前言

Kubernetes 想要感謝每一位在清晨三點被叫起來重新啟動程式的系統管理員。以及感謝每一位單純為了測試程式是否跟自己筆電上運作相同，而把程式碼推到線上環境的開發者。還有不小心忘記改掉主機名稱，而把壓力測試流量導引到線上環境的系統架構師。這些奇怪的錯誤，詭異的上班時間所帶來的痛苦，成為了 Kubernetes 被開發的起點。簡單來說，Kubernetes 存在的目的是希望從根本簡化程式碼的建構、部署、以及分散式系統的維護。它的靈感來自於數十年來維護大型系統可靠度累積下來的實務經驗，並從底層開始設計，希望這個新的系統就算不能讓大家使用時感到興奮，至少能讓人用起來的感受是舒服的。希望你也能喜歡這本書。

誰應該閱讀這本書

無論是分散式系統的新手，或者是擁有多年雲端經驗的老手，容器與 Kubernetes 都能幫助你在速度、效率、靈活性和可靠性更上一層樓。本書介紹 Kubernetes 編排器（orchestrator）以及如何利用它的工具和 API 來改善分散式應用程式的開發、交付，以及提升安全性和維護性。儘管沒有 Kubernetes 的使用經驗也沒關係，但你至少還是需要對伺服器端的應用程式建構及部署有基本的理解，例如瞭解負載平衡器、網路儲存、Linux、Linux 容器以及 Docker 這類的技術會在閱讀這本書的時候非常有幫助。

為什麼我們寫這本書

我們在 Kubernetes 創建之初就參與這個計畫。看著這套系統從基於好奇心而開發的實驗品，一路走向生產環境中非常重要的基礎建設，乘載著不同領域的大型應用程式，例如機器學習及線上服務。基於這樣的轉變，我們認為撰寫一本同時收錄 Kubernetes 的基本概念及背後開發動機的書很重要，希望為雲端原生程式的開發之旅提供一些貢獻。我們希望在閱讀這本書的同時，你不只能學會如何在 Kubernetest 上建構可靠、可擴展的程式，同時也能理解分散式系統所帶來的挑戰以及導致我們開發 Kubernetes 的原因。

為什麼我們修訂這本書

在本書的第一版、第二版之後，Kubernetes 的生態系不斷的成長演進，隨著不斷推陳出新的 Kubernetes，越來越多的工具及使用模式成為 Kubernetes 的標準。在第三版中，我們針對利用程式存取 Kubernetes 的方式，Kubernetes 的安全性，以及跨叢集部署的應用程式這幾個日益興盛的題目做介紹，同時我們也更新過去所有章節的內容來對應近期 Kubernetes 版本更新所做出的調整。期待隨著 Kubernetes 不斷演進，我們可以在未來幾年後再度更新這本書。

今日雲端原生程式的演進

從最初的程式語言到物件導向程式開發，接著虛擬化技術的產生到雲端基礎建設服務，計算機科學的演進史總是試著把複雜的細節抽象化，來作為建造更複雜應用程式的基礎。但是建立一個可靠，可擴展的應用程式始終比想像中困難。直到近幾年，容器或 Kubernetes 這類的容器 API 被證實可以大幅降低分散式系統開發的複雜度，並成為提高可靠度且可擴充的重要工具。幾年前如同科幻小說般的情節，直到容器以及容器編排工具的出現，才真正地讓開發者可以快速、敏捷，且可靠的建構並部署程式。

本書導覽

本書的編排如下。第 1 章將簡單的介紹 Kubernetes 的好處，如果是第一次接觸 Kubernetes，你可以從這裡開始閱讀，並慢慢瞭解為何需要繼續讀這本書後續的章節。

第 2 章將介紹容器以及容器化應用程式的開發細節。如果你過去從來沒有使用過 Docker，這個章節將非常有幫助，如果你已經是 Docker 專家的話，可以利用這個章節回憶一下相關細節。

第 3 章將介紹如何部署 Kubernetes。雖然本書大部分重點都放在如何使用 Kubernetes，但你還是會需要在開始練習前先準備一個可以使用的 Kubernetes。安裝設定一個生產環境等級的 Kubernetes 叢集不在本書的範圍內，這個章節僅提供幾種簡單的方法來建立一個足以作為練習用的 Kubernetes 叢集。第 4 章涵蓋了與 Kubernetes 叢集互動的常見指令。

從第 5 章開始，我們將開始深入瞭解如何使用 Kubernetes 部署應用程式，我們將會提到 Pod（第 5 章），Label 和 Annotation（第 6 章），Services（第 7 章），Ingress（第 8 章），ReplicaSets（第 9 章），這些章節是在 Kubernetes 上建立服務的基本元件。接著我們將介紹 Deployment（第 10 章），同時介紹一個應用程式在 Kubernetes 上的生命週期。

後續章節我們將會探討一些特別的 Kubernetes 物件，例如 DaemonSet（第 11 章），Job（第 12 章），ConfigMap 及 Secret（第 13 章），儘管上面這些章節是構成一個生產環境應用程式必要的元素，但是如果你只是想學習 Kubernetes 的話可以暫時跳過這幾章，待逐漸累積經驗後再回頭閱讀這些內容。

再來我們將介紹角色權限控制（Role-based access control）（第 14 章），Service Mesh（第 15 章），整合儲存空間到 Kubernetes（第 16 章），擴充 Kubernetes 的功能（第 17 章），利用程式語言存取 Kubernetes（第 18 章），然後專注在 Pod 的安全性（第 19 章）以及利用 Kubernetes 的 Policy 來管理 Kubernetes 叢集（第 20 章）。

最後，我們將利用一些範例來總結如何在跨叢集環境中開發並部署應用程式（第 21 章），並討論該怎麼在原始碼管理環境中組織你的程式（第 22 章）

線上資源

你需要安裝 Docker（*https://docker.com*），如果你對 Docker 不熟的話，建議看看 Docker 的官方文件。

同時你也需要安裝 `kubectl` 命令列工具（*https://kubernetes.io*）。你可能也會想要加入 Kubernetes 的 Slack 頻道（*http://slack.kubernetes.io*），在這裡你可以找到很多 Kubernetes 社群的同好，在任何時間與你討論並回答對 Kubernetes 的疑問。

最後當你成為 Kubernetes 專家後，你可能也會想到 Kubernetes 在 GitHub 上的開放原始碼專案貢獻一份心力（*https://github.com/kubernetes/kubernetes*）。

第一章

緒論

Kubernetes 是部署容器化應用程式的開源編排器（orchestrator）。最初由 Google 開發，靈感來自於十年間透過應用程式導向 API 部署可擴展且可靠的容器系統經驗[1]。

自從 2014 年推出以來，Kubernetes 已經是為全球最大且最受歡迎的開源項目之一。也成為構建雲端原生應用程式的 API 標準，幾乎在每個公有雲都可以見到它的身影。小至樹莓派（Raspberry Pi）叢集，大至充滿最新設備的資料中心，Kubernetes 已經被證實是適合雲端原生開發者在分散式系統使用的基礎架構。Kubernetes 提供了建置和部署可靠、可擴展的分散式系統所需的相關工具。

你可能會好奇「可靠、可擴展的分散式系統」是什麼？越來越多的服務透過 API 的形式存在於網路上，這些 API 通常透過**分散式系統**的概念實作，各個部件運行在不同的機器上透過網路連接並溝通協調。因為我們在日常生活中越來越倚賴這些 API（例如：尋找最近醫院的路徑），所以這些系統必須有很高的**穩定性**，即使只是小規模的系統當機或是其他的問題都不允許發生。也因此，就算軟體更新或是其他的系統維護，都要維持其**可用性**。同時由於越來越多的人使用這些服務，因此必須具有高度的可擴展性，以便追上持續增加的用量，不需要為了改善服務而徹底的重新設計整個分散式系統。在許多案例中，這代表你的程式可以透過自動的擴充或縮小容量來達到效率最大化。

不論你基於什麼原因開始閱讀這本書，也不論你在容器、分散式系統，還是 Kubernetes 的經驗是如何，抑或是你正在規劃使用公有雲、私有資料中心、或者混和雲環境，這本書應該都能讓你充分瞭解 Kubernetes。

1 由 Brendan Burns 等人所著作「Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade,」*ACM Queue* 第 14 卷（2016）70-93 頁，網址：*https://oreil.ly/ltE1B*。

使用容器或是 Kubernetes 這類容器 API 的原因有很多，但是我們可以大致上歸納為下列好處：

- 開發速度
- 擴展性（對於容器跟團隊而言都是）
- 抽象化基礎架構
- 效率
- 雲端原生的生態環境

以下各節，將分別說明 Kuberbetes 如何帶來這些好處。

速度

「速度」可說是當今軟體開發中的關鍵。在軟體產業中交付方法不斷在演進，從盒裝的 CD/DVD 到每小時進行更新的網路線上服務。在這種持續變化的環境中，你與競爭對手的區別通常是開發和部署新組件或功能的時間長短，或是應變對手創新的速度。

但要注意，這裡談的「速度」不是單純像字面上所代表的意思而已。雖然用戶期待著你持續改進服務，但他們更重視的是高可用性。從前我們或許可以每天半夜進行停機維護，但現在用戶期待的是在服務不中斷的前提下，軟體可以持續不斷地更新。

所以，速度的衡量標準不只是每天或每小時增加多少功能，而是如何在保持高可用性服務的前提下做到不斷的創新。

在這個條件之下，容器和 Kubernetes 可以成為你快速迭代服務的工具且同時維持可用性。主要實現這個核心概念的包含：

- 不可變性
- 宣告式組態
- 即時自我修護系統
- 可重複使用的共通程式庫及工具

這些概念都互相關聯著，從根本提升穩定部署的速度。

不可變性的重要性

容器和 Kubernetes 都鼓勵開發者建置分散式系統的時候，遵循不可變基礎架構原則。在不可變的基礎架構上，一旦 artifact 被建立，就不能被其他人改變。

傳統的軟體開發，電腦和軟體系統都當成可變基礎架構來對待。在可變基礎架構上，更新是一層一層地疊加在既有系統上。這些變動可以一次全部更新，或是在漫長的時間中一點一點更新。舉例來說，使用 `apt-get update` 工具升級系統，`apt` 依序下載需要更新的 binary，覆蓋到舊的 binary 上，並對配置文件進行修改。在可變基礎架構中，整個架構的目前狀態不能使用一個 artifact 來表示，而是一連串累積的系統更新與變更。大多的系統增量更新不只來自系統升級，還有操作人員的修改。不僅這些，在一個有大型團隊維護的系統中，這些變動或許是由不同人所操作，而且很多時候不會有任何紀錄。

相較之下，在不可變的系統中，不再是一連串的遞增更新與修改，而是建立一個全新的完整映像檔，只需要一個操作，將新的映像檔替換成舊的映像檔就完成了，不會再有增量修改。可以想像這對於傳統組態管理上是個重大轉變。

為了讓各位更加具體了解這個概念，我們以容器來舉例。在容器中可以用這兩種方式升級你的軟體：

- 登入容器中，執行指令下載新軟體，刪除舊的伺服器再啟用新的。
- 構建新的容器映像檔，放到容器儲存庫，刪除舊的容器再運行新的。

乍看之下，這兩個方法看似沒什麼分別，但哪個可以讓構建新的容器提升可靠性？

關鍵在於建立的 artifact 和如何建立的紀錄。這些紀錄可以讓你輕鬆了解新舊版本的差別，假設有錯誤發生，能夠定位修改的項目以及如何修復。

此外，構建新版的映像檔，而不是修改原來的映像檔，這代表原有映像檔依然存在，當發生問題，我們可以馬上回復原有的版本，相較之下，一旦用新的執行檔覆蓋舊的執行檔，要回復舊版幾乎是不太可能。

不可變的容器映像檔是使用 Kubernetes 的關鍵核心概念。Kubernetes 可以強制改變運作中的容器，不過這是當你已經無計可施時所用的極端做法（例如：這是唯一的方法暫時修復生產環境系統中的關鍵系統）。即便如此，在災難發生過後，這些變更也必須在稍後透過宣告式組態記錄下來。

宣告式組態

容器的不可變性也延伸到在 Kubernetes 叢集中如何描述運行中的應用程式，Kubernetes 中所有的物件都是利用**宣告式組態**，代表系統期望的運作狀態，而 Kubernetes 的工作就是確保應用程式符合你宣告的期望狀態。

就像可變對比不可變的基礎架構，宣告式組態成為**命令式**組態的替代選擇。因為命令式組態透過一連串的指令來定義，而不像宣告式組態直接宣告最終期望的狀態。命令式組態定義一連串的操作，宣告式組態定義最終狀態。

為了理解這兩種方法，想要為軟體產生三個一模一樣副本，命令式組態的做法會說「運行 A、運行 B 然後運行 C」，而宣告式組態則會說「三份相同的應用程式」。

因為宣告式組態直接描述狀態，所以它並不需要透過執行來被理解。它的作用會被具體的宣告。因為宣告式組態的結果在執行前就能夠被理解，所以不易出錯。此外，軟體開發中的傳統工具，像是原始碼管理（source control）、程式碼審查（code review）和單元測試（unit testing），皆可用於宣告式組態，而在命令式指令是不可能的實現的。在原始碼管理中儲存宣告式組態的概念被稱為「基礎架構即程式碼（infrastructure as code）」。

最近，GitOps 已經開始使用基礎架構即程式碼的概念，並將其與原始碼管理結合成為整個基礎架構唯一的事實來源。當你開始採用 GitOps 概念，任何對生產環境的改動，都必須將修改內容提交到 Git 存儲庫中，配合自動化工具才能生效。除此之外，GitOps 開始大量被雲端供應商提供的 Kubernetes 服務整合，以提供更簡單的方法利用宣告式組態管理整個雲端原生基礎架構。

Kubernetes 確保應用程式狀態符合宣告式組態的特性，搭配上版本控制系統，讓版本復原（Rollback）變得容易。只需要簡單地按照先前的宣告組態重新啟動即可。在命令式組態裡通常是不可能的，因為命令式組態描述著如何從 *A* 點轉變到 *B* 點，但通常不包含反向指令。

自我修護系統

Kubernetes 是個線上自我修護的系統。當它收到需求變更時，並不是一次性的將當下的變動進行修改，而是**持續地**確認目前整個叢及狀態是否符合最終狀態。這表示 Kubernetes 不僅會初始化系統，而且它會保護系統不被任何的故障或干擾破壞和影響可靠度。

比較傳統的操作人員會手動進行一些緩解程序或是人為干預來應對某些警報。命令式組態修復成本比較高（因為通常要一個 on-call 操作人員進行修復）。而且通常也比較慢，因為人必須要先醒來，然後才能登入系統進行回應。此外，因為就像前一節中所述，命令式組態可能存在著各種問題，用它來進行修復操作並不可靠。像 Kubernetes 的自我修護系統，能夠降低操作人員的負擔，而且能夠透過快速可靠的修復，進而提高系統整體可靠度。

我們用一個比較具體例子來說，如果你與 Kubernetes 提出三個副本的需求，Kubernetes 不會只是幫你把三個副本建立後就結束了，它會持續的確認三個副本是否存在。這時候你手動建立第四個，為了保持三個副本的數量，它會將第四個移除。同樣若你手動移除了一個副本，它將會建立一個副本以符合三個副本的最終狀態。

線上自我修護系統，改善了開發人員的效率，有效將運維的時間和精力，轉移到開發及測試新功能上。

近期 Kubernetes 也花了許多努力在 *operator* 設計模式，提供了更進階的自我修復模式。對於維護、擴展和修復特定軟體（例如：MySQL）所需要更複雜的邏輯，可以透過 operator 容器在叢集中提供服務所需的處理邏輯，與 Kubernetes 的通用自我修復功能相比，operator 的客製化程式碼讓它能夠處理更明確的進階健康檢測和修復，我們將這些概括為「operators」，這將在第 17 章談到。

擴展你的服務和團隊

隨著產品的增長，為了開發速度，擴展軟體和團隊是不可避免的。幸好 Kubernetes 可以透過支援**去耦合**架構達到可擴展性。

去耦化

在去耦合架構下，每個元件透過定義各自的 API 和服務負載平衡器與其他元件分離。API 和負載平衡器分割每一個不同的系統，並為實作者（implementer）和消費者（consumer）之間提供緩衝，而負載平衡器則為每個服務的運行實體提供緩衝。

透過負載平衡器使得去耦化元件更容易擴展程式，因為只要增加了大小（和容量），不用做任何的調整或重新配置其他層面的服務，就可以擴展程式。

透過 API 使得去耦化伺服器更容易擴展開發團隊，因為每一個團隊只要專注在單一小型的微服務的領域就行了。微服務裡簡潔的 API，降低了構建和部署軟體時跨團隊的溝通成本，而溝通成本通常是限制擴展團隊的主要因素。

讓擴展應用程式和叢集變簡單

具體來說，當你需要擴展服務時，Kubernetes 不可變與宣告式的特性，讓改變這事情變得很平常。因為你的容器是不可變，而且副本數量只是個被定義在宣告式組態的數字，向上擴展服務只要修改組態檔、跟 Kubernetes 說新的宣告式狀態，最後讓它處理剩下的事，或者直接設定自動擴容的最終狀態，讓 Kubernetes 幫你處理擴容的事。

當然，這是假設你的叢集中擁有足夠的可用資源，但有時候還是需要擴充整個叢集的資源，來滿足更多的需求。Kubernetes 同樣能非常簡單的達成這件事，因為叢集中每台機器都有相同的配置，且應用程式本身透過容器與機器硬體規格分離，因此新增資源只要新增機器並加入叢集中即可。透過簡單的指令或預先製作的映像檔就能夠完成。

這個過程中的其中一個挑戰是如何預估所需擴展機器的資源。如果你運行在實體機的基礎架構上，要取得新機器的時間會是以天或週為單位來計算。而無論在實體和雲端基礎架構上，預估費用是很困難的，因為很難估計特定的應用程式成長量或擴展需求。

Kubernetes 可以簡化對預測未來的運算成本。想像一下，考慮擴編 A、B 和 C 三個團隊。過去因為每個團隊成長的起起伏伏造成成本難以估計。假設你是為每個服務配一台機器，因為資源不能共享，所以只能根據每個服務去預測其最大成長。但如果你使用 Kubernetes 將機器與開發團隊分開，便可以基於三個服務的總成長量來預估所需的容量。將三個個別波動的成長率合併成一個，能夠降低統計雜訊並且產生較可靠的預測成長量。除此之外，將機器與特定團隊分開後，我們可以讓團隊之間共享機器的資源，減少不必要的開銷與過多的硬體資源。

Kubernetes 讓自動擴容（包含增加及減少資源）變得可能。尤其是在雲端環境上可以透過 API 來新增機器，配合 Kubernetes 上的自動擴容機制，讓應用程式跟基礎建設都可以配合當前容量進行調整，達到成本的最佳化。

利用微服務擴展開發團隊

正如各個研究指出，理想的團隊規模是「兩個披薩團隊（two-pizza team）」或大約六至八人，因為這樣的規模往往會有良好共享知識、快速制定決策和共同的使命感。較大的團隊往往會受到職級階層、能見度低及內部爭鬥的影響，這會阻礙團隊的成功和敏捷性。

但為了達成目標，許多專案需要非常多的資源，但是維持敏捷的團隊大小與達成目標所需的團隊人數往往存在著拉扯。

常見解決這種矛盾的方法是，將開發去耦化，以提供服務為導向的團隊，各自構建單一的微服務。每一個小團隊負責服務的設計並交付給其他團隊，將所有服務集結在一起成為整個產品的最終實作。

Kubernetes 提供許多的抽象和 API，能簡單的構建去耦合微服務架構，像是：

- *Pod* 或稱容器群，可以將不同團隊開發的容器映像檔劃分成單一個部署單位。
- Kubernetes *service* 提供負載平衡、命名以及發現其他服務的能力，來達成每個微服務的各自獨立性。
- *Namespace* 提供隔離和存取控制，定義每個微服務彼此授權存取的程度。
- *Ingress* 提供一個易於使用的程式前端入口，可以結合多個微服務成單一的外部 API。

去耦化應用程式的容器映像檔與機器，意味著將微服務放在同一台機器而不互相干擾，進而降低微服務架構的成本。Kubernetes 支援健康檢查（Health-checking）和滾動更新（Rollout）的功能，確保應用程式的部署有可靠的一致性，也確保團隊的微服務增加不會導致服務的生命週期和運維會有所不同。

一致性和擴展性之間不同的考量

除了 Kubernetes 帶來的操作一致性之外，Kubernetes 堆疊帶來的解耦合可以讓底層的基礎架構更容易維持一致性，如此一來小而精實的團隊，能夠管理很多機器。我們已經詳細討論了應用程式容器和機器 / 作業系統之間的去耦化，但重要的是容器編排的 API 成為一個乾淨俐落的分界，它將應用程式與叢集的操作人員職責分開。稱作「各司其職，各安其位；不在其位，不謀其政（not my monkey, not my circus）」。應用程式開發者，依據容器編排 API 提供的服務層級協議（service-level agreement, SLA），不用擔心實踐 SLA 的細節。同樣的，容器編排 API 可靠性工程師，專注在交付容器編排 API 的 SLA，而不用擔心運行在其上的應用程式。

去耦化考量的點是，一個小團隊運行的 Kubernetes 叢集可以負責支援數百甚至是數千團隊的應用程式（圖 1-1），同樣的，一個小團隊可以在全球負責十二個（或更多）的叢集。

必須注意的是，容器和作業系統的去耦合，能夠使作業系統可靠度工程師專注在個別機器上作業系統的 SLA。這成為另一種職責分離，Kubernetes 操作人員依靠作業系統的 SLA，作業系統操作人員僅需要關心作業系統本身的 SLA。而且使你可以擴展一個小規模的作業系統專家團隊去負責數千台機器。

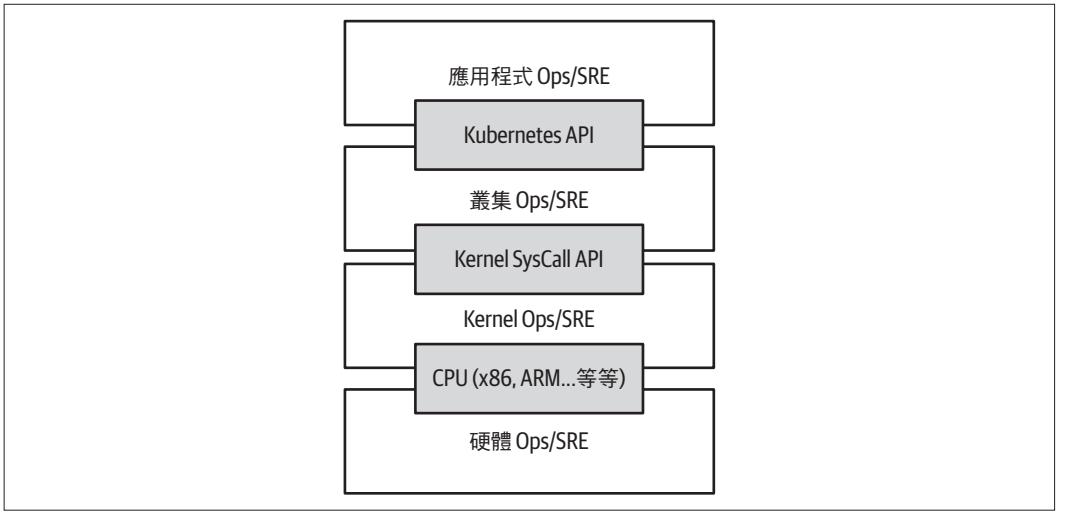

圖 1-1　各個不同維運團隊之間利用 API 解耦合的示意圖

當然，投入一個團隊去管理作業系統，也超出許多組織的能力範圍。這個時刻，公有雲供應商提供的 Kubernetes-as-a-Service（KaaS）也是不錯的選擇。隨著 Kubernetes 變得越來越普遍，相對的 KaaS 也越來越成熟可用，以至於你所知的公有雲都有這項服務。當然，使用 KaaS 有些限制，因為 KaaS 供應商會替你決定如何構建和配置 Kubernetes 叢集。像是許多 KaaS 供應商禁止使用 alpha 功能，因為它認為可能會破壞託管叢集的穩定性。

除了全托管的 Kubernetes 服務外，Kubernetes 也造就了另一種商機機，許多顧問公司協助安裝和管理 Kubernetes。在「純手工安裝（the hard way）」和全託管的服務之間有各種各樣的解決方案。

因此，是否使用 KaaS 或自行管理 Kubernetes（或許是介於兩者之間），是按照個人的技能和需求決定。就小型團隊而言，會選擇 KaaS 作為易於使用的解決方案，這樣能夠將時間和精力全心集中在構建軟體上，而不是管理叢集。對於一個能夠負擔 Kubernetes 叢集專業管理團隊的大型組織來說，自行管理是有意義的，因為在叢集的功能和操作方面提供了更大的靈活性。

抽象化你的基礎架構

公有雲的目標是為開發人員提供易於使用的自助服務基礎設施。然而，雲端 API 往往圍繞在 IT 所期望的基礎架構而非開發人員想要使用的功能（例如：「虛擬機」而不是「應用程式」）。此外，常常會因為使用不同雲端服務供應商所提供的服務，而使用不同的實作方式。直接使用這些 API 會使應用程式難以運行在多個環境中，或將應用程式同時部署在雲端跟實體環境。

像 Kubernetes 採用應用程式導向容器 API 有兩個好處。第一個，就如我們上面所述，可以將特定機器從開發人員中抽離出來。這使機器導向的 IT 角色更輕鬆，因為擴展叢集只要新增機器進去就行了。而在雲端的環境中，由於開發人員正使用根據特定雲端基礎架構實作的高階 API，因此也具有高度的可移植性。

當開發人員基於容器映像檔構建應用程式，並且基於便利的 Kubernetes API 部署它們，之後在不同甚至是混用環境間運行應用程式，只要把宣告式組態送進新的叢集即可。Kubernetes 有很多外掛程式，可以抽象化特定雲的實作方式。例如：Kubernetes 服務能夠在各個主要的公有雲以及部分私有雲上建立負載平衡器。同樣，Kubernetes 的 PersistentVolumes 和 PersistentVolumeClaims 可用在將應用程式抽象其儲存層。當然，要實現這種可移植性，你就要避免使用專屬某個雲端供應商所管理的服務（例如：Amazon DynamoDB、Azure CosmosDB 或 Google Cloud Spanner），你可以專注於部署和管理開源儲存解決方案，像是 Cassandra、MySQL 或 MongoDB。

綜合以上考量，將一切建立在 Kubernetes 應用導向的抽象概念之上，可以確保在構建、部署和管理應用程式所付出的努力，並真正達到各種環境間的可移植性。

效率

容器和 Kubernetes 除了對開發人員和 IT 管理有好處之外，抽象化也有具體的經濟效益，因為開發人員不再思考機器本身的問題，應用程式可以集中在同一台機器上，而不會影響應用程式本身的運作，這表示多個服務可以緊密安裝在少量的機器上以達到成本效益。

效率可以被機器或程式執行的比例所來衡量。當部署或管理應用程式時，很多工具和程式（例如：bash scripts、`apt` update 或命令式組態管理）是沒有效率的。當討論效率的時候，一般來說要考量到伺服器及人力成本。

運行一個伺服器，會產生耗電量、冷卻需求、資訊中心空間和原始運算能力的成本。一旦伺服器被裝上和打開電源（或點擊和轉開），儀表板就開始運行了，任何的閒置 CPU 時間都是在浪費金錢。因此，持續保持且管理伺服器達到可接受的使用率，成為系統管理員工作的一部分。這剛好是容器和 Kubernetes 工作流程能帶來的好處，Kubernetes 提供讓應用程式自動分配在機器叢集上的方法，確保比傳統方法提供更高的使用率。

進一步提高效率的方法，來自於開發人員個人專屬的一整組測試環境容器，能夠快速且便利地建立在共享的 Kubernetes 叢集中（使用 *namespaces* 功能）。過去為開發者準備一個測試環境，可能意味著要開啟三台機器，透過 Kubernetes 讓所有開發人員共享一個測試環境變得非常簡單，並將所有人集中在少量的機器群組內。降低整體機器數量，反而提高每個系統效率：因為每台單獨機器上的資源（CPU、RAM…等等的）被充分利用，所以每個容器的整體成本變得更低。

為了降低實踐完整開發測試流程所需的過高開發機器成本。例如：透過 Kubernetes 部署應用程式，可以想像成開發人員貢獻的每一個 commit，都可以有自己的完整系統來部署及測試。

當每次部署的成本是以少量容器而不是多個完整的虛擬機器（VM）來衡量時，測試所產生的成本將顯著降低。而 Kubernetes 的價值在於增加測試的同時也提高速度。因為程式碼的可靠度將會提升，完整的測試也有助於快速識別潛在問題。

最後，如同前面小節所述，透過自動化的擴充容量並在不需要的時候縮減機器的數量，可以提升整個程式運作的效率且確保程式可以有效的運作。

雲端原生的生態環境

Kubernetes 在設計之初就考量到未來的擴充性，並歡迎廣大的社群加入並強化 Kubernetes。這樣的設計目標及普遍性，帶來了圍繞著 Kubernets 開發，大量且活躍的工具及服務生態系。跟隨著 Kubernetes（在這之前還有 Docker 與 Linux）開放原始碼的腳步，許多生態系中的工具也同樣以開源的方式存在，意味著開發者可以不必從頭開始。從 Kubernetes 釋出這些年來，幾乎所有的任務都可以找到基於 Kubernetes 的工具，包含機器學習（Machine Learning）、持續開發（Continous Development），無伺服器程式運算（Serverless Programming Models）等。當然，許多挑戰並非在於尋求潛在解決方案，而是在眾多解決方案中決定較合適的特定方案。充足的雲端原生工具生態系已經

成為許多人選擇 Kubernetes 的理由。當你選擇融入雲端原生生態系，可以在系統的任何元件中使用社群開發支援的專案，讓你專注在核心業務邏輯並提供獨一無二屬於你的服務。

如同任何的開源生態系，最大的挑戰來自於擁有各種不同的解決方案，卻缺少端到端的整合方案。來自雲源生計算基金會（Cloud Native Computing Foundation, CNCF）的技術指引也許可以解決這個複雜的問題。CNCF 作為一個產業中立性的組織，負責雲端原生的專案及智慧財產權。包含三個專案的階段，作為你選擇解決方案的參考。大部分的專案在 CNCF 中是屬於沙盒階段。沙盒階段代表專案本身還在早期開發階段，不建議採用，除非你是想對專案進行貢獻的早期參與者。下一個階段稱為孵化期，孵化期的專案證實在生產環境中的可靠度及實用性，但是依舊在持續開發成長中。儘管沙盒專案有上百個，但是孵化中的專案卻僅有二十多個。最後一個 CNCF 的專案階段稱為畢業生，這些專案已經完全成熟並且大量被採用。僅有少數的專案成功的畢業，其中包含 Kubernetes 專案本身。

另一種探索雲端原生生態系的方法是透過 Kubernetes-as-a-Service 供應商的整合清單。目前大多數的 KaaS 透過雲端原生生態系的開源專案提供額外的服務，也因為這些服務被整合在雲端供應商的產品中，你也可以假設這些專案的成熟度足以運用在生產環境。

總結

Kubernetes 徹底改變了在雲端構建和部署的方式。旨在為開發人員提供更快的速度、效率和敏捷性。你正在使用的網路服務跟應用程式，目前已經有許多是建構在 Kubernetes 之上的。在你不知道的時候，其實已經成為一位 Kubernetes 的使用者。希望這個章節已經讓你了解到為什麼應該選擇 Kubernetes 來部署你的應用程式。既然你被說服了，接下來的章節將會教導你如何部署你的應用程式。

第二章

建立和運行容器

Kubernetes 提供了一個建立、部署和管理分散式應用程式的平台。這些應用程式有不同的規模、組成，但總而言之，都是由一個或多個在個別機器上運行的應用程式組成。這些應用程式接收輸入，處理資訊，輸出結果。我們在思考如何建構分散式系統前，首先應該思考如何建立**應用程式容器映像檔**，這些應用程式映像檔將成為分散式系統的主要組成元件。

應用程式通常由語言執行階段（runtime）、程式庫（library）和原始碼組成的。許多應用程式會依賴像是 `libc` 和 `libssl` 的外部程式庫。這些外部程式庫通常會安裝在作業系統中提供其他元件共用。

這類的共享程式庫經常在程式部署到生產環境的時造成問題，因為程式所需的共享程式庫僅存在於開發者筆電的作業系統中，即使開發環境與生產環境使用相同版本的作業系統，還是有可能發生開發者忘記把相依賴的程式庫或檔案放到生產環境而發生錯誤。

在傳統的軟體開發上，通常會要求在同一台機器上運行的所有程式，都需要使用相同版本的共享程式庫。當程式由不同的團隊開發的時候，這些共享的依賴關係會在各個團隊之間造成了不必要的複雜性和耦合。

程式只有在正確的環境中才能成功運作。但大多數的時候，部署應用程式有許多不同的腳本需要執行，這些腳本可能因為不同的先後順序相互影響，導致無法預期的結果，有時稱之為拜占庭將軍問題（Byzantine failure）。這會使得推出新版本（無論是整套或部分系統），會成為工作量巨大且困難的任務。

在第 1 章，我們強調不可變映像檔和基礎架構的重要性。而這正是容器映像檔所帶來的價值。稍後章節我們會提到，容器映像檔如何解決剛剛提到依賴管理和封裝的問題。

在開發應用程式的時候，如果能將應用程式打包成易於分享的格式，將會帶來許多好處。Docker 是多數人選擇的的容器工具，它讓打包應用程式變得容易，並且推送到遠端容器儲存庫（registry）供他人直接取用。撰寫本文時，主要的公有雲都已經提供了容器儲存庫（registry），同時也包含了建置映像檔所需的工具。當然你也可以選擇自行架設開源或商業版的容器儲存庫（registry），這些儲存庫可以讓使用者輕鬆管理和部署私有映像檔，而映像檔建置服務可以輕鬆地與持續交付系統整合。

在接下來及後續的章節中，我們會透過一個範例應用程式來演示整個流程，你可以在 GitHub（*https://oreil.ly/unTLs*）上找到範例應用程式。

容器映像檔將根目錄檔案系統下的應用程式和相依程式庫打包成單一 artifact。最受歡迎的容器映像檔格式是 Docker，該格式已由開放容器計畫（Open Container Initiative）認可為 OCI 映像檔的標準。Kubernetes 可以透過 Docker 和其他執行階段，支援 Docker 和 OCI 兼容的映像檔。Docker 映像檔還包括額外的中繼資料，容器執行時會根據這些資訊啟動映像檔中應用程式。

本章包含下列主題：

- 如何使用 Docker 映像檔格式打包應用程式
- 如何使用 Docker 容器執行階段啟動應用程式

容器映像檔

對大部分的人來說，第一次接觸任何容器技術，都是從映像檔開始。**容器映像檔**是一個二進位包裝檔，將所有執行應用程式必要的檔案封裝在作業系統容器中。第一次與容器接觸的經驗，可能是從本機的檔案系統建構容器映像檔，或是從**容器儲存庫**下載已存在的映像檔。無論是哪個情況，一旦映像檔儲存於本機中，你就可以透過執行映像檔來啟動作業系統容器中運行的應用程式。

最受歡迎並廣為使用的容器映像檔格式為 Docker，由 Docker 開放原始碼計畫提供的 docker 指令進行容器的包裝、分發，執行，這個格式起先由 Docker, Inc. 所開發，隨後經過開放容器計畫（Open Container Initiative）將其標準化。雖然 OCI 的標準在 2017 年中發布 1.0 版本，但是實際套用 OCI 標準的進度還是很緩慢。Docker 映像檔仍是業界標準，並且由一層層檔案系統堆疊組成。每一層都會根據前一層檔案系統的內容進行新增、刪除或修改。這是**覆蓋型**（*overlay*）檔案系統的一種。在封裝及使用映像檔時都會使用覆蓋型檔案系統，在執行階段時，此類檔案系統有多種不同的實作方式，包括 `aufs`、`overlay` 和 `overlay2`。

容器分層

「Docker 映像檔格式（Docker image format）」和「容器映像檔（container images）」這兩個名詞或許有點令人困惑。映像檔不是一個文件檔案，它的內容其實是一份清單（manifest），清單中定義其他檔案所在的位置。這份清單跟相關聯的其他檔案組合成一個映像檔。把檔案本身跟清單做一定程度的分離有助於節省儲存空間以及提升映像檔的傳輸效率。連同這個映像檔格式定義的，還有一組 API 提供映像檔的上傳與下載，以利於將映像檔推送到映像檔儲存庫上。

容器映像檔是由一系列的檔案系統層所構成的，每一層都繼承和修改上層。試著透過建立一些容器來了解這中間是如何運作的。這邊必須提醒一下，正確的階層順序應該是從下往上，但為了容易理解，我們將反過來解釋。

```
.
└─ container A: 基本的作業系統，像是 Debian
    └─ container B: 構建在 #A 之上， 新增 Ruby v2.1.10
    └─ container C: 構建在 #A 之上，新增 Golang v1.6
```

此時，我們有三個容器：A、B 和 C。B 和 C 是由 A **分支**出來的，除了容器的基本檔案之外，其他的都不會共享。接下來，我們可以基於 B 之上，新增 Rails（版本 4.2.6）。有時候可能為了支援舊版的應用程式，需要有舊版本的 Rails（例如：版本 3.2.x）。我們可以基於 B 的應用程式，構建容器映像檔，並在未來計畫遷移至 Rails 4。

```
.（從上面接著繼續）
└─ container B: 構建在 #A 之上， 新增 Ruby v2.1.10
    └─ container D: 構建在 #B 之上， 新增 Rails v4.2.6
    └─ container E: 構建在 #B 之上， 新增 Rails v3.2.x
```

概念上，每個容器映像檔層，構建在前一層之上。前一層的參考是個指標。這只是個簡單的例子，但在真實世界中，容器可以是更大更廣泛的有向無環圖的一部分。

容器映像檔，通常包含容器組態檔，說明如何設定容器環境和應用程式進入點。容器組態，包含如何設定網路、namespace 隔離、資源限制（cgroups），以及運行的容器實例上應放置哪些**系統調用**（*syscall*）限制的訊息。容器的根檔案系統和組態檔，通常透過 Docker 映像檔配合使用。

容器種類可分為兩大類：

- 系統容器
- 應用程式容器

系統容器模仿虛擬機器執行開機程序，同時也可能包含系統服務，像是 `ssh`、`cron` 和 `syslog`。Docker 剛誕生時，系統容器是最為常見的類型。隨著時間的流逝，系統容器已被認為是不好的做法，而應用程式容器才是首選。

應用程式和系統容器的差異在於應用程式容器通常運行單一應用程式。雖然說限制一個容器只能運行一個應用程式看似有點太嚴格，但是這樣的粒度大小卻非常適合可擴充應用程式的組成，而且也成為 Kubernetes 中 Pod 的設計理念。我們在第 5 章會更詳細介紹 Pod 是如何運作。

利用 Docker 建立應用程式映像檔

通常像 Kubernetes 這樣的容器編排系統，專注於構建和部署由應用程式容器組成的分散式系統，因此我們將在本章節接下來的部分，重點介紹應用程式容器。

Dockerfile

Dockerfile 可用於建立 Docker 容器映像檔。

我們從建立一個 Node.js 的應用程式映像檔開始介紹，其他動態語言（例如：Python 或 Ruby）也可以參考下面的範例。

這個應用程式中會包含 npm、Node 和 Express，裡面有兩個檔案：*package.json*（範例 2-1）和 *server.js*（範例 2-2）。將檔案都放在目錄中，然後執行 `npm install express --save` 建立對 Express 的相依關係並安裝。

範例 2-1　package.json

```
{
  "name": "simple-node",
  "version": "1.0.0",
  "description": "A sample simple application for Kubernetes Up & Running",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
```

```
  "author": ""
}
```

範例 *2-2*　*server.js*

```
var express = require('express');

var app = express();
app.get('/', function (req, res) {
  res.send('Hello World!');
});
app.listen(3000, function () {
  console.log('Listening on port 3000!');
  console.log('  http://localhost:3000');
});
```

要將它們打包為 Docker 映像檔，需要建立另外兩個檔案：*.dockergnore*（範例 2-3）和 Dockerfile（範例 2-4）。Dockerfile 是一份描述如何建立一個容器映像檔的步驟清單，而 *.dockerignore* 定義了將檔案從本機複製到映像檔時應該忽略的部分。可以至 Dockerk 提供的網站上找到更完整的 Dockerfile 用法（*https://dockr.ly/2XUanvl*）。

範例 *2-3*　*.dockerignore*

```
node_modules
```

範例 *2-4*　*Dockerfile*

```
# 從 Node.js 16 (LTS) 映像檔作為最初映像檔 ❶
FROM node:16

# 指定所有的指令都在指定的目錄中執行 ❷
WORKDIR /usr/src/app

# 複製套件檔案及安裝相依套件 ❸
COPY package*.json ./
RUN npm install
RUN npm install express

# 複製所有 app 的檔案進映像檔 ❹
COPY . .

# 指定映像檔啟動後預設指令 ❺
CMD [ "npm", "start" ]
```

❶ 每個 Dockerfile 的都建立在其他容器映像檔的基礎上，這行我們指定 Docker Hub 中的 `node:16` 為基礎映像檔，這是 Node.js 16 的預設映像檔。

❷ 此行指示容器映像檔為接下來的指令定義工作目錄。

❸ 這三行初始化 Node.js 的相依關係，第一步，我們將套件的檔案複製到映像檔中，包含 *package.json* 和 *package-lock.json*，接著利用 `RUN` 指令在容器中執行指令以安裝必要的相依套件。

❹ 現在，我們將其餘檔案複製到映像檔中，有些檔案是被 *.dockerignore* 定義排除的，所以除了 *node_modules* 這個資料夾以外的其他檔案都會複製到映像檔中。

❺ 最後，指定在啟動容器時應執行的指令。

執行下面的指令構建 `simple-node` 的 Docker 映像檔：

```
$ docker build -t simple-node .
```

當你要運行該映像檔時，可以透過以下指令來執行，然後，打開網址 *http://localhost:3000* 來存取在容器中運作的程式：

```
$ docker run --rm -p 3000:3000 simple-node
```

此時 `simple-node` 映像檔存在於本機的 Docker 儲存庫，而且只能從本機存取，而 Docker 的強項是可以讓數以千計的機器和廣大的 Docker 社群共享映像檔。

最佳化映像檔大小

建置映像檔時，隨著時間，會開始遇到映像檔越來越大的問題。首先要記住的是，藉著後續資料層刪除的系統檔案，依然存在於映像檔中，只是無法被存取。就像以下情況：

```
.
└── 資料層 A：包含大檔案，名稱為「BigFile」
    └── 資料層 B：移除「BigFile」
        └── 資料層 C：建構於 B 之上，新增一個靜態 binary
```

你以為 *BigFile* 不再存在於這個映像檔內了嗎？畢竟，當你在運行這個映像檔時，該檔案已經無法被存取。但事實上，該檔案仍然存在資料層 A，意思是每當你推送或提取這個映像檔，即使不能存取 *BigFile*，但這個檔案仍然透過網路傳送著。

另一個需要注意的地方是映像檔構建快取。請記住每一個資料層都只記錄跟上一層之間的改變。每次修改一個資料層，在該層之後的每一資料層都會跟著受到影響。修改之前的資料層，意味著需要重建、重推送和重提取映像檔以部署到開發環境中。

要更理解深入，可以參考這兩個映像檔：

```
.
└── 資料層 A：包含基本作業系統
    └── 資料層 B：新增程式碼 server.js
        └── 資料層 C：安裝「node」套件
```

對比

```
.
└── 資料層 A：包含基本作業系統
    └── 資料層 B：安裝「node」套件
        └── 資料層 C：新增程式碼 server.js
```

在第一次映像檔被提取執行的時候，兩個映像檔執行的結果都是相同的。想像一下，如果 *server.js* 改變，會發生什麼事。在第二種情況下，只需提取和推送 *server.js* 資料層。但是在第一種情況下，*server.js* 和 node 套件的資料層都需提取和推送，因為 node 資料層依賴 server.js 資料層。一般來說，為了最佳化推送和存取映像檔的大小，會按照修改頻率，從最少修改到最常修改來排序資料層。這就是為什麼在範例 2-4 中，我們先複製 *package*.json*，並安裝相依套件，最後才再複製其他檔案的原因，因為開發人員修改應用程式的頻率比相依套件還要來的高。

映像檔的安全

談到安全性，沒有任何一條捷徑。建置會運行在生產環境的 Kubernetes 叢集中的映像檔時，請確保遵守封裝及發布應用程式的最佳實踐原則。舉例來說，不能將密碼放在容器裡，不只是最後一層，在映像檔的任一層都不行。容器資料層有個不怎麼直觀的問題，就是刪除一個資料層的檔案並不會真正從之前的資料層中刪除該檔案。它仍然佔用空間，任何擁有正確工具的人都可以存取，攻擊者可以簡單地創建一個包含密碼資料層的映像檔。

機敏資訊和映像檔**千萬不能**放在一起。這麼做容易被駭客攻擊，並且會給你的公司或部門帶來羞辱。我們都想上電視，但有更好的方式可以辦到。

此外，映像檔通常用來執行單一應用程式，因此最佳的做法是盡可能減少單一映像檔中的檔案數量，因為每一個額外的程式庫都有可能增加攻擊的面向，進而使你的應用程式產生漏洞。依照每個程式語言特性的不同，有些程式語言的映像檔可以非常的小，僅包含最小相依性的檔案。越少的相依性確保你的映像檔不會暴露在未使用的程式庫漏洞下。