序

軟體產業一直認為開發人員必須在寫出第一行程式之前，就開發並完成軟體架構。受建築業啟發，一般認為，在開發的過程中，成功的軟體架構可以避免更改，大家不喜歡更改的原因是重新建立架構意味著重新工作，它也會產生高成本的報廢。

之所以預先規劃架構，也是因為開發人員認為應該在編寫程式之前確定需求，這種看法導致階段式（或瀑布式）方法的出現：先確定需求，才能設計架構，先設計架構，才能實際建構（編程）。但是，敏捷軟體方法的興起對這種觀點發起猛烈的挑戰，挑戰“需求不變”的概念，認為定期變更才能夠滿足現代的商業需求，敏捷方法也提供許多專案規劃技術，可在控制之下執行變更。

在這個全新的敏捷領域中，很多人質疑架構的功用。預先規劃的架構確實不適合現代的動態，但我們可以採取另一種方式來建立架構，以敏捷的方式來配合變更。我們認為，建立架構是一種持續性的工作，應該與編程緊密合作，如此一來，架構不但可以對持續變動的需求做出反應，也可以回應編程的回饋。我們將它稱為演進式架構（evolutionary architecture），藉此強調，儘管變動是無法預測的，我們仍然可以讓架構朝著良好的方向前進。

我們在 ThoughtWorks 一直沉浸在這種架構世界觀之中。Rebecca 在這個千禧年之初帶領我們進行許多重要的專案，並且培養我們首席技術長的能力。Neal 一直都很細心地觀察我們的作品，歸納並傳達我們學到的教訓。Pat 使用他的專案成果來栽培我們的技術經理。我們一直認為架構非常重要，不能坐視不管。我們曾經犯錯，但也從中吸取教訓，因此更瞭解如何組建一個“能夠優雅地回應目的變更”的基礎程式。

建立演進式架構的核心是採取小規模變更，並且植入回饋迴圈，讓所有人都可以在開發系統的過程中學習。持續交付的興起是演進式架構越來越可行的關鍵因素。這三位作者使用適應度函數的概念來監視架構的狀態。他們研究各種架構演進風格，並將重點放在與長時間存留的資料有關的問題上——這是一種經常被忽視的主題。Conway 定律的重要性遠超過大部分的討論，確實也該如此。

雖然我認為，以演進式風格來建立軟體結構還有許多需要學習的地方，但本書是一張重要的路線圖，指出目前為止的理解狀態。隨著越來越多人認識到軟體系統在 21 世紀的世界發揮核心的作用，任何一位軟體領導者都必須知道如何既保持良好的狀態，又能夠對變化做出最佳反應。

— *Martin Fowler*
martinfowler.com
2017 年 9 月

第一章

軟體架構

軟體架構的範圍廣大且不斷變化，所以長期以來，開發人員很努力地為它找出一個簡潔的定義。Ralph Johnson 將軟體架構定義成“重要的東西（無論它是什麼）”。架構師的工作，就是了解並平衡所有這些重要的事情（無論它們是什麼）。

在一開始，架構師要先了解解決方案的商業或領域需求，這些需求是使用軟體來解決問題的動機，但它們只不過是架構師在建立架構時需考慮的因素之一，架構師還有許多其他因素需要考慮，有一些是明確的（例如，性能服務層級的協議），有一些是暗藏在商務性質底下的（例如，公司正熱切地進行併購）。因此，要建立軟體架構，架構師必須分析商業和領域需求，以及其他的重要因素，來找到一個最能夠平衡所有關注點的解決方案。軟體架構的範圍，就是由這些架構因素組成的，如圖 1-1 所示。

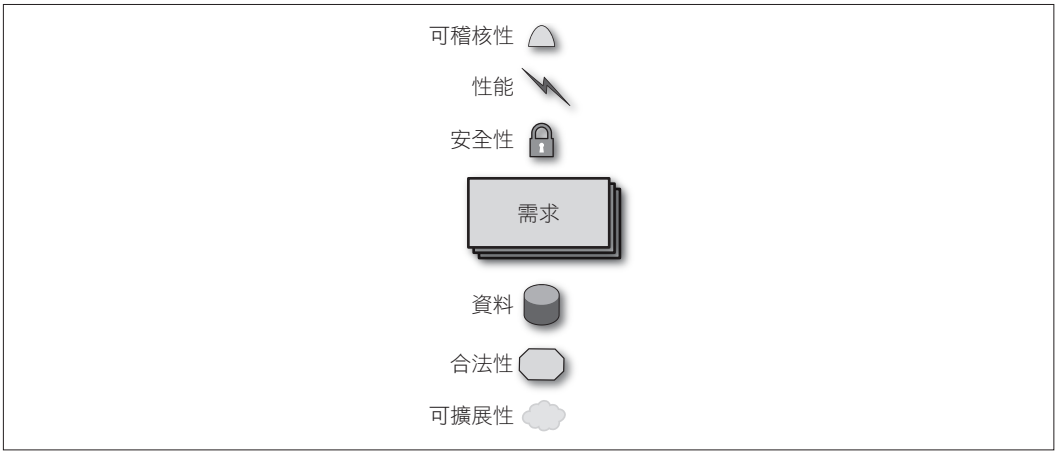

圖 1-1　架構的整個範圍，包括需求與“-ility（性）”

在圖 1-1 中，商業和領域需求是與其他的架構關注點並存的，其中包含許多外部因素，它們都有可能改變“該建構哪種系統”以及“該如何建構系統”的決策過程。見表 1-1：

表 1-1　部分的“-ility”

accessibility	accountability	accuracy	adaptability	administrability
affordability	agility	auditability	autonomy	availability
compatibility	composability	configurability	correctness	credibility
customizability	debugability	degradability	determinability	demonstrability
dependability	deployability	discoverability	distributability	durability
effectiveness	efficiency	usability	extensibility	failure transparency
fault tolerance	fidelity	flexibility	inspectability	installability
integrity	interoperability	learnability	maintainability	manageability
mobility	modifiability	modularity	operability	orthogonality
portability	precision	predictability	process capabilities	producibility
provability	recoverability	relevance	reliability	repeatability
reproducibility	resilience	responsiveness	reusability	robustness
safety	scalability	seamlessness	self-sustainability	serviceability
securability	simplicity	stability	standards compliance	survivability
sustainability	tailorability	testability	timeliness	traceability

在建構軟體時，架構師必須確定哪個“-ility”是最重要的，但是，許多因素都是互相矛盾的。例如，你很難同時實現高性能和極度可擴展性，因為這必須仔細地平衡架構、運維和許多其他因素才能做到。因此，當你設計架構時，必須進行分析，並且平衡難免互相衝突的各種因素，但是，當你平衡各個架構決策的優缺點時，會產生架構師很不喜歡的**權衡取捨**。在過去幾年中，軟體開發的核心工程方法——漸進開發——已經奠定了基礎，讓人們重新思考如何讓架構隨著時間變化，以及如何在變化發生時保護重要的架構特性。本書會把這些部分聯繫在一起，用全新的方式來思考**架構**和**時間**。

我們幫軟體架構加入一種新的標準“-ility”——**可演進性**（*evolvability*）。

演進式架構

儘管我們已經盡了最大的努力，但隨著時間的過去，軟體也會變得越來越難以變更。由於各種原因，組成軟體系統的各個部分會變得不容易修改，隨著時間越來越脆弱且難以管理。因此人們通常會先重新評估功能與範圍之後，再決定是否變更軟體專案。但是有另一種變更是架構師和長期規劃人員沒辦法控制的——雖然架構師希望進行戰略性的未來規劃，但是不斷變化的軟體開發生態系統讓他們很難做這件事。既然無法避免變化，我們就要利用它。

既然一切都在不斷變化，怎麼可能進行長期規劃？

在生物世界中，由於自然和人為的原因，環境會不斷變化。例如，在 1930 年代初，澳洲遇到甘蔗甲蟲災害，使得甘蔗作物的生產和收穫利潤減少。在 1935 年 6 月，糖業實驗管理局引進一種食肉動物——甘蔗蟾蜍，在那之前，這種蟾蜍只棲息在美國的南部和中部[1]。在 1935 年 7 月和 8 月，經過人工飼養後，澳州政府將許多年輕的蟾蜍放到昆士蘭北部。甘蔗蟾蜍皮膚有毒、沒有天敵，因此廣泛地繁殖，目前估計有 2 億隻。這個故事告訴我們，對高度動態的（生態）系統進行變更，可能會產生不可預測的結果。

軟體開發生態系統是由所有的工具、框架、程式庫和最佳實踐法（在軟體開發領域中，持續累積的最新技術）組成的，這個生態系統形成一種均衡狀態（很像生物系統），可讓開發人員在裡面理解並建構東西。但是這種均衡是動態的——新事物會不斷出現，破壞均衡，再出現新的均衡。想像有一位獨輪車手扛著箱子，因為他必須不斷調整身體才能保持直立，所以產生**動態**，因為他要不斷保持平衡，所以產生**均衡**。在軟體開發生態系統中，每一個創新或新的實踐法都有可能打破現狀，建立新均衡。打個比方，這就像我們不斷把更多箱子扔到獨輪車上，迫使車手重新建立平衡。

架構師很像那位不幸的獨輪車手，必須不斷地平衡和適應持續變化的條件。持續交付這種工程實踐法就是在這種均衡之下的結構性轉變，它將之前獨立的職能（例如運維）併入軟體開發生命週期，讓人們對於**變更**有了全新的看法。企業架構師再也不能依賴靜態的 5 年計畫了，因為整個軟體開發領域都會在這段時間內不斷發展，使得每一項長期決策都可能變得毫無意義。

1 Clarke, G. M., Gross, S., Matthews, M., Catling, P. C., Baker, B., Hewitt, C. L., Crowther, D., & Saddler, S. R. 2000, Environmental Pest Species in Australia, Australia: State of the Environment, Second Technical Paper Series (Biodiversity), Department of the Environment and Heritage, Canberra.

我們很難預測顛覆性的改變，即使是精明的實踐者也是如此。用 Docker（*https://www.docker.com*）等工具產生的容器就是無法預期的產業轉變案例，但是，我們可以藉由一系列小規模的、漸進的步驟，來追蹤容器的興起。在以前，作業系統、應用伺服器和其他基礎設施都是商用產品，需要付出巨額的開銷購買許可權，在那個時代設計出來的架構，大都把注意力放在有效地使用共享資源上面。漸漸地，許多企業認為 Linux 已經夠用了，使得作業系統的**金錢**成本降為零。接下來，DevOps 實踐法，例如透過 Puppet（*https://puppet.com/*）或 Chef（*https://www.chef.io/*）等工具來自動供應機器，也讓 Linux 的**運維**成本歸零。當生態系統變成免費，並且被廣泛使用時，大家就一定會圍繞著常見的可移植格式進行整合，因此，Docker 出現了。但是如果沒有之前不斷演進的步驟，容器化就不可能發生。

我們的編程平台也是不斷演進的例子。新版的程式語言提供更好的應用程式編程介面（API），可提升處理問題的靈活性或適用性；新程式語言則提供不同的範本和不同的構造。例如，Java 是 C++ 的替代品，目的是降低編寫網路程式的難度，以及改善記憶體管理問題。回顧過去 20 年，我們可以發現許多語言仍然不斷地發展它們的 API，而全新的程式語言之所以出現，似乎都是為了處理新問題。圖 1-2 是程式語言的演進過程。

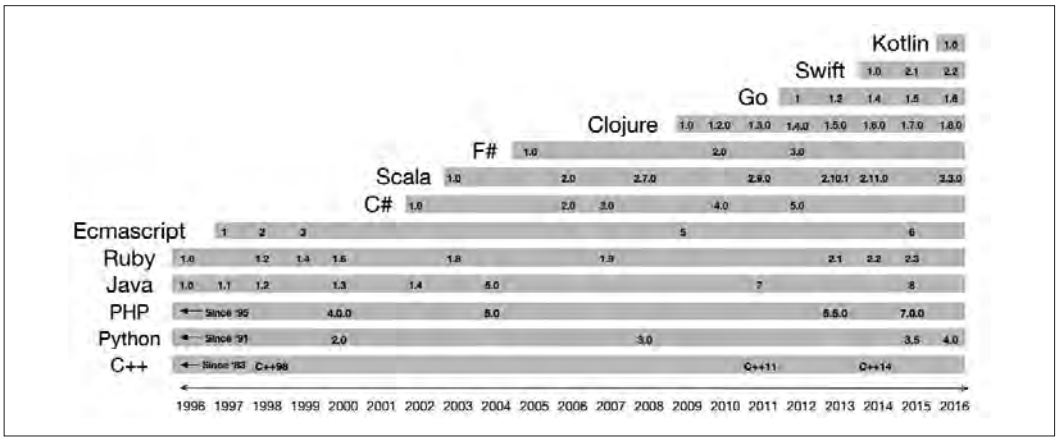

圖 1-2　熱門程式語言的演進過程

無論在哪個軟體開發層面（編程平台、語言、作業環境、持久保存技術等），我們都可以預期，變化將會不斷發生。儘管我們無法預測技術或領域的變化何時發生，或哪些變化會不斷持續下去，但我們知道，變化是不可避免的。因此，當我們架構系統時，應該有“技術領域將不斷變化”的覺悟。

既然生態系統總會以意想不到的方式不斷變化，而且根本無法預測，我們該用什麼方案來**取代**原本的固定式計畫？企業架構師和其他開發人員都必須學會適應環境。制定傳統的長期計劃有一部分的原因與財務有關，因為變更軟體的成本很高，但是，現代的工程實踐法已經推翻這個前提了，因為它們可以將過往的人工程序自動化，並且運用 DevOps 等先進的技術來降低變更的成本。

多年來，許多機靈的開發人員發現系統中有些部分比其他部分更難修改，這就是有人將**軟體架構**定義成“日後難以變更的部分”的原因，這個方便的定義區分了**可以**輕鬆修改的部分與難以變更的部分。不幸的是，在規劃架構時，這個定義變成一個盲點：當開發者假設“變更是困難的”時，它就變成一種自證預言（self-fulfilling prophecy）了。

幾年前，有一些有創意的軟體架構師重新思考“以後很難更改”的問題：如果將可變性**植入**架構會怎樣？換句話說，如果**易於更改**是架構的基本原則，更改就不難了。將可演進性植入架構之後，我們就容許全新行為的出現，並且再次顛覆動態均衡。

即使生態系統沒有改變，我們又該如何應對架構的特性被逐漸侵蝕的情況？架構師好不容易設計了架構，卻又在混亂的世界中公開它們，任由許多人以它們為基礎進行**實作**。架構師該如何保護他們定義好的重點區域？

建立架構之後，我該如何防止它隨著時間逐漸劣化？

許多組織都會發生一種不幸的衰退，通常稱為**位元衰減**（*bit rot*）。架構師已經選擇特定的架構模式來處理業務需求和“性能”了，但是那些特性經常隨著時間而意外地劣化。例如，架構師已經建立一個分層的架構了，該架構的頂層是表示層，底層是持久保存層，之間有許多中間層。出於性能原因，負責報表的開發人員經常要求繞過其他層，直接從表示層造訪持久保存層。架構師建立階層是為了隔離變更，但開發人員繞過那些階層卻增加耦合，使得階層最初的功用失效。

一旦架構師定義了重要的架構特性，他們該如何**保護**那些特性，使它們免受侵蝕？在架構中加入**可演進性**的意思就是在系統演進時保護其他的特性。例如，架構師為可擴展性設計了一個架構，他不希望那個特性隨著系統的發展而退化，此時，**可演進性**是一種高階特性，一種保護所有其他架構特性的架構包裝。

我們在本書指出，演進式架構有一種副作用是它可以保護重要的架構特性。我們將探討**持續型架構**背後的理念：建立沒有最終狀態的架構，並且讓它隨著不斷演進的軟體開發生態系統而演進，同時內建保護機制，保護重要的架構特性。我們不打算全面性地定義軟體架構，你可以在這裡找到許多其他的定義（*https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf*）。我們的重點是擴展當前的定義，將**時間**和**變更**當成一級架構元素加入定義。

下面是我們對演進式架構的定義：

> **演進式架構可支援橫跨多個維度且引導式的漸進變更。**

漸進變更

漸進變更指的是軟體架構的兩個層面：團隊如何逐步建構軟體，以及他們如何部署軟體。

在開發過程中，允許小規模漸進變更的架構比較容易演進，因為它可讓開發人員以較小的範圍進行變更。在部署時，漸進變更代表商業功能的模組化與解耦程度，以及它們如何對映到架構。我們舉個例子。

假設 PenultimateWidgets 是一家銷售 widget 的大型公司，它有一個使用微服務架構及現代工程方法製作的目錄網頁。這個網頁可讓用戶對各種 widget 進行星級評分。PenultimateWidgets 公司的其他服務也需要評分（客戶服務代表、貨運商評估等等），它們共享星級評分服務。有一天，星級評分團隊打算在既有的版本上發表一個新版本，提供半星評分，這是一種規模不大，但很重要的升級。需要使用評分的其他服務不一定要遷移到新版本，但可以視情況逐步遷移。PenultimateWidgets 有部分的 DevOps 方法不僅會監視服務的架構，也會監視服務之間的路由。當運維團隊發現在某個時間範圍內，沒有人被路由到特定服務時，他們就會自動將該服務從生態系統中移除。

這是個架構級漸進變更案例：只要有其他的服務需要使用原始的服務，原始的服務就可以和新服務一起運行。團隊可以在有空的時候（或者視需求）遷移到新的行為，舊的版本則會被自動回收。

要成功實施漸進變更，你必須協調一些持續交付的做法。你不一定要在所有情況下都實踐所有方法，但它們在實務上經常同時發生。我們會在第 3 章討論如何實現漸進變更。

引導式變更

架構師都希望他們選出重要的特性之後，可以**引導**架構的變更，來保護那些特性。為此，我們借用進化計算（evolutionary computing）的**適應度函數**（*fitness function*）概念。適應度函數是一種目標函數，用來算出“潛在的設計方案距離既定的目標有多遠”。在進化式計算中，適應度函數的用途是計算演算法可不可以隨著時間而改善，換句話說，當每一版的演算法產生時，適應度函數都會根據演算法的設計者如何定義“適應”，來算出每個版本的“適應”程度。

在演進式架構中，我們也有類似的目標——隨著架構的演進，我們需要透過一些機制來評估變更如何影響架構的重要特性，並防止那些特性隨著時間而劣化。適應度函數是確保架構不被各種不受歡迎的方式變更的機制，這種機制包括指標、測試程式和其他驗證工具。架構師指定他們想要保護的架構特性之後，還要定義一或多個適應度函數來保護那個（些）特性。

長久以來，架構有一些部分經常被視為治理活動，直到最近，架構師才接受透過架構來實現變更的概念。架構適應度函數可讓架構師根據組織的需求和商務功能進行決策，同時奠定一個基礎，將那些決策明確化，並且讓它們可被測試。演進式架構並不是一種無限制的、不負責任的軟體開發方法，它是一種可在“快速變更的需求”與“圍繞著系統和架構特性的嚴格需求”之間取得平衡的方法。適應度函數驅動架構的決策，可在指引架構的同時，容許人們做必要的變更，來支援不斷變化的業務和技術環境。

我們使用**適應度函數**來建立架構的演進方針；第 2 章將詳細討論它們。

多個架構維度

> 世上沒有獨立的系統。這個世界是一個連續體。系統的邊界取決於討論的目的。
>
> —Donella H. Meadows

古希臘的物理學家漸漸學會以固定點來分析宇宙，並在古典力學中達到了頂峰（*http://farside.ph.utexas.edu/teaching/301/lectures/node3.html*）。然而，在 20 世紀初，更精確的儀器和更複雜的現象讓相對論的定義越來越完善。科學家們意識到，他們以前認為是孤立的現象，事實上是相互作用的。自 1990 年代以來，開明的架構師漸漸將軟體架構視為多維的，持續交付進一步擴展這一個觀點，將運維加入其中。然而，軟體架構師在乎的通常是**技術**架構，而它只是軟體專案的一個維度。如果架構師想要建立可演進的架構，就必須考慮系統裡面會受變更影響的每一個部分。正如物理學所言，萬物都是相對的，架構師應該知道軟體專案有很多維度。

為了建立可演進的軟體系統，架構師不能只考慮技術架構。例如，如果專案包含一個關聯式資料庫，那麼資料庫實體之間的結構和關係也會隨著時間而演進。類似的情況，架構師不希望他建構的系統在演進的過程中暴露安全漏洞。這些都是架構**維度**的例子——架構的各個部分通常以正交的方式組合在一起。有些維度符合所謂的**架構關注點**（上述的“-ility”清單），但其實**維度**比較廣泛，它也包括傳統上在技術架構範圍之外的東西。每一個專案都有規劃架構如何演進時必須考慮的維度，以下是可能影響現代軟體架構的可演進性的維度：

技術

架構的實作部分：框架、程式庫與實作語言。

資料

資料庫的 schema（架構定義）、資料表布局、優化計畫等等。這種架構通常是資料庫管理員負責處理的。

安全性

定義安全策略、方針，以及協助找出缺陷的工具。

運維 / 系統

關注如何將架構對映到既有的物理或虛擬基礎：伺服器、機器叢集、斷路器、雲端資源等等。

上述的每一個層面都形成一個架構**維度**——將支援特定層面的因素刻意分出來的部分。架構維度的概念包含傳統的架構特徵（“-ility”），以及任何其他協助建構軟體的因素。維度形成我們想要在問題不斷發展和周遭世界不斷變化的過程中保留的架構觀點。

目前有各式各樣的技術可在概念上劃分架構。例如，4 + 1 架構 View Model（*https://en.wikipedia.org/wiki/4%2B1_architectural_view_model*）把重心放在各種角色的不同層面上，將生態系統劃分成**邏輯**、**開發**、**程序**與**實體觀點**，已被納入 IEEE 軟體架構定義。在著名的書籍 Software Systems Architecture（*https://www.viewpoints-and-perspectives.info/home*）中，作者提出一系列關於軟體架構的觀點，涵蓋了更大規模的角色群。Simon Brown 的 C4（*http://www.codingthearchitecture.com/*）符號可劃分各種關注點，以協助組織概念。相較之下，本書的目的不是將各種維度分門別類，而是從既有的專案中，辨識既有的維度。從實用的角度來看，無論特別重要的關注點屬於哪一類，架構師都必須保護那個維度。不同的專案有不同的關注點，因此每個專案都有獨特的維度組合。上述的技術提供了實用的觀點，很適合用於新專案，但你必須根據現實的情況來處理既有的專案。

藉著站在架構維度的角度來思考，架構師可以評估每一個重要的維度對變更的反應，來分析架構的可演進性。隨著系統與彼此競爭的關注點（可擴展性、安全性、分布、交易等等）越來越緊密地糾纏在一起，架構師也必須追蹤更多維度。為了建立可演進的系統，架構師必須考慮系統如何橫跨所有重要的維度進行演進。

專案的整個架構範圍是由軟體需求及其他維度組成的。我們可以在架構與生態系統一起隨著時間演進時，使用適應度函數來保護特性，如圖 1-3 所示。

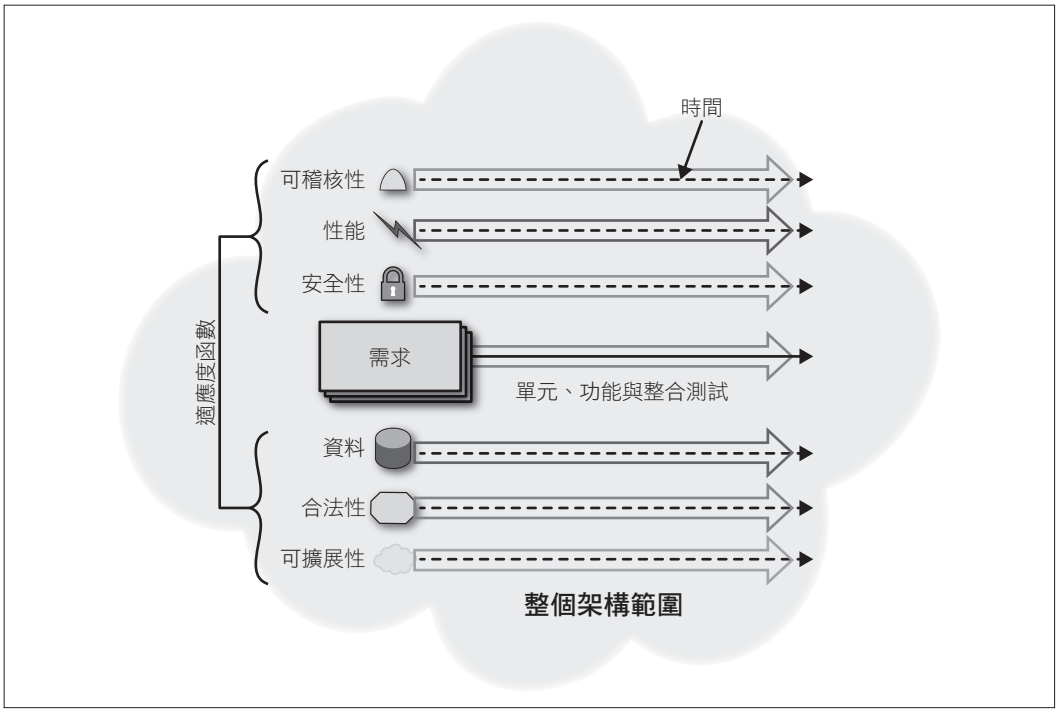

圖 1-3　架構是由需求與其他維度組成的，每一個維度都被適應度函數保護

在圖 1-3 中，架構師已經確定**可稽核性**、**資料**、**安全性**、**性能**、**合法性**和**可擴展性**是這個應用程式很重要的架構特性了。你可以隨著商業需求的變化，不斷演變各種架構特性，並且利用適應度函數來保護完整性。

雖然本書的作者強調整體架構觀點的重要性，但是我們也認識到，可演進的架構與技術架構模式及其相關主題（例如耦合和內聚）有很大的關係。我們將在第 4 章討論技術架構耦合如何影響可演進性，並在第 5 章討論資料耦合的影響。

這裡的耦合指的不是只有軟體專案的結構元素而已，最近有許多軟體公司發現，團隊結構對軟體架構等人們意想不到的事項有深遠的影響。我們會討論軟體的所有耦合層面，但是團隊的影響會在早期出現，而且經常出現，所以我們也必須討論它。

Conway 定律

1968 年 4 月，Melvin Conway 向 Harvard Business Review 提出一篇論文，題目是“How Do Committees Invent?”（*http://www.melconway.com/research/committees.html*）。在這篇論文中，Conway 提到，社會結構，尤其是人與人之間的溝通途徑，必然會影響最終的產品設計概念。

正如同 Conway 所述，在設計的早期階段，我們要對系統進行了高層次的瞭解，以理解如何將職責區域劃分成不同的模式。團隊分解問題的方式，將會影響他們日後的選擇。他提出了日後稱為 *Conway 定律*的看法：

> 設計系統的組織…產生的設計都被限制了，它們只是那些組織的溝通結構的複製品。
>
> —Melvin Conway

正如同 Conway 所言，當技術人員將問題分解成更小的區塊來委託工作時，就開始引入協調問題了。乍看之下，許多組織的溝通結構或嚴格的層次結構可以解決這種協調問題，但這些結構也經常產生不靈活的解決方案。例如，在分層架構中，團隊是根據技術功能（用戶介面、商業邏輯等）組成的，因此在解決直向跨越各個階層的共同問題時，協調開銷將會提升。曾經在初創企業工作，後來加入大型跨國公司的人，可能經歷過前者靈活、適應性強的文化，以及後者僵化的溝通結構。Conway 定律有一個很好的例子，在更改兩項服務之間的合約時，如果其中一個團隊為了更改它負責的服務，必須協調另一個團隊，並且讓他們同意一起努力的話，那項工作可能極為困難。

Conway 在他的論文中警告軟體架構師不僅要注意軟體的架構和設計，還要注意團隊之間工作的委託、分配和協調。

許多組織都是用職能技術來組織團隊的。常見的例子包括：

前端開發者

> 具有特定使用者介面（UI）技術（例如 HTML、行動裝置、桌機）的專業技能的團隊。

後端開發者

> 具備建構後端服務技能的團隊，有時候是 API 層。

料庫開發者

具備建構儲存體及邏輯服務技術的團隊。

在含有職能 silo[譯註]的組織中，管理層是根據人力資源部分的喜好來組織團隊的，沒有考慮工程效率。雖然各個團隊都有他們擅長設計的部分（例如建立螢幕畫面、加入後端 API 或服務，或開發新的儲存機制），但是為了發表新的商業功能，三個團隊都必須參與功能的構建。組織經常針對眼前的工作效率而優化團隊，而不是根據更抽象的商務戰略目標，在面臨進度壓力時更是如此。團隊經常把目標放在提供可彼此搭配或無法搭配的組件上，而不是提供端對端的功能價值。

在這種組織劃分機制之下，公司要花更多時間才能完成需要三個團隊一起完成的功能，因為每一個團隊都是在不同時間處理他們的元件。例如，考慮更新目錄網頁這種常見的商業變更，這種變更必須修改 UI、商業規則和資料庫 schema，如果每個團隊都在他們自己的 silo 內工作，他們就必須協調時間的規劃，延長完成功能所需的時間。這個例子可充分說明團隊結構如何影響架構及演進能力。

正如同 Conway 在他的論文中指出的，**每當有人將工作委託出去，而且有人的調查、詢問的範圍被縮小時，有效的替代方案的種類也會減少**。換句話說，如果有人想要改變一樣東西，但那樣東西是別人擁有的，那個人就很難改變它。軟體架構師應該注意如何劃分和指派工作，好讓架構的目標符合團隊的結構。

許多製作微服務等架構的公司都是根據服務邊界來組織團隊的，而不是劃分孤立的技術架構。我們在 ThoughtWorks Technology Radar（*https://www.thoughtworks.com/radar*）中，將這種情況稱為 Inverse Conway Maneuver（逆 Conway 調度）（*https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver*）。以這種方式組織團隊是很好的做法，因為團隊結構最好可以反映問題的大小和範圍，它也會影響無數的軟體開發維度。例如，在建立微服務架構時，為了建立類似這種服務的團隊架構，公司通常會橫跨職責 silo，讓團隊的成員涵蓋商務及技術架構的每一個層面。

設法讓團隊的結構符合目標架構，如此一來，團隊將更容易實現它。

譯註 silo 直譯為穀倉，商學經常用它來形容公司或組織中，因為科技的限制或政治因素，只對內而不對外溝通，造成資訊不對稱或不流通的現象。

介紹 PenultimateWidgets，以及它們的反 Conway 時刻

本書使用 PenultimateWidgets 公司來示範，它是一家大型的線上 Widget 銷售商，目前的規模倒數第二。這間公司正逐步更新大部分的 IT 基礎設施，他們有一些舊系統想要保留一段時間，以及一些需要採取更多迭代方法的新戰略系統。在各章中，我們會重點介紹 PenultimateWidgets 的許多問題，以及為了滿足這些需求而開發的解決方案。

他們的架構師發現一件與軟體開發團隊有關的事情。舊的單體應用程式使用分層架構、分離展示、商業邏輯、持久保存和運維。他們的團隊與這些功能相應：所有的 UI 開發人員都坐在一起，開發人員和資料庫管理員都有自己的 silo，運維則外包給第三方。

當開發人員開始處理新的架構元素時（也就是具有粒狀服務的微服務架構），協調成本就開始急劇上升。由於服務是圍繞著各個領域（例如 *CustomerCheckout*）而不是技術架構建立的，因此若要對單一領域進行更改，就要進行大量的跨 silo 協調。

於是，PenultimateWidgets 運用 Inverse Conway Maneuver，建立符合服務範圍（或權限，purview）的跨職能團隊：每個服務團隊都是由服務負責人、幾位開發人員、一位商業分析師、一位資料庫管理員（DBA）、一個品保（QA）人員和一個運維人員組成的。

本書會在許多地方展示這種團隊造成的影響，並舉例說明它帶來的後果。

為何要演進？

關於演進式架構，人們經常問一個與名稱有關的問題：為什麼稱它為**演進式**架構，而不是其他名稱？其他的名稱包括**漸進式**、**持續式**、**敏捷式**、**反應式**和**應急式**（僅舉幾個例子）。原因是它們都偏離主題了，我們定義的演進式架構包括兩項主要的特性：漸進與引導。

持續式、**敏捷式**與**應急式**都帶有“隨著時間演進”的概念，這的確是演進式架構的重要特性，但是這些術語都沒有明確地指出架構**如何**變更，或架構的最終狀態應該是什麼。雖然所有術語都暗指一個不斷變化的環境，但是它們都沒有涵蓋架構應該長怎樣。定義中的**引導**反映了我們想要實現的架構，也就是我們的終極目標。

我們比較喜歡**演進性**而不是**可適應**（*adaptable*）的原因是，我們想要看到的是經歷演進變更的架構，而不是經歷修補與調整，意外地越來越難理解的複雜架構。**適應**意味著無論解決方案優雅與否或壽命長短，都要設法讓某些事物可以動作。為了建立真的能夠演進的架構，架構師必須支援真正的變更，而不是臨時、應急的解決方案。回到我們的生物學比喻，**演進**是讓一個系統符合我們的目標，並且在不斷變化的環境中生存的過程。或許各個系統都有各自的適應性，但是作為架構師，我們要關心的，該是整體的可演進系統。

小結

演進式架構有三個主要成分：漸進變更、適應度函數，和適度耦合。在本書的其餘部分，我們將分別討論這些因素，再將它們組合起來，建立和維護可支援不斷變更的架構，並且解決在過程中遇到的問題。