**淺談「基礎架構即程式碼」**

**何謂基礎架構即程式碼**

基礎架構即程式碼 (英語：Infrastructure as Code，縮寫：IaC) 是一種通過軟體語法就可以定義出雲端服務所需架構的方法，而這可定義的架構可包括了：軟體、伺服器、儲存、加速器、網路等等。由於基礎架構能以程式碼的形式來定義，所以管理者就可以很快地自動把雲端服務部署起來；管理者也可以把這些程式碼存放到軟體版本控制系統如 Git、Gitlab 之內，以確保可複製性與版本控制的好處。

Infrastructure as Code 之所以盛行，主要是因為「持續整合」、「持續交付」、「持續部署」的需要 (以下簡稱為CI/CD)。在沒有實踐 CI/CD 之前，測試人員想要部署用來測試的軟體、或者MIS人員想要部署用於營運的應用程式，通常需要大量人工執行、或者有人在場待命，以免部署失敗。如果部署能夠自動化，開發人員就能夠透過申請的方式，來取得根據程式碼而配置出來的虛擬機，那麼部署作業就會更加順暢也更加迅速，省下了大量成本與時間。

「可以用程式碼來呈現」，這才是真正理想的軟體定義 (Software Defined)：

從作業系統到應用程式，都可以用程式來自動安裝；

多層的網路架構，可以用程式來自動配置；

隨需的儲存空間，可以用程式來自動開通；

當所有的IT基礎架構都可以用程式碼來呈現，就實現了理想中的「軟體定義資料中心 (Software Defined Data Center)」！

下圖就說明Infrastructure as Code在 DevOps 開發流程中所扮演的角色。

一張含有 文字, 螢幕擷取畫面, 字型, 圖表 的圖片

自動產生的描述

**基礎架構即程式碼的由來**

Infrastructure as Code 基礎架構即程式碼的由來比較特別，因為它沒有一個明顯的發明人或研發機構。比較有跡可循的是知名軟體架構師 Martin Flower 與 Kief Morris 在 2006 年所撰文的 《IT 運算架構鐵器時代》(見 Reference #2 & #3)：

在90年代，運算架構與機房硬體採購是緊密相連的。採購一台新的伺服器，通常都需要幾個月的時間，因此對於部署軟體的時程，也就沒有太大的壓力。當伺服器到貨，IT人員通常會用上幾天的時間，才把所需的軟體裝好、測試完成，交給研發人員使用。

但是到了2000年代初期，隨著運算虛擬化與VM的成熟，大家都開始使用VM 映像檔把作業系統及軟體自動安裝起來，因此部署一個雲端服務，可能只需要幾分鐘的時間。可是，這還是沒有解決整體架構、網路規劃、共用儲存等的問題，因此研發人員與IT之間，常常還是需要數日，甚至數星期去把架檬給規劃與部署出來。

基礎架構即程式碼 (Infrastructure as Code)就是在這種壓力下發展出來的。許多開源軟體如 CFEngine、Puppet、Chef 等出現，愈來愈多領先的科技公司組織開始採納敏捷式開發，開始發展自動化以部署IT服務所需的環境架構，開發人員開始使用程式碼來編譯及註明他們所需要的環境，IT運營人員可以直接部署。

**Kubernetes 容器叢集與基礎架構即程式碼**

順帶一提，目前流行的雲原生容器 ＋Kubernetes 叢集架構，和基礎架構即程式碼是相當匹配的。Kubernetes 容器的部署規格，就是以 YAML 來註明。YAML 就像是架構的程式碼，它不但可以版控，又沒有跟任何具體的硬體規格有所關連，因此，我們撰寫了在個人開發環境上部署 Kubernetes 容器的 YAML，也可以用來部署容器服務到不同的機房硬體上。這過程中不需要再經過IT處理，就可以輕鬆地把一個雲端服務搬到測試環境、開發環境、或者不同地區的生產環境上。

在營運考量面，我們也可以利用 Kubernetes 的 YAML 規格，註明 Replication Controller，若一組服務容器中有一個容器損毀了，Kubernetes 就會自動根據 YAML 的規格，把一個新的容器開起來，以實現服務的高可用性。

同樣的觀念，也可以應用到「升級不中斷」(Non-Disruptive Upgrade, NDU) 的情境。我們可以用同樣的 YAML 規格，開啟一個升版的容器來做測試，若測試順利通過，就可以把服務轉到新版容器上，Replication Controller 就會自動把其他容器部署起來，可保持穩定的服務水準。

**為何企業會接受基礎架構即程式碼**

有了基礎架構即程式碼，自動部署服務，能讓企業能更有效率地控制環境的變更與組態設定，讓開發人員與營運人員更緊密地合作建置虛擬機器、虛擬網路、虛擬儲存，以及虛擬機器內的應用程式與服務。一般來說，以基礎架構即程式碼來自動化部署服務，將有下列好處：

集中管理、部署、以及設定整體環境。自動化的部署步驟都是標準化的，因此可以確保環境與流程，並可以預期後續的結果。

增加系統間的一致性。

用程式碼來定義系統配置的標準，讓人們可以不必手動執行重複且單調的工作，當然也可以減少人為錯誤。

採用基礎架構即程式碼，雖然在前期需要一些時間編譯，但是因為自動化部署比手工的速度更快，也可以不斷地重複部署，而每次執行也無須工程人員在旁待命，從中長期來看，自動化是可以大幅降低成本的。

架構程式碼可以整合到持續整合/持續交付的開發流程中，IT/開發人員能大量利用成熟的第三方或開源軟體，促進各部門協同合作。

IT 可以將編譯好的架構程式碼，在不同的硬體環境試行，協助測試。

當營運環境需要調整時，IT人員可以快速修改架構程式碼，增加開發效率。

**實踐「架構自動化」的步驟**

如上所述，以架構程式碼自動部署服務，既可以增加效率，又可減少人工，避免不必要的錯誤，那麼實踐架構的自動化部署，又會包含那些步驟呢？

一般來說，實踐架構的自動化，可分為以下幾個步驟：

1. 服務執行環境的規格化

服務執行環境，簡單分為硬體與軟體。硬體部份包括了伺服器、加速器如GPU、儲存系統、網路等等。由於虛擬資料中心的概念已經行之有年，因此 IT系統運維人員通常都會把現有硬體切成不同的資源池。把基礎架構視為程式碼的好處，就是做為一個資源的需求者，我不必知道實際硬體的所在地，也不必知道實體網路的介接，我只需要把服務所需規格訂出來，IT人員就可以依照虛擬規格，從不同的資源池上，將資源配發給我這個需求者。

架構即程式碼的另一個優點，就是環境的一致性。一般的軟體開發流程，通常都會隔離出開發環境、整合測試環境、與正式上線環境。雖然這三座環境是分隔的，但是在規格上，則是愈相像愈好。架構程式碼就可以確保這件事：以程式碼分別多次部署環境，仍然得到不變的規格。

2. 服務虛擬化的映像

服務執行環境的軟體部份，包括了系統的作業系統和應用程式。下一步，就是把系統的作業系統，以及跟應用系統的容器，製作其服務虛擬化的映像檔。幾乎所有作業系統的映像檔都可以從互聯網下載，而容器則是更為接近架構即為程式碼的概念，開發人員可以用 dockerfile，彈性定義如何製作 docker Image。Docker 基本上就是 docker image 的腳本，它是一個文字設定檔 ，使用者透過 Dockerfile 快速建立自訂的映象檔。

3. 服務應用程式的參數配置

架構自動化的過程，比較費時間的，是如何去設定各個元件模組的參數，以及模組與模組之間的介接。這些系統架構中的重要模組，例如 SQL、 MongoDB、MQ、Celery、ELK、Prometheus 等等，都有很多個參數要設定，像是輸入/輸出的目錄、權限設定、資源空間、網路埠等等，這些都需要開發人員不斷地試錯來能得到結果。有幸的是，在架構即程式碼的概念下，使用者也可以使用如 Ansible 跟 Puppet 這樣的工具，將參數程式化，穩定地配置參數以確保模組正確運行。

4. 服務運行時的資源規劃

另一項可以用程式碼定義的，是服務運行時的資源配給的策略。尤其是在現代 Kubernetes 或 Openstack 的雲服務叢集架構下，都可以彈性定義各個微服務的資源配給，例如 Replication factor，Load Balancer、Quota等等。

5. 上線前測試

上述各項工作做好後，在自動部署之前，必須通過一定的功能測試與迴歸測試 (Regression test)，服務才可以 上線。這也是可以利用架構程式碼執行的自動化程序。測試人員通常都會撰寫好一些測試腳本以及測試框架，交給運維人員做上線前的測試。

應用服務部署的程序，也有不同的實踐方式，例如藍綠部署、紅黑部署、AB測試、灰金絲雀釋出、滾動釋出等等。有興趣的讀者可參考 Reference #10。

**基礎架構即程式碼對雲端服務的意義**

基礎架構即程式碼無疑是雲端虛擬化與DevOps/敏捷開發流程的結晶，但是基礎架構即程式碼的好處絕對不只是自動化、快速部署、版本控制而已。這是因為，基礎架構即程式碼本身就是 一種系統化與模組化的概念，它可以讓架構師很清楚地把雲端服務架構以「抽象層」(abstraction)的形式做表達， 以便收集團隊意見，提早移除潛在的問題，並加速計劃的進度。另一方面，經過這二十年的發展(參考前段 《基礎架構即程式碼的由來》的演進)，基礎架構即程式碼也衍生出自己一套編譯模式及理論，可應用到不同的架構及開發模式上。

**「架構程式碼」的編譯模式**

接下來，我們就來討論幾個 「架構程式碼」的編譯模式：

1. 虛擬機房部署 (Hardware Environment Provisioning) 與服務參數配置管理(Service Configuration Management)
2. 指令式(Imperative)與宣示式(Declarative)程式碼編譯法
3. 直接式與間接式的自動化(Direct vs. Indirect automation)
4. 可變與不可變的基礎架構

**虛擬機房部署與服務參數配置管理 (Service Configuration Management)**

我在標題「機房部署」之前加了「虛擬」兩個字，就是因為這個「虛擬」是 雲端世代才開始普及的概念。傳統的機房，都是用 Ghost Image 或 ISO 檔來部署。用 Ghost image 雖然也算自動化，但事前 IT 必需把硬體的拓樸架構先接好，才來裝 OS、Storage server、和 Network Management Software。這個過程還是常常出錯，曠日費時。虛擬機房的概念，就是把儲存跟網路也虛擬化，整個拓樸架構其實都是虛擬軟體。只要把Openstack 或 Kubernetes 這類的雲端叢集裝好，那麼其餘的都是用映像檔 (image) 安裝出來的。

相反的，服務參數的配置向來都是由軟體負責的。 服務參數的意思是，應用軟體裝好後，運行時還是需要一些環境參數做服務的最佳呈現。舉個例子：磁碟分區 patition 的大小、佇列的深度、第三方軟體的介接等等，都算是服務參數。

傳統 的 Puppet 與 saltstack，算是 Configuration Management 工具，比較常用於服務參數配置。另一方面，比較新的軟體，如 Terraform 跟 Heat，則是 常用於虛擬機房部署。Ansible 最初也是專長於服務參數配置，但這幾年 也加了不少虛擬機房配置的功能。

我們日後會另以專文，深入介紹這些軟件。

**指令式 (Imperative) 與宣示式 (Declarative) 程式碼編譯法**

「指令式」 vs. 「宣示式」，最簡單 的了解方法，就是「如何去做」 (How) vs. 「要達成什麼目標」 (what)。顧名思義，指令式是要一步一步地告訴部署程式，要怎樣去把服務給裝起來；宣示式則是只要告訴部署程式，你最後需要一個什麼元件，部署程式內部會發送API給下層架構系統，去把元件部署起來。

舉例來說，要裝一個 Database (e.g. SQL、MySQL)，如果用的是「指令式」，就是在一個VM內一步一步把MySQL裝起來，並且把MySQL的參數都配置好。如果用的是「宣示式」，就只需告訴 Terraform 機房部署軟件，你需要一個 MySQL，Terraform 就會發送 API 給 IaaS vendor ，去把 MySQL 帶起來。

如下圖所示：

一張含有 文字, 圖表, 設計, 螢幕擷取畫面 的圖片

自動產生的描述

使用指令式自動化對基礎架構做變更，您要透過指令行介面 (CLI) 來修改雲端設定，首先要從容器內部開始變更，接下來是虛擬伺服器 (VM)，然後是虛擬私有雲，將所有的變更指令全部列在一個腳本當中。這份清單會非常詳盡，但如果在變更發布之後又要對很多台機器做組態設定調整，那就必須整個腳本再重新執行一次。

若採用宣示式自動化方法，則是要先建立目標，例如，您不必使用 CLI 來列出執行 VM 組態設定變更的每一道步驟，您只需宣告您要一台配置在網域下的 VM 即可，剩下的就交由自動化工具幫您搞定。宣示式方法可讓您更容易指定您希望自動化工具幫您完成的需求。

直接與間接的自動化模式(Direct vs. Indirect automation)

「直接式」vs.「間接式」的自動化，跟「指令式」vs. 「宣示式」的概念有點相近，只是專案執行的型態有所分別。

間接式的自動化，是指服務中各應用的配置參數，是經由開發人員在開發過程中一起編譯出來。IT人員只是把所有開發人員的參數配置程式，包在服務的安裝程式包內，對IT人員來說，這些參數的作用是不透明的。間接式的好處，是專案始創期比較簡單。實際上大部份比較傳統的服務也是這樣做的。

但隨著服務的持續維運，軟體的升級，間接式的安裝程序跟參數都要經常更改。久而久之，整個服務自動化的安裝流程，就會愈來愈複雜，難以維護。甚至到某一個階段後，整個安裝包就需要重寫。

直接式的自動化，需要IT人員從一開始，就對整個雲端服務的資源或第三方軟體的需求都有充份的掌握。IT對於安裝所用的工具跟慣例，也有一定的標準。IT會堅持在整合測試的環境，就用這些指定的工具，跟第三方軟體及其配置參數，一併進行測試。換句話說，直接式的自動化是比較合乎DevOps精神的。直接式也許在專案始創期會比較費時，也需要各部門用多一些時間來溝通。但它的好處，是比較不會因為程式或軟體的更新而受到影響。長遠來說，面對客戶的服務，還是用直接式的自動化，其投資回報會比較高。

**可變 (Mutable) 與不可變 (Immutable) 的基礎架構**

讀者可能會問，互聯網服務本來就是經常更新的，怎麼可能有不可變的基礎架構呢？事實上，在傳統的 IT作業裡，伺服器以及其各子系統軟體，都是常常要更新的。這裡說的「不可變」(Immutable)，是指開發人員與 IT 人員，都不能直接登入到生產環境裡面去安裝任何 patch 或升版軟體。反過來說，可變 (Mutable) 就是直接登入生產環境中變更。因为任何小修改都有服務損壞的風險，所以在開發時，就必須把「不可變架構」當做持續開發的架構程式碼來完成。

尤其是在 IDC 大量接納雲端架構後，虛擬機都是用映像檔來安裝的，因此這種不可變的基礎架構理論，就愈來愈合理。到了在雲原生的環境上，所有的 容器也都是從映像帶起來，映像檔也通常會跟應用以及第三方軟體綑綁在一起。在容器的架構上，IT 都不會登入到容器內做更新，而是製作一個新的映像檔，採用無中斷服務的流程，去把新的一版容器帶起來。因此在微服務跟 雲原生的生態系統上，不可變的基礎架構，成為當前主流。

不可變的基礎架構，有一個很重要的隱含假設，那就是基礎架構的更新，一定要非常快速，而且是高度可複製化的，而這也正是架構程式碼工具的強項。但在另一方面，雲端服務的最佳化呈現，往往跟硬體環境、工作負載很有關係，所以在系統部署好了以後，還是會有一些 配置參數，必需因應這個特定環境而作最後的調整。

**總結**

基礎架構即程式碼算是軟體開發、互聯網服務、CI/CD等技術實踐交互影下的自然進化。不論是 SaaS 服務或人工智慧應用，所有相關服務都需要頻繁地更新。在這個前題下，開發、整合、測試，IT 營運人員都需要緊密合作，能夠應付這種需求快速變化的技術實踐，就是直接用程式碼來溝通回應環境與服務的需求。DevOps 其實就是 Development (開發)和 Operations (營運) 的合名，可見基礎架構即程式碼必然會被企業廣泛接受的。

近年來容器化、微服務的生態環境下，採用宣示式的基礎架構程式碼，與直接式的自動化來部署虛擬機房環境，已經愈來愈流行。這部份的基礎架構，也是採用不可變的模式。但是在雲端服務的最佳化上，指令式的架構程式碼 還是有其需求。故此，近年來很多架構即程式碼的工具，都慢慢演變成可以 同時支援指令式與宣示式的架構 (如下圖)。

一張含有 文字, 螢幕擷取畫面, 字型, 數字 的圖片

自動產生的描述

**References**

1. <https://en.wikipedia.org/wiki/Infrastructure_as_code>
2. <https://www.thoughtworks.com/insights/blog/infrastructure-code-iron-age-cloud-age>
3. <https://martinfowler.com/bliki/InfrastructureAsCode.html>
4. <https://cfengine.com/product/what-is-cfengine/>
5. <https://techbeacon.com/enterprise-it/infrastructure-code-engine-heart-devops>
6. <https://www.plutora.com/blog/infrastructure-as-code>
7. <https://www.trendmicro.com/zh_tw/what-is/cloud-security/infrastructure-as-code.html>
8. <https://stackoverflow.blog/2021/03/08/infrastructure-as-code-create-and-configure-infrastructure-elements-in-seconds/>
9. <https://fpalomba.github.io/pdf/Conferencs/C42.pdf> (Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry )
10. <https://www.itread01.com/content/1542762148.html>
11. <https://phoenixnap.com/blog/ansible-vs-terraform-vs-puppet>

原文; <https://medium.com/gemini-open-cloud/%E6%B7%BA%E8%AB%87%E6%9E%B6%E6%A7%8B-%E7%A8%8B%E5%BC%8F%E7%A2%BC-2e83f3ff6236>