第二版 *The Pragmatic Programmer* 收到的讚賞

有人說，*The Pragmatic Programmer* 這本書，根本是 Andy 和 Dave 的神來一筆之作；近期不太可能有人能寫出一本能像它那樣推動整個行業的書。然而，神來兩筆是有可能的，而這本書就是明證。第二版更新的內容確保了它將在「軟體發展最佳書籍」的榜首再待上 20 年，這正是它該得到的位置。

➤ **VM (Vicky) Brasseur**
Director of Open Source Strategy, Juniper Networks

如果您希望您的軟體更現代化和更易於維護，請在手邊保留一本 *The Pragmatic Programmer*。本書充滿了實用的建議，既有技術上的，也有專業上的，這些建議將在未來的幾年裡提升您和您的專案。

➤ **Andrea Goulet**
CEO, Corgibytes; Founder, LegacyCode.Rocks

The Pragmatic Programmer 這本書，完全改變了我在軟體領域的職業生涯，為我指明了成功的方法。閱讀這本書讓我意識到成為一名工藝職人的可能性，而不僅僅是一架大機器上的一個齒輪。這是我生命中最重要的書之一。

➤ **Obie Fernandez**
Author, The Rails Way

對於第一次閱讀的讀者，您可以期待將看到一個吸引人的介紹，引導您進入軟體實作的現代世界，一個被第一版塑造出的世界。閱讀過第一版的讀者將在第一時間，重新發現使這本書如此重要的洞察力和實作智慧、專業策劃和更新，與時並進。

➤ **David A. Black**
Author, The Well-Grounded Rubyist

我的書架上有一本紙本舊版的 *The Pragmatic Programmer*。我把它讀了又讀，而且在很久以前它就改變了我作為一個程式設計師的工作方式。在新版中，一切都改變了，一切也都沒有改變：我現在改為在 iPad 上閱讀它，程式碼範例改為使用現代程式設計語言，但是底層的概念、思想和態度是永恆的、普遍適用的。二十年後，這本書依然與我們息息相關。我很高興地知道，現在和將來的開發人員都將有機會像以前一樣，從 Andy 和 Dave 的深刻見解中學習。

➤ **Sandy Mamoli**
Agile coach, author of How Self-Selection Lets People Excel

20 年前，第一版的 *The Pragmatic Programmer* 完全改變了我的職涯軌跡。這個新版本會對您產生一樣的效用。

➤ **Mike Cohn**
Author of Succeeding with Agile, Agile Estimating and Planning, and User Stories Applied

前言

我記得當 Dave 和 Andy 第一次在推特上談論這本書的新版時，這可是條大新聞。我看到了程式設計社群的興奮回應，我的 feed 也因著這些期待而嗡嗡作響。二十年後的今天，《*The Pragmatic Programmer*》這本書的地位和在過去一樣地重要。

一本這樣具歷史背景的書會得到如此迴響，代表了很多事。我有幸在本書未出版前先閱讀它，並寫這篇序言，我明白它為什麼會引起這麼大的轟動。雖然這是一本技術書籍，但給它這樣的名稱完全是幫倒忙。技術書籍常常令人生畏，它們充斥著誇大、晦澀的術語和令人費解的例子，無意中讓您覺得自己很愚蠢。作者越有經驗，就越難把自己放在一個初學者的角度，越容易忘記學習新概念的感覺。

儘管 Dave 和 Andy 有幾十年的程式設計經驗，他們卻已經克服了寫作的困難挑戰，他們和剛接觸這些知識的人一樣興奮。他們不會居高臨下地對您說話，他們不會假定您是專家，他們甚至不假設您讀過第一版，他們把您當成是想要變得更好的程式設計師。他們用這本書一步一腳印地來幫助您達到目標。

憑良心講，他們以前就已經這樣做了。本書第一版中同樣包含了許多具體的例子、創新想法和實用的技巧，可以幫助您建立寫程式碼的肌肉記憶和開發您的寫程式碼的大腦，這些技巧今天仍然適用，但是本書的新版本做了兩個改進。

第一個改進是顯而易見的：它刪除了一些較老的參考和過時的例子，並用新鮮、現代的內容替換它們，您不會再看到關於迴圈不變數或建構機器的例子。Dave 和 Andy 已經擷取了它們的強大的內涵，並確保不會錯過任何重

點，不會被舊的例子分心。它重新詮釋原有的概念，如 DRY（don't repeat yourself），並給它們塗上一層新的油漆，真正讓它們閃閃發光。

第二個改進是這次改版真正令人興奮的地方。寫完第一版後，他們有機會反省他們想要說什麼，他們想讓讀者帶走什麼，以及讀者是如何接收這些訊息的。他們利用之前的回饋，看到了卡住的地方、需要改進的地方、需要解釋的地方。在這本書透過全世界程式設計師的雙手和心靈傳播的 20 年裡，Dave 和 Andy 研究了這些回饋並形成了新的想法和概念。

他們已經認識到行動力的重要性，並認識到開發人員可能比大多數其他專業人員更需要行動力。所以他們以簡單而深刻的訊息開始這本書：「這是您的人生」，它提醒了我們，在我們撰寫的程式碼中、在我們的工作中、在我們的職業生涯中我們擁有力量，這句話為本書奠定了基調，它不只是另一本塞滿程式碼範例的技術書籍。

讓它在技術書籍的書架上真正脫穎而出原因，是它知道作為一個程式設計師代表著什麼。程式設計的目的是為了減少未來的痛苦，是為了讓我們的隊友更輕鬆，是關於把事情做錯並能夠重新振作起來，是要養成良好的習慣，是要去瞭解您的工具集。撰寫程式碼只是一個程式設計師工作的一部分，這本書探索了這樣的一個世界。

我花了很多時間思考撰寫程式碼的旅程，由於我不是生來就會寫程式；我在大學裡沒有學過，我的青少年時期並沒有花時間在擺弄技術上。我在 25 歲左右進入了程式碼的世界，才開始學習成為一名程式設計師的意義。這個社群和我曾經接觸過的其他社群非常不同，為學習特別的奉獻又著重實用性，既令人耳目一新，又令人生畏。

對我來說，這真的像是進入了一個新的世界，或至少是個新城市。我得去認識鄰居，挑選買東西的雜貨店，找最好的咖啡店，需要花了一段時間來瞭解這塊土地，以找到最有效的路線，避開交通最繁忙的街道，知道什麼時候交通可能會受到影響。這裡的氣候不一樣，而我整個衣櫃的衣服都需要重新買過。

在一個新城市的頭幾週，甚至是頭幾個月可能會令人感到害怕。若有一個友好的、知識淵博的鄰居不是很好嗎？鄰居可以帶您參觀，帶您去那些咖啡店？鄰居是一個在那裡待了足夠長的時間，瞭解當地文化、瞭解當地脈搏的人，這樣您不僅得到家的感覺，在未來也能成為一個有貢獻的成員？ Dave 和 Andy 就是您的鄰居。

對一個身為相對新手的人來說，很容易被成為程式設計師的過程而不是程式設計的行為所淹沒。一個完整的心態轉變需要發生，在習慣、行為和期望上的改變。成為一個更好的程式設計師的過程並不止是您知道如何撰寫程式碼；它必須透過刻意和深思熟慮的實作來實行。這本書是一本指南，有效率地引導您成為一個更好的程式設計師。

但是請不要誤會，本書不會告訴您應該如何做程式設計，這不是要做哲學，也不是充滿批判。本書簡單明瞭地告訴您，什麼是務實的程式設計師，他們會做什麼，如何處理程式碼。作者們讓您自己決定您是否想成為其中一員。如果您覺得那不適合您，他們也不會反對您。但如果您認為自己想成為其中一員，那麼他們就是您的友善鄰居，他們會為您指路。

➤ **Saron Yitbarek**
Founder & CEO of CodeNewbie
Host of Command Line Heroes

第二版前言

回到 1990 年代，當時我們與一些專案出了問題的公司合作。我們發現自己對每個人都說了同樣的話：也許您應該在發佈之前測試一下；為什麼程式碼只能在 Mary 的機器上建構？為什麼沒有人問過使用者呢？

為了節省與新客戶打交道的時間，我們開始做筆記。這些筆記最後變成了 *The Pragmatic Programmer* 這本書。令我們驚訝的是，這本書似乎引起了共鳴，在過去的 20 年裡，這本書一直很受歡迎。

但是 20 年對於軟體來說是很長的壽命。若將一個從 1999 年的開發人員丟到今天的一個團隊裡，他們將在這個陌生的新世界中掙扎。而 1990 年代的世界對今天的開發者來說同樣陌生。書中對 CORBA、CASE 工具和索引迴圈（indexed loop）的引用，輕則令人覺得離奇古怪，有更大機會讓人混淆。

與此同時，20 年對常識沒有任何影響。技術可能改變了，但人沒有。以前曾經是好的實務和方法論，現在仍然是好的，這本書把那些方面保存得很好。

所以，當我們要撰寫這個 *20th* 周年紀念版時，我們必須做出一個決定。我們可以花一天時間將全書所引用到的技術更新成現今的技術。或者，我們可以根據另外 20 年的經驗，重新檢查我們建議實作背後的假設。

最後，我們兩者都做到了。

因此，這本書有點像忒修斯的船（*Ship of Theseus*）[註 1]。書中大約三分之一的主題是全新的。其餘的大部分，也都被部分或全部重寫了。我們的目的是讓事情變得更清晰、更相關、更永恆。

我們做了一些艱難的決定，我們刪除了資源（*Resources*）附錄，因為它不可能保持最新，也是因為現在您更容易藉由搜尋找到您想要的內容。考慮到現今平行硬體豐富而且缺乏處理平行的好方法，我們重新組織和重寫了與平行相關的主題。我們添加了一些內容來說明持續變化的態度和環境，這些內容從我們幫助發起的敏捷運動，到對函式式程式設計習慣的逐漸普及，以及對隱私和安全性的日益需求。

然而有趣的是，關於這個版本內容，我們之間的爭論也比寫第一個版本時要少得多，我們都覺得更容易識別出什麼才是重要的東西。

無論如何，這本書就是成果，請您享受它，也許您也可以採納一些新的做法，或判定我們建議的一些東西是錯的。請讓這本的內容參與您的工作，並給我們回饋。

但是，最重要的是，記住要讓過程保持有趣。

本書組織

這本書是由許多短篇集合而成，每個主題都是獨立完整、並且針對特定的主題，您會發現大量的交叉引用，這有助於您理解每個主題前後關係。您可以以任何順序隨意閱讀主題，這不是一本需要您從頭到尾閱讀的書。

在書本，您偶爾會看到一個標示為提示 *nn*（例如提示 1，重視您的手藝，在第 xviii 頁）的方塊，這些提示除了強調文字中的要點外，我們覺得這些提示

註 1 如果隨著幾年過去，一艘船的每個零件壞掉的時候，都被換新，那最後這艘船還是原來那艘船嗎？

有自己的生命，我們每天都和它們生活在一起。您會在 Appendix C 的提示卡中找到所有提示的匯總摘要。

我們已經在適當的地方放了練習題和挑戰題。練習題通常有相對簡單的答案，而挑戰題的答案則比較開放。為了讓您對我們的想法有個概念，我們已經把練習題的答案列在附錄裡了，但是它們通常不是唯一正確的解答。這些挑戰題可能成為高級程式設計課程中，小組討論或論文寫作的基礎。

還有一個簡短的參考書目，列出了我們明確引用的書籍和文章。

書名的含意？

> 「當我用一個詞時」，*Humpty Dumpty* 輕蔑地說到「它的意思就是我想要它該有的意思——不能多也不能少」。
>
> ➤ **Lewis Carroll, Through the Looking-Glass**

在整本書中，您會發現各式各樣的術語——要麼由英語單詞曲解來的技術術語，要麼是由對某種語言充滿怨恨的電腦科學家編造出來令人髮指的單詞。當我們第一次提到這些術語時，我們會嘗試去定義它，或者至少給它一些說明。然而，我們確信有些術語還是被遺漏了，而其他的，例如物件（*object*）和關聯式資料庫（*relational database*），由於這些術語被非常普遍的使用，所以硬是要為它們添加一個定義說明將會很無聊。如果您真的遇到一個您以前沒見過的術語，請不要跳過它，請花點時間去查一下，也許在網路上查，也許在電腦科學課本上查。或如果可以的話，請發郵件向我們投訴一下，這樣我們就可以在下一版增加一個定義說明。

說了這麼多，我們決定報復電腦科學家。有時候，的確有一些非常好的術語可表示一些概念，但我們決定忽略這些詞。為什麼？因為現有的術語通常被局限於特定的問題領域，或者特定的開發階段。然而，本書的基本思想之一是，我們推薦的大多數技術都是通用的：例如，模組化同時通用於程式碼、設計、文

件和團隊組織。若我們在更廣泛的背景下使用傳統術語時，它們將變得令人困惑，我們似乎無法克服術語最初的定義所帶來的包袱。當這種情況發生時，我們選擇發明了自己的術語，而不使用原來術語。

原始程式碼和其他資源

本書中的大部分程式碼都是從可編譯的原始檔案中提取出來的，可編譯的原始檔案可以從我們的網站上下載[註 2]。

在我們的網站上，您還可以找到我們認為有用的資源連結，以及本書的更新和其他關於本書的消息更新。

請發送回饋訊息給我們

我們很高興收到您的來信，請發郵件到 `ppbook@pragprog.com`。

第二版致謝

在過去的 20 年裡，我們已經享受了成千上萬的關於程式設計的有趣對話，在研討會認識的人時，在教授課程時，有時甚至在坐飛機時。每一個有趣對話都增加了我們對開發過程的理解，這些對話也為本版本的更新做出了貢獻。謝謝您們所有人（還有也請在我們錯了的時候，持續提醒我們）。

感謝參與本書預發行版的所有人員，您的問題和評論幫助我們把事情解釋得更好。

在我們進行預發行版之前，我們與一些人分享了這本書，並請這些人給予評論指導。感謝 VM（Vicky） Brasseur、Jeff Langr 和 Kim Shrier 的詳細評論，感謝 José Valim 和 Nick Cuthbert 的技術評論。

註 2 *https://pragprog.com/titles/tpp20*

感謝 Ron Jeffries 讓我們用他數獨（Sudoku）的範例。

非常感謝 Pearson 的工作人員，是他們讓我們以自己的方式來創作這本書。

特別感謝不可或缺的 Janet Furlow，她掌控著一切，讓我們在正確的路上前進。

最後，向所有在過去的 20 年裡一直致力於讓程式設計變得更好的務實程式設計師們發出一聲吶喊，後面是另一個 20 年。

第一版序言

這本書將幫助您成為一個更好的程式設計師。

不論您是單獨的開發人員、一個大型專案團隊的成員，或者一個同時與許多客戶一起工作的顧問，這都沒有關係；這本書將都能幫助您，作為一個個體，做更好的工作。這本書不是理論性的書籍，我們關注的是務實的主題，利用您的經驗來做出更明智的決定。務實這個詞來自於拉丁語 *pragmaticus*「專精事務」，而後者來自於希臘語的 πραγματικός，它的意義為「適合使用」。

這是一本關於做（doing）的書。

程式設計是一門手藝。簡單地說，就是讓電腦做您想讓它做的事情（或您的使用者想讓它做的事情）。作為一名程式設計師，您既是聽眾，又是顧問，既是解譯者，又是獨裁者。您試圖捕獲難以捉摸的需求，並找到一種表達它們的方式，以便只用機器就可以很好地處理它們。您試著記錄您的工作，這樣別人就能理解它，您試著策劃您的工作，這樣別人就利用您的工作產生更多建樹。更重要的是，您試圖在專案時程的滴答聲中完成所有這些工作，您每天都在創造奇跡。

這是一項困難的工作。

有許多人能提供您幫助，例如吹捧他們的產品能創造奇跡的工具供應商，方法論大師拍胸保證他們的技巧能保證成果。每個人都聲稱他們的程式設計語言是最好的，每個作業系統能解決所有的問題。

當然，這些都不是真的。從來就沒有簡單的答案。沒有最佳的解決方案，無論是工具、語言還是作業系統。只有在特定的環境下才有更合適的系統。

這就是務實主義派上用場的地方。您不應該拘泥於任何特定的技術，而應該擁有足夠廣泛的背景和經驗基礎，以便在特定的情況下選擇合適的解決方案。您的背景源於對電腦科學基本原理的理解，而您的經驗來自於廣泛的專案實作。理論和實作相結合使您強大。

您應調整方法以適應當前的情況和環境，您可以判斷影響專案的所有因素的相對重要性，並使用您的經驗來產生適當的解決方案。隨著工作的進展，要不斷地這樣做。務實的程式設計師最終能完成工作，並且做得很好。

誰應該讀這本書

這本書的目標讀者是那些希望成為更高效、更有生產力的程式設計師的人。也許您感到沮喪，因為您沒有好好利用您的潛力。也許您會注意到，有些同事似乎利用工具使自己比您更有效率。也許您現在的工作使用的是較老的技術，您想知道如何應用新想法到您的工作中。

我們不會假裝擁有所有（甚至大部分）答案，也不會假裝我們所有的想法都適用於所有情況。我們只能說，如果遵循我們的方法，您將快速獲得經驗，生產力將提高，並能更理解整個開發過程，您會寫出更好的軟體。

什麼造就了一個務實的程式設計師

每個開發人員都是獨特的，具有各自的優勢和劣勢、偏好和討厭的東西。隨著時間的推移，每個人都將打造自己的個人環境。這種環境將像程式設計師的愛好、衣服或髮型一樣強烈地反映出他 / 她的個性。然而，如果您是一個務實的程式設計師，您會有以下許多共同點：

早期採用者 / 快速適應

> 您對技術和技巧有一種直覺，您喜歡嘗試。當接觸到新的東西時，您可以很快地掌握它，並把它與您其他的知識結合起來。您的信心來自於經驗。

好奇

您很愛問問題，您是怎麼做到的？那個函式庫用起來有問題？我聽說過的量子計算是什麼？如何實作符號連結？您會對小的事情起疑，而這些小事情都可能會影響幾年後的決定。

批判性的思想家

您很少在不了解事實的情況下就接受別人給您的東西。當同事們說「因為這就是要這樣做」，或者供應商承諾會解決您所有的問題時，您會聞到挑戰的味道。

現實主義

您試圖理解您面臨的每個問題的本質。這種現實主義讓您對事情有多困難、需要多長時間有一個很好的感覺。深刻理解一個過程應該有難度，或會花上一段時間，可以給您堅持下去的毅力。

萬事通

您努力熟悉各種技術和環境，並努力跟上新開發的步伐。雖然您目前的工作可能要求您成為該領域的專家，但您總是能夠進入新的領域，迎接新的挑戰。

我們把越基本的特徵留到越後面，所有實用的程式設計師都有這些特徵。這些特徵可以用一個提示表達：

提示 1	重視您的手藝

除非您想把開發軟體做好，否則我們覺得開發軟體是沒有意義的。

提示 2	思考！您的工作

為了成為一個務實的程式設計師，我們要求您在做事的時候思考一下您在做什麼。這不是只對當前在做的事情進行一次檢查，而是對您每天作的決策進行批判性評估，每天做，對每一個專案都做。不要用直覺，而是要不斷地思考，即時批判您的工作。IBM 公司的老格言 *THINK!*，是務實的程式設計師的口頭禪。

如果這對您來說聽起來很困難，那麼您正在展示的就是現實主義特徵。這樣做會佔用您一些寶貴的時間，這些時間可能已經處於巨大的壓力之下。而這麼做的回報是可以更積極地投入到您喜歡的工作中，對越來越多的主題有掌控感，對不斷進步的感覺有愉悅感。從長期來看，您的時間投資將得到回報，因為您和您的團隊將得到更高的效率，撰寫更容易維護的程式碼，並在會議上花費更少的時間。

個體實用主義者與大型團隊

有些人認為在大型團隊或複雜的專案中不該有個人的空間。「軟體是一種工程紀律」，他們說，「如果單個團隊成員只站在自己的角度做決定，它就會崩潰」。

我們強烈反對這句話。

的確應該將工程概念應用在軟體建構上。然而，這並不妨礙個人的技藝。想想中世紀在歐洲建造的大教堂，每一個都需要花上數千人年，時間跨度幾十年。吸取的經驗教訓被傳遞給下一代的建設者，他們用自己的成就推動了結構工程的發展。但木匠、石匠、雕刻師和玻璃工人都是獨立的手工匠人，他們解譯各種工程需求，以生產出一個整體，一個超越機械製造的整體。正是他們對個人貢獻的信念支撐著這些專案：雖然只是採石者，但仍心心念念展望未來的大教堂。

在一個專案的整體結構中，總是會有給個人特色與工匠的空間。考慮到現代軟體工程的情況，這一點尤其正確。一百年後，我們的工程可能會像中世紀大教堂建造者使用的技術，在今日的土木工程師眼中看來雖然古老，但我們的技藝仍將受到尊重。

這是一個持續的過程

> 一位到英國伊頓公學旅遊的遊客問園丁他是如何把草坪修剪得如此完美。「那很容易，」他回答說，「您只要每天早上拂去露水，隔天修剪一次，一週滾壓一次就行了」。
>
> 「這樣就可以了嗎？」遊客問。「沒錯」園丁回答，「如果您這樣做了*500*年，您會有一個很好的草坪。」

好的草坪需要少量的日常照顧，好的程式設計師也是如此。管理顧問們喜歡在談話中使用*改善*（*kaizen*）這個詞，「改善」是一個日語單詞，意思是不斷地做出許多小的改進，這被認為是日本製造業生產率和品質大幅提高的主要原因之一，並被全世界廣泛仿效。改善也適用於個人。每天都要努力完善您的技能，並在您的技能庫裡添加新的工具。不用像伊頓公學草坪那麼久，您會在幾天內就可以看到成果。執行數年，您會驚訝於您的經驗是如何開花結果的，您的技能是如何成長的。

Chapter 1

務實的哲學

這本書是關於您的。

毫無疑問，這本書寫的是您的事業，或甚至是您的生活。您現在之所以會閱讀這本書，是因為您知道自己可以成為一個更好的開發人員，並能幫助其他人也變得更好。您可以成為一個務實的程式設計師。

務實的程式設計師特別在哪裡呢？我們覺得是一種態度、一種風格、一種處理問題和解決方案的哲學。他們超越眼前的問題，把問題放在更大格局下思考，尋求更大的願景。畢竟，沒有這種更大的格局，您怎麼可能是務實的？如何做出聰明的妥協和明智的決定？

務實的程式設計師會成功的另一個關鍵原因，是他們對自己所做的一切負責，我們將在貓吃了我的原始碼中討論了這一點。務實的程式設計師是有責任心的，他們不會坐視專案因疏忽而崩潰。在軟體亂度中，我們將告訴您如何保持專案的原始狀態。

大多數人覺得改變是困難的，這種困難有時有一些好的理由，有時則只是純粹的習慣。在石頭湯與煮青蛙小節中，我們著眼於一種激發漸進變化的策略，並且（為了平衡）會再以一個兩棲動物的寓言故事說明，漸進變化的危險很容易被忽略。

瞭解與您工作相關的狀況的好處之一，您可以更容易地瞭解您的軟體必須有多好。有時，近乎完美是唯一的選擇，但常常需要權衡利弊。我們在夠好的軟體中對此進行了探討。

當然，您需要有廣泛的知識和經驗基礎來完成這一切，而學習是一個持續不斷的過程。在您的知識資產，我們會討論一些保持成長的策略。

最後，沒有人是在真空中工作的。我們都花大量的時間與他人互動。在溝通！中，將會列出我們可以做得更好的方法。

務實程式設計源於一種務實的思考哲學，本章將會說明這種哲學的基本概念。

1 這是您的人生

> 我活在這個世界上不是為了滿足您的期望，就像您活著也不是為了滿足我的期望。
>
> ➤ 李小龍

這就是您的人生。您擁有它，您執行它，您創作它。

許多與我們聊過的開發人員都呈現一種沮喪的狀態，他們有各式各樣的擔憂。一些人覺得他們的工作停滯不前，另一些人則認為自己追不上技術。員工們覺得自己沒有得到賞識，或者薪水太低，或者他們團隊的嚴重問題。也許他們想搬到亞洲，或者歐洲，或者在家裡工作。

而我們總是給出一樣的答案。

「為什麼您不能改變它？」

在您所有可以考慮從事的職業清單中，軟體開發一定是排在最前面的選擇。因為，我們的技能是有市場需求的，我們的知識不受地理限制，我們可以遠端工作，我們薪資不錯，我們幾乎真的可以做任何我們想做的事情。

但是，出於某種原因，開發人員似乎抗拒改變。他們蹲守原地，一心希望情況會好轉。當他們的技能過時時，他們只被動地旁觀，並抱怨公司沒有培訓他們。他們會在公車上看異國風情的廣告，然後走進寒冷的雨中，艱難地去上班。

所以，以下是本書中最重要的提示。

提示 3 您擁有改變的能量

您的工作環境糟糕嗎？您的工作無聊嗎？請試著改變它，但也不要一直試到天荒地老。正如 Martin Fowler 所說的：「您可以改變現有員工做事的方法，也可以改為僱用其他的員工」（you can change your organization or change your organization）[註 1]。

如果您擁有的技術看起來已落後，請（從自己的時間裡）擠出時間來學習看起來有趣的新東西。您是在為自己投資，所以趁下班時間做這件事是合理的。

想要遠端工作嗎？您有問過可行性嗎？如果得到的答案是「不行」，那就找一個說「可以」的人。

這個行業給您提供了一系列非凡的機會，請積極主動地把握這些機會。

相關章節包括

- 主題 4，*石頭湯與煮青蛙*，第 10 頁
- 主題 6，*您的知識資產*，第 16 頁

註 1 *http://wiki.c2.com/?ChangeYourOrganization*

2 貓吃了我的原始碼

在所有弱點中，最大的弱點是害怕顯得軟弱。

➤ *J.B. Bossuet，Politics from Holy Writ，1709* 年

務實思考哲學的基石之一，是為您自己和您的行動負責，包括您的職業發展、您的學習和教育、您的專案、和您的日常工作。務實的程式設計師掌控自己的職業生涯，不害怕承認無知或錯誤。當然，這不是程式設計中令人愉快的一面，但它確實會發生，即使在最好的專案中也是如此。儘管有全面的測試、良好的文件和可靠的自動化，事情還是會出錯、交付還是會延遲、還是會出現不可預見的技術問題。

這些事情總會發生，我們會盡我們所能專業地處理它們，這代表著誠實和直接的態度。我們可以為自己的能力感到自豪，但也必須承認自己的不足，我們還是會有無知和做錯的時候。

團隊信任

最重要的事情是，您的團隊必須能夠信任和依賴您，您也需要放心地依賴他們每一個人。根據研究文獻[註 2]，對團隊的信任對於創造力和協作是絕對必要的。在一個以信任為基礎的健康環境中，您可以安全地說出您的想法，展示您的想法，並依靠您的團隊成員，他們也可以依靠您。如果缺少了信任，嗯…。

想像一下，一支擁有高科技的隱形忍者隊伍潛入了反派的邪惡巢穴。經過幾個月的計畫和精準的執行，您終於成功到達了那個邪惡巢穴。現在輪到你設計雷射引導網格時，您卻說：「對不起，夥計們，我沒有雷射。因為貓在玩那個紅點，所以我把它留在家裡了。」

註 2　舉例來說其中一篇參考文獻是 *Trust and team performance: A meta-analysis of main effects, moderators, and covariates*（*http://dx.doi.org/10.1037/apl0000110*），此參考文獻中有很好的描述分析。

這種對信任造成的破壞可能很難修復。

負責

責任感代表您積極認同的東西。因為您作的承諾是要確保某件事要被正確完成，但您卻不一定能直接控制該件事的所有細節。除了盡自己最大的努力之外，您還必須分析形勢，找出自己無法控制的風險。您有權利不為做不到、風險太大或道德含意太模糊的情況承擔責任。您必須根據自己的價值觀和判斷做出決定。

當您接受承擔結果的責任時，別人就會認為您要為結果負責。所以，當您犯了一個錯誤或判斷錯誤時（就像我們所有人一樣），誠實地承認它，並嘗試提供解決問題的選擇。

不要怪東怪西，也不要編造藉口。不要把所有的問題都歸咎於供應商、程式設計語言、管理層或您的同事。雖然這些人事物都參與其中，但主要成敗還是取決於您所提供的解決方案，責無旁貸。

如果存在供應商不幫您解決的風險，那麼您應該有一個應急計畫。如果您的儲存裝置燒掉了，而且裡面存了所有的原始程式碼，而您沒有備份，那就是您的錯。告訴老闆「貓吃了我的原始碼」是行不通的。

提示 4	請提供解決問題的選擇，停止製造爛藉口

在您打算去見任何人，並告訴他們為什麼有些事情做不到，為什麼有些事情會延後，為什麼有些事情會失敗之前，請先暫停一下，先把自己的想法告訴您螢幕上的橡皮小鴨或貓看看。您的藉口聽起來是合理的還是愚蠢的？您的老闆會如何看待？

請在腦海中演練對話，猜測別人可能會怎麼回應？他們會不會問您：「您試過這個嗎？」或者「您沒有考慮過嗎？」您將如何回應？在您去告訴他們壞消息

之前還能試試別的嗎？有時候，您根本就知道他們會說什麼，所以就不用去麻煩他們了。

不要找藉口，請提供解決問題的選擇，而且不要說做不到；說明一下做什麼才能挽救這種情況。必須捨棄目前的程式碼嗎？答案如果為是，就這樣告訴他們，並解釋重構的價值（參見主題 40，重構，第 247 頁）。

問問自己，您是否需要花費時間來做原型設計，以確定最佳的開發方式（參見主題 13，原型和便利貼，第 64 頁）？您是否需要引入更好的測試（參見主題 41，測試對程式碼的意義，第 252 頁，和無情且持續的迴歸測試，第 326 頁）或自動化來防止問題再次發生？

也許您需要額外的資源來完成這項任務，又或者您需要的是花更多的時間和使用者相處？或者您需要的就是您自己：例如，您需要去學習一些技術或更深入地技術嗎？閱讀書籍或進修課程會有幫助嗎？不要害怕去要求或承認您需要幫助。

在大聲說出那些站不住腳的藉口之前，試著先把這些藉口找出來銷毀。如果您無法忍住，那就把那些藉口告訴您的貓吧。畢竟，如果那個小屁孩要幫你背黑鍋的話…。

相關章節包括

- 主題 49，務實的團隊，第 312 頁

挑戰題

- 當有人，如銀行出納員、汽車修理工或職員，拿蹩腳的藉口搪塞您時，您會作何反應？您如何看待他們和他們的公司？
- 當您發現自己在說「我不知道」的時候，一定要接著說「但是我會知道的」。「這是承認您不知道的東西的好方法，而且同時也像專業人士那樣承擔責任。」

3 軟體亂度

雖然軟體發展幾乎不受所有物理定律的影響，但不可阻擋的亂度（熵 / *entropy*）的增長給我們帶來了沉重的打擊。亂度是物理學中的一個術語，指的是系統中「無序」的數量。不幸的是，熱力學定律保證宇宙中的亂度趨向於最大值。當軟體的無序性增加時，我們稱之為「軟體凋零」（software rot）。有些人可能會用更樂觀的術語「技術負債」（technical debt）來稱呼它，這個術語隱含著他們總有一天會償債的概念，但他們通常不會。

不過，不管叫什麼名字，凋零和負債都可能失控地蔓延。

導致軟體凋零的因素有很多，其中最重要的似乎是做專案時的心理或文化。即使您是一人團隊，您的專案心理可能是一件非常微妙的事情。儘管有最好的計畫和最好的人員，一個專案在它的生命週期中仍然會經歷毀滅和腐朽。然而，儘管存在巨大的困難和不斷的挫折，還是有其他一些專案能成功地克服大自然對無序的偏愛，並取得了不錯的成果。

是什麼造成了這種差異？

在市中心，有一些建築美麗而乾淨，也有另一些殘破不堪，為什麼呢？犯罪和城市衰敗領域的研究人員發現了一種「令人著迷」的觸發機制，它能迅速地將一棟乾淨、完整、有人居住的建築變成一座破碎、廢棄的廢墟[註 3]。

這就是破窗效應。

若有一扇被打破的窗戶，而且在相當長的一段時間內沒有得到修理，會給建築物的居民灌輸了一種被遺棄的感覺，一種當權者不關心建築物的感覺，所以導致另一扇窗戶也被打破了，人們開始亂扔垃圾，出現塗鴉，開始嚴重的破壞結構。在相對較短的時間內，建築被破壞的程度超出了業主修復的意願，被遺棄從一種感覺變成了現實。

註 3　參見 *The police and neighborhood safety [WH82]*。

為什麼一扇破窗會造成這樣的事情呢？心理學家的研究表明[註 4] 絕望是會傳染的，就像附近的流感病毒一樣。忽視一個明顯被破壞的情況會強化以下的想法：就算東西都壞了也不會有人在意，一切都是註定的；所有消極的想法會在團隊成員之間傳播，形成惡性循環。

提示 5	不要讓破窗存在

不要放任「破窗」（糟糕的設計、錯誤的決策或糟糕的程式碼）壞在那裡不修。一旦發現，立即修復每一個。如果沒有足夠的時間來修復它，那麼請將它用木板封住，比方說您可以註解掉有問題的程式碼，或者顯示一個「功能未實作」的訊息，或者用虛擬資料代替。請採取一些行動來防止進一步的傷害，這會顯示您已經控制了局面。

我們已經看到，一旦有窗戶開始崩壞，乾淨、功能良好的系統就會迅速劣化。還有其他因素會導致軟體凋零，我們將在本書其他地方討論其中的一些因素，但是忽視這個因素比其他任何因素更加速凋零的速度。

您可能會想，沒有人有時間去清理一個專案中所有的碎玻璃。如果是這樣，您最好計畫買一個垃圾箱，或者搬到另一個社區。不管如何，請不要讓亂度贏了這一局。

首先，不要傷害

Andy 曾經認識一個非常有錢的人。他的房子一塵不染，堆滿了價值連城的古董、藝術品等等。一天，掛在離壁爐太近的掛毯著火了。消防隊衝了進來，拯救了他和他的房子。但當時在他們把又大又髒的水管拖進屋裡之前，在火災仍在肆虐時停下來，於前門和火源之間鋪了一張墊子。

註 4 參見 *Contagious depression: Existence, specificity to depressed symptoms, and the role of reassurance seeking [Joi94]*。

他們不想把地毯弄髒。

這事情聽起來很極端。毫無疑問，消防部門的首要任務是撲滅大火，以免造成不必要的損失。但他們顯然已經評估了形勢，對自己控制火勢的能力充滿信心，並且小心翼翼地不讓財產造成不必要的損失。軟體也必須要這樣：不要因為出現了某種危機就造成附帶損害，一扇破窗的損害就夠多了。

一個破碎的破窗、一段設計糟糕的程式碼、一個糟糕的管理決策（在專案進行期間，團隊必須忍受的東西）就是開始衰敗的全部原因。如果您發現自己在做一個專案的時候，很多窗戶都是破的，您很容易就會陷入「其他部分的程式碼也都是垃圾，那我也跟著做垃圾」的思維。輕視了到目前為止，專案是否進展良好的事實。在「破窗理論」的最初實驗中，一輛被遺棄的汽車被完好無損地放置了一個星期。但一旦有一扇窗戶被打破，這輛車就會在數小時內從裡到外被奪取得體無完膚。

同樣地，如果您發現自己在一個專案中，程式碼寫得非常漂亮，乾淨、設計良好、優雅，您可能會格外小心，不把它弄糟，就像那些消防員一樣。即使有火災肆虐（比喻截止日期、發佈日期、商貿展等等），您也不會想成為第一個把東西弄得一團糟，造成額外的損害的人。

告訴您自己「不可以有破碎的窗戶」。

相關章節包括

- 主題 10，正交性，第 45 頁
- 主題 40，重構，第 247 頁
- 主題 44，命名，第 281 頁

挑戰題

- 透過調查您的專案的裡裡外外，來加強幫助您的團隊。請選擇兩到三個被打破的窗戶，與您的同事討論問題是什麼，以及可以做些什麼來修復它們。
- 您能知道第一次打破一扇窗戶是什麼時候嗎？您對此有何反應？如果這是別人決定的結果，或者是一個管理命令，您能做什麼呢？

4 石頭湯與煮青蛙

三個士兵餓著肚子從戰場回家。當他們看到前面的村莊時，他們的精神為之一振，他們相信村民們會給他們一頓飯吃。但是當他們到了那裡，他們發現門都鎖上了，窗戶也關著。經過多年的戰爭，村民們缺乏食物，當有食物時也會把所有的食都祕密貯藏起來。

士兵們沒有放棄，他們煮了一鍋水，小心地放了三塊石頭進去。驚訝的村民們都出來觀看。

士兵們解釋說：「這是石頭湯。」「這就是您放進去的所有東西嗎？」村民們問。「沒錯，雖然有人說加一點胡蘿蔔會更好喝…」，此時一個村民跑開了，很快就從他的貯藏裡拿出一籃子胡蘿蔔回來了。

幾分鐘後，村民們又問：「有變好喝嗎？」

「這個嘛，」士兵們說，「加幾個馬鈴薯能讓它更濃稠。」，此時又一個村民跑開了。

在接下來的一個小時裡，士兵們列出更多可以讓湯更美味的配料：牛肉、韭菜、鹽和香草。而每次都有不同的村民跑去他們的私人倉庫拿東西。

最終，他們做好一大鍋熱氣騰騰的湯。士兵們把石頭拿走，他們和全村的人坐在一起，享用他們幾個月來吃過的第一頓豐盛的飯。

石頭湯的故事裡有幾個寓意。首先，村民們被士兵們耍了，士兵們利用村民們的好奇心從他們那裡獲取食物。但更重要的是，士兵們發揮了催化劑的作用，讓整個村莊團結起來，這樣他們就可以聯合生產一些他們自己無法完成的東西，這就是協作的結果。最終每個人都贏了。

有時，您可能想要模仿士兵。

比方您可能處於這樣一種情況：您確切地知道需要做什麼以及如何去做，仿佛整個系統就出現在您眼前，您確信方向是正確的。但是，如果您請求處理整個事情，您會看到一些呆滯和茫然的眼神。人們將傾向成立委員會，預算將需要被批准，事情將變得複雜。每個人都會保護自己的資源。有時這被稱為「啟動疲勞」（start-up fatigue）。

此時該是時候把石頭拿出來的時候了，同時弄清楚您能合理要求的是什麼，然後好好地發展它。一旦您完成了它以後，請展示給人們看，讓他們驚歎。然後說「當然，如果我們添加…它會變得更好」，一邊假裝它不重要。然後坐下來，等待他們開始要求您添加最初您就想要的功能。人們更容易加入一個持續成功的行動。讓他們瞥一眼未來，您就能讓他們團結起來[註 5]。

提示 6	成為改變的催化劑

村民的角度

另一方面，石湯的故事也是一種溫和而漸進的欺騙。村民們太過專注地想著石頭，忘記了世界上的一切。我們每天都會上這種當，事情就這樣悄悄發生在我們身上。

註 5　當您這樣做的時候，來自海軍少將博士 Grace Hopper 的一句話可能可以安慰您：「請求原諒比獲得許可容易。」

我們都見過這些症狀。專案緩慢而無情地完全失去控制，大多數軟體災難開始的時候都很小，不會引起人們的注意，專案超時每天延後一點點，系統的各個功能漸漸逐一地偏離了它們的規格，而程式碼中被添加一個又一個補丁，直到都看不出原始程式碼的樣貌。往往是一些小事情的累積破壞了士氣和團隊。

> **提示 7** 記得大方向

說實話，我們從來沒試過。但是「他們」說，如果您把一隻青蛙扔進沸水裡，它會馬上跳出來。然而，如果您把青蛙放在一鍋冷水裡，然後逐漸加熱，青蛙不會注意到溫度的緩慢上升，直到煮熟為止。

請注意，青蛙的問題與第 7 頁的主題 3，軟體亂度中討論的破窗問題不同。在破窗理論中，人們失去了對抗亂度的意志，因為他們認為沒有人關心，而青蛙只是沒有注意到變化。

不要像傳說中的青蛙。請您關注大局，經常環顧您周圍發生的事情，而不僅僅是您自己在做什麼。

相關章節包括

- 主題 1，這是您的人生，第 2 頁
- 主題 38，靠巧合寫程式，第 232 頁

挑戰題

- 當初在審閱第一版的草稿時，John Lakos 提出了以下問題：士兵們漸進地欺騙了村民，但他們促成的變化對他們都有好處。然而，透過逐步欺騙青蛙，您正在傷害它。當您嘗試催化改變時，您能確定您是在做石頭湯還是青蛙湯嗎？這個決定是主觀的還是客觀的？

- 請不要看並快速回答，您頭頂的天花板上有多少盞燈？房間裡有幾個出口？有多少人？有沒有什麼東西被亂放，看起來不屬於這裡？這是一項關於「態勢感知」（*situational awarenss*）的練習，從童子軍到海豹突擊隊，都在練習這種技巧。養成仔細觀察周圍環境的習慣，然後對您的專案做同樣的事情。

5 夠好的軟體

為了追求更好，我們毀損了原已夠好的。

➤ 莎士比亞，李爾王 *1.4*

有一個（有點）古老的笑話，一家公司向一家日本製造商下了 100,000 個積體電路的訂單。該規格的一部分是不良率：1/10,000。幾週後，貨物來了：一個大盒子裡裝著大量的積體電路，另一個小盒子裡只有 10 個。小盒子上貼著一張標籤，上面寫著：「這些是有缺陷的」。

要是我們真能控制品質就好了。但現實世界就是不讓我們生產出真正完美的產品，尤其是沒有 bug 的軟體。時間、技術和性情都對我們不利。

然而，這並不令人沮喪。正如 Ed Yourdon 在《*IEEE Software*》的一篇文章中所描述的，足夠好的軟體就是最好的軟體 *[You95]* 時，您可以訓練自己撰寫足夠好的軟體——對於您的使用者、對於未來的維護者、對於您自己的內心平靜來說足夠好的軟體。您會發現您更有效率，您的使用者也更滿意。您可能會發現您的程式實際上越早完成越好。

在我們進一步討論之前，我們需要特別說明一下將要討論的內容。這裡所稱的「足夠好」一詞並不代表著程式碼可以隨便寫或效率低下。所有系統都必須滿足使用者的要求才能成功，並滿足基本的性能、隱私和安全標準。我們只是主張給使用者一個機會，讓他們參與到這個過程中來，決定什麼時候您的產品能夠滿足他們的需求。

讓您的使用者參與功能取捨

通常您是為別人寫軟體，所以通常您會記得要去找出他們想要什麼[註6]。但是您可曾問過他們想要多好的軟體嗎？雖然有些情況下我們沒得選，例如您正在開發的是心臟起搏器、自動駕駛軟體或一個將被廣泛傳播的底層函式庫，那麼對於「好」的定義將更加嚴格，您的選擇將更加有限。

但是，如果您正在開發一個全新的產品，您將面臨不同的限制。行銷人員必須遵守承諾，最終的終端使用者可能已經根據交付計畫制訂了計畫，而您的公司肯定會有現金流限制。如果只是因為想給程式添加新功能，或者多做一次程式碼優化，就忽略上面這些使用者的需求，是一種不專業的表現。我們並不是在恐嚇您：承諾不可能實現的完成時間，和為了滿足最後期限而偷工減料，兩者不專業的程度是相同的。

您生產的系統的範圍和品質應該作為系統需求的一部分進行討論。

提示 8	把品質看成一種需求

一般情況來說，您會遇到需要權衡利弊的情況。令人驚訝的是，許多使用者寧願馬上就有堪用的軟體，也不願花一年的時間等待光鮮亮麗、功能齊全的版本（事實上，他們一年之後所需要的可能完全不同），許多預算緊張的 IT 部門會同意這一點。今天的可用軟體往往比明天的完美軟體更受歡迎。如果您儘早給您的使用者提供可以上手使用東西，他們的回饋通常會引導您找到更好的最終解決方案（參見主題 12，曳光彈，第 58 頁）。

知道何時停手

在某些方面，程式設計就像繪畫。您從一張空白的畫布和一些基本的原材料開始。您使用科學、藝術和工藝的結合來決定如何使用它們。您可以勾畫出一個

註 6　我試圖在這埋一個笑話梗！

整體形狀，繪製底層環境，然後填充細節。您總是帶著批判的眼光回顧自己所做的事情，您會時不時地扔掉一張畫布，然後重新開始。

但是藝術家會告訴您，如果您不知道什麼時候停止，就會毀掉之前所有的努力。如果您不停地一層一層地添加細節，您的畫作最後將會被顏料淹沒。

請不要因為過分的修飾和精煉而破壞了一個完美的程式，去做下一件事，讓您的程式碼獨處一段時間。它可能並不完美，但您也無需擔心：它不可能是完美的（在第 7 章，當您寫程式時，第 225 頁，我們將討論在不完美的世界中開發程式碼的哲學）。

相關章節包括

- 主題 45，需求坑，第 288 頁
- 主題 46，解開不可能的謎題，第 298 頁

挑戰題

- 查看您經常使用的軟體工具和作業系統。您能找到任何證據來證明這些組織和 / 或開發人員願意交付他們知道並不完美的軟體嗎？身為一個使用者的您，您是願意（1）等待他們把所有的錯誤都清除掉，（2）擁有複雜的軟體並接受一些錯誤，還是（3）選擇更簡、單錯誤也更少的軟體？
- 讓我們思考一下模組化對軟體交付的影響。若有一個使用非常鬆散耦合設計的模組或微服務的系統，和另一個緊密耦合的軟體相比，提升到所需的品質所需要的時間是比較多還是少？這兩種方法的優點或缺點是什麼？
- 您能想到受功能膨脹（*feature bloat*）所害的熱門軟體嗎？也就是說，這種軟體含的功能比您可能使用的多得多，每個功能都有可能帶來更多 bug 和安全性漏洞，使得您真的需要使用的功能更難被找到和使用。您自己有掉入這樣的陷阱的經驗嗎？