.NET 相依性注入

使用Unity

DEPENDENCY INJECTION

蔡煥麟/著

.NET 相依性注入

使用 Unity

Michael Tsai

這本書的網址是 http://leanpub.com/dinet

此版本發布於 2019-10-16

ISBN 9789574320684

Contents

序

老實說，我寫程式時也不總是遵循最佳實務與設計原則；我也會因為趕時間而「姑息養蟲」，或者未加思索地把一堆實作類別綁得很緊，造成日後維護的麻煩。的確，當我們不知道問題在哪裡，自然也就不容易發覺哪些寫法是不好的，以及當下能夠用什麼技術來解決，以至於技術債越揹越多。在學習 Dependency Injection（以下簡稱 DI）的過程中，我覺得身上逐漸多了一些好用的武器裝備，可用來改善軟體設計的品質，這感覺真不錯。於是，我開始有了念頭，把我理解的東西比較有系統地整理出來，而結果就是您現在看到的這本書。

撰寫本書的過程中，.NET 技術平台陸續出現一些新的消息。其中一則令人矚目的頭條新聞，便是下一代的 ASP.NET 框架——ASP.NET vNext ——將會更全面地支援 Dependency Injection 技術，其中包括一個 DI 抽象層與一些轉換器（adapters）類別來支援目前幾種常見的 DI 容器，例如 Unity、Autofac、Ninject、Structuremap、Windsor 等等。

雖然 ASP.NET vNext 離正式版本發布還有一段時間，將來的實作規格仍有變數，但我們至少可以確定一件事：DI 技術在未來的 .NET 平台已愈形重要，可說是專業開發人員的一項必備技能了。故不惴淺薄，希望本書能夠適時在 DI 技術方面提供一些學習上的幫助；另一方面，則拋磚引玉，盼能獲得各方高手指教，分享寶貴的實戰心得。

簡單地說，「寫自己想看的書」，我是抱持著這樣的想法來寫這本書。希望裡面也有您想要的東西。

蔡煥麟 於台灣新北市
2014 年 7 月

致謝

寫書是很私人的事情，感謝家人的支持與忍耐。

感謝「恆逸（UCOM）」的前輩們，特別是 Richard、Vivid、Adams、Anita、John、Jerry 等 .NET 講師。少了在恆逸講台上的那些日子，大概也就不會有這本書。

從本書發行 beta 版開始就支持本書的讀者，感謝你們的大力支持！也謝謝曾經在部落格、社群網站上留言鼓勵我寫書（推坑？）的朋友們，像是 91、Cash、IT Player 等等（無法一一羅列，請自行對號入座）。雖然很少外出參加活動、與人交流，但我想，寫作也算是一種不錯的交流方式吧。

本書有許多寫作靈感來自於 Mark Seemann 的《Dependency Injection in .NET》，儘管他不知道我，仍要謝謝他。

關於本書

本書內容是關於 .NET 相依性注入（Dependency Injection，簡稱 DI）程式設計的相關議題。透過本書，您將會了解：

- 什麼是 DI、它有什麼優點、何時使用 DI、以及可能碰到的陷阱。
- 如何運用 DI 應付容易變動的軟體需求，設計出更彈性、更好維護的程式碼。
- 與 DI 有關的設計模式。
- DI 於 .NET 應用程式中的實務應用（如 ASP.NET MVC、ASP.NET WEB API、WCF 等等）。
- 如何在應用程式中使用現成的 DI 框架來協助實現 DI。本書支援的 DI 框架主要是 Unity，部分章節有提供 Autofac 的範例（如第 5 章、第 6 章）。

誰適合閱讀本書

這不是 .NET 程式設計的入門書。以下是閱讀本書的基本條件：

- 熟悉 C# 語法，包括擴充方法（extrension methods）、泛型（generics）、委派（delegates）等等。如果精通的是 VB（或其他 .NET 語言）但不排斥 C# 語法，也是 OK 的。
- 具備物件導向的基礎概念，知道何謂封裝、繼承、多型（polymorphism）。

倒不是說，不符合以上條件的讀者就無法從本書汲取有用的東西；只是就一般的情況而言，讀起來會比較辛苦一些。

如果您曾經接觸、研究過設計模式（design patterns），有些章節閱讀起來會輕鬆一些。然而這並非基本要求，因為本書也會一併介紹相關的設計模式與原則，例如 **Decorator 模式**、**Factory 模式**、開放／封閉原則（**Open/Closed Principle**）、單一責任原則（**Single Responsibility Principle**）等等。

此外，如果下列描述有一些符合您現在的想法，那麼本書也許對您有幫助：

- 我的日常開發工作需要設計共用的類別庫或框架，供他人使用。
- 我經常使用第三方（third-party）元件或框架，而且它們都提供了某種程度的 DI 機制。為了充分運用這些機制，我必須了解 DI 的各種用法。
- 我希望能夠寫出寬鬆耦合、容易維護的程式碼。
- 我已經開始運用寬鬆耦合的設計原則來寫程式，我想知道更多有關 DI 的細節，以了解有哪些陷阱和迷思，避免設計時犯了同樣的毛病。
- 我正在開發 ASP.NET MVC 或 ASP.NET Web API 應用程式，想要了解如何運用 DI 技術來改善我的應用程式架構。

無論如何，自己讀過的感覺最準。建議您先讀完本書的試閱章節（包含本書第 1 章完整內容），以評估這本書是否適合你。

如何閱讀本書

您並不需要仔細讀完整本書才能得到你需要的 DI 觀念與實務技巧。如果不趕時間，我的建議是按本書的章節順序仔細讀完「基礎篇」（一至三章）。讀完前面三章，掌握了相關基礎概念和常用術語之後，接下來的「實戰篇」與「工具篇」，則可依自己的興趣來決定閱讀順序。

舉例來說，如果你希望能夠盡快學會使用 Unity 框架的各項功能，則可嘗試在讀過第一和第二章以後，直接跳到「工具篇」。等到將 DI 技術實際應用於日常開發工作時，便可能會需要知道如何運用 .NET Framework 提供的 DI 機制，屆時本書「實戰篇」的內容也許就有你需要的東西。

在撰寫第七章〈Unity 學習手冊〉時，我大多採取碰撞式寫法，讓一個主題帶出另一個主題。因此，即使是介紹 Unity 框架的用法，應該也能夠依序閱讀，而不至於像啃 API 參考文件那般枯燥。

無論使用哪一種方法來閱讀本書，有一項功課是絕對必要的，那就是：動手練習。碰到沒把握的範例程式或有任何疑惑時，最好都能開啟 Visual Studio，親手把程式碼敲進去，做點小實驗，學習效果肯定更好。

書寫慣例

技術書籍免不了夾雜一堆英文縮寫和不易翻譯成中文的術語，而且有些術語即使譯成中文也如隔靴搔癢，閱讀時反而會自動在腦袋裡轉成英文來理解。因此，對於這些比較麻煩的術語，除了第一次出現時採取中英並呈，例如「服務定位器」（Service Locator），往後再碰到相同術語時，我傾向直接使用英文——這並非絕對，主要還是以降低閱讀阻力為優先考量。

書中不時會穿插一些與正文有關的補充資料，依不同性質，有的是以單純加框的側邊欄（sidebar）圈住，有的則會佐以不同的圖案。底下是書中常用的幾個圖案：

此區塊會提示當前範例原始碼的所在位置。

注意事項。

相關資訊。

通常是筆者的碎碎念、個人觀點（不見得完全正確或放諸四海皆準），或額外的補充說明。

需要準備的工具

本書範例皆以 C# 寫成，使用的開發工具是 Visual Studio 2017。為了能夠一邊閱讀、一邊練習，您的 Windows 作業環境至少需要安裝下列軟體：

- .NET Framework 4.5
- Visual Studio 2015 Community 或 Professional 以上的版本

範例程式與補充資料

本書的完整範例程式與相關補充資料都放在 github 網站上。網址如下：

https://github.com/huanlin/di-book-support

版本更新紀錄

2019 年 8 月

- 第 7 章：針對程式碼太長而超出 PDF 文件邊界的地方調整版面編排。

2018 年 9 月

- 第 6 章與第 7 章：針對 Unity 套件從 v3.x 升級至目前最新版本（v5.8.11）而修改一些過時的文字敘述和程式碼。主要是 Unity 相關組件以及 namespace 的名稱，以及少數的屬性與方法名稱。
- 第 7 章：補強 Unity 的〈攔截〉一節的內容。
- 第 7 章：範例程式專案的套件參考形式由 packages.config 改為使用 PackageReference。

2018 年 8 月

- 修正第 1 章：原「對開放擴充」，改為「對擴充開放」。
- 修正第 7 章與第 8 章錯誤（感謝 Allen Kuo 大大）：

  ```
      - EmailService：透過電子郵件發送訊息
      - SmsService：透過簡訊服務發送訊息

    以上三個類別...
  ```

  應為「以上**兩**個類別...」
- 更新範例原始碼，確保所有範例專案所參考的套件是最新版，且都能夠以 Visual Studio 2017 編譯（感謝 Allen Kuo 大大的提醒）。

2014 年 12 月

2014/12/8 正式發布初版。

2014 年 7 月

2014/7/7 首次對外發布 beta 版。

關於作者

.NET 程式設計師，現任 C# MVP（2007 年至今），有幸曾站在恆逸講台上體會「好為人師」的滋味，也翻譯過幾本書。

著作：

書名	初版／更新日期	網址
Leanpub 自出版實戰	2018 年 9 月	https://leanpub.com/selfpub
C# 本事	2018 年 9 月	https://leanpub.com/csharp-kungfu
.NET 非同步程式設計	2015 年	https://leanpub.com/dotnet-async
.NET 相依性注入	2018 年 9 月	https://leanpub.com/dinet

除了 leanpub.com，上列書籍大多也有上架至其他線上書店，包括 Google Play 圖書、讀墨、Kobo 等等。

陳年譯作：

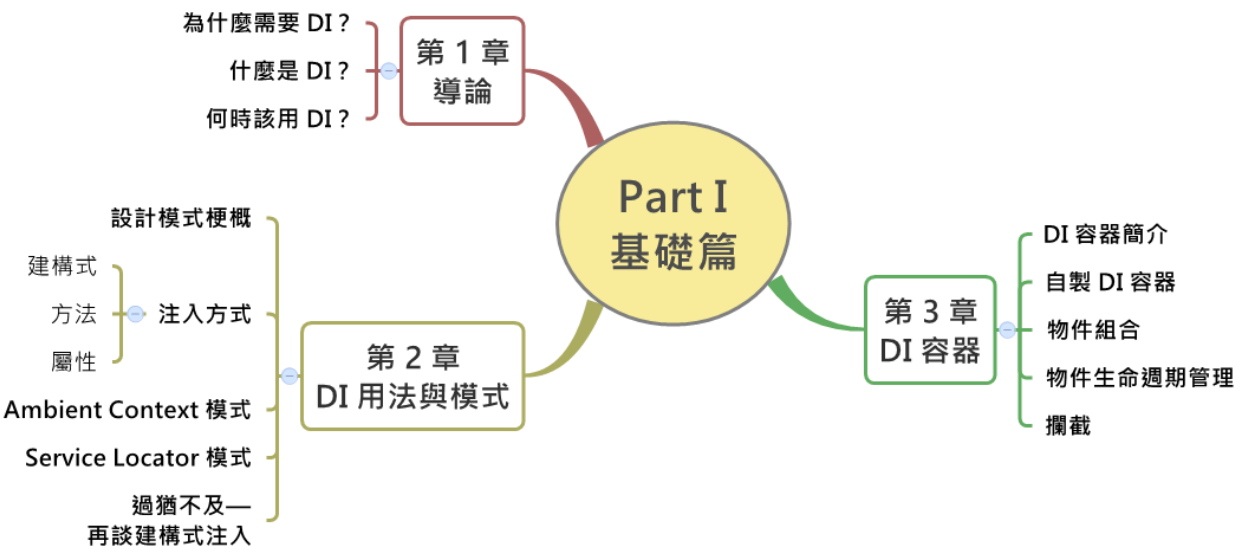
- 《軟體構築美學》，2010。原著：Brownfield Application Development in .NET。
- 《物件導向分析設計與應用》[1]，2009。原著：Object-Oriented Analysis and Design with Applications 3rd Edition。
- 《ASP.NET AJAX 經典講座》，2007。原著：Introducing Microsoft ASP.NET AJAX。
- 《微軟解決方案框架精要》，2007。原著：Microsoft Solution Framework Essentials。
- 《軟體工程與 Microsoft Visual Studio Team System》，2006。原著：Software Engineering and Mcirosoft Visual Studio Team System。

[1] http://www.books.com.tw/products/0010427868

您可以透過下列管道得知作者的最新動態：

- 作者部落格：https://www.huanlintalk.com
- Facebook 專頁：https://www.facebook.com/huanlin.notes
- Twitter：https://twitter.com/huanlin

Part I：基礎篇


Part I
基礎篇
第 1 章
導論
為什麼需要 DI？
什麼是 DI？
何時該用 DI？
第 2 章
DI 用法與模式
設計模式梗概
注入方式
建構式
方法
屬性
Ambient Context 模式
Service Locator 模式
過猶不及—
再談建構式注入
第 3 章
DI 容器
DI 容器簡介
自製 DI 容器
物件組合
物件生命週期管理
攔截

第 1 章：導論

本章從一個基本的問題開始，點出軟體需求變動的常態，以說明為什麼我們需要學習「**相依性注入**」（**Dependency Injection；簡稱 DI**）來改善設計的品質。接著以一個簡單的入門範例來比較沒有使用 DI 和改寫成 DI 版本之後的差異，並討論使用 DI 的時機。目的是讓讀者先對相關的基礎概念有個概括的理解，包括可維護性（maintainability）、寬鬆耦合（loose coupling）、控制反轉（inversion of control）、動態繫結、單元測試等等。

內容大綱：

本章範例原始碼位置：

`https://github.com/huanlin/di-book-support` 裡面的 Examples/ch01 資料夾。

若您從未接觸過 DI，建議精讀並且動手練習書中的範例。

為什麼需要 DI？

或許你也曾在某個網路論壇上看過有人提出類似問題：「如何利用字串來來建立物件？」

欲了解此問題的動機與解法，得先來看一下平常的程式寫法可能產生什麼問題。一般來說，建立物件時通常都是用 new 運算子，例如：

```
ZipCompressor obj = new ZipCompressor();
```

上面這行程式碼的作用是建立一個 ZipCompressor 物件，或者說，建立 ZipCompressor 類別的執行個體（instance）。從類別名稱不難看出，ZipCompressor 類別會提供壓縮資料的功能，而且是採用 Zip 壓縮演算法。假如有一天，軟體系統已經部署至用戶端，後來卻因為某種原因無法再使用 ZipCompressor 了（例如發現它有嚴重 bug 或授權問題），必須改用別的類別，比如說 RarCompressor 或 GZip。那麼，專案中所有用到 ZipCompressor 的程式碼全都必須修改一遍，並且重新編譯、測試，導致維護成本太高。

為了降低日後的維護成本，我們可以在設計軟體時，針對「將來很可能需要替換的元件」，在程式中預留適度的彈性。簡單地說，就是一種應付未來變化的設計策略。

回到剛才的範例，如果改寫成這樣：

```
var className = ConfigurationManager.AppSettings["CompressorClassName"];
Type aType = Type.GetType(className);
ICompressor obj = (ICompressor) System.Activator.CreateInstance(aType);
```

亦即建立物件時，是先從應用程式組態檔中讀取欲使用的類別名稱，然後透過 Activator.CreateInstance 方法來建立該類別的執行個體，並轉型成各壓縮器共同實作的介面 ICompressor。於是，我們就可以將類別名稱寫在組態檔中：

```
<appSettings>
    <add key="CompressorClassName" value="MyLib.ZipCompressor, MyLib" />
<appSettings>
```

將來若需要換成其他壓縮器，便無須修改和重新編譯程式碼，而只要修改組態檔中的參數值，就能切換程式執行時所使用的類別，進而達到改變應用程式行為的目的。這裡使用了動態繫結的技巧。

> **動態繫結**
>
> 動態繫結又稱為晚期繫結（late binding），指的是程式執行時才決定物件的真正型別，而非在編譯時期決定（靜態繫結）。舉例來說，瀏覽器的附加元件（add-ins）就是一種晚期繫結的應用。

舉這個例子，重點不在於「以字串來建立物件」的程式技巧，而是想要點出一個問題：當我們在寫程式時，可能因為很習慣使用 new 運算子而不經意地在程式中加入太多緊密的相依關係——即**「相依性」（dependency）**。進一步說，每當我們在程式中使用 new 來建立第三方（third party）元件的執行個體，我們的程式碼就在編譯時期跟那個類別固定綁（bind）在一起了；這層相依關係有可能是單向依賴，也可能是彼此相互依賴，形成更緊密的**「耦合」（coupling）**，增加日後維護程式的困難。

可維護性

就軟體系統而言，**「可維護性」（maintainability）**指的是將來修改程式時需要花費的工夫；如果改起來很費勁，我們就說它是很難維護的、可維護性很低的。

有軟體開發實務經驗的人應該會同意：軟體需求的變動幾乎無可避免。如果你的程式碼在完成第一個版本之後就不會再有任何更動，自然可以不用考慮日後維護的問題。但這種情

況非常少見。實務上，即使剛開始只是個小型應用程式，將來亦有可能演變成大型的複雜系統；而最初看似單純的需求，在實作完第一個版本之後，很快就會出現新的需求或變更既有規格，而必須修改原先的設計。這樣的修改，往往不只是改動幾個函式或類別而已，還得算上重新跑過一遍完整測試的成本。這就難怪，修改程式所衍生的整體維護成本總是超出預期；這也難怪，軟體系統交付之後，開發團隊都很怕客戶改東改西。

此外，我想大多數人都是比較喜歡寫新程式，享受創新、創造的過程，而少有人喜歡接手維護別人的程式，尤其是難以修改的程式碼。然而，程式碼一旦寫完，某種程度上它就已經算是進入維護模式了[2]。換言之，程式碼大多是處於維護的狀態。既然如此，身為開發人員，我們都有責任寫出容易維護的程式碼，讓自己和別人的日子好過一些。就如 Damian Conway 在《Perl Best Practices》書中建議的：

> 「寫程式時，請想像最後維護此程式的人，是個有暴力傾向的精神病患，而且他知道你住哪裡。」

寬鬆耦合

在 .NET （或某些物件導向程式語言）的世界裡，任何東西都是「物件」，而應用程式的各項功能便是由各種物件彼此相互合作所達成，例如：物件 A 呼叫物件 B，物件 B 又去呼叫 C。像這樣由類別之間相互呼叫而令彼此有所牽連，便是耦合（coupling）。物件之間的關係越緊密，耦合度即越高，程式碼也就越難維護；因為一旦有任何變動，便容易引發連鎖反應，非得修改多處程式碼不可，導致維護成本提高。

為了提高可維護性，一個常見且有效的策略是採用**「寬鬆耦合」（loose coupling）**，亦即讓應用程式的各部元件適度隔離，不讓它們彼此綁得太緊。一般而言，軟體系統越龐大複雜，就越需要考慮採取寬鬆耦合的設計方式。

當你在設計過程中試著落實寬鬆耦合原則，剛開始可能會看不太出來這樣的設計方式有什麼好處，反而會發現要寫更多程式碼，覺得挺麻煩。但是當你開始維護這些既有的程式，

[2]《Brownfield Application Development》第 7 章。

你會發現自己修正臭蟲的速度變快了，而且那些類別都比以往緊密耦合的寫法要來得更容易進行獨立測試。此外，修改程式所導致的「牽一髮動全身」的現象也可能獲得改善，因而降低你對客戶需求變更的恐懼感。

基本上，「可維護性」與「寬鬆耦合」便是我們學習 Dependency Injection 的主要原因。不過，這項技術還有其他附帶好處，例如有助於單元測試與平行開發，這裡也一併討論。

可測試性

> I've focused almost entirely on the value of Dependency Injection for unit testing and I've even bitterly referred to using DI as "Mock Driven Design." That's not the whole story though.[3] ——Jeremy Miller
>
> （對於 Dependency Injection 所帶來的價值，我把重點幾乎全擺在單元測試上面，有人甚至挖苦我是以「模仿驅動設計」的方式來使用 DI。但那只是事實的一部分而已。）

當我們說某軟體系統是「可測試的」（testable），指的是有**「單元測試」（unit test）**，而不是類似用滑鼠在視窗或網頁上東點西點那種測試方式。單元測試有「寫一次，不斷重複使用」的優點，而且能夠利用工具來自動執行測試。不過，撰寫單元測試所需要付出的成本也不低，甚至不亞於撰寫應用程式本身。有些方法論特別強調單元測試，例如**「測試驅動開發」（Test-Driven Development；TDD）**，它建議開發人員養成先寫測試的習慣，並盡量擴大單元測試所能涵蓋的範圍，以達到改善軟體品質的目的。

有些情況，要實施「先寫測試」的確有些困難。比如說，有些應用程式是屬於分散式多層架構，其中某些元件或服務需要運行於遠端的數台伺服器上。此時，若為了先寫測試而必須先把這些服務部署到遠端機器上，光是部署的成本與時間可能就讓開發人員打退堂鼓。像這種情況，我們可以先用**「測試替身」（test doubles）**來暫時充當真正的元件；如此

[3] The Dependency Injection Pattern –What is it and why do I care? by Jeremy Miller

一來，便可以針對個別模組進行單元測試了。Dependency Injection 與寬鬆耦合原則在這裡也能派上用場，協助我們實現測試替身的機制。

平行開發

分而治之，是對付複雜系統的一個有效方法。實務上，軟體系統也常被劃分成多個部分，交給多名團隊成員同時分頭進行各部元件的開發工作，然後持續進行整合，將它們互相銜接起來，成為一個完整的系統。要能夠做到這點，各部分的元件必須事先訂出明確的介面，就好像插座與插頭，將彼此連接的介面規格先訂出來，等到各部分實作完成時，便能順利接上。DI 的一項基本原則就是「**針對介面寫程式**」（**program to an interface**），而此特性亦有助於團隊分工合作，平行開發。

了解其優點與目的之後，接著要來談談什麼是 **Dependency Injection**。

什麼是 DI？

如果說「容易維護」是設計軟體時的一個主要品質目標，「寬鬆耦合」是達成此目標的戰略原則，那麼，**「相依性注入」（dependency injection；DI）**就是屬於戰術層次；它包含一組設計模式與原則，能夠協助我們設計出更容易維護的程式。

DI 經常與**「控制反轉」（Inversion of Control；簡稱 IoC）**相提並論、交替使用，但兩者並不完全相等。比較精確地說，IoC 涵蓋的範圍比較廣，其中包含 DI，但不只是 DI。換個方式說，DI 其實是 IoC 的一種形式。那麼，IoC 所謂的控制反轉，究竟是什麼意思呢？反轉什麼樣的控制呢？如何反轉？對此問題，我想先引用著名的軟體技術問答網站 Stack Overflow 上面的一個妙答，然後再以範例程式碼來說明。

該帖的問題是：「如何向五歲的孩童解釋 DI？[4]」在眾多回答中，有位名叫 John Munch 的仁兄假設提問者就是那五歲的孩童而給了如下答案：

[4] http://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old

> 當你自己去開冰箱拿東西時，很可能會闖禍。你可能忘了關冰箱門、可能會拿了爸媽不想讓你碰的東西，甚至冰箱裡根本沒有你想要找的食物，又或者它們早已過了保存期限。
>
> 你應該把自己需要的東西說出來就好，例如：「我想要一些可以搭配午餐的飲料。」然後，當你坐下用餐時，我們會準備好這些東西。

如此精準到位的比喻，網友一致好評。

接下來，依慣例，我打算用一個 Hello World 等級的 DI 入門範例來說明。常見的 Hello World 範例只有短短幾行程式碼，但 DI 沒辦法那麼單純；即使是最簡單的 DI 範例，也很難只用兩三行程式碼來體現其精神。因此，接下來會有更多程式碼和專有名詞，請讀者稍微耐心一點。我們會先實作一個非 DI 的範例程式，然後再將它改成 DI 版本。

為了避免文字敘述過於冗長，往後將盡量以縮寫 **DI** 來代表「相依性注入」，以 **IoC** 來代表「控制反轉」。

入門範例—非 DI 版本

這裡使用的範例情境是應用程式的登入功能必須提供雙因素驗證（two-factor authentication）機制，其登入流程大致有以下幾個步驟：

1. 使用者輸入帳號密碼之後，系統檢查帳號密碼是否正確。
2. 帳號密碼無誤，系統會立刻發送一組隨機驗證碼至使用者的信箱。
3. 使用者收信，獲得驗證碼之後，回到登入頁面繼續輸入驗證碼。
4. 驗證碼確認無誤，讓使用者登入系統。

依此描述，我們可以設計一個類別來提供雙因素驗證的服務：`AuthenticationService`。底下是簡化過的程式碼：

```
class AuthenticationService
{
    private EmailService msgService;

    public AuthenticationService()
    {
        msgSevice = new EmailService(); // 建立用來發送驗證碼的物件
    }

    public bool TwoFactorLogin(string userId, string pwd)
    {
        // 檢查帳號密碼，若正確，則傳回一個包含使用者資訊的物件。
        User user = CheckPassword(userId, pwd);
        if (user != null)
        {
            // 接著發送驗證碼給使用者，假設隨機產生的驗證碼為 "123456"。
            this.msgService.Send(user, " 您的登入驗證碼為 123456");
            return true;
        }
        return false;
    }
}
```

AuthenticationService 的建構函式會建立一個 EmailService 物件，用來發送驗證碼。TwoFactorLogin 方法會檢查使用者輸入的帳號密碼，若正確，就呼叫 EmailService 物件的 Send 方法來將驗證碼寄送至使用者的 e-mail 信箱。EmailService 的 Send 方法就是單純發送電子郵件而已，這部分的實作細節並不重要，故未列出程式碼；CheckPassword 方法以及 User 類別的程式碼也是基於同樣的理由省略（User 物件會包含使用者的基本聯絡資訊，如 e-mail 位址、手機號碼等等）。

主程式的部分則是利用 AuthenticationService 來處理使用者登入程序。這裡用一個簡化過的 MainApp 類別來表示，程式碼如下，我想應該不用多做解釋。

```
class MainApp
{
    public void Login(string userId, string password)
    {
        var authService = new AuthenticationService();
        if (authService.TwoFactorLogin(userId, password))
        {
            if (authService.VerifyToken(" 使用者輸入的驗證碼"))
            {
                // 登入成功。
            }
        }
        // 登入失敗。
    }
}
```

此範例目前涉及四個類別：MainApp、AuthenticationService、User、EmailService。它們的相依關係如圖 1-1 所示，圖中的箭頭代表相依關係的依賴方向。

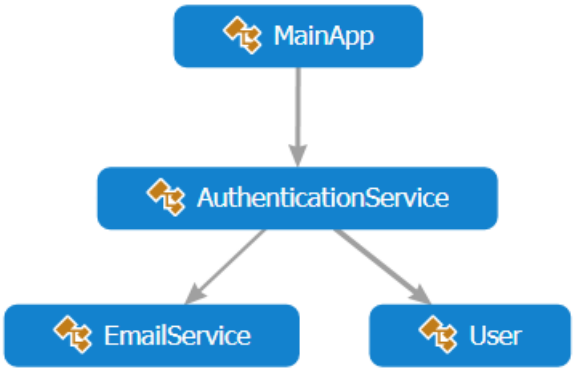


圖 1-1：類別相依關係圖

透過這張圖可以很容易看出來，代表主程式的 MainApp 類別需要使用 Authentication-Service 提供的驗證服務，而該服務又依賴 User 和 EmailService 類別。就它們之間的角色關係來說，AuthenticationService 對 MainApp 而言是個「服務端」（service）的角色，對於 User 和 EmailService 而言則是「用戶端」（client；有時也說「呼叫端」）的角色。

目前的設計，基本上可以滿足功能需求。但有個問題：萬一將來使用者想要改用手機簡訊來接收驗證碼，怎麼辦？稍後你會看到，此問題凸顯了目前設計上的一個缺陷：它違反了「開放／封閉原則」。

開放／封閉原則

「開放／封閉原則」（Open/Close Principle；OCP）指的是軟體程式的單元（類別、模組、函式等等）應該要夠開放，以便擴充功能，同時要夠封閉，以避免修改既有的程式碼。換言之，此原則的目的是希望能在不修改既有程式碼的前提下增加新的功能。須注意的是，遵循開放／封閉原則通常會引入新的抽象層，使程式碼不易閱讀，並增加程式碼的複雜度。在設計時，應該將此原則運用在將來最有可能變動的地方，而非試圖讓整個系統都符合此原則。

一般公認最早提出 **OCP** 的是 Bertrand Meyer，後來由 Robert C. Martin（又名鮑伯[Uncle Bob]）重新詮釋，成為目前為人熟知的物件導向設計原則之一。鮑伯在他的《Agile Software Development: Principles, Patterns, and Practices》書中詳細介紹了五項設計原則，並且給它們一個好記的縮寫：**S.O.L.I.D.**。它們分別是：

- **SRP**（Single Responsibility Principle）：單一責任原則。一個類別應該只有一個責任。
- **OCP**（Open/Closed Principle）：開放／封閉原則。對擴充開放，對修改封閉。
- **LSP**（Liskov Substitution Principle）：里氏替換原則。物件應該要可以被它的子類別的物件替換，且完全不影響程式的既有行為。
- **ISP**（Interface Segregation Principle）：介面隔離原則。多個規格明確的小介面要比一個包山包海的大型介面好。
- **DIP**（Dependency Inversion Principle）：相依反轉原則。依賴抽象型別，而不是具象型別。

後續章節若再碰到這些原則，將會進一步說明。您也可以參考剛才提到的書籍，繁體中文版書名為《敏捷軟體開發：原則、樣式及實務》。出版社：碁峰。譯者：林昆穎、吳京子。

針對「使用者想要改用手機簡訊來接收驗證碼」的這個需求變動，一個天真而快速的解法，是增加一個提供發送簡訊服務的類別：ShortMessageService，然後修改 AuthenticationService，把原本用到 EmailService 的程式碼換成新的類別，像這樣：

```
class AuthenticationService
{
    private ShortMessageService msgService;

    public AuthenticationService()
    {
        msgSevice = new ShortMessageService(); // 建立用來發送驗證碼的物件
    }

    public bool TwoFactorLogin(string userId, string pwd)
    {
        // 沒有變動，故省略。
    }
}
```

其中的 TwoFactorLogin 方法的實作完全沒變，是因為 ShortMessageService 類別也有一個 Send 方法，而且這方法跟 EmailService 的 Send 方法長得一模一樣：接受兩個傳入參數，一個是 User 物件，另一個是訊息內容。底下同時列出兩個類別的原始碼。

```
class EmailService
{
    public void Send(User user, string msg)
    {
        // 寄送電子郵件給指定的 user (略)
    }
}

class ShortMessageService
{
    public void Send(User user, string msg)
    {
        // 發送簡訊給指定的 user (略)
    }
}
```

你可以看到，這種解法僅僅改動了 AuthenticationService 類別的兩個地方：

1. 私有成員 msgService 的型別。
2. 建構函式中，原本是建立 EmailService 物件，現在改為 ShortMessageService。

剩下要做的，就只是編譯整個專案，然後部署新版本的應用程式。這種解法的確改得很快，程式碼變動也很少，但是卻沒有解決根本問題。於是，麻煩很快就來了：使用者反映，他們有時想要用 e-mail 接收登入驗證碼，有時想要用手機簡訊。這表示應用程式得在登入畫面中提供一個選項，讓使用者選擇以何種方式接收驗證碼。這也意味著程式內部實作必須要能夠支援執行時期動態切換「提供發送驗證碼服務的類別」。為了達到執行時期動態切換實作類別，相依型別之間的繫結就不能在編譯時期決定，而必須採用動態繫結。

接著就來看看如何運用 DI 來讓剛才的範例程式具備執行時期切換實作類別的能力。

入門範例—DI 版本

為了讓 AuthenticationService 類別能夠在執行時期才決定要使用 EmailService 還是 ShortMessageService 來發送驗證碼，我們必須對這些類別動點小手術，把它們之間原本緊密耦合的關係鬆開——或者說「解耦合」。有一個很有效的工具可以用來解耦合：介面（interface）。

說得更明白些，原本 AuthenticationService 是依賴特定實作類別來發送驗證碼（如 EmailService），現在我們要讓它依賴某個抽象介面，而此介面會定義發送驗證碼的工作必須包含那些操作。由於介面只是一份規格（specification；或者說「合約」），並未包含任何實作，故任何類別只要實作了這份規格，便能夠與 AuthenticationService 銜接，完成發送驗證碼的工作。有了中間這層介面，開發人員便能夠**「針對介面、而非針對實作來撰寫程式。」（program to an interface, not an implementation）**[5]，使應用程式中的各部元件保持「有點黏、又不會太黏」的適當距離，從而達成寬鬆耦合的目標。

[5]Gamma、Erich 等人合著之《Design Patterns: Elements of Reusable Object-Oriented Software》，Addison-Wesley, 1994. p.18。這裡的「實作」指的是包含實作程式碼的具象類別。

提煉介面（Extract Interface）

開始動手修改吧！首先要對 EmailService 和 ShortMessageService 進行抽象化（abstraction），亦即將它們的共通特性抽離出來，放在一個介面中，使這些共通特性成為一份規格，然後再分別由具象類別來實作這份規格。

以下程式碼是重構之後的結果，其中包含一個介面和兩個實作類別。我在個別的 Send 方法中使用 Console.WriteLine 方法來輸出不同的訊息字串，方便觀察實驗結果（此範例是個 Console 類型的應用程式專案）。

```
// EmailService 和 ShortMessageService 都有 Send 方法，
// 故將此方法提出來，放到一個抽象介面中來定義。
interface IMessageService
{
    void Send(User user, string msg);
}

class EmailService : IMessageService
{
    public void Send(User user, string msg)
    {
        // 寄送電子郵件給指定的 user (略)
        Console.WriteLine(" 寄送電子郵件給使用者，訊息內容：" + msg);
    }
}

class ShortMessageService : IMessageService
{
    public void Send(User user, string msg)
    {
        // 發送簡訊給指定的 user (略)
        Console.WriteLine(" 發送簡訊給使用者，訊息內容：" + msg);
    }
}
```

看類別圖可能會更清楚些：

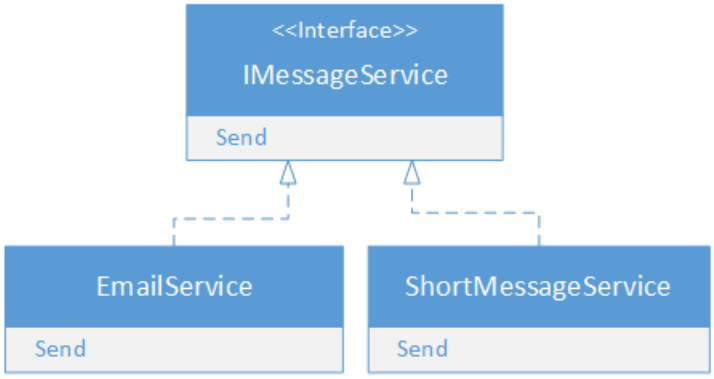


圖 1-2：抽離出共通介面之後的類別圖

介面抽離出來之後，如先前提過的，AuthenticationService 就可以依賴此介面，而不用再依賴特定實作類別。為了方便比對差異，我將修改前後的程式碼都一併列出來：

```
class AuthenticationService
{
    // 原本是這樣：
    private ShortMessageService msgService;

    public AuthenticationService()
    {
        this.msgSevice = new ShortMessageService();
    }

    // 現在改成這樣：
    private IMessageService msgService;

    public AuthenticationService(IMessageService service)
    {
        this.msgService = service;
    }
}
```

修改前後的差異如下：

- 私有成員 msgService 的型別：修改前是特定類別（EmailService 或 ShortMessage-Service），修改後是 IMessageService 介面。
- 建構函式：修改前是直接建立特定類別的執行個體，並將物件參考（reference）指定給私有成員 msgService；修改後則需要由外界傳入一個 IMessageService 介面參考，並將此參考指定給私有成員 msgService。

控制反轉（IoC）

現在 AuthenticationService 已經不依賴特定實作了，而只依賴 IMessageService 介面。然而，介面只是規格，沒有實作，亦即我們不能這麼寫（無法通過編譯）：

```
IMessageService msgService = new IMessageService();
// 錯誤：不能建立抽象介面的執行個體!
```

那麼物件從何而來呢？答案是由外界透過 AuthenticationService 的建構函式傳進來。請注意這裡有個重要意涵：非 DI 版本的 AuthenticationService 類別使用 new 運算子來建立特定訊息服務的物件，並控制該物件的生命週期；DI 版本的 AuthenticationService 則將此控制權交給外層呼叫端（主程式）來負責——換言之，相依性被移出去了，**「控制反轉了」**。

最後要修改的是主程式（MainApp）：

```
class MainApp
{
    public void Login(string userId, string pwd, string messageServiceType)
    {
        IMessageService msgService = null;

        // 用字串比對的方式來決定該建立哪一種訊息服務物件。
        switch (messageServiceType)
        {
            case "EmailService":
                msgService = new EmailService();
                break;
            case "ShortMessageService":
                msgService = new ShortMessageService();
                break;
            default:
                throw new ArgumentException(" 無效的訊息服務型別!");
        }

        var authService = new AuthenticationService(msgService);  // 注入相依物件
        if (authService.TwoFactorLogin(userId, pwd))
        {
            // 此處沒有變動，故省略.
        }
    }
}
```

現在主程式會負責建立訊息服務物件，然後在建立 `AuthenticationService` 物件時將訊息服務物件傳入其建構函式。這種由呼叫端把相依物件透過建構函式注入至另一個物件的作法是 DI 的一種常見寫法，而這寫法也有個名稱，叫做「**建構式注入**」（**Constructor Injection**）。「建構式注入」是實現 DI 的一種方法，第 2 章會進一步介紹。

現在各型別之間的相依關係如下圖所示。請與先前的圖 1-1 比較一下兩者的差異（為了避免圖中箭頭過於複雜交錯，我把無關緊要的配角 `User` 類別拿掉了）。

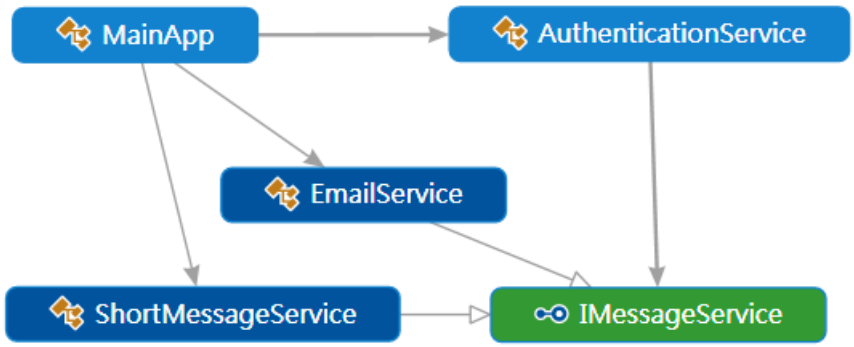


圖 1-3：改成 DI 版本之後的類別相依關係圖

你會發現，稍早的圖 1-1 的相依關係，是上層依賴下層的方式；或者說，高層模組依賴低層模組。這只符合了先前提過的 S.O.L.I.D. 五項原則中的**「相依反轉原則」（Dependency Inversion Principle；DIP）**的其中一小部分的要求。**DIP** 指的是：

- 高層模組不應依賴低層模組；他們都應該依賴抽象層（abstractions）。
- 抽象層不應依賴實作細節；實作細節應該依賴抽象層。

而從圖 1-3 可以發現，DI 版本的範例程式已經符合「相依反轉原則」。其中的 `IMessageService` 介面即屬於抽象層，而高層模組 `AuthenticationService` 和低層模組皆依賴中間這個抽象層。

這裡順便提供兩個有助於實踐「相依反轉原則」的小技巧：

- 宣告變數時盡量使用抽象型別，而不使用具象類別（concrete class）。
- 碰到需要繼承類別的場合，只繼承自抽象類別，或讓類別實作某個介面。理由：一旦你的類別繼承自特定具象類別，就等於是依賴特定實作，且與之緊密耦合。

這些原則並非鐵律，仍須因地制宜。比如說，你絕對不會想要把所有變數都宣告成 `Object` 型別，然而在碰到需要操作串流資料的時候，將變數宣告成抽象型別 `Stream`（而不是 `FileStream`）所帶來的彈性，可能正好是你要的。

此 DI 版本的範例程式有個好處，即萬一將來使用者又提出新的需求，希望發送驗證碼的方式除了 e-mail 和簡訊之外，還要增加行動裝置平台的訊息推播服務（push notification），以便將驗證碼推送至行動裝置的 app。此時只要加入一個新的類別（可能命名為 `PushMessageService`），並讓此類別實作 `IMessageService`，然後稍微改一下 `MainApp` 便大致完工，`AuthenticationService` 完全不需要修改。簡單地說，應用程式更容易維護了。

有注意到嗎？現在這個版本的 `AuthenticationService` 類別也符合了稍早提及的 S.O.L.I.D. 設計原則中的 **OCP**（開放／封閉原則）。

當然，這個範例的程式寫法還是有個缺點：它是用字串比對的方式來決定該建立哪一種訊息服務物件。想像一下，如果欲支援的訊息服務類別有十幾種，那個 `switch...case` 區塊不顯得太冗長累贅嗎？如果有一個專屬的物件能夠幫我們簡化這些型別對應以及建立物件的工作，那就更理想了。這個部分會在第 3 章〈DI 容器〉中進一步說明。

何時該用 DI？

一旦你開始感受到寬鬆耦合的好處，在設計應用程式時，可能會傾向於讓所有類別之間的耦合都保持寬鬆。換言之，碰到任何需求都想要先定義介面，然後透過 DI 模式（例如

先前範例中使用的**「建構式注入」**）來建立物件之間的相依關係。然而，天下沒有白吃的午餐，寬鬆耦合也不例外。每當你將類別之間的相依關係抽離出來，放到另一個抽象層，再於特定時機注入相依物件，這樣的動作其實多少都會產生一些額外成本。不管三七二十一，將所有物件都設計成可任意替換、隨時插拔，並不總是個好主意。

設計時盡量採取寬鬆耦合原則並搭配 DI 技術，通常意味著應用程式的各部元件更容易隨時抽離替換，更接近軟體組裝的境界。但這並不是說，我們應該在任何場合做到所有類別皆可隨時替換。軟體系統的架構設計涵蓋許多面向，例如開發時程、維護成本、功能需求與非功能需求等等，往往須要經過一番取捨，才能做出比較正確的設計決策。再次強調，提升可維護性才是終極目標，而 DI 只是達成此目標的手段；切莫因為別人都在談論 DI，或只因為我們會使用此技術，就在程式裡面任意使用 DI。

以 .NET 基礎類別庫（Base Class Library；簡稱 BCL）為例，此類別庫包含許多組件，各組件又包含許多現成的類別，方便我們直接取用。每當你在程式中使用 BCL 的類別，例如 `String`、`DateTime`、`Hashtable` 等等，就等於在程式中加入了對這些類別的依賴。此時，你會擔心有一天自己的程式可能需要把這些 BCL 類別替換成別的類別嗎？如果是從網路上找到的開放原始碼呢？答案往往取決於你對於特定類別／元件是否會經常變動的信心程度；而所謂的「經常變動」，也會依應用程式的類型、大小而有不同的定義。

相較於其他在網路上找到或購買的第三方元件，我想多數人都會覺得 .NET BCL 裡面的類別應該會相對穩定得多，亦即不會隨便改來改去，導致既有程式無法編譯或正常執行。這樣的認知，有一部分可能來自於我們對該類別的提供者（微軟）有相當程度的信心，另一部分則是來自以往的經驗。無論如何，在為應用程式加入第三方元件時，最好還是審慎評估之後再做決定。

以下是幾個可能需要使用或了解 DI 技術的場合：

- 如果你正在設計一套框架（framework）或可重複使用的類別庫，DI 會是很好用的技術。

- 如果你正在開發應用程式，需要在執行時其動態載入、動態切換某些元件，DI 也能派上用場。
- 希望自己的程式碼在將來需求變動時，能夠更容易替換掉其中一部份不穩定的元件（例如第三方元件，此時可能搭配 Adapter 模式[6]使用）。
- 你正在接手維護一個應用程式，想要對程式碼進行重構[7]，以降低對某些元件的依賴，方便測試並且讓程式碼更好維護。

以下是一些可能不適合、或應該更謹慎使用 DI 的場合：

- 在小型的、需求非常單純的應用程式中使用 DI，恐有殺雞用牛刀之嫌。
- 在大型且複雜的應用程式中，如果到處都是寬鬆耦合的介面、到處都用 DI 注入相依物件，對於後續接手維護的新進成員來說可能會有點辛苦。在閱讀程式碼的過程中，他可能會因為無法確定某處使用的物件究竟是哪個類別而感到挫折。比如說，看到程式碼中呼叫 `IMessageService` 介面的 `Send` 方法，卻沒法追進去該方法的實作來了解接著會發生什麼事，因為介面並沒有提供任何實作。若無人指點、也沒有文件，每次碰到疑問時就只能以單步除錯的方式來了解程式實際運行的邏輯，這確實需要一些耐心。
- 對老舊程式進行重構時，可能會因為既有設計完全沒考慮寬鬆耦合，使得引入 DI 困難重重。

總之，不加思索地使用任何技術總是不好的；沒有銀彈[8]。

本章回顧

就本章的範例而言，從非 DI 版本改成 DI 版本的寫法有很多種，而作為 Hello World 等級的入門範例，這裡僅採用其中一種最簡單的作法：從類別的建構式傳入實際的物件，並透

[6] http://en.wikipedia.org/wiki/Adapter_pattern
[7] http://refactoring.com/
[8] http://en.wikipedia.org/wiki/No_Silver_Bullet

過這種方式，將兩個類別之間的相依性抽離至外層（即範例中的 `MainApp` 類別），以降低類別之間的耦合度。底下是本章的幾個重點：

- 可維護性是終極目標，寬鬆耦合是邁向此目標的基本原則，DI 則是協助達成目標的手段。
- DI 是一組設計原則與模式，其核心概念是「針對介面寫程式，而非針對實作」（program to an interface, not an implementation），並透過將相依關係抽離至抽象層（abstraction layer）來降低各元件之間的耦合度，讓程式更好維護。
- DI 是 IoC（Inversion of Control；控制反轉）的一種形式，兩者經常交替使用，但並不完全相等。
- 相依反轉原則（Dependency Inversion Principle；DIP）包含兩個要點：(1) 高層模組不應依賴低層模組，他們都應該依賴抽象層（abstractions）；(2) 抽象層不應依賴實作細節，而應該是實作細節依賴抽象層。
- DI 不是銀彈，使用時仍須思考該用於何處、不該用於何處，以期發揮最大效益。

DI 的內涵包括四大議題：注入方式、物件組合、物件生命週期管理、攔截。本章的範例已稍微觸及前兩項，後續章節會進一步討論這些議題。

本書內容大綱

The End