次逾時,也只會退一次,額度不會被「退」成負

的。

- **第 3 章的口訣在這裡重演**:先扣後跑、確定失敗才退。只是生圖的「確定失敗」可能發生在十分鐘後,所以 Worker 得把單號、IP、建立時間記在資料庫裡,事後才對得起帳。

4. 前端輪詢:5 秒一問,90 秒後放慢

前端的 `submitWallpaper` 先看 Worker 的回應決定路線:主路會直接給 `{mode:'sync', images}`——拿圖收工,根本不用輪詢;拿到 `{mode:'job', jobId}`(備援路徑)才進輪詢主迴圈。`app/js/api.js` 的輪詢節奏:

```
export function pollDelay(attempt) {
  return attempt < 18 ? 5000 : 10000; // 5s × 18 次(90s)後改 10s
}
```

前 90 秒每 5 秒問一次(多數圖在這個區間完成,問太慢使用者白等),之後降到每 10 秒(已經等很久了,問密一點也不會更快,只是浪費流量)。輪詢主迴圈(`pollWallpaper`,節錄):

```
while (true) {
  const delay = pollDelay(attempt);
  if (elapsed + delay > 720000) throw new ApiError(0, { error: 'poll_timeout' }); // 720s 放棄
  await sleep(delay);
  elapsed += delay;
  attempt += 1;

  try {
    res = await fetch(`${workerBase()}/image/${jobId}`);
    body = await res.json();
  } catch {
    consecutiveErrors += 1;                          // 網路抖一下不算失敗
    if (consecutiveErrors >= 4) throw new ApiError(0, { error: 'network' });
    continue;                                        // 連續錯 4 次才放棄
  }
  consecutiveErrors = 0;

  if (body.status === 'succeeded') return body.images;
  if (body.status === 'failed') throw new ApiError(200, { error: 'job_failed', detail: body.error
});
  // queued / running → 繼續下一輪
}
```

注意兩個數字的關係:前端等到 720 秒才放棄,**比 Worker 的 660 秒寬**——這樣設計,使用者一定會先收到 Worker 那句「逾時、額度已退還」,而不是前端自己先放棄、留下「到底退了沒」的懸念。超時這種事,要讓「管帳的那一方」先開口。

5. pending job 持久化:重新整理也不丟單

單號拿到手的那一刻就立刻寫進 localStorage(`app/js/main.js`):

```
api.savePendingJob({
  jobId: sub.jobId,
  createdAt: new Date().toISOString(),   // 續傳時用來換算「已經等了多久」
  orientation: state.wallpaper.orientation,
  styleId: state.wallpaper.styleId,
  gridMode: wallpaperMode,
});
images = await api.pollWallpaper(sub.jobId, { byo: sub.byo });
api.clearPendingJob();                   // 終局(成功)才清掉
```

頁面載入時檢查有沒有「未終局」的單:

```
const pending = api.loadPendingJob();
if (pending) {
  goToStep(4); // 直接帶回步驟四
  showMsg(bgMsgEl, '偵測到上次離開時背景還在生成,已自動接續等待(不會重複扣額度)。', 'ok');
  const elapsedMs = Date.now() - Date.parse(pending.createdAt);
  startGenUi(Date.now() - elapsedMs);    // 進度條從「已經過的時間」接著走,不歸零
  const images = await api.pollWallpaper(pending.jobId, { elapsedMs });
  api.clearPendingJob();
  await applyWallpaperImages(images);
}
```

幾個細節讓續傳「像沒斷過」:進度條從實際經過時間接著走(不會重新從 0 開始騙人);輪詢節奏用 `elapsedMs` 換算回正確的位置(已經過了 90 秒就直接用 10 秒節奏);成功、失敗、逾時這些「終局」都會清掉 pending 單,但**純網路錯誤不清**——網路恢復或下次載入頁面時還能接著等。另外,使用者手癢再按一次「生成」時,程式發現有 pending 單會改成恢復輪詢而不是重送——重送只會撞上 Worker 的 409(同 IP 同時只能一張)。

6. 等待 UX:三分鐘等得下去的進度條

上游不會告訴你「畫到 73% 了」,所以進度條必然是「演的」——但可以演得誠實(`app/js/main.js`):

```
const GEN_STAGES = ['AI 構圖中', 'AI 正在打草稿', '上色中', '收尾修飾中'];

const tick = () => {
  const elapsed = Date.now() - startedAt;
  const idx = Math.floor(elapsed / 9000) % GEN_STAGES.length;   // 階段文案每 9 秒輪播
  const mm = String(Math.floor(elapsed / 60000));
  const ss = String(Math.floor((elapsed % 60000) / 1000)).padStart(2, '0');
  genStageEl.textContent = `${GEN_STAGES[idx]}…(已 ${mm}:${ss},一般約 1-3 分鐘)`;
  pbarFill.style.width = `${Math.min(96, (elapsed / 180000) * 100).toFixed(1)}%`;
};
```

這短短幾行裡有四個行銷人秒懂的心理設計:

1. **預告時間**:「一般約 1-3 分鐘」先講,使用者的耐心是用預期管理出來的,不是用動畫拖出來的。

2. **顯示已經過時間**:`已 1:24` 證明系統活著,沒有什麼比一條不動的進度條更勸退。
3. **階段文案輪播**:構圖中 → 打草稿 → 上色中 → 收尾修飾——把黑盒子講成一個有進展的故事(文案是演的,但「還在跑」是真的)。
4. **進度條封頂 96%**:用預估 180 秒當分母,但永遠不走到 100%。走到 100% 卻還沒好,是等待 UX 的頭號災難;96% 的「快好了」可以體面地撐到真的好。

最後一哩:成功拿到圖之後,**立刻把圖抓回來存 IndexedDB**(`applyWallpaperImages`)。上游給的圖片網址 7 天就過期,自己的成品要放在自己手上——這也呼應本課一直講的資料自主。

給 AI 的 prompt 範本

這段貼給 ChatGPT/Claude,可以重現本章的「長任務輪詢」骨架(對著任何 job 式 API 都適用;搭配本課 Worker 時把網址換成你自己部署的——要走備援路徑、也就是不設 `OPENAI_API_KEY`,才會拿到 202 與 jobId;主路會直接同步回圖):

```
請幫我做一個單檔網頁(所有 CSS/JS 內嵌),示範「AI 長任務的送單與輪詢」:

【送單】一顆「開始生成」按鈕,POST {WORKER_URL}/image,
body 是 {goal, subGoals, styleId, orientation} 的 JSON;
成功會拿到 202 和 {jobId}。送出的瞬間把
{jobId, createdAt: 現在時間 ISO 字串} 存進 localStorage(key 'pending-job')。

【輪詢】拿 jobId 每 5 秒 GET {WORKER_URL}/image/{jobId} 問一次,
問滿 18 次(90 秒)後改成每 10 秒一次;總共等超過 720 秒就放棄並顯示逾時訊息。
回應的 status 是 queued 或 running 就繼續等;
succeeded 就把 images[0].url 顯示成圖片並清掉 localStorage 的 pending-job;
failed 就顯示錯誤訊息(error 欄位)也清掉 pending-job。
fetch 本身丟錯(網路抖動)不算失敗:連續錯 4 次才放棄,而且不清 pending-job。

【續傳】頁面載入時檢查 localStorage 有沒有 pending-job:
有的話不要重送,直接用那個 jobId 接著輪詢,
並依 createdAt 算出已經過的時間,讓進度顯示從正確位置接著走。

【等待 UI】等待期間顯示:
1. 一條進度條,以 180 秒為分母推進,但最多走到 96%、不到 100%;
2. 階段文案每 9 秒輪播:「AI 構圖中」「AI 正在打草稿」「上色中」「收尾修飾中」,
   後面接已經過時間(分:秒)和「一般約 1-3 分鐘」;
3. 生成期間按鈕停用,結束(成功或失敗)後恢復。

【驗收】生成中重新整理頁面,要能自動接著等、進度不歸零、不重送請求。
深夜藍底(#0e1420)配金色(#c9a44e),繁體中文,不用 emoji。
```

常見坑

1. **同步硬等三分鐘**:代理層 90-100 秒就可能切線,手機網路更脆,而且 504 之後結果拿不回——錢花了、圖生了、人看不到。長任務一律 job + 輪詢;同步只留給「上游夠快夠穩、又有明確截斷」的場合(本

課主路打 OpenAI、90 秒截斷,就是這種例外)。

2. **輪詢太密**:每秒問一次不會讓圖快一秒,只會把流量和伺服器額度燒掉。5 秒起步、90 秒後退到 10 秒,夠了。
3. **重新整理就重送**:沒存單號的頁面,reload 等於棄單重排,還會撞 Worker 的 409(同 IP 一次一張)。單號到手立刻進 localStorage,載入時先查再說。
4. **終局忘了清 pending 單**:成功/失敗/逾時都要清,否則使用者永遠卡在「偵測到上次的生成」;但網路錯誤不清——那不是 job 的終局,留著才能續傳。
5. **進度條走到 100% 卡住**:不知道真實進度就不要演到滿,封頂 96% 是誠實與體驗的平衡點。
6. **退款不冪等**:輪詢會對同一個失敗 job 問很多次,退款邏輯沒有「退過就跳過」的檢查,額度就會被多退。Worker 的 `refundJob` 先查 `refunded` 旗標就是在防這個。
7. **直接引用上游圖片網址**:那個網址 7 天過期。成功當下就把圖抓回存 IndexedDB,桌布才真正是使用者的。

對照成品

- `demos/03-wallpaper/` — 本章+第 6 章的練習版:漸層背景 + canvas 合成可離線玩,AI 背景為選配(自帶 key)。
- `app/` — 完整版:送單與查單在 `worker-deploy/src/index.js`(`handleImage` / `handleImageStatus`),主路 OpenAI 介接在 `worker-deploy/src/lib/openaiimage.js`,備援上游介接與同步 fallback 在 `worker-deploy/src/lib/imagejobs.js`,前端輪詢與續傳在 `app/js/api.js`(`submitWallpaper` / `pollWallpaper` / `savePendingJob`),等待 UI 在 `app/js/main.js`(搜尋 `GEN_STAGES`)。備援上游的 job API 出處:codex-image-service PR #5(https://github.com/yazelin/codex-image-service/pull/5)。

模組 6 — canvas 文字合成:AI 畫背景、canvas 畫字

學完能做什麼

- 講得出為什麼即使現代繪圖模型已經會寫中文,正式產品仍該用 canvas 疊字:關鍵在「之後怎麼改」。
- 用 canvas 把九宮格文字精準疊在任何背景圖上:cover 縮放鋪滿、暗化遮罩保字、字級自動縮放、中文斷行(英文單字不被攔腰折斷)。
- 套用「中央格最強色、八個子目標各配一色」的配色慣例,輸出一張可直接下載的 PNG 桌布。

先想清楚:文字讓 AI 畫,還是程式畫?

開始之前,先做一個五分鐘的對照實驗。打開任何一個生圖工具(ChatGPT、Gemini 都行),貼這段:

```
畫一張 3×3 的目標九宮格表格圖,深藍色底、金色框線。
中央格寫「完成全馬 42.195km」,
周圍八格依序寫:「每週跑 40km」「核心訓練」「睡滿 7 小時」
「補給策略」「配速練習」「比賽報名」「飲食管理」「跑姿調整」。
要求:每一格的繁體中文都清晰可讀、一字不差。
```

先說句公道話:「AI 寫不好中文字」這件事,在新一代的強繪圖模型(gpt-image 新版、Gemini 的新繪圖模型)上已經進步很多——九格大字常常一次就寫對。但在許多免費管道、較舊或較小的模型上,你仍會看到三種典型災難:偽漢字(筆畫黏成字典裡沒有的字)、缺字漏字、簡繁混雜;而且同一個 prompt 重抽三次,錯的地方都不一樣。

所以真正的決策點不是「AI 會不會寫中文」,而是——**字畫進圖裡之後,你要怎麼改?**

1. **改字成本**:文字烤進圖裡,改一個字 = 整張重生。一次重生是幾十秒到幾分鐘加一筆生圖費,而且構圖會跟著變,「只改那個字、其他都不動」做不到。canvas 疊字改文字是即時、免費、其他像素一動不動。
2. **驗收成本**:模型再強也是機率性輸出,9 格要逐格人工檢查,81 格小字更要;程式畫的字是確定性的,**100% 一字不差,不用驗**。
3. **版型保證**:自動縮字、斷行、絕不重疊、八色對應——這些是排版引擎的工作,canvas 給你保證,模型只能給你機率。

一句話總結這個架構決定:**文字是「資料」,背景是「藝術」**。資料要可程式化地修改與驗收,藝術才交給模型發揮。這也是為什麼「之後要改文字比較方便」是 canvas 合成的第一理由,模型寫不寫得好中文只是次要的歷史包袱。

核心觀念:分工——AI 畫氛圍,程式畫文字

正路是把工作拆成兩層:

- **AI 負責它擅長的**:氛圍、光線、風格、構圖——生成一張沒有任何文字的背景圖(模組 5 的生圖 prompt 裡有一條鐵律:`Do NOT render any text`)。
- **程式負責機器擅長的**:精準、可驗收的文字排版——用瀏覽器內建的 canvas,把九宮格一格一格畫上去。

背景是一張圖,文字是一層程式畫的圖,兩層合成後輸出 PNG。這個「解耦」還有一個下一章才兌現的紅利:改文字不必重新生圖,迭代零成本。

步驟

以下程式碼取自成品 `app/js/wallpaper.js`(有簡化,邏輯一致)。

1. 開一張「輸出尺寸」的 canvas

桌布要的是實際畫素,不是網頁排版尺寸:

```
const CANVAS = { portrait: { w: 1170, h: 2532 }, landscape: { w: 1920, h: 1080 } };

const { w: W, h: H } = CANVAS[orientation];
canvas.width = W;   // 這是輸出畫素,不是 CSS 樣式
canvas.height = H;
const ctx = canvas.getContext('2d');
```

直式 1170×2532 是 iPhone 等級的桌布解析度;預覽時用 CSS 把 canvas 縮小顯示即可,輸出尺寸不受影響。

2. 背景 cover 縮放鋪滿

AI 生的背景是 1024×1536(或 1536×1024),跟畫布比例不完全相同。處理方式跟 CSS 的 `background-size: cover` 同一個邏輯:放大到剛好蓋滿、超出的部分置中裁掉。

```
// 算出來源圖要裁切的區域 {sx, sy, sw, sh}
function coverRect(srcW, srcH, dstW, dstH) {
  const scale = Math.max(dstW / srcW, dstH / srcH);
  const sw = dstW / scale;
  const sh = dstH / scale;
  return { sx: (srcW - sw) / 2, sy: (srcH - sh) / 2, sw, sh };
}

const { sx, sy, sw, sh } = coverRect(bgBitmap.width, bgBitmap.height, W, H);
ctx.drawImage(bgBitmap, sx, sy, sw, sh, 0, 0, W, H);
```

好處是這個函式不挑圖:任何比例的圖丟進來都能鋪滿(下一章「匯入背景圖」靠的就是它)。

3. 暗化遮罩:字的可讀性不能賭 AI 聽話

生圖 prompt 雖然要求「中央區域保持低對比留白」,但模型不一定每次都聽話。可讀性要靠自己保證,雙保險:

```
// 保險一:全幅半透明暗化遮罩
ctx.fillStyle = 'rgba(8, 10, 16, 0.30)';
ctx.fillRect(0, 0, W, H);

// 保險二:每一格自己再墊一層更深的格底(畫格子時用)
const CELL_BG = 'rgba(12, 16, 24, 0.58)';
```

這樣不管背景多花,字底下永遠是深色,白字、淺色字一定讀得到。

4. fitText:二分搜尋找出塞得下的最大字級

九宮格每格的文字長度差異很大(「英文」兩個字 vs「睡前聽英文 podcast」九個字),固定字級必爆版。做法是用二分搜尋找「換行後仍塞得進格子的最大字級」:

```
function fitText(measureFactory, text, opts) {
  const { maxWidth, maxHeight, maxFontSize, minFontSize = 14, lineHeight = 1.32 } = opts;
  const tryFit = (fs) => {
    const measure = measureFactory(fs);          // 該字級下的量寬函式
    const lines = wrapCJK(text, maxWidth, measure); // 先斷行(見下一步)
    const heightOk = lines.length * fs * lineHeight <= maxHeight;
    const widthOk = lines.every((l) => measure(l) <= maxWidth + 0.01);
    return heightOk && widthOk ? lines : null;
  };
  let lo = minFontSize;
  let hi = Math.max(minFontSize, Math.floor(maxFontSize));
  let best = null;
  while (lo <= hi) {
    const mid = (lo + hi) >> 1;
    const lines = tryFit(mid);
    if (lines) { best = { fontSize: mid, lines }; lo = mid + 1; }
    else { hi = mid - 1; }
  }
  return best || { fontSize: minFontSize, lines: wrapCJK(text, maxWidth, measureFactory(minFontSiz
e)) };
}
```

量字寬用 canvas 自己的 `measureText`,但**量之前一定要先把 `ctx.font` 設成要量的字級**,不然量出來的是上一次的字寬:

```
const factory = (fs) => {
  ctx.font = `500 ${fs}px 'Noto Sans TC', 'Microsoft JhengHei', sans-serif`;
  return (s) => ctx.measureText(s).width;
};
const { fontSize, lines } = fitText(factory, '睡前聽英文 podcast', {
  maxWidth: cellW - pad * 2, maxHeight: cellH - pad * 2, maxFontSize: cellW * 0.15,
});
```

5. 中文斷行:逐字可斷,英數段不拆

中文沒有空格,canvas 也沒有自動換行,得自己斷。原則是:**中文字逐字都可以斷,但英數連續段(`160km`、`TOEIC`、`podcast`)是一個不可拆的單位**,放不下就整段移到下一行——拆成 `TOE` / `IC` 就不是人話了。

```
function wrapCJK(text, maxWidth, measure) {
  const lines = [];
  let line = '';
  const push = (unit) => {
    const next = line + unit;
    if (line && measure(next) > maxWidth) { lines.push(line); line = unit; }
    else { line = next; }
  };
  // 英數連續段視為不可拆單位;其餘逐字
  const units = String(text).match(/[A-Za-z0-9]+|[\s\S]/gu) || [];
  for (const unit of units) {
    if (unit === '\n') { lines.push(line); line = ''; continue; }
    if (unit.length > 1 && measure(unit) > maxWidth) {
      for (const ch of unit) push(ch); // 整段比格子還寬才退回逐字硬拆
    } else {
      push(unit);
    }
  }
  lines.push(line);
  return lines.length ? lines : [''];
}
```

關鍵在那行正規表達式:`/[A-Za-z0-9]+|[\s\S]/gu` 把文字切成「英數段」與「單一字元」兩種單位,後面就用同一套邏輯排。

6. 配色慣例:中央格最強、八色各自呼應

網路上流傳的大谷翔平目標表轉載圖有一套大家看慣的配色語言,我們沿用它:

- **中央格(核心目標)**:整張圖最強的顏色——風格主色 85% 不透明度當底、3px 同色邊框、白色粗體字。
- **八個子目標格**:深色格底 + 各自一條 4px 彩色左邊條,八格八色。
- **9×9 模式**:外圍每個區塊的中心格,用對應子目標的同一個顏色(填色 32% 透明度 + 同色邊框)——一眼看出「這個區塊在展開哪個子目標」。

```
// 顏色來自風格預設(styles-presets.js):每個風格帶 accent + sub[8]
const palette = presetById(styleId).palette;

// 位置 i(0-8,跳過中央 4)對應第幾個子目標
const subIndexOf = (pos) => (pos < 4 ? pos : pos - 1);

// hex 轉帶透明度的 rgba
function hexA(hex, alpha) {
  const n = parseInt(hex.slice(1), 16);
  return `rgba(${(n >> 16) & 255}, ${(n >> 8) & 255}, ${n & 255}, ${alpha})`;
}

// 中央格:fill = hexA(palette.accent, 0.85),邊框 palette.accent 3px,白字 700
// 子目標格 i:fill = CELL_BG,左邊條 palette.sub[subIndexOf(i)] 4px,米白字 500
```

7. 輸出 PNG

合成完直接從 canvas 匯出檔案,全程不經過任何伺服器:

```
canvas.toBlob((blob) => {
  const url = URL.createObjectURL(blob);
  const a = document.createElement('a');
  a.href = url;
  a.download = '目標九宮格.png';
  a.click();
  setTimeout(() => URL.revokeObjectURL(url), 4000);
}, 'image/png');
```

給 AI 的 prompt 範本

貼給 ChatGPT 或 Claude,可重現本章成品(單檔、雙擊能開):

```
做一個單檔 HTML(CSS/JS 全內嵌)的「九宮格桌布合成器」:

【輸入】九個文字框(3×3 排列,中央是核心目標),加一個「上傳背景圖」
檔案選擇器(沒上傳就用深藍到金色的 linearGradient 漸層當背景)。

【畫布】直式 1170×2532 / 橫式 1920×1080,一組切換鈕。canvas 的
width/height 設成輸出畫素,預覽用 CSS 縮小顯示(max-width: 100%)。

【合成順序】
1. 背景圖 cover 縮放鋪滿(模仿 background-size: cover:取
   max(dstW/srcW, dstH/srcH) 的縮放比,置中裁切),用 drawImage 的
   九參數形式畫。
2. 疊一層 rgba(8,10,16,0.3) 全幅暗化遮罩。
3. 畫 3×3 九宮格:格子總寬 = min(0.86*畫布寬, 0.52*畫布高),格間距 =
   格寬*0.06,圓角 = 格寬*0.08,直式置中於 44% 高、橫式置中。
4. 底部置中一行半透明小字「目標九宮格」。

【格子配色】中央格:金色 #c9a44e 85% 透明度當底 + 3px 金色邊框 + 白色
粗體字;其餘八格:rgba(12,16,24,0.58) 格底 + 各自一條 4px 彩色左邊條
(八格八色,在深色底上要可讀)。

【文字排版】每格文字要自動縮放字級:用二分搜尋找「斷行後寬高都塞得進
格子」的最大字級,最小 14px,行高 1.32。斷行規則:中文逐字可斷,
連續英數段(如 160km、TOEIC)視為不可拆單位整段換行。量字寬用
ctx.measureText,量之前先設好 ctx.font。字型 'Noto Sans TC' 加系統
fallback。

【輸出】「下載 PNG」按鈕用 canvas.toBlob 下載。

【驗收】
1. 某格填「睡前聽英文 podcast 三十分鐘」:不溢出格子、podcast 沒被拆半。
2. 上傳一張直的照片配橫式畫布:鋪滿、置中裁切、不變形。
3. 上傳一張很亮的圖:每格文字仍清楚可讀。
4. 下載的 PNG 實際尺寸是 1170×2532(或 1920×1080)。
```

常見坑

1. **還是想叫 AI 畫字**:短英文偶爾成功,會讓人誤以為中文也行。記住判準:錯的位置每次不一樣 = 無法驗收 = 不能上線。
2. **用 CSS 思維設 canvas 尺寸**:`canvas.width` 是輸出畫素、CSS 的 `width` 只是顯示大小,兩者各管各的。只設 CSS 會得到一張 300×150 的糊圖。
3. **忘了遮罩,字看不清**:別依賴生圖 prompt 的「中央留白」約束,模型不一定聽話。全幅遮罩 + 格底是自己掌握的保證。
4. **英文單字被攔腰折斷**:純逐字斷行會把 `podcast` 拆成 `pod` / `cast`。英數連續段要當一個單位處理。
5. **measureText 量錯寬**:量之前沒先設 `ctx.font`,量到的是上一個字級的寬度,版面就會時好時壞。

6. **跨網域圖片污染 canvas**:直接 `drawImage` 一張外站圖,沒過 CORS 的話 `toBlob` 會丟 SecurityError。成品的做法是把生成圖先 `fetch` 回來變 Blob、存進 IndexedDB,再 `createImageBitmap` 來畫——順便解掉生成圖網址七天過期的問題。
7. **toBlob 是非同步的**:結果在 callback 裡,別寫成同步取值。

對照成品

- `demos/03-wallpaper/`:本章成品(漸層或上傳背景 + 文字合成 + 下載),雙擊能開。
- `app/js/wallpaper.js`:完整版,多了 9×9 全圖版型與風格預設;`app/` 走完五步就能體驗「AI 背景 + canvas 文字」的完整管線。
- 建議把開頭對照實驗的圖跟本章成品擺在一起看一次——同樣的九格文字,一邊要逐格驗收、改字要整張重生,一邊是一字不差、改字即時。

模組 7 — AI 編輯迴圈與降級設計:prompt 是可攜帶的資產

學完能做什麼

- 讓使用者用一句自然語言修改 AI 背景(「整體亮一點,天空加一點晚霞」),而不是整張重抽碰運氣。
- 善用上一章「背景與文字解耦」的紅利:改文字、換配色即時重繪,零等待、零成本。
- 設計一條零後端依賴的降級路徑:服務忙線或沒設金鑰時,「複製生圖 prompt」讓使用者自己去 ChatGPT/Gemini 生,「匯入背景圖」把成品帶回來繼續合成——行銷頁永遠走得完。

核心觀念:迭代有兩種,成本差一千倍

使用者拿到第一張桌布,幾乎一定會想改。這時要分清楚改的是哪一層:

- **改背景**:要重新生圖,快則數十秒(OpenAI 主路)、慢則兩三分鐘(自架備援),扣一次額度。
- **改文字、換風格配色**:只是 canvas 重繪,不到 0.1 秒,免費,可以改一百次。

上一章把「AI 畫背景、canvas 畫字」拆成兩層,紅利就在這裡兌現:貴的迭代與便宜的迭代被分開了,使用者可以放心玩文字,只有真的不滿意背景時才動用生圖額度。

至於「修改背景」是怎麼運作的——先打破一個錯覺:**生圖 API 沒有記憶**。你在 ChatGPT 裡說「再亮一點」它聽得懂,是因為對話介面幫你把上一張圖帶著;直接呼叫 API 時,所謂「編輯」其實是「把上一張圖當參考圖附上 + 在 prompt 裡加一句修改要求,整張重新生成」。理解這件事,後面的程式碼就都顯而易見了。

步驟

以下程式碼取自成品 `app/js/api.js` 與 `app/js/main.js`(有簡化,邏輯一致)。

1. edit 模式的請求長什麼樣

跟模組 5 的生成請求只差兩件事:prompt 多一行「修改要求」、參考圖欄位帶上一張背景。
`app/js/api.js` 的 `submitWallpaper` 是這樣組的:

```
// 原始 prompt 照舊用同一個模板組(風格詞、構圖約束、禁文字鐵律都還在)
let prompt = imagePrompt(req.goal, req.subGoals, presetById(req.styleId).words, req.orientation);

// 修改模式:把使用者那句話接在後面,明說「套用在上一版設計上」
if (req.editInstruction) {
  prompt += `\nRevision request from the user (apply it to the previous design): ${req.editInstruct
ion}`;
}

const payload = {
  prompt,
  size: req.orientation === 'portrait' ? '1024x1536' : '1536x1024',
};
// 上一張背景圖放進參考圖欄位(純 base64,去掉 data: 前綴)
if (req.referenceImagesBase64 && req.referenceImagesBase64.length) {
  payload.reference_images_base64 = req.referenceImagesBase64;
}
```

注意:**原始 prompt 一個字都沒少**。如果只送「亮一點」三個字,模型就失去了風格與構圖的所有上下文,生出來的會是另一張不相干的圖。

2. 把上一張背景帶上(縮圖再上傳)

上一張背景在生成成功時已經存進 IndexedDB(模組 5),修改時撈出來、縮小、轉 base64:

```
// main.js:editInstruction 有值 = 修改模式
if (editInstruction && bgRecord) {
  referenceImagesBase64.push(
    await blobToBase64(await resizeImageBlob(bgRecord.blob, 1280)),
  );
}
```

兩個工具函式都在上一章的 `wallpaper.js` 裡:`resizeImageBlob` 把長邊縮到 1280(原圖直接 base64 會把請求撐爆,伺服器有 20MB 上限),`blobToBase64` 轉完會把 `` 前綴去掉——上游 API 要的是純 base64。

送出之後走哪條路,跟模組 5 的三層一樣由 Worker 決定:主路(OpenAI)上,帶參考圖的請求會改打官方的 `/v1/images/edits` 端點(文生圖才是 `/v1/images/generations`),一樣同步回圖;自架備援路徑才是同一個 job 端點、同一套輪詢、同一條進度條。對 Worker 來說「生成」與「修改」是同一件事——差別只在有沒有參考圖。

3. 文字迭代零成本:只重繪,不重生

這層幾乎不用寫新程式碼——因為背景圖(`bgBitmap`)還在手上,任何文字或配色改動只要重跑一次上一章的 `renderWallpaper`:

```
function rerenderWallpaper() {
  renderWallpaper(wallCanvas, {
    bgBitmap,                       // 背景不變,不重新生圖
    cells: state.cells,
    actions: state.actions,
    styleId: state.wallpaper.styleId,
    orientation: state.wallpaper.orientation,
    mode: wallpaperMode,
  });
}
```

成品在背景生成完成時,直接把這條規則講給使用者聽:

> 背景完成。不滿意可以輸入修改指示重生背景;改文字或換風格則即時重繪、不用重生。

這句話就是本章的 UX 核心:把「貴的路」和「免費的路」明白地標出來,使用者自然會把額度花在刀口上。

prompt 是可攜帶的資產:降級設計

接下來這節是本章真正想教的行銷頁設計觀。

想一個問題:生圖服務在排隊(`queue_full`)、站方金鑰沒設(`not_configured`)、或今天額度用完了——使用者walk away,漏斗就斷在這裡。大多數網站的答案是一句「請稍後再試」,等於送客。

但回頭看第 1 步:我們手上其實握著一份完整的、品質很好的生圖 prompt——風格詞、構圖約束、主題意象,全是現成的。這份 prompt 不是只有自家後端能用,**貼到 ChatGPT 或 Gemini 一樣能生**,而使用者多半本來就有這些帳號。

所以降級路徑是兩個純前端按鈕,零後端依賴。成品把它們收在步驟四一個「服務忙線或想自己生?」的摺疊面板(`<details>`)裡,生圖出錯時自動展開:

(a) 複製生圖 prompt

用前端那份相同的 `imagePrompt` 模板(`app/js/prompts.js` 與 Worker 端逐字同步,BYO 直連時走的就是它)組出完整 prompt,補一句長寬比提示(外面的工具不吃 `size: '1024x1536'` 這種 API 參數,要用人話講),一鍵複製:

```
import { imagePrompt } from './prompts.js';
import { presetById } from './styles-presets.js';

function buildPortablePrompt(goal, subGoals, styleId, orientation, editInstruction) {
  let p = imagePrompt(goal, subGoals, presetById(styleId).words, orientation);
  if (editInstruction) {
    p += `\nRevision request from the user (apply it to the previous design): ${editInstruction}`;
  }
  // API 參數換成人話:長寬比提示
  p += orientation === 'portrait'
    ? '\nAspect ratio 2:3, vertical phone wallpaper.'
    : '\nAspect ratio 3:2, horizontal desktop wallpaper.';
  return p;
}

copyBtn.addEventListener('click', async () => {
  await navigator.clipboard.writeText(buildPortablePrompt(
    core.text, subGoals, state.wallpaper.styleId, state.wallpaper.orientation, null,
  ));
  showMsg(bgMsgEl, '已複製。貼到 ChatGPT 或 Gemini 生成,參考圖自己附上,生好的圖用下方「匯入背景圖」帶回來。',
'ok');
});
```

引導文案要把三步講完:貼過去生 → 下載圖 → 帶回來。使用者上傳過的參考照片在他自己手上,叫他直接附給 ChatGPT 即可——這條路上連參考圖都不用經過我們。

(b) 匯入背景圖

一個檔案上傳框,把使用者自己生好的圖收回來當背景:

```
importInput.addEventListener('change', async () => {
  const file = importInput.files[0];
  if (!file) return;
  try {
    bgBitmap = await createImageBitmap(file);
    // 跟 AI 生成的背景同等對待:存進 IndexedDB,標記來源是手動匯入
    bgRecord = {
      blob: file,
      meta: { source: 'manual', styleId: state.wallpaper.styleId,
              orientation: state.wallpaper.orientation },
      createdAt: new Date().toISOString(),
    };
    bgRecord.id = await idbPut('wallpapers', bgRecord);
    rerenderWallpaper(); // cover 裁切吃任何比例,直接重繪
  } catch {
    showMsg(bgMsgEl, '這張圖讀不進來,請換一張(支援一般圖片格式)。');
  }
});
```

不用檢查圖的長寬比——上一章的 `coverRect` 本來就吃任何比例,置中裁滿。匯入後的圖跟 AI 生的背景走完全相同的後續流程:文字合成、即時重繪、下載 PNG,連「修改背景」都能接著用(它會被當成下一輪的參考圖)。

最後一塊拼圖:**錯誤訊息要把這條路標成主要出路**。`not_configured`、`queue_full`、額度用完、生成逾時——這些錯誤面板上,第一順位的建議不是「稍後再試」,而是「複製生圖 prompt 自己生,生好匯入回來」。服務掛掉的瞬間,頁面從「壞掉的產品」變成「換一條路的產品」。

這就是「prompt 是可攜帶的資產」的意思:你花心思調出來的 prompt 模板,價值不綁死在自家後端上。後端是加速器(免操作、有額度管理),不是必經之路;漏斗的最後一步——下載桌布、留 email——永遠走得到。

給 AI 的 prompt 範本

在模組 6 成品的基礎上加功能,貼給 ChatGPT 或 Claude:

```
我有一個九宮格桌布合成頁(canvas 把文字疊在背景圖上),請加三個功能:

【修改背景】背景生成成功後,顯示一個輸入框加「修改背景」按鈕。按下時:
1. 沿用原本那份完整的生圖 prompt,在最後面加一行:
   "Revision request from the user (apply it to the previous design): " +
   使用者輸入的修改句
2. 把目前這張背景圖縮到長邊 1280 的 JPEG、轉 base64(去掉 data: 前綴),
   放進請求的 reference_images_base64 陣列
3. 之後走跟初次生成完全相同的送出與輪詢流程
另外:改九宮格文字或換風格時只重繪 canvas,絕不重新生圖。

【複製生圖 prompt】一顆按鈕,把完整生圖 prompt(含風格描述與構圖約束)
組出來,結尾補一行長寬比提示(直式 "Aspect ratio 2:3, vertical phone
wallpaper." / 橫式 "Aspect ratio 3:2, horizontal desktop wallpaper."),
用 navigator.clipboard.writeText 複製;若 clipboard 不可用(非 https 或
file:// 開啟),退而把 prompt 顯示在一個唯讀 textarea 讓使用者自己全選
複製。按鈕旁附引導文案:「貼到 ChatGPT 或 Gemini 生成,參考圖自己附,
生好的圖用『匯入背景圖』帶回來」。

【匯入背景圖】一個檔案上傳框,選圖後 createImageBitmap 當作背景,
cover 置中裁切鋪滿畫布重繪(任何長寬比都收),後續文字合成、下載 PNG
照常運作。

【錯誤處理】生圖服務回傳「未配置」「排隊滿」「額度用完」「逾時」時,
錯誤訊息的第一建議是「複製生圖 prompt 自己生成,再匯入回來」,
而不是只說稍後再試。

【驗收】
1. 修改背景後,新圖看得出保留原構圖氛圍且套用了修改句。
2. 改文字後畫面瞬間更新,Network 面板沒有任何生圖請求。
3. 完全離線:複製 prompt(或 textarea 後備)、匯入圖、改字重繪、
   下載 PNG 全部可用。
4. 匯入一張正方形的圖配直式畫布:鋪滿、置中裁切、不變形。
```

常見坑

1. **以為 AI 記得上一張圖**:對話式工具寵壞了我們。API 是無狀態的——不附參考圖的「修改」等於完全重抽,跟上一張一點關係都沒有。
2. **只送修改句、丟掉原 prompt**:風格詞與構圖約束(含「不准畫字」鐵律)全在原 prompt 裡,修改句是追加,不是取代。
3. **參考圖沒縮就上傳**:手機原圖好幾 MB,三張 base64 直接撐爆請求(伺服器 20MB 上限)。先縮長邊 1280 再轉。
4. **base64 忘了去前綴**:`reference_images_base64` 要的是純 base64,`` 開頭的字串會被上游拒絕。
5. **clipboard API 默默失敗**:`navigator.clipboard` 在非 https 頁面或拿檔案直接開(file://)時可能不可用。降級路徑自己也要有降級:備一個唯讀 textarea。

6. **降級路徑藏太深**:做了複製與匯入,卻只放在頁面角落,錯誤訊息照樣只說「稍後再試」——使用者看不到出口等於沒有出口。出錯的當下就要把路指出來。
7. **匯入的圖比例不合就拒收**:不必。cover 裁切吃任何比例;頂多在引導文案提醒「跟 ChatGPT 要 2:3 直式,裁掉的部分會少一點」。

對照成品

- `app/`:步驟四的完整體驗——「AI 修改背景」輸入框、「複製生圖 prompt」與「匯入背景圖」都在,試著故意不設金鑰走一次降級路徑,看錯誤面板怎麼引導你。
- `demos/03-wallpaper/`:本章的合成底盤(模組 6 的成品),匯入背景圖後接著玩文字重繪,就能體會「貴的迭代」與「免費的迭代」差在哪。

第 08 章 — 防刷與額度:你的金鑰背後是真鈔

學完能做什麼

- 給公開的 AI 端點裝上四層防線:Turnstile 無感驗證、每日配額、同 IP 同時只跑一張的 in-flight lock、先扣後跑加失敗退款。
- 看懂「併發繞過」這種抽象風險的具體長相,並用 D1 的一條原子 SQL(`ON CONFLICT ... RETURNING`)把計數做到繞不過。
- 分清楚哪些是真防線(配額層)、哪些不是(CORS),以及金鑰還沒備齊的部署過渡期怎麼設計 fail-open。

核心觀念:免費給人玩,不等於免費給腳本玩

這個產品對訪客全免費:AI 補格、AI 生桌布,點了就跑。但每一次點擊背後都是真實成本——`/fill` 燒 Groq 額度;`/image` 更貴:主路打 OpenAI Images API,每生一張就是一筆按張計費的 API 帳單;沒設 OpenAI 金鑰時退到作者自架的 codex-image-service,燒的是作者個人的 ChatGPT 訂閱額度、實質併發只有一到二張。不管走哪條路,每次點擊都是真錢——這套防線同時護著 OpenAI 帳單與自架服務的額度。你照本課自己部署一套時,燒的就是你的錢。而 Worker 端點是公開網址,任何人寫一個十行的迴圈就能整夜打它。防線的目標不是把門鎖死,是讓「正常人的一天」剛好夠用、讓「腳本的一秒鐘」必然碰壁。

本章還保留了一個誠實的工程教訓。第一版設計裡,配額計數打算用 Cloudflare KV——「讀出來、加一、寫回去」,看起來會動,單人測試也都過。但在上線前的對抗式 review 被抓出來:兩個請求同時讀到同一個數字,就會雙雙通過檢查,防線在併發下是擺設。修法不是加更多檢查,是把「檢查與加一」交給資料庫的一條原子 SQL。我們把這個發現原樣寫進教材,因為「看起來會動」和「攻擊下仍然正確」之間的差距,正是這一章要教的東西。

步驟

1. 先盤點你要保護的東西

端點	燒什麼	每日配額(每 IP)
`POST /fill`(AI 補格/展開)	Groq API 額度	12 次
`POST /image`(AI 桌布背景)	OpenAI Images API(按張計費);備援自架服務燒 ChatGPT 訂閱	6 次
`POST /signup`(email 留資)	D1 寫入(便宜,但會被灌垃圾)	10 次

數字放在 `wrangler.toml` 的 `[vars]`(`IMG_PER_DAY`、`FILL_PER_DAY`),要調不用改程式。原則:最貴的資源配額壓最低。

2. 第一層:Turnstile 無感驗證

Turnstile 是 Cloudflare 的免費人機驗證,多數真人完全無感(不用點紅綠燈),但腳本拿不到 token。前端每次呼叫前在背景渲染一個隱形 widget 取得一次性 token,隨請求送出;Worker 拿 token 向 Cloudflare 驗證:

```
// worker-deploy/src/lib/turnstile.js
const VERIFY_URL = "https://challenges.cloudflare.com/turnstile/v0/siteverify";

export async function verifyTurnstile(env, token, ip) {
  if (!env.TURNSTILE_SECRET) {
    console.warn("TURNSTILE_SECRET 未設,跳過 Turnstile 驗證");
    return true; // fail-open:過渡期不擋人(見下方說明)
  }
  if (!token) return false;
  const form = new URLSearchParams();
  form.set("secret", env.TURNSTILE_SECRET);
  form.set("response", token);
  if (ip && ip !== "unknown") form.set("remoteip", ip);
  const res = await fetch(VERIFY_URL, {
    method: "POST",
    headers: { "Content-Type": "application/x-www-form-urlencoded" },
    body: form.toString(),
  });
  return (await res.json()).success === true;
}
```

注意第一段:**secret 沒設就放行,只留警告**。這是刻意的部署過渡設計——Turnstile 要前端 sitekey 和 Worker secret 兩邊同時就位才能動,部署總有先後;如果 secret 未設就回 403,整個站會在過渡期全掛。fail-open 的代價是「還沒開驗證的期間少一層防線」,但後面還有三層,而且每一層都比 Turnstile 更硬。反過來的情境也要處理:站方已開驗證、自部署的前端卻沒填 sitekey,使用者會收到 403——本課前端的錯誤文案會明講「此頁缺 sitekey 設定」,而不是讓人對著「驗證失敗」發呆。

3. 第二層:每日配額——以及那個 KV 教訓

先看會被繞過的版本(第一版設計,沒上線):

```
// 反面教材:KV 讀-改-寫,併發下是擺設
const used = Number(await env.QUOTA.get(key)) || 0; // 請求 A、B 同時讀到 2
if (used >= limit) return tooMany();                // 兩個都通過檢查
await env.QUOTA.put(key, String(used + 1));         // 兩個都寫 3:多跑一次,少算一次
```

「讀出來、檢查、寫回去」是三個動作,中間沒有任何機制阻止別人插隊。攻擊者只要同時發 20 個請求,大部分都會在「還沒寫回」的窗口通過檢查。何況 Cloudflare KV 是最終一致的全球儲存,連「寫完馬上讀」都不保證讀到新值——它本來就不是拿來做計數器的。

正解:換 D1(Cloudflare 的免費 SQLite),把「檢查+加一」壓進一條原子 SQL:

```
CREATE TABLE IF NOT EXISTS quota_counts (
  day TEXT NOT NULL, ip TEXT NOT NULL, kind TEXT NOT NULL,
  count INTEGER NOT NULL DEFAULT 0, PRIMARY KEY (day, ip, kind));
```

```
// worker-deploy/src/lib/quota.js — 先扣後跑:單條 upsert 原子扣額
export async function takeQuota(db, { kind, ip, limit, day }) {
  const row = await db.prepare(
    `INSERT INTO quota_counts (day, ip, kind, count) VALUES (?, ?, ?, 1)
     ON CONFLICT(day, ip, kind) DO UPDATE SET count = count + 1
     WHERE count < ? RETURNING count`,
  ).bind(day, ip, kind, limit).first();
  if (!row) return { ok: false };
  return { ok: true, count: row.count };
}
```

逐句讀:第一次見到這組(日期、IP、種類)就插入 `count=1`;已存在就走 `DO UPDATE` 加一,但 `WHERE count < ?` 擋住超限;`RETURNING` 的意思是「真的有插入或更新才回一列」——**沒有列回來,就是超限**。檢查與加一發生在同一條 SQL 裡,資料庫保證兩個併發請求必有先後,誰都插不了隊。超限時回 `429` 並附 `resetAt`(台北時間明日 00:00),前端能告訴使用者幾點回來,以及引導 BYO(自帶金鑰)出路。

4. 第三層:in-flight lock——同一時間只跑一張

生圖一張要跑幾十秒到三分鐘。就算每日配額是 3,有人同時送 3 張,主路是三筆同時開跑的 OpenAI 帳單、備援則直接塞滿自架服務的隊列,都在排擠其他訪客。所以同一個 IP 同時只能有一個生圖 job:

```
// 先清過期鎖,再搶鎖;RETURNING 無列 = 沒搶到(回 409 in_flight)
export async function acquireLock(db, ip, jobId, now = Date.now()) {
  await db.prepare("DELETE FROM locks WHERE expires_at < ?").bind(now).run();
  const row = await db.prepare(
    `INSERT INTO locks (ip, job_id, expires_at) VALUES (?, ?, ?)
     ON CONFLICT(ip) DO NOTHING RETURNING ip`,
  ).bind(ip, jobId, now + 900_000).first();
  return !!row;
}
```

同一招式:用 `ON CONFLICT ... RETURNING` 把「檢查+佔位」做成原子。兩個細節:鎖一定要有期限(這裡 15 分鐘),不然使用者關掉網頁,他的 IP 就永遠被鎖死;前端收到 `409` 時文案要講人話——「另一個生成正在進行(可能是你稍早送出的請求),最多 15 分鐘後自動解鎖」。

5. 第四層:先扣後跑,失敗退款

扣額度和呼叫上游,哪個先?**一定先扣**。如果「跑完才扣」,那麼在跑的這幾十秒到幾分鐘裡,同 IP 的請求個個都是「還沒扣」狀態,配額形同虛設。`/image` 的完整順序:

```
// worker-deploy/src/index.js(節錄)
if (!(await verifyTurnstile(env, body.turnstileToken, ip)))   // 1. Turnstile
  return json({ error: "turnstile_failed" }, 403, cors);
if (!(await acquireLock(env.DB, ip, "pending")))               // 2. lock
  return json({ error: "in_flight" }, 409, cors);
const quota = await takeQuota(env.DB, { kind: "img", ip, limit, day }); // 3. 先扣
if (!quota.ok) {
  await releaseLock(env.DB, ip);
  return json({ error: "quota_exceeded", resetAt: taipeiResetAt() }, 429, cors);
}
// 4. 扣完才呼叫上游;送單失敗 → 退 1 + 解鎖
```

先扣後跑對使用者不公平的地方,用「失敗退款」補回來:主路 OpenAI 回錯或撞上 90 秒截斷、備援送單失敗、job 失敗、或 job 超過 660 秒還沒結果(視同逾時),Worker 都自動退還那 1 次額度並解鎖。「不退」只有一種情況——送出後使用者自己放棄或斷線,因為 Worker 無從分辨那和正常等待的差別。

退款有個陷阱:前端每 5 秒輪詢一次 job 狀態,失敗狀態會被讀到很多次,如果每次都退 1,等於失敗一次反而送額度。所以退款必須冪等:

```
// 退款冪等:搶到 refunded 0→1 這個翻轉的那一次,才真的退
const marked = await db.prepare(
  "UPDATE jobs SET refunded = 1 WHERE id = ? AND refunded = 0 RETURNING id",
).bind(jobId).first();
if (!marked) return false; // 已退過,跳過
await refundQuota(db, { kind: "img", ip: job.ip, day: job.day }); // count-1,不低於 0
await releaseLock(db, job.ip);
```

又是 `RETURNING`:第一次呼叫翻轉 `refunded` 拿到列、執行退款;之後的呼叫條件不成立、拿不到列、直接跳過。三層原子操作,同一個句型。

6. CORS 不是濫用邊界

Worker 的 CORS 設定只允許 `yazelin.github.io` 和 localhost,你可能以為這就擋掉了別站盜用。要想清楚:**CORS 是瀏覽器的自律規範,只有瀏覽器會遵守**。用 curl 直打,根本沒有 Origin 這回事:

```
curl -s -X POST https://goal-grid.yazelinj303.workers.dev/fill \
  -H 'Content-Type: application/json' \
  -d '{"mode":"core","cells":[null,null,null,null,"學好英文",null,null,null,null]}'
# 不經瀏覽器、沒有 Origin,請求照樣進來 —— 接住它的是配額層,不是 CORS
```

CORS 的功能是保護「使用者在別的網站上不被偷偷代打」,不是保護你的額度。防濫用是配額層的事,這就是為什麼四層防線一層都不能省。

7. 驗收

- 開兩個分頁同時按「生成 AI 背景」:第二個要立刻收到「另一個生成正在進行」(409),而不是兩張都開跑。
- 同一天生圖 3 次後再按:要看到「今日 AI 額度已用完,將於(明日 00:00)重置」與 BYO 引導,而不是無聲失敗。
- 用上面的 curl 直打一次:確認回應來自 Worker 的配額/驗證邏輯(403 或正常回應),體會「CORS 擋不住它」。
- 自部署版還沒設 `TURNSTILE_SECRET` 時,功能照常可用,Worker log 出現跳過驗證的警告——fail-open 生效。

給 AI 的 prompt 範本

```
幫我的 Cloudflare Worker(已有 /fill 與 /image 兩個 AI 端點)加上四層防刷防線,
用 D1 做配額,不要用 KV 或記憶體變數計數。

【資料表】在 schema.sql 加三張表:
quota_counts(day, ip, kind, count,主鍵 day+ip+kind)
locks(ip 主鍵, job_id, expires_at)
jobs(id 主鍵, ip, day, refunded 預設 0, created_at)

【第一層 Turnstile】每個 POST 帶 turnstileToken,Worker 向
https://challenges.cloudflare.com/turnstile/v0/siteverify 驗證。
Secret 名稱 TURNSTILE_SECRET;未設定時跳過驗證並 console.warn(部署過渡期 fail-open)。

【第二層 每日配額】每 IP 每日 /image 6 次、/fill 60 次(數字放 env vars,依漏斗轉換需要調整 — 額度太緊,使用者連桌布都生不出來,就更不會留 email)。
扣額必須是一條原子 SQL:
INSERT INTO quota_counts (day, ip, kind, count) VALUES (?, ?, ?, 1)
ON CONFLICT(day, ip, kind) DO UPDATE SET count = count + 1
WHERE count < ? RETURNING count
沒有列回傳就是超限,回 429 加 resetAt(台北時間明日 00:00 的 ISO 字串)。
day 用台北時區的 YYYYMMDD。禁止「先 SELECT 再 UPDATE」的寫法。

【第三層 in-flight lock】同 IP 同時只能有一個 /image job:
先 DELETE 過期鎖,再 INSERT ... ON CONFLICT(ip) DO NOTHING RETURNING ip,
無列回 409 {error:"in_flight"};鎖的 expires_at = 現在 + 900 秒。

【第四層 先扣後跑+退款】順序固定:Turnstile → lock → 扣額 → 呼叫上游。
上游送單失敗、job 失敗、job 超過 660 秒未完成:退 1 次額度(不低於 0)並解鎖。
退款要冪等:UPDATE jobs SET refunded=1 WHERE id=? AND refunded=0 RETURNING id,
有列才執行退款,輪詢重複讀到失敗狀態不能重複退。

【驗收】寫一個用 Map 模擬 D1 的最小單元測試,證明:
1. 同 IP 連續扣到上限後被拒
2. 兩個併發扣額不會超過上限
3. 退款呼叫兩次只實際退一次
4. lock 互斥、過期後可重新取得
全部通過才算完成,並附 curl 指令讓我手動驗 429 與 409。
```

前端這側的配額體驗(429 顯示重置時間、409 講人話、退款後提示可重試),可以另外丟一句:

```
幫我的前端把 API 錯誤翻成人話:429 顯示「今日額度已用完,將於 {resetAt 台北時間} 重置」
並附自帶金鑰的引導;409 顯示「另一個生成正在進行,最多 15 分鐘後自動解鎖」;
退款類錯誤(逾時、生成失敗)明說「額度已退還,請點重試」。不要只顯示錯誤代碼。
```

常見坑

1. **任何「讀-改-寫」三步的計數都有併發洞**:不只 KV,存在 Worker 記憶體裡的 Map 也一樣(而且 Worker 會多實例、會重啟歸零,記憶體限流只能當 best-effort 的第一層)。計數一律交給資料庫的原子操作。
2. **以為 CORS 能防盜用**:curl 和任何後端腳本都不理 CORS。CORS 只管瀏覽器,額度要靠配額層。
3. **退款不冪等**:輪詢會重複讀到失敗狀態,沒做 `refunded` 翻轉檢查就會重複退款,失敗反而變成送額度。
4. **lock 沒有期限**:使用者關頁斷線,IP 永遠被鎖。鎖要帶 `expires_at`,取鎖前先清過期的。
5. **「跑完才扣」**:長任務跑著的幾分鐘裡,後續請求全都在「還沒扣」的窗口通過。先扣後跑,失敗再退。
6. **配額日期沒帶時區**:用 UTC 切日,台北使用者會在早上 8 點看到「額度重置」,訊息全亂。本課用台北時區的 YYYYMMDD 當 key。
7. **Turnstile 上線順序想錯**:secret 與 sitekey 要兩邊就位;secret 未設就硬擋會讓過渡期全站 403。fail-open 過渡,上線後再補驗證。
8. **429 文案冷冰冰**:「額度用完」就結束,訪客流失。要給重置時間,並給出路(自帶金鑰、或第 07 章的「複製 prompt 自己去生」降級路徑)。

對照成品

- 互動體驗:demos/04-quota/ ——用前端模擬配額遞減、429、409 與退款重試的完整 UX,不用真的燒額度就能玩到每一種狀態。
- 真實實作:`worker-deploy/src/lib/quota.js` 與 `worker-deploy/src/index.js`(在 dev repo,不隨教材發佈;本章程式碼塊皆直接取自該處)。線上的 完整版 App 跑的就是這套四層防線。

第 09 章 — Email 收集與後台:留下名單,當場兌現承諾

學完能做什麼

- 在成品的最後一步加上 email 收集:honeypot 擋機器人、UNIQUE 去重不報錯、誘因當場兌現(不開空頭支票)。
- 做一個只有你進得去的後台 `admin.html`:Bearer token 驗證、名單表格、一鍵匯出 CSV 進任何 EDM 工具。
- 知道後台 token 為什麼存 sessionStorage 而不是 localStorage——共用 origin 的 XSS 爆炸半徑。

核心觀念:免費價值先給,留資換加值,承諾當場兌現

前課模組 9 講過:社群觸及是租來的,email 名單才是自己的資產。本章把那套「收集、儲存、後台」放進完整產品的漏斗位置,而位置決定成敗——桌布 PNG 直接給,**不設牆**;email 表單放在下載旁邊,用「留下 email,立即拿 30 天行動追蹤模板」交換。訪客已經拿到主要價值,留資是加值交易,不是門票,轉換才不會帶著怨氣。

「立即拿」三個字是設計,不是文案修辭。第一版產品最容易在「我之後會寄給你」這句話上失信:寄信要串 EDM 服務、會進垃圾桶、會晚到。所以這裡的誘因是一個靜態網頁(`tracker.html`),註冊成功的回應裡直接帶連結,當場點開、當場可印——承諾在同一個畫面裡兌現,不依賴任何寄信系統。名單先收著,真正的群發等你哪天要做再匯出去。

步驟

1. 表單與 honeypot

表單只收 email(欄位越少轉換越高),外加一個真人永遠看不到的「蜜罐」欄位:

```
<!-- app/index.html 步驟五 -->
<form class="signup-form" id="signup-form" novalidate>
  <input type="email" name="email" id="signup-email" autocomplete="email"
         placeholder="you@example.com" aria-label="Email">
  <input type="text" name="company" id="signup-company" class="hp-field"
         tabindex="-1" autocomplete="off" aria-hidden="true">
  <button class="btn btn-primary" type="submit" id="signup-submit">送出拿模板</button>
</form>
```

```
/* honeypot:移出視野但仍在 DOM(真人看不到,機器人照填) */
.signup-form .hp-field {
  position: absolute; left: -9999px; top: 0;
  width: 1px; height: 1px;
  opacity: 0; pointer-events: none;
}
```

笨的灌名單機器人會把頁面上每個欄位都填滿。`company` 有值,就當機器人——但 Worker 不回錯誤,而是**假裝成功、不寫入**,讓機器人以為得逞,不會回頭換手法。

2. Worker 端:/signup 的完整防線

```
// worker-deploy/src/lib/signup.js(節錄)
export async function handleSignup({ db, env, ip, body }) {
  // 第一層:每 IP 每分鐘 5 次(記憶體 Map,best-effort —— 見第 08 章的提醒)
  if (limited(ip)) return { status: 429, body: { error: "rate_limited" } };

  // 第二層:Turnstile(有設 secret 才驗,與其他端點同一套)
  if (!(await verifyTurnstile(env, body.turnstileToken, ip))) {
    return { status: 403, body: { error: "turnstile_failed" } };
  }

  // honeypot:有值就當機器人,假裝成功不寫入
  if (body.company) {
    return { status: 200, body: { ok: true, gift: env.GIFT_URL } };
  }

  // 正規化:trim + 轉小寫,Abc@Mail.com 和 abc@mail.com 才會被當同一人
  const email = String(body.email || "").trim().toLowerCase();
  const name = String(body.name || "").trim().slice(0, 60);
  if (!EMAIL_RE.test(email) || email.length > 120) {
    return { status: 400, body: { error: "bad_email" } };
  }

  // 第三層:D1 每日配額 10 次/IP(原子扣額,第 08 章的同一個 takeQuota)
  const quota = await takeQuota(db, { kind: "signup", ip, limit: 10, day: taipeiDay() });
  if (!quota.ok) return { status: 429, body: { error: "rate_limited" } };

  try {
    await db.prepare("INSERT INTO signups (email, name, created_at, ip) VALUES (?, ?, ?, ?)")
      .bind(email, name || null, now, ip).run();
    return { status: 200, body: { ok: true, gift: env.GIFT_URL } };
  } catch (e) {
    if (String(e).includes("UNIQUE")) {
      return { status: 200, body: { ok: true, already: true, gift: env.GIFT_URL } };
    }
    return { status: 500, body: { error: "db_failed" } };
  }
}
```

留意分鐘限流和每日配額是兩層:記憶體 Map 擋連點(快、零成本,但 Worker 多實例或重啟就歸零),D1 配額才是繞不過的那層——這正是第 08 章的教訓在便宜端點上的應用。

3. 去重:重複送出不是錯誤

資料表給 email 上 UNIQUE 約束(`schema.sql`:`email TEXT NOT NULL UNIQUE`),同一人按三次也只會有一筆。重點在回應方式:UNIQUE 衝突**不回錯誤**,照樣回 `200` 加 gift 連結,只多一個 `already: true` 讓前端把文案換成更誠實的一句:

```
// app/js/signup.js — 成功回應轉文案
message: already
  ? '你已經在名單上了 — 模板連結照樣再給你一次:'
  : '完成,Email 已登記。這是你的 30 天行動追蹤模板:',
```

為什麼不報錯?第一,體驗:使用者換了台電腦回來再填,看到「此 email 已存在」只會困惑,他要的是模板,再給一次就好。第二,不從狀態碼洩漏:外人無法用「會不會報錯」來探測某個 email 在不在你的名單上。要誠實補一句:回應裡的 `already` 欄位本身還是可被探測的——這份名單只是「誰領了模板」,取捨偏向體驗;若你收的是敏感名單(客戶、病患、會員),連 `already` 都拿掉,讓兩種回應完全一致。

4. 誘因即時兌現:tracker.html

gift 連結指向一個純靜態頁 `app/assets/tracker.html`:30 天行動追蹤表,八個子目標乘三十天勾選格,A4 橫式列印 CSS,在瀏覽器按一鍵就變白底黑字的紙本。它還有個小巧思——和 App 同源,所以能直接讀 localStorage 把使用者自己的九宮格帶進表格:

```
// app/assets/tracker.html — 從 localStorage 帶入八個子目標(沒有 state 就留空白手寫)
const STATE_KEY = 'goal-grid-state-v1';
const SUB_POSITIONS = [0, 1, 2, 3, 5, 6, 7, 8]; // 略過中央 4 = 核心目標
const state = readState();
const core = cellText(4);
if (core) coreEl.textContent = core;
```

領到的不是一張通用模板,是「印著你自己目標」的追蹤表。誘因的價值感就差在這裡。

5. 後台 admin.html:token gate

後台是一個獨立靜態頁:輸入管理密碼、向 Worker 的 `GET /list` 要名單、畫成表格。兩個前課就講過的鐵則,這裡照辦:密碼存在 Worker 的 Secret(`ADMIN_TOKEN`),永遠不出現在前端程式碼;傳輸走 `Authorization` 標頭,不放網址:

```
// app/admin.html — token 走標頭,不放進網址(避免被瀏覽器歷史 / CDN log 記下)
const r = await fetch(apiBase() + '/list', { headers: { Authorization: 'Bearer ' + t } });
```

```
// worker-deploy/src/lib/signup.js — Worker 端比對
export async function handleList({ db, env, bearer }) {
  if (!env.ADMIN_TOKEN || bearer !== env.ADMIN_TOKEN) {
    return { status: 401, body: { error: "unauthorized" } };
  }
  const { results } = await db
    .prepare("SELECT id, email, name, created_at, ip FROM signups ORDER BY id DESC").all();
  return { status: 200, body: { signups: results, count: results.length } };
}
```

6. token 存哪裡:sessionStorage,不是 localStorage

前課的後台把 token 記在 localStorage 圖個方便。本課改成 sessionStorage,原因值得搞懂:

這個站部署在 `yazelin.github.io`——GitHub Pages 上,**同一個帳號的所有專案頁共用同一個 origin**。瀏覽器的儲存以 origin 為界,意思是這個帳號下任何一個專案(包括多年前的舊 demo)只要有一個 XSS 漏洞,注入的腳本就能讀到「整個 origin」的 localStorage——包括你後台的 token。把長效憑證放 localStorage,等於讓爆炸半徑涵蓋你所有專案的歷史包袱。

sessionStorage 的差別:只活在這個分頁,分頁關了就消失。代價是每次開後台要重新輸一次密碼;換來的是被偷的窗口從「永久」縮成「這個分頁開著的期間」。後台一天開不了幾次,這筆交易很划算。同樣的理由,輸入框上方那行說明文字也直接告訴使用者「只記到這個分頁關閉為止」——安全設計講出來,就是信任感。

7. 匯出 CSV,接進 EDM 工具

名單的出口是 CSV——MailerLite、Mailchimp、Brevo 任何群發工具都吃。前端把 JSON 轉 CSV 有兩個容易踩的細節,這裡都處理了:

```
// app/admin.html — CSV(RFC 4180):每個欄位都加雙引號、引號翻倍跳脫、列尾 CRLF
function buildCsv(items) {
  const q = (v) => '"' + String(v ?? '').replace(/"/g, '""') + '"';
  const rows = [['email', 'name', 'created_at', 'ip']]
    .concat(items.map((s) => [s.email, s.name || '', s.created_at, s.ip || '']));
  return rows.map((r) => r.map(q).join(',')).join('\r\n') + '\r\n';
}
// 開頭加 UTF-8 BOM:讓 Excel 直接開啟時正確識別中文
const blob = new Blob(['\uFEFF' + buildCsv(list)], { type: 'text/csv;charset=utf-8' });
```

欄位一律包雙引號並把引號翻倍,稱呼裡出現逗號或引號才不會把表撐破;開頭的 `\uFEFF`(UTF-8 BOM)讓 Excel 雙擊開啟時中文不變亂碼。匯出後丟進 EDM 工具的「匯入聯絡人」,你的第一波 email 行銷就有對象了。

8. 驗收

- App 步驟五填一個 email 送出:畫面當場出現模板連結,點開 `tracker.html` 看到自己的子目標已帶入,按列印預覽是白底 A4。
- 同一個 email 再送一次:看到「你已經在名單上了」,連結照給,不報錯。
- 開 `admin.html` 輸入 `ADMIN_TOKEN`:剛剛那筆在表格最上面;故意輸錯密碼,要看到「密碼不對」而不是名單。
- 匯出 CSV 用 Excel 開:中文正常、欄位對齊。
- 關掉後台分頁重開:要求重新輸入密碼(sessionStorage 生效)。

給 AI 的 prompt 範本

```
幫我做一套 email 名單收集:Cloudflare Worker 端點 + 前端表單 + 管理後台,規格如下。

【資料表】D1 一張表 signups:id 自增主鍵、email(NOT NULL UNIQUE)、name、created_at、ip

【POST /signup】收 {email, name?, company?, turnstileToken?}:
- company 是 honeypot:有值就當機器人,回假成功(200 {ok:true, gift})但不寫入
- email 先 trim 再轉小寫,正規驗證格式,超過 120 字回 400 {error:"bad_email"}
- 防線三層:每 IP 每分鐘 5 次(記憶體)、Turnstile(secret 未設則跳過)、
  D1 每日配額每 IP 10 次(原子 upsert,不可用先讀後寫)
- 寫入成功回 200 {ok:true, gift:<模板網址>}
- email 重複(UNIQUE 衝突)不報錯:回 200 {ok:true, already:true, gift:同上}

【GET /list】讀 Authorization: Bearer <token> 標頭,與 Secret ADMIN_TOKEN 比對,
不對回 401;對了回 {signups:[...], count}。token 絕不接受放在網址 query。

【前端表單】email 輸入框 + 送出鈕 + honeypot 欄位(position:absolute 移出視野、
tabindex=-1、aria-hidden);成功顯示「完成,這是你的模板:」加 gift 連結,
already 時顯示「你已經在名單上了,連結照樣再給你一次」;送出中要鎖按鈕。

【後台 admin.html】單檔靜態頁,meta robots noindex:
- 密碼輸入 → 帶 Bearer 標頭打 /list → 名單畫成表格(欄位要做 HTML 跳脫)
- token 存 sessionStorage 不是 localStorage(共用 origin 的 XSS 爆炸半徑),
  並在介面註明「只記到分頁關閉為止」
- 匯出 CSV 按鈕:RFC 4180(欄位全包雙引號、引號翻倍、CRLF 列尾),
  檔案開頭加 UTF-8 BOM 讓 Excel 認得中文

【驗收】用 curl 測五種情境:正常註冊、重複註冊、honeypot、爛 email、錯 token,
每種的狀態碼與 body 都符合上面規格才算完成。
```

常見坑

1. **裸表單沒擋機器人**:公開的收集端點一定被灌垃圾。本章疊了四道:honeypot、分鐘限流、Turnstile、D1 日配額——便宜端點也值得上原子配額(第 08 章)。
2. **把重複註冊當錯誤**:體驗中斷、還從狀態碼洩漏名單成員。UNIQUE 去重 + 照樣回成功;敏感名單連 `already` 欄位都別回。

3. **誘因開空頭支票**:「之後寄給你」依賴寄信系統,第一版最常在這失信。用靜態資產讓承諾在同一個畫面兌現。
4. **管理密碼放進網址**:`?token=xxx` 會被瀏覽器歷史、CDN 日誌、referer 記下。一律走 `Authorization` 標頭。
5. **token 存 localStorage**:在 `github.io` 這種共用 origin 上,任何一個舊專案的 XSS 都能撈走它。用 sessionStorage 縮小爆炸半徑;名單升級成真實客戶個資時,改用前課模組 9 的 Cloudflare Access 實名門禁。
6. **CSV 沒處理跳脫與 BOM**:稱呼裡一個逗號就讓整列錯位;沒有 BOM,Excel 開起來中文全是亂碼。
7. **個資責任**:收了 email 就有保管責任——頁面寫清楚用途、不需要的欄位不收、名單不外流;正式營運要補退訂機制(前課模組 9 講過,這裡同樣適用)。

對照成品

- 互動體驗:demos/05-email/ ——表單 + honeypot 的單檔版本,可指向真 Worker 實際送一筆。
- 完整版:App 步驟五(送出後當場拿模板)、admin.html(輸入管理密碼看名單、匯 CSV)、tracker.html(30 天行動追蹤模板,會帶入你的九宮格)。
- Worker 端原始碼:`worker-deploy/src/lib/signup.js` 與 `schema.sql`(在 dev repo,不隨教材發佈;本章程式碼塊取自該處)。

第 10 章 — 進階 9×9 與發佈:展開 81 格,然後讓全世界(和 AI)看見

學完能做什麼

- 把 3×3 一鍵展開成大谷正宗的 81 格:AI 對八個子目標序列展開、有進度、失敗可從中斷點續跑。
- 做出 9×9 桌布版型:中央 3×3 放大當視覺主體、81 全圖縮小擺旁邊,手機上依然可讀。
- 用 publish.sh 把私有工作 repo 發佈成公開課程站,配齊 llms.txt、sitemap、JSON-LD、OG 圖——讓搜尋引擎和 AI 都讀得懂你。

核心觀念:81 格的勸退門檻,AI 把它變成一鍵

大谷翔平那張目標達成表流傳了十幾年,看過的人成千上萬,真正填完 81 格的人極少。不是方法不好,是門檻設計使然:中央 3×3 填完正有成就感,接著要為八個子目標**各想八個具體行動**——64 個空格瞪著你,多數人在這裡合上筆記本。這正是 AI 該站的位置:不是替你想得更好,是把「64 個空格」這個心理門檻變成一鍵,生出一版夠好的草稿,你只負責改與刪。改一個字的門檻,遠低於想一個詞。

本章的後半是發佈。做完的東西沒被看見等於沒做(前課模組 10 的開場白,在這裡依然成立),而這門課自己的發佈流程就是教材:**dev 私有 repo 是工作檯,公開 repo 是櫥窗**——設計文件、驗證截圖、Worker 原始碼、管理 token 全留在工作檯,一支 `publish.sh` 只把櫥窗該有的東西搬出去。分離的好處不只安全,還有自由:工作檯可以亂,櫥窗永遠乾淨。

步驟

1. 資料模型:actions 是 8×8

3×3 的九格存在 `state.cells`(第 02 章);展開後的 64 個行動存在 `state.actions`——八列,每列對應一個子目標的八個行動,沿用「null = 留給 AI、有物件 = 使用者鎖定」的同一套模型:

```
// localStorage 'goal-grid-state-v1'
state.cells   = Array(9);  // null | {text, source:'user'|'ai'},index 4 = 核心目標
state.actions = Array(8);  // null | Array(8) of (null | {text, source})
```

2. AI 展開:每個子目標一次呼叫

Worker 的 `/fill` 端點第 03 章就建好了,展開只是它的第二種模式(`mode:'expand'`):送核心目標、一個子目標、該列已填的行動,拿回八個行動。prompt 把方法論直接寫進規則——行動要具體到能每日檢核,並拿大谷當示範:

```
// worker-deploy/src/lib/prompts.js(節錄)
規則:
1. 行動要具體可執行、最好可每日/每週檢核
   (例:大谷在「運」底下寫「撿垃圾」「打招呼」這種日常小事)。
2. 每格 2-10 個字,繁體中文(台灣用語),彼此不重複,不用標點結尾,不用 emoji。
3. 已填的原樣保留,一字不改。
```

3. 序列展開:八次呼叫,一個迴圈

「展開 81 格」按下去是八次 API 呼叫。直覺會想用 `Promise.all` 八個一起發,這裡刻意用**序列**:

```
// app/js/main.js — runExpand(節錄)
const coreGoal = state.cells[4].text;
for (let s = 0; s < 8; s++) {
  aiProgressEl.textContent = `展開中 ${s + 1}/8`;
  const existing = (state.actions[s] || Array(8).fill(null)).map((a) => (a ? a.text : null));
  if (existing.every(Boolean)) continue;          // 這個子目標已滿,不重抽
  const subGoal = state.cells[posOfSubIndex(s)].text;
  const actions = await api.expandSub(coreGoal, subGoal, existing);
  actions.forEach((text, j) => {
    if (!existing[j]) setAction(state, s, j, text, 'ai');
  });
  persist();                                      // 每完成一列就存檔
}
```

序列的三個理由,都跟長流程的體驗有關:

- **進度感**:「展開中 3/8」讓使用者知道沒當機;八個並發只能給一顆轉圈圈。
- **失敗只損一格**:第五列失敗,前四列已各自 `persist()` 進 localStorage;再按一次,`existing.every(Boolean)` 會跳過已完成的列,從斷點續跑,不重抽、不重複扣配額。
- **對上游友善**:八個請求同時打,容易撞 LLM 服務的速率限制,整批失敗。

注意展開的前提:3×3 必須先填滿(自己填或 AI 補),因為每一列展開都要拿「核心目標 + 該子目標」當脈絡。程式裡有守門,文案直接告訴使用者先做哪一步。

4. 9×9 桌布版型:主體與全圖分離

81 格直接鋪滿手機桌布,每格字只剩幾個像素——不可讀的桌布沒人會設。本課的版型把「情感主體」和「完整資訊」分開:中央 3×3 放大當視覺主角,81 全圖縮小放下方(直式)或右側(橫式),想細看時放大圖片看得到,平時鎖定畫面看到的是九個大字:

```
// app/js/wallpaper.js — layoutWallpaper(節錄)
if (mode === '9x9') {
  // 直式:3×3 中心移到 0.30H(騰出下方);橫式:3×3 置左 0.27W
  if (orientation === 'portrait') { cx = 0.5 * W; cy = 0.3 * H; }
  else { cx = 0.27 * W; cy = 0.5 * H; }
}
// 81 全圖:直式寬 0.92W、中心 0.74H;橫式寬 0.5W、中心 0.72W
```

色彩上沿用大谷表轉載圖的慣例:八個子目標各有一色,81 全圖裡每個外圍區塊的中心格用同一色呼應,視線能立刻對上「這個區塊在展開哪個子目標」:

```
// drawGrid81(節錄):外圍區塊中心格 = 子目標,同色呼應
drawCell(ctx, rect, g.radius, {
  fill: hexA(subColor, 0.32),
  border: hexA(subColor, 0.9),
  text: cellText(cells[subPos]),
  ...
});
```

文字縮放交給第 06 章寫好的 `fitText`(二分搜尋字級 + CJK 逐字斷行),81 格的字長差異再大也不會爆框。

5. 發佈:dev 私有 + 公開雙 repo,一支腳本同步

repo 有兩個:`ai-goal-grid-course-dev`(私有,日常工作都在這)和 `ai-goal-grid-course`(公開,GitHub Pages 從這裡服務)。同步靠一支三十行的腳本,值得整段讀懂:

```
#!/usr/bin/env bash
# scripts/publish.sh — 把教材從 dev repo 發佈到公開 repo
set -euo pipefail
cd "$(dirname "$0")/.."

PUB_REPO="https://github.com/yazelin/ai-goal-grid-course.git"
PUBLISH_PATHS=(index.html .nojekyll slides demos course app \
               assets sitemap.xml llms.txt robots.txt)

TMP=$(mktemp -d)
trap 'rm -rf "$TMP"' EXIT
git clone -q --depth 1 "$PUB_REPO" "$TMP/pub"

# 清空(留 .git)後重放發佈集
find "$TMP/pub" -mindepth 1 -maxdepth 1 ! -name .git -exec rm -rf {} +
for p in "${PUBLISH_PATHS[@]}"; do
  [ -e "$p" ] && cp -r "$p" "$TMP/pub/"
done
# app 的 node 測試屬開發資產,不發佈
rm -rf "$TMP/pub/app/test"
cp README-public.md "$TMP/pub/README.md"

cd "$TMP/pub"
git add -A
if git diff --cached --quiet; then
  echo "publish: 沒有變更,跳過"
else
  git commit -q -m "publish: $(date +%Y-%m-%d) 教材更新"
  git push -q origin main
  echo "publish: 已推送 $(git rev-parse --short HEAD)"
fi
```

三個設計決定:

- **白名單,不是黑名單**:`PUBLISH_PATHS` 列出「要公開什麼」。黑名單(排除 docs/、worker-deploy/…)總有一天漏掉新增的私有資料夾,白名單預設安全——新東西不會被意外公開。
- **清空重放**:先把公開 repo 清空(留 `.git`)再複製,dev 這邊改名或刪除的檔案才會同步消失,不會在公開站留殭屍頁。
- **可重複執行**:沒變更就跳過 commit。跑十次和跑一次結果相同,你永遠不用怕「再跑一次會不會壞」。

公開 repo 的 README 也是發佈品(`README-public.md` 改名而成)——dev repo 的 README 寫給自己,公開的寫給訪客,兩份各司其職。

6. AEO:讓 AI 搜尋引擎讀懂你的課

越來越多人問 ChatGPT / Perplexity 而不是 Google,你的課要讓 AI 講得出來(完整原理在前課模組 10,這裡列本課的實際配備):

llms.txt ——放在站點根目錄、專門寫給 LLM 的純文字說明書:一句話講清楚你是誰,然後把重要連結攤開:

```
# AI 互動行銷頁實戰 — 目標九宮格

> 教行銷人員把 AI 行銷頁零件整合成完整產品的實戰課程。
> 範例產品:訪客用 AI 填好「目標九宮格」……生成專屬桌布下載,
> 並以 30 天行動追蹤模板為誘因留下 email。

## 成品
- [目標九宮格 App](https://yazelin.github.io/ai-goal-grid-course/app/)
## 課程章節(course/)
- 08 防刷與額度(Turnstile + D1 原子配額)
……
```

sitemap.xml ——列出所有公開頁(首頁、app、slides、五個 demo),提交 Google Search Console。

JSON-LD(Course schema) ——課程首頁 `<head>` 裡給機器讀的身分證,AI 和 Google 都吃:

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Course",
  "name": "AI 互動行銷頁實戰 — 目標九宮格",
  "description": "教行銷人員把 AI 行銷頁零件組成完整產品的實戰課程。",
  "provider": { "@type": "Person", "name": "Yaze Lin", "url": "https://yazelin.github.io/" },
  "isAccessibleForFree": true,
  "inLanguage": "zh-Hant"
}
</script>
```

robots.txt 的誠實註記 ——GitHub Pages 專案頁(`帳號.github.io/repo名/`)的 robots.txt 與 llms.txt 放在子路徑,爬蟲實際讀的是網域根。本課的 robots.txt 開頭就註明這件事,真正的站根檔案由 `yazelin.github.io` 主 repo 管理(在它的 llms.txt 列出所有子專案)。這是前課模組 10 用檢測工具實測過的坑:AEO 分數低,先確認工具掃的是哪個 URL。

另外兩個本課一直在做、容易忘記歸進 AEO 的事:後台與模板頁都加了 `<meta name="robots" content="noindex">`(admin.html、tracker.html 不該出現在搜尋結果);關鍵資訊全用文字寫在頁面上,不藏在圖裡。

7. OG 圖:分享出去的第一印象

課程連結被貼到 LINE / FB 時長什麼樣,由 OG tags 決定。本課的 `assets/og-course.jpg` 是 1200×630、壓在 200KB 內的 AI 生成圖(深夜藍加金的九宮格意象,不含文字——中文字交給 og:title 顯示,不讓 AI 畫)。鐵則跟前課一樣:`og:image` 必須是完整 https 網址,相對路徑抓不到;改圖後用 FB Sharing Debugger 強制重抓快取。

8. 驗收

- 3×3 填滿後按「展開完整 81 格」:進度從 1/8 走到 8/8;中途斷網再按一次,從斷掉的那列續跑,已展開的列不重抽。
- 桌布步驟切到 9×9:直式是「上 3×3、下全圖」,橫式是「左 3×3、右全圖」;手機鎖定畫面上中央九格清晰可讀。
- `bash scripts/publish.sh` 跑兩次:第一次推送,第二次顯示「沒有變更,跳過」。
- 公開 repo 檢查:沒有 docs/、scripts/、worker-deploy/,沒有 `app/test/`,README 是公開版。
- 把課程網址貼給 ChatGPT 或 Claude 問「這在講什麼」:答得出是一門做目標九宮格產品的行銷課,AEO 就生效了;再貼到 LINE 看 OG 預覽卡。

給 AI 的 prompt 範本

展開 81 格與 9×9 桌布(接在你既有的九宮格 App 上):

```
我的九宮格 App 已有 3×3 填寫、AI 補格、桌布合成。幫我加「進階 9×9」:

【資料模型】localStorage state 加 actions:長度 8 的陣列,每項是 null 或
長度 8 的 (null | {text, source:'user'|'ai'});actions[i] = 子目標 i 的八個行動。

【AI 展開】「展開完整 81 格」按鈕:
- 前提:3×3 九格都有字,否則提示「先補滿空格再展開」
- 對八個子目標逐一(序列,不要並發)呼叫 AI:送核心目標、該子目標、
  該列已填的行動,要求回 JSON {"actions":[8 個字串]};已填的原樣保留
- 行動要具體可每日檢核、每格 2-10 個字、繁體中文(台灣用語)、不用 emoji
- 顯示進度「展開中 x/8」;每完成一列就存進 localStorage
- 已填滿的列跳過不重抽 ── 失敗後再按一次要能從斷點續跑

【9×9 檢視】巢狀 grid:外圈 8 區塊各 3×3,區塊中心 = 對應子目標的唯讀鏡像,
其餘 8 格可編輯寫回 actions;中央區塊 = 整張 3×3 的唯讀鏡像。
每個區塊用該子目標的代表色標示。

【9×9 桌布版型】直式(1170x2532):中央 3×3 放大置於上方(中心約 0.30 高),
81 全圖縮小置於下方(寬 0.92,中心約 0.74 高);橫式(1920x1080):3×3 置左、
81 全圖置右。81 全圖中,每個外圍區塊的中心格用對應子目標的顏色呼應。
文字用「二分搜尋字級 + 中文逐字斷行」塞進格子,最小 14px。

【驗收】展開到一半模擬失敗(斷網),再按一次能續跑;9×9 直式桌布輸出 PNG 後,
中央九格在手機螢幕上可讀。
```

發佈與 AEO 資產:

```
幫我的課程 repo 做發佈系統與 SEO/AEO 資產:

【雙 repo】私有 dev repo 是工作檯;公開 repo 只放發佈品,GitHub Pages 從它服務。

【publish.sh】bash 腳本,set -euo pipefail:
- 白名單陣列 PUBLISH_PATHS 列出要公開的路徑(index.html、course、demos、app、
  assets、sitemap.xml、llms.txt、robots.txt…),絕不用「排除法」
- 流程:clone 公開 repo 到暫存資料夾 → 清空(保留 .git)→ 複製白名單路徑 →
  刪掉 app 裡的測試資料夾 → README-public.md 改名 README.md →
  有變更才 commit push,沒變更就印「跳過」
- 可重複執行,跑幾次結果都一樣

【AEO 資產】
- llms.txt:站點一句話定位 + 成品/章節/demo 連結清單(純文字、給 LLM 讀)
- sitemap.xml:列出所有公開頁
- 課程首頁 <head> 加 JSON-LD,@type 用 Course,內容必須與頁面一致
- 後台頁與贈品頁加 meta robots noindex
- robots.txt 註明:GitHub Pages 專案頁的 robots/llms 在子路徑,
  爬蟲讀的是網域根,真正站根檔案由主 repo 管理

【OG】og:title / og:description / og:image(1200x630、完整 https 網址、200KB 內)

【驗收】publish.sh 連跑兩次,第二次顯示無變更;公開 repo 裡 grep 不到
dev 專用資料夾;把首頁網址貼給 AI 問「這在講什麼」,要答得出課程主題。
```

常見坑

1. **八個展開請求一起發**:撞速率限制整批失敗、沒有進度感、失敗全毀。序列 + 逐列存檔 + 已滿跳過,才有「斷點續跑」。
2. **展開前沒守門**:3×3 還有空格就展開,AI 拿不到脈絡,生出來的行動空泛。先補滿再展開,程式擋 + 文案引導。
3. **81 格直接鋪滿桌布**:手機上每格字小到不可讀。主體(3×3 放大)與全圖(縮小)分離,是版型的核心決定。
4. **手動複製資料夾來「發佈」**:漏刪舊檔、誤帶私有檔案,遲早出事。白名單 + 清空重放 + 腳本化。
5. **直接改公開 repo 的檔案**:下次 publish 整個被覆蓋。公開 repo 是輸出物,所有修改回 dev repo 再發佈。
6. **og:image 用相對路徑**:LINE / FB 抓不到,分享變破圖卡。完整 https 網址,改圖後用 Sharing Debugger 重抓。
7. **JSON-LD 跟頁面內容對不上**:亂填會被搜尋引擎降權。schema 寫的必須是頁面上真的有的東西。
8. **以為子路徑的 robots/llms 全站生效**:爬蟲讀網域根。要嘛自訂網域,要嘛把根域站的 llms.txt 做好並列出子專案(前課模組 10 的實測教訓)。
9. **sitemap 忘了更新**:新增 demo 或頁面後,sitemap.xml 與 llms.txt 要跟著加,發佈前看一眼。

對照成品

- 完整版:App ——第三步「展開完整 81 格」、第四步切 9×9 版型,整條路徑可離線玩到輸出 PNG。
- 發佈系統:`scripts/publish.sh`(在 dev repo,不隨教材發佈;本章已全文引用)。
- AEO 資產就在你眼前:這個公開站的 llms.txt、sitemap.xml、robots.txt,以及課程首頁的 JSON-LD ——把首頁網址貼給任何 AI 問「這在講什麼」,自己驗收一次。