對本書的讚譽

「這本書補足了在整個技術堆疊中構建微服務和分析架構決策細微差別中所缺少的部分。在這本書中，你可以得到在構建分散式系統時可以做出的架構決策清單，以及與每個決策相關的優缺點。這本書是每個構建現代分散式系統的架構師所必須具備的。」

—*Aleksandar Serafimoski*，*Thoughtworks* 首席顧問

「對於熱衷於架構的技術專家來說，這是一本必讀的書。本書對模式的表達清晰扼要。」

—*Vanya Seth*，*Thoughtworks India* 技術主管

「無論你是位有抱負的架構師，還是經驗豐富帶領團隊的架構師，都不要手忙腳亂，這本書將具體引導你在如何建立企業應用程式和微服務的過程中獲得成功。」

—*Venkat Subramaniam* 博士，獲獎作家和 *Agile Developer, Inc.* 創始人

「《軟體架構：困難部分》在讓你瞭解如何拆開高度耦合的系統並重新構建它們上，為讀者提供了寶貴的見解、實踐和真實世界的範例。藉由獲得有效的權衡分析技能，你將開始做出更好的架構決策。」

—*Joost van Wenen*，*Infuze Consulting* 聯合創始人兼執行合夥人

「我喜歡閱讀這本關於分散式架構全方位的著作！它完美結合了基本概念的扎實討論與大量實用的建議。」

—*David Kloet*，獨立軟體架構師

「分割一個大泥球並不是件容易的事。從程式碼開始到資料，這本書將幫助你明白應該提取的服務和應該保持在一起的服務。」

—*Rubén Díaz-Martínez*，*Codesai* 的軟體開發人員

「本書將為你提供理論背景和實用框架，協助解答在現代軟體架構上所面臨的最困難問題。」

—*James Lewis*，*Thoughtworks* 技術總監

第一章

當沒有「最佳做法」時，會發生什麼？

為什麼像軟體架構師這樣的技術專家會出席會議或寫書？因為他們發現了俗稱的「最佳做法」，這個術語被過度濫用，以致於那些說這個術語的人越來越受到強烈反對。不管是哪種說法，當技術專家對一個一般性的問題發現新的解決方案，並想把它傳播給更多人的時候，他們就會寫書。

但是，對於那些沒有好解決方案的大量問題會發生什麼？在軟體架構中存在所有類型的問題，這些問題卻沒有好的一般性解決方案，而是呈現一組凌亂的權衡，與一組（幾乎）同樣凌亂的集合。

軟體開發人員在網上搜尋當前問題的解決方案方面建立了出色的技能。例如，如果他們需要弄清楚如何在他們的環境中配置一個特定工具，專業地使用 Google 就會找到答案。

但對架構師來說，情況並非如此。

對於架構師來說，許多問題都帶來了獨特的挑戰，因為它們混淆了你組織的確切環境和情況 —— 有人遇到了這種情況並將它發布在博客或貼在 Stack Overflow 上的機會有多少？

架構師可能想知道，與框架、API 等技術主題相比，為什麼關於架構的書籍這麼少。架構師很少會經歷常見的問題，但卻在新情況下為了做決定而不斷地掙扎。對於架構師來說，每一個問題都是一片雪花。在許多情況下，問題不僅僅是出現在特定的組織內，而對整個世界都是新的。對於這些問題，沒有任何的書籍或是會議討論它！

架構師不應該一直為他們的問題尋找靈丹妙藥；現在這些靈丹妙藥和 1986 年 Fred Brooks 創造這個術語時一樣罕見：

> 無論是技術還是管理方面的技術，都沒有任何一個發展能夠承諾十年內在生產力、可靠性和簡單性方面有平均一個數量級 [十倍] 的改善。
>
> —出自 Fred Brooks 的「No Silver Bullet」

因為幾乎每一個問題都會帶來新的挑戰，架構師的真正工作在於他們能夠客觀地決定和評估一個最終決定任何一方的權衡，以盡可能好地解決它。作者沒有談論「最佳解決方案」（在本書或現實世界中），因為「最佳」意味著架構師已經設法將設計中所有可能的競爭因素最大化。相反地，我們提出以下半開玩笑的建議：

不要試圖在軟體架構中找到最好的設計；相反地，要爭取最不差的權衡組合。

通常情況下，架構師可以建立的最好設計是最不差的權衡集合——沒有一個單一架構特徵能超過它單獨的那樣，但所有競爭架構特徵的平衡能促進專案成功。

這就引出了一個問題，「架構師如何才能**找到**最不差的權衡組合（並有效地記錄它們）？」本書主要是關於決策的，使架構師在面對新情況時能做出更好的決策。

為什麼是「困難部分」？

為什麼我們把這本書叫做《**軟體架構：困難部分**》？實際上，書名中的「hard」一字有雙重含義。首先，*hard* 意味著**困難**，而架構師們不斷面臨著字面上（及象徵性地）以前沒有人所面臨過的困難問題，涉及許多具有長期影響的技術決策，這些決策必須發生在人際和政治環境之上。

第二，*hard* 意味著**硬**——就如同在分離的**硬體**和**軟體**中，**硬**的部分應該改變得比較少，因為它為**軟**的部分提供了基礎。同樣地，架構師也討論了**架構**和**設計**之間的區別，前者是結構性的，而後者更容易改變。因此，在本書中，我們討論的是架構的基礎部分。

軟體架構的定義本身已經為它的參與人員提供了許多時間的非生產性對話。一個最受歡迎半開玩笑的定義是「軟體架構是那些以後很難改變的**東西**」，這些**東西**就是我們這本書的內容。

提供關於軟體架構的永恆建議

軟體開發的生態系統不斷地、混亂地轉變和成長。幾年前風靡一時的話題要麼被生態系統所包含並消失，要麼被不同 / 更好的東西所取代。例如，10 年前，大型企業的主流架構樣式是協作驅動、服務導向的架構。現在，幾乎沒有人再採用這種架構的樣式了（原因我們將在後面揭曉）；目前許多分散式系統所青睞的樣式是微服務，這種轉變是如何以及為什麼發生的？

當架構師注視一個特定的樣式（尤其是歷史性的），他們必須考慮導致這架構成為主流的約束因素。當時，許多公司合併成為企業，伴隨著這種過渡隨之而來的是整合問題。此外，對於大公司來說，開源並不是一個可行的選項（通常是出於政治而非技術原因）。因此，架構師強調共用資源以及集中協作作為一種解決方案。

然而，在這幾年中，開源和 Linux 成為可行的替代方案，使作業系統在商業上免費。然而，真正的轉捩點發生在 Linux 隨著像是 Puppet 和 Chef 等工具的出現而在操作上變得免費，這些工具允許開發團隊以編程方式啟動他們的環境，作為自動建構的一部分。一旦有了這種能力，它就透過微服務和迅速出現的容器基礎架構，以及像是 Kubernetes 等協作工具而促進了一場架構革命。

這說明了軟體開發的生態系統以完全無法預期的方式擴展和演進。一種新的能力導致了另一種能力，這又不預期地創造了新的能力。隨著時間的推移，這個生態系統一次一個地完全取代了自己。

這對一般技術以及特別是軟體架構書籍的作者提出了一個古老的問題——我們怎樣才能寫出不會立即變老的東西？

在本書中，我們並不關注技術或其他實作的細節。相反地，我們關注的是架構師如何做決策，以及在遇到新情況時如何客觀地權衡取捨。我們使用當時的情景和例子來提供細節和上下文，但基本原則著重於面對新問題時的權衡分析和決策。

資料在架構中的重要性

> 「資料是一種珍貴的東西，並會比系統本身更持久。」
>
> —Tim Berners-Lee

對於架構界許多人來說，資料就是一切。每個建立任何系統的企業都必須處理資料，因為它的壽命往往比系統或架構還長得多，需要勤奮的思考和設計。然而，資料架構師建

立緊密耦合系統的許多本能在現代分散式架構中產生了衝突。例如，架構師和 DBA 必須確保商務資料在整體式系統被拆散後仍能存在，並且無論架構如何波動，商務仍可以從它的資料中獲得價值。

有人說，**資料是一個公司最重要的資產**。企業希望從他們擁有的資料中提取價值，並在決策中找到新的方法部署資料。現在企業的每一部分都是由資料驅動的，從服務現有客戶到獲得新客戶、增加客戶保留率、改善產品、預測銷售和其他的趨勢。這種對資料的依賴意味著所有的軟體架構都在為資料服務，確保正確的資料可以被企業的各部門使用。

幾十年前，當分散式系統剛開始流行時，作者就建立了許多分散式系統，但是現代微服務的決策似乎更為困難，我們想弄清楚為什麼。我們最終意識到，在分散式架構的早期，我們大多仍然是堅持資料在一個單一的關聯式資料庫中。然而，在微服務中，以及從「Domain-Driven Design」（*https://oreil.ly/bW8CH*）中對有界上下文的哲學堅持，作為限制實作細節耦合範圍的一種方式，資料已經和交易性一起移到了架構關注點上。現代架構的許多困難部分都來自於資料和架構關注點之間的緊張關係，我們會在第一部分和第二部分解開這個問題。

我們在不同章節中涵蓋的一個重要區別，是在**操作資料**與**分析資料**之間的分割。

操作資料

用於商務運作的資料，包括銷售、交易資料、庫存等等。這些資料是公司運行的依據，如果這些資料受到干擾，組織就無法長期運作。這種類型的資料被定義為**線上交易處理**（OLTP），它通常涉及在資料庫中插入、更新和刪除資料。

分析資料

資料科學家和其他商務分析師用於預測、趨勢分析及其他商務智慧的資料。這種資料通常不是交易性的，而且通常不是關聯性的——它可能位於圖形資料庫或快照中，其格式與原來的交易形式不同。這些資料對日常運作並不重要，但是對長期策略方向和決策都很重要。

我們在全書中涵蓋操作和分析資料的影響。

架構決策記錄

記錄架構決策的最有效方法之一是透過**架構決策記錄**（ADR（*https://adr.github.io*））。ADR 最早是由 Michael Nygard 在一篇博客貼文（*https://oreil.ly/yDcU2*）中宣揚的，後來在 Thoughtworks Technology Radar（*https://oreil.ly/0nwHw*）中被標記為「採用」。一個 ADR 由描述一個特定架構決策的簡短本文檔案組成（通常為一到兩頁）。雖然 ADR 也可以用明文編寫，但它們通常是用某種本文檔案格式編寫的，像是 AsciiDoc（*http://asciidoc.org*）或 Markdown（*https://www.markdownguide.org*）。或者，ADR 也可以用 wiki 頁面模板來編寫。在我們的前一本書《*Fundamentals of Software Architecture*》（O'Reilly）中，我們用了整整一章來討論 ADR，繁體中文版《軟體架構原理｜工程方法》由碁峰資訊出版。

我們將利用 ADR 作為記錄本書中各種架構決策的一種方式。對於每個架構決策，我們將使用以下的 ADR 格式，並假設每個 ADR 都得到正式的認可。

ADR：一個包含架構決策的短名詞片語。

上下文
在 ADR 的部分，我們將增加簡短一兩句話的問題描述，並列出替代的解決方案。

決策
在這部分我們將陳述架構決策，並提供決策的詳細理由。

結果
在 ADR 的這部分中，我們將描述應用決策後的任何結果，並討論所考慮的權衡。

本書中建立的所有架構決策記錄清單可以在附錄 B 中找到。

記錄一個決策對架構師很重要，但管理決策的正確使用是一個單獨的主題。幸運的是，現代工程實作允許透過使用架構適應度函數自動化許多常見的管理問題。

架構適應度函數

一旦架構師確定了組件之間的關係並將其編入設計中，他們如何確保實作者會遵守這設計？更廣泛地說，如果架構師不是實作者，他們如何能確保他們定義的設計原則會成為現實？

這些問題屬於**架構管理**的範疇，它適用於對軟體開發的一個或多個面向的任何有組織的監督。由於本書主要涵蓋架構結構，我們在很多地方都介紹了如何經由適應度函數使設計和品質的原則自動化。

軟體開發隨著時間的推移慢慢演變以適應獨特的工程做法。在軟體開發的早期，無論是大的（如瀑布式開發過程）還是小的（專案上的整合實作），製造業的比喻通常被應用於軟體實作。在 1990 年代初期，由 Kent Beck 和 C3 專案的其他工程師領導的對軟體開發工程實作的重新思考，被稱為 eXtreme 編程（XP），說明了增量回饋和自動化作為軟體開發生產力的關鍵推動者的重要性。在 2000 年代初期，同樣的教訓被應用於軟體開發和作業的路口，催生了 DevOps 的新角色，並使許多以前手動作業的雜事自動化。就像以前一樣，自動化使團隊走得更快，因為他們不必擔心沒有好的回饋事情就會中斷。因此，**自動化**和**回饋**已經成為有效軟體開發的核心原則。

考慮導致自動化突破的環境和情況，在持續整合之前的年代，大多數軟體專案都包括一個漫長的整合階段。每個開發者都被期望與其他人在某種程度上隔離的工作，然後在最後將所有的程式碼整合到一個整合階段。這種做法的痕跡仍然存在於版本控制工具中，強迫分支並阻止持續整合。毫不奇怪地，專案規模與整合階段的痛苦之間存在著強烈的關聯。透過開創性的持續整合，XP 團隊說明了迅速、持續回饋的價值。

DevOps 革命遵循類似的歷程。隨著 Linux 和其他開源軟體對企業來說變得「足夠好」，再加上允許虛擬機器編程定義（最終）工具的出現，作業人員意識到他們可以將機器定義和許多其他重複性工作自動化。

在這兩種情況下，技術和洞察力的進步導致由昂貴角色處理的重複性工作的自動化——這描述了大多數組織中架構管理的目前狀態。例如，如果一個架構師選擇了一個特定的架構樣式或通訊媒介，他們如何確保開發者正確地實作它？手動完成時，架構師會執行程式碼審查或召開架構審查委員會來評估管理狀態。然而，就像在作業中手動配置電腦一樣，重要的細節很容易落入膚淺的審查中。

使用適應度函數

在 2017 年的《*Building Evolutionary Architectures*》（O'Reilly）一書中，作者（Neal Ford、Rebecca Parsons 和 Patrick Kua）定義了**架構適應度函數**的概念：對某些架構特徵或架構特徵的組合進行客觀完整性評估的機制。以下是對這定義的逐點分解：

任何機制

架構師可以使用多種工具來實作適應度函數；我們將在書中展示大量的例子。例如，有專門的測試庫來測試架構結構，架構師可以使用監視器來測試作業架構的特性，像是性能或可擴展性等，並且用混沌工程框架測試可靠性和彈性。

客觀的完整性評估

自動化管理的一個關鍵推動因素是對架構特徵的客觀定義。例如，架構師不能指定他們想要一個「高性能」網站；他們必須提供一個可以用測試、監控或其他適應度函數測量的目標值。

架構師必須注意**複合架構的特徵**——那些不能客觀測量但實際上是其他可測量事物的組合。例如，「敏捷性」是不可測量的，但如果架構師開始把廣義的**敏捷性**拉開，目標是讓團隊能夠迅速、自信地回應生態系統或領域的變化。因此，架構師可以找到有助於敏捷性的可測量特徵：可部署性、可測試性、週期時間等等。通常，缺乏測量架構特徵的能力表示定義太模糊。如果架構師朝著可測量的性質努力，就可以允許他們將適應度函數應用自動化。

某些架構特徵或架構特徵的組合

這個特徵描述了適應度函數的兩個範圍：

原子的

這些適應度函數單獨的處理單一架構特徵。例如，檢查程式碼庫中組件週期的適應度函數在範圍上是原子的。

整體的

整體的適應度函數驗證了架構特徵的組合。架構特徵的一個複雜特點是它們有時會展現出與其他架構特徵協同作用。例如，如果架構師想提高安全性，它將很有可能會影響性能。同樣地，可擴展性和彈性有時也是不一致——支援大量的併發使用者會使處理突發事件變得更加困難。整體的適應度函數行使了互鎖架構特徵的組合，以確保組合效果不會對架構產生負面影響。

架構師實作適應度函數，以圍繞架構特徵的意外變化建立保護。在敏捷軟體開發領域，開發者實作單元、功能以及使用者驗收測試，以驗證**領域**設計的不同維度。然而，直到現在，還沒有類似的機制來驗證設計中的**架構特徵**部分。事實上，適應度函數和單元測試之間的分離為架構師提供了一個很好的範圍指引。適應度函數驗證架構特徵，而不是領域標準；單元測試則正好相反。因此，架構師可以透過詢問以下問題來決定是否需要適應度函數或單元測試：「執行這個測試是否需要任何的領域知識？」如果答案是「是」，那麼單元 / 功能 / 使用者驗收測試是合適的；如果答案是「否」，則需要一個適應度函數。

例如，當架構師談論**彈性**時，指的是應用程式能夠承受使用者突然爆發的能力。請注意，架構師不需要知道有關領域的任何細節——這可以是電子商務網站、線上遊戲、或其他的東西。因此，**彈性**是一個架構上的問題，並且在適應度函數的範圍內。另一方

面，如果架構師想要驗證一個郵寄位址的正確部分，則可以透過傳統測試來涵蓋。當然，這種分離並不是純粹的二分法——有些適應度函數會觸及到領域，反之亦然，但不同的目標提供了一個很好的方法來從精神上將它們分開。

這裡有幾個使這個概念不那麼抽象的例子。

架構師的一個共同目標是在程式碼庫中保持良好的內部結構完整性。然而，在許多平台上，惡意的力量與架構師的良好意圖作對。例如，當在任何流行的 Java 或 .NET 開發環境中編碼時，只要開發者引用了尚未導入的類別，IDE 就會協助地顯示一個對話框，詢問開發者是否想自動導入這個引用。這種情況經常發生，以致於大多數程式師都養成了像反射動作一樣打發掉自動導入對話框的習慣。

然而，在彼此之間任意地導入類別或組件會對模組化造成災難。例如，圖 1-1 說明了架構師希望避免的一種特別有害的反模式。

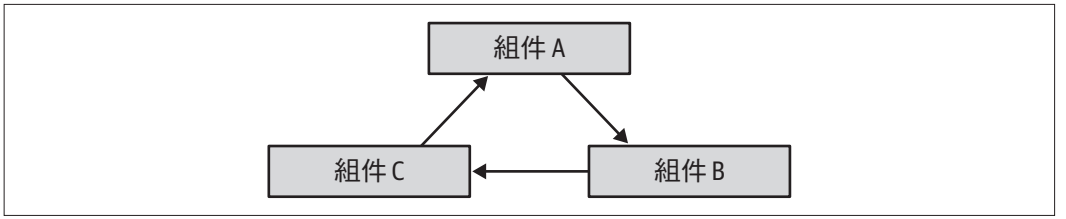

圖 1-1　組件之間的循環相依性

在這種反模式中，每個組件都引用了其他組件中的某些東西。有像這樣的組件網路會損害模組化，因為開發者如果不把其他組件也帶上，就不能重用一個組件。當然，如果其他組件被耦合到另外的組件上，架構就會越來越傾向於大泥球（*https://oreil.ly/usx7p*）的反模式。架構師如何管理這種行為，而又不至於持續盯著那些會亂開槍的開發者的肩膀？程式碼審查雖有幫助，但因在開發週期中發生的太晚了而無效。如果架構師允許開發團隊在程式碼審查前的一週內肆意導入程式碼庫，則程式碼庫中已經發生了嚴重的損害。

這個問題的解決方法是編寫一個適應度函數來避免組件循環，如範例 1-1 所示。

範例 1-1　檢測組件循環的適應度函數

```
public class CycleTest {
    private JDepend jdepend;

    @BeforeEach
```

```
        void init() {
              jdepend = new JDepend();
              jdepend.addDirectory("/path/to/project/persistence/classes");
              jdepend.addDirectory("/path/to/project/web/classes");
              jdepend.addDirectory("/path/to/project/thirdpartyjars");
        }

        @Test
        void testAllPackages() {
              Collection packages = jdepend.analyze();
              assertEquals("Cycles exist", false, jdepend.containsCycles());
        }
    }
```

在程式碼中，架構師用度量工具 JDepend（*https://oreil.ly/ozzzk*）來檢查套件之間的相依性。這工具了解 Java 套件的結構，如果存在任何循環，則測試失敗。架構師可以將這個測試連接到專案的持續建構中，不再擔心亂開槍的開發者意外引入循環。這是一個很好的例子，說明適應度函數可以守護軟體開發重要而不緊急的做法：這對架構師來說是一個重要的問題，但對日常的編碼影響不大。

範例 1-1 顯示了一個非常低層次、以程式碼為中心的適應度函數。許多流行的程式碼保健工具（如 SonarQube（*https://www.sonarqube.org*））以完整立即可以使用的方式實作了許多常見的適應度函數。但是，架構師可能還想驗證架構的宏觀結構以及微觀結構。當設計一個像圖 1-2 中的分層架構時，架構師定義了各層以確保分離關注點。

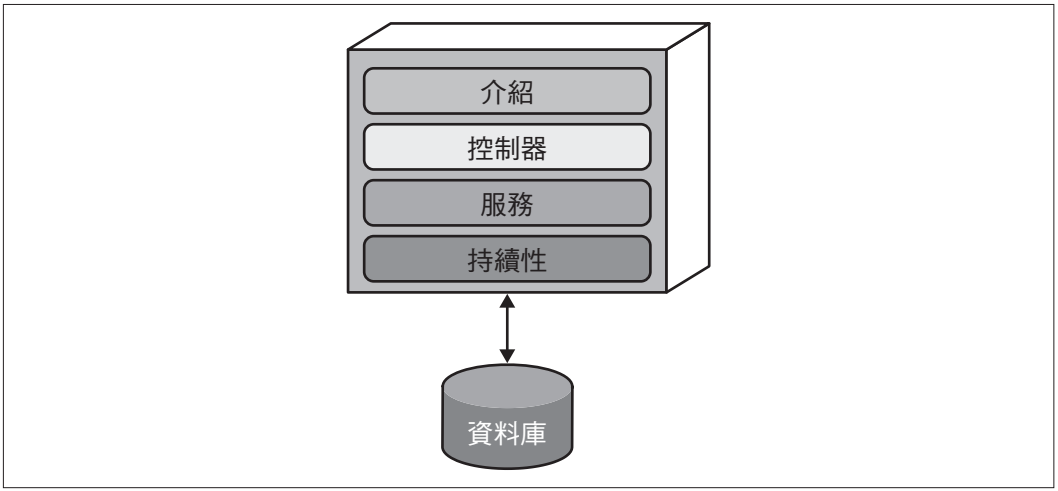

圖 1-2　傳統的分層架構

但是，架構師如何能確保開發者會尊重這些層呢？一些開發者可能不理解這些模式的重要性，而另一些開發者則可能會因為一些像是性能等壓倒一切的局部考慮，而採用「寬恕好於允許」的態度。但是，允許實作者侵蝕架構的原因會傷害架構的長期健康。

ArchUnit（*https://www.archunit.org*）允許架構師透過適應度函數解決這個問題，如範例 1-2 所示。

範例 1-2　管理層次的 ArchUnit 適應度函數

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

在範例 1-2 中，架構師定義了各層間的理想關係，並編寫了一個驗證適應度函數來管理它。這允許架構師在圖表和其他資訊工件之外建立架構原則，並持續地驗證它們。

在 .NET 領域有類似的工具 NetArchTest（*https://oreil.ly/EMXpv*），允許對這個平台做類似的測試。範例 1-3 中顯示一個 C# 中的層次驗證。

範例 1-3　層次相依性的 NetArchTest

```
// 在介紹中的類別不應該直接引用儲存庫
var result = Types.InCurrentDomain()
    .That()
    .ResideInNamespace("NetArchTest.SampleLibrary.Presentation")
    .ShouldNot()
    .HaveDependencyOn("NetArchTest.SampleLibrary.Data")
    .GetResult()
    .IsSuccessful;
```

在這領域持續出現複雜程度越來越高的工具，我們將繼續強調其中的許多技術，因為我們在說明許多解決方案的同時也說明了適應度函數。

為一個適應度函數找到一個客觀的結果是至關重要的。然而，**客觀**並不意味著**靜態**。一些適應度函數會有非上下文的回傳值，像是真 / 假或一個如性能閾值的數值。但是，其他適應度函數（被視為是**動態的**）會根據某些上下文回傳一個值。例如，當測量**可擴展性**時，架構師測量併發使用者的數量，且一般也測量每個使用者的性能。通常，架

構師設計系統時，隨著使用者數量的增加，每個使用者的性能會稍微下降，但不會急遽降低。因此，對於這些系統，架構師在設計性能適應度函數時要考慮到併發使用者的數量。只要對架構特徵的測量是客觀的，架構師就可以對它進行測試。

雖然大多數適應度函數應該自動化，並持續運行，但某些函數必須是手動的。手動的適應度函數需要一個人來處理驗證。例如，對於有敏感法律資訊的系統，律師可能需要審查關鍵部分的改變以確保合法性，這不能自動化。大多數部署管道支援手動階段，允許團隊容納手動適應度函數。理想情況下，這些函數在合理的可能範圍內頻繁地執行——不執行的驗證就不能驗證任何事。團隊可以因為需要（很少）或作為持續整合工作流的一部分（最常見）執行適應度函數。為了完全實現像是適應度函數驗證的好處，它們應該持續執行。

連續性很重要，正如這個使用適應度函數的企業級管理例子所說明的。考慮以下場景：當企業使用的某個開發框架或資料庫被發現有零日漏洞時，公司會怎麼做？如果它像大多數公司一樣，安全專家會偵查專案以找到有問題的框架版本並確保它已經被更新，但這個過程很少是自動化的，而是依賴許多手動步驟。這不是一個抽象的問題，這個確切的場景影響了「Equifax 資料外洩」中描述的一個主要金融機構。就像前面描述的架構管理一樣，手動流程容易出錯，並讓細節被忽略。

Equifax 資料外洩

2017 年 9 月 7 日，美國一家主要的信用評等機構 Equifax 宣佈發生了資料外洩。最終，這問題被追溯到 Java 生態系統中流行的 Struts 網路框架的駭客攻擊漏洞（Apache Struts vCVE-2017-5638）。這機構在 2017 年 3 月 7 日發表一份聲明，宣佈這個漏洞並發布補丁。國土安全部第二天聯繫了 Equifax 和類似的公司，警告他們這個問題，他們在 2017 年 3 月 15 日進行掃描，並沒有發現所有受影響的系統。因此，直到 2017 年 7 月 29 日，當 Equifax 的安全專家確認導致資料外洩的駭客行為時，關鍵補丁才被應用於許多舊的系統上。

想像在一個替代的世界中，每個專案都執行一個部署管道，且安全團隊在每個團隊的部署管道中都有一個「插槽」，他們可以在那裡部署適應度函數。大多數的時候，這些將是對安全防護的普通檢查，像是防止開發者在資料庫中存儲密碼和類似的常規管理工作。然而，當出現零日漏洞時，在各處都有相同機制允許安全團隊在每個專案中插入測試，以檢查框架和版本編號；如果發現危險的版本，則構建失敗並通知安全團隊。團隊

配置部署管道會被生態系統的任何改變喚醒：程式碼、資料庫模式、部署配置和適應度函數。這允許企業普遍地自動化重要管理工作。

適應度函數為架構師提供了許多好處，其中最重要的是有機會再次進行編碼！架構師一個普遍的抱怨問題是他們不再寫很多程式碼了——但適應度函數往往就是程式碼！透過建立一個可執行的架構規範，任何人都可以藉由執行專案構建來隨時驗證，架構師必須很好地了解系統和它持續的演進，這與在專案發展過程中跟上專案程式碼的核心目標重疊。

無論適應度函數多麼強大，架構師應該避免過度使用它們。架構師不應該形成一個小集團，並撤回到象牙塔中去建立一個難以置信的複雜、互鎖的一組適應度函數，這只會使開發者和團隊感到沮喪。相反地，這是為架構師在軟體專案上建立一個**重要**但不**緊急**原則可執行清單的方法。許多專案都淹沒在緊迫性中，讓一些重要的原則從旁邊溜走。這是技術債務的常見原因：「我們知道這很不好，但我們以後會回來修正它」——而這個以後卻永遠不會來。透過將關於程式碼品質、結構和其他防衰減措施的規則編入持續執行的適應度函數，架構師建立了一個開發者不能跳過的品質查核表。

幾年前，Atul Gawande（Picador）的優秀著作《*The Checklist Manifesto*》強調了像外科醫生、飛行員等專業人士對查核表的使用，以及那些經常使用（有時是法律強制使用）查核表作為他們工作的一部分領域。這不是因為他們不了解自己的工作或特別健忘；當專業人士反復執行相同的工作，當不小心跳過時，就很容易自欺欺人，而查核表可以防止這種情況。適應度函數代表了由架構師定義的重要原則查核表，並作為構建的一部分執行，以確保開發者不會意外地（或因為像是進度壓力等外部力量而有目的地）跳過它們。

當有機會說明管理架構解決方案以及初始設計時，我們會在整本書中使用適應度函數。

架構與設計：保持定義簡單性

架構師不斷奮鬥的領域是保持**架構**和**設計**為獨立但相關的活動。雖然我們不想涉入關於這區別永無止境的爭論中，但我們在本書中努力堅定地站在這範圍中**架構**的這一邊，有以下幾個原因。

首先，架構師必須了解底層架構的原則以做出有效的決策。例如，在架構師實作細節分層之前，**同步**與**異步**通訊之間的決策有許多權衡。在《*Fundamentals of Software Architecture*》一書中，作者創造了軟體架構的第二定律：**為什麼**比**如何做**更重要。雖然

最終架構師必須了解如何實作解決方案，但他們首先必須了解為什麼一種選擇會比另一種選擇有更好的權衡。

第二，透過專注在架構概念上，我們可以避免這些概念的大量實作。架構師可以透過各種方式實作異步通訊；我們專注於為什麼架構師會選擇異步通訊，並將實作細節留在另一個地方。

第三，如果我們開始在我們顯示所有種類選項的實作路上，這將會是曾經有過最長的一本書。專注於架構原則使我們能夠盡可能地保持事物的通用性。

為了使主題盡可能地扎根於架構，我們對關鍵概念使用了盡可能簡單的定義。例如，架構中的**耦合**就可以寫滿整本書（而且已經寫了）。為此，我們使用以下簡單的、近乎簡化的定義：

服務

在口語上，**服務**是作為獨立可執行文件部署功能的凝聚集合。我們對於服務討論的大多數概念廣泛適用於分散式架構，且特別是對微服務架構。

我們在第 2 章定義的術語中，**服務**是架構量子的一部分，其中包括服務和其他量子之間的靜態和動態耦合的進一步定義。

耦合

如果一個工件（包括服務）的改變可能需要另一個工件的改變以維持適當的功能，那麼這兩個工件就是耦合的。

組件

應用程式的一個架構建構區塊，用於做某種商務或基礎架構的功能，通常是透過套件結構（Java）、命名空間（C#），或某種目錄結構中的原始程式碼檔案的實體分組表現出來。例如，訂單歷史這個組件可以透過位於命名空間 `app.business.order.history` 中的一組類別檔案實作。

同步通訊

如果呼叫者在繼續進行之前必須等待回應，那麼這兩個工件的通訊為同步。

異步通訊

如果呼叫者在繼續之前不用等待回應，那麼這兩個工件的通訊為異步。可以選擇的是，當請求已經完成時，接收者可以透過單獨的通道通知呼叫者。

協作的協調

如果一個工作流程包括一個主要責任是協調工作流程的服務，那它就是協作的。

編排的協調

當一個工作流程缺少協作器時，它就是編排的；相反地，工作流程中的服務分享工作流程的協調責任。

原子性

如果一個工作流程所有部分始終都保持一致的狀態，那麼這工作流程就是**原子的**；相反地，則由**最終一致性**的範圍表示，這涵蓋在第 6 章中。

合約

我們廣泛地使用**合約**這個術語來定義兩個軟體部分之間的介面，其中可能包括方法或函數呼叫、整合架構遠端呼叫、相依性等。只要兩個軟體連接的地方，都涉及到合約。

軟體架構本質上是抽象的：除了沒有兩個是完全相同的以外，我們無法知道平台、技術、商務軟體以及讀者可能會有的其他令人眼花繚亂的獨特組合。我們涵蓋了許多抽象的想法，但必須以一些實作細節為基礎，使它們具體化。為此，我們需要一個問題來說明架構概念——這將我們引向了 Sysops Squad。

Sysops Squad 傳奇的介紹

傳奇

一個關於英雄成就的長篇故事。

—牛津英語詞典

我們在本書中討論了一些傳奇，包括字面的和比喻的。架構師共同選擇了**傳奇**這個術語來描述分散式架構中的交易行為（我們將在第 12 章中詳細介紹）。然而，關於架構的討論往往會變得抽象，尤其是在考慮像架構的困難部分這樣抽象的問題時。為了幫助解決這個問題，並為我們討論的解決方案提供一些真實世界的上下文，我們開啟了一個關於 *Sysops Squad* 的字面傳奇。

我們在每一章中使用 Sysops Squad 傳奇來說明本書所描述的技術和權衡。雖然許多關於軟體架構的書籍涵蓋了新的開發工作，但許多真實世界的問題仍然存在於現有系統中。因此，我們的故事用這裡所強調現有的 Sysops Squad 架構開始。

Penultimate Electronics 是一家大型電子巨頭，在全國有許多零售店。當客戶購買電腦、電視、音響和其他電子設備時，他們可以選擇購買一個支援計畫。當發生問題時，面向客戶的技術專家（Sysops Squad）會到客戶的住所（或辦公室）解決電子設備的問題。

Sysops Squad 單據應用程式的四個主要使用者如下：

管理者

管理者維護系統的內部使用者，包括專家名單和他們對應的技能集合、位置和可用性。管理者還管理使用這系統客戶的所有帳單處理，並維護靜態參考資料（例如支援的產品、系統中的名 - 值對等）。

客戶

客戶註冊 Sysops Squad 服務並維護他們的客戶個人資料、支援合約和帳單資訊。客戶在系統中輸入問題單，並在工作完成後填寫調查表。

Sysops Squad 專家

專家被分配到問題單，並根據問題單修復問題。他們還與知識庫互動以搜尋客戶問題的解決方案，並輸入有關維修的注釋。

經理

經理保持持續追蹤問題單的作業，並接收關於整個 Sysops Squad 問題單系統的操作和分析報告。

非單據工作流程

非單據工作流程包括管理者、經理和客戶執行的與問題單據無關的行動。這些工作流程概述如下：

1. Sysops Squad 專家是經由輸入他們位置、可用性和技能的管理者，在系統中添加和維護。
2. 客戶在 Sysops Squad 系統上註冊，並根據他們購買的產品擁有多個支援計畫。
3. 客戶根據他的個人資料中包含的信用卡資訊按月自動開具帳單。客戶可以經由系統查看帳單歷史和敘述。
4. 管理者要求並接收各種操作和分析報告，包括財務報告、專家績效報告和單據報告等。

單據工作流程

當客戶在系統輸入問題單後，單據工作流程就開始了，且當客戶在維修完成後完成調查時結束。這個工作流程概述如下：

1. 已購買支援計畫的客戶藉由使用 Sysops Squad 網站輸入問題單。
2. 一旦一個問題單進入系統，系統就會根據技能、目前位置、服務區域和可用性決定哪個 Sysops Squad 專家最適合這項工作。
3. 一旦被分配，問題單會上傳到 Sysops Squad 專家行動裝置上的專用定制行動應用程式。專家也會收到簡訊通知，告訴他們分配到一張新的問題單。
4. 客戶透過 SMS 簡訊或電子郵件（根據他們個人資料的偏好）得到通知專家正在來的路上。
5. 專家使用他們手機上的定制行動應用程式取得問題單資訊和位置。Sysops Squad 專家還可以透過行動應用程式存取知識庫，找出過去為解決這種問題所採取的措施。
6. 一旦專家修復了這個問題，他們將問題單標記為「完成」。然後，Sysops Squad 專家可以將有關問題及修復的資訊加入知識庫。
7. 在系統收到問題單完成的通知後，它向客戶發送一封附有調查鏈結的電子郵件，然後由客戶填寫。
8. 系統從客戶那裡接收到完成的調查表並記錄調查資訊。

糟糕的場景

最近 Sysops Squad 問題單應用程式的情況並不好。目前故障的單據系統是多年前開發的大型整體式應用程式。客戶抱怨說，因為問題單遺失，所以顧問們從來沒來過，而且也經常是錯誤的顧問來維修他們一無所知的東西。客戶也抱怨，系統並不總是可以輸入新的問題單。

在這個龐大的整體中，更改也很困難和危險。每當進行更改時，通常會花費很多時間，而且常會出現其他問題。由於可靠性問題，Sysops Squad 系統經常「當機」或崩潰，當問題被確認並重新啟動應用程式的 5 分鐘到 2 小時不等時間內，所有應用程式的功能都無法使用。

如果不趕快處理，Penultimate Electronics 將被迫放棄利潤豐厚的支援合約業務線，並解雇所有 Sysops Squad 管理者、專家、經理和 IT 開發人員——包括架構師。

Sysops Squad 的架構組件

Sysops Squad 應用程式的整體式系統處理單據管理、操作報告、客戶註冊和帳單，以及一般性的管理功能，像是使用者維護、登錄、專家技能和個人資料維護等。圖 1-3 和對應的表 1-1 說明並描述現有整體式應用程式的組件（`ss.` 部件命名空間指定了 Sysops Squad 應用程式的上下文）。

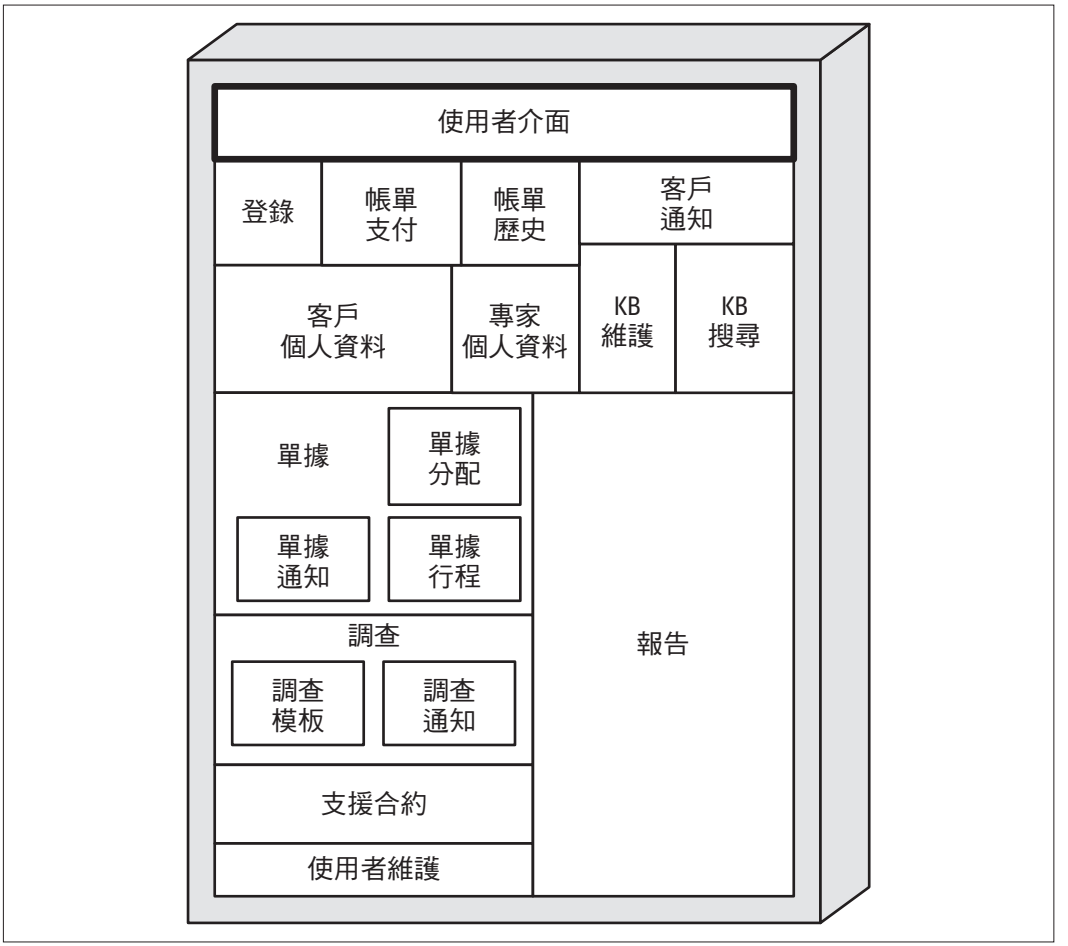

圖 1-3　現有 Sysops Squad 應用程式中的組件

表 1-1　現有的 Sysops Squad 組件

組件	命名空間	責任
登錄	`ss.login`	內部使用者和客戶的登錄和安全邏輯
帳單支付	`ss.billing.payment`	客戶按月帳單和客戶信用卡資訊
帳單歷史	`ss.billing.history`	付款歷史和以前的帳單敘述
客戶通知	`ss.customer.notification`	通知客戶帳單、一般資訊
客戶個人資料	`ss.customer.profile`	維護客戶個人資料、客戶註冊
專家個人資料	`ss.expert.profile`	維護專家個人資料（姓名、地點、技能等）
KB 維護	`ss.kb.maintenance`	維護和查看知識庫中的項目
KB 搜尋	`ss.kb.search`	用於搜尋知識庫的查詢引擎
報告	`ss.reporting`	所有報告（專家、單據、財務）
單據	`ss.ticket`	單據創建、維護、完成、常用程式碼
單據分配	`ss.ticket.assign`	尋找專家並分配單據
單據通知	`ss.ticket.notify`	通知客戶專家已經在路上了
單據行程	`ss.ticket.route`	將單據發送到專家的行動裝置應用程式
支援合約	`ss.supportcontract`	對客戶、計畫中的產品提供支援合約
調查	`ss.survey`	維護調查、捕捉和記錄調查結果
調查通知	`ss.survey.notify`	發送調查郵件給客戶
調查模板	`ss.survey.templates`	根據服務類型維護各種調查
使用者維護	`ss.users`	維護內部使用者和角色

這些組件將在後續章節中處理將應用程式分解成分散式架構時，用來說明各種技術和權衡。

Sysops Squad 的資料模型

Sysops Squad 應用程式以及它在表 1-1 中列出的各種組件，在資料庫中使用單一的模式來託管它所有的表格和相關的資料庫程式碼。這資料庫用於保存客戶、使用者、合約、帳單、付款、知識庫和客戶調查；這些表格的清單列於表 1-2 中，ER 模型則說明於圖 1-4。

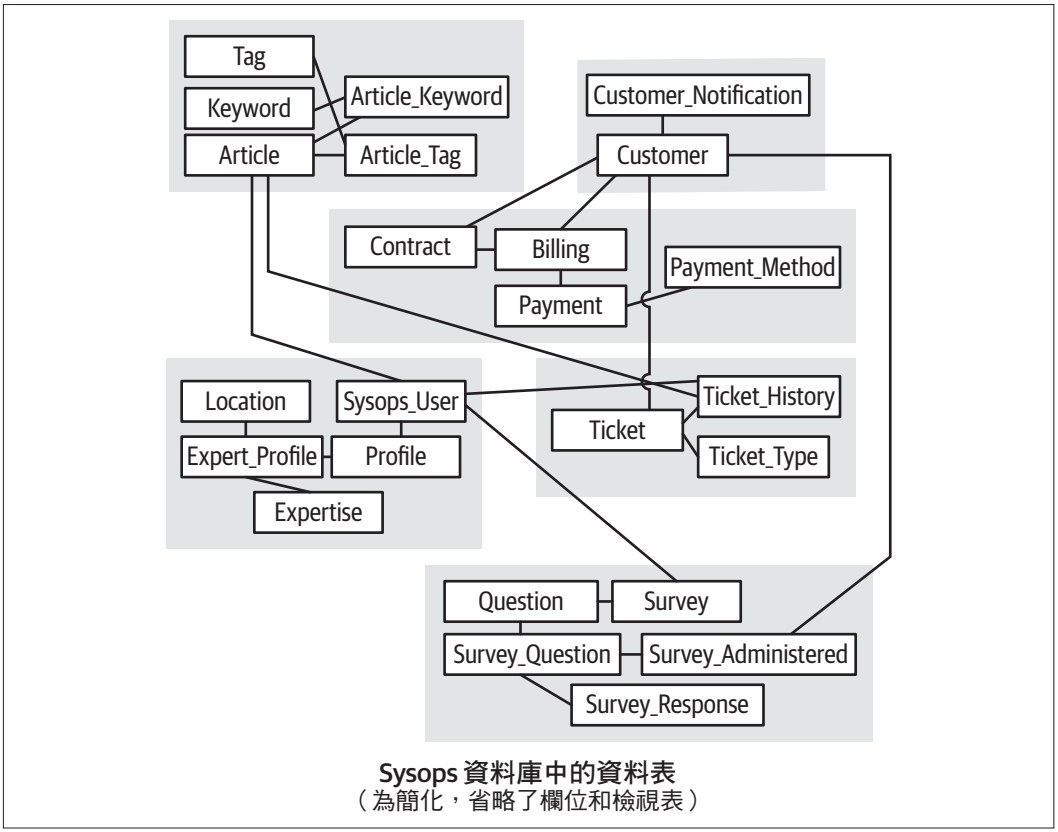

圖 1-4　現有 Sysops Squad 應用程式中的資料模型

表 1-2　現有 Sysops Squad 資料庫資料表

表格	責任
Customer	需要 Sysops 支援客戶的實體
Customer_Notification	客戶的通知偏好
Survey	支援後客戶滿意度調查
Question	調查中的問題
Survey_Question	分配給調查的一個問題
Survey_Administered	分配給客戶的調查問題
Survey_Response	客戶對調查的回應
Billing	支援合約的帳單資訊

表格	責任
Contract	產品與 Sysops 之間的支援合約
Payment_Method	對支援的付款方式
Payment	帳單付款處理
Sysops_User	Sysops 中的各種使用者
Profile	Sysops 使用者的個人資料
Expert_Profile	專家個人資料
Expertise	Sysops 內的各種專業知識
Location	專家服務的地點
Article	知識庫內的文章
Tag	文章的標籤
Keyword	文章的關鍵字
Article_Tag	與文章相關的標籤
Article_Keyword	關鍵字和文章連接表
Ticket	客戶提出的支援單
Ticket_Type	不同類型的單據
Ticket_History	支援單的歷史

Sysops 資料模型是標準第三種正常形式的資料模型，只有少數存儲過程或觸發器。但是，存有相當多主要提供給報告組件使用的檢視表。當架構團隊試圖分解應用程式並轉成分散式架構時，它將不得不與資料庫團隊合作以完成資料庫層級的工作。這種資料庫資料表和檢視表的設置將用於本書中所討論的各種技術和權衡，以完成資料庫的分割工作。