前言

歡迎大家選購《網路可程式性與自動化》一書！

網路業界正在發生翻天覆地的變化。而如今驅使各機構及網路專家們擁抱網路可程式性與自動化的想法及觀念的動力，也較過往更為強烈，在這背後推動的是各種革命性的新協定、新技術及新交付模式，再加上對於新業務的需求，若要提升競爭力，就必須更為敏捷及彈性化。然而網路可程式性與自動化的本質究竟為何？本書就從回答這個問題開始。

本書涵蓋的內容

正如書名字面所示，本書主要專注在網路的可程式性與自動化上面。其中心觀念在於，凡是涉及網路設備、網路技術、網路服務及網路連結的設定、管理及運作，網路可程式性與自動化就是要簡化與它們相關的各種任務。

這中間牽涉到數不清的部件——包括如今在網路上運用的廣泛程度更勝以往的作業系統、還有像是持續整合（continuous integration）之類的新方法論、以及在過去只會出現在系統管理員領域的工具（像是原始碼控管及組態管理系統）等等。由於這些內容都會在網路可程式性與自動化的核心定義中占有一席之地，我們會全數一一介紹。作者們為本書所設的目標，就是要讓讀者們能建立起關於網路可程式性與自動化的基礎知識。

本次改版的新增內容

本書此次再版，比初版多了以下四個全新章節：

- 第四章，〈雲端〉
- 第五章，〈網路開發人員的環境〉
- 第七章，〈Go〉
- 第十四章，〈網路自動化架構〉

此外，我們也改寫了既有章節，納入以下的更新內容：

- Google Protocol Buffers
- gRPC/gNMI
- Terraform
- Nornir

除了上述新增的內容，我們也逐章翻新並擴充內容，以便反映自從本書出版以來，業界發生的種種進展及變化。

我們很高興能有機會為本書添加這些新的有趣題材。但囿於篇幅，有些初版時的內容便不得不割愛。然而我們並未將這些舊內容完全捨棄，而是改放到 *https://oreilly-npa-book.github.io* 當中供大家免費展讀。

本書章節編排

本書並不要求讀者們必須從頭讀到尾；相反地，我們盡量將各種題材拆開來，以便讀者們輕易就能找到自己最有興趣先讀的內容。也許你會覺得先依序讀完前三章會很有幫助，因為這三章提供了背景資訊，並替本書隨後的章節打好了基礎。接下來你可以隨意地跳到你覺得最有用、或最有興趣的題材，開始讀下去。

筆者們嘗試讓各章之間盡量自成一格，但就像任何技術一樣，你不太可能完全不涉及其他內容。因此我們會盡量地提供交叉參閱說明，協助讀者們找出自己需要的參照內容。

為何要網路自動化？

網路自動化就跟大多數自動化的類型一樣，被視為是更快速的工作方式。雖說能更迅速地完成任務是件好事，但縮短部署及組態變更所需的時間，卻並不一定是許多 IT 組織急需解決的問題。

在這個小節裡，我們要來檢視幾種因素，亦即任何形式及規模的 IT 組織為何應該考慮逐步導入網路自動化，其中也包括了速度。你應該注意，相同的原理也適用於任何導入自動化的類型（像是應用程式、系統、儲存、電話等等）。

簡化架構

如今依然很容易見到所謂獨特雪花組態的網路設備（snowflakes，意指內有大量只在此設定過一次的非標準組態），而網路工程師依舊為了能以一次性的網路變更來解決傳輸及應用程式問題而感到滿意，但這種變更方式到頭來反而讓網路不只更難以維護和管理、也妨礙了自動化的進行。

網路自動化及管理不該在一開始被視為次要專案或附加功能，而是應在一開始建立新架構時便包括在內。包括要確保充裕的人員及工具預算。但是很不幸的是，工具常是預算短缺時第一個被犧牲的項目。

點對點架構及相關的第二日（day 2）操作，都應該是一體且一致的。在建立架構時，你應該思考以下問題：

- 哪些功能可以跨廠商運作？
- 哪些延伸功能可以跨平台運作？
- 什麼樣的 API 或自動化工具只能搭配特定網路裝置平台運作？
- 是否有可靠的 API 文件？
- 特定產品有哪些程式庫可用？

如果你能在設計階段及早考慮以上問題並取得解答，結果的架構必能變得更簡化、易於重複實現（repeatable）、也更容易維護以及自動化，並且減少網路中所需的廠商專屬延伸功能。

即使在部署了配有正確管理及自動化工具的簡化架構之後，請記住，減少特立獨行的變更仍然有其必要，這是為了確保網路組態不再出現上述的雪花狀況。

確定性的結果

在一個企業組織中，通常會舉行變更審查會議，過程中會檢視即將來臨的網路變更、以及變更會對外部系統的影響、還有還原計畫等等。在一個人們還能透過 CLI 來變更內容的世界裡，打錯命令的影響可能是一場大災難。設想一個由 3 位、4 位、5 位、甚至是 50 名工程師組成的團隊。每個人可能都有自己的一套進行變更調整的作法。此外，能夠使用 CLI 甚至是 GUI，並沒有辦法消除或是減少在變更控制窗口期間出錯的機會。

但是與手動變更相比，改用經過驗證和測試的網路自動化方式來進行變更，有助於達成更容易預測的行為，而且執行團隊有更多機會達成有確定性（deterministic）的結果，亦即更能確保第一次就把手中任務做對，不會出現人為錯誤。這類任務可能是任何動作內容，從 VLAN 調整、到需要在網路內四處進行變更的新客戶登錄等等。

此外，確定性的結果代表更低廉的營運成本（operating expenses，OpEx），因為在執行網路變更時涉及的人力會更少，也提升了整體網路營運的效率（譬如把在網路裝置上升級作業系統這種耗時的過程加以自動化）。網路工程師所省下的營運時間可以拿來花在更富策略性的專案上，以便持續地改進程序。

業務敏捷性

自從伺服器虛擬化興起以來，系統管理員幾乎可以在一瞬間完成新應用程式的部署。而隨著應用程式的部署越發迅速，人們便越會開始質疑，如果只不過是要部署一個三層式（three-tier）的新應用程式，何以像是設定 VLAN、路由、防火牆原則、負載平衡原則這些網路資源設定要花那麼久的時間。

很顯然地，如果能採取網路自動化，網路工程及營運團隊便能更快速地回應其他 IT 團隊部署應用程式的需求。更重要的是，自動化讓業務更形靈活。從採用的角度來看，無論你原本要讓業務敏捷化的立意有多良善，重點在於你嘗試採用任何一種自動化之前，都必須先理解既有的（通常也是手動進行的）工作流程。

如果你對於要把什麼東西自動化一無所知，缺乏知識會使得採用的過程更複雜、更冗長。我們的首要建議是，當你展開網路自動化之旅時，務必了解既有的手動工作流程、加以記錄（像是平均完成時間、每個步驟發生的次數），並了解它們對於業務的影響。然後自動化解決方案的實施才會更為簡化、也更為有效。

改進安全性並減少風險

以程式碼的形式來描述網路相關工作流程，會隱然帶來另一項自動化的好處：程式碼會自然而然地捕捉並記錄每個步驟的程序定義及分析，這樣一來，任何人都隨時可以取得其內容。這樣的公開方式就等於為工作流程引進了同儕審閱（peer review，亦即其他工程師也分擔了變更的責任）及自動化的處理方式，可以在實際將變更部署至網路之前就找出潛在的安全問題或組態錯誤。而且一旦實施自動化，就能確保網路必定按照定義繼續運作，並依需求修正偏差。

在傳統的網路營運方式中，任務的結果有很大程度取決於負責執行的網路工程師是否經驗豐富、以及相關文件是否存在，而且文件總是難以隨時保持更新。但是當這樣的任務自動化以後，每次當工作流程執行時，所有曾對該流程做出貢獻的工程師，他們的知識都會濃縮在這個流程當中。

自動化可以讓整個團隊一起分擔責任，並讓程序更為有效、也更不容易出現問題，因為它一定會強制實施額外的人為審閱。自動化也會以工具來進行處理，這樣就能強制施行原則（譬如安全原則）、甚至是連結人工分析工具，根據場合來建議改進方式。

還有，網路自動化並非一次性的工作。總是會出現意料之外的問題、或是錯失某些要點，只要發生這種事，工作流程的程式碼就需要加以調整，使其更為完善。這種隨時間進展而持續改進的手法，會在逐步改善的過程中，將前人及現有成員的所有貢獻合而為一。

從簡化架構到持續改進，這個小節介紹了若干你之所以該考慮網路自動化的高階因素，下一小節則要來看看網路自動化的類型。

網路自動化的類型

自動化通常意味著速度，但考量到某些網路任務並不十分在意速度，因此就很容易看出何以有些 IT 團隊不覺得自動化有其價值。VLAN 的設定就是一個很好的例子；你會自忖「建立 VLAN 的速度要快到什麼地步？平常一天會建立幾個 VLAN？我真的需要這樣的自動化嗎？」這些都是有意義的問題。

這個小節會專注在幾項自動化的確有用的任務上：像是設備的配置（device provisioning）、資料的擷取及補足、轉移、組態管理、組態的合規遵循、狀態驗證、故障排除以及報表等等。但是請記住我們先前提過的，自動化並不只是速度和敏捷性而

檢視自動化工具

雖說這些工具都著重在自動化上面，但每一種都有它自己的架構，並以略有出入的手法來達成自動化。因而它們各自有其長處和短處。在這個小節裡，我們會很快地檢視一下每一種工具，這樣你就能著手判斷它們可能如何運用在你的環境當中。

從高階觀點來看，這些工具的主要架構 / 概念之間具有以下的差異：

組態管理相較於基礎架構配置

基礎架構配置（infrastructure provisioning）所指的是建立基礎架構的過程——基礎架構包括網路服務、虛擬機器、資料庫等等。組態管理（configuration management）則是將安裝軟體元件及進行組態管理任務等事情自動化的過程。因此你可以把基礎架構配置看成 day 0 的動作，而組態管理是 day 1 的動作。儘管大部分的自動化工具都可以做到這兩點，但本章依舊會協助你理解各種工具特有的長處。

有代理程式相較於無代理程式

有些工具需要讓代理程式——這也是軟體的一種——在要被管理的系統或裝置上運行。以網路自動化的運用案例來說，這也許會是一個問題，因為並非所有的 NOS 都支援能在網路裝置上運行代理程式。如果 NOS 不支援在裝置上直接運行代理程式，就必須採迂迴方式，譬如 proxy agent。而無代理程式的工具顯然並不需要代理程式，因而可能較適合網路自動化的運用案例。

集中式相較於非集中式

基於代理程式的架構通常還需要一套中央式的主伺服器（centralized master server）。有些無代理程式產品也會用到主伺服器，但它們多半是分散式的。

自訂協定相較於標準式協定

有的工具會採用自訂協定；這通常與基於代理程式的架構有關。其他工具則會利用 SSH。由於 SSH 常見於網路裝置，因此採用 SSH 作為傳輸協定的工具也許更適於網路自動化運用案例。

DSL 相較於標準式的資料格式和一般通用程式語言

有些工具擁有自己的領域專屬語言（domain-specific language，DSL）；使用這類工具的人必須以該種 DSL 建立適當的檔案，以便讓自動化工具讀取。DSL 是一種針對特定領域（或工具）為目的而打造的語言。對於尚不熟悉 DSL 的人們來說，這可能會衍生額外的學習曲線。其他工具則會使用 YAML，在網路自動化背景中它被視為通用目的類型的語言。我們在第八章時已介紹過 YAML。

宣告式相較於命令式

有些工具以宣告式的手法（declarative approach）來定義基礎架構的最終狀態：這與定義的順序無關，工具自會推斷出正確的依存關係和任務並加以執行，以便達到目標的狀態。其他工具則會使用命令式手法（imperative approach）：每個步驟都是一個要進行的程序，而定義的順序便代表了執行的順序。

延伸性

這類自動化工具大部分都允許你藉由像是 Python 和 Go 等高階指令稿語言來添加或延伸功能。

推送式相較於拉取式或事件驅動式

有些自動化工具會以推送模式（push model）運作：資訊會從一個地方被推送給受管控的裝置或系統。其他則採的是拉取模式（pull model），通常用來拉取組態資訊或指令（多半以某種定期方式來進行）。最後，事件驅動式的工具則會根據對於別的事件或觸發器做出的回應而進行動作。

可變相較於不可變

傳統上，當你需要更改基礎架構的組態時，指的就是你對於當前狀態套用了異動內容，因而變更（*mutated*）了其狀態。這個過程又被稱為可變式手法（*mutable approach*），通常就是組態管理工具運作的方式。相較之下，不可變（*immutable*）的手法則代表你得用全新的基礎架構來取代 / 重啟原本的基礎架構，才能改變其狀態。即使只是些微的變更，譬如更改伺服器的主機名稱，你都得從零開始重新配置整套伺服器。但依據著重之處的差異，每種案例會適用不同的工具。

狀態管理

組態管理工具不會去管遠端基礎架構的生命週期（lifecycle）。你可以為每個步驟蒐集當前狀態，但它不會自行追蹤狀態。相反地，基於不可變樣式的工具則需要判斷何時需要重新建立某個基礎架構的元件，因此只要是經由它們配置的遠端基礎架構，它們便會保有其狀態。

在心目中對這些高階架構差異有了概念之後，我們可以快速地瀏覽一下本章所介紹的三種工具了：

Ansible

Ansible 採用的是分散式、無代理程式的架構，以 SSH 作為底層傳輸協定。它通常以推送模式運作，但它也支援拉取模式。Ansible 以 Python 建置，並廣泛地運用了 Python。Ansible 支援的範本撰寫方式採用 Jinja 範本語言。Ansible 原本的目標是以能夠在伺服器上運行特定命令為主，但它已逐漸發展成任務調度工具，會按照 *playbooks* 指示，在目標系統上進行 idempotent 的任務。Playbooks 可以用標準的 YAML 來撰寫、也可以用 Ansible 衍生版本的 YAML 來寫。

Nornir

Nornir 屬於輕量型的 Python 框架（但並非 CLI），該框架係針對網路自動化而設計，可以透過添加新功能（例如 tasks 或 inventory sources）的外掛程式來擴充，而無須從頭開始撰寫。它可以做為 Ansible 這類自動化工具的替代品，因為它是純粹的 Python 工具，讓你可以完全掌控程式邏輯，並可直接取用所有的程式庫及工具。Nornir 含有若干 Python 的 constructs 型別資料，因而能夠更容易運用主機構成的 inventory、運行 tasks 並處理其結果，完全以結構化的方式進行。此外，Nornir 預設便是使用多執行緒處理方式，因此所有的 tasks 都是每台主機平行進行的。

Terraform

Terraform 所採的則是與此處其他工具截然不同的手法。Terraform 著重於基礎架構的配置、而非組態管理，它允許你以容易閱讀而且是宣告式的組態檔案來定義自己的基礎架構（並管理其生命週期），該檔案以 Terraform 組態語言寫成。它獨立於基礎架構的服務供應端之外，可以輕易經由 providers 擴充，因而成為與 IaaS 週邊系統關聯最密切的工具。

介紹網路自動化架構

儘管每一種網路自動化解決方案也許都自有不同的實作方式或工具，但它們全都擁有某種共通模式。能夠識別及分類這些模式，將有助於你採行系統化的手法來建置並持續地發展新的解決方案、同時令解決方案能呼應其特定功能性。這便是此處提議的網路自動化架構的目標所在。

圖 14-1 描繪出了網路自動化解決方案的六大元件。架構中的每個元件都擁有至少一個以上的功能性目標（基於其實作方式），而且元件彼此之間皆有連結，以便達成解決方案的預期目標。了解每一種元件，你才能決定應該在何處進行實現網路操作工作流程所需的必要動作，並輕易看出需要應付的挑戰以及可行的實作選項。

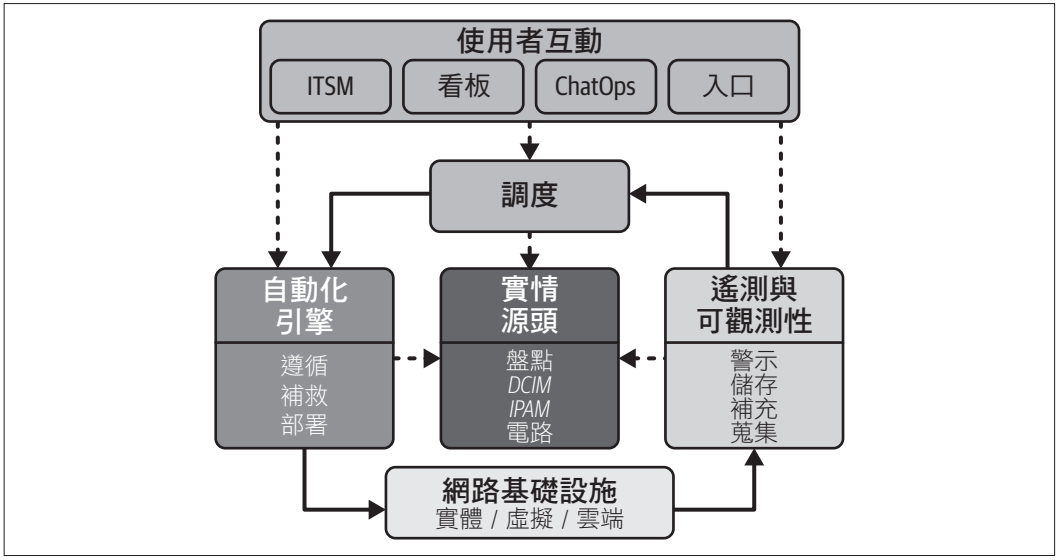

圖 14-1　網路自動化架構

首先我們來簡短介紹一下每一項架構元件：

網路基礎架構

第一章已介紹過了網路基礎架構的新趨勢（例如 NFV 和雲端 / 控制器式的服務）。所有這些包括傳統的網路實體裝置，都涵蓋在網路基礎架構的範疇之下。本書通篇已涵蓋了此一元件：第十章探索了可用的管理介面、第四章則提出了新的網路功能事例、並在第五章當中加以模擬。

使用者互動

人類終須以某種方式與網路自動化解決方案互動。如果你採用的是 GitOps 工作流程，也許會與 Git 互動；如果你為非技術人員採用了自助式服務，也許就會與案件通報系統（ticketing system）互動。每一種使用案例都需要適當的介面，為每一種案例選擇正確的介面，會對解決方案的採用產生大幅影響。走對方向至關緊要。與使用者互動的方式會讓文化和人們能夠向網路自動化靠攏。最壞的示範，就是一個到頭來沒人要用的介面。在 748 頁的「使用者互動」一節中，我們會描述互動所採行的各種形式，像是服務入口網站、看板以及訊息應用程式等等。

實情來源

每一種網路自動化解決方案都需要資料可以作為操作的對象。在不需持續管理網路的簡單解決方案中，資料可以就地提供。譬如一個使用者可以觸發一支小型命令稿，去進行小型的自動化任務。但是當網路自動化解決方案必須持續地管理網路狀態時，該解決方案就得參照那些我們稱之為意欲達成狀態（*intended state*，指的是你希望自己的網路應該呈現的狀態）。正如你在 756 頁的「實情來源」一節中會讀到的，描述網路狀態這件事可能涵蓋了需要部署及管理網路的多個資料運用案例——像是 IP 定址、VLANs、路由協定、ACL 等等。還有，要正確地實作出實情的源頭，你必須要了解關於資料管理的限制及取捨。

自動化引擎

此一元件包括了所有與變更網路基礎架構狀態（更一般化的說法是與網路基礎架構互動）有關的任務。它透過使用者互動或實情來源取得資料作為輸入，以便透過適當的介面與網路互動。正如你在 775 頁的「自動化引擎」一節中會看到的，自動化引擎涵蓋了一切與組態管理及其他操作任務（例如裝置重啟）相關的面向。在這個小節裡，讀者們將能辨識出第十二章所介紹的工具，或是第十章引用程式庫的命令稿等所扮演的角色。

遙測與可觀測性

與自動化引擎正好相反，遙測與可觀測性元件主要只著重於取回（以唯讀方式）網路的運作狀態。在 781 頁的「遙測與可觀測性」一節中，大家會學到如何蒐集、儲存以及運用狀態，在網路的實際狀態不符合實情來源所定義的期望狀態時，提供適當的視覺化效果或警示，以便觸發其他的自動化任務。

REST API 與 GraphQL

第十章解釋過的 REST API，是在以程式化的方式與應用程式互動時最受歡迎的方式。然而，當資料需要由用戶端整合與自訂時，其他的 API 協定也許會更有用。由 Meta（舊稱 Facebook）開發的 *GraphQL*，就能只用一個 API 呼叫做到 REST 需要好幾個呼叫才能做到的事情。

在這個例子裡，你可以輕易地比較如何取得相同資料的方式。GraphQL 的 API 允許你嵌入一筆查詢、其中定義了所需傳回的資料。如果是使用 REST API，你得分別存取個別的 REST API 端點（下圖中的綠色細虛線），最後才在用戶端組合成資料。而另一方面，GraphQL 的用戶端只需存取單一端點（下圖中的紅色粗虛線），查詢中則只需準確地定義所需的資訊。

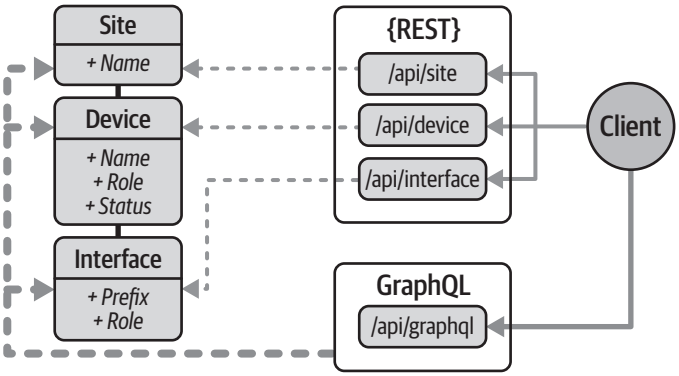

圖 14-5　REST 與 GraphQL API 的比較

現在我們要重現範例 14-1，但我們改成用 POST 方法把一個查詢的定義傳給 API 路徑。該筆查詢也許包含了變數過濾器，以及連結模型中的巢狀屬性選擇（譬如選出 `platform` 之下的 `name`）：

```
>>> query = """
... query {
...   devices (name: "ams01-edge-01"){
...     name
...     platform {
...       name
...     }
...   }
```

```
... }
... """
>>> response = requests.post(
...     f"{url}/api/graphql/",
...     json={"query": query},
...     headers=headers,
... )
>>> print(json.dumps(response.json(), indent=4))
{
    "data": {
        "devices": [
            {
                "name": "ams01-edge-01",
                "platform": {
                    "name": "Arista EOS"
                }
            }
        ]
    }
}
```

我們在此不會對 GraphQL 多所著墨，只希望以上簡短的比較可以讓大家體會到 GraphQL 相較於 REST API 的若干獨特優勢。GraphQL 允許你可以有效地取回所需資料，同時因為只需較少的查詢就能取回資料，進行起來也更有效率。請參閱官方 GraphQL 學習網頁（*https://graphql.org/learn*）以擴展你對它的知識。

一旦盤點資料就緒，就可以用其他的相關資料來補強資料內容了。

資料中心的基礎架構管理。 在實體網路環境裡，資料中心基礎架構管理（*data center infrastructure management*，*DCIM*）定義了網路裝置應位於何處、應如何彼此串接。它也許還包含了關於機櫃、裝置硬體、供電、纜線、電路等資訊，或任何其他建構實體網路所需的一切細節。

範例 14-2 便沿用了來自範例 14-1 的硬體，從中取回從一個介面到另一個裝置應有的纜線追蹤資訊。

範例 14-2　探索 Nautobot 的纜線追蹤資訊

```
>>> device_id = '9d512f81-5523-456d-8508-506da78fe6e2'
>>> response = requests.get(
>>>     f"{url}/api/dcim/interfaces/?device_id={device_id}", headers=headers)      ❶
>>> interfaces = response.json()["results"]
```

現在你對自動化解決方案要做些什麼已有明確的腹案，可以動手把它對應到網路自動化架構上了。

把自動化任務對應到架構元件上

定義好步驟之後，你就會開始注意到它們之間的協同作用，這會有助於對其中部分步驟進行分組。然後這樣的分組會對應到不同的架構元件上。正如我們在 801 頁的「調度」一節所述，每種任務都應縮小到可以納入到一個區塊當中。如果不是這樣，請改選另一個更符合任務目標的任務。

圖 14-14 描繪了一個流水圖中的自動化工作流程；物件皆為架構中的元件。

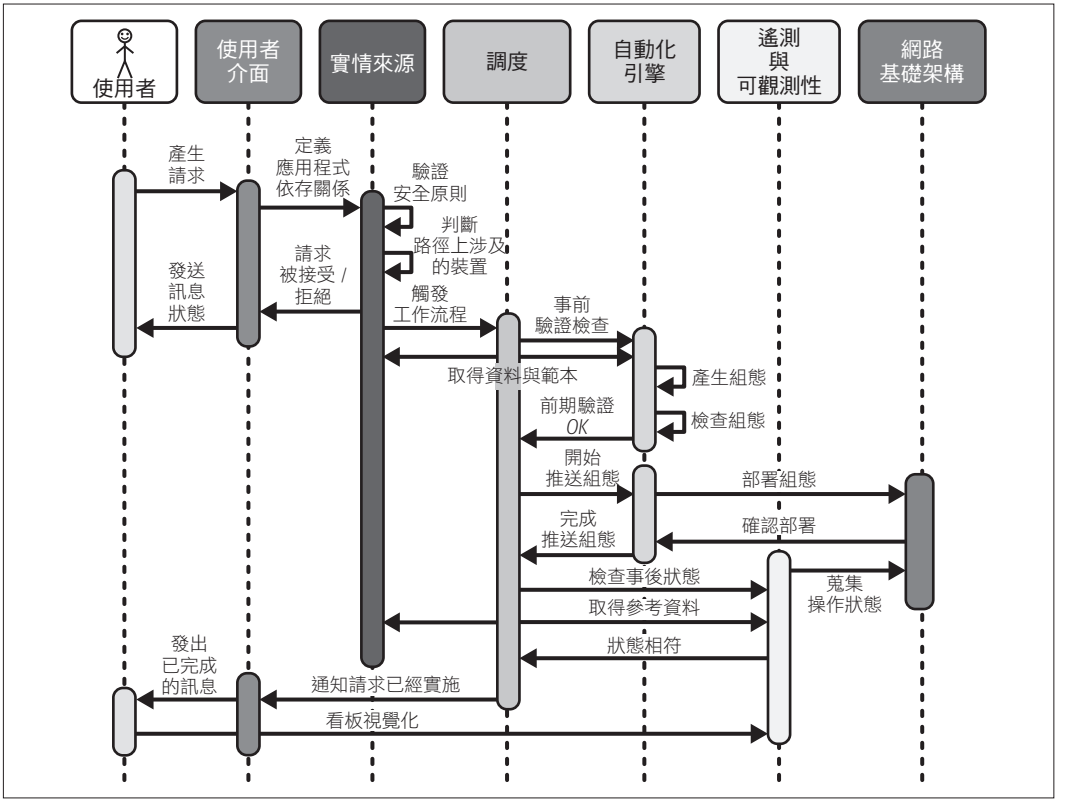

圖 14-14　流水圖中的自動化工作流程