序

我一直對谷歌（Google）如何做事的細節著迷不已。我曾向我的谷歌員工（Googler）朋友詢問公司內部實際運作方式。他們如何管理如此龐大的單一程式碼儲存庫（monolithic code repository）而不會陷入困境？成千上萬的工程師如何在數千個專案上成功協作？他們如何保持系統的品質？

與前谷歌員工一起工作只會增加我的好奇心。如果你曾經和一位前谷歌工程師（或有時被稱為 Xoogler）合作過，你無疑聽過『在谷歌，我們 ......』這句話。從谷歌進入其他公司似乎是一次令人震驚的經歷，至少從工程方面來看是這樣。據這位局外人所知，考慮到公司的規模和人們讚美的頻率，谷歌的系統和編寫程式碼的流程一定是世界上最好的。

在《Google 的軟體工程之道》中，有一組谷歌員工（以及一些前谷歌員工）為我們提供了一份關於 Google 軟體工程中許多實施方法、工具甚至文化元素的詳細藍圖。人們很容易過度關注 Google 為支援程式碼編寫而建構的神奇工具，而本書提供了許多關於這些工具的細節。但它不僅僅是簡單地描述工具，還為我們提供了 Google 團隊遵循的理念和流程。這些內容適用於各種情況，無論你是否擁有足夠的規模和工具。令人高興的是，有幾章深入探討了自動化測試的各個方面，這個話題在我們的行業中仍然遇到太多的阻力。

技術的偉大之處在於，做一事從來都不只有一種方法。取而代之的是，我們都必須根據團隊的處境和情況做出一系列的權衡。我們可以從開放原始碼中廉價地獲得什麼？我們的團隊可以建構什麼？對我們的規模來說，什麼是有意義的支援？當我詢問我的 Googler 朋友時，我想聽聽這個處於極端規模的世界：資源豐富，既有人才又有錢財，對正在建構的軟體有很高的要求。這些軼事給了我一些想法，讓我有了一些原本不會考慮的選項。

碁峰 www.gotop.com.tw

透過這本書，我們把這些選項寫下來供大家閱讀。當然，Google 是一家獨特的公司，如果認為運行你的軟體工程組織之正確方法是精確地複製他們的公式，那就太愚蠢了。從實際應用來講，本書將為你提供有關如何完成工作的想法，以及許多資訊，你可以用來支持你採用最佳實施方法（如測試、知識共享和建構協作團隊）的論點。

你可能永遠不需要自己構建 Google，甚至可能不想在你的組織中達到他們所應用的同樣技術。但是，如果你不熟悉 Google 所開發出的實施方法，就會錯過一個關於軟體工程的觀點，這個觀點來自二十多年來成千上萬的工程師在軟體上的協同工作。這些知識太有價值了，不容輕忽。

— Camille Fournier（卡米爾 • 富尼耶）

《The Manager's Path》（經理人之道）作者

碁峰 www.gotop.com.tw

前言

這本書的標題是「Google 的軟體工程之道」。我們所說的軟體工程究竟是什麼意思？「軟體工程」（software engineering）與「程式設計」（programming）或「計算機科學」（computer science）的區別是什麼？為什麼 Google 會有獨特的觀點被添加到過去 50 年來所寫之軟體工程文獻的語料庫中？

在我們的行業中，「程式設計」和「軟體工程」這兩個術語已經被交替使用了相當長的時間，儘管此二者都有不同的重點和不同的含義。大學生往往會想要學習「計算機科學」，成為「程式員」（programmers）以獲得撰寫程式碼（writing code）的工作。

然而，「軟體工程」聽起來更嚴肅，好像它意味著，應用一些理論知識來建構真實而精確的東西。機械工程師、土木工程師、航空工程師和其他工程學科的人都在從事工程工作。他們都在現實世界中工作，並利用他們的理論知識的應用來創造真實的東西。軟體工程師也創造了「真實的東西」 儘管它不如其他工程師創造的東西那麼有形。

與那些更成熟的工程專業不同，當前的軟體工程理論或實施方法並不那麼嚴謹。航空工程師必須遵守嚴謹的指導方針和實施方法，因為他們的計算錯誤會造成真正的損害；總體而言，程式設計傳統上沒有遵循這種嚴格的實施方法。但是，隨著軟體越來越融入我們的生活，我們必須採用並依賴更嚴格的工程方法。我們希望這本書能幫助其他人看到一條通往更可靠之軟體實施方法的道路。

隨著時間推移的程式設計

我們認為，「軟體工程」不僅包括編寫程式碼的行為，還包括一個組織隨著時間的推移用於建構和維護該程式碼的所有工具和流程。一個軟體組織可以導入哪些實施方法，進而得以長期保持其程式碼的價值？工程師如何才能使一個程式碼基底（codebase）更具

碁峰 www.gotop.com.tw

可持續性（sustainable）並使軟體工程學科本身更加嚴謹？我們對這些問題沒有基本答案，但我們希望 Google 在過去二十年的集體經驗能夠為這些答案指出可能的道路。

我們在本書中分享的一個關鍵見解是，軟體工程可以被認為是「隨著時間的推移而整合的程式設計」。我們可以在我們的程式碼中導入哪些實施方法，使其能夠在生命週期內（從概念到導入，從導入到維護，再從維護到棄用）對必要的變更做出反應，進而使其具有可持續性？

本書強調了我們認為軟體組織在設計、建構和編寫程式碼時應該牢記的三項基本原則：

時間與變化
: 程式碼需要如何適應其生命週期

規模和成長
: 組織需要如何適應其發展

權衡和成本
: 組織如何根據時間、變化、規模和成長的經驗做出決策

在本書的所有章節中，我們都試圖與這些主題聯繫起來，並指出這些原則如何影響工程實施方法，並使其具有可持續性。（完整的討論，請參閱第 1 章。）

Google 的觀點

Google 對可持續軟體生態系統的成長和演變有著獨特的觀點，這源於我們的規模和壽命。我們希望，隨著你的組織的發展和採用更可持續的做法，我們所吸取的教訓，將是有用的。

我們將本書中的主題分為 Google 軟體工程領域的三個主要方面：

- 文化
- 過程
- 工具

Google 的文化是獨一無二的，但我們在發展自己的工程文化方面所吸取的教訓是廣泛適用的。我們關於文化的章節（第二部分）強調了軟體開發企業的集體性質，軟體開發是團隊的努力，適當的文化原則對於組織成長和保持健康至關重要。

碁峰
www.gotop.com.tw

大多數軟體工程師都熟悉流程章節（第三部分）中概述的技術，但 Google 之大型和長期的程式碼基底（codebase）為開發最佳實施方法（best practices）提供了更完整的壓力測試（stress test）。在這些章節中，我們試圖強調，隨著時間的推移和規模的擴大，我們發現到的一些可行的方法，同時也指出了我們尚未找到令人滿意答案的領域。

最後，我們的工具章節（第四部分）說明了，我們如何利用在工具基礎架構方面的投資，在程式碼基底的成長和老化過程中為其帶來好處。在某些情況下，這些工具是 Google 特有的，儘管我們會在適用的情況下指出開源或第三方替代方案。我們希望這些基本見解，適用於大多數工程組織。

本書中概述的文化、流程和工具，描述了典型的軟體工程師希望在工作中學到的教訓。Google 當然不會壟斷好的建議，我們在這裡介紹經驗，並不是為了決定你的組織應該做什麼。本書的內容是我們的觀點，但我們希望你會發現它很有用，無論是直接採用這些教訓，還是在針對你自己的問題領域，考慮你自己的實施方法時，將它們視為一個起點。

本書也不是為了說教。Google 本身仍然無法完美應用書中這些概念。我們從失敗中吸取了教訓：我們仍然會犯錯誤，實作不完美的解決方案，並且需要反覆改進。然而，Google 之工程組織的龐大規模，確保了每個問題都有多樣化的解決方案。我們希望本書包含該群體中的佼佼者。

本書不涵蓋…

本書並不打算涵蓋軟體設計，這門學科需要自己的書（並且已經存在很多內容）。儘管本書中有一些程式碼用於說明，但這些原則是語言中立的，並且這些章節中幾乎沒有實際的「程式設計」建議。因此，本書內容沒有涵蓋軟體開發中的許多重要問題：專案管理、API 設計、安全強化、國際化、用戶介面框架或其他特定於語言的問題。本書遺漏這些內容並不意味著它們不重要。相反，我們選擇不涵蓋它們，因為我們無法提供它們應有的待遇。我們試圖讓本書的討論更多的是關於工程，而不是關於程式設計。

最後感言

本書是所有貢獻者的心血，希望你能夠接受它：因為它是一個窗口，讓你得以看到一個大型軟體工程組織如何建構其產品。我們還希望，這有助於推動我們的行業，採用更具前瞻性和可持續性的做法。最重要的是，我們進一步希望你喜歡閱讀它，並能將其中一些教訓用於你自己的問題。

— Tom Manshreck（湯姆・曼什雷克）

碁峰 www.gotop.com.tw

何謂軟體工程？

作者：Titus Winters（泰特斯・溫特斯）

編輯：Tom Manshreck（湯姆・曼斯瑞克）

> 沒有什麼是建立在石頭上的；一切都是建立在砂子上，但我們必須把沙子當作石頭。
>
> ——Jorge Luis Borges（豪爾赫・路易士・博爾赫斯）

我們在程式設計（programming）與軟體工程（software engineering）之間看到三個關鍵區別：時間、規模以及權衡。在軟體專案中，工程師需要更加注意時間的流逝以及最終的需求變更。在軟體工程組織中，我們需要更加關注規模（scale）和效率（efficiency），無論是對我們生產的軟體，還是對生產軟體的組織。最後，做為軟體工程師，我們被要求做出更複雜的決策，其結果風險更大，因為這往往是基於對時間和成長不精確的估計。

在 Google 內部，我們有時會說：「軟體工程是隨著時間的推移不斷整合的程式設計。」程式設計無疑是軟體工程的重要部分：畢竟，程式設計是你首先產生新軟體的方式。如果你接受這種區別，那麼顯然，我們可能需要在程式設計任務（開發）和軟體工程任務（開發、修改、維護）之間進行劃分。時間的加入，為程式設計添加了一個重要的維度。正如立方體不是正方形，距離不是速度。軟體工程不是程式設計。

觀察時間對程式之影響的一種方法是，思考這樣一個問題：「你的程式碼的預期壽命（life span）[1] 是多久？」這個問題的各個合理答案大約可以相差到 10 萬倍。程式碼存

1 我們不是指「執行壽命」（execution lifetime），而是指「維護壽命」（maintenance lifetime）——程式碼將繼續被建構、執行和維護多長時間？

碁峰 www.gotop.com.tw

活幾分鐘是合理的，存活幾十年的程式碼一樣是合理的。一般來說，位於該光譜短端（short end）的程式碼將不受時間的影響。對於效用只有一個小時的程式，你不太可能需要去適應（adapt）其背後之程式庫（library）、作業系統（OS）、硬體或程式語言的新版本。這些短命（short-lived）的系統實際上「只是」一個程式設計問題，就像是立方體在一個維度上被壓縮得很厲害時就成為正方形一樣。當我們把時間擴大到允許更長的壽命時，變化就會變得更為重要。在十年或更多的時間裡，大多數程式的依賴關係（dependency），無論是隱性的還是顯性的，都可能發生變化。這種認識是我們區分軟體工程和程式設計的根源。

這種區別是我們所謂的軟體的「可持續性」（sustainability）之核心。如果在你的軟體的預期壽命內，你能夠出於技術（technical）或業務（business）的原因，對任何有價值的變化做出反應，則你的專案是「可持續的」（sustainable）。重要的是，我們只看重能力——你可能選擇不進行某項升級，無論是因為缺乏價值，還是因為其他優先事項。[2] 當你完全無法對底層技術或產品方向做出反應時，你就是在下一個高風險的賭注，希望這種變化永遠不會成為關鍵。對於短期的專案來說，這可能是一個安全的賭注。但是在幾十年的時間裡，這可能並不安全。[3]

另一種看待軟體工程的方法是考慮規模（scale）。涉及多少人？隨著時間的推移，他們在開發和維護中扮演什麼角色？程式設計任務通常是個人創造的行為，但是軟體工程任務是團隊的工作。早期定義軟體工程的嘗試為此觀點提供了一個很好的定義：『多人開發的多版本程式』（The multiperson development of multiversion programs）[4]。這說明了軟體工程與程式設計的區別是時間與人員的區別。團隊協作帶來了新的問題，但也提供了比任何單個程式員更多的潛力來營運有價值的系統。

團隊組織、專案組成以及軟體專案的政策和實施方法都主導著軟體工程複雜性這一方面。這些問題是規模固有的：隨著組織的成長和專案的擴展，它在生產軟體方面是否會變得更有效率？我們的開發流程是隨著我們的成長變得更有效率，還是我們的版本控制政策和測試策略會按比例增加我們的成本？從軟體工程早期開始，人們就一直在討論關於溝通（communication）和人員擴展（human scaling）的規模問題（scale issues），這

2 這也許是一個合理的技術債務（technical debt）定義：「應該」做但尚未做的事情——我們的程式碼與我們希望的程式碼之間的差距。

3 另外還要考慮到一個問題，就是我們是否提前知道，這一個專案是否會長期存在。

4 關於這句話的原始出處有一些疑問；共識似乎是，它最初是由 Brian Randell 或 Margaret Hamilton 表述的，但它可能完全是由 Dave Parnas 編造的。它的常見引文是「軟體工程技術：北約科學委員會主辦之會議的報告」1969 年 10 月 27 日至 31 日，義大利羅馬，北約科學事務部。（"Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee," Rome, Italy, 27–31 Oct. 1969, Brussels, Scientific Affairs Division, NATO.）

碁峰 www.gotop.com.tw

可追朔到《人月神話》（Mythical Man Month）[5]。這樣的規模問題通常是政策問題，也是軟體可持續發展的根本問題：重複進行我們需要做的事情，成本有多少？

我們也可以說，軟體工程與程式設計的不同之處在於需要做出決策的複雜性及其利害關係。在軟體工程中，我們經常被迫在多條前進道路之間做權衡，有時風險很大，有時價值指標（value metrics）並不完善。軟體工程師或軟體工程領導者的工作目標是實現組織、產品和開發工作流程的可持續發性（sustainability）和擴展成本（scaling costs）管理。考慮到這些問題，評估你的權衡並做出合理的決定。我們有時可能會推延維護方面的變更，或甚至會接受那些擴展性不好的政策，因為我們知道，我們將需要重新審視這些決策。這些選擇應該明確和清楚地說明推延的成本。

在軟體工程中，很少有一個萬能的（one-size-fits-all）解決方案，本書也是如此。對於「軟體的使用壽命是多久」，合理的答案是 100,000（十萬）的因數；對於「你的組織中有多少工程師」的範圍，可能是 10,000（一萬）的因數；而對於「你的專案有多少運算資源可用」，誰知道是多少呢；Google 的經驗可能與你的經驗不符。在本書中，我們的目的是介紹我們在軟體建構和維護中發現的行之有效的方法，我們預期這些軟體能夠持續數十年，擁有數以萬計的工程師，以及跨越世界的運算資源。我們發現，在這種規模下，大多數必要的實施方法也適用於較小規模的工作：考慮到這是一份關於工程生態系統（engineering ecosystem）的報告，我們認為隨著規模的擴大，這種生態系統可能會有不錯的效果。在某些地方，超大的規模是有固定的成本，因此如果不用支付額外的費用，我們會很樂意的。我們將此視為一個警訊。希望如果你的組織發展到需要擔心這些成本的程度時，你可以找到一個更好的答案。

在探討團隊合作、文化、政策和工具之前，讓我們先詳細說明一下時間、規模和權衡等主題。

時間與變化

當新手學習程式設計時，所產生的程式碼之壽命通常以小時或日數為單位。程式設計作業和練習往往只寫一次，幾乎沒有重構（refactor），當然也沒有長期維護。這些程式在最初製作後往往不會再被重建（rebuilt）或執行。這在教學環境中並不奇怪。也許在中等或專上教育中，我們會看到小組專題課程（team project course）或動手論文（hands-on thesis）。如果是這樣的話，這可能是學生的程式碼之壽命，時間將超過一個月左右的

5 Frederick P. Brooks Jr. 所著之《人月神話：軟體專案管理之道》（波士頓：艾迪生韋斯利，1995）。

碁峰 www.gotop.com.tw

唯一一次。這些開發者可能需要重構一些程式碼，也許是為了回應不斷變化的需求，但他們不太可能被要求去處理更廣泛的環境變化。

我們還發現在常見的行業環境（industry settings）中也可以找到短命程式碼的開發者。行動應用程式（mobile apps）的壽命通常相當短，[6] 而且無論好壞，完全重寫是相對常見的。工程師在創業公司早期階段理所當然會選擇將重點放在立即目標，而非長期投資上：該公司的壽命可能不夠長，無法從回報緩慢的基礎架構投資中獲得收益。一個連續創業（serial startup）開發者擁有 10 年的開發經驗是非常合理的，但很少人或根本沒人能夠具有維護任何預期會存在超過一兩年之軟體的經驗。

光譜的另一端，一些成功專案的有效壽命是無限的：我們無法合理的預測 Google Search、Linux kernel 或 Apache HTTP Server 等專案的終點。對於大多數的 Google 專案，我們必須假定它們可以無限期地存在著，因為我們無法預測何時不需要升級依賴項目、程式語言的版本…等等。隨著壽命的延長，這些長期存在的專案最終會有不同於程式設計作業（programming assignments）或創業開發（startup development）的感覺。

請參考圖 1-1，它呈現了位於此「預期壽命」（expected life span）光譜兩端的兩個軟體專案。對於從事預期壽命為數小時之任務的程式員，什麼類型的維護是合理的？ 也就是說，如果你正在編寫一支僅執行一次的 Python 命令稿，你的 OS（作業系統）出現了新版本，你是否應該放棄正在做的事情並進行升級？當然不是：升級並不重要。但另一方面，將 Google Search 困在上世紀 90 年代的 OS 版本上，明顯是一個問題。

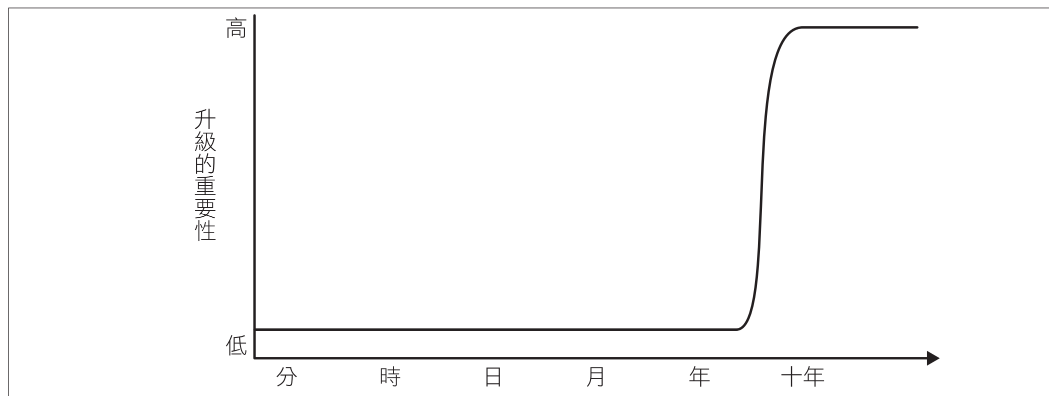

圖 1-1　壽命和升級的重要性

6　參見 Appcelerator，2012/12/6 於 Axway Developer 部落格的文章〈Nothing is Certain Except Death, Taxes and a Short Mobile App Lifespan〉（除了死亡、稅收和行動應用程式的短暫壽命，沒有什麼是確定的）（*https://oreil.ly/pnT2_*）。

碁峰 www.gotop.com.tw

從預期壽命光譜的低點和高點可看出，在某個地方存在一個轉變。在一次性的程式與持續十年的專案之間的某個地方，發生了一個轉變：專案必須對不斷變化的外在因素做出反應。[7] 對於任何從一開始就沒有計劃升級的專案來說，這種轉變非常痛苦，原因有三，每一個都會使其他因素更加複雜：

- 你正在進行的任務尚未完成此專案；更多隱藏的假設已被引入。
- 嘗試進行升級的工程師不太可能有這方面的經驗。
- 升級的規模往往比平時大；一次做幾年的升級，而不是多次進行累加式升級。

因此，在實際經歷過一次這樣的升級（或放棄部分升級）之後，高估後續升級的成本並決定「再也不升級」是合理的。得出這一結論的公司最終只會走放棄一途，重寫程式碼，或是決定不再升級。與其採取自然的方法來避免痛苦的任務，有時更負責任的答案是投資於減少痛苦。這完全取決於升級的成本、它所提供的價值，以及有關專案的預期壽命。

不僅要度過第一次大的升級，而且要達到能夠可靠地與時俱進的程度，這是你的專案長期可持續性的本質所在。可持續性需要規劃和管理所需變更的影響。對於 Google 的許多專案來說，我們相信我們已經實現了這種可持續性，主要是透過反覆試驗。

那麼，具體來說，短期程式設計與產生預期壽命更長的程式碼有何不同？隨著時間的推移，我們需要更清楚認識到「碰巧可以工作」和「可維護」之間的區別。對於這些問題的確認，沒有完美的解決方案。這是不幸的，因為長期維護軟體的可維護性是一場持續的戰鬥。

海勒姆法則

如果你要維護供其他工程師使用的專案，那麼關於「它是有效的」與「它是可維護的」的最重要教訓是所謂的海勒姆法則（*Hyrum's Law*）：

> 如果有足夠的 API 用戶，那麼你在契約中的承諾就無關緊要：你的系統所有可觀察到的行為都會被某個人所依賴。

根據我們的經驗，這個原則（axiom）是任何關於軟體隨時間變化之討論中的主導因素（dominant factor）。從概念上講，它類似於熵（entropy）：在討論隨時間變化和維持時，必須注意到海勒姆法則[8]，正如討論效率和熱力學時，必須注意到熵一樣。僅僅因

7 你自己的優先事項和品味將告訴你轉變發生的具體位置 。我們發現，大多數專案似乎都願意在五年內升級。一般來說，5 到 10 年之間似乎是這種轉變的保守估計。

8 值得稱讚的是，海勒姆真的很努力地想要謙虛地將其稱為「隱性依賴法則」（The Law of Implicit Dependencies），但海勒姆法則是 Google 大多數人已經確定的簡寫。

碁峯 www.gotop.com.tw

為熵永遠不會減少，並不意味著，我們不應該嘗試提高效率。僅僅因維護軟體時適用海勒姆法則，並不意味著，我們不能未雨綢繆或者不能更好地瞭解它。儘管我們可以減輕它的影響，但我們知道，它永遠無法根除。

海勒姆法則代表了這樣的實務知識（practical knowledge）：即使有最好的意圖、最好的工程師以及程式碼審查（code review）方法，我們也不能假設你會完全遵守已發布的契約或可靠的最佳實施方法（best practices）。做為一個 API 擁有者，你可以透過清楚瞭解介面承諾來獲得一定的靈活性和自由度，但實際上，一個給定之變更的複雜性和困難度還取決於用戶發現 API 的某些可觀察到之行為的有用程度。如果用戶不能倚賴這樣的東西，你的 API 將很容易變更。如果有足夠的時間和足夠的用戶，那麼即使是無害的變更也會破壞一些東西；[9] 你對這些變更的價值分析必須將調查、確定和解決這些問題的困難度納入其中。

例子：雜湊排序

思考下面的雜湊迭代排序（hash iteration ordering）例子。如果我們在一個基於雜湊的集合（hash-based set）中插入 5 個元素，我們會按照什麼順序將它們取出來？

```
>>> for i in {"apple", "banana", "carrot", "durian", "eggplant"}: print(i)
...
durian
carrot
apple
eggplant
banana
```

大多數程式員都知道雜湊表（hash tables）是無明顯順序的。很少人知道他們使用的特定雜湊表是否打算提供特定排序的細節。這看起來似乎並不明顯，但是過去的一、二年裡，資訊業（computing industry）使用這種資料類型的經驗發生了變化：

- 雜湊氾濫（hash flooding）[10] 攻擊為不確定的雜湊迭代（hash iteration）提供了更大的動力。
- 想從雜湊演算法（hash algorithms）或雜湊容器（hash containers）的改進，獲得可能的效率提升，需要改變雜湊迭代順序（hash iteration order）。
- 根據海勒姆法則，程式員將可按照雜湊表的遍歷順序（traversal order）來編寫程式，如果他們有能力這樣做的話。

9 參見 xkcd 漫畫 “Workflow”（*http://xkcd.com/1172*）。

10 這是一種 Denial-of-Service（阻斷服務，或簡寫為 DoS）攻擊，其中不受信任的用戶知道雜湊表和雜湊函式的結構，並以降低雜湊表操作之演算法性能的方式來提供資料。

碁峰 www.gotop.com.tw

因此，如果你問任何一個專家「我可以為我的雜湊容器假設一個特定的輸出序列嗎？」這個專家大概會說「不行」。大體上這是正確的，但可能過於簡單化。更細緻的答案是：「如果程式碼的壽命短，沒有對硬體、語言運行環境（language runtime）或資料結構的選擇進行變更，那麼這樣的假設是可以的。如果你不知道自己的程式碼可以使用多長時間，或者你不能保證你所依賴的任何東西都不會改變，這樣的假設是不正確的。」此外，即使你自己的實作不倚賴於雜湊容器順序（hash container order），它也可能被其他程式碼使用，暗中建立此類依賴關係。例如，如果你的程式庫將值序列化（serialize）為遠端程序調用（Remote Procedure Call，或簡寫為 RPC）回應，那麼 RPC 調用程序（caller）最後可能會依賴於這些值的順序。

這是說明「有效」（work）和「正確」（correct）有所不同的一個非常基本的例子。對於短命的程式，倚賴你的容器之迭代順序（iteration order）不會引起任何技術問題。另一方面，對於一個軟體工程專案來說，如此倚賴已定義的順序（defined order）是一種風險——給予足夠的時間，某些東西將讓變更該迭代順序變得有價值。該價值可以透過多種方式體現出來，無論是效率、安全性，還是僅讓資料結構面向未來（future-proofing）以允許將來的變更。當這個價值變得清晰時，你將需要權衡「該價值」和「讓你的開發者或客戶承受崩潰的痛苦」之間的輕重。

有些語言為了防止依賴性，專門在程式庫版本之間，或甚至在同一程式的執行之間，隨機化（randomize）雜湊排序。但即便如此，海勒姆法則仍然讓人感到意外：有些程式碼會使用雜湊迭代排序（hash iteration ordering）做為一種效率低下的亂數產生器。現在去掉這種隨機性，會讓用戶崩潰。就像熵在每個熱力學系統中都會增加一樣，海勒姆法則適用於每個可觀察到的行為。

讓我們思考以「現在就能使用」的心態和「無限期地使用」的心態來編寫程式碼之間的差異，我們可以獲得一些明確的關係。將程式碼視為具有（高度）可變壽命需求（variable lifetime requirement）的產出物（artifact），我們就可以開始對程式設計風格進行分類：程式碼若依賴於其依賴項之脆弱和未發布的功能，則可能被描述為靈活（hacky）或巧妙（clever）；若程式碼遵循最佳實施方法並對未來有規劃，則可能被描述為無瑕（clean）或可維護（maintainable）。兩者都有各自的目的，但選擇哪一種取決於相關程式碼的預期壽命。我們常說，如果「巧妙」是一種恭維，那就是程式設計（programming）；如果「巧妙」是一種指責，那就是軟體工程（software engineering）。

碁峯 www.gotop.com.tw

為什麼不以「不變」為目標呢？

在所有關於時間和對變化做出反應之必要性的討論中，隱含著這樣一個假設，即改變可能是必要的。是嗎？

與本書中的其他所有內容一樣，這要視情況而定。我們將欣然承諾：「對於大多數專案來說，在足夠長的時間內，可能需要更改其下的所有內容。」如果你有一個用純 C 語言編寫的專案，沒有任何外部的依賴項（或者只具有保證長期穩定性的外部依賴項，例如 POSIX），那麼你或許可以避免任何形式的重構或升級困難的問題。C 語言在提供穩定性方面做得很出色——在很多方面，這都是其主要目的。

大多數專案都有更多機會去改變底層技術。大多數程式語言和執行環境的變化都比 C 語言要大得多。即使用純 C 語言實作的程式庫，也可能為了支援新的功能而改變，進而影響下游的用戶。從處理器到網路程式庫，再到應用程式碼，各種技術都會暴露出安全問題。你的專案所倚賴的每一項技術都有一定的風險（希望很小）：可能會包含一些關鍵的錯誤和安全的漏洞，只有在你倚賴它時才會暴露出來。你無法為 Heartbleed（*http://heart bleed.com*）部署補丁或無法緩解諸如 Meltdown 和 Spectre（*https://meltdownattack.com*）之類的推測執行（speculative execution）問題，因為你已經假設（或承諾）一切都不會改變，所以這是一場重大的賭博。

效率的提高使情況更加複雜。我們希望為我們的資料中心配備具有成本效益的運算設備，尤其是提高 CPU 效率。但是，來自早期 Google 的演算法和資料結構，在現代設備上的效率根本不高：鏈結串列（linked-list）或二元搜尋樹（binary search tree）仍然可以正常工作，但是 CPU 週期與記憶體延遲之間不斷擴大的差距會影響「高效」程式碼的外觀。隨著時間的推移，如果不對軟體進行相應的設計變更，升級到新硬體的價值就會降低。向後相容性（backward compatibility）可確保舊系統仍能正常工作，但不能保證舊的優化仍然有用。如果不願意或不能利用這些機會，將招致巨大的成本。這樣的效率問題特別微妙：原始設計可能完全合乎邏輯並且遵循合理之最佳的實施方法。只有在向後相容的變化演變之後，新的、更有效的選擇才變得重要。雖然沒有犯錯誤，但隨著時間的推移，改變仍然很有價值。

像剛才提到的那些擔憂，說明了何以長期專案沒有投資於可持續性，會有很大的風險。我們必須有能力應對這類問題，並利用機會，而不管它們是直接影響我們，還是僅體現在我們所依賴之技術的遞移閉包（transitive closure）上。改變本來就不是好事。我們不應該僅為了改變而改變。但是我們有改變的能力。如果我們允許這種最終的必要性，我們還應該考慮是否投資以使這種能力變得便宜。每個系統管理員都知道，理論上來說，你可以從磁帶來復原是一回事，而在實施方法中確切知道如何做以及在需要時知道需要多少成本是另一回事。實施方法和專業知識是提高效率和可靠性的巨大動力。

碁峰 www.gotop.com.tw

規模與效率

正如《網站可靠性工程》（Site Reliability Engineering，或簡寫為 SRE）[11] 一書中所指出的那樣，Google 的整個生產系統（production system）是人類所創造的最複雜的機器之一。建構這樣一台機器並使其維持平穩的運行，所涉及的複雜性，需要我們的組織和全球各地的專家進行無數小時的思考、討論和重新設計。所以，我們編寫了一本書，講述維持機器在該規模下運行的複雜性。

本書大部分的內容都集中在生產這種機器之組織規模的複雜性，以及我們用來維持機器長期運行的過程。再次思考「程式碼基底可持續性」（codebase sustainability）的概念：「當你能夠安全地變更所有你應該變更的內容，並且可以在程式碼基底的生命週期內這樣做時，你的組織的程式碼基底便是可持續的。」在對能力的討論中還隱藏著一個成本問題：如果改變某件事的成本過高，它很可能會被推遲。如果成本隨著時間的推移而呈超線性成長，那麼該業務顯然是不可擴展的。[12] 最終，時間會佔上風，並且出現你絕對必須變更的意外情況。當你的專案之範圍擴大一倍，並且需要你再次執行該任務時，是否會耗費兩倍的人力？下一次你是否有足夠的人力資源來解決這個問題？

人力成本並不是唯一需要擴展的有限資源。就像軟體本身需要利用傳統資源（如運算、記憶體、存儲和頻寬）以進行良好的擴展一樣，該軟體的開發也需要擴展，無論是在人時（human time）的投入，還是在為你的開發工作流程（development workflow）提供動力的運算資源上都是如此。如果你的測試叢集（test cluster）之運算成本呈超線性成長，每個季度（quarter）每人消耗更多的運算資源，那麼你的測試叢集就走上了不可持續的道路，需要儘快進行改變。

最後，軟體組織（software organization）最寶貴的資產（程式碼基底（codebase））本身也需要擴展。如果你的建構系統或版本控制系統會隨著時間的推移以超線性的方式擴展，可能是由於「變更日誌歷史紀錄」（changelog history）成長和不斷增加的結果，到了你無法繼續下去的時候。許多問題，比如「進行完整的建構需要多長時間？」、「提取儲存庫的一個新副本需要多長時間？」或者「升級到新的語言版本需要多少費用？」，沒有得到積極監控，而且變化速度緩慢。它們很容易變得像溫水煮青蛙（*https://oreil.ly/clqZN*）；問題很容易慢慢惡化，讓人察覺不到危險。只有在整個組織都意識到並致力於擴展，你才有可能繼續關注這些問題。

---

11 Beyer, B. 等人所著之《網站可靠性工程：Google 的系統管理之道》（Site Reliability Engineering: How Google Runs Production Systems）（Boston：O'Reilly Media，2016 年）。

12 本章中，每當我們在非正式的情況下使用「可擴展」（scalable）時，我們的意思是「關於人際互動的次線性擴展（sublinear scaling）」。

碁峰 www.gotop.com.tw

你的組織賴以生產和維護程式碼的所有東西，在總體成本和資源消耗方面應該是可擴展的。特別是，你的組織必須反覆做的所有事情，在人力方面應該是可擴展的。從這個意義上講，許多常見的政策似乎都無法擴展。

無法擴展的政策

只要稍加操練，就能更容易發現具有不良擴展性的政策。最常見的是，透過考慮強加在一個工程師身上的工作，並想像組織規模擴大 10 倍或 100 倍的規模，就可以發現這些問題。當我們的規模擴大 10 倍時，我們的樣本工程師（sample engineer）需要跟上的工作量會增加 10 倍嗎？我們的工程師必須完成的工作量是否會隨組織規模的增加而增加？

工作量是否會隨著程式碼基底的擴展而擴展？如果以上皆成立，那麼我們是否有任何機制可以自動化或優化該工作？如果沒有，我們就有擴展的問題。

思考一下傳統的棄用（deprecation）做法。我們將在第 15 章中更詳細地討論「棄用」，但常見的棄用做法是擴展問題的一個很好的例子。一個新的 Widget（小部件）已經被開發出來，並決定每個人都應該使用新的，停止使用舊的。為了激勵大家，專案負責人說：「我們將在 8 月 15 日刪除舊的 Widget；請確保你已經轉換到新的 Widget。」

這種做法在小型軟體設置中可能有效，但隨著依賴關係圖（dependency graph）之深度和廣度的增加，很快就會失敗。團隊依賴於不斷增加的 Widget 數量，而一次建構中斷可能會影響公司的比例越來越大。要以可擴展的方式解決這些問題，意味著需要改變我們進行棄用的方式：團隊無須把遷移工作推給客戶，而是可以透過提供的所有規模經濟（economies of scale）自行內部化。

2012 年，我們試圖用緩解客戶流失（mitigating churn）的規則來阻止這一現象：基礎架構團隊必須自己動手將內部用戶遷移到新版本，或者以向下相容（backward-compatible）的方式進行適當的更新。這項政策，我們稱之為「客戶流失規則」（Churn Rule），規模更大：依賴的專案不再為了跟上進度而逐步加大努力。我們還瞭解到，讓一支專門的專家小組來變更規模，比要求每個用戶付出更多的維護努力要好：專家花一些時間深入瞭解整個問題，然後將專業知識應用到每個子問題。強迫用戶（user）對客戶流失（churn）做出反應，意味著每個受影響的團隊都會做得更糟，解決他們眼前的問題，然後丟棄那些現在無用的知識。專業知識的擴展性更好。

開發分支（development branch）的傳統用法是另一個有內在擴展問題（built-in scaling problem）之政策的例子。一個組織可能會發現，將大型功能合併到主線（trunk）中，會破壞產品的穩定性，並得出此結論：「我們需要對何時合併進行更嚴格的控制。我們應該減少合併的頻率。」這很快導致每個團隊或每個功能都有單獨的開發分支。每

碁峰 www.gotop.com.tw

當有任何分支被確定為「完成」時，都會對其進行測試並合併到主線中，以重新同步（resyncing）和測試（testing）的形式，對仍在開發分支上工作的其他工程師，觸發（triggering）一些潛在之代價昂貴的工作（expensive work）。這樣的分支管理（branch management）對一個要處理 5 到 10 個這樣分支的小組織來說是可行的。隨著組織規模（以及分支數量）的增加，很快就會發現，我們為完成同樣任務而付出的開銷越來越大。當我們擴大規模時，我們需要使用不同的做法，我們將在第 16 章中進行討論。

可擴展的政策

隨著組織的發展，什麼樣的政策，成本較優？或者說，更好的是，隨著組織的發展，可以制定什麼樣的政策來提供超線性的價值？

我們最喜歡的一個內部政策是基礎架構團隊的一大助力，保護他們安全地變更基礎架構的能力。「如果一個產品由於基礎架構變更而出現運行中斷（outage）或其他問題，但我們的持續整合（Continuous Integration，或簡寫為 CI）系統中的測試沒有發現問題，那麼就不是基礎架構變更的錯。」更通俗地說，這句話的意思就是「如果你喜歡它，就應該為它進行 CI 測試」，我們稱之為「碧昂絲法則」（Beyoncé Rule）。[13] 從擴展的角度來看，碧昂絲法則意味著複雜的、一次性的定制測試（bespoke test），如果不是由我們通用的 CI 系統觸發的，則不算數。否則，基礎架構團隊中的工程師，可能需要追蹤每個具有受影響程式碼的團隊，並詢問他們是如何進行測試的。如果我們有一百名工程師，我們可以這樣做。但我們絕對不能再這樣下去了。

我們發現，隨著組織規模的擴大，專業知識（expertise）和共享交流論壇（shared communication forums）會帶來巨大的價值。隨著工程師在共享論壇中討論和回答問題，知識會被傳播開來。於是新的專家會成長起來。如果你有一百個工程師在編寫 Java 程式，只要有一位友善且樂於助人的 Java 專家願意回答問題，就會使一百名工程師編寫出更好的 Java 程式碼。知識是病毒，專家是載體，對於清除工程師常見之絆腳石的價值，有很多可以說的。我們將在第 3 章中更詳細地討論這個議題。

13 這是對流行歌曲 Single Ladies 的引用，其中包括一句歌詞 “If you liked it then you shoulda put a ring on it.”（如果你喜歡它，就應該為它戴上戒指。）

碁峰 www.gotop.com.tw

例子：編譯器升級

思考一下升級編譯器的艱巨任務。理論上講，考慮到語言的向下相容性（backward compatible）需要花費的精力，編譯器的升級應該是很便宜的，但實際操作起來有多便宜呢？如果你之前從未做過這樣的升級，那麼你將如何評估你的程式碼基底（codebase）與該變更的相容性？

根據我們的經驗，即使普遍認為語言和編譯器的升級具向下相容性，但它們仍是微妙和艱巨的任務。編譯器升級幾乎總是會導致行為上的細微變化：修正錯誤的編譯、調整優化，或有可能改變任何以前未定義的結果。你會如何根據所有這些潛在的結果來評估你整個程式碼基底的正確性？

Google 歷史上最有名的編譯器升級發生在 2006 年。當時，我們已經運營了幾年，並擁有數千名工程師。我們已經有大約五年沒有更新編譯器了。我們的大部分工程師都沒有更換編譯器的經驗。我們的大部分程式碼只處於一個編譯器版本的作用之下。對於一個由（大部分）志願者組成的團隊來說，這是一項艱難和痛苦的任務，最終變成了尋找捷徑和化繁為簡的問題，以便繞過我們不知道如何採用的上游編譯器和語言變更。[14] 最後，2006 年的編譯器升級非常痛苦。許多的海勒姆法則問題，無論大小，都已潛入程式碼基底，並加深了我們對特定編譯器版本的依賴。要打破這些隱含的依賴關係是很痛苦的。相關的工程師是在冒險：我們還沒有「碧昂絲法則」，也沒有普及的 CI 系統，所以很難提前知道變更的影響，也很難確定他們不會因為倒退（regression）而受到指責。

這個故事一點也不稀奇。許多公司的工程師都能講述類似的故事：痛苦的升級。不尋常的是，我們在事後認識到這項任務是痛苦的，並著手關注技術和組織的變革，以克服規模的問題，並將規模轉化為我們的優勢：自動化（因此，一個人可以做更多的事情）、整合／一致性（因此，低級別之變更的問題範圍是有限的）以及專業知識（因此，少數人可以做更多的事情）。

你變更基礎架構的頻率越高，就越容易做到這一點。我們發現，大多數情況下，當程式碼做為編譯器升級的一部分而被更新時，它會變得不那麼脆弱，將來也更容易升級。在大多數程式碼都經過多次升級的生態系統中，它不再倚賴於底層實作的細微差別；而是倚賴於語言或作業系統所保證的實際抽象概念。無論你要進行什麼升級，即使控制了其他因素，程式碼基底的首次升級成本都會比以後升級的成本高很多。

14 具體來說，C++ 標準程式庫中的介面需要在 `std` 命名空間中被引用，而對 `std::string` 的優化修改，對我們的使用來說被證明一個是顯著的劣化（pessimization），因此需要一些其他的解決方案。

碁峰 www.gotop.com.tw

透過這些經驗和其他經驗，我們發現了許多影響程式碼基底靈活性的因素：

專業知識

我們知道如何做到這一點；對於某些語言，我們現在已經在許多平台上完成了數百次的編譯器升級。

穩定性

由於我們定期採用新的版本，所以版本之間的變化較小；對於某些語言，我們現在每隔兩週就會部署一次編譯器升級程序。

一致性

沒有經過升級的程式碼已經比較少了，這也是因為我們有定期進行升級。

熟悉性

因為我們經常這樣做，我們會在進行升級的過程中發現冗餘並嘗試實現自動化。這與 SRE（網站可靠性工程）對於勞力的觀點有很大的重合。[15]

政策

我們有如同「碧昂絲法則」（Beyoncé Rule）的流程和政策。這些流程的最終效果是，升級仍然可行，因為基礎架構團隊不需要擔心每一個未知的使用情況，只需要擔心在我們的 CI 系統中可見的使用情況。

這給我們的啟示不是關於編譯器升級的頻率或難度，而是當我們意識到編譯器升級任務是必要的時候，我們就找到了方法，確保在程式碼基底不斷成長的情況下，由固定數量的工程師來進行這些任務。[16] 如果我們認為這項任務成本過高，將來應該避免，那麼我們可能仍然使用十年前的編譯器版本。由於錯過了優化的機會，我們可能要為運算資源多支付 25% 的代價。例如，使用 2006 年的編譯器肯定無助於緩解「推測執行」（Speculative Execution）漏洞，我們的中央基礎架構可能會面臨重大的安全風險。停滯不前是一種選擇，但往往不是明智之舉。

15 Beyer, B. 等人所著之《網站可靠性工程：Google 的系統管理之道》，第 5 章〈減少瑣事〉。

16 根據我們的經驗，一個普通的軟體工程師（software engineer，或簡寫為 SWE）在單位時間內產生的程式碼列數是相當穩定的。對於一個固定的 SWE 群體，隨著時間的推移，程式碼基底會與 SWE 月數成線性成長。如果你的任務需要的努力與程式碼的列數成比例，那就令人擔憂了。

碁峰 www.gotop.com.tw

左移

我們看到的一個廣泛的真理是，在開發人員工作流程的早期發現問題，通常可以降低成本。讓我們來思考一個功能之開發人員工作流程的時間軸（timeline）：從左側進展到右側，由概念和設計開始，一直到實作（implementation）、審查（review）、提交（commit）、發行（canary）和最終的生產環境部署（production deployment）。如圖 1-2 所示，在此時間軸上較早地將問題偵測移到「左側的」位置，將使修正成本比「等待時間較長的」位置更便宜。

這個詞似乎源自這樣的論點：安全問題不能推遲到開發流程的最後，並要求「將安全問題左移」（shift left on security）。這種情況下的論點相對簡單：如果僅在產品投入生產環境之後才發現安全問題，那麼你所面臨的是成本非常昂貴的問題。如果在將其部署到生產環境之前被發現，則可能仍需要大量的工作來確定並解決問題，但較便宜。如果你能在最初的開發人員將缺陷提交到版本控制之前就發現它，那就更便宜了：他們已經瞭解此功能；根據新的安全限制進行修改，比提交並強迫其他人對其進行分類和修正要便宜得多。

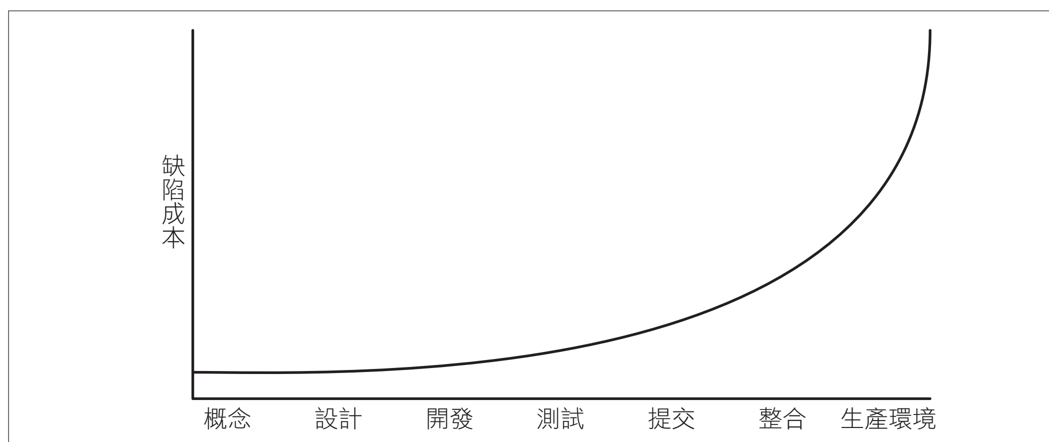

圖 1-2　開發人員工作流程的時間軸

本書中曾多次出現同樣的基本模式（basic pattern）。在提交之前，透過靜態分析和程式碼審查捕捉到的錯誤，要比進入生產環境的錯誤便宜得多。在開發過程的早期提供突出的品質、可靠性和安全性之工具和實施方法，是我們的許多基礎架構團隊之共同目標。這不需要任何一個流程或工具就可以完美實現，因此我們可以採用深度防護（defense-in-depth）的做法，希望盡可能在圖的左側捕捉到任何錯誤。

碁峯 www.gotop.com.tw

權衡與成本

如果我們瞭解如何進行程式設計，瞭解我們正在維護之軟體的壽命，以及隨著我們規模規的擴大有更多的工程師製作和維護新的功能時，瞭解如何維護它，剩下的就是做出好的決策。這似乎是顯而易見的：在軟體工程中，就像是在生活中一樣，好的選擇會帶來好的結果。然而，此一觀點的影響很容易被忽略。在 Google 內部，人們對「因為我說過」（because I said so）有強烈的反感。重要的是，任何題目（topic）都要有一個決策者，當決策似乎是錯誤的時候，要有明確的上報途徑（escalation paths），但目標是達成共識，而不是達成一致。看到一些「我不同意你的衡量／評價，但我知道你是如何得出這個結果的」之情況是好的，也是意料之中的。所有這一切蘊含著這樣一種想法，即任何事情都需要一個理由；「僅僅因為」、「因為我說過」或「因為其他人都是這樣做的」這些都是潛伏著錯誤決策的地方。只要這樣做是有效的，我們就應該能夠在決定兩種工程方案的一般成本時，解釋我們的工作情況。

我們所說的成本（cost）是指什麼？我們在這裡談論的不僅僅是錢。「成本」大致相當於工作量，可能涉及以下任何或所有因素：

- 財務成本（例如，錢）
- 資源成本（例如，CPU 時間）
- 人員成本（例如，工程工作量）
- 交易成本（例如，採取行動的成本是多少？）
- 機會成本（不採取行動的代價是什麼？）
- 社會成本（這種選擇對整個社會將產生什麼影響？）

從歷史上看，人們特別容易忽略社會成本的問題。然而，Google 和其他大型科技公司，現在能夠可靠地為數以億計的用戶部署產品。在許多情況下，這些產品都有明顯的淨收益（net benefit），但當我們以這樣的規模運作時，即使是可用性、可存取性、公平性或潛在濫用方面的微小差異，也會被放大，這往往會損害已經被邊緣化的群體。軟體會滲透到社會和文化的許多方面；因此，明智的做法是，當我們做出產品和技術的決策時，既要意識到好的一面，也要注意到壞的一面。我們將在第四章詳細探討這個問題。

除了上述的成本（或我們對成本的估計），還有一些偏見：現狀偏差（status quo bias）、損失規避（loss aversion）等。當我們評估成本時，我們需要記住之前列出的所有成本：一個組織的健康不僅僅是銀行裡有沒有錢，更重要的是它的成員是否感到有價值和有效率。在軟體工程等富有創造力和利潤豐厚的領域，財務成本通常不是限制因素，而人員成本通常是限制因素。保持工程師的快樂、專注和投入，所帶來的效率指標（efficiency gains）很容易主導其他因素，只因為專注度和生產力如此的多變，10% 到 20% 的差異是很容易想像的。

碁峯 www.gotop.com.tw

例子：白板筆

在許多組織中，白板筆被視為珍貴的用品。它們受到嚴格控制，始終供不應求。在任何一個白板上，總是有一半的白板筆是乾的，無法使用。有多少次，因為無白板筆可用，導致會議中斷？有多少次，因為白板筆斷水，導致思路受阻？有多少次，所有的白板筆就這樣不見了，大概是因為其他團隊的白板筆用完了，不得不拿走你的白板筆？都是為了一個成本不到 1 美元的用品。

Google 往往會在大多數工作區中打開裝滿辦公室用品的櫥櫃，其中包含白板筆。只要稍加注意，就可以輕易抓出各種顏色的白板筆。我們在某個時刻做了明確的取捨：優化無障礙的腦力激盪，比防止有人拿著一堆白板筆亂跑更重要。

我們的目標是對我們所做的每一件事都保持同等的眼光，並明確權衡成本／效益的輕重，從辦公室用品和員工福利到開發人員的日常經驗，再到如何配置（provision）和運行全球規模的服務（global-scale services）。我們經常說：「Google 是一種資料驅動文化。」事實上，這是一種簡化：即使沒有資料，也可能仍然有證據、先例和論據。做出好的工程決策，就是權衡所有可用的輸入，並就取捨做出明確的決定。有時，這些決策是基於本能或公認的最佳做法，但只有在我們用盡了各種方法來衡量或估計真正的潛在成本之後。

最後，工程團隊的決策應該歸結為以下幾點：

- 我們這樣做是因為我們必須這樣做（法律要求、客戶要求）。
- 我們這樣做是因為根據目前的證據，這是我們當時能看到的最佳選擇（決定自某個適當的決策者）。

決策不應該是「我們這樣做是因為我這樣說。」[17]

對決策的投入

當我們衡量資料時，我們發現有兩種常見的情況：

- 涉及到的所有量（quantities）都是可以衡量的，或至少是可以估算出來。這通常意味著，我們正在評估 CPU 和網路之間的平衡，或者金額和 RAM 之間的平衡，或者考慮是否要花兩個星期的工程師時間（engineer-time）來節省整個資料中心的 *N* 個 CPU。

17　這並不是說，決策需要一致，或需要達成廣泛的共識；最後必須有人做為決策者。這主要是說明決策流程應如何流向實際負責決策的人。

碁峰
www.gotop.com.tw

- 有些量是微妙的，或者我們不知道如何量測它們。有時這表現為「我們不知道這將花費都少工程師時間」。有時甚至更含糊：你如何衡量設計不良之 API 的工程成本？或者產品選擇的社會影響？

在第一種決策上，沒有什麼不足之處。任何軟體工程組織都可以且應該追蹤運算資源、工程師工時（engineer-hours）和經常與你互動的其他量（other quantities）之當前成本。即使你不想向你的組織公布確切的金額，你仍然可以產生一個轉換表：這麼多的 CPU 成本跟這麼多的 RAM 或這麼多的網路頻寬成本是一樣的。

手上有商定的換算表（conversion table），每個工程師都可以進行自己的分析。「如果我花兩週的時間，將這個鏈結串列（linked-list）改成一個高性能的結構，我將使用 5GB 以上的生產環境記憶體（production RAM），但會節省 2000 個 CPU。我應該這樣做嗎？」這個問題不僅取決於 RAM 和 CPU 的相對成本，還取決於人員成本（為軟體工程師提供兩週的支援）和機會成本（該工程師在兩週內還能生產出什麼產品？）。

對於第二種決策，沒有簡單的答案。我們倚靠經驗、領導力和先例來協商這些問題。我們所投資的研究能幫助我們量化難以量化的東西（見第七章）。然而，我們得到的最廣泛建議是，要認識到並非一切都是可以衡量或可預測的，並努力以同樣的優先次序和謹慎的態度來對待這些決策。它們通常同樣重要，但更難管理。

例子：分散式建構

以建構（build）為例。根據完全不科學的 Twitter 調查，大約有 60-70％的開發人員在本地進行建構，即使是今日之複雜的大型建構。這直接導致了嚴肅的課題，正如 Compiling 這幅漫畫（*https://xkcd.com/303*）所要表達的那樣：在你的組織中，等待建構會浪費多少生產時間？將其與為一個小型團體運行 `distcc` 之類東西的成本做比較。或者，為一個大型團隊運行一個小型的建構場（build farm）需要多少錢？這些費用需要幾週／幾個月的時間才能實現淨收益？

早在 2000 年代中期，Google 完全倚賴於本地建構系統：簽出（check out）程式碼，然後在本地編譯。在某些情況下（使用桌機來建構 Maps ！），我們會使用大量的本地機器，但隨程式碼基底（codebase）的成長，編譯時間越來越長。不出所料，由於時間的流失，我們在人員成本上產生越來越大的開銷，同時也為了更大、更強的本地機器增加了資源成本…等等。這些資源成本特別麻煩：當然，我們希望人們盡快進行建構，但大多數情況下，高性能的桌上開發機器將處於閒置狀態。這並非投資這些資源的恰當方式。

碁峰 www.gotop.com.tw

最後，Google 開發了自己的分散式建構系統。開發這個系統當然要付出一定的代價：工程師要花時間開發它，而且改變每個人的習慣和工作流程、學習新的系統需要更多的工程師時間，當然還需要額外的運算資源。但從整體上的節省來看，顯然是值得的：建構的速度變得更快，工程師的時間被收回，硬體投資可以集中在託管之共享基礎架構（實際上，是我們的生產團隊的一部分）上，而不是功能越來越強大的桌機上。第 18 章將詳細介紹我們的分散式建構做法和相關的折衷方案。

因此，我們建構了一個新系統，將其部署到生產環境中，並加快了所有人的建構速度。那是故事的幸福結局嗎？不完全是：提供分散式建構系統大大提高了工程師的工作效率，但隨著時間的流逝，分散式建構本身也變得擁腫了。在前一個例子中，個別工程師所受之限制（想要盡快進行本地建構以從中受益）在分散式建構系統中並不存在。建構圖（build graph）中，擁腫或非必要的依賴關係變得非常普遍。當每個人都直接感受到不理想建構所帶來的痛苦並被激勵提高警惕時，激勵措施就會被更好地結合在一起。透過移除這些激勵措施，並在平行分散式建構中隱藏擁腫的依賴關係，我們創造了一種局面，此局面中資源的消耗可能會氾濫，幾乎沒有人被激勵去關注變得擁腫的建構。這讓人聯想到 Jevons（傑文斯）悖論（*https://oreil.ly/HL0sl*）：資源的消耗可能會隨著其使用效率的提高而增加。

總的來說，添加分散式建構系統所節省的成本遠遠超過了與「建構和維護」相關的負成本。但是，正如我們看到的消耗增加，我們並沒有預見到所有這些成本。開拓了視野後，我們發現自己處於這樣一種局面：我們需要重新認識系統的目標以及系統的限制和我們的用法，確定最佳的實施方法（小型的依賴關係、依賴關係的機器管理），並為新生態系統的工具和維護提供資金。即使是「我們將花費 $$$ 的運算資源來收回工程師的時間」這種相對簡單的權衡，也有不可預見的下游效應（downstream effects）。

例子：在時間和規模之間做出決定

很多時候，我們的時間和規模之主題是重疊的，而且是結合在一起的。像碧昂絲法則這樣的政策可以很好地擴展，並幫助我們長期維護事物。更改作業系統的介面可能需要進行許多小的重構，以適應這些變更，但是大多數變更都會有很好的擴展，因為它們的形式都是相似的：作業系統的變更不會因為調用者（caller）和專案的不同而有不同的表現。

偶爾，時間和規模會發生衝突，沒有什麼比這個基本問題更清楚了：我們是應該添加一個依賴關係，還是應該分支／重新實作（fork/reimplement）它，以更好地滿足我們的本地需求？

碁峰 www.gotop.com.tw

這個問題可能會在軟體堆疊（software stack）的許多層級上出現，因為通常情況下，為你的狹窄問題空間（narrow problem space）定制的專門解決方案可能會勝過需要處理所有可能性的通用解決方案。透過分支或重新實作公用程式碼（utility code），並為你的狹窄領域進行定制，你可以更輕鬆地添加新的功能，或者更確定地進行優化，而不管我們談論的是微服務、記憶體中的快取（in-memory cache）、壓縮常式（compression routine）還是軟體生態系統中的任何其他東西。也許更重要的是，從這樣的分支中獲得的控制權使你免受「基礎依賴關係（underlying dependencies）之變更」的影響：這些變更不是由另一個團隊或第三方供應商決定的。你可以控制如何以及何時對時間的流逝和變更的必要性做出反應。

另一方面，如果每個開發人員將軟體專案中使用的東西都分支出去，而不是重用現有的東西，那麼可擴展性（scalability）和可持續性（sustainability）都會受到影響。對基礎程式庫（underlying library）中的安全問題做出反應，不再是更新單一依賴關係及其用戶的問題：現在需要確定該依賴關係的每個易受攻擊的分支，以及這些分支的用戶。

與大多數軟體工程決策一樣，對於這種情況並沒有放之四海皆準（one-size-fits-all）的答案。如果你的專案壽命很短，那麼分支的風險就比較小。如果有關的分支在範圍上是有限的，那就會有所幫助，同時也要避免對「可能會跨時間或專案時間邊界進行操作的介面（資料結構、序列化格式、網路協定）」進行分支。一致性（consistency）有很大的價值，但普遍性（generality）也有其自身的代價，如果你謹慎行事，你通常可以透過做自己的事情來取勝。

回顧決策、所犯下錯誤

致力於資料驅動文化（data-driven culture）的不明顯好處（unsung benefits）之一就是承諾錯誤的能力和必要性的組合。在某個時候，我們會根據現有資料做出決定——希望是基於良好的資料和一些假設，但隱含在現有資料的基礎上。隨著新資料的出現，環境的變化，或者假設的破滅，可能會清楚看到決策是錯誤的，或者當時是有意義的，但現在已經沒意義了。這對於一個壽命較長的組織來說尤為關鍵：時間不僅會引發技術依賴關係和軟體系統的變化，還會引發用於驅動決策之資料的變化。

我們堅信資料可以為決策提供信息，但我們認識到，資料會隨著時間的推移而變化，而且新的資料可能會出現。這意味著，在相關系統的壽命內，將需要不時重新審視決策。對於長期的專案來說，在做出最初的決策後，擁有變更方向的能力往往是至關重要的。而且，重要的是，這意味著，決策者需要有承認錯誤的權利。與一些人的直覺相反，承認錯誤的領導者受到的尊重更多，而不是更少。

碁峰 www.gotop.com.tw

要以證據為導向，但也要意識到那些無法衡量的事情可能仍然有價值。如果你是一個領導者，這就是你被要求做的事情：運用判斷力，斷言事情很重要。我們將在第 5 和 6 章中詳細介紹領導力（leadership）。

軟體工程與程式設計

當看到我們所指出之軟體工程和程式設計的差異時，你可能會問，是否在進行內在的價值判斷。程式設計是否不如軟體工程？一個擁有數百人的團隊之預計能持續 10 年的專案，是否比一個只用了一個月、由兩個人建構的專案更有價值？

當然不是。我們的意思並不是說軟體工程是優越的，而只是說它們代表了兩個不同的問題領域，有著不同的限制、價值和最佳實施方法。相反的，指出這種差異的價值來自於認識到某些工具在一個領域中很出色，而在另一個領域中卻不是。對於一個僅持續幾天的專案，你可能不需要依賴整合測試（見第 14 章）和持續部署（Continuous Deploymen，或簡寫為 CD）實施方法（見第 24 章）。同樣的，我們對軟體工程專案中的語意化版本控制（SemVer）[譯註] 和依賴性管理（見第 21 章）的所有長期關注並不適用於短期程式設計專案：利用一切可以利用的東西來解決手頭上的任務。

我們認為區分「程式設計」和「軟體工程」這兩個相關但又有差異的術語很重要。這種差異很大程度源於隨著時間的推移對程式碼的管理，時間對規模的影響，以及面對這些想法的決策。程式設計是產生程式碼的直接行為。軟體工程是一組策略、實施方法和工具，而這些策略、實施方法和工具都是「讓程式碼在需要使用時一直有用並允許跨團隊協作」所必需的。

結語

本書討論了以下這些主題：組織和單一程式員的策略、如何評估和完善你的最佳實施方法以及可維護軟體中使用的工具和技術。Google 一直在努力建立可持續的程式碼基底（codebase）和文化。我們未必認為我們的方法是做事情的唯一方法，但它確實提供了一個例子，證明它是可以做到的。我們希望它能提供一個有用的框架來思考一般問題：在程式碼需要繼續工作的情況下，如何維護程式碼？

譯註　見 *https://semver.org/lang/zh-TW/*。

碁峰 www.gotop.com.tw

摘要

- 「軟體工程」在維度上與「程式設計」不同：程式設計是關於程式碼的製作。而軟體工程將其擴展到包括維護該程式碼的有效壽命。
- 壽命短的程式碼和壽命長的程式碼之間至少有 10 萬倍的因數。認為同樣的最佳實施方法普遍適用於光譜的兩端，是愚蠢的。
- 當在程式碼的預期壽命內，我們有能力回應依賴性、技術或產品需求的變化時，軟體就是可持續的。我們可以選擇不做改變，但我們需要有此能力。
- 海勒姆法則：若有足夠數量的 API 用戶，你在契約中承諾什麼並不重要：你的系統的所有可觀察到的行為，都將被某人所倚賴。
- 你的組織必須重複執行的每項任務都應該在人力投入（human input）方面具有可擴展性（線性或更好的形式）。政策是使流程可擴展的絕佳工具。
- 流程效率不高和其他軟體開發任務往往會慢慢擴大規模。要小心溫水煮青蛙的問題。
- 與規模經濟相結合時，專業知識的回報尤其顯著。
- 「因為我說過」（Because I said so）這是一個可怕的理由。
- 資料驅動是一個好的開始，但實際上，大多數決策都是基於資料、假設、先例和論證的混合。當客觀資料（objective data）佔這些輸入的大部分時，是最好的，但很少能做到全部。
- 隨著時間的推移，資料驅動（data driven）意味著，當資料發生變化時（或當假設破滅時）需要改變方向。錯誤或修訂計劃是不可避免的。

碁峰 www.gotop.com.tw