部署與維護講義

第一章-Docker目錄

Docker 簡介

為什麼要用 Docker？

在團隊協作開發應用程式時，常會遇到「在我電腦上可以跑，但在你電腦上卻不行」的困境。主要原因包括：

- **環境不一致**：開發者、組員和伺服器可能使用不同的作業系統（Windows、Mac、Linux）
- **套件版本衝突**：不同專案可能需要不同版本的相同套件，造成互相干擾
- **相依性問題**：缺少必要的執行環境、套件或設定檔
- **路徑與權限差異**：不同系統的檔案路徑結構和權限設定不同

過去我們使用虛擬機來解決隔離問題，但虛擬機需要模擬整個作業系統，資源消耗大且啟動緩慢。

Docker 容器技術提供了更好的解決方案：

1. **輕量化**：只打包應用程式及其相依環境，不需要完整的作業系統，啟動速度快且資源消耗少
2. **環境統一**：確保開發、測試和生產環境完全一致，解決「在我電腦上可以跑」的問題
3. **高度隔離**：每個容器獨立運行，不會互相干擾
4. **易於遷移**：透過映像檔可以快速在不同機器上部署相同環境
5. **簡化協作**：組員只需執行一個指令就能獲得完全相同的開發環境，不需要手動安裝各種套件

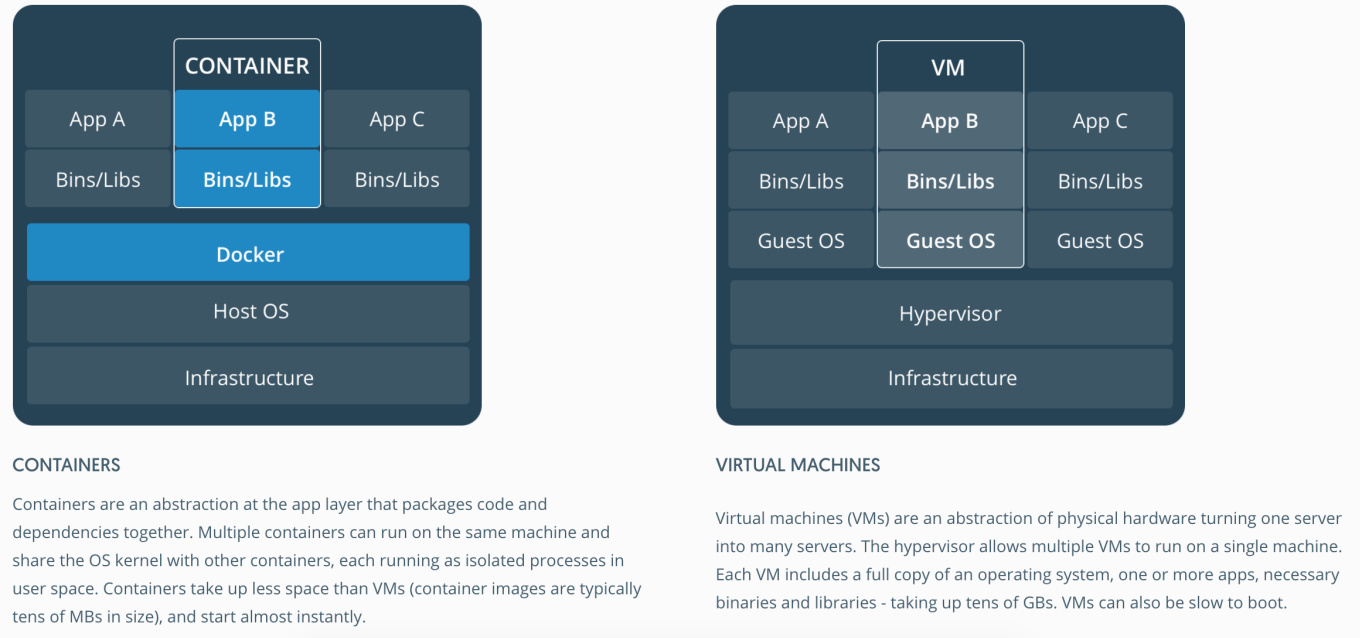
例如，當你用 Python 3.10 開發了一個 Flask 應用，只要將其打包成 Docker 映像檔，組員就能在 Mac、Windows 或 Linux 上以相同方式執行，無需擔心 Python 版本、套件安裝或系統差異問題。

為什麼我們常說Docker就會輕量化？

Docker 之所以輕量化，主要有以下幾個原因：

1. **共享作業系統核心**：Docker 容器共享宿主機的作業系統核心，而不是每個容器都需要一個完整的作業系統。這大大減少了資源消耗，因為不需要為每個容器分配獨立的 OS 資源。
2. **精簡的映像檔**：Docker 映像檔通常只包含應用程式及其必要的依賴，而不包含不必要的系統元件。這使得映像檔更小，下載和啟動速度更快。
3. **快速啟動**：由於 Docker 容器不需要啟動完整的作業系統，它們可以在幾秒鐘內啟動，這對於開發和測試環境非常有利。

Docker 架構圖



Docker 採用客戶端-伺服器架構，主要組件包括：

- **Docker Client**: 使用者與 Docker 互動的介面
- **Docker Daemon**: 負責建立、運行和管理容器的背景服務
- **Docker Registry**: 存儲和分發映像檔的倉庫

Docker 核心概念

容器 (Container)

容器是 Docker 的核心單位，它是一個輕量級、獨立的執行環境，包含應用程式及其所有依賴。容器利用宿主機的作業系統核心，實現資源隔離和高效利用。 Docker Container

映像檔 (Image)

映像檔是用來建立容器的藍圖，包含應用程式的程式碼、運行時、庫和設定檔。映像檔是不可變的，可以通過 Dockerfile 來定義和建立。

Dockerfile

Dockerfile 是一個文本檔案，包含一系列指令，用於自動化建立 Docker 映像檔的過程。開發者可以在 Dockerfile 中指定基礎映像檔、安裝軟體包、複製檔案等操作。

Registry

Registry 是一個集中存儲和分發 Docker 映像檔的服務。Docker Hub 是最常用的公共 Registry，開發者也可以搭建私有 Registry 來管理自己的映像檔。

Docker 安裝

Docker 可以在多種作業系統上安裝，包括 Windows、macOS 和各種 Linux 發行版。安裝過程通常包括下載 Docker 安裝程式，執行安裝，並完成初始設定。

下載點: https://docs.docker.com/get-docker/

Docker 基本指令

映像檔管理

- `docker pull <image>`: 從 Registry 下載映像檔。
- `docker images`: 列出本地所有映像檔。
- `docker search <keyword>`: 在 Docker Hub 上搜尋映像檔。
- `docker rmi <image>`: 刪除本地映像檔。
- `docker save -o <output-file.tar> <image>`: 將映像檔匯出為 tar 檔案，方便離線傳輸或備份。
- `docker load -i <input-file.tar>`: 從 tar 檔案載入映像檔，與 `docker save` 配對使用，可在無網路環境下部署映像檔。

容器管理

- `docker run <image>`: 建立並啟動一個新的容器。
- `docker ps`: 列出正在運行的容器。
- `docker stop <container>`: 停止一個正在運行的容器。
- `docker rm <container>`: 刪除一個已停止的容器。

網路管理

- `docker network ls`: 列出所有 Docker 網路。
- `docker network create <network>`: 建立一個新的 Docker 網路。
- `docker network inspect <network>`: 顯示 Docker 網路的詳細資訊,包括連接的容器、子網路配置、閘道器等。
- `docker inspect <container|image>`: 顯示容器或映像檔的詳細資訊,包括設定、網路、掛載點等,以 JSON 格式輸出。

如何把一個python程式打包成Docker Image

1. 建立一個新的目錄，並將你的 Python 程式碼放入該目錄。
2. 在該目錄中建立一個名為 `Dockerfile` 的檔案，內容如下：

```
# 使用官方 Python 映像檔作為基礎映像檔
FROM python:3.9-slim
# 設定工作目錄
WORKDIR /app
# 複製需求檔案並安裝依賴
COPY requirements.txt .
```

```
RUN pip install --no-cache-dir -r requirements.txt
# 複製應用程式檔案
COPY . .
# 暴露應用程式埠
EXPOSE 5000
# 設定環境變數
ENV FLASK_APP=app.py
# 啟動應用程式
CMD ["flask", "run", "--host=0.0.0.0"]
```

常用指令參數

- `-t <tag>`: 為建立的映像檔指定標籤（tag），方便管理和識別。
- `.`: 指定 Dockerfile 所在的目錄,通常使用當前目錄（`.`）。
- `-p <主機端口>:<容器端口>`: 端口映射，將容器內部端口映射到主機端口，例如 `-p 8080:80` 將容器的 80 端口映射到主機的 8080 端口。
- `-d` 或 `--detach`: 在背景模式運行容器,容器啟動後會立即返回命令提示符,不會顯示容器的輸出日誌。適合需要長時間運行的服務。
- `-v` 或 `--volume`: 掛載資料卷,將主機的目錄或檔案掛載到容器內部,格式為 `-v <主機路徑>:<容器路徑>`。例如 `-v /data:/app/data` 將主機的 `/data` 目錄掛載到容器的 `/app/data` 目錄,實現資料持久化和共享。
- `-i` 或 `--interactive`: 保持容器的標準輸入 (STDIN) 開啟,即使沒有附加終端機也能與容器互動。通常與 `-t` 參數一起使用（`-it`）來建立一個互動式終端機會話,適合需要輸入指令或調試的情境。例如 `docker run -it ubuntu bash` 可以進入 Ubuntu 容器的互動式 bash 終端機。 增加-f參數的說明
- `-f <Dockerfile 路徑>`: 指定要使用的 Dockerfile 檔案路徑。如果你的 Dockerfile 不在當前目錄或名稱不是預設的 `Dockerfile`，可以使用此參數來指定。例如 `-f ./path/to/Dockerfile`。

`-f` 參數說明

`-f` 或 `--format` 參數用於格式化輸出結果,讓你可以從 JSON 格式的詳細資訊中提取特定欄位。

使用範例

```
docker inspect -f '{{.Mounts}}' tomcat
```

這個指令會:

- 使用 `-f` 參數指定輸出格式
- `{{.Mounts}}` 是 Go template 語法,用於提取容器的掛載資訊
- 只顯示 Mounts 欄位的內容,而不是完整的 JSON 輸出

其他常用格式範例

```
# 查看容器 IP 位址
docker inspect -f '{{.NetworkSettings.IPAddress}}' container_name
```

```
# 查看容器狀態
docker inspect -f '{{.State.Status}}' container_name

# 查看容器映像檔
docker inspect -f '{{.Config.Image}}' container_name
```

優點

- 快速提取所需資訊,不需要手動解析完整的 JSON 輸出
- 可以在腳本中使用,方便自動化處理
- 支援 Go template 語法,可以進行複雜的格式化操作

如何把 Docker Image Push 到 Docker Hub

1. 登入 Docker Hub 帳號

```
docker login
```

2. 標記映像檔

```
docker tag <local-image>:<tag> <dockerhub-username>/<repository>:<tag>
```

3. 推送映像檔到 Docker Hub

```
docker push <dockerhub-username>/<repository>:<tag>
```

- reference: https://ithelp.ithome.com.tw/articles/10191139

Docker Desktop 使用介面介紹

Containers

- 顯示目前所有的容器狀態（運行中、停止等）
- 提供啟動、停止、刪除容器的操作按鈕

Images

- 顯示本地所有的映像檔

Volumes

- 顯示所有的資料卷，方便管理持久化資料

Builds

- 提供建立映像檔的功能，支援從 Dockerfile 建立

Docker Scout

- 提供映像檔安全性掃描與分析功能

Extensions

- 提供安裝和管理 Docker 擴充功能的介面，增強 Docker Desktop 的功能

Dockerfile 撰寫

說明：

- Dockerfile 是一個文本檔案，包含一系列指令，用於自動化建立 Docker 映像檔的過程。

為什麼要使用 Dockerfile？

- 自動化：透過 Dockerfile，可以自動化映像檔的建立過程，減少手動操作的錯誤。
- 可重複性：Dockerfile 確保每次建立的映像檔都是一致的，方便在不同環境中部署。
- 版本控制：Dockerfile 可以與程式碼一起存放在版本控制系統中，方便追蹤變更歷史。

指令

撰寫 Dockerfile 時，需遵循特定的語法和指令。常用指令包括：

- `FROM`: 指定基礎映像檔。
- `COPY`: 複製檔案到映像檔中。
- `RUN`: 執行命令來安裝軟體包或進行設定。
- `CMD`: 指定容器啟動時執行的命令。

實作範例 (以python flask網站, sql server資料庫為例)

```
# 使用官方 Python 映像檔作為基礎映像檔
FROM python:3.9-slim
# 設定工作目錄
WORKDIR /app
# 複製需求檔案並安裝依賴
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
# 複製應用程式檔案
COPY . .
# 暴露應用程式埠
EXPOSE 5000
# 設定環境變數
ENV FLASK_APP=app.py
# 啟動應用程式
CMD ["flask", "run", "--host=0.0.0.0"]
```

SQL Server 資料庫設定

在使用 Docker 部署 SQL Server 時,需要注意以下幾點:

requirements.txt 範例

```
Flask==2.3.0
pyodbc==4.0.39
python-dotenv==1.0.0
```

連線設定

在 Flask 應用程式中連接 SQL Server 資料庫:

```
import pyodbc
import os

# 從環境變數讀取資料庫連線資訊
db_host = os.getenv('DB_HOST', 'localhost')
db_port = os.getenv('DB_PORT', '1433')
db_user = os.getenv('DB_USER', 'sa')
db_password = os.getenv('DB_PASSWORD')
db_name = os.getenv('DB_NAME', 'mydb')

# 建立連線字串
conn_str = f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={db_host},
{db_port};DATABASE={db_name};UID={db_user};PWD={db_password}'

# 連線資料庫
conn = pyodbc.connect(conn_str)
cursor = conn.cursor()
```

注意事項

- SQL Server 容器首次啟動時需要時間初始化,建議在應用程式中加入重試機制
- 密碼必須符合 SQL Server 的複雜度要求(大小寫字母、數字、特殊符號)
- 生產環境中應使用 Docker Secrets 或環境變數檔案來管理敏感資訊
- 建議使用 Volume 來持久化資料庫資料,避免容器重啟後資料遺失

實作範例二 (以python flask網站, sql server資料庫為例, 且運行使用gunicorn, 並搭配nginx)

```
# 使用官方 Python 映像檔作為基礎映像檔
FROM python:3.9-slim
# 設定工作目錄
WORKDIR /app
# 複製需求檔案並安裝依賴
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

```
# 複製應用程式檔案
COPY . .
# 安裝 Gunicorn
RUN pip install gunicorn
# 暴露應用程式埠
EXPOSE 5000
# 設定環境變數
ENV FLASK_APP=app.py
# 啟動應用程式使用 Gunicorn
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

Docker Compose 撰寫

說明：

- Docker Compose 是一個用於定義和管理多容器 Docker 應用程式的工具。通過撰寫 `docker-compose.yml` 檔案，開發者可以輕鬆地配置應用程式的服務、網路和卷。

Docker Compose 跟 Dockerfile 的關係

- Dockerfile 用於定義單個容器的映像檔，而 Docker Compose 用於定義和管理多個容器的協同工作。
- Docker Compose 可以引用多個 Dockerfile，並定義它們之間的關係和依賴。

重要提醒

- 自 Docker Compose v2 起，`version` 欄位已被棄用，不再需要指定
- 必須使用 `name` 欄位來指定專案名稱，方便管理和識別

實作範例 (以python flask網站, sql server資料庫為例)

```
name: flask-sqlserver-app

services:
  web:
    build: .
    ports:
      - "5000:5000"
    environment:
      - DB_HOST=sqlserver
      - DB_PORT=1433
      - DB_USER=sa
      - DB_PASSWORD=YourStrong@Passw0rd
      - DB_NAME=mydb
    depends_on:
      - sqlserver

  sqlserver:
    image: mcr.microsoft.com/mssql/server:2022-latest
    environment:
      - ACCEPT_EULA=Y
      - SA_PASSWORD=YourStrong@Passw0rd
```

```
    ports:
      - "1433:1433"
    volumes:
      - sqlserver_data:/var/opt/mssql

volumes:
  sqlserver_data:
```

實作範例2 (以python flask網站, sql server資料庫為例, 且運行使用gunicorn, 並搭配nginx)

```
name: flask-gunicorn-sqlserver-app

services:
    web:
        build: .
        ports:
        - "5000:5000"
        environment:
        - DB_HOST=sqlserver
        - DB_PORT=1433
        - DB_USER=sa
        - DB_PASSWORD=YourStrong@Passw0rd
        - DB_NAME=mydb
        depends_on:
        - sqlserver

    sqlserver:
        image: mcr.microsoft.com/mssql/server:2022-latest
        environment:
        - ACCEPT_EULA=Y
        - SA_PASSWORD=YourStrong@Passw0rd
        ports:
        - "1433:1433"
        volumes:
        - sqlserver_data:/var/opt/mssql

volumes:
  sqlserver_data:
```

最佳實踐

- 使用輕量級的基礎映像檔以減少映像檔大小。
- 定期更新映像檔以包含最新的安全修補程式。
- 使用多階段構建來優化映像檔。
- 適當配置資源限制以防止容器過度使用系統資源。

常見問題

- **Docker 與虛擬機有何不同？** Docker 使用容器技術，共享宿主機的作業系統核心，而虛擬機則是完整模擬一個獨立的作業系統，資源開銷較大。
- **如何在 Docker 中持久化資料？** 可以使用 Docker 卷 (Volume) 或綁定掛載 (Bind Mount) 來持久化資料，確保資料在容器重啟或刪除後仍然存在。

第二章-K8S目錄 (參考用資料)

Kubernetes 簡介

Kubernetes（簡稱 K8s）是一個開源的容器編排平台，用於自動化部署、擴展和管理容器化應用程式。Kubernetes 提供了強大的功能來管理多個容器的生命週期，確保應用程式的高可用性和彈性。

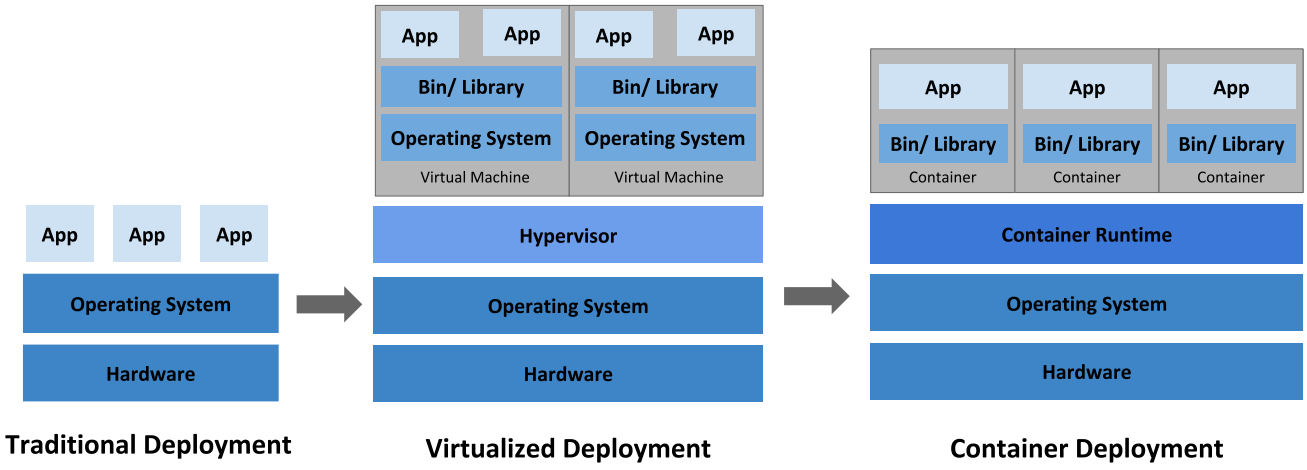
甚麼時候該用K8s

- 當應用程式需要高可用性和彈性時
- 當需要管理大量容器時
- 當需要自動化部署和擴展時

一個可能的例子

假設你有一個電子商務網站，使用 Docker 容器來部署前端、後端和資料庫服務。隨著業務的增長，網站流量增加，單一伺服器無法滿足需求。這時候，你可以使用 Kubernetes 來管理多個容器實例，實現自動擴展和負載均衡，確保網站在高流量時仍能穩定運行。

K8s 與容器的關係圖解



上圖展示了應用程式部署方式的演進：

- **傳統部署**：多個應用直接運行在實體伺服器上，容易發生資源競爭問題
- **虛擬化部署**：透過虛擬機隔離應用，但資源開銷較大
- **容器化部署**：使用容器技術輕量化隔離，而 Kubernetes 則負責管理這些容器的生命週期

當容器數量增加到一定規模時，手動管理變得困難，這時就需要 Kubernetes 來：

- 自動調度容器到合適的節點
- 監控容器健康狀態並自動重啟故障容器
- 根據負載自動擴展或縮減容器數量
- 提供服務發現和負載均衡
- 管理容器的網路和儲存資源

為什麼用了容器就要用K8s

- 容器雖然解決了應用程式的打包和部署問題，但在大規模應用中，管理大量容器變得複雜。Kubernetes 提供了自動化的容器管理功能，如自動擴展、負載均衡和自我修復，簡化了容器的運營和維護。
- Kubernetes 支援多種雲端平台和本地環境，提供一致的部署體驗，讓開發者能夠專注於應用程式的開發，而不必擔心底層基礎設施的差異。

K8s架構與組件


K8s Architecture

Kubernetes 採用 Master-Worker 架構，由多個 Node 組成 Cluster。以下用船隊的比喻來說明：

Node（節點）

Node 是 Cluster 的基本硬體單位，代表一台伺服器（實體或虛擬機），負責提供執行環境給 Pod。如果 Cluster 是一個船隊，那麼 Node 就是船，船上載著貨櫃（Pod）。

每個 Node 都必須安裝以下三個核心組件：

1. Kubelet

- 相當於每艘船上的船長
- 負責接收 Master Node 的指令並執行

- 主要工作包括：建立/刪除 Pod、監控 Pod 狀態、向 Master Node 回報狀態

2. Container Runtime

- 負責實際執行容器的底層引擎
- Kubernetes 透過 Container Runtime Interface (CRI) 與 Container Runtime 溝通
- 常見的 Container Runtime 有：containerd、CRI-O 等
- 歷史演進：Docker → Docker + shim → containerd

3. Kube-proxy

- 負責 Pod 之間的網路流量轉發
- 監控 Service 與 Pod 的變化
- 將發送到 Service 的流量正確轉發到對應的 Pod

Master Node（控制平面）

Master Node 是整個船隊的總指揮，又稱為 Control Plane，負責管理與指揮整個 Cluster。除了上述三個基本組件外，還包含四個特殊組件：

1. kube-apiserver

- Cluster 的通訊中樞，所有訊息傳遞都必須經過它
- 負責身分驗證與授權
- 管理員透過 CLI（kubectl）或 API 與它互動
- 是 Cluster 中最重要的組件之一

2. etcd

- 以 key-value 方式存放 Cluster 的所有資料
- 儲存 Cluster 狀態、資源配置等重要資訊
- 定期備份 etcd 是重要的維運工作

3. kube-scheduler

- 負責資源調度，決定 Pod 該放到哪個 Node 上
- 考量因素：資源使用率、節點狀態、Pod 需求等
- 找到合適的 Node 後，通知 kube-apiserver

4. kube-controller-manager

- 各種控制器的集合體（Node Controller、Replication Controller 等）
- 負責將資源狀態調整至「期望狀態」
- 例如：當 Pod 故障時，自動重新啟動

Worker Node（工作節點）

Worker Node 是接收 Master Node 命令並執行任務的節點，負責實際運行 Pod。如果 Cluster 是船隊，Worker Node 就是負責執行命令的小船。

Pod 建立流程範例

當使用者要建立一個 Pod 時，會發生以下流程：

1. 使用者透過 kubectl 告訴 kube-apiserver 要建立 Pod
2. kube-apiserver 驗證使用者身分與權限
3. kube-apiserver 將 Pod 資訊寫入 etcd
4. kube-scheduler 找到適合的 Node
5. scheduler 透過 apiserver 將 Node 資訊寫入 etcd
6. 目標 Node 的 kubelet 收到通知，調用 Container Runtime 建立容器
7. Pod 狀態回報給 apiserver，並寫入 etcd

HA Cluster（高可用性叢集）

為避免單一 Master Node 故障導致管理功能癱瘓，生產環境通常會建立 HA Cluster：

Stacked etcd topology

- 至少需要 3 台 Master Node
- etcd 與 Master Node 部署在同一台機器
- 優點：容易部署與管理
- 缺點：承擔同時失去 Master Node 與 etcd 的風險

External etcd topology

- 至少需要 3 台 Master Node + 3 台 etcd server
- etcd 獨立部署在另一台機器
- 優點：避免同時失去 Master Node 與 etcd
- 缺點：部署複雜，成本較高

為什麼是奇數個 Master？

- K8s 採用 RAFT 演算法確保資料一致性
- 需要「超過半數」的節點同意才能寫入資料
- 偶數個節點容錯能力較差，實務上建議 3-5 個 Master Node

Kubernetes 安裝

Kubernetes 可以透過多種方式安裝，包括使用 Minikube、kubeadm 或在雲端平台上部署。安裝過程通常包括設定 Master Node 和 Worker Node，並建立 Cluster。