前言

在你的手中，你正拿著一本多年前我就希望擁有的 C++ 書籍。不是作為我的第一本書，不是的，而是作為在我已經消化了這種程式語言的機制，並且能夠超越 C++ 語法進行思考之後一本進階的書籍。是的，這本書必定會幫助我更理解可維護軟體的基本面向，而且我確信它也會對你有所幫助。

我為什麼寫這本書

到我真正開始鑽研這種程式語言的時候（那是在第一個 C++ 標準發佈後幾年），我幾乎已經讀遍了所有 C++ 的書籍。但是，儘管這些書中有很多很棒的內容，而且肯定能為我目前擔任 C++ 培訓師和顧問的職涯做好準備，但這些書都太專注於小的細節和具體實作，而離可維護軟體的整體情況太遠。

當時，真正關注整體情況，處理大型軟體系統開發的書非常少。其中有 John Lakos 的《*Large Scale C++ Software Design*》[1]，這是一本很好的書，但太偏重於逐字介紹依賴性管理；以及所謂的四人幫書籍，它是關於軟體設計模式的經典著作[2]。不幸的是，多年來，這種情況並沒有真正地改變：大多數的書籍、講座、部落格等，主要關注在語言的技術和特徵——小細節和特定面向。很少有，在我看來是太少了，關注在可維護的軟體、可改變性、可擴展性和可測試性的新版本。而且如果他們試圖這樣做，非常不幸地，他們很快就會回到解釋語言機制和展示功能的常見習慣之中。

1 John Lakos，《*Large-Scale C++ Software Design*》（Addison-Wesley，1996）。

2 Erich Gamma 等人，《*Design Patterns: Elements of Reusable Object-Oriented Software*》（Addison-Wesley，1994）。

這就是我寫這本書的原因。與其他大多數書籍相比，這本書沒有花時間在技術或語言的許多特徵上，而是主要關注於軟體一般的可改變性、可擴展性和可測試性。這本書並不假裝使用新的 C++ 標準或功能會使軟體的好壞有所區別，而是清楚地展現出，對依賴性的管理才是決定性的，我們程式碼中的依賴性決定了程式的好壞。因此，這確實是 C++ 世界裡罕見的一本書，因為它專注在整體的情況：軟體設計。

關於本書

軟體設計

在我看來，好的軟體設計是每個成功軟體專案的要素。然而，不管它基本的作用是什麼，在這個主題上的文獻卻很少，關於該做什麼和如何正確做的建議也很少，為什麼呢？嗯，因為這很困難，非常困難，這可能是我們撰寫軟體時必須面對的最困難的面向。而且這是因為沒有單一「正確」的解決方案，沒有可以代代相傳給軟體開發者的「黃金」建議，它總是視情況而定。

儘管有這個限制，我還是會就如何設計好的、高品質的軟體提供建議。我將提供能幫助你更好地理解如何管理依賴性，並將你的軟體變成可以運作數十年的設計原則、設計指南和設計模式。如之前所說的，沒有什麼「黃金」建議，而且這本書也不會擁有任何終極或完美的解決方案。取而代之的，我將嘗試展示好軟體最基本層面，最重要的細節，不同設計的多樣性和利弊。我也將規劃固有的設計目標，並且演示如何用 Modern C++ 完成這些目標。

Modern C++

十多年來，我們一直在歌頌 Modern C++ 的來臨，為這個語言許多新的特徵和擴展而喝彩，並透過這樣做，建立了 Modern C++ 將幫助我們解決所有軟體相關問題的印象。在本書中不是這樣，本書不會假裝在程式碼中扔幾個智慧型指標就能使程式碼「Modern」或自動產生好的設計。此外，本書也不會將 Modern C++ 展示為各式各樣的新特性，而是會展現程式語言的哲理是如何演進，以及現今我們實作 C++ 解決方案的方式。

但當然，我們也會看到程式碼，很多的程式碼。而且當然，這本書也會使用比較新的 C++ 標準（包括 C++20）的特性。然而，它也會努力地強調設計與實作細節和使用的特性無關。新的特性不會改變關於什麼是好設計或壞設計的規則；它們只是改變了我們實作好設計的方式，它們使好設計更容易實作。因此，本書展示並且討論了實作細節，但（希望）不會在這些細節之中迷失，而是始終保持關注整體的情況：軟體設計和設計模式。

設計模式

一開始提到設計模式，你就會不經意地聯想到對物件導向程式設計和繼承階層結構的期望。是的，本書將展示許多設計模式物件導向的起源。然而，它將著重於強調，善用設計模式不是只有一種方法的事實。我將用許多不同的範例演示，設計模式的實作是如何演進和多樣化，包括物件導向程式設計、通用程式設計和函數式程式設計。本書承認沒有一種真正範例的現實，以及不要妄求只有單一的方法，一種對所有問題總是有用的解決方案。相反地，本書試圖展示 Modern C++ 到底是什麼：結合所有的範例，將它們編織成強大且持久的網，並創造出能持續數十年軟體設計的機會。

我希望證明這本書是 C++ 文獻中缺少的一塊，我希望它能像幫助我一樣盡可能地幫助你。我希望它擁有一些你一直在尋找的答案，並提供你一些你所缺少的關鍵見解。而且我也希望這本書能讓你感到一些興趣，並激勵你閱讀它所有的內容。然而，最重要的是，我希望這本書能讓你看到軟體設計的重要性，以及設計模式所發揮的作用。因為，就像你將看到的，設計模式無所不在！

誰應該讀這本書

本書對每一位 C++ 開發者都有用。它特別是為了每一位有興趣了解可維護性軟體通常的問題，並學習這些問題常見解決方案的 C++ 開發者所寫（而我假設的確是*每一位* C++ 開發者）。然而，這本書不是寫給 C++ 初學者的。事實上，本書中大多數的指導原則都需要對一般軟體開發有一定的經驗，特別是對 C++ 開發。例如，我假設你已經牢牢掌握了繼承階層結構的語言技術，並且對模板有一些經驗。然後，我就能夠在必要和適當的時候拿出相應的特徵。偶爾，我甚至會拿出一些 C++20 的特徵（特別是 C++20 的概念）。然而，由於重點是軟體設計，所以我很少會沉浸在某個特定特徵的說明上，因此如果你對某個特徵不清楚的話，請查閱你最喜歡的 C++ 語言參考資料，我只是偶爾會添加一些提醒，主要是關於常見的 C++ 慣用法（像是 5 的規則（*https://oreil.ly/fzS3f*））。

本書結構

本書分為一些章節，每一章都包含一些指導原則，而每個指導原則都專注在可維護性軟體的一個關鍵面向、或一個特定的設計模式。因此，這些指導原則代表了主要的精華，是我希望能帶給你最大價值的面向。它們的寫法是讓你可以從前到後地閱讀所有指導原則，但是因為它們只是鬆散地結合在一起，所以你也可以從吸引你注意的指導原則開始閱讀。然而，它們不是各自獨立的；因此，每個指導原則都包含了與其他指導原則必要的交叉引用，以讓你能看到所有東西都是互有關聯的。

第一章

軟體設計的藝術

什麼是軟體設計？你為什麼要關心它？在本章中，我將為這本軟體設計的書打好基礎。我將從總體上說明軟體設計，幫助你理解為什麼它對專案的成功非常重要，以及為什麼它是你應該做對的一件事。但是你也將看到，軟體設計很複雜，非常複雜。事實上，它是軟體開發中最複雜的部分。因此，我也會說明一些能幫助你保持在正確道路上的軟體設計原則。

在第 2 頁的「指導原則 1：理解軟體設計的重要性」中，我將著眼於整體情況上，並說明軟體是被期待會改變的。因此，軟體應該有能力應付改變。然而，說比做要容易，因為在現實中，耦合和依賴性使身為開發者的我們生活更為困難。這個問題會經由軟體設計來解決。我將介紹軟體設計是管理依賴性和抽象化的藝術——是軟體工程的一個重要部分。

在第 10 頁的「指導原則 2：為改變而設計」中，我將明確地討論耦合和依賴性，並幫助你理解如何為改變而設計，以及如何使軟體更具適應性。為了這個目標，我將介紹**單一責任原則**（*SRP*）和**不要重複自己**（*DRY*）兩個原則，這將有助於你達成這目標。

在第 22 頁的「指導原則 3：分離介面以避免人為的耦合」中，我將擴展關於耦合的討論，並特別地定位在經由介面的耦合。我也會介紹**介面隔離原則**（*ISP*），作為減少由介面造成人為耦合的一種方法。

在第 26 頁的「指導原則 4：為可測試性而設計」中，我將重點放在因人為耦合而造成的可測試性問題。特別是，我將提出如何測試一個私有成員函數的問題，並證明真正的解決方案是隨之而來的分離關注點應用。

在第 33 頁的「指導原則 5：為擴展而設計」中，我將討論一種重要的改變：擴展。就如同程式碼應該很容易改變一樣，它也應該很容易擴展。我將提供一個如何實現這目標的想法，並且將展示**開放 - 封閉原則**（*OCP*）的價值。

指導原則 1：理解軟體設計的重要性

如果我問你哪些程式碼屬性對你最重要，你思考之後可能會說像是可讀性、可測試性、可維護性、可擴展性、可重用性和可擴充性等屬性；而且我也完全同意。但是現在，如果我問你如何達成這些目標，這將可能是你開始列出一些 C++ 的特徵：RAII、演算法、lambda、模組等等的好機會。

特徵不是軟體設計

沒錯，C++ 提供了很多特徵，非常多！在列印出的近 2000 頁 C++ 標準中，大約有一半是用來解釋語言的機制和特徵[1]。而且從 C++11 發佈以來就明確的承諾這方面將會有更多：每三年，C++ 標準化委員都會提供我們一個新的 C++ 標準與補充的、全新的特徵。了解了這點後，對於 C++ 社群非常強調特徵和語言機制，就不會感到太大的驚訝。大多數的書籍、講座和部落格都專注在特徵、新函數庫和語言的細節上[2]。

這似乎讓人覺得特徵是關於用 C++ 撰寫程式中最重要的事情，而且對於 C++ 專案的成功至關重要。但老實說，它們並不是。無論是關於所有特徵的知識，或是對 C++ 標準的選擇，都不負有專案成功的責任。不，你不應該期望特徵會拯救你的專案。相反地：即使使用較舊的 C++ 標準，而且即使只用了可用特徵的一個子集，專案也可能會非常成功。將軟體發展中人的方面丟到一邊，對於一個專案的成功或失敗問題，更重要的是軟體的整體**結構**。這是最終負有可維護性責任的結構：它對修改程式碼、擴展程式碼和測試程式碼有多容易？如果不能輕鬆地改變程式碼、新增功能，並且因為測試的驗證而對它正確性具有信心，那麼專案就已經處於它生命週期的末期了。結構也負有專案可擴充性的責任：在專案被自己的重量壓垮之前，它能成長到多大？在踩到彼此的腳趾頭之前，有多少人可以從事於實現專案的願景？

1 但是，當然你甚至不會嘗試列印目前的 C++ 標準。你要麼使用官方 C++ 標準的 PDF 檔案（*https://oreil.ly/bZUDd*），要麼使用目前的工作草案（*https://oreil.ly/r46ta*）。然而，對於你日常大部分的工作，你可能想參考 C++ 的參考網站（*https://oreil.ly/z0tKS*）。

2 不幸地，我無法呈現任何的數字，因為我很難說我對 C++ 廣闊的領域有完整的概述。相反地，我甚至可能對我所知道的來源都沒有完整的概述！所以，請將此視為我的個人印象和我對 C++ 社群的看法，而你可能會有不同的印象。

整體結構是專案的設計，設計在專案的成功中扮演著比任何特徵都重要的角色。好的軟體主要不是指正確的使用任何特徵，而是指堅固的架構和設計。好的軟體設計可以容忍一些不好的實作決策，但不好的軟體設計不能只藉由英雄式的使用特徵（舊的或新的）來挽救。

軟體設計：管理依賴性和抽象化的藝術

為什麼軟體設計對專案的品質如此重要？假設現在所有事情都很完美，只要你的軟體中沒有任何改變，而且只要不必增加任何東西，那你就沒事。但是，這種狀態可能不會持續太久。期待某些事會有一些改變是合理的。畢竟，在軟體開發中經常發生事的就是改變。改變是我們所有問題（也是大部分我們解決方案）背後的驅動力。這就是為什麼軟體會被稱為**軟體**：因為與硬體比較，它是柔軟與可塑的。是的，**軟體**被期待能夠容易地適應千變萬化的需求。但是如同你可能知道的，在現實中這種期待可能並不總是真實的。

為了說明這一點，讓我們想像你從你的問題跟蹤系統中選擇了一個被團隊評為預期努力為 2 的問題。不管 2 在你自己的專案中表示什麼，它聽起來肯定不像是一個大的工作，所以你相信它將會很快地完成。誠心誠意地，你首先花了一些時間了解所預期的是什麼，然後你開始在某個實體 A 中做了一個改變。因為來自測試的即時回饋（你很幸運有測試！），你很快地被提醒還必須解決實體 B 中的問題。這讓人感到驚訝！你並沒有預期會牽涉到 B。不過，你還是繼續前進，無論如何都要遷就 B。然而，在一次出乎意料的、夜間的建構顯示，這會導致 C 和 D 停止工作。在繼續之前，你現在對這個問題進行深入一點的研究，發現問題的根源分散在程式碼庫的大部分範圍內。這個小的、起初看起來無辜的工作已經演變成一個大的、有潛在風險的程式碼修改[3]。你對快速地解決這個問題的信心已經消失，而且你對本週剩下時間的計劃也是如此。

也許這個故事你聽起來很熟悉，也許你甚至可以貢獻一些自己的艱苦經歷。事實上，大多數開發者都有類似的經歷，而且大多數這些經歷都有相同的問題根源。通常，這個問題可以簡化成一個字：**依賴性**。如同 Kent Beck 在他關於測試驅動開發的書中所表達的[4]：

> 依賴性是所有規模軟體開發中的關鍵問題。

3 程式碼修改是否有風險，在很大程度上可能取決於你的測試覆蓋率。好的測試覆蓋率實際上可能會吸收一些不好的軟體設計可能造成的損害。

4 Kent Beck，《*Test-Driven Development: By Example*》（Addison-Wesley，2002）。

依賴性是每個軟體開發者生存的禍根。「但當然會有依賴性，」你爭辯道。「總是會有依賴性的，否則不同的程式碼片段要如何一起工作？」當然，你是對的。不同的程式碼片段需要一起工作，而且這種互動將總是會產生某種形式的耦合。但是，雖然這是必須、不可避免的依賴性，但也有一些是因為我們缺乏對根本問題的理解，對整體情況沒有清楚的認識，或者只是沒有足夠的重視，而不小心引入的人為依賴性。當然，這些人為依賴性是有害的，它們使我們對軟體的理解、軟體的改變、增加新的功能、以及撰寫測試變得更困難。因此，將人為依賴性保持在最低限度，就算不是軟體開發者的主要工作，也會是主要工作之一。

這種最小化依賴性是軟體架構和設計的目標。用 Robert C. Martin 的話來說就是[5]：

> 軟體架構的目標是在建構和維護所需系統時，最小化所需要的人力資源。

架構和設計是在任何專案中最小化工作努力所需要的工具。它們處理依賴性，並且經由抽象化來降低複雜性。用我自己的話說[6]：

> 軟體設計是管理軟體元件之間相互依賴性的藝術。它的目的是最小化人為的（技術）依賴性，並且引入必須的抽象化和妥協。

是的，軟體設計是一門藝術。它不是一門科學，而且沒有一套簡單而清楚的答案[7]。很多時候，設計的整體情況使我們困惑，而我們被軟體實體複雜的相互依賴性所淹沒。但我們試圖要處理這種複雜性，並且透過引入正確的抽象化來減少它。這樣一來，我們將細節程度保持在一個合理的水準。然而，太多時候團隊中的個別開發者可能對於架構和設計有不同的想法，我們可能無法實現自己對設計的願景，為了向前推進而被迫做出妥協。

抽象化這個術語用於不同的背景。它用於將功能和資料項目組織成資料類型和函數。但它也用於描述共同行為的建模，以及一組需求和期望的表示。在這本關於軟體設計的書中，我主要將這個術語用於後者（請參考第 2 章）。

5 Robert C. Martin，《*Clean Architecture*》（Addison-Wesley，2017）。

6 這些事實上是我自己說的，因為關於軟體設計並沒有一個單一、共同的定義。因此，你可能對什麼是軟體設計的含義有自己的定義，這非常好。但是，注意在本書，包括對設計模式的討論，都是基於我的定義。

7 先說清楚：計算機科學是一門科學（名字裡就有）。而軟體**工程**似乎是科學、工藝和藝術的混合形式。而且後者的一個面向就是軟體**設計**。

注意在前面的引言中，**架構**和**設計**這兩個術語可以互換，因為它們非常相似，而且有相同的目標；然而，它們並不一樣。如果你看一下軟體發展的三個層次，這具有差異的相似性，就會變得很清楚。

軟體發展的三個層次

軟體架構和**軟體設計**只是軟體發展三個層次中的兩個，它們被**實作細節**層次所補充。圖 1-1 提供了這三個層次的概觀。

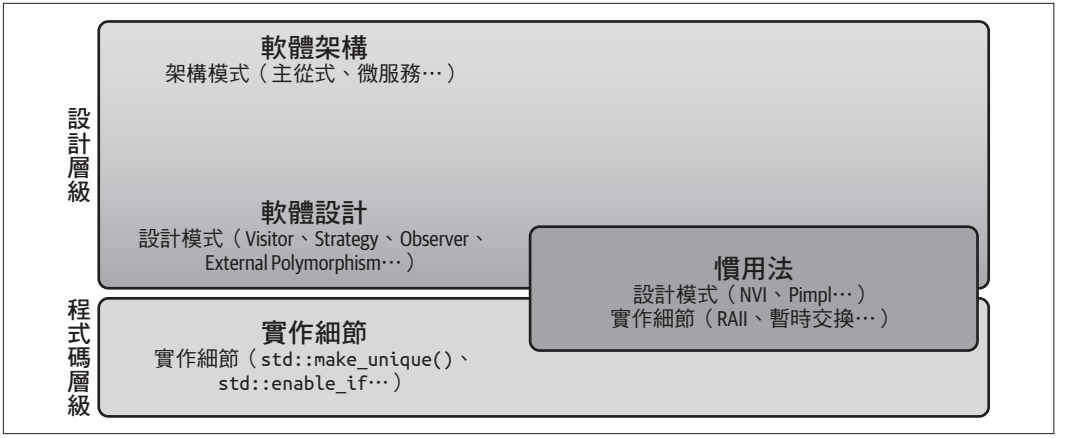

圖 1-1　軟體開發的三個層次：軟體架構、軟體設計、和實作細節，慣用法可以是設計或實作模式

為了讓你對這三個層次有感覺，讓我們從了解架構、設計和實作細節之間關係的一個現實世界例子開始。將你自己當成是一個建築師的角色。不，請不要想像自己坐在電腦前舒適的椅子上，旁邊還有一杯熱咖啡，而是想像自己在外面的建築工地上。是的，我談的是建築的建築師[8]。像這樣的建築師，你將負責房子的所有重要屬性：它與周圍環境的融入、它的結構完整性、房間的安排、管道等等。你也將負責賞心悅目的外觀和功能品質——也許是一個大客廳，廚房和餐廳之間方便的通道等等。換句話說，你將負責整個建築，那些以後很難改變的東西，但是你也會處理與建築相關的小設計面向。然而，要說出這兩者之間的區別很難：建築和設計之間的界限似乎是流動的，而且沒有明確的區分。

8　用這個比喻，我不是嘗試要暗示建築業的建築師整天在建築工地工作。很可能，這種建築師和你我這樣的人一樣，花很多時間在舒適的椅子上和電腦前。但我想你會抓到重點的。

然而，這些決定將是你責任的終點。作為一位建築師，你不用擔心冰箱、電視或其他傢俱的擺放位置。你不用處理關於圖片和其他裝飾品要放哪裡的所有枝微末節。換句話說，你不會處理這些細節；你只需確保屋主有好好生活所需要的結構。

在這個比喻中的傢俱和其他「枝微末節」對應於軟體開發中最低和最具體的層次，即實作細節；這個層次處理如何實作一個解決方案。你選擇必要的（和可用的）C++ 標準或它的任何子集，以及適當的特徵、關鍵字和使用的語言特性，並且處理像是記憶體獲取、異常安全、性能等面向。這也是**實作模式**的層次，像 `std::make_unique()` 作為**工廠函數**，`std::enable_if` 作為明確受益於 SFINAE 的循環解決方案等等[9]。

在軟體設計中，你開始關注全域。關於可維護性、可改變性、可擴展性、可測試性和可擴充性的問題，在這個層次上更為明顯。軟體設計主要處理軟體實體的相互作用，在前面的比喻中，是由房間、門、管道和電纜的排列表示。在這個層次，你處理組件（類別、函數等）的實體和邏輯上的依賴性[10]。這是設計模式的層次，像是 Visitor、Strategy 和 Decorator，如第 3 章所說明的，它們定義了軟體實體之間的依賴結構。這些模式，通常可以從一種語言轉換到另一種語言，幫助你將複雜的事情分解成可消化的片段。

軟體架構是三個層次中最模糊、最難用語言表示的，這是因為沒有共同、普遍接受的軟體架構定義。雖然對於架構確切是什麼可能有很多不同的看法，但有一個似乎是每個人都同意的面向：架構通常意味著重大的決策，是在你的軟體中未來最難改變的那些面向：

> 架構是你希望你能在專案早期就做出正確的決策，但並不一定會比其他的決策做得更正確[11]。
>
> —Ralph Johnson

9 替換失敗不是錯誤（SFINAE）是一個基本的模板機制，通常用於替代 C++20 概念以約束模板。關於 SFINAE 和 `std::enable_if` 的具體說明，請參考你喜歡的與 C++ 模板相關的書籍。如果你沒有，一個很好的選擇是 C++ 模板聖經：David Vandevoorde、Nicolai Josuttis 和 Douglas Gregor 所著的《*C++ Templates: The Complete Guide*》（Addison-Wesley）。

10 對實體和邏輯依賴性管理的更多資訊，請參考 John Lakos 的「壩堤」著作，《*Large-Scale C++ Software Development：Process and Architecture*》（Addison-Wesley）。

11 Martin Fowler，「Who Needs an Architect?」*IEEE Software*，20，no.5（2003），11-13，*https://doi.org/10.1109/MS.2003.1231144*。

在軟體架構中，你使用像是**主從架構**、**微服務**等架構模式[12]。這些模式也處理如何設計系統、你可以改變軟體一部分而不影響任何其他部分的問題。與**軟體設計**模式類似，它們定義並解決軟體實體之間的結構和相互依賴性。然而對比於設計模式，它們通常處理的是關鍵角色，即軟體的大型實體（例如，模組和組件，而不是類別和函數）。

從這個角度看，軟體架構表示軟體方法的整體策略，而軟體設計則是使這個策略發揮作用的戰術。這描述的問題出在沒有定義所謂的「大型」，特別是微服務的出現，使得在小型實體和大型實體之間越來越難畫出一條清楚的界限[13]。

因此，架構通常被描述為專案中精湛的開發者所察覺的關鍵決策。

造成架構、設計和細節之間分隔更困難的是**慣用法**的概念。**慣用法**是循環問題常用、但針對特定語言的解決方案。因此，慣用法也表示一種模式，但它可以是一種**實作模式**，也可以是一種**設計模式**[14]。一般說來，C++ 慣用法是 C++ 社群在設計或實作上最佳的實踐。在 C++ 中，大多數慣用法都屬於實作細節的範疇。例如，例如，有一個**複製和交換慣用法**（*https://oreil.ly/hioCd*），你可能會從複製指定運算子的實作中熟知它，以及 *RAII* **慣用法**（*https://oreil.ly/55blq*）（資源獲取是初始化——你一定對它很熟悉了；如果不是如此，請看你第二喜歡的 C++ 書籍[15]）。這些慣用法都沒有引入抽象化，也都沒有幫助解耦。儘管如此，它們對於實作好的 C++ 程式碼是不可或缺的。

我聽到你問：「你能不能說得更具體一些？RAII 不是也提供某種形式的解耦嗎？它沒有將資源管理從業務邏輯中解耦嗎？」你是對的。RAII 分離了資源管理和業務邏輯。然而，它不是藉由解耦，也就是抽象的方式完成，而是藉由封裝的方式。抽象和封裝都能幫助你使複雜的系統更容易理解和改變，但是抽象解決的是出現在軟體設計層次的問題，而封裝解決的是出現在實作細節層次的問題。這引用自維基百科的內容（*https://oreil.ly/BeFXr*）：

> 作為資源管理技術，RAII 的優點是它提供了封裝、異常安全 [...] 和局部性 [...]。提供封裝是因為資源管理邏輯只在類別中定義一次，而不是在每個呼叫的位置定義。

12 在 Sam Newman 的著作《*Building Microservices: Designing Fine-Grained Systems*》第 2 版（O'Reilly）中，可以找到關於微服務非常好的介紹。

13 Mark Richards 和 Neal Ford 所著的《*Fundamentals of Software Architecture: An Engineering Approach*》（O'Reilly，2020）。

14 **實作模式**這個術語最早用於 Kent Beck 的著作《*Implementation Patterns*》（Addison-Wesley）。在本書中，我用這個術語提供了與**設計模式**這個術語清楚的區別，因為**慣用法**這個術語可以指軟體設計層次或是實作細節層次的模式，我將一致地使用這個術語來指實作細節層次上的常用解決方案。

15 當然，在這本書之後的第二喜歡。如果這是你唯一的一本書，那麼你可以參考 Scott Meyers 的經典著作《*Effective C++: 55 Specific Ways to Improve Your Programs and Designs*》第三版（Addison-Wesley）。

雖然大多數慣用法屬於實作細節的範疇，但也有一些慣用法屬於軟體設計的範疇。其中兩個例子是**非虛擬介面（*NVI*）慣用法**和***Pimpl* 慣用法**。這兩個慣用法是基於兩個傳統的設計模式：分別是 *Template Method* 設計模式和 *Bridge* 設計模式[16]。它們引入了抽象化，並且幫助解耦和為了改變和擴展而設計。

專注在特徵上

如果軟體架構和軟體設計如此重要，那麼為什麼我們在 C++ 社群中會如此強烈地專注在特徵上？為什麼我們要建立 C++ 標準、語言機制和特徵對專案來說是決定性的錯覺？我認為這有三個強大的原因。首先，因為有這麼多的特徵，有時還會有複雜的細節，我們需要花很多時間來談論如何適當地使用它們全部。我們需要對哪些使用是好的以及哪些使用是壞的建立一個共識。我們作為一個社群，需要發展一種慣用法的 C++ 觀念。

第二個原因是，我們可能會把錯誤的期望放在特徵上。例如，讓我們考慮 C++20 模組。不談細節，這個特徵的確被認為是自 C++ 開始以來最大的技術革命。模組可能最終會結束將標頭檔包含到原始檔案中這種有問題和繁瑣的做法。

由於有這種可能，對這特徵的期待是巨大的。有些人甚至期待模組透過修復結構問題而拯救他們的專案。不幸的是，模組很難滿足這些期待：模組不能改善程式碼的結構或設計，而只能代表目前的結構和設計。模組不能修復設計的問題，但是它們也許能夠使缺陷成為可見。因此，模組根本不能拯救你的專案。所以，我們的確可能放了太多或錯誤的期待在特徵上。

最後，但並非最不重要的，第三個原因是，儘管有大量的特徵和它們的複雜性，但與軟體設計的複雜性相比，C++ 特徵的複雜性還算小。解釋一組給定特徵的規則，不管它們包含了多少特殊情況，都比解釋軟體實體解耦的最佳方式要容易得多。

雖然對所有與特徵相關的問題通常都有一個好的答案，但軟體設計中常見的答案是「看情況再說」。這個答案甚至可能不是缺乏經驗的證據，而是領悟到使程式碼更可維護、可改變、可擴展、可測試和可擴充的最佳方法，高度依賴於許多專案具體的因素。在許多實體之間有複雜相互作用的解耦的確可能是大家曾經面臨過最具挑戰性的工作之一：

> 設計和撰寫程式是人類的活動；忘記這一點，那一切都會失去[17]。

16 Template Method 和 Bridge 設計模式是由 Erich Gamma 等人在所謂的四人幫（GoF）書籍《*Design Patterns: Elements of Reusable Object-Oriented Software*》中介紹的 23 種經典設計模式中的 2 種。本書中我不會進入 Template Method 的細節，但是你可以在包括 GoF 書籍本身在內的各種教科書中找到很好的說明。然而，我將在第 424 頁的「指導原則 28：建構 Bridge 以移除實體依賴性」中說明 Bridge 設計模式。

17 Bjarne Stroustrup，《*The C++ Programming Language*》，第三版（Addison-Wesley，2000）。

對我來說，這三個原因的結合就是為什麼我們會如此專注在特徵上。但是，請不要誤會我的意思。這並不是說特徵不重要。剛好相反，特徵很重要。是的，討論特徵並學習如何正確使用它們是必要的，但再次強調，光靠它們是無法拯救你的專案。

專注在軟體設計和設計原則上

雖然特徵很重要，而且雖然討論它們當然很好，但軟體設計更重要。軟體設計是必不可少的。我甚至主張，它是專案成功的基礎。因此，在這本書中，我將嘗試真正專注在軟體設計和設計原則上，而不是特徵。當然，我仍然會展示好的、最新的 C++ 程式碼，但我不會強力推動使用最新的和最重大的語言擴充部分[18]。在合理和有益的情況下，我會使用一些新的特徵，像是 C++20 的概念，但我不會關心 `noexcept` 或是廣泛地使用 `constexpr`[19]。反而，我將嘗試處理軟體困難的面向。大多數情況下，我將專注在軟體設計、設計決策背後的推理、設計原則、管理依賴性和處理抽象化等。

總之，軟體設計是撰寫軟體的關鍵部分。軟體開發者應該對軟體設計有很好的理解，以便寫出好的、可維護的軟體。因為畢竟，好的軟體是低成本的，而壞的軟體則是昂貴的。

指導原則 1：理解軟體設計的重要性

- 將軟體設計當成撰寫軟體的一個必不可少的部分。
- 少專注在 C++ 語言的細節，多專注在軟體設計上。
- 避免不必要的耦合和依賴性，使軟體對頻繁的改變更能適應。
- 理解軟體設計是管理依賴性和抽象化的藝術。
- 將軟體設計和軟體架構之間的界限看作是流動的。

18 很佩服 John Lakos，在他的《*Large-Scale C++ Software Development: Process and Architecture*》（Addison-Wesley）書中有類似的主張並使用了 C++98。

19 是的，Ben 和 Jason，你們沒有看錯，我不會將所有事情視為 constexpr。參考 Ben Deane 和 Jason Turner 的文章「constexpr ALL the things」（*https://oreil.ly/Pazfb*），CppCon 2017。

指導原則 2：為改變而設計

對好軟體基本的期望之一是它具有容易改變的能力。這個期望甚至是**軟體**這個字的一部分。與**硬體**相比，軟體被期望能夠很容易地適應改變的需求（參考第 2 頁「指導原則 1：理解軟體設計的重要性」）。然而，從你自己的經驗中，你可能會知道，往往改變程式碼並不容易。相反地，有時候一個看似簡單的改變，結果會是一個星期的努力。

分離關注點

減少人為依賴性和簡化改變的最好和成熟的解決方案之一是分離關注點。這個想法的核心是分割、隔離或提取功能片段[20]：

> 將系統分解成小的、命名良好的、可理解的部分，能使工作更快。

分離關注點背後的目的是為了更好地理解和管理複雜性，因此設計出更加模組化的軟體。這個想法可能和軟體本身一樣老舊，因此被賦予許多不同的名稱。例如，同樣的想法被 Pragmatic Programmers 稱為「正交」[21]。他們建議分割軟體的正交面向。Tom DeMarco 稱它為**凝聚力**[22]：

> 凝聚力是模組內各元素關聯強度的測量標準。一個高凝聚力的模組是一個敘述和資料項目的集合，因為它們之間密切相關而應該被當作一個整體處理。任何將它們分割的企圖只會造成增加耦合並且降低可讀性。

這是 *SOLID* 原則[23]中最被廣泛接受的一組設計原則，這個想法被稱為**單一責任原則**（*SRP*）：

> 一個類別應該只有一個改變的理由[24]。

雖然這個概念很老舊，而且通常有很多名稱，但許多試圖說明分離關注點的嘗試都會引起比回答更多的問題。對 SRP 來說更是如此。單單這個設計原則的名稱本身就引起了一

20 Michael Feathers，《*Working Effectively with Legacy Code*》（Addison-Wesley，2013）。

21 David Thomas 和 Andrew Hunt，《*The Pragmatic Programmer: Your Journey to Mastery*》，20 週年紀念版（Addison Wesley，2019）。

22 Tom DeMarco，《*Structured Analysis and System Specification*》（Prentice Hall，1979）。

23 SOLID 是首字母的縮寫，是接下來幾個指導原則中所描述的五個原則的縮寫：SRP、OCP、LSP、ISP 和 DIP。

24 關於 SOLID 原則的第一本書是 Robert C. Martin 的《*Agile Software Development:Principles, Patterns, and Practices*》（Pearson）。另一本較新且較便宜的書也是 Robert C. Martin 的作品《*Clean Architecture*》（Addison-Wesley）。

些問題：什麼是責任？什麼是單一責任？為了澄清關於 SRP 的模糊性，一個常見的嘗試如下：

> 一切都應該只做一件事。

不幸的是，這種說明很難超越出模糊性。就像責任這個字並沒有太多的含義，只一件事也無助於讓它更容易理解。

不管名稱如何，想法總是相同的：將那些真正屬於一起的事情群組起來，將不是嚴格互屬的事情分開；或者換句話說：把那些因為不同原因而改變的事情分開。藉由這樣做，你程式碼不同面向之間的人為耦合會減少，並且它會幫助你使你的軟體更能適應改變。在最好的情況下，你可以在一個確切的位置改變你軟體的一個特定面向。

人為耦合的一個例子

讓我們藉由一個程式碼例子使分離關注點更容易理解。我的確有一個很好的例子。讓我為你展示抽象的 `Document` 類別：

```
//#include <some_json_library.h>  // 潛在的實體依賴性

class Document
{
 public:
   // ...
   virtual ~Document() = default;

   virtual void exportToJSON( /*...*/ ) const = 0;  ❶
   virtual void serialize( ByteStream&, /*...*/ ) const = 0;  ❷
   // ...
};
```

這看起來像是一個對所有文件類型都非常有用的基礎類別，不是嗎？首先，有一個 `exportToJSON()` 函數（❶）。為了從文件中產生 JSON 檔案（*https://oreil.ly/YWrsw*），所有衍生類別都必須實作 `exportToJSON()` 函數。這將被證明很有用：不需要知道是某種特定的文件（我們可以想像，我們最後會有 PDF 文件、Word 文件和更多其他的文件），我們總是可以匯出 JSON 的格式。很好！其次，有一個 `serialize()` 函數（❷），這個函數讓你透過 `ByteStream` 將 `Document` 轉換成位元組。你可以將這些位元組儲存在某些持久的系統中，像是檔案或資料庫。當然，我們可以期待還有許多其他的、有用的函數可以使用，這讓我們幾乎可以在任何事情上使用這個文件。

我看到你皺眉了。不，你看起來並沒有完全相信這是好的軟體設計。這可能是因為你只是很懷疑這個例子（它看起來好得不像是真的），也可能是因為你已經學習到這種設計最後會導致問題。你可能已經經歷過，用常見的物件導向設計原則來束縛資料和對操作它們的函數，可能很容易導致不幸的耦合。我同意：儘管這個基礎類別有看起來是很好的一體化套件的事實，甚至看起來它擁有我們可能需要的一切，但這種設計很快就會導致問題。

這是糟糕的設計，因為它包含了許多依賴性。當然，有一些明顯的、直接的依賴性，例如在 `ByteStream` 類別的依賴性。然而，這種設計也促成了引入人為的依賴性，這會使後續難以修改。在這種情況下，有三種人為的依賴性，其中兩種是由 `exportToJSON()` 函數所引入的，而另一種是由 `serialize()` 函數引入。

首先，`exportToJSON()` 需要在衍生類別中實作。是的，別無選擇，因為它是一個純虛擬函數（*https://oreil.ly/1u9at*）（用序列 `=0` 表示，即所謂的**純指定器**）。因為衍生類別很可能不想帶有手動實作 JSON 匯出的負擔，它們將依賴於外部、第三方的 JSON 函數庫：*json*（*https://oreil.ly/MqB03*）、*rapidjson*（*https://oreil.ly/jNMsz*）或 *simdjson*（*https://oreil.ly/5dBzC*）。不管你為此選擇什麼函數庫，因為 `exportToJSON()` 成員函數，將使衍生的文件突然依賴於這個函數庫。而且，很可能的，只是為了一致性的理念，所有的衍生類別會依賴於同一個函數庫。因此，衍生類別不是真正的獨立；它們被人為地耦合到一個特定的設計決定中[25]。還有，在特定 JSON 函數庫的依賴性肯定會限制階層結構的重用性，因為它不再是羽量級了。而且換到另一個函數庫將會造成重大的改變，因為所有衍生類別都必須調整[26]。

當然，`serialize()` 函數會引入同種類的人為依賴性。`serialize()` 很可能也會用像是 protobuf（*https://oreil.ly/z6Kgr*）或 Boost.serialization（*https://oreil.ly/ySJLk*）等第三方函數庫來實作。這相當程度地惡化了依賴性的情況，因為它引入了兩個正交的、不相關的設計面向（即 JSON 匯出和序列化）之間的耦合，一個面向的改變可能會造成另一個面向的改變。

在最壞的情況下，`exportToJSON()` 函數可能會引入第二個依賴性。在 `exportToJSON()` 呼叫中預期的引數，可能會意外地反映了所選 JSON 函數庫的一些實作細節。在這種情況下，最後換到另一個函數庫可能會造成 `exportToJSON()` 函數的簽章改變，這隨後將造成所有呼叫者的改變。因此，在所選 JSON 函數庫上的依賴性可能會意外地比預期的還要更廣泛。

---

25 不要忘記，藉由外部函數庫所做的設計決定可能會影響你自己的設計，這顯然會增加耦合。

26 這包括其他人可能撰寫的類別，也就是你無法控制的類別。而且，其他的人對這種改變也不會感到高興。因此，這個改變可能**真的**很難。

第三種依賴性是由 serialize() 函數引入的。由於這個函數，從 Document 衍生的類別依賴於文件如何被序列化的全域決策。我們使用什麼格式？我們是使用小的位元組順序還是大的位元組順序？我們是否必須添加位元組表示 PDF 檔案或 Word 檔案的資訊？如果是的話（我認為這非常有可能），我們要如何表示這樣的文件？是否借助於一個整數值？例如，我們可以為這個目的使用一個列舉 [27]：

```
enum class DocumentType
{
   pdf,
   word,
   // ... 可能有更多的文件類型
};
```

對序列化這種方法非常常見。然而，如果在 Document 類別的實作中使用這種低階的文件表示法，我們會意外地結合所有不同種類的文件。每個衍生類別都會隱含地知道所有其他的 Document 類型。因此，增加一種新的文件種類會直接影響所有現存的文件類型。這將是一個嚴重的設計缺陷，再一次，因為這將使改變更困難。

不幸的是，Document 類別促成了許多不同種類的耦合。所以不行，Document 類別不是一個好類別設計的例子，因為它不容易改變。相反地，它很難改變，因此是違反 SRP 的很好例子：從 Document 衍生的類別和 Document 類別的使用者有很多原因改變，因為我們已經在一些正交的、不相關的面向之間建立了強大的耦合。總之，衍生類別和文件的使用者可能會因為以下的任何原因而改變：

- exportToJSON() 函數的實作細節因為與所用的 JSON 函數庫上的直接依賴關係而改變
- exportToJSON() 函數的簽章因為底層實作改變而改變
- Document 類別和 serialize() 函數因為與 ByteStream 類別上的直接依賴關係而改變
- serialize() 函數的實作細節因為與實作細節的直接依賴關係而改變
- 所有類型的文件都因為與 DocumentType 列舉上的直接依賴關係而改變

很明顯地，這種設計促進了更多的改變，而且每一個改變都很難。當然，在一般的情況下，還有額外的正交面向被人為地在文件內耦合的危險，這將進一步增加做出改變的複雜性。另外，這些改變中的部分絕對不會侷限在程式碼庫中的某一個地方。特別是，對 exportToJSON() 和 serialize() 實作細節的改變不會只侷限於一個類別，而可能會發生在所有種類的文件（PDF、Word 等）。因此，一個改變會影響到整個程式碼庫的大部分地方，這就引起了維護的風險。

27 列舉似乎是一個明顯的選擇，但當然還有其他的選擇。最後，我們需要一組已經取得共識的數值，這些數值代表以位元組表示的不同文件格式。

邏輯耦合相對於實體耦合

耦合並不限於邏輯耦合，也會擴展到實體耦合，圖 1-2 說明了這種耦合。讓我們假設在我們架構的低層次有一個 User 類別，它需要使用放在架構較高層次的文件。當然，User 類別直接依賴於 Document 類別，這是一個必要的依賴性——一個給定問題的固有依賴性。因此，它不應該是我們所擔憂的。然而，Document 對所選 JSON 函數庫的（潛在的）實體依賴關係和在 ByteStream 類別的直接依賴關係，造成了 User 對 JSON 函數庫和 ByteStream 間接的、過渡的依賴關係，這些都放在我們架構的最高層次。在最壞的情況下，這意味著對 JSON 函數庫或 ByteStream 類別的改變會對 User 有影響。希望這是很容易可以看出是一種人為的，而不是有意的依賴性：User 不應該依賴 JSON 或序列化。

我應該明確地指出，Document 在所選擇的 JSON 函數庫存有潛在的實體依賴性。如果 <Document.h> 標頭檔包括任何來自所選擇 JSON 函數庫的標頭檔（如第 11 頁「人為耦合的一個例子」開始程式碼片段所示），例如因為 exportToJSON() 函數期待某些基於這函數庫的引數，那麼在這函數庫上就有明顯的依賴性。但是，如果介面可以適當地從這些細節中抽取出來，而且 <Document.h> 標頭檔不包括任何來自 JSON 函數庫的事物，那這實體依賴性就有可能避免。因此，這取決於依賴性抽象化的程度。

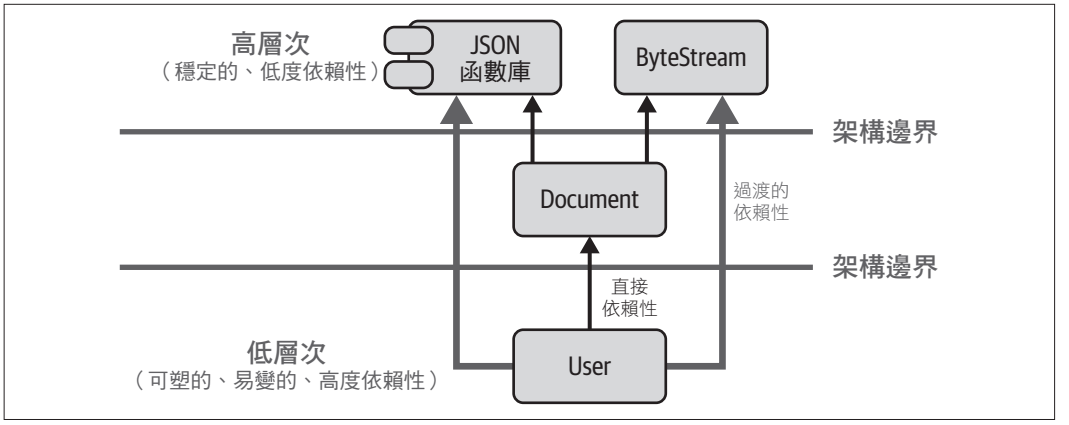

圖 1-2　User 與像 JSON 和序列化的正交面向之間強烈的過渡、實體的耦合

「高層次，低層次——現在我很困惑，」你發出抱怨。是的，我知道這兩個術語通常會造成一些困惑。所以，在我們繼續之前，讓我們對高層次和低層次的術語取得一致意見。這兩個術語的起源與我們在**統一模組化語言**（*UML*）（*https://oreil.ly/s0ID2*）中畫圖的方式有關：我們認為穩定的功能出現在頂部，在高層次上；經常變化的功能，因此被認為是易變的或可塑的，出現在底部，也就是低層次。不幸的是，當我們繪製架構時，我們經常試圖顯示事物是如何相互依存的，所以最穩定的部分會出現在架構的底部。當然，這也造成了一些困惑。不管事物是如何畫出，只需要記住這些術語：**高層次**指的是你架構中穩定的部分，而**低層次**指的是那些經常改變或更可能改變的面向。

回到問題上：SRP 建議我們應該把關注點和不是真正屬於的事物，也就是非黏性的（有黏性的）事物分開。換句話說，它建議我們將因為不同原因而改變的事物分離成**變動點**。圖 1-3 顯示了如果我們將 JSON 和序列化面向隔離到分離的關注點時的耦合情況。

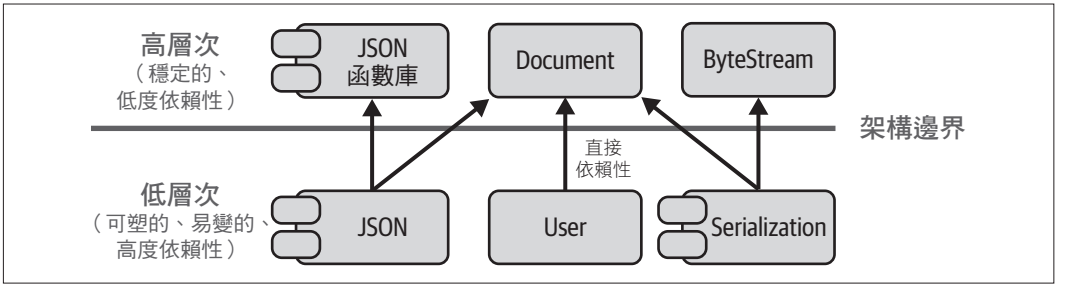

圖 1-3　遵守 SRP 以解決 User、JSON 和序列化之間的人為耦合

基於這個建議，`Document` 類別用以下的方式重構：

```
class Document
{
 public:
   // ...
   virtual ~Document() = default;

   // 不再有「exportToJSON()」和「serialize()」函數。
   // 只有非常基本的文件操作，這不會
   // 引起強烈耦合，其餘不變。
   // ...
};
```

JSON 和序列化面向只是非 Document 類別基本的功能部分。Document 類別應該僅僅表示不同種類文件的最基本操作。所有正交的面向都應該被分開。這將使改變相當容易。例如，透過將 JSON 面向隔離到一個分離的變動點和新的 JSON 組件，那從一個 JSON 函數庫換到另一個將只影響這一個組件。這種改變可以在一個地方完成，並且與所有其他的、正交的面向隔離發生。而藉由一些 JSON 函數庫也比較容易支援 JSON 的格式。另外，文件如何序列化的任何改變將只影響程式碼中的一個組件：新的 Serialization 組件。還有，Serialization 將充當一個能夠隔離的、容易改變的變動點，這將是最佳的情況。

在你最初對 Document 的例子感到失望之後，我可以再次看到你顯得較為高興。也許你臉上甚至有「我就知道！」的笑容。然而，你還沒有完全地滿意。「是的，我同意分離關注點的一般想法。但是，我必須如何建構我的軟體來分離關注點呢？我必須做什麼以使它有作用？」這是一個很好的問題，但這問題有很多答案，我將在接下來的章節中討論。然而，第一也是最重要的一點是識別一個變動點，也就是在你的程式碼中預期會改變的某些面向。這些變動點應該被提取、隔離和包裝起來，這樣在這些變動點上就不再有任何依賴性，這最後將有助於使改變更容易。

「但這仍然只是表面上的建議！」我聽到你說，而你是對的。不幸的是，沒有單一的答案，而且沒有簡單的答案。這要視情況而定，但我承諾會在後續的章節中提供許多關於如何分離關注點的具體答案。畢竟，這是一本軟體設計的書，也就是說，是一本管理依賴性的書。作為一個小小的預告，在第 3 章，我將對這個問題介紹一個一般和實用的方法：設計模式。在這種想法下，我將顯示如何用不同的設計模式來分離關注點。例如，我想到了 *Visitor*、*Strategy* 和 *External Polymorphism* 等設計模式。所有這些模式都有不同的強處和弱點，但它們會引入某種抽象化以幫助你減少依賴性的共同特性。此外，我承諾將仔細觀察在現代 C++ 中如何實作這些設計模式。

我將在第 107 頁的「指導原則 16：用 Visitor 來擴展操作」中介紹 Visitor 設計模式，在第 134 頁的「指導原則 19：用 Strategy 來隔離事物如何完成」中介紹 Strategy 設計模式。External Polymorphism 設計模式將是第 271 頁的「指導原則 31：為非干擾性執行期使用 External Polymorphism」的主題。

不要重複自己

可改變性還有第二個重要的面向。為了說明這個面向，我將介紹另一個例子：一個專案的階層結構。圖 1-4 提供了這種階層結構的圖像。