第一章

挑戰與原則

新一代的基礎架構管理技術，有望改善我們管理 IT 基礎架構的方式。但現今許多組織仍看不出有任何顯著的差異，此外有些組織發現，這些工具甚至會讓事情變得更複雜。正如我們將看到的，「基礎架構即程式碼」（infrastructure as code）所提供的原則（principle）、實施方法（practice）和模式（pattern），讓我們得以有效地使用這些技術。

為何需要「基礎架構即程式碼」？

虛擬化、雲端、容器、伺服器自動化和軟體定義網路，應該可以簡化企業的 IT 維運工作（operations work），讓服務的配置（provision）、組態設定（configure）、更新（update）和維護（maintain）得以花費較少的時間和精力。此外，問題應該可以迅速偵測並得到解決，而且系統的組態應該具一致性和最新的狀態。花在例行工作所花的時間少了，IT 人員就有時間迅速進行變更和改善，協助自己的組織適應現代世界不斷變化的需求。

不過，即使有最新、最好的新工具和平台，IT 維運團隊（operations teams）仍然發現他們無法跟上每天的工作量。他們沒有時間解決系統長期存在的問題，更不會有時間去改造系統以充分利用新工具。事實上，雲端（cloud）和自動化（automation）往往讓事情變得更糟。輕易就能配置（provisioning）新的基礎架構，將導致一個不斷擴展的系統組合，但是為了避免發展到不可收拾的地步，所需花的時間將不斷增加。

採用雲端服務和自動化工具，可立即降低對基礎架構進行變更的門檻。雖然這種管理變更的方式改善了一致性和可靠性，但仍無法脫離軟體的框架。人們需要思考如何使用這些工具，如何建立系統、流程和習慣，以便能夠有效地使用它們。

在雲端和自動化普及之前，有些 IT 組織仍舊透過（以往管理基礎架構和軟體的流程、結構、規定等）老方法來回應此一挑戰。然而這些老法僅適用於需要幾天或幾星期才能配置一台新伺服器的時代，但是現在只需要幾分鐘或幾秒的時間。

傳統的「變更管理流程」（change management processes）常會被需要把事情做好的人給忽略、跳過或否決[1]。那些實施這些流程較為成功的組織，會漸漸發現自己被技術上更為靈活的競爭者所超越。

面對雲端與自動化的變化速度，傳統的變更管理方式顯得左支右絀。但是仍然需要應付雲端與自動化工具所建立之不斷成長、持續變化的系統。這就是為何要導入「基礎架構即程式碼」[2] 的原因。

鐵器時代和雲端時代

在 IT 的「鐵器時代」（iron age），系統被侷限在實際的硬體中。配置和維護基礎架構全是人工作業（manual work），為了讓事情能夠進行下去，人們必須花時間操作滑鼠和鍵盤。變更牽涉到如此多的工作，故「變更管理流程」注重謹慎的事前考量、設計和審查工作，因為犯錯的代價非常昂貴。

在 IT 的「雲端時代」（cloud age），系統與實際的硬體是分離的。例行的配置和維護工作可以委派給軟體系統來進行，這讓人們得以從苦差事中解脫。任何的變更若不能在幾秒內，亦可在幾分鐘內完成。變更管理可以利用這樣的速度，提供更好的可靠度以及更快的上市時間。

什麼是「基礎架構即程式碼」？

「基礎架構即程式碼」係基於「軟體開發實務」（software development practice）的一種「基礎架構自動化」（infrastructure automation）解決方案。它著重在，為系統的配置（provisioning）、變更（changing）以及組態（configuration），提供一致、可重複的程序。變更的進行方式為：撰寫定義檔，然後透過無人參與的流程（包括充分的驗證）來變更系統。

1 「影子 IT」（Shadow IT）是指人們置正式的 IT 管控規定於不顧，帶自己擁有的設備到公司，購買並安裝未經許可的軟體，或採用雲端託管服務。這通常意味著，內部的 IT 部門無法跟上其所服務之組織的需求。

2 「基礎架構即程式碼」（infrastructure as code）這個專有名詞並沒有明確的出處或作者。撰寫本書當時，我試著找出明確的出處，但並沒有明確的結果。我能找到最早的參考資料是在 2009 年 Velocity 會議中 Andrew Clay-Shafer 和 Adam Jacob 的演講。John Willis 的一篇關於該會議的文章，可能是第一份記錄此一名詞的文件（*http://itknowledgeexchange.techtarget.com/cloud-computing/infrastructure-as-code/*）。Luke Kaines 已公開承認，這可能是從他開始的，相關人士都認同這個說法。

自動化恐懼

在 DevOpsDays 會議（*http://www.devopsdays.org/*）的一場關於「組態自動化」（configuration automation）的開放空間對談（*http://en.wikipedia.org/wiki/Open_Space_Technology*）中，我問大家有多少人使用過自動化工具，像是 Puppet 或 Chef。大多數的人都舉了手。我又問有多少人在自動化排程（automatic schedule）中執行那些無人監督的工具。大部分的人便放下了手。

許多人都有同樣的問題。早期我自己在使用自動化工具的時候，也是選擇性地使用自動化工具——例如，用於協助新伺服器的建構，或是進行特定組態的變更。每次執行它的時候，我都會調整它的組態，以便配合我正在進行的特定任務。

我不敢把我的自動化工具放著不管，因為我對它們所做的事沒有信心。

我之所以會對我的自動化沒有信心，是因為我的伺服器的組態並不一致。

我的伺服器的組態之所以會缺乏一致性，是因為我並未頻繁且一貫地運行自動化。

這就是自動化恐懼螺旋，如圖 1-1 所示，而基礎架構團隊需要打破此螺旋，好讓自動化的使用能夠順利。打破此螺旋的最有效方法就是面對恐懼。挑擇一組伺服器，調整其組態定義，這樣你就會清楚它們的作用，接著把它們安排到無人監督的排程中執行，至少一小時運行一次。然後再挑選另一組伺服器，重複此過程，直到你所有的伺服器都能持續不斷更新。

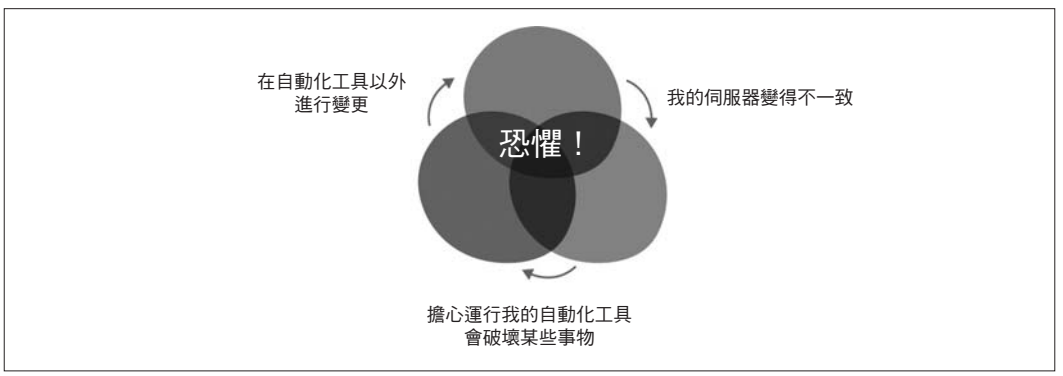

圖 1-1　自動化恐懼螺旋

良好的監控和有效的自動化測試機制（本書第三篇所要探討的內容）讓我們有信心建構可靠的組態，以及能夠快速解決問題。

無論動態基礎架構平台是雲端服務、虛擬化或實體裸機都沒關係，重要的是命令稿和工具可以用來自動建立（create）和銷毀（destroy）基礎架構的元素、回報這類元素的狀態以及管理中介資料（metadata）。

在本章稍後會探討不同類型的平台以及如何選擇它們的考慮因素。

實作你的動態基礎架構平台

雖然 AWS 之類的公有基礎架構雲端服務，是最著名之動態基礎架構平台的案例，但許多組織還是會實作自己的私有基礎架構雲端服務。本書側重於，在現有的平台上建構基礎架構，所以迴避了如何建構自己的基礎架構平台的議題。即便如此，本章應該協助讀者理解，為了支援「基礎架構即程式碼」，一個平台需要具備的特徵。

動態基礎架構平台的特徵

「基礎架構即程式碼」就是把基礎架構視為一個軟體系統；這意味著，動態基礎架構平台具有某些特徵。也就是說，這樣的平台必須是：

- 可程式化（Programmable）
- 隨選即用（On-demand）
- 自助式服務（Self-service）

接下來的幾節中，我們將會討論這些特徵的細節。

依據 NIST 的定義，雲端服務所具有的特徵

美國國家標準暨技術研究院（NIST），在所發表的一篇文章中，對雲端運算做了極好的定義（*http://www.nist.gov/itl/cloud/*），文章中列出了雲端運算的五個基本特徵：

- 隨選即用、自助式服務（配置）
- 寬泛的網路存取（以「標準機制」透過網路來提供）
- 資源彙整（多重租賃）
- 迅速、彈性（可以快速新增和刪除元素，甚至是自動化的）
- 量測服務（可以量測資源的使用情況）

動態基礎架構平台的定義比雲端運算還要廣泛。資源彙整（resource pooling）並非「基礎架構即程式碼」的基本特徵，而這也意味著，量測可能是非必要的。

可程式化

動態基礎架構平台必須是可程式化的。能有一個使用者介面可用會很方便，而且大多數虛擬化產品和雲端服務供應商都有一個這樣的介面，但是命令稿、軟體和工具必須能夠與平台互動，而這需要用到一個 API 來進行程式設計。

即便使用現成的工具，無論如何，大多數團隊最終還是會撰寫一些小型的客製化命令稿和工具。因此，基礎架構平台的 API 必須對命令稿語言有良好的支援，讓團隊方便使用。

大多數的基礎架構平台，係透過一個網路 API 來公開其管理功能，而且使用的是基於 REST 的 API（*http://en.wikipedia.org/wiki/Representational_state_transfer*），由於其易用性和靈活性（圖 2-1），因此廣為人知。

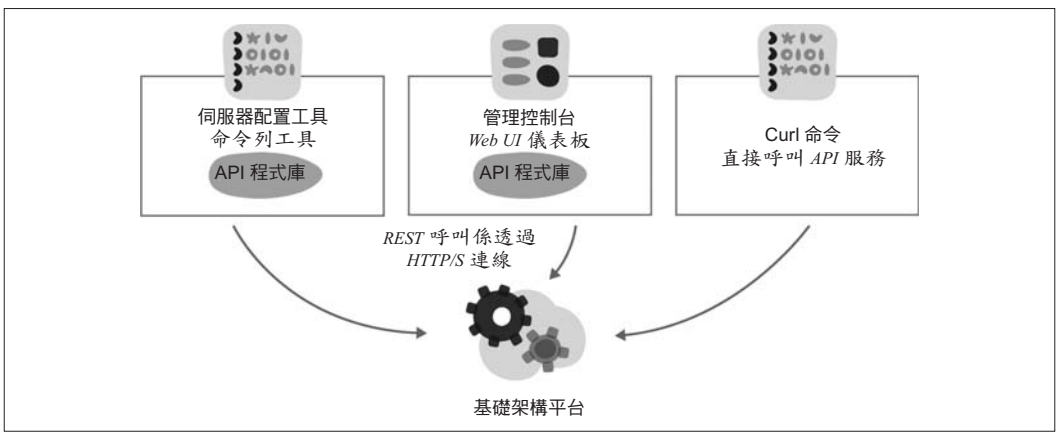

圖 2-1　基礎架構平台的 API 用戶端

根據 REST API 來進行編程和命令稿編寫並不難，但是使用特定語言的程式庫（其中封裝了 API 的細節，並提供代表基礎架構之元素的類別［ classes ］和結構［ structures ］，以及操作它們的方法）可能會有所幫助。動態基礎架構平台的開發人員通常會為幾個熱門的語言提供專屬的 SDK，還有許多開源專案為多種平台提供全面的 API，像是 Ruby Fog（*http://fog.io*）和 Python Boto（*http://boto3.readthedocs.org/*）等程式庫。

平台提供之基礎架構資源

基礎架構的管理具有許多可移動的部分。動態基礎架構平台提供了三塊關鍵積木：運算、儲存和網路（圖 2-2）。

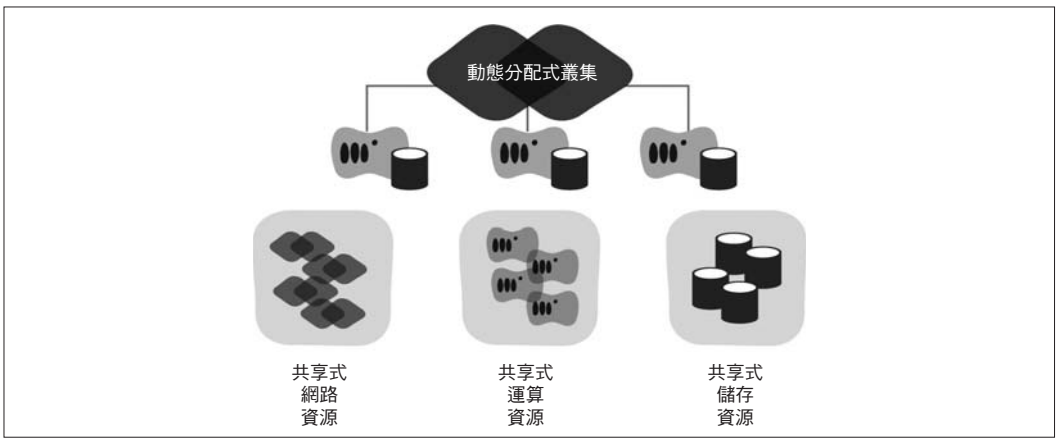

圖 2-2　動態基礎架構平台所提供的核心資源

儘管大多數平台所提供的服務不僅僅這三種，但幾乎所有其他的服務都只是這三種服務的變形（例如，不同類型的儲存裝置）或是把服務結合起來並運行在其上。實際的例子包括：伺服器映像檔管理（以儲存裝置來支援運算）、負載平衡（網路，可能會使用運算實例）、訊息傳送（網路和運算）以及身分驗證（一個在運算實例上執行的應用程式）。即使無伺服器運算服務，例如 Kinesis（*https://aws.amazon.com/kinesis/*），也是透過在一般的運算資源上執行工作來實現的。

運算資源

運算資源（compute resources）就是伺服器實例（server instances）。任何的動態基礎架構平台都會提供建立和銷毀虛擬伺服器（虛擬機）的方法，以及一系列的服務和功能，使得伺服器的管理更簡單、更強大。

建構、設定和管理伺服器的系統和流程，對大多數基礎架構團隊來說，是最耗費時間和精力，也是最頭大的部分，因此本書中有很多內容都著重於此。

私有的「基礎架構即服務」（IaaS）雲端服務

私有雲（private cloud）則是為單一組織內的多個客戶（例如，公司內的部門或團隊）所建構和運行的雲端服務。供應商可以為組織建構、運行甚至代管私有雲，但資源並不與其他組織共享。該組織通常需要為「可用總容量」支付費用，即便不是全部使用，不過該組織可能會根據其用途和會計預算成本，制定內部的收費或容量限制規則。

如同所有的雲端服務，私有的 IaaS 雲端服務允許客戶在自助式服務模型（self-service model）中按需要自動配置（provision）資源，而且可以自動建立和銷毀這些資源。有些組織設置（set up）了沒有這些特性的虛擬化基礎架構，並稱之為私有雲，但正如下一節所解釋的，這實際上是手動（hand-cranked）虛擬化。

IaaS 雲端產品的例子包括 CloudStack、OpenStack 和 Vmware vCloud。

雲端的類型：IaaS、PaaS 和 SaaS

IaaS、PaaS 和 SaaS 係用來理解雲端上不同服務模型的術語。它們每一個在意義上皆是雲端服務，允許多個用戶共享運算資源，但它們的用戶往往是非常不同的。

NIST 的雲端定義對這三種服務模型有非常清楚且有用的定義。下面是我對它們的描述：

軟體即服務（*Software as a Service*，亦即 *SaaS*）
: 一個讓終端用戶共享的應用程式。這可能是使用者面向的，例如 Web 模式（web-hosted）電子信箱，但事實上存在一些針對基礎架構團隊的 SaaS 產品，包括監控，甚至是組態管理伺服器。

平台即服務（*Platform as a Service*，亦即 *PaaS*）
: 一個讓應用程式開發者得以建構、測試及託管其軟體的共享平台。它會將底層的基礎架構抽象化，因此開發人員不需要擔心，像是需要為他們的應用程式或資料庫叢集分配（allocate）多少伺服器之類的事情：它就是會發生。許多的 IaaS 雲端供應商會提供本質上是 PaaS 元素的服務（例如，託管資料庫，像是 Amazon RDS）。

基礎架構即服務（*Infrastructure as a Service*，亦即 *IaaS*）
: 一個共享的硬體基礎架構，系統管理員可以用它來為服務建構虛擬基礎架構。

雲端服務與虛擬化的機械同理心

Martin Thompson 從一級方程式賽車選手 Jackie Stewart 那裡借用了「機械同理心」（Mechanical Sympathy）一詞，並把它帶到了資訊產業（IT）[3]。像 Stewart 這樣一位成功的車手，天生具備理解其賽車是如何運作的能力，因此他可以發揮出賽車的最大效益，並避免故障的發生。就一位 IT 專業人員來說，對系統的運作原理（從堆疊到硬體）有越深入、越深刻的瞭解，就越能夠從中獲得最大的收益。

整個軟體的歷史離不開將一個抽象層置於另一個抽象層上。作業系統、程式語言以及今日的虛擬化，都會透過簡化人們與電腦系統的互動方式，來幫人們提高工作效率。

你不需要擔心哪一個 CPU 的暫存器已經被儲存了特定值，也不需要思考如何為不同的物件分配堆積（heap）以避免重疊，甚至不需要在乎特定的虛擬機運行在哪一個硬體主機上。

除非由你來做的時候。

硬體仍舊隱藏在抽象層之下，而了解 APIs 和虛擬 CPU 單元的背後發生了什麼事，非常有用（圖 2-3）。它可以幫助你建構出「能夠從容處理，硬體之故障、避開隱藏之效能瓶頸、利用潛在之同理心的系統」——比起單純的軟體撰寫，透過調校（tweak）讓軟體與底層系統結合，更可靠、更有效率。

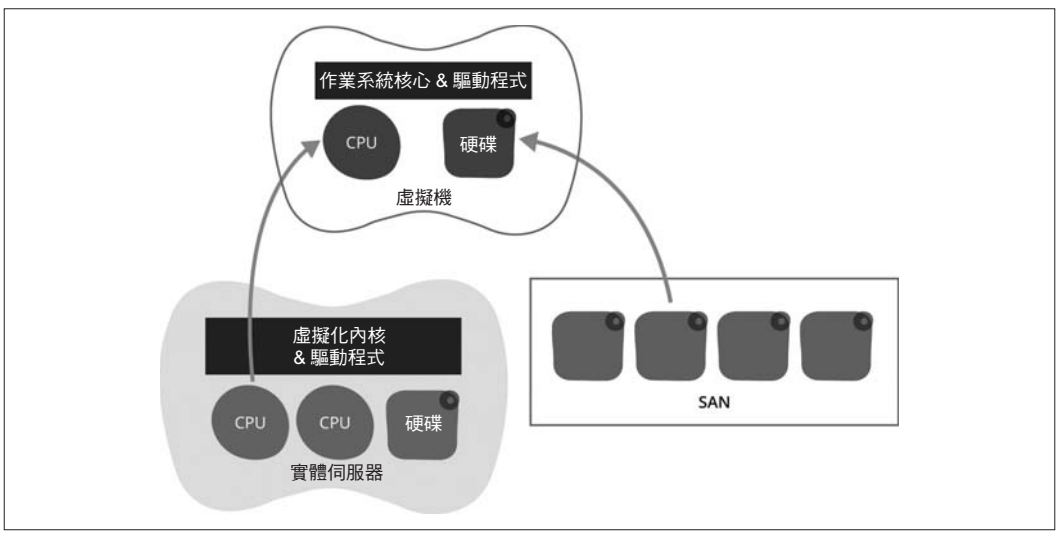

圖 2-3　虛擬化的抽象示意圖

3　見 Martin 的部落格貼文“Why Mechanical Sympathy”（*http://mechanical-sympathy.blogspot.co.uk/2011/07/why-mechanical-sympathy.html*）。

一些黑箱式自動化（black box automation）工具內建了自己的 VCS 功能。但因為那不是這類工具的核心功能，它們通常不具備獨立之 VCS 所提供的完整功能。外部化組態允許團隊去選擇他們需要的 VCS 功能，之後如果他們找到更好的系統，可以直接抽換。

使用 VCS 的基礎知識

現代的軟體開發團隊不加思索就會使用 VCS。然而，對於許多系統管理團隊來說，這並不是他們的工作流程（workflow）中自然而然的一部分。

VCS 對「基礎架構即程式碼」扮演著樞紐（hub）的角色。凡可以對基礎架構進行的任何操作，皆可記錄下來並轉換成命令稿、組態檔和定義檔，並簽入（check into）VCS。當有人想要做出變更，他們可以簽出（check out）VCS 中的檔案，透過編輯檔案進行變更，然後將新的檔案版本提交回 VCS。

將變更提交到 VCS，使其能夠應用到基礎架構。如果團隊使用了變更管理流水線（如第 12 章所述），可讓變更自動應用於測試環境並被測試。這些變更是藉由一系列的測試階段來進行的，只有通過所有測試後才能在重要的環境上使用。

VCS 是真相的唯一來源。如果兩個團隊成員簽出（check out）檔案並對基礎架構進行變更，他們可能會做出彼此不相容的變更。例如，我可能要對應用伺服器增加組態，而當下 Jim 正好在修改防火牆規則，恰好阻擋我存取自己的伺服器。如果我們每個人都直接對基礎架構應用各自的變更，我們無法得知當時其他人正在做什麼。我們可能會不斷應用變更，試圖獲得我們想要的結果，但我們也將持續抵銷彼此所做的努力。

但是，如果我們每個人都需要透過提交各自的變更，才能應用它，那麼我們將馬上瞭解，我們的變更是如何發生衝突的。VCS 中檔案的當前狀態，可以準確地表示將應用於我們的基礎架構的內容。

如果我們每次提交（commit）時，皆使用持續整合（continuous integration）自動應用我們的變更來測試基礎架構，那麼一旦有人做出導致損害的變更時，我們就會收到通知。如果我們養成在提交自己的變更之前，先拉取（pull）最後一版變更的好習慣，那麼當第二個人要提交時（在其提交之前）將發現，已經有人做了可能會影響到自己工作的變更。

基礎架構團隊的工作流程在第 13 章將會有更詳細的討論。

第四章

伺服器組態工具

使用命令稿和自動化來建立、配置和更新伺服器並非什麼新鮮事，但在過去這十年左右的時間裡，出現了新一代的工具。CFEngine、Puppet、Chef、Ansible…等等對這類工具做了最好的定義。虛擬化和雲端服務讓大量建立和更新伺服器實例的工作變得容易許多，從而推動了這些工具的普及。

最近才出現的容器化工具（containerization tools），例如 Docker，經常被用於包裝（packaging）、派送（distributing）和運行（running）應用程式及流程。容器將作業系統的元素與應用程式綑綁在一起，這對伺服器的配置和更新方式造成了影響。

正如前一章所提到的，並不是所有的工具都被設計來應用於「基礎架構即程式碼」。前一章挑選工具的準則，同樣也適用於伺服器組態工具；它們應該可以被寫成命令稿、在無人監督的情況下執行，以及使用外部的組態檔。

本章將介紹伺服器自動化工具如何應用於「基礎架構即程式碼」。這包括工具可以採取的不同方法，以及團隊為自己的基礎架構實作這些工具時，可以使用的不同方法。

伺服器管理模式

第二篇所包含的章節將以本章的內容為基礎：第 6 章將討論配置伺服器的通用模式和方法，第 7 章會更深入探討伺服器模板的管理辦法，第 8 章則會討論伺服器變更管理的模式。

自動化伺服器管理的目標

使用「基礎架構即程式碼」來管理伺服器組態將導致以下結果：

- 新的伺服器可以按需要配置[1]，不用等幾分鐘即可完成。
- 新的伺服器無須人工干預即可完成配置——例如，對事件的回應。
- 一旦伺服器組態變更已定義，無須人工干預即可將其應用到伺服器。
- 每次變更都將應用到與之相關的所有伺服器，並反映在變更完成後新配置的所有伺服器上。
- 配置和將變更應用到伺服器的過程是可重複、一致、自帶文件（self-documented）且通透（transparent）的。
- 將變更用於配置伺服器和更改其組態的過程，既簡單又安全。
- 每次變更伺服器組態定義的時候，便會進行自動測試，而且當涉及到配置和修改伺服器的任何程序時，也會進行自動測試。
- 對組態的變更，以及對在基礎架構上執行任務之流程的變更，被版本化（versioned）並應用於不同的環境，以便支援受控測試（controlled testing）和分階段發布（staged release）策略。

用於不同伺服器管理功能的工具

為了瞭解伺服器管理工具，將伺服器的生命週期（lifecycle）看作有幾個階段是有幫助的（如圖 4-1 所示）。

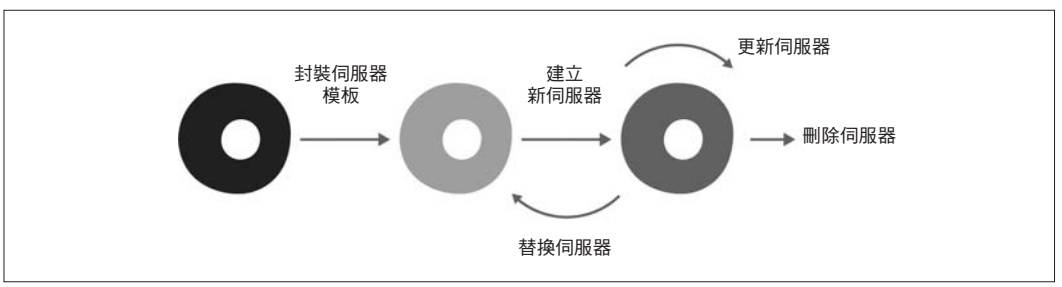

圖 4-1　伺服器的生命週期

1　請參閱第 3 章對「配置」（Provisioning）的定義，以瞭解我如何在本書中使用該術語。

- 來自 Google 的 lmctfy（*https://github.com/google/lmctfy*），該專案已經終止並併入 Docker 使用的 libcontainer[6]
- 來自 VMware 的 Bonneville（*http://blogs.vmware.com/cloudnative/introducing-project-bonneville/*）

將執行期需求（runtime requirements）與主機系統分離的好處，對於基礎架構管理來說尤其強大。它在基礎架構和應用程式之間的關係做了清楚的切割。主機系統只需要安裝容器的執行期軟體，然後它就可以執行幾乎任何的容器映像[7]。應用程式、服務、任務及它們所有的依賴關係一同被封裝到容器中，如圖 4-3 所示。這些依賴關係可能包括作業系統套件、程式語言的執行期、程式庫和系統檔案。不同的容器可能具有不同的、甚至是衝突的依賴關係，但在相同的主機上運行仍舊沒有問題。我們可以對依賴關係進行變更，而無須對主機系統進行任何變更。

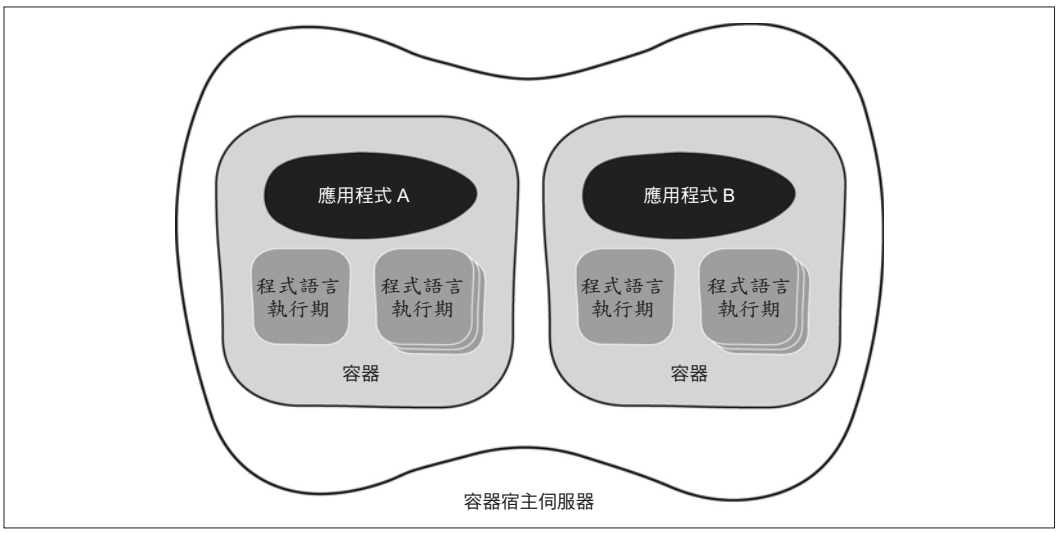

圖 4-3　在容器中隔離套件和程式庫

6　目前容器化的變化速度相當快。在寫這本書的過程中，我不得不將內容，從原先幾個段落擴展成一節，並以其做為管理伺服器組態的主要模型之一。許多我所描述的細節在你閱讀本文的同時，可能已經改變了。但希望這一般的概念，特別是容器如何對應到「基礎架構即程式碼」的原則與實施方法，仍然是相關的。

7　實際上，在主機和容器之間仍存在一些依賴關係。特別是，容器實例使用了主機系統的 Linux 核心，因此當運行不同版本的核心時，映像檔可能會有不同的行為，甚至會失敗。