前言

C# 不斷的演進與改變。在這過程中，C# 的社群也在改變。現在更多的開發者視 C# 語言為他們首要的專業程式語言。我們社群的這些成員，並不會像一些在使用另一個以 C 為基礎的語言多年之後才開始使用 C# 的人一樣，有先入為主的成見。縱使是那些使用 C# 多年的開發者，近來的改變已帶來採用許多新習慣的需要。C# 語言自成為開源軟體以來革新的速度明顯是在加速中。現在檢視 C# 提議的功能之工作已納入整個社群，而不僅由一小群語言專家進行。整個社群也可以參與新功能的設計。

建議的架構改變以及配置也在改變 C# 開發者的語言慣用語法。以結合微服務（microservices）、分散式的程式，以及把資料與演算法分離等建立應用程式，是現代應用程式開發的所有要素。C# 語言也採納了擁抱這些慣用語法的步驟。

我在組織《*More Effective C#*》第二版時考慮了語言以及社群的改變。《*More Effective C#*》並不會帶領您回顧語言變革的歷史旅程，而是在如何使用現在的 C# 語言方面提出建議。從目前版本中移除的一些作法是在目前的 C# 語言或應用程式中不再適合的。新的做法包括新的語言與架構的功能，以及社群在使用 C# 建造數個版本的軟體產品時所學到的常規做法。早期版本的讀者會注意到《*Effective C#*》早期版本被納入目前版本的內容，同時大量的做法也已被移除。就目前的版本中，兩本書都已被我重新組織。整體而言，這 50 個做法是協助您作為一個專業開發者，能更有效使用 C# 的一組建議。

本書假設您是使用 C# 7，但這不是一個新語言功能的詳細解說。就像 Effective 軟體開發系列中所有的書一樣，這本書只是針對如何使用這些功能

去解決您每天可能遇到的問題，提出實務上的建議而已。本書所涵蓋的 C# 7 功能是因為這些新的語言功能，可以帶來新的與更好的方式撰寫常用的慣用語法。網路搜尋依然可以找到一些已使用多年的早期解決方案。本書會特別指出這些較舊的建議，並解釋為何語言強化的項目能帶來較好的方法。

本書對象

《*More Effective C#*》是為想以 C# 作為主要程式語言的專業開發者所寫。本書假設您熟悉 C# 語法及語言特色，並且是大致上精通 C#。本書在語言功能方面並沒有包含教學式的指示。取而代之的是本書討論如何把現行版本 C# 的所有功能整合到您的日常開發中。

除了熟悉 C# 語言特色之外，本書假設您對共同語言執行環境（Common Language Runtime，**CLR**）及 just-in-time（**JIT**）編譯器有些認識。

關於內容

在現今的世界，到處都有資料。物件導向的思維是資料及程式碼為型別的一部分，並且是型別的責任所在。函數式的思維則視方法為資料。而服務導向的思維則把資料與處理資料的程式碼分離。C# 已演進為包含所有這些規範的共同語言慣用語法。這使您設計時的選擇較從前複雜。第 1 章將會討論這些選擇，並對不同用途挑選不同語言慣用語法提供指導。

程式設計主要是 API 設計。你就是如此向使用者表達你希望他們該如何使用你的程式碼，同時這也清楚的表明你理解其他開發者的需求及期盼。在第 2 章中，您將學到使用 C# 語言豐富的功能作為表達您意圖的最佳方式。您將會看到如何運用惰性求值（lazy evaluation）、建立可合成介面（composable interfaces），以及避免在您的公開介面中不同語言元素之間的混淆。

以 Task 為基礎的非同步程式設計提供由非同步的結構單元合成應用程式的新慣用語法。專精這些功能代表您可以為非同步操作建立能清楚地反映出程式碼會如何執行並且容易使用的 API。在第 3 章中，您將學到如何使用以

task 為基礎的非同步語言功能表達您的程式碼，如何跨多個服務執行並且使用不同的資源。

第 4 章探討非同步程式設計其中之一的特定子集：多執行緒平行處理（multithreaded parallel execution）。您將會看到 PLINQ 如何允許我們把複雜的演算法輕易的分解至多個核心及多個 CPUs。

第 5 章討論使用 C# 作為一個動態語言（dynamic language）。C# 是一個有強型別、靜態型別的語言。但是現在越來越多的程式同時具有動態和靜態型別。C# 提供您運用動態程式設計慣用語法的方法，而不會使整個程式喪失靜態型別的好處。在第 5 章中，您將會學習到如何運用動態功能，並且避免動態型別在整個程式中帶來麻煩。

第 6 章以如何參與全球 C# 社群來結束本書。我們有很多種方式可以參與社群，並且協助塑造我們每日使用的語言。

程式碼規範

在書中展示程式碼需要在空間和清晰度做某種妥協。我試圖把範例精簡到只專注於範例的焦點所在。通常這意味著需要省略一個類別或方法的其他部分。有時候這表示為節省空間起見而省略錯誤回復的部分。公開的方法應該驗證其引數和其他輸入，但是程式碼一般是因為空間的考量而予以省略。同樣因為空間的緣故，方法呼叫的驗證以及 `try/finally` 等通常在複雜的演算法中會出現的語句也在此省略。

當範例中用到一些常見的命名空間時，我通常會假設大部分開發者可找到適當的命名空間。您可以很安全的假設每一個範例都會自動包含下列 `using` 敘述：

```
using System;
using static System.Console;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

1 處理資料型別

C# 最初設計應用於支援物件導向設計技巧，把資料和功能性結合在一起。隨著它越來越成熟，增加了新的慣用語法以支援日漸受歡迎的程式設計規範。其中之一的趨勢就是把資料的儲存與處理資料的方法分離。這種趨勢是受分散式系統的風潮運動所驅使，其中應用程式被分解為數個較小的服務，每一個服務只實作單一功能或一小組相關的功能。採用新的策略來分離關注點，自然就會帶來新的程式設計技巧。同理，使用新的程式設計技巧也會帶來新的語言特色。

在本章中，你將會學到分離資料與處理資料的技巧。這些資料不一定是物件，有時候它們是函式和被動資料容器（passive data containers）。

作法 01 使用屬性取代可存取的資料成員

屬性始終都是 C# 功能之一，但自 C# 語言問世以來，數次的改進已使屬性更有表達力。舉例來說，你可以針對 getter 與 setter 宣告不同的存取限制。自動實作的屬性（auto properties）減少定義屬性（而不是資料成員）時打字的量，包括唯讀屬性。運算式主體（expression-bodied）成員則啟用更精簡的語法。如果你仍在你的型別中定義 public 欄位，請停止。如果你還在手動建立 get 與 set 方法，請停止。屬性讓你揭露資料成員作為 public 介面的一部分，而仍然讓你保有物件導向環境中的封裝特性。屬性是在存取時如同資料成員的語言元素，但其實作卻像方法。

型別的一些成員很適合被表示為資料：如客戶的名字、一個點的位置 (x, y) 或去年的收入。

屬性允許你建立一個介面，在使用時就好像直接存取資料成員一樣，但仍然保有方法的好處。客戶端程式碼存取屬性時和存取 public 欄位是一樣的。實際上，屬性使用方法實作，定義屬性存取子（accessors）的行為。

.NET Framework 假設你會使用屬性作為你的 public 資料成員。事實上，.NET Framework 中的資料繫結類別支援屬性而不支援 public 資料欄位。這在所有的資料繫結程式庫中都是成立的：如 WPF、Windows Forms 與 Web Forms。資料繫結把一個物件的屬性連結到使用者介面的控制項。資料繫結的機制使用反映（reflection）找到一個型別中的具名屬性（named property）。

```
textBoxCity.DataBindings.Add("Text",
    address, nameof(City));
```

上述程式碼把 textBoxCity 控制項的 Text 屬性繫結到 address 物件的 City 屬性。上述程式碼如改用一個名稱為 City 的 public 資料欄則不會成功，因為 Framework 類別庫的設計人不支援這種用法。使用 public 資料成員被視為一種壞習慣，所以在 Framework 類別庫中並沒有加入對它們的支援。這一項省略給了你跟隨適當的物件導向技術的另一個理由。

是的，資料繫結只能用在那些包含要顯示在你的使用者介面（UI）邏輯中的元素的類別。但這並不表示屬性只能應用在 UI 邏輯：你在其他類別與結構中也應該使用屬性。隨著時間過去，當你發現新的需求或行為時，屬性是更易於更改的。舉例來說，你可能決定你的客戶型別應該永遠不允許一個空白的名稱。如果你使用一個 public 屬性作為 Name（名稱），這個需求很容易變更，因為要更改的全都位於同一處：

```
public class Customer
{
    private string name;
    public string Name
    {
        get => name;
        set
        {
            if (string.IsNullOrWhitespace(value))
                throw new ArgumentException(
                    "Name cannot be blank",
```

```
                    nameof(Name));
            name = value;
        }
        // 更多省略
    }
}
```

如果你使用 public 資料成員，則會被卡住而到處尋找設定客戶名稱的程式碼，並試圖在該處修正。這會浪費你更多的時間－非常多的時間。

因為屬性是以方法實作，加入對多執行緒的支援更加容易。你可以加強 get 與 set 存取子的實作以提供對資料（更多細節請見作法 39）的同步存取：

```
public class Customer
{
    private object syncHandle = new object();

    private string name;
    public string Name
    {
        get
        {
            lock (syncHandle)
                return name;
        }
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Name cannot be blank",
                    nameof(Name));
            lock (syncHandle)
                name = value;
        }
    }
    // 更多省略
}
```

屬性與方法有著相同的語言特色。特別是，屬性可以是 virtual：

```
public class Customer
{
```

```
    public virtual string Name
    {
        get;
        set;
    }
}
```

請注意先前幾個例子用的都是隱含型屬性（implicit property）語法。建立一個屬性來包裝一個支援存放區（backing store）是一種常見的模式。通常屬性的 getters 或 setters 都不需要驗證邏輯。C# 語言支援經過簡化的隱含型屬性，以減少揭露一個簡單的欄位作為屬性時形式上必須要寫的程式碼。編譯器會為你建立一個 private 欄位（通常稱為一個支援存放區）並為 get 與 set 存取子實作明顯的邏輯。

你可以使用和隱含型屬性類似的語法，把屬性擴充為 abstract，定義屬性為介面定義的一部分。以下的例子將示範一個泛型介面中的屬性定義。雖然語法和隱含型屬性一致，但是這個介面的定義不包含任何實作。它定義了任何實作這個介面的型別所必須滿足的協定。

```
public interface INameValuePair<T>
{
    string Name { get; }

    T Value { get; set; }
}
```

屬性是完整的、一級的、延伸方法的語言元素，可用來存取或修改內部資料。任何你可以用成員函式做的事，都可以用屬性做。屬性同時可避免在欄位可能發生的重大缺陷：你不可以使用 `ref` 或 `out` 關鍵字把屬性傳遞給方法。

屬性的存取子是編譯到你的型別中的兩個不同方法。你可以在 C# 的屬性中為 get 和 set 指定不同的存取修飾詞。這個彈性使你對被揭露為屬性的資料成員在可見性方面有更大的操控權：

```
public class Customer
{
    public virtual string Name
    {
```

```
        get;
        protected set;
    }
    // 省略其餘實作
}
```

屬性語法可以延伸到簡單資料欄位以外。如果你的型別包含有索引的項目作為介面的一部分，你也可以使用索引子（indexers，也就是參數化的屬性）。這是建立一個屬性供傳回序列（sequence）中項目的好方法：

```
public int this[int index]
{
    get => theValues[index];
    set => theValues[index] = value;
}
     private int[] theValues = new int[100];

// 存取索引子：
int val = someObject[i];
```

索引子和單一項目屬性（single-item properties）有相同的語言支援：索引子在寫的時候是以方法實作，因此你可以在索引子內做任何驗證或計算。索引子可以是 virtual 或 abstract、可以在介面中宣告，也可以是唯讀或可讀寫。單一維度使用數值引數的索引子可參與資料繫結。其他的索引子可使用非整數型引數定義對應：

```
public Address this[string name]
{
    get => addressValues[name];
    set => addressValues[name] = value;
}
private Dictionary<string, Address> addressValues;
```

在 C# 中維護多維度陣列時，你可以建立多維度索引子，在每個維度使用相似或不同的型別：

```
public int this[int x, int y]
    => ComputeValue(x, y);

public int this[int x, string name]
    => ComputeValue(x, name);
```

請注意所有的索引子都是用 this 關鍵字宣告。在 C# 中你不可以命名索引子，因此在一個型別中每一個索引子必須有一個不同的引數列以免混淆。幾乎屬性中所有的功能在索引子都有：索引子可以是 virtual 或 abstract；索引子的 setters 與 getters 可以有不同的存取限制。但是有一個差異，即不可以像屬性一樣定義隱含型索引子。

以上屬性的功能非常完善與好用，與先前 C# 版本相比是好的改進。縱使如此，你可能依然試圖先以資料成員開始實作，然後當你需要使用上述優點才開始使用屬性替換資料成員。這聽起來像是一個合理的策略－但這是錯誤的。請參考類別定義以下所列的部分：

```
// 使用 public 資料成員，不良習慣：
public class Customer
{
    public string Name;

    // 省略其餘實作
}
```

這個類別定義描述了一個有名字的客戶。你可以用熟悉的成員記號 get 或 set 名字：

```
string name = customerOne.Name;
customerOne.Name = "This Company, Inc.";
```

這很簡單且直接。你可以想像隨後你用一個屬性去替換 Name 資料成員，而且其他程式不需更改依然可運作。但這只有部分是真的。屬性在被存取時本來就是要像資料成員一樣；這本來就是語法的目的。但是屬性不是資料；一個屬性的存取和資料成員的存取會產生不同的 Microsoft 中繼語言（Microsoft Intermediate Language，MSIL）指令。

雖然屬性和資料成員的資料來源是相容的，但是它們的機器碼是不相容的。顯而易見的是這個限制代表你由一個 public 資料成員改為一個相應的 public 屬性時，必須重新編譯原先使用 public 資料成員的程式碼。C# 視二進位的 assemblies 為一級成員。語言的其中一個目標是允許你釋出一個單一 assembly 的更新而不需要更新整個應用程式。由一個資料成員改為一個屬

性的簡單動作打破了機器碼的相容性，使得更新一個已經部署的 assembly 更為困難。

當我們看到一個屬性的 MSIL 指令碼時，可能會想到屬性和資料成員相對的效能比如何？屬性的效能將不會比存取資料成員快，但是也不會比較慢。just-in-time（JIT）編譯器會內嵌一些呼叫，其中包含屬性的存取子。當 JIT 編譯器內嵌屬性存取子時，資料成員和屬性的效能是一樣的。縱使屬性存取子沒有被內嵌時，實際的效能差異也只有一個函式呼叫，差異甚小。這個差異只在少部分情況下可測量到。

屬性是在呼叫的程式碼中可看到的方法，就像資料一樣，這一點可能為你使用者的想法中帶來一些期待。他們看到屬性的存取，就以為那是一個資料的存取。畢竟，表面上看起來是如此。你的屬性存取子就被期待滿足這些預期。get 存取子不應有任何可觀察到的副作用。相對的，set 存取子更新狀態，而使用者應該可以看到這些改變。

屬性存取子也會使你的使用者有效能上的期待。屬性的存取和資料欄位的存取相像。屬性的存取在效能上的特徵不應該和簡單資料存取有太大的差異。屬性存取子不應該進行長的計算，或跨應用程式的呼叫（如進行資料庫查詢），或做其他較耗時的操作，以免和使用者對屬性存取子的預期不符。

每當你要透過型別的 public 或 protected 介面揭露資料，則使用屬性。針對序列或 dictionaries 則使用索引子。所有資料成員都是 private 而沒有例外。有了這些選擇，你立即會取得資料繫結的支援，而且在未來更改方法的實作也會更容易。把任何的變數封裝在你的屬性內也只不過是在一天中會多打字一兩分鐘而已。相較之下，才發現你之後需要用屬性才能正確反應你的設計會需要數小時的工作。現在多花一點點時間，就能節省你未來的許多時間。

作法 02　可變動的資料優先使用隱藏屬性

在 C# 屬性語法上新增的部分可讓你使用屬性以更清楚的表達你設計的意圖。現代的 C# 語言也支援你在一段時間之後改變你的設計。只要開始使用屬性，你就開啟了許多未來的可能性。

當你加入可存取的資料到類別時，通常屬性存取子只是資料欄位的簡單包覆。如果是這個情況，你可以用隱含式屬性增加程式碼的可讀性：

```
public string Name { get; set; }
```

編譯器使用由編譯器產生的名稱建立支援欄位（backing field）。因為支援欄位的名稱是由編譯器產生的，縱使在類別中你也需要呼叫屬性存取子，而不是直接更動支援欄位。這不是一個問題：呼叫屬性存取子的工作是相同的，而且因為所產生的屬性存取子是一個單一的指派敘述，屬性存取子的呼叫很可能是內嵌的。隱含式屬性在執行期行為和存取支援欄位的執行期行為是一樣的，甚至在效能方面也是如此。

隱含式屬性支援相同的屬性存取修飾詞，和一般的屬性相同。你可以定義任何限制性更大的 set 存取子：

```
public string Name
{
    get;
    protected set;
}
// 或
public string Name
{
    get;
    internal set;
}
// 或
public string Name
{
    get;
    protected internal set;
}
// 或
```

```
public string Name
{
    get;
    private set;
}
// 或
// 只能在建構函式中設定：
public string Name { get; }
```

隱含式屬性和你在先前版本 C# 中使用一個支援欄位手動建立一個屬性的模式是相同的。使用隱含式屬性的好處是生產力更高了，而且你的類別可讀性更高。一個隱含式屬性宣告顯示其他人所讀到的程式和你想要寫的程式是完全相同的，不會因為檔案中有額外的資訊而隱藏了真正的含意。

當然，因為隱含式屬性產生的程式碼和一般的屬性相同，你也可以用隱含式屬性定義 virtual 屬性、override virtual 屬性，或實作一個介面中定義的屬性。

當你建立一個 virtual 的隱含式屬性時，衍生（derived）的類別無法存取編譯器產生的支援存放區。但是 overrides 可以存取基底屬性 get 與 set 方法，就如同它們可以存取任何其他 virtual 方法一般：

```
public class BaseType
{
    public virtual string Name
    {
        get;
        protected set;
    }
}

public class DerivedType : BaseType
{
    public override string Name
    {
        get => base.Name;
        protected set
        {
            if (!string.IsNullOrEmpty(value))
                base.Name = value;
        }
```

```
    }
}
```

使用隱含式屬性你可以得到兩個額外的好處。第一，當需要把隱含式屬性用具體的實作取代以達成資料驗證或其他動作的時候，你將會是針對你的類別做機器碼相容的改變。第二，你的驗證只會出現在一個位置。

在早期 C# 語言版本中，大部分開發者直接存取支援欄位來改變他們的類別。這種習慣在整個檔案中到處散佈攜帶有驗證與錯誤檢查的程式碼。每一個對隱含式屬性的支援欄位之變動都會呼叫（可能是 private 的）屬性存取子。你必須把隱含式屬性存取子改為明確的屬性存取子，然後在新的存取子中寫出所有的驗證邏輯。

```
// 原來版本
public class Person
{
    public string FirstName { get; set;}
    public string LastName { get; set; }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}

// 後來加入驗證部分
public class Person
{
    public Person(string firstName, string lastName)
    {
        // 運用屬性 setters 中的驗證：
        this.FirstName = firstName;
        this.LastName = lastName;
    }
    private string firstName;
    public string FirstName
    {
        get => firstName;
        set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "First name cannot be null or empty");
            firstName = value;
```

```
        }
    }

    private string lastName;
    public string LastName
    {
        get => lastName;
        private set
        {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException(
                    "Last name cannot be null or empty");
            lastName = value;
        }
    }
    public override string ToString() =>
        $"{FirstName} {LastName}";
}
```

當你使用隱含式屬性時，把所有的驗證碼建在同一個位置。如果你可以持續使用你的存取子而不是直接存取支援欄位，則所有的欄位驗證就可以集中在一個位置。

隱含式屬性有一個重要的限制：你不可以把隱含式屬性用在具有 `Serializable` attribute 的型別。儲存的檔案格式會和支援存放區中編譯器所產生的欄位名稱有關。該欄位名稱無法保證會保持不變，即表示任何時候只要你更動了類別，該欄位名稱都可能改變。

儘管有兩個限制，但隱含式屬性可節省開發時間、產出可讀性較佳程式碼，而且促進一種把所有欄位驗證集中於一處的開發風格。當你建立了更清晰的程式碼，就可能用更好的方法去維護它。