推薦序

在我作為電腦科學和軟體工程教育者的 30 年職業生涯中，特別是自從我 2001 年在工業界短暫任職以來，很少有其他技術會像自動化單元測試——也就是將測試驅動開發（test-driven development, TDD）操作化為一種特定但廣泛適用的技術的泛用作法——那樣的影響並滲透到我的教學（和研究）中。

就在採用了 Martin Fowler 2003 年的教科書《*UML Distilled*》（第 3 版），作為我的物件導向開發課程的 UML 參考書之後，我仍然記得我就能夠具體的瞭解 TDD 了，這就像是它的副作用一樣。在那本書中，Martin 討論了成功的迭代開發過程中通常存在的三個關鍵實務：自動化迴歸測試、重構和持續整合。這樣簡潔的描述引起了我的強烈共鳴，我一直很享受說服我的學生要透過編寫額外的程式碼，來測試他們的其餘程式碼，並以所得到的各色測試結果來接收即時回饋，以獲得更多的樂趣。

大概在十年後，也就是大約 2012 年時，當我開始收聽一些關於軟體架構的 *Software Engineering Radio* 播客（podcast）時，我的另一個驚喜時刻出現了。我正在閱讀播客中提到的一些參考資料，並在 “Uncle Bob” Robert C. Martin 的書《*Agile Software Development: Principles, Patterns, and Practices*》中看到了一個名為 “偶然的架構（Serendipitous Architecture）” 的簡短小節，其中的討論聚焦於如何使程式碼自動可測試，從而引致良好的、可維護的架構。

總之，上面這兩點強調了自動化測試是如何將流程和架構、以及功能性和非功能性需求聯繫在一起的方式：藉由讓我們對程式碼可以滿足功能性需求的程度這件事更有信心之後，可測試性可以認為是最重要的非功能性需求。

今年夏天，差不多又過了十年後，Saleem Siddiqui 因為他的書而與我聯繫。順道一提，明年將是 Saleem 和我一起修讀三門研究生課程的第 25 週年！看到他成為一名成功的技術專業人士——就像 Martin Fowler 一樣的思想工作者——和作家，我感到非常欣慰。我很榮幸他讓我為他的書寫前言，也讓我想更加了解他對 TDD 的想法。

Saleem 的書最讓我興奮的是，它使用日常生活中非常熟悉的執行範例，以一種動手操作但又有條不紊的方式讓讀者參與到 TDD 過程裡。無論是哪種程式語言，紅－綠－重構週期都會為流程定下基調。這個金融貨幣領域的接續性功能是具體且易於關聯的，但會引導讀者逐步應對更複雜的挑戰，從而建立信心、揭示微妙的取捨、並喚醒進一步探索的好奇心。沿著輪廓（profile）、目的（purpose）和程序（process）三個維度進行的最終程式碼審查，更整合了在這個過程中所收集的洞察。

透過使用三種具有相當互補設計理念的常用語言——其中 JavaScript 和 Python 已經在市場上佔據領先地位，而 Go 正在迅速崛起—— Saleem 為 TDD 方法的廣泛適用性提供了強而有力的理由。此外，他還為讀者提供了額外的接觸點、和對語言設計與剛才提到的 “ 三個 P” 之間關係的認識。

我非常希望 Saleem 的書，能夠和那些被 Go、JavaScript 和 Python 等有影響力的語言所吸引的新一代軟體開發人員產生共鳴，並將他們拉上測試驅動開發的良性路徑，從而產生加乘效應。借用偉大的爵士薩克斯風演奏家 Cannonball Adderley 在向紐約現場觀眾描述時髦（hipness）一詞時所說的話：這不是一種心態，而是生活中的事實。

—— Konstantin Läufer
電腦科學教授，
芝加哥洛約拉大學
芝加哥，伊利諾州，2021 年 9 月

前言

測試驅動開發是在程式設計過程中管理恐懼的一種方式。

—Kent Beck

我們真是太幸運了！多年來，我們一直在進行測試驅動開發。

自從為水星太空計畫（Mercury Space Program）編寫程式碼的開發人員實踐了 Punch Card（打孔卡片）TDD（test-driven development，測試驅動開發）（*https://oreil.ly/pKpSZ*）以來，已經過去了幾十年了。促進採用測試驅動開發的 XUnit 程式庫的問世可以追溯到世紀之交。事實上，撰寫了《*Test-Driven Development: By Example*》（Addison-Wesley Professional，2002）並且開發了 Junit 框架的 Kent Beck，稱自己“重新發現”（而不是發明）了 TDD 的實務（*https://oreil.ly/zDyBr*）。這句話是他謙卑的證據，但也是事實。TDD 與軟體開發本身一樣古老。

那為什麼測試驅動開發和標準的程式碼編寫方式還離的很遠呢？為什麼當有進度壓力時、或者需要削減 IT 預算時、或者（這是我個人最喜歡的原因）希望“提高軟體交付團隊的速度”時，它都是首先被犧牲的實務呢？這些原因都會被拿出來講，儘管有現成的經驗和實驗證據（*https://oreil.ly/2Xxyb*）來證明 TDD 可以減少缺陷數量、建立了更簡單的設計、並能提高開發人員對自己程式碼的信心。

為什麼 TDD 會被勉強的採用而也會被輕易的放棄呢？或許下面這些經常會從那些不願意實踐這件事的人那裡聽到的論點可以解釋原因：

我不知道從哪裡以及如何開始。

也許最常見的原因是缺乏認識和曝光。像任何其他技能一樣，以測試驅動的風格來編寫程式碼是需要學習的。許多開發人員不是沒有外部誘因（時間、資源、指導、鼓勵），就是沒有內部動機（克服自己的不情願和恐懼）來學習這項技能。

***TDD* 適用於玩具程式（*toy program*）或程式設計面試，但不適用於編寫“真實世界”程式碼。**

這個想法雖然不正確，但可以理解。大多數測試驅動的開發教程和書籍——包括這本書——都被限制成只能從一個常見領域中挑選一些相對簡單的例子來說明。我們很難使用從商業性部署的應用程式（例如，從金融機構、醫療保健管理系統或自動駕駛汽車）中所提取的實際程式碼來編寫 TDD 文章或書籍。一方面，許多這樣的真實世界程式碼是專屬的，而不是開源的。另一方面，作者的工作應該是展示來自於那些會對最大受眾具有最廣泛吸引力的領域的程式碼。在高度專業的領域背景之下來展示 TDD 是不合邏輯的，而且近乎蒙昧主義（obscurantism）。要這樣做首先必須對該領域的晦澀行話進行冗長的解釋。而這將違背本書作者的目的：讓 TDD 易於理解、平易近人、甚至是可愛的。

儘管在 TDD 文獻中使用真實世界的程式碼存在著這些障礙，但開發人員經常使用測試驅動開發來編寫（production）生產軟體。也許最好和最有說服力的例子是 JUnit 框架（framework）（*https://oreil.ly/UCPcg*）本身的單元測試（unit test）套件。另外 Linux 核心程式（Kernel）——可能是世界上最頻繁被使用的軟體——正在透過單元測試獲得改進（*https://oreil.ly/hBbq0*）。

事後再編寫測試就足夠了；*TDD* 過於嚴格和 / 或迂腐。

這種說法比聽到有人偶爾抱怨說“單元測試被高估了”（*https://oreil.ly/Y7S5M*）還更令人耳目一新！在編寫生產程式碼之後再編寫測試，還是對根本不編寫測試這個作法的改進。任何能夠提高開發人員對其程式碼的信心、降低意外的複雜性、並提供真實說明文件的東西都是一件好事。但是，在編寫生產程式碼之前就編寫單元測試提供了一種強制性，可以防止產生不可預期的複雜性。

TDD 引導我們進行更簡單的設計，因為它提供了以下兩個實用規則作為保護：

1. 只編寫生產程式碼來修復失敗的測試。
2. 當且僅當測試為綠色時才積極重構（refactor）。

測試驅動開發能否保證我們編寫的所有程式碼，都會自動且不可避免的成為最簡單的程式碼？不是，並不會如此。沒有任何實務、規則、書籍或宣言（manifesto）可以做到這一點。而是由將這些實務帶入生活的人們來確保會達成和保留簡單性。

本書的內容會解釋並指引測試驅動開發是如何在三種不同的程式設計語言中運作的。其目的是向開發人員灌輸使用 TDD 作為常規實務的習慣和自信。這可能是一個雄心勃勃的目標，但我希望它不是難以捉摸的。

什麼是測試驅動開發？

測試驅動開發是一種設計和結構化程式碼的技術，它鼓勵簡單性並增加人們對程式碼的信心，即使程式碼的大小會有所增加。

讓我們來看一下這個定義的各個部分。

一種技術

測試驅動開發是一種技術。的確，這種技術源於一組關於程式碼的信念，也就是：

- 簡單性——也就是將未完成的工作量最大化的藝術，是必不可少的[1]
- 顯而易見性（obviousness）和清晰性（clarity）比起聰明（cleverness）而言更是美德
- 編寫整潔的程式碼是成功的關鍵組件

儘管植根於這些信念，但實際上，TDD 就是一種技術。就像騎自行車、揉麵團或解微分方程一樣，這是一項沒有人天生就會的技能，每個人都必須要學習。

除了本節之外，本書並未詳述測試驅動開發背後的信念系統。我們假設了您已經相信了它，或者您願意嘗試將 TDD 作為一項新的（或被遺忘的）技能。

該技術的機制——首先編寫一個失敗的單元測試、然後輕快的編寫一個剛好夠讓它通過測試的程式碼、然後再花時間清理——佔據了本書的大部分內容。會有足夠的機會讓您親自嘗試這種技術。

歸根究柢來說，學習一項技能並且讓自己充滿了支持它的信念會更令人滿意——就像騎自行車時提醒自己騎自行車對健康和環境是有益的，會讓自己更愉快一樣！

[1] 這種簡單性的定義體現在敏捷宣言（Agile Manifesto）（*https://agilemanifesto.org/principles.html*）的 12 條原則之一裡面。

設計和結構程式碼

請注意，TDD 並不是在基本上就與測試程式碼相關。我們確實使用了單元測試來驅動程式碼，但 TDD 的目的是改進程式碼的設計和結構。

這個重點是至關重要的。如果 TDD 只是和測試相關，我們就無法真正的提出一個在編寫業務程式碼**之前**而不是**之後**來編寫測試的有效案例。設計更好的軟體是激勵我們前進的目標；測試只是達成這項進步的工具。我們最終透過 TDD 所完成的單元測試只是一種額外的好處；主要的好處是我們所得到的設計的簡單性。

我們如何達成這種簡單性呢？它是透過**紅－綠－重構**（*red-green-refactor*）的機制達成的，第 1 章開始會有詳細描述。

對簡單性的偏見

簡單性不僅僅是一個深奧的概念。在軟體中，我們甚至可以測量它。每個功能的程式碼行數變得更少、循環複雜度（cyclomatic complexity）變得更低（*https://oreil.ly/5Gj2b*）、副作用變的更少、執行時期或記憶體要求變得更小——這些（或其他）要求的任何一部分組合都可以作為簡單性的客觀衡量標準。

測試驅動開發，透過強迫我們製作出“最簡單的可行東西”（也就是能通過所有測試的東西），來不斷的推動我們朝著這些簡單性指標前進。我們不允許因為“以備不時之需”或“我們可以預期它快被用到”而添加多餘的程式碼。我們必須首先編寫一個失敗的測試來證明編寫這樣的程式碼是合理的。先編寫測試這樣的行為是一種強制功能——這迫使我們**儘早**處理不可預期的複雜性。如果我們即將開發的功能的定義並不明確，或者我們對它的理解存在著缺陷，我們會發現很難先編寫出一個好的測試。這將迫使我們在編寫出一行生產程式碼**之前**先解決這些問題。這就是 TDD 的優點：透過了運用測試來驅動我們的程式碼這樣的紀律的練習，讓我們在每個關鍵點都消除了不可預期的複雜性。

這種優點並不神秘：使用測試驅動開發不會讓您的開發時間、程式碼行數或者缺陷數量減少一半。它**可以**做的是去除您引入人造和人為複雜性的這類誘惑。最終的程式碼——由先編寫出失敗測試的這個原則來驅動——將成為完成工作的最直接方式，也就是滿足測試之需求的最簡單程式碼。

增加信心

程式碼，尤其是我們自己編寫的程式碼，應該要能夠激發信心。這種信心雖然本身是一種模糊的感覺，但奠基於對可預測性的期望。我們會對可以預測其行為的事物充滿信心。如果街角的咖啡店在某天少收了我的錢，然後第二天時再多收了同樣多的錢，即使我在兩天內是收支平衡的，我也可能會對員工失去信心。我們更看重**規律性**（*regularity*）和**可預測性**（*predictability*），而不是**淨值**（*net value*），這是人類的天性。世界上最幸運的賭徒，可能剛剛在輪盤賭桌上連續贏了 10 次，他不會說他們“信任”輪盤或對它有“信心”。我們對可預測性的喜好即使對於愚蠢的運氣來說也是存在的。

測試驅動開發增加了我們對程式碼的信心，因為每個新的測試都以新的和以前未經測試的方式來改變系統——至少從字面上看來就是如此！隨著時間的推移，我們所建立的測試套件可以保護我們免受迴歸失敗的影響。

這種不斷增加的測試組合正是為何會隨著程式碼大小的增長，程式碼的品質和我們對它的信心也會增加的原因。

這本書是給誰看的？

這是一本給開發人員（即編寫軟體的人）的書。

這個職業有許多專業頭銜：“軟體工程師”、“應用程式架構師”、“devops 工程師”、“測試自動化工程師”、“程式設計師”、“駭客”、“程式碼低語者（code whisperer）”等等。這些頭銜可能會是令人印象深刻的或謙遜的、時尚的或莊重的、傳統的或現代的。然而，自稱這些頭銜的人都有一個共通點：他們每星期至少有一部分時間——如果不是每天的話——在電腦前閱讀和 / 或編寫程式原始碼。

我選擇了**開發人員**（*developer*）這個詞來代表這個社群，而我是其中一個謙虛又感激的成員。

編寫程式碼是人們最自由和最平等的活動之一。理論上，一個人要進行這件事唯一所需要的生理能力就是擁有一個大腦。年齡、社會性別（gender）、生理性別（sex）、國籍、出生地等都不應該成為障礙。身體殘疾也不應成為障礙。

然而，如果認為現實是如此乾淨或公平的，那就太過於天真了。對計算資源的存取是不公平的。一定程度的財富、免於匱乏的自由和安全還是必要的。不良的軟體編寫、不良的硬體設計，以及無數其他的可用性限制都會進一步阻礙了存取，這些限制造成不是所有的人都可以只基於他們的興趣和努力來學習程式設計。

我試圖讓盡多的人能夠閱讀這本書。特別一提的是，我試圖讓身體殘疾的人可以接受它。影像都具有替代文本（alt-text）以方便電子方式閱讀。程式碼可透過 GitHub 獲得。並且文體很簡單。

就經驗而言，本書既適用於仍在學習如何設計程式的人，也適用於那些已經知道如何設計程式的人。如果您正在學習本書中三種語言中的其中一種（或多種），那麼您就屬於目標受眾。

但是，本書並不會教授任何語言的程式設計**基礎知識**，包括 Go、JavaScript 或 Python。能以其中的至少一種程式設計語言來讀寫程式碼的能力是一項要求。如果您對程式設計完全陌生，那麼在繼續閱讀本書之前，最好先去鞏固運用這三種語言之一來編寫程式碼的基礎。

本書的甜蜜點涵蓋了那些已經初嚐程式設計滋味的開發人員，一直到經驗豐富的架構師，如圖 P-1 所示（Kent Beck 是一個異常值）。

圖 P-1　這是一本寫給軟體開發人員的書

編寫程式碼時而令人振奮，時而令人惱火。然而，即使在最令人沮喪的情況下，我們也應該**始終**保持一絲絲樂觀和一蒲式耳（bushel）的信心，也就是我們終究可以讓程式碼按照我們的意願行事。持之以恆進行，您會發現讀完這本書的旅程是富有成果的，並且在讀完第 14 章之後，您還會想要長期享受以測試驅動方式來編寫程式碼的樂趣。

閱讀本書的先決條件是什麼？

在設備和技術實力上，您應該：

- 可以存取具有網際網路連接的電腦。
- 能夠在該電腦上安裝和移除軟體。也就是說，您在該電腦上的存取不應受到限制；在大多數情況下，這需要在該電腦上具有“管理員（Administrator）”或“超級使用者（Superuser）”的存取權限。
- 能夠在該電腦上啟動和使用殼層（shell）程式、Web 瀏覽器、文本編輯器以及可選的整合開發環境（integrated development environment, IDE）。
- 已安裝（或能夠安裝）本書所使用的任一種語言的執行時期（runtime）工具。
- 能夠用本書所使用的任一種語言來編寫和執行一個簡單的程式——“Hello World”。

第 0 章第 1 頁的“設定您的開發環境”有更多的安裝細節。

如何閱讀本書

本書的主題是“如何在 Go、JavaScript 和 Python 中進行測試驅動開發”。雖然其中所討論的概念都適用於所有的三種語言，但對於每種語言的處理還是需要對每章中的材料進行一些切割。學習測試驅動開發（就像任何其他您已獲得的技能一樣）的最佳方式就是透過練習。我鼓勵您閱讀內文並自己編寫程式碼。我將這種風格稱為“跟隨書本（*following the book*）”——因為它包括主動的閱讀和主動的程式設計。

要充分利用本書，請用所有三種語言來編寫 Money 範例的程式碼。

大多數章節都有適用於所有三種語言的通用部分。接下來是特定於語言的小節，其中描述和開發了三種語言其中之一的程式碼。這些特定於語言的部分總是用它們的標題清楚的標記出來：*Go*、*JavaScript* 或 *Python*。在每一章的末尾都有一兩個小節來總結，我們到目前為止已經完成的工作以及接下來會進行什麼。

第 5 章到第 7 章是獨一無二的，因為它們分別專門處理以下三種語言中的其中一種：Go、JavaScript 和 Python。

圖 P-2 顯示了一個流程圖，描述了本書的佈局以及遵循它的不同方法。

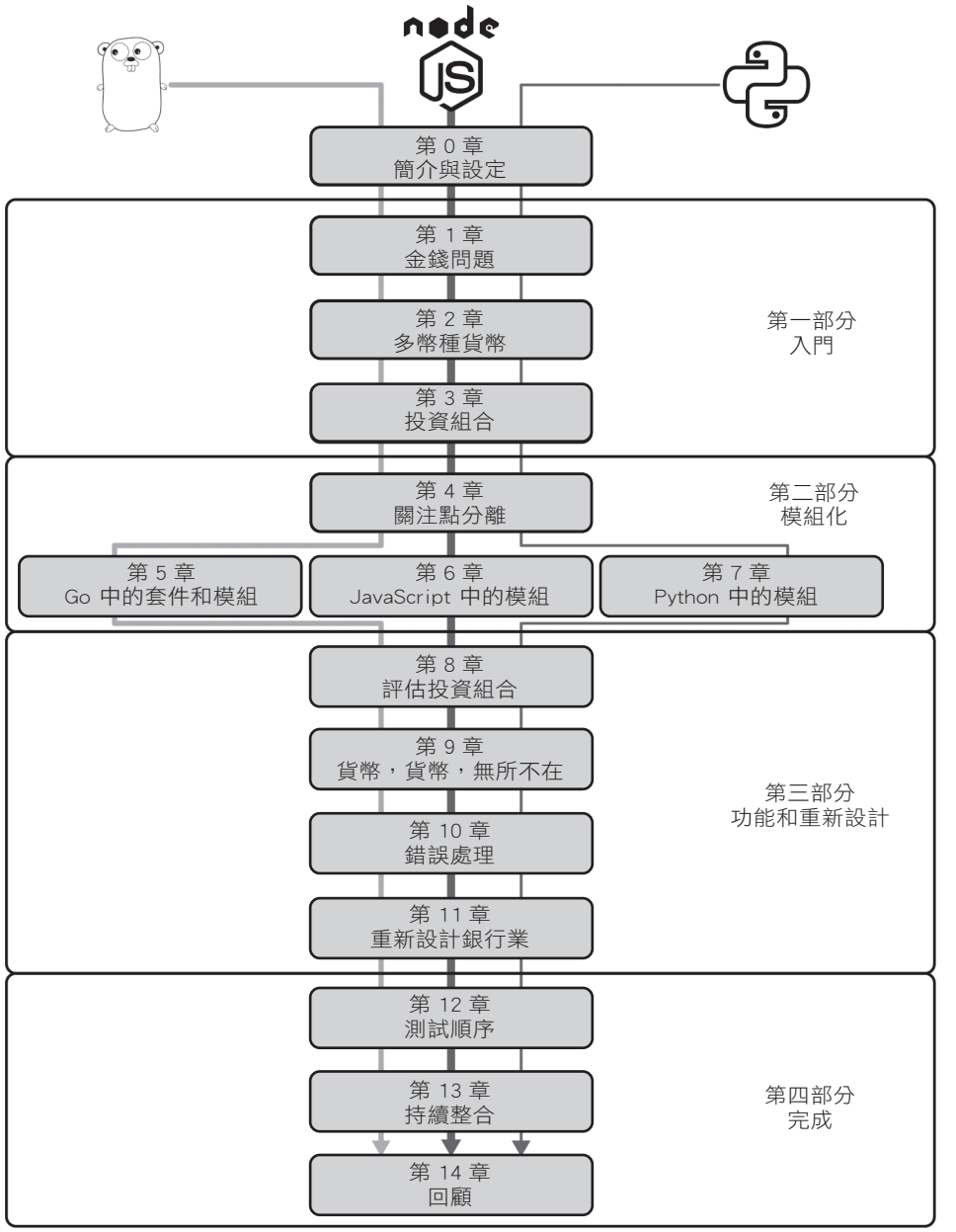


圖 P-2　閱讀本書的流程圖

以下是一些關於如何最能夠遵循這本書的“閱讀路徑”。

一次遵循本書的一種語言

如果以下這些條件中的一個或多個適用於您，我推薦您遵循此路徑：

1. 在處理其他兩種語言之前，我渴望先深入研究其中一種語言。

2. 我特別好奇（或懷疑！）關於 TDD 是如何在三種語言中的其中一種中運作的。

3. 我最好的學習方式是一次只使用一種語言，而不是同時使用多種語言。

請按照圖 P-2 所示的流程圖進行，一次遵循一條線。例如，如果您渴望首先學習 Go 中的 TDD，第一次閱讀中請跳過標記為 JavaScript 和 Python 的部分。然後再讀一遍這本書的 JavaScript 部分，第三遍再用 Python 來完成。或者您可以按不同的順序來選擇語言。第二次和第三次時*應該*會比第一次更快；但是，請為每種語言的獨特怪癖做好心理準備！

如果您以這種方式閱讀本書，您會發現用每種語言連續編寫程式碼，可以讓您更深入的瞭解 TDD 並把它*當作是原則*——而不只是把測試細節*當作是語言特性*。養成編寫測試的習慣是必要的；但是，瞭解測試驅動開發為何要跨語言工作的原因更重要。

先用兩種語言遵循本書，然後用第三種語言遵循本書

如果您適用於以下的任何陳述，我推薦此路徑：

1. 我想用兩種語言來建構和比較同一個問題的解決方案。

2. 我對其中一種語言不太熟，想在其他兩種語言之後再處理它。

3. 我可以同時使用兩種語言來編寫程式碼，但很難同時兼顧這三種語言。

請按照圖 P-2 所示的流程圖進行，一次遵循二條線。在您完成了兩種語言的金錢問題之後，再透過這本書來學習第三種語言。

您可能想在第一遍時就用其中兩種語言學習，但無法決定要將哪種語言延後到第二遍才閱讀。以下是有關如何從三種語言中選擇兩種語言的一些建議：

1. 您想對動態型別語言與靜態型別語言進行比較、並讓語言技術堆疊保持簡單嗎？請先遵循 Go 和 Python，然後是 JavaScript。

2. 您準備要學習用兩種不同語言以對比方式來建構程式碼的方法，並準備處理技術堆疊的變化了嗎？請先遵循 Go 和 JavaScript，再遵循 Python。

3. 您想比較和對比兩種動態型別語言嗎？請先遵循 JavaScript 和 Python，然後是 Go。

如果您以這種方式閱讀本書，您將很快發現用多種語言進行 TDD 的異同。雖然語言中的語法和設計的變化造成了明顯的差異，但您可能會驚訝於 TDD 規則在您如何編寫程式碼的作法的滲透程度，無論您是使用哪種語言來編寫程式碼。

同時用所有的三種語言來遵循本書

如果您適用於以下任何陳述，我推薦此路徑：

1. 您想透過學習三種語言的對比和相似之處來獲得最大的價值。

2. 您發現從頭到尾閱讀一本書比多次閱讀更容易。

3. 您對所有的三種語言都有一定的經驗，但沒有在其中任何一種語言中練習過 TDD。

如果您可以同時用三種語言來編寫程式碼而不會不知所措，我推薦這條路徑。

無論您選擇哪種路徑，請注意，當您編寫程式碼時，您可能會面臨與您的特定開發環境有關的挑戰。雖然本書中的程式碼已經過正確性測試（並且它的持續整合建構是綠色的（*https://github.com/saleem/tdd-book-code/actions*）），但這並不意味著它一開始就可以在您的電腦上運行。（相反的，我幾乎可以保證您會在學習曲線上發現有趣的陡峭部分）。TDD 的主要好處之一是您可以控制前進的速度。當您被卡住時，請放慢速度。如果您用較小的增量來取得進展，則更容易找到程式碼誤入歧途的地方。編寫軟體意味著要處理錯誤的依賴關係、不可靠的網路連接、古怪的工具以及程式碼所繼承的數千種天然衝擊。當您感到不知所措時請放慢速度：讓您的更改變得更小且更離散。請記住：TDD 是一種管理對寫程式的恐懼的方法！

本書字體慣例

本書中使用了兩類需要解釋的慣例：編排的和語境的。

編排慣例

本書中的內文採用這句話中所使用的字體類型。它是用來閱讀的，而不是作為程式碼逐字輸入的。當內文中使用的單字也用於程式碼中時——例如 `class`、`interface` 或 `Exception` ——則使用固定寬度的字體。這會提醒您該術語正在或將在程式碼中使用（拼字完全相同）。

較長的程式碼片段被分成各自的區塊，如下所示。

```
package main

import "fmt"

...❶

func main() {
    fmt.Println("hello world")
}
```

❶ 省略號表示不相關的程式碼或輸出已被省略。

程式碼區塊中的所有內容可以是您逐字輸入的內容，**或者**是程式產生的文字輸出，但有兩個例外。

1. 在程式碼區塊中，**省略號**（`...`）用於表示省略的程式碼或省略的輸出。在這兩種情況下，省略的內容都與當前主題無關。您**不**應在程式碼中鍵入這些省略號或期望在輸出中看到它們。上面的程式碼區塊中顯示了一個這樣的範例。
2. 在顯示輸出的程式碼區塊中，可能有**暫時值**——記憶體位址、時間戳記、經過的時間、行號、自動產生的檔案名稱等——對您來說幾乎一定會和這裡有所不同。閱讀此類輸出時，您可以放心的忽略**特定的**暫時值，例如以下區塊中的記憶體位址：

```
AssertionError: <money.Money object at 0x10417b2e0> !=
                <money.Money object at 0x10417b400>
```

提示是在您編寫程式碼時對您有幫助的建議。它們與正文分開以便於參考。

對主題至關重要的重要資訊是這樣標識的。通常會有資源的超連結或註腳，可提供有關該主題的更多資訊。

在大多數章節中，都會對這三種語言的程式碼進行深入的開發和討論（例外的是第 5、6 和 7 章，它們分別專門討論 Go、JavaScript 和 Python）。為了區分每種語言的討論，標題以及位於頁邊空白處的圖示被用來指出該語言的專有段落。請留意這三個標題和圖示：

- *Go* 
- *JavaScript* 
- *Python* 

詞彙慣例

這本書討論了核心軟體概念，並用三種不同語言的程式碼來支持這些討論。這些語言在各自的術語上存在很大差異，因此在討論共同概念時會出現一些挑戰。

例如，Go 沒有類別或基於類別的繼承。JavaScript 的型別系統有基於原型（prototype）的物件——這意味著一切都是真正的物件，其中包括通常會被認為是類別的東西。

本書中使用的 Python 具有更“傳統的”基於類別的物件。[2] 像是“我們將建立一個名為 Money 的新類別”這樣的句子不僅令人困惑，而且在 Go 的語境中解釋時會是完全錯誤的。

為了減少可能出現的混淆，我採用了表 P-1 中所示的通用術語來參照關鍵概念。

表 P-1　本書使用的通用術語

術語	意義	Go 中的等價物	JavaScript 中的等價物	Python 中的等價物
實體（**Entity**）	一個單一的、獨立有意義的領域概念；一個關鍵名詞	**Struct** 型別	類別（**Class**）	類別（**Class**）
物件（**Object**）	一個實體的實例；物化名詞	**Struct** 實例	物件（**Object**）	物件（**Object**）
序列（**Sequence**）	動態長度物件的循序串列	切片（**Slice**）	陣列（**Array**）	陣列（**Array**）
雜湊圖（**Hashmap**）	一組（鍵值）對，其中鍵和值都可以是任意物件，並且沒有兩個鍵可以相同	映射（**Map**）	映射（**Map**）	字典（**Dictionary**）
函數（**Function**）	具有給定名稱的運算集合；函數可能（或可能沒有）具有輸入和輸出的實體，但它們不直接與任何一個實體相關聯	函數（**Function**）	函數（**Function**）	函數（**Function**）
方法（**Method**）	與實體關聯的函數。一個方法被稱為“呼叫在”該實體的一個實例（即一個物件）	方法（**Method**）	方法（**Method**）	方法（**Method**）
發出錯誤信號（**Signal an error**）	函數或方法用來指出失敗的機制	錯誤傳回值（通常是函數 / 方法的最後一個傳回值）	拋出例外（**Throw an exception**）	引發例外（**Raise an exception**）

我們的目標是使用能夠解釋概念的術語，而不是偏愛某一種程式語言的術語。畢竟，本書最大的收穫應該是，測試驅動開發是一門可以在任何程式語言中實踐的學科。

在本書涉及到三種語言之一的那些部分（在標題中會明確標記）中，在其內文中會使用特定於該種語言的術語。例如，在 Go 段落中，將有“定義一個名為 Money 的新結構（struct）”這樣的指令。這樣的語境清楚的表明該指令是特定於某個特定語言的。

[2] Python 在支援物件導向程式設計（OOP）方面非常順暢。例如，請參閱 prototype.py（*https://oreil.ly/ZKivt*），它在 Python 中實作了基於原型的物件。

使用程式碼範例

本書的原始碼可以在 *https://github.com/saleem/tdd-book-code* 取得。如果您有任何技術上或使用程式碼範例的問題，請發送電子郵件至 *bookquestions@oreilly.com*。

本書旨在幫助您學習和實踐測試驅動開發的藝術。通常，您可以在程式和說明文件中使用本書中提供的任何程式碼。除非您要複製程式碼的重要部分，否則您無需聯繫我們以獲得許可。例如，編寫一個使用了本書中多個程式碼區塊的程式不需要獲得許可。銷售或散佈 O'Reilly 書籍中的範例確實需要獲得許可。透過引用本書和引用範例程式碼來回答問題不需要獲得許可。將本書中的大量範例程式碼合併到您的產品說明文件中確實需要許可。

我們很感謝您在引用它們時標明出處（但不強制要求）。出處通常包括標題、作者、出版商和 ISBN。例如："*Learning Test-Driven Development* by Saleem Siddiqui（O'Reilly). Copyright 2022 Saleem Siddiqui, 978-1-098-10647-8"

如果您認為您對程式碼範例的使用超出了合理使用或上述許可的範圍，請隨時透過 *permissions@oreilly.com* 與我們聯繫。

TDD — 為什麼

對 TDD ——以及隱含著就是對這本書——的批評可以有許多種形式。其中一些具有創造性的幽默感，例如圖 P-3 中 Jim Kersey 令人耳目一新的漫畫。

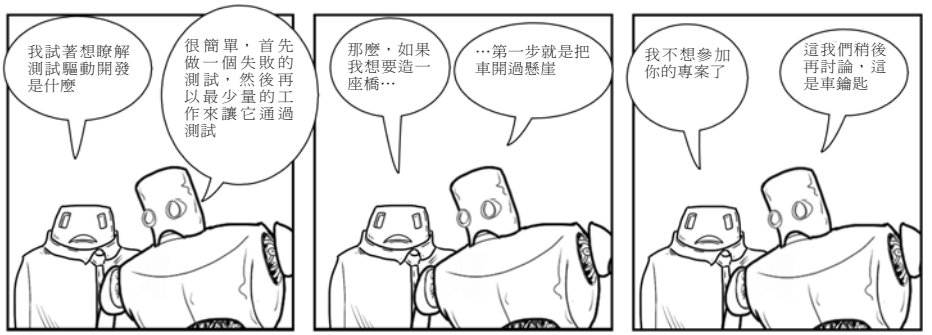


圖 P-3 TDD 幽默：不要跨過尚未建造的橋梁！（來源：*https://robotkersey.com*）

正經一點，對這本書的內容和結構有疑問是很自然的。以下是幾個對此類問題的回答。

為什麼本書要使用 Go、JavaScript 和 Python ？

本書使用 Go、JavaScript 和 Python 作為三種語言來示範測試驅動開發的實務。一個合理的問題是：為什麼是這三種語言？

以下是一些原因。

1. 變化性

本書中的三種語言代表了多種設計選項，如表 P-2 所示。

表 P-2　Go、JavaScript 和 Python 的比較

特徵	Go	JavaScript	Python
物件導向	"是與否"（*https://oreil.ly/M3u1P*）	是（作為符合 **ES.next** 的語言）	是
靜態與動態型別	靜態型別	動態型別	動態型別
外顯式或內隱式型別	大部分為外顯式，變數型別可以是內隱式的	內隱式型別	內隱式型別
自動型別強制	無型別強制	部分型別強制（用於 **Boolean**、**Number**、**String**, **Object**）。對任意 `Class` 型別沒有強制	一些內隱式型別強制（例如 `0` 和 `""` 代表否）
例外機制	按照慣例，方法的第二種傳回型別是 `error`，呼叫者必須外顯式的檢查這是否為 `nil`	關鍵字 `throw` 用於發出例外信號，`try ... catch` 用於回應它	關鍵字 `raise` 用於發出例外信號，`try ... except` 用於回應它
泛型（**Generics**）	還沒有！（*https://oreil.ly/ORveC*）	由於使用動態型別，不需要	由於使用動態型別，不需要
測試支援	語言的一部分（也就是 `testing` 套件和 `go test` 命令）；有可用的程式庫（例如，**stretchr/testify**）	不是語言的一部分，有許多程式庫可用（例如 **Jasmine**、**Mocha**、**Jest**）	語言的一部分（也就是 `unittest` 程式庫）；有可用的程式庫（例如，**PyTest**）

2. 人氣

根據 Stack Overflow 2017 年（*https://oreil.ly/CbnCx*）、2018 年（*https://oreil.ly/uhhLx*）、2019 年（*https://oreil.ly/BdAQJ*）和 2020 年（*https://oreil.ly/mHqNs*）的多項年度調查，Python、JavaScript 和 Go 是開發人員最想學習的三大新語言。圖 P-4 顯示了 2020 年調查的結果。

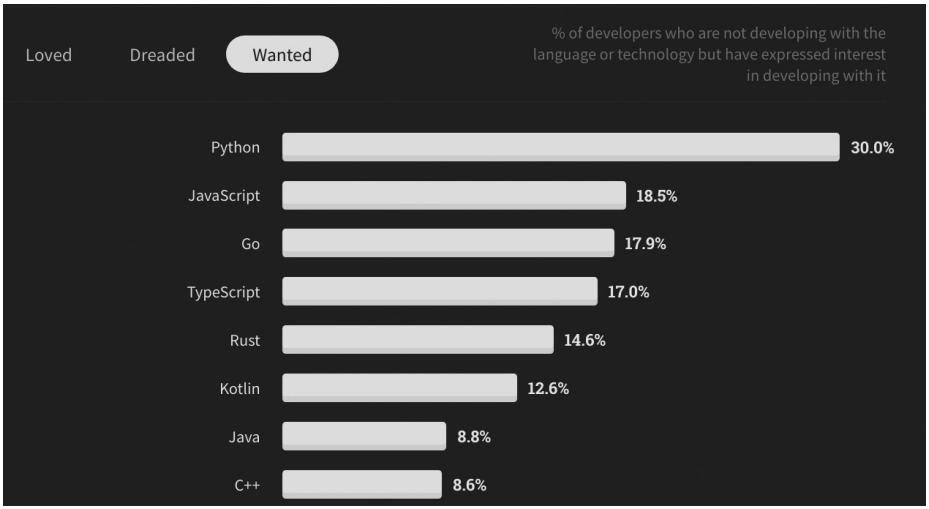


圖 P-4　Stack Overflow 對開發人員的一項調查指出的最需要學習的新語言

在 2021 年 Stack Overflow 的調查（*https://oreil.ly/hzMVk*）中，TypeScript 攀升至第二位，將 JavaScript 和 Go 分別擠到第三和第四位。 Python 保住了第一把交椅。

從語法上講，TypeScript 是 JavaScript 的嚴格超集合（*https://oreil.ly/aATAD*）。因此，我們可以說每個想要學習 TypeScript 的開發人員都必須要瞭解 JavaScript。我希望 TypeScript 開發人員也會發現這本書會很有價值。

3. 個人原因

在過去五年左右的時間裡，我有機會參與了幾個專案，其中的技術堆疊會將這三種語言其中的一種作為主要程式語言。在與其他開發人員一起工作時，我發現一般來說，他們對學習和實踐 TDD 的渴望，與他們無法找到資源（或培養紀律）來做到這一點相提並論。他們想練習 TDD，但不知道如何或找不到時間進行。很明顯的，這種情況對於經驗豐富的開發人員和“新手”一樣適用。

我希望這本書對於那些想用任何語言來學習和實踐 TDD（不僅僅是 Go、JavaScript 或 Python）的人來說，既可以作為實用指南，也可以作為靈感來源。

為什麼不是其他的語言？

對於初學者來說，有大量的程式語言可用。可以想像，或許有一個人可以寫出六本這樣的書，但這樣仍然只涵蓋了世界各地開發人員每天用於為學術、商業和娛樂目的編寫程式碼的語言中的一小部分。[3]

此外，已經有一本優秀的書籍可用在 Java 中的測試驅動開發。Kent Beck 的開創性工作啟發了我，就像無數其他開發人員一樣，我愛上了 TDD 的藝術和科學。這也引發了本書的“金錢問題”這個主要主題。

我確信有許多其他語言可以提供實用的 TDD 指南。R 呢？還是 SQL ？甚至 COBOL ？

讓我向您保證：提到 COBOL 並不是稻草人的論證（straw man argument），也不是陰險毒辣的誹謗。在 2000 年代中期，我參與了一個專案，在該專案中我示範了使用 COBOLUnit 在 COBOL 中執行 TDD 的能力。這是我用比我大上十多歲的語言所獲得的最有趣的東西！

我希望您能繼承這個衣缽。您會學習、教導和擁護用其他語言來實踐測試驅動開發所需的技能和紀律。您會撰寫部落格、開源專案或本系列的下一本書。

為什麼這本書有“第 0 章”？

絕大多數程式語言對陣列和其他可數序列會使用從 0 開始的索引。[4] 這對於構成本書基礎的三種程式語言來說當然也是正確的。從某種意義上說，本書用從 0 開始的章節編號來紀念程式設計文化的豐富歷史。

我也想向零本身致敬，它是一個激進的想法。Charles Seife 就這個孤獨的數字寫了一整本書。在追溯零的歷史時，Seife 注意到希臘人對一個不代表什麼的數字持保留態度：

[3] 雖然有一本關於某種語言的 TDD 的書不太可能得到出版商的認可。它的書名以“大腦”開頭，並以髒話結尾！

[4] Lua 是一個明顯的例外。我的朋友 Kent Spillner 曾經就這個主題發表過一次引人入勝的演講，我在此進行了總結（*https://oreil.ly/E9M41*）。

> 在那個 [也就是希臘] 宇宙中，沒有什麼是虛無的。沒有零。正因為如此，西方近兩千年來無法接受零。後果是可怕的。零的缺席會阻礙數學的發展，扼殺科學的創新，順便說一句，還會把日曆弄得一團糟。在他們接受零之前，西方的哲學家必須摧毀他們的宇宙。
>
> —— Charles Seife, *Zero: The Biography of a Dangerous Idea*

以下這段話冒著過於崇高的風險：測試驅動開發在今天的程式設計文化中佔據了相似的位置，就像幾千年前在西方哲學中的零一樣。採用它時會遇到阻力，這是由於不屑一顧、不安以及對一無所有的過度大驚小怪的奇怪組合而產生的。“我為什麼會對先編寫測試這件事挑剔呢——因為我已經知道我會如何編寫這個功能了！”、“測試驅動開發是迂腐的：它只在理論上有效，而不是在實務上。”、“在編寫完生產程式碼後再編寫測試至少會和先編寫測試一樣的有效，甚至還會更有效。”這些還有其他對 TDD 的反對意見致使它在激進程度上類似於數字零！

無論如何，一本書有第 0 章的作法並不完全是激進的。Carol Schumacher 寫了一整本書，標題為第零章：抽象數學的基本概念（*Chapter Zero: Fundamental Notions of Abstract Mathematics*）（*https://oreil.ly/nXJdV*），這是許多大學課程中高等數學的標準教科書。猜到那一本書是從哪一章開始的並不會有獎品可拿！

Schumacher 博士在她那本書的教師手冊中，說了一些我覺得很有啟發性的話：

> 作為作家，您的任務是給您的讀者正確的暗示，讓他們盡可能容易的理解您想說的話。
>
> —— Carol Schumacher，與第零章一起使用的教師資源手冊

我把這個建議牢記在心。務實的說，包含 “0” 的標題有助於將第 0 章與其後面的內文區分開來。本書的第 1 章將我們帶入了一個 TDD 之旅，並在接下來的十幾章中繼續進行。第 0 章是用來描述那段旅程會是什麼、在我們開始之前我們需要知道和擁有什麼、以及我們在此過程中會發生什麼。

解釋完之後，讓我們直接進入第 0 章吧！

第 0 章

簡介與設定

乾淨俐落的程式碼對成功至關重要。

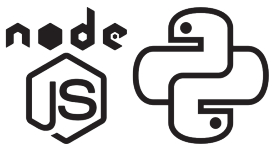
— Ron Jeffries，“Clean Code: A Learning”，2017 年 8 月 23 日，*ronjeffries.com*

在我們開始進入測試驅動開發的費時耗力但報酬豐富的世界之前，我們需要確保我們有一個可以工作的開發環境。本章全是關於準備和設定的內容。

設定您的開發環境

無論您遵循哪種閱讀路徑（參見圖 P-2），您都需要一個乾淨的開發環境來閱讀本書。本書的其餘部分假定您已按照本節所述來設定了開發環境。

無論您從 Go、JavaScript 或 Python 中的哪一個開始，都應按照本節中的說明來設定您的開發環境。



通用設定

資料夾結構

請建立一個資料夾，作為我們將在本書中所編寫的所有原始碼的根目錄。將它命名為從現在起幾個星期後對您來說還是清晰明確的名稱，例如 *tdd-project*。

請在此資料夾下，建立一組資料夾，如下所示：

```
tdd-project
├── go
├── js
└── py
```

請在編寫第一行程式碼之前建立所有的資料夾，即使您打算多次學習本書且一次只使用一種語言也一樣。建立此資料夾結構具有以下好處：

1. 它將三種語言的程式碼分開但又彼此靠近。
2. 確保本書中的**大多數**命令無需更改即可執行。
   - 要處理那些會完全限定檔案 / 資料夾名稱的命令是例外——這樣的命令很少見。其中之一會出現在本節中。
3. 它允許跨越三種語言來輕鬆的採用進階功能，例如持續整合。
4. 這會與隨書所附的程式碼庫（*https://github.com/saleem/tdd-book-code*）中的資料夾結構相匹配。隨著程式碼的發展，這對於比較和對比您的程式碼很有用。

在本書的其餘部分，*TDD 專案根資料夾*（應用格式）這個術語，是用來參照包含了所有原始碼的根資料夾——在上文的名稱為 `tdd-project`。名為 `go`、`js` 和 `py` 的資料夾所參照的就是它們的名稱所示——從語境中可以清楚的看出它們的含意。

*TDD 專案根資料夾*是用來參照包含了本書中所開發的所有原始碼的資料夾的名稱。它是名為 `go`、`js` 和 `py` 的三個資料夾的父層級。

宣告一個名為 `TDD_PROJECT_ROOT` 的環境變數，並將其值設定為 *TDD 專案根資料夾*的完全合格名稱。在每個殼層（shell）中執行一次（或者更好的是，在您的殼層初始化腳本（例如 `.bashrc` 檔案）中執行一次），以確保所有後續命令都能無縫運行。

```
export TDD_PROJECT_ROOT=/完全/合格/路徑/到/tdd-project
```

例如，在我的 macOS 系統上，`TDD_PROJECT_ROOT` 的完全合格路徑是 `/Users/saleemsiddiqui/code/github/saleem/tdd-project`。

消除無聊

您需要在您啟動的每個新殼層中定義環境變數 `TDD_PROJECT_ROOT`。如果您覺得這很麻煩，您可以在相對應的配置檔案中設定一次即可。如何設定它的方式因作業系統以及殼層而異。對於我們將在本書中使用的類似 Bash 的殼層，可以在配置檔案中定義環境變數；雖然在細節上仍然不同。例如，在大多數 Linux（和 macOS）上，您可以在主資料夾中名為 `.bashrc` 的檔案裡添加 `export TDD_PROJECT_ROOT=...` 敘述（*https://oreil.ly/SMmFc*）。如果您在 Windows 上使用 Git BASH（如本章後面所述），您可能需要改用 `.bash_profile` 檔案（*https://oreil.ly/pkgxg*）。

簡而言之：消除工作中的無聊是一件好事。請使用適當的機制以在本書所使用的所有殼層中，可靠且一致的定義環境變數。

文本編輯器或 IDE

我們需要一個文本編輯器來編輯原始檔。整合開發環境（*integrated development environment*, IDE）提供了一個工具來幫助我們編輯、編譯和測試多種語言的程式碼。然而，這是一個攸關選擇和個人喜好的問題；請選擇最適合您的那一個。

附錄 A 中更詳細的描述了 IDE。

殼層

我們需要一個殼層（一個命令行直譯器（command-line interpreter））來執行我們的測試、檢查輸出並執行其他任務。與 IDE 一樣，殼層的選擇也很多，而且通常是開發人員之間熱烈分享想法的主題。本書在需要輸入的命令時假設使用了一個類似 *Bash* 的殼層。在大多數（如果不是全部）的類 Unix 作業系統（以及 macOS）上，Bash 殼層是現成可用的。

在 Windows 中可以使用 Git BASH（*https://gitforwindows.org*）等殼層。在 Windows 10 上，Windows Subsystem for Linux（*https://oreil.ly/UZ0KU*）提供了對 Bash 殼層的原生支援，以及許多其他的 “Linux 好東西 ”。這些任何一個選項或其他類似的選項都足以（並且必須）遵循本書中的程式碼範例。

圖 0-1 顯示了一個類似 Bash 的殼層，其中也包含輸入命令的結果。

```
tdd-project> python3
Python 3.9.6 (default, Jun 29 2021, 05:25:02)
[Clang 12.0.5 (clang-1205.0.22.9)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

圖 0-1　遵循本書中的編寫程式碼範例需要一個類似 Bash 的殼層，就像這裡所顯示的那樣

Git

第 13 章會介紹如何使用 GitHub Actions 來進行持續整合（CI）的實務。要遵循該章的內容，我們需要建立自己的 GitHub 專案並將程式碼推送到該專案中。

Git 是一個開源的分散式版本控制系統。GitHub 是一個協作式網際網路託管平台，允許人們相互保存和共享專案的原始碼。

Git（*https://git-scm.com*）是一個免費的、開源的、分散式的版本控制系統。GitHub（*https://www.github.com*）是一個使用 Git 的程式碼共享平台。

為了確保我們能夠採用持續整合，我們現在要做一些準備，並將一些工作延遲到第 13 章。更具體的說，我們將會在我們的開發環境中設定 Git 版本控制系統。我們會把 GitHub 專案的建立延遲到第 13 章。

首先，下載安裝 Git 版本控制系統（*https://git-scm.com/downloads*）。它適用於 macOS、Windows 和 Linux/Unix。安裝後，透過在終端機視窗上輸入 `git --version`，並按 Enter 來驗證它是否可以運作。您應該會看到已安裝 Git 版本的回應，如圖 0-2 所示。

```
tdd-project> git --version
git version 2.32.0
tdd-project> _
```

圖 0-2　透過鍵入 `git --version` 並在殼層上按 Enter 來驗證 Git 是否已安裝

接下來，我們將在 TDD_PROJECT_ROOT 中建立一個新的 Git 專案。請在殼層視窗中，鍵入以下命令：

```
cd $TDD_PROJECT_ROOT
git init .
```

這應該會產生一個 `Initialized empty Git repository in /您的/完全/合格/專案/路徑/.git/` 的輸出。這會在我們的 TDD_PROJECT_ROOT 中建立一個閃閃發亮的新的（目前為空的）Git 儲存庫。現在我們在 TDD-PROJECT-ROOT 資料夾下應該包含這些資料夾：

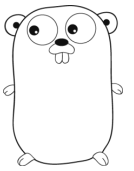
```
tdd-project
├── .git
├── go
├── js
└── py
```

Git 使用 `.git` 資料夾進行簿記（bookkeeping）。我們無需對其內容進行任何更改。

當我們在接下來的章節中編寫原始碼時，我們會定期將我們的變更提交到這個 Git 儲存庫。我們將使用 Git CLI（command line interface，命令行介面）來執行此操作。

在本書的其餘部分，我們會經常將我們的程式碼變更提交到 Git 儲存庫中。為了突顯這一點，我們將使用 git Git 圖示。

Go



閱讀本書時我們需要安裝 Go 版本 1.17。它適用於不同的作業系統的版本可在此下載（*https://golang.org/dl*）。

要驗證 Go 是否已正確安裝，請在殼層上鍵入 `go version` 並按 Enter。這應該會印出您的 Go 安裝的版本號碼。請參見圖 0-3。

```
tdd-project> go version
go version go1.17 darwin/amd64
tdd-project> _
```

圖 0-3　透過鍵入 `go version` 並在殼層上按 Enter 來驗證 Go 是否正常運作

我們還需要設定幾個 Go 特定的環境變數：

1. GO111MODULE 環境變數應設定為 on。
2. GOPATH 環境變數不應該包含 TDD_PROJECT_ROOT 或其下的任何資料夾，例如 go 資料夾。

請在殼層中執行這兩行程式碼：

```
export GO111MODULE="on"
export GOPATH=""
```

我們需要建立一個骨幹的 go.mod 檔案來準備編寫程式碼。這些是執行此操作的命令：

```
cd $TDD_PROJECT_ROOT/go
go mod init tdd
```

這將建立一個名為 go.mod 的檔案，其內容應為：

```
module tdd

go 1.17
```

對於從現在開始的所有 Go 程式的開發，請確保殼層正位於 TDD_PROJECT_ROOT 下的 go 資料夾中。

對於本書中的 Go 程式碼，請確保在執行任何 Go 命令之前先輸入 cd $TDD_PROJECT_ROOT/go。

快速瞭解 Go 套件管理

Go 的套件管理正處於翻天覆地的變化之中。使用 GOPATH 環境變數的舊樣式正在逐步的被淘汰，取而代之的是使用 go.mod 檔案的新樣式。這兩種風格在許多情況下是互不相容的。

我們上面所定義的兩個環境變數，以及我們產生的骨幹 go.mod 檔案，確保了 Go 工具可以正確的運作在我們的原始碼上，尤其是在我們建立套件時。我們將在第 5 章建立 Go 套件。

JavaScript

我們需要 Node.js v14（“Fermium”）或 v16 來遵循本書。適用於不同的作業系統的這兩個版本的 JavaScript 都可以從 Node.js 網站（*https://nodejs.org/en/download*）取得。

要驗證 Node.js 是否已經正確安裝，請在殼層上鍵入 `node -v` 並按 Enter。該命令應會印出一行訊息，列出 Node.js 的版本。請參見圖 0-4。

```
tdd-project> node -v
v16.6.2
tdd-project> _
```

圖 0-4　透過鍵入 `node -v` 並在殼層上按 Enter 來驗證 Node.js 是否正常運作

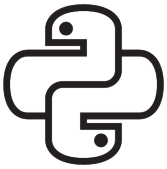
快速瞭解測試程式庫

Node.js 生態系統中有幾個單元測試框架。總體而言，它們非常適合編寫測試和進行 TDD。然而，這本書避開了*所有的*這些框架。它的程式碼使用了 `assert` NPM 套件來進行斷言（assertion），並使用一個簡單的類別來組織測試。簡單性是要讓我們專注於 TDD 的*實務和語意*，而不是任何一個程式庫的*語法*。第 6 章更詳細的描述了測試的組織。附錄 B 列舉了測試框架以及我們不使用其中任何一個的詳細原因。

另一個快速瞭解，關於 JavaScript 套件管理

與測試框架類似，JavaScript 有很多方法來定義套件和依賴項。本書採用了 CommonJS 風格。第 6 章討論了其他風格：我們用原始碼來詳細展示 ES6 和 UMD 風格，而 AMD 風格則不使用原始碼來進行簡要的展示。

Python



我們需要 Python 3.10 來學習本書，適用於不同的作業系統的版本可從 Python 網站（*https://oreil.ly/xNLPa*）中取得。

Python 語言在 “Python 2” 和 “Python 3” 之間經歷了重大變化。雖然 Python 3 的舊版本（例如 3.6）可能還可以運作，但 Python 2 的任何版本都不足以滿足本書的學習目的。

您的電腦上可能已經安裝了 Python 2。例如，許多 macOS 作業系統（包括 Big Sur）都與 Python 2 捆綁在一起。並沒有必要（或不推薦）卸載 Python 2 來遵循本書；但是，有必要確保 Python 3 是我們所使用的版本。

為了避免歧義，本書明確的使用 `python3` 作為命令中可執行檔的名稱。將 `python` 命令 “別名（alias）” 為參照 Python 3 是有可能的（儘管也是沒有必要的）。

這裡有一個簡單的方法來找出您需要輸入哪個命令才可以確保使用了 Python 3。在殼層上鍵入 `python --version` 並按 Enter。如果您獲得一些從 `Python 3` 開始的輸出訊息，那麼就沒錯。如果您得到以 `Python 2` 開頭的東西，那您可能需要為本書中的所有命令外顯式的輸入 `python3`。

圖 0-5 顯示了一個同時使用了 Python 2 和 Python 3 的開發環境。

```
tdd-project> python3 --version
Python 3.9.6
tdd-project> python --version
Python 2.7.16
tdd-project> _
```

圖 0-5　驗證 Python 3 是否已安裝以及您需要輸入才能使用它的命令（如此處所示的 `python3`）

使用 Python 3 來遵循本書中的程式碼。不要使用 Python 2 ——它無法運作。

圖 0-6 顯示了一個助記符（mnemonic），用於簡化前面的 Python 版本的冗長廢話！

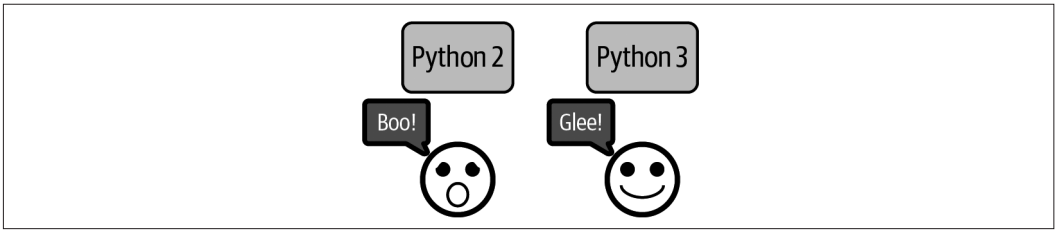


圖 0-6　簡單的助記符，釐清本書需要哪個版本的 Python ！

譯註：Boo!（喝倒彩的聲音）發音類似於 two；Glee!（高興）發音類似於 three。

我們在哪裡

在這個預備章節中，我們熟悉了要開始以測試驅動方式來編寫程式碼所需的工具鏈。我們還學習了如何準備我們的開發環境並驗證它是否處於運作狀態。

現在我們知道了這本書是有關什麼、內容是什麼、其閱讀方式、以及最重要的如何設定我們的工作環境來遵循它之後，我們已經準備好解決我們的問題了，一次發掘一個功能，並藉由測試來進行驅動。我們將在第 1 章開始這一旅程。讓我們開始吧！