前言

著名的軟體工程師和企業家 Marc Andreesen 曾宣稱「軟體正在吞噬世界（software is eating the world）」（*https://oreil.ly/tYaNz*）。這是 2011 年的事情了，而且這隨著時間的推移變得更加真實。軟體系統的複雜性繼續增長，並且在現代生活的所有面向都可以找到它們的蹤影。立足於這頭貪婪野獸中心的，正是 Python 語言。程式設計師們經常把 Python 列為最喜愛的語言（*https://oreil.ly/RUNNh*），它隨處可見：從 Web 應用，到機器學習，到開發者工具等等更多地方都可以看到它。

不過，並非所有會閃閃發光的東西都是金子。隨著我們的軟體系統變得越來越複雜，要理解我們的心智模型（mental models）是如何映射到現實世界的，變得越來越難。如果不加以控制，軟體系統就會膨脹變大並變得脆弱，從而獲得「舊有程式碼（legacy code）」這一可怕稱號。這些源碼庫（codebases）往往伴隨著這樣的警告：「不要碰這些檔案，我們不知道為什麼，但一碰就壞」，還有「哦，只有某某某懂這些程式碼，而且他們兩年前就為了矽谷的高薪工作離職了」。軟體開發是一個年輕的領域，但是這類陳述應該會讓開發人員和業務人士都感到害怕。

事實是，要想寫出能持續發揮作用的系統，你需要做出慎重的選擇。正如 Titus Winters、Tom Manshreck 和 Hyrum Wright 所說的：「軟體工程是隨著時間推移而整合的程式設計（Software engineering is programming integrated over time）」[1]。你的程式碼可能會存在很長的時間，我碰過的一些專案，其程式碼是在我上小學的那個年代寫的。你的程式碼會持續存在多久？它會比你在當前工作中的任期（或者你維護該專案的時期）更長嗎？幾年後，當有人用你的程式碼建置核心元件時，你希望你的程式碼如何被

1 Titus Winters、Tom Manshreck 與 Hyrum Wright。*Software Engineering at Google: Lessons Learned from Programming over Time*。Sebastopol, CA: O'Reilly Media, Inc., 2020。

人看待？你希望你的繼任者感謝你的遠見卓識，還是喊著你的名字叫罵，因為你給這個世界帶來了麻煩的問題？

Python 是一個美好的語言，但它偶爾也會使未來的建設工作變得棘手。其他程式語言的一些支持者責難 Python「不是生產級（production-grade）的」或「只對原型設計（prototyping）有用」，但事實是，許多開發者只接觸到最表層，而沒有學習編寫強健 Python 程式所需的全部工具和技巧。在本書中，你將學習如何做得更好。你將經歷許多使 Python 變得潔淨且可維護的方式。你未來的維護者會很喜歡使用你的程式碼，因為它最初的設計就是為了讓事情變得簡單。所以，出發吧，閱讀這本書、展望未來，並建置出能長久使用的強大軟體。

誰應該閱讀本書

本書適用於任何希望能以永續且可維護的方式發展他們手頭程式碼的 Python 開發人員。本書並不打算成為你的第一本 Python 教材，我預期你以前寫過 Python。你應該對 Python 的流程控制（control flow）感到熟悉，並且曾經使用過類別（classes）。如果你想找一本更入門的書，我建議先讀 Mark Lutz（O'Reilly）所著的《*Learning Python*》（*https://oreil.ly/iIl2K*）。

雖然我會涵蓋許多進階的 Python 主題，但本書的目標並不是要成為如何使用 Python 所有功能的指南。取而代之，這些功能為更廣大的對話設置好了背景：關於強健性（robustness）和你的選擇將如何影響可維護性。有時，我會討論一些你應該很少使用的策略，如果真的要用的話。這是因為我想闡明強健性的首要原則；了解我們為什麼以及如何在程式碼中做出決定的歷程，比知道在最佳情況下要使用什麼工具更為重要。在實務上，最佳情況是很少發生的。使用本書介紹的原則，從你的源碼庫得出你自己的結論。

本書不是一本參考書。你可以說它是一本討論書。每一章都應該是你組織中開發人員的一個起點，讓他們一起討論如何最好地應用這些原則。發起讀書會、討論小組或餐會來學習，以促進交流。我在每一章中都提出了討論主題，以開啟對話。當你遇到這些話題，我鼓勵你停下來，反思一下你目前的源碼庫。與你的同儕交談，把這些話題當作討論你們程式碼、程序和工作流程狀態的跳板。如果你對 Python 語言的相關參考書感興趣，我衷心推薦 Luciano Ramalho 所著的《*Fluent Python*》（*https://oreil.ly/PVbON*，O'Reilly；第二版於 2021 年底已出版）。

一個系統可以透過許多不同方式變得強健。它可以是安全性強化過（security hardened）的、規模可擴充（scalable）的、容錯（fault-tolerant）的，或者不太可能引入新的錯誤。強健性的這每一個面向都值得寫一整本書來探討；本書的焦點放在如何避免繼承你程式碼的開發者在你的系統中製造出新的錯誤。我將告訴你如何與未來的開發者溝通，如何透過架構模式（architectural patterns）使他們的生活更輕鬆，以及如何在你源碼庫中的錯誤進入到生產環境之前抓住它們。本書將重點放在你 Python 源碼庫的強健性上，而非你系統整體的強健性。

我將涵蓋豐富的資訊，源於電腦軟體的許多不同領域，包括軟體工程、電腦科學、測試、函式型程式設計（functional programming）和物件導向程式設計（object-oriented programming，OOP）。我並不預期你有這些領域的背景。在某些章節中，我以初學者的水準來解釋事情；這通常是為了解構我們思考此語言核心基本原理的思維方式。也就是說，本書絕大部分都是中階水平的內容。

理想的讀者包括：

- 目前在大型源碼庫中工作的開發人員，希望找到更好的方法與同事交流
- 源碼庫的主要維護者，正在尋找可以幫忙減輕未來維護者負擔的方法
- 自學出身的開發者，他們可以把 Python 寫得非常好，但需要更深入了解為什麼我們要做這些事情
- 需要人提醒實用的開發建議的軟體工程畢業生
- 在找方法想要將他們設計背後的原由與強健性的第一原則聯繫起來的資深開發人員

本書的重點是撰寫會隨著時間推移而演進的軟體。如果你的很多程式碼都只是原型、用完即扔的，或以其他任何方式一次性使用的，那麼本書的建議最終會導致超出你專案原本所需的更多工作。如果你的專案很小，例如說，少於 100 行的 Python 程式碼，也會是如此。要讓程式碼變得可維護，確實會增加複雜性，這一點是毫無疑問的。然而，我將指引你把這種複雜性降到最低。如果你的程式碼壽命超過幾週，或者會增長到相當大的規模，你就得考量你源碼庫的永續性（sustainability）。

關於本書

本書涵蓋的知識面很廣，跨越許多章節。它被分成四個部分：

第一部，以型別注釋你的程式碼

我們將從 Python 中的型別（types）開始。型別是語言的基礎，但不常被詳細研究。你選擇的型別很重要，因為它們傳達了一個非常特定的意圖（intent）。我們將研究型別注釋（type annotations），以及特定的注釋向開發者傳達了什麼。我們還將討論型別檢查器（typecheckers），以及它們如何幫助早期捕獲錯誤。

第二部，定義你自己的型別

在介紹了如何思考 Python 的型別之後，我們將集中討論如何創建你自己的型別。我們將深入了解列舉（enumerations）、資料類別（data classes）和類別。我們將探討設計一個型別時所做出的某些設計抉擇，會如何增加或減低你程式碼的強健性。

第三部，可擴充的 *Python*

在學習了如何更好地表達你的意圖之後，我們將專注討論如何能讓開發者毫不費力地變更你的程式碼，在你強大的基礎上充滿信心地建置其他東西。我們將討論可擴充性（extensibility）、依賴關係（dependencies）和架構模式（architectural patterns），使你能修改你的系統並只帶來最小的影響。

第四部，構建安全網

最後，我們將探討如何建立一個安全網（safety net），這樣你就可以在你未來的合作者被絆倒時輕輕地拉住他們。他們的信心將會增加，因為他們知道自己有一個強大且健全的系統，可以無所畏懼地依據他們的使用案例做調整。最後，我們將介紹各種靜態分析（static analysis）和測試（testing）工具，這些工具將幫助你抓住反常行為。

每章基本上都是自成一體的，並在適當的地方引用其他章節的內容。你可以從頭到尾地閱讀這本書，或者跳到適合你目標的章節。被分組在每個部分中的章節是相互關聯的，但書中各部分之間的關聯性會比較少。

所有的程式碼範例都是使用 Python 3.9.0 執行的，若需要某個特定的 Python 版本或更高版本來執行範例時，我會試著特別標明（例如用於資料類別的 Python 3.7）。

在本書中，我將在命令列（command line）上完成大部分的工作。我在 Ubuntu 作業系統上執行所有的這些命令，但大多數工具在 Mac 或 Windows 系統上應該也能正常運作。在某些情況下，我將展示某些工具如何與整合式開發環境（integrated development environments，IDE）互動，例如 Visual Studio Code（VS Code）。大多數 IDE 在幕後都是使用命令列選項；你在命令列上學到的大部分內容將直接轉化為 IDE 選項。

本書會介紹許多不同的技術，可以提高你程式碼的強健性。然而，軟體開發中沒有一勞永逸的銀彈（silver bullets）。權衡取捨是堅實工程的核心，在我介紹的方法中也不例外。在討論這些話題時，我會清楚指出好處和壞處。你比我更了解你的系統，你最適合挑選哪種工具適合哪項工作。我所做的只是儲備、充實你的工具箱。

本書編排慣例

本書使用下列排版慣例：

斜體字（*Italic*）

代表新名詞、URL、電子郵件位址、檔名和延伸檔名。中文以楷體表示。

定寬字（`Constant width`）

用於程式碼列表，還有正文段落裡參照到程式元素的地方，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。

定寬粗體字（**`Constant width bold`**）

顯示應該逐字由使用者輸入的命令或其他文字。

定寬斜體字（*`Constant width italic`*）

顯示應該以使用者所提供的值或由上下文決定的值來取代的文字。

這個元素代表訣竅或建議。

這個元素代表一般註記。

第一章

強健的 Python 簡介

本書主題是如何使你的 Python 更容易管理。隨著你源碼庫成長，你需要一個由訣竅、技巧和策略所組成的特定工具箱，來構建可維護的程式碼。本書將指導你如何減少錯誤，成為更加快樂的開發者。你將認真審視自己的程式碼編寫方式，並了解你的決定所帶來的影響。討論如何編寫程式碼時，我想起了 C.A.R. Hoare 的這些至理名言：

> 建構軟體設計的方法有兩種：一種方法是使其簡單到一看就知道沒有缺陷，另一種方法是使其複雜到看不出有缺陷。第一種方法要難得多[1]。

本書談論的是如何用第一種方法開發系統。沒錯，這將會更困難的，但不要害怕。我是你提高 Python 水準旅程中的嚮導，會帶領你讓程式碼一看就知道沒有缺陷（*there are obviously no deficiencies*），正如 C.A.R. Hoare 在上面所述。歸根究柢，這就是一本關於如何撰寫強健（*robust*） Python 的書。

在這一章中，我們將討論強健性（*robustness*）的含義，以及為什麼你應該在意它。我們將討論你的溝通方式隱含著哪些好處和壞處，以及如何以最佳方式表達你的意圖。The Zen of Python（*https://oreil.ly/SHq8i*）指出，開發程式碼時「應該用一種明顯的方式來進行，而且最好只用一種」。你將學會如何評估你的程式碼是否以明顯的方式編寫，以及你可以做什麼來修補它。首先，我們需要解決一些基本問題。那麼，什麼是強健性呢？

1 Charles Antony Richard Hoare. “The Emperor’s Old Clothes.” *Commun. ACM* 24, 2 (Feb. 1981), 75–83. *https://doi.org/10.1145/358549.358561*.

強健性

每本書都至少需要一個字典定義，所以我要及早將之清楚列出。Merriam-Webster 字典為 *robustness*（*https://oreil.ly/2skKO*）提供了許多定義：

1. 具有或展現出強度或生機勃勃的健康狀態
2. 具有或表現出活力、強度或堅實度
3. 牢固地被建構出來或形成
4. 能夠在廣泛的條件範圍之下正常工作而不發生故障

這些都是對我們應該追求的目標之精彩描述。我們希望有一個**健康**的系統，一個歷經多年都還是能符合期待的系統。我們希望我們的軟體**展現出強度**，而且很明顯地，這種程式碼應該要經得起時間的考驗。我們希望有一個**結構強大**的系統，一個建立在堅實基礎上的系統。最重要的是，我們想要一個**能夠順利運行，不會有故障**的系統，這種系統不應該因為引入變化而變得脆弱。

人們通常認為軟體就像一座摩天大樓，有一些宏偉的結構作為堡壘，抵禦一切變化，是不朽的楷模。遺憾的是，事實是更加混亂的。軟體系統不斷演進。修補漏洞、調整使用者介面，功能新增又刪除，然後重新添加。框架變遷、元件過時了，安全性臭蟲也出現了。軟體會改變。軟體開發更類似於處理城市規劃中的無序擴張，而不是建造一座靜態的建築物。在不斷變化的源碼庫中，如何能使你的程式碼更強健呢？你怎樣才能建立一個對臭蟲有抵抗力的強大基礎呢？

事實是，你必須接受變化。你的程式碼會被拆開、縫合，並重新加工。新的用例會改變大量的程式碼，而這都沒有問題，擁抱它吧。要明白，你的程式碼光是容易修改是不夠的，在它過時之後，它最好能被刪除和重寫。這並不會削弱它的價值；它仍然會在聚光燈下存活很長一段時間。你的工作是使改寫系統的部分內容變得容易。一旦你開始接受你程式碼的短暫性，你就會開始意識到，僅僅為現在編寫無缺陷的程式碼是不夠的，你還得讓源碼庫的未來擁有者能夠放心地變更你的程式碼。這就是本書主旨所在。

你將學習如何建置強大的系統。這種力量並非來自於剛性（rigidity），就像鐵條所表現出來的那樣。相反地，這源於彈性（flexibility）。你的程式碼需要像一棵高聳的柳樹一樣強大，在風中搖曳，彎曲但不斷裂。你的軟體將會需要處理你做夢也想不到的狀況。你的源碼庫需要能夠適應新的環境，因為負責維護它的不會永遠都是你。那些未來的維護者需要知道他們是在一個健康的源碼庫中工作。

你的源碼庫需要傳達它的力量。你必須以一種能夠減少失誤的方式來編寫 Python 程式碼，即使是在未來維護者將其拆開並重新構建它的過程中也是如此。

編寫強健的程式碼意味著要刻意考慮到未來。你會希望未來的維護者看到你的程式碼時，能輕鬆理解你的意圖，而不是在深夜的除錯過程中咒罵你的名字。你必須傳達你的想法、推理和警告。未來的開發者需要把你的程式碼變成新的形狀，而且他們希望這樣做的時候，不用去擔心每一次的修改是否會使它像一座搖搖欲墜的紙牌屋一樣坍塌。

簡單地說，你不希望你的系統失敗，特別是在意外發生時。測試（testing）和品質保證（quality assurance）是其中的重要部分，但這兩者都無法完全烘培出高品質，它們更適合用來揭示與期望的差距，並提供一個安全網。因此，你必須使你的軟體經得起時間的考驗。為了做到這一點，你必須編寫**潔淨且可維護**（*clean and maintainable*）的程式碼。

潔淨程式碼（clean code）能清晰並簡潔地表達其意圖，並依照先清晰再簡潔的順序進行。當你看著一行程式碼時，會對自己說：「啊，這完全合理」，那就是潔淨程式碼的標誌。你越是需要透過除錯器逐步追蹤，越是需要看很多其他的程式碼來弄清楚發生了什麼事，越是需要停下來盯著程式碼思考，它就越是不潔淨。如果那會使其他開發者難以閱讀，那麼潔淨程式碼就不會採用聰明的技巧。就像 C.A.R. Hoare 之前說的那樣，你不會想讓你的程式碼變得如此晦澀難懂，到了實際查看時難以理解的程度。

潔淨程式碼的重要性

擁有潔淨的程式碼對於擁有強健的程式碼而言至關緊要，對於任何實質的專案來說，這都是一個不可或缺的要素。通常會有一些特定的實務做法與編寫潔淨的程式碼相關，包括：

- 以適當的單元組織你的程式碼
- 提供良好的說明文件
- 妥善命名你的變數 / 函式 / 型別
- 保持函式的簡短和單純

雖然潔淨程式碼的主題貫穿本書，但我不會用很多的時間來討論這些特定的實務做法。有一些其他的書籍可以更好地掌握潔淨程式碼的實踐方式。我推薦 Robert C. Martin 所著的《*Clean Code*》（Prentice Hall）、Andy Hunt 和 Dave Thomas 的《*The Pragmatic Programmer*》（Addison-Wesley），以及 Steve McConnell 的《*Code Complete*》（Microsoft Press）。這三本書都大幅提高了我作為一名開發者的技能，對任何希望成長的人來說，都是很好的資源。

雖然你絕對應該努力寫出潔淨的程式碼，但你必須準備好在那些並不是潔淨性之光榮典範的源碼庫中工作。軟體開發是一種雜亂的事業，有時會因為各種原因（包括商業和技術原因）而犧牲潔淨程式碼的純度。利用本書給出的建議，藉由討論可維護性來幫助推動程式碼的清潔。

可維護的程式碼（maintainable code）是指…嗯，沒錯，就是可以輕鬆維護的程式碼。維護工作從第一次提交（commit）後立即開始，一直持續到沒有任何開發人員還在關注該專案為止。開發人員會修復錯誤、新增功能、閱讀程式碼，提取出程式碼用於其他程式庫，諸如此類的。可維護的程式碼使這些任務變得毫無障礙。軟體的壽命長達數年，甚至數十年。現在就要關注你的可維護性（maintainability）。

你不希望成為系統失敗的原因，無論你是否仍然活躍地在它們底下作業。你需要積極主動地使你的系統經得起時間的考驗。你需要一個測試策略來作為你的安全網，但你也必須能在一開始就避免跌倒。因此，考慮到所有的這些，我提供了我對源碼庫強健性的定義：

> 強健的源碼庫（*robust codebase*）即使是在持續不斷的變化之下，都是有彈性（*resilient*）且無失誤（*error-free*）的。

為什麼強健性很重要？

為了讓軟體完成它應該做的事情，我們花費了很多精力，但要知道什麼時候算完成了，並不容易。開發的里程碑並不容易預測。人為因素，如 UX（使用者體驗）、無障礙輔助（accessibility）和說明文件，只會增加複雜性而已。現在再加上測試，以確保你已經涵蓋了已知和未知行為的一小部分，你就會看到漫長的開發週期。

軟體的目的是為了提供價值。如何儘早提供全部的價值是每個利害關係者最在意的事情。鑒於開發時間表的不確定性，通常會有額外的壓力來促使期望被滿足。我們都曾背負過不實際的時間表或最後期限。遺憾的是，許多能使軟體難以置信地強健的工具，在短期內只會增加我們的開發週期。

誠然，在立即交付價值和使程式碼強健之間存在著固有的緊張關係。如果你的軟體已經「夠好」，為什麼還要增加更多的複雜性呢？要回答這個問題，請考量該軟體被反覆修訂的頻率。交付軟體價值通常不是一種靜態的工作，很少有一個系統在提供價值後不再需要被修改的情況。軟體就其本質而言就是不斷演進的。源碼庫需要做好準備，以便頻繁且長期地提供價值。這就是強健的軟體工程實務做法發揮作用的地方。如果你無法在不影響品質的情況下輕鬆且快速地交付功能，你就需要重新評估所用的技巧，以使你的程式碼更容易維護。

如果你延遲交付你的系統，或者交出沒辦法順利運作的系統，你就會招致即時的成本。仔細考量你的源碼庫。問問你自己，如果你的程式碼在一年後因為有人無法理解而導致損壞，會發生什麼事？你會損失多少價值？你的價值可以用金錢、時間、甚至生命來衡量。問問自己，如果價值沒有準時交付會怎樣？會有什麼反響？如果這些問題的答案很可怕，那麼好消息是，你正在做的工作是有價值的。但這也強調了為何消除未來的錯誤是如此的重要。

多名開發人員同時在同一個源碼庫上工作。許多軟體專案會比這些開發人員中的大多數人都還要長壽。你需要找到一種方法來與現在和未來的開發者溝通，因為你沒辦法親自到場解釋。未來的開發者將以**你的**決定為基礎。每一條走錯的路、每一個難以捉摸的舉措和每一次令人分心的冒險（yak-shaving[2] adventure）都會拖慢他們的速度，這就阻礙了價值的實現。你需要同理那些在你之後的人，你需要站在他們的角度思考。這本書是讓你開始思考你的協作者和維護者的入口。你需要考慮永續性的工程實務。你需要寫出能持續存在的程式碼。編寫長壽程式碼的第一步是能夠透過你的程式碼進行交流。你需要確保未來的開發者可以理解你的意圖。

2 Yak-shaving（犛牛剃鬚）描述了一種情況，即你經常需要解決不相關的問題，然後才能開始解決原來的問題。你可以在 *https://oreil.ly/4iZm7* 了解這個術語的起源。

你的意圖是什麼？

你為什麼要努力編寫潔淨和可維護的程式碼？為什麼你要如此關心強健性？這些答案的核心在於溝通（communication）。你不是在交付靜態系統。程式碼會不斷地變化。你還必須考慮到維護者也會隨著時間的推移而交替。在編寫程式碼時，你的目標就是提供價值。同時還要把你的程式碼寫得能讓其他開發者也能同樣快速地提供價值。為了做到這一點，你需要在見不到你未來的維護者的情況下，傳達你的推理過程和意圖。

讓我們來看看一個假想的舊有系統（legacy system）中的程式碼區塊。我想讓你估計一下，你要花多長時間才能理解這段程式碼在做什麼。如果你不熟悉這裡的所有概念，或者你認為這段程式碼很複雜（這是故意的！），也沒關係。

```
# 接受一個膳食食譜並透過
# 調整每種成分來改變份量
# 一個 recipe（食譜）的第一個元素是份數，而剩餘
# 的元素是 ( 名稱 , 數量 , 單位 )，例如 ("flour", 1.5, "cup")
def adjust_recipe(recipe, servings):
    new_recipe = [servings]
    old_servings = recipe[0]
    factor = servings / old_servings
    recipe.pop(0)
    while recipe:
        ingredient, amount, unit = recipe.pop(0)
        # 請只使用易於測量的數字
        new_recipe.append((ingredient, amount * factor, unit))
    return new_recipe
```

這個函式接受一個食譜（recipe），並調整每一種成分（ingredient）以應付新的份數（number of servings）。然而，這段程式碼喚起了許多問題。

- `pop` 的用處是什麼？
- `recipe[0]` 代表什麼？為什麼那是舊的份數（old servings）？
- 為什麼我需要為可以輕易測量的數字提供一個註解？

這肯定是有點問題的 Python 程式碼。如果你覺得有必要重寫它，我不會怪你。像這樣寫看起來會好得多：

```
def adjust_recipe(recipe, servings):
    old_servings = recipe.pop(0)
    factor = servings / old_servings
    new_recipe = {ingredient: (amount*factor, unit)
```

```
                for ingredient, amount, unit in recipe}
    new_recipe["servings"] = servings
    return new_recipe
```

那些喜歡潔淨程式碼的人可能更喜歡第二個版本（我當然喜歡）。沒有明顯的迴圈、變數不會變動（mutate）。我回傳一個字典（dictionary），而不是由元組（tuple）所成的一個串列（list）。取決於周遭情況，所有的這些變更都可以被看作是正面的。但是我剛才可能引入了三個細微難察的錯誤。

- 在最初的程式碼片段中，我有清除原本的食譜，而現在我沒有那麼做。即使呼叫端程式碼只有一個區域仰賴這種行為，我還是打破了那段呼叫端程式碼的假設。
- 由於回傳的是一個字典，我移除了在串列中能夠擁有重複成分的能力。這可能會對有多個部分（如一道主菜和一個醬汁）的食譜產生影響，因為這些部分都使用相同的原料。
- 若有任何成分被命名為「servings」，那我就引入了一個衝突的名稱。

這些是否為錯誤取決於兩個相互關聯的東西：原作者的意圖和呼叫端程式碼。作者打算解決一個問題，但我不確定他們為什麼要以這種方式撰寫程式碼。為什麼他們要 pop 出元素？為什麼「servings」是串列裡的一個元組？為何要用串列？據推測，原作者應該知道原因，並在當時當地與他們的同儕進行了交流。他們的同儕基於這些假設編寫了呼叫端程式碼，但隨著時間的推移，這個意圖變得越來越模糊。若沒有傳達到未來的訊息，我在維護這段程式碼時就只有兩種選擇：

- 在實作之前，查看所有的呼叫端程式碼，先確認這種行為沒有被依存。如果這是有外部呼叫者的程式庫的公開 API，那只能祝我好運了。我可能要花很多時間來做這件事，這將使我感到沮喪。
- 做出變更，然後等著看後果是什麼（客戶投訴、壞掉的測試等等）。如果我運氣好，可能不會發生什麼壞事。若非如此，我就得花很多時間來修復用例，這同樣會讓我感到沮喪。

在維護的情境之下，這兩種選擇感覺起來都不怎麼有成效（尤其是當我必須修改這段程式碼時）。我不想浪費時間；我想迅速處理好當前的任務，然後繼續下一個任務。如果我考慮到如何呼叫這段程式碼，情況會變得更糟。想一想你是如何與前所未見的程式碼互動的。你可能是看到其他呼叫端程式碼的例子，複製它們並調整以適應你的用例，卻從未意識到你需要傳遞一個叫作「servings」的特定字串來作為你串列的第一個元素。

這些都是會讓你大感不解的決定。我們都在較大型的源碼庫中見過它們。它們並不是惡意編寫的，而是隨著時間的推移，在最好的意圖之下，隨機應變寫出來的。函式一開始都很簡單，但隨著用例的增加和多名開發人員的貢獻，這些程式碼往往會變形並掩蓋住最初的意圖。這是可維護性正受到不良影響的明顯跡象。在你的程式碼中，你一開始就要先表達意圖。

那麼，如果原作者使用了更好的命名模式和較佳的型別用法呢？這段程式碼會是什麼樣子呢？

```
def adjust_recipe(recipe, servings):
    """
    接受一個膳食食譜並變更份數
    :參數 recipe：一個 `Recipe`，指出什麼需要做調整
    :參數 servings：份數
    :回傳 Recipe：份量跟成分根據新的份數
                  調整過的一個食譜
    """
    # 建立成分（ingredients）的一個拷貝
    new_ingredients = list(recipe.get_ingredients())
    recipe.clear_ingredients()

    for ingredient in new_ingredients:
            ingredient.adjust_proportion(Fraction(servings, recipe.servings))
    return Recipe(servings, new_ingredients)
```

這看起來要好得多，有更好的說明，並且清楚地表達了原本的意圖。原開發者將他們的想法直接編碼到程式碼中。從這個片段中，你能知道下列為真：

- 我使用一個 `Recipe` 類別。這使得我可以取出某些運算作為抽象層。可想而知，在類別本身之中，有一個允許重複成分的不變式（invariant）存在（我將在第 10 章中更詳細談論類別和不變式）。這提供了一個共通的詞彙，使函式的行為更加明確。
- 食材（servings）現在是 `Recipe` 類別的一個明確的部分，而不需要成為串列的第一個元素，作為一種特殊情況處理。這大大簡化了呼叫端的程式碼，並防止了不經意的名稱碰撞。
- 很明顯，我想清除舊食譜上的成分。沒有模棱兩可的理由需要我去做一次 `.pop(0)`。
- 成分（ingredients）是一個單獨的類別，並負責處理分數（fractions，*https://oreil.ly/YxUHK*），而不是一個明確的 `float`。對所有參與的人來說，都可以更清楚看到，我處理的是分數單位（fractional units），而且可以很容易地做一些事情，例如 `limit_denominator()`，當人們想限制測量的單位時，就能呼叫它（而非仰賴註解）。

我使用型別取代了變數，例如一個食譜（recipe）型別和成分（ingredient）型別。我還定義了一些運算（`clear_ingredients`、`adjust_proportion`）來表達我的意圖。透過這些改變，我已經讓程式碼的行為對未來的讀者而言，變得非常清楚。他們不再需要和我討論才能理解這些程式碼。取而代之，他們甚至不用跟我交談就能理解我在做什麼。這就是**非同步通訊**（*asynchronous communication*）的精華所在。

非同步通訊

在一本 Python 書中寫到非同步通訊而不提及 `async` 和 `await` 是很奇怪的事情。但我恐怕得在一個更複雜的地方討論非同步通訊：**現實世界**。

非同步通訊意味著資訊的生產（production）和消耗（consumption）是相互獨立的。生產和消耗之間有一個時間差存在。這可能是幾個小時，如不同時區的協作者的情況。也可能是幾年，例如未來的維護者試圖對程式碼的內部運作進行深入研究之時。你無法預測什麼時候會有人需要了解你的邏輯。他們消耗你所產生的資訊時，你甚至可能已經不在那個源碼庫上工作（或為那個公司工作）。

這與**同步通訊**（*synchronous communication*）形成鮮明對比。同步通訊是現場（即時）交流思想。這種形式的直接溝通是表達你的想法的最好方式之一，但不幸的是，它沒有規模擴充性，你不會總是在身邊回答問題。

為了估算每一種溝通方式在試圖理解意圖時有多合適，讓我們看看兩個軸線：接近度（proximity）和成本（cost）。

接近度是指溝通者需要在多接近的時間內進行交流，以取得溝通成效。有些溝通方式在即時（real-time）資訊傳遞方面表現出色。其他溝通方式則擅長處理在多年後的交流。

成本衡量溝通的努力程度。你必須權衡為溝通所花費的時間和金錢，以及所提供的價值。然後，你未來的消費者必須權衡消耗資訊的成本和他們試著提供的價值。編寫程式碼並不提供任何其他溝通管道是你的基線（baseline），你必須這樣做才能產生價值。為了評估其他溝通管道的成本，這裡列出我會考量的因素：

可發現性（*Discoverability*）

在正常工作流程之外找到這些資訊有多容易？這些知識的存在時間有多短暫？搜尋資訊是否容易？

維護成本（*Maintenance cost*）

資訊的準確度如何？需要多久更新一次？如果這些資訊過時了，會出什麼問題？

生產成本（*Production cost*）

達成這種溝通要投入多少時間和金錢？

在圖 1-1 中，我根據自己的經驗，繪製了一些常見溝通方式的成本和所需的接近度。

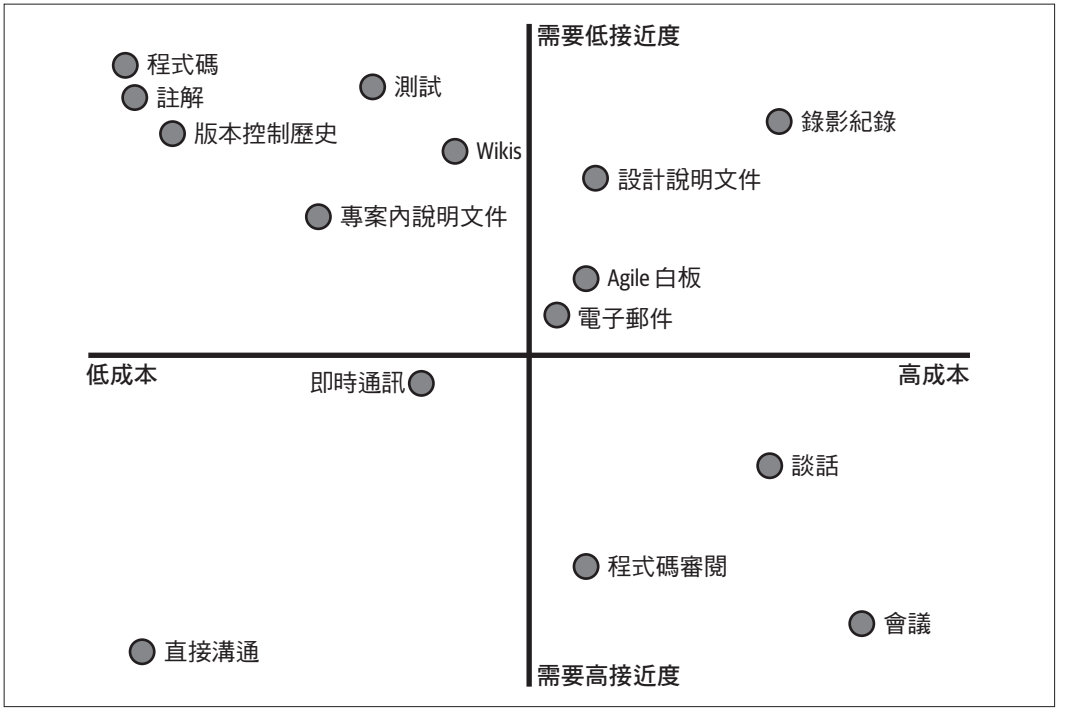

圖 1-1　繪製出溝通方式的成本與接近度

有四個象限構成了成本 / 接近度（cost/proximity）圖。

低成本、所需接近度高

這類方法的生產和消耗都很便宜，但不能跨越時間擴充規模。直接溝通和即時通訊是這些方法很好的例子。把這些當作資訊在某個時間點上的快照（snapshots）；它們只有在使用者主動傾聽時才有價值。不要依靠這些方法來與未來溝通。

高成本、所需接近度高

這些都是昂貴的事件，而且往往只發生一次（如會議或研討會）。這些事件應該在交流時提供大量的價值，因為它們對未來沒有提供多少價值。有多少次你參加了一個感覺是浪費時間的會議？你所感受到的就是價值的直接損失。談話需要每個與會者付出倍數型的成本（花費的時間、舉行的空間、交通物流等等）。程式碼審查一旦完成，就很少有人回顧了。

高成本、所需接近度低

這些都是昂貴的，但由於所需的接近性較低，這種成本可以透過所交付的價值隨著時間得到回報。電子郵件和敏捷白板（agile boards）包含豐富的資訊，但不是其他人可以發現的。這對於不需要經常更新的大概念來說是很好的。但若是得從那所有的雜訊中篩選出你要找的一小塊資訊，這就成了一場噩夢。錄影紀錄和設計說明文件（design documentation）對於理解時間點的快照而言是很好的，但維持最新資訊的成本很高。不要仰賴這些溝通方式來了解日常的決策。

低成本，所需接近度低

這些東西建立起來很便宜，而且可以輕鬆消耗。程式碼註解、版本控制歷史和專案的 README 都屬於這一類，因為它們與我們編寫的原始碼放在一起。使用者可以在這些交流產生的數年後查閱它們。開發人員在其日常工作流程中會遇到的任何東西都是本質上可發現的。這些溝通方式自然適合用在有人會關注原始碼的第一個地方。然而，你的程式碼是你最好的說明工具之一，因為它是你系統的生動紀錄，而且是唯一的真相來源。

討論主題

圖 1-1 中的這個圖是根據一般化的用例所繪製的。想一想你和你的組織所使用的溝通途徑，你會把它們畫在圖的什麼地方呢？消耗準確的資訊有多容易？生產資訊的成本有多高？你對這些問題的答案可能會導致一個稍有不同的圖表，但真理的唯一來源將存在於你所交付的可執行軟體中。

低成本、低接近度的溝通方式是傳達資訊到未來的最佳工具。你應該努力使溝通的生產成本和消耗成本最小化。無論如何，你都得編寫軟體來傳遞價值，所以最低成本的選擇就是使你的程式碼成為你的主要溝通工具。你的源碼庫成為清楚表達你決定、意見和變通方法的最佳選擇。

然而，為了使這一論斷成立，程式碼也必須是消耗起來成本低廉的。你的意圖必須清楚地體現在你的程式碼中。你的目標是儘量減少讀者理解你程式碼所需的時間。理想情況下，讀者沒必要細讀你的實作，只需要閱讀你的函式特徵式（function signature）。透過使用好的型別、註解和變數名稱，你的程式碼做了什麼事，應該是清楚易懂的。

自我說明的程式碼（Self-Documenting Code）

對圖 1-1 的錯誤反應是「自我說明的程式碼就是我所需要的一切！」。程式碼絕對應該自我說明做了什麼事，但這無法涵蓋每一種溝通的用例。舉例來說，版本控制會給你變更的歷史。設計說明文件討論的是整體的理想，並不單屬於任何一個程式碼檔案。會議（如果做得好的話）可以成為同步計畫執行的一個重要事件。談話是與眾多聽眾同時分享想法的好辦法。雖然本書的重點是你可以在你的程式碼中做到什麼，但可別捨棄任何其他有價值的交流手段。

Python 中意圖的例子

現在我已經談過什麼是意圖以及它的重要性，讓我們透過 Python 的視角來看看實際例子。如何確保你有正確表達你的意圖呢？我會帶你看一下決策如何影響意圖的兩個不同例子：群集（collections）和迭代（iteration）。

群集

挑選一個群集時，你就是在傳達特定的訊息。你必須為手頭的任務選出適當的群集。否則，維護者會從你的程式碼中推斷出錯誤的意圖。

考慮一下這段程式碼，它接受烹飪書（cookbooks）的一個串列，並提供了作者和所寫書籍數量之間的一個映射（mapping）。

```
def create_author_count_mapping(cookbooks: list[Cookbook]):
    counter = {}
    for cookbook in cookbooks:
        if cookbook.author not in counter:
            counter[cookbook.author] = 0
        counter[cookbook.author] += 1
    return counter
```

我所用的群集告訴了你什麼呢？為什麼我沒有傳入一個字典（dictionary）或一個集合（set）呢？為什麼我不是回傳一個串列（list）呢？根據我目前對群集的使用，你可以這樣假設：

- 我傳入由烹飪書組成的一個串列。這個串列中可能有重複的烹飪書（我可能是在書店裡計算書架上有多本重複的烹飪書）。
- 我回傳的是一個字典。使用者可以查找（look up）一名特定的作者，或者迭代過（iterate over）整個字典。我不用擔心回傳的群集中會有重複的作者。

如果我想表達的是傳入這個函式中的東西不應該有重複（no duplicates），要怎麼辦呢？串列傳達了錯誤的意圖。取而代之，我應該選擇一個集合（set）來表達這段程式碼絕對不會處理重複的內容。

挑選一種群集（collection）告訴了讀者你的具體意圖。下面列出了常見的群集型別，以及它們所傳達的意圖：

List（串列）

這是一種要被迭代過（iterated over）的群集。它是**可變**（*mutable*）的：任何時候都能被變更。你很少會想從串列的中間位置取回特定的元素（使用一個靜態的串列索引）。可能會有重複的元素（duplicate elements）。書店書架上的烹飪書就可以被儲存在一個串列中。

String（字串）

一種不可變（immutable）的字元群集。烹飪書的書名就會是一個字串。

Generator（產生器）

要被迭代過而且永遠都不會被索引的一種群集。每個元素的存取都是惰性（lazily）進行的，所以每次迴圈的迭代可能都需要花費時間或資源。它們很適合用於計算成本昂貴或無限的群集。一個線上食譜資料庫就可以回傳作為一個產生器。當使用者只想看搜尋到的前 10 筆結果時，你不會希望還得先擷取世界上所有食譜才行。

Tuple（元組）

一種不可變的群集。你不預期它發生變動，所以更可能是從元組中間提取特定的元素（透過索引或拆分動作）。它極少被迭代。關於特定一本烹飪書的資訊可能被表示為一個元組，例如 `(cookbook_name, author, pagecount)`。

Set（集合）

一種可迭代（iterable）的群集，不包含重複的內容。你不能仰賴其中元素的順序。烹飪書中的成分可能被儲存為一個集合。

Dictionary（字典）

從鍵值到值（keys to values）的一種映射（mapping）。鍵值在整個字典中是唯一（unique）的。字典通常會被迭代，或使用動態鍵值進行索引。烹飪書的書籍索引就是從鍵值到值（從主題到頁碼）映射的一個好例子。

不要為你的目的用錯了群集。我遇過不應該有重複的串列太多次了，或是實際上並沒有被用來映射鍵值到值的字典。每當你的意圖與程式碼中的內容脫節時，就會造成維護的負擔。維護者必須暫停工作，找出你真正的意思，然後才能為他們錯誤的假設（也包括你的錯誤假設）找出變通之道。

動態索引 vs. 靜態索引

取決於你所用的群集型別，你可能會想要使用**靜態索引**（*static index*）。靜態索引就是當你使用一個常數字面值（constant literal）來索引群集時，會得到的東西，例如 `my_list[4]` 或 `my_dict["Python"]`。一般來說，串列和字典通常不會這樣被使用。由於它們的動態特性，你不能保證這種群集在那個索引上有你要找的元素。如果你在這些型別的群集中尋找特定的欄位（fields），那就是很好的跡象，指出你需要一個使用者定義的型別（user-defined type，會在第 8、9、10 章中探討）。對元組進行靜態索引是安全的，因為它們的大小是固定的。集合和產生器則永遠不會被索引。

這一規則的例外情況包括：

- 取得一個序列（sequence）的第一個或最後一個元素（`my_list[0]` 或 `my_list[-1]`）
- 使用字典作為一種中介的資料型別（intermediate data type），如讀取 JSON 或 YAML 時
- 在一個序列上進行的運算，特別是處理固定的區塊時（例如，總是在第三個元素之後分割，或者在一個格式固定的字串中檢查某個特定的字元）
- 特定群集型別的效能因素

> 相較之下，每當你使用一個執行時期（runtime）才會知道其值的變數對一個群集進行索引，就會發生**動態索引動作**（*dynamic indexing*）。這是選用串列和字典最合適的時機。在對群集進行迭代或用 `index()` 函式搜尋一個特定的元素時，你就會看到這種情況。

這些是基本的群集，但要表達意圖，還有更多的方式可用。這裡有一些特殊的群集型別，它們在與未來的溝通中更具有表達力：

`frozenset`

一種不可變的集合（set）。

`OrderedDict`

會依據插入時間（insertion time）保留元素順序的一種字典（dictionary）。從 CPython 3.6 和 Python 3.7 開始，內建的字典也會依據插入時間保留元素的順序。

`defaultdict`

欠缺鍵值（key）時會提供一個預設值（default value）的一種字典。舉例來說，我可以把前面的例子改寫為：

```
from collections import defaultdict
def create_author_count_mapping(cookbooks: list[Cookbook]):
    counter = defaultdict(lambda: 0)
    for cookbook in cookbooks:
        counter[cookbook.author] += 1
    return counter
```

這為終端使用者引入了一種新的行為：如果他們在字典中查詢一個不存在的值，就會得到一個 0。這在某些用例中可能是有益的，如果沒有，你大可改為單純回傳 `dict(counter)`。

`Counter`

用來計數（counting）一個元素出現多少次的一種特殊的字典型別。這能大幅簡化上面的程式碼為這樣：

```
from collections import Counter
def create_author_count_mapping(cookbooks: list[Cookbook]):
    return Counter(book.author for book in cookbooks)
```

花點時間思考一下那最後一個例子。注意到，在不犧牲可讀性的前提之下，使用 Counter 如何讓我們的程式碼更加簡明。如果你的讀者熟悉 Counter，那麼這個函式的含義（以及該實作的運作方式）就會立即顯現出來。這是透過群集型別更妥善的選擇來向未來傳達意圖的好例子。我將在第 5 章進一步探討群集。

還有很多其他的型別可以探索，包括 array、bytes 與 range。每當你遇到一種新的群集型別，不管是內建的還是其他，問問自己它與其他群集有何不同，以及它向未來的讀者傳達了什麼。

迭代

迭代（iteration）是「你所選的抽象層（abstraction）決定了你傳達的意圖」的另一個例子。

你見過多少次這樣的程式碼？

```
text = "This is some generic text"
index = 0
while index < len(text):
    print(text[index])
    index += 1
```

這段簡單的程式碼將每個字元列印在單獨的一行。這對於第一次使用 Python 解決這種問題來說是完全沒問題的，但是這個解決方案很快就演變成了更 *Pythonic*（以一種慣用風格撰寫的程式碼，目的是強調簡單性，對大多數 Python 開發者來說都很容易識別）的形式：

```
for character in text:
    print(character)
```

花點時間反思一下，為什麼這種選項更合適。for 迴圈是一個更恰當的選擇，它更清楚地傳達了意圖。就像群集型別一樣，你挑選的迴圈構造（looping construct）明確地傳達了不同的概念。這裡列出了一些常見的迴圈構造和它們所傳達的意義：

for 迴圈

for 迴圈用來迭代過（iterating over）一個群集或範圍（range）的每個元素，並進行某個動作或產生某種副作用（side effect）。

```
for cookbook in cookbooks:
    print(cookbook)
```

while 迴圈

若要在某個特定條件為真的情況下都持續迭代，就使用 `while` 迴圈。

```
while is_cookbook_open(cookbook):
    narrate(cookbook)
```

概括式（*Comprehensions*）

概括式被用來把一個群集變換（transforming）為另一個（正常來說，這不會有副作用，特別是在概括式是惰性的時候）。

```
authors = [cookbook.author for cookbook in cookbooks]
```

遞迴（*Recursion*）

遞迴用在一個群集的子結構（substructure）與該群集本身的結構完全相同之時（舉例來說，一個樹狀結構的每個子節點也是樹狀結構）。

```
def list_ingredients(item):
    if isinstance(item, PreparedIngredient):
        list_ingredients(item)
    else:
        print(ingredient)
```

你希望你源碼庫的每一行都能提供價值。此外，你希望每一行都能向未來的開發者清楚傳達那個價值是什麼。這促使我們要儘量減少樣板程式碼（boilerplate）、鷹架（scaffolding）和多餘程式碼的數量。在上面的例子中，我迭代過每個元素，並產生一個副作用（印出一個元素），這使得 `for` 迴圈成為理想的迴圈構造。我沒有浪費程式碼。相較之下，`while` 迴圈要求我們明確地追蹤迴圈動作，直到某個條件發生為止。換句話說，我得追蹤一個特定的條件，並在每次迭代中變動一個變數。這分散了迴圈所提供的價值，並帶來了沒必要的認知負擔。

最不意外法則

分散對於意圖的注意力是不好的，但有一類溝通更糟糕：當程式碼不斷讓你未來的協作者感到意外。你會想要遵守**最不意外法則**（*Law of Least Surprise*）。有人讀過源碼庫時，他們幾乎永遠都不應該對行為或實作感到意外（而當他們感到意外時，在程式碼的附近應該要有很好的註解來解釋為什麼是如此）。這就是為什麼傳達意圖是最重要的。清晰、潔淨的程式碼可以降低溝通不良的可能性。

最不意外法則（*Law Of Least Surprise*），也被稱為最小驚奇法則（*Law of Least Astonishment*），指出程式應該總是以讓使用者最不驚訝[3]的方式來回應他們。意外的行為導致困惑，困惑會導致錯置的假設，而錯置的假設導致了臭蟲的產生，你就是這樣得到不可靠的軟體的。

請記住，你可以寫出完全正確的程式碼，但在未來還是讓人感到驚訝。在我職業生涯的早期，我追查過一個討厭的臭蟲，因為記憶體的毀損而導致當機。把程式碼放在除錯器底下執行，或是放入過多的列印述句，都會影響到時機，使錯誤無法顯現（一個真正的「海森堡（heisenbug）」）[4]。

因此，我不得不手動平分，把程式碼分成兩半，藉由移除另一半程式碼來查看到底是哪一半的程式碼導致了崩潰，然後在那一半的程式碼中重新再做一遍同樣的事。抓頭苦思兩個星期之後，我最後決定去檢查一個聽起來無害的函式，叫作 `getEvent`。結果發現，這個函式實際上會設定（*setting*）帶有無效的資料的一個事件。不用說，我感到非常意外。就它所做的事情而言，這個函式是完全正確的，但由於我搞錯了程式碼的意圖，我至少有三天的時間都漏看了那個臭蟲。讓你的協作者感到驚訝會導致他們時間的損失。

很多的這種意外，最終都來自於複雜性。有兩種類型的複雜性：**必要的複雜性**（*necessary complexity*）和**意外的複雜性**（*accidental complexity*）。必要的複雜性是你領域中固有的複雜性。深度學習模型（deep learning models）必然是複雜的，它們不是你瀏覽一下內部工作原理，在幾分鐘就能理解的東西。物件關聯映射（object–relational mapping，ORM）的最佳化必然是複雜的，有大量可能的使用者輸入要被考慮在內。你無法消除必要的複雜性，所以你最好的選擇是試圖控制它，以免它在你的源碼庫中蔓延，最後反而成為意外的複雜性。

相較之下，意外的複雜性是指會在程式碼中產生多餘的、浪費的或令人困惑的述句（statements）的那種複雜性。當一個系統隨著時間的推移不斷發展，而開發人員沒有重新評估舊程式碼以確定他們原來的假設是否依然成立，就把功能塞進去，那時就會發生這種狀況。我曾經參與過的一個專案，在其中，增加單一個命令列選項（以及設定它的相關程式化方法）所涉及的檔案，不會少於 10 個。為什麼增加一個簡單的值就得在整個源碼庫中修改那麼多地方呢？

3 Geoffrey James. *The Tao of Programming*. *https://oreil.ly/NcKNK*。

4 被觀察時會顯現不同行為的臭蟲。*SIGSOFT '83: Proceedings of the ACM SIGSOFT/SIGPLAN software engineering symposium on High-level debugging*。

如果你有過以下經歷，你就知道有意外的複雜性存在：

- 聽起來很簡單的事情（新增使用者、改變 UI 控制項等）卻不容易實作。
- 很難讓新的開發人員了解你的源碼庫。一個專案的新開發者是你程式碼可維護性的最佳指標，不需要等待多年就可以知道。
- 增加功能性的預估時間總是很長，但你依然趕不上既定時程表。

盡可能消除意外複雜性並分離出必要複雜性。這些會是你未來協作者的絆腳石。這些複雜性來源使溝通不良的情況更加惡化，因為它們使整個源碼庫中各處的意圖都變得模糊和分散。

討論主題

你的源碼庫中有哪些意外的複雜性？如果你被丟到源碼庫中，而且沒有與其他開發者交流，要理解其中簡單的概念會是多大的挑戰？你可以做些什麼來簡化在這個練習中所識別出來的複雜性呢（尤其是在經常變化的程式碼中）？

在本書的其餘部分，我將研究在 Python 中溝通意圖的不同訣竅。

結語

強健的程式碼很重要。潔淨的程式碼也很重要。你的程式碼需要在源碼庫的整個生命週期內都是可維護的，而為了確保這一結果，你需要對你所傳達的內容和方式有實質的遠見。你得在盡可能靠近程式碼的地方清楚地體現你的知識。不斷地展望未來感覺像是一種負擔，但只要多練習，那就會變得很自然，而你維護自己源碼庫的過程中，就會開始收穫好處。

每次你把一個現實世界的概念映射到程式碼中時，你就是在創造一個抽象層（abstraction），不管是透過群集的使用，還是你把函式分離的決定。每一個抽象層都是一種選擇，而每個選擇都傳達了一些東西，不管是有意還是無意。我鼓勵你思考你所寫的每一行程式碼，並問自己：「未來的開發者會從這裡學到什麼？」。為了未來的維護者，你有責任讓他們能夠以與你當下相同的速度提供價值。否則，你的源碼庫會變得臃腫，進度會推遲，複雜性也會增長。作為一名開發者，你的工作就是要緩解這種風險。

尋找潛在的熱點，如不正確的抽象層（例如群集或迭代）或意外的複雜性。這些都是溝通可能隨著時間的推移而中斷的主要區域。如果這些類型的熱點出現在經常變化的地方，那麼它們就是現在要優先解決的問題。

在下一章中，你會把從本章學到的東西應用於一個基本的 Python 概念：型別（types）。你所選的型別向未來的開發者表達了你的意圖，選擇正確的型別將促成更好的可維護性。