.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：

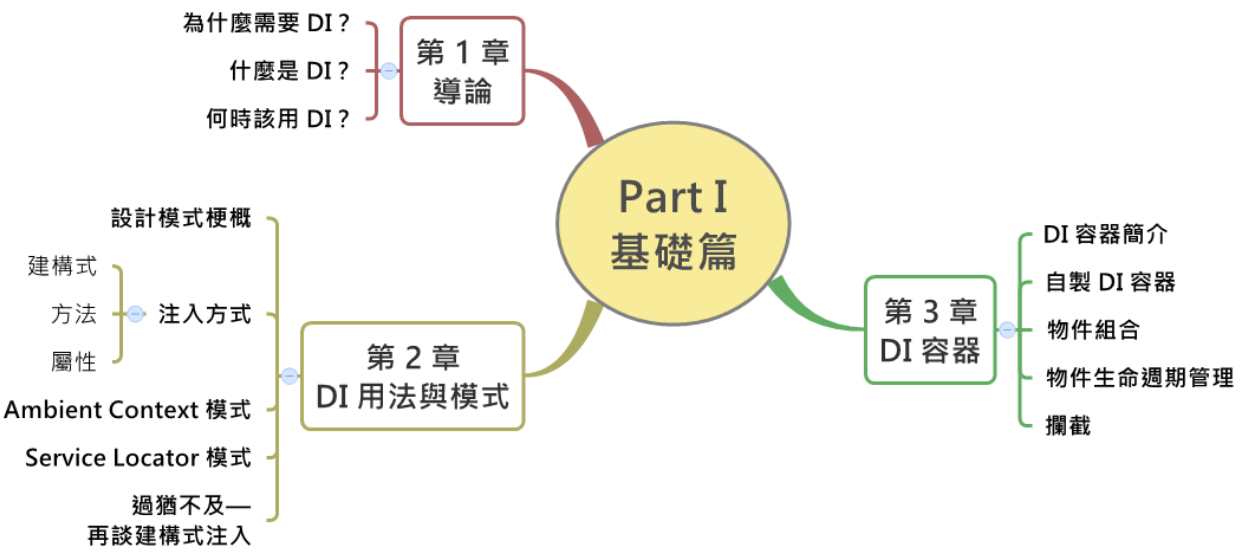
- 《軟體構築美學》，2010。原著：Brownfield Application Development in .NET。
- 《物件導向分析設計與應用》[1]，2009。原著：Object-Oriented Analysis and Design with Applications 3rd Edition。
- 《ASP.NET AJAX 經典講座》，2007。原著：Introducing Microsoft ASP.NET AJAX。
- 《微軟解決方案框架精要》，2007。原著：Microsoft Solution Framework Essentials。
- 《軟體工程與 Microsoft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

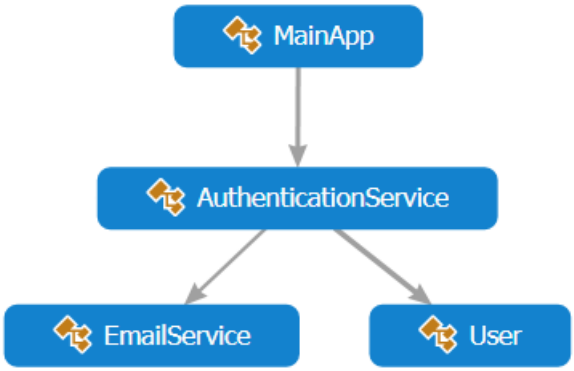


圖 1-1：類別相依關係圖

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：

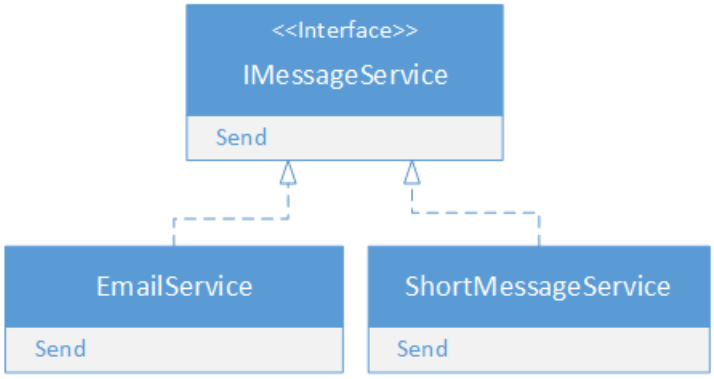


圖 1-2：抽離出共通介面之後的類別圖

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）。

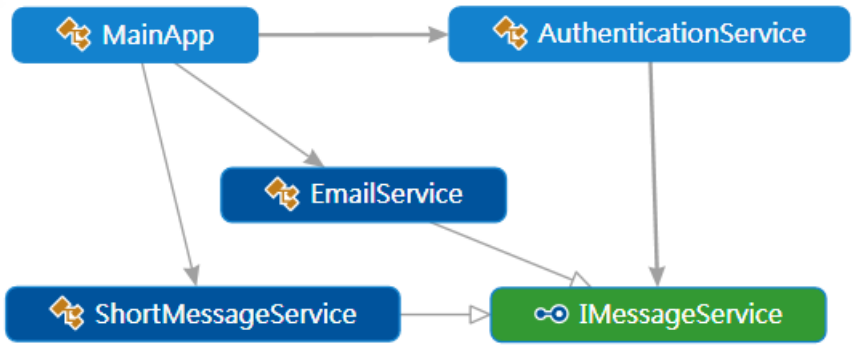


圖 1-3：改成 DI 版本之後的類別相依關係圖

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。

第 2 章：DI 用法與模式

讀完第 1 章以後，您應該已經了解 DI 的用途與目的，接著要來進一步了解的是 DI 的實作技術，也就是注入相依物件的方式。本章所介紹的 DI 用法，又稱為「窮人的 DI」（poor man's DI），因為這些用法都與特定 DI 工具無關，亦即不使用任何現成的 DI 框架（例如 Unity、Autofac）。畢竟，DI 只是一組設計原則與模式，不依賴第三方類別庫也照樣能實現。

內容大綱：

本章範例原始碼位置：

https://github.com/huanlin/di-book-support 裡面的 Examples/ch02 資料夾。

設計模式梗概

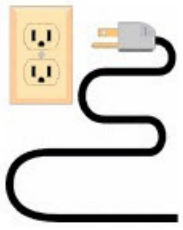
> 每個模式都描述了一個不斷發生在我們周遭的問題，然後描述該問題的核心解法，於是你便可以一再使用該解法，而無須對同樣的事情做兩次工。
>
> ——Christopher Alexander. *A Pattern Language*.

除了第 1 章提到的 **S.O.L.I.D.** 設計原則，在運用 DI 技術時，也經常需要搭配一些設計模式（design patterns），例如 **Factory Method**（工廠方法）、**Decorator**（裝飾）、**Composite**（組合）、**Adapter**（轉換器）等等。基於後續章節討論的必要，本節將介紹幾個相關的設計模式。如需比較完整深入的介紹，可參考相關書籍，例如：《物件導向設計模式》[9]、《深入淺出設計模式》[10]、《重構－向範式前進》[11]等等。

閱讀提示：若您已經很熟悉設計模式，可直接跳到下一節〈注入方式〉。

小引－電器與介面

日常生活中，四處可見電器用品，例如電視、微波爐、電腦等等。這些電器通常都有條電線，電線尾端是個插頭，而當我們要使用這些電器時，就把插頭插在牆壁或電源插座上，電器便能夠獲得



[9] 《物件導向設計模式》，作者：Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides。翻譯：葉秉哲。
[10] 《深入淺出設計模式》，作者：Eric Freeman、Bert Bates、Kathy Sierra、Elisabeth Robson。翻譯：蔡學鏞。
[11] 《重構－向範式前進》，作者：Joshua Kerievsky。翻譯：侯捷、陳裕城。

所需之電力。一般情況下，沒有人會捨插座不用，而把電器的電源線固定焊在牆壁的電源插座。假使真這麼做，萬一有一天電視或電腦故障而需要維修，那可就麻煩了。

不只電源插座，電腦的 USB 插槽也一樣——它們都具備寬鬆耦合的特性。這裡的電源插座或 USB 插槽，對應到軟體世界裡的概念，便是介面。一個介面就等於是一份規格，而各家廠商所生產的各式各樣的電源插座或 USB 插槽，就是遵照其標準規格（介面）所實作出來的產品，或簡稱實作品。用軟體的術語來說，這些實作品就是類別——實作了特定介面的類別。

介面的威力即在於一旦訂出標準規格，各家廠商便可依照標準介面來製作各類產品。對使用者來說，好處則是享有多種選擇，因為他們不會被特定廠商的產品綁住；只要他們高興，隨時可以更換不同的產品，而且通常是隨插即用。在軟體的世界裡，介面也有同樣的好處：讓類別與類別之間保持寬鬆耦合，以便提供隨時抽換實作類別的彈性。

Null Object 模式

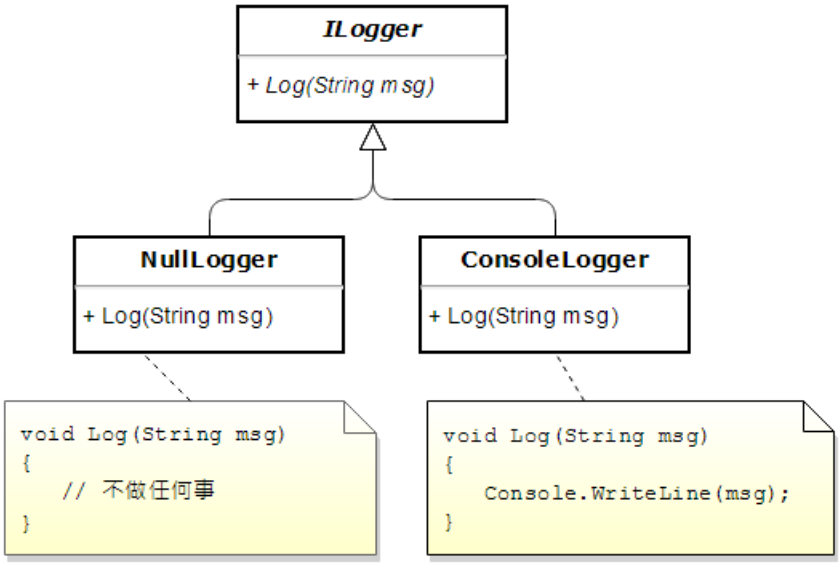
回到電源插座的例子。如果我們將電腦的電源線從插座上拔起，它們就只是彼此不再連接而已，電腦和插座並不會因此而著火或爆炸。但是在軟體程式的世界裡，若物件 A 會呼叫物件 B（物件 A 依賴物件 B），而當你將物件 B 移除，亦即物件 B 不存在時，程式就會發生 `NullReferenceException` 類型的錯誤。於是，我們常常會在程式裡面加入檢查物件參考是否為 `null` 的邏輯，例如：

```
if (anObject != null)
    anObject.DoSomething();
else
    DoSomethingElse();
```

如果在程式中一再重複寫這些檢查 `null` 的邏輯，程式碼便會膨脹，而且在解讀程式的主要邏輯時，常常得要跳過這些檢查邏輯，多少會形成閱讀程式碼的阻礙。針對此問題，我們可以設計一個空的、完全不做任何事的類別，然後在變數有可能是 `null` 的地方，讓它們指向那個空的物件。這種模式叫做 **Null Object**。

Null Object 的優點：可減少撰寫判斷物件參考是否為 `null` 的防錯邏輯。但前提是開發人員得知道有 **Null Object** 可用，否則還是會寫出多餘的防錯程式碼。

Null Object 類別通常要實作某個介面（或繼承自抽象類別），但實作程式碼完全沒做任何事，即所有方法都只是個空殼子，或僅提供無害的預設行為。以程式中常用的 logging（日誌）機制為例，我們可以將寫入日誌的操作定義成一個 `ILogger` 介面，然後依實際需要撰寫不同的實作類別，例如用來將日誌訊息輸出至 Console 的 `ConsoleLogger`。此外，考慮到應用程式有時候可能不需要紀錄任何訊息，我們可以寫一個 `NullLogger` 類別，作為 **Null Object** 之用。結構圖如下。



底下分別是 `ILogger` 介面以及 `NullLogger` 和 `ConsoleLogger` 類別的程式碼：

```
public interface ILogger
{
    void Log(string msg);
}

public class NullLogger : ILogger
{
    public void Log(string msg)
    {
        // 不做任何事
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string msg)
    {
        Console.WriteLine(msg);
    }
}
```

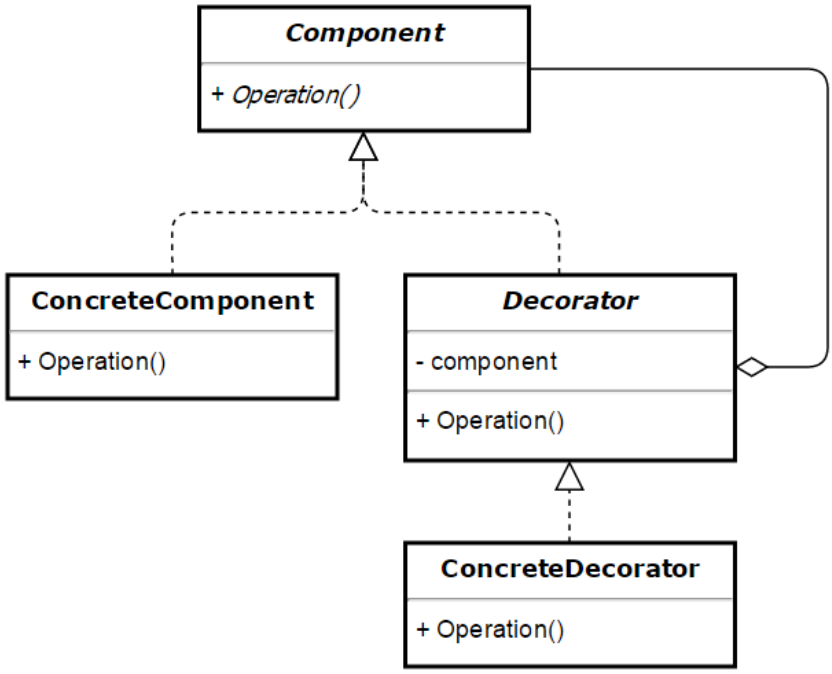
像底下這個函式，呼叫端只要傳入 `ConsoleLogger` 物件，日誌訊息就會輸出至 Console；而當呼叫端想要停止記錄日誌，便可傳入 `NullLogger` 物件。如此一來，就不用在每次寫入日誌訊息時都重複寫一遍檢查 `logger` 物件是否為 `null` 的防錯邏輯。

```
void DoSomething(ILogger logger)
{
    logger.Log(" 開始執行 DoSomething 函式。");
    ....
}
```

Null Object 本身並不需要「進化」成真正有做事的物件，因為它的存在就是為了提供一個完全不做任何事、不具任何意義的物件。

Decorator 模式

一般情況下，如果在使用電腦時突然停電了，尚未儲存的資料就會消失不見。為了解決此問題，我們可以在牆壁的電源插座與電腦電源線之間加入一個不斷電系統（UPS）。此時，UPS 的電源線會接在牆壁的電源插座上，而電腦的電源則改接在 UPS 上。此三者在串接的時候，都是透過單一的標準介面：插座。類似這種透過同一介面來串接多個不同物件的作法，叫做 **Decorator Pattern**（裝飾模式）。此模式可以讓我們為物件層層疊加新的功能上去，而無須修改既有的類別。下圖為 **Decorator 模式**的結構圖。



延續前面的 logging 範例，假設想要在每次輸出 log 訊息時額外加上當時的日期時間，而且前提是不可修改既有的 `ILogger` 和 `ConsoleLogger` 類別，該怎麼做？

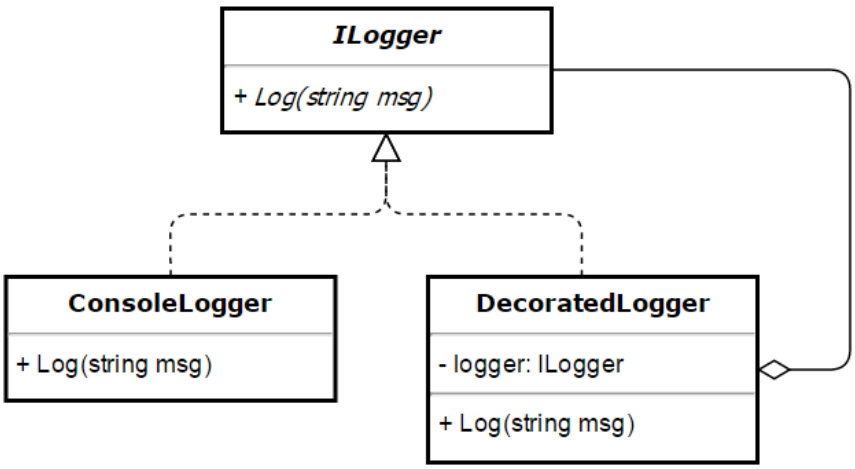
我們可以使用 **Decorator 模式**。作法為：設計一個新的類別，此類別不僅要實作 `ILogger` 介面，而且還需要使用既有的 `ConsoleLogger` 物件來輸出 log 訊息。簡單起見，我就把這

個類別命名為 DecoratedLogger。程式碼如下：

```
public class DecoratedLogger : ILogger
{
    private ILogger logger;

    public DecoratedLogger(ILogger aLogger)
    {
        logger = aLogger;
    }

    public void Log(string msg)
    {
        logger.Log(DateTime.Now.ToString() + " - " + msg);
    }
}
```

基本上，DecoratedLogger 是把另一個 ILogger 物件包起來，並對該物件加一些額外的「裝飾」（這裡是對 ILogger.Log 方法加點裝飾）。

下圖描繪了這個簡略版本的 **Decorator 模式**範例的類別結構：



於是，在用戶端程式中使用這個新的 DecoratedLogger 來輸出 log 訊息時，可以這麼寫：

```
    void DoSomething()
    {
        ILogger logger = new DecoratedLogger(new ConsoleLogger());
        logger.Log("Hello, 裝飾模式!");
    }
```

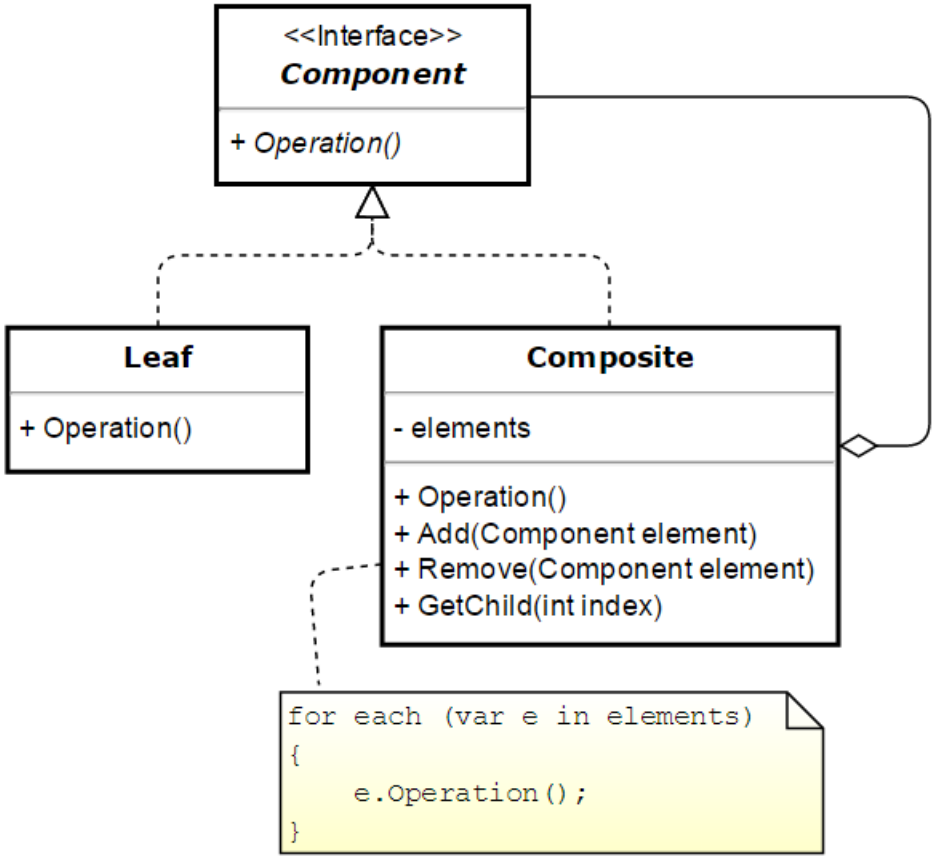
你可以看到，在這次的修改當中，既有的 `ILogger` 和 `ConsoleLogger` 完全沒有動到。我們只增加了一個新類別（`DecoratedLogger`），就為應用程式加上了新功能。這也就符合了第 1 章提過的 **OCP**（開放／封閉原則）。

Composite 模式

延續先前的電器比喻。現在，如果希望 UPS 不只接電腦，還要接電風扇、除濕機，可是 UPS 卻只有兩個電源輸出孔，怎麼辦？

我們可以買一條電源延長線，接在 UPS 上面。如此一來，電風扇、除濕機、和電腦便都可以同時插上延長線的插座了。這裡的電源延長線，即類似 **Composite Pattern**（組合模式），因為電源延長線本身又可以再連接其他不同廠牌的延長線（這又是因為插座皆採用相同介面），如此不斷連接下去。

呃....延長線的比喻有個小問題：它在外觀上看起來也像是層層串接，容易和 **Decorator 模式**混淆。事實上，這兩種設計模式在結構上的確有相似之處。下圖所示為 **Composite 模式**的結構圖。

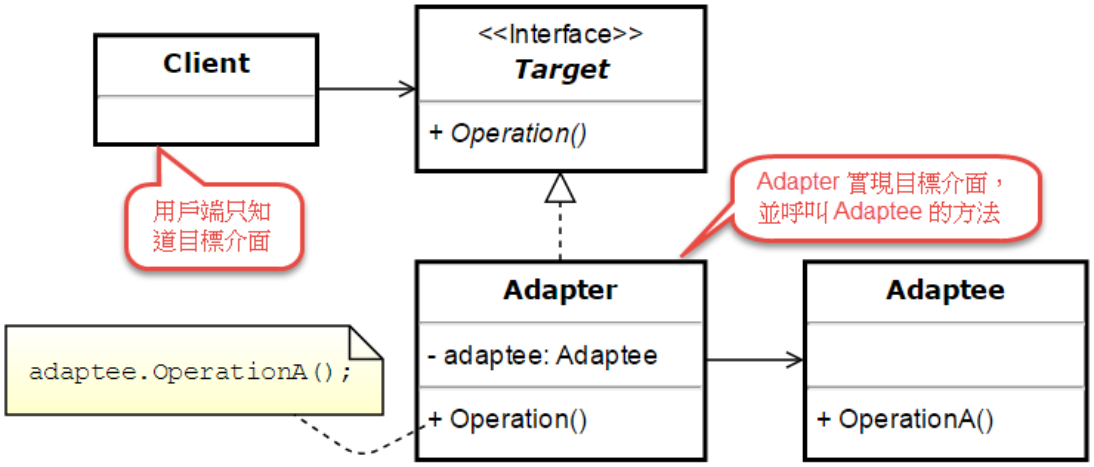


Composite 模式的結構圖

由此結構圖可以看得出來，**Composite 模式**其實是個樹狀結構，呈現的是「整體－包含」（whole-part）的關係。樹上的每個節點（Leaf）都實作了相同的介面，而每個節點又可以包含多個子節點；就像檔案目錄結構那樣，每個資料夾底下都可以有零至多個資料夾。相較之下，**Decorator 模式**則是讓裝飾者看起來長得和被裝飾者一樣，但其實還加上了額外的裝飾。

Adapter 模式

當你的手機沒電，需要充電時，就算有電源延長線也沒用，因為手機充電時所需的電壓並不是一般家庭用電的 110 伏特交流電壓。此時我們通常會使用手機隨附的變壓器（adapter），將變壓器的電源插頭插在牆壁的電源插座，然後將變壓器的另一端連接至手機。像這樣把一種規格（介面）轉換成另一種規格的設計，就叫做 **Adapter Pattern**（轉換器模式）。下圖所示為 **Adapter 模式**的結構。



Adapter 模式的結構圖

仍使用先前 logging 範例來說明。假設我們沒有實作自己的 logging API，而是直接使用第三方元件。然而，考慮到將來很可能會改用另一套 logging 元件，於是決定使用 **Adapter 模式**來保護自己的程式碼。首先，必須先訂出 logging API 的介面，讓應用程式只針對此介面來寫入 log。此介面只定義了一個寫入日誌的方法，叫做 Log，參考以下程式片段。

```
public interface ILogger
{
    void Log(string msg);
}
```

接著設計 **Adapter** 類別。此類別須實作 ILogger 介面，並且在 Log 方法中轉而呼叫第三方元件的方法。程式碼如下：

```
public class CommonLogger : ILogger
{
    private ThirdPartyLogger logger = new ThirdPartyLogger();

    public void Log(string msg)
    {
        logger.WriteEntry(msg);  // 轉呼叫第三方元件的方法。
    }
}
```

如此一來，以後如果真的需要改用另一套 logging 元件，程式修改的範圍就只限定在 CommonLogger 類別而已。

Factory 模式

第一章曾經提過，每當我們在程式中使用 new 運算子來建立類別的執行個體，我們的程式碼就在編譯時期跟那個類別固定綁（繫結）在一起了。其實用 new 來建立物件還有個缺點：C# 的建構函式名稱就是類別名稱，不可任意命名；於是當類別有數個多載的（overloaded）建構函式時，光是閱讀傳入建構函式的參數列，有時不見得那麼容易明白程式的意圖。舉例來說：

```
var user1 = new User("Mike", 101, true);
var user2 = new User("Jane", 102, flase);
```

不如下列程式碼清楚：

```
var user1 = UserFactory.CreateAdministrator("Mike", 101);
var user2 = UserFactory.CreateDomainUser("Jane", 102);
```

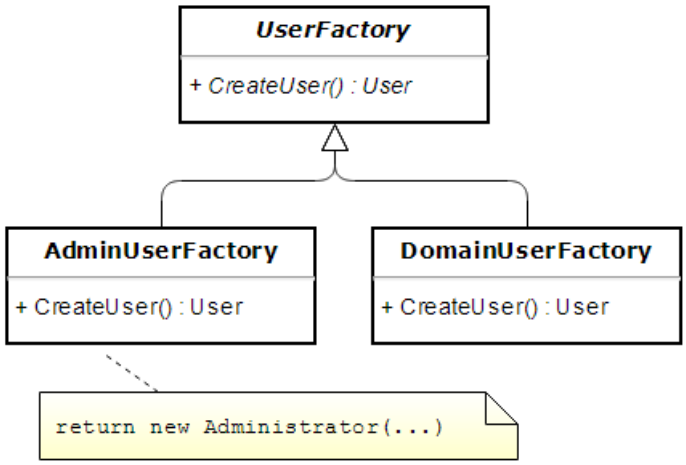
其中的 `UserFactory` 就是擔任物件工廠的角色，它是個 `static` 類別，且唯一的任務就是生產特定類型的物件。程式碼如下所示。

```
public static class UserFactory
{
    public static User CreateAdministrator(string name, int id)
    {
        // 略
    }

    public static User CreateDomainUser(string name, int id)
    {
        // 略
    }
}
```

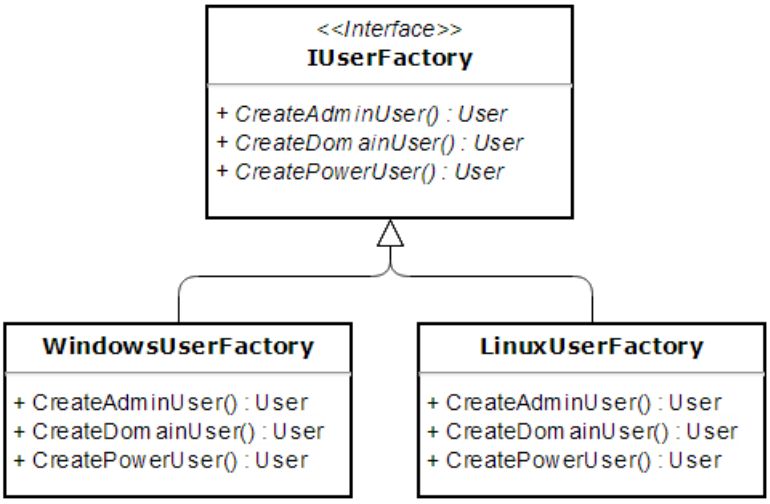
一般而言，**Factory 模式**泛指各種能夠生產物件的工廠，可再細分成三種模式：**Factory Method**（工廠方法）、**Simple Factory**（簡單工廠）、和 **Abstract Factory**（抽象工廠）。剛才的 `UserFactory` 就是一個 **Simple Factory**。

假設我們完全不知道 DI，或者覺得沒必要使用 DI，可是又希望程式碼不要和特定實作類別綁太緊（不想要直接 `new` 一個物件），此時 **Factory 模式**就是個值得考慮的方案。

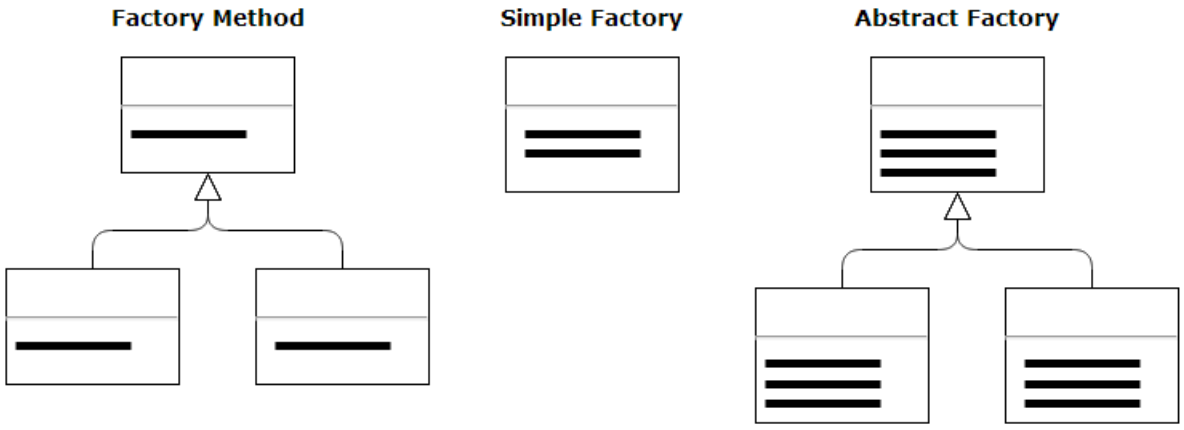
實作 **Factory Method** 模式時，通常會在一個基礎類別中定義建立物件的抽象方法（難怪叫做「工廠方法」），然後由各個子類別來實作該方法。若將先前的 `UserFactory` 改成以 **Factory Method** 來實作，其類別結構如下圖所示。



Abstract Factory 比前面兩種工廠模式要稍微複雜一些，它是用來建立多族系的相關或相依物件，且無須指名物件的具象類別。實作此模式時，會將一組建立物件的方法定義成一個介面，代表抽象工廠。然後，你可以撰寫多個類別來實作該介面，而這些類別的角色就像真實世界中的工廠，類別中的每一個工廠方法則有點像是真實工廠裡的一條生產線。當用戶端需要建立該族系的物件時，就是利用其中一種具象工廠（concrete factory）來生成物件。此外，由於具象工廠都實作了同一組介面，所以用戶端甚至可以在執行時期動態切換成不同的工廠，以建立一組相關的物件。



如果你跟我一樣，常常搞混這三種 **Factory 模式**，我發現《Refactoring to Patterns》這本書的 6.2 節裡面有一張簡略的結構圖挺有用。我依樣畫了一張，如下圖所示，其中的粗黑線代表建立物件的函式。下次忘記時，不妨回來瞄一眼底下這張圖，也許能幫你回想起來它們之間的差異。



設計模式的部分就概略介紹到此，後續章節中如碰到其他模式，也會一併介紹（例如

Strategy、**Repository**、**Service Locator** 等等）。

> 我曾在這樣的十字路口：努力學習各種模式，希望成為一個更好的軟體設計師；但現在，為了真正成為更優秀的軟體設計師，我必須降低對模式的依賴。
>
> ——Joshua Kerievsky. 《Refactoring to Patterns》作者

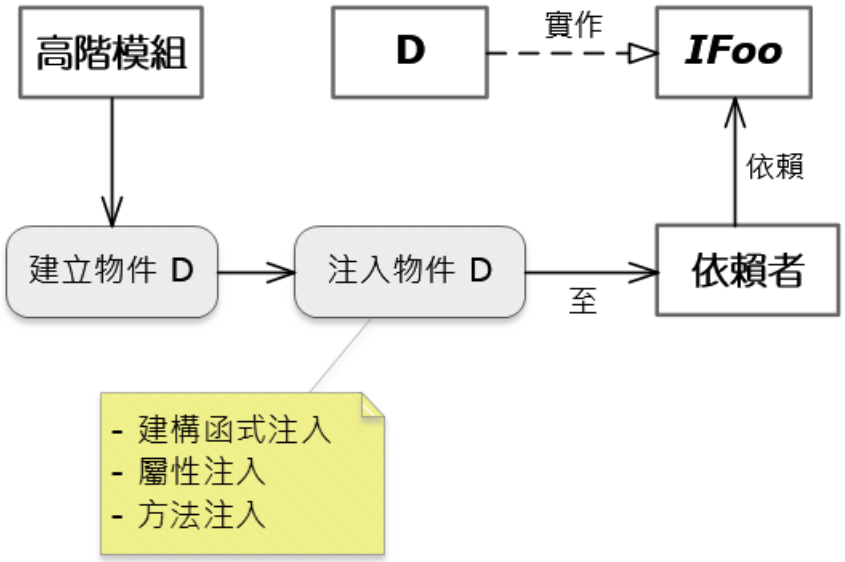
注入方式

DI 的核心概念是寬鬆耦合，是「針對介面寫程式」，故一旦開始在程式中運用 DI 技術，你可能會開始對「new 一個物件」的寫法更敏感。你可能會開始考慮，這個地方如果用 new 來建立特定實作類別的物件，將來需要修改程式時會不會很麻煩？如果只依賴介面或抽象類別會不會比較好？一旦開始出現這種現象，您已經朝向寬鬆耦合之路邁進了。

本節將介紹 DI 的三種注入方式，包括：

- 建構式注入（**Constructor Injection**）
- 屬性注入（**Property Injection**）
- 方法注入（**Method Injection**）

這三種注入方式基本上都符合下圖所描繪的模式。



建構式注入

建構式注入（Constructor Injection）指的是類別所需要的物件是由外界透過該類別的公開建構函式傳入。若需要注入 N 個相依物件，則建構函式通常至少要有 N 個引數，且引數的型別為介面或抽象類別。

已知應用例

.NET 基礎類別庫的 `System.IO.Compression.ZipArchive` 類別有一個建構函式提供了外界注入物件的機制，其函式原型如下：

```
public ZipArchive(Stream stream)
```

用法

假設類別 `AuthenticationService` 需要使用符合 `IMessageService` 介面的物件，可按以下步驟實現「建構式注入」：

1. 在 AuthenticationService 類別中宣告一個型別為 IMessageService 的成員變數。令此成員變數名稱為 msgService。
2. 撰寫建構函式，在參數列中加入一個 IMessageService 型別的參數，然後將此參數值設定給成員變數 msgService。

範例程式

```
class AuthenticationService
{
    private readonly IMessageService msgService;

    public AuthenticationService(IMessageService service)
    {
        if (service == null)
        {
            throw new ArgumentException("service");
        }
        this.msgService = service;
    }
}
```

程式說明：

- 在此範例中，AuthenticationService 採用建構式注入，要求外界傳入一個符合 IMessageService 介面的物件。若外界傳入不相容於此介面的型別，程式碼將無法通過編譯。
- 私有成員 msgService 的變數宣告之所以加上 readonly 關鍵字，是為了確保相依物件一旦在 AuthenticationService 的建構函式中設定完成後，就不能再改變。
- 建構函式還檢查了傳入的相依物件是否為 null，以確保一旦建構函式執行完畢，在此類別中的任何地方都可以使用相依物件，而無須擔心它是否為無效參考。

每當需要注入相依物件時，一般建議優先考慮「建構式注入」，因為其用法對呼叫端來說相當明確、直覺——建立物件時就要一併傳入所有相依物件，所以呼叫端透過建構函式便可得知某物件相依於哪些第三方元件。

屬性注入

顧名思義，**屬性注入（Property Injection）**就是透過物件的屬性來注入相依物件，另一個稱呼是「**設定函式注入**」（**Setter Injection**）。它與「建構式注入」相似，類別本身也需要有個成員變數來保存對相依物件的參考，但有個主要區別：「屬性注入」的時機比「建構式注入」來得晚，而且外界不一定會設定該屬性。也就是說，如欲採用「屬性注入」，類別本身通常要能夠取得相依物件的預設實作。如此一來，即使外界沒有注入相依物件，類別仍有預設的相依物件可用。

已知應用例

ASP.NET MVC 的 `ControllerBuilder.SetControllerFactory` 方法。此方法可用來切換 ASP.NET MVC 的 `DefaultControllerFactory`（第 4 章有範例程式）。此方法之原型宣告如下：

```
public void SetControllerFactory(IControllerFactory controllerFactory)
```

用法

在類別中定義一個公開屬性，以便用戶端可以隨時設定（或完全不設定）該屬性來切換相依物件。若用戶端未曾設定該屬性，則由類別本身提供預設實作，或撰寫 `null` 檢查邏輯來避免執行時期發生 `NullReferenceException`。

範例程式

```
class AuthenticationService
{
    private IMessageService msgService;

    public IMessageService MessageService
    {
        get { return this.msgService; }
        set { this.msgService = value; }
    }

    public void Login(string userId, string password)
    {
        MessageService.Send(...);          // 使用相依物件。
    }
}
```

屬性 MessageService 背後所使用的私有成員 msgService 不能宣告為 readonly，因為「屬性注入」允許外界於任何時候注入相依物件，甚至切換相依物件或根本不注入相依物件。對於不注入相依物件的情況，類別本身必須做些防護措施，以免其他地方存取相依物件時，因物件參考為 null 而引發 NullReferenceException 或其他異常。常見的防護措施是由類別本身提供預設的相依物件，時機可以選在建構函式中設定，或者更晚一點，在屬性的 getter 區塊中設定，參考以下程式範例：

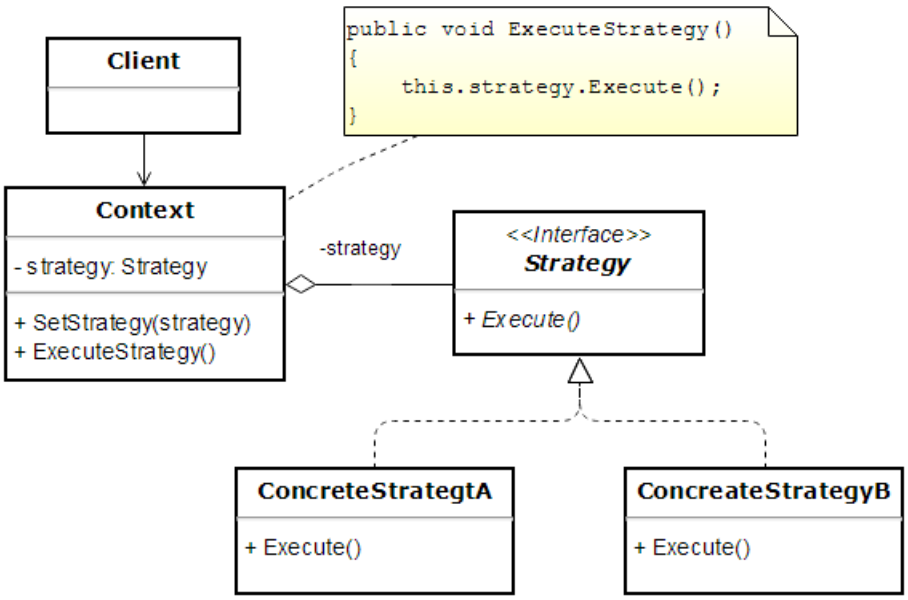
```
public IMessageService MessageService
{
    get
    {
        if (this.msgService == null)
        {
            // 外界沒有注入物件，就自己建立一個預設的物件。
            this.msgService = new DefaultMessageService();
        }
        return this.msgService;
    }
    set { this.msgService = value; }
}
```

只是，如此一來，此類別又和特定實作 DefaultMessageService 綁在一起了。如果你覺得

這種寫法不好，亦可考慮使用稍後介紹的 **Method Injection**（方法注入）或 **Ambient Context**（環境脈絡），或先前提過的 **Factory 模式**來解決，例如 **Factory Method**。

Strategy 模式

看完「建構式注入」和「屬性注入」這兩個小節之後，如果您熟悉設計模式，會不會有一種感覺：DI 骨子裡根本就是 **Strategy 模式**嘛！底下附上 **Strategy 模式**的結構圖，讀者不妨把它當作一個練習，揣摩看看兩者的異同。



方法注入

「**方法注入**」（**Method Injection**）指的是用戶端每次呼叫某物件的方法時都必須透過方法的引數來傳入相依物件。此注入方式的適用時機如下：

- 提供服務的類別不需要在整個類別中使用特定相依物件，而只有在用戶端呼叫某些方法時才需要傳入那些物件。

- 用戶端每次呼叫特定方法時可能會傳入不同的物件，而這些物件唯一相同之處是它們都實作了同一個介面（或繼承自同一個抽象類別）。

已知應用例

ADO.NET 的 `DbDataAdapter.Fill` 方法有提供「方法注入」的機制，其函式原型如下：

```
protected virtual int Fill(
    DataTable dataTable,     // 查詢結果會填入此物件。
    IDbCommand command,      // 提供查詢命令的物件。
    CommandBehavior behavior // 細部控制命令的行為。
)
```

用法

針對類別中的特定方法，把需要的相依物件加入方法的參數列。用戶端在每次呼叫這些方法時必須建立並傳入這些相依物件。

採用此注入方式時，除了傳入相依物件之外，經常會一併傳入其他相關的參數值，以便完成特定任務。例如：

```
public void ExecuteTask(ITaskContext context, int value1, int value2)
{
    // 略
}
```

範例

仍以先前的 `AuthenticateService` 為例，如果 Login 方法允許外界指定使用何種驗證碼發送機制，便可改成這樣：

```
public void Login(string userId, string password, IMessageService service)
{
    if (service == null)
    {
        throw new ArgumentException("service");
    }
    service.Send(...);
}
```

軟體框架和底層基礎建設（infrastructure）類型的函式庫也經常用到「方法注入」，因為這些類別庫通常不需要保存外界所提供的相依物件，而大多是針對每一次方法呼叫來讓多個物件共同達成任務。這同時也意味著，那些在方法呼叫之間傳遞的相依物件，通常也是在某高層模組需要完成特定工作時才臨時建立，而不會在應用程式的進入點或初始化階段就全都準備好這些物件。另一方面，如前兩節討論過的，提供「建構式注入」和「屬性注入」的類別則通常會在類別本身利用一個私有成員變數來保存外界注入的相依物件，而那些相依物件很有可能是高層模組初始化的時候、甚至在應用程式一開始執行時就已經預先建立的。

Ambient Context 模式

前述三種注入相依物件的方式，有些場合可能不適用，例如：應用程式特定執行環境的範圍內需要共享特定物件。碰到這種場合，便可以考慮採用 **Ambient Context**（環境脈絡）模式來解決。

Context 的中文術語

要為 context 這個名詞找個適當的中文挺傷腦筋。我有時候用「脈絡」，例如 **Ambient Context** 便是「環境脈絡」；有時候用「環境」，例如 **Context Object** 會寫成「環境物件」。往後提及與 context 有關的術語時，會盡量使用英文。

Ambient Context 又叫做 **Context Object**（環境物件），是一種常見的設計模式，主要用於跨階層、跨模組共享物件、界定程式執行區塊的範圍、以及提供橫切面的功能

（cross-cutting concerns）。這些到處都需要的物件或服務，不太可能一一注入到每個需要它們的地方：一來過於繁瑣，二來有些子模組或程式區塊是碰觸不到、或不在控制範圍內的。因此，**Ambient Context** 沒有明顯「注入物件」的味道；它不是侵入性的，而是在某個地方已經準備好、被動地等著別人來取用。此特性在某些場合正好可以彌補前述注入方式的不足，故在此一併討論。

已知應用例

.NET 類別庫中提供交易管理功能的 `System.Transactions.TransactionScope` 就是 **Ambient Context** 的一個例子。以下程式片段示範了基礎用法。

```
using (TransactionScope trxScope = new TransactionScope())
{
    // 執行多項資料異動作業。
    order.Add(newOrder);
        customer.LastOrderDate = DateTime.Now;

    trxScope.Complete();  // 確認交易。
}
```

此外，ASP.NET 應用程式經常會用 `Http.Web.HttpContext.Current` 來取得目前的 `HttpContext` 物件。這也是一個常見的例子。

範例程式（一）

如前面提過的，**Ambient Context** 模式可用於程式特定執行範圍內共享物件狀態，此「特定範圍」可以是整個應用程式、特定執行緒、或其他自訂的執行範圍。如果是整個應用程式範圍內皆可存取的共享物件，實作起來相當容易，通常用一個公開的靜態類別和靜態屬性就能達成。例如以下程式片段：

```
public static class AppShared
{
    private static ILogger _logger = new MyLogger();

    public static ILogger Logger
    {
        get { return _logger; }
        set { _logger = value; }
    }
}
```

每當應用程式需要寫入日誌訊息時，在任何地方皆可使用如下方式達成：

```
AppShared.Logger.Info(" 請謹慎使用靜態變數和全域變數。");
```

範例程式（二）

這裡再提供一個範例，示範如何實作一個依個別執行緒（per thread）共享物件資訊的 **Ambient Context** 類別。此類別會使用 .NET Framework 4.0 之後提供的 `ThreadLocal<T>` 來保存個別執行緒的狀態資訊。

令此 **Ambient Context** 類別名稱為 `PerThreadContext`，而且它要提供一個靜態的 `Current` 屬性，供外界取得當前的 context 物件。如此一來，用戶端程式可以透過以下方式取得當前執行緒 context 中的共享物件：

```
var obj = PerThreadContext.Current.SomeMember;
```

`PerThreadContext` 類別的程式碼如下：

```
public class PerThreadContext
{
    // 用一個靜態的 ThreadLocal<T> 來管理各執行緒的 context 物件。
    private static ThreadLocal<PerThreadContext> _threadedContext;

    static PerThreadContext()
    {
        _threadedContext = new ThreadLocal<PerThreadContext>();
    }

    // 共享的狀態
    public DateTime OnceUponATime { get; set; }

    // 把建構函式宣告為私有，不讓外界任意 context 物件。
    private PerThreadContext()
    {
        OnceUponATime = DateTime.Now;
    }

    public static PerThreadContext Current
    {
        get
        {
            // 如果目前的執行緒中沒有 context 物件...
            if (_threadedContext.IsValueCreated == false)
            {
                // 就建立一個，並保存至 thread-local storage。
                _threadedContext.Value = new PerThreadContext();
            }
            return _threadedContext.Value;
        }
    }
}
```

這裡使用了延遲初始化（lazy initialization）的技巧：當用戶端程式透過靜態屬性 `Current` 取得當下的 context 物件時，先檢查目前的執行緒中有沒有 context 物件，有則直接傳回物件參考，若沒有，便建立一個，並保存至目前執行緒專屬的儲存區（thread-local storage）。其中的公開物件屬性 `OnceUponATime` 代表要與其他物件共享的狀態。

我們可以用一個簡單的 Console 程式來觀察其運作機制：

```
static void Main(string[] args)
{
    ShowTime();
    System.Threading.Thread.Sleep(2000);

    var t1 = new Thread(ShowTime);
    var t2 = new Thread(ShowTime);

    t1.Start();
    System.Threading.Thread.Sleep(2000);
    t2.Start();
    System.Threading.Thread.Sleep(2000);

    ShowTime();

    /* 執行結果：
       Thread 1: 2014/5/4 下午 01:37:09
       Thread 3: 2014/5/4 下午 01:37:11
       Thread 4: 2014/5/4 下午 01:37:13
       Thread 1: 2014/5/4 下午 01:37:09
     */
}

static void ShowTime()
{
    Console.WriteLine("Thread {0}: {1} ",
        Thread.CurrentThread.ManagedThreadId,
        PerThreadContext.Current.OnceUponATime);
}
```

執行結果顯示，同樣是印出 `PerThreadContext.Current.OnceUponATime` 屬性值，不同的執行緒會有不同的結果。

在決定使用 **Ambient Context** 模式之前，請先考慮「建構式注入」和「屬性注入」是否可行。若採用注入的方式會導致過多相依物件入侵應用程式的各層 API，那表示太多地方都需要它了，該物件很可能屬於基礎建設或橫切面的功能，此時便可考慮使用 **Ambient Context** 模式。

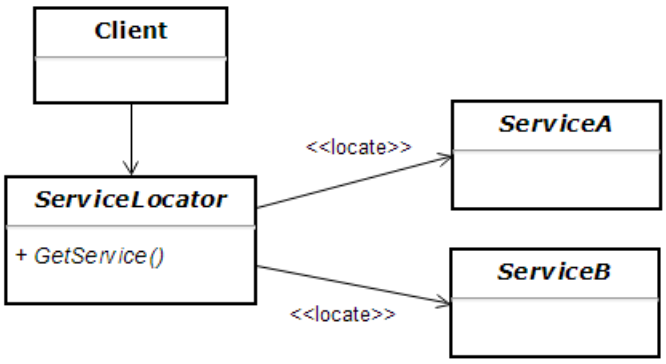
Service Locator 模式

Service Locator（服務定位器）是一種設計模式，它同時具有前面提過的 **Ambient Context** 和 **Factory 模式**的性質，而且經常與 DI 搭配使用（儘管頗具爭議），故在此一併介紹。

顧名思義，**Service Locator** 的功能是用來尋找應用程式所需的服務，並返回該服務的執行個體。說得更具體些，當用戶端需要特定介面（或抽象類別）的物件時，既不使用 `new` 來建立物件，也不使用注入物件的機制，而是向 **Service Locator** 要一個物件。其基本運作機制如下：

1. 用戶端向 **Service Locator** 提出請求，要求一個符合 `IServiceA` 的物件。
2. **Service Locator** 透過本身的型別搜尋／對應機制來尋找符合（相容於）`IServiceA` 介面的具象類別，然後建立該類別的物件實體，並回傳給用戶端。

下圖為 **Service Locator 模式**的結構圖。



Service Locator 經常以 **Singleton** 的方式實作。這裡我採用 `static` 類別的方式來實作一個極陽春的 **Service Locator**。你也可以將它視為全域共享的 **Ambient Context**。類別名稱就叫做 `ServiceLocator`，程式碼如下。

```
public static class ServiceLocator
{
      public static object GetService(Type requestedType)
      {
          if (requestedType is IMessageService)
         {
             return new EmailService();
         }
         else
         {
             // 略
         }
      }
}
```

第 3 章介紹自製 DI 容器時，會有比較像樣的範例。

若把先前的 AuthenticationService 範例改成使用此 ServiceLocator 來取得符合 IMessageService 介面的服務，程式碼會像這樣：

```
class AuthenticationService
{
    private readonly IMessageService msgService;

    // 原本使用建構式注入
    public AuthenticationService(IMessageService service)
    {
        this.msgService = service;
    }

    // 現在改用 Service Locator
    public AuthenticationService()
    {
        this.msgService = ServiceLocator.GetService(IMessageService);
    }
}
```

由於這種寫法太方便了，我們甚至可能懶得在建構函式中取得物件參考並保存至私有變數，而變成在程式中的任何地方、任何時候呼叫 `ServiceLocator` 來取得物件。然而，這裡有兩個問題必須注意。

首先，程式的語意變得比較隱晦，因而增加理解上的困難。進一步說，用戶端由於不再需要傳入相依物件至類別的建構函式，所以「瞄一眼建構函式就知道類別依賴哪些型別」的優點已經消失。同樣地，從用戶端程式碼也通常不容易看出此類別需要哪些相依物件。換言之，**Service Locator** 模式把物件實體化的相關資訊都隱藏起來了；這也是前面提過的，**Service Locator** 具有 **Factory** 性質的原因。

第二個問題是，`AuthenticationService` 原本使用「建構式注入」時並未依賴任何實作類別，改用 **Service Locator 模式**之後，卻依賴特定的具象類別 `ServiceLocator` 了。若推而廣之，在應用程式中大量使用此模式來取得相依物件，就會變成到處都依賴這個 `ServiceLocator` 類別。這於種大量依賴同一個類別的情形，如果該類別不是自己寫的，而是採用第三方元件，就得更慎重考慮其穩定性，以及是否會增加日後維護的麻煩。

基於上述兩個原因，許多人建議 **Service Locator** 少用為妙。Mark Seemann 甚至直接把這種用法歸類為「反模式」（anti-pattern）[12]。

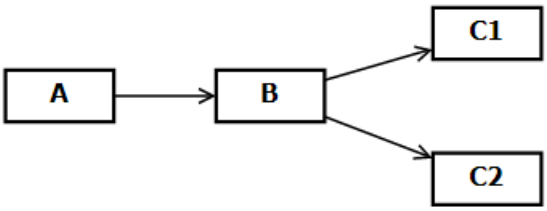
過猶不及－再談建構式注入

經過前面的說明，您已經知道三種注入相依物件的方式，以及它們的適用時機。一般的建議是優先考慮採用「建構式注入」，而此注入方式的最大好處是我們只要瞄一眼建構函式就能知道類別依賴哪些外部元件。然而，在實際運用時，「建構式注入」還有些細節必須注意，否則可能反而寫出更難維護的程式碼。本節的用意即在點出常見的陷阱與迷思，並嘗試提供一些解決問題的辦法與思考方向。

[12]《Dependency Injection in .NET》by Mark Seemann. Manning 2011.

半吊子注入

先以一個比較抽象的例子來說明：類別 A 會用到類別 B，而類別 B 會用到類別 C1 和 C2。為了避免範例程式太複雜而失去焦點，我刻意省略定義介面的部分。閱讀範例程式碼的時候，請著重在注入物件的方式，而毋需計較哪個地方的型別應該宣告為介面或抽象類別。下圖為各類別的關係。



為了讓 B 可以切換 C1 與 C2 的實作，故於類別 B 的建構函式注入這兩個物件。程式碼如下：

```
class B
{
    private readonly C1 objC1;
    private readonly C2 objC2;
    public B(C1 c1, C2 c2)
    {
        this.objC1 = c1;
        this.objC2 = c2;
    }

    public void ExecuteTask()
    {
        objC1.Task();
        objC2.Task();
    }
}
```

類別 B 本身並不建立 C1 和 C2 物件，而是將此責任往外拋給高層模組，由呼叫端將相依物

件傳入 B 的建構函式。於是類別 A 在使用類別 B 的時候，就必須提供 C1 和 C2 物件。然而我們也不想讓 A 和 C1、C2 綁太緊，於是 A 的建構函式也提供對應的參數，讓更高層的模組來提供 C1 與 C2 物件。類別 A 的程式碼如下：

```
class A
{
    private readonly B objB;

    public A(C1 c1, C2 c2)
    {
        // 建立 B 物件時，將外界注入的 c1 和 c2 再注入至 B 的建構函式。
        this.objB = new B(c1, c2);
    }

    public void ExecuteTask()
    {
        objB.ExecuteTask();
    }
}
```

類別 C1 和 C2 的實作程式碼與重點無關，故此處不列出它們的程式碼。

你可以看到，由於 B 要求注入 C1 與 C2，所以 A 的建構函式也同樣要求外界注入 C1 與 C2。問題是，A 本身完全沒有呼叫 C1 與 C2 的任何方法，而只是單純傳遞給 B 的建構函式。換句話說，A 之所以需要知道 C1 與 C2，完全是為了別人，是為了提供下一層模組所需之物件。在更複雜的場合中，甚至有可能下一層也同樣只是讓這些物件「過家門而不用」，轉手又交給下一層。

想像一下，如果下層模組需要注入的物件總數多達 10 個以上，類別 A 的建構函式不就得傳入 10 個以上的參數嗎？這裡顯然不對勁——相依關係疑似過度向外蔓延了。

此外，也許你已經發現另一個可疑之處：A 既然不想要跟 C1 和 C2 緊密耦合，卻直接使用 `new B()` 的方式直接與類別 B 綁在一起。這種寫法，我戲稱為「半吊子注入」。

阻止相依蔓延

針對上述情境，一種可能的修改方式為：在某一層（某個類別）中阻止相依物件繼續向上（外）蔓延。以此例來說，可能就是由 A 或 B 來負責建立 C1 與 C2。假設我們決定把這責任交給類別 A，修改後的程式碼如下：

```
class A
{
    private readonly B objB;

    public A()
    {
        var c1 = new C1();
        var c2 = new C2();
        this.objB = new B(c1, c2);
    }
}
```

如此一來，類別 A 便阻斷了相依物件的向上蔓延，即 A 的上一層呼叫端不再需要為下層模組建立相依物件，同時也不會知道下層模組需要什麼物件。換句話說，此寫法意味著：

- 類別 A、B、C1 和 C2 四者形成一個黑箱，上一層模組只知道 A，而不知有 B，無論 C1、C2。類別 A 的角色如同一個對外的窗口，有可能實作成 **Façade** 物件。
- 高層模組失去了切換 C1、C2 實作類別的機會，也失去了組合物件的權利，因為 A 完全掌控了 B 和 C1、C2 之間的關係。

當你很篤定某一層模組當中的物件不需要動態切換實作，就不見得要使用 DI 來注入相依物件。而且，從另一個角度來看，這種寫法在某種程度上提供了更好的封裝性。

解決「半吊子注入」

現在假設我們有很好的理由（例如單元測試需要切換別的替身物件），需要由高層模組來動態決定 C1 與 C2 的實作類別。那麼，剛才的「阻止相依蔓延」策略便無法解決「半吊子

注入」的問題了。比較可能的作法，是修改 A 的建構函式，使其改為注入 B 物件。像這樣：

```
class A
{
    private readonly B objB;

    public A(B b)
    {
        this.objB = b;
    }

    public void ExecuteTask()
    {
        objB.ExecuteTask();
    }
}
```

那麼，C1 和 C2 物件從何而來呢？答案是來自 A 之上的更高層模組。在此範例中，A 之上的更高層模組就是程式的進入點，也就是 Main 函式。程式碼如下：

```
class Program
{
    public static void Main()
    {
        // 在這裡把所有需要的物件一次組合好
        var c1 = new C1();
        var c2 = new C2();
        var b = new B(c1, c2);
        var a = new A(b);
        a.ExecuteTask();
    }
}
```

如此一來，各模組所需之相依物件都是在應用程式啟動時建立並組合完成。這個用來組合物件的集中處所，就是所謂的 **Composition Root**（組合根）。

Composition Root（組合根）：用來組合相依物件的集中處所，通常是應用程式初始化的地方。例如，Console 應用程式的組合根會在 `Main` 函式；ASP.NET 應用程式則可能是 `Global.Application_Start` 方法。

再次提醒，我的意思並不是建議讓所有的相依物件都一律從外部注入，而不去衡量這麼做是否有其必要，以及能否產生實質效益。重點在於，當你使用「建構式注入」時，如果覺得程式碼看起來怪怪的，那最好檢查一下，是否真有什麼怪味道（bad smells）。過與不及都不好，如果不管三七二十一，讓所有的相依物件一律從外部注入，則可能產生另一種狀況：**過度注入**。

過度注入

一旦大量使用「建構式注入」，便可能碰到一個現象：某些類別的建構函式需要注入的物件數量特多，多到讓你覺得有壞味道（bad smells）。例如以下程式片段，建構函式需要傳入 5 個相依物件：

```
class AuthenticationService
{
    public AuthenticationService(
        IMessageService service,
        IUserRepository userRepo,
        IUser user,
        IValidator validator,
        ILogger logger)
    {
        // 略
    }
}
```

當然，傳入建構函式的參數要多到什麼程度才會讓人不自在，每個人或每個開發團隊都有自己的界線。有的人可能只接受最多五個相依物件，有的人可能覺得三個就太多了。主要的問題是，當注入的物件數量太多時怎麼辦？

此問題的解法不只一種，條列如下：

1. 改用「屬性注入」和「方法注入」。
2. 使用 **Factory 模式**、**Ambient Context**、或 **Service Locator 模式**。
3. 將建構函式的多個參數重構成 **Parameter Object**（參數物件）。
4. 多載建構函式（overloaded constructors）。
5. 重構成 **Façade** 模式。

第一種解法，是將一部分或全部的相依物件改用其他方式注入，例如「屬性注入」和「方法注入」。如先前提過的，這兩種注入方式各有其適用時機，雖然不見得能夠紓解建構函式參數過多的問題，但至少能讓你再想一下，那些物件是否真的都有必要在物件初始化的時候傳入，抑或其實有些物件只是臨時需要罷了。

第二種解法所涉及的設計模式都已經在前面介紹過，這裡就不再重複相關細節。簡單地說，**Factory 模式**可讓你將「建立特定物件或某族系物件的責任」移轉給特定的工廠類別，從而減少注入物件的程式碼；甚至於，你可以結合 DI 與 **Factory**：不使用特定工廠類別來建立相依物件，而是將抽象工廠注入你的類別。相依物件如果有「多處共享」或橫切面（cross-cutting）服務的性質，則可考慮 **Ambient Context**，甚至 **Service Locator** 模式。

其餘三種解法比較難用三言兩語交代完，故以接下來的幾個小節分別說明。

重構成參數物件

第二種解法是利用重構技巧 **Parameter Object**（參數物件）來將這些傳入參數包在一個類別裡面。例如稍早的例子：

```
class AuthenticationService
{
    public AuthenticationService(
        IMessageService msgService,
        IUserRepository userRepo,
        IUser user,
        IValidator validator,
        ILogger logger)
    {
        this.messageService = msgService;
        this.userRepository = userRepo;
        this.user = user;
        this.validator = validator;
        this.logger = logger;
    }
}
```

經過重構之後，可能會新增一個參數類別，叫做 `AuthenticationParameter`。於是，原本的建構函式會變成只傳入一個參數：

```
class AuthenticationService
{
    public AuthenticationService(AuthenticationParameter param)
    {
        this.messageService = param.MessageService;
        this.userRepository = param.UserRepository;
        this.user = param.User;
        this.validator = param.Validator;
        this.logger = param.logger;
    }
}
```

如此一來，傳入建構函式的參數「看起來」就只有一個了。

多載建構函式

當類別的建構函式需要注入太多相依物件時，另一個常見的解法，是利用多載（overloaded）建構函式，並由類別本身提供預設的相依物件，藉此提供用戶端一點

彈性。也就是說，定義兩個或兩個以上的建構函式，讓用戶端可以更彈性地選擇要注入哪些相依物件。比如說，預設建構函式由於不帶任何參數，所以當外界使用預設建構函式來建立類別的執行個體時，此類別將在內部自行建立所有的相依物件。

以下程式片段示範了此注入方式的寫法：

```
class AuthenticationService
{
    private readonly IMessageService msgService;
    private readonly ILogger logger;

    // 預設建構函式：自行建立所有相依物件。
    public AuthenticationService()
    {
        this.msgService = new EmailService();
        this.logger = new MyLogger();
    }

    // 可由外界提供一個相依物件的建構函式。
    public AuthenticationService(IMessageService service)
    {
        this.msgService = service;
        this.logger = new MyLogger();
    }

    // 可由外界提供兩個相依物件的建構函式。
    public AuthenticationService(IMessageService service, ILogger log)
    {
        this.msgService = service;
        this.logger = log;
    }
}
```

如此一來，用戶端便有多種選擇，可視需要使用特定版本的建構函式來建立 AuthenticationService 物件。用戶端沒有注入的物件，將會由類別本身自行提供預設實作，以確保本身的功能得以正常運行。這是此解法的一個優點。

嚴格來說，這有點走回頭路的味道。畢竟使用 DI 的目的是解耦合，減少對特定實作類別

的依賴，並將物件的生殺大權交出去。一旦採用此作法，類別本身勢必仍得依賴特定實作。取捨之間，確有值得斟酌之處。

Mark Seemann 在他的《Dependency Injection in .NET》書中將此作法視為反模式（anti-pattern），還給它取了個不太好聽的名稱：**私生注入（Bastard Injection）**。此名稱的由來是 Ayende Rahien 在他的部落格文章〈Reviewing NerdDinner[13]〉裡面對此寫法所下的一句評論：

「....if you want to do poor man's IoC, go ahead. But please don't create this bastard child.」（如果你想用窮人的 IoC，請便。但請別建立這個私生子。）

我的看法，儘管這不算是嚴謹的 DI 解法，但在某些場合，它所帶來的好處可能相對低於對可維護性的傷害。更何況，並非應用程式的任何地方都得要完全用 DI 來拆解耦合。

重構成 Façade 模式

前述幾種解決方案或重構技巧雖然能夠紓緩建構函式參數太多所產生的怪味道，但是在運用這些技巧之前，更重要的是進一步思考為什麼需要傳入這麼多物件。明確地說，問自己這個問題：

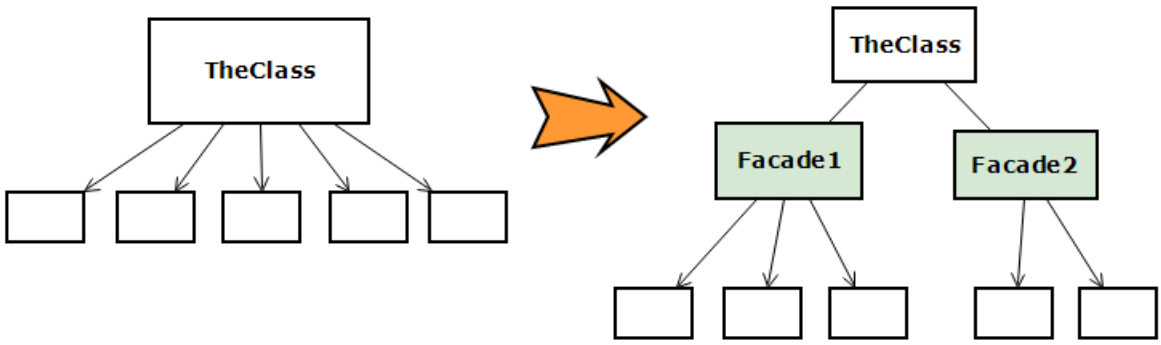
「我們是否賦予此類別太多責任，以至於嚴重違反 **SRP**（單一責任原則）?」

當一個類別經由建構函式注入的相依物件數量達到 3 個時，開始思考上述問題。若發現有依賴過多物件的跡象，則挑選適當的解法來避免未來引入更多相依物件。

若目前的設計嚴重違反 **SRP**，就該從設計面著手，採取根本的解決之道。例如使用 **Extract Class（提煉類別）**或 **Extract Subclass**（提煉子類別）等重構[14]技巧來將大型類別的責任加以細分，或搭配 **Façade 模式**來減少類別直接依賴的物件個數，如下圖所示。

[13] http://ayende.com/blog/4092/reviewing-nerddinner

[14] 參見 Martin Fowler、Kent Beck 等人合著的《Refactoring: Improving the Design of Existing Code》。繁體中文版：《重構：改善既有程式的設計》，譯者：侯捷、熊節。



由以上討論可以發現，採用「建構式注入」有個明顯的好處：你可以很容易從建構函式的參數列發現某類別的相依物件是否太多，進而評估該類別是否嚴重違反 **SRP**。

單一責任原則

第 1 章曾簡單帶過 **SRP**（Single Responsibility Principle），也就是「單一責任原則」。既然多次提及，就順便進一步談談這個原則。

SRP 的原始定義是：「要修改一個類別時，應該只基於一個原因。」換言之，若每一個物件都只負責一件工作，理想狀況下，當日後碰到需求變動時，每一個變動應該就只需要改一個類別就能解決。然而，如何界定「一個原因」和「一件工作」的範圍多大呢？也許有人會說：「是啊，修改這些程式的唯一原因就是客戶的需求又變了。」這話儘管有點開玩笑，但也的確凸顯了 **SRP** 的模糊地帶。

我認為可以從這樣的角度來看待 **SRP** 以及其他設計原則：當你碰到一個需求異動時，實際評估一下，為了因應這次需求變動，得要修改多少類別。如果只改一個類別就能收工，恭喜你！如果要改兩個以上，想想看是需求異動的幅度本身就很大，故牽連較廣，還是現有的設計有需要改進的地方（例如一個類別同時負責兩種以上性質截然不同的工作，或必須依賴五、六個商業邏輯元件才能完成任務），然後趁此機會做些適當幅度的重構。此外，在設計一個新系統時，經常檢查現有設計是否違反 **SRP** 是好事，但可別一開始就把完美

> 主義發揮到極致，要求自己第一次設計就得符合 **S.O.L.I.D.** 全部的設計原則——那很可能會導致過度設計，讓你的生產力像車輪卡在坑洞裡一樣動彈不得。隨著每一次經驗的累積來改善設計，通常更容易貼近「剛剛好的設計」。

本章回顧

讀完本章以後，你可能已經發現，DI 技術並不只是「把建立物件的工作丟給別人」這麼簡單。除了涉及各種物件導向設計原則，如 **S.O.L.I.D.**、設計模式，同時也必須知道一些重構技巧。唯有具備這些知識與技術，才能更正確地運用 DI，並選擇最適方案來解決設計上的根本問題。

本章除了概略介紹了設計模式，另一項重點就是與 DI 相關的模式與常見的陷阱與迷思。以下是幾個重點：

- 注入相依物件的基本方式有三種：建構式注入、屬性注入、方法注入。
- 一般建議盡量採用「建構式注入」，好處是很容易從建構函式的參數列得知某類別所相依的外部物件，從而協助研判該類別是否違反 **SRP**（單一責任原則）。
- 採用「建構式注入」時，可能會寫出「半吊子注入」或「過度注入」的程式碼，本章提供了幾種可能的解決方案，包括 **Factory**、**Ambient Context**、**Service Locator** 模式，以及重構成 **Parameter Object** 或 **Façade**，以及多載建構式注入等等。
- **Composition Root**（組合根）指的是用來組合物件的地方，通常是應用程式初始化的時候。例如，Console 應用程式的組合根會在 `Main` 函式；ASP.NET 應用程式則可能是 `Global.Application_Start` 方法。

下一步：雖然不需要額外工具也能使用 DI，但身為開發人員，提高生產力總是很重要的。因此，下一章便會開始介紹協助實現 DI 的工具：DI 容器。同時也會進一步討論相依物件的生命週期管理等基本觀念。

第 3 章：DI 容器

當你開始在應用程式中大量使用 DI，便會發現 DI 不只是「注入物件」那麼單純而已。在寬鬆耦合、任意切換元件等好處的背後，它同時也帶來了一些問題，以及解決這些問題所衍生的開發成本。我想多數人應會同意，與其自己寫一堆重複的程式碼來處理這些繁瑣細節，不如把寶貴時間花在其他能夠產生核心商業價值的工作上。於是，DI 容器應運而生，有了用武之地。

內容大綱：

本章範例原始碼位置：

https://github.com/huanlin/di-book-support 裡面的 Examples/ch03 資料夾。

DI 容器簡介

從實作的角度來看，DI 容器（或者說 IoC 容器）指的是一套類別庫，可用來協助開發人員實現 DI，並減輕開發人員的負擔。換句話說，DI 容器就是協助實現 DI 的類別庫或框架（framework），其角色如同資料存取或日誌（logging）框架，可視為應用程式的基礎建設（infrastructure）。

那麼，DI 容器究竟提供了什麼功能來減輕開發人員的負擔呢？主要是：物件組合、物件生命週期管理、和攔截（interception）。這三項功能，物件組合是實戰應用時最先碰到的議題，而且其實上一章也已經大致提過。因此，接著我們要先複習一下物件組合的基礎概念，等到介紹完自製 DI 容器之後，再來討論物件生命週期與攔截等相關議題。

物件組合

在前兩章的範例中，我們已經看過如何將一個類別中「建立相依物件的工作」外移至高層模組；其手段主要是「**建構式注入**」（**Constructor Injection**），目的在於寬鬆耦合，而附帶效果則是釋出建立物件的控制權。方便閱讀起見，這裡再貼一遍第 2 章的範例程式片段：

```
class AuthenticationService
{
    private readonly IMessageService msgService;

    public AuthenticationService(IMessageService service)
    {
        if (service == null)
        {
            throw new ArgumentException("service");
        }
        this.msgService = service;
    }
}
```

原本 AuthenticationService 的建構函式是直接用 new 來建立特定實作類別的執行個體（例如 EmailService）。改成 DI 版本之後，這個建立相依物件的工作便移至外層了，而且外層必須把建立好的物件透過 AuthenticationService 的建構函式注入進來。

乍看之下，AuthenticationService 失去了兩項控制權：

- 建立何種物件的控制權；它只知道介面，不知是哪個特定實作。
- 何時建立物件的控制權（以及何時摧毀物件）。

然而從另一個角度來看，它的責任也減輕了；AuthenticationService 不用再綁住特定實作類別，不用擔心它們是怎麼建立、何時建立的，當然它也就不用擔心何時摧毀這些相依物件（以及它們佔據的資源有沒有釋放乾淨）。當一個類別不再負責建立與摧毀相依物件之後，類別本身便輕盈一些。

另一方面，為了移除對實作類別的依賴，AuthenticationService 引進了 IMessageService 介面。由於介面只是一份規格，是抽象的，不具備任何實作，故只要任何類別實作了 IMessageService，其執行個體便可注入至 AuthenticationService 物件。換言之，掌控物件生殺大權的高層模組便擁有更多彈性，可透過注入物件的方式來替換實作類別，就好像把一堆物件組裝成完整的軟體系統一樣。

這又引出一個新議題：當應用程式裡面有許多地方都需要組裝物件，而使得組裝物件的程式碼分散在各處，這種狀況可能會增加日後理解程式結構的困難——你可能會經常需要查找多個程式檔案來確認某個地方注入的相依物件究竟屬於哪個實作類別。因此，組合物件的動作，一般建議是盡量集中寫在應用程式的進入點或接近初始化的地方。為了方便溝通，我們將這個適合用來組合物件的地方稱之為「**組合根**」（**Composition Root**）。對 Console 應用程式而言，合適的「組合根」地點就是 Main 函式；ASP.NET 應用程式的「組合根」通常是 Global.Application_Start 方法或每一次 HTTP Request 開始的地方；Windows Forms 應用程式可能選在個別 Form 的 Load 事件處理常式；Windows Azure 應用程式則可以選在角色的 OnStart 方法中處理。

建議放在「組合根」處理的工作：

- 建立與設定 DI 容器
- 組合物件

以上所述，大多已在第 2 章提過，這裡只是扼要複習一下相關概念。實際寫程式來組合物件時，往往需要處理一些瑣碎細節，並寫出一長串看似重複的程式碼。試看底下這個程式片段：

```
static void Main()
{
    IFoo foo = new Foo();
    IBar bar = new Bar(foo);
    IDuck duck = new Duck(bar);
    IRobot robot = new Robot(duck);
    IAmLegend legend = new Legend(robot);

    // 或者以上敘述全部寫成一行：
    legend = new Legend(new Robot(new Duck(new Bar(new Foo()))));
}
```

想像一下，類似這些用來組合物件的程式碼有數十行，寫起來是否太過瑣碎？

如果可以改寫成像底下這樣，而且達到相同的目的：

```
static void Main()
{
    var container = new MagicDIContainer();
    container.RegisterAllInterfaces(); // 自動註冊所有介面與其對應之實作型別。
    var legend = container.Resolve<IAmLegend>();
}
```

這樣的程式碼你覺得如何？寫起來是不是輕鬆許多？

其中的 `MagicDIContainer` 雖然是虛構的，但「自動註冊」功能倒是真有其事。當然啦，看似神奇的技術，背後總是有些需要注意和取捨的地方，這些相關細節都會在稍後進一步討論。這裡想要凸顯的是 DI 容器能夠在「組合物件」這項工作上提供開發人員什麼好處。

自製 DI 容器

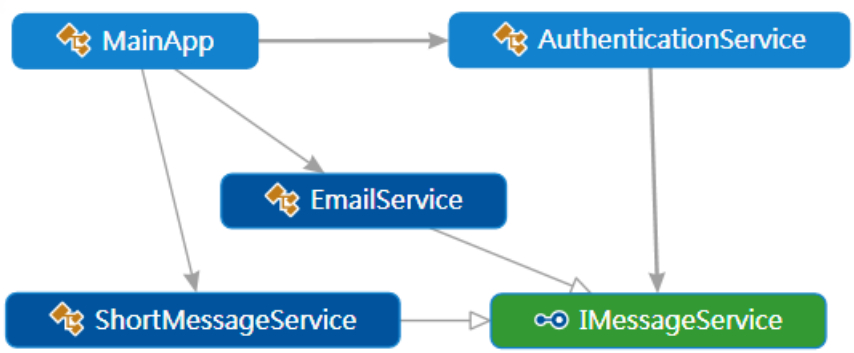
大致了解 DI 容器所涉及的主要議題之後，接下來要看看如何自製簡易的 DI 容器。其目的不是鼓勵您自己寫一套 DI 容器，而是希望透過這些實作範例讓您了解 DI 容器的基本功能與運作方式。一旦了解背後的機制，將來直接使用現成工具時，往往更得心應手。倒不是說，非得知道怎麼修車才能開車上路，但如果你具備汽車變速箱和剎車系統等相關知識，開車的時候可能就會更注意一些細節操作，以減少誤動作的機會，並減緩機件的損耗（於是降低了維護成本）。

此外，可能還有一些別的原因令你想要自己寫一個 DI 容器，例如：

- 政策限制。有些團隊或軟體專案對於第三方元件的使用採取比較嚴格的限制，以盡量減少軟體專案對外來元件的依賴。
- 現成的 DI 容器和框架都太複雜了、不合用，或執行效能不佳。
- 練功，作為深入研究其他 DI 容器的入門磚。
- 我不想用別人寫的東西，程式還是自己寫的好。**（請自行刪除這個理由）**

基於上述原因，本節的內容對你或許仍有些閱讀價值。

接下來，我要繼續使用第 1 章的範例來示範如何自行設計簡易 DI 容器。先回顧一下第 1 章最後修改完成的 DI 範例程式的相依關係圖。



第 1 章範例程式改成 DI 版本之後的相依關係圖

請留意圖中的 AuthenticationService 並未依賴實作類別 EmailService 和 ShortMessageService，而只依賴 IMessageService 介面。程式執行時，AuthenticationService 實際會使用哪個實作類別，係由其外層（高層模組）的 MainApp 來決定；說得更仔細點，MainApp 的 Login 方法還同時肩負了物件組合的任務。所以從相依關係圖中也能清楚看到，MainApp 相依於三個型別，即 AuthenticationService、EmailService、和 ShortMessageService。

在第 1 章的〈入門範例—DI 版本〉一節結束前曾經提到，MainApp 類別的 Login 方法有個缺點：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。方便閱讀起見，這裡再貼一遍先前的範例程式碼：

```
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))
        {
            // 與主題無關，故省略。
        }
    }
}
```

想像一下，如果欲支援的訊息服務類別有十幾種，那個用來處理型別對應和建立物件的 `switch...case` 區塊不就顯得冗長累贅嗎？為了解決這個問題，我們得稍微重構一下目前的程式碼，把「型別對應與建立物件」的工作移出去，交給另一個類別來負責。這個新的類別，姑且就將它命名為 `MyDIContainer`，即自製 DI 容器的意思。程式碼如下：

```
// 自製 DI 容器 v1 （範例程式的專案名稱：Ch03.MyDIContainerV1）
public static class MyDIContainer
{
    private static Dictionary<string, Type> typeMap; // 用來保存型別對應表。

    static MyDIContainer()
    {
        // 建立型別對應表。
        typeMap = new Dictionary<string, Type>();
        typeMap.Add("EmailService", typeof(EmailService));
        typeMap.Add("ShortMessageService", typeof(ShortMessageService));

        // 若往後需要增加其他型別對應，可以加在這裡。
    }

    public static object Reslove(string typeName)
    {
        // 查表，取得型別名稱所對應的型別物件。
        var resolvedType = typeMap[typeName];

        // 利用 reflection 機制來呼叫型別的預設建構函式，以建立物件。
        object instance = Activator.CreateInstance(resolvedType);
        return instance;
    }
}
```

程式說明：

- 由於只是示範用途，不需要多個容器物件，也不需要切換多種容器，故簡單設計成 `static` 類別。
- 型別對應和建立物件的功能，一般統稱為「型別解析」（**type resolving**），故 `MyDIContainer` 使用一個 `Resolve` 方法來負責處理這件工作。
- `Resolve` 方法會根據呼叫端傳入的型別名稱（是個字串）找到對應的實際型別，然後利用反射（reflection）機制來建立該型別的執行個體。其中的「字串—型別」對應資訊，是儲存在一個 `Dictionary<string, Type>` 集合物件中，並由類別的靜態建構函式來初始設定此對應表的內容。

於是，原先的 `MainApp.Login` 方法中的 `switch...case` 區塊可以簡化成這樣：

```
    var msgService = (IMessageService) MyDIContainer.Reslove(messageServiceType);
```

這行程式碼是利用自製 DI 容器的 `Resolve` 方法來建立物件，而物件的型別名稱是個字串，由呼叫端透過 Login 方法的參數 `messageServiceType` 傳入。

> **DI 容器與 Factory 模式**
>
> DI 容器提供了建立物件的功能，**Factory 模式**（參見第 2 章）也是。兩者的主要差異在於，**Factory 模式**是用來建立特定類型或同一族系的物件，而 DI 容器可建立任何型別的物件，只要型別對應表中存在對應的型別即可。

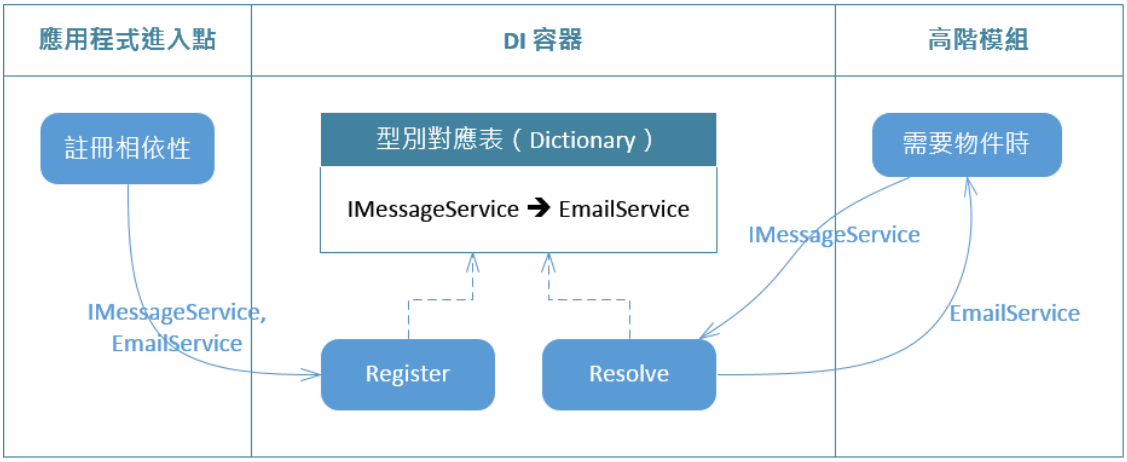
這只是重構的第一步。程式碼的修改幅度不大，是為了讓你更容易看到哪些程式碼移到了新類別中的哪些地方，而新類別又增加了什麼方法來提供型別對應和建立物件的功能。其中一個明顯的缺點是，由於此 DI 容器在建構函式中初始設定了型別對應表，綁住了特定實作類別，故只能給特定場合使用。因此，下一個重構的目標就是把這部分的耦合關係拆出去，讓 DI 容器變成通用的工具類別，以便重複使用。

自製 DI 容器—2.0 版

第一個版本的自製 DI 容器只支援特定實作型別（`EmailService` 和 `ShortMessageService`），幫助有限。為了讓它更通用些，我們可以做下列改進：

- 將「設定型別對應」的工作改由一個公開方法來實作，要求呼叫端提供欲對應之型別。這個用來「註冊型別對應」的方法，就姑且命名為 `Register` 吧。
- 型別對應的形式，目前是「字串—實作類別」，即透過字串來對應至實作類別。在程式中使用固定的字串有個問題：若打錯字，編譯器檢查不到。因此，接著要將型別對應的形式改成「抽象型別—實作類別」。

在閱讀程式碼之前，先來看一下這個自製 DI 容器的運作機制，以便大致了解此自製 DI 容器的用法。如下圖所示。



自製 DI 容器的用法示意圖

修改後的 MyDIContainer 類別如下：

```
// 自製 DI 容器 v2 （範例程式的專案名稱：Ch03.MyDIContainerV2）
public static class MyDIContainer
{
    static readonly Dictionary<Type, Type> typeMap =
        new Dictionary<Type, Type>();

    public static void Register<TypeToResolve, ConcreteType>()
    {
        typeMap[typeof(TypeToResolve)] = typeof(ConcreteType);
    }

    public static TypeToResolve Resolve<TypeToResolve>()
    {
        Type concreteType = typeMap[typeof(TypeToResolve)];
        Object instance = Activator.CreateInstance(concreteType);
        return (TypeToResolve)instance;
    }
```

```
}
```

程式說明：

- Register 是個泛型方法，用來供呼叫端註冊一項型別對應資訊。呼叫端可透過泛型參數 TypeToResolve 和 ConcreteType 來分別指定當應用程式需要型別 TypeToResolve 時，此 DI 容器實際上會建立 ConcreteType 的執行個體。
- Resolve 現在也是泛型方法了。當應用程式需要建立 TypeToResolve 型別的物件時，便可呼叫此方法。

使用此版本的 DI 容器，外層呼叫端必須先呼叫 DI 容器的 Register 方法來註冊型別對應。此處 MainApp 的外層呼叫端就是整個 Console 應用程式的進入點，亦即 Main 函式。程式碼如下：

```
class Program
{
    static void Main(string[] args)
    {
        // 向 DI 容器註冊型別對應。
        MyDIContainer.Register<IMessageService, ShortMessageService>();

        var app = new MainApp();
        app.Login("michael", "1234");
    }
}
```

如此一來，物件型別解析的工作改由 DI 容器負責，故 MainApp.Login 方法的第三個字串參數 messageServiceType 可以拿掉，並簡化成這樣：

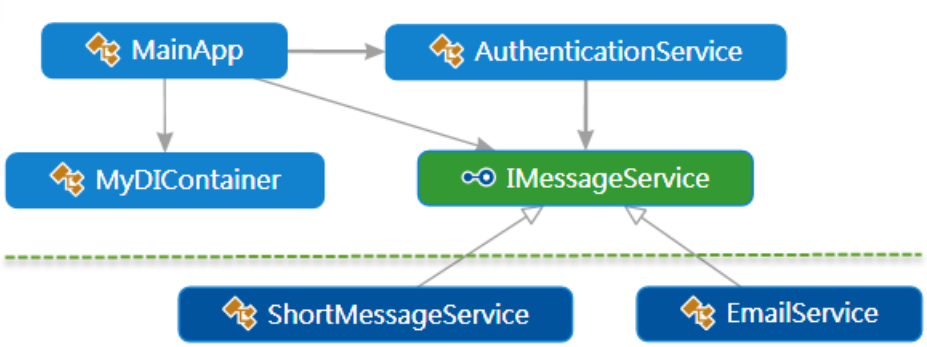
```
class MainApp
{
    public void Login(string userId, string pwd)
    {
        var msgService = MyDIContainer.Resolve<IMessageService>();

        var authService = new AuthenticationService(msgService);  // 注入相依物件。
        if (authService.TwoFactorLogin(userId, pwd))
        {
            // 與主題無關，故省略。
        }
    }
}
```

其中的 `MyDIContainer.Resolve<IMessageService>()` 呼叫會傳回一個相容於 `IMessageService` 介面的物件，並將此物件參考指派給變數 `msgService` 來保存。當變數 `msgService` 不在當前程式執行區塊的有效範圍時，亦即離開 `Login` 方法之後，它所參考的物件個體也就成為可回收的資源了。

另外，也許您已經發現了，這裡的 `MainApp` 類別並未要求外界注入任何物件，而是自行透過 DI 容器來取得符合 `IMessageService` 介面的物件。像這種在應用程式進入點（或某高層模組）預先向 DI 容器註冊型別，然後等到程式中某處需要特定型別的物件時，再透過 DI 容器的 `Resolve` 方法來建立物件的寫法，等於是把 DI 容器當成 **Service Locator**（服務定位器）來使用。你可以把這裡的 `MyDIContainer` 視為第 2 章的 `ServiceLocator` 範例的進化版。

再次提醒，把 DI 容器——尤其是全域的靜態 DI 容器——單純當作 **Service Locator** 來用，會讓程式碼不易閱讀和理解。你可能會因為 DI 容器的 `Resolve` 方法太方便了，而逐漸習慣在程式中任何需要物件的地方都給它「解析」一下。長此以往，「解析特定抽象型別」的程式碼四處分散，就很難單單透過閱讀程式碼來了解某類別究竟依賴了哪些外部元件。

目前為止，我們的自製簡易 DI 容器已大致完成，而且整個拼圖的各個部分都已逐一看過。現在來看看加入 DI 容器後，各型別之間的相依關係，以獲得全局概觀，如下圖。



相依關係圖

進入下一個議題之前，再整理本小節的幾個重點（如果仍有不清楚的地方，請回頭參照程式碼和相依關係圖）：

- 高層模組 `MainApp` 和 `AuthenticationService` 不直接依賴實作類別 `EmailService` 和 `ShortMessageService`，而只依賴 `IMessageService` 介面（可視為抽象層）。屬於低層模組的實作類別同樣也只依賴抽象層的介面，而不依賴高層模組。這符合了先前提過的 **DIP**（相依反轉原則）。
- 主程式的進入點 `Main` 函式會向 DI 容器（`MyDIContainer`）註冊 `IMessageService` 所欲對應之實作類別，然後 `MainApp` 會透過 DI 容器的 `Resolve` 方法來建立發送訊息的物件，並將此物件注入 `AuthenticationService` 物件（建構式注入）。
- 當我們要更換驗證碼的發送方式時，只要修改 `Main` 函式的第一行程式碼，將註冊的實作類別替換掉就行了。
- 如先前提過的，註冊型別的程式碼是寫在應用程式的進入點（`Main` 函式），統稱為 **Composition Root**（組合根）的地方。

本節的目的是介紹 DI 容器在實作「型別註冊」與「型別解析」時的一種常見作法，而不是要提供一個面面俱到、功能強大的元件，故對於物件生命週期管理以及其他涉及許多複雜實作細節的部分皆略過不提。這些進階議題會在介紹現成 DI 容器之後進一步說明。

現成的 DI 容器

目前既有的 DI 容器可謂百花齊放，競爭激烈。在重新發明輪子之前，不妨優先考慮現成的工具。實際用過幾個現成的 DI 容器之後，應該會更清楚 DI 容器幫你處理掉哪些繁瑣細節，以及它們的優點和使用上的限制。

以下列舉幾個比較常見的 DI 容器（依字母順序排列）：

- Autofac
- Ninject
- Spring.NET
- StructureMap
- Unity
- Windsor

MEF 是 DI 容器嗎？

MEF（Managed Extensibility Framework）是用來協助開發人員設計高度彈性、容易組合的應用程式，而其中一種最常見的應用就是讓軟體系統具有自動加載附加元件（plugin）的能力。比如說，應用程式部署之後，可將其他人開發的第三方元件的 .DLL 檔案複製到應用程式的安裝目錄下（也許是 bin 或 plugins 子目錄），等到下次重新啟動應用程式時，便會自動發現這些新的附加元件，並自動載入。為了實現這種自動偵測與加載外掛元件的能力，MEF 便與 DI 容器有了重疊的地方，也因此常有人問：「MEF 究竟是不是 DI 容器？」

這裡引述 Glenn Block（MEF 的計畫經理）在接受 Scott Hanselman 訪問時所說的一段話來回答這個問題：「...兩者主要的差異在於，IoC 容器是在管理一群已知的東西....而 MEF 其實是在管理未知的東西。」

如欲取得這段訪問的錄音與原文，可透過此網址：http://tinyurl.com/mef-block

至於如何選擇 DI 容器，各人著重的面向不同，也許是執行效能、也許是功能強大與否（有些人喜歡輕量級元件），又或者是團隊既定的政策。因此，我無法在這裡告訴你哪一個 DI 容器最適合你。另一方面，我也沒有完全用過上列每一種 DI 容器，不能空口說白話。

如果你想知道各家 DI 容器的差別，除了針對自己感興趣的功能，實際寫程式測試一下，另一個比較快的方法是上網搜尋。我發現 Daniel Palme 有維護一份各家 DI 容器的效能比較表：IoC Container Benchmark - Performance comparison[15]，可以參考看看。我從這篇文章摘出上列 DI 容器的功能差異，再加上我自己上網查證的結果，整理成一張簡表（結果與 Palme 的文章略有出入），如下圖所示。

容器	效能	設定方式			功能			環境		
		程式碼	XML	自動	自動匹配	自訂生命週期	攔截	.NET	WP8	WinRT
AutoFac	中等	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ninject	慢	✓	✓	✓	✓	✓	✓	✓	✗	✓
Spring.NET	很慢	✓	✓	✗	✓	✗	✓	✓	✗	✗
StructureMap	中等	✓	✓	✓	✓	✓	✓	✓	✗	✓
Unity	中等	✓	✓	✓	✓	✓	✓	✓	✓	✓
Windsor	中等	✓	✓	✓	✓	✓	✓	✓	✗	✗

DI 容器功能比較表（更新日期：2014-06-16）

說明：

- 「設定方式」指的是設定 DI 容器的方式，包括：以程式碼來設定、使用 XML、以及自動設定。設定的內容則主要是型別註冊的相關資訊。
- 「自動匹配」的英文是 auto-wiring，指的是 DI 容器在建立元件的執行個體時，會根據目前的設定來尋找合適的建構函式。
- Autofac 目前的版本是否具備「自訂生命週期」功能，由於每個人對此功能的認知和定義不盡相同，看法也不一致，故特別以橘色勾來標示。

[15] http://www.palmmedia.de/Blog/2011/8/30/ioc-container-benchmark-performance-comparison

除了「設定方式」和「自動匹配」（auto-wiring）之外，「攔截」與「自訂生命週期」也都是 DI 容器的重要功能。接下來，我們要透過一些簡短的範例來了解 DI 容器提供的主要功能和基本用法。

物件組合

經由前面介紹的自製 DI 容器範例，我們知道 DI 容器內部有個型別對應表，而當應用程式向 DI 容器註冊型別對應關係（抽象型別對應至實作類別）時，DI 容器便將此對應關係保存於內部的型別對應表，以便將來解析型別之用。「解析」這個動作即在根據用戶端要求之型別來取得對應的實作類別，並建立其執行個體。物件使用完畢之後，最終會在某個時機進行釋放資源的動作。因此，**「註冊、解析、釋放」**這三個步驟皆與物件組合息息相關，而 DI 容器也提供了相應的服務來完成這些工作。

釋放物件資源的部分，會在稍後討論物件生命週期時說明。這裡要先介紹的是與註冊和解析有關的基本用法。大體而言，為了滿足各種需求，DI 容器提供的設定方式有三種：使用 XML 組態檔、使用程式碼、以及自動註冊。

使用 XML

使用 XML 來設定 DI 容器的主要優點是只要修改檔案內容就能改變用應用程式的行為，而無須重新編譯程式碼。這項優點卻也附帶了缺點：萬一在 XML 組態檔中打錯字（例如類別名稱寫錯），編譯器沒辦法幫你抓出來；而即使型別名稱都寫對，也可能在搜尋路徑中找不到相應的 DLL 組件，而造成應用程式執行到某處時才突然異常終止。XML 本身的另一個問題是語法格式比較不適合人眼閱讀，而更適合用程式處理。無論如何，XML 是目前相當普遍的格式，擁有眾多工具的支援，仍有其方便之處。

有些 DI 容器使用自訂的 XML 組態檔（例如 Ninject），有些是使用標準的 .NET 應用程式組態檔（例如 Unity），有些則兩種都支援（例如 Autofac）。以下分別列出 Unity、Autofac、和 Ninject 的組態檔設定範例，讓你對於「如何使用 XML 來設定 DI 容器」先有個概念。

Unity 範例：

```
<configuration>
   <configSections>
      <section
         name="unity"
         type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
               Microsoft.Practices.Unity.Configuration" />
   </configSections>

   <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
      <namespace name="DIExamples.Ch02" />
      <container>
         <register type="IMessageService" mapTo="EmailService" />
      </container>
   </unity>
</configuration>
```

Autofac 範例：

```
<configuration>
   <configSections>
      <section name="autofac"
               type="Autofac.Configuration.SectionHandler, Autofac.Configuration"/>
   </configSections>
   <autofac defaultAssembly="DIExamples.Ch02">
      <components>
         <component type="DIExamples.Ch02.EmailService, DIExamples.Ch02"
                    service="DIExamples.Ch02.IMessageService" />
         </component>
      </components>
   </autofac>
</configuration>
```

Ninject 範例：

```
<module name="messageServiceModule">
   <bind service="DIExamples.Ch02.IMessageService, DIExamples.Ch02"
         to="DIExamples.Ch02.EmailService, DIExamples.Ch02" />
</module>
```

如果你的應用程式不需要「晚期繫結」(late binding)，亦即不需要「修改組態檔來切換元件」的能力，建議盡量以程式碼來設定 DI 容器。此外，有些場合甚至只能以程式碼來設定 DI 容器，例如 Unity 就不支援在 Windows 市集應用程式（Store apps）中使用組態檔來進行配置[16]。

> 「我的建議是一律提供程式化介面（API）來設定所有的組態，而把外部組態檔視為附加功能。如此一來，你就可以輕易透過 API 來存取組態檔。如果你正在開發軟體元件，便可讓用戶自行決定要使用 API 還是組態檔，或者他們也可以使用自訂格式的組態檔，並搭配 API 來存取檔案內容。」
>
> 摘自 Martin Fowler 的文章：Inversion of Control Containers and the Dependency Injection pattern[17]

使用程式碼

以下列舉幾個以程式碼來註冊型別的範例，其中包括建立與設定 DI 容器，以及註冊型別。同樣地，這些範例也只是先讓您有個概略印象，方便參考、比較，而不是要提供完整 API 的用法說明。

Unity：

[16] Unity v3 不支援在 Windows 市集應用程式中使用組態檔來設定，是因為 `Unity.Configuration` 組件不相容於 Windows 市集用程式。相關細節可參考 Unity 技術文件：Appendix A - Unity and Windows Store apps。

[17] http://martinfowler.com/articles/injection.html

```
var container = new UnityContainer();
container.RegisterType<IMessageService, EmailService>();
```

Autofac：

```
var builder = new ContainerBuilder();
builder.RegisterType<EmailService>().As<IMessageService>();
var container = builder.Build();
```

Ninject：

```
var kernel = new Ninject.StandardKernel();
kernel.Bind<IMessageService>().To<EmailService>();
```

無論採用 XML 還是以程式碼的方式註冊型別，在執行時期需要解析型別時，皆可透過程式碼來達成。如以下範例所示。

Unity：

```
var instance = container.Resolve<IMessageService>();
```

Autofac：

```
var instance = container.Resolve<IMessageService>();
```

Ninject：

```
var instance = kernel.Get<IMessageService>();
```

不難想像，一旦你在應用程式中大量使用 DI 容器來組合物件，就會需要寫一堆註冊型別的程式碼，例如：

```
var container = new UnityContainer();
container.RegisterType<IMessageService, EmailService>();
container.RegisterType<ILogger, FileLogger>();
container.RegisterType<IOrderRepository, OrderRepository>();
container.RegisterType<IOrderService, OrderService>();
container.RegisterType<IBlahBlahBlahService, BlahBlahBlahService>();
....
```

為了讓開發人員少寫一些程式碼，許多 DI 容器都還提供了自動註冊的功能。欲知詳情，請接著看下一節。

自動註冊

「自動註冊」的另一種說法是「依慣例註冊」（registration by convention），其目的在於讓開發人員可以少寫一些註冊型別的程式碼。說得更明白些，DI 容器的「自動註冊」功能會進行組件掃描（assembly scanning）與型別探索（type discovery）的工作，也就是依你指定的條件去掃描應用程式的某些組件（通常是 DLL），並尋找符合特定條件的型別，然後自動註冊這些型別。

以下程式片段示範了如何讓 Unity 容器自動掃描組件並註冊型別：

```
var container = new UnityContainer();
container.RegisterTypes(
    AllClasses.FromLoadedAssemblies(),   // 掃描目前已經載入此應用程式的全部組件。
    WithMappings.FromAllInterfaces);     // 尋找所有介面。
```

這段程式碼會令 Unity 容器掃描目前已經載入此應用程式的全部組件，並註冊所有已知介面的實作型別。假設此範例應用程式目前已載入的組件當中已經有一個 `ConsoleLogger` 類別，而且該類別實作了 `ILogger` 介面，如下所示：

```
public interface ILogger
{
    void Info(string msg);
}

public class ConsoleLogger : ILogger
{
    public void Info(string msg)
    {
        Console.WriteLine(msg);
    }
}
```

那麼當應用程式需要取得符合 `ILogger` 介面的物件時，由於 Unity 已經自動尋找並註冊了相關型別，故可透過先前介紹過的型別解析方法來取得物件：

```
var logger = container.Resolve<ILogger>();
logger.Info("Hello Auto-registration!");
```

以下是 Autofac 的自動註冊範例：

```
var asmb = Assembly.GetExecutingAssembly();
var builder = new ContainerBuilder();
builder.RegisterAssemblyTypes(asmb)
    .Where(t => t.Name.EndsWith("Logger")) // 指定型別篩選條件
    .AsImplementedInterfaces(); // 為找到的類別註冊它所實作的全部介面
```

此範例會尋找所有類別名稱以 “Logger” 結尾的類別，然後將這些類別與它們所實作的全部公開介面各配成一對（IDisposable 除外），並註冊到容器裡。比如說，`EmailLogger` 若同時實作了 `ILogger` 和 `IMailer` 介面，那麼當你向 Autofac 容器要求解析 `ILogger` 或 `IMailer` 時，容器都會傳回 `EmailLogger` 的執行個體。

自動或手動？隱含或明確？

只要你了解 DI 容器提供的型別探索機制和語法，就可以用短短幾行程式碼來完成註冊型別的工作。而且，如果你的介面和類別都有遵循特定慣例來命名，那麼即使日後增加了新的介面和類別，原先寫的那些自動註冊的程式碼可能不用修改就能運作。但你知道，簡單與神奇的背後往往要付出一些代價。

由於「自動註冊」需要在執行時期掃描多個組件的型別資訊，當你的應用程式包含大量 DLL 組件時，依組件數量而定，掃描型別資訊的工作可能要數秒甚至超過十秒才能完成。自動註冊的程式碼通常是寫在應用程式初始化的地方，這表示使用者可能會發現從應用程式啟動到出現第一個畫面的等待時間過長。

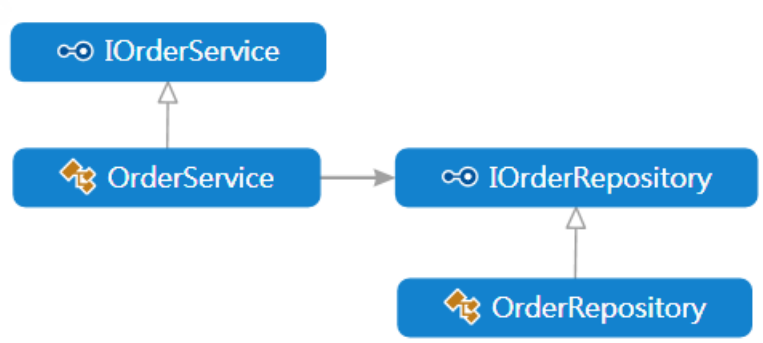
另一個問題與團隊開發有關。如果整個應用程式的架構是你設計的，你自然很清楚介面與實作類別應該按照什麼命名慣例，才能順利由 DI 容器搜尋並自動註冊。你非常清楚背後的運作機制，所以對這短短幾行自動註冊的神奇程式碼既不驚訝也不疑惑。然而其他團隊成員呢？他們需要知道某些介面實際對應到哪些實作類別嗎？如果需要，找答案的過程會很麻煩嗎？以及，你需要花多少時間向新進成員解釋背後的機制與命名慣例，並且說服大家，那短短三五行的隱含寫法確實優於撰寫數十行類似底下的明確註冊的程式碼？

```
// 撰寫明確註冊的程式碼，好處是更容易理解程式的結構。
container.RegisterType<ILogger, ConsoleLogger>();
```

若上述問題不至於對你和你的開發團隊造成困擾，那麼 DI 容器的自動註冊功能會是很好的選擇。

自動匹配

「自動匹配」（auto-wiring）或「自動選擇建構函式」指的是 DI 容器在建立特定型別的物件時，會根據容器現有的設定來使用合適的建構函式。舉例來說，假設我們正在撰寫訂單服務，類別 `OrderService` 需要從建構函式注入一個符合 `IOrderRepository` 介面的物件來存取訂單資料。各型別的關係如下圖所示：



底下是 OrderService 和 OrderRepository 類別的程式碼。此範例所要展示的重點在於，當實作類別有多個建構函式時，DI 容器將如何選擇適當的建構函式來建立相依物件，因此這裡只列出與主題相關的程式碼。

```
public class OrderService : IOrderService
{
    private IOrderRepository _orderRepo;

    // 預設建構函式。
    public OrderService()
    {
    }

    // 需要注入 IOrderRepository 物件的建構函式。
    public OrderService(IOrderRepository orderRepo)
    {
        _orderRepo = orderRepo;
    }
}

public class OrderRepository : IOrderRepository
{
    // OrderRepository 的建構函式不用傳入任何參數。
    public OrderRepository()
    {
        // 略
    }
}
```

在不使用 DI 容器的情況下，組合物件的程式碼可能類似這樣：

```
static void Main()
{
    var orderRepos = new OrderRepository();
    var orderService = new OrderService(orderRepos);

    // 接著利用 orderService 處理訂單....
}
```

由於這裡使用 new 來建立 OrderService 物件，我們可以輕易看出來，實際上呼叫的是有帶一個傳入參數的建構函式。

如果使用 Unity，程式碼可以改成這樣：

```
static void Main()
{
    // 設定 DI 容器。
    IUnityContainer container = new UnityContainer();
    container.RegisterType<IOrderRepository, OrderRepository>();
    container.RegisterType<IOrderService, OrderService>();

    // 在程式某處，使用 DI 容器來解析物件。
    IOrderService orderService = container.Resolve<IOrderService>();

    // 接著利用 orderService 處理訂單....
}
```

請注意，在設定 DI 容器時，只是註冊抽象型別與實作型別的對應關係，而等到需要處理訂單時，也只是告訴 DI 容器：我現在需要一個 IOrderService 的物件。我們並沒有特別指定要使用哪一個建構函式，但 Unity 會自動選擇那個需要傳入 IOrderRepository 參數的建構函式，而不是選擇預設建構函式。這就是 DI 容器的自動匹配機制。

顯然地，碰到多載建構函式時，Unity 自有一套規則來決定該呼叫哪一個建構函式。關於 Unity 的自動匹配規則，第 7 章的〈自動匹配〉一節有更詳細的說明。

深層解析

DI 容器大多具備自動深層解析（有時我會說「連鎖解析」）的功能，而這項基本功能是為了讓開發人員能夠更輕鬆地解決相依關係鍊（dependency chain）所衍生的物件組合問題。怎麼說呢？

以上一個小節的範例來說，`OrderService` 依賴 `IOrderRepository`，故採用建構式注入，於 `OrderService` 的建構函式注入一個 `IOrderRepository` 物件。然而，如你所見，在透過 Unity 容器來解析 `IOrderService` 時，並不需要撰寫解析與注入 `IOrderRepository` 的程式碼，而只寫了底下這樣單純的一行：

```
IOrderService orderService = container.Resolve<IOrderService>();
```

這是因為容器在解析 `IOrderService` 時，會自動進行深層解析。中間的過程大概是這樣：

1. 容器發現 `IOrderService` 有註冊過，且對應的實作類別是 `OrderService`，於是準備呼叫此類別的建構函式來建立執行個體。
2. 按照上一節介紹的「自動匹配」規則來選擇 `OrderService` 的建構函式，結果是決定使用這個版本建構函式：

   ```
   public OrderService(IOrderRepository orderRepo)
   {
       _orderRepo = orderRepo;
   }
   ```

3. 由於選擇的建構函式又需要注入一個 `IOrderRepository` 物件，所以在呼叫此建構函式之前，容器必須先嘗試解析 `IOrderRepository`。
4. 容器發現 `IOrderRepository` 有註冊過，且對應的實作類別是 `OrderRepository`，於是準備呼叫此類別的建構函式來建立執行個體。
5. 如同步驟 2，此時容器選擇的是不帶任何參數的建構函式，於是建立了 `OrderRepository` 物件，然後將此物件傳入 `OrderService` 的建構函式。接著便建立了 `OrderService` 物件，並將此物件參考傳回呼叫端。整個解析 `IOrderService` 的程序至此完成。

同樣地，如果 OrderRepository 的建構函式又需要注入其他物件，那麼容器又會先解析這些更內層的相依物件，依此類推。這樣的解析過程即所謂「深層解析」。

「深層解析」的方便之處，正如剛才提過的，寫程式時只要告訴 DI 容器需要解析哪個上層物件就夠了；DI 容器為了完成這項任務，自會想辦法把建立該物件所需注入的其他相依物件一併連帶解析。

物件生命週期管理

前面提過，當一個類別不再負責建立其相依物件，它也同時失去了摧毀物件的控制權。這不僅是基於單一責任原則，而是還有個重要的理由：如果一個類別 Foo 所依賴的物件是由外界所建立，可是 Foo 卻主動去摧毀那些相依物件，這種作法可能會造成災難，因為當某個相依物件被釋放時，可能有別的物件還需要使用它們。也就是說，同一個物件可能被注入至多個模組，由多個用戶端共享，故任何一個用戶端都不應任意嘗試釋放由外界注入的相依物件。

那麼，摧毀相依物件的工作該由誰負責？何時執行呢？

.NET 執行環境提供了資源回收的機制，能夠在適當時機釋放已經沒人使用的物件（即沒有任何變數參考到的物件），故一般而言，我們大多不用操心物件何時摧毀。需要特別注意的，是那些實作了 IDisposable 介面的類別。想必你已經很熟悉 IDisposable 介面，它只定義了一個方法：

```
public interface IDisposable
{
    void Dispose();
}
```

那些有實作 IDisposable 介面的類別通常會包含一些需要開發人員配合手動回收的外部資源（例如資料庫連線、通訊埠等等），以便讓用戶端程式透過 Dispose 方法來釋放那

些資源。DI 容器既然提供了建立相依物件的功能，故它也必須負起釋放物件資源的責任，特別是那些有實作 `IDisposable` 介面的物件——用英文的術語來說，叫做 disposable objects。這些有關物件的生與滅的處理機制，一般統稱為「物件生命週期管理」。

建立物件的部分，無論是用 `new` 運算子、**Factory 模式**、還是 DI 容器，前面都已經談得很多，這裡就不再贅述了。接下來要說明的，是比較令人頭疼的問題：如何保證物件資源獲得釋放，以避免產生記憶體洩漏的狀況。

記憶體洩漏問題

Nicholas Blumhardt 的部落格有一篇介紹 Autofac 生命週期管理的文章，標題是 An Autofac Lifetime Primer[18]。我覺得這篇文章寫得很好，故仿照其作法，寫一點小程式來觀察不同 DI 容器在處理物件資源回收的作法上有何差異。本節範例程式的專案名稱是 Ch03.TestObjectLifetime。

由於 .NET 執行環境本身已提供自動資源回收的功能，故本節探討的物件資源釋放議題，以及 DI 容器需要處理的對象，都是針對有實作 `IDisposable` 介面的物件，也就是剛才提過的 disposable objects。

假設有個類別 `Foo`，它實作了 `IFoo` 和 `IDisposable` 介面，程式碼如下（`IFoo` 的內容不重要，故未列出）。

[18] http://nblumhardt.com/2011/01/an-autofac-lifetime-primer/

```
class Foo : IFoo, IDisposable
{
    byte[] buf = new byte[1024 * 1024];

    public void Dispose()
    {
        Console.WriteLine("Entering Foo.Dispose()");
    }
}
```

Foo 類別只是單純用來消耗記憶體用的，每次建立一個 Foo 物件就會消耗 1MB 左右的記憶體。接著撰寫用來觀察 Autofac 的測試類別：

```
class TestAutofac
{
    public void Run()
    {
        var builder = new ContainerBuilder();
        builder.RegisterType<Foo>().As<IFoo>(); // 註冊型別
        var container = builder.Build();        // 建立容器

        while (true)
        {
            var obj = container.Resolve<IFoo>(); // 解析型別（建立物件）

            // 顯示此程式目前占用多少記憶體
            Process proc = Process.GetCurrentProcess();
            Console.WriteLine(" 已配置記憶體: {0} MB",
                              proc.PrivateMemorySize64 / (1024 * 1024));
        }
    }
}
```

其中的 Run 方法會先向 Autofac 容器註冊型別對應資訊，然後用一個無限迴圈來不斷取得 IFoo 的物件。預設情況下，每呼叫容器的 Resolve 方法一次，就會建立一個新的物件。我另外在迴圈裡面加了兩行程式碼來觀察目前記憶體的使用情形。

按照我們對 .NET 資源回收機制的理解，每一次迴圈結束之後，該迴圈的執行範圍內所建立之物件已經沒有被任何變數參考，所以記憶體的使用情形應該是反覆的配置然後釋放，

而得以維持在一個固定水準。但實際上執行時，才不到幾秒的時間，此範例程式就會引發記憶體不足的錯誤。這是因為每一次迴圈所建立的物件資源並未真正獲得釋放，最終導致程式的記憶體用量超過限制。

那麼，為什麼每一次迴圈所建立的 Foo 物件不會在下一次迴圈開始時釋放呢？原因在於：

預設情況下，Autofac 會追蹤任何由它所建立的 disposable objects。

而且這個追蹤的動作是深層的、是連鎖反應的（即本章稍早提過的「深層解析」）。也就是說，如果在解析 IFoo 的過程中發現實作類別 Foo 的建構函式需要注入 IBar 物件，Autofac 就會先解析 IBar 並建立 Bar 物件。同理，在解析 IBar 時，如果它的實作類別的建構函式又需要 IThud 物件，則又會先解析 IThud，依此類推。在這個一路連動解析的過程中所建立的相依物件，只要它們有實作 IDisposable 介面，Autofac 都會加以追蹤——亦即 Autofac 容器內部會有變數一直參考著那些物件，而這就是剛才的範例程式中，每一次迴圈結束時不會釋放 Foo 物件的原因。

接著來看看 Unity 是否有同樣情形。

只要稍微修改剛才的 TestAutofac 就能測試 Unity 容器是否具有相同行為。程式碼如下：

```
class TestUnity
{
    public void Run()
    {
        var container = new UnityContainer();
        container.RegisterType<IFoo, Foo>();  // 註冊型別

        while (true)
        {
            var obj = container.Resolve<IFoo>(); // 解析型別（建立物件）

            // 顯示此程式目前占用多少記憶體
            Process proc = Process.GetCurrentProcess();
            Console.WriteLine(" 已配置記憶體: {0} MB",
                              proc.PrivateMemorySize64 / (1024 * 1024));
        }
```

```
    }
}
```

結果是，`TestUnity` 的 `Run` 方法可以持續正常執行，並不會引發記憶體不足的錯誤，而且螢幕上輸出的記憶體使用狀況始終維持在一個固定水平。

由這個小實驗可以確認，**預設情形下，Unity 不會追蹤任何由它建立的物件**。說得更明白些，Unity 對「釋放物件」這件事情所採取的作法與 Autofac 截然不同：Autofac 預設會追蹤它所建立的 disposable 物件，並提供了配套機制讓開發人員能夠更細緻地控制何時釋放物件；Unity 則是預設不追蹤它所建立的物件。

另一方面，此實驗也提醒了我們，各家 DI 容器對於如何管理物件生命週期這件事可能採取不同觀點。一旦選定你想要使用的 DI 容器，就必須了解它在這方面的實作策略，以及提供了哪些進階選項來讓你更細緻地控制物件生命週期。這也就是接著要討論的議題。

生命週期選項

剛才討論的重點不是要比較孰優孰劣，而是在凸顯一個事實：當建立物件的責任轉移至 DI 容器身上，物件的生命週期管理就成為 DI 容器的一項重要工作，而且這項工作的背後還牽涉到不少細節。比如說，有些物件是每次需要時就建立一個新的，有些場合則是永遠只用同一個物件（全域共享物件）；甚至有些情境是必須在應用程式的特定執行範圍內共享同一個物件（例如每一個 HTTP 請求的範圍內共享一個資料庫連線或 `DbContext` 物件）。於是，功能比較完整的 DI 容器通常會提供多種物件生命週期的選項，以便應付各種不同的場合與需求。

以下列舉幾種常見的選項：

- 臨時的（**Transient**）：亦即每次需要時建立新的。每當用戶端要求相依物件時，便建立一個新物件，該物件的釋放時機通常由 .NET 執行環境決定，亦即無人參考時便可釋放。這種物件生命週期選項，一般稱之為「臨時的」（transient）。Autofac 稱之為 **Instance Per Dependency**。

- 唯一的（**Singleton**）：只建立單一執行個體（singleton）。類別的執行個體一旦建立，往後有其他用戶端要求相同型別時，便直接返回該物件，而不再建立新的。
- 依每條執行緒（**Per Thread**）：每一條執行緒只會有一個物件。也就是說，若同一條執行緒當中有多處要求建立某抽象型別的執行個體，DI 容器只會建立一個物件。也就是執行緒內共享一個物件的意思。
- 依每次 HTTP 請求（**Per HTTP Request**）：每一個 HTTP 請求範圍內只會有一個物件。舉例來說，在 ASP.NET 應用程式中使用 Entity Framework 來存取資料，一般建議是讓每一個 HTTP Request 擁有各自的 `DbContext` 物件。此時便很適合採用這個物件生命週期選項。

有些 DI 容器甚至支援自訂範圍。比如說，你可以要求 DI 容器總是為特定型別返回同一個物件（即該物件會一直存活），直到另一個相關的物件參考產生變化（變數指向其他物件），之後的解析動作才會另外建立新的執行個體。

以 Unity 為例，若未特別指定，其預設採用的物件生命週期範圍是 **Transient**。如欲使用特定生命週期，我們可以在註冊型別時額外傳入一個生命週期管理員。比如說，**Singleton** 生命週期管理員就是 `ContainerControlledLifetimeManager`。參考以下程式片段：

```
var container = new UnityContainer();
container.RegisterType<IFoo, Foo>(new ContainerControlledLifetimeManager());
```

有關 Unity 生命週期管理的議題，在本書的第 7 章〈Unity 學習手冊〉中有更詳細的介紹。

攔截

有時候，在呼叫某些函式時，我們會想要在這些函式開始執行之前或之後安插額外的程式碼，藉以改變該函式的行為，或提供額外的處理。例如進入函式時寫一筆 log，函式返

回時也寫一筆 log。同時，為了不違反 **SRP**（單一責任原則）或 **OCP**（開放／封閉原則），我們希望在不修改既有類別的前提之下改變其行為。**攔截（interception）**就是達成上述需求的一種方法。

繼承現有類別然後改寫其方法，也可以達到類似效果。但一般還是建議盡量以物件組合（包含關係）來取代繼承，因為類別繼承階層如果太深，基礎類別只要稍微改動，就會影響所有的子類別，進而波及所有的用戶端程式。

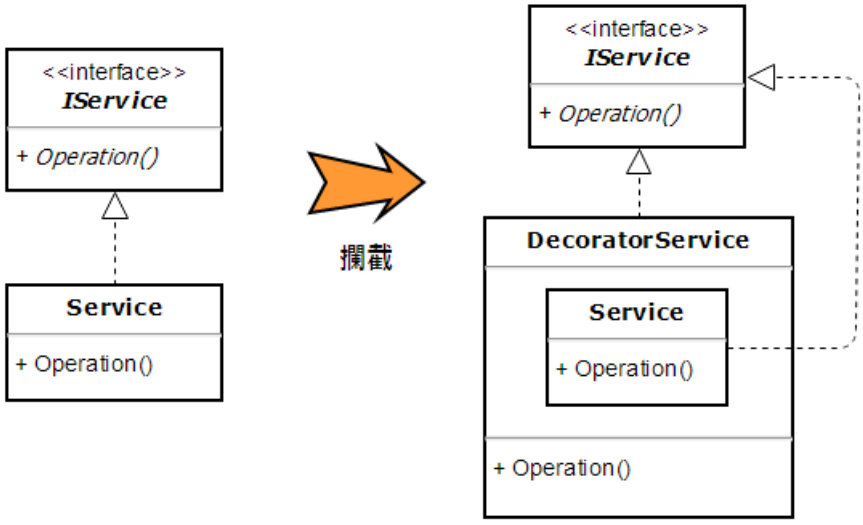
其實第 2 章介紹的 **Decorator 模式**就是實現攔截機制的一種常見作法。因此，這裡會延續第 2 章的範例來說明攔截的概念，然後再看看如何使用 DI 容器來實現攔截機制。

AOP 與攔截

AOP（Aspect Oriented Programming；切面導向程式設計）也是實現攔截的一種常見方法，它經常用來處理橫切面（cross-cutting）的功能，像是日誌（logging）、交易管理、安全控管等等。所謂的切面（aspect）指的是你所欲封裝的功能（例如 logging），而這些「切面」會攔截類別的方法，並從中改變方法的既有行為或增加功能——用 AOP 術語來說，這叫做提供 **Advice**。

使用 Decorator 模式實現攔截

使用 **Decorator 模式**來設計攔截機制時，首先當然要知道你想攔截（裝飾）哪個類別的哪些方法，而這些方法必須是定義於該類別所實作的抽象介面中。接著，建立一個新的裝飾類別，此類別也必須實作被攔截（被裝飾）類別的介面，而且要注入符合該介面的物件。這段描述可能有點抽象，請搭配底下這張圖和範例程式碼來理解。



圖中左邊的區塊是尚未加入攔截機制的類別結構，右邊則是加了攔截機制的結構。如果和第 2 章的 **Decorator 模式**結構圖比較，你會發現這裡的畫法有點不一樣。其實兩者沒有差別，這裡的呈現方式只是為了凸顯攔截者其實是個包覆類別（wrapper class）的概念。

再拿第 2 章的 `DecoratedLogger` 範例來說明，程式碼僅稍微修改，如下所示。

```
public interface ILogger
{
    Log(string msg);
}

public class ConsoleLogger : ILogger
{
    public void Log(string msg)
    {
        Console.WriteLine(msg);
    }
}

public class DecoratedLogger : ILogger
{
    private ILogger logger;
```

```
    public DecoratedLogger(ILogger aLogger)
    {
        logger = aLogger;
    }

    public void Log(string msg)
    {
        BeforeLog(); // 執行原本的 Log 方法之前所做的額外處理。

        logger.Log(msg);

        AfterLog(); // 執行原本的 Log 方法之後所做的額外處理。
    }

    private void BeforeLog()
    {
        // 略
    }

    private void AfterLog()
    {
        // 略
    }
}
```

由於 ConsoleLogger 和 DecoratedLogger 都實作了 ILogger 介面，所以高層模組在進行物件組合時，便可以注入 DecoratedLogger 物件，像這樣：

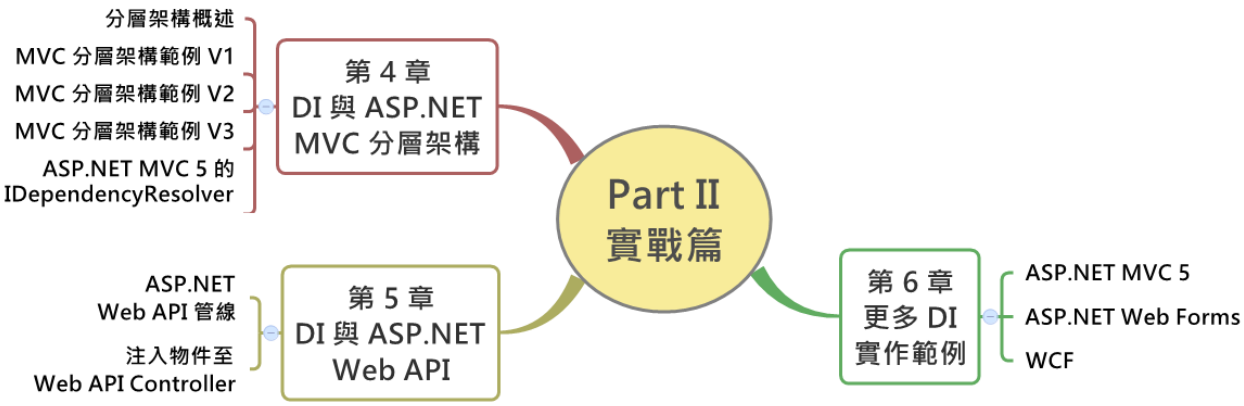
```
static void Main(string[] args)
{
    var logger = new DecoratedLogger(new ConsoleLogger());
    var orderService = new OrderService(logger);
    // ....略
}
```

本書的〈工具篇〉會介紹如何使用現成的 DI 框架來實現攔截機制。

本章回顧

- 在實現物件組合時，如果沒有使用 DI 容器，我們管它叫「窮人的 DI」（poor man's DI）。這種作法雖然需要開發人員多花一點點工夫，但除此之外，它並沒有違反任何 DI 設計原則，也沒有犧牲 DI 所提供的任何好處。
- DI 容器是提供 DI 功能的類別庫，目的在於減輕開發人員的負擔。DI 容器通常具備三大功能：物件組合、物件生命週期管理、攔截。
- 當用戶端（呼叫端）程式對 DI 容器提出要求，說它需要某個介面、基礎類別、或特定具象類別的時候，DI 容器就會去尋找符合其需求的型別，然後建立該型別的物件實體，並回傳給用戶端。此過程稱為「型別解析」（type resolving），或簡稱「解析」。
- DI 容器通常會提供多種物件生命週期選項，包括：臨時的（transient）、唯一的（singleton）、依每條執行緒、依每次 HTTP 請求等等，有些甚至提供自訂生命週期的管理機制。
- DI 容器的設定方式主要有三種：以程式碼來設定、使用 XML、以及自動註冊。
- DI 容器的「自動註冊」功能會自動掃描特定組件的型別資訊，並註冊符合條件的型別。使用此功能時，須留意是否影響應用程式的執行效能，以及是否增加程式碼理解上的困難。
- 攔截（interception）的目的是在不修改既有類別實作的前提下改變其行為。**Decorator 模式**和 **AOP**（Aspected-Oriented Programming；切面導向程式設計）都是實現攔截機制的常見方法。

Part II：實戰篇



〈第 4 章：DI 與 ASP.NET MVC 分層架構〉主要在介紹 ASP.NET MVC 分層架構的設計，並以逐步演進的範例程式來展示如何將原本緊密耦合的設計改成寬鬆耦合。除了分層架構的基礎觀念之外，同時也會說明如何將相依物件注入至 controller 類別的建構函式（兩種作法，分別使用 `IControllerFactory` 以及 `IDependencyResolver`）。本章內容適用於 ASP.NET MVC 5.x。

〈第 5 章：DI 與 ASP.NET Web API〉會詳細拆解 ASP.NET Web API 的管線架構，並說明此架構所提供的擴充點，以及如何利用這些擴充點來將自訂服務注入至 Web API 框架，以便取代或擴充預設的服務。本章內容適用於 ASP.NET Web API 2.x。

〈第 6 章：更多 DI 實作範例〉偏重實作練習，而較少著墨在基礎觀念和底層框架背後的運作機制會詳細拆解。而且，本章不再使用純手工 DI，而是示範如何使用現成的 DI 框架來減輕開發人員的負擔。

第 4 章：DI 與 ASP.NET MVC 分層架構

讀過前面幾章，相信你已經大致了解 DI 與 DI 容器對於打造寬鬆耦合的應用程式能夠帶來哪些幫助，以及有哪些實務上可能會碰到的麻煩與陷阱了。從這一章開始，重點會放在「純手工 DI」與特定開發技術平台或框架有關的實務應用，包括 ASP.NET MVC 應用程式的分層架構，以及 MVC 框架本身提供的 DI 擴充機制。

本章內容適用於 ASP.NET MVC 5。

內容大綱：

從內容大綱可以看得出來，分層架構的議題占了本章相當大的篇幅，甚至比主角 DI 的戲份還要多。如果您並不需要了解這個部分，而只想快點知道如何利用「建構式注入」的方式將相依物件注入至你的 controller 類別，您可以直接閱讀以下兩個小節：

- 〈MVC 分層架構範例 V2 －寬鬆耦合〉一節中的〈組合物件〉 小節
- 〈ASP.NET MVC 5 的 `IDependencyResolver`〉

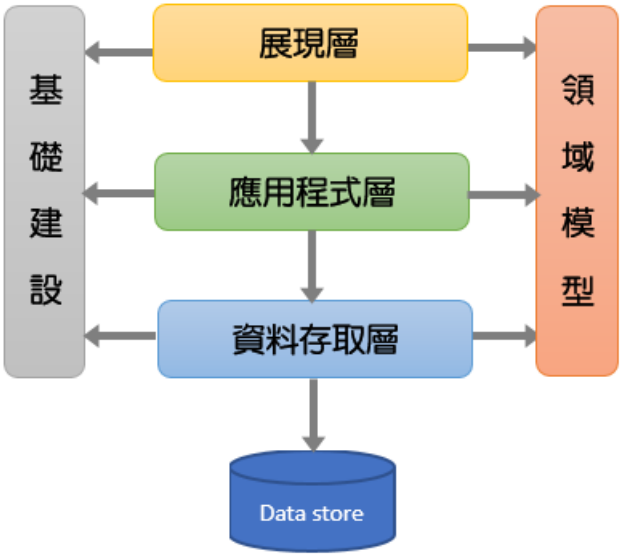
本章範例原始碼位置：

https://github.com/huanlin/di-book-support 裡面的 Examples/ch04 資料夾。

分層架構概述

分層架構（layered architecture）是一種將應用程式中各元件依其功能加以分組的邏輯架構，傳統上分為三層，即展現層（presentation layer）、商業邏輯層（business layer）、以及資料存取層（data access layer）。實務上，依應用程式的規模與複雜程度而定，亦可能切分得更多更細，而且哪些元件要歸類在哪一層，設計時也因人而異。也就是說，並沒有所謂唯一正確而能適用於任何場合的終極架構。下圖 所示即為本章範例所採用的分層架構。

N-layer 與 N-tier 這兩個名詞的中文通常都是「N 層」。為了避免混淆，我選擇用「分層」（layered）這個名詞。分層指的是應用程式邏輯層次的切分，N-tier 則與應用程式實際部署的組態有關，例如用戶端、應用程式伺服器、資料庫伺服器等等。



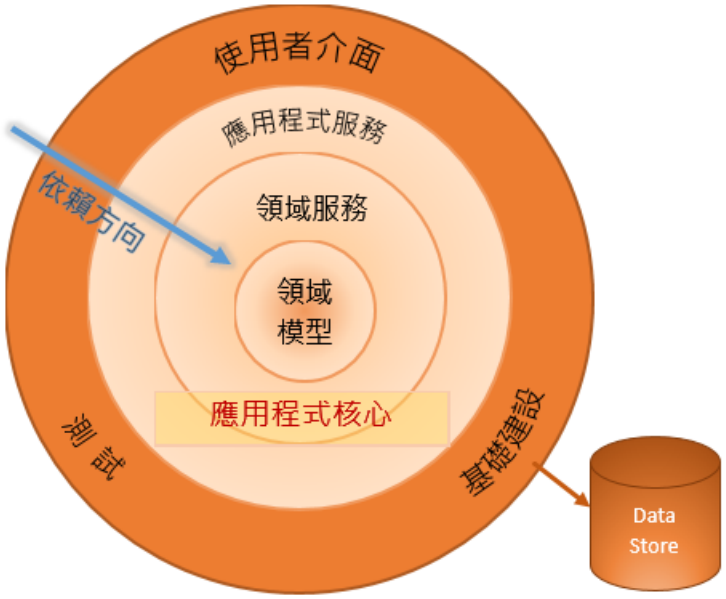
分層架構

圖中各層的功能如下所述：

- 展現層：提供使用者介面（UI），例如網頁、視窗應用程式、行動應用程式等等。
- 應用程式層：或稱為服務層（service layer），服務的對象是展現層。其中可能包含商業邏輯、資料驗證邏輯。
- 領域模型：包含領域物件（domain objects）、資料傳送物件（Data Transfer Object；**DTO**），其中亦可能包含資料驗證邏輯和部分商業邏輯。
- 資料存取層：即所謂的 **DAL**（Data Access Layer）。DAL 主要的功能是提供資料操作的相關服務。本章範例將使用 Entity Framework 來存取資料。
- 基礎建設：提供各層所需之基礎功能，如日誌（logging）、快取（caching）、異常處理（exception handling）機制、安全機制等等。

基本上，展現層係利用應用程式層所提供的服務來完成使用者的操作，應用程式層則透過資料存取層來操作資料。展現層不會直接使用最底層資料存取層的元件，應用程式層和資料存取層也不會反過來參考或使用展現層的元件。此外，領域物件會在各層之間傳遞，故展現層、應用層、資料存取層都會用到（相依於）領域模型。如果用同心圓的方式來

呈現此架構，就會變成如下圖 那樣，以領域模型為應用程式核心的**「洋蔥架構」（Onion Architrecture）**。



洋蔥架構

關於洋蔥架構

洋蔥架構（Onion Architecture）是由 Jeffrey Palermo 於 2008 年提出，其主要精神為：

- 應用程式係圍繞著一個獨立的物件模型來建構。
- 內層定義介面，外層實作介面。
- 耦合的方向是朝向中央。
- 應用程式的所有核心程式碼可以在與基礎建設分離的情況下正常運行。

還有一種也同樣強調以介面來將領域模型與外界隔離的架構，是 Alistair Cockburn 於 2005 年提出的**「六角架構」（Hexagonal Architecture）**，又名 **Ports and Adapters 模**

> **式**。如有興趣近一步了解，可參考這篇文章：〈應用程式的分層設計 (3) - DDD、六角、與洋蔥架構[a]〉。
>
> [a] http://huan-lin.blogspot.com/2012/10/designing-layered-application-3-onion.html

如本節一開始提過的，實務上的設計依需求不同、因人而異，即使採用相同的架構模型，實作時多少都有些差別。比如說，你可能會在某些場合直接把領域物件傳遞至展現層，而在某些場合可能會另外設計專供展現層使用的「檢視模型」（view model）類別。這些實作細節與本章主旨無關，故不在此詳述。

分層架構不是寬鬆耦合的保證。事實上，如果沒有遵循「針對介面寫程式」的原則，最終設計出來的程式往往還是緊密耦合且不易維護。在分層架構中使用 DI 技術，除了能夠達到寬鬆耦合，另一個目的就是提供可抽換元件的能力。需要抽換的元件可能是基礎建設中的工具類別（例如 logging 框架），也可能是其他抽象層。其中比較常被提及可能需要更換的，是資料存取層。比如說，你可能在某些技術論壇或會議中聽過有人這麼問：「萬一資料存取框架要從 Entity Framework 改成 NHibernate 怎麼辦？」或者「後端資料庫從 SQL Server 換成 Oracle 甚至 NoSQL 怎麼辦？」

為了解決這個問題，一個常見的解決方法是在資料存取層之上再疊加另一個抽象層：**Repository**，也就是接著要介紹的 **Repository 模式**。

抽象層本身也是有維護成本的。當你想要為應用程式增加抽象層的時候，不妨稍微想一下，多加這一層是否值得。我是說，它帶來的效益總和是否明顯大於多寫那些程式碼所帶來的維護成本？會不會，某些抽象層對於軟體系統的交付（delivery）並未產生相應的價值？

Repository 模式

關於 Repository 模式，Martin Fowler 這麼解釋[19]：

[19] http://martinfowler.com/eaaCatalog/repository.html

> 「Repository 是領域層與資料對應層之間的媒介，如同存在記憶體中的物件集合。用戶端會建構查詢規格（query specification），並傳遞給 Repository 以執行資料查詢。Repository 可以加入或移除物件，就如同操作一般的物件集合那樣.....概念上，Repository 將資料與其相關操作封裝成物件集合，方便以貼近物件導向的方式來存取資料。」

也就是說，Repository 是介於領域層與資料存取層之間的一個抽象層。當用戶端程式需要處理資料時，使用的是 Repository 提供的服務，而不需要知道底層的資料來源類型或資料存取技術（如 Entity Framework、NHibernate）。如果你覺得目前開發的軟體系統「很可能」需要切換底層資料來源的時候，便可以考慮採用 Repository 模式。

在設計 Repository 時，常見的做法是為每一個「主要的 entity」設計一個 Repository 類別。這些對應至主要 entity 的類別，以領域驅動設計[20]（Domain-Driven Design；簡稱 DDD）的術語來說，叫做 **Aggregate Root**（聚合根）。例如，用來處理客戶資料的 Repository 通常命名為 `CustomerRepository`，產品資料則是 `ProductRepository`，欲處理訂單資料則透過 `OrderRepository`；這些都是 aggregate roots。至於訂單細項，則通常由 `OrderRepository` 一併負責處理，就不另外設計 `OrderItemRepository`。此外，由於這些 Repository 類別大都有提供增、刪、改、查等方法，而且彼此長得很像，於是為了避免寫一堆重複的程式碼（DRY 原則[21]），便衍生出泛型 Repository 的設計。

在設計泛型 Repository 時，通常會把一些共同的基礎操作抽出來，定義在一個介面裡，以確保所有 Repository 實作類別都具有一組相同名稱的操作，程式碼寫起來也就比較一致。此介面一般命名為 `IRepository`，像這樣：

[20] http://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215
[21] http://en.wikipedia.org/wiki/Don't_repeat_yourself

```
public interface IRepository<TEntity>
{
    IEnumerable<TEntity> GetAll();
    IEnumerable<TEntity> Find(Expression<Func<TEntity, bool>> query);
    T GetByID(int id);
    void Add(T item);
    void Update(T item);
    void Delete(T item);
    void SaveChages();
}
```

然後，實作一個通用的泛型類別 `Repository<TEnity>` 來提供這些基本操作。至於其他特定的 Repository 類別，則可以繼承自 `Repository<TEntity>`，或者將它包在類別裡面（即採用組合而非繼承的作法）。

如何實作 Repository 模式並非本書主題，故簡單介紹到此。稍後的範例實作也會採取比較簡易的設計，以免出現太多無關主題的程式碼，影響閱讀。

Repository 參考資料

若您對 Repository 模式有興趣，網路上有許多教學和範例可以參考，例如：

- ASP.NET MVC 專案分層架構 Part.2 抽出 Repository 裡相同的部份[a] by Kevin Tseng (mrkt)
- Learning MVC Part 6: Generic Repository Pattern in MVC3 Application with Entity Framework[b] by By Akhil_Mittal
- Generic implementation of Repository pattern in C# .NET[c] by besnik

[a] http://kevintsengtw.blogspot.tw/2012/10/aspnet-mvc-part2-repository.html
[b] http://www.codeproject.com/Articles/640294/Learning-MVC-Part-Generic-Repository-Pattern-in
[c] https://github.com/besnik/generic-repository

接著要開始進入分層架構範例的實作部分。此範例的第一個版本會採取緊密耦合的設計，

之後會修改第二版，改成比較寬鬆耦合的設計。然後，還會有稍微簡化的第三版（拿掉 Repository）。

建議您一邊閱讀，一邊把程式碼敲進去，透過實際的練習來體會不同做法的差異。這樣做有個好處：當你需要從零開始建構一個分層式架構時，你會比較能夠判斷實作類別之間的耦合關係究竟要拆分到多細才適當，以及哪些地方保留的彈性已經足夠，將來即使出現變動，也只需要花一點重構的工夫就能搞定（希望如此）。

MVC 分層架構範例 V1－緊密耦合

現在要建立一個 ASP.NET MVC 應用程式，以作為分層架構的基本範例。首先，新建一個空白方案（blank solution），命名為 `LayeredMvcDemo`。接著照以下各小節的順序完成各層的實作。

本節範例的原始碼位置：

Examples/ch04/LayeredMvcDemo_V1_Coupled 資料夾下，方案名稱是 LayeredMvcDemo.sln。

領域模型

實作步驟：

1. 在方案中加入一個新的 Class Library 專案，命名為 `LayeredMvcDemo.Domain`。
2. 在此專案中建立一個資料夾：Models。
3. 在 Models 資料夾中加入新類別：Customer.cs。此類別的全名會是 `LayeredMvcDemo.Domain.Models.Customer`。程式碼如下：

```
public class Customer
{
    public int Id { get; set; }
    public string CompanyName { get; set; }
    public string Contact { get; set; }
}
```

資料存取層

實作步驟：

1. 在方案中加入一個新的 Class Library 專案，命名為 LayeredMvcDemo.DataAccess。
2. 利用 NuGet 管理員加入組件參考：Entity Framework。
3. 加入專案參考：LayeredMvcDemo.Domain。
4. 加入新類別 SouthwindContext.cs。此檔案包含兩個類別：SouthwindContext 和 SouthwindDBInitializer。程式碼如下：

```
public class SouthwindContext : DbContext
{
    public SouthwindContext() : base("SouthwindDB")
    {
        Database.SetInitializer<SouthwindContext>(new SouthwindDBInitializer());
    }

    public DbSet<Customer> Customers { get; set; }
}

public class SouthwindDBInitializer
        : CreateDatabaseIfNotExists<SouthwindContext>
{
    public override void InitializeDatabase(SouthwindContext context)
    {
        base.InitializeDatabase(context);

        context.Customers.Add(new Customer {
            Id = 1, CompanyName = "MikeSoft", Contact = "Michael"
        });
        context.Customers.Add(new Customer {
```

```
            Id = 2, CompanyName = "OralCall", Contact = "Vivid"
        });
        context.Customers.Add(new Customer {
            Id = 3, CompanyName = "SimonTech", Contact = "Simon"
        });
        context.Customers.Add(new Customer {
            Id = 4, CompanyName = "IoSee", Contact = "Mark"
        });

        context.SaveChanges();
    }
}
```

這裡是使用 Entity Framework 的 **Code First** 模型。其中的 `SouthwindDBInitializer` 提供自訂的資料庫初始化程序，用途是當資料庫不存在時自動建立資料庫，同時加入幾筆 `Customer` 資料，方便稍後觀察程式執行結果。

5. 加入新類別：CustomerRepository.cs。

此類別的用途是提供 Customer 資料的存取操作。簡單起見，僅提供兩個方法：

- `GetCustomerById` 一取得指定 ID 的客戶資料。
- `GetCustomerList` 一取得指定篩選條件的客戶清單。

程式碼如下：

```
public class CustomerRepository
{
    private SouthwindContext db = new SouthwindContext();

    public Customer GetCustomerById(int id)
    {
        var query = from t in db.Customers
                    where t.Id == id
                    select t;
        return query.FirstOrDefault();
    }

    public IEnumerable<Customer> GetCustomerList(Func<Customer, bool> filter)
    {
        var query = from t in db.Customers
                    select t;
```

```
            return query.Where(filter);
        }
    }
```

從 DDD（領域驅動設計）的角度來看，Repository 比較貼近領域層，而非資料存取層。這裡將 Repository 類別放在資料存取層，使它帶有 DAO（Data Access Object）的味道。只是讓你知道一下，在寫這些範例程式時，我沒有特別去考慮它們是否為「正宗的 DDD」。如欲進一步了解 DDD，可參考 Eric Evan 的《Domain-Driven Design》。

應用程式層

實作步驟：

1. 在此方案中加入一個新的 Class Library 專案，命名為 `LayeredMvcDemo.Application`。
2. 加入專案參考：`LayeredMvcDemo.Domain` 以及 `LayeredMvcDemo.DataAccess`。
3. 在此專案中建立一個資料夾：Services。
4. 在 Services 資料夾中加入一個新類別：CustomerService.cs。此類別的全名會是 `LayeredMvcDemo.Application.Services.CustomerService`。

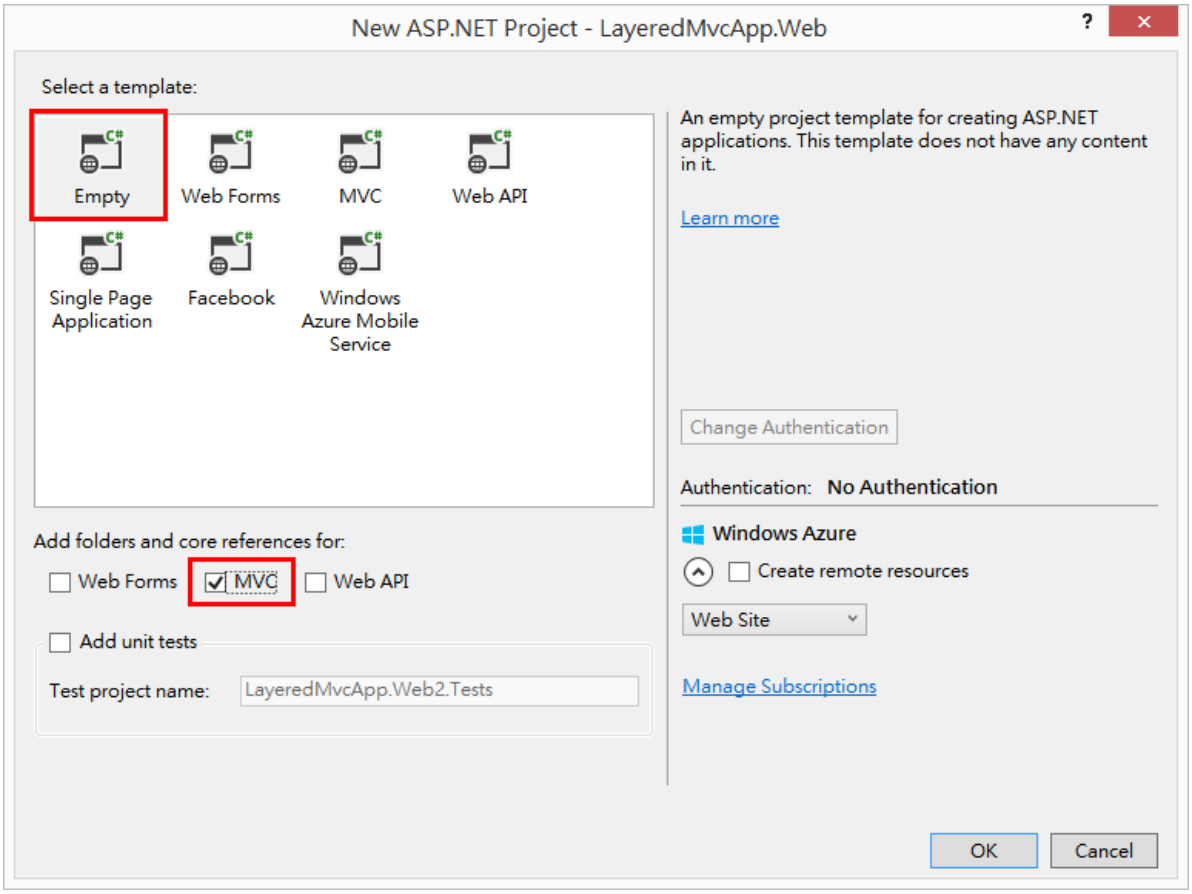
   程式碼：

```
public class CustomerService
{
    private CustomerRepository repository = new CustomerRepository();

    public Customer GetCustomerById(int id)
    {
        return repository.GetCustomerById(id);
    }

    public List<Customer> GetCustomerList(Func<Customer, bool> filter)
    {
        return repository.GetCustomerList(filter).ToList();
    }
}
```

展現層

實作步驟：

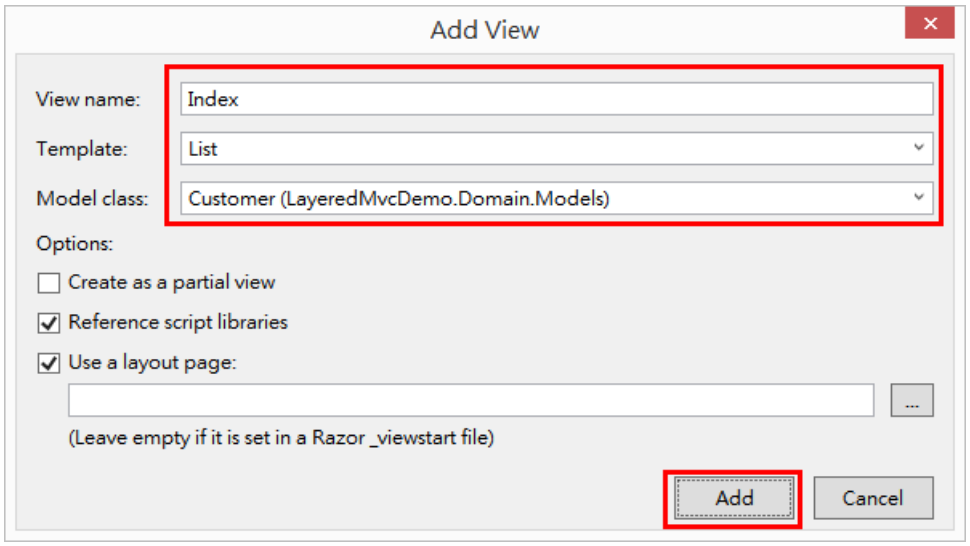
1. 在此方案中加入一個新專案，專案範本選擇 ASP.NET Web Application，專案名稱為 `LayeredMvcDemo.Web`。接著開啟對話窗進一步挑選範本時，選擇【Empty】，並勾選【MVC】，然後按【OK】。參考下圖操作。



2. 利用 NuGet 管理員加入組件參考：Entity Framework。
3. 加入專案參考：`LayeredMvcDemo.Domain` 以及 `LayeredMvcDemo.Application`。
4. 在 Controllers 資料夾中加入一個新類別：CustomerController.cs。程式碼如下：

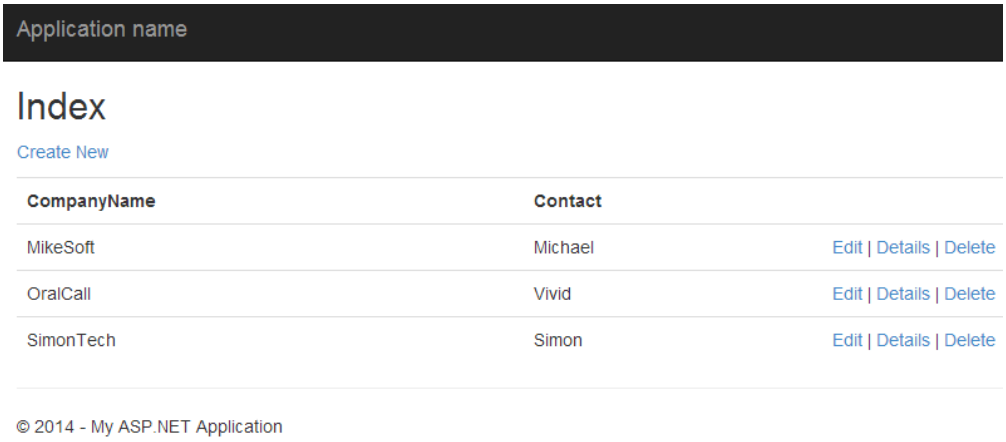
```
public class CustomerController : Controller
{
    private CustomerService customerService = new CustomerService();

    // GET: Customer
    public ActionResult Index()
    {
        var customers = customerService.GetCustomerList(cust => cust.Id < 4);
        return View(customers);
    }
}
```

接著加入一個 View。作法是：在 Index 方法上點右鍵，選【Add View】，接著會開啟一個對話窗，讓你設定此 View 的名稱、範本、和模型類別。參考下圖操作。



在此對話窗中點【Add】按鈕之後，便會在此專案的 Views\Customer\ 資料夾下產生 Index.cshtml。此檔案的內容與主題無關，故不列出其原始碼。

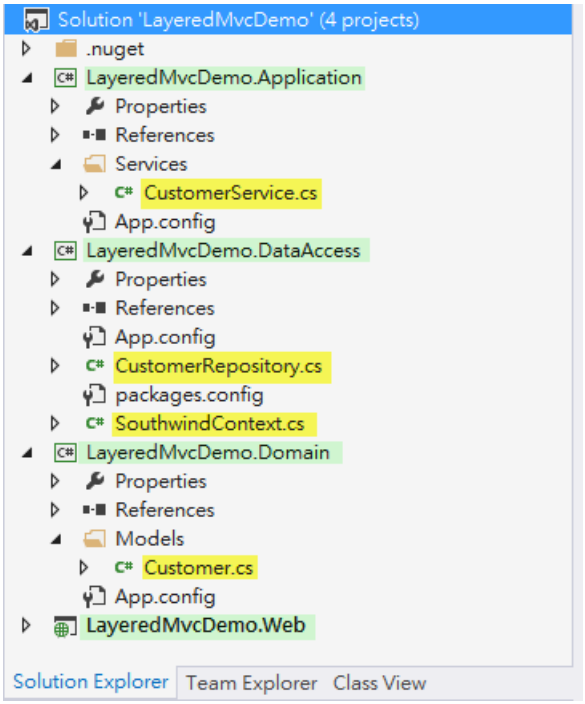
5. 建置整個方案，然後在瀏覽器中檢視 Index.cshtml，以查看執行結果，如下圖所示。此動作的目的只在確定前面的實作步驟都正確完成，且程式碼能夠通過編譯。



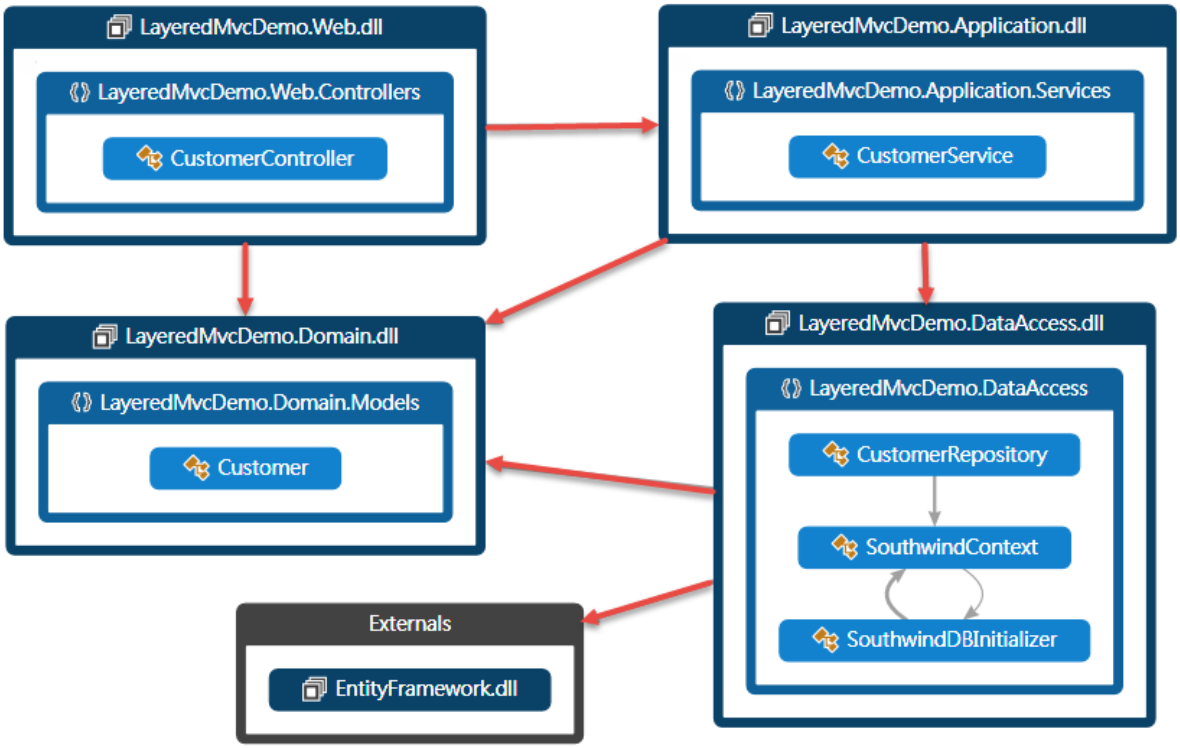
（執行結果）

審視目前設計

以下兩張圖分別呈現了整個範例程式的專案結構，以及此範例程式的組件相依關係。從圖中可以看得出來各專案的名稱、目錄結構、以及包含哪些類別。其中的 LayeredMvcDemoWeb 節點並未展開，因為展現層並不是我們關心的重點。


Solution 'LayeredMvcDemo' (4 projects)
.nuget
LayeredMvcDemo.Application
Properties
References
Services
CustomerService.cs
App.config
LayeredMvcDemo.DataAccess
Properties
References
App.config
CustomerRepository.cs
packages.config
SouthwindContext.cs
LayeredMvcDemo.Domain
Properties
References
Models
Customer.cs
App.config
LayeredMvcDemo.Web
Solution Explorer
Team Explorer
Class View

專案結構



組件相依關係圖

透過組件相依關係圖，我們知道：

- 應用程式中所有的組件都有參考領域模型（LayeredMvcDemo.Domain.dll）。
- 資料存取層有參考 EntityFramework.dll，領域模型與應用程式層則沒有。
- 展現層是整個生態系統的最上層，因此沒有人參考它。
- `CustomerController` 直接使用（相依於）`CustomerService`。
- `CustomerService` 直接使用（相依於）`CustomerRepository`。
- 完全沒有介面。

是的，目前的設計並沒有定義任何抽象介面，全都是具象類別（concrete classes）。那麼，如果稍稍修改一下 `CustomerRepository` 類別，把它的公開方法抽離出來，定義成一個 `ICustomerRepository` 介面，並且讓 `CustomerRepository` 實作此介面呢？如此一來，我們的 `CustomerService` 不就可以「針對介面寫程式」了嗎？像下面這段程式碼：

```
public class CustomerService
{
    private readonly ICustomerRepository repository = new CustomerRepository();

    // 其餘不變，故省略。
}
```

就耦合程度而言，這樣做並沒有什麼差別，因為 CustomerService 還是直接使用 new 來建立 CustomerRepository 的執行個體，亦即直接依賴特定實作。緊密的耦合關係並不會因為我們單單在宣告變數時改用抽象型別而有所鬆動。不過話說回來，拆分介面這個動作倒是個好的開始，也是接下來修改第二版時的重點之一。

MVC 分層架構範例 V2－寬鬆耦合

假設需要考慮將來可能有如下需求變動：

- 原本使用的 ORM（object-relation mapping；物件關聯對應）框架是 Entity Framework，將來可能會改用 NHibernate。理想狀況下，此時只需修改資料存取層的 Repository 類別。
- 基於測試需要，得將應用程式層的 CustomerService 類別暫時替換成其他測試替身（test double）。

為了應付這些可能的變動，我們想要在設計時就預先保留一點彈性，讓以後需要切換特定元件時輕鬆一點、少改一些程式碼。

當然，這些只是為了示範 DI 所假設的情境。切換 ORM 這個動作，實際上可能會麻煩些，因為不同 ORM 有各自特殊功能和行為，可能影響其他高層模組程式的寫法，所以也有可能不是只修改 Repository 抽象層這麼簡單。同理，若資料庫要從 Oracle 換成 SQL Server，也不會是單單換掉 Repository 層就能解決（除非應用程式完全不使用資料庫特有的功能和 SQL 語法）。總之，能夠任意「切換實作」的確很酷，但實際上會不會用到，成本與效益之間還是得斟酌一下。

真實案例：為了方便測試而切換資料來源

我曾參與過一個專案，負責設計其中一塊中介層服務。該服務需提供資料給網頁、行動裝置、桌上型應用程式等用戶端，而那些資料是去遠端主機上的 IBM WebSphere MQ 佇列中取得。在佇列的另一頭，提供資料的又是另一個遠端大型主機，而那個大型主機又是去跟其他遠端資料中心查詢資料，然後將執行結果塞入佇列。由於該主機位於國外，且因法令限制而無法讓我遠端連線測試，故在開發程式時，我是用對方提供的一個模擬程式來當作資料來源，該模擬程式會從佇列讀取查詢請求，然後產生模擬的查詢結果，再將查詢結果塞到佇列，供我的程式讀取。經過一段時間，我發現每次要建置與設定此測試環境都要花好大一番工夫，於是我另外設計了一個類別來當作測試資料的提供者，而其資料來源是一個專供測試用的 SQL Server 資料庫。如此一來，每當需要測試或除錯時，便可以將原本的 WebSphere MQ 佇列資料提供者切換成 SQL Server 資料提供者，省下了許多設定測試環境的時間。

話題扯遠了。接著就來看看如何利用介面與 DI 來進一步鬆綁此範例應用程式各層的耦合關係。

此範例的原始碼位置：

Examples/ch04/LayeredMvcDemo_V2_Decoupled 資料夾下，方案名稱是 LayeredMvcDemo.sln。

領域模型

就此範例而言，我覺得並沒有必要額外定義一個 `ICustomer` 介面，然後讓 `Customer` 類別實作此介面。故 `Customer` 類別無須變動。

資料存取層

由於先前的一個假設是資料存取技術可能會從 Entity Framework 改成其他 ORM，所以需要將 `CustomerRepository` 類別的功能定義成一個新的介面，好讓高層模組只依賴此抽象介面，而不依賴特定實作。這個新介面的名稱是 `ICustomerRepository`，而原本的 `CustomerRepository` 類別必須實作此介面，程式碼如下。

```
public interface ICustomerRepository
{
    Customer GetCustomerById(int id);
    IEnumerable<Customer> GetCustomerList(Func<Customer, bool> filter);
}

public class CustomerRepository : ICustomerRepository
{
    // 其餘不變。
}
```

應用程式層

為了讓 `CustomerService` 不依賴特定的 Repository 實作類別，我們得將原本有用到 `CustomerRepository` 的地方改成使用 `ICustomerRepository` 介面。同時，這裡使用「**建構式注入**」（**Constructor Injection**；參見第 2 章）的技巧來讓外界提供 Repository 物件。

```
public class CustomerService
{
    private readonly ICustomerRepository repository; // 改用介面

    public CustomerService(ICustomerRepository repo) // 建構式注入
    {
        this.repository = repo;
    }

    public Customer GetCustomerById(int id)
    {
        return repository.GetCustomerById(id);
    }

    public List<Customer> GetCustomerList(Func<Customer, bool> filter)
    {
        return repository.GetCustomerList(filter).ToList();
    }
}
```

同樣地，為了讓展現層不直接依賴 `CustomerService` 類別，我們可以將服務類別的功能萃取出來，定義成 `ICustomerService` 介面，然後讓 `CustomerService` 實作此介面：

```
public interface ICustomerService
{
    Customer GetCustomerById(int id);
    List<Customer> GetCustomerList(Func<Customer, bool> filter);
}

public class CustomerService : ICustomerService
{
    // 其餘不變。
}
```

展現層

Controller 的部分也一樣只依賴介面，並使用「建構式注入」。修改後的程式碼如下：

```
public class CustomerController : Controller
{
    private readonly ICustomerService customerService;  // 改用介面

    public CustomerController(ICustomerService service)  // 建構式注入
    {
        this.customerService = service;
    }

    public ActionResult Index()
    {
        var customers = customerService.GetCustomerList(cust => cust.Id < 4);
        return View(customers);
    }
}
```

現在各層之間的關係都是透過介面來連接了。然而，介面是抽象的，沒有實作。所以剩下的問題便是：程式執行時所需要的物件由誰提供、在何處建立？

組合物件

正如〈基礎篇〉中提過的，建立並組合這些相依物件的程式碼應該要寫在應用程式的「組合根」（Composition Root）。以 ASP.NET 應用程式來說，組合根常常會是 Global.asax 檔案中的 `Application_Start` 方法。

但有個問題：ASP.NET MVC 的 controller 物件並非由我們自行建立，而是由 MVC 框架本身自動建立的。比如說，當使用者透過瀏覽器檢視網址 `http://.../Customer/Index` 的時候，ASP.NET MVC 框架會去尋找 `CustomerController` 類別，並呼叫其預設建構函式(不帶任何參數的建構函式）來建立 controller 物件。你可以看到，目前的 `CustomerController` 已經沒有預設建構函式，因此會造成 ASP.NET MVC 框架在執行時期找不到可用的預設建構函式而發生如下錯誤：

System.MissingMethodException: No parameterless constructor defined for

this object.
(這個物件沒有定義無參數的建構函式。)

還好，ASP.NET MVC 有提供擴充機制，讓我們可以自行建立 controller 物件。欲使用此擴充機制，關鍵在於了解如何切換 controller 工廠。(**Factory 模式**的相關概念請參閱第 2 章)

切換 Controller 工廠

預設情況下，ASP.NET MVC 框架在建立 controller 物件時，係透過預先註冊的 `IControllerFactory` 物件來負責這件工作，而這個物件的實際型別是 `DefaultControllerFactory`。換言之，`DefaultControllerFactory` 類別實作了 `IControllerFactory` 介面，並且已經預先註冊到 MVC 框架裡，成為預設的 controller 物件工廠。這個預設的物件工廠是可替換的，即我們也可以寫一個類別來實作 `IControllerFactory` 介面，並取而代之。

`IControllerFactory` 定義了三個方法，如下所示：

```
// 命名空間：System.Web.Mvc
public interface IControllerFactory
{
    IController CreateController(RequestContext requestContext,
                                 string controllerName);
    void ReleaseController(IController controller);
    SessionStateBehavior GetControllerSessionBehavior(
        RequestContext requestContext, string controllerName);
}
```

其中的 `CreateController` 方法的用途就是建立一個 controller 物件來處理當前的 HTTP 請求。實作此方法時，你要怎麼建立 controller 物件都可以，只要返回的物件型別是 `IController` 就行。

簡單起見，這裡的自訂類別就直接繼承自 MVC 內建的 `DefaultControllerFactory` 類別，然後視需要改寫父類別的方法。程式碼如下：

```
public class MyControllerFactory : DefaultControllerFactory
{
    public override IController CreateController(
        RequestContext requestContext, string controllerName)
    {
        if (controllerName == "Customer")
        {
            // 建立相依物件並注入至新建立的 controller。
            var repository = new CustomerRepository();
            var service = new CustomerService(repository);
            var controller = new CustomerController(service);
            return controller;
        }
        // 其他不需要特殊處理的 controller 型別就使用 MVC 內建的工廠來建立。
        return base.CreateController(requestContext, controllerName);
    }

    public override void ReleaseController(IController controller)
    {
        // 如果需要釋放其他物件資源，可寫在這裡。
        base.ReleaseController(controller);
    }
}
```

改寫的 CreateController 方法是根據傳入參數 controllerName 來判斷該建立哪一種 controller 物件。你可以看到，建立 CustomerService 物件時需要先注入 CustomerRepository 物件，之後再把 CustomerServer 物件注入至 CustomerController 的建構函式。

最後一步，就是在應用程式啟動時告訴 ASP.NET MVC 框架：請使用我提供的 controller 物件工廠，而不要用你預設的。作法是在 Global.asax 檔案的 Application_Start 方法中呼叫 ControllerBuilder 物件的 SetControllerFactory 方法。參考以下程式片段：

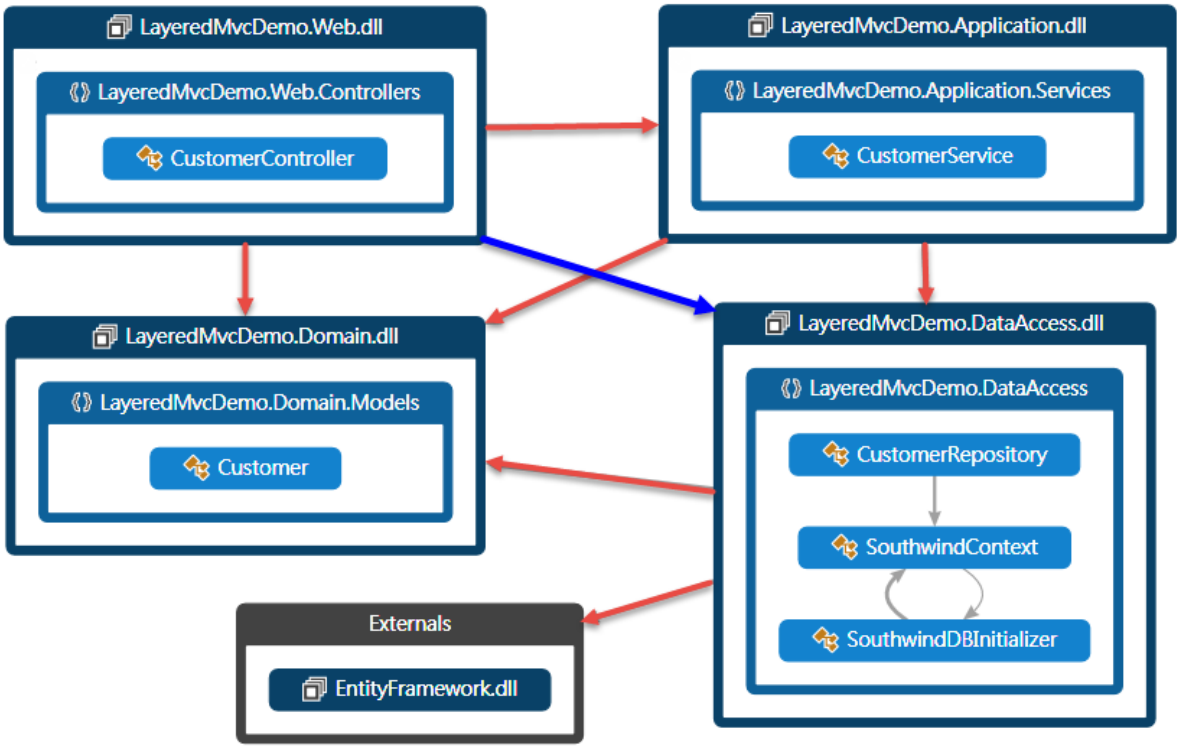
```
public class MvcApplication : System.Web.HttpApplication
{
   protected void Application_Start()
   {
        AreaRegistration.RegisterAllAreas();
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        // 把 MVC 框架的預設 controller factory 換掉。
        var ctrlFactory = new MyControllerFactory();
        ControllerBuilder.Current.SetControllerFactory(ctrlFactory);
   }
}
```

這個 SetControllerFactory 方法就是先前介紹 DI 注入模式中的「屬性注入」（**Property Injection**）。雖然它是個函式，但其作用其實是個設定函式（setter），故等同於透過設定屬性來注入相依物件。

此範例程式的第二版到此完成。網頁的執行結果和先前的版本 一樣，就不重複貼圖了。

審視目前設計

下圖為此範例程式改版之後的組件相依關係圖。請注意展現層(LayeredMvcDemo.Web.dll)現在多了一個箭頭指向資料存取層（LayeredMvcDemo.DataAccess.dll）。這是因為，此應用程式的「組合根」正好是在展現層，而「組合根」需要參考所有實作類別，才有辦法建立各相依型別的執行個體。



DI 版本的組件相依關係圖

避免過度設計

現在假設要加入訂單管理功能。照先前的設計方式，我們可能會在各層加入以下類別：

- 展現層：OrderController
- 領域模型：Order、OrderItem、Product
- 應用程式層：OrderService、ProductService
- 資料存取層：OrderRepository、ProductRepository

由於訂單服務也需要存取客戶資料和產品資料，所以 `OrderService` 類別可能需要注入三個（甚至更多）Repository 物件，像這樣：

```
public class OrderService
{
    private readonly IOrderRepository orderRepository;
    private readonly ICustomerRepository customerRepository;
    private readonly IProductRepository productRepository;

    public OrderService(
        IOrderRepository orderRepo,
        ICustomerRepository customerRepo,
        IProductRepository productRepo)
    {
        this.orderRepository = orderRepo;
        this.customerRepository = customerRepo;
        this.productRepository = productRepo;
    }
}
```

於是負責組合物件的 `MyControllerFactory` 也需要注入相應的物件：

```
public class MyControllerFactory : DefaultControllerFactory
{
    public override IController CreateController(
        RequestContext requestContext, string controllerName)
    {
        if (controllerName == "Customer")
        {
            var repository = new CustomerRepository();
            var service = new CustomerService(repository);
            var controller = new CustomerController(service);
            return controller;
        }
        else if (controllerName == "Order") // 此區塊是新增加的程式碼
        {
            var orderRepo = new OrderRepository();
            var customerRepo = new CustomerRepository();
            var productRepo = new ProductRepository();

            var service = new OrderService(orderRepo, customerRepo, productRepo);
            var controller = new OrderController(orderService);
            return controller;
        }
```

```
        else
        {
            return base.CreateController(requestContext, controllerName);
        }
    }
}
```

真實世界的情形可能會比這裡的範例要來得更複雜些。當你碰到這種狀況，可以先檢查一下，是不是出現了第 2 章提過的〈過度注入〉的陷阱，並考慮幾種可能的解法，例如：

1. 有些相依物件可能可以改用「屬性注入」和「方法注入」。
2. 使用 **Factory 模式**、**Ambient Context**、或 **Service Locator 模式**。
3. 將建構函式的多個參數重構成 **Parameter Object**（參數物件）。
4. 多載建構函式（overloaded constructors）。下一節修改此範例的第三個版本時會示範此作法。
5. 使用 **Façade 模式**來重構現有設計。

此外，如果某些相依類別負責提供的是橫切面（cross-cutting）的功能（如 logging），還可以考慮第 3 章介紹的「攔截」（interception）技術，運用 **Decorator 模式**或 **AOP**（Aspect-Oriented Programming）來實作這些橫切面功能。

問自己下列問題，來決定該採用何種解法：

- 類別是不是被賦予太多責任了？

  如果是這種情形，可嘗試重構程式碼，將類別的責任再適度切割一下。
- 傳入建構函式的相依物件，其中有一些是要傳遞給下一層或更深層的物件用的，自己根本沒有使用？

  這種情況，可以看看有些相依物件是不是到處都有用到。如果是的話，考慮使用 **Ambient Context 模式**（參閱第 2 章），甚至使用 **Service Locator 模式**。
- 是否切出太多不必要的抽象層？有些元件不太可能抽換，卻仍然用介面去拆解耦合？

  對於根本不可能抽換、或相對穩定的元件，不排除直接使用 new 來建立其物件實體。

最後一點似乎有背道而馳、走回頭路的感覺。關於這點，我想再多解釋一下（雖然我不見得是對的）。

寬鬆耦合也是有成本的。如果把 DI 當成一把鋒利的手術刀，看到任何東西就不由分說把它切開來，然後用介面和注入物件的方式在執行時期將它們黏起來，那麼，切得越多越細，黏合的工夫自然也越多。我們絕對不希望為了解決一個問題，卻因為解法本身的副作用而引進更多麻煩，反而讓程式更難維護。先研究不傷身體，再講求效果，大概是這個意思。

寬鬆耦合、單一責任（SRP）、開放關閉（OCP）、里氏替換（LSP）、介面隔離（ISP）、相依反轉（DIP）....這些設計原則都值得我們學習且牢記在心，但也別忘了，我們還有 **YGANI**（You Ain't Gonna Need It）這個好朋友。

YAGNI

YAGNI（You Ain't Gonna Need It）原則是 KISS（Keep It Simple and Stupid）原則的兄弟，用意都在提醒我們：設計應保持簡單，不應該為程式碼加入一堆用不到的功能。

身為開發人員，有時候，我們可能會想得很多、很遠。也許是因為需求太模糊、經常變動而產生的危機感，也許是對於追求完美的堅持、技術的狂熱、或其他原因，我們可能會難以抗拒某種具有高度儀式、規範、和一堆術語的設計方法或框架，以至於未加思索地套用在自己的設計中。對老手來說，這樣的不加思索可能是經驗累積出來的直覺，但是對於仍在摸索學習階段的人來說，卻可能見樹不見林，照本宣科，因而出現為 pattern 而 pattern、為 DI 而 DI 的情形。

對此情形，YAGNI 扮演著一種平衡的角色，或者說，鎮靜劑。將此原則一併納入你的設計工具箱，也許可在深陷迷霧、糾結徬徨時讓自己稍稍放鬆一下，想一想，現在這個程式，真的需要使用某某模式嗎？一定要用 DI 嗎？花這些時間所寫出來的程式碼，有產生相對的實質效益嗎？無論答案為何，我相信這樣的質疑與思考過程是有幫助的。

MVC 分層架構範例 V3－簡化一些

在這個小節裡，我會嘗試將範例程式 V2 的版本再稍微簡化一下，成為分層架構範例的第三個版本。此版本與 V2 版本主要的差異是拿掉 Repository 抽象層，理由是：

- 假定目前的應用程式會使用 Entity Framework 來存取關聯式資料庫，而且以後需要切換成其他 ORM 框架的可能性極低。
- Entity Framework 本身即具備了 Repository 的特性，而且包含了變更追蹤、交易管理等功能。既然前提是不會更換 ORM 框架，我們想要省略 Repository 抽象層，讓應用程式架構輕巧一些。

如果你的評估是未來可能需要切換資料存取層，那麼採用前一版本（有 Repository 層）的設計可能比較適合。相對的，若實際情形比較符合上述條件，則建議採用本節的版本。

此範例的原始碼位置：

Examples/ch04/LayeredMvcDemo_V3.0_NoRepo 資料夾下，方案名稱是 LayeredMvcDemo.sln。

資料存取層

去除 Repository 之後，原本的資料存取層裡面就只剩下 `SouthwindContext` 類別（繼承自 `DbContext`）和 `SouthwindDBInitializer` 類別了。這兩個類別的程式碼不需要任何修改。

應用程式層

原本應用程式層中的服務類別是使用 Repository 來存取資料，現在改為直接使用 `SouthwindContext`，也就是使用 Entity Framework 來操作資料。因此，`LayeredMvcDemo.Application` 專案現在必須加入 Entity Framework 組件參考。

ICustomerService 介面不變，但 CustomerService 類別需要修改。程式碼如下：

```
public class CustomerService : ICustomerService
{
    private readonly SouthwindContext db;

    public CustomerService()
    {
        // 提供預設的 DbContext 物件。
        db = new SouthwindContext();
    }

    public CustomerService(SouthwindContext dbContext)
    {
        // 呼叫端有注入 DbContext 物件，就用對方提供的。
        this.db = dbContext;
    }

    public Customer GetCustomerById(int id)
    {
        return db.Customers.Find(id);
    }

    public List<Customer> GetCustomerList(Func<Customer, bool> filter)
    {
        return db.Customers.Where(filter).ToList();
    }
}
```

值得注意的地方：

- 拿掉 Repository 之後，也拿掉了一些重複的程式碼（轉呼叫 Repository 方法的部分）。
- 現在提供兩個版本的建構函式，一個不帶參數，一個則接受外界傳入 SouthwindContext 物件。如此一來，即使外界沒有提供，也能自給自足，照常運作。

提供多載版本的建構函式可以確保當呼叫端沒有注入相依物件時，本身仍會自行建立所需之物件。第 2 章曾提過，Mark Seemann 視之為反模式（anti-pattern）。

提供一個建構函式讓外界傳入 DbContext 的另一個用意是，應用程式層中經常會有多個服務類別彼此合作共同完成任務的情況，例如訂單服務 OrderService 可能需要呼叫 ProductService、CustomerService、和 ShippingService 等物件的方法來完成特定商業流程，而這些操作都是在同一個 HTTP 請求的範圍（context）內，所以這些物件最好也都共享同一個 DbContext，以方便管理交易，並且避免出現資料不一致的情形。以上述範例的做法來說，OrderService 可將本身的 DbContext 物件傳遞給其他共同合作的物件，以便共享同一個 DbContext 物件。參考以下程式片段：

```
public class OrderrService : IOrderService
{
    private readonly SouthwindContext db;

    public OrderService()
    {
        // 提供預設的 DbContext 物件。
        db = new SouthwindContext();
    }

    public OrderService(SouthwindContext dbContext)
    {
        // 呼叫端有注入 DbContext 物件，就用對方提供的。
        this.db = dbContext;
    }

    public void CreateOrder(Order order)
    {
        // 建立相關服務時一併傳入 DbContext 物件。
        var customerSvc = new CustomerService(this.db);
        var shippingSvc = new ShippingService(this.db);

        customerSvc.DoSomething(order);
        shippingSvc.DoSomething(order);

        db.Orders.Add(order);
        db.SaveChanges();
    }
}
```

實作幾個服務類別之後，會發現它們都有一些相同的程式碼（例如建構函式）。此時亦可

考慮將這些重複的程式碼抽出來，放到一個基礎類別裡面，讓所有的服務類別繼承。

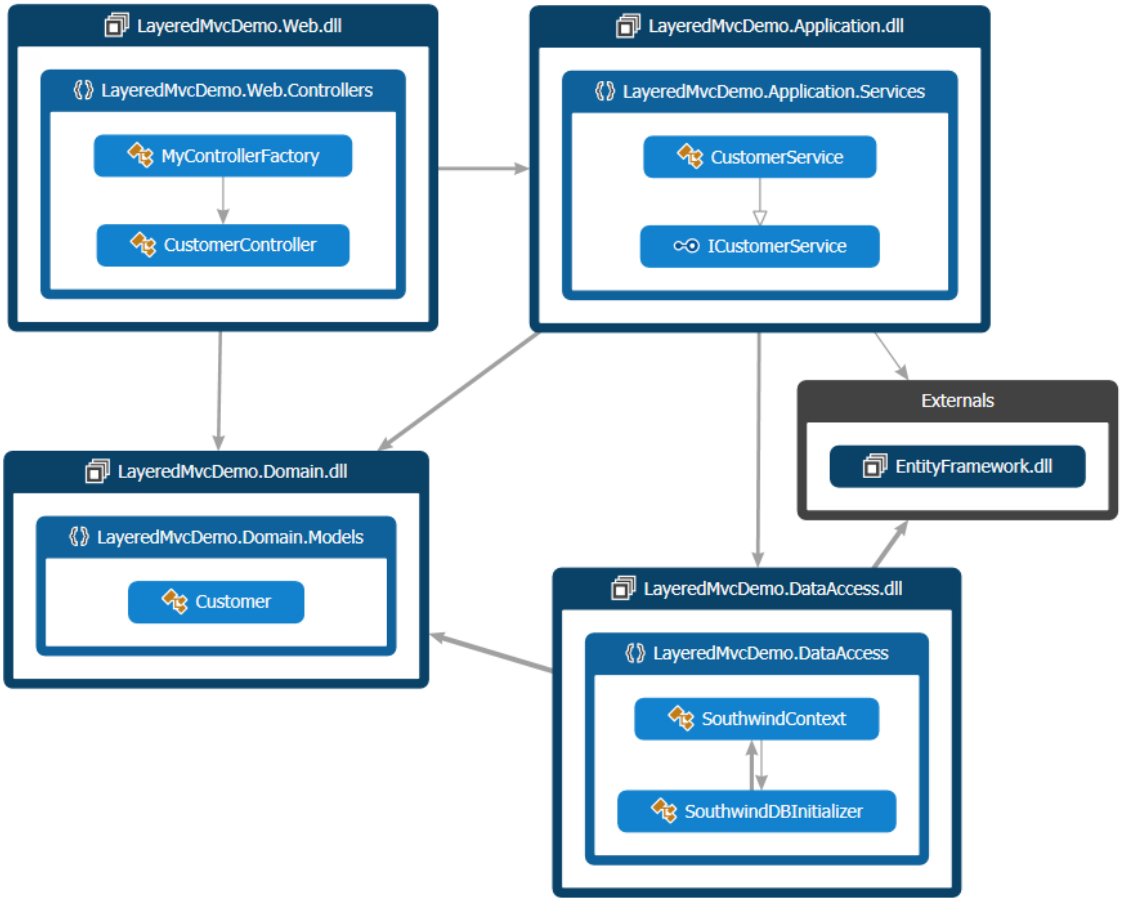
展現層

`CustomerController` 類別可維持原樣，不用修改。需要修改的是 `MyControllerFactory`，如下所示：

```
public class MyControllerFactory : DefaultControllerFactory
{
    public override IController CreateController(
        RequestContext requestContext, string controllerName)
    {
        if (controllerName == "Customer")
        {
            var service = new CustomerService();
            var controller = new CustomerController(service);
            return controller;
        }
        else
        {
            return null;
        }
    }
}
```

審視目前設計

移除 Repository 抽象層之後，應用程式各組件的相依關係如下圖所示。



DI 版本的組件相依關係圖

幾個值得觀察的地方：

- 和上一個版本一樣，領域模型仍然是位於相依關係的中心，亦即其他各層皆依賴領域模型。
- 展現層依然沒有依賴資料存取層。
- 由於拿掉了 Repository 抽象層，應用程式層現在也直接依賴 ORM 框架，即 Entity Framework。

一個 HTTP 請求搭配一個 DbContext

在上一節的範例中，服務類別（例如 `OrderService`）所需要的 `DbContext` 物件可透過建構函式傳入。實務上，在 ASP.NET 應用程式中，每一次 HTTP 請求的處理過程可能涉及多個服務類別來完成特定任務，而在該次 HTTP 請求的過程中，各服務類別通常都會使用同一個 `DbContext` 物件。那麼，既然需要「在同一個 HTTP 請求的範圍內共享同一個 `DbContext` 物件」，何不使用 **Ambient Context 模式**來簡化程式寫法？這樣就不用層層傳遞 `DbContext` 物件了。

本節範例的原始碼位置：

Examples/ch04/LayeredMvcDemo_V3.1_DbContextPerRequest 資料夾下，方案名稱是 LayeredMvcDemo.sln。

以下是實作步驟：

1. 修改 Global.asax.cs，在你的 `Application` 類別中加入以下兩個方法：

```
protected void Application_BeginRequest()
{
    HttpContext.Current.Items["DbContext"] = new SouthwindContext();
}

protected void Application_EndRequest()
{
    var db = HttpContext.Current.Items["DbContext"] as SouthwindContext;
    if (db != null)
    {
        db.Dispose();
    }
}
```

這樣會在每一次 HTTP 請求開始時建立一個新的 `SouthwindContext` 物件，並保存於目前 `HttpContext` 物件的 `Items` 集合屬性中。每當一個 HTTP 請求結束時，則將原先保存的 `SouthwindContext` 物件清除。

2. 為 SouthwindContext 類別加入一個靜態屬性 InstanceInCurrentRequest，用來傳回目前 HttpContext 中保存的 DbContext 物件：

```
public class SouthwindContext : DbContext
{
    public static SouthwindContext InstanceInCurrentRequest
    {
        get
        {
            return HttpContext.Current.Items["DbContext"] as SouthwindContext;
        }
    }
}
```

3. 每當程式中需要 SouthwindContext 物件時，便可透過靜態屬性 InstanceInCurrentRequest 取得與當前 HTTP 請求關聯的物件。如下所示：

```
public class CustomerService : ICustomerService
{
    private readonly SouthwindContext db;

    public CustomerService()
    {
        db = SouthwindContext.InstanceInCurrentRequest;
    }
}
```

如你所見，使用純手工 DI 時，許多細節都得自行處理（純手工的意思是不依靠任何 DI 容器）。在第 6 章，你將會看到使用 DI 容器來解決相同問題時能夠幫我們省下多少工夫。比如說，本章範例若捨棄純手工 DI 而改用 Unity，由於它會自動解析類別的建構函式所需之相依物件，而且是連鎖反應般的深層解析，所以像底下這些手動注入物件至建構函式的程式碼都免了：

```
var orderRepo = new OrderRepository();
var customerRepo = new CustomerRepository();
var productRepo = new ProductRepository();
var service = new OrderService(orderRepo, customerRepo, productRepo);
```

本書附的範例程式也提供了改用 Unity 的版本，如果你想要看程式碼，可以開啟書附範例的 Examples/ch04/LayeredMvcDemo_V4_Unity 資料夾下的 LayeredMvcDemo.sln。

到目前為止，有關分層架構的議題已經談得夠多了，接著要補充說明 ASP.NET MVC 5 提供的一個重要擴充機制：`IDependencyResolver` 介面。

ASP.NET MVC 5 的 `IDependencyResolver`

在前面的 MVC 分層架構範例 V2－寬鬆耦合 一節中，我們已經看過如何撰寫自訂的 `IControllerFactory` 類別來解決注入相依物件至 controller 建構函式的問題。這只是其中一種解法，而本節將介紹另一種解法：`IDependencyResolver` 介面。

`IDependencyResolver` 定義了 ASP.NET MVC 的型別解析服務所需之標準功能，讓外界得以透過實作此介面來建立自訂的型別解析器，以便進一步控制或抽換 MVC 框架所使用的一些內部元件。也就是說，ASP.NET MVC 框架在處理 HTTP 訊息的過程中，有許多地方都是仰賴此型別解析器來取得特定元件（介面）的執行個體，而我們只要實作了自己的型別解析器，並註冊至 ASP.NET MVC 框架，就能夠在 HTTP 訊息處理過程中加入額外的功能或改變其預設行為。

IDependencyResolver 介面的定義如下：

```
// 命名空間：System.Web.Mvc
public interface IDependencyResolver
{
    object GetService(Type serviceType);
    IEnumerable<object> GetServices(Type serviceType);
}
```

說明：

- GetService 方法接受外界指定要解析的服務型別，並傳回該服務的執行個體。若無法解析型別，則應傳回 null。
- GetServices 方法的功能同上，但是它可以傳回多個物件，特別用於一個型別對應多個物件的場合。如果無法解析指定之型別，則應傳回空集合。

實作這兩個方法時請特別注意：無論發生什麼意外都不可以拋異常（exception），否則會破壞 MVC 框架的後續處理流程。

實作自訂的 IDependencyResolver 元件

如同 IControllerFactory 的預設實作是 DefaultControllerFactory，ASP.NET MVC 框架也內建了 IDependencyResolver 介面的預設實作：DefaultDependencyResolver。同樣地，我們也可以撰寫自訂類別來取代既有的預設實作。

這裡延續先前的範例程式，將原本使用自訂 IControllerFactory 元件來建立 controller 物件的部分改為透過自訂的 IDependencyResolver 元件來處理。

此範例的原始碼位置：

Examples/ch04/LayeredMvcDemo_V3.2_DependencyResolver 資料夾下，方案名稱是 LayeredMvcDemo.sln。

我將這個自訂的類別命名為 MyDependencyResolver，程式碼如下：

```
public class MyDependencyResolver : System.Web.Mvc.IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        // 觀察 MVC 框架有哪些服務會透過 dependency resolver 來解析。
        System.Diagnostics.Debug.WriteLine(serviceType.FullName);

        // 解析特定 controller。
        if (serviceType == typeof(CustomerController))
        {
            var customerSvc = new CustomerService();
            var controller = new CustomerController(customerSvc);
            return controller;
        }

        // 不需要在此解析的型別，必須傳回 null。(不可拋異常！)
        return null;
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        // 沒有特別要解析的型別，故傳回空集合。(不可拋異常！)
        return new List<object>();
    }
}
```

注意 `GetService` 方法的 `serviceType` 參數，它就是呼叫端要求解析的服務類型。我在此方法中加入了除錯訊息，方便觀察 ASP.NET MVC 框架在處理 HTTP 請求的過程中會解析哪些型別。

接著要把 MVC 框架預設的 dependency resolver 換成我們自己的。作法是在 Global_-asax.cs 的 Application_Start 方法中呼叫 `System.Web.Mvc.DependencyResolver` 類別的靜態方法 `SetResolver`。如下所示：

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();
        RouteConfig.RegisterRoutes(RouteTable.Routes);

        // 設定 MVC 應用程式的全域 dependency resolver 物件。
        var myResolver = new MyDependencyResolver();
        DependencyResolver.SetResolver(myResolver);
    }
}
```

如此便完成了。你可以用 Visual Studio 開啟書附範例的專案，執行看看能否順利解析 `CustomerController`。

剛才提過，`MyDependencyResolver` 類別的 `GetService` 方法中有一行程式碼是用來觀察 MVC 框架解析服務的過程：

```
System.Diagnostics.Debug.WriteLine(serviceType.FullName);
```

以除錯模式執行此範例程式，然後查看 Visual Studio 的 Output 視窗，便可得知每當用戶端對某個 controller 發出 HTTP 請求時，ASP.NET MVC 框架會透過 `IDependencyResolver` 物件解析哪些元件。在我的開發環境上觀察的結果，應用程式第一次啟動時會解析下列型別：

```
System.Web.Mvc.IControllerFactory
System.Web.Mvc.IControllerActivator
LayeredMvcDemo.Web.Controllers.CustomerController
System.Web.Mvc.ITempDataProviderFactory
System.Web.Mvc.ITempDataProvider
System.Web.Mvc.Async.IAsyncActionInvokerFactory
System.Web.Mvc.IActionInvokerFactory
System.Web.Mvc.Async.IAsyncActionInvoker
System.Web.Mvc.IActionInvoker
```

後續發出的 HTTP 請求則只有解析當下請求的 controller：

```
LayeredMvcDemo.Web.Controllers.CustomerController
```

根據以上線索，你已經知道有哪些元件會在應用程式初次執行時透過 `IDependencyResolver` 物件來解析。如果你想要替換其他元件，只要以該介面的名稱作為關鍵字，上網搜尋一下，應該就能找到相關的技術文件和範例。

ASP.NET MVC 5 應用程式的生命週期

官方網站有一份精美的海報，裡面描繪了 MVC 5 的應用程式生命週期，亦有助於了解 MVC 框架的底層運作。網址如下：

http://i3.asp.net/media/4773381/lifecycle-of-an-aspnet-mvc-5-application.pdf

順便提及，`System.Web.Mvc.IControllerActivator` 也是一個可用來建立 controller 物件並注入相依物件的擴充點。從 ASP.NET MVC 3 開始，controller 工廠已經把「建立 controller」的責任切給了 `IControllerActivator` 的 `Create` 方法。所以，單就解析 controller 型別與注入相依物件的問題，你至少有三種解法可以選擇：`IControllerFactory`、`IControllerActivator`、以及 `IDependencyResolver`。一般的建議是盡量使用 `IDependencyResolver`，彈性較大。

本章回顧

本章首先概略介紹了分層架構的概念，且順便提及**「洋蔥架構」（Onion Architrecture）**和 **Repository 模式**。接著以三個版本的 ASP.NET MVC 範例程式來逐一示範從緊密耦合到寬鬆耦合的設計與修改過程，以及如何讓我們的 controller 類別也能使用建構式注入（**Constructor Injection**）。

談完分層架構的設計之後，本章最後一節則將焦點放在 MVC 框架提供的另一個擴充機制：`IDependencyResolver` 介面。其中的範例也展示了如何實作此介面來取代 ASP.NET MVC 預設的型別解析元件，以及注入相依物件至 controller 類別的建構函式。

本章範例程式皆採用純手工 DI，完全沒有使用 DI 容器，所以實際練習時會需要寫較多程式碼。到了第 6 章，DI 容器的份量加重，所有的範例都會使用 DI 容器來加速開發。

第 5 章：DI 與 ASP.NET Web API

ASP.NET Web API 框架的設計遵循「針對介面寫程式」的原則，其內部諸多元件不僅能夠替換成你的自訂實作，而且也能夠與其他 DI 容器整合。本章將以問題導向的方式，透過解題的範例程式帶出與 DI 擴充機制有關的幾個重要型別，包括 `ServicesContainer`、`DefaultServices`、以及 `IDependencyResolver`，並詳細說明這些擴充機制背後的運作原理。

內容大綱：

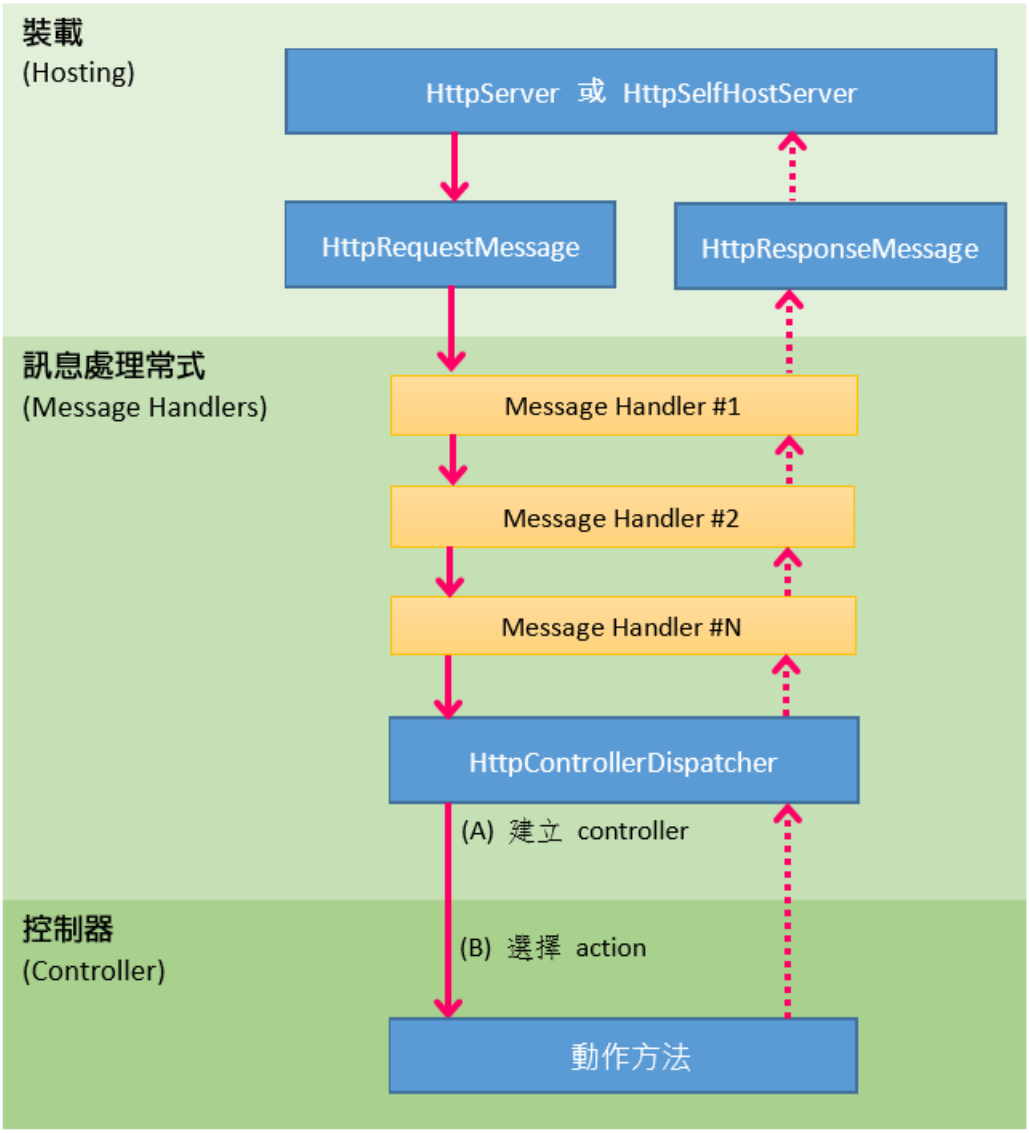
本章範例原始碼位置：

`https://github.com/huanlin/di-book-support` 裡面的 Examples/ch05 資料夾。Ch05.sln 即包含了本章全部的範例專案。

ASP.NET Web API 管線

ASP.NET Web API 框架為了提供彈性的擴充機制，本身在設計上便採用 DI 技術，而且能夠讓你運用自己熟悉的 DI 容器來將你的自訂元件／服務插入至 Web API 框架，或替換掉既有的服務。那麼，在整個 Web API 框架當中，有哪些擴充點可供我們利用，以及有哪些服務是可以替換的呢？欲回答此問題，我們得先了解 ASP.NET Web API 的 HTTP 訊息處理流程，或者說，Web API 的管線（pipeline）架構。

下圖即為簡化過的 ASP.NET Web API 訊息處理流程圖[22]：



ASP.NET Web API 管線架構）

[22]為了避免圖片太大以至於超過版面，上圖中的「HTTP 訊息處理常式」區塊省略了 HttpRoutingDispatcher 處理路由分派的部分。「控制器」區塊則省略了篩選條件（filter）的處理細節。微軟網站有提供一份比較完整的 Web API 訊息處理流程圖，網址是 http://www.microsoft.com/zh-tw/download/details.aspx?id=36476。

對於 HTTP request ／ response、controller、action 等名詞，筆者有時直接用英文，有時則直接用中文表達。例如：HTTP 請求／回應、控制器、動作方法（action method）等等。之所以會不一致，一方面因為平日講話時，有些名詞已經慣用英文了，比較自然；另一方面，有些地方為了避免閱讀時產生誤解而採用英文術語。請讀者明察。

此訊息管線架構圖分為三層，由上至下，分別是裝載（Hosting）、訊息處理常式（Message Handlers）、以及控制器（Controller）。圖中的紅色實心箭頭代表 HTTP 請求訊息，虛線箭頭代表 HTTP 回應訊息。訊息處理流程如下：

1. 當用戶端對伺服器發出的 HTTP 請求開始進入 ASP.NET Web API 框架時，該 HTTP 請求訊息會被包裝成 `HttpRequestMessage` 物件，進入圖中最頂端「裝載」方塊的 `HttpServer`（web 裝載）或 `HttpSelfHostServer`（自我裝載）。接著該訊息便流入管線的下一個階段，直到整個訊息流程處理完畢，會得到一個代表 HTTP 回應訊息的 `HttpResponseMessage` 物件，並將此物件的訊息內容傳回用戶端。
2. `HttpRequestMessage` 物件進入「訊息處理常式」管線。在此階段，HTTP 訊息行經數個訊息處理常式（message handlers），並且在返回 HTTP 回應訊息時以相反的順序執行。
3. 在各個訊息處理常式之後，HTTP 請求訊息接著會傳遞給 `HttpControllerDispatcher`，並且由這個物件來建立 Web API controller，然後將 HTTP 請求傳遞給 controller 物件（圖中標示「**(A) 建立 controller**」的步驟）。
4. Controller 會先決定目標動作方法（即圖中標示「**(B) 選擇 action**」的步驟），然後呼叫它。動作方法將負責產生回應內容，之後便依前述管線流程的反方向沿路返回。

以上便是 Web API HTTP 訊息管線的大致處理流程。其中關於建立 Web API controller 的部分與本章主題密切相關，所以接下來我們要進一步了解這個部分的細節。

訊息處理常式

如果您想要在 Web API 訊息管線中安插自己的訊息處理常式來對進入與返回的訊息做些額外的處理，例如：修改 HTTP 訊息標頭（headers）、身分驗證（authentication）、記錄訊息等等，你可以從 `DelegatingHandler` 衍生新的類別，並改寫其 `SendAsync` 方法；此方法接受一個 `HttpRequestMessage` 參數，並且傳回 `HttpResponseMessage` 物件。寫好的自訂訊息處理常式可以在應用程式啟動的時候加入 `HttpConfiguration` 的 `MessageHandlers`

> 集合，便可註冊成為 Web API 訊息管線中的一環。
>
> 進一步說，抽象類別 `DelegatingHandler` 定義了 `HttpRequestMessage` 和 `HttpResponseMessage` 物件的串接機制，讓 Web API 框架能依序呼叫各個訊息處理常式。那麼，如何串接呢？`DelegatingHandler` 有個 `InnerHandler` 屬性，型別是 `HttpMessageHandler`（它就是 `DelegatingHandler` 的父類別），而這個 `InnerHandler` 就是用來指向下一個訊息處理常式。好比拆禮物時，外包裝盒打開，裡面還有個盒子，再打開，裡面又還有個盒子……如此層層串接，各層之間的連結就是透過 `InnerHandler` 屬性。你也可以把它想成俄羅斯娃娃的結構，如果你知道這玩意的話。
>
> 此外，在剛才的管線架構圖裡面，頂端「裝載」區塊中的 `HttpServer` 也是繼承自 `DelegatingHandler` 類別，而 `HttpSelfHostServer` 又是繼承自 `HttpServer`。

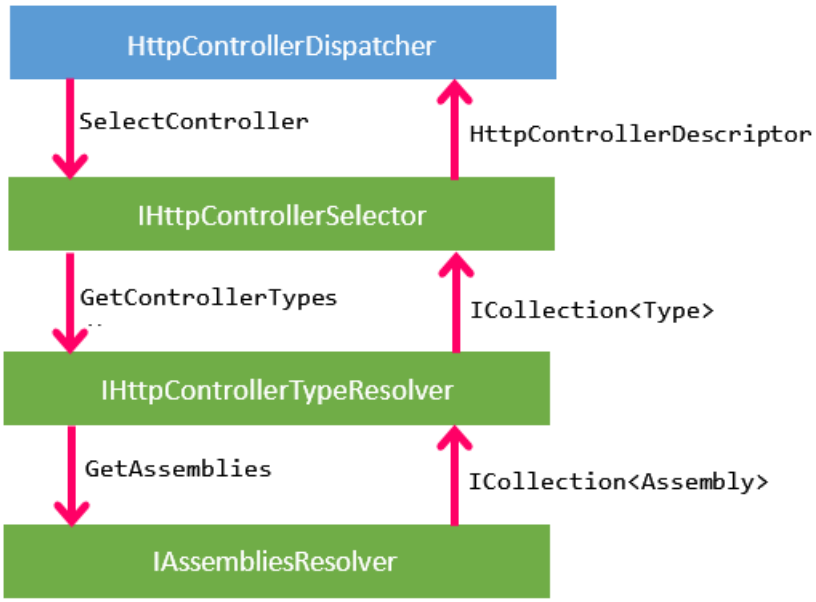
Controller 是怎樣建成的？

剛才只說明了 Web API HTTP 訊息管線的大致處理流程，而欲注入相依物件至 controller 類別的建構函式，或從中動些手腳來改變預設行為，必得了解 Web API 框架建立 controller 的內部過程。本節將進一步說明其中的複雜環節，其中會反覆提及多個抽象介面，第一次閱讀時可能略感吃力，並難免心生疑惑，但等到實際寫過、跑過一遍後面的範例程式，再回頭來看這一節的說明，整個拼圖應該就會漸漸明朗了。

剛才提到，`HttpControllerDispatcher` 會建立目標 controller 物件，亦即先前 ASP.NET Web API 管線架構圖中標示「**(A) 建立 controller**」的步驟。此步驟其實包含兩件工作：

1. 解析目標 controller。亦即決定該使用哪一個 controller 類別。
2. 建立目標 controller 類別的執行個體，並將 HTTP 請求（`HttpRequestMessage` 物件）傳遞給它，以便由 controller 進行後續處理。

首先，「解析目標 controller」的工作主要是從應用程式的 DLL 組件中尋找所有可用的 controller 類別，再從中選擇一個與當前 HTTP request 匹配的。其處理邏輯如下圖所示：



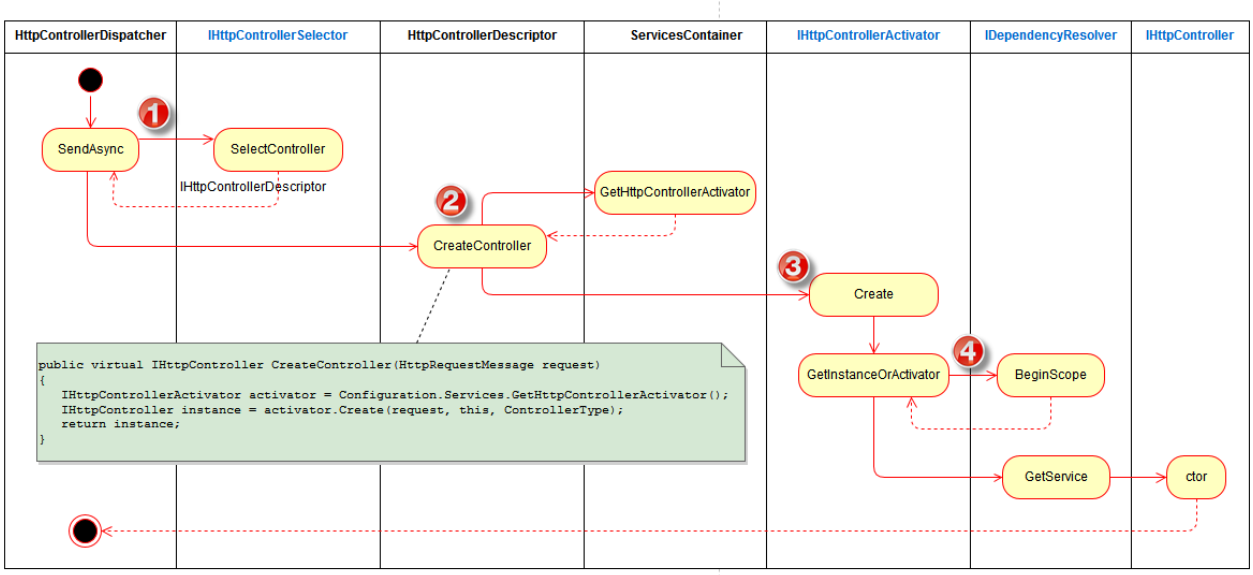
說明：

- 圖中下方的 `IAssembliesResolver` 物件的 `GetAssemblies` 方法將提供應用程式的組件清單，並由 `IHttpControllerTypeResolver` 物件的 `GetControllerTypes` 方法取得可用的 controller 類別清單。
- `IHttpControllerSelector` 負責決定要選擇哪一個 controller 類別，然後返回一個包含其型別資訊的 `HttpControllerDescriptor` 物件給 `HttpControllerDispatcher`。

此處「`IHttpControllerSelector` 物件」的意思是「實作了 `IHttpControllerSelector` 介面的物件」。其餘同樣的寫法也都是這個意思。ASP.NET Web API 框架大多有為這些核心標準介面提供預設實作，而且允許我們將預設實作換掉（稍後會示範作法）。例如，Web API 框架為 `IHttpControllerSelector` 提供的預設實作類別是 `DefaultHttpControllerSelector`；而在使用 web 裝載（而非自我裝載）的情況下，`IAssembliesResolver` 的預設實作類別是 `WebHostAssembliesResolver`。

從確定目標 controller 型別之後，到建立完成 controller 執行個體的過程中，還有經過一些核心標準介面所提供的擴充點，而這些擴充點就是稍後範例程式的主角。底下再用一張

UML 活動圖搭配 Web API 原始碼的方式來解構其內部處理過程（如果這張圖在你的閱讀裝置上看不清楚，請利用此網址查看原圖：http://goo.gl/OwiMWY）。



建立 controller 物件的內部流程

說明如下（與上圖中的數字編號對應）：

1. HttpControllerDispatcher 透過 IHttpControllerSelector 物件的 SelectController 方法來取得目標 controller 型別資訊，這型別資訊是包在一個 HttpControllerDescriptor 物件裡。
2. HttpControllerDispatcher 接著呼叫 HttpControllerDescriptor 物件的 CreateController 方法，而該方法又會去呼叫 ServicesContainer 物件的 GetHttpControllerActivator 方法來取得 IHttpControllerActivator 物件。以下程式片段摘自 Web API 原始碼，涵蓋了此步驟至下一步驟的部分邏輯：

```
// HttpControllerDescriptor 類別的 CreateController 方法。
public virtual IHttpController CreateController(HttpRequestMessage request)
{
    // 底下的 Configuration.Services 是個服務容器，後面會說明。
    IHttpControllerActivator activator =
        Configuration.Services.GetHttpControllerActivator();
    IHttpController instance = activator.Create(request, this, ControllerType);
    return instance;
}
```

3. 取得 `IHttpControllerActivator` 物件之後，便接著呼叫它的 `Create` 方法，而此方法會呼叫自己的 `GetInstanceOrActivator` 方法，以便取得 controller 執行個體。以下程式片段摘自 `DefaultHttpControllerActivator` 類別的原始碼，筆者將錯誤處理以及快取機制的部分拿掉，並加上了中文註解：

```
// DefaultHttpControllerActivator 類別的 Create 方法（重點摘錄）
public IHttpController Create(HttpRequestMessage request,
    HttpControllerDescriptor controllerDescriptor, Type controllerType)
{
    Func<IHttpController> activator;

    IHttpController controller =
        GetInstanceOrActivator(request, controllerType, out activator);
    if (controller != null)
    {
        // 註冊至 Web API 框架的 dependency resolver
        // 已經建立此 controller 型別的執行個體。
        return controller; // 那就直接使用此物件。
    }
    // 目標 controller 物件尚未建立，就用 GetInstanceOrActivator 方法
    // 所提供的委派來建立物件。
    return activator();
}
```

4. `IHttpControllerActivator` 物件的 `GetInstanceOrActivator` 方法會呼叫 `HttpRequestMessage` 的擴充方法 `GetDependencyScope` 來取得與當前 request 關聯的 `IDependencyScope` 物件（其實就是個 **Service Locator**），並利用它的 `GetService` 方法來取得 controller 物件。若 `GetService` 方法並未傳回 controller 物件，而是傳回 `null`（代表無法解析服務型別），則退而求其次，改用型別反射（reflection）機制來建立 controller 物件。一樣搭配原始碼來看：

```
// 摘自 DefaultHttpControllerActivator.cs
private static IHttpController GetInstanceOrActivator(
    HttpRequestMessage request, Type controllerType,
    out Func<IHttpController>> activator)
{
   // 若 dependency scope 有傳回 controller 物件，便使用它。
   IHttpController instance = (IHttpController) request
           .GetDependencyScope().GetService(controllerType);
   if (instance != null)
   {
      activator = null;
      return instance;
   }

   // 否則，建立一個委派來創建此型別的執行個體。
   activator = TypeActivator.Create<IHttpController>(controllerType);
   return null;
}
```

其中的 `request.GetDependencyScope()` 就是對應到剛才說的「呼叫 `HttpRequestMessage` 的擴充方法 `GetDependencyScope` 來取得與當前 request 關聯的 `IDependencyScope` 物件。」而這裡實際取得的 `IDependencyScope` 物件會是 Web API 框架提供的預設實作：`EmptyResolver`。從類別名稱可知，這類別其實啥事也沒做——它的 `GetService` 方法一律傳回 `null`。因此，在預設情況下，Web API 框架會一律使用型別反射（reflection）機制來建立 controller 物件，而這也就是為什麼我們的 controller 類別一定要有預設建構函式（default constructor）的緣故。

基本上，controller 物件就是這麼建成的。

Controller 建立完成後

Controller 物件建立完成後，HTTP request 就進入了 controller 的控制範圍。接下來，`ApiController` 會呼叫 `IHttpActionSelector` 物件的 `SelectAction` 方法來決定目標動作方法（action method）。Web API 框架也為 `IHttpActionSelector` 介面提供了一個預設實

> 作：ApiControllerActionSelector，而且我們也可以撰寫自訂類別把它給換掉。
>
> 此外，我們的 Web API controller 類別通常繼承自 ApiController，這是因為它實作了 IHttpController 介面，並提供了一些現成的功能——當然我們的 controller 類別也可以直接實作 IHttpController。

等等！仔細看一下，剛才描述的處理過程，最後一個步驟提到了 IDependencyScope，還提到了預設型別解析器 EmptyResolver，可是前面的活動圖裡面卻沒有它們，反而是 IDependencyResolver，為什麼呢？

欲回答此問題，還得再花些篇幅，再挖一些 Web API 原始碼，但我覺得該是動手寫點程式的時候了。目前請暫且記住以下兩點：

- IDependencyResolver 定義了 Web API 框架的型別解析操作之標準介面。
- EmptyResolver 是 Web API 框架提供的預設型別解析服務（但沒做任何事），它實作了 IDependencyResolver 介面，而 IDependencyResolver 又繼承自 IDependencyScope 介面。後面會有個範例展示如何抽換這個預設實作。

爬梳 Web API 管線架構的工作到此告一段落。接著要透過一個範例來展示說明如何將 Web API 框架提供的預設實作（預設服務）替換成我們的自訂服務——終於要開始寫程式啦！

注入物件至 Web API Controller

經過上一節的抽絲剝繭，您已經知道為什麼在預設情況下，我們的 API controller 必須提供預設建構函式。那麼，如果不提供預設建構函式會怎麼樣呢？比如說，我們很可能需要透過建構函式來為各個 controllers 動態注入所需之服務，例如底下這個 CustomerController 就沒有預設建構函式，而是要求外界必須從建構函式注入一個實作 ICustomerService 介面的服務：

```
public class CustomerController : ApiController
{
    private ICustomerService _customerService;

    public CustomerController(ICustomerService customerService)
    {
        _customerService = customerService;
    }

    public Customer Get(int id)
    {
        return _customerService.Get(id);
    }
}
```

程式執行時，若用戶端送出 *http://你的網站域名/api/Customer/Get?id=1* 請求，ASP.NET Web API 框架就會拋出以下錯誤訊息：

> An error occurred when trying to create a controller of type 'CustomerController'. Make sure that the controller has a parameterless public constructor.
>
> （嘗試建立 'CustomerController' 時發生錯誤。請檢查該 controller 類別是否有提供不帶任何參數的公開建構函式。）

針對注入物件至 Web API controller 的問題，常見的解法有兩種：

1. 抽換 `IHttpControllerActivator` 服務。亦即寫個類別來實作 `IHttpControllerActivator` 介面（命名空間 `System.Web.Http.Dispatcher`），並取代 Web API 提供的預設實作（`DefaultHttpControllerActivator`）。
2. 抽換 `IDependencyResolver` 服務。亦即寫個類別來實作 `IDependencyResolver` 介面（命名空間 `System.Web.Http.Dependencies`），並取代 Web API 提供的預設實作（`EmptyResolver`）。

這兩種方法都能夠讓你將自訂物件插入至 ASP.NET Web API 框架中，以取代或擴充預設的功能或服務。接下來的兩個小節將分別進一步說明這兩種解法。

抽換 IHttpControllerActivator 服務

IHttpControllerActivator 在 ASP.NET Web API 框架中的角色，是負責定義「如何建立一個 controller」，這點在稍早的〈Controller 是如何建成的？〉一節中已略見端倪。此介面只包含一個方法：Create。如下所示：

```
public interface IHttpControllerActivator
{
    IHttpController Create(
        HttpRequestMessage request,
        HttpControllerDescriptor controllerDescriptor,
        Type controllerType
    )
}
```

每當 ASP.NET Web API 框架要為當前 HTTP request 建立相應的 controller 時，就會呼叫 IHttpControllerActivator 物件的 Create 方法來建立目標 controller 物件。Web API 已經內建了一個 IHttpControllerActivator 預設實作，其類別名稱是 DefaultHttpControllerActivator。我們的目標就是要把這個預設實作換掉，換成我們自己的。

純手工打造

先來看一個完全不使用 DI 容器的簡單範例：

```
public class MyHttpControllerActivator : IHttpControllerActivator
{
    public IHttpController Create(
        HttpRequestMessage request,
        HttpControllerDescriptor controllerDescriptor,
        Type controllerType)
    {
        if (controllerType == typeof(CustomerController))
        {
            var customerSvc = new CustomerService(); // 建立相依物件
            request.Properties.Add("Time", DateTime.Now); // 傳遞額外資訊
            return new CustomerController(customerSvc); // 建立控制器並注入物件
        }
        return null;
    }
}
```

本節範例的專案名稱是 Ex01.HttpControllerActivator.csproj。

程式說明：

- `MyHttpControllerActivator` 類別實作了 `IHttpControllerActivator` 介面，而建立 controller 和注入相依物件的處理都是寫在 `Create` 方法中。
- 這裡還示範了透過 `Create` 方法傳入的 `HttpRequestMessage` 物件來傳遞額外資訊（目前時間）給目標 controller。雖然與主旨無關，但可以發現實作自訂的 `IHttpControllerActivator` 物件的一個附帶好處，就是能夠取得當前的 `HttpRequestMessage` 物件，並且視需要利用該物件的 `Properties` 屬性來傳遞一些額外資訊給 controller。

寫完 `IHttpControllerActivator` 的實作類別之後，還要把 Web API 內部的預設實作換成我們的 `MyHttpControllerActivator` 物件才行。以下程式片段即示範了如何在應用程式啟動時將預設的 `IHttpControllerActivator` 服務換成我們的自訂服務：

```
public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        var myControllerActivator = new MyHttpControllerActivator();
        GlobalConfiguration.Configuration.Services.Replace(
            typeof(IHttpControllerActivator), myControllerActivator);
    }
}
```

程式說明：

- Web API 的標準服務是統一註冊在一個全域的服務容器裡，而我們可以透過 `GlobalConfiguration.Configuration` 提供的 `Services` 屬性來取得此服務容器（其型別為 `ServicesContainer`）。
- 取得服務容器之後，便可呼叫它的 `Replace` 方法來抽換容器內部的既有服務。

> 如果你的專案是利用 Visual Studio 的 ASP.NET Web API 專案範本（template）所建立的，那麼你的 Global_asax.cs 中的 `Application_Start` 方法會有一行敘述將設定 Web API 的工作委派給另一個函式，像這樣：
>
> ```
> protected void Application_Start()
> {
>     GlobalConfiguration.Configure(WebApiConfig.Register);
> }
> ```
>
> 同時，Web API 專案範本會在你的專案的 App_Start 資料夾下產生 WebApiConfig.cs。於是我們可以將「替換 `IHttpControllerActivator` 實作」的程式碼寫在靜態類別 `WebApiConfig` 的 `Register` 方法裡，如下所示：
>
> ```
> public static class WebApiConfig
> {
>     public static void Register(HttpConfiguration config)
>     {
>         // （省略其他設定 Web API 的程式碼）
>
>         // 替換 IHttpControllerActivator 實作
> ```

```
            config.Services.Replace(typeof(IHttpControllerActivator),
                                    new MyHttpControllerActivator());
        }
    }
```

這裡再重點整理一下抽換 IHttpControllerActivator 服務的基本步驟：

1. 撰寫一個自訂類別（例如 MyHttpControllerActivator），此類別須實作 IHttpControllerActivator 介面。
2. 在應用程式啟動時（Application_Start 方法）將 Web API 的預設實作（Default-HttpControllerActivator）替換成我們的自訂實作（MyHttpControllerActivator）。如何替換？透過全域的 HttpConfiguration 物件的 Services 屬性來取得服務容器，然後呼叫它的 Replace 方法。

使用 DI 容器：Unity

先前的範例程式係採用純手工 DI 的作法。若使用 DI 容器，程式碼可以再簡潔一些。這裡用 Unity 來示範，以便與前面的範例做個對比。對於 Unity API 的部分，本書第 7 章有比較詳細的說明，這裡就不多做解釋。

本節範例的專案名稱是 Ex01.HttpControllerActivator.Unity.csproj。

首先，為專案加入 Unity 組件參考，然後修改 MyHttpControllerActivator：

```
public class MyHttpControllerActivator : IHttpControllerActivator
{
    private IUnityContainer _container;

    // 由外界注入 DI 容器。
    public MyHttpControllerActivator(IUnityContainer container)
    {
        _container = container;
    }

    public IHttpController Create(
        HttpRequestMessage request,
        HttpControllerDescriptor controllerDescriptor,
        Type controllerType)
    {
        if (controllerType == typeof(CustomerController))
        {
            var customerSvc = new CustomerService(); // 建立相依物件
            return new CustomerController(customerSvc); // 建立 controller 並注入物件
        }
        return null;

        var controller = _container.Resolve(controllerType);
        return controller as IHttpController;
    }
}
```

跟原先未使用 DI 容器的程式碼比較一下（標示刪除線的部分），使用 DI 容器之後，程式碼可以省去一堆 `if...else` 的判斷邏輯，變得更簡潔。當然，我們得先設定 DI 容器並註冊型別才行。設定 DI 容器的工作可以寫在 `WebApiConfig.Register` 方法裡面，像這樣：

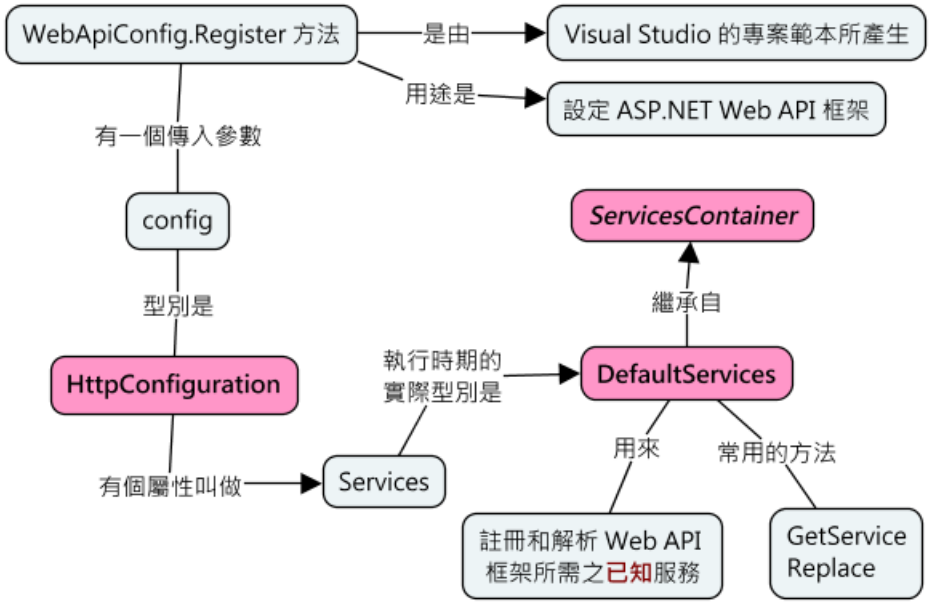
```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var container = new UnityContainer();
        container.RegisterType<CustomerController>();
        container.RegisterType<ICustomerService, CustomerService>();
        var myControllerActovator = new MyHttpControllerActivator(container);
        config.Services.Replace(typeof(IHttpControllerActivator),
                                myControllerActovator);
    }
}
```

如果您只想知道如何替換預設的 IHttpControllerActivator 物件，以便注入相依物件至 controller 的建構函式，那麼看到這裡就夠了，可以跳到下一節接著讀。但如果您想要進一步了解「替換 IHttpControllerActivator 實作」背後的細節，請耐心接著往下看。

現在，請把焦點放在剛才的範例程式的最後一行敘述：

```
// 替換 IHttpControllerActivator 實作
config.Services.Replace(typeof(IHttpControllerActivator),
                        new MyHttpControllerActivator());
```

由前述範例可知，變數 config 的型別是 HttpConfiguration 類別，而這個類別有個 Services 屬性，型別是 ServicesContainer，用來記錄已註冊之服務。然而，ServicesContainer 是個抽象類別，所以這裡的 Services 屬性在執行時期的實際型別肯定另有其人——那就是 System.Web.Http.Services.DefaultServices。好像有點複雜？那麼試試搭配以下概念圖一起看，也許會清楚些：



概念圖：設定 ASP.NET Web API 服務容器

ServicesContainer 類別隸屬於命名空間 System.Web.Http.Controllers，類別名稱中的「Services」是複數，不要跟另一個名稱很像的 ServiceContainer 類別搞混喔。

顧名思義，ServicesContainer 就是「服務的容器」，這個容器的設計是採用 **Service Locator 模式**的概念，而 DefaultServices 即實作了 ServicesContainer 抽象概念所定義的功能。基本上，你可以把 DefaultServices 視為 Web API 內建的一個簡易 DI 容器。此容器類別的 Replace 方法可以讓我們把原先已經加入至容器的「型別－執行個體」對應關係替換掉；換言之，即重新指定某型別（上述範例中的 IHttpControllerActivator）所對應之物件（上述範例中的 MyHttpControllerActivator 執行個體）。如果要向此容器取得指定類型的服務，則可利用 GetService 方法。底下是 GetService 和 Replace 方法的原型宣告：

```
public abstract Object GetService(Type serviceType)
public void Replace(Type serviceType, Object service)
```

可想而知，DefaultServices 內部必然有一個類似「型別－物件」對應表的結構。一旦將某服務類型的執行個體（instance）註冊到這個容器裡，以後每當用戶端需要該類型的服務時，只要呼叫容器的 GetService 方法，就能取得指定類型服務的的執行個體。這也是為什麼剛才說 ServicesContainer 是採用 **Service Locator 模式**的原因。

以下程式片段示範兩種取得特定服務的方法：

```
var svc1 = config.Services.GetService(typeof(IHttpControllerActivator))
    as MyHttpControllerActivator; // GetService 回傳的型別是 Object，須手動轉型。
var svc2 = config.Services.GetHttpControllerActivator(); // 效果同上，但免轉型。
```

擴充方法 GetHttpControllerActivator 來自 System.Web.Http.ServicesExtensions 類別，好處是提供了強型別，不用像使用 GetService 方法那樣得將回傳的 Object 型別手動轉型。

如需複習 **Service Locator 模式**的基礎觀念，請參閱第 2 章 〈Service Locator 模式〉 一節的說明。

如欲了解 ServersExtensions 類別提供了哪些擴充方法，可參閱 MSDN 線上文件：http://msdn.microsoft.com/zh-tw/library/system.web.http.servicesextensions_methods.aspx

這裡有個小細節，與第 3 章提過的物件生命週期選項有關，必須特別說明。

請再看一眼剛才列出的 ServicesContainer 類別的 Replace 方法的原型宣告，它的第一個參數是個 Type（型別），而第二個參數是個 Object（物件）。如果你到 MSDN 網站上查看該類別的 Add 方法，也會發現該方法需要的兩個參數也同樣是「型別」和「物件」。由這些線索可以得出一個結論：ServicesContainer 這個抽象類別並不打算幫你建立服務的物件實體，而只是單純記住你傳入的物件參考而已。換句話說，它並沒有像一般 DI 容器那樣

提供多種物件生命週期選項。原因在於，抽象類別 `ServicesContainer` 假定它所支援的服務都是採用「註冊一次，不斷重複使用」的方式。換言之，註冊至該容器中的服務是以類似 **Singleton** 的方式運作。

一般而言，在向 ASP.NET Web API 的服務容器註冊你的自訂服務時，你通常可以選擇要直接提供一個現成的物件，或者一個具象類別（concrete class）。若選擇前者，亦即在註冊服務時提供一個現成的物件，那麼往後 ASP.NET Web API 框架在解析該服務時便會使用你提供的那個物件，而不會去另外建立新的執行個體——換言之，它是以 **Singleton** 的方式來使用這些服務。若選擇後者，即註冊服務時提供的是一個具象類別，此時你通常還可以選擇該服務的物件生命週期，例如 **Transient**、**Singleton**、**Per HTTP Request** 等等（這些物件生命週期選項都在第 3 章提過）。

還有一點也很重要：`DefaultServices` 是 ASP.NET Web API 框架內部專用的容器，它只支援它認得的「已知服務類型」（即先前提過的「標準服務」），而且對這些已知服務的操作有諸多限制。簡單地說，你不能拿它來當作一般的通用容器，任意對它註冊自訂型別或移除已知服務類型。例如底下這些程式碼都會導致執行時期拋出 `ArgumentException`，錯誤訊息是「不支援服務類型 Xyz」：

```
// 以下程式碼都會造成執行時期錯誤！（變數 config 的型別是 HttpConfiguration）
config.Services.Add(typeof(IMyService), new MyService());
config.Services.Remove(typeof(IHttpControllerActivator), myControllerActivator);
var svclist = config.Services.GetServices(typeof(IHttpControllerActivator));
```

那麼，ASP.NET Web API 的 `DefaultServices` 允許外界抽換哪些抽象型別的實作呢？底下這張表格列出了幾個相關服務：

服務型別	說明
IActionValueBinder	繫結某個 action 的參數。
IAssemblyResolver	取得應用程式的組件（集合）。
IApiExplorer	負責蒐集 API 相關資訊。
IContentNegotiator	在 HTTP 管線中進行內容協商（content negotiation）。
IHttpActionInvoker	執行選定之 action。
IHttpActionSelector	選擇用來處理當前 request 的 action。
IHttpControllerActivator	建立當前 request 所需之 controller 執行個體。
IHttpControllerSelector	尋找並選擇應該負責處理某 HTTP request 的 controller。
IHttpControllerTypeResolver	用來取得應用程式的 controller 型別（集合）。
ITraceWriter	負責輸出追蹤訊息。

可替換的服務不少，限於篇幅，這裡無法一一舉例說明其用法。若您需要抽換其他服務類型，不妨參考前面的範例，同時搭配 MSDN 線上文件的說明，應該就能有一些頭緒。

接著來看另一種解法：抽換 Web API 的內建的 `IDependencyResolver` 服務。

抽換 IDependencyResolver 服務

經過上一節的說明，我們已經知道：`HttpConfiguration` 類別有一個 `Services` 屬性代表一個全域的服務容器，它可以讓我們取得或替換 Web API 標準服務。當你很清楚想要抽換哪一個標準服務時，就可以參考上一節範例的作法，將你撰寫的自訂服務註冊到這個服務容器中。

除了透過 `Services` 屬性來操作 Web API 已知服務，`HttpConfiguration` 還提供了另一個擴充點，就是 `DependencyResolver` 屬性，其型別為 `IDependencyResolver` 介面。此介面

在稍早說明 controller 的建立過程時曾概略提過，但尚未詳細說明背後的運作機制。現在就要來補上這個缺口，然後再示範其用法。

在稍早的〈Controller 是如何建成的？〉一節中所摘錄的最後一段原始碼當中，有一行比較不好理解：

```
IHttpController instance = (IHttpController)request.GetDependencyScope()
                                                    .GetService(controllerType);
```

當時有解釋，其中的 `request.GetDependencyScope()` 是呼叫 `HttpRequestMessage` 的擴充方法 `GetDependencyScope` 來取得一個 `IDependencyScope` 物件，然後緊接著呼叫它的 `GetService` 方法來取得 controller 物件。這裡實際取得的 `IDependencyScope` 物件會是 Web API 框架提供的預設實作：`EmptyResolver`。

請先看一下擴充方法 `GetDependencyScope` 的原始碼，如下所示（中文註解是筆者自行加上的）：

```
// 摘自 HttpRequestMessageExtensions.cs
static IDependencyScope GetDependencyScope(this HttpRequestMessage request)
{
    IDependencyScope result;

    // 嘗試從當前 request 物件的 Properties 屬性中取得 IDependencyScope 物件。
    if (!request.Properties.TryGetValue<IDependencyScope>(
            HttpPropertyKeys.DependencyScope, out result))
    {
        // 當前的 request 物件並沒有關聯的 IDependencyScope 物件，
        // 那就從全域組態中取得已註冊的 IDependencyResolver 物件
        IDependencyResolver dependencyResolver =
            request.GetConfiguration().DependencyResolver;

        // 並且利用此 dependency resolver 來建立一個 dependency scope（子範圍）。
        result = dependencyResolver.BeginScope();

        // 建立好 scope 之後，將此 scope 物件關聯至當前的 request
        request.Properties[HttpPropertyKeys.DependencyScope] = result;
```

```
            // 並且向當前 request 註冊：這個 scope 物件是可隨著 request 物件一起回收。
            request.RegisterForDispose(result);
        }
        return result;
    }
```

程式碼中的註解已經點出了幾個重點：

- 每當 Web API 要解析 controller 型別時，會先嘗試取得「與當前 request 關聯的 IDependencyScope 物件」。稍後你將會明白，剛才用引號包住的文字其實也可以改成「與當前 request 關聯的 DI 容器」。
- 如果當前的 HTTP request 並沒有任何關聯的 IDependencyScope 物件，那就透過 HttpConfiguration.DependencyResolver 來取得全域的 DI 容器，並利用此全域 DI 容器來建立一個新的 IDependencyScope 物件，然後存入當前的 request 物件。如此一來，當前的 HTTP request 物件一定會關聯一個（它專屬的）IDependencyScope 物件。

擴充方法 GetDependencyScope 傳回 IDependencyScope 物件之後，呼叫端便接著呼叫它的 GetService 方法來取得指定 controller 型別的執行個體（instance）。

打通此環節的最後一哩，就是釐清 IDependencyResolver 和 IDependencyScope 的角色，以及它們之間的關係。

IDependencyResolver 與 IDependencyScope

IDependencyResolver 介面隸屬於命名空間 System.Web.Http.Dependencies，其定義如下：

```
public interface IDependencyResolver : IDependencyScope, IDisposable
{
    IDependencyScope BeginScope();
}
```

如您所見，IDependencyResolver 介面只定義了一個 BeginScope 方法，此方法會傳回一個 IDependencyScope 物件。同時，IDependencyResolver 又繼承自 IDependencyScope 介面，因此它也繼承了該介面所定義的兩個方法：

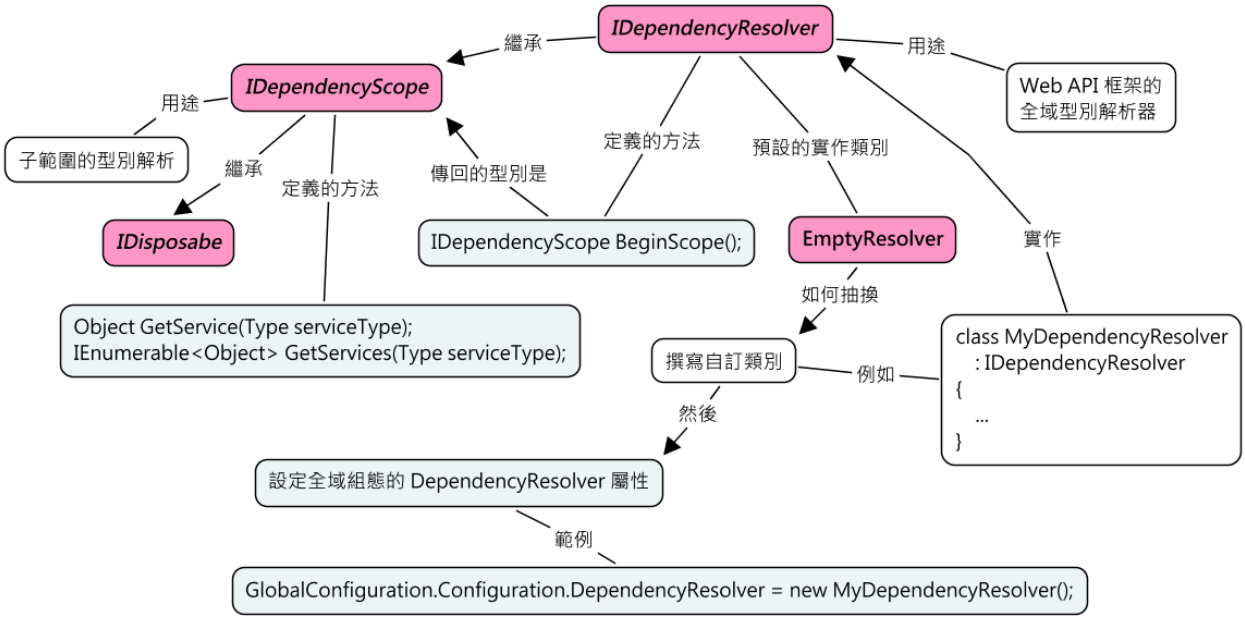
```
public interface IDependencyScope
{
    Object GetService(Type serviceType);
    IEnumerable<Object> GetServices(Type serviceType);
}
```

函式名稱已經清楚表達了它們的用途：

- GerService 是用來取得指定型別之執行個體。
- GetServices 則是用來取得指定型別之執行個體集合。有些服務類型可以對應至多個物件，故需要此方法。

ASP.NET Web API 框架會知道哪些情況該呼叫 GetService 還是 GetServices 來取得一個或多個物件。

現在比較清楚了，IDependencyResolver 或 IDependencyScope 都有 GetService 方法，這表示它們都有「型別解析」的功能。IDependencyResolver 的角色就像是 Web API 框架的全域型別解析器，而它的 BeginScope 方法可用來建立一個新的「子範圍」或「子容器」；這個子容器（IDependencyScope）就不必是唯一或全域物件，而且它有自己的生命周期，可在適當時機回收（disposable）。下圖描繪了這兩個介面的角色與關係，以及如何抽換 Web API 預設的型別解析器（若圖片不夠清晰，可點此連結查看原圖：http://goo.gl/Ut5JBy）。



概念圖：Web API 的 Dependency Resolver

請注意，Web API 框架在處理每一個 HTTP request 時都會呼叫一次 `BeginScope` 方法來取得一個新的「子範圍」或「子容器」，然後在 request 結束時呼叫 `IDependencyScope.Dispose()`。正如先前提過的，子容器的一個用途是關聯至當前的 HTTP request，亦即讓每一個 `HttpRequestMessage` 物件都有自己專屬的子容器來提供型別解析的服務，而且個別子容器將隨著它所屬的 request 結束時一併消滅。

純手工 DI 範例

接著便以一個範例來說明如何透過實作自訂的 `IDependencyResolver` 來解決先前的問題：注入相依物件至 controller 的建構函式。

此範例的專案名稱是 Ex02.DependencyResolver.csproj。

步驟 1：實作 IDependencyResolver 介面

首先，撰寫一個類別來實作 IDependencyResolver 介面，姑且將它命名為 MyDependencyResolver，程式碼如下：

```
public class MyDependencyResolver : IDependencyResolver
{
    public IDependencyScope BeginScope()
    {
        return this;
    }

    public object GetService(Type serviceType)
    {
        if (serviceType == typeof(CustomerController))
        {
            var service = new CustomerService();
            return new CustomerController(service);
        }
        return null;
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return new List<object>();
    }

    public void Dispose()
    {
    }
}
```

程式說明：

- BeginScope 方法只是單純地傳回物件本身，因為此範例不需要建立特定類型的子容器。
- GetService 方法使用了一個簡單的邏輯來判斷當前要解析的 controller 型別是否為 CustomerController；如果是，便自行建立相依物件，並注入至

CustomerController 的建構函式，然後傳回這個物件。於是，Web API 框架就會直接使用我們建立的 controller 物件，而不會使用預設的型別反射機制來建立 controller（原因詳見〈Controller 是如何建成的？〉一節的說明）。

- GetServces 方法在這裡沒有作用。

基本上，實作 IDependencyResolver 就等於是在實作一個 **Service Locator**。

要特別說明的是，此範例的 Dispose 方法不需要作任何事。如果你在 Dispose 方法中加了一些釋放 instance 內部資源的程式碼，反而可能造成應用程式執行時發生錯誤。原因在於，此範例的 BeginScope 方法傳回的是物件本身（this），也就是說，MyDependencyResolver 在這裡同時擔任了 resolver 和 scope 的角色，而 ASP.NET Web API 每處理一個 request 就會呼叫一次 BeginScope 方法；因此，如果你在 Dispose 方法中釋放 instance 資源，那麼當應用程式收到第二次 request 時，便可能因為上次建立的內部資源已經被回收，而發生執行時期錯誤。

注意：GetService 方法若無法解析指定之型別，則必須傳回 null，不可拋出異常。同樣的，GetServices 若無法解析型別，亦不可拋異常，而應傳回空集合。Web API 框架如果發現 IDependencyResolver 物件傳回 null，它會採用其他方式（例如 reflection）解析型別；拋出異常將導致此解析流程中斷。

步驟 2：替換預設的型別解析器

接著就是要把剛才寫好的 MyDependencyResolver 注入至 Web API 框架中。注入的時機是應用程式初始設定的時候，注入的管道則是先前提過的 HttpConfiguration 物件的 DependencyResolver 屬性。參考以下程式片段：

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.DependencyResolver = new MyDependencyResolver();
    }
}
```

如此便完成了，效果與上一節〈抽換 IHttpControllerActivator 服務〉的範例相同。

MyDependencyResolver 在整個應用程式生命週期中只會建立一次。

如果您想要進一步了解 Web API 框架的型別解析過程，這裡有個簡單的小實驗可以試試。作法很簡單：在剛才的 MyDependencyResolver 類別的 GetService 方法中加入一行程式碼，用來輸出目前欲解析的型別名稱，以便觀察整個型別解析的過程。參考以下程式片段：

```
public class MyDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        // 觀察 Web API 框架在哪些時候會呼叫此方法來解析什麼型別
        System.Diagnostics.Debug.WriteLine(serviceType.FullName);
    }
}
```

以下是在我的開發環境上觀察到的，初次啟動應用程式時解析的型別：

```
System.Web.Http.Metadata.ModelMetadataProvider
System.Web.Http.Tracing.ITraceManager
System.Web.Http.Tracing.ITraceWriter
System.Web.Http.Dispatcher.IHttpControllerSelector
System.Web.Http.Dispatcher.IAssembliesResolver
```

```
System.Web.Http.Dispatcher.IHttpControllerTypeResolver
System.Web.Http.Controllers.IHttpActionSelector
System.Web.Http.Controllers.IActionValueBinder
System.Web.Http.Hosting.IHostBufferPolicySelector
```

而在第一次發出 Web API 請求時（例如 /api/Customer/Get?id=1），則會解析下列型別：

```
System.Web.Http.Dispatcher.IHttpControllerActivator
Ex02_DependencyResolver.Controllers.CustomerController
System.Web.Http.Validation.IModelValidatorCache
System.Web.Http.Controllers.IHttpActionInvoker
System.Net.Http.Formatting.IContentNegotiator
System.Web.Http.ExceptionHandling.IExceptionHandler
```

對同一個 `CustomerController` 發出第二次請求時，需要解析的型別就只有一個：

```
Ex02_DependencyResolver.Controllers.CustomerController
```

使用 DI 容器：Unity

透過上一節的 `MyDependencyResolver` 範例，您已經知道如何自行實作 `IDependencyResolver` 來取代 Web API 的預設型別解析器（`EmptyResolver`）。然而我們也不見得要自己寫，因為現成的 DI 容器（例如 Unity、Autofac）大多有提供對應的功能，很容易和 `IDependencyResolver` 銜接。底下便是一個範例，示範如何搭配 Unity 容器來實作 `IDependencyResolver`。

此範例的專案名稱是 Ex02.DependencyResolver.Unity.csproj。

```
public class UnityDependencyResolver : IDependencyResolver
{
    protected readonly IUnityContainer _container;

    public UnityDependencyResolver(IUnityContainer container)
    {
        _container = container;
    }

    public IDependencyScope BeginScope()
    {
        IUnityContainer childContainer = _container.CreateChildContainer();
        return new UnityDependencyResolver(childContainer);
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return _container.Resolve(serviceType);
        }
        catch
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return _container.ResolveAll(serviceType);
        }
        catch
        {
            return new List<object>();
        }
    }

    public void Dispose()
    {
        _container.Dispose();
    }
```

```
}
```

如您所見，UnityContainer 提供了 CreateChildContainer 方法來建立子容器，此功能正好能滿足 IDependencyResolver.BeginScope 方法的需要。

關於 Unity 的子容器以及 HierarchicalLifetimeManager 的用法，本書第 7 章〈Unity 學習手冊〉有更詳細的介紹。

剩下的工作，就是在應用程式的「組合根」進行設定容器與註冊型別的動作，並且將 Web API 預設的型別解析器給換掉。參考以下範例：

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // 其他 Web API 組態設定的程式碼已省略。

        // 使用簡易的 MyDependencyResolver 來取代預設的 EmptyResolver
        config.DependencyResolver = new MyDependencyResolver();

        // 使用 UnityDependencyResolver 來解析
        var container = new UnityContainer();
        var lifetimeMngr = new HierarchicalLifetimeManager();
        container.RegisterType<ICustomerService, CustomerService>(lifetimeMngr);

        var resolver = new UnityDependencyResolver(container);
        config.DependencyResolver = resolver;
    }
}
```

如果你覺得自行實作 IDependencyResolver 太麻煩（如剛才的 UnityDependencyResolver），那麼不妨試試一個名叫 Unity.WebApi 的第三方套件。該套件提供了現成的 IDependencyResolver 實作類別，可以讓你少寫一些程式碼。

使用 DI 容器：Autofac

Autofac 框架已內建整合 Web API 的功能，而且有提供現成的 `IDependencyResolver` 實作類別。欲使用 Autofac 來達到上一節範例的效果，首先要用 NuGet 套件管理員來為專案加入「Autofac.WebApi2」套件。此一動作會加入下列組件：

- Autofac（核心組件）
- Autofac.Integration.WebApi（Web API 整合組件）

然後，在 `WebApiConfig` 類別的 `Register` 方法中設定 Autofac 容器，並將 Web API 預設的型別解析器換成 Autofac 提供的 `AutofacWebApiDependencyResolver`。參考以下程式碼：

```
using Autofac;
using Autofac.Integration.WebApi;
....
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // 其他 Web API 組態設定的程式碼已省略。

        var builder = new ContainerBuilder();
        builder.RegisterApiControllers(typeof(WebApiConfig).Assembly);
        builder.RegisterType<CustomerService>().As<ICustomerService>();

        var container = builder.Build();
        var resolver = new AutofacWebApiDependencyResolver(container);
        config.DependencyResolver = resolver;
    }
}
```

此範例的專案名稱是 Ex02.DependencyResolver.AutofacDemo.csproj。

本章回顧

本章從 ASP.NET Web API 的管線處理流程開始談起。然後，藉由爬梳 Web API 原始碼，我們看到了 controller 的創建過程，以及此過程所涉及的相關介面與類別，包括 `HttpControllerDispatcher`、`IHttpControllerActivator`、`IHttpController`、`IDependencyResolver`、`IDependencyScope` 等等。

接著，針對注入物件至 Web API controller 的問題，介紹了兩種常見解法：

- 解法一：實作自訂的 `IHttpControllerActivator` 服務來取代 Web API 內建的預設實作 `DefaultHttpControllerActivator`。
- 解法二：實作自訂的 `IDependencyResolver` 服務來取代 Web API 內建的預設實作 `EmptyResolver`。

一般建議是採用第二種解法，也就是實作自訂的 `IDependencyResolver` 服務；主要是因為彈性較大，而且能夠與第三方 DI 容器銜接。

除了純手工 DI 的寫法，本章最後也分別以兩個範例程式來說明如何利用 Unity 和 Autofac 框架來解決 controller 建構式注入的問題。從這些範例可以看得出來，許多瑣碎的細節都可以由現成的 DI 框架代勞，確實能夠讓開發人員輕鬆一些。下一章會繼續介紹其他 DI 應用範例，而且這些範例也都會使用現成的 DI 框架。

第 6 章：更多 DI 實作範例

前兩章用了不少篇幅探討 MVC 分層式架構以及 ASP.NET Web API 的管線處理流程，所占分量，甚至超過了主角 DI 的戲份。本章將提供更多 DI 與 .NET 應用程式的實作範例，包括 ASP.NET MVC、Web Forms、WCF（Windows Commnication Foundation）等等。

與先前各章不同的是，本章偏重實作練習，而較少著墨在基礎觀念和底層框架背後的運作機制；此外，範例程式將直接使用現成的 DI 框架——Unity 或 Autofac，而不再示範純手工 DI 的寫法，以免過於冗贅（先前各章已談得夠多了）。

內容大綱：

本章範例的專案原始碼位置：

https://github.com/huanlin/di-book-support 裡面的 Examples/ch06 資料夾。Ch06.sln 即包含了本章全部的範例專案。

共用程式碼

為了盡量減少重複的程式碼和文字敘述，這裡先列出往後各節範例所共用的程式碼。假想的情境是應用程式需要一個訊息服務，而這個訊息服務只提供一項功能：傳回一個簡單的「Hello World」字串。姑且將此訊息服務的介面命名為 `IMesssageService`，其定義如下：

```
public interface IMessageService
{
    string Hello(string name);
}
```

其實作類別則命名為 `MessageService`，程式碼如下：

```
public class MessageService : IMessageService
{
    public string Hello(string name)
    {
        return "Hello, " + name;
    }
}
```

上述程式碼都放在一個名為 `Examples.Common` 的類別庫專案中，隸屬於命名空間 `Examples.Common.Services`。在實作後續各節的範例時，請記得加入此專案的組件參考。

DI 與 ASP.NET MVC 5

如第 4 章提過的，當我們想要透過 controller 類別的建構函式來注入相依物件時，由於 ASP.NET MVC 框架預設會使用不帶任何參數的建構函式，因而會在程式執行時出錯：

System.MissingMethodException: 這個物件沒有定義無參數的建構函式。

此問題的解法，在第 4 章已經介紹過純手工 DI 的方式，並且示範了兩種作法：實作 `IControllerFactory` 或者 `IDenepdencyResolver` 介面來取代 ASP.NET MVC 的預設元件。這裡的範例會直接使用現成的 DI 框架來解決此問題。

練習：使用 Unity

接下來的實作步驟會示範如何運用 ASP.NET MVC 的 `IDependencyResolver` 介面所提供的擴充機制，並搭配 Unity 容器來將物件注入至 controller 類別的建構函式。

本練習的原始碼專案名稱為 Mvc5Demo.Unity.csproj。

Step 1：建立新專案

1. 建立一個新的 ASP.NET Web Application 專案，目標平台選擇 .NET Framework 4.5.2，專案名稱命名為：**Mvc5Demo.Unity**。
2. 專案範本選擇【Empty】，並且在【Add folder and core references for】項目上勾選【MVC】。
3. 專案建立完成後，透過 NuGet 套件管理員加入 Unity.Mvc 套件。為專案加入此套件之後，會連同相關的 Unity 套件一併加入，包括：
   - Unity
   - Unity.Container
   - Unity.Abstractions
   - Unity.Mvc

同時，App_Start 資料夾底下會多出兩個檔案：UnityConfig.cs 和 UnityMvcActivator.cs。後者暫時用不到，可以先不去看它。UnityConfig.cs 裡面則需要我們加入一些程式碼來設定 Unity 容器。

Step 2：設定 Unity 容器

開啟 App_Start 資料夾下的 UnityConfig.cs，然後在 `RegisterTypes` 函式中加入你的註冊型別的程式碼。以下是此類別的完整原始碼（註解原本是英文）：

```
using System;
using Unity;
using Examples.Common.Services;

namespace Mvc5Demo.Unity.App_Start
{
    /// <summary>
    /// 用來設定主要的 Unity 容器。
    /// </summary>
    public class UnityConfig
    {
        private static Lazy<IUnityContainer> container =
            new Lazy<IUnityContainer>(() =>
            {
                var container = new UnityContainer();
                RegisterTypes(container);
                return container;
            });

        /// <summary>
        /// 取得 Unity 容器的執行個體。
        /// </summary>
        public static IUnityContainer Container => container.Value;

        /// <summary> 向 Unity 容器註冊型別對應。</summary>
        /// <param name="container"> 欲設定的 Unity 容器。</param>
        /// <remarks>
        ///   除非你想要改變預設行為，否則不需要自行註冊具象類別，例如 controllers 或
        ///   API controllers；因為無論是否預先註冊具象類別，Unity 都能自動解析它們。
        /// </remarks>
        public static void RegisterTypes(IUnityContainer container)
        {
            // 如欲透過 web.config 來設定 Unity 容器，可恢復底下這行註解掉的程式碼。
            // 別忘了還要引用命名空間 Unity.Configuration。
            // container.LoadConfiguration();
```

```
            // 在此註冊你的型別。
            container.RegisterType<IMessageService, MessageService>();
        }
    }
}
```

程式說明：

- 註冊型別的程式碼是寫在 `RegisterTypes` 方法中。
- 靜態方法 `UnityConfig.GetConfiguredContainer()` 能夠傳回一個靜態 Unity 容器物件，而且這裡還使用了 .NET 4.0 之後才有的 `Lazy<T>` 泛型類別來提供延遲初始化的機制。簡單地說，就是當用戶端程式真的需要存取此類別的 `Value` 屬性時，才會去執行初始化物件的動作。如果你的 DI 容器的初始設定工作比較費時，那麼此延遲初始化的寫法便有些用處，因為它可以減少 ASP.NET 應用程式第一次啟動所需要花費的時間。

在一個 HTTP 請求的範圍內共享同一個物件

順便提及，第 4 章的 MVC 分層架構範例曾示範如何在一個 HTTP 請求的範圍內共享同一個 `DbContext` 物件的作法。這個部分也同樣可以利用 Unity 容器達成。以剛才的範例來說，就是在 `UnityConfig` 類別的 `RegisterTypes` 方法中註冊你的 `DbContext` 類別，像這樣：

```
container.RegisterType<MyDbContext>(new PerRequestLifetimeManager());
```

其實，預設情況下，具象類別是不用預先註冊的（Unity 能夠自動解析具象類別）。這裡之所以註冊具象類別 `SouthwindContext`，只是為了使用特殊的生命週期管理員：`PerRequestLifetimeManager`。其效果為：無論呼叫 Unity 容器的 `Resolve` 方法幾次，只要目前仍屬於同一個 HTTP 請求的範圍內，容器都會傳回同一個 `SouthwindContext` 物件，而且該物件會在 HTTP request 結束時釋放。

接著，修改 App_Start 資料夾的 UnityMvcActivator 類別的 Start 方法，找到以下程式碼：

```
// TODO: Uncomment if you want to use PerRequestLifetimeManager
// Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility
      .RegisterModule(typeof(UnityPerRequestHttpModule));
```

然後把最後一行程式碼的註解取消。若漏掉此步驟，PerRequestLifetimeManager 不會產生預期的作用。

如此一來，第 4 章範例中的 CustomerService 類別只要在建構函式的參數列加入 SouthwindContext 參數，如下所示：

```
public class CustomerService : ICustomerService
{
    private readonly SouthwindContext db;

    public CustomerService(SouthwindContext dbContext)
    {
        this.db = dbContext;
    }
}
```

等到你在程式中解析 ICustomerService 時，像這樣：

```
var svc = UnityConfig.GetConfiguredContainer().Resolve<ICustomerService>();
```

容器便會自動連帶解析 SouthwindContext，而且在當前的 HTTP 請求範圍內，即使呼叫多次 Resolve<SouthwindContext>()，得到的物件都會是同一個。

Step 3：建立 Controller

在專案的 Controller 資料夾下加入一個新的 controller 類別：HomeController。程式碼如下：

```
public class HomeController : Controller
{
    private readonly IMessageService _messageService;

    public HomeController(IMessageService msgSvc)
    {
        _messageService = msgSvc;
    }

    public ActionResult Index()
    {
        return Content(_messageService.Hello("MVC5 with Unity!"));
    }
}
```

這樣就行了。現在執行看看，瀏覽器應該能順利顯示字串：「Hello, MVC5 with Unity!」

如果你曾試過不靠任何 DI 容器，完全以純手工的方式注入相依物件至 controller 的建構函式（例如第 4 章的範例），就會發現「Unity.Mvc」套件的確能夠幫我們省一些工夫。

查看步驟 1 的 UnityMvcActivator.cs 程式碼，你會發現此套件是透過自訂的 `UnityDependencyResolver` 類別來取代 ASP.NET MVC 框架的 `IDependencyResolver` 的預設實作類別，並藉此改變建立 controller 的過程。

DI 與 ASP.NET Web Forms

跟 ASP.NET MVC 與 Web API 比起來，在 Web Forms 應用程式中使用 DI 會比較麻煩些。畢竟是比較早期發展的框架，在 DI 方面的支援自然沒有 MVC 與 Web API 那麼友善與彈性。不過，透過撰寫自訂的 HTTP 處理常式（handler），還是有辦法解決的。

本節會用一個範例來說明如何注入外部元件至 Web Forms 的 ASPX 頁面。這一次，注入方式不再是「建構式注入」（**Constructor Injection**），而是「屬性注入」（**Property Injection**）。

問題描述

基於測試或其他原因，希望 ASPX 網頁都只依賴特定服務的介面，而不要依賴具象類別。比如說，假設首頁 Default.aspx 需要利用 `MessageService` 來取得一個訊息字串，該網頁的 code-behind 類別大概會像這樣：

```
public partial class Default : System.Web.UI.Page
{
    private IMessageService _msgService;

    protected void Page_Load(object sender, EventArgs e)
    {
        // 在網頁上輸出一段字串訊息。訊息內容由 MessageService 提供。
        Response.Write(_msgService.Hello("DI in ASP.NET Web Forms!"));
    }
}
```

問題來了：Web Forms 的 `Page` 物件是由底層框架所建立，而不是我們自己 `new` 出來的，我們如何從外界動態注入 `IMessageService` 物件呢？

解法

一般的建議是儘量採用「建構式注入」（**Constructor Injection**）來注入相依物件，可是此法很難用在 Web Forms 的 **Page** 物件上。一個便宜行事的解法是採用 Mark Seemann 所說的「私生注入」（**Bastard Injection**），像這樣：

```
public partial class Default : System.Web.UI.Page
{
    public IMessageService MessageService { get; set; }

    public Default()
    {
        // 透過一個共用的 Container 物件來解析相依物件。
        MessageService = AppShared.Container.Resolve<IMessageService>();
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        // 在網頁上輸出一段字串訊息。訊息內容由 MessageService 提供。
        Response.Write(MessageService.Hello("DI in ASP.NET Web Forms!"));
    }
}
```

也就是說，解析相依物件的工作由每一個 Web Form 頁面自行處理。

此解法的一個問題是，你必須在每一個 ASPX 頁面的 code-behind 類別中引用 DI 框架（例如 Unity）的命名空間，而這樣就變成到處都依賴特定的 DI 容器了。我們希望盡可能把呼叫 DI 容器的程式碼集中寫在少數幾個地方就好。

因此，這裡要示範的解決方法，是採用「屬性注入」（**Property Injection**）的方式。我們會撰寫一個自訂的 `PageHandlerFactory` 來介入 Web Forms 的頁面創建過程，從中動點手腳，以便將 `Page` 物件所需的相依元件給注入進去。

`PageHandlerFactory` 類別是 ASP.NET Web Forms 的預設 HTTP handler 工廠，它負責建立那些繼承自 `Page` 類別並實作了 `IHttpHandler` 介面之類別的執行個體。

練習：使用 Unity

接下來的實作步驟會示範如何撰寫自訂的 `PageHandlerFactory` 來攔截 `Page` 物件的建立程序，以便在 `Page` 物件建立完成後，緊接著以 **Property Injection** 的方式將 `Page` 物件需要的服務給注入進去。相依物件的型別註冊與解析工作都是透過 Unity 容器來處理。

本練習的原始碼專案為 WebFormsDemo.Unity.csproj。

Step 1：建立新專案

1. 建立一個新的 ASP.NET Web Application 專案，目標平台選擇 .NET Framework 4.5，專案名稱命名為：**WebFormsDemo.Unity**。
2. 專案範本選擇【Empty】，然後在【Add folder and core references for】項目上勾選【Web Forms】。
3. 專案建立完成後，透過 NuGet 管理員加入 Unity 套件。

Step 2：註冊型別

在應用程式的「組合根」建立 DI 容器並註冊相依型別。這裡選擇在 Global_asax.cs 的 Application_Start 方法中處理這件事：

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        var container = new UnityContainer();
        Application["Container"] = container; // 把容器物件保存在共用變數裡

        // 註冊型別
        container.RegisterType<IMessageService, MessageService>();
    }
}
```

Step 3：撰寫 HTTP Handler

在專案根目錄下建立一個子目錄：Infrastructure，然後在此目錄中加入一個新類別：UnityPageHandlerFactory.cs。程式碼：

```
public class UnityPageHandlerFactory : System.Web.UI.PageHandlerFactory
{
    public override IHttpHandler GetHandler(
        HttpContext context, string requestType, string virtualPath, string path)
    {
        Page page = base.GetHandler(
            context, requestType, virtualPath, path) as Page;
        if (page == null)
            return null;

        var container = context.Application["Container"] as IUnityContainer;
        var properties = GetInjectableProperties(page.GetType());

        foreach (var prop in properties)
        {
            try
            {
                var service = container.Resolve(prop.PropertyType);
                if (service != null)
                {
                    prop.SetValue(page, service);
                }
            }
            catch
            {
                // 沒辦法解析型別就算了。
            }
        }
        return page;
    }

    public static PropertyInfo[] GetInjectableProperties(Type type)
    {
        var propFlags = BindingFlags.Public | BindingFlags.Instance |
                        BindingFlags.DeclaredOnly;
        var props = type.GetProperties(propFlags);
        if (props.Length == 0)
        {
            // 傳入的型別若是由 ASPX 頁面所生成的類別，
            // 那就必須取得其父類別（code-behind 類別）的屬性。
            props = type.BaseType.GetProperties(propFlags);
        }
        return props;
```

```
    }
}
```

程式說明：

- ASP.NET Web Forms 框架會呼叫此 HTTP handler 的 `GetHandler` 方法來建立 `Page` 物件。
- 在 `GetHandler` 方法中，先利用父類別來建立 `Page` 物件，然後緊接著進行 **Property Injection** 的處理。首先，從 `Application["Container"]` 中取出上一個步驟所建立的 DI 容器，接著找出目前的 `Page` 物件有宣告哪些公開屬性，然後利用 DI 容器來逐一解析各屬性的型別，並將建立的物件指派給屬性。
- 靜態方法 `GetInjectableProperties` 會找出指定型別所宣告的所有公開屬性，並傳回呼叫端。注意這裡只針對「`Page` 類別本身所宣告的公開屬性」來進行 **Property Injection**，這樣就不用花時間在處理由父類別繼承而來的數十個公開屬性。

Step 4：註冊 HTTP Handler

在 web.config 中註冊剛才寫好的 HTTP handler：

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>

  <system.webServer>
    <handlers>
       <add name="UnityPageHandlerFactory" path="*.aspx" verb="*"
            type="WebFormsDemo.Unity.Infrastructure.UnityPageHandlerFactory"/>
    </handlers>
  </system.webServer>
</configuration>
```

基礎建設的部分到此步驟已經完成，接著就是撰寫各個 ASPX 頁面。

Step 5：撰寫測試頁面

在專案中加入一個新的 Web Form，命名為 Default.aspx。然後在 code-behind 類別中宣告相依服務的屬性，並且在其他地方呼叫該服務的方法。參考以下範例：

```
public partial class Default : System.Web.UI.Page
{
    public IMessageService MessageService { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        // 在網頁上輸出一段字串訊息。訊息內容由 MessageService 提供。
        Response.Write(MessageService.Hello("DI in ASP.NET Web Forms!"));
    }
}
```

你可以看到，各 Web Form 頁面並不需要引用 Unity 容器，而只需要把相依的物件宣告成公開屬性而已。注入相依物件的動作都由我們的自訂 HTTP handler 代勞了。

練習：使用 Unity 的 BuildUp 方法

與「範例一」的作法雷同，唯一的差別是這次使用 Unity 容器內建的 `BuildUp` 方法來幫我們處理 **Property Injection** 的工作。換言之，上一個範例的靜態方法 `GetInjectableProperties` 可以拿掉。

本練習的原始碼專案為 WebFormsDemo.UnityBuildUp.csproj。

直接用上一個範例來修改，首先修改 `UnityPageHandlerFactory` 類別：

```
public class UnityPageHandlerFactory : System.Web.UI.PageHandlerFactory
{
    public override IHttpHandler GetHandler(
        HttpContext context, string requestType, string virtualPath, string path)
    {
        Page page = base.GetHandler(
            context, requestType, virtualPath, path) as Page;
        if (page == null)
        {
            return null;
        }

        var container = context.Application["Container"] as IUnityContainer;
        container.BuildUp(page.GetType(), page);

        return page;
    }
```

跟先前的版本比較，程式碼顯然精簡多了。但還沒完呢，我們還得在 ASPX 類別裡面的屬性上面套用 Unity 的 `DependencyAttribute`，如下所示：

```
public partial class Default : System.Web.UI.Page
{
    [Dependency]   // <---- 加這行
    public IMessageService MessageService { get; set; }

    // 其餘程式碼不變，故不列出。
}
```

如此一來，Unity 的 `BuildUp` 方法就會針對有套用 `DependencyAttribute` 的屬性來解析相依物件。此法挺方便，唯一的缺點是，你的各個 ASPX 類別都得要引用 Unity 組件的命名空間，那麼將來如果有一天要改成其他 DI 容器，就比較麻煩了。

練習：使用 Autofac

Autofac 核心套件已經有提供對 Web Forms 網頁進行「屬性注入」的功能，相當方便。本練習將示範其用法。

本練習的原始碼專案為 WebFormsDemo.Autofac.csproj。

Step 1：建立新專案

1. 建立一個新的 ASP.NET Web Application 專案，目標平台選擇 .NET Framework 4.5，專案名稱命名為：**WebFormsDemo.Autofac**。
2. 專案範本選擇【Empty】，然後在【Add folder and core references for】項目上勾選【Web Forms】。
3. 專案建立完成後，透過 NuGet 管理員加入 Autofac 套件。

Step 2：註冊型別

在 Global_asax.cs 的 Application_Start 方法中建立容器並註冊型別：

```
public class Global : System.Web.HttpApplication
{
    protected void Application_Start(object sender, EventArgs e)
    {
        ContainerBuilder builder = new ContainerBuilder();
        builder.RegisterType<MessageService>().As<IMessageService>();
        Application["Container"] = builder.Build(); // 把容器物件保存在共用變數裡
    }
}
```

Step 3：撰寫 HTTP Handler

在專案根目錄下建立一個子目錄：Infrastructure，然後在此目錄中加入一個新類別：AutofacPageHandlerFactory.cs。程式碼如下：

```
public class AutofacPageHandlerFactory : System.Web.UI.PageHandlerFactory
{
    public override IHttpHandler GetHandler(
        HttpContext context, string requestType, string virtualPath, string path)
    {
        Page page = base.GetHandler(
            context, requestType, virtualPath, path) as Page;

        if (page == null)
        {
            return null;
        }

        IContainer container = context.Application["Container"] as IContainer;
        container.InjectProperties(page);

        return page;
    }
}
```

其運作原理與先前的 Unity 範例中的 `UnityPageHandlerFactory` 相同，主要差異是 Autofac 提供了 `InjectProperties` 方法，只要一行程式碼就能為指定的網頁完成「屬性注入」的工作。

Step 4：註冊 HTTP Handler

在 web.config 中註冊剛才寫好的 HTTP handler：

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="AutofacPageHandlerFactory" path="*.aspx" verb="*"
           type="WebFormsDemo.Autofac.Infrastructure.AutofacPageHandlerFactory"/>
    </handlers>
  </system.webServer>
</configuration>
```

基礎建設的部分到此步驟已經完成，接著就是撰寫各個 ASPX 頁面。

Step 5：撰寫測試頁面

在專案中加入一個新的 Web Form，命名為 Default.aspx。然後在 code-behind 類別中宣告相依服務的屬性，並且在其他地方呼叫該服務的方法。參考以下範例：

```
public partial class Default : System.Web.UI.Page
{
    // 此屬性會由 Autofac 負責注入相依物件。
    public IMessageService MessageService { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        // 在網頁上輸出一段字串訊息。訊息內容由 MessageService 提供。
        var msg = MessageService.Hello("DI in ASP.NET Web Forms using Autofac!");
        Response.Write(msg);
    }
}
```

你可以看到，Web Form 頁面並不需要引用 Autofac 容器，而只需要把相依物件宣告成屬性而已。注入相依物件的動作都由 Autofac 代勞了。

DI 與 WCF

如同 ASP.NET MVC 與 Web API 的 controller 類別，WCF（Windows Communication Foundation）的 service 類別也是由底層框架呼叫類別的預設建構函式來創建執行個體

的。因此，若要注入相依物件至 WCF 服務類別的建構函式，也同樣得費一番工夫。本節將以一個範例來說明如何解決這個問題。

問題描述

假設有個 WCF service 類別，名為 `Service1`，此服務需要使用 `MessageService` 來輸出一段訊息，程式碼如下：

```
public class Service1 : IService1
{
    private IMessageService _messageService;

    public Service1()
    {
        _messageService = new MessageService();
    }

    public string Hello()
    {
        return _messageService.Hello("DI in WCF service.");
    }
}
```

`MessageService` 的介面與實作與上一節相同，這裡就不重複。

現在，我們希望讓 `IMessageService` 物件由外界注入，且注入的管道是透過此服務類別的建構函式，像這樣：

```
public class Service1 : IService1
{
    private readonly IMessageService _messageService;

    public Service1(IMessageService msgService)
    {
        _messageService = msgService;
    }
    // 其餘省略。
}
```

改成這樣之後，WCF 服務啟動時會出現錯誤：

> System.InvalidOperationException: 由於服務類型沒有預設 (無參數) 建構函式，因此無法將提供的服務類型載入成為服務。若要修正此問題，請將預設建構函式新增至類型，或將類型例項傳至主機。

解法

透過 WCF 的擴充機制，我們可以寫一些程式碼，從中介入底層框架創建 WCF 服務的過程，以實現 **Constructor Injection**。與此問題有關的 WCF 擴充機制包括：

- 撰寫自訂的 `ServiceHostFactory`（命名空間：`System.ServiceModel.Activation`）
- 撰寫自訂的 `ServiceHost` 類別（命名空間：`System.ServiceModel`）
- 實作 `IInstanceProvider` （命名空間：`System.ServiceModel.Dispatcher`）
- 實作 `IContractBehavior` （命名空間：`System.ServiceModel.Description`）

它們之間的合作關係是：

- `ServiceHostFactory` 會建立 `ServiceHost` 物件；
- `ServiceHost` 會將自訂的 `IContractBehavior` 附加至特定 WCF service 的合約清單裡；

- 自訂的 IContractBehavior 物件會改寫執行時期分派器（DispatchRuntime) 的 InstanceProvider 屬性，把它設定成我們的自訂 IInstanceProvider 物件‘，以便取代預設的「創建服務執行個體」的行為。
- 實作自訂的 IInstanceProvider 類別時，在它的 GetInstance 方法中解析服務類型，並注入相依物件。

練習：使用 Unity

以下實作範例是搭配 Unity 容器來解決前述問題，分別需要實作服務端和用戶端應用程式。

本練習的原始碼專案為 WcfDemo.ServerApp.Unity.csproj 和 WcfDemo.ClientApp.csproj。

Step 1：建立 WCF 服務

1. 建立一個新的 ASP.NET Web Application 專案，目標平台選擇 .NET Framework 4.5，專案名稱命名為：**WebDemo.ServerApp.Unity**。專案範本選擇【Empty】，其餘額外選項（例如 MVC、Web API）都不勾選。
2. 專案建立完成後，手動加入 .NET Framework 的 System.ServiceModel.Activation.dll 組件參考，然後利用 NuGet 管理員加入 Unity 套件。
3. 在專案根目錄下加入一個新的 WCF Service，命名為 **Service1.svc**。

參考以下程式片段來修改 Service1.svc 的 code-behind 類別：

```
public class Service1 : IService1
{
    private IMessageService _messageService;

    public Service1(IMessageService msgService)
    {
        _messageService = msgService;
    }

    public string Hello()
    {
        return _messageService.Hello("DI in WCF service using Unity.");
    }
}
```

其中的 IMessageService 是沿用上一節範例的程式碼，這裡就不重複列出。

接著要實作 WCF 擴充機制的相關類別。我把這些類別全放在專案的 Infrastructure 子目錄下。

Step 2：撰寫自訂的 ServiceHostFactory

1. 在專案的根目錄下建立一個新的資料夾：Infrastructure。
2. 在 Infrastructure 資料夾中加入一個新類別：MyServiceHostFactory。

MyServiceHostFactory 類別只是單純改寫父類別 ServiceHostFactory 的 CreateService-Host 方法，並傳回自訂的 ServiceHost 物件，亦即接下來要實作的 MyServiceHost。

```
public class MyServiceHostFactory : ServiceHostFactory
{
    protected override System.ServiceModel.ServiceHost CreateServiceHost(
        Type serviceType, Uri[] baseAddresses)
    {
        return new MyServiceHost(serviceType, baseAddresses);
    }
}
```

Step 3：撰寫自訂的 ServiceHost

在 Infrastructure 資料夾中加入一個新類別：MyServiceHost，讓此類別繼承自 ServiceHost。程式碼如下：

```
public class MyServiceHost : ServiceHost
{

    public MyServiceHost(Type serviceType, params Uri[] baseAddresses)
        : base(serviceType, baseAddresses)
    {
        var contracts = this.ImplementedContracts.Values;
        foreach (var contract in contracts)
        {
            var behavior = new MyContractBehavior();
            contract.ContractBehaviors.Add(behavior);
        }
    }
}
```

MyServiceHost 類別改寫了父類別的建構函式，並在其中對傳入的服務類型動手腳：附加自訂的合約行為至其合約清單。下一個步驟就要實作這個自訂的合約行為。

Step 4：實作 IContractBehavior 介面

在 Infrastructure 資料夾中加入一個新類別：MyContractBehavior。此類別要實作 IContractBehavior 介面，如下所示：

```
public class MyContractBehavior : IContractBehavior
{
    public void AddBindingParameters(ContractDescription contractDescription,
        ServiceEndpoint endpoint, BindingParameterCollection bindingParameters)
    {
    }

    public void ApplyClientBehavior(ContractDescription contractDescription,
        ServiceEndpoint endpoint, ClientRuntime clientRuntime)
    {
    }

    public void ApplyDispatchBehavior(ContractDescription contractDescription,
        ServiceEndpoint endpoint, DispatchRuntime dispatchRuntime)
    {
        dispatchRuntime.InstanceProvider =
            new MyInstanceProvider(contractDescription.ContractType);
    }

    public void Validate(ContractDescription contractDescription,
        ServiceEndpoint endpoint)
    {
    }
}
```

一共有四個方法，但我們真正需要寫程式的地方的是 `ApplyDispatchBehavior` 方法。在此方法中，我們將執行時期分派器的 `InstanceProvider` 屬性指派為自訂的 `MyInstanceProvider` 物件。

Step 5：實作 IInstanceProvider 介面

在 `Infrastructure` 資料夾中加入一個新類別：`MyInstanceProvider`。此類別要實作 `IInstanceProvider` 介面，如下所示：

```
public class MyInstanceProvider : IInstanceProvider
{
    private Type _contractType;

    public MyInstanceProvider(Type contractType)
    {
        _contractType = contractType; // 記住目前要創建的服務類型。
    }

    public object GetInstance(InstanceContext instanceContext,
        System.ServiceModel.Channels.Message message)
    {
        // 呼叫自己的另一個多載方法。
        return this.GetInstance(instanceContext);
    }

    public object GetInstance(InstanceContext instanceContext)
    {
        try
        {
            // Note: Unity 容器在解析 IService1 時會一併解析 IMessageService。
            return App.UnityContainer.Resolve(_contractType);
        }
        catch
        {
            // 若無法解析服務，則使用 Reflection 機制來建立物件。
            return Activator.CreateInstance(_contractType);
        }
    }

    public void ReleaseInstance(InstanceContext instanceContext, object instance)
    {
        // 釋放資源。
        if (instance is IDisposable)
        {
            (instance as IDisposable).Dispose();
        }
    }
}
```

其中的 GetInstance 方法透過另一個靜態類別 App 的 UnityContainer 屬性來取得全域的

Unity 容器，並呼叫容器的 Resolve 方法來解析當前的服務型別，然後將容器所建立的服務執行個體傳回上一層呼叫端。

Step 6：設定 Unity 容器

設定 Unity 容器的動作都放在一個名為 App 的靜態類別裡。此類別也是放在 Infrastructure 資料夾下。程式碼如下：

```
public static class App
{
    private static Lazy<IUnityContainer> _container =
        new Lazy<IUnityContainer>(() =>
        {
            var container = new UnityContainer();
            ConfigUnity(container);
            return container;
        });

    private static void ConfigUnity(IUnityContainer container)
    {
        container.RegisterType<IMessageService, MessageService>();
        container.RegisterType<IService1, Service1>();
    }

    public static IUnityContainer UnityContainer
    {
        get
        {
            return _container.Value;
        }
    }
}
```

這裡使用了 Lazy<T> 來提供延遲初始化的效果，亦即 .NET CLR 載入靜態類別 App 時並不會立刻建立這個「懶惰物件」，而是當外界第一次存取 App.UnityContainer 屬性時才執行建立容器以及註冊型別的動作。

Step 7：修改 Web.config

開啟 web.config，在 <system.serviceModel> 的 <serviceHostingEnvironment> 區段中加入 <serviceActivations>，如下所示：

```
<serviceHostingEnvironment aspNetCompatibilityEnabled="true"
                           multipleSiteBindingsEnabled="true" >
  <serviceActivations>
    <add service="WcfDemo.ServerApp.Unity.Service1"
         relativeAddress="./Service1.svc"
         factory="WcfDemo.ServerApp.Unity.Infrastructure.MyServiceHostFactory" />
  </serviceActivations>
</serviceHostingEnvironment>
```

其中關鍵在於利用 <serviceActivations> 區段來告訴 WCF 框架：請使用我提供的 MyServiceHostFactory 來取代預設的 ServiceHostFactory。

除了在 web.config 中註冊自訂的 ServiceHostFactory 類別，還有一種方法是直接修改個別 WCF service 的 .svc 檔案，將自訂的 factory 類別設定給 Factory 屬性。像這樣：

```
<%@ ServiceHost Language="C#" Debug="true"
    Service="WcfDemo.ServerApp.Unity.Service1"
    CodeBehind="Service1.svc.cs"
    Factory="WcfDemo.ServerApp.Unity.Infrastructure.MyServiceHostFactory"
%>
```

WCF 伺服器端應用程式至此大功告成。接著只剩下撰寫簡單的用戶端程式來測試此服務。

Step 8：撰寫用戶端程式

1. 在目前的方案中加入一個新的 Console 應用程式專案，目標平台選擇 .NET Framework 4.5，專案名稱命名為：WebDemo.ClientApp。

2. 加入 WCF 服務參考：在 Solution Explorer 視窗裡，此專案名稱底下的 References 項目上點一下滑鼠右鍵，選【Add Service Reference...】，然後在新開啟的「Add Service Reference」對話窗裡點【Discover】按鈕，即可發現剛才實作的 `Service1`。在此對話窗的【Namespace】欄位中輸入「**WcfDemoService_Unity**」，然後點 OK 鈕。
3. 開啟 Program.cs，撰寫 `Main` 方法。如下所示：

```
static void Main(string[] args)
{
    var serviceProxy = new WcfDemoService_Unity.Service1Client();
    Console.WriteLine(serviceProxy.Hello());

    Console.WriteLine(" 程式執行完畢，按 Enter 鍵結束。");
    Console.ReadLine();
}
```

4.以除錯模式執行此程式（按【F5】），應該會看到螢幕上輸出一段訊息：

```
Hello, DI in WCF service using Unity.
程式執行完畢，按 Enter 鍵結束。
```

如果執行此用戶端程式時出現如下錯誤訊息：

System.ServiceModel.ServiceActivationException: 無法啟動要求的服務 'http://[主機名稱]: 埠號/Service1.svc'。如需詳細資訊，請參閱伺服器的診斷追蹤記錄檔。

那表示先前實作的 WCF 服務無法順利啟動，請回頭檢查每一個步驟，看看是否遺漏了什麼。或者，你可以將以下 XML 內容貼到 WcfDemo.ServerApp.Unity 專案的 web.config 的 `<configuration>` 區段裡，以便將伺服器端的 WCF 診斷追蹤訊息輸出至 `Error.svclog` 檔案。

```
<!-- 底下這段配置是用來啟動診斷追蹤紀錄，若執行時發生伺服器端錯誤，
    可查看應用程式所在目錄的 Error.svclog 檔案內容 -->
<system.diagnostics>
 <sources>
   <source name="System.ServiceModel"
```

```
            switchValue="Information, ActivityTracing"
            propagateActivity="true" >
        <listeners>
          <add name="xml"/>
        </listeners>
      </source>
      <source name="System.ServiceModel.MessageLogging">
        <listeners>
          <add name="xml"/>
        </listeners>
      </source>
      <source name="myUserTraceSource"
            switchValue="Information, ActivityTracing">
        <listeners>
          <add name="xml"/>
        </listeners>
      </source>
    </sources>
    <sharedListeners>
      <add name="xml"
           type="System.Diagnostics.XmlWriterTraceListener"
           initializeData="Error.svclog" />
    </sharedListeners>
  </system.diagnostics>
```

將上述 XML 加入 web.config 以後，再次以除錯模式執行 `WcfDemo.ClientApp`，然後到 `WcfDemo.ServerApp` 所在目錄下查看 `Error.svclog` 檔案的內容，以獲得詳細的錯誤訊息。

檢視 WCF 診斷追蹤紀錄檔時，你會需要檢視工具：SvcTraceViewer.exe。若以滑鼠雙擊 .svclog 檔案時沒有直接啟動此工具，那麼你可能得自行安裝此工具。此工具附在 Windows SDK 裡，請搜尋關鍵字「Windows SDK download」即可找到下載頁面（安裝時只需勾選「.NET Development」底下的全部元件即可）。

如果你覺得寫那麼多程式碼實在太麻煩了，也可以試試第三方套件 **Unity3.Wcf**，網址是 https://www.nuget.org/packages/Unity3.Wcf/。使用此套件至少可省下撰寫三個類別的工夫，即 `MyServiceHost`、`MyContractBehavior`、以及 `MyInstanceProvider`。同樣地，Autofac 也有類似功能的套件，叫做 **Autofac.Wcf**。下一個練習就來看看怎麼使用此套

件來達到相同目的。

練習：使用 Autofac.Wcf 套件

此練習的步驟與前一個練習類似，只是省了不少工夫。

本練習的原始碼專案為 WcfDemo.ServerApp.Autofac.csproj 和 WcfDemo.ClientApp.csproj。

Step 1：建立 WCF 服務

1. 建立一個新的 ASP.NET Web Application 專案，目標平台選擇 .NET Framework 4.5，專案名稱命名為：WebDemo.ServerApp.**Autofac**。專案範本選擇「Empty」，其餘額外選項（例如 MVC、Web API）都不勾選。
2. 專案建立完成後，手動加入 .NET Framework 的 `System.ServiceModel.Activation.dll` 組件參考，然後利用 NuGet 管理員加入 Autofac 核心套件以及提供 WCF 整合功能的 **Autofac.Wcf** 套件。
3. 在專案根目錄下加入一個新的 WCF Service，命名為 **Service1.svc**。

參考以下程式片段來修改 Service1.svc 的 code-behind 類別：

```
public class Service1 : IService1
{
    private IMessageService _messageService;

    public Service1(IMessageService msgService)
    {
        _messageService = msgService;
    }

    public string Hello()
    {
        return _messageService.Hello("DI in WCF service using Autofac.Wcf.");
    }
}
```

其中的 IMessageService 是沿用上一節範例的程式碼，這裡就不重複列出。

Step 2：撰寫自訂的 ServiceHostFactory

1. 在專案的根目錄下建立一個新的資料夾：Infrastructure。
2. 在 Infrastructure 資料夾中加入一個新類別：MyServiceHostFactory。

MyServiceHostFactory 類別只是單純改寫父類別 ServiceHostFactory 的 CreateServiceHost 方法，並利用 Autofac.Wcf 套件為 ServiceHost 所提供的擴充方法 AddDependencyInjectionBehavior 來幫我們處理「屬性注入」的瑣碎工作。程式碼如下：

```
public class MyServiceHostFactory : ServiceHostFactory
{
    protected override ServiceHost CreateServiceHost(
        Type serviceType, Uri[] baseAddresses)
    {
        var host = base.CreateServiceHost(serviceType, baseAddresses);

        // 呼叫 Autofac.Wcf 套件所提供的擴充方法。
        host.AddDependencyInjectionBehavior(serviceType, App.Container);

        return host;
    }
}
```

Step 3：設定 Autofac 容器

設定 Autofac 容器的動作都放在一個名為 App 的靜態類別裡。此類別也是放在 Infrastructure 資料夾下。程式碼如下：

```
public static class App
{
    static Lazy<IContainer> _container = new Lazy<IContainer>(ConfigAutofac);

    private static IContainer ConfigAutofac()
    {
        var builder = new ContainerBuilder();
        builder.RegisterType<MessageService>().As<IMessageService>();
        builder.RegisterType<Service1>();

        return builder.Build();
    }

    public static IContainer Container
    {
        get
        {
            return _container.Value;
        }
    }
}
```

程式寫法與前一個 Unity 版本的練習類似，就不再重複解釋了。

Step 4：修改 Web.config

開啟 web.config，在 `<system.serviceModel>` 的 `<serviceHostingEnvironment>` 區段中加入 `<serviceActivations>`，如下所示（已省略與上個範例重複的部分）：

```
<serviceHostingEnvironment aspNetCompatibilityEnabled="true"
                           multipleSiteBindingsEnabled="true" >
  <serviceActivations>
    <add service="WcfDemo.ServerApp.Autofac.Service1"
       relativeAddress="./Service1.svc"
       factory="WcfDemo.ServerApp.Autofac.Infrastructure.MyServiceHostFactory" />
  </serviceActivations>
</serviceHostingEnvironment>
```

四個步驟就完成伺服器端的 WCF 程式，比前一個範例程式省事多了。

Step 5：撰寫用戶端程式

直接沿用前一個範例的用戶端應用程式專案，只需要兩個動作便可完成：

1. 為專案加入服務參考。加入服務參考時，在新開啟的「Add Service Reference」對話窗裡點【Discover】按鈕，即可發現剛才實作的 `Service1`。在此對話窗的【Namespace】欄位中輸入「**WcfDemoService_Autofac**」，然後點 OK 鈕。
2. 開啟 Program.cs，修改 `Main` 方法。如下所示：

```
static void Main(string[] args)
{
    var serviceProxy = new WcfDemoService_Autofac.Service1Client();
    Console.WriteLine(serviceProxy.Hello());

    Console.WriteLine(" 程式執行完畢，按 Enter 鍵結束。");
    Console.ReadLine();
}
```

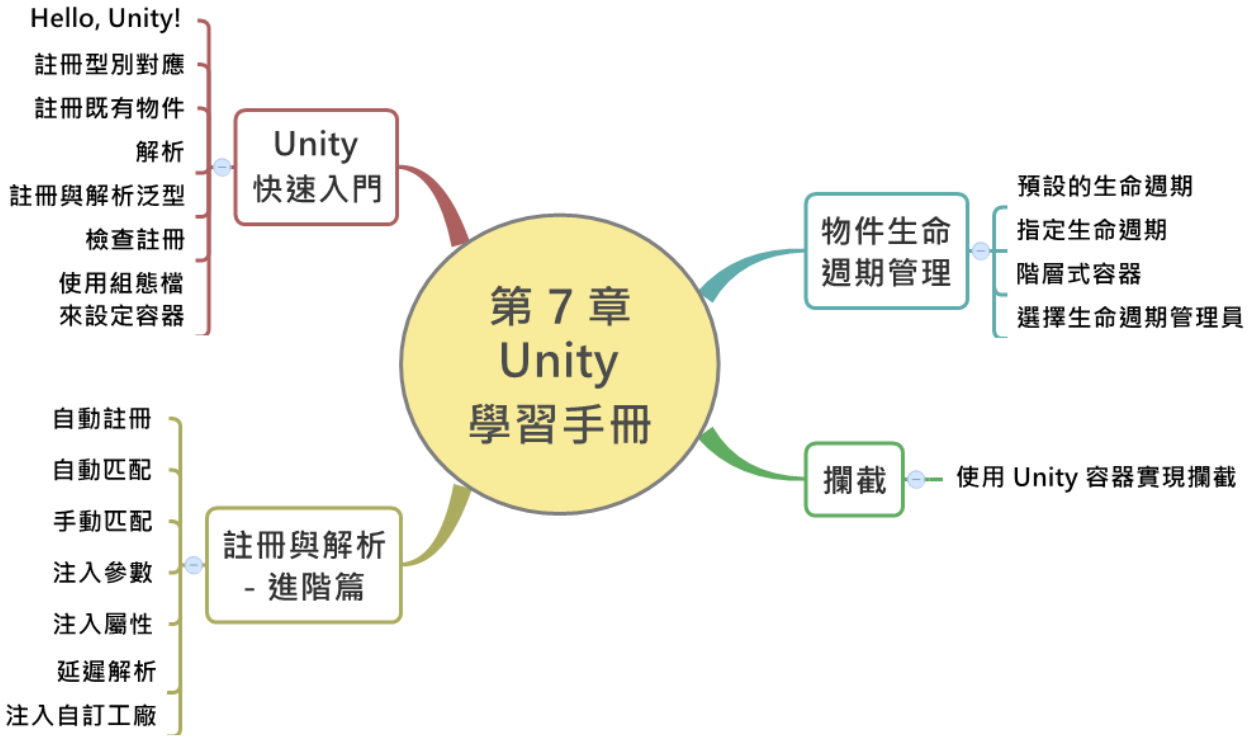
本章回顧

本章提供了更多 DI 範例與實作練習，這些範例大多以「解決特定 DI 問題」為出發點，如同坊間的食譜書（cookbook），從問題帶出解法，然後以逐步練習來示範如何以現成的 DI 框架來解決這些問題。本章範例所涉及的 .NET 應用類型包括：

- ASP.NET MVC 5 —使用「建構式注入」來將相依物件注入至 controller。使用的容器是 Unity。
- ASP.NET Web Forms —分別示範 Unity 和 Autofac 的作法。其中 Unity 的部分還分成「屬性注入」和 `BuildUp` 兩種寫法。
- WCF —分別以 Unity 和 Autofac 來示範如何將物件注入至 WCF 服務的建構函式。

Part III：工具篇

這是本書的第三個部分，主要是提供 DI 容器的入門教學與常見的應用技巧。目前僅提供 Unity 框架的學習手冊（第 7 章），將來也許會在改版時加入其他 DI 框架的用法介紹（例如 Autofac）。



由於本書範例程式皆以 Visual Studio 英文版來撰寫，故有關 IDE 操作畫面的截圖以及某些文字敘述會出現英文術語，例如 Solution Explorer、Console Application、NuGet Package Manager 等等。

第 7 章：Unity 學習手冊

本章將介紹 Unity 框架的一些常用寫法與技巧，包括型別註冊與解析的一些常見用法、自動註冊、延遲解析（deferred resolution）、自動工廠（automatic factories）、物件生命週期管理等等。

閱讀本章前，最好先讀過本書的前三章。或者，至少讀過第 1 章和第 2 章，以免閱讀時被一些陌生名詞或縮寫卡住。

這不是完整的 Unity API 參考手冊。本章大多採取碰撞式寫法，由一個議題帶出其他相關議題，以便依序閱讀。此外，由於筆者本身並不熱衷於編寫 XML 組態檔，常以程式碼為主，組態檔為輔，故寫作時亦偏重以程式碼來操作 Unity 容器，而對於組態檔的部分較少著墨。請讀者諒察。

如果您需要查閱完整詳細的 API，可以參考線上文件：Unity API Documentation[23] 。

本章範例程式所使用之 Unity 套件的版本是 v5.8.x。

內容大綱：

- Unity 快速入門
  - Hello, Unity!、註冊型別對應、註冊既有物件、解析、註冊與解析泛型、檢查註冊、使用組態檔來設定容器
- 註冊與解析－進階篇
  - 自動註冊、自動匹配、手動匹配、注入參數、注入屬性、延遲解析、注入自訂工廠

[23] https://unitycontainer.github.io/api/index.html

- 物件生命週期管理
  - 預設的生命週期、指定生命週期、階層式容器、選擇生命週期管理員
- 攔截
  - 使用 Unity 容器實現攔截

本章範例原始碼位置：

https://github.com/huanlin/di-book-support 裡面的 Examples/ch07 資料夾。Ch07.sln 即包含了本章全部的範例專案。共用程式碼的專案名稱是 Ch07.Common.csproj。

Unity 快速入門

Unity 是由微軟的 Patterns & Practices（P&P）小組所開發之眾多 application blocks 的其中一塊。所謂的 application block，其實就是可重複使用的類別庫，其中提供了應用程式基礎建設的常用功能，例如日誌紀錄、異常處理等等。Unity 的角色則是提供一般 DI 容器常見的功能，協助開發人員撰寫寬鬆耦合的 .NET 應用程式。

Unity 和 Enterprise Library 雖然系出同門，而且 Enterprise Library 套件中有包含 Unity，但 Unity 其實也是個獨立類別庫，並不須要跟 Enterprise Library 一起使用。

Hello, Unity!

如果你還沒有在應用程式中用過 Unity，這裡有個簡單的入門練習，可以讓你在短時間內大致了解 Unity 的基本用法。本章只有這個範例會提供詳細的操作步驟，主要是希望儘量讓初次使用 Unity 容器的讀者能夠順利完成第一個實作練習。

讓我們開始吧！

本節範例的專案名稱是 Ch07/Ex01.HelloUnity.csproj。

練習步驟：

1. 建立一個新的 Console Application 專案，將專案名稱命名為 Ex01.HelloUnity。
2. 加入 Unity 套件。你可以用 Visual Studio 的 Package Manager 視窗來尋找並安裝 Unity 套件，或者也可以透過命令列的方式來安裝。命令列的操作方法是從 Visual Studio 主選單點【TOOLS | NuGet Package Manager | Package Manager Console】，接著在「Package Manager Console」視窗裡輸入以下命令（不包含前面的 `PM>` 提示字元）：

   ```
   PM> Install-Package Unity 【Enter】
   ```

3. 加入一個新介面：`ISayHello`。程式碼如下：

   ```
   interface ISayHello
   {
       void Run();
   }
   ```

4. 加入兩個新類別：`SayHelloInEnglish` 與 `SayHelloInChinese`。這兩個類別都要實作 `ISayHello` 介面。

```
class SayHelloInEnglish : ISayHello
{
    public void Run()
    {
        Console.WriteLine("Hello, Unity!");
    }
}

class SayHelloInChinese : ISayHello
{
    public void Run()
    {
        Console.WriteLine(" 哈囉, Unity!");
    }
}
```

5. 修改 program.cs，引用必要的命名空間，並撰寫 Main 函式如下。

```
using Unity;

namespace Ex01_HelloUnity
{
    class Program
    {
        static void Main(string[] args)
        {
            // (1) 建立 Unity 容器。
            IUnityContainer container = new UnityContainer();

            // (2) 向 Unity 容器註冊型別。
            container.RegisterType<ISayHello, SayHelloInEnglish>();

            // (3) 在程式某處，要求解析型別，以取得元件的執行個體。
            ISayHello hello = container.Resolve<ISayHello>();

            // (4) 呼叫元件的方法。
            hello.Run();

            Console.ReadKey();  // 等待按任意鍵結束程式。
        }
    }
}
```

6. 執行看看（按【F5】），程式應能正常執行，並且在螢幕上輸出字串「Hello, Unity!」。

說明：

- 抽象介面 `IUnityContainer` 定義了 Unity 容器的主要功能，而 `UnityContainer` 類別實作了此介面。
- 註冊型別的動作是在告訴 DI 容器：當用戶端要求某個抽象介面（`ISayHello`）的物件時，請呼叫其對應的實作類別（`SayHelloInEnglish`）的建構函式來建立物件。
- 解析型別的意思就是尋找用戶端所要求的相容型別，然後建立該型別的執行個體，並回傳給用戶端。
- `SayHelloInChinese` 類別在此範例中沒有特別作用，但後續各節會用到。

透過此練習，你至少已經知道兩件事：

- 註冊型別對應關係時呼叫的方法是 `RegisterType` 方法；
- 等到需要解析物件時，則是呼叫 `Resolve` 方法來取得特定類別的執行個體。

接著要進一步說明「註冊」與「解析」的相關細節，並介紹幾種常用的寫法。

註冊型別對應

前面提過，`UnityContainer` 類別實作了 `IUnityContainer` 介面。此介面定義了十幾個多載（overloaded）版本的 `RegisterType` 方法，你可以依實際需求選擇自己慣用的方法。這些多載方法最終大多是呼叫底下這個弱型別的（weakly typed）版本：

```
IUnityContainer RegisterType(
    Type from,  // 來源型別，通常是抽象型別（例如介面）
    Type to,    // 對應的目標型別（具象類別，或者實作類別））
    LifetimeManager lifetimeManager,  // 生命週期管理員
    params InjectionMember[] injectionMembers  // 注入成員
)
```

除此之外，Unity 框架還另外提供了其他方便的擴充方法，其中有些是強型別的（strongly typed）泛型方法，例如本書範例經常使用的：

```
container.RegisterType<IMessageservice, EmailService>();
```

其中第一個泛型參數是來源型別，第二個泛型參數則是對應的實作類別。

串接呼叫

每一個 `RegisterType` 方法都會傳回同一個 Unity 容器物件，因此我們甚至可以採用串接呼叫的（fluent）寫法，例如：

```
var container = new UnityContainer()
    .RegisterType<IFoo, Foo>()
    .RegisterType<IBar, Bar>()
    .RegisterType<IDuck, Duck>();
```

此寫法的好處是讓你不用一直重複寫好多「container.」。

註冊既有物件

除了 `RegisterType` 方法，Unity 還提供了 `RegisterInstance` 來讓你直接註冊既有的物件。

範例：

```
var emailSvc = new EmailService();

// 註冊既有的物件
IUnityContainer container = new UnityContainer();
container.RegisterInstance<IEmailService>(emailSvc);
```

需要解析物件時，一樣是呼叫 Resolve 方法，並傳入先前註冊時指定的抽象型別：

```
var svc = container.Resolve<IEmailService>();
```

解析

IUnityContainer 介面所定義的諸多方法當中，用來解析型別的並不多，就兩個：Reslove 和 ResolveAll。前者返回一個物件，後者則是返回一群物件。這兩個解析方法的原型宣告如下：

```
object Resolve(Type t, string name, params ResolverOverride[] resolverOverrides);
IEnumerable<object> ResolveAll(Type t, params ResolverOverride[] resolverOverrides)
```

接著先談談 Resolve 方法。等到介紹完「具名註冊」之後，再回頭來看 ResolveAll。

接下來的幾個註冊與解析 API 用法，可搭配參考書附範例的 Ch07/Ex02.RegisterResolveBasic.csproj 專案。

解析一個物件：Resolve

跟 RegisterType 方法一樣，Unity 框架也針對解析型別的操作提供了數個方便的擴充方法。只要你的程式有引用命名空間 Unity，就能使用其他擴充版本的 Resolve 方法了，例如：

```
    ISayHello hello = container.Resolve<ISayHello>();
```

值得一提的是，只有抽象型別才需要事先向 Unity 註冊。也就是說，當用戶端向 Unity 要求解析一個具象類別時，該類別並不需要事先註冊。參考以下範例：

```
static void Main(string[] args)
{
    // (1) 建立 Unity 容器。
    IUnityContainer container = new UnityContainer();

    // (2) 向 Unity 容器註冊型別。
    // container.RegisterType<ISayHello, SayHelloInEnglish>();

    // (3) 在程式某處，要求解析型別，以取得元件的執行個體。
    // ISayHello hello = container.Resolve<ISayHello>();
    ISayHello hello = container.Resolve<SayHelloInEnglish>();

    // (4) 呼叫元件的方法。
    hello.Run();
}
```

這一次，ISayHello 並未事先註冊至 Unity 容器，解析時則是改為要求解析實作類別 SayHelloInEnglish。

具名註冊與解析

如果對同一個抽象型別註冊重複許多次，那麼最後註冊的型別會取代先前的註冊。例如：

```
container.RegisterType<ISayHello, SayHelloInEnglish>();
container.RegisterType<ISayHello, SayHelloInChinese>(); // 取代先前的型別對應
```

解析 ISayHello 型別時，將會得到 SayHelloInChinese 的執行個體。

如果你真的需要讓同一抽象型別對應至多個實作類別，則可採用具名註冊的方式，例如：

```
container.RegisterType<ISayHello, SayHelloInEnglish>();
container.RegisterType<ISayHello, SayHelloInChinese>("CHN");
```

第一行程式碼會註冊為 ISayHello 的預設型別對應，而第二行程式碼則將此型別對應關係命名為「CHN」，於是 Unity 容器不至於把先前的預設型別對應蓋掉。等到需要解析型別時，便可以傳入此自訂名稱來指定要使用哪一個型別對應：

```
var hello = container.Resolve<ISayHello>("CHN");
```

呼叫 RegisterType 方法時若沒有指定註冊名稱，則稱之為「預設註冊」或「未具名的註冊」；有時也會說「預設型別對應」。

解析多個物件：ResolveAll

ResolveAll 可用來取得特定型別所對應之多個具象類別的執行個體，但只對具名註冊起作用。參考以下範例：

```
// 註冊
var container = new UnityContainer()
    .RegisterType<IHelloService, SayHelloInEnglish>("")      // (1)
    .RegisterType<IHelloService, SayHelloInEnglish>("ENG")   // (2)
    .RegisterType<IHelloService, SayHelloInChinese>("CHN")   // (3)
    .RegisterType<IHelloService, SayHelloInChinese>("CHN")   // (4)
    .RegisterType<IHelloService, SayHelloInChinese>("CHN2"); // (5)

// 解析全部
var services = container.ResolveAll<IHelloService>();
foreach (var svc in services)
{
    svc.Run();
}
```

說明：

- 此範例所註冊之抽象型別都是 `IHelloService`。
- 標示 (1) 者為預設註冊。由於 `ResolveAll` 方法只處理具名註冊，故這一筆型別註冊在稍後進行全部解析時會被忽略。
- 標示 (2)、(3)、(4)、(5) 者皆採用具名註冊，但 (4) 與 (3) 的名稱相同，後者會取代前者。

執行結果：

```
Hello, Unity!    <= 來自 (2) 的具名註冊 "ENG"
哈囉, Unity!      <= 來自 (4) 的具名註冊 "CHN"
哈囉, Unity!      <= 來自 (5) 的具名註冊 "CHN2"
```

`ResolveAll` 方法只處理「具名註冊」，碰到「預設註冊」時會直接忽略。

註冊與解析泛型

泛型的註冊與解析寫法並無特殊之處。僅舉一簡短範例：

```
// 註冊
container.RegisterType<IRepository<Customer>, Repository<Customer>>();

// 解析
var repo = container.Resolve<IRepository<Customer>>();
```

檢查註冊

Unity 容器提供了 `IsRegister` 方法，可用來檢查指定的型別是否已經註冊。用法很簡單，參考以下範例：

```
IUnityContainer container = new UnityContainer();
if (container.IsRegistered<EmailService>())
{
    // 已向 Unity 容器註冊。
}
```

`IsRegistered` 方法也支援具名註冊，只要在呼叫此方法時傳入欲檢查的註冊名稱即可。

另外，Unity 容器還提供了 `Registrations` 屬性，讓你能夠取得已註冊至容器的的全部型別資訊。此功能的主要用途是方便除錯，例如找出不正確的型別對應關係。以下範例會將容器中的註冊資訊逐一取出，並以格式化字串來顯示每一筆型別對應關係。

```
foreach (ContainerRegistration item in container.Registrations)
{
    var s = String.Format(
        "Name: '{0}', map from: {1}, map to: {2}",
        item.Name, item.RegisteredType.Name, item.MappedToType);
    Console.WriteLine(s);
}
```

使用組態檔來設定容器

除了透過程式碼來註冊型別對應關係，Unity 也有提供設計時期組態（design-time configuration）的作法，也就是利用組態檔來完成相關設定。這種作法的最大好處就是調整設定時只需修改組態檔，而不用重新編譯程式。

參考書附範例原始碼的 Ch07/Ex03.ConfigFile.csproj 專案。

Unity 的組態設定係使用 .NET 應用程式組態檔。在使用 Unity 組態 API 時，你的專案至少要加入 `System.Configuration` 組件參考，並引用下列命名空間：

- Unity
- Microsoft.Practices.Unity.Configuration

Unity 組態檔基本格式

編輯組態檔時，我們得先加入一個自訂區段，而這個自訂區段的名稱通常命名為「unity」。底下是一個基本範例：

```
<configSections>
  <section
    name="unity"
    type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
          Unity.Configuration"/>
</configSections>

<unity>
  <container>
    <register
      name=""
      type="Examples.Common.Contracts.IMessageService, Examples.Common"
      mapTo="Examples.Common.Services.EmailService, Examples.Common" />
  </container>
</unity>
```

注意：從 Unity v5.2.1 開始，類別 `Microsoft.Practices.Unity.Configuration.UnityConfigurationSection` 被搬到 Unity.Configuration.dll 組件中。

在自訂區段 `<unity>` 裡面的 `<container>` 區段中所包含每一個 `<register>` 元素即代表一筆型別對應的註冊資訊，其中：

- 屬性 `name` 是註冊名稱。若不指定，即為預設註冊（未具名的註冊），亦即 Unity 解析 `type` 屬性所指定之型別時的預設型別對應。
- 屬性 `type` 是這筆型別對應關係中的來源型別。
- 屬性 `mapTo` 是這筆型別對應關係中的目標型別。

以上範例是使用全名（full qualified name）來指定型別名稱。當註冊項目很多時，這種寫法恐怕太過冗長。此時可以在 `<unity` 區段中先宣告組件名稱與命名空間。像這樣：

```
<unity>
  <assembly name="Examples.Common" />
  <namespace name="Examples.Common.Contracts" />
  <namespace name="Examples.Common.Services" />

  <container>
    <register name="" type="IMessageService" mapTo="EmailService" />
    <register name="sms" type="IMessageService" mapTo="SmsService" />
  </container>
</unity>
```

如此一來，每一個 `<register>` 元素的內容就簡潔多了。

載入組態檔設定

組態檔編寫完成後，接著就能夠在程式中呼叫 Unity 容器的擴充方法 `LoadConfiguration` 來載入組態檔的相關設定。例如：

```
IUnityContainer container = new UnityContainer();
container.LoadConfiguration();
```

載入組態檔的設定之後，接著就可以呼叫容器的 `Resolve` 方法來解析型別。這個部分與先前介紹的寫法並無二致，故不再贅述。

> **執行時期組態 vs. 設計時期組態**
>
> 雖然用組態檔來控制介面與實作類別的對應有其方便之處——無須重新編譯程式碼就能抽換實作類別——但它也不是沒有缺點。相較於利用程式碼來設定組態的作法，組態檔比較容易出錯，而且可能更不好維護。至於不利維護，這點恐怕就見仁見智了。我總覺得 XML 格式不利人眼閱讀，尤其當組態檔越長越大時。

因此，最好只有在真的需要不重新編譯程式就任意抽換類別的情況才使用組態檔；其餘的場合，筆者仍建議優先考慮使用程式碼來設定型別對應。這裡特別強調「真的需要」，是因為我們有時候會不加思索地將所有東西一股腦兒塞進組態檔案裡，或者認為「任何東西都可以隨時抽換就是最好最彈性的作法。」其實都還有商榷餘地。

當然，執行時期組態（Code as Configuration）和設計時期組態（XML 組態檔）這兩種作法並不互相排斥，它們是可以一起搭配運用的。例如我們可以將大部分的型別對應都以程式碼來設定，而少數需要隨時任意切換的類別則放在組態檔案裡。這是一種搭配方式。又或者，以程式碼來設定所有的型別對應關係，之後再接著載入 XML 組態的方法，讓 XML 組態設定蓋掉原先程式碼的設定，而且是只有重複的部分才蓋掉；對於 XML 組態沒有涵蓋的部分，仍然會使用程式碼的設定。

註冊與解析－進階篇

當你在應用程式中使用 Unity 容器的分量愈來愈多，可能會碰到一些問題，例如：

- 需要註冊的型別好多，能不能少寫一點註冊型別的程式碼？（參閱〈自動註冊〉一節的說明）
- 欲解析的類別包含數個多載建構函式，我怎麼知道 Unity 會使用哪一個建構函式？或者，我如何明白要求 Unity 使用某個特定的建構函式？（參閱〈自動匹配〉一節的說明）
- 使用 **Constructor Injection** 時，目標建構函式需要傳入基本型別(primitive types)，這是 Unity 容器的自動深層解析（參閱第 3 章）功能無法處理的部分，那有辦法讓我注入像 `String`、`Int32` 等基本型別的參數值嗎？（參閱〈注入參數〉）
- 由於 Unity 會自動深層解析相依物件，可是我不希望在解析某個高層模組的元件時就連帶建立一大堆的物件，導致應用程式初始化時間拉得太長。我可以延遲這些相依物件的建立時機嗎？（參閱〈延遲解析〉）

本節的目的便在於介紹 Unity 的一些進階功能，以解決實際開發時可能碰到的狀況。這些功能包括：自動註冊（auto-registraction）、自動匹配（auto-wiring）、延遲解析（deferred resolution）等等。其中關於延遲解析的部分，還包括了自動工廠（automatic factories）、注入自訂工廠等技巧。至於其他同等重要的進階議題，例如物件生命週期管理、攔截（interception）等等，也會在後面陸續介紹。

共用的範例程式

除了先前「Hello, Unity!」範例程式中的介面與類別，從本節開始會加入其他型別。為了避免往後重複太多相同的程式碼，這裡先列出後續各節所共用的介面與類別。

本章共用範例程式的專案名稱是 Ch07.Common.csproj。

情境

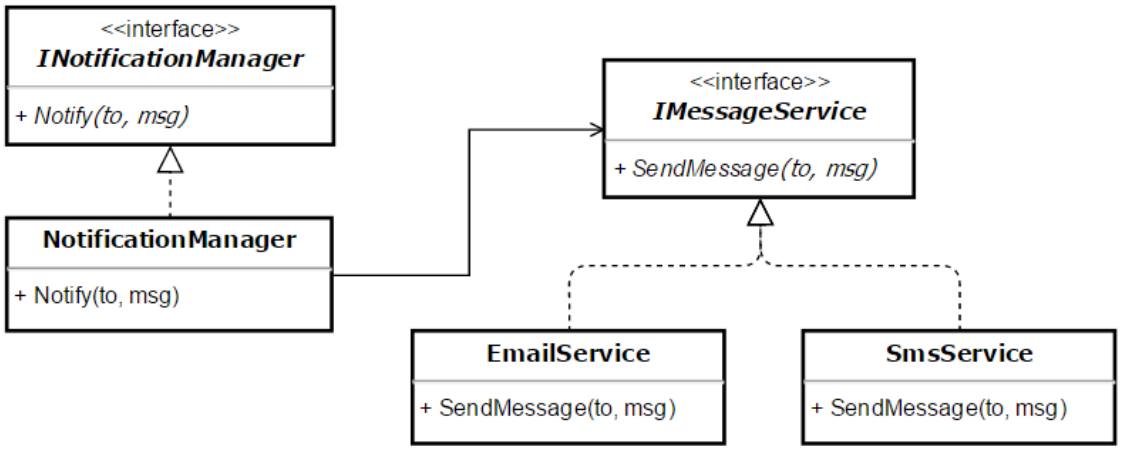
假設應用程式需要提供訊息通知機制，而此機制需支援多種發送管道，例如：電子郵件、簡訊服務（Short Message Service）、行動應用程式的訊息推送（push notification）等等。簡單起見，這裡僅實作其中兩種服務，而且發送訊息的部分都使用簡單的 `Console.WriteLine()` 來輸出訊息，方便觀察程式的執行結果。

設計

用一個 `NotificationManager` 類別來作為整個訊息通知功能的管理員。各類訊息通知機制則由以下類別提供：

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

以上兩個類別均實作同一個介面：IMessageService，而且 NotificationManager 只知道 IMessageService 介面，而不直接依賴具象類別。底下的類別圖描繪了它們的關係：



程式碼

訊息通知管理員的相關程式碼：

```
public interface INotificationManager
{
    void Notify(string to, string msg);
}

public class NotificationManager : INotificationManager
{
    private readonly IMessageService _msgService = null;

    // 從建構函式注入訊息服務物件。
    public NotificationManager(IMessageService svc)
    {
        _msgService = svc;
    }

    // 利用訊息服務來發送訊息給指定對象。
```

```
    public void Notify(string to, string msg)
    {
        _msgService.SendMessage(to, msg);
    }
}
```

這裡採用了 Constructor Injection 的注入方式，由建構函式傳入訊息服務。其中的 Notify 方法則利用事先注入的訊息服務來發送訊息給指定的接收對象（引數 'to'）。在真實世界中，你可能會需要用額外的類別來代表訊息接收對象（例如設計一個 MessageRecipient 類別來封裝收件人的各項資訊），這裡為了示範而對這部分做了相當程度的簡化。

底下是各類訊息服務的程式碼：

```
public interface IMessageService
{
    void SendMessage(string to, string msg);
}

public class EmailService : IMessageService
{
    public void SendMessage(string to, string msg)
    {
        Console.WriteLine(" 透過 EmailService 發送郵件給 {0}。", to);
    }
}

public class SmsService : IMessageService
{
    public void SendMessage(string to, string msg)
    {
        Console.WriteLine(" 透過 SmsService 發送簡訊給 {0}。", to);
    }
}
```

往後各節若有特殊的狀況和需求，會個別再列出修改後的程式碼。

自動註冊

「自動註冊」的另一種說法是「依慣例註冊」（registration by convention），主要用意是讓開發人員能夠少寫一些註冊型別的程式碼。Unity 是透過擴充方法 `RegisterTypes`（注意 “Types” 是複數）來提供自動註冊的功能。此方法有兩個多載版本，底下是它們的原型宣告：

```
// (1)
public static IUnityContainer RegisterTypes(
    this IUnityContainer container,
    IEnumerable<Type> types,                                  // (1)
    Func<Type, IEnumerable<Type>> getFromTypes=null,          // (2)
    Func<Type, string> getName=null,                          // (3)
    Func<Type, LifetimeManager> getLifetimeManager=null,  // (4)
    Func<Type, IEnumerable<InjectionMember>> getInjectionMembers=null, // (5)
    bool overwriteExistingMappings=false)                     // (6)
    /* 以下依上列標示的註解編號分別說明：
      (1) 具象類別清單（欲解析之目標類別）。
      (2) 委派：傳回抽象型別清單。
      (3) 委派：傳回特定類別的註冊名稱。
      (4) 委派：傳回生命週期管理員。
      (5) 委派：傳回注入成員。
      (6) 類別重複註冊時，後者是否蓋掉前者？
    */

// (2)
public static IUnityContainer RegisterTypes(
    this IUnityContainer container,
    RegistrationConvention convention,
    bool overwriteExistingMappings = false)
```

先來看一個簡單的用法，使用的是上列標示 (1) 的 `RegisterTypes` 多載方法：

```
// 範例：讓 Unity 容器自動掃描組件並註冊型別。
var container = new UnityContainer();
container.RegisterTypes(
    AllClasses.FromLoadedAssemblies(), // 掃描目前已載入此應用程式的全部組件。
    WithMappings.FromAllInterfaces);   // 尋找所有介面。
```

這裡省略了大部分非必要的參數，而只傳入兩個參數：

- 參數 `types` 是具象類別清單。這裡使用了 Unity 提供的輔助類別 `AllClasses` 的 `FromLoadAssemblies` 方法來提供類別清單。
- 參數 `getFromTypes` 是個用來取得來源型別清單的委派（delegate）。「來源型別」指的就是註冊型別對應關係時的抽象型別。這裡傳入的委派是指向 Unity 的 `WithMappings.FromAllInterfaces` 方法，作用是取得所有類別所實作的介面。

假設此範例應用程式目前已載入的組件當中已經有一個 `EmailService` 類別，而且該類別實作了 `IMessageService` 介面，那麼當應用程式需要取得符合 `IMessageService` 介面的物件時，由於 Unity 已經自動尋找並註冊了相關型別，於是我們便可透過先前提過的型別解析方法來取得物件：

```
var svc = container.Resolve<IMessageService>();
```

參考書附範例：Ch07/Ex04.AutoRegistration.csproj。

解決重複型別對應的問題

預設情況下，「自動註冊」會採用未具名的「預設註冊」（這名詞前面有提過），而且在碰到欲解析的類別有多載建構函式時，會根據一套既定規則來挑選建構函式。因此，如果只用剛才的簡單範例來進行自動註冊，很可能會在程式執行時發生型別註冊失敗或無法解析特定型別的錯誤。

就拿前面提過的訊息通知服務的範例來說，由於 EmailService 和 SmsService 都實作了 IMessageService 介面，若以上述範例程式來進行自動註冊，那麼當程式執行時，將會在呼叫 RegisterTypes 方法的地方發生重複型別對應的錯誤：

> An unhandled exception of type 'Microsoft.Practices.Unity.DuplicateTypeMappingException' occurred in Microsoft.Practices.Unity.RegistrationByConvention.dll

碰到這種情形，你可以透過 RegisterTypes 方法的 getName 參數（它是個委派）來為不同的類別提供不同的註冊名稱。

```
var container = new UnityContainer();
container.RegisterTypes(
    AllClasses.FromLoadedAssemblies(),  // 掃描目前已載入此應用程式的全部組件。
    WithMappings.FromAllInterfaces,     // 尋找所有介面。
    getName: WithName.TypeName);        // 使用類別名稱來當作註冊名稱。
```

其中的 WithName 是 Unity 提供的輔助類別，而它的 TypeName 方法可傳回指定型別的名稱。也就是說，如果沒有特別複雜的需求，我們可以直接把 WithName.TypeName 方法傳遞給 getName 參數。如此一來，每一個具象類別的型別對應關係就會以該類別的名稱來命名。於是，在解析物件的時候，也必須明白指定註冊名稱。像這樣：

```
var svc = container.Resolve<IMessageService>("EmailService");
```

另一個避免型別對應重複的解決方法，是利用參數 overwriteExistingMappings。當你只需要找到任何一個可用的實作類別，而不在乎 Unity 最終會採用哪一個，此時便可將參數 overwriteExistingMappings 指定為 true，如此亦可避免型別對應重複的問題，而且無須使用具名註冊。參考以下範例：

```
var container = new UnityContainer();
container.RegisterTypes(
    AllClasses.FromLoadedAssemblies(),
    WithMappings.FromAllInterfaces,
    overwriteExistingMappings: true); // 有多個符合條件的類別時，用最後那個。
```

接著要來進一步了解 `AllClasses` 和 `WithMappings` 類別還提供了哪些輔助功能。

`AllClasses` 類別

`AllClasses` 是 Unity 提供的一個工具類別，它提供了下列方法來協助取得類別清單：

方法名稱	說明	適用平台
`FromAssemblies`	從你提供的組件清單中取得所有可見的（visible）類別。	桌上型、Windows Store app
`FromAssembliesInBasePath`	從目前應用程式的基礎路徑取得組件清單，然後從這些組件當中找出所有可見的類別。	桌上型應用程式
`FromLoadedAssemblies`	從目前應用程式已經載入的組件中取得所有可見的類別。	桌上型應用程式
`FromApplication`	從目前應用程式安裝目錄下載入的組件中取得所有可見的類別。	Windows Store app

這些方法的回傳型別都是 `IEnumerable<Type>`，方便我們在呼叫 `RegisterTypes` 方法時直接傳遞給 `types` 參數，以提供欲自動註冊的類別清單。

在 web 應用程式中呼叫 `FromAssembliesInBasePath` 方法時，它並不會處理 Bin 目錄底下的組件。如欲自動註冊 Bin 目錄下各組件的型別，應呼叫 `System.Web.Compilation.BuildManager` 類別的 `GetReferencedAssemblies` 方法來取得組件清單，然後將這份清單當作輸入參數，餵給 `AllClasses.FromAssemblies` 方法。

有些方法（例如 FromAssembliesInBasePath）傳回的類別數量可能太多，使得 Unity 浪費時間去處理不相關的類別。為了節省時間，除了使用 FromAssemblies 來自行提供組件清單之外，你也可以在將類別清單傳遞給 RegisterTypes 方法之前先行過濾。參考以下範例：

```
var container = new UnityContainer();

// 在自動註冊之前，先行篩選類別。
var classes = from t in AllClasses.FromLoadedAssemblies()
              where t.Name.IndexOf("Service") > 0
              select t;
container.RegisterTypes(classes);
```

參考書附範例：Ch07/Ex05.AllClassesDemo.csproj。

WithMappings 類別

顧名思義，WithMappings 這個輔助類別的用途在於讓我們方便控制自動註冊時的型別對應關係。下表列出了 WithMappings 提供的方法：

方法名稱	說明
FromAllInterfaces	傳回指定類別所實作的所有介面。
FromAllInterfacesInTheSameAssembly	傳回指定類別所實作的所有介面，而且那些介面必須和實作類別位於相同組件。
FromMatchingInterface	傳回符合命名慣例的介面。所謂的命名慣例指的是在類別名稱前面冠上字母 'I' 則為對應之介面名稱。例如 IOrderService 與 OrderService。
None	傳回空集合。這表示不為指定的類別註冊任何型別對應關係。

這些方法都有一個型別為 Type 的傳入參數，代表特定實作類別；回傳型別則

為 IEnumerable<Type>。它們的目的都是要提供了特定的介面集合，以便包裝成委派，並直接傳遞給 RegisterTypes 方法的 getFromTypes 參數（型別是 Func<Type, IEnumerable<Type>>）。

自動匹配

採用 **Constructor Injection** 的方式來注入物件時，如果欲解析的類別僅提供一個建構函式，這種情形通常不會產生任何疑慮，因為 Unity 在進行解析時一定是呼叫那個唯一的建構函式來生成物件。然而，如果欲解析的類別提供了數個多載（overloaded）建構函式，Unity 就會使用其內建的一套規則來挑選建構函式。身為開發人員，我們有必要了解這套規則，一方面有助於正確解讀程式碼的運作邏輯，一方面則可避免因為不解其中規則而寫出了產生執行時期錯誤的程式碼。

以先前的 EmailService 為例，如果為這個類別增加一個帶參數的建構函式，使之具有兩個建構函式：

```
public class EmailService : IMessageService
{
    public EmailService()
    {
        Console.WriteLine("EmailMessageService ctor()");
    }

    public EmailService(string smtpHost)
    {
        Console.WriteLine("EmailMessageService ctor(smtpHost)");
    }
}
```

那麼，當我們以下列程式碼來解析 EmailService 時，Unity 會拋出 ResolutionFailedException 錯誤。

```
var container = new UnityContainer();
var svc = container.Resolve<EmailService>();
```

這是因為預設情況下，Unity 會使用參數最多的那個建構函式，而非不帶任何參數的預設建構函式。於是，在解析 `EmailService` 時，Unity 容器會呼叫帶有 `smtpHost` 參數的建構函式，並試圖解析這個參數。在沒有提供額外設定的情況下，Unity 無法解析 `string` 型別的參數，於是拋出異常。

自動匹配規則

承上所述，這種根據特定規則來自動挑選建構函式的機制叫做**自動匹配**（**auto-wiring**）。底下是完整的自動匹配邏輯：

1. 當目標型別具有多個建構函式，則選擇有套用 `InjectionConstructor` 特徵項（attribute）的那個建構函式；
2. 如果全部的建構函式皆未套用 `InjectionConstructor` 特徵項，則使用參數個數最多的那個建構函式；
3. 如果參數個數最多的建構函式並不只一個（例如有兩個建構函式都需要傳入 5 個參數），Unity 會拋出異常。
4. 如果同一類別有兩個或兩個以上的建構函式套用了 `InjectionConstructor` 特徵項，Unity 也會拋異常。

了解這些規則之後，先前的問題便迎刃而解。例如，我們可以為預設建構函式套上 `[InjectionConstructor]`，像這樣：

```
public class EmailService : IMessageService
{
    [InjectionConstructor]
    public EmailService()
    {
        Console.WriteLine("EmailMessageService ctor()");
    }

    public EmailService(string smtpHost)
    {
        Console.WriteLine("EmailMessageService ctor(smtpHost)");
    }
}
```

如此一來，Unity 在進行解析 `EmailService` 時便會改用它的預設建構函式了。

參考書附範例：Ch07/Ex06.AutoWiring.csproj。

手動匹配

除了剛才介紹的 `InjectionConstructor` 特徵項，Unity 也支援手動匹配的方式，以應付各種需求。關鍵的類別名稱也叫做 `InjectionConstructor`，它可以用於 **Constructor Injection** 的場合，讓我們在註冊類別時就明確指定要使用哪一個建構函式，並且連同目標建構函式所需要的傳入參數也都預先準備好。

如果同時使用了 `InjectionConstructor` 類別和 `InjectionConstructor` 特徵項，Unity 會採用前者。

沿用先前範例的 `NotificationManager` 類別。現在假設要讓它支援多種訊息通知機制，於是寫了三個建構函式，分別可傳入零個、一個、和兩個 `IMessageService` 物件。如下所示：

```
class NotificationManager
{
    public NotificationManager()
    {
        Console.WriteLine(" 呼叫了無參數的建構函式。");
    }

    public NotificationManager(IMessageService svc)
    {
        Console.WriteLine(" 呼叫了一個參數的建構函式。");
    }

    public NotificationManager(IMessageService svc1, IMessageService svc2)
    {
        Console.WriteLine(" 呼叫了兩個參數的建構函式。");
    }
}
```

底下是註冊與解析的程式碼：

```
    var container = new UnityContainer();

    // (1) 具名註冊 EmailService 和 SmsService 類別。
    container.RegisterType<IMessageService, EmailService>("email");
    container.RegisterType<IMessageService, SmsService>("sms");

    // (2) 註冊 NotificationManager 類別。
    container.RegisterType<NotificationManager>(
        new InjectionConstructor(
            new ResolvedParameter<IMessageService>("email"),
            new ResolvedParameter<IMessageService>("sms")
        )
    );

    // 解析 NotificationManager。
    container.Resolve<NotificationManager>();
```

說明：

1. 註冊 `EmailService` 與 `SmsService` 採用具名註冊。註冊名稱分別是 “email” 和 “sms”。

2. 註冊 NotificationManager 類別時傳入一個 InjectionConstructor 物件，此物件帶有兩個已解析的參數（ResolvedParameter）。於是，Unity 在解析 NotificationManager 時，便知道要呼叫帶有兩個 IMessageService 參數的建構函式。同樣地，如果想要使用預設建構函式，則只要寫成 new InjectionConstructor() 即可。

其中的 ResolvedParameter 類別是用來包裝「需要進一步解析的參數」。請注意，在 new 一個 ResolvedParamter 物件的時候並沒有立刻執行解析型別的動作，而是會等到將來呼叫 Resolve 方法時才會解析參數。

參考書附範例：Ch07/Ex07.OverrideAutoWiring.csproj。

循環參考問題

如果兩個類別的建構函式都需要注入對方，像這樣：

```
public class Foo : IFoo
{
    public Foo()
    { }

    public Foo(IBar bar)  // Foo 需要注入 Bar
    { }
}

public class Bar : IBar
{
    public Bar(IFoo foo)  // Bar 又需要 Foo
    { }
}
```

那麼當 Unity 在解析 IFoo 或 IBar 時，便會因為循環參考而引發 StackOverflowException。原因即在於 Unity 的自動匹配（auto-wiring）機制預設會使用最多參數的那個建構函式。碰到這種情形，可在註冊型別時使用 InjectionConstructor 來指定其他建構函式。參考以下範例：

```
// 註冊時一併指定將來解析元件時應使用不帶任何參數的建構函式。
container.RegisterType<IFoo, Foo>(new InjectionConstructor());
```

注入參數

剛才提過，`InjectionConstructor` 類別可用來控制解析時要使用哪一個多載版本的建構函式，並預先準備傳入建構函式的參數。底下是此類別的建構函式的原型宣告：

```
public InjectionConstructor(params object[] parameterValues);
```

由此可見，其建構函式可接受任意個數、任意型別的物件。

上一節的範例是透過 `ResolvedParameter` 來包裝「需要解析的」傳入參數，那麼現在考慮另一種情況：類別的建構函式需要傳入的參數是基本型別（primitive types）。由於基本型別不需要解析，故可直接傳入，而無須動用 `ResolvedParameter`。舉例來說，如果 `NotificationManager` 有一建構函式需要傳遞 `IMessageService` 和 `int`，像這樣：

```
public NotificationManager(IMessageService svc, int timeout)
{
    // (略)
}
```

那麼，在註冊類別時可以這麼寫：

```
container.RegisterType<NotificationManager>(
    new InjectionConstructor(
        new ResolvedParameter<IMessageService>("email"),
        500 // 傳遞整數 500 給參數 'timeout'。
));
```

解析類別的寫法不變：

```
container.Resolve<NotificationManager>();
```

解析物件時改寫注入之參數與物件

Unity 不僅在註冊類別時提供了 API 讓你動態注入相依物件和參數，它還能夠讓你在解析物件時改寫（override）先前註冊型別時提供的參數與相依物件。也就是說，如果有需要的話，呼叫 `RegisterType`（或 `RegisterInstance`）時注入的參數和相依物件，是可以在呼叫 `Resolve` 方法時把它們給替換掉的。這個解析時期的改寫機制主要涉及三個類別：

- `ParameterOverride`：用來改寫（取代）建構函式的傳入參數。
- `PropertyOverride`：用來改寫（取代）屬性值。
- `DependencyOverride`：用來改寫（取代）特定型別的相依物件。

如需進一步資訊，可參考官方文件：Resolving Objects by Using Overrides[a]。

[a] http://msdn.microsoft.com/en-us/library/dn507420.aspx

注入屬性

除了建構函式的傳入參數，Unity 亦可於註冊類別時指定將來解析時需要一併注入的屬性。提供此功能的類別是 `InjectionProperty`。

假設 `NotificationManager` 有個屬性叫做 `Timeout`，型別是 `int`。如需注入屬性值，可以這麼寫：

```
container.RegisterType<NotificationManager>(
    new InjectionConstructor(new ResolvedParameter<IMessageService>("email")),
    new InjectionProperty("Timeout", 500)
);
```

參考書附範例：Ch07/Ex07.OverrideAutoWiring.csproj。

呼叫 `RegsiterType` 方法時，你還可以透過 Unity 的 `InjectionFactory` 類別來注入自訂的物件工廠。稍後會有進一步的說明。

延遲解析

當我們說「解析某個型別／元件」時，意思通常是呼叫某類別的建構函式，以建立其執行個體（instance）。但有些場合，我們會希望解析時先不要生成物件，而是等到真正要呼叫物件的方法時才建立物件。這種延後建立物件的解析方式，叫做「延遲解析」（deferred resolution）。

延遲解析通常用在哪裡呢？一個典型的場合是欲解析的物件的創建過程需要花較多時間（例如解析時可能因為建構函式需要注入其他物件，而產生多層巢狀解析的情形），而我們希望能加快這個過程，以提升應用程式的回應速度。

本節提供兩種實現延遲解析的作法，一種是 `Lazy<T>`，另一種是「自動工廠」（automatic factories）。以下兩個小節將分別介紹這兩種作法，以及相關的程式技巧（注入自訂工廠）。

使用 `Lazy<T>`

Unity 可以讓我們直接使用 .NET Framework 內建的 `Lazy<T>` 來實現延遲解析。試比較底下兩個範例，首先是一般的寫法：

```
// 一般寫法
var container = new UnityContainer();
container.RegisterType<IMessageService, EmailService>();  // 註冊

// 解析元件（呼叫實作類別的建構函式）
var svc = container.Resolve<IMessageService>();

// 使用元件
svc.SendMessage("Michael", "Hello!");
```

然後是 `Lazy<T>` 的延遲解析寫法，由於註冊型別的程式碼不變，故只列出解析的部分：

```
var lazyObj = container.Resolve<Lazy<IMessageService>>(); // 延遲解析
var svc = lazyObj.Value;  // 此時才會呼叫類別的建構函式
svc.SendMessage("Michael", "Hello!"); // 使用元件
```

參考書附範例：Ch07/Ex08.DeferredResolution.csproj。

使用自動工廠

「自動工廠」（automatic factories）指的是 DI 容器能夠自動生成一個輕量級的工廠類別，這樣我們就不用花工夫自己寫了。那麼，什麼情況會用到自動工廠呢？通常是用來實現「延遲解析」，或者應付更複雜、更動態的晚期繫結（late binding）的需求。

仍舊以先前提過的 `NotificationManager` 和 `IMessageService` 為例。假設 `EmailService` 這個元件在建立執行個體時需要花費較長的時間（約五秒），參考以下程式片段：

```
public class EmailService : IMessageService
{
    public EmailService()
    {
        // 以暫停執行緒的方式來模擬物件生成的過程需要花費較長時間。
        System.Threading.Thread.Sleep(5000);
    }

    // 其餘程式碼在這裡並不重要，予以省略。
}
```

如果照一般的元件解析方式，會這麼寫：

```
    // (1) 註冊
    var container = new UnityContainer();
    container.RegisterType<IMessageService, EmailService>();

    // (2) 解析
    var notySvc = container.Resolve<NotificationManager>();

    // (3) 呼叫
    notySvc.Notify("Michael", " 自動工廠範例");
```

這種寫法，在其中的「**(2) 解析**」這個步驟會發生下列動作：

1. DI 容器欲解析 `NotificationManager`，發現其建構函式需要傳入 `IMessageService` 物件，於是先解析 `IMessageService`。
2. 由於先前向容器註冊元件時已經指定由 `EmailService` 來作為 `IMessageService` 的實作類別，故容器會先建立一個 `EmailService` 物件，然後將此物件傳入 `NotificationManager` 的建構函式，以便建立一個 `NotificationManager` 物件。

也就是說，在解析 `NotificationManager` 時便一併建立了 `EmailService` 物件，故此步驟至少要花五秒的時間才能完成。然而，現在我們想要延後相依物件的創建時機，亦即等到真正呼叫元件的方法時，才真正建立其相依物件的執行個體。像這種場合，我們可以利用 `Func<T>` 與 Unity 的「自動工廠」來達到延遲解析相依物件的效果。作法很簡單，只要修改 `NotificationManager` 類別就行了。如下所示：

```
class NotificationManager
{
    private IMessageService _msgService;
    private Func<IMessageService> _msgServiceFactory

    public NotificationManager(Func<IMessageService> svcFactory)
    {
        // 把工廠方法保存在委派物件裡
        _msgServiceFactory = svcFactory;
    }

    public void Notify(string to, string msg)
    {
        // 由於每次呼叫 _msgServiceFactory() 時都會建立一個新的
        // IMessageService 物件，這裡用一個私有成員變數來保存先前建立的
        // 物件，以免不斷建立新的執行個體。當然這並非必要；有些場合，你可能
        // 會想要每次都建立新的相依物件。
        if (_msgService == null)
        {
            _msgService = _msgServiceFactory();
        }

        _msgService.SendMessage(to, msg);
    }
}
```

另一方面，原先的「註冊、解析、呼叫」三步驟的程式碼都不用任何改變。方便閱讀起見，這裡再將註冊元件的程式碼貼上來：

```
// (1) 註冊
var container = new UnityContainer();
container.RegisterType<IMessageService, EmailService>();
```

請注意，`NotificationManager` 的建構函式要求注入的明明是 `Func<IMessageService>`，可是向容器註冊元件時，卻依舊寫 `IMessageService`，而不用改成 `Func<IMessageService>`（要五毛，給一塊）。如此一來，當你想要為既有程式碼加入延遲解析（延遲建立相依物件）的能力時，就可以少改一些程式碼。這是 Unity 容器的「自動工廠」提供的好處。

進一步解釋，當 Unity 容器欲解析 `NotificationManager` 時，發現其建構函式需要一個 `Func<IMessageService>` 委派（delegate），於是便自動幫你生成這個物件，並將它注入至 `NotificationManager` 類別的建構函式。由於注入的是委派物件（你可以把它當作是個**工廠方法**），故此時並沒有真正建立 `IMessageService` 物件，而是等到上層模組呼叫此元件的 `Notify` 方法時，才透過呼叫委派方法來建立 `IMessageService` 物件。

參考書附範例：Ch07/Ex09.AutoFactory.csproj。

當然，Unity 容器的「自動工廠」可能無法滿足某些需求。比如說，有些相依物件的創建邏輯比較複雜，需要你撰寫自訂的物件工廠。這個時候，你可能會想要知道如何注入自訂工廠。那麼請接著往下看。

注入自訂工廠

當你想要讓 Unity 容器在解析特定元件時使用你的自訂工廠來建立所需之相依物件，Unity 框架的 `InjectionFactory` 類別可以派上用場。

參考書附範例：Ch07/Ex10.CustomFactory.csproj。

延續上一個小節的 `NotificationManager` 範例。現在假設你寫了一個物件工廠來封裝 `IMessageService` 的創建邏輯，像這樣：

```
class MessageServiceFactory
{
    public IMessageService GetService()
    {
        bool isEmail = CheckIfEmailIsUsed();
        if (isEmail)
        {
            return new EmailService();
        }
        else
        {
            return new SmsService();
        }
    }
}
```

此物件工廠的 GetService 方法會根據執行時期的某些變數來決定要返回 EmailService 還是 SmsService 的執行個體。EmailService 與 SmsService 這兩個類別都實作了 IMessageService 介面，它們的程式碼在這裡並不重要，故未列出。如需查看這些類別的程式碼，可參閱稍早的〈共用的範例程式〉一節的內容。

NotificationManager 的建構函式與上一節的範例相同，仍舊是注入 Func<IMessageService>。如下所示：

```
class NotificationManager
{
    private IMessageService _msgService;
    private Func<IMessageService> _msgServiceFactory

    public NotificationManager(Func<IMessageService> svcFactory)
    {
        // 把工廠方法保存在委派物件裡
        _msgServiceFactory = svcFactory;
    }

    // （已省略其他不重要的程式碼）
}
```

剩下的工作，就是告訴 Unity 容器：「在需要解析 IMessageService 的時候，請使用我的 MessageServiceFactory 來建立物件。」參考以下程式片段：

```
using Unity;
using Unity.Injection;

......

var container = new UnityContainer();

// 註冊
Func<IMessageService> factoryMethod = new MessageServiceFactory().GetService;
container.RegisterType<IMessageService>(new InjectionFactory(c => factoryMethod()));

// 解析
container.Resolve<NotificationManager>();
```

註冊元件的部分需要加以說明，如下：

- 先建立一個 Func<IMessageService> 的委派物件，讓它指向 MessageServiceFactory 物件的 GetService 方法。
- 接著呼叫 Unity 容器的 RegisterType 方法，告訴容器：解析 IMessageService 時，請用我提供的自訂工廠的 GetService 方法，而這個工廠方法已經包在剛才建立的委派物件（變數 factoryMethod)，並透過 Unity 的 InjectionFactory 將此工廠方法再包一層，以便保存於 Unity 容器。

此範例所使用的 RegisterType 是個擴充方法，其原型宣告如下：

```
public static IUnityContainer RegisterType<T>(
    this IUnityContainer container,
    params InjectionMember[] injectionMembers);
```

InjectionFactory 類別繼承自 InjectionMember，而此範例所使用的建構函式之原型宣告為：

```
public InjectionFactory(Func<IUnityContainer, object> factoryFunc);
```

如需 InjectionFactory 類別的詳細說明，可參考線上文件，網址是：

https://unitycontainer.github.io/api/Unity.Injection.InjectionFactory.html

物件生命週期管理

我們已經知道，DI 容器在「解析型別」時也包含了「建立物件」的動作。既然肩負了建立物件的責任，DI 容器自然也應具備摧毀物件的功能，以便在適當時機釋放物件所占用的記憶體。然而，什麼時候才是「適當時機」呢？這得依不同的需求而定。

在使用 DI 容器時，開發人員得要知道，每當透過容器來解析某型別時，應當建立新物件、還是直接使用先前已經建立好的物件；以及，先前由容器所建立的物件，是要一直保留到容器本身摧毀時才一併釋放、還是在符合特定條件時就要先行釋放。這些涉及物件生成方式與存活時間長短的議題，一般統稱為「物件生命週期管理」。

針對各種不同的需求，Unity 分別實作了不同類型的生命週期管理員。在介紹這些生命週期管理員之前，讓我們先來了解，預設情況下，Unity 是採取什麼方式來控制物件的生命週期。

本節所對應的書附範例：Ch07/Ex11.LifetimeDemo.csproj。

預設的生命週期

預設情況下——亦即沒有明白指定時——Unity 容器採取的生命週期模式會依註冊時呼叫的方法而不同：

- 若是呼叫 RegisterType 方法來註冊型別，則每次解析時（呼叫 Resolve 或 ResolveAll 方法時）都會建立新的物件，且 Unity 容器不負責釋放那些物件。這種生命週期模式一般稱為 **Transient**（「臨時」之意）。
- 若是呼叫 RegisterInstance 方法來註冊既有物件，則預設的物件生命週期等同於 **Singleton** 模式，也就是每次解析時，Unity 都會傳回同一個物件。

實際上，Unity 提供了 TransientLifetimeManager 和 ContainerControlledLifetimeManager 類別來分別實作 **Transient** 和 **Singleton** 這兩種生命週期模式。相關細節稍後再談，且先以一個簡單的範例來測試各種生命週期模式。程式碼如下：

```
interface ILifetimeTest : IDisposable
{
    // 空介面，不定義任何方法。注意：此介面繼承自 IDisposable。
}

class LifetimeTest : ILifetimeTest
{
    public static int ObjectCounter { get; private set; }

    public static void ResetCounters()
    {
        ObjectCounter = 0;
    }

    public static void PrintCounter()
    {
        Console.WriteLine("ObjectCounter={0}", ObjectCounter);
    }

    public LifetimeTest()
    {
        LifetimeTest.ObjectCounter++;
    }

    public void Dispose()
    {
        LifetimeTest.ObjectCounter--;
    }
}
```

說明：

- 這個 `LifetimeTest` 類別實作了 `ILifetimeTest` 和 `IDisposable` 介面，並且分別在建構函式和 `Dispose` 方法中遞增與遞減此類別的靜態屬性 `ObjectCounter`。
- 靜態方法 `PrintCounter` 是用來觀察目前總共創建了幾個物件。
- 靜態方法 `ResetCounter` 是用來清除物件計數。

以下範例可用來觀察 `RegisterInstance` 方法是否真的預設採用 **Singleton** 模式（即 `ContainerControlledLifetimeManager`）來控制物件的生命週期。

```
LifetimeTest.ResetCounter();
using (var container = new UnityContainer())
{
    var theObj = new LifetimeTest();
    container.RegisterInstance<ILifetimeTest>(theObj);

    var obj1 = container.Resolve<ILifetimeTest>();
    var obj2 = container.Resolve<ILifetimeTest>();
    System.Diagnostics.Debug.Assert(obj1 == obj2); // 不會出錯
    LifetimeTest.PrintCounters(); // 印出 ObjectCounter=1
}

// container 物件至此已成為可回收的資源，故其內部管理的物件也已經釋放。
LifetimeTest.PrintCounters(); // 印出 ObjectCounter=0
```

其中的 `Debug.Assert(obj1 == obj2)` 敘述不會出錯，是因為兩次呼叫 `Resolve` 的結果都是傳回同一個物件。觀察第一次呼叫 `LifetimeTest.PrintCounters()` 的輸出結果（物件總數為 1）也能得到相同結論。最後一行呼叫 `LifetimeTest.PrintCounters()` 的輸出結果顯示物件總數為 0，可確認 Unity 容器在本身釋放時，會一併釋放先前所建立的物件（呼叫它們的 `Dispose` 方法）。這是因為背後使用了 `ContainerControlledLifetimeManager` 來控制物件生命週期所產生的效果。

前面提過，`RegisterType` 方法所採用的預設生命週期模式為 **Transient**，即每次解析時一律建立新物件，且 Unity 容器不負責釋放物件。以下範例可驗證此說法。

```
LifetimeTest.ResetCounter();
using (var container = new UnityContainer())
{
    container.RegisterType<ILifetimeTest, LifetimeTest>();

    var obj1 = container.Resolve<ILifetimeTest>();
    var obj2 = container.Resolve<ILifetimeTest>();

    LifetimeTest.PrintCounters(); // 印出 ObjectCounter=2
}
LifetimeTest.PrintCounters();     // 印出 ObjectCounter=2
```

兩次 `PrintCounters()` 呼叫的輸出結果都是「ObjectCounter=2」，顯示 Unity 容器本身釋放時並沒有一併釋放它先前解析時所建立的兩個物件。

了解 Unity 預設的生命週期模式之後，接著要看的是如何明白指定生命週期。

指定生命週期

向 Unity 容器註冊型別時，可以傳入特定的生命週期管理員來明白指定要使用哪一種生命週期模式。以下是 Unity 提供的生命週期管理員：

- **TransientLifetimeManager**：每次解析時一律建立新的物件，且 Unity 容器不會保存該物件的參考（即容器不負責釋放該物件）。此控制模式一般稱為 **Transient**。
- **ContainerControlledLifetimeManager**：每次解析時都固定傳回同一個既有物件，並且在容器本身摧毀時一併釋放該物件。此生命週期模式的效果等同於 **Singleton** 模式。這也是呼叫 `RegisterInstance` 方法時的預設生命週期管理員。
- **PerResolveLifetimeManager**：類似 `TransientLifetimeManager`，主要差別在它在深層解析的過程中會重複使用同一個物件，而不會每次都建立新的物件。
- **PerThreadLifetimeManager**：基本上，此生命週期等於是執行緒範圍內的 **Singleton**。其行為與 `ContainerControlledLifetimeManager` 雷同，差別僅在於它會依執行緒來決定是否要建立新的物件。此生命週期管理員不負責釋放物件。

- **ExternallyControlledLifetimeManager**：與 `ContainerControlledLifetimeManager` 一樣具有 **Singleton** 的效果，但主要的差別在於，`ExternallyControlledLifetimeManager` 是保存物件的弱參考（weak reference）。如此一來，你就可以完全掌控釋放物件的時機，例如在用完物件時立刻呼叫它的 `Dispose` 方法來釋放該物件。顯然，此生命週期管理員不負責釋放物件。
- **HierarchicalLifetimeManager**：用於階層式容器，亦即容器本身又包含了子容器的場合。基本上，此生命週期的行為等同於 `ContainerControlledLifetimeManager` 再加上對子容器的額外控制：子容器有自己的 **Singleton** 範圍，不會跟父容器共享同一個物件。
- **PerRequestLifetimeManager**：此生命週期管理員只適用於 Web 應用程式。它能夠確保在同一個 HTTP 請求的範圍內解析某型別時總是傳回同一個物件。基本上，它就是一次 HTTP 請求範圍內的 **Singleton**。

先來看一個使用 `ContainerControlledLifetimeManager` 的範例：

```
LifetimeTest.ResetCounter();
using (var container = new UnityContainer())
{
    var lifeManager = new ContainerControlledLifetimeManager();

    // 註冊時一併指定此型別對應關係所欲使用的生命週期模式。
    container.RegisterType<ILifetimeTest, LifetimeTest>(lifeManager);

    var obj1 = container.Resolve<ILifetimeTest>();
    var obj2 = container.Resolve<ILifetimeTest>();
    LifetimeTest.PrintCounter(); // 印出 ObjectCounter=1
}
LifetimeTest.PrintCounter();     // 印出 ObjectCounter=0
```

此範例在呼叫 `RegisterType` 時傳入了 `ContainerControlledLifetimeManager` 物件，等於是為這一筆型別對應關係打上一個標籤，告訴容器：將來解析此型別時，請使用這個生命週期模式。因此，後來兩次呼叫 `Resolve` 方法所取得的物件都是同一個（**Singleton**）。而且，容器釋放時也會一併釋放它所建立的物件。

若把 ContainerControlledLifetimeManager 換成 ExternallyControlledLifetimeManager，則兩次呼叫 PrintCounter() 所顯示的物件數量都會是 1；這表示兩次解析傳回的是同一個物件，且容器不負責釋放物件。

如果換成 TransientLifetimeManager 或 PerResolveLifetimeManager，則兩次呼叫 PrintCounter() 所顯示的物件數量都會是 2；這表示容器每一次解析該型別時都會建立新物件，且容器不負責釋放物件。那麼，這兩種生命週期有何差別呢？請接著看下去。

Transient vs. Per-Resolve

TransientLifetimeManager 與 PerResolveLifetimeManager 的主要差異在於自動深層解析（參閱第 3 章）的過程中，碰到同一個型別需要解析多次時採取的策略。前者是無論如何都會建立新的物件，後者則是在該次解析的過程中對同一個型別只建立一個執行個體。

為了觀察兩者的差異，這裡延續先前的 LifetimeTest 範例，並且再增加兩個類別：

```
class Foo
{
    public Foo(ILifetimeTest lifetimeTest, Bar bar)
    {
    }
}

class Bar
{
    public Bar(ILifetimeTest lifetimeTest)
    {
    }
}
```

此例中，Foo 的建構函式需要注入 Bar 和 ILifetimeTest 物件，而 Bar 也需要注入 ILifetimeTest 物件。那麼，在解析 Foo 時，DI 容器便會執行一連串的深層解析。請看底下的範例：

```
LifetimeTest.ResetCounter();
using (var container = new UnityContainer())
{
    var lifeManager = new PerResolveLifetimeManager();
    container.RegisterType<ILifetimeTest, LifetimeTest>(lifeManager);

    var obj = container.Resolve<Foo>(); // 解析 Foo，發起一連串深層解析。
    LifetimeTest.PrintCounter();  // 印出 ObjectCounter=1
}
LifetimeTest.PrintCounter();      // 印出 ObjectCounter=1
```

說明：

- 註冊 ILifetimeTest 時，一併傳入 PerResolveLifetimeManager 物件。這表示將來容器在解析此型別時，會使用 **PerResolve** 生命週期。
- 解析 Foo 時，容器會呼叫其建構函式來建立執行個體，但此時發現建構函式需要注入 Bar 和 ILifetimeTest 物件，於是得先解析這兩個物件。然而解析 Bar 的時候，又再一次碰到需要解析 ILifetimeTest。在第二次碰到需要解析 ILifetimeTest 的時候，由於此型別在註冊時有指定使用 PerResolveLifetimeManager 來控制物件的生命週期，所以 Unity 容器不會另外建立新物件，而會重複使用第一次解析 ILifetimeTest 時所建立的物件。也正因為如此，Foo 解析完成之後的 PrintCounter() 印出的物件數量是 1。
- 在釋放物件方面，**PerResolve** 與 **Transient** 一樣，即容器不會去呼叫物件的 Dispose 方法。故第二次呼叫 PrintCounter() 印出的物件數量仍是 1。

你不妨試試將此範例的 PerResolveLifetimeManager 改成 TransientLifetimeManager，結果會變成兩次呼叫 PrintCounter() 所印出的物件數量皆為 2。這是因為在深層解析過程中，只要碰到有建構函式需要注入 ILifetimeTest，不管碰到幾次，容器都會建立新的物件。

PerResolve 只在當次解析的過程中維持某物件的唯一性，而 **ContainerControlled** 則會確保某物件在該容器的整個掌控範圍中必定只有一個執行個體。

Per-Request 生命週期

如先前提過的，`PerRequestLifetimeManager` 只適用於 ASP.NET 應用程式，而且你的專案必須引用「Unity bootstrapper for ASP.NET MVC」套件，才能使用此類別。此生命週期管理員能夠確保在同一個 HTTP 請求的範圍內解析某型別時總是傳回同一個物件。基本上，可以說是一次 HTTP 請求範圍內的 **Singleton**。

由於 `PerRequestLifetimeManager` 的用法涉及較多操作步驟，而這個部分已在第 6 章介紹過，這裡就不再重複。如需相關的範例和說明，請參閱本書第 6 章，標題為「在一個 HTTP 請求的範圍內共享同一個物件」的註記欄。

階層式容器

Unity 支援容器階層的概念，也就是容器裡面還包含其他子容器。子容器本身自成一個管轄範圍（稱為子範圍），在此範圍內，子容器不僅可以使用父容器的型別註冊資訊，同時又擁有自己專屬的型別對應關係（不會影響父容器），並且管理由它所建立之物件的生命週期。

針對階層式容器，Unity 提供了 `HierarchicalLifetimeManager` 來管理物件的生命週期。請看以下範例：

```
LifetimeTest.ResetCounter();
using (var parentContainer = new UnityContainer())
{
    var lifeManager = new HierarchicalLifetimeManager();
    parentContainer.RegisterType<ILifetimeTest, LifetimeTest>(lifeManager);

    // 建立子容器
    var childContainer = parentContainer.CreateChildContainer();

    var obj1 = parentContainer.Resolve<ILifetimeTest>(); // 使用父容器解析
    var obj2 = childContainer.Resolve<ILifetimeTest>();  // 使用子容器解析
    LifetimeTest.PrintCounter();  // 印出 ObjectCounter=2
}
LifetimeTest.PrintCounter();      // 印出 ObjectCounter=0
```

此範例值得注意的地方：

- childContainer 本身並沒有額外註冊型別對應關係，但依然能夠解析 ILifeTimeTest。這裡顯然使用了父容器的註冊資訊。
- parentContainer 和 childContainer 各執行了一次解析，而且分別建立了不同的執行個體。這點可以從 PrintCounter() 輸出的結果看得出來（物件總數為 2）。
- 子容器的生命與父容器綁在一起，故當父容器的生命結束時，它們會分別呼叫自身所管理之物件的 Dispose 方法來釋放物件。所以最後一次呼叫 PrintCounter() 的結果顯示物件總數為 0。

容器在進行解析時，會依先前註冊型別時指定的生命週期模式來決定要不要「管理這個物件」。上一段所說的「自身所管理之物件」，指的是容器在建立物件之時，一併記住了該物件的參考，以便將來可呼叫這些物件的 Dispose 方法。說得更精確一點，容器記住的物件參考必須是強參考（strong reference），才會由容器控制該物件的釋放時機；使用 ExternallyControlledLifetimeManager 的物件不在此列，因為對於使用此生命週期模式的物件，容器所保存的物件參考是弱參考（weak reference）。

註：當你指派一個物件給某變數時，該變數所持有的便是強參考。強／弱參考會影響 CLR 回收垃圾物件的時機，詳細用法可參考 MSDN 線上文件的 WeakReference 類別的說明。

選擇生命週期管理員

經過前面的文字說明與範例程式練習，相信讀者已經大致認識 Unity 內建的幾種物件生命週期管理員，以及它們之間的差異。這裡再將前述概念扼要整理成一張心智圖，以便協助判斷什麼情況選擇哪一個生命週期管理員比較合適。

選擇 Unity 的物件生命週期管理員

若此圖片在您的閱讀裝置上不夠清楚，可試試這個未經縮放的圖片連結：http://bit.ly/1wPPOHW。

攔截

有時候，為了不違反 **SRP**（單一責任原則）或 **OCP**（開放／封閉原則）或其他原因，我們會想要在不修改既有類別的前提下改變其行為，或安插額外的程式碼。比如說，呼叫某函式時，在真正執行函式的第一行程式碼之前寫一筆 log，函式返回時也寫一筆 log。**攔截（interception）**就是達成上述需求的一種方法。

有關攔截的基礎觀念以及相關的 **Decorator** 模式，本書在第 2 章與第 3 章已經提過。這裡要進一步說明的是 Unity 如何實現攔截機制，以及我們要怎麼樣來使用它。

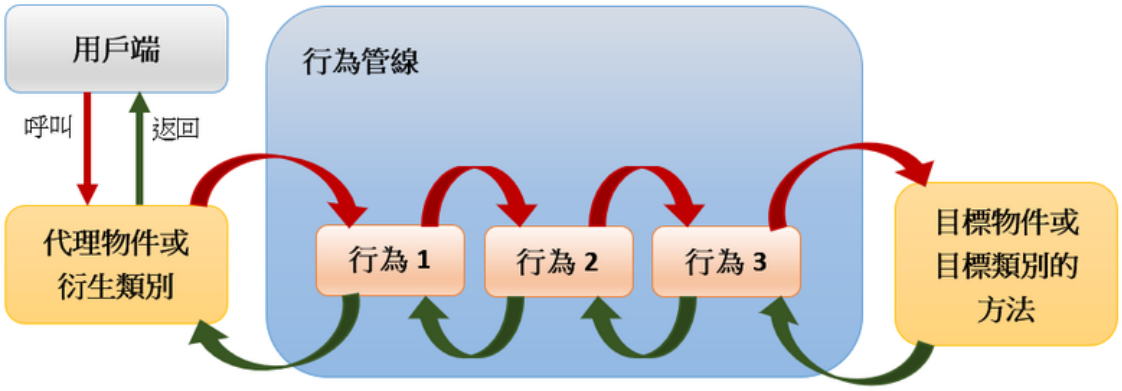
首先要知道的是，Unity 的攔截機制提供了兩種作法：

- **型別式攔截**（type-based interception）

  若選擇型別式攔截，你需要使用一個類別名為 `VirtualMethodInterceptor` 的攔截器（虛擬方法攔截器）。該攔截器會從你的目標類別衍生出一個新的子類別，而當 Unity 在解析物件時，便會建立那個子類別的執行個體（而不是目標類別）。
- **物件式攔截**（instance-based interception）

  若選擇物件式攔截，Unity 有兩個攔截器供你選擇：`InterfaceInterceptor`（介面攔截器）或 `TransparentProxyInterceptor`（通透代理攔截器）。前者只能攔截一個介面中的方法，後者則可以攔截目標物件的所有方法（包括介面方法、虛擬和非虛擬方法），但缺點是執行速度比 `InterfaceInterceptor` 慢很多（因為使用了 .NET Remoting 技術）。無論選擇哪一種物件攔截器，其運作方式都是為目標類別動態產生一個代理類別（proxy class），並在實際解析物件時建立該代理類別的執行個體。

下圖即描繪了 Unity 的攔截機制，其中的行為 1～3 代表我們自訂的攔截行為，而這些層層串接的行為便形成了所謂的「**行為管線**」（behaviors pipeline）。



Unity 的攔截機制

接著就來用一個範例來說明如何利用 Unity 容器的攔截機制來攔截物件的方法。

InterfaceInterceptor 範例

本節範例：Ch07/Ex12.InterceptionDemo.csproj。

這裡用本章前面提過的共用類別 `EmailService` 來當作攔截的目標。方便閱讀起見，這裡再貼一次相關的程式碼：

```
public interface IMessageService : IDisposable
{
    void SendMessage(string to, string msg);
}

public class EmailService : IMessageService
{
    public void SendMessage(string to, string msg)
    {
        Console.WriteLine(" 透過 EmailMessageService 發送郵件給 {0}。", to);
    }
}
```

先來看一下，加入攔截機制之前的用戶端程式碼：

```
static void Main(string[] args)
{
    var container = new UnityContainer();
    container.RegisterType<IMessageService, EmailService>();

    var msgService = container.Resolve<IMessageService>();
    msgService.SendMessage("Michael", " 此範例尚未加入攔截機制。");
}
```

程式碼很簡單，裡面用到的註冊與解析方法，前面出現過很多次，應該不用多作解釋了。此範例的執行結果如下：

```
    透過 EmailMessageService 發送郵件給 Michael。
```

現在假設我們希望每當應用程式呼叫 `EmailService` 的 `SendMessage` 方法時，在進入與離開此方法的時候輸出 log 訊息，以便觀察該方法的使用頻率或執行效能。當然，前提是不能修改 `EmailService` 類別，而必須使用攔截機制來實現。接著來看實作步驟。

Step 1：加入 Unity 攔截套件

Unity 核心組件並未包含攔截功能，此擴充功能是放在一個名為「Unity.Interception」的套件裡。你可以在 Visual Studio 中使用 NuGet Package Manager 來為專案加入此套件，

或者在 Package Manager Console 視窗裡輸入以下命令（不包含前面的 PM> 提示字元）：

```
PM> Install-Package Unity.Interception 【Enter】
```

加入此套件之後，在你的程式中引用所需之命名空間，然後呼叫 Unity 容器的 AddExtension 或 AddNewExtension 方法來加入攔截功能。參考以下程式片段：

```
using Unity;
using Unity.Interception.ContainerIntegration;
...
IUnityContainer container = new UnityContainer();
container.AddNewExtension<Interception>(); // 為容器加入攔截功能。
```

其中的 Interception 類別隸屬於命名空間 Unity.Interception.ContainerIntegration，組件檔名是 Unity.Interception.dll。此類別繼承自 Unity.Extension.UnityContainerExtension，主要的用途是為 Unity 容器添加「攔截」功能。

Step 2：實作攔截行為

接著你要寫一個類別來提供攔截行為，也就是當攔截的動作發生時，你想要做什麼事情，就把它們寫在這個類別裡。

用來封裝攔截行為的類別必須實作 Unity 的 IInterceptionBehavior 介面。此介面定義了三個方法，如下所示：

```
// 命名空間：Unity.Interception.InterceptionBehaviors
public interface IInterceptionBehavior
{
    bool WillExecute { get; }
    IEnumerable<Type> GetRequiredInterfaces();
    IMethodReturn Invoke(IMethodInvocation input,
                         GetNextInterceptionBehaviorDelegate getNext);
}
```

其中的

- `WillExecute` 是個唯讀屬性，它必須傳回一個旗號，用來控制此攔截行為要不要執行。若傳回 `false`，則此攔截行為將不起作用。
- `GetRequiredInterfaces` 方法須傳回一個集合，該集合裡面可包含零至多個介面型別。如果你在此集合中加入特定介面，那麼等到攔截機制要建立代理類別（proxy class）時，便會讓該類別實作這些介面。

  註：這裡的攔截機制，背後其實會動態建立一個代理類別來來包住欲攔截之類別。
- `Invoke` 方法會在被攔截之物件的方法正被呼叫時觸發，真正的攔截行為就是寫在此方法中。引數 `input` 封裝了目前正要執行的方法，而引數 `getNext` 是個委派，用來呼叫下一個攔截行為——如此便可將多個攔截行為串在一起，成為一條**行為管線**。

以下類別即在示範如何實作攔截行為：

```
1  public class MyLoggerBehavior : IInterceptionBehavior
2  {
3      public IMethodReturn Invoke(
4              IMethodInvocation input,
5              GetNextInterceptionBehaviorDelegate getNext)
6      {
7          // 呼叫目標方法之前。
8          Console.WriteLine(" 正要執行此方法：{0}", input.MethodBase.Name);
9
10         // 執行管線中的下一個行為。
11         var result = getNext()(input, getNext);
12
13         // 呼叫目標方法之後。
14         Console.WriteLine(" 正要離開此方法：{0}", input.MethodBase.Name);
15
16         return result;
17     }
18
19     public IEnumerable<Type> GetRequiredInterfaces()
20     {
21         return new Type[] { };
22     }
23
24     public bool WillExecute
25     {
26         get { return true; }
27     }
28 }
```

注意其中的 getNext 委派方法呼叫（第 11 行），它會呼叫行為管線中的下一個行為的 Invoke 方法，如此層層進入，直到最內層的原始目標方法執行完畢之後，再逐層返回。如果你知道 ASP.NET MVC 或 Web API 的訊息處理管線模型[24]，應該會覺得它們的運作方式非常相似。

第 19 行的 GetRequiredInterfaces 方法傳回的是空集合，因為這裡並不需要替攔截的物件加上額外的介面。

[24] http://huan-lin.blogspot.com/2013/01/aspnet-web-api-message-handlers.html

Step 3：註冊攔截行為

最後一個步驟，就是應用程式在向容器註冊型別時一併設定攔截行為。程式碼如下：

```
1  static void Main(string[] args)
2  {
3      var container = new UnityContainer();
4
5      // 為容器加入攔截功能。
6      container.AddNewExtension<Interception>();
7
8      // 註冊型別時，一併設定攔截器。
9      container.RegisterType<IMessageService, EmailService>(
10         new Interceptor(new InterfaceInterceptor()),
11         new InterceptionBehavior(typeof(MyLoggerBehavior)));
12
13     // 解析並呼叫物件方法。
14     var msgService = container.Resolve<IMessageService>();
15     msgService.SendMessage("Michael", "Hello, 利用 Unity 攔截物件的方法!");
16 }
```

設定攔截器的動作是在註冊型別對應關係時一併完成（第 9 行），這裡呼叫的 `RegisterType` 擴充方法是這個版本：

```
public static IUnityContainer RegisterType<T>(
    this IUnityContainer container,
    params InjectionMember[] injectionMembers);
```

透過此方法，這裡傳入了兩個 `InjectionMember` 物件，一個是 `Interceptor`，另一個是 `InterceptionBehavior`。以下分別說明之：

- 第 10 行：建立 `Interceptor` 物件時，傳入一個 `InterfaceInterceptor`，表示我們要使用「介面攔截器」，也就是稍早提過的兩種物件式攔截器的其中之一。
- 第 11 行：建立 `InterceptionBehavior` 物件時，將我們的自訂攔截行為類別 `MyLoggerBehavior` 傳入其建構函式。如此一來，以後呼叫 Unity 的 `Resolve` 方法

來解析 IMessageService 時，我們自訂的攔截行為就會套用至對應的 EmailServer 物件。

程式的執行結果如下：

```
正要執行此方法：SendMessage
透過 EmailMessageService 發送郵件給 Michael。
正要離開此方法：SendMessage，時間：9/23/2018 5:49:59 AM
```

本章重點回顧

DI 容器涵蓋的三大功能是型別的註冊與解析、物件生命週期管理，以及攔截（interception）。本章從 Unity 框架最基礎的 API 開始介紹註冊與解析的各種寫法，以及其他衍生的相關議題，包括：建構函式的自動匹配規則、自動註冊（可少寫一些程式碼）、如何注入建構函式的參數、延遲解析、以及自動工廠等等。

基本的 API 介紹完畢之後，接著是 Unity 支援的物件生命週期，包括 **Transient**、**ContainerControlled**、**PerResolve**、**ExternalControlled**、**PerThread**、**Hierarchical**、**PerRequest** 等等，並以範例程式來觀察各種生命週期管理員的差異。

最後是 Unity 提供的攔截功能。Unity 提供了三種攔截技術，並且分別封裝成三個類別：VirtualMethodInterceptor（虛擬方法攔截器）、InterfaceInterceptor（介面攔截器）和 TransparentProxyInterceptor（通透代理攔截器）。VirtualMethodInterceptor 是型別式攔截器，亦即以衍生類別的方式來實現攔截；InterfaceInterceptor 和 TransparentProxyInterceptor 則屬於物件式攔截器，亦即透過動態產生代理物件的方式來實現攔截。

如本章開頭所說，這不是完整的 Unity API 參考手冊，而比較像是（至少我希望它是）學習 Unity 框架的入門磚。我想，只要實際練習過本章的範例，掌握其中介紹的幾個常用 API，大致上應該能夠解決日常開發所碰到的 DI 相關問題。

附錄一：DI 容器實務建議

此附錄整理了一些有關使用 DI 容器的一些建議事項，主要的參考資料來源是 Jimmy Board 的文章：Container Usage Guidelines[25]。

容器設定

避免對同一個組件（DLL）重複掃描兩次或更多次

掃描組件的目的是為了自動註冊型別對應關係，故其過程涉及了探索組件內含之型別資訊。依應用程式所包含的組件與型別數量而定，掃描組件與探索型別的動作可能在毫秒內完成，亦可能需要花費數十秒。因此，當你在應用程式中使用 DI 容器的自動掃描功能來註冊型別時，應注意避免對同一個組件重複掃描兩次以上，以免拖慢應用程式的執行效能（通常是影響應用程式啟動的時間，因為註冊型別的動作大多集中寫在 **Composition Root**）。

為了能夠自動找到型別對應關係，型別的命名通常會遵循特定慣例，比如說，在掃描組件過程中，自動把 `Foo` 註冊成為 `IFoo` 的實作類別。

本書第 7 章的〈自動註冊〉一節有介紹自動註冊的基本用法。

[25] http://lostechies.com/jimmybogard/2014/09/17/container-usage-guidelines/

使用不同類別來註冊不同用途的元件

例如，你可能有一個 `WebApiConfig` 類別負責註冊 ASP.NET Web API 相關型別，以及用一個 `DalConfig` 類別來註冊資料存取層（Data Access Layer）的相關型別。然後，在應用程式的組合根（Composition Root）呼叫這些類別的註冊型別方法。一種常見的寫法是把組合根放在一個叫做 `Bootstrapper` 的類別裡，像這樣：

```
public static class Bootstrapper
{
    private static readonly IUnityContainer _container = new UnityContainer();

    public static void RegisterDependencies()
    {
        WebApiConfig.RegisterTypes(_container);
        DalConfig.RegisterTypes(_container);
        BllConfig.RegisterTypes(_container);
    }
}
```

此範例使用了一個靜態（`static`）類別 `Bootstrapper` 來總管容器物件的建立與型別註冊的工作，而此作法對於某些需要更大彈性的場合來說可能不適用，此時可參考下一個建議。

使用非靜態類別來建立與設定 DI 容器

有時候，應用程式可能需要在不同時機建立多個 DI 容器，而這些 DI 容器的生命週期可能不完全相同。若應用程式只有一個全域共享的靜態 DI 容器，便無法滿足上述需求。提供非靜態的 API，例如提供 instance 層級的方法和屬性（而不是 class 層級的 static 方法和屬性），可提供用戶端程式更多彈性，同時也更方便測試時抽換特定元件。

不要另外建立一個 DLL 專案來集中處理相依關係的解析

DI 容器的設定應該寫在需要解析那些相依型別的應用程式裡面，而不要把它們集中放在一個單獨的 DLL 專案（例如 DependencyResolver.csproj）。

為個別組件加入一個初始化類別來設定相依關係

如果你的 DLL 組件會給多個專案共用，而且它依賴某些外部型別，此時最好為該組件提供一個初始化類別（如前面提過的 `Bootstrapper`）來設定相依關係。若該組件需要掃描其他組件來尋找相依型別，此動作也是寫在初始化類別裡。

掃描組件時，盡量避免指定組件名稱

組件名稱可能會變，所以在掃描組件時，最好避免指定特定名稱的組件。

生命週期管理

多數 DI 容器都有提供物件生命週期管理的功能。一般而言，應用程式比較常用的是 Transient（每次要求解析時都建立新的物件）、Singleton（全都共享同一個物件）、以及 Per-HTTP request（每一個 HTTP 請求範圍內共享同一個物件），而只在某些少數場合才需要用到特殊的自訂生命週期。

優先使用 DI 容器來管理物件的生命週期

若無特殊原因，最好使用 DI 容器來管理物件的生命週期，而不要自行撰寫物件工廠來管理相依物件的生命週期。這是因為，目前大家喊得出名號的 DI 容器都經過多年的實務考驗與多方測試，不僅在設計上考量得更全面，發生 bug 的機率也比自己寫的元件來得低。有時候，物件生滅的時機比較特殊，或者需要更細緻地管理物件的生命週期，這些情況則不妨自行撰寫物件工廠來管理。

考慮使用子容器來管理 Per-Request 類型的物件

對於 Per HTTP Request 或類似的場合，一種常見的作法是把物件保存在當前的 `HttpContext` 物件裡，例如在 HTTP Request 建立之初便一併建立相依物件並將它們保存於 `HttpContext.Current.Items` 集合裡，然後在 Request 結束前一併釋放這些相依物件。DI 容器提供了另一個選擇來處理這類需求：子容器（child container）。

所有透過子容器解析（建立）的物件，其壽命不會超過子容器。也就是說，子容器一旦消失，它所管理的物件亦隨之消失。基於此特性，我們可以利用子容器來管理如 Per HTTP Request 或其他類似性質的物件生命週期。

本書第 7 章的〈階層式容器〉一節有介紹子容器的用法。

在適當時機呼叫容器的 Dispose 方法

DI 容器通常有實作 `IDisposable` 介面，亦即提供了 `Dispose` 方法。當你呼叫某容器物件的 `Dispose` 方法來釋放該容器時，它會找出內部管理的所有物件，只要是支援 `IDisposable` 的物件，就呼叫它的 `Dispose` 方法，以確保相依物件的資源得以釋放。

元件設計相關建議

避免建立深層的巢狀物件

雖然我們偏好物件組合而不是類別繼承，但如果相依物件的巢狀關係太多且深，這樣的程式架構仍然不好維護。DI 容器的一個方便卻也危險之處，是它具備自動解析巢狀相依物件的能力。於是，我們甚至只要寫一行程式碼就能解析所有深層的相依物件。這的確減

輕了開發人員的負擔，但同時也隱藏了背後的複雜性，因而導致開發人員更容易忽略設計的缺陷：太多零碎的小介面，組合出非常複雜的物件圖。對此設計面的問題，DI 容器沒法幫忙，唯有靠開發人員自己多加注意。

考慮使用泛型來封裝抽象概念

當你發覺整個程式架構太過笨重，可試著從現有的元件中找出同性質的功能，並為它們定義基礎的泛型介面，例如 `ICommand<T>`、`IValidator<T>`、`IRequestHandler<TRequest, TResponse>` 等等。

考慮使用 Adapter 或 Façade 來封裝 3rd-party 元件

開發應用程式時，難免會使用一些現成的外來（third-party）元件。有時候，外來元件可能會因為版本更新而造成既有應用程式無法運行或產生錯誤的結果。為了避免這種狀況，我們可以利用 Adapter、Façade、若 Proxy 等模式來為自己的應用程式建立一個反腐敗層（anti-corruption layer）。

不要一律為每個元件定義一個介面

對 **S.O.L.I.D.** 設計原則的一個常見誤解是：每個元件都應該要有一個相應的介面。當元件本身確實具有某個抽象概念的意涵，那麼為它定義一個抽象介面是合理的；但如果它不具備抽象概念，就不要硬為它生出一個介面。此外，你也不應該因為在使用 DI 容器時想要一致透過介面來解析，而為特定元件定義介面。DI 容器可以直接解析具象型別（concrete type），故從技術上而言，直接使用具象型別並不是問題。

看一下應用程式中的元件，如果許多元件的類別名稱正好就是它所實作的介面名稱去掉 'I'（例如 `Foo` 與 `IFoo`），這可能是一個訊號：有些介面可能只是單純因為「每個元件都要有個介面」的想法而產生的。

對於同一層（layer）的元件，可依賴其具象型別

如果某元件與其所依賴的其他元件都位在應用程式的同一層（layer），而且沒有抽換元件實作的需要（例如單元測試），這種情況，可直接依賴具象型別無妨。這就好像在同一間辦公室裡面工作的同事，通常是直接當面溝通；除非有特殊原因，否則沒必要透過中間人傳話，或透過即時訊息軟體的方式溝通。

動態解析

雖然元件或服務的類型大多能夠在應用程式初始化的時候決定，但有時候還是需要依執行時期的特定條件來動態決定使用哪個類別。比如說，在 ASP.NET MVC 應用程式中，可能會需要根據每一次前端網頁發出請求的動作參數來決定該繫結哪一個 model 類別，而無法在建立 controller 時預先得知欲繫結的 model 類別。

盡量避免把 DI 容器直接當成 Service Locator 來使用

如需動態解析元件類型，比較偷懶的辦法是把 DI 容器當成全域共享的 **Service Locator** 來使用。像這樣：

```
MyApp.Container.Resolve<IValidationProvider>();
```

這種用法很方便，只要在程式的某處向 DI 容器預先註冊元件類型，之後在程式中的任何地方都可以透過 DI 容器來解析元件。正因為它方便，所以容易濫用，使得元件之間的相依關係變得複雜紊亂，增加程式碼閱讀與維護的困難。

因此，若有其他選項，例如 **Constructor Injection**、**Factory 模式**等，應優先考慮。若都沒有合適的辦法，最後才使用 **Service Locator**。

此外，如果應用程式框架本身已經有提供元件解析的機制，也應該優先採用。例如 ASP.NET MVC 和 Web API 提供的 `IDependencyResolver` 及其相關機制。

本書第 4 章與第 5 章都有介紹 `IDependencyResolver` 的用法。

考慮使用物件工廠或 `Func<T>` 來處理晚期繫結

有時候，必須等到程式執行時，依當下的某些變數值來動態決定該建立何種物件。例如：

```
class NotificationService
{
    private IMessageService _emailService; // 郵件服務
    private IMessageService _smsService;   // 簡訊服務

    public NotificationService(IMessageService emailService, IMessage smsService)
    {
        _emailService = emailService;
        _smsService = smsService;
    }

    public void Notify(string to, string msg)
    {
        if (當前某些變數值符合特定條件)
        {
            _emailService.SendMessage(to, msg);
        }
        else
        {
            _smsService.SendMessage(to, msg);
        }
    }
}
```

其中「當前某些變數值符合特定條件」所涉及的相關資訊如果不存在 `NotificationService` 類別裡面，則可以考慮將「建立物件」的工作委外，亦即由呼叫端或其他特定類別來負責

建立所需之物件。一種常見的委外方式是撰寫特定用途的物件工廠，把建立物件的邏輯包在工廠類別裡，例如：

```
class MessageServiceFactory : IMessageServiceFactory
{
    public IMessageService GetService()
    {
        if (當前某些變數值符合特定條件)
        {
            return new EmailService();
        }
        else
        {
            return new SmsService();
        }
    }
}

class NotificationService
{
    public NotificationService(IMessageServiceFactory msgServiceFactory)
    {
        _msgServiceFactory = msgServiceFactory;
    }

    public void Notify(string to, msg)
    {
        using (var msgService = _msgServiceFactory.GetService())
        {
            msgService.SendMessage(to, msg);
        }
    }
}
```

於是用戶端可以這麼寫：

```
    var notySvc = new NotificationService(new MessageServiceFactory());
    notySvc.Notify("Michael", "DI Example");
```

另一種可行的作法，是利用 `Func<T>` 來讓外界提供建立物件的函式。像這樣：

```
class NotificationService
{
    private Func<IMessageService> _msgServiceFactory;

    public NotificationService(Func<IMessageService> svcFactory)
    {
        _msgServiceFactory = svcFactory;
    }

    public void SendMessage(string to, string msg)
    {
        using (var msgService = _msgServiceFactory())
        {
            msgService.Send(to, msg);
        }
    }
}
```

如果已經了解上述兩種做法，還可以試著再進一步，使用 DI 框架提供的「自動工廠」（automatic factory）來達到相同目的。第 7 章的〈使用自動工廠〉一節有更詳細的介紹。

附錄二：初探 ASP.NET 5 的內建 DI 容器

在 ASP.NET 5（vNext）之前，亦即 MVC 4/5、Web API 2 的時代，MVC 與 Web API 框架彼此有非常相似的設計，卻是以不同的程式碼來實作。現在，ASP.NET 5 整合了 MVC、Web API、與 Web Pages 程式模型於單一框架，統稱為 MVC 6。

ASP.NET 5 的另一個亮點是內建 Dependency Injection 容器。在此之前的 MVC 與 Web API 框架對於 DI 的支援則相對薄弱，主角是 `IDependencyResolver` 介面。

ASP.NET 5 的內建 DI 容器可能已經能夠滿足大部分的 DI 基礎操作，這表示將來我們對其他 DI 框架（如 Unity、Autofac）的依賴程度可能會逐漸降低。

這裡就要來牛刀小試一下 ASP.NET 5 內建的 DI 容器。

需要的工具：

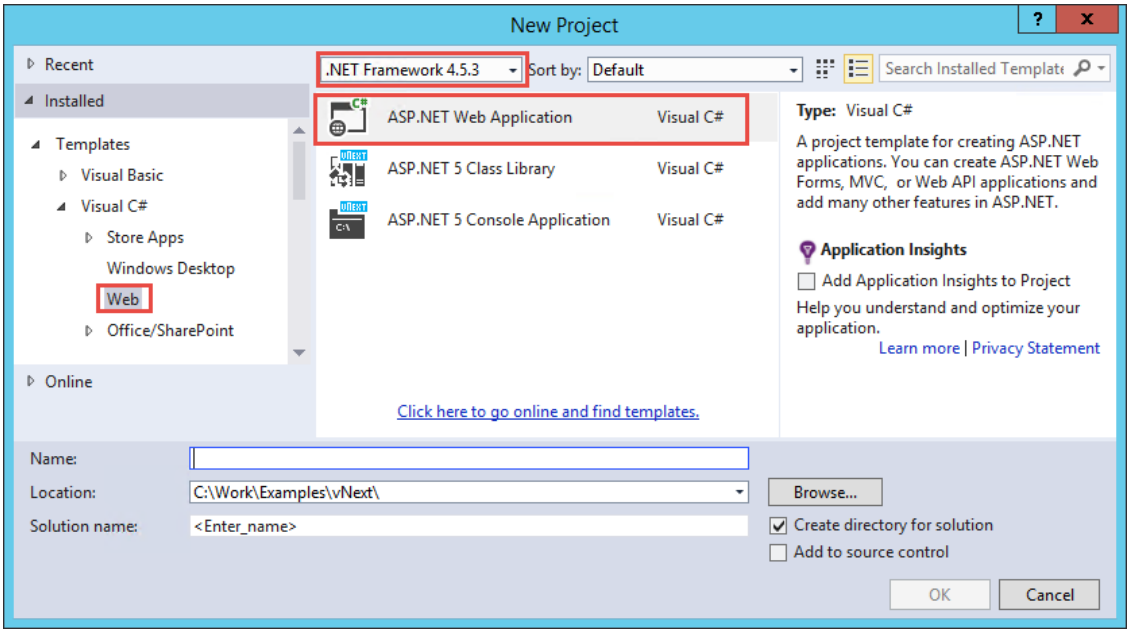
- Visual Studio 2015 Preview

撰寫本文時，ASP.NET 5 仍在 beta 階段，Visual Studio 2015 也是預覽版，所以本文的操作畫面截圖和程式範例可能會跟將來的正式版有些出入。

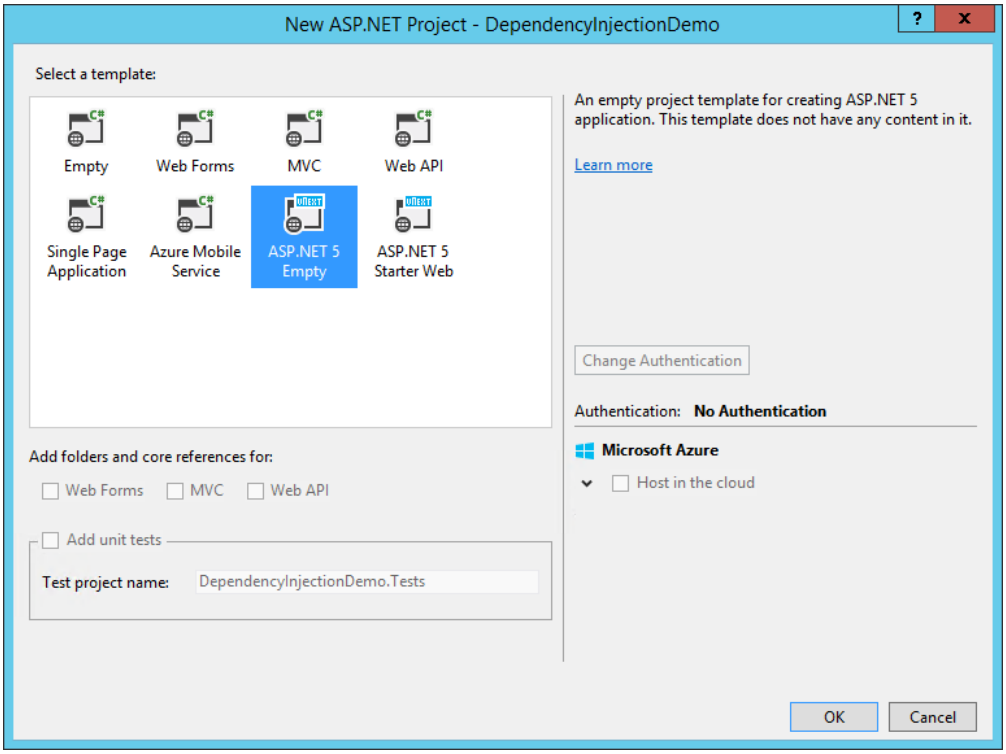
練習步驟

步驟 1：建立專案

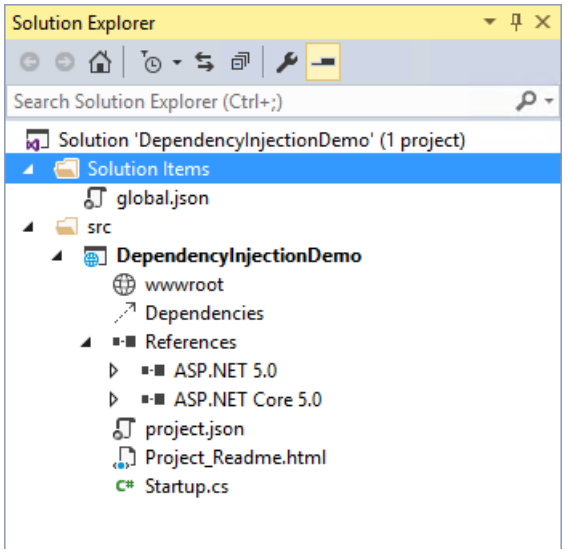
開啟 Visual Studio 2015，建立一個新的 ASP.NET Web Application 專案，參考下圖：



專案名稱命名為 DependencyInjectionDemo。按 OK 之後，接著選擇範本「ASP.NET 5 Empty」：



專案建立完成後，大概看一下 Solution Explorer 裡面有哪些東西：



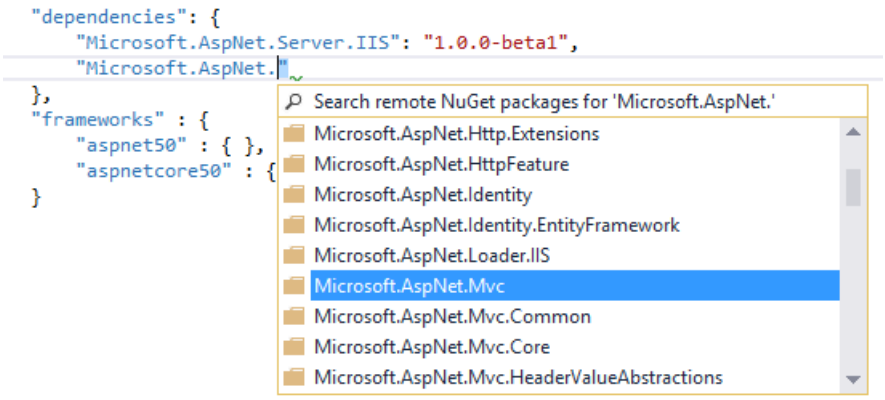
根目錄下的 project.json 即是此專案的設定檔，其中包含此專案所依賴的框架與元件。Startup.cs 則會包含應用程式啟動時所需執行的初始化工作。

步驟 2：加入必要組件

專案剛建立完成時的 project.json 內容如下：

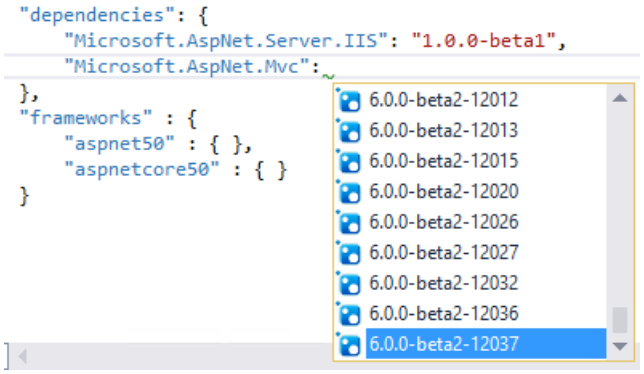
```
{
    "webroot": "wwwroot",
    "version": "1.0.0-*",
    "exclude": [
        "wwwroot"
    ],
    "packExclude": [
        "**.kproj",
        "**.user",
        "**.vspscc"
    ],
    "dependencies": {
        "Microsoft.AspNet.Server.IIS": "1.0.0-beta1",
    },
    "frameworks" : {
        "aspnet50" : { },
        "aspnetcore50" : { }
    }
}
```

我們得在「dependencies」區段中加入 ASP.NET MVC 組件：`"Microsoft.AspNet.Mvc": "6.0.0-beta1"`（你的開發環境可能是別的版本號）。這些文字都要自己敲進去，不過還好，Visual Studio 有智慧提示功能，如下圖：



若沒出現智慧提示，可按【Alt+ 右方向鍵】令它顯現。

輸入冒號之後，會接著提示版本，如下圖：



雖然移到最底下就能選擇最新的 beta 版本，但它可不一定能在你目前的開發環境上順利運行。保險起見，還是選最上方的「6.0.0-beta1」。

修改完畢之後，「dependencies」區塊的內容會像這樣：

```
    "dependencies": {
        "Microsoft.AspNet.Server.IIS": "1.0.0-beta1",
        "Microsoft.AspNet.Mvc": "6.0.0-beta1"
    },
```

說明：

- **Microsoft.AspNet.Server.IIS** －由於我們要使用 IIS 來做為此應用程式的裝載平台，所以必須加入此套件。
- **Microsoft.AspNet.Mvc** －這是 MVC 與 Web API 的核心套件。

步驟 3：將 Web API 元件加入 ASP.NET 管線

開啟 Startup.cs，參考以下範例來修改程式碼：

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.Http;
using Microsoft.Framework.DependencyInjection; // 別忘了這個!
using Microsoft.AspNet.Hosting; // 別忘了這個!

namespace DependencyInjectionDemo
{
    public class Startup
    {
        public void Configure(IApplicationBuilder app)
        {
            app.UseMvc();
        }

        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc();
        }
    }
}
```

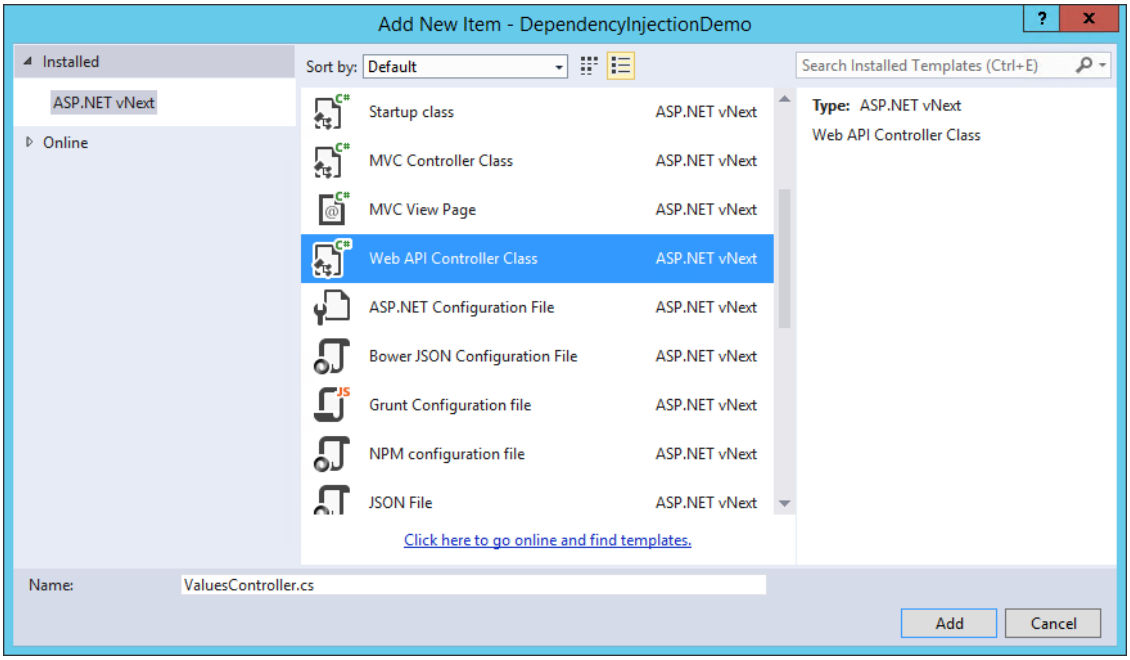
說明：

- `Configure` 方法有一個傳入參數 `app`，型別是 `IApplicationBuilder`。這裡使用它的 `UseMvc` 方法來將 MVC/Web API 元件加入至應用程式的管線（pipeline）。ASP.NET 框架會在應用程式啟動時主動呼叫此方法。
- `ConfigureServices` 方法也會由 ASP.NET 框架主動呼叫，我們可以在這裡設定 DI 容器，向 DI 容器註冊應用程式所需之服務。此方法的傳入參數 `services` 的型別是 `IServiceCollection`，它就像個 DI 容器，可以讓我們註冊相依服務。這個部分稍後會有範例程式。

`IServiceCollection` 介面隸屬於命名空間 `Microsoft.Framework.DependencyInjection`。

步驟 4：加入 API Controller

在 Solution Explorer 中，專案的根目錄下建立一個資料夾：Controllers。然後在此資料夾上點右鍵，選 Add ＞ New Item。在新開啟的對話窗中選擇「Web API Controller Class」，並將檔案命名為 ValuesController.cs。參考下圖：



產生的程式碼大致如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNet.Mvc;

namespace DependencyInjectionDemo.Controllers.Controllers
{
    [Route("api/[controller]")]
    public class ValuesController : Controller
    {
        // GET: api/values
        [HttpGet]
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // 省略其餘 Get/Post/Put/Delete 方法。
```

```
    }
}
```

OK! 現在按 F5 或 Ctrl+F5，看看應用程式能否正常運作。請注意，瀏覽器的網址列必須手動修改成這樣：

```
http://[主機名: 埠號]/api/Values
```

若沒出現錯誤訊息，便可繼續下一個步驟。

步驟 5：撰寫測試用的服務類別

寫一個簡單的類別來作為注入至 controller 的物件。如下所示：

```
namespace DependencyInjectionDemo
{
    public interface ITimeService
    {
        string Now { get; }
    }

    public class TimeService : ITimeService
    {
        public string Now
        {
            get
            {
                return DateTime.Now.ToString();
            }
        }
    }
}
```

程式碼很簡單，就不多解釋了。

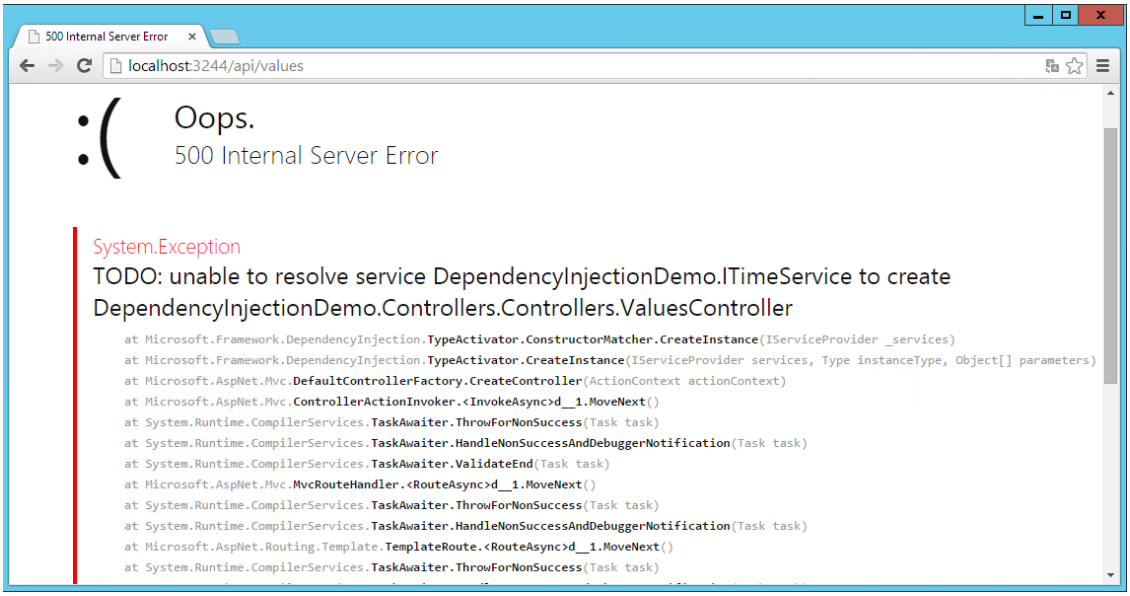
步驟 6：注入相依物件至 Controller 的建構函式

修改 ValuesController 類別，讓它看起來像這樣：

```
public class ValuesController : Controller
{
    private readonly ITimeService _timeService;

    public ValuesController(ITimeService timeService)
    {
        _timeService = timeService;
    }

    // GET: api/values
    [HttpGet]
    public IEnumerable<string> Get()
    {
        return new string[] { _timeService.Now };
    }

    // 省略其餘 Get/Post/Put/Delete 方法。
}
```

這表示我們希望 ASP.NET 框架在建立此 controller 物件時，能夠一併注入它需要的 `ITimeService` 物件。用 DI 術語來說，這裡使用了「建構式注入」（Constructor Injection）來避免我們的 API Controller 跟特定實作類別綁太緊——`ValuesController` 依賴的是抽象的 `ITimeService` 介面，而非具象類別 `TimeService`。

再執行一次應用程式看看，由於 MVC 框架找不到 `ValuesController` 的預設建構函式，瀏覽器應該會顯示錯誤訊息，如下圖：



解決方法很簡單，只要在你的 Startup 類別的 ConfigureServices 方法中註冊 ITimeService 型別所對應的實作類別就行了。

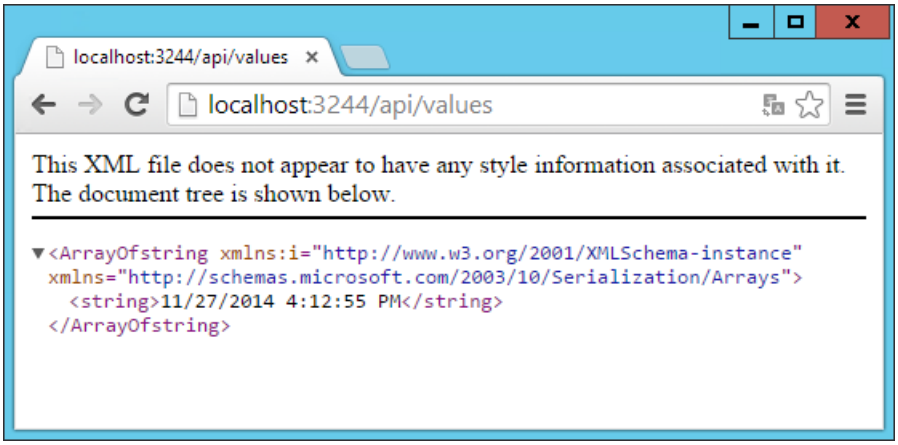
```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.UseMvc();
    }

    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();

        services.AddScoped<ITimeService, TimeService>(); // 加這行!
    }
}
```

這裡是透過 IServiceCollection 的擴充方法 AddScope 來向 ASP.NET 內建的容器註冊型別對應關係，意思是：碰到需要 ITimeService 物件的時候，使用 TimeService 來建

立物件實體。此外，從方法的名稱大約可以猜得出來，這個 AddScope 方法還有另一個意義，那就是：將來容器在建立 ITimeService 物件時（用 DI 術語來說，就是「解析 ITimeService」），會建立一個「活在特定範圍內」的物件。就此範例而言，這個特定範圍就是一個 HTTP 請求的範圍。

如此一來，當 MVC 框架在建立 ValuesController 時，發現它的建構函式需要一個 ITimeService 物件，於是 MVC 框架就會跟內建容器要一個 ITimeService 物件，然後將此物件傳入 ValuesController 的建構函式。

再執行一次應用程式，這次應該能夠順利執行了。執行結果如下圖：



很簡單吧？這裡完全沒用到第三方 DI 框架。

運用相同技巧，你可以將任何物件注入至你的 Controller 類別，例如應用程式層（application layer）的各類服務／元件。

結語

ASP.NET 5 的內建 DI 容器支援四種生命週期模式：**Instance**、**Singleton**、**Transient**、**Scoped**。本文範例中使用的 AddScoped 方法即為 **Scoped**，或者說 **Per Request** 生命週期模式，亦即物件只存活在目前請求的範圍內，且同一請求範圍內會共享同一個物件實體。

如欲進一步了解 ASP.NET 5 的 DI 功能與限制，可參考下列文章：

- Dependency Injection in ASP.NET vNext[26]
- Getting started with ASP.NET 5 MVC 6 Web API & Entity Framework 7[27]
- ASP vNext Dependency Injection Lifecycles[28]

The End

[26] http://blogs.msdn.com/b/webdev/archive/2014/06/17/dependency-injection-in-asp-net-vnext.aspx
[27] http://bitoftech.net/2014/11/18/getting-started-asp-net-5-mvc-6-web-api-entity-framework-7/
[28] http://www.khalidabuhakmeh.com/asp-vnext-dependency-injection-lifecycles

版權頁

.NET 相依性注入

副標：使用 Unity

檔案格式：EPUB、PDF、MOBI

ISBN：9789574320684
書籍定價：540
初版日期：2014-12-08
更新日期：2018-09-01

作者：蔡煥麟 (Michael Tsai)
出版：微步 (Ministep Books)
電郵：huanlin.tsai@gmail.com
網址：https://www.huanlintalk.com
地址：237 新北市三峽區大觀路 132 號 2 樓
電話：0936-681639