前言

本書是專為任何需要實際了解 Linux 容器、以及如何才能用容器來改善開發及生產實況的讀者所設計的。大多數的現代整合工作流程和正式環境系統，都要求開發人員和營運工程師要充分地了解 Linux 容器、以及如何利用容器來大幅改善整個系統中的可重複性與可預測性。本書會探索如何在 Docker 的周邊生態環境中建置、測試和部署 Linux 容器，並進行除錯。我們也會談到若干運用了 Linux 容器的知名調度工具。最後還會總結出一份有關容器環境安全性及最佳實施方式的指南。

誰應閱讀本書

任何人，只要是想要針對大規模正式環境中所進行的開發及部署軟體的複雜工作流程，解決其中涉及的問題，就是本書預設的讀者。如果你對於 Linux 容器、Docker、Kubernetes、DevOps，以及大型可擴展的軟體基礎架構有興趣，這本書就適合你。

為何要讀這本書？

如今的網際網路上早已充斥著各種關於 Docker 的對談、專案及文章，有些甚至還開始預言 Docker 即將過氣。

那為何還要花費寶貴的時間閱讀本書？

雖說坊間還有其他的替代方案，但 Docker 可是只憑一己之力便讓所有工程師都能取用 Linux 容器的方案。在 Docker 建立容器映像檔標準、並協助打造出許多當今容器化系統所使用的核心程式庫之前，Linux 容器可是十分難用，而且主要都只是超大型雲端託管業者手中的工具，用來提供可擴充性，同時保護他們的系統不受不可靠的使用者程式碼所影響。

Docker 改變了這一切。

即使外界已有大量關於 Docker 和 Linux 容器的資訊，此一領域仍在蓬勃發展之中，而最佳實施方式也一直在演變當中。想像一下，你剛讀過一篇四年前發表的、關於 Docker 的部落格貼文。它也許仍然可以拿來參考引用，但卻可能已不再是最好的做法。光是在我們撰寫本書初版期間，Docker, Inc. 便釋出了四個版本的 Docker、外加幾種周邊生態系統中的主要工具。在本書初版與再版之間的七年當中，該技術版圖又發生了劇烈變化。Docker 已經趨於穩定，而且有許多額外的工具可以扮演類似的角色。Docker 非但不缺工具，它甚至在 DevOps 工作流程的每個面向都有許多穩健可靠的選擇。想要完全擁抱 Linux 容器與 Docker 所提供的一切，同時理解如何將其融入你的工作流程，並正確進行所有各種整合，可不是一件一蹴可幾的事。

九年多來，我們曾與多家公司合作，建置並營運各種 Linux 容器平台的正式環境，包括了 Docker、Mesos 和 Kubernetes。我們原本在 Docker 初次面世的幾個月內便實作了它的正式環境，從那時開始，我們便與讀者們分享了若干我們自己在發展正式環境平台時所獲得的親身經歷。我們撰寫本書的目標，是希望讀者們能汲取我們的經驗、避開我們吃過的苦頭。雖說 Docker 專案的線上文件（*https://docs.docker.com*）非常有用，我們還是會嘗試為讀者描繪出更完備的輪廓，並把我們一路以來所領悟的許多最佳實施方式提供給讀者們。

一旦你讀完本書，應該就能取得足夠的資訊，了解何謂 Linux 容器、Docker 又提供了什麼、為何它們如此重要，以及你如何能利用它們，把從本地端開發一路直到正式環境的一切事物順暢化。透過一些已有實際應用的有趣技術，這會是一趟迷人無比的旅程。

本書概覽

本書章節編排如下：

- 第 1 與第 2 章介紹了 Docker，並說明其本質、以及你可以如何使用它。
- 第 3 章引導讀者們安裝 Docker 的相關步驟。

- 第 4 到第 6 章會深入介紹 Docker 的用戶端、映像檔、以及容器，並探索其本質、以及如何使用它們。
- 第 7 章討論如何為映像檔及容器除錯。
- 第 8 章介紹 Docker Compose，以及如何利用它大幅簡化複雜容器式服務的開發。
- 第 9 章探索了能確保順利過渡到正式環境的各項重要考量。
- 第 10 章深入探討如何在公有及私有雲大規模部署容器。
- 第 11 章深入探討各種需要對 Docker 已有若干認識的進階題材，當你開始在正式環境中採用 Docker 時，這些題材就可能十分重要。
- 第 12 章探索了若干在容器化 Linux 環境中會很有用的替代性工具。
- 第 13 章探索了若干業界已經確信的、在設計新一代網際網路等級軟體的正式環境時所需的核心概念。
- 第 14 章總結了所有的內容。其中包括已經談過的內容，以及這些內容如何能協助讀者們改進軟體服務的交付與擴展方式。

我們很清楚有很多人不會從頭到尾詳讀技術書籍，像這段序言就是很容易為人所忽略的，但如果你已讀到這裡，以下是一些關於如何閱讀本書的快速指南：

- 如果你是 Linux 容器新手，就從頭開始讀。前兩章旨在協助你搞懂 Docker 和 Linux 容器的基本知識，包括其本質、如何運作、以及為何你應該在乎這些事。
- 如果你想直接跳去學習如何在自己的工作站上安裝及運行 Docker，就從第 3 和第 4 兩章開始，這兩章會解釋如何安裝 Docker、以及如何產生和下載映像檔、還有如何運行容器等等。
- 如果你已熟悉 Docker 的基礎知識，但還是想多學一點關於如何在開發時的運用，就去讀第 5 至 8 章，這四章會談到大量的技巧，可以讓你的日常 Docker 操作更為輕鬆，同時也詳盡地探索了 Docker Compose。
- 如果你已經在開發時使用 Docker，但需要有人指導你如何將開發成果送上正式環境，就從第 9 章開始、一路讀到第 12 章。這些章節會深入探討容器的部署、如何運用高階的容器平台、以及許多其他的進階題材。
- 如果你擔任的是軟體或平台架構師，你會發現第 13 章是一個值得鑽研的有趣章節，因為我們會在此深入探討若干目前對於容器化應用程式的思維、以及可以水平擴展的服務設計。

第五章

操作容器

在前一章當中，我們學到了如何建置一個 Docker 映像檔、以及如何在容器中運行一個結果映像檔所需的種種基本步驟。至於這一章，我們首先要來介紹容器技術的過往，然後深入研究容器的運行，並探討各種 Docker 命令，它們控制了容器如何接收所有的組態、資源及權限。

何謂容器？

你或許已經十分熟悉 VMware 或 KVM 之流的虛擬化系統，它們允許你在一個硬體虛擬層上運行完整的 Linux 核心及作業系統，這個虛擬層就是大家熟知的 *hypervisor*。這種手法在個別的工作負載間提供了十分徹底的隔離機制，因為每個 VM 都擁有自己的作業系統核心、並使用硬體虛擬層所提供的個別記憶體空間。

容器的本質則不一樣，因為它們都共用一個核心，而工作負載之間的隔離則是完全在單一核心中實作出來的。這又稱為**作業系統虛擬化**（*operating system virtualization*）。

`libcontainer` 的 README（*https://github.com/opencontainers/runc/blob/main/libcontainer/README.md*）中提出了對於容器的簡單扼要定義：

> 容器所指的是一個自給自足的執行環境，它與主機系統共用核心，而且（也許）會與系統上其他容器隔離。

容器的主要優點之一，在於其資源效率，因為你並不需要為個別隔離的工作負載提供完整的作業系統執行個體。由於核心是共享的，在隔離的工作負載與底層實際硬體之間便少了一個間接層。當程序運行在容器當中之時，只有少數程式碼位於管理容器的核心當中。相較於 VM，VM 會在容器不具備的間接層中運行這個部分。在 VM 裡，程序

對硬體或 hypervisor 的呼叫必須兩度進出處理器的特權模式，因而導致許多呼叫的速度變慢。

libcontainer（*https://github.com/opencontainers/runc/tree/main/libcontainer*）是一個 Go 語言程式庫，其設計係用來為應用程式提供標準的 Linux 容器管理介面。

但容器的做法意味著你運行的程序必須與底層核心相容。譬如說，容器不像 VMware 或 KVM 提供的硬體虛擬化技術，你無法在 Linux 主機上的 Linux 容器當中直接運行 Windows 應用程式，而是只能在 Windows 主機上的 Windows 容器之中運行 Windows 的應用程式。因此你最好是把容器想像成一種與特定 OS 相關的技術，你可以從中運行任何與容器的伺服端核心相容的應用程式或 daemons。當你面對容器時，請儘量拋棄以往對 VM 的一切認知，而是把容器概念視為伺服器上所運行程序的一個外包裝層。

除了可以在 VM 當中運行容器以外，你當然也可以在容器當中運行 VM。如果採用這種方式，那你當然可以在一個運行於 Linux 容器上的 Windows 虛擬機當中運行 Windows 應用程式。

容器簡史

所謂革命性的技術，常常不過是另一個早年既存的技術，只是終於得到人們關注罷了。技術此起彼落，某些 1960 年代便已出現的想法又再度流行起來。同樣地，Docker 雖屬於較新的技術，其易用性才是令它一炮而紅的關鍵，但它並非憑空誕生的。Docker 所憑依的許多技術皆源於過去 30 年間在各種領域累積的成就。我們可以輕易地追溯容器概念的演進，從 1970 年代晚期為 Unix 核心加入的簡易系統呼叫，直到近年來各家網際網路巨擘（像是 Google、Twitter 和 Meta）所仰賴的現代化容器工具等等。值得花一點時間來快速回顧一下，這些技術是如何演進並導致了 Docker 的誕生，因為理解這些背景會有助於你用已經熟悉的舊知識來認識容器的新面貌。

容器並不是什麼新的觀念。它們也不過是將運行中系統的一部分加以隔離和封裝的方式而已。此一領域最早的技術包括批次處理系統（batch processing systems）。在使用早期的電腦時，系統只會一次運行一個程式，並且在前一支程式已經完成、或是已屆滿預定的運行期間之時，切換到另一支程式去運作。這樣的設計可以達到某種程度的隔離：你可以確保你的程式不至於與他人的程式重疊，因為電腦一次只能運行一件事物。雖說現代的電腦還是會在任務之間頻繁切換，但速度卻快到大部分使用者感覺不出來。

第七章

為容器除錯

一旦你將應用程式交付至正式環境，總有一天它會出毛病無法如預期般運作。而若是能提前得知這一天何時會到來，總是一件好事。此外，在展開更複雜的部署之前，你應該要對容器的除錯有良好的認識。若是缺乏除錯技巧，就很難看出調度系統哪裡出了問題。所以這一章我們就要來談談容器的除錯。

說到底，為容器化應用程式進行除錯，跟原本替系統上的正常程序除錯並無多大差異，只不過使用的工具略有不同罷了。Docker 提供了一些很出色的工具來協助你！有些工具的概念可以和尋常的系統工具相呼應、有些則更上一層樓。

此外還有一點很要緊，那就是你的應用程式和其他的 Docker 程序也都運行在相同的系統上、並未分離。它們都共用一個核心，而且根據你的容器組態，它們甚至還共用其他的事物，像是儲存子系統和網路介面等等。這意味著你可以從底層系統本身獲得大量關於容器動作的資訊。

如果你早已習於在 VM 環境中為應用程式除錯，也許你直覺就會認為應該進到容器裡面才能檢查應用程式的記憶體或 CPU 使用狀況、或是為系統呼叫除錯。然而實情並非如此！儘管容器中的程序在很多方面給人的觀感都像是虛擬化的層級，但它們其實也不過就是 Linux 主機本身的程序之一而已。如果你想觀察一部機器上跨越所有 Linux 容器當中的程序清單，只需登入伺服器、以你偏好的命令列選項執行 `ps` 即可。但是，你也可以從底層 Linux 核心的觀點出發，從任何位置用 `docker container top` 命令觀察運行在容器中的程序清單。且讓我們仔細研究一下，看看在不動用到 `docker container exec` 或 `nsenter` 的情況下，還能做哪些事情來為容器化應用程式除錯。

程序的輸出

當你在為容器除錯時，你會想知道的首要之務，便是容器裡正在運行些什麼事物。先前曾經提過，Docker 有一個內建命令可以得知這類資訊：`docker container top`。它並非唯一可以得知容器內動態的方式，但卻是目前為止最方便的一個。來看看它是如何運作的：

```
$ docker container run --rm -d --name nginx-debug nginx:latest
796b282bfed33a4ec864a32804ccf5cbbee688b5305f094c6fbaf20009ac2364

$ docker container top nginx-debug

UID   PID  PPID C STIME TTY TIME  CMD
root  2027 2002 0 12:35 ?   00:00 nginx: master process nginx -g daemon off;
uuidd 2085 2027 0 12:35 ?   00:00 nginx: worker process
uuidd 2086 2027 0 12:35 ?   00:00 nginx: worker process
uuidd 2087 2027 0 12:35 ?   00:00 nginx: worker process
uuidd 2088 2027 0 12:35 ?   00:00 nginx: worker process
uuidd 2089 2027 0 12:35 ?   00:00 nginx: worker process
uuidd 2090 2027 0 12:35 ?   00:00 nginx: worker process
uuidd 2091 2027 0 12:35 ?   00:00 nginx: worker process
uuidd 2092 2027 0 12:35 ?   00:00 nginx: worker process

$ docker container stop nginx-debug
```

要執行 `docker container top`，我們得把容器的名稱或識別碼提供給它，然後便會看到一長串容器內正在執行事物的漂亮清單，就像 Linux 的 `ps` 輸出一樣，按照 PID 排序陳列。

不過這裡仍有一些怪異之處。主要是使用者名稱的命名空間和檔案系統。

你必須理解，特定使用者識別碼（user ID，UID）所屬的使用者名稱，在每個容器和主機系統中可能完全不同。甚至有可能某個特定的 UID 在容器中、或在主機的 */etc/passwd* 檔案裡完全對應不到任何具名使用者。這是因為 Unix 並不一定要 UID 能對應到一個具名的使用者，我們稍後會在第 303 頁的「命名空間」一節中詳細介紹 Linux 的命名空間（namespaces），它在容器的有效使用者概念、以及底層主機的有效使用者概念之間，提供了某種程度的區隔。

來看一個更實際的例子。設想有一部運行了 Ubuntu 22.04 的 Docker 伺服端正式環境，而其中運行的容器也使用了 Ubuntu 發行版。如果你在 Ubuntu 主機上執行以下命令，你會看到 UID 7 被命名為 `lp`：

但除非你把容器頂層的程序（即容器內的 PID 1）清除，光是清除程序是無法將容器本身清除的。如果你只是要清除失控的程序，這也許是可行的，但這會讓容器處於預期外的狀態。開發人員如果能用 `docker container ls` 看到他們的容器，就可能會預期所有的程序都在運行當中。但這也可能會誤導 Mesos 或 Kubernetes 這類的排程工具、或是任何其他會為你的應用程式進行健康檢測的系統。請記住，容器對於外界來說是一個完整個體。如果你需要清除容器裡的某些部分，最好是把整個容器都抽換掉。容器提供了一個可以與工具互動的抽象層。而這些工具會認定容器內部是可預期而且始終一致的。

我們之所以要對程序發送訊號，目的並不僅限於要終結程序而已。由於容器化的程序在許多方面也不過就是正常程序，因此它們也可以接收 Linux 的 `kill` 命令的說明頁所列的整套 Unix 訊號。很多 Unix 程式會在接收到特定的預定訊號時執行特殊動作。舉例來說，`nginx` 就會在接收到 `SIGUSR1` 訊號時重新開啟日誌。你可以利用 Linux 的 `kill` 命令將任何 Unix 訊號發送給本地端伺服器上的容器程序。

容器內的程序控制

即使你可以運行像是 Kubernetes 這種能在較大抽象環境（例如 pod）中處理多個容器的調度工具，我們仍認為在正式環境容器中運行某種程度的程序控制是最佳實施方式。不論是 `tini`（*https://github.com/krallin/tini*）、`upstart`（*https://upstart.ubuntu.com*）、`runit`（*http://smarden.org/runit*）、`s6`（*https://skarnet.org/software/s6*）還是其他工具，這種方法都讓你可以在容器中含有一個以上的程序時，以最細緻的方式對待容器。但是你仍應盡力不要在容器內執行一個以上的事物，以確保容器只會限定處理定義良好的單一任務，不至於膨脹成一個一應俱全（monolithic）的容器。

不論何種狀況，你都會希望 `docker container ls` 能反映出整個容器的存在現況，這樣你就不必煩惱其中是否有個別程序已經掛掉。如果只要容器存在、加上沒有錯誤日誌出現這兩件事，就能假設一切運作無誤，你就可以把 `docker container ls` 的輸出視為 Docker 系統正在發生的真實情況。這也意味著你使用的任何調度系統也能這樣做。

你應該也要能掌握你所偏好的程序控制服務的整體行為，包括記憶體或磁碟運用狀況、Unix 的信號處理（signal handling）等等，因為這會影響容器的效能和行為。一般說來，以輕量型系統為佳。

由於容器的運作就像任何其他程序一樣，你必須了解它們是如何與你的應用程式實際互動的。在容器中，程序若要分生出背景子程序，是有一些特殊需求的——亦即任何程序都會分支（forks）並將自身化為 daemon，使得父程序不再管控子程序的生命週期。Jenkins 建置容器就是一個人們會看到這種錯誤的常見範例。當 daemons 在背景進行分支時，這些分支出來的內容會變成 Unix 系統上 PID 1 的子程序。而 Process 1 是個特殊程序，通常就是某種 `init` 程序[譯註]。

PID 1 負責確保子程序都會得到清理。在你的容器裡，根據預設，主程序就會是 PID 1。由於你也許不會從應用程式來處理子程序的清理，於是就可能在容器中衍生出殭屍程序。這個問題有幾種解法。第一種是在你的容器中運行一個 init 系統——讓它負責處理 PID 1 的職掌。而 `s6`、`runit` 和其他上頁註解中描述的工具就能輕易地在容器中使用。

但是 Docker 本身提供了一個更簡單的選項，可以解決這個案例、同時又不必用到整套 init 系統的全部功能。如果你在執行 `docker container run` 命令時加上 `--init` 旗標，Docker 便會啟動一個源於 `tini` 專案（*https://github.com/krallin/tini*）的極小 init 程序，它會在容器啟動時取得 PID 1。不論你在 *Dockerfile* 以 `CMD` 指定了什麼內容，都會被轉交給 `tini`，以你預期的方式運作。但它確實也取代了你的 *Dockerfile* 中原本在 `ENTRYPOINT` 段落裡所有的任何內容。

當你啟動一個 Linux 容器但沒有加上 `--init` 旗標時，你會在程序清單中看到像這樣的內容：

```
$ docker container run --rm -it alpine:3.16 sh
/ # ps -ef

PID   USER     TIME   COMMAND
    1 root       0:00 sh
    5 root       0:00 ps -ef

/ # exit
```

注意，上例中我們啟動的 `CMD` 佔用了 PID 1。這代表它必須負責子程序的清理。如果我們啟動的容器至關緊要，可以再加上 `--init`，以確保當父程序退出時，子程序會得到清理：

```
$ docker container run --rm -it --init alpine:3.16 sh
/ # ps -ef

PID   USER     TIME   COMMAND
```

譯註 於是 Jenkins 自己便變成了原本應該由 `init` 擔任的 PID 1 程序。

第十章

大規模容器環境

容器主要的強項，就是它可以把底層的硬體及作業系統抽象化，因而使得你的應用程式不再受限於特定的主機或環境。不只在你的資料中心裡、它甚至也可以跨越雲端供應商水平擴充無狀態應用程式，而不會受到許多傳統上的障礙限制。正如同貨櫃輸送的譬喻一般，在某一個雲端的容器，就算換到另一個雲端，看起來也不會有差異。

許多機構發覺現成部署的 Linux 容器十分具有吸引力，因為它們馬上就能獲得許多可擴充容器式平台的優點、但卻不必在自家當中從頭建置某些事物。雖說如此，但其實不論是在雲端、還是在你自己的資料中心裡建置自家的平台，門檻都很低，我們馬上就會談到幾種選項。

主流的公有雲供應商都已可在自家服務上直接支援 Linux 容器。其中對 Linux 容器的支援規模最大的公有雲如下：

- Amazon Elastic Container Service（*https://aws.amazon.com/ecs*）
- Google Cloud Run（*https://cloud.google.com/run*）
- Azure Container Apps（*https://azure.microsoft.com/en-us/services/container-apps*）

上述業者同時也提供了非常紮實的 Kubernetes 代管服務：

- Amazon Elastic Kubernetes Service（*https://aws.amazon.com/eks*）
- Google Kubernetes Engine（*https://cloud.google.com/kubernetes-engine*）
- Azure Kubernetes Service（*https://azure.microsoft.com/en-us/services/kubernetes-service*）

在任何公有雲的一個 Linux 執行個體上安裝 Docker 並不困難。但是把 Docker 裝進伺服器，只不過是產生整個正式環境的步驟之一而已。你可以獨力進行，或是也可以利用主流雲端業者、Docker, Inc. 及廣大容器社群所提供的豐富工具。這些工具不論是在公有雲或是你自有的資料中心裡都可以運作得一樣好。

在排程工具或是更複雜工具系統的領域裡，我們有許多系統可以選擇，它們都能重現公有雲供應商所提供的多項功能。即使你已經在公有雲上運作，還是會有一些避免不了的因素，讓你選擇運行自己的 Linux 容器環境、而不採用那些現成的產品。

在本章當中，我們會談到若干大規模運行 Linux 容器的選項，首先我們會介紹相對簡單的 Docker Swarm 模式，然後再深入 Kubernetes 這種更為進階的工具、以及若干大型的雲端產品。所有這些範例應該可以讓你體會，如何利用 Docker 來為你的應用程式工作負載提供出色的彈性平台。

Docker Swarm Mode

在以 Docker 引擎的形式建置出容器執行環境之後，Docker 的工程師轉頭考慮另一個問題，就是如何調度個別的 Docker 主機、並有效地在這些主機上裝滿容器。第一項成果就是名為 Docker Swarm 的工具。我們先前也解釋過，很多人會困惑為何有兩款名字裡有「Swarm」字樣的產品，還都來自 Docker, Inc.？

原本獨立的 Docker Swarm，現在又被稱作是 Docker Swarm (classic)（*https://github.com/docker-archive/classicswarm*），而第二種「Swarm」的實作則又被精確地稱為 Swarm 模式（*https://docs.docker.com/engine/swarm*）。後者不再是獨立產品，而是內建在 Docker 用戶端當中。內建的 Swarm 模式，其功能比先前的 Docker Swarm 更豐富，而其用意則在完全取代前者。Swarm 模式的一大優點，就是它不需要分開安裝任何事物。你已經可以在任何運行 Docker 的系統上取得叢集（clustering）功能！這才是我們在此要鑽研的 Docker Swarm 實作版本。希望讀者們現在已經理解為何會有兩種不一樣的 Docker Swarm 實作，而且也不至於被網際網路上種種彼此牴觸的資訊搞糊塗。

Docker Swarm 模式背後的概念，在於向 `docker` 用戶端工具提供單一介面，但該介面背後則是一整套的叢集、而不再只是單獨一個 Docker daemon。 Swarm 主要就是以透過 Docker 工具管理叢集化運算資源為目標。自從它面世以來，已經有了極大的成長和變化，現已含有數種排程工具的外掛程式，它們各自有自己的策略，能把容器指派給叢集中的主機，而且也都內建若干基本的服務尋覓功能。但它依然只是更為複雜解決方案的建構區塊之一。

```
$ kubectl delete -f ./lazyraster-service.yaml

service "lazyraster" deleted
persistentvolumeclaim "cache-data-claim" deleted
deployment.apps "lazyraster" deleted
```

最後，如果你已經做夠了實驗，就可以把 Minikube 叢集移除了：

```
$ minikube delete

🔥  Deleting "minikube" in docker …
🔥  Deleting container "minikube" …
🔥  Removing /Users/spkane/.minikube/machines/minikube …
💀  Removed all traces of the "minikube" cluster.
```

Kubernetes 是非常龐大的系統，社群參與程度非常高。我們在此不過是藉由 Minikube 向讀者們展現了冰山的一角而已，但如果你有興趣繼續鑽研，坊間有很多其他 Kubernetes 發行版及工具可供探索。

與 Docker Desktop 整合的 Kubernetes

Docker Desktop 支援內建單一節點的 Kubernetes 叢集，只需在應用程式的偏好設定裡（preferences）啟用一個選項，便能將其運行起來。

要設定整合的 Kubernetes 叢集並不容易，但它確實提供了一個非常容易取用的選項，可以讓那些只需對當前安裝的 Kubernetes 驗證若干基本功能的人使用。

要啟用 Docker Desktop 內建的 Kubernetes 功能，請啟動 Docker Desktop、接著從工具列 / 選單的 Docker 鯨魚圖示開啟 Preferences。然後再選擇 Kubernetes 分頁、點選 Enable Kubernetes、最後點選「Apply & Restart」按鈕，讓必要的變更套用在 VM 上。當你頭一回這樣做時，Docker 會利用 `kubeadm`（*https://kubernetes.io/docs/reference/setup-tools/kubeadm*）命令來設置 Kubernetes 叢集。

如果你有興趣進一步了解 Docker Desktop 整合的 Kubernetes 是如何設置的，Docker 有一篇極好的部落格貼文（*https://www.docker.com/blog/how-kubernetes-works-under-the-hood-with-docker-desktop*），其中有談到一些細節。

進階組態

Docker 的外部網路介面十分簡潔，表面上看起來相當單純。但其實它的背後有許多可以調整設定的東西在運作，而我們在第 152 頁的「日誌紀錄」一節中描述過的日誌紀錄後端，便是一個這樣的例子。你可以為整個 daemon 抽換容器映像檔的儲存後端、或是改用完全不一樣的執行環境（runtime）、甚至指定個別容器使用不同的網路組態，這些都是可以做得到的。上述的切換功能都十分強大，但你在沒弄清楚其作用前不應隨便使用。首先我們會來談談網路組態，然後我們會探討儲存用的後端，最後才會嘗試完全抽換不同的容器執行環境，把 Docker 預設提供的 `runc` 給換掉。

網路功能

早先我們曾經描述過位在 Linux 容器及真實網路之間的網路層。現在我們來仔細地觀察其運作。Docker 支援為數豐富的網路組態，但我們要先從預設的設置方式開始。圖 11-1 顯示的便是典型的 Docker 伺服端，右方有三個容器正在自己的私有網路上運作。其中之一在 Docker 伺服端開放了公用通訊埠 port（10520 號 TCP 通訊埠）。我們會追蹤對內連線請求如何抵達 Linux 容器、也會檢視 Linux 容器的對外連線是如何抵達外部網路的。

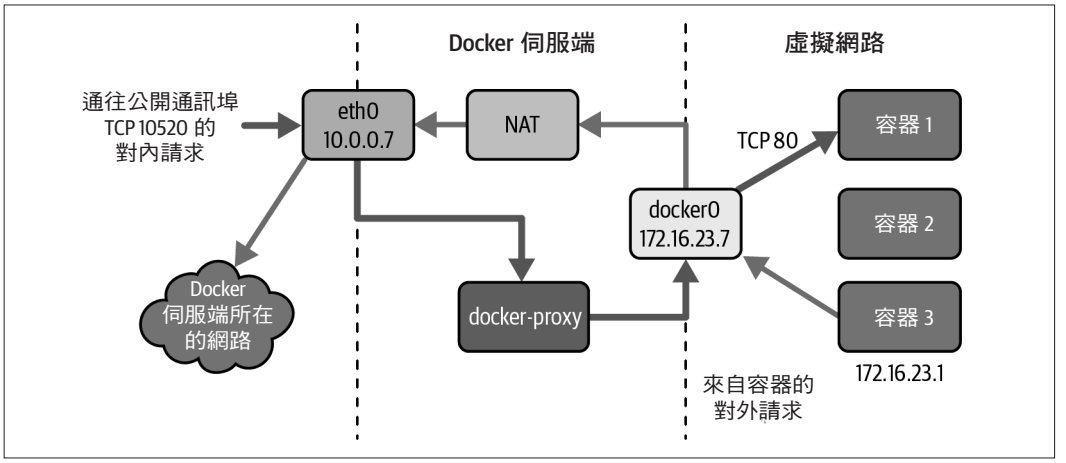

圖 11-1　典型 Docker 伺服端的網路

第十三章

容器平台設計

當你需要在正式環境中實作任何一種新技術時，通常都得先設計出一個具備充分韌性的平台，它必須能承受必定會發生的意外問題，如此方可達到最佳成效。Docker 雖說是威力強大的工具，但也需要充分注意細節，才能讓平台順利運作。作為一項發展飛快的技術，在構成容器平台的各種組成元件之間，總免不了會不時出現種種令人失望的錯誤或問題。

如果你並非只是想把 Docker 草草部署到現有環境當中，而是願意花上一點時間來打造以 Docker 作為核心元件、並且設計完善的容器平台，你就能享受到容器化工作流程的諸多好處，同時不至於受到這樣迅速發展的專案所導致的一些麻煩的問題所傷害。

Docker 就跟所有其他技術一樣，並不能神奇地解決你所有的問題。要發揮其真正的潛力，你任職的機構必須審慎地決定採用它的緣由及方式。對於小型專案來說，可以用很簡單的方式來運用 Docker；不過如果你打算支援一個能隨需求擴展的大型專案，那麼審慎地設計你的應用程式及平台便十分重要。如此可確保你在該項技術上最大的投資報酬率。多花些時間仔細設計你的平台，也能保障日後能更輕鬆地修改你的正式環境工作流程。一個設計良好的容器平台及部署過程，應該盡可能地保持輕便和直覺化，同時仍能支援各種功能，以便滿足一切技術上及法規遵循上的需求。一個考量完善的設計，要能確保你的軟體可以在動態的基礎上運行，亦即要能輕易地隨著技術和公司流程的發展而升級。