第一章

大致接近正確的軟體

譯註

倘若讀者曾經搭乘飛機，則表示您已經採取世界上其中一種最安全的旅行方式。根據統計，遭逢空難身故的可能性為 2,940 萬分之一（*http://www.statisticbrain.com/airplane-crash-statistics/*），這意味著讀者可以選擇成為一名航空公司的飛行員，並在 40 年的飛航職業生涯中，能夠避免飛機失事而安然無恙的度過。飛機是相當複雜的結構體，會導致過去這些意外著實令人吃驚，然而整體來說並非總是如此。

2014 年是航空災難年；與航空相關的死亡人數共計有 824 人（*http://bit.ly/2014-aviation*），其中包括馬來西亞航空飛機失蹤一案。而在 1929 年的空難傷亡則有 257 人（*http://bit.ly/casualties-1929*）。表面上看起來在飛航安全方面越來越糟，但仔細分析會發現，單以美國而言，每年有超過 1000 萬次的航班起降，而 1929 年的航班數量則比較少——大約 5 萬到 10 萬班次起降。這意味著從 1929 年到 2014 年，一架飛機遇難的總體可能性大幅降低，從 0.25％下降到 0.00824％。

飛航歷經多年的變革，軟體開發也該如此。以軟體業而言，儘管 1929 年當時軟體開發工作尚未存在，然而在往後的 85 年期間，開發人員建立許多的軟體專案，也搞砸其中不少的軟體專案。

最近搞砸的軟體案例像是：health.gov 的啟動事件，這是一場財政災難，耗費的成本約 6.34 億美元（*http://bit.ly/cost-healthcaregov*）。更糟糕的例子是含有重大災難性漏洞的軟體專案，譬如納斯達克證券交易所（NASDAQ）由於軟體故障而被迫交易停擺，並因此處以一千萬美元的罰款（*http://reut.rs/2i2HfgS*）。另外在 2014 年，心血漏洞（Heartbleed bug）的影響，造成許多使用 SSL 的網站受害，因此導致 CloudFlare 註銷超過 100,000 個 SSL 憑證，以及耗費數百萬美元的相關支出（*http://bit.ly/cost-heartbleed*）。

譯註：本章標題為大致接近正確的軟體（Probably Approximately Correct Software）又稱為 PAC 軟體。

軟體和飛機的一個共通點是：兩者皆為複雜的結構體，當功能失效時，會造成公然災難性的影響。隨著時間的進展，航空公司能夠確保飛行安全，降低 96%以上飛航災難發生的可能性。然而，對於越來越複雜的軟體而言，卻不能像飛航安全有相同的結果。實際上，軟體的災難性錯誤經常發生，也因此浪費數十億美元。

為何飛航變得如此安全，而軟體卻依然錯誤百出呢？

將軟體撰寫正確

從 1929 年到 2014 年期間，對於結構越來越複雜、飛行越來越快速與規模越來越大型的飛機來說，伴隨著 FAA 與國際機構的監管項目也越來越多，而飛行員之間也造就檢查清單的工作文化。

同一期間，儘管電腦硬體技術迅速提升，但是執行於其中的軟體卻沒有跟上腳步。開發人員主要還是採用程序性程式或物件導向程式，而這些程式並沒有充分利用平行運算的硬體資源。不過程式設計師對於撰寫軟體與建立測試文化兩方面的指引，則獲得十足的進展，因而推演出 SOLID 與 TDD 的運用。SOLID 是一組能夠寫出良好程式碼的引導原則，TDD 則是測試驅動設計（test-driven design）或測試驅動開發（test-driven development）模型。接下來會探討這兩個與撰寫正確軟體有關的心智模型，然後會討論以軟體為中心的重構（refactoring）議題。

SOLID

SOLID 是一個協助設計良好物件導向程式的軟體框架。與 FAA 明定航空公司或飛機應該做什麼的方式雷同，SOLID 描述應該如何建構軟體。飛航偶爾會違反 FAA 規定，其中可能造成嚴重或輕微程度的飛安影響；SOLID 也是如此，這些原則有時會產生巨大的差異，但通常只是指引的效果。Robert Martin 提出 SOLID 的內容有「五項原則」，主要目的是寫出可維護、可理解與穩定的優良程式碼。而 Michael Feathers 將其中每項原則的開頭第一個英文字母組合成 *SOLID* 以方便記住這些原則。

SOLID 的五項原則是：

- 單一職責原則（SRP — Single Responsibility Principle）
- 開放 / 封閉原則（OCP — Open/Closed Principle）
- 里氏替換原則（LSP — Liskov Substitution Principle）
- 介面隔離原則（ISP — Interface Segregation Principle）
- 相依性反轉原則（DIP — Dependency Inversion Principle）

單一職責原則

SRP 已經成為寫出良好物件導向程式碼最普遍的原則之一。原因是單一職責用於定義簡單的類別或物件。同樣的思維可以應用於具有單一功能的函數式程式設計（functional programming）。然而在此的想法相當簡單：一份軟體只做一件事。違反 SRP 的貼切例子是多功能工具（如圖 1-1 所示）。一個多功能工具能夠實現多項功能，然而每項功能只在必要時才能單獨運用。

圖 1-1　具有過多職責的多功能工具

開放 / 封閉原則

OCP 有時也稱為封裝（encapsulation），此原則是物件的功能得以擴展但不能修改。例如與內部計數相關的計數器物件，其中具有 `increment` 和 `decrement` 方法。除非是透過已定義的 API，否則此物件不應該讓任何人直接變更內部計數值，然而可對此物件進行擴展（譬如：利用 `Notifier` 物件進行計數值變更的通知）。

里氏替換原則

LSP 表示任何子類型應該能夠輕易的從某個物件樹（object tree）之下進行替換，而不會產生副作用。例如：一台模型車可以取代一輛真實的汽車。

介面隔離原則

ISP 是「具有多個用戶端特定介面會優於所有用戶端只有一個通用介面」原則。這個原則用於簡化實體（entities）之間的資料交換作業。一個貼切的例子是將垃圾區分出一般垃圾、廚餘與資源垃圾等三類，而非將這三種內容混雜成為一大包垃圾。

相依性反轉原則

DIP 是「強調依據抽象面而非仰賴具體項」原則，其中應該建立物件階層或物件繼承樹（inheritance tree）。在 Robert Martin 的 DIP 原始論文中[1]列舉的例子是：KeyboardReader 繼承自一般的 Reader 物件而非實作類別中的所有內容。這也符合 Arthur Riel 所著的《*Object Oriented Design Heuristics*》一書中關於避免 *god classes*（神類別）的說法。雖然可以將吉他與放大器直接以導線焊接，但很有可能效率不高，而且聽起來的效果不會太好。

SOLID 框架歷經時間的考驗，並呈現於 Martin 與 Feathers 撰寫的許多書籍當中，也納入 Sandi Metz 所著的《***Practical Object-Oriented Design in Ruby***》一書中（***http://poodr.info/***）。這個框架是個指引，然而也含有簡單事物的提醒，使得開發人員能夠盡力寫出最佳的程式碼。這些指引有助於寫出正確的結構化軟體。

測試與 TDD

早期航太業的飛行員沒有使用檢查清單測試飛機是否能夠起飛。在 Tom Wolfe 的《*The Right Stuff*（太空英雄）》書中，大多數像 Chuck Yeager 這樣早期的試飛員會以自身的感受與能力駕馭複雜的飛行器。如此的作為也造成當中四分之一的試飛員在飛行時因而喪生[2]。

如今的情勢大不相同，飛機在起飛之前，飛行員會進行一系列的檢查作業。其中有些檢查項目似乎頗為費工，譬如與其他機組人員逐一打招呼以確認彼此。然而想像一個情境：倘若發現自己處在某個慌亂情況中，而需要將某個問題立即通知某人，此時不曉得人員的名字，那就很難進行溝通。

好的軟體也是如此，需要進行一系列的系統檢查與定期執行，測試軟體的作業是否正常，以讓軟體維持一致性的運作。

早期的軟體業，大多數的測試工作是在原版軟體撰寫完成之後進行（請參閱瀑布式模型（*https://en.wikipedia.org/wiki/Waterfall_model*），這是 NASA 和其他組織曾經針對軟體設計與上線測試所採用的軟體模型）。這個做法與當時的工程管理風格能夠達到良好的結合。與飛航持續的建置過程類似，開發人員習慣先進行軟體設計，根據規格內容進行

1 Robert Martin, “The Dependency Inversion Principle,” http://bit.ly/the-DIP.

2 Atul Gawande, The Checklist Manifesto (New York: Metropolitan Books), p. 161.

程式碼撰寫，然後在交付給客戶之前才進行軟體的測試。由於技術的保質期限短，而這種測試方式可能需要數月甚至數年的週期，時常發生緩不濟急的情況。也因此誕生敏捷軟體開發宣言（Agile Manifesto —— *http://agilemanifesto.org/*）以及 Kent Beck、Ward Cunningham 等人所倡導的測試與 TDD（測試驅動開發）相關軟體開發文化。

測試驅動開發的想法很簡單：撰寫一個測試項，記錄要實現的內容；首先進行測試以確定此測試項運作失敗，接著撰寫程式碼修正此測試項，並在通過測試之後，調整相關程式碼以符合 SOLID 原則。雖然許多人認為這種做法會增加軟體開發的時間，但如此可以大幅降低程式碼的錯誤與缺陷，並提高正式上線作業的穩定性[3]。

失誤容忍度低的飛機大多以相同的方式進行操作。飛行員在準備駕駛波音 787 客機之前，會在飛行模擬器中花費數小時的試驗飛行，了解並測試此類型飛機的性能與情境。在飛機起飛前，飛行員會進行飛航測試，而在飛行期間，也會再次進行飛航測試。現代的軟體開發也採用非常相似的方式。在軟體部署前後，軟體人員會撰寫測試項測試相關性能（部署後會透過監督方式進行）。

然而此種做法會造成一個問題，實際的情況是：並非一切皆保持不變，測試項的撰寫並不能代表必定保有良好的程式碼。在 David Heinemer Hanson 發表與測試驅動損害相關病毒呈現的文章中（*http://bit.ly/test-induced-damage*），針對盲目依循 TDD 與 SOLD 原則而產出複雜程式碼的結果，已經提出一些不錯的觀點；其中大部分的觀點與非必要的複雜性有關，這些複雜性的產生是因為將軟體每段程式碼取出放入不同的類別中，或寫出可測試卻難以閱讀的程式碼。不過筆者認為這是將軟體撰寫正確而需要關注的最後一個因素：重構。

重構

重構是最難向非程式設計師解釋的程式設計實務之一，通常外人難以窺視重構物表面底下的內容。當乘客搭乘飛機時，只能看到飛行器整體的 20% 內容。機體的鋁與鈦金屬板底下有乘客看不到的複雜電氣系統，這個系統包含供應緊急照明使用的電力，以防在飛行過程中發生的各種故障狀況而失去電能，另外也包含設計輕巧堅固的管線與桁架——有不少機體單元陳列在內。廣泛而言，要解釋飛機含有的內容，就如同向某人說明某個華麗水龍頭下方水槽底下所有的管線，難以言喻。

3 Nachiappan Nagappan et al., “Realizing Quality Improvement through Test Driven Development: Results and Experience of Four Industrial Teams,” Empirical Software Engineering 13, no. 3 (2008): 289–302, http://bit.ly/Nagappanetal.

重構會將現有的結構調整到更好的狀態。就像拿起一個凌亂的斷路器（circuit breaker），對它進行整理，而當再次使用時，就會確切知道其中的運作內容。飛機屬於剛性的設計，但軟體不是，軟體的內容變化快速。許多企業會持續不斷的將軟體部署到正式上線環境（production environment）中。所有的功能開發有時會累積某些程度的技術債（technical debt）。

技術債，也稱為**設計負債**（*design debt*）或**程式碼負債**（*code debt*），是軟體專案隨著時間衍生出系統設計不良的一種隱喻。技術債造成的衰弱問題是：其所積累的利息代價終將阻礙未來的功能開發。

如果在某個專案著墨的時間夠長，會感受到只有起初能夠迅速釋出軟體版本，而越接近專案時程尾聲越會發生停滯不前的窘境。在許多情況下，通常是因為缺少軟體測試項或沒有依循 SOLID 原則才會產生技術債。

技術債的存在並不是一件壞事——有時專案需要早點推出，以便擴展相關的業務；但倘若不償還過程累積的負債，最終會債台高築而將專案搞砸。然而可以針對具有負債的程式碼進行重構作業而克服上述的議題。

重構作業能夠讓程式碼更貼合 SOLID 指引與 TDD 程式碼庫（codebase），會對現有的程式碼進行清理工作，並可以讓新加入的開發人員輕鬆進行開發工作。對於現有程式碼的處理步驟如下：

1. 依循 SOLID 指引
   a. 單一職責原則
   b. 開放 / 封閉原則
   c. 里氏替換原則
   d. 介面隔離原則
   e. 相依性反轉原則
2. 實作 TDD（測試驅動開發 / 設計）
3. 重構現有的程式碼以避免技術債的累積

此刻，實際的問題是：怎樣把軟體內容弄得正確？

撰寫正確的軟體

撰寫正確的軟體比將軟體撰寫正確要困難得多。在《*Specification by Example*》這本書中，作者 Gojko Adzi 認定撰寫軟體的最佳做法是：先制定規格然後直接與使用者合作。只有在規格完成之後，才能撰寫符合該規格的程式碼。然而，這種做法會有問題——有時世界不是我們所認定的樣子。最初認為可行的模式時常到最後都會變得不可行。

例如，美國生鮮電商 Webvan 在建立線上生鮮雜貨銷售業務所面臨的慘敗狀況，其中投入資金額將近 4 億美元，並迅速完成基礎設施的建構以便配合公司認定會蓬勃發展的業務推行。然而，因為食品宅配的成本以及對線上生鮮雜貨零售市場的高估，而使得公司被迫關門。藉由多項措施，Webvan 在軟體撰寫與業務拓展方面取得成功，然而市場還來不及接納這家企業的新興業務，公司就迅速宣告破產。而這家企業當時建構的許多基礎設施，如今卻被 Amazon.com 用於 AmazonFresh 之中。

> In theory, theory and practice are the same. In practice they are not.
>
> —Albert Einstein
>
> 理論上，理論與實際會一樣。實際上，兩者卻不一樣。
>
> ——愛因斯坦

目前處於理論上可以將軟體撰寫正確並且讓軟體正常運作的位置點，然而要撰寫正確的軟體是個定位非常模糊的議題，而這就是真正需要引入機器學習的所在之處。

用機器學習撰寫正確的軟體

在《*The Knowledge-Creating Company*（創新求勝——智價企業論）》一書中，作者 Nonaka 與 Takeuchi 概述日本企業在二十世紀八〇年代相當成功的原因，主要不是使用由上而下的做法解決問題，而是隨著時間不斷的學習，其中麵包機揉麵團的過程是個迭代作業的貼切範例，可輕易套用於軟體開發中。

接著要仔細探討機器學習的內容。

究竟何謂機器學習？

根據大多數的定義，機器學習是讓機器透過資料——即以數值格式（矩陣、向量等）表示的內容——進行學習的交易策略之相關演算法、技術與技巧的集合。

為了更容易理解機器學習的意義，在此需要說明其誕生過程。在二十世紀五〇年代，人們對於西洋跳棋遊戲有廣泛的研究，許多模型都聚焦於將此遊戲玩得更好而提出最佳遊玩策略。如今讀者也許可以提出一個相當簡單的西洋跳棋遊玩程式，僅由一盤獲勝的結局倒推下棋的過程，而映射出相關的決策樹（decision tree），並將獲勝的過程進行優化。

不過，這是一個非常狹隘與演繹的推理方式，實際上代理程式（agent）必須納入軟體中。然而，早期的程式大多數並沒有納入情境或非理性行為。

大約經過 30 年之後，機器學習逐漸興起，開始有許多類似的思維用於處理垃圾郵件過濾、分類與一般資料分析的問題。

在此的重要轉變是從電腦化演繹到電腦化歸納的移轉。就像福爾摩斯一樣，演繹會使用複雜的邏輯模型得出結論。相較之下，歸納會認真看待資料，並嘗試將某個模型與資料進行配適（fit）。這個轉變對於尋找一般問題的夠好解法方面已經取得非常不錯的進展。

然而，歸納式推理所遭遇的議題是：只能提供我們所知的演算法資料，要將某些事物進行量化處理是非常困難的工作。例如，如何將圖中某隻小貓的可愛成分進行量化作業呢？

在過去十年中，持續目睹深度學習領域的復興內容，也因此從中找到緩解此議題的方式。不須依賴由人類進行編碼的資料，而是採用像自動編碼器（autoencoders）這樣的演算法就可以找到之前無法量化的資料節點。

上述的結果聽起來都很棒，但是採用這項功能，伴隨而來的是相當高的成本負擔。

機器學習的高利貸技術債

最近 Google 公布一篇名為〈Machine Learning: The High Interest Credit Card of Technical Debt〉的論文（*http://research.google.com/pubs/pub43146.html*），作者 Sculley 等人表明機器學習專案遭遇比文中論述還多的技術債議題（如表 1-1 所示）。

文中指出機器學習專案本質上是複雜且邊界模糊，重度仰賴資料相依性內容，遭受系統級的麵條式程式碼（spaghetti code）影響，以及可能因為外界的變化而徹底改變。作者的論點是這些問題頗與機器學習專案有關，而且大多數的情況下皆是如此。

在此並不會逐一探討這些議題，而主要的安排是回溯關聯本章針對 SOLID、TDD 以及重構作業的初始論述內容，並且明瞭這些議題與機器學習程式的相關性。

表 1-1　機器學習的高利貸技術債

機器學習問題	現象表露	違反原則
糾結（Entanglement）	變更一個因子就改變一切	SRP
隱藏回饋循環（Hidden feedback loops）	在模型中具有內建隱藏特徵	OCP
未宣告的使用者 / 可見性負債（Undeclared consumers/ visibility debt）		ISP
不穩定資料相依性（Unstable data dependencies）	消逝性資料	ISP
未充分利用的資料相依性（Underutilized data dependencies）	未使用的維度	LSP
Correction cascade		*
黏合程式碼（Glue code）	撰寫能處理所有工作的程式碼	SRP
管道叢林（Pipeline jungles）	透過複雜的工作流程發送資料	DIP
實驗路徑（Experimental paths）	無路可走的死路徑	DIP
組態負債（Configuration debt）	對於新資料使用舊組態	*
動態情境的固定臨界值（Fixed thresholds in a dynamic world）	對相關性的變化不具彈性	*
相關性變化（Correlations change）	超越因果關係的相關性建模	機器學習特定

將 SOLID 應用於機器學習

如之前所述，SOLID 只是一份指引原則，提醒開發人員在撰寫物件導向程式碼時可以依循的一些目標。許多機器學習演算法本質上不屬於物件導向型態，通常是函數式的數學類型並採用大量的統計運算，但不一定都是這樣的型態。其中可以嘗試在資料向量與資料矩陣的每一列（row）中使用物件，而不是以單純的功能項目進行思量。

SRP

在機器學習程式中，人們意識到的最大挑戰之一是：程式碼與資料彼此相依。若缺少資料，則機器學習演算法毫無價值可言；如果沒有機器學習演算法，就不曉得該怎麼處理資料。因此根據定義，兩者是緊密相互交織與耦合。這種緊密耦合的相依性可能是機器學習專案失敗的最大原因之一。

這種相依性內容顯現出機器學習程式碼中的兩個問題：糾結（entanglement）與黏合程式碼（glue code）。糾結有時又稱為 Changing Anything Changes Everything（稍微變更就改變一切）或 CACE 原則。最簡單的範例是機率議題。如果從某個分布中移除一項機率運算，那麼必須調整其餘的內容，而因此會違反 SRP。

其中可能的緩解策略包括分離模型、分析維度相依性[4]以及應用正則化 (regularization) 技術[5]。之後討論到貝氏模型（Bayesian models）與機率模型時，會回來探討上述問題。

黏合程式碼是在某個程式專案中隨時間累積的程式碼，其目的通常是將兩個分開的程式片段粗糙的黏合。往往也屬於嘗試解決所有問題而非只處理一個問題的程式碼類型。

無論機器學習研究人員願意承認與否，通常實際的機器學習演算法本身相當簡單，而相關的程式碼是專案組成的大部分內容。依據其中所使用的函式庫，無論是 GraphLab、MATLAB、scikit-learn 還是 R，皆有各自所屬的向量與矩陣相關實作內容，這些實作是機器學習主要處理的項目。

OCP

回顧之前所述，OCP 容許開放類別的功能擴充但不可進行內容修改。機器學習程式中 CACE 的問題會顯露出 OCP 議題。OCP 問題可能會在任何軟體專案中表露，但在機器學習專案中，通常被視為隱藏回饋循環。

一個隱藏回饋循環的明顯例子是**預測性警務**（*predictive policing*）。在過去幾年中，許多研究人員已經表明：機器學習演算法可以應用於判斷犯罪發生所在之處。初步結果顯示這些演算法運作得相當好。然而，這些演算法也有黑暗的一面。

雖然這些演算法可以顯示犯罪發生的地方，但自然導致的現象是：警方會開始頻繁巡視這些高風險地區，並在那裡發現更多的犯罪事件，因而造成自我強化此一演算法。這種現象也可以稱為確認偏誤（confirmation bias），或者是確認先入為主觀念的偏見，也會產生對某些人口統計資料或鄰里（neighborhoods）衍生系統性歧視的缺失。

儘管不容易發現隱藏回饋循環，還是得用敏銳的眼力監測而將其排除。

LSP

現今較少談論 LSP，因為許多程式設計師都在倡導繼承的組成內容。然而在機器學習的世界中，違反 LSP 的情況頻繁。往往已知的資料集，尚未完全有對應解，有時這些資料集的維度有數千個。

4 H. B. McMahan et al., “Ad Click Prediction: A View from the Trenches.” In The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, August 11–14, 2013.

5 A. Lavoie et al., “History Dependent Domain Adaptation.” In Domain Adaptation Workshop at NIPS ’11, 2011.

對這些資料集執行相關的演算法，實際上可能會違反 LSP。機器學習程式中一個常見的顯現議題是未充分利用的資料相依性。往往已知的資料集包含數千個維度，有時可以衍生出相關資訊而有時則不會產生。模型可能會採用所有的維度，然而可能不常使用到某個維度。例如，針對蘑菇進行有毒性或可食用的分類過程中，像氣味這樣的資訊可能是個很重要的區分指標，然而蕈環值就不是重要資訊。蕈環值的粒度（granularity）小，只有零、一、二的可能值；因此對於蘑菇分類模型確實沒有多大的幫助，所以將此資訊從模型中剔除，並不會大幅降低整體效能。

讀者可能會覺得：為什麼這會與 LSP 相關，原因是如果只能使用資料點（或特徵）的最小集合，則已經建立出最佳的模型。這也與 Ockham's Razor（奧坎剃刀）的概念相仿，表明最簡單的解決方案最好。

ISP

ISP 是用戶端特定介面優於通用介面的概念。在機器學習專案中，由於資料與程式碼的緊密耦合，使得這個概念往往難以執行。機器學習程式通常會因為兩種類型的問題而違反 ISP：**可見性負債**（*visibility debt*）與**不穩定資料**（*unstable data*）。

例如，某家企業有個報表資料庫用於收集銷售、貨運與其他重要項目的相關資訊。其中都是透過將資料導入資料庫的某種專案進行管理。此資料庫所定義的客戶是個機器學習專案，這個專案採用以前的銷售資料預測未來的銷售結果。而某天在資料庫清理過程中，某人對一份資料表重新命名，將過去非常困擾的稱呼換成非常合宜的名稱。因此讓所有人亂成一團，大家都想知道發生什麼狀況。

事件最終的結果是機器學習專案不是資料的唯一使用者；其中還附有六個 Access 資料庫相互關聯。事實上，許多未宣告的使用者本身就是機器學習專案的負債。

這種類型的負債稱為可見度負債，而這項負債通常不會影響專案的穩定性，有時候，隨著特徵的建立，在某種程度上能恢復一切。

資料會與對它進行歸納的程式碼相依，因此要建立穩定的專案需有穩定的資料。然而情況往往並非如此。以股票價格為例；上午開盤時可能具有投資價值，但經過幾個小時後卻變得毫無價值。

這終究是違反 ISP，因為正在查看的是一般資料流，而非用戶端特定的資料流，如此可能導致投資組合的交易演算法非常難以建構。一個常用的技巧是針對資料而建立某種指數加權方案；另一個重點是資料流版本化，這個版本化方案能作為模型預測波動性限制的可行方式。

DIP

相依性反轉原則是針對資料的組織進行限制，並針對將來的變化而讓程式碼更具彈性。在機器學習專案中會以兩種特定方式具體呈現相關議題：**管線叢林**與**實驗路徑**。

管線叢林在資料驅動的專案中很常見，幾乎屬於一種黏合程式碼。這是把備妥的資料進行合併的作業，在某些情況下，此程式碼將所有內容都綁在一起，因此模型可以利用這些備妥的資料。然而這些叢林會隨著時間逐漸變得複雜而不能使用。

機器學習程式碼需求軟體和資料，兩者相互交織而不可分割。有時必須在軟體正式上線中進行測試。某些時候在機器上執行測試會帶來不實的期待，而需要用一行程式碼進行實驗。這些實驗路徑會隨著時間累積，最終污染相關的工作空間。減少此一相關負債的最好方式：引用 tombstoning，這是引自 C 語言的一種舊有技術。

Tombstones 是一種將準備刪除的項目進行標記的方法，如果在軟體正式上線中呼叫這個方法，它會將此事件紀錄到某個記錄檔（logfile）中，以便在稍後對程式碼庫進行批次清除作業。

對於那些研究記憶體回收（garbage collection）的人來說，最有可能聽聞過以此方法進行標記與清除作業。基本上將準備刪除的物件作個標記動作，稍後才將這個標記物件清除。

複雜但並非不可能的機器學習程式

有時，機器學習程式格外難以撰寫與理解，但絕非不可能達成。之前筆者使用飛航作類比，並使用 SOLID 指引作為「起航之前」的檢查清單，進而撰寫出成功的機器學習程式——雖然複雜（complex），但不一定棘手（complicated）。

同樣的道理，可以將機器學習程式與太空船飛航相比——當然這樣的比擬在以前曾經做過，不過至今仍屬前瞻作為。利用 SOLID 檢查清單模型，可以使用 TDD 與重構作業而有效的啟用相關程式碼。實際上，想寫出成功的機器學習程式就要盡力依循本章列出的設計原則，並撰寫測試項用於支持程式碼形式的假設因果。寫出有效程式碼的另一個關鍵因素是保有靈活性，以便適應現實世界中會遇到的變化。

TDD：科學方法 2.0

每位實際的科學家都是一位夢想家兼質疑者。敢把人類送到月球上是大膽的作為，然而透過系統性的研發，因而取得相當大的成就；機器學習程式碼也是如此，一些應用程式令人著迷，但也難以順利完成。

成功的秘訣是：針對機器學習專案，使用 SOLID 檢查清單、TDD 工具以及重構作業實現。

TDD 比起上述的原則更會是一種解決問題的風格。測試作業提供一種回饋循環，而能夠用來解決棘手的問題。科學家主張需要事先假設、進行測試並依據理論，而筆者主張身為一名 TDD 實踐者，對於 Red（測試失敗）- Green（測試通過）- Refactor（重構程式）的程序正是可行的作業。

本書將深入探討 TDD 與 SOLID 原則應用於機器學習的內容，並以重構作業為目的而建構穩定、可擴展且易於使用的模型。

重構知識之路

如上所述，重構是工作修訂的能力，並重新思考曾經陳述的內容。本書的其他章節會在相關演算法運用時針對機器學習的易犯錯誤所進行的重構作業，加以探討。

本書的預期

本書主要涵蓋機器學習的基礎內容，並期望讀者閱讀本書之後，對於如何撰寫機器學習相關的程式來說，能夠獲得更佳的理解，以及針對如何將程式部署到正式上線環境與達到一定的運作規模而言，能夠獲得有效的認識。機器學習是一個令人著迷的專業領域，可以實現很多的需求，但如果缺少紀律、檢查清單與指引，則許多的機器學習專案注定要面臨失敗。

後續的內容會涉及 SOLID 原則探討、程式碼測試（利用各種途徑為之）以及重構作業的項目，屆時回溯關聯本章提到的這些初始原則，進而讓讀者能夠持續學習與改善程式的效能。

往後的每一章會說明章節所使用的 Python 套件，並描述一般性的測試計劃。儘管機器學習程式無法以一對一的明顯方式進行測試，但依然可以撰寫測試項協助了解問題。