



Docker 實踐手冊

從容器基礎、Compose 到 Kubernetes

Wayne Chang 編著



免費預覽

這是《Docker 實踐手冊：從容器基礎、Compose 到 Kubernetes》的免費預覽，從容器觀念、開發環境與常用指令讀起，並進入 Dockerfile 與鏡像優化。

完整版另含資料與網路、Compose、FastAPI 本地開發、運行時與 Podman、通往 Kubernetes 的橋接，以及在 Google Cloud 虛擬機上的三個實驗與三篇附錄。



第一章：容器技術與 Docker 基礎概念

本章宗旨： 建立正確的容器化思維，釐清虛擬機與容器的本質差異，並掌握 Docker 主從式架構與核心三大元素（Image、Container、Registry）。

學完本章後，你應該能回答：

- 容器為什麼比虛擬機更輕、更快？
- 容器如何靠 Namespaces / Cgroups 隔離視野並限制資源用量？
- Docker Client、Daemon、Host 各自負責什麼？
- Image、Container、Registry 如何串成一個完整運作流程？

1.1 什麼是容器與容器化 (Container & Containerization)

1. 容器 (Container) 的定義

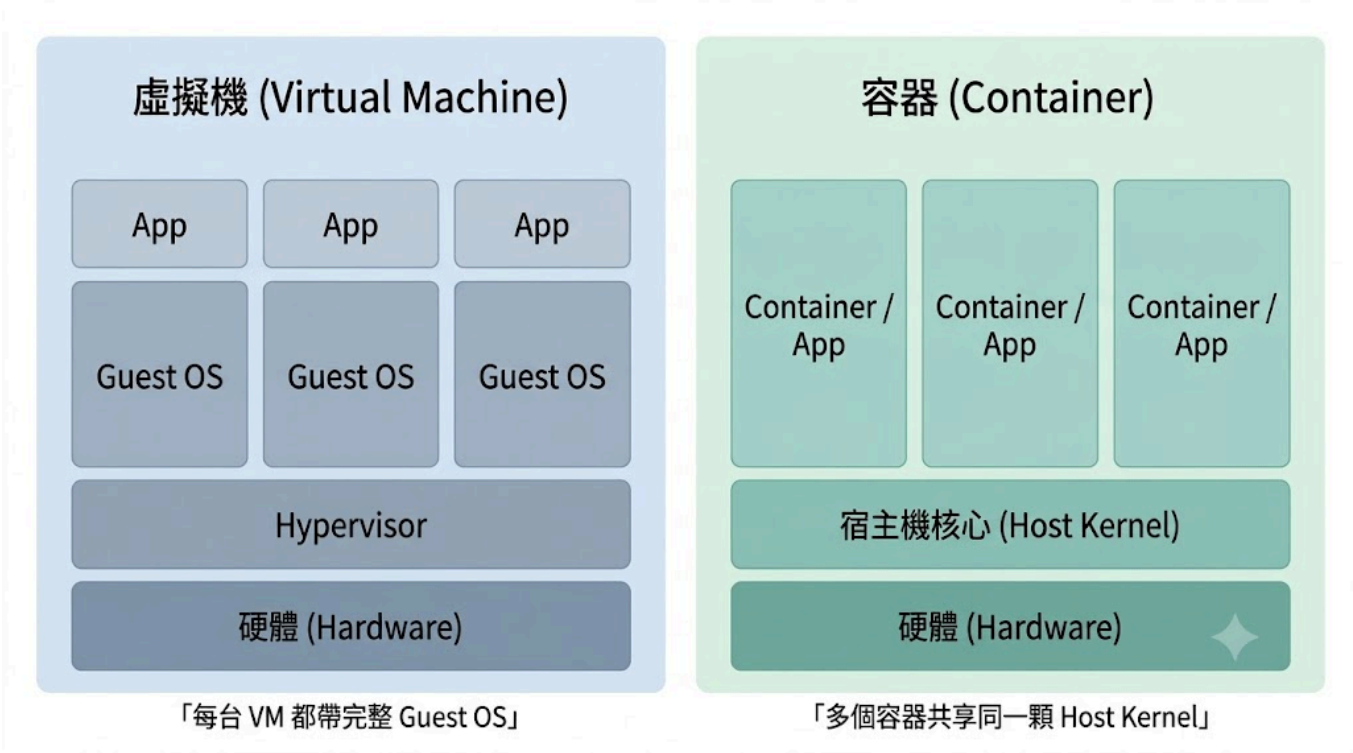
- **輕量化打包套件：** 容器是一個輕量級、可移植的軟體套件，它將應用程式的程式碼、所需的函式庫、運行時 (Runtimes)、設定檔以及所有系統依賴項完整打包在一起 [Chronosphere 容器解析](#)。
- **行程級隔離：** 容器在宿主機 (Host System) 上運行，本質上是一個被隔離的行程 (isolated process)，並直接共享宿主機的作業系統核心 (OS Kernel)。
- **一致的執行行為：** 因為容器攜帶了自身所需的執行環境，有助於讓應用程式在開發者的筆電、CI 測試平台或生產伺服器等不同環境下，表現得更接近一致（仍須留意底座 OS / 架構差異與設定是否對齊）。

補充 (Mac / Windows)： 「共享宿主機核心」這句話在 Linux 上最直接成立。若你使用 Docker Desktop [Docker Desktop 官方說明](#) (macOS / Windows)，Docker 實際上會在背景跑一個輕量 Linux 虛擬機，容器共享的是該 Linux VM 的核心，而非 macOS / Windows 本體核心。這不影響日常使用觀念，但有助於理解為什麼 Desktop 需要安裝與啟動背景服務。

2. 容器與虛擬機 (VM) 的本質差異

- **虛擬機 (Virtual Machine)：**
 - 需要在實體硬體與 Hypervisor 之上，為每個實例載入一個完整的虛擬作業系統 (Guest OS)，這使得虛擬機體積龐大、資源開銷高，且啟動速度通常明顯慢於容器。
 - 隔離邊界較厚 (各自 Guest OS)；相對地，啟動與資源開銷也較高。
- **容器 (Container)：**

- 不需要包含獨立的作業系統核心，直接共享宿主機的核心（Host Kernel）；鏡像裡仍常帶精簡的 userland（如 Alpine 的系統工具與函式庫），但不會各自再帶一顆完整 Guest OS，因此體積極小（通常僅數 MB 至數百 MB），資源消耗極低。
- 啟動速度極快，屬於行程級啟動（常見為數毫秒至數秒），能更高效地利用底層實體基礎設施的運算資源。
- 故障處理較快，但不是絕對隔離：單一應用崩潰時，通常可只重啟該容器，不必連帶重開整台 Guest OS。不過容器仍共享核心——隔離是 Namespaces / 行程層級，不是硬體虛擬化；未設限的資源耗盡、核心層問題，或把宿主敏感路徑掛進容器，仍可能影響同機其他容器或宿主。Cgroups 用來限制用量；安全邊界會在第九章再談。

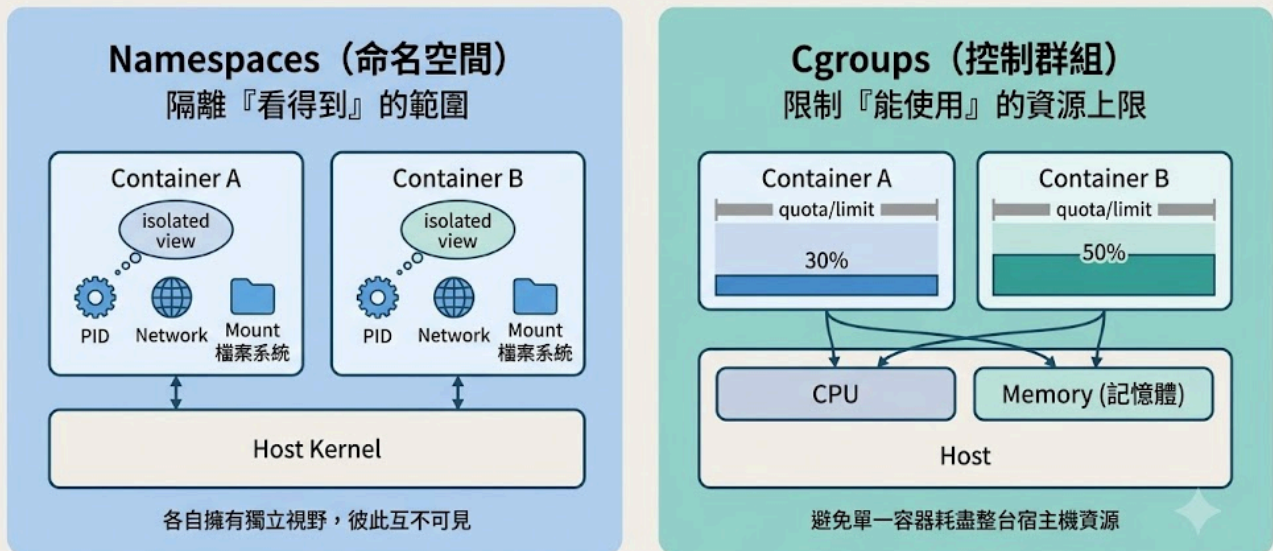


比較維度	虛擬機 (Virtual Machine)	容器 (Container)
作業系統 (OS)	包含完整的 Guest OS	共享宿主機核心，無獨立作業系統
資源消耗 (Size)	體積龐大 (通常數 GB 起步)	極為輕量 (通常僅數 MB 至數百 MB)
啟動時間 (Speed)	較慢 (通常明顯慢於容器)	極快 (行程級啟動，常見為毫秒至數秒)
隔離邊界	硬體層級虛擬化隔離	核心命名空間行程級隔離

3. 底層隔離機制 (Linux Kernel 隔離技術)

容器並非真正的虛擬機，它只是宿主機上一個受到資源限制與安全隔離的普通行程。此隔離機制主要依賴 Linux 核心的兩大關鍵技術：

容器底層隔離：Namespaces 與 Cgroups



Namespaces 負責隔離視野，Cgroups 負責限制用量

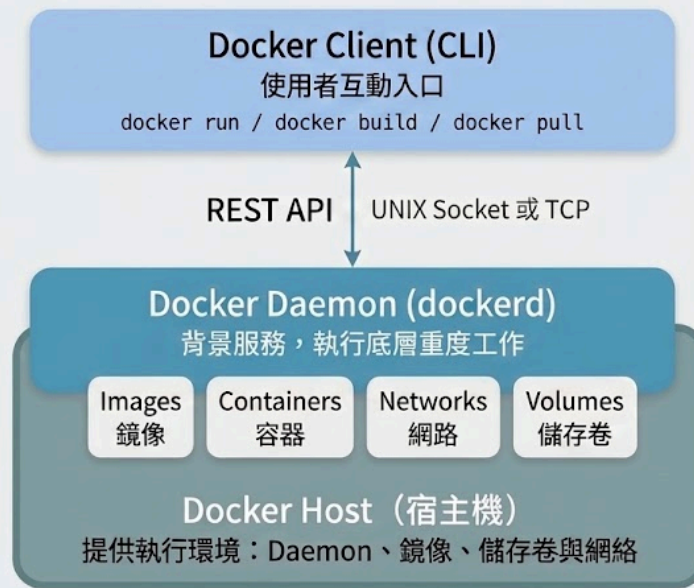
- **Namespaces (命名空間)**：隔離行程「看得到」的範圍。負責劃分行程識別碼 (PID)、網路空間、檔案系統掛載點等，防止容器之間互相干擾或看見彼此的資源。
- **Cgroups (Control Groups, 控制群組)**：控管行程「能使用」的資源上限。負責強制限制容器的 CPU 運算量、記憶體使用量等硬體資源，避免單一容器因行程異常而耗盡整台宿主機的資源。

現階段只要抓住一句話即可：**Namespaces 負責隔離視野，Cgroups 負責限制用量**——容器因此能輕量地共享同一顆核心；「輕」來自共享，「有限度的隔離」也來自共享。

1.2 Docker 的主從式架構 (Client–Server Architecture)

Docker 採用 **Client–Server (主從式)** 架構 [Docker Overview 官方文件](#)。整個運作生態主要由 **Docker Client**、**Docker Daemon** 與 **Docker Host** 組成，彼此透過標準的 **REST API** 進行通訊。

Docker 主從式架構 (Client-Server Architecture)



1. Docker Client (docker)

- 這是使用者與 Docker 互動的最主要入口。
- 當使用者在終端機輸入指令（如 `docker run`、`docker build`、`docker pull`）時，Client 會利用 **Docker API** 將這些指令轉譯為 REST API 請求，並發送給背景的 Docker Daemon 執行。
- 一個 Docker Client 可以根據設定，同時連接並控制多個本地或遠端的 Docker Daemon。

2. Docker Daemon (dockerd)

- 常駐在宿主機背景執行的監聽行程（Daemon）。
- 它專門接收來自 Client 的 API 請求，並負責處理所有耗費資源的底層重度工作，包括建立、運行、分發與管理 **Images (鏡像)**、**Containers (容器)**、**Networks (網路)** 及 **Volumes (資料卷)**。

3. Docker Host (宿主機環境)

- 提供容器運行的實體機或虛擬機環境。
- 宿主機內部包含 Docker Daemon 服務，並集中管理本地下載快取的鏡像、運作中的容器實例、虛擬網路配置以及持久化儲存空間。
- **通訊通道**：Client 與 Daemon 在本地端預設使用 **UNIX Sockets**（如 `/var/run/docker.sock`）進行通訊，若要跨網路連接遠端主機，則會透過 **TCP 網路介面** 傳輸。

1.3 Docker 核心三大元素 (The Core Elements)

理解 Docker 的運作，必須掌握以下三個彼此關聯的核心元件 [Docker Overview 官方文件](#)。可先用物件導向比喻快速建立直覺：

元素	比喻	一句話
Image (鏡像)	類別 (Class) / 藍圖	唯讀模板，描述「要跑什麼」
Container (容器)	物件 (Object) / 一台小電腦	由鏡像啟動、正在 (或可) 運行的實例
Registry (倉庫)	套件庫 / 雲端倉庫	存放與分發鏡像的服務系統 (公有或私有)

1. 鏡像 (Image)

- **定義：**鏡像 (Image；舊稱「映像檔」) 是一個**唯讀 (Read-only)** 的靜態模板，裡面包含了建立與啟動容器所需的一切指令與打包環境 (包括程式碼、運行時、系統工具、作業系統層與組態設定)。本書正文統一使用「鏡像」。
- **分層與快取機制 (Image Layering)：**
 - 鏡像採用分層疊加的結構，Dockerfile 中的每一行指令都會在鏡像中建立一個唯讀層 (Layer)。
 - 當你修改 Dockerfile 並重新建置時，**Docker 只會重新建置發生變更的層及其後的層**，其餘未修改的層會直接重用本地快取，這使得鏡像建置極快且傳輸體積非常輕量。
 - 本章先建立分層觀念即可；撰寫策略與最佳實踐會在後續 Dockerfile 章節再深入。

2. 容器 (Container，容器實例)

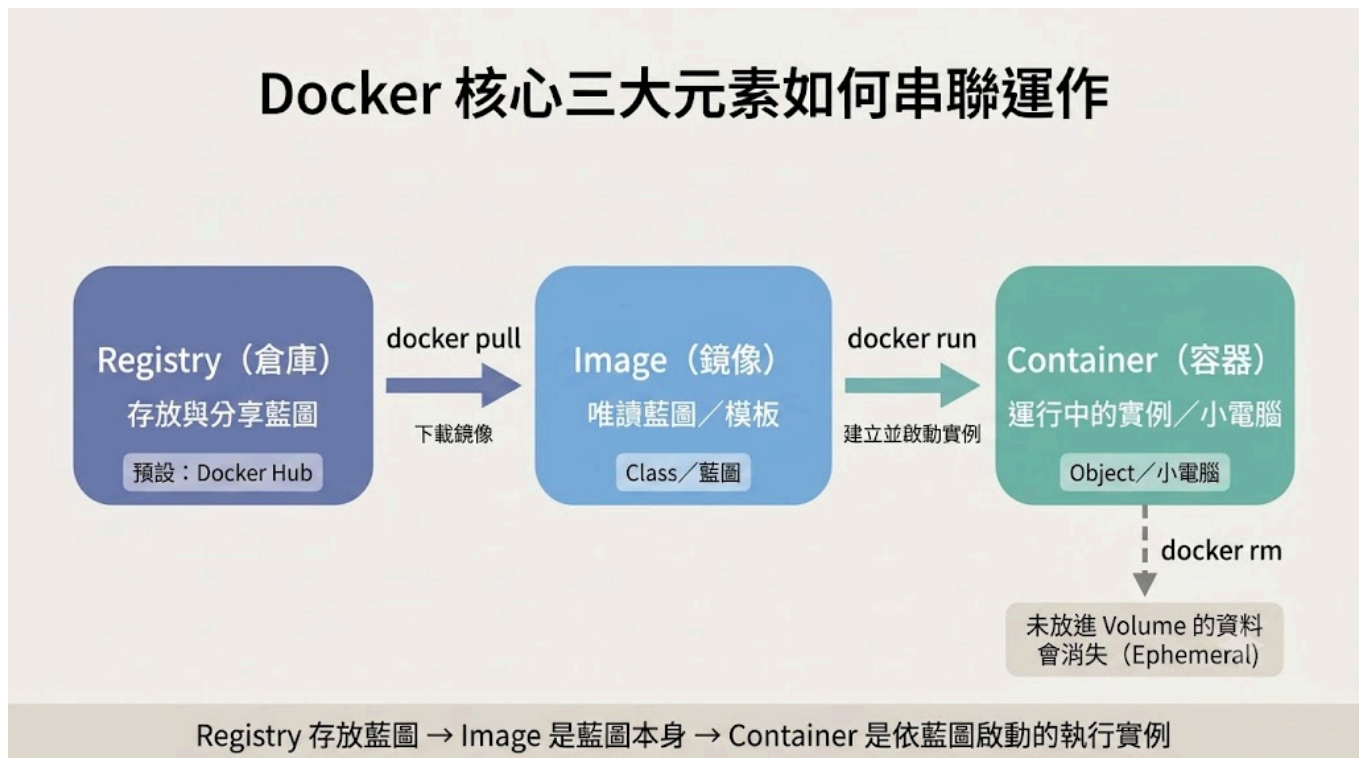
- **定義：**容器是由鏡像啟動的運行中實例 (Runnable Instance) ——對應上表的「物件 / 小電腦」比喻。(此處「運行中」指生命週期狀態，**不是**第九章的 Container Runtime / 運行時鏈。)
- **生命週期與揮發性 (Ephemeral)：**容器預設具揮發性 (Ephemeral)，一旦被刪除，任何未儲存在持久化機制 (如 Docker Volumes / 資料卷) 的內部檔案變更與資料都會隨之消失。

3. 倉庫 (Registry)

- **定義：**倉庫是用於集中儲存、管理與分發容器鏡像的服務系統。
(勿與 `docker images` 欄位 **REPOSITORY** 混淆：那是本機列出的**鏡像名稱路徑**，第三章會再標明。)
- **公有與私有倉庫：**
 - **公有倉庫 (Public Registry)：**最著名的代表為官方的 **Docker Hub**。當我們在終端機輸入下載指令 (如 `docker pull`) 而未特別指定倉庫來源時，Docker 預設會直接前往 Docker Hub 下載鏡像。
 - **私有倉庫 (Private Registry)：**企業通常會建置內部專用的私有倉庫，用來安全地儲存與分享組織內部的商務機密鏡像，避免機密程式碼外洩。

4. 三大元素如何串聯運作

單獨認識三個名詞還不夠，關鍵是理解它們如何串成一條日常工作流：



1. 從 Registry 取得 Image (對應 `docker pull`)：本機沒有需要的鏡像時，Docker 會向倉庫（預設 Docker Hub）下載。
2. 以 Image 建立並啟動 Container (對應 `docker run`)：鏡像是唯讀藍圖；真正執行時會再加上一層可寫層，形成正在運行的容器——也就是依藍圖啟動的一台「小電腦」。
3. 容器結束後資料可能消失 (對應 `docker rm`)：容器刪除後，未放進 Volume 的變更不會保留——這也是後續章節會談「持久化」的原因。

心智模型小結： Registry 負責「存放與分享藍圖」；Image 是「藍圖本身」；Container 是「依藍圖啟動起來的執行實例」。日常指令多半就是在這三者之間轉換；實際操作與參數細節，會在第二章開始動手練習。

1.4 本章小結與動手練習

小結

1. **容器 ≠ 虛擬機**：容器共享宿主機核心、以行程級隔離運行，因此更輕、更快；虛擬機則各自攜帶完整 Guest OS。共享核心也代表隔離有限度，不是「一個容器壞掉就絕不會影響整機」。
2. **隔離靠兩招**：Namespaces 負責隔離視野，Cgroups 負責限制用量。

3. **Docker 是主從架構**：你下的 CLI 指令由 Client 發出，真正做事的是背景的 Daemon（`dockerd`），兩者透過 REST API 通訊。
4. **三大元素缺一不可**：Registry 存放 Image，Image 是唯讀模板，Container 是運行實例；刪除容器不會自動刪除鏡像，但容器內未持久化的資料會消失。

動手練習

本章尚不需安裝 Docker；請用文字回答（可對照本節圖表），第二章再動手操作。

1. 用自己的話說明：為什麼容器通常比虛擬機更輕、更快？「共享宿主機核心」這句話，在 macOS / Windows 的 Docker Desktop 上要怎麼理解？
2. 對照 Namespaces 與 Cgroups：各負責「看得到什麼」還是「能用多少」？若某個容器記憶體用量失控，比較可能靠哪一個機制擋住？
3. 依序寫出日常流程：從 Registry 取得鏡像 → 啟動容器 → 刪除容器後，哪些東西通常還在、哪些可能消失？
4. （選做）為什麼說「重啟故障容器很快」**不等於**「容器絕對不會影響宿主或其他容器」？舉一個本章提過的反例或風險即可。

下一章預告：完成 Docker Desktop 安裝與設定，親手運行 `hello-world`，並逐步拆解 `docker run` 背後發生的完整流程。

Reference

1. Chronosphere 容器解析 — <https://chronosphere.io/learn/kubernetes-pod-vs-container/>
2. Docker Desktop 官方說明 — <https://docs.docker.com/desktop>
3. Docker Overview 官方文件 — <https://docs.docker.com/get-started/docker-overview/>



第二章：開發環境準備與第一個容器

本章宗旨： 完成開發工具安裝，並親手運行第一個 Docker 程式；不只會下指令，也要理解 `docker run` 在背景實際做了什麼。

學完本章後，你應該能回答：

- 如何安裝並確認 Docker Desktop / Daemon 正常運行？
- 如何用 `hello-world` 完成第一次 pull 與 run？
- 為什麼重複執行 `docker run` 會產生多個容器？該用什麼指令重跑既有容器？
- `docker run` 背後依序發生哪些步驟？

2.1 Docker Desktop 安裝、設定與背景原理

要在個人電腦上開始使用 Docker，第一步是安裝官方提供的整合開發環境工具 —— Docker Desktop。

1. 跨平台安裝路徑

請依你的作業系統，到官方文件完成安裝：

- **Windows**（建議啟用 WSL2 以獲得最佳效能與相容性） — [Install Docker Desktop on Windows](#)
- **macOS**（支援 Apple Silicon 與 Intel 晶片） — [Install Docker Desktop on Mac](#)
- **Linux**（如 Ubuntu、Debian、Fedora 等） — [Install Docker Desktop on Linux](#)

Linux 補充： 本教材以 Docker Desktop 作為統一入門路徑 [Docker Desktop 官方說明](#)。若你在 Linux 上只想安裝引擎、不需要 Desktop GUI，也可以改裝 **Docker Engine**（參見 [Docker Engine 安裝總覽](#)）。後續指令在 Desktop 與 Engine 上大多相通；本課程示範仍以 Desktop 為主。

安裝完成後，**必須實際啟動 Docker Desktop**（或確保 Engine / Daemon 已在運行），使其在背景保持可用，後續在終端機（Terminal）輸入的 Docker 指令才能正常生效。

2. 如何確認安裝成功

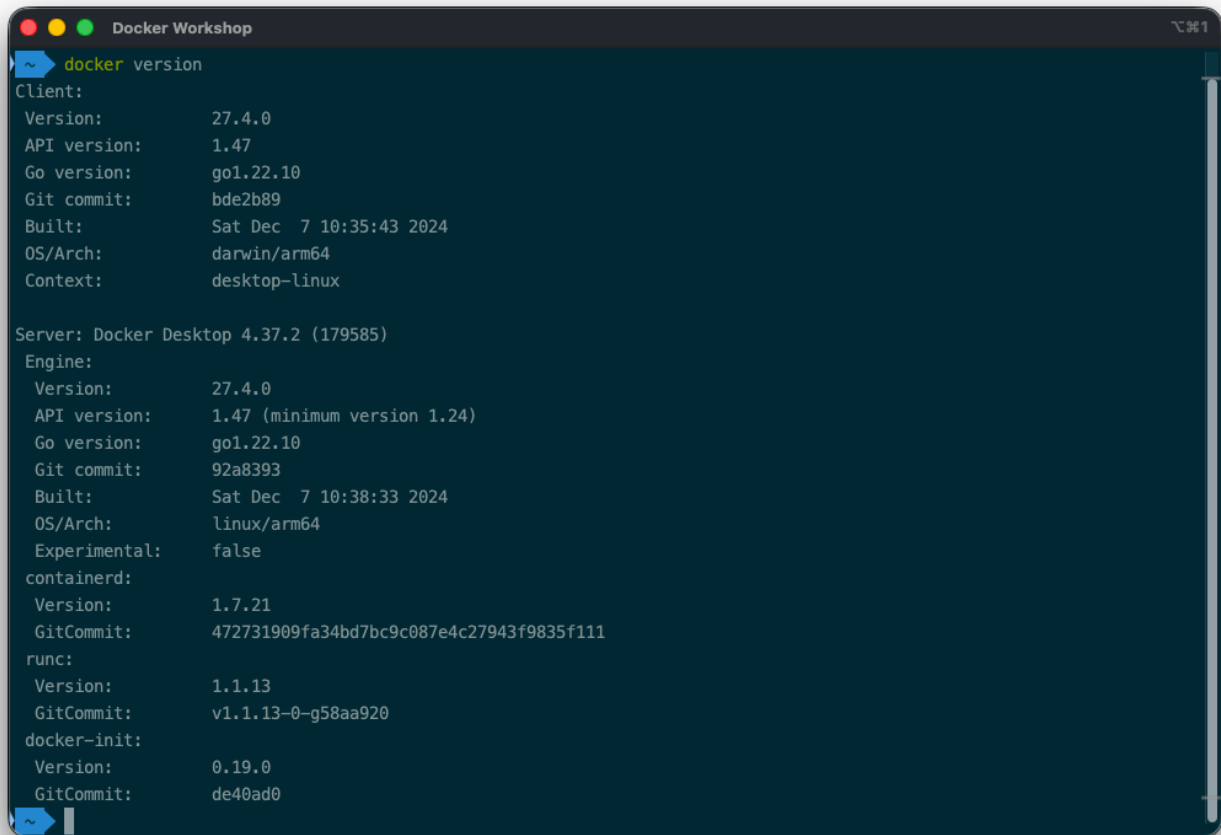
啟動 Docker Desktop，等待狀態顯示為 Running（選單列 / 系統匣圖示不再轉圈）後，在終端機執行：

```
docker version
```

若 Client 與 Server（Engine / Daemon）兩側都有版本資訊，代表本機已能連上背景 Daemon。也可再執行：

```
docker info
```

若這裡就失敗，多半是 Desktop 尚未就緒，請先回到上一步確認軟體已開啟。



```
~ docker version
Client:
 Version:           27.4.0
 API version:       1.47
 Go version:        go1.22.10
 Git commit:        bde2b89
 Built:             Sat Dec  7 10:35:43 2024
 OS/Arch:           darwin/arm64
 Context:           desktop-linux

Server: Docker Desktop 4.37.2 (179585)
Engine:
 Version:           27.4.0
 API version:       1.47 (minimum version 1.24)
 Go version:        go1.22.10
 Git commit:        92a8393
 Built:             Sat Dec  7 10:38:33 2024
 OS/Arch:           linux/arm64
 Experimental:      false
containerd:
 Version:           1.7.21
 GitCommit:         472731909fa34bd7bc9c087e4c27943f9835f111
runc:
 Version:           1.1.13
 GitCommit:         v1.1.13-0-g58aa920
docker-init:
 Version:           0.19.0
 GitCommit:         de40ad0
```

3. 常見新手連線錯誤剖析

如果在執行任何 `docker` 指令時，終端機出現以下錯誤訊息：

```
Cannot connect to the Docker daemon at unix:///Users/.../docker.sock. Is the
docker daemon running?
```

這代表 **Docker Desktop 尚未啟動**，或背景的 Docker Daemon 服務已停止。請開啟 Docker Desktop，待其完全就緒後再重試指令。

```
Docker Workshop
~ docker version
Client:
Version:      27.4.0
API version:  1.47
Go version:   go1.22.10
Git commit:   bde2b89
Built:        Sat Dec  7 10:35:43 2024
OS/Arch:      darwin/arm64
Context:      desktop-linux
Cannot connect to the Docker daemon at unix:///Users/*****/.docker/run/docker.sock. Is the docker daemon running?
```

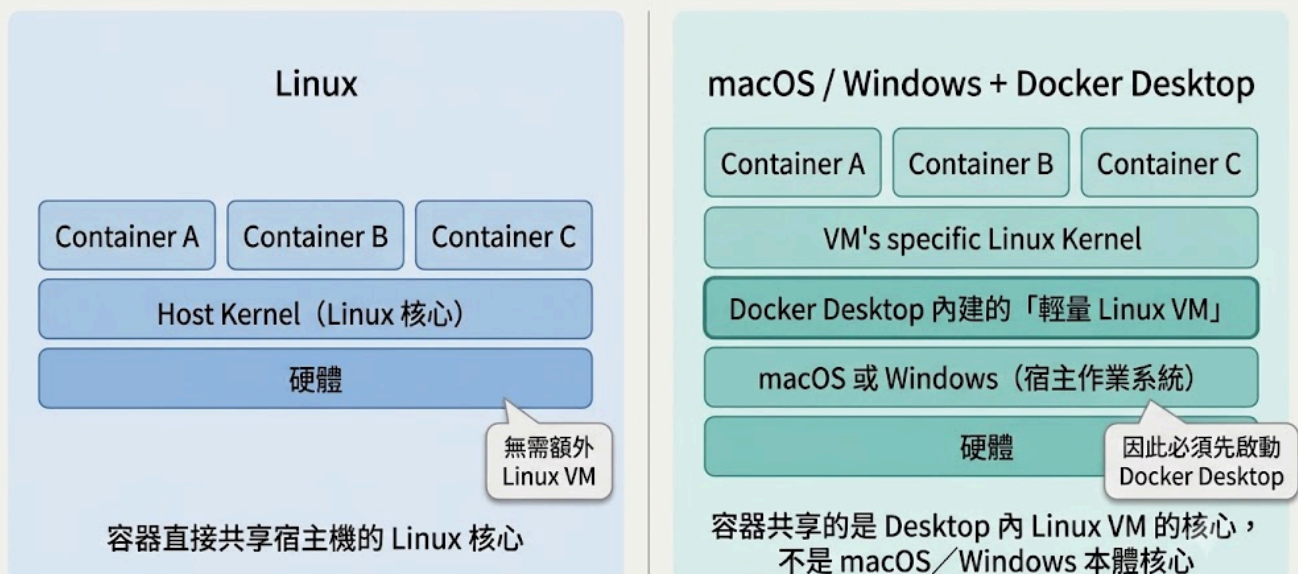
4. macOS / Windows 共享宿主機核心的本質

在第一章我們提到「容器與宿主機共享作業系統核心（OS Kernel）」。這裡再複習一次：

- **Linux 系統**：容器直接共享 Linux 宿主機的核心。
- **macOS / Windows 系統**：由於這兩個作業系統並非 Linux，Docker Desktop 實際上會在背景跑一個輕量級的 Linux 虛擬機（VM）。此時，容器共享的是該 Linux VM 的核心，而非 macOS 或 Windows 本體的核心。

這也解釋了為什麼在 Mac 與 Windows 上使用 Docker 時，必須先啟動 Desktop（及其背景 VM 服務）。

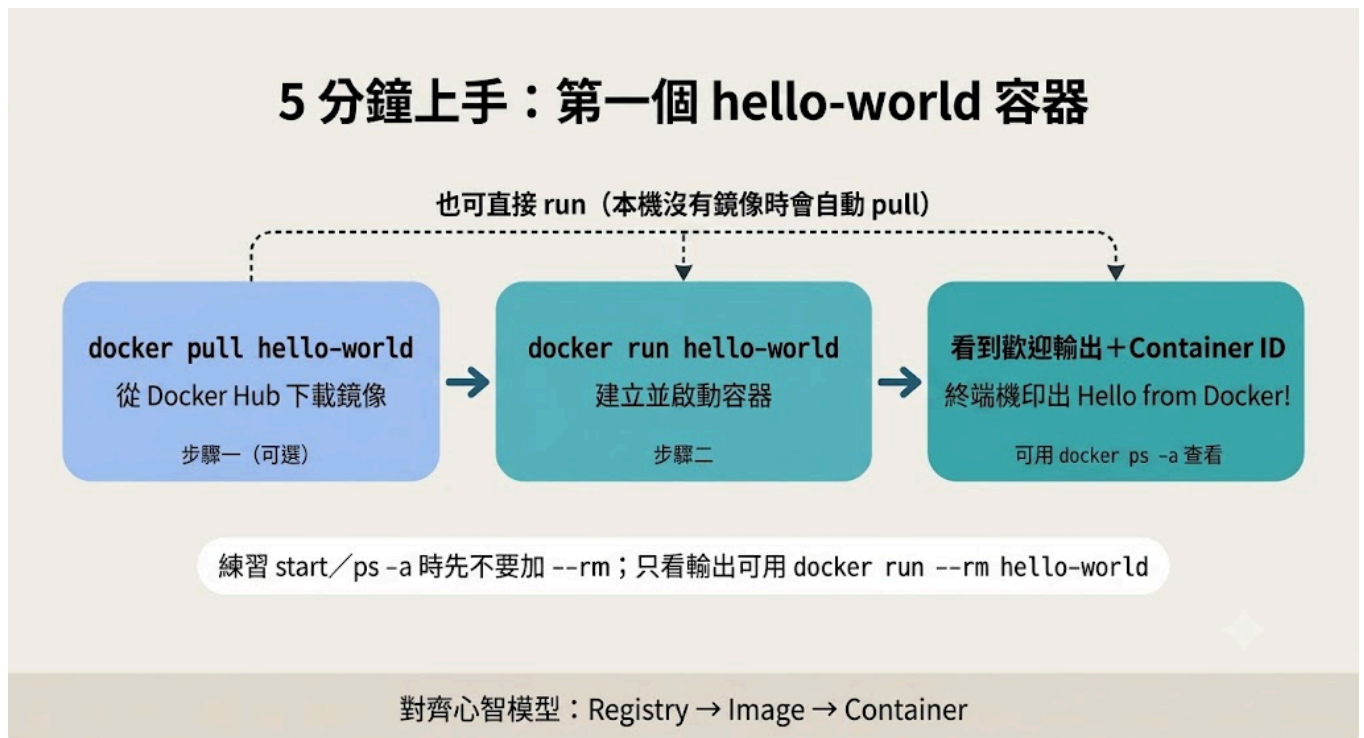
為什麼 Mac / Windows 上必須先啟動 Docker Desktop？



這也解釋了為什麼 Desktop 沒開時，會出現 Cannot connect to the Docker daemon

2.2 5 分鐘快速上手實驗：第一個 hello-world 容器

接下來，我們將實際從雲端倉庫拉取鏡像，並利用它建立並運行容器。本章先求跑通；明確指定鏡像版本 (Tag)、避免依賴 `latest` 的好習慣，會留到第三章再系統強調。



【步驟一：下載鏡像 (Image)】

在終端機中輸入以下指令，從預設的官方倉庫 (Docker Hub) 下載經典的 `hello-world` 鏡像 [hello-world 官方鏡像](#)：

```
docker pull hello-world
```

概念對照：`pull` 會從遠端倉庫 (Registry) 將鏡像下載到本機。你可以透過 Docker Desktop 介面，或指令 `docker images` 查看已下載的鏡像。

【步驟二：運行容器 (Container)】

下載完成後，輸入以下指令來建立並啟動容器 [docker run 官方說明](#)：

```
docker run hello-world
```

執行後，你會在主控台 (Console) 看到 Docker 輸出的歡迎與說明資訊，代表已成功在隔離的容器環境中執行該程式。此時若回到 Docker Desktop 的 Containers 畫面，會看到一個新生成的容器，並標示其隨機分配的 **Container ID** (例如 `f10ff7cdfb54`；請以你畫面上實際顯示的 ID 為準)。

```
Docker Workshop
docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
58dee6a49ef1: Pull complete
Digest: sha256:c3cbe1cc1aa588a64951ac6286e0df7b27fe2e6324b1001c619bb358770c0178
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm64v8)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

```
Docker Workshop
docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c397a04a7fbc	hello-world	"/hello"	2 minutes ago	Exited (0) 2 minutes ago		sleepy_wright

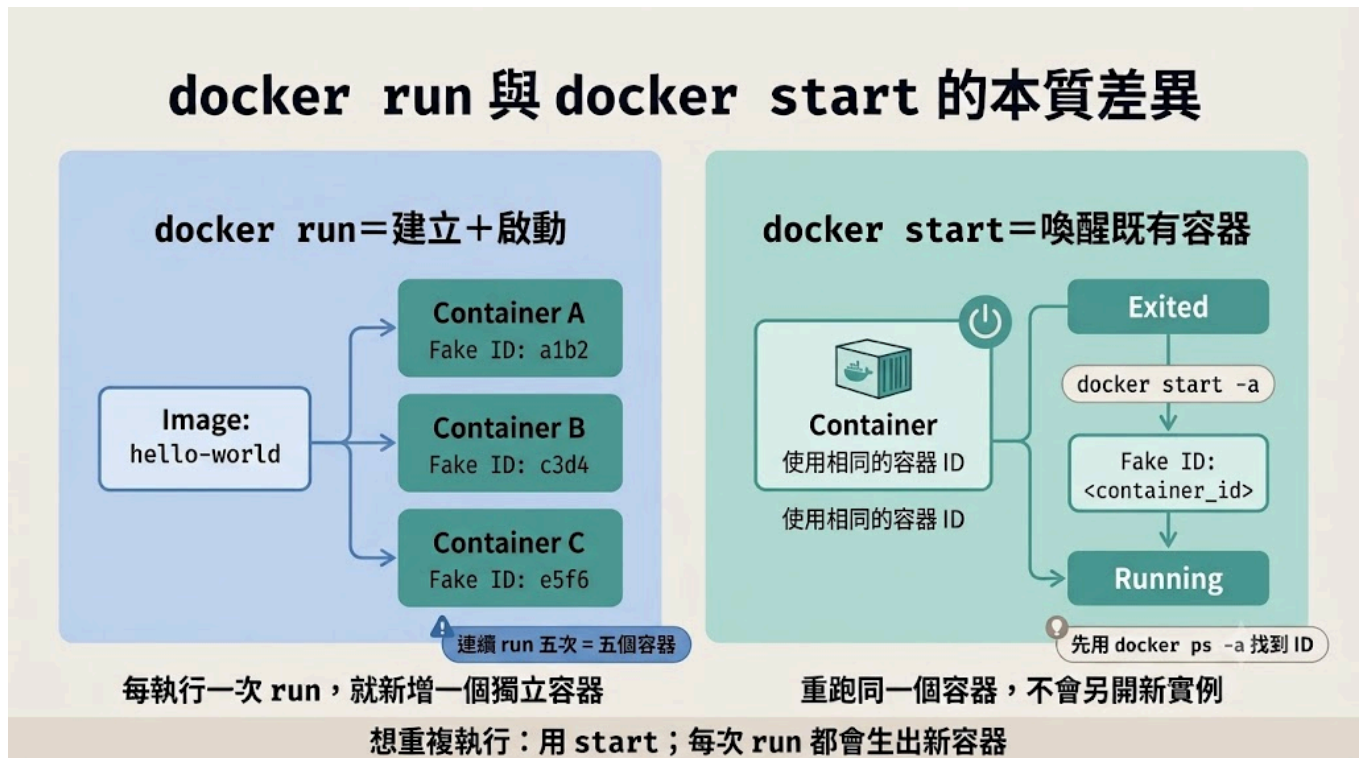
補充：其實也可以略過步驟一，直接執行 `docker run hello-world`。若本機尚無該鏡像，Docker 會自動先 pull 再建立容器——這正是下一節「docker run 六大步驟」的第一步。本章分開練習，是為了先對齊第一章的 Registry → Image → Container 心智模型。

若不想在練習過程中累積一堆已結束（Exited）的容器，可改用：

```
docker run --rm hello-world
```

`--rm` 會在容器結束後自動刪除該容器，適合這種「跑完看輸出就好」的實驗；若你還想練習 `docker start` / `docker ps -a`，請先不要加 `--rm`。

核心觀念釐清：`docker run` 與 `docker start` 的本質差異



許多初學者會重複執行 `docker run` 來啟動程式，這是非常常見的誤區：

- `docker run` 是「建立 + 啟動」：** 每一次執行 `docker run <image>`，Docker 均會全新建立一個完全獨立的容器實例。
 - 舉例來說，如果你在終端機中連續運行 `docker run hello-world` 五次，你的電腦實際上會建立起 5 個彼此獨立的容器！這些已停止的舊容器會持續佔用硬碟空間，造成不必要的資源浪費。
- `docker start` 是「喚醒既有容器」：** 如果你只想重新執行已經建立過的容器，不應該再使用 `run`，而是要先用 `docker ps -a` 找出該容器的 Container ID（或名稱），再使用 `start` 重新運行它：

```
1 # 請把 <container_id> 換成 docker ps -a 列出的實際 ID 或名稱
2 docker start -a <container_id>
```

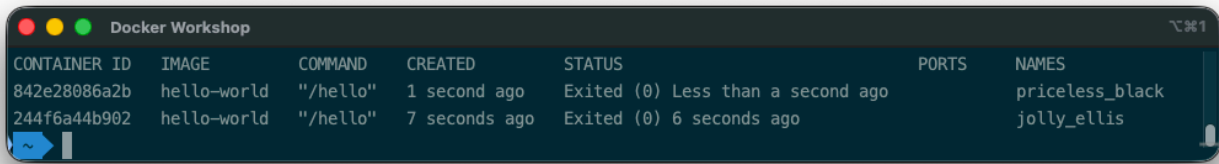
實用排查指令：檢視容器狀態

若想查看目前電腦中所有的容器（不論正在運行，或已經結束運作），請在終端機輸入：

```
docker ps -a
```

提示：一定要加上 `-a`（All）參數。如果不加 `-a`，預設只會列出當前「正在執行中」的容器，而

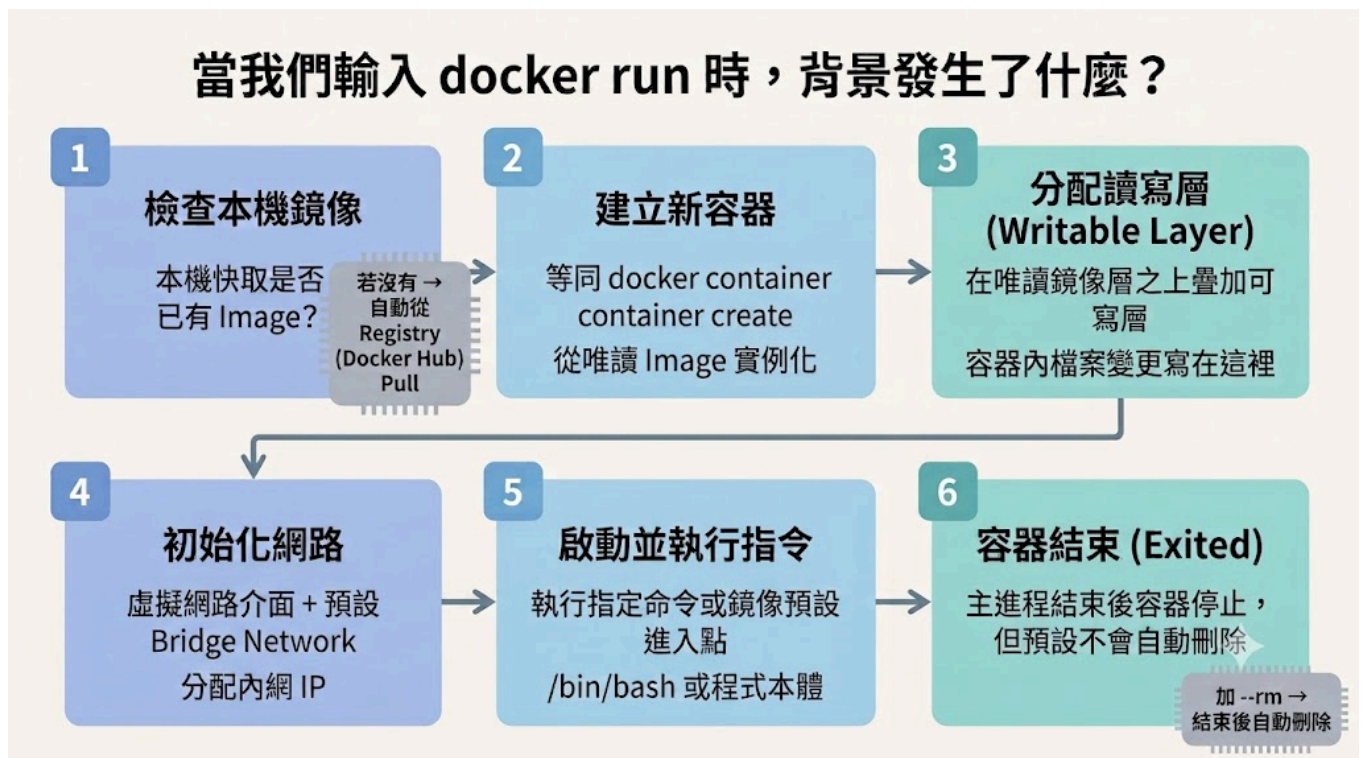
hello-world 這類執行完即結束的容器將不會顯示在列表上。



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
842e28086a2b	hello-world	"/hello"	1 second ago	Exited (0) Less than a second ago		priceless_black
244f6a44b902	hello-world	"/hello"	7 seconds ago	Exited (0) 6 seconds ago		jolly_ellis

2.3 深度剖析：當我們輸入 `docker run` 時，背景到底發生了什麼事？

當我們在終端機敲下 `docker run <image_name>`（例如 `docker run -it ubuntu /bin/bash`）並按下 Enter 時，Docker 背景會依序自動執行以下 6 大完整流程：



1. 本地鏡像檢查與 Registry 拉取

Docker Client 會向背景的 Docker Daemon 發送請求。Daemon 首先檢查本機快取中是否存在該鏡像。

- 如果本機有，直接使用該鏡像。
- 如果本機沒有，Docker 會自動前往配置的倉庫（預設為 Docker Hub）進行 `pull` 下載。

這也呼應上一節：直接 `docker run hello-world` 時，若本機沒有鏡像，就會先走完這一步再繼續往下。

2. 建立新容器實例

鏡像準備就緒後，Docker 會利用這個唯讀的靜態模板，在宿主機中「實例化」出一個全新的容器，其效果等同於手動下達了 `docker container create` 命令。

3. 分配讀寫檔案系統層 (Writable Layer)

由於鏡像的所有分層都是唯讀且不可變的 (Immutable)，為了讓容器能正常執行與讀寫資料，Docker 會在唯讀的鏡像層之上，額外疊加一個可寫的檔案系統層 (Read-Write Writable Layer)。

- 容器運行期間所有的新增、修改與刪除檔案操作，都會被記錄在這一層。
- 當容器被刪除時，若資料沒有保存在持久化儲存區，這個可寫層內的所有變更都會隨之消失。

4. 初始化網路介面與 IP 位址

為了讓容器能與外界或其他容器通訊，Docker 會為其建立一個虛擬網路介面，並將其連接至預設的橋接網路 (Default Bridge Network)。在此步驟中，Docker 會自動為該容器分配一個獨立的虛擬內網 IP 位址。

5. 啟動容器並執行指定指令

網路、檔案系統與命名空間隔離就緒後，Docker 正式啟動容器行程，並執行你在命令列尾端指定的指令（例如 `/bin/bash`）或鏡像預設指定的進入點程式。

6. 容器結束運作 (Exit) 與生命週期

當容器內運行的主行程 (PID 1) 執行結束（例如你在 `bash` 中輸入 `exit` 退出，或是內部的程式碼執行完畢）時，容器便會自動停止運行。

- **重要提醒：**容器停止 (Status: Exited) 並不代表它被刪除了！該容器的 Writable Layer、Container ID 與網路設定依然保留在你的硬碟中，你可以隨時透過 `docker start` 將其再次喚醒。若要徹底清除，需要手動執行 `docker rm`，或是在一開始 `run` 時加上 `--rm` 參數。

2.4 本章小結與動手練習

小結

1. **Docker Desktop 是入口：**安裝後要啟動並用 `docker version` 確認 Daemon 可連線；在 macOS 與 Windows 上，它以輕量 Linux VM 提供容器所需的核心理環境。
2. **`docker run` 會一直生出新實例：**重複執行時應使用 `docker start`，並定期用 `docker ps -a` 檢視容器狀態；本機沒有鏡像時，`run` 也會自動先 `pull`。

3. 了解 `run` 的 6 大步驟：有助於你在後續章節學習「網路配置」與「Volumes 持久化」時，迅速理解底層為什麼需要特別對接。

動手練習

1. 啟動 Docker Desktop，執行 `docker version`，確認 Client 與 Server 都有輸出。
2. 執行 `docker run hello-world`（可先 `pull`，或直接 `run` 觀察自動下載），再執行 `docker ps -a`，找出剛產生的容器 ID。
3. 用 `docker start -a <container_id>` 重跑同一個容器；接著再執行一次 `docker run hello-world`，用 `docker ps -a` 對照：為什麼容器數量變多了？
4. （選做）再跑一次 `docker run --rm hello-world`，然後 `docker ps -a`：確認這次是否留下新的 Exited 容器。

下一章預告：第三章會系統整理日常 CLI——鏡像管理（`pull` / `images` / `rmi`）、容器生命週期（`run` / `start` / `stop` / `rm`）、日誌與 `exec` 除錯，並強調避免濫用 `latest` 標籤等好習慣。本章出現的 `run` / `start` / `ps`，到時會放到完整工具箱裡一起複習。

Reference

1. Docker Desktop 官方說明 — <https://docs.docker.com/desktop/>
2. Docker Engine 安裝總覽 — <https://docs.docker.com/engine/install/>
3. hello-world 官方鏡像 — https://hub.docker.com/_/hello-world
4. docker run 官方說明 — <https://docs.docker.com/engine/containers/run/>



第三章：Docker 常用 CLI 指令操作實務

本章宗旨： 熟練掌握日常開發與運維必備的命令列（CLI）指令。不只是硬背指令，更要理解容器生命週期的流轉，並建立安全、規範的容器管理習慣。

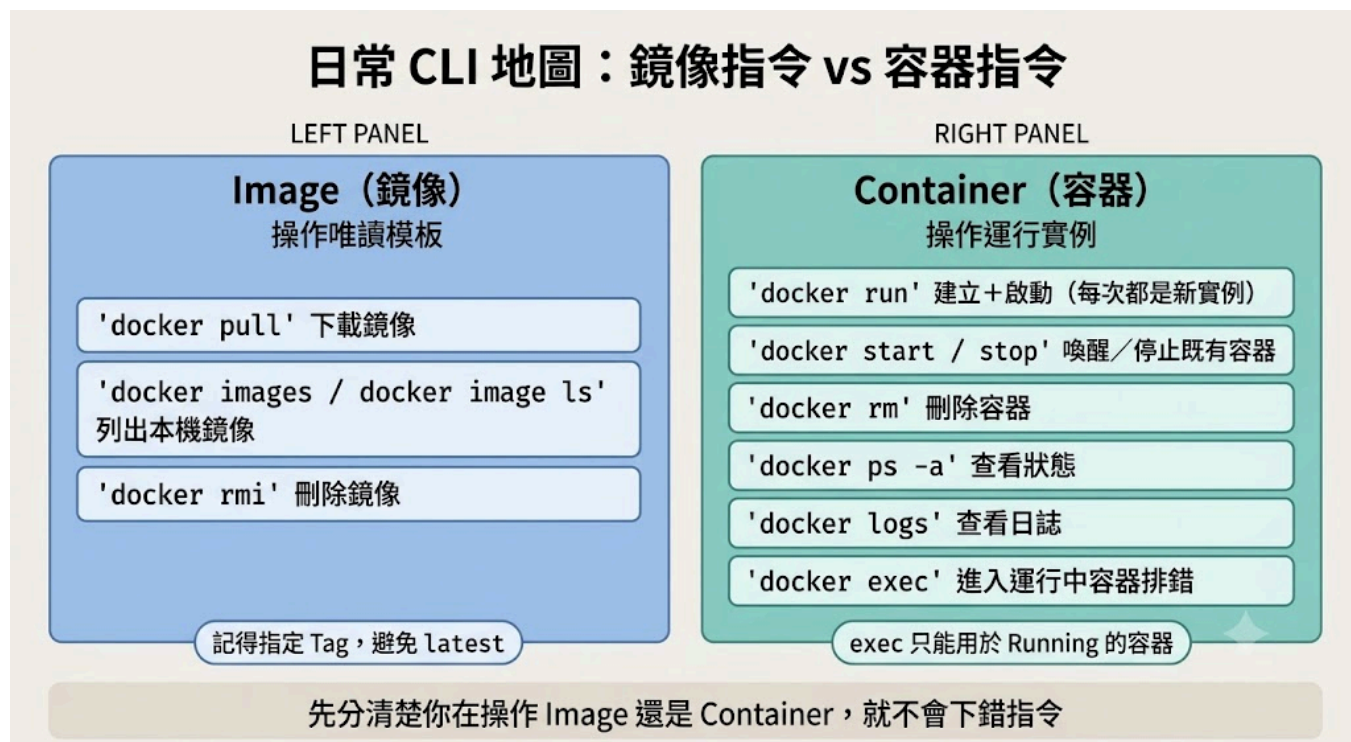
第二章你已經會安裝環境並跑通 `hello-world`；本章要把日常 CLI 收成完整工具箱——從鏡像管理、容器生命週期，到日誌、排錯與安全清理習慣。

學完本章後，你應該能回答：

- 如何用 `pull` / `images` / `rmi` 管理本機鏡像？為什麼不該隨使用 `latest`？
- `run` / `start` / `stop` / `rm` 分別對應生命週期的哪個階段？為什麼重複 `run` 會產生多個容器？
- 如何用 `ps -a` 與 `logs` 觀察容器狀態與輸出？
- 何時、如何用 `exec` 進入運行中的容器排錯？

3.1 鏡像管理 (Image Management)

鏡像（Image）是唯讀的應用程式模板。在日常開發中，我們會頻繁地下載、檢視與清理本機的鏡像。



1. `docker pull` 下載鏡像

從遠端倉庫（預設為 Docker Hub）下載鏡像到本機：

```
docker pull [image_name]:[image_tag]
```

- 實務範例：

```
1 # 下載指定版本 (Tag) 的 Nginx 鏡像
2 docker pull nginx:1.28
```

2. 安全指引：避免使用 `latest` 標籤

如果下載時不指定 tag（例如只輸入 `docker pull nginx`），Docker 預設會下載 `latest`（最新版）。

在生產環境與團隊協作中，強烈建議明確指定版本號（如 `nginx:1.28`）。因為 `latest` 是一個會隨時指向新版的「動態指標（Mutable Tag）」——今天與下個月的 `latest` 可能根本不是同一個內容。若習慣使用 `latest`，極容易因為鏡像在背景悄悄更新、引入不相容的變更，而導致系統突然崩潰或建置失敗。

為什麼不要隨便使用 `latest` 標籤？

使用 `latest`（有風險）



環境難以重現，容易突然壞掉

明確指定 Tag（建議）



下載、執行、文件都寫死明確版本

好習慣：永遠帶上明確 Tag，不要依賴 `latest`

好習慣：下載、執行、寫進文件時，都帶上明確 Tag；需要可重現的環境時，更不要依賴 `latest`。

本書 Nginx 版本線：CLI / 一般示範用 `nginx:1.28`；後續章節需要較小基底時用 `nginx:1.28-alpine`（同一 minor）。官方 Tag 會演進，實作前請至 [Docker Hub nginx](#) 核對，並可替換成當下可用的明確版本。

3. `docker images` 列出本機鏡像

查看目前電腦中已下載並快取的所有鏡像：

```
1 docker images
2 # 或者使用現代標準指令
3 docker image ls
```

- 此指令會顯示鏡像的 REPOSITORY（儲存庫 / 鏡像名稱）、TAG（標籤 / 版本）、IMAGE ID（鏡像 ID）以及 SIZE（大小）。

欄位名 REPOSITORY 指的是本機列出的鏡像名稱路徑（如 nginx、library/nginx），不是第一章的 Registry（倉庫＝存放與分發鏡像的服務系統）。

4. docker rmi 刪除鏡像

當硬碟空間不足，或舊版本鏡像不再需要時，可以使用此指令將其自本機刪除：

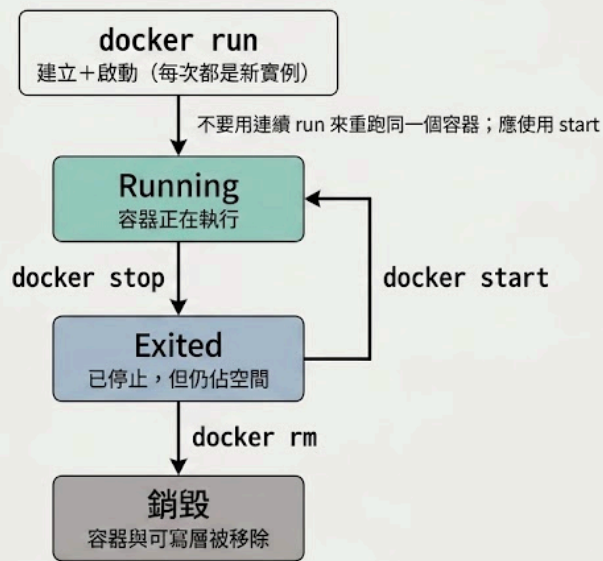
```
1 # 依據 Image ID 刪除
2 docker rmi [image_id]
3
4 # 依據 名字與 Tag 刪除
5 docker rmi nginx:1.28
```

提示：如果要刪除的鏡像目前正被某個容器（不論是運行中或已停止）所使用，必須先刪除該容器，才能順利刪除該鏡像。

3.2 容器生命週期 (Container Lifecycle)

容器是由鏡像啟動的運行中實例，其生命週期包含建立、啟動、停止與刪除。（「運行中」指生命週期，不是 Container Runtime。）

容器生命週期：run / start / stop / rm



Running ⇌ Exited；要清除才用 rm

1. docker run 建立並運行全新容器

`run` 是 Docker 中最核心的指令，它會利用指定的鏡像，全新建立並啟動一個容器 [docker run 官方說明](#)：

```
docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
```

• 常用擴展參數 (OPTIONS)：

- `-d`：在背景 (Detached mode) 運行容器，釋放終端機控制權。
- `-p [宿主機Port]:[容器Port]`：將容器內部監聽的 Port 對接並發布到本機的 Port。
- `--name [自訂名稱]`：為容器指定一個好記的自訂名稱 (若不指定，Docker 會隨機分配名字)。
- `-it`：以互動模式 (Interactive) 並分配一個虛擬終端 (pseudo-TTY)，讓你可以輸入指令。
- `--rm`：容器退出運作時，自動將該容器及其讀寫層檔案系統徹底刪除，非常適合臨時測試。

• 實務範例：

```
1 # 在背景啟動一個名為 my-web 的 Nginx 容器，並將本機 8080 Port 映射到容器的 80 Port
2 docker run -d --name my-web -p 8080:80 nginx:1.28
```

- **⚠ 核心觀念釐清：** `docker run` 每次都會建立全新容器 `run` 的本質是「建立 (Create) + 啟動 (Start)」。如果在終端機中重複輸入 `docker run hello-world` 五次，你的硬碟裡就會多出 5 個彼此完全獨立、各佔空間的容器實例！因此，若要重用已經建立過的舊容器，切記要改用 `docker start`。(此觀念與第二章呼應，本章會放到完整生命週期裡一起練習。)

2. docker start / docker stop 啟動與停止容器

針對已經存在（但處於停止狀態）的容器進行重啟，或是將運作中的容器優雅關閉：

```
1 # 啟動既有容器（會保留先前的所有檔案與狀態）
2 docker start [container_id_or_name]
3
4 # 優雅停止正在運行的容器（發送 SIGTERM 訊號）
5 docker stop [container_id_or_name]
```

補充（較少用）：Docker 還有 `docker pause` / `docker unpause`，可暫時凍結 / 恢復正在運行容器的行程（不同於 `stop` 會結束主行程）。日常開發較少用到；本章以 `start` / `stop` / `rm` 為主即可。

3. docker rm 刪除容器

當容器已經停止（Exited），它依然會殘留在硬碟中。若要徹底釋放資源，必須進行手動刪除：

```
1 docker rm [container_id_or_name]
2
3 # 強制刪除（包含正在運行中的容器，發送 SIGKILL 訊號）
4 docker rm -f [container_id_or_name]
```

3.3 狀態與日誌監控 (Status & Logs Monitoring)

1. docker ps -a 檢視容器狀態

在終端機中確認目前電腦中所有容器的健康與存活狀態：

```
docker ps -a
```

- 不加 `-a`：預設只會列出當前「正在執行中」的容器。
- 加上 `-a`（All）：會列出所有容器，包括已經停止（Exited）、啟動中（Up）或建立但未運行的容器。這對於排查「為什麼我的容器一啟動就立刻掛掉」至關重要。

2. docker logs 即時日誌追蹤

不登入容器，直接在宿主機的終端機上即時抓取並追蹤容器輸出的標準日誌（Stdout / Stderr）：

```
docker logs -f --tail 10 [container_id_or_name]
```

- `-f`：持續追蹤 (Follow) 最新日誌，畫面會保持更新。
- `--tail 10`：僅顯示最後 10 筆歷史日誌，避免被過去大量的舊 Log 刷屏。

3.4 容器除錯 (Debugging)

1. `docker exec` 深入容器內部

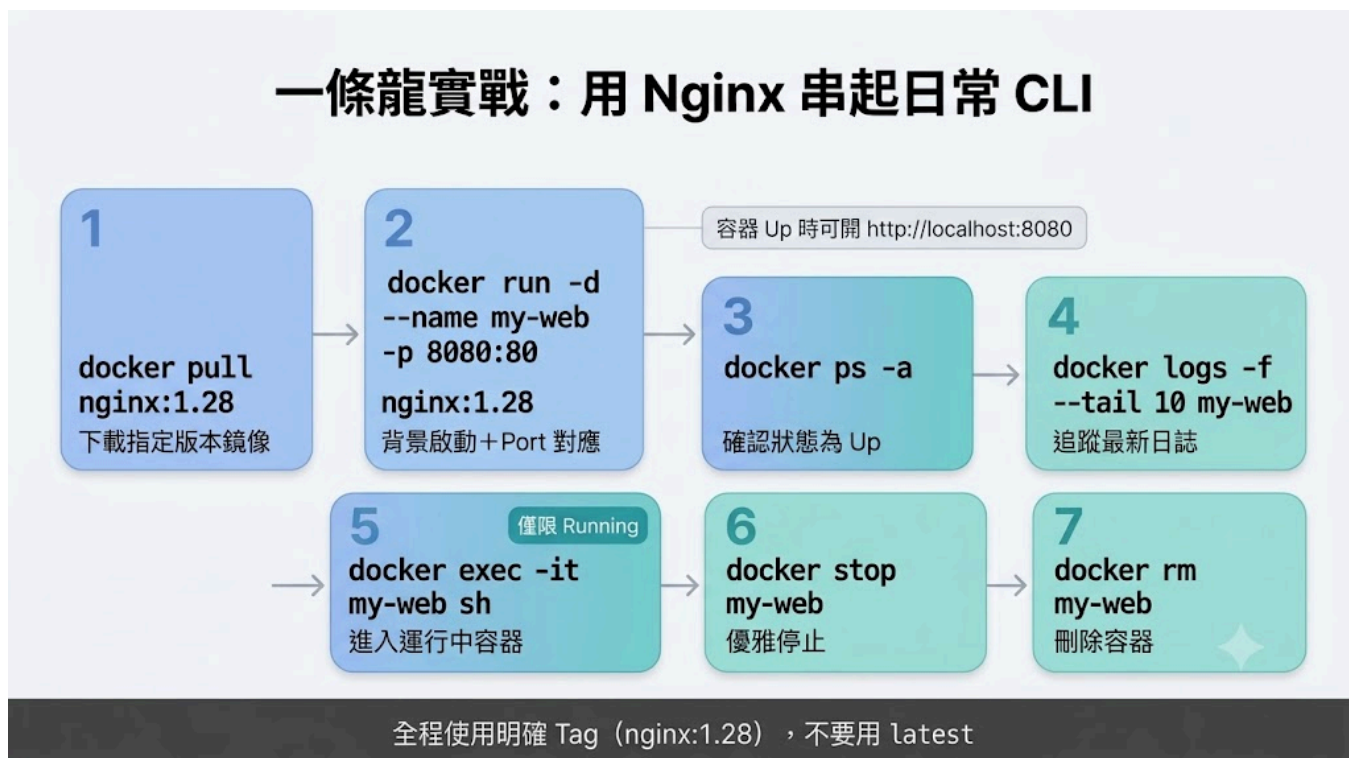
當容器運作異常，或者需要進入容器內檢查設定檔、臨時排錯時，這是最常使用的除錯指令 `docker exec` [官方說明](#)：

```
1 docker exec -it [container_id_or_name] sh
2 # 或者 (如果容器內有安裝 bash)
3 docker exec -it [container_id_or_name] /bin/bash
```

- **運作邏輯**：`exec` 會在已經運行中的容器內部，額外開啟並執行一個新的行程（例如 `sh` 或 `/bin/bash` 殼層），讓你就像登入一台小電腦一樣，可以直接在容器的檔案系統中進行探索與測試。
- **使用前提**：`exec` 只能用於狀態為 `Up / Running` 的容器。若容器已是 `Exited`，執行 `exec` 會失敗——請先用 `docker start` 將它喚醒，或改用 `docker logs` 查看它退出前的輸出。不要把 `exec` 當成「對任何容器都能進去」的萬用指令。

3.5 一條龍實戰：用 Nginx 串起日常指令

把前面各節串成一條可實際跟著做的流程。請使用**明確 Tag**（不要用 `latest`）：



```
1 # 1) 下載指定版本鏡像
2 docker pull nginx:1.28
3
4 # 2) 背景啟動，並做 Port 對應
5 docker run -d --name my-web -p 8080:80 nginx:1.28
6
7 # 3) 確認容器狀態（應看到 Up）
8 docker ps -a
9
10 # 4) 追蹤最新日誌（看幾行後可用 Ctrl+C 結束追蹤）
11 docker logs -f --tail 10 my-web
12
13 # 5) 進入運行中的容器查看（Nginx 鏡像通常有可用的 shell）
14 docker exec -it my-web sh
15 # 在容器內可試：ls /usr/share/nginx/html 然後輸入 exit 離開
16
17 # 6) 瀏覽器驗收（須在容器仍為 Up、尚未 stop / rm 時）
18 # 開啟 http://localhost:8080 確認 Nginx 歡迎頁
19 # 若 Port 被占用，可把步驟 2 的 -p 8080:80 改成其他宿主機埠（如 8081:80）後重跑
20
21 # 7) 優雅停止
22 docker stop my-web
23
24 # 8) 刪除容器（此時才比較好刪對應鏡像，若你要清空間）
25 docker rm my-web
```

步驟 6 請務必在 `stop / rm` 之前完成；容器刪除後 `localhost:8080` 就不會再回應。

3.6 開發日常實用：Docker 指令組合技

在密集的本地開發階段，常會累積大量廢棄容器與未使用鏡像。以下提供兩個常用的指令組合技：

組合技 A：一鍵停止並刪除所有容器

當你想乾淨重置本地環境時，可用以下指令清除本機容器：

```
1 # 停止所有「運行中」的容器（只要 -q，不要 -a）
2 docker stop $(docker ps -q)
3
4 # 刪除所有本機容器（含已停止；-a 才會列出 Exited）
5 docker rm $(docker ps -aq)
```

`-q` (Quiet) 代表僅輸出 Container ID，方便餵給 `stop / rm`。若本機沒有符合條件的容器，`$(...)` 會是空的，指令可能報 `requires at least 1 argument` ——屬預期，不是環境壞掉。

組合技 B：docker system prune 系統大掃除

當本機硬碟空間被 Docker 佔用時，可用 prune 清除無用殘留 [Docker Prune 官方說明](#)：

```
docker system prune -a --volumes
```

docker system prune：你到底會清掉什麼？



執行前務必確認：這些清理通常不可復原

不確定時：先用 `docker system prune`；不要一開始就上 `-a --volumes`

⚠ **高風險警告：** `-a` 會刪除所有未被任何容器使用的鏡像（不只是懸空鏡像）；`--volumes` 會一併刪除未被使用的匿名 Volume（不會清掉未使用的具名卷，例如練習用的 `db_data`）。這代表本機快取的鏡像、匿名測試卷都可能被清掉，且通常無法復原。請只在確認不需要這些本機資源時執行；若不確定，可先執行不帶 `--volumes` 的 `docker system prune`，或省略 `-a` 做較保守的清理。若還要清未使用的具名 Volume，見第六章的 `docker volume prune -a`（範圍不同，勿混為同一指令）。

它會清除：已停止的容器、未使用的網路、未使用的鏡像（加 `-a` 時範圍更大），以及未使用的匿名資料卷（加 `--volumes` 時）。

3.7 本章小結與動手練習

小結

1. 鏡像三件套：`pull` 下載、`images` 列出、`rmi` 刪除；下載與執行時都應明確指定 Tag，避免依賴 `latest`。

2. 容器生命週期：`run` = 建立 + 啟動（每次都是新實例）；重跑用 `start`，關閉用 `stop`，清除用 `rm`。
3. 觀察與排錯：`ps -a` 看狀態、`logs` 看輸出；`exec` 只能進入正在運行的容器。
4. 清理要謹慎：組合技與 `system prune -a --volumes` 很方便，但後者破壞性高，執行前務必確認。

動手練習

1. 分別執行 `docker pull nginx` 與 `docker pull nginx:1.28`，再用 `docker images` 對照：兩者的 TAG 欄位有何不同？為什麼教材建議指定版本？
2. 依 §3.5 跑完 Nginx 一條龍（`run -d -p` → `ps -a` → `logs` → `exec` → `stop` → `rm`）。中途若 `exec` 失敗，先用 `ps -a` 確認容器是否仍在 Running。
3. 故意連續執行兩次 `docker run -d nginx:1.28`（可加不同 `--name`），再用 `ps -a` 觀察：為什麼會出現多個容器？若要重跑「同一個」容器，正確指令是什麼？
4. （選做）在確認不需要本機殘留資源後，先執行較保守的 `docker system prune`；**不要一開始就用 `-a --volumes`**，除非你完全了解後果。

下一章預告：第四章會進入 Dockerfile——學習常用指令、鏡像分層原理，以及如何自己寫出結構清楚、體積精簡的鏡像建置檔。本章的 `pull` / `run` / Tag 習慣，會直接用在之後的建置與驗證流程中。

Reference

1. `docker run` 官方說明 — <https://docs.docker.com/engine/containers/run/>
2. `docker exec` 官方說明 — <https://docs.docker.com/reference/cli/docker/container/exec/>
3. Docker Prune 官方說明 — <https://docs.docker.com/engine/manage-resources/pruning/>



第四章：Dockerfile 指令詳解與鏡像分層原理

本章宗旨： 學會自行撰寫結構清楚、安全且體積精簡的 Dockerfile；先建立鏡像分層心智模型，再掌握常用指令與避坑原則。

第三章你已經會用 CLI 下載、運行與管理鏡像 / 容器；本章要更進一步——自己寫 Dockerfile，把「應用環境」變成可重複建置的藍圖。

學完本章後，你應該能回答：

- 為什麼 Dockerfile 每條指令會形成一個唯讀層？可寫層又在何時出現？
- FROM / RUN / COPY / ADD / CMD / ENTRYPOINT 各自負責什麼？
- 為什麼 WORKDIR 要用絕對路徑、USER 不該用 root？
- 如何寫出一個兼顧快取與安全的基礎 Node.js Dockerfile？

4.1 鏡像分層機制（Image Layering）與 Union FS 原理

在動手撰寫 Dockerfile 之前，先建立關鍵心智模型：鏡像是由多個「唯讀層（Read-Only Layers）」疊加而成的。



1. 什麼是唯讀層（Immutable Layers）？

當你使用 Dockerfile 建置鏡像時，每一條指令通常都會產生一個全新的唯讀層。

- 每一層只記錄與前一層相比的**檔案系統變更**（新增、修改或刪除）。
- 這些唯讀層一旦建置完成，就是**不可變的（Immutable）**。
- 分層帶來的好處是**重用與節省頻寬**：若多個鏡像共享相同基礎層，宿主機通常只需儲存該層一次，有助加速分發與建置。

2. Union FS（聯合檔案系統）與目錄視野

既然鏡像由多個獨立唯讀層組成，容器啟動時應用程式如何讀到完整檔案系統？

- **Union FS（聯合檔案系統）**：Docker 會把這些唯讀層**疊加（Stack）**成一個統一的虛擬目錄檢視。
- **根目錄隔離**：容器行程看到的是這個融合後的檔案系統視角；對行程而言，通常感覺不到「分層」的存在，只會看到一套完整目錄樹。

實務上不必先精通 Union FS 實作細節；重點是記住：**建置產生唯讀層，運行時再疊上可寫層。**

3. 可寫層（Writable Layer）與運行時狀態

當容器被建立並運行（`docker run`）時，Docker 會在唯讀層最頂端，再疊加一個**可寫層（Writable Layer）**——這也呼應第二章對 `docker run` 的說明。

- 容器運行期間的新增、修改或寫入 Log，**主要落在這個可寫層**。
- 底層鏡像唯讀層保持不變，因此同一個鏡像可同時啟動多個彼此隔離的容器。
- **重要提醒**：容器被刪除後，若未接上 Volume 等持久化機制，可寫層內容會一併消失。持久化細節會在第六章展開。

4.2 核心指令詳解與最佳實踐

接下來逐一拆解常用 Dockerfile 指令，以及初學者常踩的效能與安全陷阱。[Dockerfile 官方參考](#) / [Docker 建置最佳實踐](#)

1. FROM：挑選合適與安全的基礎鏡像

`FROM` 是 Dockerfile 的起點，用來指定基礎鏡像並開啟建置階段。

- **實務建議**：優先考慮精簡基礎鏡像。常見選擇是 **Alpine**（如 `FROM alpine:3.21`），體積很小（常在數 MB 等級），有助加快下載並縮小攻擊面。
- **Alpine 注意事項**：並非所有應用都適合 Alpine。它使用 `musl`，部分依賴 `glibc` 的原生模組 / 二進位可能不相容。選基礎鏡像時，要同時考慮體積與相容性。

- 本書示範版本線：正文與 Digest 示範統一釘選 **alpine:3.21**（明確 minor，避免 latest）。Hub 上常有更新的 minor（如 3.22 / 3.23）；實作時可換成當下可用的明確 Tag，觀念相同。核對請至 [Docker Hub alpine](#)。

基礎鏡像怎麼選？Alpine 的優點與取捨

Alpine（精簡）



- 1 下載較快
- 2 攻擊面較小
- 3 常見精簡首選

優先考慮精簡基礎鏡像

相容性注意（musl）



並非所有應用都適合 Alpine

KEY TAKEAWAY

選基礎鏡像時：同時考慮體積與相容性

至少指定明確 Tag（避免 latest）；Digest 鎖定留待第五章

- 版本習慣：至少指定明確 Tag（延續第三章，避免 latest）。若要更嚴格的鎖定，可使用 Digest（SHA-256）；完整的 Pinning 策略會在第五章深入。本章先建立「不要裸用浮動標籤」的意識即可：

```
1 # 範例：使用明確 Tag（本書示範 3.21；可依 Hub 換成當下明確 minor）
2 FROM alpine:3.21
3
4 # 進階（第五章再細講）：以 Digest 鎖定
5 # FROM alpine:3.21@sha256:<請填入官方查得的真實 Digest>
```

- 補充：FROM 之前，全域範圍通常只允許出現 ARG（可用於動態指定基礎鏡像版本）。

2. RUN：建置階段執行命令

RUN 在 docker build 時執行命令，並把結果提交為新的鏡像層。

- RUN（建置時）vs CMD（運行時）：RUN 會在建置階段真正執行並寫入層；CMD 只是宣告容器啟動時的預設主行程，建置時不會執行它。
- Debian / Ubuntu 體系常見錯誤：

```
1 # 錯誤示範：拆成兩層，容易造成過時套件 / 快取問題
2 RUN apt-get update
```

```
3 RUN apt-get install -y curl
```

Docker 建置快取會把兩條指令當成獨立層。[Docker Build Cache 說明](#) 若你只改第二行，第一行的 `apt-get update` 可能被直接重用，導致套件索引過舊。

- 建議寫法：更新、安裝、清理放在同一個 `RUN`：

```
1 # 正確示範：合併指令，並清除 apt lists 以縮小體積
2 RUN apt-get update && apt-get install -y --no-install-recommends \
3     curl \
4     git \
5     && rm -rf /var/lib/apt/lists/*
```

RUN 指令：為什麼要把 apt-get 合併在一起？

錯誤示範：拆成兩個 RUN

```
RUN apt-get install -y curl
```

```
RUN apt-get update
```

快取重用舊的
「apt-get update 層」

之後只修改 install 那一行
(例如改成 nginx)

快取重用舊的 update

可能裝到過時套件／安裝失敗

兩條指令 = 兩層；改第二層時第一層常被直接重用

正確示範：合併同一個 RUN

```
RUN apt-get update && apt-get install -y \
    curl git --no-install-recommends \
    && rm -rf /var/lib/apt/lists/*
```

update + install + 清理在同一層

避免把 apt lists 垃圾打進鏡像

同一層一起失效、一起重建；體積也比較小

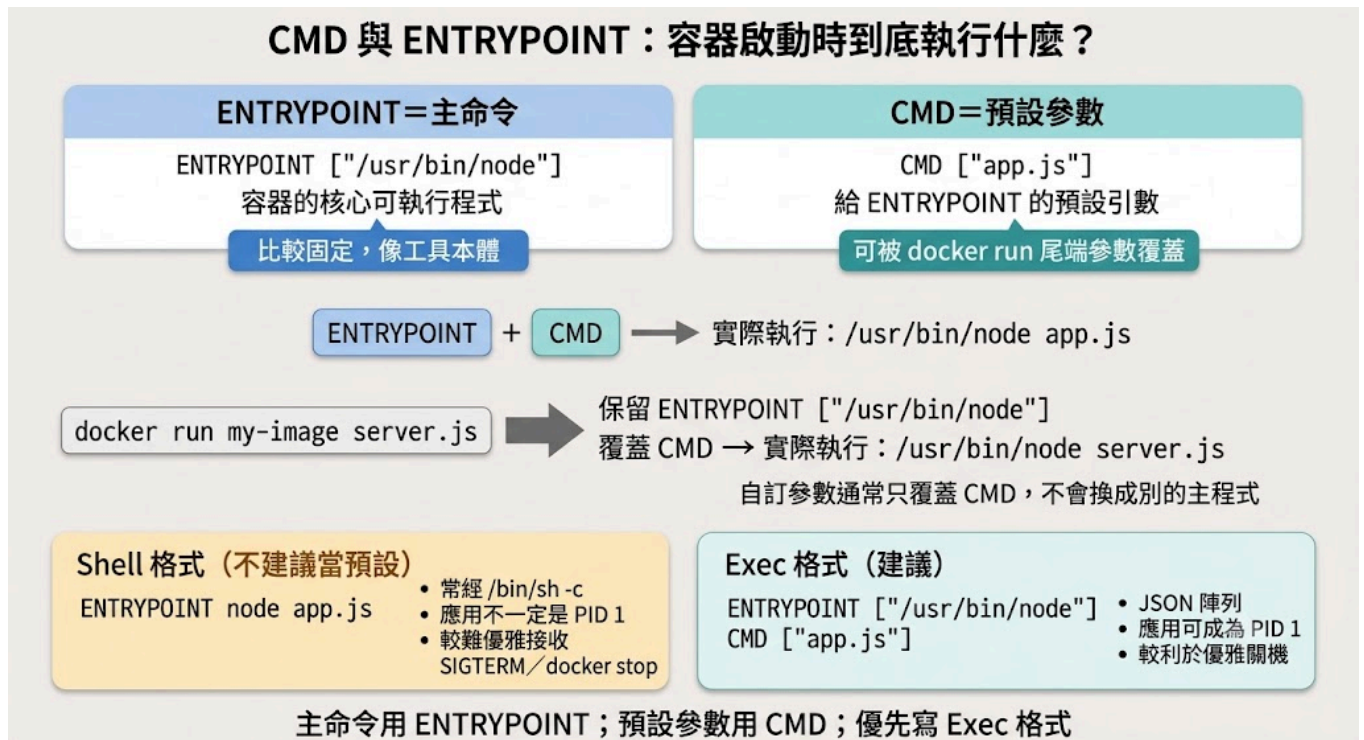
Debian / Ubuntu：把更新、安裝、清理寫在同一個 RUN

- 管線 (pipe) 安全：若 `RUN` 使用 `|`，預設只有最後一個命令的結束碼會影響建置成敗——前面失敗仍可能被當成成功。可用 `pipefail` 讓任一環節失敗就失敗，但預設 `/bin/sh` 常不支援：Debian / Ubuntu 多為 `dash`，Alpine 多為 `BusyBox ash`。需要時請改用 `bash`（基礎鏡像須已安裝；Alpine 需先 `apk add --no-cache bash`）：

```
1 # 僅在需要管線嚴格檢查時切到 bash；鏡像內需有 bash
2 SHELL ["/bin/bash", "-o", "pipefail", "-c"]
3 RUN wget -O - https://example.com/file | wc -l
```

若不想到處改 shell，也可單條寫：`RUN ["/bin/bash", "-o", "pipefail", "-c", "...|..."]`。精簡基底若不想裝 `bash`，就避免在 `RUN` 裡用 `|`，改成可單獨檢查結束碼的寫法。

3. CMD 與 ENTRYPOINT : 容器啟動行為



兩者都與「容器啟動時做什麼」有關，但角色不同：

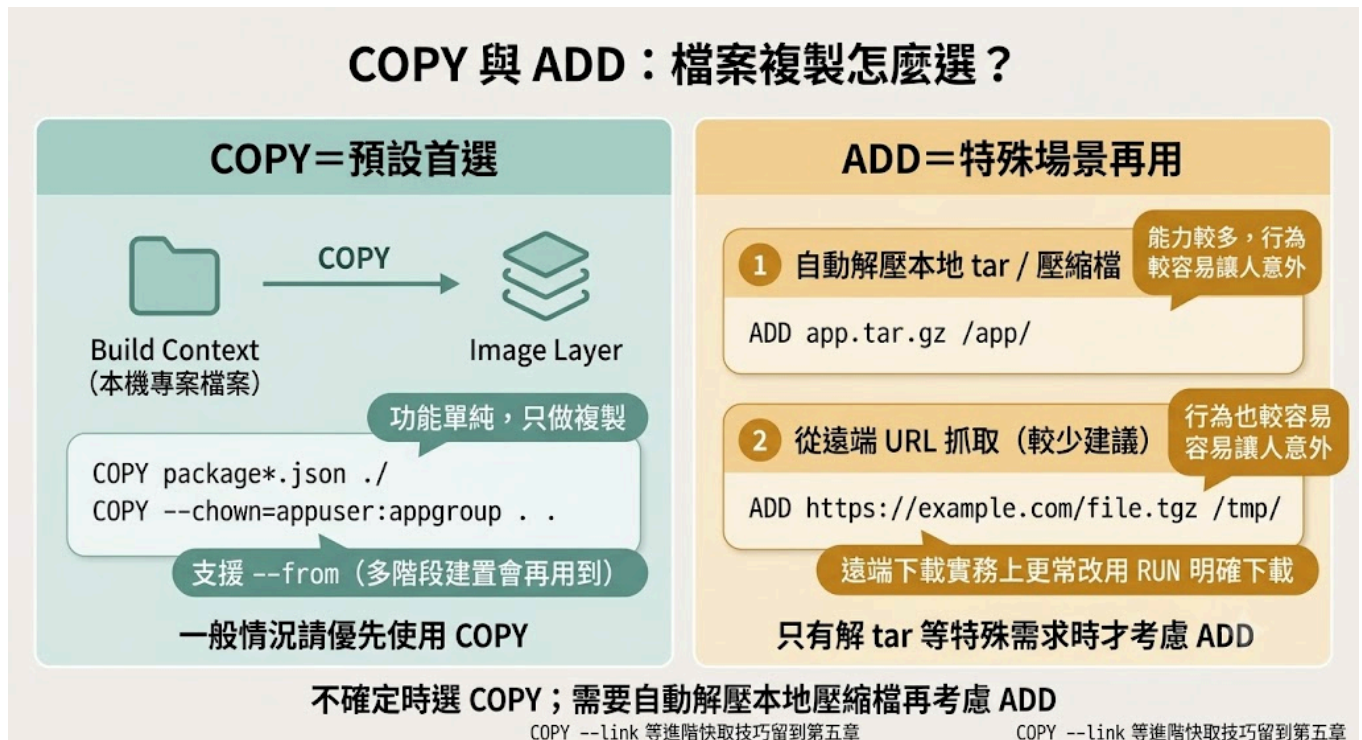
1. **ENTRYPOINT (主命令)**：定義容器的核心可執行程式，讓鏡像用起來像一個固定工具。
 2. **CMD (預設參數)**：常作為 ENTRYPOINT 的預設引數；也可單獨作為預設啟動命令。
- **互動規則**：若執行 `docker run <image> echo hi` 這類寫在鏡像名之後的尾端參數，通常會覆蓋 **CMD**，但仍會交給 **ENTRYPOINT**（在兩者搭配使用時）。寫在鏡像名之前的 `-v / -p` 等則是 Docker CLI 旗標，不要和容器引數混淆。
 - **格式**：優先使用 **Exec 格式 (JSON 陣列)**：

```
1 # 正確示範：Exec 格式
2 ENTRYPOINT ["/usr/bin/node"]
3 CMD ["app.js"]
```

在此搭配下，`docker run <image>` 等於跑 `node app.js`；若改成 `docker run <image> echo hi`，實際會變成 `node echo hi` 並失敗——正好證明尾端參數是接到 **ENTRYPOINT** 之後，而不是整段取代它。

- **避免 Shell 格式作為預設習慣**（如 `ENTRYPOINT node app.js`）。Shell 格式常會經由 `/bin/sh -c` 啟動，可能讓應用程式無法成為 PID 1，因而較難正確接收 `SIGTERM` 等訊號；`docker stop` 時就比較不容易優雅退出。

4. COPY 與 ADD：檔案複製怎麼選



兩者都能把檔案放進鏡像，但建議分工如下：

- **COPY (預設首選)**：功能單純，從建置上下文 (Build Context) 複製檔案 / 目錄。也支援 `--from`，方便之後做多階段建置時跨階段複製產物。
- **ADD (特殊需求再用)**：
 1. 需要把本地 tar 等壓縮檔複製進去時自動解壓；
 2. 需要從遠端 URL 抓取內容 (實務上仍常更建議在 RUN 裡明確下載，以便控管與除錯)。

進階預告 (第五章 / BuildKit)：COPY `--link` 等技巧可進一步優化快取與傳輸；本章先掌握「一般情況用 COPY」即可。

另外，建置上下文若混入 `.git`、說明文件、本機密鑰，不但拖慢建置，也可能把不該進鏡像的東西打進去。實務上會搭配 `.dockerignore` 排除雜訊——細節與快取策略會在第五章系統整理。

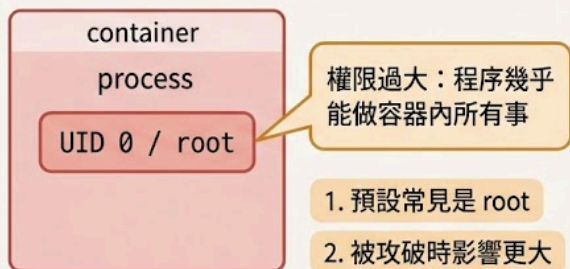
5. ENV、EXPOSE、WORKDIR 與 USER

- **ENV (環境變數)**：設定後會保留在鏡像與容器環境中。
安全提醒：不要把 API Key、密碼寫進 ENV。即使後面 `unset`，歷史唯讀層仍可能留下痕跡。
- **EXPOSE (埠聲明)**：主要是文件 / 溝通用途，宣告應用預設監聽哪些 Port。它不會自動完成宿主機埠對應；實際發布仍需 `docker run -p`。

- **WORKDIR** (工作目錄)：請使用絕對路徑 (如 `/app`)。避免到處寫 `cd /app && ...`；設定後，後續 `RUN` / `CMD` / `COPY` 會以此為預設目錄。
- **USER** (非 root 執行)：
容器內預設常是 `root`，這不安全。應建立低權限使用者，並在啟動應用前切換：

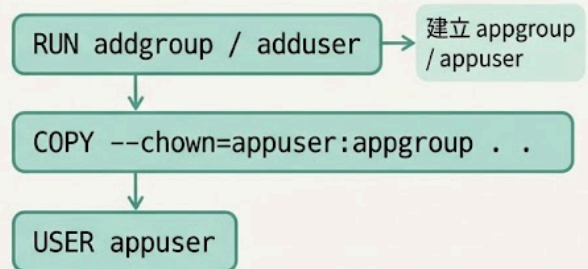
為什麼容器不該用 root 執行？

預設：以 root 執行 (不安全)



方便，但不安全

建議：建立低權限使用者後再切換



啟動應用前切換為非 root

安全底線：以 USER 非 root 執行

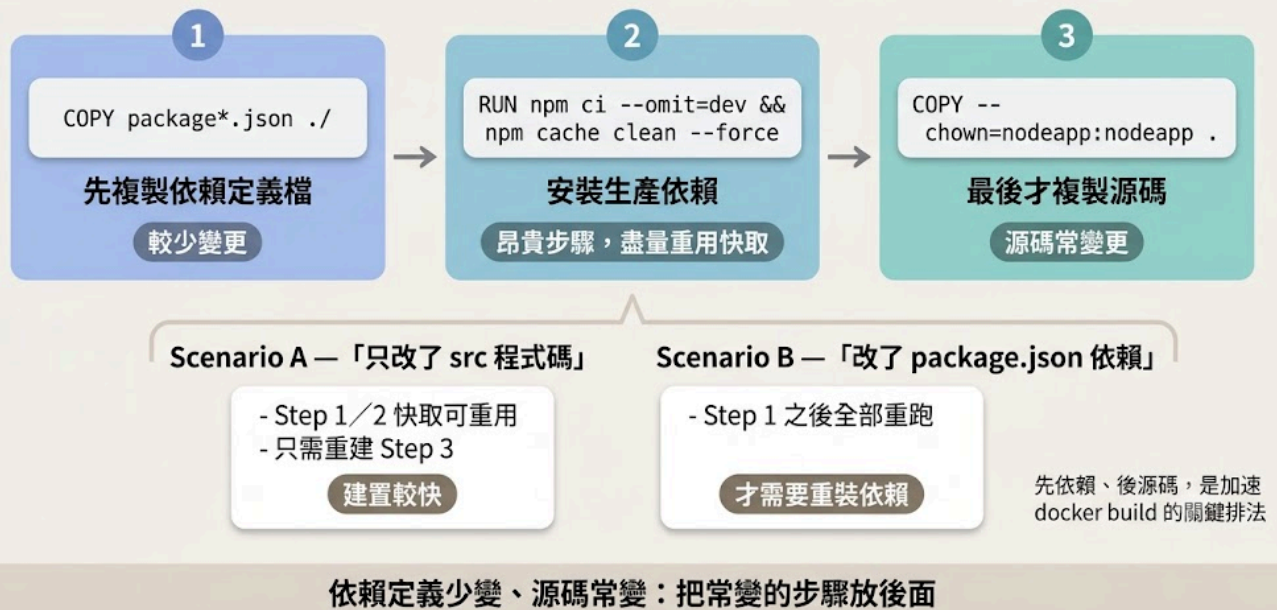
WORKDIR 用絕對路徑；不要把機密寫進 ENV

USER appuser

```
1 # 下列為 Alpine / BusyBox 語法；Debian / Ubuntu 系請改用
2 # groupadd --system appgroup && useradd --system --gid appgroup appuser
3 RUN addgroup -S appgroup && adduser -S appuser -G appgroup
4 WORKDIR /app
5 COPY --chown=appuser:appgroup . .
6 USER appuser
```

4.3 實務練習：撰寫安全、高效的 Node.js Dockerfile

Node Dockerfile 快取順序：為什麼先複製 package.json？



結合以上原則，下面是一個兼顧安全與快取順序的 Node.js Dockerfile 範例（教學用最小 HTTP 服務；不必安裝 Express 或其他框架）：

```
1 # 使用 Alpine 變體，並指定明確 Tag（請依專案更新版本）
2 FROM node:22-alpine
3
4 # 聲明生產環境
5 ENV NODE_ENV=production
6
7 # 建立無特權使用者
8 RUN addgroup -S nodeapp && adduser -S nodeapp -G nodeapp
9
10 # 設定絕對路徑工作目錄
11 WORKDIR /app
12
13 # 快取優化 1：先只複製依賴定義檔
14 COPY package*.json ./
15
16 # 快取優化 2：安裝生產依賴，並清理 npm 快取
17 # （本節最小骨架無第三方依賴；npm ci 仍需要 package-lock.json）
18 RUN npm ci --omit=dev && npm cache clean --force
19
20 # 快取優化 3：再複製其餘原始碼；原始碼常變，依賴層可盡量重用
21 COPY --chown=nodeapp:nodeapp . .
22
23 # 聲明監聽 Port（仍需 docker run -p 才會對外）
24 EXPOSE 3000
25
26 # 以非 root 運行
27 USER nodeapp
28
```

```
29 # Exec 格式，讓 node 成為 PID 1
30 ENTRYPOINT ["node"]
31 CMD ["src/index.js"]
```

最小可驗證流程

若尚無專案，可在空目錄先建立下列最小檔案（與 Dockerfile 同層），再執行建置。正式專案請換成你的應用入口與依賴清單。

package.json :

```
1 {
2   "name": "my-node-app",
3   "private": true,
4   "version": "0.1.0"
5 }
```

src/index.js :

```
1 const http = require("http");
2 const port = Number(process.env.PORT) || 3000;
3 http
4   .createServer((_req, res) => {
5     res.writeHead(200, { "Content-Type": "text/plain; charset=utf-8" });
6     res.end("dockerfile ok\n");
7   })
8   .listen(port, "0.0.0.0");
```

產生 package-lock.json（npm ci 需要它）。不一定要本機安裝 Node——前幾章只保證有 Docker 即可。擇一：

```
1 # 首選（無本機 Node 也能做）：用官方 Node 鏡像在目前目錄寫出 lockfile
2 docker run --rm -v "$(pwd)":/app -w /app node:22-alpine \
3   npm install --package-lock-only
4
5 # 若本機已有 npm，也可以：
6 # npm install --package-lock-only
```

若暫時不想產生 lockfile：可把上方 Dockerfile 的 npm ci 改成 npm install 再 docker build（逃生門）；可重現性較差，教材仍建議養成 npm ci + lockfile 的習慣。

確認目錄已有 Dockerfile、package.json、package-lock.json、src/index.js 後：

```
1 # 建置鏡像
2 docker build -t my-node-app:0.1.0 .
3
```

```
4 # 運行並對應埠號（依應用實際監聽埠調整）
5 docker run --rm -p 3000:3000 my-node-app:0.1.0
```

瀏覽器或 `curl http://localhost:3000` 應看到 `dockerfile ok`。若建置失敗，常見原因包括：建置上下文缺少 `package-lock.json`（`npm ci` 需要它）、基礎鏡像 Tag 不存在、或應用入口路徑與 `CMD` 不一致。

4.4 本章小結與動手練習

小結

1. 分層心智模型：Dockerfile 指令形成唯讀層；`docker run` 時再疊可寫層。刪容器不等於資料自動永久保存。
2. 指令選擇：FROM 選對基礎鏡像；RUN 注意合併與清理；RUN 管線若要用 `pipefail`，確認 shell 是 `bash`（預設 `sh` 常不行）；一般複製用 `COPY`；啟動指令優先 `Exec` 格式的 `ENTRYPOINT` / `CMD`。
3. 安全底線：`WORKDIR` 用絕對路徑；不要把機密寫進 `ENV`；以 `USER` 非 `root` 執行。
4. 與後續章節：Digest 鎖定、`.dockerignore`、多階段建置與更完整快取策略，會在第五章展開。

動手練習

1. 依 §4.3 「最小可驗證流程」建立最簡 Node 專案（或使用自有專案），執行 `docker build` 與 `docker run`，確認容器能啟動並回應。
2. 故意把 `apt-get update` 與 `install` 拆成兩個 `RUN`（若你改用 Debian 基底做實驗），觀察修改安裝清單時快取行為有何不同。
3. 將 `USER` 拿掉後重建並運行，再與加上 `USER` 的版本對照：為什麼教材要求非 `root`？
4. （選做）嘗試把 `CMD` 改成 `Shell` 格式，執行 `docker stop` 時觀察停止行為是否較不乾淨（需自行計時 / 觀察）。

下一章預告：第五章會進入鏡像優化與建置策略——多階段建置、`.dockerignore` / 快取失效控制、以及用 Digest 做更嚴格的版本鎖定，幫你把本章的 Dockerfile 再往「更小、更快、更安全」推進。

Reference

1. Dockerfile 官方參考 — <https://docs.docker.com/reference/dockerfile/>
2. Docker 建置最佳實踐 — <https://docs.docker.com/build/building/best-practices/>
3. Docker Build Cache 說明 — <https://docs.docker.com/build/cache/>

第五章：鏡像優化與建置策略

本章宗旨： 學會企業級的鏡像優化技術，把第四章的 Dockerfile 再往「更小、更快、更可重現」推進，並建立更穩定的容器供應鏈習慣。

第四章你已經會寫結構清楚、兼顧快取與安全的基礎 Dockerfile；本章要處理生產與 CI/CD 更常碰到的三件事：多階段建置瘦身、快取 / 建置上下文控管，以及用 Digest 做更嚴格的版本鎖定。

學完本章後，你應該能回答：

- 多階段建置如何把「編譯環境」與「運行環境」分開？最終鏡像為什麼能明顯變小？
- 延續第四章的 Layer 快取觀念：多階段與 `.dockerignore` 還能再優化什麼？
- 何時該用 `--pull` / `--no-cache`？平常為什麼不該當成預設？
- 為什麼要用 Digest 鎖定基礎鏡像？怎麼查、怎麼寫進 `FROM`？鎖定後如何做多平台 / 更新 / 掃描治理？

5.1 多階段建置 (Multi-stage Builds)

在傳統單階段建置中，常會碰到兩難：編譯需要 JDK、Maven、Node 工具鏈或測試依賴；但這些東西在「運行期」通常用不到。若全部打進最終鏡像，體積會膨脹，攻擊面也跟著變大。

多階段建置允許在同一個 Dockerfile 裡使用多個 `FROM`。每個 `FROM` 開啟一個獨立建置階段 (Stage)；最終只把需要的產物帶進運行鏡像。[多階段建置 \(Multi-stage builds\)](#)

Alpine 提醒 (延續第四章)： 最終階段若用 `*-alpine`，請確認應用 / 原生依賴與 `musl` 相容；不相容時可改用非 Alpine 變體。

1. 核心觀念：編譯環境與運行環境分離

- **Build Stage (建置階段)：** 選功能較完整的基礎鏡像 (含編譯器 / SDK)，負責安裝依賴、編譯與測試。
- **Run Stage (運行階段)：** 選精簡底座 (如 Alpine、Nginx、JRE)。只用 `COPY --from=<stage>` 把編譯產物複製過來；編譯器、暫存檔與多餘工具都留在前一階段，不會進入最終鏡像。
- **最終產物：** 通常可把「數百 MB 級、含整套建置工具」的單階段鏡像，收斂成「數十 MB 級、只含執行所需」的運行鏡像 (實際數字依專案而異)。
- **預設輸出：** 執行 `docker build` 時，預設只有最後一個 Stage 會成為最終鏡像；前面的 Stage 是中間產物，不會自動全部打進結果裡。若要單獨建置某個中間階段 (例如排錯)，可用 `docker build --target <stage_name> -t my-app:debug .`

體積對照（概念）：

單階段：基礎鏡像 + 編譯工具 + 依賴 + 原始碼 / 產物 → 偏大

多階段最終鏡像：精簡底座 + 必要產物（與運行依賴）→ 明顯較小

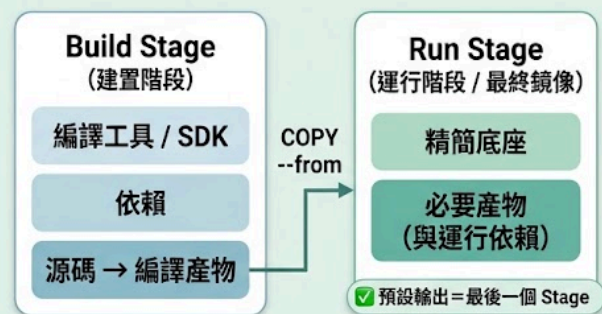
多階段建置：為什麼最終鏡像可以更小？

單階段建置（全部打進最終鏡像）



體積偏大、攻擊面較大

多階段建置（只帶走必要產物）



體積明顯較小

KEY TAKEAWAY: 編譯環境與運行環境分離：最終只保留執行所需

前面的 Stage 是中間產物；可用 `--target` 單獨建置以排錯

也可用階段名稱或序號引用上一階段：

```
1 # 用名稱（建議，可讀性較好）
2 COPY --from=build /app/target/app.jar /app/app.jar
3
4 # 用序號（0 起算；較不直觀，維護時較易搞混）
5 COPY --from=0 /app/target/app.jar /app/app.jar
```

2. 經典應用場景示範

場景 A：Java（Maven + Tomcat）

編譯需要 JDK 與 Maven；運行時通常只需要 Servlet 容器與 JRE：

Tag 提醒： 官方 Tomcat 目前沒有 `*-jre17-alpine` 這類 Tag（Alpine 變體多停在舊 jre8 線）。jre17 最終階段請用 Temurin 變體（如下）；實作前請至 [Docker Hub tomcat](#) 核對當下可用 Tag。這也呼應章首：最終階段不必硬用 Alpine。

```

1 # 第一階段：命名為 build，專門用於編譯
2 FROM maven:3.9-eclipse-temurin-17 AS build
3 WORKDIR /app
4
5 # 可先只複製 pom.xml，再下載依賴，有助重用依賴層快取（進階用法）
6 COPY pom.xml .
7 # RUN mvn -B dependency:go-offline # 選做：先暖依賴快取
8
9 COPY src ./src
10 RUN mvn -B clean package
11
12 # 第二階段：最終運行階段，不帶走 Maven 與完整 JDK 工具鏈
13 FROM tomcat:10-jre17-temurin
14
15 # war 檔名依專案而變；請改成你實際產出的路徑
16 COPY --from=build /app/target/my-app.war /usr/local/tomcat/webapps/

```

場景 B：React 前端靜態網頁

建置需要 Node 來打包成靜態 HTML / JS / CSS；若不做 SSR，運行時通常不需要 Node，用 Nginx 託管即可：

```

1 # 第一階段：Node 環境編譯
2 FROM node:22-alpine AS builder
3 WORKDIR /app
4
5 COPY package*.json ./
6 RUN npm ci
7
8 COPY . .
9 RUN npm run build
10 # 產出目錄名稱依專案而異（常見 dist/ 或 build/）
11
12 # 第二階段：Nginx 靜態託管
13 FROM nginx:1.28-alpine
14
15 COPY --from=builder /app/dist /usr/share/nginx/html
16 EXPOSE 80

```

`npm ci` 需要 lockfile（如 `package-lock.json`）一併存在於建置上下文中。

5.2 建置快取與 Layer 順序優化

承接第四章：Layer 快取失效規則、以及「先 `package.json` 再原始碼」的排法，第四章已說明。本節重點是把它放進多階段情境，並補上 `.dockerignore` 與強制重建參數的實務界線。

Docker 建置時依序執行指令；某一層快取失效後，其後續下游層通常也必須重建。[建置快取說明](#)

1. 延續第四章：依賴與原始碼拆開複製

許多新手會寫成：

```
1 # 錯誤示範：糟糕的快取順序
2 FROM node:22-alpine
3 WORKDIR /app
4 COPY . .
5 RUN npm install
```

只要改一行原始碼，`COPY . .` 失效，後面的依賴安裝也常被拖著重跑。

建議寫法：先複製少變的依賴定義，再複製常變的原始碼（與第四章一致；下列為「只跑生產依賴、不再編譯」的單階段示意）：

```
1 # 建議：優化後的快取順序（生產依賴示意）
2 FROM node:22-alpine
3 WORKDIR /app
4 COPY package*.json ./
5 RUN npm ci --omit=dev
6 COPY . .
```

防混淆：若此階段還要編譯（例如需要 TypeScript 的 `tsc` 等 `devDependencies`），**不要加** `--omit=dev`；應先完整 `npm ci` 完成編譯，再於另一個「只裝生產依賴」的階段使用 `--omit=dev`（見 §5.4）。

在多階段 Dockerfile 裡，同一個原則仍適用：把「安裝依賴」放在「複製大量原始碼」之前；各 Stage 的快取大致獨立，改 runner 階段不一定會逼 builder 整段重跑，但 Stage 內的順序依然重要。

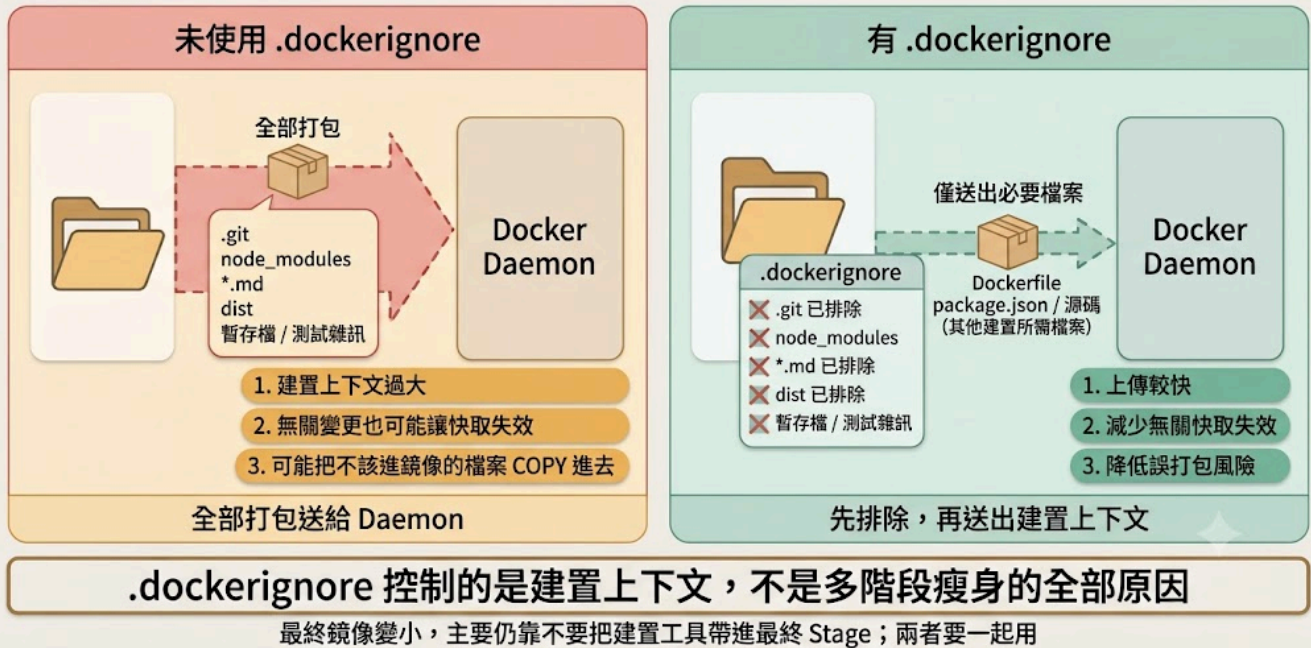
2. 善用 `.dockerignore`

執行 `COPY . .` 時，Docker 會把建置上下文（Build Context）送給 Daemon。若未過濾，`.git`、本地 `node_modules`、測試產物等可能被送進上下文，造成：

1. 上傳 / 傳輸變慢；
2. 無關檔案變更也可能讓依賴 `COPY` 的層快取失效；
3. 更糟時，機密或不該進鏡像的檔案被 `COPY` 進去。

注意：`.dockerignore` 直接控制的是「哪些檔案進入建置上下文」，**不是**多階段瘦身的全部原因。最終鏡像變小，主要仍靠「不要把建置工具帶進最終 Stage」；兩者要一起用。[建置上下文與 .dockerignore](#)

.dockerignore：縮小建置上下文



在專案根目錄建立 `.dockerignore`（語法近似 `.gitignore`）：

```
1 # .dockerignore 範例
2 .git
3 .github
4 node_modules
5 npm-debug.log
6 dist
7 *.md
8 *.tmp
```

3. `--pull` 與 `--no-cache`：何時用、何時別當預設

- `--pull`：建置前重新檢查 / 拉取 FROM 所用基礎鏡像的較新內容（依 Tag 解析結果），避免一直沿用過舊的本地基底。
- `--no-cache`：停用建置快取，所有指令重跑。

```
1 # 排錯或釋出前需要「乾淨重編」時再用
2 docker build --pull --no-cache -t my-app:0.1.0 .
```

平常開發請以快取為主。每次都 `--no-cache` 會讓建置明顯變慢；這兩個旗標較適合：懷疑快取髒掉、要驗證「從零建置也會過」、或要刻意對齊較新基底時。

5.3 供應鏈安全與版本鎖定 (Pinning)

在生產與 CI 中，可重現（今天建得出來，之後盡量建出一樣的底座）比「永遠自動追最新 Tag」更重要。[Docker 建置最佳實踐（含 Pinning）](#)

1. 可變 Tag 的風險

`FROM alpine:3.21` 或 `FROM node:22-alpine` 這類 Tag 可能被發行商改指向新的鏡像內容（安全修補或行為變更）。結果是：同一份 Dockerfile，隔一段時間重建，底座可能已不同，輕則行為改變，重則建置失敗。

與第四章銜接：本書 Digest / Alpine 示範統一用 `alpine:3.21`。若你改用較新的明確 minor（如 3.22 / 3.23），下列查 Digest 指令請一併替換 Tag；重點是「明確 Tag + 必要時 Digest」，不是追最新數字。

這也延續第三章對 `latest` 的提醒：浮動標籤方便，但可重現性較差。

2. 用 Digest (SHA-256) 鎖定

更嚴格的作法是在 Tag 之外再加上不可變的 Digest：

- 1 # 範例格式 (Digest 請自行查官方 / Registry 的真實值，勿照抄過期雜湊)
- 2 **FROM** `alpine:3.21@sha256:<請填入官方查得的真實 Digest>`

即使同名 Tag 被覆寫，只要 Digest 不變，你鎖到的仍是同一份內容。

為什麼要用 Digest 鎖定基礎鏡像？

LEFT PANEL 只使用可變 Tag

Tag: `alpine:3.21` or `node:22-alpine`



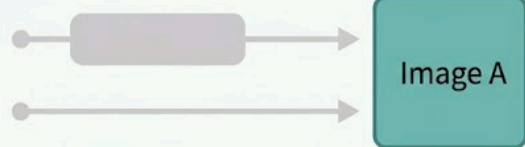
⚠ Tag 可能被發行商改指向新內容

⚠ 同一份 Dockerfile，重建結果可能不同

方便，但可重現性較差

RIGHT PANEL Tag + Digest 鎖定

`alpine:3.21@sha256:...`



✓ Digest 不可變

✓ 鎖定同一份鏡像內容

可重現性較高

KEY TAKEAWAY 明確 Tag 是基本盤；Digest 提供更強的可重現性
先跑通再用 `imagetools inspect / pull` 查 Digest；更新安全修補時要主動改 Digest

如何查 Digest（擇一即可）：`docker buildx imagetools inspect`

```
1 # 方式 A：用 buildx 檢視（常見於教材 / CI）
2 docker buildx imagetools inspect alpine:3.21
3
4 # 方式 B：pull 後從輸出或 docker image inspect 查看 RepoDigests
5 docker pull alpine:3.21
6 docker image inspect alpine:3.21 --format '{{index .RepoDigests 0}}'
```

```
Docker Workshop
docker image inspect alpine:3.21 --format '{{index .RepoDigests 0}}'
3.21: Pulling from library/alpine
Digest: sha256:48b0309ca019d89d40f670aa1bc06e426dc0931948452e8491e3d65087abc07d
Status: Image is up to date for alpine:3.21
docker.io/library/alpine:3.21
alpine@sha256:48b0309ca019d89d40f670aa1bc06e426dc0931948452e8491e3d65087abc07d
~/Desktop/docker-shot/ch05
```

把查到的 `sha256:...` 寫進 `FROM image:tag@sha256:...` 即可。

3. Digest 治理：鎖定之後怎麼維護

Digest 不是「釘死一次就結束」。可重現性與安全修補之間要有節奏：

(1) 多平台 Digest 別混用

許多官方 Tag (如 `alpine:3.21`、`node:22-alpine`) 在 Registry 上是 `manifest list / index` (一份 Tag 底下掛多個架構)。`imagetools inspect` 常會同時看到：

- **Index / list 的 Digest**：代表「這個 Tag 的多平台目錄」
- **單一平台 (如 `linux/amd64`、`linux/arm64`) 的 Digest**：只對應某一個架構的實際鏡像

實務建議：

- 本機 (例如 Apple Silicon → `arm64`) 與 CI (常為 `amd64`) 若各自 `pull` 再抄 `RepoDigests`，有可能拿到不同的平台 Digest。
- 團隊應約定：鎖 **index Digest** (多平台同一鎖定點)，或在 CI / 本機都對同一目標平台查 Digest 再寫進 `Dockerfile`。
- 查閱時可指定平台，例如：

```
1 docker buildx imagetools inspect alpine:3.21 --format '{{json .Manifest}}'
2 # 或在輸出中對照 Platform 與對應的 Digest，避免把 arm64 的雜湊用到只建 amd64 的流水線
```

(2) 定期更新流程 (建議最小節奏)

1. **鎖定**：開發 / CI 使用 `image:tag@sha256:...` (先跑通 Tag，再加 Digest)。
2. **排程複查**：依風險訂週期 (例如每兩週或每月；有 CVE 通報時提前)，對照官方 Tag 是否已指向新內容。
3. **驗證後改鎖**：在測試 / CI 用新 Digest 建置與跑煙霧測試，通過後再合併更新後的 `FROM`。
4. **記錄**：PR / 變更說明寫清「為何更新、舊→新 Digest」，方便回溯。

(3) 掃描與政策 (工具點到即可)

- 建置或推送前對**最終鏡像**做弱點掃描 (如 Docker Scout、Trivy、Grype 等；選團隊既有工具即可)。
- 基底過期 / 已知高危 CVE 時，優先更新基底 Digest，而不是只掃應用依賴。
- 政策類檢查 (例如「基底是否過舊」) 可當提醒，**不能取代人工確認與回歸測試**。

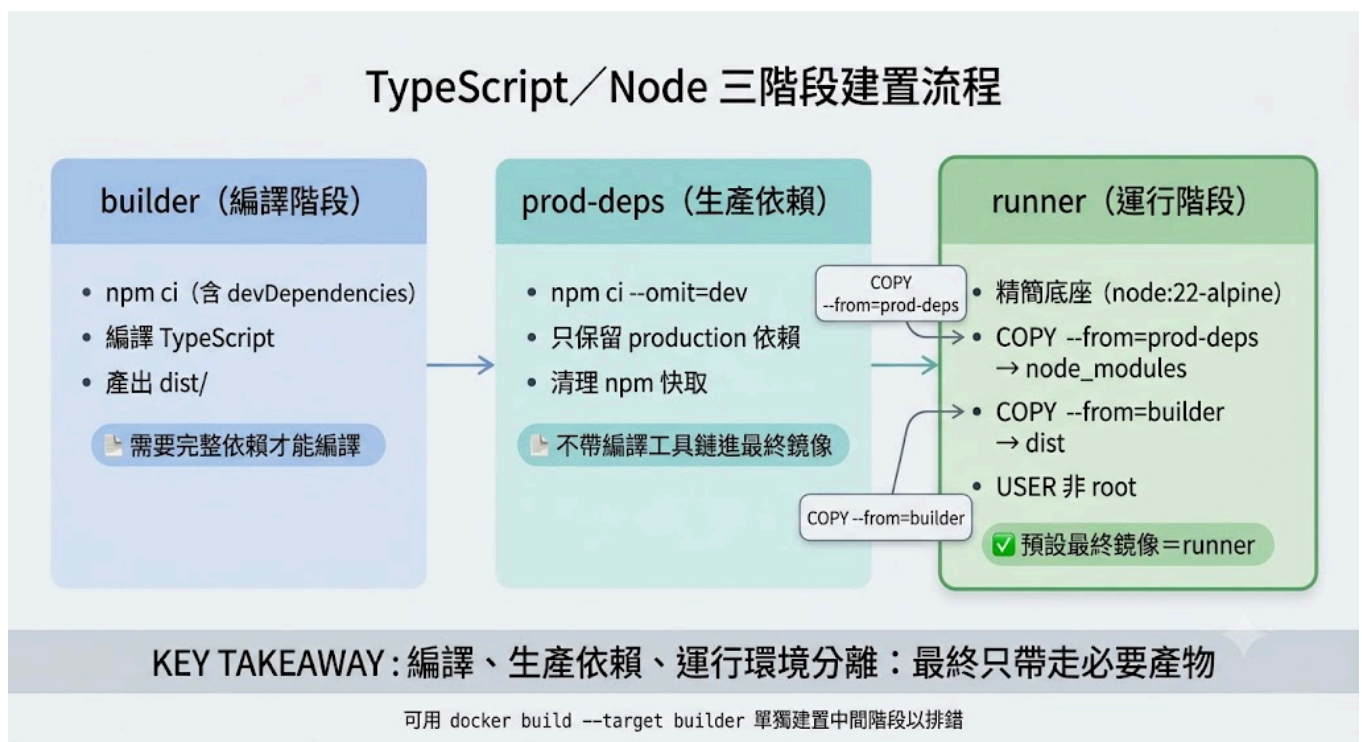
一句話：Digest 解決「建得出一樣的底座」；治理解決「底座該換的時候有人換、換完有人驗」。

5.4 實務範例：TypeScript / Node 多階段 Dockerfile

跟做前提（請自備專案）：本節示範多階段拆分與快取順序，**不是**第四章 / 第七章那種「空目錄 + 最小骨架就能整段貼上」的入門流程。請準備已能在本機（或容器外）跑通的 TypeScript / Node 專案：至少含 `package.json`、`package-lock.json`、可產出 `dist/index.js` 的 `build` 腳本，以及根目錄的 `.dockerignore`。若暫時沒有 TS 專案，可先精讀 Dockerfile 結構與大小對照圖，或改用你已有的前端建置專案對照「場景 B」；動手練習亦同此前提。

以下範例結合本章重點：多階段分離、依賴 / 原始碼快取順序、非 root、以及 Digest 占位。三個階段分別是：

1. **builder**：安裝完整依賴並編譯 TypeScript
2. **prod-deps**：只安裝 production 依賴
3. **runner**：精簡運行環境，只帶入 `node_modules` 與 `dist`（預設最終鏡像）



先跑通再鎖定：初學驗證可先把下方 `FROM` 寫成 `FROM node:22-alpine`（不要加 `@sha256:...`），確認 `build / run` 流程沒問題後，再用 §5.3 的方法查 Digest 並加上鎖定。

實戰 Dockerfile

```
1 # syntax=docker/dockerfile:1
2 # 上方 syntax 列會啟用 Docker BuildKit 的 Dockerfile 前端，以便使用較新指令選項（如 COPY --link）
```

```

3
4 # 共用基底
5 # 初學：可先改成 FROM node:22-alpine AS base
6 # 進階：填入真實 Digest，例如 FROM node:22-alpine@sha256:<真實 Digest> AS base
7 FROM node:22-alpine AS base
8 WORKDIR /app
9
10 # =====
11 # 階段一：編譯 (builder)
12 # =====
13 FROM base AS builder
14 COPY package*.json ./
15 RUN npm ci
16 COPY . .
17 RUN npm run build
18
19 # =====
20 # 階段二：生產依賴 (prod-deps)
21 # =====
22 FROM base AS prod-deps
23 COPY package*.json ./
24 RUN npm ci --omit=dev && npm cache clean --force
25
26 # =====
27 # 階段三：運行 (runner) ← 預設最終鏡像
28 # =====
29 # 沿用 base，Digest 只需在上方 AS base 鎖一次
30 FROM base AS runner
31 ENV NODE_ENV=production
32
33 RUN addgroup -S nodeapp && adduser -S nodeapp -G nodeapp
34
35 # COPY --link 為進階選項 (需 BuildKit)：有助讓複製進來的內容以更獨立的層處理，
36 # 改善部分快取 / 傳輸情境；不是「一定要會」的入門必備，可先當加分項。
37 COPY --link --from=prod-deps --chown=nodeapp:nodeapp /app/node_modules ./node_modules
38 COPY --link --from=builder --chown=nodeapp:nodeapp /app/dist ./dist
39
40 EXPOSE 3000
41 USER nodeapp
42
43 ENTRYPOINT ["node"]
44 CMD ["dist/index.js"]

```

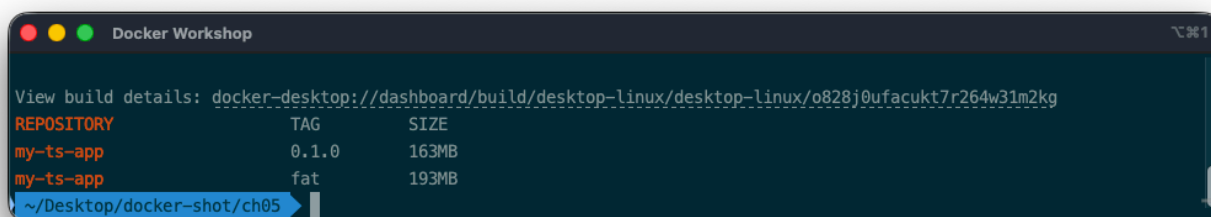
最小可驗證流程

在已符合上方前提的專案根目錄放入本節 Dockerfile（與 `.dockerignore`）後：

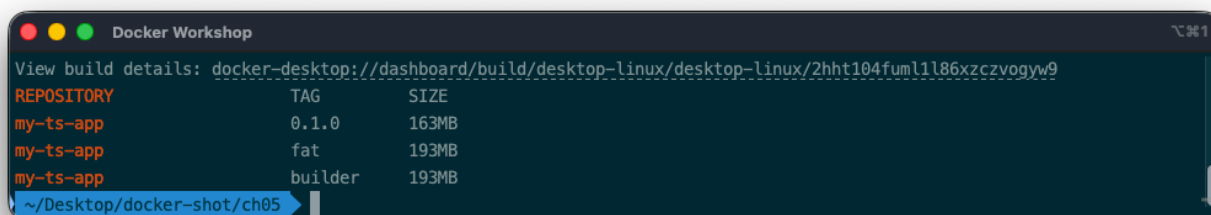
```

1 docker build -t my-ts-app:0.1.0 .
2 docker run --rm -p 3000:3000 my-ts-app:0.1.0
3
4 # 選做：只建置中間階段以便排錯
5 # docker build --target builder -t my-ts-app:builder .

```



```
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/o828j0ufacukt7r264w31m2kg
REPOSITORY      TAG      SIZE
my-ts-app        0.1.0    163MB
my-ts-app        fat      193MB
~/Desktop/docker-shot/ch05
```



```
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/2hht104fum1l186xzczyvgyw9
REPOSITORY      TAG      SIZE
my-ts-app        0.1.0    163MB
my-ts-app        fat      193MB
my-ts-app        builder  193MB
~/Desktop/docker-shot/ch05
```

若建置失敗，先檢查：lockfile 是否存在、`npm run build` 產出路徑是否為 `dist/`。若你已改為 Digest 鎖定，再確認 `@sha256:...` 為查得的真實值（占位字串無法建置）。

5.5 本章小結與動手練習

小結

1. 多階段建置：用多個 `FROM` 分離編譯與運行；預設只有最後一個 Stage 成為最終鏡像；以 `COPY --from` 帶入必要產物，體積與攻擊面通常都能下降。
2. 快取與上下文：延續「先依賴、後原始碼」；編譯階段勿貿然 `--omit=dev`；用 `.dockerignore` 縮小建置上下文，減少無關失效與誤打包。
3. 強制重建：`--pull` / `--no-cache` 是排錯與乾淨重編工具，不是日常預設。
4. Pinning 與治理：明確 Tag 是基本盤；Digest 提高可重現性。注意多平台 Digest 別混用；鎖定後要有定期更新+掃描，而不是釘死一次就不管（先跑通再鎖定）。
5. 與後續章節：容器刪了資料就沒了——持久化儲存與容器網路，會在第六章展開。

動手練習

1. 在自備的前端 / Node / TS 專案（見 §5.4 前提）放入場景 B 或 §5.4 Dockerfile，比較「單階段含建置工具」與「多階段最終鏡像」的 `docker images` 大小差異。
2. 故意拿掉 `.dockerignore` 中的 `node_modules`，觀察建置上下文與快取行為有何不同（可搭配建置輸出體感）。

3. 用 `docker buildx imagetools inspect` 查一個基礎鏡像 Digest，寫進 `FROM`，確認可成功 `docker build`。並在輸出中指出：你鎖的是 `index` 還是某一個 `平台` Digest？若本機是 `arm64`、`CI` 是 `amd64`，會有什麼風險？
4. (選做) 用 `docker build --target builder` 建出中間階段，再與預設最終鏡像對照差異。
5. (選做) 拿掉 `COPY --link`，改成一般 `COPY --from`，確認行為仍正確，並理解它是加分項而非必要條件。
6. (選做) 為「每月更新一次基底 Digest」寫三步檢查清單（查新 Digest → 建置 / 煙霧測試 → 合併），並想一項你會搭配的掃描工具名稱即可。

下一章預告：第六章會進入持久化資料與容器網路——Volume / Bind Mount、以及容器如何連線與對外發布，補上「鏡像優化之後，資料與網路怎麼接」的實務。

Reference

1. 多階段建置 (Multi-stage builds) — <https://docs.docker.com/build/building/multi-stage/>
2. 建置快取說明 — <https://docs.docker.com/build/cache/>
3. 建置上下文與 `.dockerignore` — <https://docs.docker.com/build/concepts/context/>
4. Docker 建置最佳實踐 (含 Pinning) — <https://docs.docker.com/build/building/best-practices/>
5. `docker buildx imagetools inspect` — <https://docs.docker.com/reference/cli/docker/buildx/imagetools/inspect/>

預覽到此

以上是免費預覽的範圍：第一章至第五章。

後續章節會處理持久化資料與容器網路、Compose 多容器、FastAPI 本地開發、執行時與 Podman，以及從 Compose 走到 Kubernetes。實驗則在 Google Cloud 的 Ubuntu 虛擬機上自架 Docker、n8n 與 Supabase。