前言

雖然關於重構（refactoring）的書籍有很多，但其中大多數都涉及到如何逐行改善程式碼的細節。我認為，重構最困難的部分通常不是找到改進手邊程式碼的精確方法，而是如何促成圍繞在其周圍必須發生的那一切。其實我也可以說，對於任何大型的軟體專案，那些小事情很少是重要的，協調複雜的變化才是最大的挑戰。

在《**大規模重構**》（*Refactoring at Scale*）這本書中，我試著幫助你找出那些困難的部分。這是多年實踐各種規模重構專案所累積的經驗。在 Slack 工作的期間，我領導的許多專案讓公司得以大幅擴張，我們的產品從能夠支援擁有兩萬五千名員工的客戶，成長到了有辦法處理具有五十萬名員工的客戶。我們為有效重構所發展的策略，讓我們得以承受爆炸性的組織增長，同期我們的工程團隊成長了近六倍。成功規劃和執行一個會同時影響源碼庫（codebase）可觀部分和不斷增加的工程師的專案絕非易事。我希望這本書能為你提供必要的工具和資源來做到這點。

誰應該閱讀本書

如果你經手的是大型、複雜的源碼庫，而且還需要與其他十數名（或更多）工程師合作，那本書就是為你準備的！

如果你是初級工程師，想要為你的公司做出貢獻，藉此開始累積更高級的技能，那麼進行大規模的重構工作可能是達成這一目標的絕佳途徑。這類項目具有廣泛而有意義的影響，不僅限於你的團隊（它們也不那麼光鮮耀眼，以至於資深工程師可能會馬上搶去做）。它們是你獲得新專業技能（並強化你已擁有的技能）的絕佳機會。本書會引導你如何從頭到尾順利地完成這種專案。

這本書對技術水準高，能寫出任何程式碼來解決問題但對於其他人無法理解他們工作價值而感到挫折的高級工程師而言，也是寶貴的資源。如果你感到孤立，並在尋找方法來提升身邊其他人的水準，這本書也可以教你必要的策略來協助其他人透過你的視野來看待重要的技術問題。

對於希望引導團隊完成大規模重構的技術經理，本書可以幫助你瞭解如何在過程中的每一步更好地支援你的團隊。這些書頁中並沒有包含很多的技術內容，因此，不管你以何種角色（工程經理、產品經理、專案經理）參與大規模的重構工作，你都可以從這裡所提供的經驗中獲益。

我為何撰寫此書

當我開始著手進行我第一次的大規模重構時，我理解到程式碼**為何**（*why*）需要改變，以及要**如何**（*how*）改變，但最讓我困惑的是，如何安全、漸進地引入這些變更，而且不對其他人產生負面衝擊。我急切地想要產生跨職能的影響，並沒有停下來告知這種重構可能對其他人的工作產生的衍生後果，也沒有停下來去理解我如何激勵他們幫助我完成這項工作。我只是勉強熬過去（你可以在第 10 章中閱讀有關這項重構工作的內容！）。

在接下來的幾年間，我重構了更多更多的程式碼，最終成為了幾個執行不力的重構專案的接收端。我從這些經歷中學到的經驗感覺很重要，所以我開始在一些會議上談論它們。我的演講引起了數百名工程師的共鳴，他們全都像我一樣，在實際重構他們自己公司內大量程式碼的過程中遭遇到了問題。顯然，我們的軟體教育中存在著某種漏洞，特別是關於如何專業地編寫軟體的核心面向。

在許多方面，這本書試圖教導一些在典型電腦科學課程中僅僅因為這些內容在課堂上教起來太難了而沒有涵蓋的重要內容。或許單靠一本書也無法傳授這些知識，但何不試試看呢？

本書導覽

本書分為四部分，依據規劃和執行大規模重構所需工作的概略時間順序進行組織，概述如下。

- 第一部介紹重構背後的重要概念。
  - 第 1 章討論重構的基礎知識，以及大規模重構和規模較小的重構之間的差異。
  - 第 2 章介紹程式碼劣化的各種可能方式，以及這會如何影響重構的有效性。
- 第二部涵蓋你在規劃成功重構工作時需要瞭解的一切。
  - 第 3 章概述了許多衡量指標，可用在實際進行改善之前，量測你的重構工作試圖解決的問題。
  - 第 4 章說明詳盡的執行計畫重要的組成部分，以及如何起草這樣的一份計畫。
  - 第 5 章討論獲得工程管理階層支援你重構工作的不同途徑。
  - 第 6 章介紹如何確定哪些工程師最適合接手重構工作，以及招募他們的建議。
- 第三部重點介紹如何確保你的重構在過程中順利進行。
  - 第 7 章探討如何最好地促進你團隊內部以及任何外部利益關係方的良好溝通。
  - 第 8 章介紹了在整個重構過程中保持動力的數種方法。
  - 第 9 章提供了一些建議，說明如何確保重構所引入的變更得以維持。
- 第四部包含兩個案例研討，都是取自於我在 Slack 工作時所參與的專案。這些重構影響到了我們核心應用很大的一部分，是真正大規模的重構。我希望這些能有助於闡明本書第一至第三部所討論的概念。

這種順序並非規定性的處方，僅僅因為我們已經進入了一個新的階段，並不代表我們不應該在必要時重新審視我們之前的假設。舉例來說，你可能在剛開始重構時，對你將與之合作的團隊有強烈的設想，但在擬定執行計畫的半途中，發現你需要引進的工程師比最初預期的要多。這沒有關係，因為總是這樣的！

本書編排慣例

本書中使用的編排慣例如下：

斜體字（*Italic*）

代表新名詞、URL、電子郵件位址、檔名和副檔名。中文以標楷體表示。

`定寬字（Constant width）`

用於程式碼列表，還有正文字裡行間參照程式元素的地方，例如變數或函式名稱、資料庫、資料型別、環境變數、述句和關鍵字。

這個圖示代表一個小技巧或建議。

這個圖示代表一般注意事項。

使用範例程式

本書的補充性素材（程式碼範例、習題等）可在此下載取用：
https://github.com/qcmaude/refactoring-at-scale

這本書是為了協助你完成工作而存在。一般而言，若有提供範例程式碼，你可以在你的程式和說明文件中使用它們。除非你要重製的程式碼量很可觀，否則無需聯絡我們取得許可。舉例來說，使用本書中幾個程式碼片段來寫程式並不需要取得許可。販賣或散布O'Reilly 書籍的範例，就需要取得許可。引用本書的範例程式碼回答問題不需要取得許可。把本書大量的程式範例整合到你產品的說明文件中，則需要取得許可。

引用本書之時，若能註明出處，我們會很感謝，雖然這並非必須。出處的註明通常包括書名、作者、出版商以及 ISBN。例如：「*Refactoring at Scale* by Maude Lemaire (O'Reilly). Copyright 2021 Maude Lemaire, 978-1-492-07553-0.」。

如果覺得你對程式碼範例的使用方式有別於上述的許可情況，或超出合理使用（fair use）的範圍，請儘管連絡我們：*permissions@oreilly.com*。

致謝

寫書不是一件容易的事，本書也不例外。沒有許多人的貢獻，《**大規模重構**》這本書是不可能成形的。

第一章

重構

有人問過我，重構（refactoring）到底有什麼可以讓我這麼喜歡的。是什麼讓我如此頻繁地回頭參考這些類型的專案？我告訴她，其中有部分會讓人上癮。或許單純是整理清潔的簡單動作，譬如精巧地分類和排列你的香料；或是出於井然有序的喜悅，還有最後妥善處理不必要的東西，例如把一袋被遺忘的衣服送去 Goodwill 那樣的釋然；又或者，也許是我腦中的小聲音提醒著我，這些微小、漸進式的變化將大大改善我同事的日常生活。我覺得是所有這些要素的結合。

重構行為中，有一些東西可能對我們所有人都有吸引力，無論我們是在構建新的產品功能，或是在擴展基礎架構。我們全都得在工作中平衡要多寫一點程式碼或少寫一點。我們必須努力去瞭解我們的變革所產生的下游效應，無論那是有意還是無意的。程式碼是有生命會呼吸的東西。當我想到我寫的程式碼又活了五年、十年，我就不禁微微皺眉蹙眼。我當然希望，到那時，會有人出現，要麼將其完全移除，要麼用更簡潔精煉的東西取代它，最重要的是，讓它更適合屆時的應用程式需求。這就是重構的意義所在。

在本章中，我們將先定義幾個概念。我們將為一般情況下的重構提出一個基本定義，並奠基於此，為大規模重構（refactoring at scale）發展出一個單獨的定義。為了明確表達本書的某些動機，我們將討論為什麼我們應該關心重構，以及如果我們練就了這種技能，可以為我們的團隊帶來哪些優勢。接下來，我們將深入探討重構所帶來的一些好處，以及在考慮是否執行重構時應牢記的一些風險。對這些權衡取捨有所瞭解之後，我們將考慮一些場景，看看哪些是正確時機、哪些是錯誤時機。最後，我們將詳細討論一個簡短範例來將這些概念付諸實踐。

什麼是重構？

非常簡單扼要的說，**重構**（*refactoring*）是我們在不改變其外部行為的情況下重組現有程式碼（即 *factoring*）的過程。如果你認為這個定義太過廣義，別擔心，這是刻意的！重構有許多同樣有效的可能形式，具體取決於套用它的程式碼。為了闡明這一點，我們將一個「系統（system）」定義為任何會從一組輸入產生一組輸出的固定程式碼集合。

假設我們已有這樣的一個稱作 S 的系統之具體實作，如圖 1-1 所示。此系統是在一個緊迫的截止期限下建置的，迫使作者偷吃步抄捷徑。隨著時間的推移，它變成了一團錯綜複雜的程式碼。值得慶幸的是，該系統的使用者並沒有直接暴露在系統內部的混亂之中，它們透過一個定義好的介面與 S 互動，並仰賴它提供一致的結果。

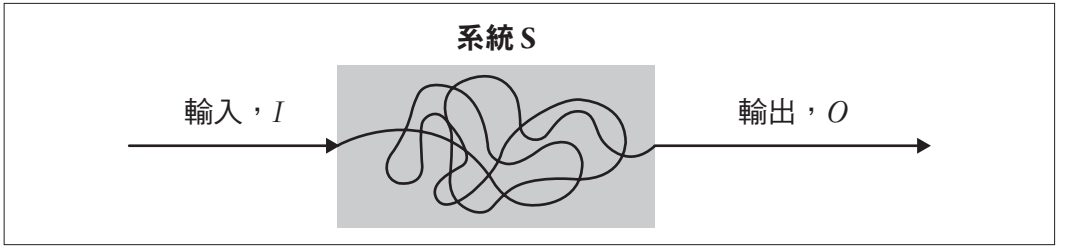

圖 1-1　具有輸入輸出的一個簡單系統

幾名勇敢的開發人員清理了該系統內部後，現在我們有了稱之為 S' 的系統，如圖 1-2 所示。雖然它是一個較乾淨整齊的系統，但對 S' 的使用者而言，什麼都沒改變。

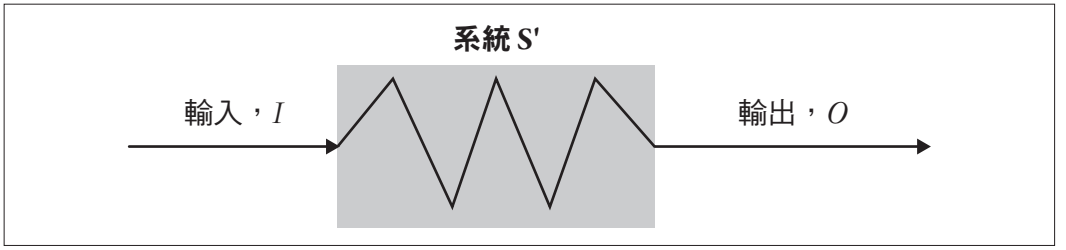

圖 1-2　具有輸入輸出的一個重構過的簡單系統

系統 S 可以是任何東西，它可以是單一個 `if` 述句（statement）、十行的函式（function）、熱門的開源程式庫、數百萬行的應用程式或介於之間的任何東西（輸入與輸出也可能同樣多變化）。這個系統可以作用在資料庫條目（database entries）、檔案集

合（collections of files）或資料串流（data streams）上。輸出不僅限於所回傳的值，還可能包括數種副作用（side effects），例如印出東西到主控台（console）或發出網路請求。你可以在圖 1-3 看到負責處理使用者請求的一個 RESTful（*https://oreil.ly/jrk1p*）服務如何對映到我們的系統定義。

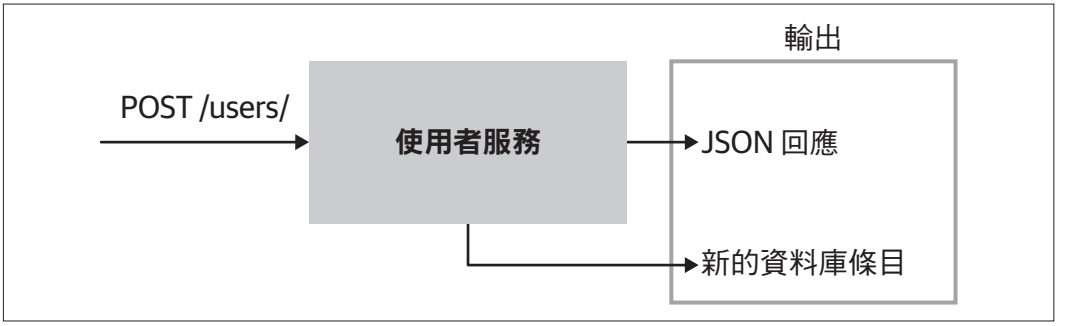

圖 1-3　一個簡單的應用程式作為一個系統

當我們繼續發展我們的重構定義，並開始探索此流程的不同面向時，確保我們都有共識的最好辦法是將每個想法連接到一個具體範例。

使用真實世界的程式設計範例是很困難的，這有幾個原因存在。鑑於我們在業界的豐富經驗，只選一個例子而非另外一些，只會立刻讓某一群讀者獲得優勢。反過來說，為了縮短篇幅而簡化某些概念或忽略某些細微差別以更簡潔地應用某個概念時，那些深諳該範例的人可能會感到無奈。為了建立一個公平的競爭環境，每當我們試圖在高層次上說明一個通用問題時，我們會以大多數人都熟悉的（希望如此）企業為例：一家知名的乾洗店。

Simon's Dry Cleaners 是當地的一家乾洗店，唯一門市位於斯普林菲爾德（Springfield）繁忙的街道上。它於星期一至星期六的正常營業時間開放。客戶既會留下一般的待洗衣物，也會留下僅限乾洗的物品。根據每個物料的數量、緊急程度和清洗難度，大約會在二到六個工作日後，將清洗過的衣物歸還給客戶。

這如何對映到我們對系統的定義呢？此店家中的乾洗作業就是系統本身。它把顧客的髒衣服作為輸入處理，並將清洗乾淨後的衣服作為輸出歸還給所有者。乾洗作業所有複雜之處都被隱藏起來，消費者看不到。我們只需把衣物留下，希望清潔公司能做好工作。系統本身相當複雜，根據輸入類型（皮革夾克、一堆襪子、絲裙等），它可能藉由執行一個或多個作業作為回應，以確保正確的輸出（乾淨的衣物）。在留下衣物到取回之

間，有很多機會發生問題：腰帶可能會丟失、污漬被忽略、某件襯衫意外退回給錯誤的顧客。但是，如果員工之間積極主動進行溝通、機器狀態良好，收據整理有序，系統將繼續平穩運行，很容易就能完成訂單。

假設 Simon's 還用紙質複印收據來營運。所有留下衣物的顧客都會在提單紙條上寫下自己的姓名和電話號碼，店員則會記錄他們的訂單。如果客戶放錯了收據，Simon's 能夠瀏覽他們最近按姓氏字母順序排列的訂單輕易查找副本。不幸的是，當顧客太晚來取回他們的乾洗衣物，而且收據放錯位置時，店員就必須從後台的箱子裡找出封存的紙條。儘管幾乎所有訂單都有成功取回，但客戶需要花費更多時間、多跑一趟才能拿到他們送洗的衣物。業主在每月底計算收益時，紙本收據也不方便，他們必須將所有交易記錄（信用卡和現金）與已完成的訂單人工手動匹配。為了達成現代化和重構他們的流程，該團隊決定升級他們的系統，使用銷售時點情報系統（point-of-sale system）消除紙面文檔的棘手問題。這樣重構就完成了！客戶繼續留下他們的乾洗衣物，並在幾天後取回它們，幾乎感受不到任何變化，但現在前台後的一切都運行得更順利。

何謂大規模重構？

2013 年末，在動蕩的發佈會中，美國所有主要新聞媒體都宣稱 Healthcare.gov 徹底失敗，該網站受到安全問題、長達數小時的停機和大量嚴重臭蟲的困擾。在推動前，不僅成本激增到近 20 億美元，其源碼庫（codebase）已經膨脹到了超過 500 萬行。雖然 Healthcare.gov 的失敗在很大程度上是由於深陷在聯邦政府官僚政策中的開發工作無法動彈所致，但當歐巴馬政府隨後宣佈要投入巨資改善這項服務的計畫時，重整和重構過度成長的軟體系統所涉及的那種不容否認的困難度，就成了主流新聞媒體所關注的焦點。在後續的幾個月裡，負責改寫 Healthcare.gov 的團隊，首先投入此源碼庫幾乎是徹底翻新的改革，這就是大規模的重構。

大規模的重構工作就是會影響到系統大量表面積的重構。它通常（但不一定是）涉及為許多用戶提供應用支援的某個大型源碼庫（100 萬行或更多的程式碼）。只要仍有舊式系統存在，就需要這些重構，其中開發人員需要在廣度上仔細考量程式碼結構，以及如何有效改善它以得到可測量的效益。是什麼使數百萬行源碼庫的重構有別於重構更小、定義更明確的應用程式？雖然我們可能很容易想到具體、迭代式的方法來改進定義明確的小系統（比如個別函式或類別），但要確定一個龐大而複雜的系統統一套用某項變更時可能產生的效應，幾乎是不可能的。有許多工具可以辨識程式碼的好壞，或自動偵測程式碼子部分中可以進行的改良，但我們基本上無法自動化人類的推理，去判斷如何重

組正在以越來越快的速度增長的源碼庫中的大型應用程式，在高速成長的公司中，更是如此。

有人可能會說，透過不斷套用可以疊加的小型變換，你就能對這種系統做出可測量的改善。這種方法或許能讓天秤開始朝好的方向傾斜，但當大部分低垂的果實都採收完畢，而小心（並逐漸）引入這些改變的工作變得更加棘手時，進展可能會顯著下降。

大規模重構是指在源碼庫中識別出某個系統性問題、構思一個更好的解決方案，並以戰略性、有紀律的方式執行對該解決方案。要辨識出系統性的問題並找出其相應的解決方案，你需要對應用程式的一或多個廣泛部分有深入的瞭解。你還需要高度耐力，才能將解決方案正確傳播到整個受影響的區域。

大規模重構還需與即時系統的重構密切配合。我們許多人所參與的應用程式都有頻繁的部署週期。在 Slack，我們每天向使用者發佈新的程式碼大約十數次。我們必須注意我們的重構工作如何融入這些週期，以將風險和對用戶的干擾降至最低。瞭解如何在重構過程中於不同時間點進行策略性部署通常可以達成安靜的推展，而不至於完全中斷服務。

Simon’s Dry Cleaners 考慮大規模重構的時候會是什麼樣子呢？假設部署銷售點系統得以大幅優化業務，事實上，好到在短短兩年時間內，它就成功地在鄰近的城鎮中開設了五家新分店！現在，在多個地點營運，業務規模不斷擴大，他們遭遇到一系列不同的問題。為了保持低成本，他們的六個地點中只有兩家實際設有乾洗設備。當客戶在沒有現場乾洗設備的四個地點之一卸下要乾洗的衣物時，這些服裝必須透過公司貨車送到最近的設施。該貨車會在四個店面停留，收取髒衣物，運送到兩個乾洗場所裝卸地點的大箱子裡。Simon’s 的員工會努力地整理一堆衣物，清理它們，然後把它們歸還給正確的店面。然而，大多數時候，這是一個令人怵目驚心的過程。兩個乾洗地點都得處理它們自己店面收受的衣物，還有從四家較小店面送來的衣物。貨車司機將它們丟進加工箱時，衣服會被分離或彼此交纏，這種情況並不少見。更緊急的訂單往往遺失在成堆的訂單裡，而清潔工必須在整批貨物中挖掘，才能找出它們。

Simon’s 如何更有效地改進其營運？是否應為每個地點設立一個專門的乾洗中心，使得每個設施最多只需要處理三個店面的訂單？若是如此，它是否應考慮以特定方式改變貨車的運送路徑？如果兩者兼而有之呢？若是它能藉此縮短處理時間，那麼再開設一個乾洗地點是否符合成本效益？它應該如何設置卸貨台，少纏幾件衣服？能否讓司機在出發前，先按緊急程度對訂單進行適當的分類和歸位？公司是否應該在緊接午餐時間後和關

店不久後限制提貨，讓乾洗地點有更多時間組織收到的衣物？有不少選項需要思量，其中有許多可以多筆訂單結合在一起執行，或同時進行。想像一下，面對著所有的這些可能性，而且必須決定先拉哪個槓桿。它肯定會陷入癱瘓！事實上，重構大型應用程式的時候，感覺也是一樣的。

為何要關心重構？

重構在理論上可能聽起來很有說服力，但你怎麼知道閱讀本書的其餘部分會不會浪費時間呢？我當然希望所有讀者讀完本書後，都能帶上幾項新工具，但若要提供單一理由，說服你繼續讀下去，那會是：

對於你的重構能力有信心，將使你傾向於採取行動，並更快開始構建一個系統，早在你對所有可動的組成部分、陷阱和邊緣案例都形成深入的瞭解之前。如果你知道你有能力在整個開發過程中識別出有效改善組成元件的機會，並且隨著系統變得更複雜時都還能繼續做到這點，那你就不需要提前花那麼多時間去架構一個程式。一旦你練就了不費吹灰之力就能操作程式碼所需的技能，你就不會再花費那麼多時間為任何單一設計決策而煩惱。寫程式的時候，你會發現自己選擇寫一些在當前環境下能夠奏效的簡單東西，而不是退後一步，規劃接下來的數個動作。你會意識到，總有（雖然有時很棘手）一條更好的解決途徑。

寫程式不是下棋。給定一組棋盤配置並假設最佳對手時，最有競爭力的玩家能在幾分鐘內敏捷地完成幾十場比賽。遺憾的是，在我們這一行中，並沒有提供一組完全列舉出來的可能動作，也沒有預先決定的結束狀態。我並不是說，在合理的需求之下，坐下來腦力激盪，為一個問題提出穩健的解決方案是沒有價值的，我要提醒你的是，不要花費大量的時間只為了把最後 10% 到 20% 的部分完美解決。如果你已磨練出重構的能力，你將能夠適時演進你的解決方案，以因應最終的規格需求。

重構的好處

除了能夠有信心地更快開始解決問題之外，重構還能帶來一些切實的好處。雖然它可能不是解決所有問題的正確工具，但它確實能對你的應用程式、工程團隊和更廣泛的組織產生持久而正面的影響。我們討論兩大好處：提升開發人員的生產力，並讓找出臭蟲的工作更為輕鬆。雖然有些人可能認為重構所帶來的好處，比這裡討論的要多得多，但我主張它們都可以歸結為這裡所介紹的兩大主題。

開發人員生產力

重構的主要目標之一是生成易於理解的程式碼。在你進行推理時簡化難懂的解決方案，不僅有助於更好地瞭解程式碼正在執行什麼操作，也能為之後接手你的每個人帶來同樣的好處。可以輕鬆理解的程式碼對於團隊中的所有人絕對都有益，無論他們的任期或經驗水準是如何。

如果你是團隊中的專職工程師，你通常會對源碼庫的某些部分非常熟悉，但是，隨著源碼庫的增長，你會對越來越多的部件感到陌生，而且你的程式碼也越來越可能對這些部件產生依賴關係。想像你正在實作一項新功能，而在將解決方案編入整個系統時，你從你非常熟悉的程式碼跳到不熟悉的領域探險。如果你所不瞭解的區域得到妥善維護，並定期進行重構以因應不斷變化的產品需求和缺陷修復，你將能縮小進行變更的理想位置，並且更快地憑直覺看到輕鬆的解決方案。但如果該程式碼隨著時間的推移而劣化，因為累積了許多不完善的臭蟲補丁，而且長度不斷膨脹，你就得花費更大量的時間瀏覽過每行程式碼，先試著理解程式碼要做什麼以及它如何做到，然後才能再花點時間推理出可接受的解決方案（將別人拖入這折磨人的程式碼「兔子洞」並不罕見，不管那是與你協作的另一名工程師，還是熟悉那些程式碼到足以回答你問題的工程師）。

源碼庫熟悉度的演進過程

對於只有少數幾個工程師維護的小型的源碼庫，工程師大部分都對源碼庫的每個部分非常熟悉的情況並不罕見。隨著更多模組的添加和修改，工程師開始專業分工，熟悉度就會逐漸降低，最終，源碼庫會到達一個臨界質量，任何一位工程師（甚至是第一位雇用的那名！）都不可能熟悉一切。

讓我們反轉這個場景。如果另一個團隊不熟悉你團隊程式碼的一名同事被迫試著閱讀該程式碼，情況會如何？他們能輕鬆瞭解它的工作原理嗎？你比較可能預期看到問題和困惑的表情，還是程式碼審閱的請求？

如果你是團隊裡的新工程師呢？也許這是你最近的情況，又或者你最近把某人帶入你的團隊，而你可以從他身上汲取經驗。他們對於源碼庫完全沒有任何心智模型可言。他們對程式碼任何區域取得信心的能力正比於程式碼的可讀性。他們不僅能夠有機地建置出源碼庫中不同單元之間關係的精確心智表徵，而且還可以解釋程式碼在做什麼，無須向隊友提出問題。（值得注意的是，知道何時可以向同事提問，以及如何提問，是一項必須磨練的重要技能。在尋求幫助之前，學著評估自己需要多長時間來建立自己的理解，

然後再去請求協助，是很困難的，但這對於身為開發者的成長來說是很關鍵的。詢問問題並不是壞事，但如果你是團隊中的專職工程師，並感受到了問題的轟炸，或許就該是時候編寫一些說明文件，並重構一些程式碼了。）

我們在開發新東西時都傾向於複製既定模式。如果我們引用的解法是清晰且簡潔的，我們就更有可能傳播清晰和簡潔的程式碼。反過來也為真：如果我們參考的唯一解決方案雜亂無章，我們將傳播凌亂的程式碼。確保最佳模式是最普遍的那種模式，是剛起步的開發人員建立正向回饋循環的關鍵所在。如果他們定期互動的程式碼易於理解，他們也會在自己的解決方案中模仿類似的重點。

辨識出臭蟲

追蹤並解決臭蟲是我們工作中必要（而且有趣！）的一部分。重構可以是完成這兩項任務的有效工具！通過將複雜述句（statements）分解成可一次消化的較短片段，並將邏輯提取到新函式中，你既可以更好地理解程式碼在做什麼，也（希望）可以隔離出臭蟲。在積極撰寫程式碼的同時進行重構，也能讓你在開發過程初期就輕易發現臭蟲，從而完全避免這些錯誤。

請考慮一個情景，你的團隊在幾個小時前將一些新的程式碼部署到生產環境中，其中有一些變更是嵌入到大家都不敢修改的幾個檔案中：那些程式碼根本不可能讀懂，而且包含大量等著發作的臭蟲雷區。不幸的是，你的測試並沒有涵蓋到多個邊緣案例中的一個，而客戶服務部門的人開始前來尋求協助，指出用戶正遭遇某個討厭的臭蟲。你和你的團隊立即開始深入調查，並迅速意識到這個臭蟲，一如預期，位在程式碼中最可怕的部分。幸好，你的隊友能夠一致地再現出問題，而且和你一起寫出一個測試，以驗證正確的行為。現在你得縮小可能有臭蟲的範圍。你採取有條不紊的步驟來拆解雜亂的程式碼：將長長的單行述句轉換為簡潔的多行述句，並將幾個條件式程式碼區塊的內容遷移到個別函式中。最後，你找出了臭蟲所在。現在程式碼已被簡化，你可以快速修復它，執行測試以確認它是否有效，並將修復程式發送給客戶。宣告勝利！

對客戶來說，有時臭蟲只是小麻煩，但有時臭蟲會使客戶完全無法使用你們的應用程式。雖然更具破壞性的錯誤通常需要緊急補救，但你的團隊必須能夠快速解決所有嚴重級別的臭蟲，才能讓用戶滿意。在維護良好的源碼庫中工作可以大大減少開發人員追查和修復錯誤的時間，讓你得以在創紀錄的時間內讓產品正式上線，並為此高興。

重構的風險

儘管重構的好處聽起來很吸引人，但在著手改進你源碼庫的每一吋（或公分）的這個旅程之前，還要考慮一些嚴重的風險和陷阱。我聽起來可能開始像是跳針的唱片，但我仍要重申：重構要求我們能夠確保行為在每次迭代中都保持相同。我們可以編寫一套測試（單元的、整合的、端到端的）來增強我們的信心，確保什麼都沒變，而在確立足夠的測試涵蓋範圍之前，我們不應考慮繼續執行任何重構工作。然而，即使經過徹底的測試，總有微小機會讓東西從縫隙中溜走。我們還必須牢記我們的最終目標：以一種對你和未來與之互動的開發人員來說都很清楚的方式改善程式碼。

嚴重的退化

重構未經測試的程式碼十分危險，非常不鼓勵這樣做。即使是配備有最周全、最精密測試套件的開發團隊仍然會將臭蟲運送到生產現場。為什麼呢？無論大小，每一次變更都會以可衡量的方式破壞系統的平衡。我們盡力讓造成的破壞降到最低，但每當我們改變系統，就有可能導致意想不到的退化（regression）。重構我們源碼庫最為可怕、令人費解的角落時，引入嚴重衰退的可能性尤其令人關切。源碼庫的這些區域之所以會處於當前狀態，正是因為它們經歷了足夠的時間劣化。在成長快速的公司中，它們經常是應用程式運作不可或缺的組成部分，也是最少被測試的。試圖整理這些檔案或函式，可能就像要毫髮無傷走過地雷區一樣，這是可能的，但是非常危險。

發掘休眠的臭蟲

正如重構可以幫助你識別臭蟲一般，它也可能無意中挖掘出休眠的臭蟲（dormant bugs）。在此，我將休眠的臭蟲歸類為最常經由程式碼的結構重組而揭露的一種退化。我們再回頭看看 Simon’s Dry Cleaners。該公司已開始以相同的交貨頻率訂購更大批的清潔用品，以便從供應商那裡獲得更好優惠。不幸的是，主店面後方沒有太多空間存放產品，所以 Simon’s 決定開始在離裝卸門更近的地方堆放貨箱。下了幾週雨後，團隊發現一些離門最近的箱子都濕了，而且分崩離析。屋主注意到後門密封性差，使得水在潮濕的日子容易滲透出來。Simon’s 從未在靠近裝卸門之處存放補給品時遭遇問題，單純因為他們以前從未這樣做過，實施新的儲存模式暴露了其基礎設施中的一個關鍵缺陷，若非如此，他們可能永遠也不會發現這個問題。

範圍蔓延（Scope Creep）

重構可能有點像吃布朗尼：剛吃幾口覺得美味，很容易就一直吃下去，不小心全都吃掉。當你吃完最後一口，會出現一點後悔的感覺，或許還會有一陣伴隨噁心的內疚感。做出準確且區域性的改變時所體驗到大幅度即時改善，是非常令人振奮的！很容易被沖昏頭，讓你允許更改的表面積超出**合理的界限**（*reasonable bounds*）。什麼是合理的界限呢？要視源碼庫而定，這可以是指單一功能區域或相互依賴的一組小型程式庫。理想情況下，重構過的程式碼所做出的修改必須限制在另一名開發人員可以輕鬆審閱的單一變更集合內。

規劃更大型的重構工作，特別是可能需要耗時數月甚至更長時間的重構時，絕對有必要維持一個緊縮的嚴格範圍。在重構少量表面積（幾行程式碼、單個函式）時，我們都會遇到意外的怪異之處；雖然我們可以持續串聯幾個增強功能有效處理這些怪事，但這種方法會在處理的表面積很可觀時變得危險。規劃的重構所觸及的表面積越大，就越有可能遭遇你沒有預料到的問題。這不代表你是一名糟糕的程式設計師，只是更加顯示你是個人類。藉由遵循一個明確的計畫，你減少了導致嚴重衰退或碰到休眠臭蟲的機會，提高了生產力。持續進行且有條不紊的重構工作就已經夠困難了，還碰上會移動的目標，這只會讓它們更難以實現。

沒必要的複雜性

要小心一開始的過度設計，並容許修改初始計畫的可能性。最主要的目標應該是產生對於人類友善的程式碼，即使代價是犧牲你原本的設計。如果把注意力集中在解決方案而不是流程上，那麼你的應用程式最終就更有可能比當初設計的更不自然且複雜。各層次的重構都應該是迭代（iterative）的。藉由在單一方向上謹慎地小步前進，並在每次迭代中都保持現有行為，你就比較能夠把焦點維持在你終極的目標上。如果只處理螢幕放得下的程式碼，而不是一次面對三十多個程式庫，這就會簡單得多。我們計劃一個新專案時，大多數人通常都會竭盡全力制定詳細的規格文件和執行計畫。即使進行了大量的重構工作，重點還是要很清楚最終產生的程式碼在完成時應該是什麼樣子。

何時需要重構

單純說「當好處勝過於風險時」是很容易的，但這不是一個有用的答案。是的，在實務上，當益處大於風險時，重構就是一種值得努力的工作，但我們如何適當地賦予拼圖的每一塊正確的權重呢？我們如何知道何時到達了臨界點，應該考慮進行重構了呢？

根據我的經驗，說是臨界點（tipping point）倒不如說更像是一個臨界**範圍**（tipping *range*），而且對於每個人和每個應用都會有所不同。判斷這個範圍的上界和下界讓重構有點像是一門主觀科學：沒有公式可以讓我們給出果斷的「是」或「否」答案。幸運的是，我們可以仰賴他人的實證經驗來引導我們做出自己的決定。

小範圍

若想重新設計經過良好測試的程式碼中簡單明瞭的一小部分，應該不會有任何阻礙存在。除非你不確定你重構過的解決方案在客觀上是否比其前身更好，或者你擔心改變會影響到過大的表面積，否則這很可能是值得的努力。小心地打造幾個 commits，並讓你做的改變推展開來！我們將在本章後面部分看到一個明顯屬於此類的例子。

程式碼的複雜性不斷阻礙開發

有些時候，我們必須冒險進入源碼庫我們害怕的部分。每當讀過那些程式碼，我們的眉頭就開始深鎖、心臟跳得更用力、神經元也開始放電。然後就是我們不得不咬緊牙關、鑽進去，做出我們想要的變更之時刻。但在脅迫下進行開發，是無意中引發更多問題的一種必然途徑。當你過度專注於做正確的事情，把問題的許多維度都同時放在腦海中，你就有對**實際**目標失焦的風險。若你的心思落在他處，你要如何充分執行那個目標呢？

如果程式碼的那個部分尚未咬我們一口，我們通常會冒這個險，努力一試。如果它已經攻擊了我們或隊友（有時不止一次），那麼現在為了防止未來失誤而把手術刀直指那段程式碼所涉及的風險，可能會超過讓它繼續停留在當前狀態的風險。如果你不確定天秤傾向哪一邊，那就和隊友談談，並收集一些資料，查看過去六個月內捕捉到的臭蟲，看看有多少可以追溯到源碼庫的這個部分。

產品需求的轉變

產品需求的劇烈變遷往往會表現在程式碼的大幅改變上。儘管我們奮力為應用程式中的每個功能編寫抽象、可擴充的解決方案，但我們仍然無法預測未來，而雖然我們的程式碼可能有辦法輕易適應小型偏差，但它們很少能完全適應大型偏差。這些轉變給了我們一個難得與業務相關的機會，讓我們回到白板前，重新構思自己的設計。

你可能在想，這些轉變不可能保留原本的行為。在相同的輸入下，現在我們必須提供不同的輸出！這為什麼是進行重構的好機會呢？如果你程式碼目前的狀態，沒辦法很好地適應新的需求，你就必須找到一個解決方案以繼續支援當前的功能，並且無縫支援未來

的功能。可以先對你的程式碼進行重構，然後（而且也只有到此時才能！）在其上實作新功能。如此，你就可以繼續設下高品質程式碼的標準，兌現重構的所有好處，同時支援業務目標。再說一次，這是勝利！勝利！勝利！

效能

提升效能可能是一項艱鉅的任務，你必須先深入瞭解現有的行為，然後識別出可以拉哪些槓桿來使天秤傾向好的那邊。從一個乾淨的平台（或作為第一步，自己打造一個）開始著手，最有利於你做到這一點。妥善分離出你找到的槓桿，使其更易於操作，而且沒有產生下游效應的風險，也是關鍵。

並非所有開發人員都認為增進效能是重構的有效理由之一，有些人認定一個系統的效能本質上是其行為的一部分，因此以某種方式改變它也會更動行為。我不同意。如果我們繼續使用我們的重構定義，其中為通用系統提供一組輸入，它就能持續產生預期的一組輸出，那麼改善生成那些輸出的速度（或減少記憶體負擔）也會是重構的有效形式。

為此目的進行重構有一個重要的獨特之處：它並不能確保結果產生的程式碼更容易親近。有時，我們讀過源碼庫時，會遇到很長的註解區塊，對其下方的程式碼發出警告。根據我的經驗，這些註解大多是要提醒讀者對一（或更多）種複雜情況保持警惕：奇怪的應用程式行為、臨時的變通方法，以及怪異的效能補丁。這些「簡短故事」所提示的大部分效能改良都是巧妙編寫的，利用到對源碼庫的深入理解，以將受影響的表面積最小化。這些「改良」更容易在較短時間內出現退化現象，因此不是重構旨在促進的可持續發展性（sustainability）的好例子。值得一試的效能改良，也就是能歸類在重構大傘之下的那些，是深刻而影響深遠的，它們是大規模部署的有效重構的好例子。我們將在第二部中更深入介紹這些變更。

使用新技術

在軟體開發領域，我們經常採用新技術。無論是為了跟上業界最新趨勢、增強擴展規模以服務更多用戶的能力，還是以新的方式使我們的產品成熟，我們都在不斷評估新的開源程式庫、協定、程式語言、服務提供商等。做出決定使用新東西並不是輕而易舉的事，這有部分是因為與現有源碼庫進行整合所需的成本。如果我們選擇以新的解決方案**取代**（*replace*）現有解決方案，我們就必須制定棄用計畫（deprecation plan），方法是識別出所有受影響的呼叫點（callsites）並遷移它們（有時是一次一個）。如果我們選擇

採用（*adopt*）一項持續發展的新技術，就必須找出高槓桿的候選產品，以便盡早採用，並計畫將利用率擴展到所有相關使用案例。

我不會列舉出運用新技術會影響系統的每一種方式（有很多種方式），但從這兩種場景可以看出，每種方式都需要對當前的系統進行仔細的審核。幸運的是，稽核可以揭示重構的最佳機會！我想花些時間告知這是一個有爭議的觀點。因為單是採用新技術就有風險，其他開發人員可能會勸阻你做出任何額外的改變。然而，我堅信，為你的系統引入新事物的最糟糕方式就是把它直接塞入，與龐大、交雜的混亂並存。為了給它最好的機會去實現它的用途，我認為最好花時間去清理它會最先接觸到的區域。

我們可以輕易將這個概念套用到 Simon's Dry Cleaners。假設它最近訂了一台最先進的環保型乾洗機。在制定安裝計畫時，業主們意識到他們現有的樓層規劃有嚴重效率不彰之處。員工們必須沿著排成一條長龍的機器，走將近三十英尺遠，才能到達機架，拿起預先準備好的服裝。如果他們重新調整機器的方位，讓員工只需走幾英尺就能抵達機架，他們就能在每個週期中少花幾分鐘。他們決定以修改後的配置安裝新的機器。藉此，Simon's 可能降低對環境的不良影響，並提高僱員的生產力。雙贏！

何時不要重構

對於開發人員來說，重構可以是非常有用的工具。許多開發人員相信，花在重構上的時間總是值得的，但實際上並沒有這麼簡單。重構需要適當的時機及合適的場合，最成熟的開發者都明白，知道何時需要重構，而何時**不要**，是很重要的。

想找樂子或覺得無聊時

閉上眼睛一分鐘，想像自己坐在電腦前。你正在看一個特別糟糕的函式。它太長了，試圖做太多事情。它的名稱早已沒辦法有意義地描述它的責任。你渴望去修復它。你很想把它分成定義清晰、簡潔的單元，裡面會有更好的變數名稱。這會很**有趣**。但這是你現在能做的最重要的事情嗎？也許你的隊友已經等你審閱程式碼好幾天了，又或者你一直在拖延，不去寫一些測試？如果你深入研究一些粗陋的舊有程式碼並且把玩它，只是為了娛樂自己，你或許是在為自己（以及你的隊友）幫倒忙。

如果你是為了好玩，很有可能你並沒有注意到你所做的改變會對周圍的程式碼、整個系統和同事造成什麼影響。我們為了樂趣而重構時，都有不同的動機存在：我們更有可能使用牽強附會的語言功能，或者嘗試一種一直想試試看的全新模式。另外還有時間和空

間讓我們去嘗試新事物、去伸展我們寫程式的肌肉，但重構並非那種時機。重構應該是一個經過深思熟慮的過程，在此過程中，焦點應該嚴格限定在提供（理想上）最小的變化以達成最大的正面效應。

因為你剛好碰上

想像這個畫面：你寫了一些程式碼，並讓它正式上線，然後開始研發一項新功能。幾個月後你回到你的那段程式碼，要擴展功能。不幸的是，它看起來和你最初寫的東西完全不同。上百萬個問題在你腦海裡湧現。這裡發生過什麼事？

你可能會成為「剛好經過的重構員（drive-by refactorer）」的獵物。這種同事經驗豐富，對如何寫程式有些真知灼見。他們是其他工程師諮詢設計決策的人。他們還有一種不幸的傾向：在遇到時改寫他人程式碼。他們認為這麼做對每個人都有好處。

你可能會傾向於同意，但想想這一點：如果這種工程師在源碼庫某個區域中修改了程式碼，而他們不是那個區域的活躍貢獻者，那麼他們很有可能會降低負責該區域的人的工作效率。我們熟悉自己負責的程式碼時，最有生產力。當我們的任務是快速解決一個問題，不管它是生產過程中的嚴重事件，還是一個小臭蟲，我們都會使用我們思維中的程式碼模型來縮小問題可能出現的檔案、類別或函式。如果我們打開編輯器時發現，東西都跟我們上次離開時不一樣了，我們會迷失方向，無法盡快解決該問題。這在工時、客戶服務時間以及可能失去的業務方面都為我們的雇主帶來了很大的成本。

沒有向原作者說明你做了重構，在兩個不同的方面都是不好的事情。首先，他們嚴重侵蝕作者的信心。不管我們多麼努力試著與我們的程式碼一刀兩斷，我們卻總是對我們所編寫的程式碼保有一點點個人自豪感和所有權。我更希望有人能坦誠地告訴我，我的解決方案有何缺點，並告訴我如何修正它，而不是在問題已經被解決後發現它們。對於新進工程師來說，這尤其有害。想像你剛從學校畢業一年，某天你上班的時候發現你花了數週的時間拼湊出來的程式碼，已經在幾個小時內被一位更資深的工程師重新寫了，而且你從未跟他說過話。這感覺真的不太好。

其次，他們可能不清楚當初編寫時圍繞在該程式碼四周的初始情況。如果面對的程式碼並不是由「剛好經過的重構員」所主動維護的，這尤其麻煩。為什麼這很重要？程式設計全都與衡量取捨有關，我們可以運用佔據更大量記憶體的資料結構來寫出更快的解決方案，或者改用近似計算而非精確計算來減少記憶體佔用空間。每行「壞」程式碼也同樣都是要試圖解決某個問題。如果盲目地重構它，你可能招惹原作者小心翼翼避開的臭蟲或陷阱。

有關原本的程式碼作者

遺憾的是，並不是每個人在職業生涯中都待在同一家公司。程式碼的原作者可能不再屬於你的團隊。幸運的是，人們傾向於在前往更好的工作機會前，把相關資訊移交給他們的團隊，因此你很有可能可以從某個團隊獲得關於那段程式碼的更多資訊。

不要當個「剛好經過的重構員」，做一個**用意良善**的重構員。不要去重構不是你負責維護的程式碼，如果真的要那麼做，請確定你有先跟負責那段程式碼的程式設計師討論過。

為了讓程式碼更容易擴充

許多重構專家提倡重構作為使程式碼更易於擴充的一種手段。儘管這可能是良好重構的明確結果，但為了未來的可延續性而改寫程式碼可能並不明智。在沒有把握得到立即且切實的勝利之情況下，花費在重構上的時間可能是一種心力的浪費，你的變更可能不會在相對短的時間內得到回報，在絕對最糟糕的情況下，可能在程式碼的生命週期內都不會發生。

如果你能夠對某個程式碼區塊進行適當的變更來推進你的專案，你大概就不應對其進行重構。大多數公司都有新的功能要開發，而且有臭蟲尚待修補。一般來說，這些幾乎總是有最高的優先序。除非您有一套具體的目標，並且有令人信服的理由認為這會直接影響公司的利潤，否則你的管理階層將無法被說服。但別失望！我們會在接下來的章節中協助你建立一個重構的商業案例。

你沒有時間的時候

唯一比急需重構的程式碼更糟糕的，是重構到一半的程式碼。陷入不確定狀態的程式碼對與之互動的開發人員來說很令人困惑。若沒有明確的時間點指出何時重構完成，它就會造成半永久性的混亂。讀到重構到一半的程式碼時，讀者往往很難分辨出要遵循的方向或實作，尤其是重構的人沒有留下任何註解的時候。你甚至可能錯誤假定哪些程式碼將被長期採用，然後在一個即將被棄用的區塊中實作某項必要變更。這類錯誤會迅速堆積起來，導致更快速、更嚴重的程式碼腐蝕現象，侵蝕你一開始希望改善的那些部分。

決定開始重構某些東西時，請確保你有足夠的時間推動計畫直至完成。如果不能，請嘗試縮小你的更改範圍，以便你仍然可以做出一些改進，但可以輕鬆抵達終點線。不完整的重構所帶來的任何暫時性好處，都絕對不會超過未來開發人員與它互動時的困惑和挫折感。

我們的第一個重構範例

現在，我們已經建立了堅實的基礎，以開始理解重構的目標，以及在正確的環境下，它如何使我們成為更好的程式設計師，讓我們透過一個小小的例子使這一切更為生動。這個例子比我們會在本書中談到的那種重構工作範圍小很多，但是它幫助我們將一些概念以較小的尺度來闡釋，以便我們及早熟悉它們。

假設我們在一所大學工作，在那裡我們開發並支援一個簡陋的程式，助教們（TA）用此程式送出作業成績。TA 們使用該程式來驗證作業成績是否落在教授指定的某個範圍內。此範圍是可配置的，因為教授設計的作業結構都不同，因此並非所有問題集都按 0 到 100 的點數級別進行評分。舉個例子，假定有包含 10 個問題的一個問題集。每個問題最多值 6 分。如果正確回答所有問題，最終的成績會 60 中的 60 分。如果沒有交出作業，你會得到 0 分。

教授們使用相同的工具以確保指定作業的平均分數落在預期範圍內。給定我們之前的例子，假設這位教授希望將該問題集的平均設定在 42 到 48 點之間（百分比分數則介於 70% 到 80% 之間）。他們可以提供此預期範圍給這個程式，然後程式會處理最終成績並判斷平均值是否位在這些界限之內。

負責此邏輯的函式稱作 `checkValid`，並顯示在範例 1-1 中。

範例 1-1　一個令人困惑的小型程式碼範例

```
function checkValid(
  minimum,
  maximum,
  values,
  useAverage = false)
{
  let result = false;
  let min = Math.min(...values);
  let max = Math.max(...values);
  if (useAverage) {
```

```
    min = max = values.reduce((acc, curr) => acc + curr, 0)/values.length;
  }

  if (minimum < 0 || maximum > 100) {
    result = false;
  } else if (!(minimum <= min) || !(maximum >= max)) {
    result = false;
  } else if (maximum >= max && minimum <= min) {
    result = true;
  }
  return result;
}
```

我們馬上就可以發現一些問題。第一，函式名稱並沒有完全捕捉到它所負責的工作。對於具有泛用名稱的函式，例如 `checkValid`，我們無法確定應該對此函式有何期望（特別是函式宣告頂端沒有任何說明文件時）。第二，並不清楚行內值 `(0, 100)` 代表什麼。根據我們對函式預期行為的瞭解，我們可以推斷這些數字代表任何作業允許的絕對最小和最大點數值。在此情境中，最小值為 `0` 是合理的，但為什麼要堅持上限為 `100` 呢？第三，其邏輯很難理解，不只是要推理的條件有不少個，行內的邏輯也可能很複雜，使我們很難快速地推理每一種情況（case）。乍看之下，幾乎不可能知道這個函式是否包含錯誤。我們可以花相當長的時間來列舉這些簡短的程式碼行中包含的許多問題，但是為了簡單起見，我們就停在這裡。

這麼少行的程式碼怎麼會如此難以理解呢？持續開發中的程式碼會定期修改，以處理小而低衝擊的變化（錯誤修復、新功能、效能調整等）。不幸的是，這些修改會累積，常常導致更冗長、更迂迴的程式碼。從程式碼結構中，我們可以識別出在函式最初寫成後可能發生過的兩個變更：

- 對所提供的那組值的平均而非那些值的總和進行範圍驗證的能力。我可以推斷此功能是之後引入的原因有二：`useAverage` 是一個選擇性的 Boolean 引數，預設值為 `false`，這表示現有的某些呼叫點並不預期第四個引數。Boolean 引數是程式碼的一種氣味，我們很快會談到這點。此外，這段程式碼覆寫 `min` 和 `max` 來反映單一的新平均值以方便處理。這表明作者正在尋找最簡單的方法來處理此需求，同時修改最少量的程式碼。