前言

很久很久以前，在很遠很遠的一間資料中心裡，有著一個古老的強大族群，人們稱其為「系統管理員」（sysadmins），他們習於手動部署基礎設施。每一部伺服器、每一套資料庫、每一組負載平衡器、還有網路組態裡的一點一滴，都是由他們親手建立及管理的。那是一個充滿黑暗與恐懼的年代：害怕當機離線、害怕意外的錯誤設定、害怕既緩慢又脆弱的部署方式、甚至擔心萬一系統管理員投向黑暗面（譬如去休假）的後果。而如今，感謝 DevOps 運動的興起，這一切有了曙光，我們有更好的方式可以完成任務了：它就是 *Terraform*。

Terraform（*https://www.terraform.io*）是一套開放原始碼的工具，由 HashiCorp 所開發，只需透過一套簡單的宣告式語言（declarative language），你就可以用它來定義自己的基礎架構即程式碼（infrastructure as code），並將相關的基礎設施部署到各式各樣的公有雲供應商環境[譯註]當中（像是 Amazon Web Services（AWS）、微軟的 Azure、Google Cloud Platform、以及 DigitalOcean）等等，並自行管理，也可以只靠少量命令便部署到私有雲及虛擬化平台上（例如 OpenStack、VMware 等等）。舉例來說，你不需再於網頁中四處亂點、或是執行成打的命令，只需以下一小段程式碼，便可在 AWS 設定好一套伺服器：

```
provider "aws" {
  region = "us-east-2"
}
```

[譯註] 原本沒打算在前言中加上「譯註」，但是 provider 一詞確實需要釐清一番。原本如果以雲端供應商來說，將 provider 一詞譯為供應商並無不妥；但是當這個術語涵蓋的範圍愈發廣泛時，譯為供應商的準確性便有待商榷了。因此下文中再出現此一名詞時，如果超出雲端業者的服務範圍，譬如某些可以接受 Terraform 操作的介面或 API，便會以較為廣義的「供應端」來稱呼、或是直接引用原文名詞 provider，這兩者可能交互出現，但涵義是一樣的。

```
resource "aws_instance" "example" {
  ami           = "ami-0fb653ca2d3203ac1"
  instance_type = "t2.micro"
}
```

若要部署，只需這樣做：

```
$ terraform init
$ terraform apply
```

多虧了它的簡約和威力，Terraform 已經成長為 DevOps 領域的要角。它幫你把原本繁瑣、脆弱、需要手動進行的基礎設施管理動作，替換成可靠的自動化根基，讓你可以在這個基礎上建構其他 DevOps 成品（例如自動化測試、持續整合（Continuous Integration）、持續部署（Continuous Delivery））和各種工具（例如 Docker、Chef、Puppet）。

本書可說是將 Terraform 運作起來的捷徑。

讀者們會先著手部署最基本的「Hello, World」這個 Terraform 範例（其實我們剛剛已經做過了！），然後一路學到如何運行整套的技術堆疊（虛擬伺服器、Kubernetes 叢集、Docker 容器、負載平衡器、資料庫等等），而這個堆疊有能力支援大量的流量和成群的開發人員——這些都會在本書章節中一一說明。本書可說是一本動手做的指南，而不只是告訴你 DevOps 和基礎架構即程式碼（infrastructure as code，IaC）的相關理論，書中會帶領讀者們遍覽成打的範例程式碼，讓你可以自行嘗試，因此請務必準備一套可以放心實驗的電腦。

一旦讀完本書，你就可以在現實中運用 Terraform 了。

誰該閱讀本書

只要是必須在程式碼寫好後擔負相關責任的任何人，這本書都適合你。包括系統管理員、營運工程師、發行（release）工程師、站台可靠性工程師（site reliability engineers）、開發暨維運（DevOps）工程師、基礎架構開發者、全端（full-stack）開發者、工程經理人、甚至技術長也不例外。無論你的頭銜為何，只要你是負責管理基礎架構、部署程式碼、設定伺服器、擴充叢集、備份資料、監視應用程式、還有得在清晨三點接聽任何緊急電話的可憐人，都可以閱讀本書。

上述的任務常被統稱為**營運**（*operations*）。以往通常都可以找得到擅長撰寫程式碼的開發人員，但他們對營運所知有限；同理，熟稔營運的系統管理員也不難找，但這類人員通常不會動手寫程式。過去這種界線涇渭分明，但在如今的世界裡，當雲端及 DevOps 運動越發蓬勃，幾乎所有的開發人員都必須學習營運的相關技術、而系統管理員也必須懂得撰寫程式碼。

本書並不會逕自認定讀者已經是老練的程式設計師或系統管理員——只需對寫程式有基本的了解、會使用命令列操作、也有可用的伺服器軟體（例如網站）就夠了。其他所需的內容都可以邊做邊學，因此一旦讀完本書，就能充分地掌握當今開發與營運領域中最基本的一塊：管理基礎架構即程式碼。

事實上，讀者們會學到的並不僅限於如何用 Terraform 管理基礎架構即程式碼，還包括如何將它運用在當今整個 DevOps 世界當中。以下就是若干你在本書結尾時能夠得到解答的問題：

- 究竟為何需要使用 IaC？
- 組態管理（configuration management）、調度（orchestration）、配置（provisioning）和伺服器範本的差異究竟何在？
- 何時應該使用 Terraform、Chef、Ansible、Puppet、Pulumi、CloudFormation、Docker、Packer 或 Kubernetes？
- Terraform 如何運作、還有如何用它管理自己的基礎架構？
- 如何建置可以一再使用的 Terraform 模組？
- 如何在操作 Terraform 時安全地管理密語（secrets）？
- 如何在多個地域、以多個帳號、在各種雲端使用 Terraform？
- 如何寫出能用在正式環境當中的可靠 Terraform 程式碼？
- 如何測試你的 Terraform 程式碼？
- 如何將 Terraform 作為自動化部署程序的一部分？
- 讓團隊使用 Terraform 的最佳實施方式為何？

你所需的工具就只有一部電腦（Terraform 幾乎可以在任何作業系統上運作）、網際網路連線，和亟欲學習的意志。

我為何要寫這本書

Terraform 是很厲害的工具。它可以搭配所有廣受歡迎的雲端供應商。其操作語言既清爽又簡潔，在重複使用、測試和版本控管等方面又有絕佳的支援。它採用開放原始碼，其社群十分友善而活躍。但它只有一項缺陷：未竟成熟境界。

Terraform 大受歡迎，但它仍然算是相當新穎的技術，而且即使受到歡迎，但要協助你精通此一工具，卻又很難找到合適的參考書、部落格貼文、或是相關的專家。官方的 Terraform 文件在介紹基礎語法和功能方面做得相當出色，但對於慣用的樣式、最佳實施方式、測試、重複使用性和團隊工作流程方面卻著墨甚微。這就像是只靠背單字學法語、卻缺乏文法或口語練習一樣，是很難達到流利程度的。

筆者撰寫本書的緣由，就是要幫開發人員流利地運用 Terraform。我自己用了六年的 Terraform，這段期間多半是在我工作的企業 Gruntwork 裡（*https://gruntwork.io*），而 Terraform 問世也不過七年而已，在 Gruntwork，Terraform 是我們用來建構程式庫的核心工具之一，其中含有 30 萬行以上可以一再使用、而且飽經實戰的基礎架構程式碼，數百間公司都在正式環境中使用它。要在這些年間撰寫及維護這樣龐大的基礎架構程式碼，還要放在這麼多不同的公司和案例當中使用，讓我們從中學到不少教訓。我的目標就是要和讀者分享這些教訓，這樣你們就可以在短期內上手，不必走我們走過的冤枉路。

當然了，光是坐著捧讀本書是不可能熟稔這套工具的。想要法語流利，你得花時間和以法語為母語的人士交談、多看法語電視節目、還有聽法語歌。同理，要熟練掌握 Terraform，你就得動手撰寫實際的 Terraform 程式碼、用它管理真正的軟體、還要將軟體部署到實際的伺服器上。因此你要做好閱讀、撰寫及執行大量程式碼的心理準備。

本書的內容

以下是本書大綱：

第 1 章，「為何是 Terraform」

DevOps 如何改變了我們運行軟體的方式；基礎架構即程式碼工具的概覽，包括組態管理、伺服器範本、調度、以及配置工具；基礎架構即程式碼的優點；Terraform、Chef、Puppet、Ansible、Pulumi、OpenStack Heat 和 CloudFormation 的比較；如何組合像是 Terraform、Packer、Docker、Ansible 和 Kubernetes 等工具。

第 *2* 章，「開始使用 *Terraform*」

安裝 Terraform；Terraform 語法的概覽；Terraform 的 CLI 工具概覽；如何部署單獨一套伺服器；如何部署一套網頁伺服器；如何部署一組網頁伺服器叢集；如何部署負載平衡器；如何清除你建立的資源。

第 *3* 章，「如何管理 *Terraform* 的狀態」

何謂 Terraform 的狀態；如何儲存狀態供給多位團隊成員取用；如何鎖定狀態檔以防止競逐狀況（race conditions）；如何隔離狀態檔以免錯誤造成破壞；如何使用 Terraform 的工作空間（workspaces）；Terraform 專案的檔案與目錄格局的最佳實施方式；如何使用唯讀狀態。

第 *4* 章，「如何以 *Terraform* 模組建立可以重複使用的基礎架構」

何謂模組；如何建立基本模組；如何透過輸入和輸出將模組設定彈性化；局部值；模組的版本控管；模組的陷阱；以模組定義可重複使用、可以設定的基礎架構零件。

第 *5* 章，「*Terraform* 的奇招異術：迴圈、*If* 敘述、部署和其他竅門」

具有 `count` 參數、`for_each` 和 `for` 表示式、以及 `for` 字串指令的迴圈；具有 `count` 參數、`for_each` 和 `for` 表示式、以及 `if` 字串指令的條件句；內建函式；零停機時間的部署方式；常見的 Terraform 陷阱，包括 `count` 和 `for_each` 的限制、零停機時間部署的陷阱、明明有效的規劃怎會失敗、以及如何安全地重構 Terraform 程式碼。

第 *6* 章，「以 *Terraform* 管理密語」

密語管理介紹；各種類型密語的概覽、不同的密語儲存方式、以及不同的密語存取方式；比較常見的密語管理工具，像是 HashiCorp Vault、AWS Secrets Manager 和 Azure Key Vault；操作 provider 時如何管理密語、包括透過環境變數、IAM 角色和 OIDC 認證；在操作資源和資料來源時如何管理密語、包括如何使用環境變數、加密檔案和集中式的密語儲存；如何安全地處理狀態檔和規劃檔（plan files）。

第 *7* 章，「搭配多種 *Provider*」

仔細觀察 Terraform provider 如何運作，包括如何安裝、如何控制版本、以及如何在程式碼中運用它們；如何使用多份 provider 相同的副本，包括如何部署到多個 AWS 地域、如何部署到多個 AWS 帳號、以及如何建置可以重複使用在多個 provider 上的模組；如何同時使用多個 provider，包括在 AWS 上以 Terraform 運行 Kubernetes 叢集（EKS）、或是將 Docker 容器化的應用程式部署到叢集中的例子。

第 *8* 章，「正式環境等級的 *Terraform* 程式碼」

DevOps 專案為何總是比你預期的更耗時；正式環境等級的基礎架構檢查表；如何建立正式環境的 Terraform 模組；小型模組；組合式模組；可測試的模組；可以發行的模組；Terraform 的登錄所（Registry）；變數驗證； Terraform 的版本控管、Terraform 供應者、Terraform 模組和 Terragrunt；Terraform 的逃生門（escape hatches）。

第 *9* 章，「如何測試 *Terraform* 程式碼」

手動測試 Terraform 程式碼；沙箱環境與清理； Terraform 程式碼的自動化測試；Terratest；單元測試；整合測試；點到點測試；注入依存關係（dependency injection）；平行執行測試；測試階段；重複嘗試；測試的金字塔；靜態分析；規劃測試；伺服器測試。

第 *10* 章，「如何以團隊方式運用 *Terraform*」

如何在團隊中採用 Terraform；如何說服主管；部署應用程式碼的工作流程；部署基礎架構程式碼的工作流程；版本控管；Terraform 的黃金守則；程式碼審閱；程式碼撰寫指南；Terraform 的風格；Terraform 的持續整合與持續部署（CI/CD）；部署過程。

請隨意閱讀本書，不論是從頭讀到尾、或是挑你有興趣的章節跳著讀都無妨。注意每一章的範例都會參照先前章節的範例建置，因此如果你是跳著看，請利用開放原始碼的範例（如同第 xxii 頁的「開放原始碼的程式範例」一節所述）來補足你沒有自行練習建置的部分。本書結尾的附錄中有一個清單，其中列出建議的參考讀物，你可以從中學到更多關於 Terraform、營運、IaC 和 DevOps 的一切。

從再版到三版的變化

本書初版始於 2017 年，兩年後的 2019 年又推出了再版，而連我自己都難以置信的是，2022 年要改三版了。當真是時光飛逝。這幾年來的世事變化之大，值得記錄一番！

如果你曾讀過本書再版，想要知道這一版有何新進內容，抑或是你純粹只是好奇 Terraform 在 2019 年到 2022 年這幾年間有何重大變化，以下便是再版到三版之間的變化重點：

第一章

為何是 Terraform

所謂軟體，並不是只要讓程式碼可以在你的電腦上執行、就算完成了。即使已經通過測試，也不算完成。甚至是完成程式碼審閱（code review）後的某人開口說「出貨」也不算完成。只有直到你將軟體**交付**（*deliver*）給使用者的那一刻，才說得上是完成。

交付軟體（*software delivery*）這件事包羅萬象，它涵蓋了你將程式碼交到客戶手上之前、一切得完成的工作，像是在正式環境伺服器（production servers）上運作程式碼、讓程式碼經得起中斷和流量巔峰的考驗、還要保護它免受攻擊者覬覦。在你一頭栽進 Terraform 的天地之前，也許該先放慢腳步，看看 Terraform 是如何在交付軟體這張廣大藍圖中佔有一席之地的。

在本章當中，你會讀到以下題材：

- 何謂 DevOps？
- 何謂基礎架構即程式碼？
- 基礎架構即程式碼有何長處？
- Terraform 如何運作？
- Terraform 與其他 IaC 工具相較如何？

何謂 DevOps？

才不過幾年之前，如果你想打造一家軟體業者，就免不了要管理大量的硬體。你得設置許多機櫃、裝滿伺服器、佈一堆網路線把它們串在一起、安裝空調、建置容錯電力系統等等。然後自然還要有一個由開發人員（Developers，俗稱「Devs」）組成的團隊，專門負責撰寫軟體，另外還有一組人馬，也就是營運（Operations，俗稱「Ops」）團隊，專門負責管理上述的硬體。

典型的 Dev 團隊會建立應用程式，然後就「甩鍋」給 Ops 團隊[譯註 1]。接下來便是 Ops 的職責，他們要設法找出如何部署及運行應用程式的方式。大部分的工作都是以手動進行的。其實從某種程度上看，手動進行也是無可厚非的，因為過程中有很多動作涉及實際操作硬體（像是將伺服器上架、牽網路線等等）。但是，甚至連 Ops 負責的軟體工作部分，像是安裝應用程式及其依存關係元件等等，通常也是在伺服器上靠手動執行命令來完成的。

上述方式在一開始時都還算順暢，但隨著公司規模日漸成長，遲早都會開始出現問題。通常問題都是這樣開始的：由於軟體釋出（release）時都是以手動進行的，隨著伺服器數量的增加，釋出的過程也越發耗時，而變得遲緩、痛苦、而且又難以預料。Ops 團隊三不五時就會出錯，因而出現了所謂的**雪花伺服器**（*snowflake servers*），這種伺服器每台的組態都與別的機器略有差異（意即俗稱的**組態漂移**（*configuration drift*）問題。於是，臭蟲的數量因此與日俱增。開發人員的答案總是千篇一律的「它在我的機器上就能跑啊！[譯註 2]」，然後中斷及停機的時間也越發地頻繁。

接下來，Ops 團隊已經厭倦了每次在釋出後的凌晨三點就要被告警訊息叫醒，甘脆就把釋出的步調減緩到每週一次。然後又拉長到一個月一次。最後索性半年才來上一次。然後，在每個半年度釋出的幾週之前，各個團隊都要焦頭爛額地整合所有的專案，然後又引發大量混亂的合併衝突。沒有人能理得清各個釋出分支版本之間的異同並加以收斂整合。然後團隊間便陷入交相指責的惡境。彼此壁壘高築。公司內部的整合幾乎陷於停滯。

譯註 1 如果是寫軟體出身的讀者，覺得對「甩鍋」一詞有不悅的感受，譯者在此致歉；因為譯者就是開發軟體的人最討厭的那一群「只會裝機又裝不好、連我的軟體都跑不動」的 infra-guy 出身的。不過原文「toss it over the wall」也有戲謔之意。

譯註 2 後面沒說的那一句就是「你裝的機器不能跑必定是你的問題」。坊間甚至流傳一個笑話：Ops 團隊把 Dev 團隊的機器拿去裝箱打包出貨，因為「既然只有你的機器能跑，那就只好拿它去交付了」。

但如今，一場無聲但翻天覆地的變革正在進行。許多公司不再自行管理自家的資料中心，而是將其移轉至雲端，改以 Amazon Web Services（AWS）、Microsoft Azure 及 Google Cloud Platform（GCP）等服務取而代之。許多 Ops 團隊也不再大手筆地投資硬體，而是透過 Chef、Puppet、Terraform、Docker 和 Kubernetes 等工具，將所有的時間花在這類軟體上。許多系統管理員（sysadmins）早已不再上架伺服器和插拔網路線，而是改為撰寫軟體。

於是，Dev 和 Ops 其實都把大部分的時間花在軟體上了，兩者的區別也日漸模糊。也許分別維持兩個團隊仍有其必要，像是讓 Dev 團隊負責應用程式的程式碼、而 Ops 團隊負責的則是營運所需的程式碼，但顯而易見的是，Dev 和 Ops 團隊必須更密切地配合工作。這就是所謂 *DevOps 運動*（*DevOps movement*）的濫觴。

DevOps 並非什麼新穎的團隊名稱、也不是職掌頭銜、更不是什麼特殊的新技術。它不過是一系列的程序、想法和技術的結合。每個人心目中對於 DevOps 的認知都略有不同，但對本書而言，筆者傾向於以下的說法：

DevOps 的目標就是要大幅地提升交付軟體的效率。

自此不再需要為了合併而傷神數日，你可以持續不斷地整合程式碼、而且自始至終都保持在隨時可以進入部署的狀態。你也不用再苦等每個月的程式碼部署，而是改為一天數度部署程式碼，甚至可以在每次單獨的提交（commit）後便進入部署階段。你可以打造出富於彈性、能自我修復的系統，並以監控及警示等機制來捕捉那些無法自動解決的問題，而不必再為每次釋出後的經常中斷及停機而苦惱。

經歷過 DevOps 轉型的公司，其成效往往令人驚豔。以 Nordstrom[譯註]為例，它在組織中採行 DevOps 的實施方法後，發現一個月當中可以交付的功能數量增長了一倍、但缺陷卻減少了一半，其**前置時間**（*lead times*，意指從想法成形到程式碼可以正式運作所需的時間）更縮短了 60%，而正式環境中的意外事件數目更減少了 60% 到 90%。另外，惠普的 LaserJet 韌體開發部門在採行 DevOps 實施方法後，其開發人員可以真正花在研製新功能上的時間，從 5% 一下增長至 40%，整體研發成本更是下降了 40%。Etsy 則是以 DevOps 實施方法擺脫了以往那種令人壓力破表、次數少得可憐、又容易引起大量中斷的部署方式，搖身一變成為一天可以部署 25 到 50 次的模式，而且中斷次數還更少[1]。

1 數字引用自 Gene Kim、Jez Humble、Patrick Debois 和 John Willis 等人合著的《*The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*》一書（IT Revolution Press，2016 年出版）。

譯註 美國的一間知名服飾百貨業者。

DevOp 運動的核心價值有四：文化（culture）、自動化（automation）、量測（measurement）和分享（sharing）（有時這四者的首字母會全部縮寫成 CAMS）。本書並非對於 DevOps 的全面概述（對此有興趣的讀者，請參閱附錄中的推薦參考讀物），因此筆者只會著重在四者之一，也就是自動化這個部分。

整體的目標可以說是要把交付軟體的過程盡量地自動化。亦即你不再仰賴在網頁中四處點來點去、或是手動執行 shell 命令的方式來管理你的基礎架構，而是透過程式碼的方式來進行。這便是俗稱的**基礎架構即程式碼**（*infrastructure as code*）的概念。

何謂基礎架構即程式碼（Infrastructure as Code）?

基礎架構即程式碼（簡稱 IaC）背後蘊含的想法是，你要定義、部署、更新及消除自己的基礎架構，就得自行撰寫和執行相關的程式碼。這意味著一項心態上的重大變革，亦即你要把一切營運現象都視為軟體——甚至是代表硬體的那一面（例如建置實體伺服器）也不例外。事實上，DevOps 的關鍵觀點之一，就是幾乎**一切事物**皆可以程式碼的形式來管理，包括伺服器、資料庫、網路、日誌檔案、應用程式組態、文件、自動化測試、部署過程等等，無一例外。

IaC 工具可分為五大類別：

- 老派的命令稿（scripts）
- 組態管理工具
- 伺服器範本編寫工具
- 調度（Orchestration）工具
- 配置（Provisioning）工具

以下我們將逐一檢視這些工具。

老派的命令稿

要將任何動作自動化，最直截了當的做法就是寫一個**老派的命令稿**（*ad hoc script*）。把原本手動進行的任務拆解成個別的步驟，再以你偏好的 scripting 語言（像是 Bash、Ruby、Python 等等）將每一個步驟定義成程式碼的形式，然後在你的伺服器上執行這份命令稿，如圖 1-1 所示。

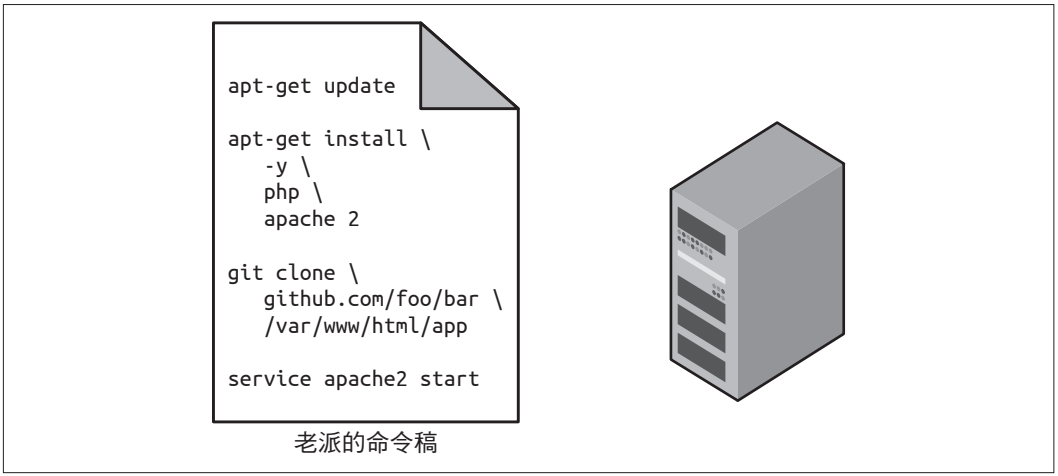

圖 1-1　要把事情自動化，最直覺的方式便是建立一支老派的命令稿，以便在伺服器上執行

舉個例子，這裡有一份名為 *setup-webserver.sh* 的 Bash 命令稿，它會逐步地安裝各種相互依存的部件、並從某個 Git 的程式庫取得某種程式碼、然後啟動一個 Apache 網頁伺服器，最終完成一套網頁伺服器的設定：

```
# Update the apt-get cache
sudo apt-get update

# Install PHP and Apache
sudo apt-get install -y php apache2

# Copy the code from the repository
sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app

# Start Apache
sudo service apache2 start
```

老派命令稿的傑出之處，在於你只需利用廣受愛用的通用型程式語言，就可以寫出你所需的任何程式碼。但是，老派命令稿的糟糕之處，也是你得靠通用型程式語言，自己寫出所有需要用到的程式碼。

然而，特別針對 IaC 用途而建置的工具，卻能以簡潔的 API 來完成複雜的任務，如果你還在使用老派的通用型程式語言，所有的任務都得靠你自行撰寫全部的程式碼才能完成。此外，專為 IaC 設計的工具通常會要求在程式碼中採行特定的結構，但每個開發者在採用通用型程式語言時，卻往往會自行決定程式碼架構風格，因而各行其是。如果像先前那樣只是安裝 Apache 的八行命令稿程式碼，以上兩種差異不會造成什麼大問題，

但是當你要用老派命令稿來管理幾十套伺服器、資料庫、負載平衡器、網路組態等標的物時，事情苗頭便不太對了。

如果你曾經受命管理過大批 Bash 命令稿的儲藏庫，就會知道到頭來總是會衍生出一大堆難以維護的麵條程式碼。對於一次性的小型任務而言，老派命令稿勝任自如，但如果你要以基礎架構即程式碼進行全面管理，最好還是考慮採用專為此項工作打造的 IaC 工具。

組態管理工具

Chef、Puppet 和 Ansible，都是所謂的**組態管理工具**（*configuration management tools*），亦即它們都是設計用來在既有伺服器上安裝及管理軟體的工具。譬如說，這裡有一個名為 *web-server.yml* 的 *Ansible role* 定義，它設定的 Apache 網頁伺服器，跟先前以 *setup-webserver.sh* 命令稿的成果是一模一樣的：

```
- name: Update the apt-get cache
  apt:
    update_cache: yes

- name: Install PHP
  apt:
    name: php

- name: Install Apache
  apt:
    name: apache2

- name: Copy the code from the repository
  git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app

- name: Start Apache
  service: name=apache2 state=started enabled=yes
```

以上程式碼看起來與 Bash 命令稿頗為相似，但利用像是 Ansible 這樣的工具來實施，有幾項優勢：

程式碼有必須遵循的慣例

Ansible 會要求遵循一系列可預測的一致性架構，像是文件、檔案佈局、命名精確的參數、密語管理（secrets management）等等。但是在老派命令稿裡，每個開發者都是自行安排以上內容的，而大部分的組態管理工具則會有自己的一套慣例，讓人更容易瀏覽程式碼。

Idempotence

要寫出可以成功運行一回的老派命令稿並不是什麼難事；但是要寫出可以一再重複地執行、都能運作無誤的老派命令稿，就是另一回事了。每當你在命令稿中建立一個資料夾時，你都必須記得先檢查該資料夾是否已經存在；每當你為檔案加上一行組態設定時，都得再檢查一遍該行設定是否已經存在；每當你要執行某支應用程式（app）時，都得先檢查它是否已在運作當中。

不論執行幾次都能正確運作的程式碼，我們會說它是 *idempotent code*。要讓上一小節的 Bash 命令稿變得 idempotent，你得加上很多行的控制用程式碼，包括大量的 if 敘述。但是對於 Ansible 而言，其大多數的函式天生就是 idempotent 的。舉例來說，上述的 *web-server.yml* 這個 Ansible role，就只會在 Apache 尚未安裝的形況下才會著手安裝，而且也只會在 Apache 網頁伺服器不曾運行的時候才會嘗試啟動它。

分散式運作

老派的命令稿是設計用來在單一的本地端機器上執行的。而 Ansible 與其他組態管理工具卻是專門設計用來管理大量遠端伺服器的，如同圖 1-2 所示。

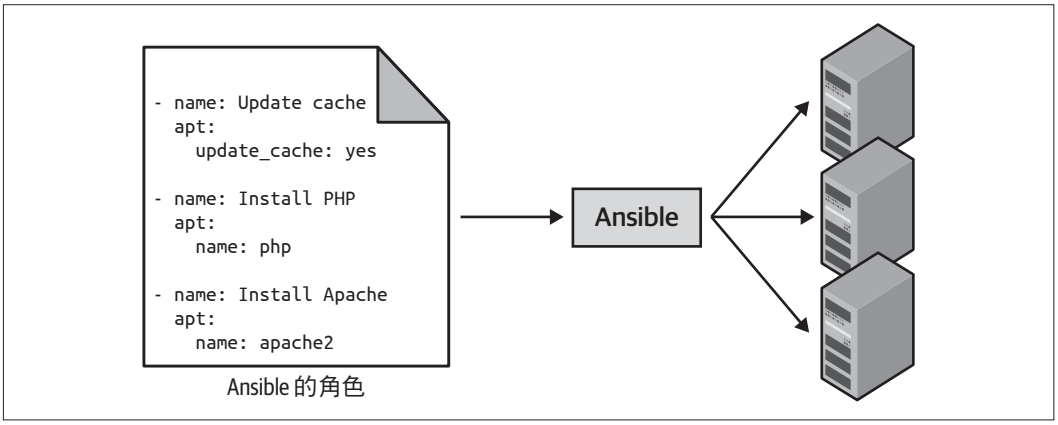

圖 1-2　像 Ansible 這樣的組態管理工具，可以在大量的伺服器上執行你的程式碼

譬如說，要把 *web-server.yml* 這個 role（角色）套用到五套伺服器上，你只需先建立一個名為 *hosts* 的檔案，其中包含這五套伺服器的 IP 位址就好：

```
[webservers]
11.11.11.11
11.11.11.12
11.11.11.13
11.11.11.14
11.11.11.15
```

然後像這樣定義你的 *Ansible playbook*：

```
- hosts: webservers
  roles:
  - webserver
```

最後像這樣執行 playbook：

```
ansible-playbook playbook.yml
```

如此便可指示 Ansible 同時設定所有五套的伺服器了。此外你還可以在 playbook 裡加上一個叫做 `serial` 的參數，以便改採所謂的**滾動式部署**（*rolling deployment*），亦即分批更新伺服器。譬如說，若是把 `serial` 的值訂為 `2`，便會指示 Ansible 一次只更新兩套伺服器，直到所有伺服器都更新完畢為止。但是在老派命令稿中，你至少得多寫好幾打、甚至上百行的控制用程式碼，才能重現同樣的邏輯與效果。

伺服器範本編寫工具

近年來，另一種替代組態管理的方式正日漸興起，並受到矚目與歡迎，它就是**伺服器範本編寫工具**（*server templating tools*），像是 Docker、Packer 及 Vagrant，都屬於這類工具。伺服器範本編寫工具所蘊含的概念，並非啟動一堆伺服器、然後在每一台上都執行相同的設定程式碼，而是建立整台伺服器的**映像檔**（*image*），這個映像檔是一個自給自足的整機系統「快照」（snapshot），其中包含作業系統（operating system，OS）、軟體、檔案、以及所有相關的細節。然後就可以再用其他 IaC 工具，把這個映像檔裝到所有的伺服器上，如圖 1-3 所示。

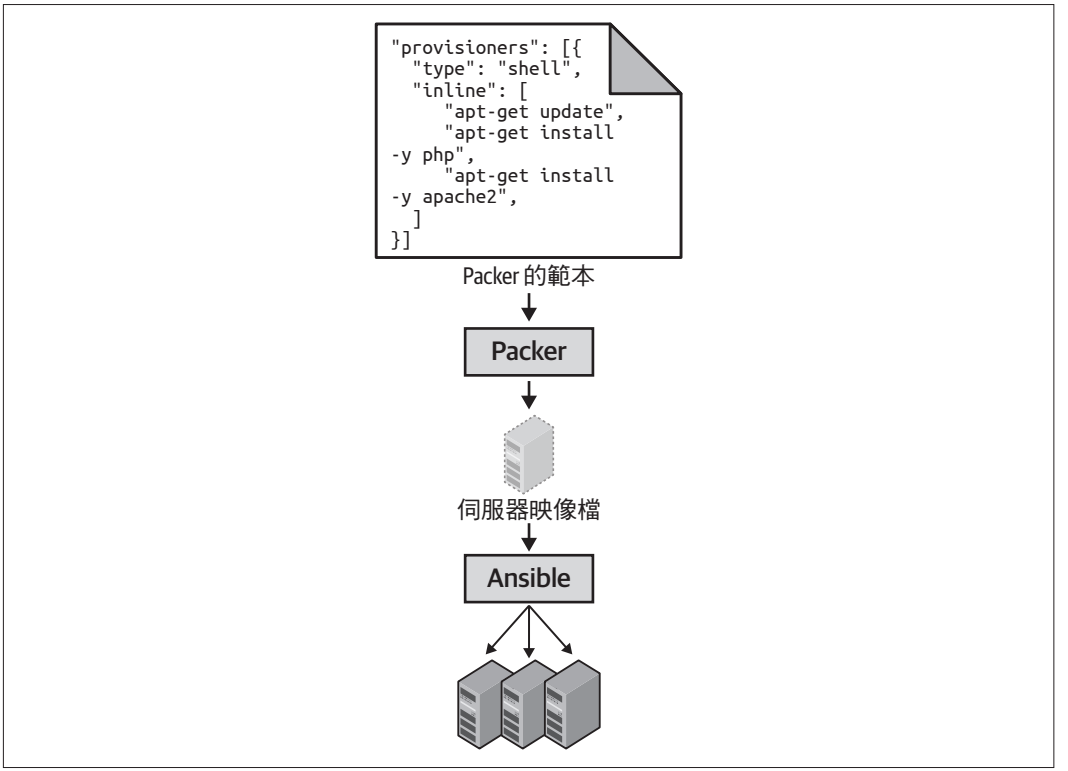

圖 1-3　你可以藉由 Packer 這類的伺服器範本編寫工具，來建置一個自給自足的伺服器映像檔。然後再以其他工具，譬如 Ansible，將該映像檔安裝至所有伺服器

操作映像檔的工具可分成兩大體系（如圖 1-4 所示）：

虛擬機器

所謂的虛擬機器（*virtual machine*，VM），係指模擬整套的電腦系統，包括硬體層。你需要先執行像是 VMware、VirtualBox、或是 Parallels 之類的一套虛擬機監視環境（*hypervisor*），才能在其中將電腦系統底層的 CPU、記憶體、硬碟及網路功能等部件加以虛擬化（virtualize，亦即上述的模擬動作）[譯註]。

[譯註] 依照維基百科的解釋，hypervisor 還分成原生或裸機、以及寄居或代管兩種形式；前者係直接執行在硬體上，例如 VMware，後者則是安裝在既有的作業系統中，例如 VirtualBox。

這種形式的優點在於，你在 hypervisor 上運作的任何 *VM 映像檔*，都只能接觸到經過虛擬化的硬體，因此它是完全與底層的宿主機器及其他 VM 映像檔隔離開來的，而且在所有環境中的運作方式都會一致（不管是在你的電腦上、在 QA 伺服器上、還是在正式環境伺服器上）。缺點則是，因為所有的硬體都必須虛擬化，而且每一套 VM 都必須運行自己獨立的作業系統，這會對 CPU 及記憶體的用量造成額外負載、啟動的時間也較長。你也可以用 Packer 和 Vagrant 之類的工具，以程式碼的形式定義 VM 映像檔。

容器

所謂的**容器**（*container*），係指模擬了作業系統的使用者空間（user space）[2]。你必須先執行像是 Docker、CoreOS rkt、或是 cri-o 之類的**容器引擎**（*container engine*），才能建置具有隔離效果的程序、記憶體、掛載點以及網路功能。

這種形式的優點在於，你在容器引擎上運行的任何容器，都只能接觸到它自己的使用者空間，因此它與底層的宿主機器及其他容器都是隔離開來的，而且在所有環境中的運作方式也都會一致（不管是在你的電腦上、在 QA 伺服器上、還是在正式環境伺服器上）。缺點則是，因為所有運行在單一伺服器上的容器，都是共用該伺服器的作業系統核心和硬體的，因此它難以達到 VM 所能提供的隔離程度及安全性[3]。然而，正由於核心及硬體都是共用的，你的容器可以在幾毫秒內便啟動，而且對 CPU 或記憶體幾乎不會造成額外負載。你也可以用 Docker 和 CoreOS rkt 之類的工具，以程式碼的形式定義容器映像檔；讀者們會在第 7 章時讀到如何運用 Docker 的例子。

2 在大多數近代的作業系統上，程式碼會在兩種「空間」之一當中運作：亦即**核心空間**或**使用者空間**。運作在核心空間當中的程式碼可以毫無限制地直接存取所有的硬體。這裡不會有安全上的限制（像是可以執行任何 CPU 指令、存取硬碟的任意部位、寫入記憶體的任一位址）、也不會有防護上的限制（譬如核心空間的故障崩毀（crash），絕對會導致整部電腦故障），因此核心空間通常都只保留給作業系統中最低階、最可信的功能（通常就是**核心**本身），才能在此運作。至於運作在使用者空間的程式碼，則是無法直接存取硬體，而只能利用作業系統核心提供的 API 來動作。這類 API 都會施加安全上的限制（例如使用者權限）和防護（例如使用者空間的 app 故障崩毀，就只會影響該 app 而已），因此幾乎所有應用程式的程式碼，都只能在使用者空間中運作。

3 一般來說，對於你自己的程式碼，容器提供的隔離程度已經足夠，但如果你需要執行第三方的程式碼（譬如你建置的是自己的雲端供應服務），而且對方有可能主動進行惡意動作，你可能就得提升隔離保障的程度，也就是改用 VM。

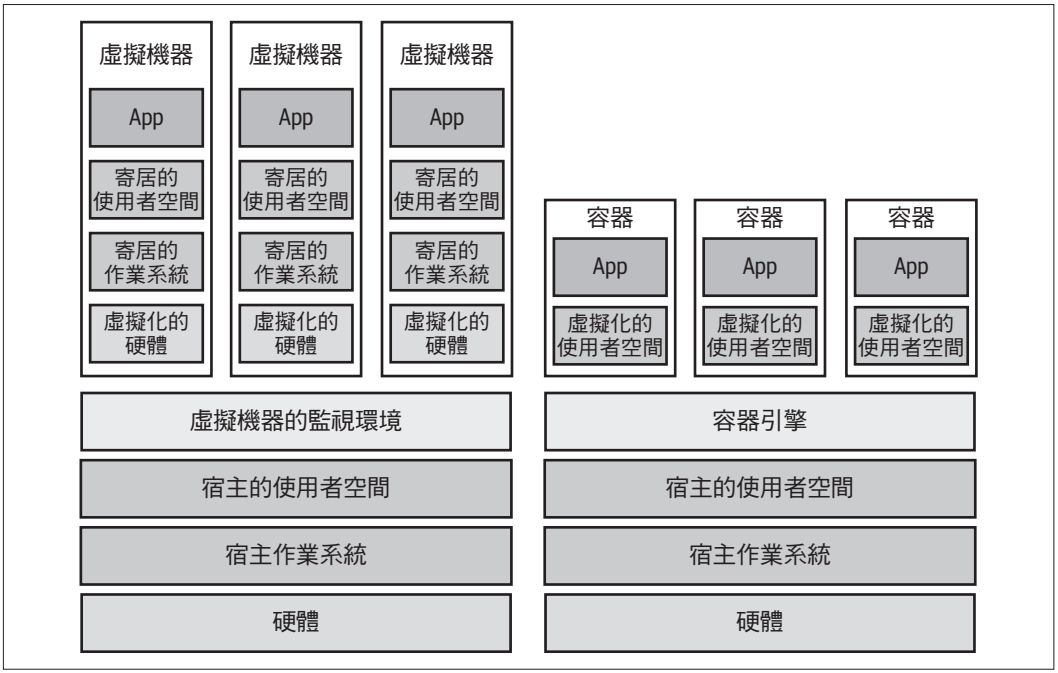

圖 1-4　映像檔的兩大類型：左邊是 VM、右邊是容器。VM 係將硬體虛擬化、而容器則是只把使用者空間虛擬化

這裡有一個 Packer 範本的例子，範本檔名是 *web-server.json*，它會產生出一個 *Amazon Machine Image*（AMI），這是一種亞馬遜 AWS 專用的虛擬機映像檔：

```
{
  "builders": [{
    "ami_name": "packer-example-",
    "instance_type": "t2.micro",
    "region": "us-east-2",
    "type": "amazon-ebs",
    "source_ami": "ami-0fb653ca2d3203ac1",
    "ssh_username": "ubuntu"
  }],
  "provisioners": [{
    "type": "shell",
    "inline": [
      "sudo apt-get update",
      "sudo apt-get install -y php apache2",
      "sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app"
    ],
    "environment_vars": [
```

```
        "DEBIAN_FRONTEND=noninteractive"
      ],
      "pause_before": "60s"
    }]
}
```

以上的 Packer 範本所設定的 Apache 網頁伺服器，與先前 *setup-webserver.sh* 用 Bash 程式碼所設定的完全一樣。兩者之間唯一的差異，就是 Packer 的範本沒有嘗試啟動 Apache 網頁伺服器（亦即呼叫 `sudo service apache2 start` 這樣的動作）。這是因為伺服器範本通常係用於在映像檔中置入軟體，因此只有當你實際運行該映像檔時——譬如將它部署到伺服器上——才會真正地運行事先安裝的軟體。

要從這份範本打造出一個 AMI，你得執行 `packer build webserver.json`。建置完成後，就可以把這個 AMI 裝到你所有的 AWS 伺服器上，並設定每部伺服器、令其開機後便執行 Apache（下一小節便會看到實際的例子），然後它們便會以完全相同的方式運作。

另外要注意的是，不同的伺服器範本編寫工具，其目的也會略有差別。像 Packer 通常都是用來建置映像檔，以便直接放到正式環境的伺服器上去運行的，就像你用正式環境的 AWS 帳號去運行一個 API 那樣。而 Vagrant 通常則是用來建置會運作在開發用電腦上的映像檔，就像你用 Mac 或 Windows 筆電運行的 VirtualBox 映像檔那樣。Docker 則通常是用來建置個別應用程式的專屬映像檔。不論正式環境還是開發用的電腦，只要該電腦上已經設有 Docker 引擎，你都可以拿來運行 Docker 映像檔。舉例來說，常見的做法會像這樣：你會用 Packer 建置一個內部裝有 Docker 引擎的 AMI，再把這個 AMI 部署到你的 AWS 帳號中的伺服器叢集，然後就只需將個別的 Docker 容器部署到叢集中，便可運作你的應用程式了。

伺服器範本編寫這個動作，可以看成是轉移至**不可變基礎架構**（*immutable infrastructure*）的關鍵元件。這個概念是啟發自函式化程式設計（functional programming），亦即變數是不輕易變動的（immutable），因此當你為變數賦值之後，就不能再變更該變數。就算你想更動某些東西，也得另外定義新變數。由於變數永遠不變，要推斷程式碼內容便容易得多。

不可變基礎架構背後蘊含的觀念也很類似：一旦你部署了伺服器，就不能再加以更改。如果你想更動些什麼，譬如要部署新版的程式碼，乾脆就從伺服器範本重新再建置一份新的映像檔，然後將其部署到新伺服器上。由於每一版伺服器都是不可變的，要推斷部署的內容自然簡單得多。

調度工具

伺服器範本編寫工具十分適合用來建置 VM 與容器，但實際上你要如何管理它們？在實際的運用範例中，你需要有辦法達成以下任務：

- 部署 VM 與容器，並有效地運用硬體。
- 更新現有的 VM 及容器群體時，以滾動式更新、兩階段式部署（又稱為藍 / 綠部署，blue-green deployment）、漸進式部署（又稱為金絲雀部署，canary deployment）等策略來進行更新。
- 監控 VM 與容器的健康狀態，並自動迭代異常的部位（亦即**自我療癒**，*auto healing*）。
- 視負載擴大或收束 VM 與容器的數量（scale up 或 scale down）（又稱為**自動規模調節**，*auto scaling*）。
- 將流量分散至各個 VM 和容器（**負載平衡**，*load balancing*）。
- 讓 VM 或容器都能透過網路找到對方並進行通訊（**服務探索**，*service discovery*）。

這些任務的處理，都算是**調度工具**（*orchestration tools*）的領域，例如 Kubernetes、Marathon/Mesos、Amazon Elastic Container Service（Amazon ECS）、Docker Swarm 和 Nomad 等等皆在此列。以 Kubernetes 為例，它允許你以程式碼的方式來定義如何管理 Docker 容器。首先你會部署一組 *Kubernetes* **叢集**（*Kubernetes cluster*），其實就是一群伺服器，Kubernetes 透過它們來管理和應用 Docker 容器的運作。大多數主流的雲端服務商都支援原生的受管 Kubernetes 叢集部署，像是 Amazon Elastic Kubernetes Service（EKS）、Google Kubernetes Engine（GKE）、還有 Azure Kubernetes Service（AKS）等等。

一旦叢集可以運作，你就可以透過 YAML 檔案、以程式碼來定義你的 Docker 容器要如何運作：

```
apiVersion: apps/v1

# Use a Deployment to deploy multiple replicas of your Docker
# container(s) and to declaratively roll out updates to them
kind: Deployment

# Metadata about this Deployment, including its name
metadata:
  name: example-app

# The specification that configures this Deployment
spec:
  # This tells the Deployment how to find your container(s)
```

```
  selector:
    matchLabels:
      app: example-app

  # This tells the Deployment to run three replicas of your
  # Docker container(s)
  replicas: 3

  # Specifies how to update the Deployment. Here, we
  # configure a rolling update.
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate

  # This is the template for what container(s) to deploy
  template:

    # The metadata for these container(s), including labels
    metadata:
      labels:
        app: example-app

    # The specification for your container(s)
    spec:
      containers:

        # Run Apache listening on port 80
        - name: example-app
          image: httpd:2.4.39
          ports:
            - containerPort: 80
```

以上檔案指示 Kubernetes，如何建立一份部署（*Deployment*），以宣告式的方式（declarative way）定義出以下的內容：

- 同時運作一個或多個 Docker 容器。這一群容器被統稱為一個 *Pod*。以上程式碼所定義的 Pod 裡只包含了一個 Docker 容器，其中執行的則是 Apache。
- Pod 中每一個 Docker 容器的設定。以上程式碼中的 Pod 會設定 Apache 要傾聽 80 號通訊埠。

- 你的叢集中要運作幾份 Pod 的複本（亦即所謂的**抄本**（*replicas*））。以上的程式碼設定了三份抄本。Kubernetes 會透過排程運算法（scheduling algorithm），根據高可用性（high availability，例如試圖讓每個 Pod 運行在不同的個別伺服器上，以免單一伺服器崩毀影響你的 app）、資源（例如選出含有容器所需通訊埠、CPU、記憶體及其他容器所需資源的伺服器）、性能（例如試圖挑出負載最輕、其中所含容器最少的伺服器）等因素，挑出最合適的伺服器，並自動判斷要在叢集中的何處部署每一個 Pod。Kubernetes 還會不斷地監控叢集，以確保始終都有三組複本正在執行，同時在任何 Pod 崩毀或是停止回應時自動進行替換。
- 如何部署更新。在部署新版的 Docker 容器時，以上的程式碼會一口氣推出三份複本，等到它們趨於穩定（healthy）時，便將另外三份舊複本移除（undeploy）。

短短幾行 YAML，威力竟強大如斯！你只需執行 `kubectl apply -f example-app.yml` 便可指示 Kubernetes 部署你的 app。只需修改 YAML 檔案、再度執行 `kubectl apply`，就可以推出更新的內容。你也可以用 Terraform 來管理 Kubernetes 叢集和其中的 apps；第 7 章就會有範例加以說明。

配置工具

雖說組態管理、伺服器範本編寫、以及調度工具都可以用來定義每部伺服器上執行的程式碼，但是像 Terraform、Cloud-Formation、OpenStack Heat 及 Pulumi 這樣的**配置工具**（*provisioning tools*）則是可以用來建置伺服器本身的。事實上，你不只可以靠配置工具來建置伺服器，還可以建置資料庫、快取服務、負載平衡器、佇列服務、監控服務、子網路配置、防火牆設定、路由選徑規則、Secure Sockets Layer（SSL）的憑證、以及幾乎所有的基礎架構內容，如圖 1-5 所示。

譬如說，以下的程式碼便會以 Terraform 部署一套網頁伺服器：

```
resource "aws_instance" "app" {
  instance_type     = "t2.micro"
  availability_zone = "us-east-2a"
  ami               = "ami-0fb653ca2d3203ac1"

  user_data = <<-EOF
              #!/bin/bash
              sudo service apache2 start
              EOF
}
```

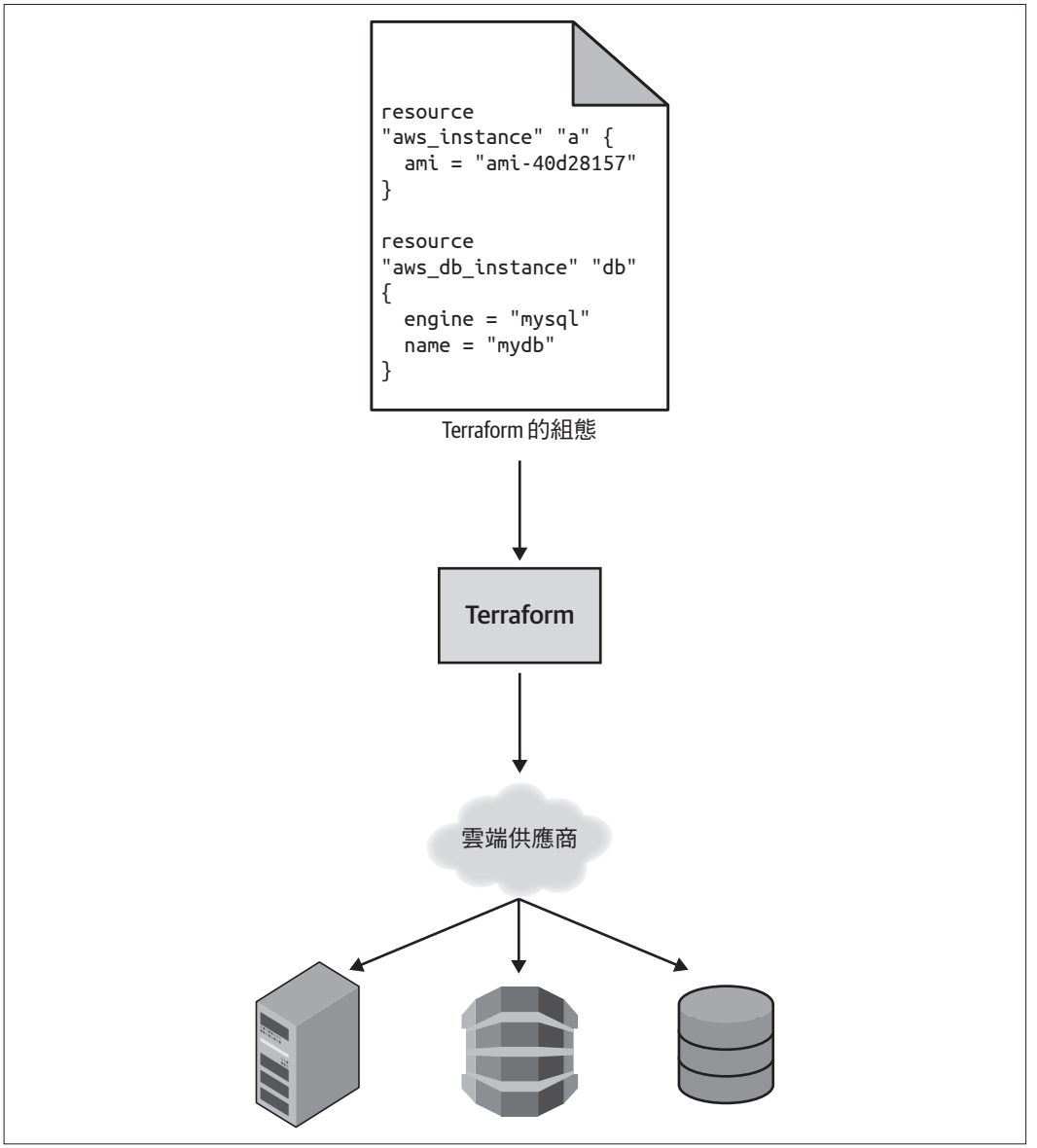

圖 1-5　你可以用配置工具來搭配雲端供應商，藉以建置伺服器、資料庫、負載平衡器及其他基礎架構所包含的部件