GOTOP

PDF

Effective C#

中文版

涵蓋 C# 6.0

寫出良好C#程式的50個具體做法 第三版

碁峰 www.gotop.com.tw

Bill Wagner 著・楊尊一 譯

Effective C# 中文版

寫出良好 C# 程式的 50 個具體做法

第三版

Bill Wagner 著

楊尊一 譯

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

獻給為所有事情提供靈感與支持的 *Marlene*。

Praise for Effective C#, Second Edition

一個好的 .NET 開發者必須對語言有深入的認識。Wagner 的書以理性討論與洞見提供讀者這樣的知識。無論是 C# 新手或有多年經驗，閱讀本書都會有收穫。

—*Jason Bock*，顧問，*Magenic*

如果跟我一樣，你一定會累積許多讓你成為專業開發者的 C# 語言精要。這一本書是最好的精要合輯。Bill 這一本書的表現超出我的預期。

—*Bill Craun*，顧問，*Ambassador Solutions*

此書是建構高效能與高擴展性應用程式的必讀項目。Bill 以罕見且了不起的能力，將複雜問題解析成易讀、易消化的內容。

—*Josh Holmes*，架構推銷員，*Microsoft*

Bill 又辦到了。這本書集合無價的 C# 開發者知識。每天學習一項，十五天，只要十五天，你就會成為更好的 C# 開發者。

—*Claudio Lassala*，開發者，*EPS Software/CODE Magazine*

C# 語言知識的源泉。Bill 指出 .NET 執行期在底層如何運行你的程式碼與如何撰寫更清楚與容易懂的程式。混合提示、技巧、與深度知識…每個 C# 開發者都應該拜讀。

—*Brian Noyes*，架構設計，*IDesign Inc*

每個 C# 開發者都應該拜讀這本書。它關於程式設計的內容是無價的。

—*Shawn Wildermuth*，*Microsoft MVP*（*C#*）兼作家兼講師

Bill Wagner 在本書提供如何使用 C# 語言重要功能的務實說明。他以淵博的知識與超凡的溝通技巧為你指引 C# 的新功能，讓你寫出更精確與容易維護的程式。

—*Charlie Calvert*，*Microsoft C#* 社群經理

目錄

前言

2016 年的 C# 社群與 2004 年 Effective C# 第一版發行時已大不相同。有更多的開發者使用 C#。C# 社群中有許多人現在以 C# 作為主要的語言，他們並沒有將不同語言的舊習慣帶到 C# 中。社群成員的經歷更為廣泛，從畢業生到具有數十年經驗的專家都在使用 C#。現在 C# 可在多種平台上執行。你可以使用 C# 語言建構多種平台上的伺服器應用程式、網站、桌面應用程式、與手機應用程式。

我在編排 Effective C# 第三版時考量了語言與 C# 社群的變化。Effective C# 並不討論語言的歷史，而是建議如何運用目前的 C# 語言。這一版刪除的做法與現在的 C# 語言或應用程式無關。新的做法涵蓋語言與架構的新功能，以及社群從使用 C# 建構軟體產品過程學到的經驗。之前版本的讀者會注意到這一版包含了 More Effective C# 的內容並刪除許多做法。我正在重新編排這兩本書，新版的 More Effective C# 會涵蓋其他概念。這五十個建議的做法可幫助你更有效的運用 C#。

本書預設為 C# 6.0，但並不是語言新功能的列舉清單。如其他 Effective Software Development Series 叢書一樣，本書提供務實的功能使用建議來解決你可能會遇到的問題。我會特別討論以 C# 6.0 語言新功能撰寫慣用語法的新方式。搜尋網路可能會找到以前的老做法，我會特別指出舊建議與語言如何使用更好的做法。

本書的許多建議可受 Roslyn Analyzer 與 CodeFixes 檢驗。其程式庫見 https://github.com/BillWagner/EffectiveCSharpAnalyzers。如果你有什麼想法或貢獻，請發出 issue 或 pull 請求。

本書對象

Effective C# 是為使用 C# 作為日常工具的專業開發者所寫。本書假設你熟悉 C# 語法與語言，內容不包括語言的教學。本書討論如何整合開發 C# 語言目前版本的功能。

除了語言功能外，我假設你具備 Common Language Runtime（CLR）與 Just-In-Time（JIT）編譯器的基本知識。

關於內容

你所寫的每個 C# 程式都會用到語言基本結構，第一章“C# 語言慣用語法”，討論這些你應該很熟悉的常用語法。它們是你建構的各種類型程式與實作的演算法的基本結構。

使用 managed 環境並不表示環境免除了你的責任，你還是必須操控環境以建構符合效能需求的正確程式。這不只是效能測試與效能調校。第二章“.NET 資源管理”，你在進行細部最佳化之前的設計慣例以讓你操控環境來達成目的。

泛型是 C# 2.0 起其他 C# 語言功能的基礎技術。第三章“使用泛型”討論以泛型取代 `System.Object` 與型別轉換，然後討論限制、泛型特化、做法限制、與向後相容性。你會學到讓你容易表達設計意圖的泛型技巧。

第四章“使用 LINQ”解釋 LINQ、查詢語法、與相關功能。你會看到使用擴充做法分離合約與實作的時機、如何有效的使用 C# 閉包，與如何設計不具名型別。你會學習編譯器如何對應查詢關鍵字與做法呼叫、如何分辨 delegate 和表示式樹（並在必要時相互轉換），與尋找數值結果時如何跳脫查詢。

第五章“例外的最佳做法”提供新式 C# 程式管理例外與錯誤的指南。你會學習到如何確保適當的回報錯誤與發生錯誤時保持程式狀態的一致與不變。你會學習如何提供給使用你的程式的開發者更好的除錯體驗。

編排慣例

在紙本書上顯示程式碼還是得在空間與清晰之間取捨。我嘗試萃取範例以展示特定要點，這通常會省略類別或做法的其他部分，有時會省略錯誤復原程式。公開的做法應該檢查參數與其他輸入，但為了節省空間也省略了。還有做法呼叫的檢驗與複雜演算法中的 `try/finally` 也同樣省略。

我通常會在範例中使用常見命名空間，並假設大部分開發者會找出適當的命名空間。你可以假設每個範例隱含下列 `using` 陳述：

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

意見提供

無論我與審稿人多麼的努力，內容與範例還是可能有錯。如果你認為找到了錯誤，請透過 bill@thebillwagner.com 或 @billwagner 與我聯絡。勘誤會放在 http://thebillwagner.com/Resources/EffectiveCS。本書的許多做法是與其他 C# 開發者討論後的成果。如果你有問題或建議，請與我聯絡。一般討論見 http://thebillwagner.com/blog。

致謝

我要感謝許多人對本書的貢獻。成為 C# 社群的一份子是一項殊榮。C# Insiders 通訊錄中的每個人（無論是否來自 Microsoft）都為這本書貢獻了想法。

我必須特別列出幾個直接幫助我的 C# 社群成員將想法轉換成可行的建議。這一版的許多新想法來自與 Jon Skeet、Dustin Campbell、Kevin Pilch-Bisson、Jared Parsons、Scott Allen，特別是 Mads Torgersen 的討論。

這一版的技術審稿團隊很棒。Jason Bock、Mark Michaelis，與 Eric Lippert 整理內文與範例以保證本書的品質。他們的審核徹底又完整，除此之外還有幫助我將主題解釋的更好的建議。

與 Addison-Wesley 的團隊合作如同夢幻。Trina Macdonald 是個非常棒的編輯、任務主管，與驅策力量。她非常依靠 Mark Renfro 與 Olivia Basegio，我也是。他們對本書從封面到封底做出完整的貢獻。Curt Johnson 完成了不起的技術內容行銷。無論你選擇什麼格式，Curt 都跟它們有關。

成為 Scott Meyers 的系列的一部分是個殊榮。他審核所有原稿並提供改善建議。他非常的仔細，雖然他的軟體經驗不包括 C#，但可以找出我沒有解釋清楚的部分。他的回饋一如往昔非常的有價值。

我的家人放棄相處時間讓我得以完成原稿。我的妻子 Marlene 犧牲許多時間讓我寫作或開發範例，沒有她的支持，我絕對無法完成任何一本書，就算完成了也不會滿意。

關於作者

Bill Wagner 是最重要的 C# 專家之一，也是 C# Standards Committee 的成員。他是 Humanitarian Toolbox 的總裁，擔任過 Microsoft Regional Director 與 .NET MVP 前後 11 年，最近被指派為 .NET Foundation Advisory Council 成員。他曾經在新創與大企業工作過，負責改善開發程序與團隊。目前在 Microsoft 的 .NET Core 內容團隊，開發 C# 語言與 .NET Core 教材。他具有 University of Illinois 的資工學士學位。

C# 語言慣用語法

1

沒有問題為何要改變？答案是你可以做得更好。你改變工具或語言是因為生產力更好。如果不改變你的習慣就不會知道有什麼收穫。這對與 C++ 或 Java 等語言有許多共通處的 C# 這種新語言更為困難。C# 也是大刮號語言（curly-braced language），這使得你很容易落入同一個語言家族的慣用語法中。這會妨礙你發揮 C# 的潛力。C# 語言從 2001 年首次商業發行以來便不斷的演進，現在的版本比原始版本與 C++ 或 Java 更不相同。如果你是從其他語言過來 C# 這邊，就必須學習 C# 的慣用語法以讓此語言為你所用而不是與你對抗。這一章討論你應該改變的習慣 - 與你應有的做法。

做法 01 偏好隱含型別的區域變數

C# 語言加入隱含型別區域變數以支援不具名型別。使用隱含型別區域的第二個原因是某些查詢產生的結果是個 `IQueryable<T>`，而其他產生的結果是 `IEnumerable<T>`。如果強制轉換 `IQueryable<T>` 的 collection 成 `IEnumerable<T>` 的 collection，你會失去 `IQueryProvider` 提供的加強功能（見做法 42）。使用 `var` 也能改善開發者對程式的理解。`Dictionary<int, Queue<string>>` 並不能幫助理解，但 `jobsQueuedByRegion` 這個變數名稱可以。

我偏好 `var` 並用它宣告區域變數，是因為我發現它讓開發者注意重要的部分（語意的意涵）而不是變數型別的特點。如果建構出沒有檢查型別的結構，編譯器還是會提出警告。變數的型別安全與開發者寫出完整的型別名稱不同。在很多情況下，`IQueryable` 與 `IEnumerable` 的不同對開發者而言沒有差別。但若你嘗試告訴編譯器它是什麼型別，你會發現弄錯型別可以改變行為（見做法 42）。有時候使用隱含型別變數比較好，因為編譯器選擇的型別比

你選的好。但有時候過度使用 var 只會降低程式的可讀性。更糟的是使用隱含型別變數會引發微妙的型別轉換 bug。

區域型別的推斷對 C# 的靜態型別沒有實際作用。為什麼？首先你必須認識區域型別推斷與動態型別不一樣。以 var 宣告的變數並非動態型別而是以等號右邊的型別隱含宣告。使用 var 時，你不是告訴編譯器你建構什麼型別；編譯器會幫你宣告型別。

讓我從可讀性的問題開始。很多時候，區域變數的型別可以從初始化陳述明顯的看出來：

```
var foo = new MyType();
```

任何一個可靠的開發者都可以從此宣告看出 foo 的型別。同樣的，大部分的 factory 方法也是清楚的：

```
var thing = AccountFactory.CreateSavingsAccount();
```

但在某些情況下，回傳的型別不一定能從方法名稱看出來：

```
var result = someObject.DoSomeWork(anotherParameter);
```

當然，這是極端的例子，我希望你的程式庫的命名能夠將回傳什麼東西表示得更好。就算是這個極端的例子，更好的變數名稱可以給開發者更好的提示：

```
var HighestSellingProduct = someObject
    .DoSomeWork(anotherParameter);
```

就算沒有型別資訊，大部分的開發者還是可以正確的假設其型別為 Product。

當然，視 DoSomeWork 實際的 signature 而定，HighestSellingProduct 不一定是 Product。它有可能是 Product 衍生的型別，甚或是 Product 實作的界面。編譯器會認為 HighestSellingProduct 是 DoSomeWork 的 signature 所表示的型別。

執行期型別是否為 Product 無所謂。當編譯期型別與執行期型別不同時，編譯器一定贏。除非你使用某種型別轉換否則輪不到你說話。

我們開始進入 var 引發的可讀性疑問的主題。使用 var 作為變數型別來儲存方法的回傳值，是一種讓閱讀程式的開發者混淆的方式。閱讀此程式的人會

假設一種型別。執行時，此人可能是對的。但編譯器沒有機會檢視該物件的執行期型別。編譯器檢視編譯期型別並根據這些宣告決定區域變數的型別。現在不一樣的部分是編譯器決定了變數的宣告型別。你自行宣告型別時，其他開發者可以看到所宣告的型別。相對的，當你使用 var 時，由編譯器決定型別，但開發者看不到。你撰寫程式的方式導致人與編譯器對型別產生不同的結論，這會引發維護問題與原本可避免的錯誤。

讓我們繼續檢視以內建數值型別宣告的變數由隱含型別區域引發的問題。內建數值型別間有無數的轉換。擴大的轉換，例如從 float 到 double，都是安全的。縮小的轉換，例如從 long 到 int，涉及精度的損失。透過明確的宣告所有數值變數的型別，你可以部分控制型別的使用並幫助編譯器對有危險的轉換發出警告。

以下列的程式來說：

```
var f = GetMagicNumber();
var total = 100 * f / 6;
Console.WriteLine(
    $"Declared Type: {total.GetType().Name}, Value: {total}");
```

total 會是多少？視 GetMagicNumber 回傳的型別而定。以下是不同的 GetMagicNumber 宣告產生的不同輸出：

```
Declared Type: Double, Value: 166.666666666667
Declared Type: Single, Value: 166.6667
Declared Type: Decimal, Value: 166.6666666666666666666666667
Declared Type: Int32, Value: 166
Declared Type: Int64, Value: 166
```

型別的差異是由編譯器推斷 f 的型別因而影響 totoal 的型別的方式導致。編譯器賦予 f 與 GetMagicNumber() 所宣告回傳型別相同的型別。由於計算 total 的常數為實字，編譯器會將這些實字轉換成 f 的型別，而計算是使用該型別的規則完成。不同型別的不同規則產生不同的檔案。

這對語言不構成問題。C# 編譯器分毫不差的執行你所要求的動作。透過區域型別推斷，你告訴編譯器它比你更清楚你的型別。它根據等號的右邊做出最佳的判斷。使用內建數值型別時必須非常小心。此外，由於各種數值型別有不同的精度，受影響的不只是可讀性，還有準確性。

當然，這不是因為使用 var 引發的問題。成因是閱讀程式並不能知道 GetMagicNumber() 回傳的型別以及用了哪些內建的轉換。從方法中移除 f 變數的宣告會發生同樣的問題：

```
var total = 100 * f / 6;
Console.WriteLine(
    $"Declared Type: {total.GetType().Name}, Value: {total}");
```

明確宣告 total 的型別也沒有影響：

```
double total = 100 * GetMagicNumber() / 6;
Console.WriteLine(
    $"Declared Type: {total.GetType().Name}, Value: {total}");
```

total 的型別是 double，但若 GetMagicNumber() 回傳整數則結果也會被四捨五入。

問題是開發者看不出來 GetMagicNumber() 回傳值的實際型別且無法輕易的判斷執行過什麼樣的數值轉換。

將同樣的程序以明確的宣告 GetMagicNumber() 回傳型別做比較，現在編譯器會告訴你是否你的假設錯誤。如果 GetMagicNumber() 回傳的型別與 f 宣告的型別間有個隱含的轉換，它是可以運作。例如 f 宣告為整數而 GetMagicNumber() 回傳 int。但若中間沒有隱含的轉換可用，你會收到編譯器錯誤。你必須改變你的假設。如此可給你機會檢視程式並理解會發生的轉換。

這個例子顯示區域變數型別推斷能讓維護程式的開發者難過。編譯器的運作方式相同，且它是執行型別檢查的工具。但開發者無法輕易的看到套用什麼規則與轉換。在這些情況下，區域型別推斷會阻礙檢視其中的型別。

但有時候編譯器在選取變數的最佳型別上比你更聰明。以下列從資料庫讀取客戶名稱開頭符合某些字串的程序為例：

```
public IEnumerable<string> FindCustomersStartingWith1(
string start)
{
    IEnumerable<string> q =
        from c in db.Customers
        select c.ContactName;
```

```
    var q2 = q.Where(s => s.StartsWith(start));
    return q2;
```

此程式有著嚴重的效能問題。定義客戶名稱的原始查詢被開發者宣告為 IEnumerable<string>。由於是對資料庫查詢，它實際上是 IQueryable<string>。但由於強宣告回傳值，你失去了這個資訊。Iqueryable<T> 繼承自 IEnumerable<T>，因此編譯器不會對這個指派提出警告。編譯查詢的第二個部分時，使用了 Enumerable.Where 而不是 Queryable.Where。此例中，編譯器可以正確的判定更好的型別（IQueryable<string>）而表示你強制使用的（IEnumerable<string>）。如果犯下這樣的錯誤而沒有可從 IQueryable<string> 進行的隱含轉換，則編譯器會給你錯誤。但由於 Iqueryable<T> 繼承自 IEnumerable<T>，編譯器可以進行轉換，所以你會害到自己。

第二個查詢沒有呼叫 Queryable.Where；相對的，它呼叫 Enumerable.Where。這對效能有很大的負面影響。做法 42 會學到以 IQueryable 組合多個查詢表示式樹成為可一起執行的單一操作，通常是對儲存資料的遠端伺服器進行。此例中，抽象的第二個部分（where 句子）視來源為 IEnumerable<string>。這個改變很重要，因為只有第一個部分是在遠端機器進行。整個客戶名稱清單會從來源回傳。第二個陳述（where 句子）在本機檢查整個客戶名稱清單並回傳符合搜尋字串的記錄。

與下列版本比較：

```
public IEnumerable<string> FindCustomersStartingWith
    (string start)
{
    var q =
        from c in db.Customers
        select c.ContactName;

    var q2 = q.Where(s => s.StartsWith(start));
    return q2;
}
```

現在 q 是個 IQueryable<string>。編譯器由查詢的來源推斷回傳型別。第二個陳述對查詢加上 Where 句子並保存新的、更完整的表示式樹。實際資料僅在呼叫方透過列舉結果時執行查詢取出。過濾查詢的表示式被傳到資料來源，結果序列只帶有與過濾條件相符的客戶名稱。網路流量減少且查詢更有效率。這是設計過的範例且你比較可能建構單一查詢，但實務上的查詢可能由多個方法組成。

此神奇的改變在於 q 被（編譯器）宣告為 IQueryable<string> 而非 IEnumerable<string>。擴充方法不能是 **virtual** 的，且 **dispatch** 不依靠物件的執行期型別決定。相對的，擴充方法是靜態方法，且編譯器根據編譯期型別而非執行期型別決定最適合的方法。此處沒有晚綁定。就算執行期型別帶有符合呼叫的實例成員，它們對編譯器是看不到的，因此不會是選項。

很重要的一點是任何擴充方法可以寫成檢查其參數的執行期型別。擴充方法可以根據執行期型別建構不同的實作。事實上，Enumerable.Reverse() 就是如此，在參數實作 IList<T> 或 ICollection<T> 時得以提升效能（見做法 3）。

身為開發者的你必須決定是否讓編譯器無聲無息的宣告變數的編譯期型別而傷害可讀性。如果他人閱讀時不能馬上看出含糊的區域變數的確實型別，最好明確的宣告其型別。但在許多例子中，程式能夠清楚的表達變數的語意資訊。在上面的例子中，你知道 q 是一系列客戶名稱（字串）。語意資訊在初始化陳述中很明顯。從查詢表示式初始化的變數通常是這樣。無論變數的語意資訊是否清楚，你都可以使用 var。回到我的第一個論點，初始化表示式不能清楚的對開發者顯示變數的語意資訊但明確的型別宣告可以表示該資訊時應該避免使用 var。

簡單說，除非開發者（包括以後的你）必須看到型別宣告才能理解程式，否則就使用 var 宣告區域變數。這個做法的標題是“偏好”而不是“總是”。我建議明確的宣告所有數值型別（int、float、double 與其他）而不要使用 var 宣告。其他東西就使用 var。多打幾個字 - 明確的宣告型別 - 不會提升型別安全或改善可讀性。如果挑錯宣告型別，你可能會造成編譯器本來能夠避免的低效率。

做法 02 偏好 readonly 而非 const

C# 有兩種不同的常數：編譯期常數與執行期常數。它們具有不同的行為，用錯了會有負面影響。你應該偏好執行期常數而非編譯期常數。編譯期常數稍快一點，但彈性較執行期常數差很多。編譯期常數留給效能關鍵且常數值絕不改變時使用。

以 readonly 關鍵字宣告執行期常數。編譯期常數以 const 關鍵字宣告：

```
// 編譯期常數：
public const int Millennium = 2000;

// 執行期常數：
public static readonly int ThisYear = 2004;
```

上面的程式顯示類別或 struct 範圍內的兩種常數。編譯期常數也可在方法中宣告。唯讀常數不能在方法範圍中宣告。

編譯期與執行期常數的行為差異在於如何存取。編譯期常數被物件程式碼中的常數的值取代。下列程式：

```
if (myDateTime.Year == Millennium)
```

編譯出的 Microsoft Intermediate Language（MSIL 或 IL）與下列寫法相同：

```
if (myDateTime.Year == 2000)
```

執行期常數則在執行時求值。產生出的 IL 在參考唯讀常數時參考的是該 readonly 變數而非其值。

這個差別對使用常數的類型的時機加上了一些限制。編譯期常數只能用於內建整數和浮點數型別、列舉或字串。它們是初始化程序中唯一能指派有意義的常數值的型別。這些原始型別是唯一能夠在編譯器產生的 IL 中被實字值取代的型別。下面的程式無法編譯。你不能使用 new 運算子初始化編譯期常數，就算被初始化的型別是值型別也一樣：

```
// 不能編譯，要改用 readonly：
private const DateTime classCreation = new
    DateTime(2000, 1, 1, 0, 0, 0);
```

編譯期常數受限於數字、字串與 `null`。唯讀值也是常數，因為在建構元執行後不能修改。但唯讀值不同處在於執行期指派。使用執行期常數更有彈性，其中一項是執行期常數可為任何型別。你必須在建構元中進行初始化，或使用初始化程序。你可以製作 `DateTime` 結構的 `readonly` 值；你不能以 `const` 建構 `DateTime` 值。

你可以對實例常數使用 `readonly` 值，讓一個類別型別的每個實例儲存不同的值。編譯期常數的定義是靜態的常數。

最重要的差別是 `readonly` 值在執行期解析。產生出的 IL 在參考唯讀常數時參考的是該 `readonly` 變數而非其值。此差異對維護有著長遠的影響。編譯期常數產生與在程式中使用數值常數相同的 IL，就算是跨不同的組件（assembly）也一樣：一個組件中的常數用於其他組件時還是會以值替換。

編譯期與執行期常數的求值方式影響執行期的相容性。假設你在一個稱為 Infrastructure 的組件中同時定義 `const` 與 `readonly` 欄位：

```
public class UsefulValues
{
    public static readonly int StartValue = 5;
    public const int EndValue = 10;
}
```

在另外一個組件參考這些值：

```
for (int i = UsefulValues.StartValue;
    i < UsefulValues.EndValue; i++)
    Console.WriteLine( “value is {0}” , i);
```

進行測試會看到下列明顯的輸出：

```
Value is 5
Value is 6
...
Value is 9
```

隨著時間過去，你修改後發行新版的 Infrastructure 組件：

```
public class UsefulValues
{
```

```
    public static readonly int StartValue = 105;
    public const int EndValue = 120;
}
```

你交付 Infrastructure 組件但沒有重新建構 Application 組件。你預期得到下列輸出：

```
Value is 105
Value is 106
...
Value is 119
```

事實上會完全沒有輸出。現在迴圈使用值 105 作為開始並以 10 作為結束條件。C# 編譯器將 const 值 10 放在 Application 組件中而不是參考 EndValue 儲存的值。比較宣告為 readonly 的 StartValue 值；它在執行期解析。因此，Application 組件會使用新值而無需重新編譯 Application 組件；只需安裝新版的 Infrastrcture 組件就可以改變使用該值的所有用戶的行為。更新公開常數的值應該視為改變界面。你必須重新編譯參考該常數的所有程式。更新唯讀常數的值是實作的改變；它與現有的用戶程式在二進位上相容。

另一方面，有時你確實需要值在編譯期固定下來。以計稅程式為例，它可能會有多個與計算有關的組件，而稅務規則可能隨時改變。這會產生一種狀況，組件可能因規則變化而在不同時間改變演算法。你想要每個類別回報依據規則適用日期進行修改的時間。使用編譯期常數來追蹤規則改版可確保每個演算法正確的回報最後更新時間。

有一個類別作為主控版本：

```
public class RevisionInfo
{
    public const string RevisionString = “1.1.R9”;
    public const string RevisionMessage = “Updated Fall 2015”;
}
```

任何執行不同計算的類別可以取得這個主控版本資訊：

```
public class ComputationEngine
{
    public string Revision = RevisionInfo.RevisionString;
```

```
public string RevisionMessage = RevisionInfo.
   RevisionMessage;

// 省略其他 API
```

重新建構會更新版本號碼至最新版。但若交付個別組件作為更新修補，新的修補會有新的版本，沒有更新的組件不會受到影響。

使用 `const` 替代 `readonly` 的最後一個優勢是效能：已知的常數值較存取 `readonly` 變數值可產生效率稍好的程式。然而效益有限且必須衡量彈性的損失。犧牲彈性前一定要記錄效能的差異（如果沒有偏好的工具，可以試試看 BenchmarkDotNet，https://github.com/PerfDotNet/BenchmarkDotNet）。

使用具名與選擇性參數時會遇到類似執行期與編譯期常數值的權衡。選擇性參數的預設值如同編譯期常數（以 `const` 宣告）的預設值一樣是放在呼叫方。如同使用 `readonly` 與 `const` 值，你會需要注意選擇性參數值的改變（見做法 10）。

必須在編譯期確定的值必須使用 `const`：屬性參數、`switch case` 標籤與 `enum` 定義，以及少數不會在版本間變化的值。其餘狀況則傾向以 `readonly` 常數提升彈性。

做法 02　偏好 is 或 as 運算子而非型別轉換

擁抱 C#，你就擁抱了靜態型別。這是件好事。強型別意味你預期編譯器找出你的程式中的型別不相符處。這也表示你的應用程式不太需要在執行期檢查型別。但有時執行期型別檢查是不可避免的。C# 有時候要撰寫以 `object` 作為參數的函式，因為架構如此定義了方法。你可能需要嘗試轉換物件成其他型別，例如類別或界面。你有兩個選擇：使用 `as` 運算子，或強制編譯器依照你的指示進行型別轉換。你還有防禦性的做法：以 `is` 測試轉換然後使用 `as` 或型別轉換。

正確的做法是在可以時使用 `as` 運算子，因為它比盲目轉換在執行期更安全且有效率。`as` 與 `is` 運算子不執行然後使用者定義的轉換。它們只能在執行期型別符合目標型別時進行；它們很少建構新的物件以滿足要求。（`as` 運算子會在轉換 boxed 值型別至 unboxed 可為空值型別時建構新型別）。

舉例來說，你撰寫需要撰寫將模糊物件轉換成 MyType 實例的程式。你可以這樣寫：

```
object o = Factory.GetObject();

// 第一版：
MyType t = o as MyType;

if (t != null)
{
    // t 是個 MyType
}
else
{
    // 回報失敗
}
```

或者這樣寫：

```
object o = Factory.GetObject();

// 第二版：
try
{
    MyType t;
    t = (MyType)o;
    // t 是個 MyType
}
catch (InvalidCastException)
{
    // 回報轉換失敗
}
```

你會同意第一版比較簡單且容易閱讀。它沒有 try/catch 句子，因此沒有這段程式與其成本。注意型別轉換版本除了捕捉例外之外還必須檢查 null。null 可使用型別轉換成為任何參考型別，但 as 運算子在用於 null 參考時回傳 null。因此型別轉換時必須檢查 null 與捕捉例外。使用 as 只需檢查回傳的參考是否為 null。

as 運算子與型別轉換最大的差別是如何處理使用者定義的轉換。as 與 is 運算子會檢查被轉換物件的執行期型別；除了必要的 **boxing**，它們不執行其他

操作。如果物件不是所要求的型別或繼承自所要求的型別，它們就失敗。另一方面，型別轉換可使用轉換運算子將物件轉換成所要求的型別。這包括任何內建的數值轉換。轉換 `long` 成 `short` 會損失資訊。

轉換使用者定義型別時會可能有相同的問題。以下面的型別為例：

```
public class SecondType
{
    private MyType _value;

    // 省略其他細節

    // 轉換運算子
    // 將 SecondType 轉換成 MyType，
    // 見做法 29
    public static implicit operator
    MyType(SecondType t)
    {
        return t._value;
    }
}
```

假設一個 `SecondType` 物件是第一個程式的 `Factory.GetObject()` 函式回傳的：

```
object o = Factory.GetObject();

// o 是個 SecondType
MyType t = o as MyType; // 失敗。o 不是個 MyType

if (t != null)
{
    // t 是個 MyType
}
else
{
    // 回報失敗
}

// 第二版：
try
{
    MyType t1;
```

```
    t1 = (MyType)o; // 失敗。o 不是個 MyType
    // t1 是個 MyType
}
catch (InvalidCastException)
{
    // 回報轉換失敗
}
```

兩個版本都失敗。但我說過型別轉換會執行使用者定義的轉換。你會認為型別轉換會成功。你是對的 - 如果你這樣想的話它應該成功。但它因為編譯器是根據物件 o 的編譯期型別來產生程式而失敗。編譯器不知道 o 的執行期型別；它視 o 為 object 的實例。編譯器認為沒有 object 至 MyType 的使用者定義轉換。它檢查 object 與 MyType 的定義。由於沒有任何的使用者定義轉換，編譯器產生程式來檢視 o 的執行期型別並檢查該型別是否為 MyType。由於 o 是個 SecondType 物件，所以失敗。編譯器沒有檢查 o 的執行期型別是否能轉換成 MyType 物件。

如果如下撰寫程式，你可以成功的將 SecondType 轉換成 MyType：

```
object o = Factory.GetObject();

// 第三版：
SecondType st = o as SecondType;
try
{
    MyType t;
    t = (MyType)st;
    // t 是個 MyType
}
catch (InvalidCastException)
{
    // 回報失敗
}
```

你不應該寫出這麼醜的程式。避免捕捉本可事先檢查的例外是比較好的程式設計方式。這顯現了一個常見的問題。雖然你不會寫出這樣的程式，你可以使用進行適當轉換的函式，以 object 為參數：

```
object o = Factory.GetObject();
DoStuffWithObject(o);
```

```
private static void DoStuffWithObject(object o)
{
    try
    {
        MyType t;
        t = (MyType)o; // 失敗。o 不是個 MyType
        // t 是個 MyType
    }
    catch (InvalidCastException)
    {
        // 回報轉換失敗
    }
}
```

記得使用者定義的轉換運算子只能操作物件的編譯期型別而非執行期型別，與是否存在 o 與 MyType 間的執行期型別轉換無關。編譯器不知道也不在乎。下面的陳述有不同的行為，視 st 的宣告型別而定：

```
t = (MyType)st;
```

下一個陳述回傳同樣的結果，無論 st 宣告什麼型別。因此你應該偏好 as 而不是型別轉換 - 它更為一致。事實上，若型別間沒有關係，但存在使用者定義的轉換運算子，下面的陳述會產生編譯器錯誤：

```
t = st as MyType;
```

現在你知道要盡可能的使用 as，讓我們討論何時不能用它。下面的程式段不能編譯：

```
object o = Factory.GetValue();
int i = o as int; // 不能編譯
```

這是因為 int 是值型別且不能為 null。如果 o 不是個整數，int 應該儲存什麼值到 i 中？你選擇的任何值都應該是有效的整數。因此不能使用 as。你或許會認為這樣只好使用拋出例外的型別轉換。相對的，使用 as 運算子來轉換可為空的型別，然後檢查是否為 null：

```
object o = Factory.GetValue();
var i = o as int?;
if (i != null)
    Console.WriteLine(i.Value);
```

這種技巧在 as 運算子左邊的運算元是個值型別或任何可為空值型別時可用。

現在你知道 is、as 與型別轉換的差別，你認為 foreach 迴圈使用什麼運算子？ foreach 迴圈可操作非泛型 IEnumerable 序列並讓型別轉換在迭代中進行。（你應該盡可能使用型別安全的泛型版本。非泛型版本因歷史因素而存在並用於支援晚綁定狀況。）

```
public void UseCollection(IEnumerable theCollection)
{
    foreach (MyType t in theCollection)
        t.DoStuff();
}
```

foreach 陳述使用轉換操作來執行物件至迴圈所用型別的轉換。foreach 陳述產生的程式約略等同下面手寫的版本：

```
public void UseCollectionV2(IEnumerable theCollection)
{
    IEnumerator it = theCollection.GetEnumerator();
    while (it.MoveNext())
    {
        MyType t = (MyType)it.Current;
        t.DoStuff();
    }
}
```

foreach 陳述需要型別轉換以支援值型別與參考型別。無論目標型別為何，透過選擇轉換運算子，foreach 陳述展現出相同的行為。但由於使用了轉換，foreach 迴圈能夠引發 InvalidCastException 例外。

由於 IEnumerator.Current 回傳沒有轉換運算子的 System.Object，沒有型別可供測試。一群 SecondType 物件因為轉換失敗而無法在前一個 UseCollection() 函式中使用。foreach 陳述（使用型別轉換）不檢查集合中的物件的執行期型別可用的型別轉換。它只檢查 System.Object 類別（IEnumerator.Current 回傳的型別）與迴圈變數的宣告型別（此例為 MyType）可用的轉換。

最後，有時你想要知道物件確實的型別而不只是目前型別是否可轉換成目標型別。is 運算子根據多形的規則：若 fido 是 dog（繼承自 Animal）則 fido is Animal 回傳 true。GetType() 方法取得物件的執行期型別。它是比 is 或 as 陳述更嚴格的測試。GetType() 回傳物件的型別並能與特定型別比較。

以下列函式為例：

```
public void UseCollectionV3(IEnumerable theCollection)
{
    foreach (MyType t in theCollection)
        t.DoStuff();
}
```

如果讓 NewType 類別繼承自 MyType，一群 NewType 物件在 UseCollection 函式中是可以運作的：

```
public class NewType : MyType
{
    // 省略內容
}
```

如果你想要撰寫所有 MyType 實例都沒問題的函式，這樣是可以的。若你想要撰寫只有 MyType 物件才可以的函式，你應該以確實的型別進行比較。此例你會在 foreach 迴圈中進行。最常見的執行期確實型別很重要的狀況是執行相等性測試。對其餘的比較，as 與 is 提供的 .isinst 比較在語意上是正確的。

.NET Base Class Library（BCL）帶有使用相同類型操作的序列元素轉換方法：Enumerable.Cast<T>() 轉換序列中支援傳統 IEnumerable 界面的每個元素：

```
IEnumerable collection = new List<int>()
    { 1,2,3,4,5,6,7,8,9,10};

var small = from int item in collection
            where item < 5
            select item;

var small2 = collection.Cast<int>().Where(item => item < 5).
    Select(n => n);
```

此查詢產生與上面最後一行程式一樣的方法呼叫。在這兩種方式中，Cast<T> 方法將序列中的每個項目轉換成目標型別。Enumerable.Cast<T> 方法使用舊式轉換而非 as 運算子。使用舊式轉換表示 Cast<T> 不必具有類別限制。使用 as 運算子會受限制，且相較於實作不同的 Cast<T> 方法，BCL 團隊選擇結構使用舊式型別轉換運算子的單一方法。這是你在你的程式中也應考量的權衡

交換。若你需要轉換的物件是個泛型型別參數，你必須衡量必要的類別限制與使用型別轉換運算子的不同行為。

還有，注意泛型的型別轉換不使用任何的轉換運算子。這變數對一系列的整數做 `Cast<double>()` 會失敗。在 C# 4.0 與之後的版本，型別系統可透過動態與執行期型別檢查來避開。有些方式可根據已知行為而非知道特定型別或界面的任何資訊來處理物件。

物件導向最佳做法要求你避開型別轉換，但有時候沒得閃避。如果你無法避免轉換，使用語言的 `as` 與 `is` 運算子以更清楚的表達你的意圖。不同的強制轉換方式有不同的規則。`is` 與 `as` 運算子通常是正確的語意，它們只有在受測物件是正確型別時才會成功。盡量使用這些陳述而非轉換運算子，後者有可能產生意圖外的副作用且結果出乎你的意料。

做法 04 以內插字串取代 string.Format()

開發者從開始撰寫程式起就在轉換電腦儲存的資訊成人類可讀的格式。在 C# 中，我們使用可追溯自 C 語言的 API 進行此工作。該是廢棄這些技術並擁抱 C# 6.0 引進的新字串內插功能的時候了。

C# 6.0 的新語法較傳統字串格式化語法有幾項優點。它產生更可讀的程式。它讓編譯器提供更好的靜態型別檢查，可降低出錯的幾率。它還提供豐富的語法供製作字串。

`String.Format()` 能用，但有幾個弱點會讓錯誤在產生字串內容時才顯現。所有的替換都根據格式化字串而定。編譯器不檢查參數的數量與字串中需要替換的引數是否相符。有錯誤時，執行程式會收到例外。

更討厭的是參數的數值記號與 `params` 陣列中的位置兩者的分離意味著很難檢查參數順序是否正確。你必須執行程式並小心的檢查產生出的內容才能確保行為正確。

這些問題都可以克服，但要花時間。你可以利用語言的功能使程式的撰寫更容易。這就是語言新的內插字串功能要做的事。

內插字串在前面有個“$”前綴。然後，相較於 {} 字元間的位置索引，你可以使用任何的 C# 表示式。如此可大幅的提升可讀性。放在格式化字串中的替換表示式很容易閱讀。其結果也容易檢驗。還有，因為表示式是放在格式化字串而非分離的陣列中，不會有數量不相符的問題。你很難在此字串中的錯誤位置加入錯誤的表示式。

這是有也不錯的語法簡化功能，字串內插真正的威力要等你見識此語言功能如何與程式設計實踐整合才會更明顯。

首先讓我們探索語法與用於替換字串的表示式的限制。

我很小心的使用“表示式”一詞來描述可用於替換字串的程式。你不能使用流程控制陳述（例如 `if/else` 或 `while`）作為替換字串。如果你需要某種流程控制，必須將它放在方法中並呼叫方法作為替換字串。

字串內插執行類似現在的 `string.Format()` 函式庫程式（如何處理多國語言見做法 5）。這包括必要時將變數轉換為字串。以下列的內插字串為例：

```
Console.WriteLine($"The value of pi is {Math.PI}");
```

字串內插產生的程式會呼叫參數為 `params` 陣列物件的格式化方法。`Math.PI` 屬性是個 `double`，是值型別。為強制轉換 `double` 成 `Object`，它會被 **boxed**。在經常執行的程式中或緊密的迴圈中，這對效能有重大的影響（見做法 9）。你應該使用表示式來轉換這些參數成字串，如此能避免 **box** 表示式中的值型別：

```
Console.WriteLine(
$"The value of pi is {Math.PI.ToString()}");
```

`ToString()` 回傳的文字可能不符合你的需要，因此你可能會想要對表示式做更多的設定以建構你想要的文字。這很容易：修改表示式以產生你想要的文字：

```
Console.WriteLine(
$"The value of pi is {Math.PI.ToString("F2")}");
```

產生你想要的文字包括其他的字串處理，或將表示式回傳的物件格式化。讓我們從簡單的標準格式化字串開始。這與使用內建格式化字串來建構你想要的輸出有關。在 {} 字元中加上一個“:”接著格式化字串。

```
Console.WriteLine($"The value of pi is {Math.PI:F2}");
```

精明的讀者會注意到“:”可能是條件表示式的一部分。這樣會產生一些衝突：C# 會找到“:”並假設它是格式化字串的開頭。下面的程式無法編譯：

```
Console.WriteLine(
  $” The value of pi is {round ? Math.PI.ToString() :
Math.PI.ToString( “F2” )}” );
```

要讓它通過編譯的方式很簡單。你只需強制編譯器認為你想要條件表示式而非格式化字串的開頭。將整個表示式放在括號中就可以獲得你想要的行為：

```
Console.WriteLine($@” The value of pi is {(round ?
    Math.PI.ToString() : Math.PI.ToString( “F2” ))}” );
```

加入語言的字串內插提供很多功能。字串內插中的表示式可以是任何有效的 C# 表示式。你已經看過變數與條件表示式，但這只是開始。你可以使用空合併或空傳播運算子來清楚的處理缺值：

```
Console.WriteLine(
   $” The customer’ s name is {c?.Name ??  “Name is missing” }” );
```

沒錯，內插表示式中套疊字串是可以的。你加入“{”與“}”字元之間的內容會被當做 C# 並作為表示式解析（只有格式化字串開頭的“:”是例外）。

這些都是好功能，當你深入檢視時，它是個大千世界。內插字串的表示式參數本身可帶有內插字串。有時這很有用。以下面顯示記錄資訊或找不到記錄的訊息的程式為例：

```
string result = default(string);
Console.WriteLine($@” Record is {(records.TryGetValue(index,out
result) ? result :
$” No record found at index {index}” )}” );
```

找不到記錄時，條件偽的句子使用另一個內插字串來回傳表示找不到的訊息。

你也可以使用 LINQ 查詢（見第四章）來建構用於內插字串的值。這些查詢本身可使用內插字串來格式化它的輸出：

```
var output = $@” The First five items are: {src.Take(5).Select(
n => $@” Item: {n.ToString()}” ).Aggregate(
(c, a) => $@” {c}{Environment.NewLine}{a}” )})” ;
```

上面的範例可能不會出現在正常程式中，但它展示內插字串與語言的良好整合。此功能甚至被整合進 ASP.NET MVC 的 Razor View 引擎中。這樣可以讓網頁應用程式的 HTML 輸出更容易。預設的 MVC 應用程式可展示 Razor View 中的字串內插如何運作。以下是控制目前登入使用者資訊顯示的例子：

```
<a asp-controller=”Manage” asp-action=”Index”
   title=”Manage”>Hello@User.GetUserName()!</a>
```

你在建構應用程式中的任何 HTML 網頁時可以使用相同的技術。這是更簡潔的表達你想要的輸出方式。

這些範例顯示字串內插功能的強大。它比傳統的字串格式化更方便。但要記住字串內插的結果還是字串。所有值都被替換，你得到一個字串。這對 SQL 命令字串的建構更重要。字串內插不會建構參數化的 SQL 查詢，它建構的是引入參數值的字串物件。使用字串內插建構 SQL 命令非常的危險。事實上，使用字串內插建構表示資料的字串供機器解析時必須非常小心。

將電腦內部儲存資訊的格式轉換成人類可讀的文字是程式設計長久以來的一部分。前述做法與許多新式語言都可以回溯到幾十年前的 C 語言。使用這些做法會導致許多潛在錯誤。新的字串內插功能比較容易正確使用。它更有威力，且容易利用新式開發中常見的技術。

做法 05　對文化特定字串偏好 FormattableString

在做法 4 中，你學到 C# 的新字串內插功能提供更容易建構結合變數值與一些格式化資訊的文字資訊的做法。你必須更深入此功能的運作方式以在需要設計不同文化與語言的程式時獲得同樣的好處。

語言設計團隊對這個問題有相當多的想法，其目標是建構支援產生任何文化的文字，同時容易撰寫產生單一文化結果的程式。平衡這兩項目標意味著要深入文化與字串內插的複雜性。

你目前為止用到的字串內插會讓你認為使用“$”語法建構字串會產生字串。這種簡化在大部分情況下可用。實際上，字串內插隱含轉換成字串或可格式化的字串。

舉例來說，下面的程式使用字串內插建構一個字串：

```
string first =
$"It's the {DateTime.Now.Day} of the {DateTime.Now.Month} month";
```

下面的程式使用字串內插來建構繼承自 FormattableString 的物件：

```
FormattableString second =
$"It's the {DateTime.Now.Day} of the {DateTime.Now.Month} month";
```

下面使用隱含型別區域變數的程式假設 third 應該是個字串並產生程式來建構字串：

```
var third =
$"It’s the {DateTime.Now.Day} of the {DateTime.Now.Month} month";
```

編譯器根據所請求的輸出編譯期型別產生不同的程式。產生字串的程式會根據目前執行程式的機器的文化來格式化該字串。如果在美國執行此程式，double 的小數點會是“.”；如果在大部分的歐洲國家執行相同的程式，小數點會是“,”。

你可以使用編譯器的功能來產生字串或 FormattableString 以使用字串內插來格式化任何文化的字串。以下列兩個使用特定語言與文化轉換 FormattableString 至字串的方法為例：

```
public static string ToGerman(FormattableString src)
{
    return string.Format(null,
      System.Globalization.CultureInfo.
         CreateSpecificCulture(“de-de”),
      src.Format,
      src.GetArguments());
}

public static string ToFrenchCanada(FormattableString src)
{
    return string.Format(null,
       System.Globalization.CultureInfo.
          CreateSpecificCulture(“de-CA”),
       src.Format,
       src.GetArguments());
}
```

這些方法以 FormattableString 作為唯一的參數。以 FormattableString 呼叫這些方法時，它們會使用特定文化（德國文化或加拿大法語）將 FormattableString 轉換成字串。你也可以直接對字串內插的結果呼叫這些方法。

首先，很重要的是不要以使用字串作為參數的類似版本過載這些方法。如果有過載可取用字串與 FormattableString，編譯器會產生程式來建構字串並呼叫以字串為參數的版本。

你還會注意到我沒有對上述方法建構擴充方法。這是因為編譯器中判斷應該建構字串或 FormattableString 的邏輯，會在結果是“.”運算子的左邊建構字串而非 FormattableString。字串內插的目標之一是讓它容易與現有的字串類別合作。但就算目標如此，開發團隊還是想要實現所有可能的全球化情境。這些情境需要額外的工作但很容易處理。

字串內插功能支援所有你所需的國際化與特定區域化。最棒的是它讓你在程式產生目前文化的文字時可忽略複雜性。當你需要特定文化時，你必須明確的告訴字串內插要建構 FormattableString，然後可以使用你指定的文化將它轉換成字串。

做法 06　避免字串型別 API

隨著我們朝向更為分散的程式，我們必須在不同的系統間傳輸更多的資料。我們還使用依靠名稱與字串識別符號操作資料的許多不同種函式庫。這些是操作不同平台與不同語言的資料的便利方法，但這種便利是有代價的。使用這些 API 與函式庫意味著損失型別安全性。你失去工具支援。你失去靜態型別語言的許多好處。

C# 語言設計團隊知道這些問題並在 C# 6.0 加上 nameof() 表示式。這個方便的關鍵字取代了名稱字串符號。最常見的例子是 INotifyPropertyChanged 界面的實作：

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

```
    {
        if (value != name)
        {
            name = value;
            PropertyChanged?.Invoke(this,
               new PropertyChangedEventArgs(nameof(Name)));
        }
    }
}
private string name;
```

使用 nameof 運算子時，任何對屬性名稱的改變都會正確的反映在用於事件參數的字串中。這是 nameof() 的基本運用。

nameof() 運算子對符號的名稱求值。它可用於型別、變數、界面、與命名空間。它可操作完整或不完整的名稱。對泛型型別定義有些限制：必須是明確列出所有參數的封閉泛型型別。

nameof 運算子必須操作這些項目且必須有相同的行為。nameof 運算子回傳的字串總是區域名稱。如此提供了一致的行為：就算變數是使用完整名稱宣告（例如 System.Int.MaxValue），回傳的還是區域名稱（MaxValue）。

許多開發者見過這種基本運用並正確的在必須使用區域變數名稱的 API 中使用 nameof。但許多開發者因為依循現有的習慣或不知道 nameof 運算子可用於不同地方而錯失良機。

許多例外型別接受參數名稱作為建構參數之一。以 nameof 取代寫死的文字可確保重新命名後名稱還是相符：

```
public static void ExceptionMessage(object thisCantBeNull)
{
    if (thisCantBeNull == null)
        throw new
            ArgumentNullException(nameof(thisCantBeNull),
             "We told you this cant be null");
}
```

靜態分析工具也可以確保參數名稱在 Exception 建構元中的正確位置。參數都是字串，所以很容易搞錯。

nameof 運算子可用於指定屬性參數的字串（位置或具名參數）。它可用於定義 MVC 應用程式或網路 API 應用程式的路徑。這是考慮使用命名空間作為路徑名稱的絕佳時機。

在這些位置使用 nameof 運算子的收穫是改變符號名稱會正確的反映在變數的名稱中。靜態分析工具也可以在參數名稱用於錯誤位置時發現錯誤與不符處。這些工具可接受多種形式，從編輯器或 IDE 中執行的診斷，到建構與 Continuous Integration 工具、重構工具等。經由盡可能地保存符號資訊，你可以讓這些自動化工具找出甚至改正各種錯誤。如此就只會剩下幾個只能透過自動化測試或人工檢驗找出的錯誤。這讓我們有更多的時間專注於更困難的問題。

做法 05　以 delegate 表示 callback

我：“兒子，去除草，我要讀書”。

兒子：“爹，我清理了草坪”。

兒子：“爹，我幫除草機加油了”。

兒子：“爹，除草機發動不了”。

我：“我來發動”。

兒子：“爹，我做完了”。

這段對話顯示出 callback。我交付任務給我兒子，他（反復的）打斷我以報告狀態。我在等待他完成任務的每個部分時並沒有停下來。他能夠在有重要（或不重要）的狀態或需要我的協助時週期性的中斷我。callback 用於從伺服器對用戶非同步的提供回饋。它們可能涉及多執行緒，或只是提供同步更新的進入點。callback 在 C# 語言中以 delegate 表示。

delegate 提供型別安全的 callback 定義。雖然大部分常見的 delegate 是用於事件，但不應該是唯一使用此功能的時機。任何時候你需要設定類別間的通訊並要求較界面更少的耦合時，delegate 是正確的選擇。delegate 讓你在執行期設定目標並通知多個用戶。delegate 是個帶有方法參考的物件。該方法

可以是靜態方法或方法實例。使用 **delegate** 時，你可以在執行期設定對一或多個用戶物件通訊。

callback 與 **delegate** 是很常見的用法，所以 **C#** 語言以 **lambda** 表示式的形式提供精簡的語法來表示 **delegate**。此外，**.NET Framework** 函式庫使用 `Predicate<T>`、`Action<>`、與 `Func<>` 定義許多常見的 **delegate**。**predicate** 是測試條件的 **Boolean** 函式。`Func<>` 取用數個參數並產生單一結果。沒錯，這表示 `Func<T, bool>` 與 `Predicate<T>` 有相同的形式。但編譯器不會視 `Func<T, bool>` 與 `Predicate<T>` 相同。一般來說，相同參數與回傳型別的不同的 **delegate** 型別定義是不同的型別。編譯器不會允許在它們之間轉換。最後，`Action<>` 取用任意數量的參數並具有 `void` 回傳型別。

LINQ 使用這些概念製作。`List<T>` 類別也帶有許多使用 **callback** 的方法。以下面這段程式為例：

```
List<int> numbers = Enumerable.Range(1, 200).ToList();

var oddNumbers = numbers.Find(n => n % 2 == 1);
var test = numbers.TrueForAll(n => n < 50);

numbers.RemoveAll(n => n % 2 == 0);

numbers.ForEach(item => Console.WriteLine(item));
```

`Find()` 方法以 `Predicate<int>` 形式的 **delegate** 對每個元素執行測試。它是個簡單的 **callback**。`Find()` 方法使用 **callback** 測試每個項目並回傳通過埋在 **predicate** 中測試的元素。編譯器將 **lambda** 表示式轉換成 **delegate** 並使用此 **delegate** 來表示 **callback**。

相似的 `TrueForAll()` 套用測試在每個元素上並判斷所有項目的 **predicate** 是否為真。`RemoveAll()` 修改清單容器將 **predicate** 為真者移除。

最後，`List.ForEach()` 方法對清單中的每個元素執行特定的動作。如前述，編譯器將 **lambda** 表示式轉換進方法並建構指向該方法的 **delegate**。

你可以在 **.NET Framework** 中發現無數個這種概念的例子。所有的 **LINQ** 是以 **delegate** 建構。**callback** 用於處理 **Windows Presentation Foundation**（**WPF**）與 **Windows Forms** 中的跨執行緒調度。**.NET Framework** 需要單一

方法時，它會使用呼叫方能以 lambda 表示式形式表示的 delegate。你在你的 API 中需要 callback 時應該依循相同的方式。

為了歷史因素，所有的 delegate 都是多點傳播的 delegate。多點傳播 delegate 將所有加入的標的函式包裝在單一呼叫中。對此方式有兩項警告：它在面對例外時不安全，且回傳值會是多點傳播最後一個叫用的標的函式的回傳值。

在多點傳播 delegate 的叫用中，每個標的是逐一呼叫。delegate 並不捕捉任何例外。因此，標的拋出的例外會終結 delegate 叫用鏈。

回傳值有類似的問題。你可以定義回傳型別而非 `void` 的 delegate。你可以撰寫出檢查使用者終止的 callback：

```
public void LengthyOperation(Func<bool> pred)
{
    foreach (ComplicatedClass cl in container)
    {
        cl.DoLengthyOperation();
        // 檢查使用者終止：
        if (false == pred())
            return;
    }
}
```

它如同單一 delegate，但用於多點傳播會有很多問題：

```
Func<bool> cp = () => CheckWithUser();
cp += () => CheckWithSystem();
c.LengthyOperation(cp);
```

叫用 delegate 所回傳的值是多點傳播鏈最後一個函式的回傳值。其他回傳值被忽略。`CheckWithUser()` 這個 predicate 的回傳會被忽略。

你可以透過自行叫用每個 delegate 標的來處理這兩個問題。你建構的每個 delegate 帶有 delegate 的清單。為自行檢查此鏈並呼叫每一個 delegate，迭代此叫用清單：

```
public void LengthyOperation2(Func<bool> pred)
{
    bool bContinue = true;
    foreach (ComplicatedClass cl in container)
    {
        cl.DoLengthyOperation();
        foreach (Func<bool> pr in pred.GetInvocationList())
            bContinue &= pr();

        if (!bContinue)
            return;
    }
}
```

此例中，我定義語意以讓每個 **delegate** 必須為真時迭代才會繼續。

delegate 提供執行期使用 **callback** 的最佳方式，簡化了用戶端的需求。你可以在執行期設定 **delegate** 標的。你可以支援多個用戶標的。用戶的 **callback** 應該以 **.NET** 的 **delegate** 實作。

做法 08 對事件叫用使用空條件運算子

發起事件在你剛開始接觸事件時似乎很簡單的任務。你定義一個事件，然後於必要時叫用依附在該事件的事件處理程序。底層的多點傳播 **delegate** 物件會管理逐一叫用所有依附其上的處理程序的複雜性。實務上，以這種天真的方式叫用事件有許多陷阱。若沒有處理程序依附在事件上會怎樣？甚至檢查處理程序的程式與呼叫它的程式間會有競爭狀況。**C# 6.0** 新引進的空條件運算子對這種運用提供更清楚的語法。你應該改變現有的習慣以盡可能的使用新語法。

讓我們檢視舊語法與必須做什麼才能寫出安全的事件叫用程式。最簡單的事件叫用如下：

```
public class EventSource
{
    private EventHandler<int> Updated;

    public void RaiseUpdates()
    {
```

```
        counter++;
        Updated(this, counter);
    }

    private int counter;
}
```

此程式有個明顯的問題。如果物件在沒有事件處理程序依附在 `Updated` 事件時執行，此程式會拋出 `NullReferenceException`。C# 的事件處理程序在沒有處理程序依附於事件時其值為 `null`。

因此，開發者必須將事件叫用包裝在檢查中以判斷事件處理程序是否為 `null`：

```
public void RaiseUpdates()
{
    counter++;
    if (Updated != null)
        Updated(this, counter);
}
```

此程式幾乎在所有情況下都可運作，但它有個潛在的 **bug**。它有可能執行第一行程式檢查事件處理程序是否為空並發現事件處理程序不是空的。然後，在檢查與叫用事件之間，另一個執行緒有可能執行並取消此唯一的事件處理程序使得處理程序為空。你還是會收到 `NullReferenceException`。然而，收到它的情況很罕見且不容易複製。

此 **bug** 很難診斷與改正。程式看起來是對的。為了看出錯誤，你必須讓執行緒以正確的順序執行。有經驗的開發者已經從這種情況得到教訓並以下列程式取代：

```
public void RaiseUpdates()
{
    counter++;
    var handler = Updated;
    if (handler != null)
        handler(this, counter);
}
```

此程式是 .NET 與 C# 推薦的發起事件做法。它可用，且具有執行緒安全性，但在可讀性上有些問題。這樣的改變如何讓程式具有執行緒安全性不是很明顯。

讓我們來認識為何它可用且具有執行緒安全性。

第一行指派目前的事件處理程序給一個新的區域變數。此新的區域變數現在帶有指向來自成員變數事件的原始處理程序的多點傳播 **delegate**。

此事件指派建構等號右邊的淺複製。該淺複製帶有每個事件處理程序的參考的新拷貝。對事件欄位沒有依附其上的處理程序的狀況，右邊為空，而指派的左邊會儲存 `null` 值。

如果其他執行緒的程式取消此事件，該程式會修改其類別的事件欄位，但它不會從區域變數刪除事件處理程序。區域變數還是持有複製當時的事件訂戶。

因此，此程式檢查 `null` 時會看到複製當時的事件訂戶。然後事件受條件叫用，而所有複製當時存在的事件處理程序都會被叫用。

它可行，但 .NET 新手不容易看出來並理解。還有，發出事件的地方都必須複製它。替代方案是建構私用方法來發起事件，並將此做法包裝在私用方法中。

這有很多程式要寫並有理解的成本，應該有個非常容易使用的功能：事件叫用。

空條件運算子使得此動作的程式更為簡單：

```
public void RaiseUpdates()
{
    counter++;
    Updated?.Invoke(this, counter);
}
```

此程式使用空條件運算子（“`?.`”）來安全的叫用事件處理程序。“`?.`” 運算子對運算子左邊求值。如果求出非空值，運算子右邊的表示式就會執行。如果表示式為空，它會跳出並繼續執行下一個陳述。

它在語意上類似前面的 `if` 陳述結構。唯一的差別是“`?.`”運算子的左邊只求值一次。

使用“`?.`”運算子時必須使用 `Invoke` 方法，因為語言並不允許“`?.`”運算子後面接著括號。編譯器會對每個 **delegate** 或事件定義產生一個型別安全的 `Invoke()` 方法。這表示 `Invoke()` 與之前直接發起事件的範例是完全相同的程式。

此程式具有執行緒安全性。它也比較精簡。由於是一行的程式，沒有理由建構另一個輔助方法來弄亂類別的設計。以一行程式發起事件 - 這是我們想要的清晰。

但舊習難改，若你已經使用 .NET 多年，你需要建立新的習慣。你手上還有使用舊做法的程式。你的團隊也難以建立新習慣。網路上還有多年累積的舊做法建議。開發者在發起事件而遇到 `NullReferenceException` 時上網找到的資源很多會指向舊做法。

新方法較簡單且清楚。每一次都採用它。

做法 09　減少 boxing 與 unboxing

值型別是資料的容器。它們不是多形型別。另一方面，.NET Framework 的設計以整個物件階層的 `System.Object` 這個單一參考型別為根。這兩個目標是不一致的。.NET Framework 使用 **boxing** 與 **unboxing** 來連接兩邊。**boxing** 將值型別複雜無型別參考物件中以讓值型別可用於需要參考型別的地方。**unboxing** 從 **box** 中擷取值型別的拷貝。在預期使用 `System.Object` 型別或界面型別的地方要使用值型別時必須動用 **boxing** 與 **unboxing**。但 **boxing** 與 **unboxing** 是消耗效能的操作。有時候 **boxing** 與 **unboxing** 還會建構物件的暫存拷貝，它們會引發微妙的 **bug**。盡可能避免 **boxing** 與 **unboxing**。

boxing 將值型別轉換成參考型別。新的參考物件稱為 **box**，分配在 **heap** 中，而值型別的拷貝儲存在參考物件中。圖 1.1 顯示 **box** 物件如何儲存與存取。**box** 帶有值型別物件的拷貝並複製被 **box** 的值型別實作的界面。你需要從 **box** 讀取時會建構並回傳值型別的拷貝。這是 **boxing** 與 **unboxing** 的關鍵概念：值的拷貝放在 **box** 中，存取時會建構另一個拷貝。

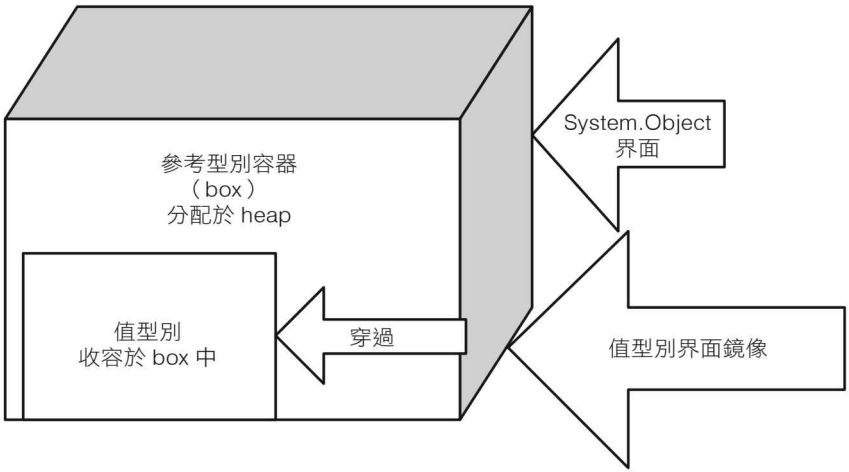

圖 1.1 box 中的值型別。將值型別轉換成 System.Object 參考時會建構不具名的參考型別。值型別儲存於不具名參考型別中。所有存取值型別的方法都穿過 box 到所儲存的值型別。

.NET 2.0 加入的泛型意味著透過泛型類別與泛型方法可避免 boxing 與 unboxing。這是撰寫使用值型別而無需 boxing 操作的最有力的方式，但 .NET Framework 中有許多方法具有型別為 System.Object 的參數。這些 API 還是會產生 boxing 與 unboxing 操作。它是自動發生。只要在預期 System.Object 等參考型別的地方使用值型別，編譯器就會產生 boxing 與 unboxing 指令。此外，透過界面指標使用值型別時就會發生 boxing 與 unboxing 操作。沒有警告 -boxing 就發生了。就算下面如此簡單的陳述也會執行 boxing：

```
Console.WriteLine($"A few numbers:{firstNumber},
{secondNumber}, {thirdNumber}");
```

建構內插字串的工作使用 System.Object 參考的陣列。整數是值型別且必須被 box 以讓它能夠傳給由編譯器產生的方法以從值建構出字串。強制轉換三個參數成 System.Object 的唯一方法是將它們 box。此外，在此方法中，程式進入 box 內部以呼叫 box 內的物件的 ToString() 方法。在某種意義上，你建構出如下的程式：

```
int i = 25;
object o = i; // box
Console.WriteLine(o.ToString());
```

boxing 與 **unboxing** 重複發生於每個物件上。在從參數建構出字串的方法中，執行了如下的程式：

```
object firstParm = 5;

object o = firstParm;
int i = (int)o; // unbox
string output = i.ToString();
```

你絕不會自己寫出這樣的程式。但由於讓編譯器自動將值型別轉換成 `System.Object`，你還是讓它發生了。編譯器只是想幫忙。它想要讓你成功。它很樂意產生必要的 **boxing** 與 **unboxing** 陳述以將任何值型別轉換成 `System.Object` 的實例。為避免這樣的懲罰，你應該在將型別傳給 `WriteLine` 之前自行轉換成字串實例：

```
Console.WriteLine($@"A few numbers:{firstNumber.ToString()},
{secondNumber.ToString()}, {thirdNumber.ToString()}");
```

此程式使用已知的整數型別，且值型別（整數）絕不會隱含轉換成 `System.Object`。這個常見的例子展示出避免 **boxing** 的第一條規則：注意隱含轉換成 `System.Object`。如果你能夠避免，值型別不應該被轉換成 `System.Object`。

另一個常見的將值型別轉換成 `System.Object` 的例子是將值型別放在 .NET 1.x 的 **collection** 中。你應該使用 .NET BCL 的 2.0 版加入的泛型 **collection** 替代 1.x 以物件為基礎的 **collection**。你應該能理解這個問題與如何避免。

.NET Framework 的第一版 **collection** 將參考儲存在 `System.Object` 實例中。任何時候將值型別加入 **collection** 就會進到 **box** 中。任何時候將物件從 **collection** 移出就會從 **box** 中複製。將物件從 **box** 中提出總是會製作拷貝。這會在你的應用程式中引發微妙的 **bug**。這些 **bug** 是定義 **boxing** 的規則的結果。從讓你修改欄位並將一些這樣的物件加入 **collection** 的簡單程式開始：

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

    public override string ToString()
    {
        return Name;
```

```
    }
}

// 在 collection 中使用 Person
var attendees = new List<Person>();
Person p = new Person { Name = "Old Name" };
attendees.Add(p);

// 嘗試改變名稱：
// 若 Person 是參考型別則可行
Person p2 = attendees[0];
p2.Name = "New Name";

// 輸出 "Old Name"：
Console.WriteLine(attendees[0].ToString( ));

int i = 25;
object o = i; // box
Console.WriteLine(o.ToString());
```

Person 是值型別。JIT 編譯器對 List<Person> 建構特定的封閉泛型型別以讓 Person 物件不被 box，因為它們儲存於 attendees 這個 collection 中。另一個拷貝在你取出 Person 物件以存取 Name 屬性供修改時產生。你只是對拷貝修改。事實上，第三個拷貝在透過 attendees[0] 物件呼叫 ToString() 函式時產生。為此與其他原因，你應該建構不可變的值型別。

是的，值型別可被轉換成 System.Object 或任何界面參考。這些轉換隱含的發生，使得尋找它們變得更為複雜。這些是環境與語言的規則。boxing 與 unboxing 操作會在你預料之外製作拷貝，這導致 bug。以多形方式處理值型別也有效能成本。注意將值型別轉換成 System.Object 或界面型別的程式：將值放入 collection、呼叫定義於 System.Object 的方法、與轉換成 System.Object。盡可能避免。

做法 10 只對基底類別更新使用 new 修飾詞

你可以對類別成員使用 new 修飾詞來重新定義繼承自基底類別的非虛擬成員。但你可以並不代表你應該這麼做。重新定義非虛擬方法會產生模糊的行

為。如果兩個類別有繼承上的關係，大部分開發者會觀察這兩段程式並立即假設它們執行相同的工作：

```
object c = MakeObject();

// 透過 MyClass 的參考呼叫
MyClass cl = c as MyClass;
cl.MagicMethod();

// 透過 MyOtherClass 的參考呼叫
MyOtherClass cl2 = c as MyOtherClass;
cl2.MagicMethod();
```

用到 new 修飾詞時就不是這樣了：

```
public class MyClass
{
    public void MagicMethod()
    {
        Console.WriteLine( “MyClass” );
        // 省略細節
    }
}

public class MyOtherClass : MyClass
{
    // 重新定義此類別的 MagicMethod
    public new void MagicMethod()
    {
        Console.WriteLine( “MyOTherClass” );
        // 省略細節
    }
}
```

這種做法會使得開發者混淆。如果你呼叫同一個物件的同一個函式，你會預期執行相同的程式。改變用於呼叫函式的參考或標籤而改變行為是不合理的，這樣會不一致。MyOtherClass 物件因參考方式不同而有不同的行為。new 修飾詞不會讓非虛擬方法變成虛擬方法。相對的，它讓你在你的類別命名範圍加入不同的方法。

非虛擬方法是靜態綁定。參考 MyClass.MagicMethod() 的程式呼叫的正是該函式。執行期不會去找繼承類別定義的不同版本。另一方面，虛擬函式是動態綁定。執行期會根據物件的執行期型別叫用適當的函式。

避免使用 new 修飾詞重新定義非虛擬函式的建議，不應解釋成建議在定義基底類別時讓所有東西都做成虛擬的。函式庫設計者讓一個函式虛擬是提出一個合約，讓你知道繼承它的類別應該要改變虛擬函式的實作。一組虛擬函式定義了繼承類別應該要改變的所有行為。“預設虛擬”的設計表明繼承類別可以改變其行為。它確實表示你不會考慮繼承類別修改行為的後果。相對的，要花時間考慮什麼方法與屬性應該多形。讓這些 - 只有這些 - 方法與屬性虛擬。不要把它想做是你的類別的使用限制，相對的，將它想做是提供自定行為的進入點。

有個地方，僅在此處，你會想要使用 new 修飾詞。你使用 new 修飾詞來利用帶有你已經使用的方法名稱的新版本基底類別。你已經寫好依靠你的類別中的方法名稱的程式。你可能已經釋出使用此方法的其他組件。你已經在你的函式庫中建構下列的類別，使用了定義於其他函式庫的 BaseWidget：

```
public class MyWidget : BaseWidget
{
    public new void NormalizeValues()
    {
        // 省略細節
    }
}
```

你寫好了程式，客戶正在使用它。然後你發現 BaseWidget 有新版本，你立刻買了新版本並嘗試重建你的 MyWidget 類別。由於 BaseWidget 團隊加上了自己的 NormalizeValues 方法使得重建失敗：

```
public class BaseWidget
{
    public void Normalizevalues()
    {
        // 省略細節
    }
}
```

這是個問題，你的基底類別在你的類別的命名範圍內搞出一個方法。有兩個方法可以改正它。你可以修改 NormalizeValues 方法的名稱。注意！我是指 BaseWidget.NormalizeValues() 與 MyWidget.NormalizeAllValues 在語意上是相同的操作。如果不是，你不應該呼叫基底類別的實作。

```
public class MyWidget : BaseWidget
{
    public void NormalizeAllValues()
    {
        // 省略細節
        // 只在（幸好）新方法執行
        // 相同操作時呼叫基底類別
        base.Normalizevalues();
    }
}
```

或者，你可以使用 new 修飾詞：

```
public class MyWidget : BaseWidget
{
    public new void NormalizeValues()
    {
        // 省略細節
        // 只在（幸好）新方法執行
        // 相同操作時呼叫基底類別
        base.Normalizevalues();
    }
```

如果你可以取得所有 MyWidget 類別的用戶的原始碼，你應該修改方法的名稱，因為長期來看這樣比較好。但若你已經將 MyWidget 類別散佈世界各地，這會迫使你的用戶必須進行修改。此時 new 修飾詞就很方便了。你的用戶可以繼續使用你的 NormalizeValues() 方法而無需改變。它們不會呼叫到 BaseWidget.NormalizeValues()，因為它不存在。new 修飾詞可處理基底類別升級與你之前在你的類別中宣告的成員衝突的情況。

當然，你的使用者可能會想要使用 BaseWidget.NormalizeValues() 方法。這時候又回到原來的問題：兩個方法看起來一樣但其實不同。考慮 new 修飾詞的長期後果。有時候修改方法的短期不方便還是比較好。

new 修飾詞必須小心的使用。如果你不假思索的套用它，你會在你的物件中產生模糊的方法呼叫。這是發生在升級你的基底類別導致與你的類別衝突的特殊情況。就算在這種情況下，使用前還是要仔細考慮。更重要的是，其他情況不要使用它。

資源管理 2

.NET 程式在 managed 環境下執行對建構高效率 C# 程式的設計方式有重大的影響。運用此環境需要你將其他環境的思考方式改變成 .NET Common Language Runtime（CLR）的思考方式。這表示要認識 .NET 的垃圾回收（GC）、物件的生命週期，以及如何控制 unmanaged 的資源。這一章的討論幫助你建構善用環境與功能的軟體的做法。

做法 11 認識 .NET 資源管理

不知道環境如何處理記憶體與其他重要資源就無法成為高效率的開發者。在 .NET 中，這表示要認識記憶體管理與垃圾回收。

GC 幫你控制 managed 記憶體。不像原生環境，你不負責大部分的記憶體洩露、懸空指標、未初始化指標或其他記憶體管理問題。但垃圾回收在你必須自行清理時做的更好。你負責資料庫連線、GDI+ 物件、COM 物件與其他系統物件等 unmanaged 資源。此外，你可能會因為在物件間使用事件處理程序或 delegate 產生連結而使得物件留在記憶體中的時間比你預期的更久。請求結果所執行的查詢也會導致物件保持被參考的時間比預期更久（見做法 14）。

以下是好消息：由於 GC 控制記憶體，特定設計方法比你自行管理所有記憶體更容易實作。循環參考，無論是簡單關係或複雜的物件網，都比你要自行管理記憶體的環境更容易正確的實作。GC 的 Mark 與 Compact 演算法有效率的檢測這些關係並完全移除無法觸及的物件網。GC 會從應用程式的根物件遍歷物件樹判斷物件是否可及，而非類似 COM 一樣強制每個物件記錄對它的參考。`EntitySet` 類別提供此演算法如何簡化決定物件所有權的例子。

一個 entity 是從資料庫載入的一群物件。每個 entity 可帶有其他 entity 物件的參考。任何一個 entity 也可以帶有對其他 entity 的連結。如同關聯式資料庫的實體集模型，這些連結與參考可能會產生循環。

有些物件網間的參考由不同的實體集表示。釋放記憶體是 GC 的責任。由於 .NET Framework 設計者無需釋放這些物件，複雜的物件參考網不會產生問題。釋放此物件網的正確順序無需做決定；它是 GC 的工作。GC 的設計簡化了辨識這種物件網是否為垃圾的問題。應用程式可以在使用完畢時停止參考任何 entity。垃圾回收會知道 entity 是否還可從應用程式中的存活物件觸及。任何不能從應用程式觸及的物件都是垃圾。

垃圾回收在每次執行時會整理 managed 的 heap。整理 heap 的動作會移動 managed 的 heap 中的每個存活物件，以讓可用空間被分配在一個連續的記憶體區塊中。圖 2.1 顯示垃圾回收前後 heap 的快照。每次 GC 操作後所有可用記憶體被放在一個連續的記憶體區塊中。

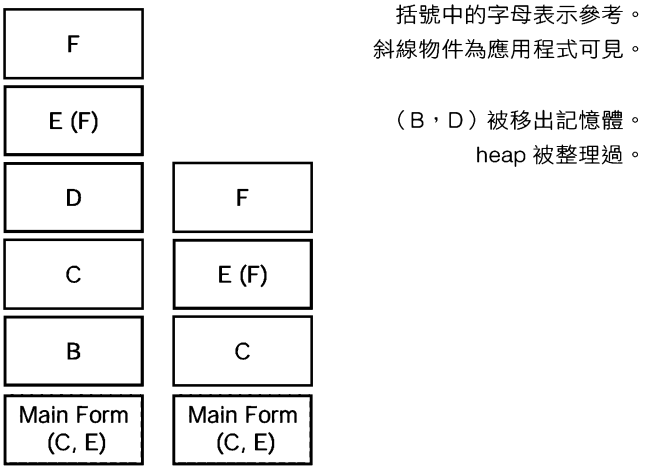

圖 2.1　垃圾回收不只移除未使用的記憶體，它還會移動記憶體中的其他物件以整理記憶體並讓可用空間最大化。

如你剛剛學到的，記憶體管理（對 managed 的 heap）完全由垃圾回收負責。其他系統資源必須由開發者管理：你與你的類別的使用者。有兩種機制可幫助開發者控制 unmanaged 資源的生命週期：finalizer 與 `IDisposable` 界面。finalizer 是確保你的物件總是有辦法釋放 unmanaged 資源的防衛機制。

finalizer 有許多缺點，因此你還有 IDisposable 界面提供較不侵入的方式及時將資源還給系統。

finalizer 由垃圾回收在物件變成垃圾後的某個時間被垃圾回收呼叫。你不知道它何時發生，你只知道在大部分環境下它會於你的物件無法觸及後的某個時間發生。這對 C++ 是個重大改變，且對你的設計有很重要的影響。有經驗的 C++ 程式設計師撰寫在建構元分配重要資源並在解構元釋放資源的類別：

```
// 好的 C++，不好的 C#：
class CriticalSection
{
    // 建構元取得系統資源。
    public CriticalSection()
    {
        EnterCriticalSection();
    }

    // 解構元釋放系統資源。
    ~CriticalSection()
    {
        ExitCriticalSection();
    }

    private void ExitCriticalSection()
    {
    }
    private void EnterCriticalSection()
    {
    }
}

// 使用：
    void Func()
    {
    // s 的生命週期控制
    // 系統資源的存取。
    CriticalSection s = new CriticalSection();

    // 執行工作
```

```
    //...

    // 編譯器產生對解構元的呼叫
    // 程式離開關鍵部分
}
```

此常用的 C++ 寫法確保資源分配的解除能夠處理例外。但這在 C# 中不可行 - 至少效果不同。決定性的 finalization 並非 .NET 環境或 C# 語言的一部分。強行在 C# 語言中使用 C++ 的決定性 finalization 慣用寫法不會運作的很好。在 C# 中，finalizer 最終會在大部分環境中執行，但並非及時執行。在上面的例子中，程式最終會離開關鍵段，但在 C# 中它不會於函式結束時離開關鍵段。它會在之後不確定的時間發生，你不知道何時，你無法得知它的發生。finalizer 是保證指定型別的物件的 unmanaged 資源分配最終會被釋放的唯一方式。但 finalizer 執行的時間不一定，因此設計與撰寫方式應該盡可能減少 finalizer 的建構，並少用已經建構的 finalizer。你在這一章會學到避免建構 finalizer 的技巧，與無可避免時如何減少它所帶來的負面影響。

依靠 finalizer 也會有效能問題。需要 finalization 的物件會拖累垃圾回收的效能。GC 發現一個需要 finalization 的垃圾物件時，它將無法立即將它從記憶體中移除。首先，它會呼叫 finalizer。finalizer 並非由垃圾回收的同一個執行緒執行。相對的，GC 將準備要 finalization 的每個物件放在一個佇列中並執行這些物件的 finalizer。它持續執行它的工作，從記憶體刪除其他垃圾。在下一個 GC 循環，被 finalized 過的物件會從記憶體中移除。圖 2.2 顯示三種不同的 GC 操作與記憶體使用。注意需要 finalizer 的物件會多在記憶體中停留幾個循環。

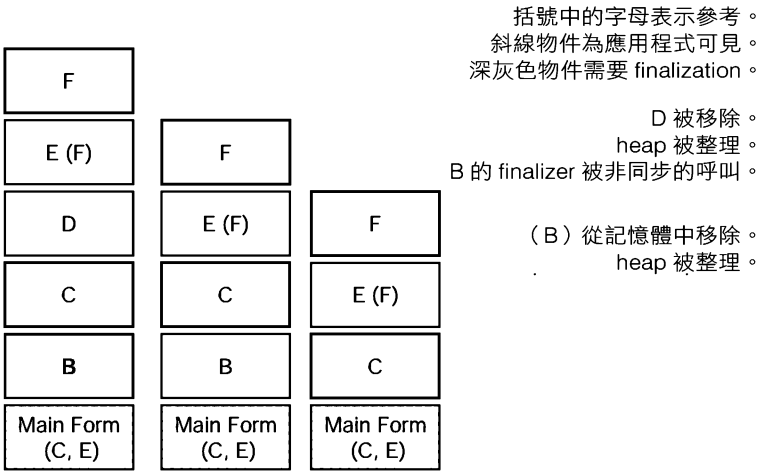

圖 2.2　此序列顯示 finalizer 對垃圾回收的效果。物件在記憶體中存留更久，需要生出額外的執行緒以執行垃圾回收。

這或許會讓你相信需要 **finalization** 的物件在記憶體中存活比所需多一個 **GC** 循環。但我已經將它簡化。實際更為複雜，因為還有另一個 **GC** 的設計決策。**.NET** 的垃圾回收定義出世代以將工作最佳化。世代幫助 **GC** 更快的辨識最有可能的垃圾候選者。任何從上次垃圾回收操作後建構的物件是世代 **0** 物件。一次 **GC** 操作後活存的物件是世代 **1** 物件。兩次或以上 **GC** 操作後活存的物件是世代 **2** 物件。世代的目的是要分離短命物件與接近應用程式壽命的物件。世代 **0** 物件大部分都是短命物件變數。成員變數與全域變數快速的進入世代 **1** 並最終進入世代 **2**。

GC 最佳化以限制檢視第一代與第二代物件的頻率達成。每個 **GC** 循環會檢視世代 **0** 物件。大約 **10** 次 **GC** 會檢查一次世代 **0** 與 **1** 物件。大約 **100** 次 **GC** 會檢視所有物件。再想一下 **finalization** 與它的成本：需要 **finalization** 的物件可能會比不需要 **finalization** 的物件多停留 **9** 次 **GC** 循環，如果它還沒有被 **finalized**，它會進入世代 **2**。在世代 **2** 中，物件多存活 **100** 次 **GC** 循環直到下個世代 **2** 回收。

我花了一些時間解釋為何 **finalizer** 不是個好方案。你還是必須釋放資源。你使用 `IDisposable` 界面與標準處置模式處理這些問題（見做法 **17**）。

結論是，要記得以垃圾回收負責記憶體管理的 **managed** 環境是正面的：記憶體洩露與其他指標問題不再是你的問題。非記憶體資源強迫你建構 **finalizer**

以確保正確的清理這些非記憶體資源。finalizer 對你的程式效能有重大的影響，但你必須使用它們來避免資源洩露。實作並使用 IDisposable 界面以避免 finalizer 引發的垃圾回收效能耗盡。下一個做法討論幫助你建構更有效率的使用此環境的技巧

做法 11　偏好成員初始化程序而非指派陳述

類別通常有一個以上的建構元，時間久了就很容易讓成員變數與建構元不同步。要確保不會發生這種情況最好的方式，是在宣告變數時加以初始化，而非在建構元中。你應該對靜態與實例變數使用初始化程序語法。

宣告變數時建構成員變數是 C# 的自然做法。在宣告時初始化變數：

```
public class MyClass
{
// 宣告 collection 並加以初始化
private List<string> labels = new List<string>();
}
```

無論你最終對 MyClass 型別加上多少個建構元，labels 都會正確初始化。編譯器在每個建構元的開始處產生程式，以執行你為實例成員變數定義的初始化程序。加入新的建構元時，labels 還是會被初始化。同樣的，如果你加入新的成員變數，你無需對每個建構元加上初始化程式；在定義處初始化變數就夠了。同樣重要的是初始化程序會被加入編譯器產生的預設建構元中。若你沒有明確的定義任何建構元，C# 編譯器會為你的型別建構預設建構元。

初始化程序不只是建構元內方便的陳述捷徑。由初始化程序產生的陳述被放在物件程式中你的建構元的前面。初始化程序在你的型別的基底類別建構元之前執行，它們是以你的類別中變數宣告的順序執行。

使用初始化程序是避免你的型別的變數未初始化的最簡單方式，但它並不完美。有三種情況不應該使用初始化程序語法。第一種是你要將物件初始化成 0 或 null 時。系統預設的初始化會在你的任何程式執行前將所有東西設為 0。系統產生的初始化 0 是在非常低階以 CPU 指令設定整個記憶體區塊為 0。你的初始化 0 是多餘的，C# 編譯器會忠實的加上額外的指令將記憶體再設一次 0。這沒有什麼不對 - 但它增加額外的程式。

```
public struct MyValType
{
    // 省略
}

MyValType myVal1; // 初始化成 0
MyValType myVal2 = new MyValType(); // 還是 0
```

兩個陳述都將變數初始化為全 0。第一個陳述將儲存 myVal1 的記憶體設為 0。第二個陳述使用 IL 指令 initobj，它導致 myVal2 變數發生 box 與 unbox 操作。這需要一些時間（見做法 9）。

第二個無效率發生於你對同一個物件建構多個初始化程序。你只應該對接收建構元中相同初始化程序的變數使用初始化程序語法。以下版本的 MyClass 在它的建構元中有個建構兩個不同 List 物件的路徑：

```
public class MyClass2
{
    // 宣告 collection 並初始化。
    private List<string> labels = new List<string>();
    MyClass2()
    {
    }

    MyClass2(int size)
    {
        labels = new List<string>(size);
    }
}
```

當你建構新的 MyClass2 並指定 collection 的大小時，你建構出兩個陣列清單。其中一個立即變成垃圾。變數的初始化程序在所有建構元之前執行。建構元本身建構第二個陣列清單。編譯器會產生出下面這種版本的 MyClass2，而你絕不會自行寫出這種程式（處理這種狀況的正確方式見做法 14）。

```
public class MyClass2
{
    // 宣告 collection 並初始化
    private List<string> labels;

    MyClass2()
```

```
    {
        labels = new List<string>();
    }

    MyClass2(int size)
    {
        labels = new List<string>();
        labels = new List<string>(size);
    }
}
```

只要你使用隱含屬性就會遇到相同的狀況。對隱含屬性是正確選擇的資料元素，做法 14 顯示如何在初始化隱含屬性保存的資料時減少重複。

最後一個將初始化移到建構元內的理由是設置例外處理。你不能將初始化程序包裝在 `try` 區塊中。任何可能會在建構成員變數時產生的例外會傳播到你的物件之外。你不能在類別中嘗試任何的復原。你應該將初始化程式移動到建構元內來實作適當的復原程式以建構你的型別並優雅的處理例外（見做法 47）。

成員初始化程序是確保你的型別的成員變數無論呼叫哪一個建構元都會被初始化的最簡單方式。初始化程序在你為你的型別製作的每一個建構元之前執行。使用此語法表示你在新增建構元時不會忘記加上適當的初始化。所有建構元以相同方式建構成員變數時使用初始化程序；它比較容易閱讀與維護。

做法 13　對靜態類別成員進行適當的初始化

你知道你應該在建構型別的實例前初始化它的靜態成員變數。為此 C# 讓你使用靜態初始化程序與靜態建構元。靜態建構元是在類別的任何其他方法、變數、或屬性首次存取之前執行的特殊函式。你使用此函式來初始化靜態變數、實行 **singleton** 模式、或在類別可用之前執行其他必要的工作。你不應該使用你的實例建構元、某些特殊的私用函式、或任何其他方法了初始化靜態變數。對需要複雜或昂貴的初始化的靜態欄位，考慮使用 `Lazy<T>` 於欄位首次存取時執行初始化。

如同使用實例初始化，你可以使用初始化語法作為靜態建構元的替代方法。如果你只是需要分配靜態成員，使用初始化程序語法。初始化靜態成員變數有更複雜的邏輯時，建構靜態建構元。

在 C# 中實作 **singleton** 模式是靜態建構元最常見的用途。將你的實例建構元設為私用，加上一個初始化程序：

```
public class MySingleton
{
    private static readonly MySingleton theOneAndOnly =
    new MySingleton();
    public static MySingleton TheOnly
    {
        get { return theOneAndOnly; }
    }

    private MySingleton()
    {
    }

    // 省略其餘內容
}
```

若 **singleton** 的初始化具有更為複雜的邏輯，**singleton** 模式可以用下面的簡單方式撰寫：

```
public class MySingleton2
{
    private static readonly MySingleton2 theOneAndOnly;

    static MySingleton2()
    {
        theOneAndOnly = new MySingleton2();
    }
    public static MySingleton2 TheOnly
    {
        get { return theOneAndOnly; }
    }

    private MySingleton2()
```

```
    {
    }

    // 省略其餘內容
}
```

如同實例初始化程序，靜態初始化程序在靜態建構元被呼叫前執行，且你的靜態初始化程序會在基底類別的靜態建構元之前執行。

CLR 在你的型別於應用程式空間（AppDomain）首次存取之前會自動呼叫你的靜態建構元。你只可以定義一個靜態建構元，它不能取用任何變數。由於靜態建構元是被 CLR 呼叫，你必須注意它產生的例外。如果讓例外溢出靜態建構元，CLR 會拋出 TypeInitializationException 來終止你的程式。呼叫方捕捉到例外的狀況更複雜。嘗試建構型別的程式會失敗，直到 AppDomain 被卸下。CLR 不能透過執行靜態建構元初始化該型別。它不會重新嘗試，且型別沒有被正確的初始化。該型別的物件（或繼承它的型別）不會被完好的定義。因此它是不被允許的。

例外是使用靜態建構元代替靜態初始化程序最常見的原因。如果你使用靜態初始化程序，你無法自行捕捉例外。用靜態建構元就可以（見方法 47）：

```
static MySingleton2()
{
    try
    {
        theOneAndOnly = new MySingleton2();
    }
    catch
    {
        // 在這裡嘗試復原
    }
}
```

靜態初始化程序與靜態建構元提供最乾淨、清楚的類別靜態成員初始化方式。它們容易閱讀與容易寫好。它們被加入語言以用於處理其他語言難以處理的靜態成員初始化。

做法 13 減少重複的初始化邏輯

撰寫建構元通常是重複性的工作。許多開發者撰寫第一個建構元然後複製到其他建構元以滿足定義於類別界面的多個過載。你最好不是其中之一。如果是，請不要再這麼做。C++ 老手會將常用演算法 factor 成私用輔助方法。也別這麼做。當你發現有多個建構元帶有相同的邏輯，將該邏輯 factor 到共通建構元中。你會享受到避免重複程式的好處，且建構元初始化程序會產生更有效率的物件程式。C# 編譯器視建構元初始化程序為特殊語法並刪除重複的變數初始化程序與重複的基底類別建構元呼叫。這樣會讓最終的物件執行最少的程式以正確的初始化物件。你也因為轉移責任給共通建構元而少寫一些程式。

建構元初始化程序讓一個建構元呼叫其他建構元。下面的例子顯示一個簡單的運用：

```
public class MyClass
{
    // 資料集
    private List<ImportantData> coll;
    // 實例名稱：
    private string name;

    public MyClass() :
        this(0, "")
    {
    }

    public MyClass(int initialCount) :
        this(initialCount, string.Empty)
    {
    }

    public MyClass(int initialCount, string name)
    {
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

C# 4.0 加入了預設參數，可將重複的建構元程式最小化。你可以使用一個指定預設值的建構元取代 MyClass 各個其他值的建構元：

```
public class MyClass
{
    // 資料集
    private List<ImportantData> coll;

    // 實例名稱：
    private string name;

    // 必須符合 new() 約束
    public MyClass() :
        this(0, string.Empty)
    {
    }

    public MyClass(int initialCount = 0, string name = "")
    {
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

選擇預設參數與使用多個過載之間需要權衡考量。預設參數讓使用者有更多選擇。這個版本的 MyClass 為兩個參數指定預設值。使用者可以對兩個或一個參數指定不同的值。使用過載建構元製作各種排列組合需要四個不同的建構元過載：無參數的建構元、要求 initialCount 的建構元、要求 name 的建構元、與全都要求的建構元。對你的類別增加成員則過載的排列組合會變得更多。複雜性讓預設參數成為減少必須建構的過載數量的有力機制。

定義你的型別的建構元的所有參數的預設值表示呼叫新的 MyClass() 時會檢驗使用者的程式。當你想要支援這種概念時，你應該建構該型別的無參數建構元，如上面的範例所示。雖然大部分程式會使用所有的預設參數，使用 new() 約束的泛型類別不會接受有預設值的建構元。為滿足 new() 約束，類別必須具有無參數建構元。因此你應該建構一個無參數建構元以讓用戶可在強制 new() 約束的泛型類別或方法中使用你的型別。這並不是說每個型別都

需要無參數建構元。但若你有支援，要確保加入此程式以讓無參數建構元可在各種情況下使用，包括具有 `new()` 約束的泛型類別。

注意第二個建構元指定 `""` 作為名稱參數的預設值而非常見的 `string.Empty`。這是因為 `string.Empty` 不是編譯期常數。它是定義於 `string` 類別的靜態屬性。由於它不是編譯期常數，你不能用它作為參數的預設值。

但使用預設參數代替過載會在你的類別與所有使用它的用戶間產生更緊密的耦合。特別是正規參數名稱成為公開界面的一部分，如目前的預設值一樣。改變參數值需要重新編譯用戶端程式以挑出改變。這會讓過載建構元面對未來的改變時更具彈性。你可以加入新的建構元，或改變沒有指定值的建構元的預設行為而不會破壞用戶端程式。

預設參數是這種問題的首選解決方案。但某些 API 使用 relfection 來建構物件並依靠無參數建構元。具有各個參數的預設值的建構元與無參數建構元不同。你必須以不同的函式撰寫不同的建構元。對建構元來說，這表示大量的重複程式。使用以一個建構元叫用同類別宣告的其他建構元的建構元鏈代替共通公用程序。這種產生共通建構元邏輯的替代方法有幾點無效率：

```
public class MyClass
{
    private List<ImportantData> coll;
    private string name;

    public MyClass()
    {
        commonConstructor(0, "");
    }

    public MyClass(int initialCount)
    {
        commonConstructor(initialCount, "");
    }

    public MyClass(int initialCount, string Name)
    {
        commonConstructor(initialCount, Name);
    }

    private void commonConstructor(int count,
```

```
    string name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

此版本看起來一樣，但它產生更沒效率的物件程式。編譯器加入程式以幫你在建構元中執行好幾個函式。它對所有的變數初始化程序加上陳述（見做法 12）。它呼叫基底類別的建構元。撰寫你自己的共通公用函式時，編譯器無法 factor 出這種重複的程式。第二個版本的 IL 與這樣撰寫相同：

```
public class MyClass
{
    private List<ImportantData> coll;
    private string name;

    public MyClass()
    {
        // 實例初始化程序放這裡
        object(); // 不合法，只是用來說明
        commonConstructor(0, "");
    }

    public MyClass(int initialCount)
    {
        // 實例初始化程序放這裡
        object(); // 不合法，只是用來說明
        commonConstructor(initialCount, "");
    }

    public MyClass(int initialCount, string Name)
    {
        // 實例初始化程序放這裡
        object(); // 不合法，只是用來說明
        commonConstructor(initialCount, Name);
    }

    private void commonConstructor(int count,
    string name)
```

```
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        this.name = name;
    }
}
```

如果你以編譯器的觀點撰寫第一個版本的建構程式，你會寫成這樣：

```
// 不合法，用來展示 IL 產生的程式
public class MyClass
{
    private List<ImportantData> coll;
    private string name;

    public MyClass()
    {
        // 此處沒有變數初始化程序
        // 呼叫下面顯示的第三個建構元
        this(0, ""); // 不合法，只是用來說明
    }

    public MyClass(int initialCount)
    {
        // 此處沒有變數初始化程序
        // 呼叫下面顯示的第三個建構元
        this(initialCount, "");
    }

    public MyClass(int initialCount, string Name)
    {
        // 實例初始化程序放這裡
        // 不合法，只是用來說明
        coll = (initialCount > 0) ?
        new List<ImportantData>(initialCount) :
        new List<ImportantData>();
        name = Name;
    }
}
```

差別在於編譯器不會對基底類別的建構元產生多個呼叫，也不會複製實例變數初始化程序到每個建構元中。基底類別的建構元只會從最後一個建構元呼

叫的這件事也有重大影響：你不能在建構元定義中引用一個以上的建構元初始化程序。你可以使用 `this()` 委派給此類別的其他建構元，或使用 `base()` 呼叫基底類別建構元。你不能兩個都做。

還不相信這種建構元初始化程序的問題？想一想唯讀常數。下面的例子中，物件的名稱不應該在其生命週期中改變。這表示你應該讓它唯讀。這會導致共通公用函式產生編譯器錯誤：

```
public class MyClass
{
    / 資料集
    private List<ImportantData> coll;
    // 實例的數目
    private int counter;
    // 實例的名稱
    private readonly string name;
    public MyClass()
    {
        commonConstructor(0, string.Empty);
    }

    public MyClass(int initialCount)
    {
        commonConstructor(initialCount, string.Empty);
    }

    public MyClass(int initialCount, string Name)
    {
        commonConstructor(initialCount, Name);
    }

    private void commonConstructor(int count,
    string name)
    {
        coll = (count > 0) ?
        new List<ImportantData>(count) :
        new List<ImportantData>();
        // 從建構元外改變名稱會產生錯誤
        //this.name = name;
    }
}
```

編譯器會強制 `this.name` 的唯讀特質且不容許建構元以外的程式改變它。C# 的建構元初始化程序提供其他方法。除了最簡單的類別外，類別都有一個以上的建構元。它們的任務是初始化物件的所有成員。由於這種本質，這些函式有類似或最好是共用的邏輯。使用 C# 的建構元初始化程序來做出共通演算法來讓你撰寫一次並讓它們執行一次。

預設參數與過載都有其用途。一般來說，你應該偏好預設值而非過載建構元。畢竟若讓用戶開發者指定所有參數值，你的建構元必須能夠處理使用者指定的任何值。你的原始預設值應該合理且不會產生例外。因此，就算改變預設值在技術上會產生問題，它也不應該讓用戶看到。它們的程式還是使用原始值，而這些原始值還是應該產生合理的行為。這樣可以將使用預設值可能產生的問題減到最小。

這是 C# 物件初始化的最後一條。是時候回顧建構型別實例的完整事件順序。你應該已經了解物件的操作與預設初始化順序。你應該會努力在建構過程將每個成員變數只初始化一次。完成的最好方式是盡早將值初始化。以下是建構型別的第一個實例的操作順序：

1. 靜態變數儲存體設為 0。
2. 執行靜態變數初始化程序。
3. 執行基底類別的靜態建構元。
4. 執行靜態建構元。
5. 實例變數儲存體設為 0。
6. 執行實例變數初始化程序。
7. 執行適當的基底類別實例建構元。
8. 執行實例建構元。

同型別的後續實例從步驟 5 開始，因為類別初始化程序只執行一次。還有，步驟 6 與 7 做過最佳化使建構元初始化程序讓編譯器移除重複的指令。

C# 語言的編譯器保證建構物件時所有東西會以某種方式被初始化。至少，你被保證你的物件使用的記憶體在建構實例時被設為 0。對靜態成員與實例成員都是如此。你的目標是確保以你想要的方式初始化所有值且只執行初始

化程式一次。使用初始化程序來初始化簡單的資源。使用建構元來初始化需要更複雜的邏輯的成員。還有，呼叫其他建構元來減少重複。

做法 15　避免建構不必要的物件

垃圾回收幫你執行非常棒的記憶體管理，並以非常有效率的方式移除無用的物件。但無論你怎麼看，分配與摧毀 heap 空間的物件所花費的處理器時間比不要分配與摧毀 heap 空間的物件要多。你可能建構大量受參考的方法區域物件而造成效能低落。

因此不要過度使用垃圾回收。你可以依循一些簡單的技巧來減少 GC 所需的工作量。所有的參考型別，包括區域變數，需要分配記憶體。這些物件在沒有根源來保持它們存活時變成垃圾。對區域變數來說，這通常是因為宣告它們的方法不再活動。一個常見的不良做法是在 Windows 的 `OnPaint` 處理程序中分配 GDI 物件：

```
protected override void OnPaint(PaintEventArgs e)
{
    // 劣。在每個 paint 事件建構同一個 font。
    using (Font MyFont = new Font("Arial", 10.0f))
    {
        e.Graphics.DrawString(DateTime.Now.ToString(),
            MyFont, Brushes.Black, new PointF(0, 0));
    }
    base.OnPaint(e);
}
```

`OnPaint()` 經常被呼叫。每次被呼叫時，你建構出另一個帶有完全相同設定的 `Font` 物件。垃圾回收必須幫你清理它們。GC 用於判斷何時執行的條件包括被分配的記憶體數量與記憶體分配頻率。更多的分配表示對 GC 更大的壓力，導致它更常執行。這非常的沒效率。

相對的，要將 `Font` 物件從區域變數提升為成員變數。每次繪製視窗時重複使用同一個字形：

```
private readonly Font myFont =
    new Font("Arial", 10.0f);
```

```
protected override void OnPaint(PaintEventArgs e)
{
    e.Graphics.DrawString(DateTime.Now.ToString(),
       myFont, Brushes.Black, new PointF(0, 0));
    base.OnPaint(e);
}
```

你的程式每次遇到繪製事件時不再製造垃圾。垃圾回收的工作減少，你的程式會跑的快一點。你提升實作 `IDisposable` 的區域變數至成員變數時，如同上例的字形，你必須在你的類別中實作 `IDisposable`。做法 17 會解釋如何適當的實作。

你應該在區域變數是參考型別（值型別就沒關係）且會在經常被呼叫的程序中使用時將它提升至成員變數。繪製程序的字形是個非常好的例子。只有經常被存取的程序中的區域變數才有需要，不常被呼叫的程序就不用。你要嘗試的是避免重複建構同一個物件，而非將每個區域變數變成成員變數。

前面使用的 `Brushes.Black` 靜態屬性展示出另一個避免重複分配相同物件的技巧。為經常使用的參考型別實例建構靜態成員變數。以前面使用的黑色筆刷為例，每次你需要使用黑色在視窗上繪製某些東西時，你在程式執行過程中建構與摧毀大量的黑色筆刷。將黑色筆刷建構為成員的第一種方式是有幫助，但還不夠好。程式可能會建構數十個視窗與控制項而建構數十個黑色筆刷。.NET Framework 的設計者預期到這種狀況並為你建構可重複使用的單一黑色筆刷。`Brushes` 類別帶有幾個靜態的 `Brush` 物件，每個都是不同的常見顏色。在內部，`Brushes` 物件使用懶演算法以只建構你有請求的筆刷。簡化過的實作如下：

```
private static Brush blackBrush;
public static Brush Black
{
    get
    {
        if (blackBrush == null)
            blackBrush = new SolidBrush(Color.Black);
        return blackBrush;
    }
}
```

第一次請求黑色筆刷時，Brushes 類別會建構它。Brushes 類別記錄這一個黑色筆刷的參考並於再次請求時回傳同一個 **handle**。結果是你建構一個黑色筆刷並一直重複使用它。此外，若你的應用程式不需要特定的資源 - 例如綠色筆刷 - 就不會建構它。此架構提供了限制物件建構的方法以減少完成任務所需的資源。在你的程式中考慮使用此技巧。從正面來看，你建構較少的物件。從反面看，這會導致物件留在記憶體的時間比實際所需長。這甚至表示無法拋棄 **unmanaged** 資源，因為你不知道何時要呼叫 Dispose() 方法。

你已經學到兩種讓你的程式處理工作時減少分配數量的技巧。你可以提升常用區域變數成為成員變數。你可以使用相依性注入來建構與重複使用特定型別的實例。後者涉及製作不可變型別的最終值。System.String 類別是不可變的：字串建構後，字串的內容不能修改。你的程式看起來像是可以修改字串的內容，實際上你是建構新的字串並讓舊字串變成垃圾。下面的程式看起來沒有問題：

```
string msg = "Hello, ";
msg += thisUser.Name;
msg += ". Today is ";
msg += System.DateTime.Now.ToString();
```

但它如同下面手寫的程式一樣無效率：

```
string msg = "Hello, ";
// 不合法，只是用來說明
string tmp1 = new String(msg + thisUser.Name);
msg = tmp1; // "Hello " 是垃圾
string tmp2 = new String(msg + ". Today is ");
msg = tmp2; // "Hello  <user>" 是垃圾
string tmp3 = new String(msg + DateTime.Now.ToString());
msg = tmp3; // "Hello  <user>. Today is " 是垃圾
```

tmp1、tmp2、tmp3 以及原來的 msg ("Hello") 字串都是垃圾。string 類別的 += 運算子建構出新的字串物件並回傳該字串。它並不是透過加入字元到原來的儲存體來修改原有的字串。對上面這種簡單的程式，你應該使用內插字串：

```
string msg = string.Format("Hello, {0}. Today is {1}",
        thisUser.Name, DateTime.Now.ToString());
```

對更複雜的字串操作，你可以使用 StringBuilder 類別：

```
StringBuilder msg = new StringBuilder("Hello, ");
msg.Append(thisUser.Name);
msg.Append(". Today is ");
msg.Append(DateTime.Now.ToString());
string finalMsg = msg.ToString();
```

上面的例子簡單到使用字串內插就好（見做法 4）。最終字串的邏輯對字串內插太過複雜時改用 StringBuilder。StringBuilder 是用於建構不可變字串物件的可變字串類別。它提供可變字串的機制以讓你在製作不可變字串物件前建構與修改文字資料。使用 StringBuilder 來建構最終版本的字串物件。更重要的是學習這種設計方式。當你的設計需要不可變型別時，考慮製作 **builder** 物件以提供多步驟建構最終物件的程式。如此能提供類別使用者一種逐步建構物件的方式並還能維持你的型別的不可變性質。

垃圾回收有效率的執行你的應用程式使用的記憶體的管理工作。但要記得建構與摧毀 **heap** 物件還是需要時間。避免建構大量的物件；不要建構你不需要的物件。還要避免在區域函式中建構多個參考型別物件。相對的，考慮將區域變數提升至成員變數，或建構靜態物件的共用實例。最後，考慮建構不可變類別的可變 **builder** 類別。

做法 16 絕不在建構元中呼叫虛擬函式

虛擬函式在物件的建構過程顯露出奇怪的行為。物件在所有建構元執行完畢前不算建構完成。同時間，虛擬函式的行為可能不如預期。以下面的簡單程式為例：

```
class B
{
    protected B()
    {
        VFunc();
    }

    protected virtual void VFunc()
    {
        Console.WriteLine("VFunc in B");
```

```
    }
}

class Derived : B
{
    private readonly string msg = "Set by initializer";

    public Derived(string msg)
    {
        this.msg = msg;
    }

    protected override void VFunc()
    {
        Console.WriteLine(msg);
    }

    public static void Main()
    {
        var d = new Derived("Constructed in main");
    }
}
```

你覺得會輸出什麼？“Constructed in main”、“VFunc in B” 或“Set by initializer” ？有經驗的 C++ 程式設計師會猜“VFunc in B”。有些 C# 程式設計師會猜“Constructed in main”。但正確答案是“Set by initializer”。

基底類別建構元呼叫定義於類別的虛擬函式但在繼承類別中會覆寫。於執行期，繼承類別版本會被呼叫。畢竟物件的執行期型別是 `Derived`。C# 語言的定義考慮到繼承物件的完全可用，因為所有成員變數在進入建構元時都已經完成初始化。畢竟，所有的變數初始化程序都已經執行。你有過初始化所有變數的機會。但這並不表示你一定要將所有成員變數初始化你想要的值。只有變數初始化程序執行過；繼承類別的建構元中的程式沒有機會執行它的工作。

無論如何，建構物件過程中呼叫虛擬函式會發生一些不一致。C++ 語言的設計者決定讓虛擬函式解析被建構物件的執行期型別。它們決定讓物件的型別在建構時盡快判定。

這背後是有邏輯的。首先，被建構的物件是個 Derived 物件；每個函式都應該為 Derived 物件呼叫正確的覆寫。此處與 C++ 的規則不同：物件的執行期型別在類別的建構元被執行時改變。其次，這個 C# 語言的功能避免了目前型別是抽象基底類別時在虛擬方法的底層實作中帶有空方法指標的問題。以下面的基底類別變化為例：

```
abstract class B
{
    protected B()
    {
        VFunc();
    }

    protected abstract void VFunc();
}

class Derived : B
{
    private readonly string msg = "Set by initializer";

    public Derived(string msg)
    {
        this.msg = msg;
    }

    protected override void VFunc()
    {
        Console.WriteLine(msg);
    }
    public static void Main()
    {
        var d = new Derived("Constructed in main");
    }
}
```

此範例可編譯，因為 B 物件沒有被建構，而任何具體的繼承物件必須提供 VFunc() 的實作。在建構元中呼叫抽象函式時，呼叫與實際執行期類別匹配的 VFunc() 版本的 C # 策略是獲取除執行期例外之外的任何東西的唯一可能方式。有經驗的 C++ 程式設計師會看出若在該語言使用相同程式可能出現的錯誤。在 C++ 中，從 B 的建構元呼叫 VFunc() 可能或當掉。

儘管如此，這個簡單的例子顯示此 C# 策略的陷阱。`msg` 變數是不可變的。它應該在物件的整個生命週期中維持相同值。由於建構元完成過程有個微小的時間窗口，期間此變數可能會具有不同的值：一個在初始化程序中設定，一個在建構元中設定。一般情況下，繼承類別的變數可能還在初始化程序或系統設定的預設狀態。它們當然不會有你認為的值，因為你的繼承類別的建構元還沒有執行。

在建構元中呼叫虛擬函式會讓你的程式對繼承類別中的實作細節非常敏感。你無法控制繼承類別的動作。在建構元中呼叫虛擬函式的程式很脆弱。繼承類別必須在變數初始化程序中正確的初始化所有實例變數。這排除了好幾種物件：大部分的建構元取用一些設定內部狀態的參數。因此你可以說在建構元中呼叫虛擬函式必須要繼承類別定義預設的建構元且沒有其他建構元。但這對所有繼承類別加上了沉重的負擔。你真的預期所有使用者都會照這個規則進行？我不這麼認為。這樣做只有一點好處，但未來可能會很痛苦。事實上，這種難得可行的狀況已經列入 FxCop 與 Visual Studio 的 Static Code Analyzer 工具中。

做法 17　實作標準的 Dispose 模式

我們討論過處置（disposing）持有 unmanaged 資源的物件的重要性。是時候討論在你建構的型別持有記憶體以外的資源時如何撰寫自己的資源管理程式。.NET Framework 有一個用來處置 unmanaged 資源的模式。你的型別的使用者會預期你依循這種標準模式。標準的處置做法在用戶記得時使用 `IDisposable` 界面釋放 unmanaged 資源，並以 finalizer 對付用戶忘記的情況。它與垃圾回收合作以確保你的物件只會在必要時付出與 finalizer 有關的效能代價。這是處理 unmanaged 資源的正確方式，因此完整的認識它會有好處。實務上，.NET 的 unmanaged 資源可透過繼承自 `System.Runtime.Interop.SafeHandle` 的類別來存取，它正確的實作此模式。

類別繼承階層的根基底類別應該要：

- 實作 `IDisposable` 界面來釋放資源。
- 若且唯若你的類別直接持有 unmanaged 資源時，加上 finalizer 作為防衛機制。

- `Dispose` 與 `finalizer`（如果有的話）均將釋放資源的工作委派給繼承類別可以覆寫的虛擬方法以供其資源管理運用。

繼承類別必須：

- 僅於繼承類別必須釋放自己的資源時覆寫該虛擬方法。
- 若且唯若其直接成員欄位之一是 unmanaged 資源時實作 finalizer。
- 記得呼叫基底類別版本的函式。

首先，若且唯若你的類別直接持有 unmanaged 資源，則它必須有個 finalizer。你不應該依賴用戶記得呼叫 `Dispose()` 方法。他們忘記時會產生資源洩漏。沒呼叫 `Dispose` 是他們的錯，但挨罵的是你。唯一保證 unmanaged 資源會被正確釋放的方式是建構 finalizer。因此若且唯若你的型別持有 unmanaged 資源就要建構 finalizer。

垃圾回收執行時，它會立即從記憶體中移除沒有 finalizer 的垃圾物件。具有 finalizer 的物件會留在記憶體中。這些物件會被加入到終結佇列，然後 GC 執行這些物件的 finalizer。在 finalizer 執行緒完成它的工作後，垃圾物件可正常的從記憶體中移除。由於在回收時存活，它們會多待一個世代。它們也會被標示無需執行 finalizer，因為已經執行過了。它們會在下一輪高世代垃圾回收中從記憶體移除。需要終結的物件在記憶體中停留比沒有 finalizer 的物件更久，但你沒有選擇。如果你需要防衛，在你的型別持有 unmanaged 資源時必須撰寫 finalizer。但還不需擔心效能。下一個步驟會確保用戶能比較容易的避免付出與 finalizer 有關的效能代價。

實作 `IDisposable` 是告知使用者與執行期系統你的物件持有必須及時釋放的資源的標準方式。`IDisposable` 界面只有一個方法：

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

`IDisposable.Dispose()` 方法的實作負責四項任務：

1. 釋放所有 unmanaged 資源。
2. 釋放所有 managed 資源（包括與事件脫鉤）。

3. 設定狀態旗標以指示物件被處置了。如果物件在處置後被呼叫，你必須檢查此狀態並在公開成員中拋出 `ObjectDisposed` 例外。
4. 呼叫 `GC.SuppressFinalize(this)` 以抑制終結程序。

你透過實作 `IDisposable` 以達成兩件事：提供機制給用戶以及時釋放所有 **managed** 資源，並讓用戶有方法釋放所有 **unmanaged** 資源。這是相當大的改善。在你的型別實作 `IDisposable` 之後，用戶可以避免終結程序的代價。你的類別是 .NET 社群中行為良好的成員。

但此機制還是有漏洞。繼承類別如何清理它的資源並還能讓基底類別也進行清理？如果繼承類別覆寫 **finalizer** 或加上自己的 `IDisposable` 實作，這些方法必須呼叫基底類別；不然基底類別不會正確的清理。還有，**finalizer** 與 `Dispose` 共同負擔某些責任；你幾乎可以確定 **finalizer** 與 `Dispose` 間會有重複的程式。覆寫界面函式不一定依照你預期的方式運作。界面函式預設上並非虛擬的。我們必須加上額外的工作來處理這些問題。標準處置模式的第三個方法，受保護的虛擬輔助函式，可提取這些共通任務並為繼承類別加上掛鉤來釋放它們分配的資源。基底類別帶有核心界面的程式。虛擬函式提供繼承類別回應 `Dispose()` 或終結程序清理資源的掛鉤：

```
protected virtual void Dispose(bool isDisposing)
```

此過載方法執行必要的工作以支援 **finalizer** 與 `Dispose`，而因為它是虛擬的，它提供所有繼承類別的進入點。繼承類別可覆寫此方法，提供合適的實作以清理它們的資源並呼叫基底類別版本。你在 `isDisposing` 為 `true` 時清理 **managed** 與 **unmanaged** 資源，在 `isDisposing` 為 `false` 時只清理 **unmanaged** 資源。在以上兩種情況下，呼叫基底類別的 `Dispose(bool)` 方法以讓它清理它自己的資源。

以下簡短的範例顯示實作此模式時的程式架構。`MyResourceHog` 類別顯示實作 `IDisposable` 與建構虛擬 `Dispose` 方法的程式：

```
public class MyResourceHog : IDisposable
{
    // 已經 disposed 的旗標
    private bool alreadyDisposed = false;

    // IDisposable 的實作
```

```
    // 呼叫虛擬 Dispose 方法
    // 抑制終結程序
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // 虛擬 Dispose 方法
    protected virtual void Dispose(bool isDisposing)
    {
        // 不要處置超過一次
        if (alreadyDisposed)
            return;
        if (isDisposing)
        {
            // 省略：釋放 managed 資源
        }
            // 省略：釋放 unmanaged 資源
            // 設定 disposed 旗標
            alreadyDisposed = true;
        }

        public void ExampleMethod()
        {
            if (alreadyDisposed)
                throw new
                    ObjectDisposedException("MyResourceHog",
                    "Called Example Method on Disposed object");
            // 省略其餘程式
        }
}
```

如果繼承類別需要執行額外的清理，它實作受保護的 `Dispose` 方法：

```
public class DerivedResourceHog : MyResourceHog
{
    // 自己的 disposed 旗標
    private bool disposed = false;

    protected override void Dispose(bool isDisposing)
    {
        // 不要處置超過一次
```

```
        if (disposed)
            return;
        if (isDisposing)
        {
            // TODO: 釋放 managed 資源
        }
        // TODO: 釋放 unmanaged 資源

        // 讓基底類別釋放它的資源
        // 基底類別負責呼叫
        // GC.SuppressFinalize( )
        base.Dispose(isDisposing);

        // 設定繼承類別的 disposed 旗標
        disposed = true;
    }
}
```

注意基底類別與繼承類別都帶有物件的 **disposed** 狀態旗標。這純粹是防禦性質。複製旗標將處置物件時可能的錯誤封裝成唯一的型別，而非所有做出物件的型別。

你必須防禦性的撰寫 `Dispose` 與 **finalizer**。它們必須是冪等的。`Dispose()` 可能會被呼叫一次以上，其效應應該要與呼叫一次相同。物件的處置會以任何順序發生。你會遇到型別的成員物件之一已經在你的 `Dispose()` 方法被呼叫前就已經被處置的狀況。你不應該將它視為問題，因為 `Dispose()` 方法可能會被呼叫好幾次。注意！ `Dispose()` 是已經被處置物件的公開方法被呼叫時要拋出 `ObjectDisposedException` 這個規則的例外。如果對已經被處置的物件呼叫它，它什麼也不做。**finalizer** 可能會在參考被處置時執行，或從未被初始化。你參考的所有物件都還在記憶體中，因此無需檢查空參考。但你參考的任何物件可能會被處置。它也可能已經被終結。

你會注意到 `MyResourceHog` 與 `DerivedResourceHog` 都沒有 **finalizer**。我寫的範例程式並未直接持有任何 **unmanaged** 資源，因此不需要 **finalizer**。這表示此範例程式不會呼叫 `Dispose(false)`。這是正確的模式。除非你的類別直接持有 **unmanaged** 資源，否則你不應該實作 **finalizer**。只有直接持有 **unmanaged** 資源的類別應該實作 **finalizer** 並加上它的成本。就算它沒有被呼叫到，光是 **finalizer** 的存在就會對你的類別產生相當大的效能傷害。除非你的型別需

要 finalizer，否則不要加上它。但你還是應該正確的實作此模式以讓確實有 unmanaged 資源的繼承類別可以加上 finalizer 並以正確處理 unmanaged 資源的方式實作 `Dispose(bool)`。

這為我帶來了與處置或清理相關的任何方法的最重要的建議：只要釋放資源就好，不要在處置方法執行其他處理。在你的 `Dispose` 或 finalizer 執行其他處理會對物件的生命週期引發嚴重的併發症。物件在你建構它時誕生，在垃圾回收時死亡。你的程式不再能存取它們時可將它們視為昏迷。如果無法碰到一個物件，你就無法呼叫它的方法。從任何目的與意義來說，它已經死了。但具有 finalizer 的物件在被宣告死亡前還有一口氣。finalizer 別的不做只清理 unmanaged 資源。如果 finalizer 最後讓物件又可以呼叫，它就死而復生。雖然從昏迷狀況被喚醒，它活著但狀況不好。下面是個明顯的例子：

```
public class BadClass
{
    // 儲存一個全域物件的參考：
    private static readonly List<BadClass> finalizedList =
        new List<BadClass>();
    private string msg;

    public BadClass(string msg)
    {
        // 快取參考：
        msg = (string)msg.Clone();
    }

    ~BadClass()
    {
        // 將物件加入清單中
        // 此物件可及，不再
        // 是垃圾。它回來了！
        finalizedList.Add(this);
    }
}
```

`BadClass` 物件執行它的 finalizer 時，它將本身的參考放在一個全域清單中。它讓自己可及。它又活了！你引發的問題讓每個人抓狂。物件已經被終結，因此垃圾回收認為無需再次呼叫它的 finalizer。如果你是要終結一個重生的物件，它不會發生。其次，有些資源不再可用。GC 不會從記憶體移除任何

在 finalizer 佇列中的物件才能觸及的物件，但它或許已經終結它們。如果是這樣，它們幾乎可以確定不再可用。雖然 `BadClass` 擁有的成員還在記憶體中，它們可能已經被處置或終結。語言中沒有任何辦法可以控制終結的順序。你無法讓這種程式可靠的運行，不要嘗試。

除了教學練習外，我從未見過以這種明顯的方式讓物件重生的程式。但我見過嘗試在 finalizer 執行一些工作並最終在 finalizer 呼叫的函式儲存物件的參考而把自己帶回來的程式。這個故事告訴我們要非常仔細的檢視 finalizer 中的程式與 `Dispose` 方法的程式。如果程式執行釋放資源以外的工作，請再檢查，這些動作可能會導致 bug。移開這些動作，要確保 finalizer 與 `Dispose()` 方法只釋放資源且沒有其他動作。

在 managed 環境中，你無需為每個型別撰寫 finalizer；只為儲存 unmanaged 型別或成員有實作 `IDisposable` 的型別撰寫。就算你只需要 `IDisposable` 界面而非 finalizer，還是要實作整個模式。不然你會限制繼承類別必須複雜的實作標準處置方式。依照我所說的標準做法。這樣會讓你、你的類別的使用者、與建構其繼承類別的那些人日子好過一些。

使用泛型

有些文章可能會讓你認為泛型只對集合有用。並非如此。有許多使用泛型的方式。你可以用它們建構界面、處理程序和常見演算法等。

許多討論將 C# 的泛型與 C++ 的模板進行比較，通常會主張其中一個優於另一個。比較 C# 的泛型與 C++ 的模板能幫助你認識語法，但比較到此就應該結束。有些做法對 C++ 的模板更自然，有些對 C# 泛型更自然。但如你會在做法 19 所見，嘗試比較哪一個“比較好”只會妨礙你對兩方的認識。加入泛型需要改變 C# 的編譯器、Just-In-Time（JIT）編譯器、Common Lanaguage Runtime（CLR）。C# 編譯器以你的 C# 程式產生 Microsoft Intermediate Language（MSIL 或 IL）的泛型型別定義。相對的，JIT 編譯器結合泛型型別定義與一組型別參數來產生封閉的泛型型別。CLR 在執行期支援這兩種概念。

泛型型別定義有些成本與好處。有時候，以泛型取代相等的程式會讓程式更小，有時候會更大。是否會遇到這種泛型程式膨脹視使用的型別參數與封閉泛型型別數量而定。

泛型類別定義被完整的編譯成 MSIL 型別。它們攜帶的程式必須完整的對任何可能使用的型別參數有效。此泛型定義被稱為泛型型別定義。所有型別參數都被指定好的一個特定泛型型別的實例被稱為封閉泛型型別（如果只有指定部分參數，稱為開放泛型型別）。

IL 中的泛型是一個真實型別的部分定義。IL 帶有特定完整泛型型別實例化的佔位符。

JIT 編譯器在建構機械碼以於執行期實例化封閉泛型型別時補完該定義。這種做法引發多個封閉泛型型別所增加的程式成本與降低儲存資料所需時間與空間兩者間的交換。

這個過程發生在每個你所建構的型別上，無論是否為泛型。對非泛型型別來說，類別的 IL 與所產生的機械碼間是 1：1 的對應關係。泛型對此交換引進一些新變化。泛型類別為 JIT 編譯時，JIT 編譯器檢視型別變數並根據型別參數發出特定指令。JIT 編譯器執行一些最佳化以將不同型別參數包進相同的機械碼中。JIT 編譯器首先會為所有參考型別建構一個泛型類別的機械碼版本。

下面這些實例化在執行期全都共用相同的程式：

```
List<string> stringList = new List<string>();
List<Stream> OpenFiles = new List<Stream>();
List<MyClassType> anotherList = new List<MyClassType>();
```

C# 編譯器於編譯時強制型別安全性，JIT 編譯器可透過假設型別是正確的而產生更為最佳化版本的機械碼。

至少有一個值型別作為型別參數的封閉泛型型別適用不同的規則。JIT 編譯器為不同的型別參數產生不同的機械指令。因此，下面三個封閉泛型型別具有不同的機械碼：

```
List<double> doubleList = new List<double>();
List<int> markers = new List<int>();
List<MyStruct> values = new List<MyStruct>();
```

這或許很有趣，但你為什麼要在乎？會被用於多種不同參考型別的泛型型別不會影響記憶體大小。所有 JIT 編譯的程式是共用的。但封閉泛型型別有值型別作為參數時，JIT 編譯的程式不是共用的。讓我們更深入此過程以檢視它是如何受影響的。

執行期需要 JIT 編譯的泛型定義（方法或類別）且至少有一個型別參數是值型別時，它會採用有兩個步驟的過程。首先，它建構代表封閉泛型型別的新 IL 類別。我把它簡化了，但基本上就是執行期在泛型定義中將所有的 `T` 以 `int` 或適當的值型別取代。替換後，JIT 編譯編譯的程式成為 x86 指令。這個兩步驟的過程有必要是因為 JIT 編譯器在載入時不會建構整個類別的 x86

程式；相對的，每個方法只會在首次呼叫時被 JIT 編譯。因此在 IL 執行區塊替換然後於被要求時 JIT 編譯所產生的 IL，如同對一般的類別定義所採取的方式。

這表示執行期的記憶體成本以這種方式增加：對每個使用值型別的封閉泛型型別增加一個額外的 IL 定義複製，對每個用於封閉泛型型別的值型別參數的每個方法呼叫再加上另一個額外的機械碼複製。

但使用有值型別參數的泛型有個好處：避免了值型別的 boxing 與 unboxing，因此減少了值型別的程式與資料的大小。此外，型別安全性受到編譯器的保證；所以需要較少的執行期檢查，如此減少了程式碼的大小並提升了效能。還有，如做法 25 所述，以泛型方法替代泛型類別可限制為每個不同的實例化產生的額外 IL 程式數量。只有確實被參考的方法會被實例化。在非泛型類別中定義的泛型方法非 JIT 編譯。

這一章討論使用泛型的各種方式，與解釋如何建構可以讓你節省時間並幫助你建構可用元件的泛型型別及方法。

做法 18 定義最少與足夠的約束

你對型別參數宣告的約束，指定你的類別需要任何用於型別參數的型別之必要行為以完成其工作。沒有滿足所有約束的型別就不能運作，而你提出的每個約束意味著使用你的型別的開發者要做更多的工作，兩者間要取得平衡。正確的選擇視情況而定，但極端做法是不對的。如果你沒有指定任何約束，你在執行期必須做更多的檢查；你會執行更多的型別轉換，可能使用到 reflection 且若開發者誤用你的型別會產生更多的執行期錯誤。指定不需要的約束意味著使用你的型別的開發者要做更多的工作。你的目標是找出中間點，指定你需要而非所有想要的約束。

約束使編譯器能夠在型別參數中超出 `System.Object` 所定義的公開界面中的功能。建構泛型型別時，C# 編譯器必須為泛型型別定義產生有效的 IL。這麼做的時候，就算編譯器對實際會用來替換型別參數的型別只有受限的知識，編譯器還是必須產生有效的組件。若沒有你提供的指引，編譯器只能假設型別的最基本能力：由 `System.Object` 顯露的方法。編譯器無法強制實行任何你已經對你的型別做出的假設。編譯器只知道你的型別必須繼

承自 System.Object（這表示你無法使用指標作為型別參數建構不安全的泛型）。只假設 System.Object 的能力是非常受限的。編譯器會對任何非定義於 System.Object 的東西產生錯誤。這包括若你定義有參數的建構元時會被隱藏的 new T() 等基本的操作。

你使用約束來（對編譯器與開發者）溝通你對泛型型別所做的假設。約束對編譯器溝通你的泛型型別預期 System.Object 的公開界面以外的功能。此溝通以兩種方式幫助編譯器。首先，它對你建構你的泛型型別提出幫助：編譯器斷言任何泛型型別參數帶有你在約束中指定的功能。其次，編譯器會確保使用你的泛型型別的人會定義符合你指定的型別參數。你可以指定型別參數必須是個 struct，或必須是個 class。你可以指定型別必須實作任何數量的界面。你還可以指定基底類別，如此就隱含了類別約束。

另一種方式是執行許多型別轉換與執行期檢查測試。舉例來說，下面的泛型方法沒有對 T 宣告任何約束，因此它必須在使用這些方法前檢查是否具有 IComparable<T> 界面：

```
public static bool AreEqual<T>(T left, T right)
{
    if (left == null)
        return right == null;

    if (left is IComparable<T>)
    {
        IComparable<T> lval = left as IComparable<T>;
        if (right is IComparable<T>)
            return lval.CompareTo(right) == 0;
        else
            throw new ArgumentException(
 “Type does not implement IComparable<T>” ,
            nameof(right));
    }
    else // 失敗
    {
        throw new ArgumentException(
 “Type does not implement IComparable<T>” ,
            nameof(left));
    }
}
```

如果你指定 T 必須實作 IComparable<T> 就簡單多了：

```
public static bool AreEqual2<T>(T left, T right)
    where T : IComparable<T> =>
        left.CompareTo(right) == 0;
```

第二個版本以執行期錯誤交換編譯期錯誤。你寫較少的程式，而編譯期會預防第一版中你必須寫程式處理的執行期錯誤。沒有約束，你沒有回報程式設計師所犯的明顯錯誤的好方法。你必須對你的泛型型別指定必要的約束，沒有這麼做意味著你的類別很容易會被誤用，在開發者猜錯時產生例外或其他執行期錯誤。他們經常猜錯，因為用戶開發者唯一能決定如何使用你的類別的方式是閱讀你的文件。你自己也是個開發者，你知道這有多少可能性。使用約束幫助編譯器強制施行你已經做出的假設。它可以減少執行期錯誤與誤用的數量。

但約束的定義很容易過量。你在泛型型別參數加上越多的約束，你想幫助的用戶開發者就會越少使用你的泛型類別。雖然你必須指定必要的約束，你還必須減少加在你的泛型參數上的約束數量。

有幾種方式可以減少指定的約束數量。最常見的方式之一是確保你的泛型型別不要求不需要的功能。以 IEquatable<T> 為例，它是常見的界面，也是許多開發者在建構新型別時會實作的界面。你可以使用 Equals 重新撰寫 AreEqual 方法：

```
public static bool AreEqual<T>(T left, T right) =>
    left.Equals(right);
```

這個版本的 AreEqual 有趣的地方在於若 AreEqual<T>() 定義於有宣告 IEquatable<T> 約束的泛型類別中，它會呼叫 IEquatable<T>.Equals。不然 C# 編譯器無法假設有 IEquatable<T>。範圍內唯一的 Equals() 是 System.Object.Equals()。

這個例子顯示 C# 泛型與 C++ 模板間主要的差異。在 C# 中，編譯器必須使用約束中指定的資訊來產生 IL。就算實例化所指定的型別具有比較好的方法，除非編譯泛型型別時有指定它，不然不會使用它。

IEquatable<T> 是型別有實作時測試相等性比較有效率的方式。您可以避免實作 System.Object.Equals() 的正確覆寫所需的執行期測試。你避免若用於泛

型型別的型別是個值型別時必要的 **boxing** 與 **unboxing**。如果你很在乎效能，`IEquatable<T>` 也可以避免虛擬方法呼叫的一點成本。

因此要求用戶開發者支援 `IEquatable<T>` 是件好事。但它是否增加了約束？有需要在有個完美但效率稍差的 `System.Object.Equals` 方法時還讓使用你的類別的人實作 `IEquatable<T>` 嗎？我建議如果可用時使用首選的方法（`IEquatable<T>`），但若不可用時再選擇次一級的 API 方法（`Equals()`）。你可以建構根據支援功能過載的內部方法來如此選擇。基本上，這如同這一節開頭所顯示的原始 `AreEqual()` 方法所示。這種方式比較花功夫，但我們會討論如何查詢型別功能，以及使用型別參數中最佳的可用界面，而不會增加用戶開發者的額外工作。

有時候約束對類別的使用產生太多限制，你應該將具有特定的界面或基底類別當做是優點而非必要條件。在這些情況中，你應該讓程式考慮型別參數可能會提供額外優點，但這些功能不一定存在。這正是 `Equatable<T>` 與 `Comparable<T>` 實作的設計。

你可以在泛型與非泛型界面上將此技巧延伸到其他約束 - 例如 `IEnumerable` 與 `IEnumerable<T>`。

另一個要仔細思考的地方是預設建構元的約束。有時候你可以用 `default()` 的呼叫取代 `new` 的呼叫以替換 `new()` 約束。後者是個 C# 的運算元，產生型別的預設值。此運算元對值型別建構預設的 0 位元樣式並對參考型別產生 `null`。因此以 `default()` 取代 `new()` 通常意味著引入類別或值約束。注意 `default()` 與 `new()` 的語意在運用參考型別時非常不同。

你經常會看到在需要型別參數物件預設值的泛型類別中使用 `default()` 的程式。下面是個搜尋第一個符合述詞物件的方法。如果目標物件存在就回傳它，不然就回傳預設值。

```
public static T FirstOrDefault<T>(this IEnumerable<T> sequence,
    Predicate<T> test)
{
    foreach (T value in sequence)
        if (test(value))
            return value;
    return default(T);
}
```

與下面的方法比較，它包裝一個 factory 方法來建構 T 型別物件。若此 factory 方法回傳 null，此方法回傳預設建構元回傳的值。

```
public delegate T FactoryFunc<T>();
public static T Factory<T>(FactoryFunc<T> makeANewT)
where T : new()
{
    T rVal = makeANewT();
    if (rVal == null)
        return new T();
    else
        return rVal;
}
```

使用 defualt() 的方法無需約束。呼叫 new T() 的方法必須指定 new() 約束。還有，由於空測試，與參考型別相比，其行為與值型別非常的不同。值型別不能為空。因此，if 陳述下面的句子絕不會被執行。Factory<T> 還是可以用於值型別，但它在內部檢查 null 值。若 T 是值型別，JIT 編譯器（將 T 以指定型別取代）會移除空測試。

你應該注意 new()、struct、與 class 的約束。前面的例子顯示加入任何約束會產生關於物件如何建構的假設，無論物件的預設值是否為全零或空參考，也無論你的泛型型別參數的實例是否能在泛型類別中建構。理想中，你應該盡量避免這三種約束。仔細思考你的泛型型別是否必須具有這些假設。你通常很少在有合適的替代方案時（例如 default(T)）心中產生假設（“我當然可以呼叫 new T()”）。注意你間接做出假設。刪除哪些非真正必要的。

為對你的用戶程式設計師溝通你的假設，你必須指定約束。但你指定的約束越多，你的類別就越少被使用。建構泛型型別的目的就是要建構盡可能在多種情況下都能有效使用的型別定義。你必須平衡指定約束的安全性與用戶程式設計師處理每個約束所需的工作量。盡量減少你所需的假設，但將你的假設做成約束。

做法 19　使用執行期型別檢查特化泛型演算法

你可以輕鬆的透過指定新型別參數重複使用泛型。新型別參數的新實例化意味著新型別具有類似的功能。

這很棒，因為你要寫的程式比較少。但有時更廣的泛型意味著無法利用特定的演算法。C# 語言的規則考慮到了這個情況。它只需要你辨識你的演算法在型別參數具有更多功能時會更有效，然後撰寫特定的程式。此外，結構建構另一個泛型型別指定不同的約束並不一定可行。泛型實例化是根據物件的編譯期型別而非執行期型別。如果你沒有考慮到這一點，可能會錯過提升效率的機會。

舉例來說，假設你要撰寫提供一系列項目的反序列舉的類別：

```
public sealed class ReverseEnumerable<T> : IEnumerable<T>
{
    private class ReverseEnumerator : IEnumerator<T>
    {
        int currentIndex;
        IList<T> collection;

        public ReverseEnumerator(IList<T> srcCollection)
        {
            collection = srcCollection;
            currentIndex = collection.Count;
        }

        // IEnumerator<T> 成員
        public T Current => collection[currentIndex];

        // IDisposable 成員
        public void Dispose()
        {
            // 沒有實作，但是必要的
            // 因為 IEnumerator<T> 實作 IDisposable
            // 無需受保護的 Dispose()
            // 因為此類別已經封印
        }

        // IEnumerator 成員
```

```
        object System.Collections.IEnumerator.Current
            => this.Current;

        public bool MoveNext() => --currentIndex >= 0;

        public void Reset() => currentIndex = collection.Count;
    }

    IEnumerable<T> sourceSequence;
    IList<T> originalSequence;

    public ReverseEnumerable(IEnumerable<T> sequence)
    {
        sourceSequence = sequence;
    }

    // IEnumerable<T> 成員
    public IEnumerator<T> GetEnumerator()
    {
        // 建構原始序列的拷貝
        // 以將其反序排列
        if (originalSequence == null)
        {
            originalSequence = new List<T>();
            foreach (T item in sourceSequence)
                originalSequence.Add(item);
        }
        return new ReverseEnumerator(originalSequence);
    }

    // IEnumerable 成員
    System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() =>
            this.GetEnumerator();
}
```

此實作假設來自參數的最少資訊量。`ReverseEnumerable` 建構元假設它的輸入參數支援 `IEnumerable<T>`。`IEnumerable<T>` 並不提供隨機存取元素。因此反轉清單的唯一方法如 `ReverseEnumerator<T>.GetEnumerator()` 所示。此處若建構元第一次被呼叫，它會逐個處理輸入序列項目並產生一個拷貝，然後套疊的類別反向處理清單項目。

它是可行的，當輸入集合不支援序列的隨機存取時，那是建構序列的反向列舉的唯一辦法。許多集合支援隨機存取，這個程式在這種情況下是非常沒效率的。當輸入序列支援 IList<T> 時，沒有理由要建構整個序列的拷貝。讓我們利用許多型別同時有實作 IEnuemrable<T> 與 IList<T> 的條件改善這個程式的效率。

唯一的改變在 ReverseEnumerable<T> 類別的建構元中：

```
public ReverseEnumerable(IEnumerable<T> sequence)
{
    sourceSequence = sequence;
    // 若序列沒有實作 IList<T>，
    // originalSequence 為 null，
    // 因此運行沒問題
    originalSequence = sequence as IList<T>;
}
```

為什麼不直接建構使用 IList<T> 的另一個建構元？這在參數的編譯期型別為 IList<T> 時有幫助。但某些情況下不行 - 例如參數的編譯期型別是 IEnumerable<T>，但執行期型別實作 IList<T>。為捕捉這些狀況，你應該同時提供執行期檢查與編譯期過載。

```
public ReverseEnumerable(IEnumerable<T> sequence)
{
    sourceSequence = sequence;
    // 若序列沒有實作 IList<T>，
    // originalSequence 為 null，
    // 因此運行沒問題
    originalSequence = sequence as IList<T>;
}

public ReverseEnumerable(IList<T> sequence)
{
    sourceSequence = sequence;
    originalSequence = sequence;
}
```

IList<T> 可使用比 IEnumerable<T> 更有效率的演算法。你無需強制此類別的使用者提供更多功能，但若有則可以利用其更好的功能。

此改變可處理大部分的情況，但還是有集合實作 ICollection<T> 而沒有實作 IList<T>。在這種情況下還是無效率。再看一下 ReverseEnumerable<T>.GetEnumerator() 方法：

```
public IEnumerator<T> GetEnumerator()
{
    // 建構原始序列的拷貝
    // 以將其反序排列
    if (originalSequence == null)
    {
        originalSequence = new List<T>();
        foreach (T item in sourceSequence)
            originalSequence.Add(item);
    }
    return new ReverseEnumerator(originalSequence);
}
```

此程式建構輸入序列的拷貝在來源集合實作了 ICollection<T> 時會執行的比所需更慢。下面的方法加上用於初始化最終儲存體的 Count 屬性：

```
public IEnumerator<T> GetEnumerator()
{
    // 字串是特例：
    if (sourceSequence is string)
    {
        // 注意型別轉換是因為 T 在
        // 編譯時可能不是 char
        return new
            ReverseStringEnumerator(sourceSequence as string)
            as IEnumerator<T>;
    }
    // 建構原始序列的拷貝
    // 以將其反序排列
    if (originalSequence == null)
    {
        if (sourceSequence is ICollection<T>)
        {
            ICollection<T> source =
                sourceSequence as ICollection<T>;
            originalSequence = new List<T>(source.Count);
        }
        else
```

```
            originalSequence = new List<T>();
        foreach (T item in sourceSequence)
            originalSequence.Add(item);
    }
    return new ReverseEnumerator(originalSequence);
}
```

此處顯示的程式類似從輸入序列建構清單的 `List<T>` 建構元：

```
List<T>(IEnumerable<T> inputSequence);
```

結束這一節討論之前我想談另一個要點。你會注意到我在 `ReverseEnumerable<T>` 中製作的測試是對執行期參數的執行期測試。這表示你預期查詢額外功能的執行期成本。在大部分情況下，執行期測試的成本大幅小於複製元素的成本。

你或許會認為我們已經看過 `ReverseEnumerable<T>` 類別所有的可能運用方式，其實還有一種：`string` 類別。`string` 提供隨機存取字元的方法，像是 `IList<char>`，但 `string` 沒有實作 `IList<char>`。使用更針對性的方法需要在你的泛型類別中撰寫更針對性的程式。套疊在 `ReverseEnumerable<T>` 中的 `ReverseStringEnumerator` 類別（如下所示）很直截了當。注意此建構元使用字串的 `Length` 參數，其他方法幾乎與 `ReverseEnumerator<T>` 類別中的一樣。

```
private sealed class ReverseStringEnumerator :
    IEnumerator<char>
{
    private string sourceSequence;
    private int currentIndex;

    public ReverseStringEnumerator(string source)
    {
        sourceSequence = source;
        currentIndex = source.Length;
    }

    // IEnumerator<char> 成員
    public char Current => sourceSequence[currentIndex];

    // IDisposable 成員
    public void Dispose()
    {
```

```
        // 沒有實作，但是必要的
        // 因為 IEnumerator<T> 實作 IDisposable
    }

    // IEnumerator 成員
    object System.Collections.IEnumerator.Current
        => sourceSequence[currentIndex];

    public bool MoveNext() => --currentIndex >= 0;

    public void Reset() => currentIndex = sourceSequence.
    Length;
}
```

要完成此特定的實作，ReverseEnumerable<T>.GetEnumerator() 需要檢視適當的型別並建構正確的列舉型別：

```
public IEnumerator<T> GetEnumerator()
{
    // 字串是特例：
    if (sourceSequence is string)
    {
        // 注意型別轉換是因為 T 在
        // 編譯時可能不是 char
        return new ReverseStringEnumerator(sourceSequence as string)
            as IEnumerator<T>;
    }
    // 建構原始序列的拷貝
    // 以將其反序排列
    if (originalSequence == null)
    {
        if (sourceSequence is ICollection<T>)
        {
            ICollection<T> source = sourceSequence as
            ICollection<T>;
            originalSequence = new List<T>(source.Count);
        }
        else
            originalSequence = new List<T>();
        foreach (T item in sourceSequence)
            originalSequence.Add(item);
    }
    return new ReverseEnumerator(originalSequence);
}
```

如同前面，其目標是要將特殊化實作隱藏在泛型類別中。這會多花一些功夫，但 `string` 類別特殊化需要完全的分離內部類別實作。

你還會注意到 `GetEnumerator()` 的實作需要在使用 `ReverseStringEnumerator` 時進行型別轉換。在編譯期，`T` 可以是任何東西，因此它可能不是個 `char`。此型別轉換是安全的；程式的唯一路徑確保 `T` 是個 `char`，因為序列是字串。這樣 OK，因為它被安全的隱藏在類別中，且不會污染公開界面。如你所見，泛型的存在不會完全消滅你比編譯器知道的更多的情況。

這個小範例顯示如何以最少的約束建構泛型類別並還能指定在型別參數支援加強功能時提供更好的實作的約束。這提供最大重複使用與特定演算法最佳實作之間的最佳平衡。

做法 20　以 IComparable<T> 與 IComparer<T> 實作排序關係

你的型別需要排序關係以描述集合應該如何儲存與搜尋。.NET Framework 定義了兩個描述排序關係的界面：`IComparable<T>` 與 `IComparer<T>`。`IComparable<T>` 定義你的型別的自然順序。型別實作 `IComparer` 以描述其他排序。你可以自定關係運算子（`<`、`>`、`<=`、`>=`）的實作以提供型別專屬的比較來避免界面實作中某些執行期的無效率。這一節討論如何實作排序關係以讓核心 .NET Framework 透過其定義的界面排序你的型別並讓其他使用者從這些操作中獲得最好的效能。

`IComparable` 界面有一個方法：**`CompareTo()`**。此方法依循 C 函式庫的 `strcmp` 函式所建立的傳統：若目前物件小於比較物件則回傳值小於 0，若相等則回傳 0，若目前物件大於比較物件則回傳值大於 0。`IComparable<T>` 用於 .NET 較新的 API。但某些舊的 API 使用傳統的 `IComparable` 界面。因此，實作 `IComparable<T>` 時，你也應該實作 `IComparable`。`IComparable` 取用 `System.Object` 型別參數。你應該對此函式的參數進行執行期檢查。每次執行比較時，你必須重新解譯參數的型別：

```
public struct Customer : IComparable<Customer>, IComparable
{
    private readonly string name;
```

```
    public Customer(string name)
    {
        this.name = name;
    }

    // IComparable<Customer> 成員
    public int CompareTo(Customer other) =>
        name.CompareTo(other.name);

    // IComparable 成員
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
            throw new ArgumentException(
                 "Argument is not a Customer" , "obj" );
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }
}
```

注意 IComparable 在此程式中明確的實作。這確保唯一會呼叫此物件型別版本的 CompareTo() 的程式是為以前的界面撰寫的。傳統版 IComparable 有很多原因不受歡迎。你必須檢查參數的執行期型別。不正確的程式會合法的以任何東西作為 CompareTo 方法的參數呼叫它。有些參數必須 boxed 與 unboxed 以進行比較。這對每個比較是額外的執行期消耗。對集合排序平均需要使用 ICompareable.Compare 方法進行 n log(n) 個比較。每個比較會導致三個 boxing 與 unboxing 操作。對 1,000 個元素的陣列會有 20,000 個以上的 boxing 與 unboxing 操作：n log(n) 在 7,000 左右，每個比較有三個 box 與 unbox 操作。

你或許會懷疑為何要實作非泛型的 IComparable 界面。有兩個原因。首先是向後相容性。你的型別會與好幾個版本以前的 .NET 2.0 之前建構的程式互動。某些 Base Class Library 類別（WinForms 或 ASP.NET Web Forms）需要向後相容 1.0 實作。這表示要支援非泛型界面。

現在由於傳統的 IComparable.CompareTo() 是個目前的界面實作，它只能透過 IComparable 參考呼叫。你的 Customer struct 的使用者能夠進行型別安全的比較，而不安全的比較是無法存取的。下面的錯誤無法通過編譯：

```
Customer c1;
Employee e1;
if (c1.CompareTo(e1) > 0)
    Console.WriteLine( “Customer one is greater” );
```

它無法編譯是因為參數對公開的 Customer.CompareTo(Customer right) 方法是錯的。IComparable.CompareTo(object right) 方法無法存取。你只能明確的轉換參考來存取 IComparable 方法：

```
Customer c1;
Employee e1;
if (c1.CompareTo(e1) > 0)
    Console.WriteLine( “Customer one is greater” );
```

實作 IComparable 時，使用目前的界面實作並通過強型別過載。強型別過載可提升效能並減少 CompareTo 方法誤用的可能性。你不會看到 .NET Framework 使用的 Sort 函式中的所有好處，因為它還是會透過界面指標存取 CompareTo()，但知道進行比較的物件雙方型別的程式會有較好的效能。

我們會對 Customer struct 做出最後一個小修改。C# 語言讓你過載標準關係運算子。它們應該利用型別安全的 CompareTo() 方法：

```
public struct Customer : IComparable<Customer>, IComparable
{
    private readonly string name;

    public Customer(string name)
    {
        this.name = name;
    }

    // IComparable<Customer> 成員
    public int CompareTo(Customer other) =>
        name.CompareTo(other.name);

    // IComparable 成員
    int IComparable.CompareTo(object obj)
    {
        if (!(obj is Customer))
            throw new ArgumentException(
                 “Argument is not a Customer”,  “obj” );
```

```
        Customer otherCustomer = (Customer)obj;
        return this.CompareTo(otherCustomer);
    }

    // 關係運算子
    public static bool operator <(Customer left,
        Customer right) =>
            left.CompareTo(right) < 0;
    public static bool operator <=(Customer left,
        Customer right) =>
            left.CompareTo(right) <= 0;
    public static bool operator >(Customer left,
        Customer right) =>
            left.CompareTo(right) > 0;
    public static bool operator >=(Customer left,
        Customer right) =>
            left.CompareTo(right) >= 0;
}
```

這就是客戶的標準順序：以名稱排序。之後你需要建構以收入排序的客戶報表。你還是需要 `Customer struct` 定義的依名稱排序功能。.NET Framework 引進泛型之後的大部分 API 會要求 `Comparison<T>` 以執行其他排序。在提供其他比較排序的 `Customer` 型別中建構靜態屬性是簡單的。舉例來說，下面的程式比較兩個客戶的收入：

```
public static Comparison<Customer> CompareByRevenue =>
    (left,right) => left.revenue.CompareTo(right.revenue);
```

舊的函式庫會要求這種功能使用 `IComparer` 界面。`IComparer` 提供型別標準的其他排序方式而不需要泛型。.NET Framework 的 1.x 版的類別函式庫中在 `IComparable` 上運行的方法提供了透過 `IComparer` 排序物件的過載。由於 `Customer struct` 是你寫的，你可以結構此新的類別（`RevenueComparer`）作為 `Customer struct` 內私用的套疊類別。它透過 `Customer struct` 中的一個靜態屬性顯露：

```
public struct Customer : IComparable<Customer>, IComparable
{
    private readonly string name;
    private double revenue;
```

```
public Customer(string name, double revenue)
{
    this.name = name;
    this.revenue = revenue;
}

// IComparable<Customer> 成員
public int CompareTo(Customer other)
{
    return name.CompareTo(other.name);
}

// IComparable 成員
int IComparable.CompareTo(object obj)
{
    if (!(obj is Customer))
        throw new ArgumentException(
             “Argument is not a Customer”, “obj”);
    Customer otherCustomer = (Customer)obj;
    return this.CompareTo(otherCustomer);
}

// 關係運算子
public static bool operator <(Customer left,
Customer right)
{
    return left.CompareTo(right) < 0;
}
public static bool operator <=(Customer left,
Customer right)
{
    return left.CompareTo(right) <= 0;
}
public static bool operator >(Customer left,
Customer right)
{
    return left.CompareTo(right) > 0;
}
public static bool operator >=(Customer left,
Customer right)
{
    return left.CompareTo(right) >= 0;
}
```

```
    private static Lazy<RevenueComparer> revComp =
        new Lazy<RevenueComparer>(() => new RevenueComparer());
    public static IComparer<Customer> RevenueCompare
        =>  revComp.Value;

    public static Comparison<Customer> CompareByRevenue =>
        (left,right) => left.revenue.CompareTo(right.revenue);

    // 比較客戶收入的類別
    // 透過界面指標使用
    // 因此僅提供界面覆寫
    private class RevenueComparer : IComparer<Customer>
    {
        // IComparer<Customer> 成員
        int IComparer<Customer>.Compare(Customer left,
            Customer right) =>
            left.revenue.CompareTo(right.revenue);
    }
}
```

崁入 `RevenueComparer` 的 `Customer struct` 版本讓你依名稱、自然順序將客戶排序，並透過顯露實作 `IComparer` 界面的類別來以收入對客戶排序。如果你無法取得 `Customer` 類別的原始碼，你還是可以提供使用任何公開屬性對客戶排序的 `IComparer`。你只應該在無法取得類別的原始碼時使用這個方法，例如需要 .NET Framework 中類別的其他排序方式時。

這一節沒有提到 `Equals()` 或 `==` 運算子。排序關係與相等性是不同的操作。排序關係不需要實作相等性比較。事實上，參考型別通常根據物件內容實作排序，而根據物件身份（identity）實作相等性。就算 `Equals()` 回傳 `false`，`CompareTo()` 還是可以回傳 0。這是完全合法的。相等性與排序關係不一定要一樣。

`IComparable` 與 `IComparer` 是你的型別提供排序關係的標準機制。`IComparable` 應該用於最自然的排序。實作 `IComparable` 時，你應該與我們的 `IComparable` 排序一致的過載比較運算子（`<`, `>`, `<=`, `>=`）。`IComparable.CompareTo()` 使用 `System.Object` 參數，因此你也應該提供指定型別的 `CompareTo()` 方法的過載。`IComparer` 可用於提供其他排序或型別並未提供而你所需要的排序。

做法 21　建構支援 Disposable 型別參數的泛型類別

約束為你與你的類別的使用者做兩件事。首先，約束將執行期錯誤轉換成編譯期錯誤。其次，約束為你的類別的使用者提供清楚的文件來描述建構你的參數化型別的實例時預期什麼。但你無法以約束指定型別參數不能做什麼。通常你不在意型別參數具有預期外的什麼能力。但在型別參數實作 IDisposable 這種特殊情況下，你有些工作要做。

能展示此問題的真實世界範例太過於複雜，因此我編造了一個簡單的範例來顯示此問題如何發生與如何改正。此問題發生於你有個泛型方法需要建構並使用其中一個方法的型別參數的實例：

```
public interface IEngine
{
    void DoWork();
}

public class EngineDriverOne<T> where T : IEngine, new()
{
    public void GetThingsDone()
    {
        T driver = new T();
        driver.DoWork();
    }
}
```

若 T 實作 IDisposable 可能會引發資源洩漏。任何情況下，建構 T 型別的區域變數時必須檢查 T 是否有實作 IDisposable，若有則需要正確的處置：

```
public void GetThingsDone()
{
    T driver = new T();
    using (driver as IDisposable)
    {
        driver.DoWork();
    }
}
```

如果你沒有看過這種在 using 陳述中的型別轉換可能會有些困惑，但它沒問題。編譯器會建構儲存 driver 轉換為 IDisposable 的參考的隱藏區域變數。若 T 沒有實作 IDisposable，則此區域變數的值為 null。在這種情況下，編譯器不會呼叫 Dispose()，因為它在執行額外工作前會檢查空值。但在 T 實作 IDisposable 的情況下，編譯器會在離開 unsing 區塊時產生對 Dispose() 的呼叫。

這是相當簡單的做法：將型別參數的區域實例包裝在 using 陳述中。你必須使用此處顯示的型別轉換，因為 T 可能有或沒有實作 IDisposable。

你的泛型類別需要結構並使用型別參數的實例作為成員變數時事情會變得更複雜。你的泛型類別現在有一個對可能有實作 IDisposable 的型別的參考。

這表示你的泛型型別必須實作 IDisposable。你必須讓你的類別檢查資源是否實作 IDisposable，若有，則必須處置該資源：

```
public sealed class EngineDriver2<T> : IDisposable
    where T : IEngine, new()
{
    // 建構代價很高，因此初始化為空
    private Lazy<T> driver = new Lazy<T>(() => new T());
    public void GetThingsDone() =>
        driver.Value.DoWork();

    // IDisposable 成員
    public void Dispose()
    {
        if (driver.IsValueCreated)
        {
            var resource = driver.Value as IDisposable;
            resource?.Dispose();
        }
    }
}
```

你的類別在這一輪過程中帶了很多包袱。你加入實作 IDisposable 的工作。其次，你得對此類別加入 sealed 關鍵字。若非如此則需實作完整的 IDisposable 模式以讓繼承類別使用你的 Dispose() 方法（見 Krzysztof Cwalina 與 Brad Abrams 的 Framework Design Guidelines 或做法 17）。封住

類別表示無需額外的工作，但限制類別的使用者不能從你的類別繼承出新的型別。

最後，注意此類別不能保證你不會呼叫 **driver** 的 `Dispose()` 一次以上。這是受允許的，任何實作 `IDisposable` 的型別必須支援多次呼叫 `Dispose()`。這是因為 `T` 沒有類別約束，因此你不能在離開 `Dispose` 方法前設定 `driver` 為 `null`（記得值型別不能設為 `null`）。

實務上，你通常可以透過改變泛型類別的界面來避免。你可以將 `Dispose` 的責任移到泛型類別外，並將 `new()` 約束移到此泛型類別外：

```
public sealed class EngineDriver<T> where T : IEngine
{
    // 建構代價很高，因此初始化為空
    private T driver;
    public EngineDriver(T driver)
    {
        this.driver = driver;
    }

    public void GetThingsDone()
    {
        driver.DoWork();
    }
}
```

當然，前面程式中的註解表示建構 `T` 物件的代價可能很高。最後一個版本忽略了這個考量。最後，如何解決這個問題視應用程式設計的其他因素而定。但有件事是確定的：如果你建構的泛型類別的型別參數描述的任何型別實例，你必須考慮到這些型別可能有實作 `IDisposable`。你必須防衛性的撰寫程式並確保這些物件離開範圍後不會洩漏資源。

有時候你可以重構程式以讓它不會建構這些實例。其他情況下最好的設計是建構並使用區域變數，撰寫程式在必要時處置它們。最後，設計可能會呼叫型別參數的懶實例建構並在泛型類別中實作 `IDisposable`。這更花功夫，但若你想要建構實用的類別時這是必要的工作。

做法 22 支援泛型的共變數與反變數

型別變化，特別是共變數（covariance）與反變數（contravariance），定義一個型別的值可以轉換成另一個型別的值的條件。有可能時，你應該修飾泛型界面並委派定義以支援泛型的共變數與反變數。這麼做可以讓你的 API 以更多方式安全的使用。如果不能替換型別，則被稱為不變（invariant）。

許多開發者遇過型別變化但不是非常理解。共變數與反變數是透過型別參數的相容性推斷兩個泛型型別相容性的能力。泛型型別 `C<T>` 與 `T` 共變數，若我們能夠從 `X` 可轉換成 `Y` 推斷 `C<X>` 可轉換成 `C<Y>`。`C<T>` 與 `T` 共變數，若我們能夠從 `Y` 可轉換成 `X` 推斷 `C<X>` 可轉換成 `C<Y>`。

大部分開發者認為你應該能夠使用 `IEnumerable<MyDerivedType>` 與具有 `IEnumerable<Object>` 參數的方法。你預期如果一個方法回傳 `IEnumerable<MyDerivedType>`，你就可以指派它給型別為 `IEnumerable <object>` 的變數。不是這樣的。在 C# 4.0 之前，所有泛型型別都是不變的。這表示有很多次當你合理地期望泛型的共變數或反變數時，只會被編譯器告知你的程式是無效的。陣列是被共變數處理的，但陣列不支援安全共變數。從 C# 4.0 之後，新的關鍵字可讓你使用泛型的共變數與反變數。這讓泛型更有用，特別是若你記得在泛型界面與 delegate 引用 in 與 out 參數時。

讓我們從認識陣列共變數的問題開始。以下面的類別階層為例：

```
abstract public class CelestialBody :
    IComparable<CelestialBody>
{
    public double Mass { get; set; }
    public string Name { get; set; }
    // 省略
}

public class Planet : CelestialBody
{
    // 省略
}

public class Moon : CelestialBody
{
```

```
    // 省略
}

public class Asteroid : CelestialBody
{
    // 省略
}
```

此方法以共變數方式安全的處理 CelestialBody 物件的陣列：

```
public static void CoVariantArray(CelestialBody[] baseItems)
{
    foreach (var thing in baseItems)
        Console.WriteLine( “{0} has a mass of {1} Kg” ,
            thing.Name, thing.Mass);
}
```

此方法也是共變數方式處理 CelestialBody 物件的陣列，但不安全。指派陳述會拋出例外：

```
public static void UnsafeVariantArray(CelestialBody[] baseItems)
{
    baseItems[0] = new Asteroid
        { Name =  “Hygiea” , Mass = 8.85e19 };
}
```

指派繼承類別的陣列到基底型別陣列的變數會有相同的問題：

```
CelestialBody[] spaceJunk = new Asteroid[5];
spaceJunk[0] = new Planet();
```

將集合當做共變數表示當兩個型別有繼承關係時，你可以想像兩個型別的陣列間有類似的繼承關係。這不是嚴格的定義，但可以幫助你記得。Planet 可以傳給任何預期 CelestialBody 的方法。這是因為 Planet 繼承自 CelestialBody。同樣的，你可以將 Planet[] 傳給預期 CelestialBody[] 的方法。但如上面的範例所示，它不一定以你預期的方式運作。

首度引進泛型時，此問題以相當嚴格的方式處理。泛型總是被 C# 編譯器以不變方式處理。泛型型別必須完全相符。但在 C# 4.0 之後，你可以修飾泛型界面以讓它們以共變數或反變數方式處理。讓我們先討論泛型共變數，然後再討論反變數。

此方法可用 List<Planet> 呼叫：

```
public static void CovariantGeneric
    (IEnumerable<CelestialBody> baseItems)
{
    foreach (var thing in baseItems)
        Console.WriteLine( “{0} has a mass of {1} Kg” ,
            thing.Name, thing.Mass);
}
```

這是因為加入 IEnumerable<T> 來限制 T 在其界面只輸出位置：

```
public interface IEnumerable<out T> : IEnumerable
{
    new IEnumerator<T> GetEnumerator();
}
public interface IEnumerator<out T> :
    IDisposable, IEnumerator
{
    new T Current { get; }
    // MoveNext(), Reset() 繼承自 IEnumerator
}
```

我在此加上 IEnumerable<T> 與 IEnumerator<T> 的定義是因為 IEnumerator<T> 有重要的限制。注意 IEnumerator<T> 現在以 out 修飾詞修飾型別參數 T。這強制編譯器限制 T 輸出位置。輸出位置限制為函式回傳值、屬性的 get 存取與特定 delegate 位置。

因此，使用 IEnumerable<out T> 讓編譯器知道你會檢視序列中的每個 T 但不會修改來源序列的內容。在此例中將每個 Planet 當做 CelestialBody 處理是可行的。

IEnumerable<T> 可為共變數僅因為 IEnumerator<T> 也是共變數。若 IEnumerable<T> 回傳沒有被宣告為共變數的界面，編譯器會產生一個錯誤。

但替換清單中第一個項目的方法在使用泛型時會是不變的：

```
public static void InvariantGeneric(
    IList<CelestialBody> baseItems)
{
    baseItems[0] = new Asteroid
```

```
        { Name = "Hygiea", Mass = 8.85e19 };
}
```

因為 `IList<T>` 在 `T` 沒有以 `in` 或 `out` 修飾，你必須使用完全相符的型別。

當然，你也可以建構反變數泛型界面與 **delegate**。將 `in` 修飾詞以 `out` 修飾詞替換。這會指示編譯器型別參數只能出現在輸入位置。**.NET Framework** 對 `IComparable<T>` 界面加上了 `in` 修飾詞：

```
public interface IComparable<in T>
{
    int CompareTo(T other);
}
```

這表示你可以讓 `CelestialBody` 實作使用物件的 **mass** 的 `IComparable<T>`。它可以比較兩個 `Planet`、一個 `Planet` 與一個 `Moon`、一個 `Moon` 與一個 `Asteroid` 或是其他組合。比較兩個物件的 **mass** 是有效的比較。

你會注意 `IEquatable<T>` 是不變的。定義上，`Planet` 不能等於 `Moon`。它們是不同的型別，因此沒有意義。兩個物件若相等則必須是相同型別。

反變數的型別參數只能出現在方法參數，與 **delegate** 參數中某些位置。

最後來討論共變數與反變數的 **delegate** 參數。**delegate** 定義可以是共變數或反變數的。它通常相當簡單：方法參數是反變數（`in`），而方法回傳型別是共變數（`out`）。**.NET Base Class Library**（**BCL**）更新了許多 **delegate** 定義以引用變化：

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg);
public delegate TResult Func<in T1, T2, out TResult>(T1 arg1, T2 arg2);
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
public delegate void Action<in T1, in T2, T3>(T1 arg1, T2 arg2, T3 arg3);
```

同樣的，這或許不是很難。但加在一起時就變得複雜了。你已經看到你不能從共變數界面回傳不變界面。你也不能使用 **delegate** 處理共變數與反變數限制。

如果不小心，**delegate** 很容易“反轉”界面中的共變數與反變數。下面是幾個例子：

```
public interface ICovariantDelegates<out T>
{
    T GetAnItem();
    Func<T> GetAnItemLater();
    void GiveAnItemLater(Action<T> whatToDo);
}

public interface IContravariantDelegates<in T>
{
    void ActOnAnItem(T item);
    void GetAnItemLater(Func<T> item);
    Action<T> ActOnAnItemLater();
}
```

我對這些界面中的方法以特別的命名來顯示為何 **delegate** 的共變數與反變數如此運作。仔細檢視 `ICovariantDelegate` 界面的定義。`GetAnItemLater()` 是方便取得項目的一個方法。之後呼叫方可叫用此方法回傳的 `Func<T>` 以取得值。`T` 還是在輸出位置。這或許還合理。`GiveAnItemLater()` 方法或許更混亂。`GiveAnItemLater()` 在你呼叫它時取用一個接受 `T` 物件的 **delegate**。因此，就算 `Action<in T>` 是個共變數，它在 `ICovariantDelegate` 界面中的位置意味著它實際上是一個從 `ICovariantDelegate <T>` 實作物件回傳 `T` 物件的方法。它看起來應該是反變數，但對於界面是共變數。

`IContravariantDelegate<T>` 也差不多，但顯示出如何在反變數界面中使用 **delegate**。同樣的，`ActOnAnItem` 方法應該是明顯的。`ActOnAnItemLater()` 方法稍微複雜一點。你在之後回傳接受 `T` 物件的方法。上一個方法同樣的可能會導致一些混淆。它與其他界面的概念相同。`GetAnItemLater()` 方法接受回傳 `T` 物件的方法。雖然 `Func<out T>` 被宣告為共變數，它的用途是將輸入帶給實作 `IContravariantDelegate` 的物件。它的用處與 `IContravariantDelegate` 相反。

確實描述共變數與反變數如何運作會很複雜，幸好語言現在支援以 `in`（反變數）與 `out`（共變數）修飾詞修飾泛型界面與 **delegate**。你應該盡可能以 `in` 與 `out` 修飾詞修飾界面與 **delegate**，然後編譯器可以糾正任何可能的變化誤用。編譯器會捕捉你的界面與 **delegate** 定義中的錯誤，且它會偵測到型別的誤用。

做法 23　使用 delegate 定義型別參數的方法約束

乍看之下，C# 的約束機制似乎過於受限：你只能指定單一基底類別、界面、類別或 struct、與無參數的建構元。還少了很多。你無法指定靜態方法（包括運算子），且無法指定其他建構元。從某個角度看，語言定義的約束可以滿足每一個約定。你可以使用參數來定義建構 T 物件的 IFactory<T> 界面，也可以定義 IAdd<T> 來加 T 物件並使用定義於 T 的靜態運算子 "+"（或使用其他方法加 T 物件）。但這不是解決此問題的好辦法，有太多額外的工作會讓你的基本設計變得困難。

以 Add() 為例。若你的泛型類別需要 T 的 Add() 方法，你必須執行幾項工作：建構 IAdd<T> 界面。對此界面寫程式。目前這樣還好。但使用你的泛型類別的開發者會需要執行更多的工作。他們必須結構實作 IAdd<T> 的類別、定義 IAdd<T> 所需的方法、然後為你的泛型類別定義指定封閉的泛型類別。為了呼叫一個方法，而讓開發者必須建構符合 API 格式的類別。這樣會使得想要使用你的類別的開發者產生麻煩與混亂。

但不一定得如此。你可以指定符合你的泛型類別需要呼叫的方法的 delegate 格式。它不會增加泛型類別作者的工作，反而可以讓使用泛型類別的開發者省下不少工作。

以下是需要將兩個 T 型別物件相加的方法的泛型類別定義。你甚至無需定義 delegate；System.Func<T1, T2, TOutput> 這個 delegate 就符合你的需求。以下是將兩物件相加的泛型方法，使用了實作 Add 的方法：

```
public static class Example
{
    public static T Add<T>(T left, T right,
        Func<T, T, T> AddFunc) =>
    AddFunc(left, right);
}
```

使用你的類別的開發者，可使用型別界面與 lambda 表示式來定義你的泛型類別必須呼叫 AddFunc() 時要呼叫的方法。你可以使用如下的 lambda 表示式呼叫 Add 這個泛型方法：

```
int a = 6;
int b = 7;
int sum = Example.Add(a, b, (x, y) => x + y);
```

C# 編譯器推斷型別並從 lambda 表示式回傳值。C# 編譯器建構一個私用靜態方法回傳兩個整數和。此方法的名稱由編譯器產生。編譯器還會建構一個 `Func<T,T,T>` 的 delegate 物件並將方法指標指向編譯器產生的方法。最後，編譯器將該 delegate 傳給泛型的 `Example.Add()` 方法。

我使用 lambda 語法指定定義 delegate 的方法以顯示為何你應該建構根據 delegate 的界面約定。此程式是刻意製作的範例，但重點在於概念。使用界面定義約束有困難時，你可以定義符合你需求的方法格式與 delegate 型別，然後將 delegate 的實例加入泛型方法的參數清單。使用你的類別的開發者可使用 lambda 表示式來定義該方法，以更清楚的方式來撰寫較少的程式。使用你的類別的開發者必須建構定義其所需功能的 lambda 表示式，此時無需額外的程式來支援界面約束的語法。

還有，你會想要使用以 delegate 為基礎的約定來建構操作序列的演算法。想象你必須撰寫組合多個探測器取樣並回傳這兩個序列至單一序列點的程式。

你的 `Point` 類別或許像這樣：

```
public class Point
{
    public double X { get; }
    public double Y { get; }
    public Point(double x, double y)
    {
        this.X = x;
        this.Y = y;
    }
 }
```

從你的裝置讀取的值是 `List<double>` 序列。你需要透過重複以成對的 (X,Y) 呼叫 `Point(double,double)` 建構元來產生序列的方法。`Point` 是個不可變型別。你不能呼叫預設建構元然後設定 X 與 Y 屬性。但你也不能建構指定建構元參數的約束。解決方案是定義取用兩個參數並回傳一個點的 delegate。同樣的，它已經在 .NET Framework 3.5 中：

```
delegate TOutput Func<T1, T2, TOutput>(T1 arg1, T2 arg2);
```

此例中，T1 與 T2 具有相同型別：double。Base Class Library 中一個建構輸出序列的泛型方法與它有點像：

```
public static IEnumerable<TOutput> Zip<T1, T2, TOutput>
    (IEnumerable<T1> left, IEnumerable<T2> right,
    Func<T1, T2, TOutput> generator)
{
    IEnumerator<T1> leftSequence = left.GetEnumerator();
    IEnumerator<T2> rightSequence = right.GetEnumerator();
    while (leftSequence.MoveNext() && rightSequence.MoveNext())
    {
        yield return generator(leftSequence.Current,
            rightSequence.Current);
    }
    leftSequence.Dispose();
    rightSequence.Dispose();
}
```

Zip 列舉兩個輸入序列，對輸入序列中的每一對項目呼叫產生 delegate，回傳新建構的 Point 物件（見做法 29 與 33）。此 delegate 的約定指定你需要以兩個不同輸入建構輸出型別的方法。注意！ Zip 被定義成兩個輸入型別無需是相同的型別。你可以使用相同的方法建構不同型別的鍵 / 值對。你只需不同的 delegate。

你可以這樣呼叫 Zip：

```
double[] xValues = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
double[] yValues = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

List<Point> values = new List<Point>(
    Utilities.Zip(xValues, yValues, (x, y) =>
    new Point(x, y)));
```

與之前相同，編譯器產生私用靜態方法、使用對該方法的參考實例化 delegate 物件、將該 delegate 物件傳給 Merge() 方法。

一般情況下，你的泛型類別需要呼叫的任何方法可以被特定的 delegate 取代。前兩個範例包含一個為泛型方法呼叫的 delegate。這種做法在你的型別

於多個位置需要 **delegate** 方法時也可行。你可以建構其中一個類別型別參數為 **delegate** 的泛型類別，然後在建構此類別的實例時，指派類別的一個成員給該型別的 **delegate**。

下面的簡單範例有個從串流讀取點呼叫 **delegate** 將文字輸入轉換成 `Point` 的 **delegate**。第一個步驟是對從檔案讀取點的 `Point` 類別加上一個建構元：

```
public Point(System.IO.TextReader reader)
{
    string line = reader.ReadLine();
    string[] fields = line.Split( ‘,’ );
    if (fields.Length != 2)
        throw new InvalidOperationException(
             “Input format incorrect” );
    double value;
    if (!double.TryParse(fields[0], out value))
        throw new InvalidOperationException(
             “Could not parse X value” );
    else
        X = value;

    if (!double.TryParse(fields[1], out value))
        throw new InvalidOperationException(
             “Could not parse Y value” );
    else
        Y = value;
}
```

建構集合類別需要一些手段。你不能約束你的泛型型別引入取用參數的建構元。但你可以要求方法執行你想要的工作。你定義從檔案建構 `T` 的 **delegate** 型別：

```
public delegate T CreateFromStream<T>(TextReader reader);
```

接下來，你建構容器類別，而此容器的建構元以該 **delegate** 型別的實例作為參數：

```
public class InputCollection<T>
{
    private List<T> thingsRead = new List<T>();
```

```
    private readonly CreateFromStream<T> readFunc;

    public InputCollection(CreateFromStream<T> readFunc)
    {
        this.readFunc = readFunc;
    }

    public void ReadFromStream(TextReader reader) =>
        thingsRead.Add(readFunc(reader));

    public IEnumerable<T> Values => thingsRead;
}
```

將一個 `InputCollection` 實例化時，你提供 **delegate**：

```
var readValues = new InputCollection<Point>(
    (inputStream) => new Point(inputStream));
```

此範例簡單到你或許會改為非泛型類別。但此技巧可幫助你建構依靠無法由正常約束指定行為的泛型型別。

通常，表示你的設計的最佳方式，是使用類別約束或界面約束來指定你的約束。**.NET BCL** 在很多地方都是這麼做，預期你的型別實作了 `IComparable<T>`、`IEquatable<T>` 或 `IEnumerable<T>`。這是正確的設計選擇，因為這些是常見的界面且許多演算法都有使用。還有，它們是清楚的表示為界面：實作 `IComparable<T>` 的型別宣告它支援排序關係。實作 `IEquatable<T>` 的型別宣告它支援相等性。

但若你需要建構自定界面約定來支援一個特定的泛型方法或類別，你會發現你的使用者使用 **delegate** 來以方法約束指定約定會比較容易。你的泛型型別會容易使用，而呼叫它的程式會易於理解。無論有無運算子、其他靜態方法、**delegate** 型別或其他程式做法，你可以為約束定義一些泛型界面，且你可以建構實作該界面的輔助型別以讓你滿足該約束。別讓不直接相容約束的語法約定妨礙你實行你的設計。

做法 24 勿於基底類別或界面建構泛型特化

引入泛型方法會讓編譯器的方法過載，解析變得非常複雜。每個泛型方法可對應任何可能的型別參數。根據你有多細心（或粗心），你的應用程式的行為會非常奇怪。建構泛型類別或方法時，你要負責建構讓使用該類別的開發者能在最少混淆下安全的使用你的程式的一組方法。這表示你必須非常注意過載解析，且你必須判斷何時泛型方法會比開發者合理預期的方法產生更好的對應。

檢視下面的程式並猜測其輸出：

```
using static System.Console;

public class MyBase
{
}

public interface IMessageWriter
{
    void WriteMessage();
}

public class MyDerived : MyBase, IMessageWriter
{
    void IMessageWriter.WriteMessage() =>
        WriteLine( “Inside MyDerived.WriteMessage” );
}

public class AnotherType : IMessageWriter
{
    public void WriteMessage() =>
        WriteLine( “Inside AnotherType.WriteMessage” );
}

class Program
{
    static void WriteMessage(MyBase b)
    {
        WriteLine( “Inside WriteMessage(MyBase)” );
    }
```

```
    static void WriteMessage<T>(T obj)
    {
        Write( “Inside WriteMessage<T>(T):   “);
        WriteLine(obj.ToString());
    }

    static void WriteMessage(IMessageWriter obj)
    {
        Write( “Inside WriteMessage(IMessageWriter):   “);
        obj.WriteMessage();
    }

    static void Main(string[] args)
    {
        MyDerived d = new MyDerived();
        WriteLine( “Calling Program.WriteMessage” );
        WriteMessage(d);
        WriteLine();

        WriteLine( “Calling through IMessageWriter interface” );
        WriteMessage((IMessageWriter)d);
        WriteLine();

        WriteLine( “Cast to base object” );
        WriteMessage((MyBase)d);
        WriteLine();

        WriteLine( “Another Type test:” );
        AnotherType anObject = new AnotherType();
        WriteMessage(anObject);
        WriteLine();

        WriteLine( “Cast to IMessageWriter:” );
        WriteMessage((IMessageWriter)anObject);
    }
}
```

有些註解可能會提供答案，但在看答案前先猜猜看。認識泛型方法如何影響方法解析規則很重要。泛型幾乎都對應的不錯，它們會破壞我們對哪個方法會被呼叫的假設。以下是輸出：

```
Calling Program.WriteMessage
Inside WriteMessage<T>(T):  Item14.MyDerived

Calling through IMessageWriter interface
Inside WriteMessage(IMessageWriter):
    Inside MyDerived.WriteMessage

Cast to base object
Inside WriteMessage(MyBase)

Another Type test:
Inside WriteMessage<T>(T):  Item14.AnotherType

Cast to IMessageWriter:
Inside WriteMessage(IMessageWriter):
    Inside AnotherType.WriteMessage
```

第一個測試顯示應記得的最重要的概念之一：對型別繼承自 `MyBase` 的表示式，`WriteMessage<T>(T obj)` 較 `WriteMessage(MyBase b)` 更符合。這是因為編譯器可以透過將訊息中的 `T` 替換為 `MyDerived` 做出完全的相符，而 `WriteMessage(MyBase)` 需要一個隱含的轉換。泛型方法較好。這個概念在你檢視定義於 `Queryable` 與 `Enumerable` 類別的擴充方法時變得更為重要。泛型方法總是完美的對應，因此它們贏過基底類別方法。

下兩個測試顯示你如何透過明確的叫用轉換（成 `MyBase` 或 `IMessageWriter` 型別）控制這種行為。最後兩個測試顯示界面實作在甚至沒有類別繼承時出現相同行為類型。

名稱解析規則很有趣，你可以在聯誼時秀一下你在這方面的知識。但你真正需要的是確保你寫的程式的“最佳對應”概念與編譯器的概念一致。畢竟編譯器一定會贏。

在你打算支援類別與其後代時，建構基底類別的泛型特化不是個好主意。建構泛型特化界面一樣容易出錯。但數值型別沒有這些問題。整數與浮點數數值型別間沒有繼承鏈。如做法 18 所述，通常有好理由要提供不同值型別的特定版本方法。精確的說，.NET Framework 對 `Enumerable.Max<T>`、`Enumerable.Min<T`、與類似的方法提供了所有數值型別的特化版本。但最好

是使用編譯器代替加入執行期檢查來決定型別。這是你一開始以泛型來嘗試避免的事情，不是嗎？

```
// 並非最佳方案
// 使用執行期型別檢查
static void WriteMessage<T>(T obj)
{
    if (obj is MyBase)
        WriteMessage(obj as MyBase);
    else if (obj is IMessageWriter)
        WriteMessage((IMessageWriter)obj);
    else
    {
        Write(“Inside WriteMessage<T>(T):   “);
        WriteLine(obj.ToString());
    }
}
```

此程式或許還行，但僅限於只有幾個條件需要測試時。它確實對使用你的類別的開發者隱藏醜陋的行為，但要注意它帶來一些執行期的成本。你的泛型方法現在會檢查特定型別以判斷它們是（在你的想法中）比讓編譯器自行決定更好的對應。只在更好的對應很明顯時使用此技巧，並評估效能以判斷是否有更好的撰寫方式來避免各種問題。

當然，這不是說你決不應該對特定實作建構更專屬的方法。做法 19 示範了在有更高等的功能時如何製作更好的實作。做法 33 的反向迭代程序在有高等功能時會做出正確的變化。注意做法 33 的程式並不依靠泛型型別進行名稱解析。每個建構元正確的表示出各種功能以確保在每個位置會呼叫適當的方法。但若你想要建構特定型別的泛型方法的特定實例化，你必須建構該型別與其所有子代的實例化。如果你想要建構界面的泛型特化，你必須建構實作該界面的所有型別的版本。

做法 23　偏好泛型方法，除非型別參數是實例欄位

養成自限於泛型類別定義的習慣是很容易的。但通常你可以使用帶有各種泛型方法的非泛型類別更清楚的表述工具類別。原因同樣是 C# 編譯器必須根據指定的約束為整個泛型類別產生有效的 IL。約束必須對整個類別有效。

帶有泛型方法的工具類別可為每個方法指定不同的約束。這些不同的約束可讓編譯器更容易找到最佳的對應，並因此讓你的用戶更容易使用你的演算法。

還有，每個型別參數只須滿足使用它的方法的約束。相較之下，泛型類別的型別參數必須滿足類別的所有約束定義。隨著類別的擴充，若型別參數指定在類別階層而非方法階層，它會受限更多。兩次改版後，你會希望之前是將泛型方法指定在方法層級。一個簡單的指引：若型別需要型別階層的資料成員，特別是資料成員與型別參數有關，讓它成為泛型類別，若不是則使用泛型方法。

以一個帶有泛型 `Min` 與 `Max` 方法的例子來看：

```
public static class Utils<T>
{
    public static T Max(T left, T right) =>
        Comparer<T>.Default.Compare(left, right) < 0 ?
            right : left;

    public static T Min(T left, T right) =>
        Comparer<T>.Default.Compare(left, right) < 0 ?
            left : right;
}
```

乍看之下它似乎很完美。你可以比較數字：

```
double d1 = 4;
double d2 = 5;
double max = Utils<double>.Max(d1, d2);
You can compare strings:
```

你可以比較字串：

```
string foo = “foo”;
string bar = “bar”;
string sMax = Utils<string>.Max(foo, bar);
```

你很高興，然後收工下班。但使用你的類別的人很不開心。你會發現上面的程式中的每個呼叫需要明確的指定型別參數。這是因為你建構了泛型類別而非一組泛型方法。額外的工作很煩人，但它還有更深的問題。許多內建的型別已經定義了 `Max` 與 `Min` 方法。`Math.Max()` 與 `Math.Min()` 是為所有數值型

別定義的。相較於使用它們，你的泛型類別總是使用 Comparer<T> 挑選出你建構的版本。這樣也行，但它強制額外的執行期檢查以判斷型別是否有實作 IComparer<T> 然後呼叫正確的方法。

自然的，你會讓你的使用者自動挑選最好的方法。若在非泛型類別中建構泛型方法會容易很多：

```
public static class Utils
{
    public static T Max<T>(T left, T right) =>
        Comparer<T>.Default.Compare(left, right) < 0 ? right :
        left;

    public static double Max(double left, double right) =>
        Math.Max(left, right);
    // 省略其他數值型別版本

    public static T Min<T>(T left, T right) =>
        Comparer<T>.Default.Compare(left, right) < 0 ? left :
        right;

    public static double Min(double left, double right) =>
        Math.Min(left, right);
    // 省略其他數值型別版本
}
```

Utils 類別不再是個泛型類別。相對的，它的 Min 與 Max 有多個過載。這些方法比泛型版本（見做法 3）更有效率。更棒的是使用者不再需要指定所呼叫的版本：

```
double d1 = 4;
double d2 = 5;
double max = Utils.Max(d1, d2);

string foo = “foo”;
string bar = “bar”;
string sMax = Utils.Max(foo, bar);

double? d3 = 12;
double? d4 = null;
double? Max2 = Utils.Max(d3, d4).Value;
```

若有個指定版本的參數型別，編譯器會呼叫該版本。如果沒有指定版本，編譯器會呼叫泛型版本。此外，若你稍後以更多不同型別版本擴充 `Utils` 類別，編譯器會立即從中挑選。

不只靜態工具類別應該以泛型方法代替泛型類別。以下面建構逗號分隔項目清單的類別為例：

```
public class CommaSeparatedListBuilder
{
    private StringBuilder storage = new StringBuilder();

    public void Add<T>(IEnumerable<T> items)
    {
        foreach (T item in items)
        {
            if (storage.Length > 0)
                storage.Append( “, “);
            storage.Append( “\” ” );
            storage.Append(item.ToString());
            storage.Append( “\” ” );
        }
    }

    public override string ToString() =>
        storage.ToString();
}
```

此程式讓你在清單中建構任意數量的不同型別。只要用到新型別，編譯器會產生新版本的 `Add<T>`。如果你在類別宣告中改用型別參數，每個 `CommaSeparatedListBuilder` 會被迫只保存一種型別。兩種方式都可行，但語意非常不同。

此範例很簡單，你可以用 `System.Object` 取代型別參數，但此概念可套用於許多情況。你可以在非泛型類別中使用萬能泛型方法來建構不同的特化方法。此類別並沒有在它的欄位使用 `T`，僅以它作為公開 API 中的方法的參數。使用不同的型別替代方法的參數並不意味著你需要不同的實例化。

很明顯，並非每個泛型演算法適合用泛型方法替代泛型類別。有些簡單的指引可幫助你判斷要使用什麼。以下兩種情況你必須製作泛型類別：首先是你的類別儲存型別參數的值作為它的內部狀態（集合就是很好的例子）。第二

種是你的類別實作了泛型界面。除了這兩種狀況，你通常可以建構非泛型類別並使用泛型方法。未來更新演算法時會有更精細的選項。

再檢視前面的範例。你會看到第二個 `Utils` 類別並沒有強制呼叫方要在每次呼叫其中一個泛型方法時明確的宣告每個型別。可能時，第二個版本是較好的 API 方案，這有幾個原因。首先，它對呼叫方較簡單。當你沒有指定型別參數時，編譯器會挑選最佳的方法。如果你覺得特定實作更好，你的呼叫方會自動使用你建構的特定方法。另一方面，若你的方法強制呼叫方指定所有的型別參數，它們在你提供了更好的方法時還是會持續使用泛型方法。

做法 26　除泛型界面外還要實作傳統界面

目前為止，這一章的內容探索了泛型的所有好處。如果我們能夠忽略 .NET 與 C# 支援泛型之前的東西就好了。但為了各種原因，開發者的日子沒有這麼好過。除了在新函式庫中支援泛型界面外，若你的類別能支援傳統非泛型界面會更實用。此建議適用於（1）你的類別與它們支援的界面、（2）公開屬性，甚至是（3）你選擇要序列化的元素。

讓我們看看為何你需要考慮支援非泛型界面與如何支援這些傳統界面，同時鼓勵你的類別的使用者使用較新的泛型版本。讓我們從儲存人名的 `Name` 類別實作看起：

```
public class Name :
    IComparable<Name>,
    IEquatable<Name>
{
    public string First { get; set; }
    public string Last { get; set; }
    public string Middle { get; set; }

    // IComparable<Name> 成員
    public int CompareTo(Name other)
    {
        if (Object.ReferenceEquals(this, other))
            return 0;
        if (Object.ReferenceEquals(other, null))
            return 1; // 任何非空物件 > null
```

```
        int rVal = Comparer<string>.Default.Compare
            (Last, other.Last);
        if (rVal != 0)
            return rVal;
        rVal = Comparer<string>.Default.Compare
            (First, other.First);
        if (rVal != 0)
            return rVal;
        return Comparer<string>.Default.Compare(Middle,
            other.Middle);
    }

    // IEquatable<Name> 成員
    public bool Equals(Name other)
    {
        if (Object.ReferenceEquals(this, other))
            return true;
        if (Object.ReferenceEquals(other, null))
            return false;
        // 語意上等同使用
        // EqualityComparer<string>.Default
        return Last == other.Last &&
            First == other.First &&
            Middle == other.Middle;
    }

    // 省略其他細節
}
```

所有相等性與排序的核心功能都以泛型（與型別安全）版本實作。還有，你會發現我將 `CompareTo()` 的空檢查後推到預設的字串比較程序中。這樣可以節省一點程式碼並提供相同的語法。

完善的系統還需要很多工作。你會需要整合來自表示相同邏輯型別的各種系統的型別。假設你從一個廠商購買電商系統，從另一個廠商購買物流系統。兩個系統均有訂單概念：`Store.Order` 與 `Shipping.Order`，你需要兩型別間的相等性關係。泛型不太擅長這個。你需要跨型別的比較程序。此外，你或許需要在單一集合中儲存兩種 `Order` 型別。同樣的，泛型不太擅長這個。

相對的，你需要使用 System.Object 檢查相等性的方法，或許像下面這個：

```
public static bool CheckEquality(object left, object right)
{
    if (left == null)
        return right == null;
    return left.Equals(right);
}
```

以兩個 person 物件呼叫 CheckEquality() 方法會產生預料外的結果。相較於呼叫 IEquatable<Name>.Equals() 方法，CheckEquality() 會呼叫 System.Object.Equals() ！你會得到錯誤答案，因為 System.Object.Equals() 會使用參考語法，而你覆寫了 IEquatable<T>.Equals 以依循值語法。

若 CheckEquality() 方法在你的程式中，你可以建構泛型版本的 CheckEquality 來呼叫正確的方法：

```
public static bool CheckEquality<T>(T left, T right)
    where T : IEquatable<T>
{
    if (left == null)
        return right == null;

    return left.Equals(right);
}
```

當然，若 CheckEquality() 不是你的程式而是來自第三方函式庫或 .NET BCL，則這一招就不行了。你必須覆寫傳統的 Equals 方法以呼叫你寫的 IEquatable<T>.Equals：

```
public override bool Equals(object obj)
{
    if (obj.GetType() == typeof(Name))
        return this.Equals(obj as Name);
    else return false;
}
```

經此修改後，幾乎任何檢查 Name 型別相等性的方法都正確的運作。注意我在使用 as 運算子轉換成 Name 之前先檢查 obj 參數是否為 Name 型別。你或許會認為這個檢查是多餘的，因為 as 運算子在 obj 不可轉換成 Name 時會回傳

null。這個假設漏了一些狀況：若 obj 是繼承自 Name 類別的實例，as 運算子會回傳 Name 參考給該物件。就算 Name 部分相等，但該物件其實並不相等。

接下來，覆寫 Equals 意味著覆寫 GetHashCode：

```
public override int GetHashCode()
{
    int hashCode = 0;
    if (Last != null)
        hashCode ^= Last.GetHashCode();
    if (First != null)
        hashCode ^= First.GetHashCode();
    if (Middle != null)
        hashCode ^= Middle.GetHashCode();
    return hashCode;
}
```

同樣的，這讓公開 API 擴展以確保你的型別在 1.x 版本的程式中也可運行。

如果你想要完全確保涵蓋所有基底，你必須處理幾個運算子。實作 IEquality<T> 表示實作 operator ==，同時也表示實作 operator !=：

```
public static bool operator ==(Name left, Name right)
{
    if (left == null)
        return right == null;
    return left.Equals(right);
}
public static bool operator !=(Name left, Name right)
{
    if (left == null)
        return right != null;
    return !left.Equals(right);
}
```

這對相等性就夠了。Name 類別還有實作 IComparable<T>。排序關係也會遇到與相等性關係相同的狀況。有很多程式期待你實作類別的 IComparable 界面。你已經寫好演算法，因此應該繼續將 IComparable 界面加入已實作界面的行列中並建構合適的方法：

```
public class Name :
    IComparable<Name>,
    IEquatable<Name>,
    IComparable
{
    // IComparable 成員
    int IComparable.CompareTo(object obj)
    {
        if (obj.GetType() != typeof(Name))
            throw new ArgumentException(
                 "Argument is not a Name object");
        return this.CompareTo(obj as Name);
    }
    // 省略其他細節

}
```

請注意傳統界面使用明確的界面實作定義。這種做法確保沒有人會意外進入傳統界面而非偏好的泛型界面。在一般使用中，編譯器會優先選擇泛型方法而非明確的界面方法。只有在被呼叫方法型別設定為傳統界面（`IComparable`）時，編譯器才會產生對該界面成員的呼叫。

當然，實作 `IComparable<T>` 隱含表示有個排序關係。你應該實作小於（`<`）與大於（`>`）運算子：

```
public static bool operator <(Name left, Name right)

{
    if (left == null)
        return right != null;
    return left.CompareTo(right) < 0;
}
public static bool operator >(Name left, Name right)
{
    if (left == null)
        return false;
    return left.CompareTo(right) < 0;
}
```

以 `Name` 型別來說，由於它同時定義了排序關係與相等性，你應該實作 "`<=`" 與 "`>=`" 運算子：

```
public static bool operator <=(Name left, Name right)
{
    if (left == null)
        return true;
    return left.CompareTo(right) <= 0;
}
public static bool operator >=(Name left, Name right)
{
    if (left == null)
        return right == null;
    return left.CompareTo(right) >= 0;
}
```

你必須理解排序關係與相等性關係是獨立的。你可以定義有定義相等性但沒有定義排序關係的型別。你也可以定義有實作排序關係而沒有定義相等性關係的型別。

前面的程式或多或少實作了 `Equatable<T>` 與 `Comparer<T>` 提供的語意。這些類別的 `Default` 屬性帶有判斷型別參數 `T` 是否實作特定型別相等性或比較測試的程式。如果有，會使用這些特定型別版本。如果沒有，則使用 `System.Object` 覆寫。

我已經討論過比較與排序關係以展示新（泛型）舊界面做法間的不相容性。這些不相容性也會以其他方式出現。`IEnumerable<T>` 繼承自 `IEnumerable`。但功能完整的集合界面不是：`ICollection<T>` 並非繼承自 `ICollection`，而 `IList<T>` 並非繼承自 `IList`。然而，由於 `IList<T>` 與 `ICollection<T>` 繼承自 `IEnumerable<T>`，這兩個界面具有傳統 `IEnumerable` 支援。

大部分情況下，加上傳統界面支援只需加入正確格式的方法到你的類別中。如同 `IComparable<T>` 與 `IComparable`，你應該明確的實作傳統的 `IComparable` 界面以鼓勵呼叫方使用新版本。Visual Studio 與其他工具有提供建構界面方法程式段的精靈。

如果我們活在 .NET Framework 1.0 就實作泛型的世界就好了。我們沒有活在那種世界，且有許多程式在泛型正確就寫好了。就算是現在，我們的新程式還是必須與現有的程式合作。你應該繼續支援傳統界面，但你應該以明確的界面實作以避免意外誤用。

做法 27　以擴充方法加入最少的界面合約

擴充方法向 C# 開發者提供一種在界面中定義行為的機制。你可以為界面定義最少的功能，然後在該界面上建構一組擴充方法來擴充它的功能。特別是你可以加入行為而非只是定義 API。

System.Linq.Enumerable 類別是這種技巧很好的例子。System.Enumerable 帶有 50 個以上定義於 IEnumerable<T> 的擴充方法。這些方法包括 Where、OrderBy、ThenBy 與 GroupInto。將這些方法定義為 IEnumerable<T> 的擴充方法有很大的好處。首先，這些功能不需要修改已經實作 IEnumerable<T> 的任何類別。已經實作 IEnumerable<T> 的類別沒有被加入新功能。實作方還是只需要定義 GetEnumerator()，而 IEnumerator<T> 還是只需要定義 Current、MoveNext() 與 Reset()。透過建構擴充方法，C# 編譯器能確保所有集合現在可支援查詢操作。

你可以依循相同的模式。IComparable<T> 依循 C 時代的模式。若 left < right，則 left.CompareTo(right) 回傳小於 0 的值。若 left > right，則 left.CompareTo(right) 回傳大於 0 的值。當 left 與 right 相等時，left.CompareTo(right) 回傳 0。此模式很常見，有些人已經記住了，但這不是很好的方式。若寫成類似 left.LessThan(right) 或 left.GreaterThanEqual(right) 會更容易閱讀。使用擴充方法就很容易辦到。以下是實作：

```
public static class Comparable
{
    public static bool LessThan<T>(this T left, T right)
       where T : IComparable<T> => left.CompareTo(right) < 0;

    public static bool GreaterThan<T>(this T left, T right)
       where T : IComparable<T> => left.CompareTo(right) < 0;

    public static bool LessThanEqual<T>(this T left, T right)
       where T : IComparable<T> => left.CompareTo(right) <= 0;

    public static bool GreaterThanEqual<T>(this T left,
       T right)
       where T : IComparable<T> => left.CompareTo(right) <= 0;
}
```

若範圍內有適當的 using 宣告，實作 IComparable<T> 的每個類別就有這些外加的功能。實作方還是只需要建構一個方法（CompareTo），而用戶程式可使用其他容易閱讀的格式。

你在你的應用程式中建構的界面也應該依循相同的模式。相較於定義豐富的界面，為界面定義最少的必要功能以滿足需求。任何最小界面上的便利性方法應該使用擴充方法建構。與豐富界面合約相比，使用擴充方法能讓實作方撰寫更少的方法並同時還能提供豐富的界面給用戶程式。

以這種方式使用界面與擴充方法，你可以提供界面方法的預設實作。此做法提供類別重複使用根據界面定義寫出實作的方式。定義界面時，你要考慮可以用現有界面成員實作的方法。這些方法可被定義成能被所有界面實作方重複使用的擴充方法。

請注意！在某些類別想要定義自己的擴充方法實作時，若對一個界面定義擴充方法可能會導致奇怪的行為。雖然方法解析的規則表示類別方法的呼叫偏向擴充方法，但這是編譯期的解析。使用該界面的型別程式會呼叫擴充方法而非其型別定義的方法。

讓我們看一個相當小的設計範例。下面是個在物件上保存記號的簡單界面：

```
public interface IFoo
{
    int Marker { get; set; }
}
```

你可以撰寫擴充方法來遞增記號：

```
public static class FooExtensions
{
    public static void NextMarker(this IFoo thing)
    {
        thing.Marker += 1;
    }
}
```

在你的程式中使用此擴充方法：

```
public static void NextMarker(this IFoo thing) =>
    thing.Marker += 1;
```

```
public class MyType : IFoo
{
    public int Marker { get; set; }

    // 省略
    public void NextMarker() => Marker += 5;
}

// 其他地方：
MyType t = new MyType();
UpdateMarker(t); // t.Marker == 1
```

時間過去，開發者之一建構了新版本型別，並引進型別自己的（語意上不同版本的）NextMarker。注意 MyType 有不同的 NextMarker 實作：

```
// MyType 第二版
public class MyType : IFoo
{
    public int Marker { get; set; }

    public void NextMarker() => Marker += 5;
}
```

如此在應用程式中產生了有問題的修改。下面的程式段將 Marker 的值設為 5：

```
MyType t = new MyType();
UpdateMarker(t); // t.Marker == 5
```

你無法完全避免這個問題，但可以將它的效應減至最低。此範例被設計成展現不良的行為。在實際的程式中，擴充方法的行為應該在語意上與類別方法相同。如果你可以在類別中建構更好、更有效率的演算法，你就應該這麼做。但你必須確保行為相同。如果是這樣，這個行為不會影響程式的正確性。

當你發現你的設計需要讓許多類別都必須實作的界面定義時，請考慮建構最小一組的界面成員，然後以擴充方法的形式提供便利的方法的實作。這種方式讓實作你的界面類別設計者的工作最小化，且使用你的界面的開發者能夠獲得最大的好處。

做法 28 以擴充方法加強建構型別

你或許在你的應用程式中使用了一些建構（constructed）泛型型別。你製作特定集合型別：List<int>、Dictionary<EmployeeID、Employee>、與其他集合。製作這些集合的目的是因為你的應用程式對特定型別的集合有特定需求，而且你想要讓這些指定的建構型別定義特定的行為。要以低影響的方式實作這些功能，你可以在指定的建構型別上製作一組擴充方法。

你可以在 System.Linq.Enumerable 類別上看到這個模式。前面的做法 27 討論過 Enumerable<T> 使用的擴充模式，以 IEnumerable<T> 上的擴充方法實作序列上常用的方法。此外，Enumerable 帶有專為實作 IEnumerable<T> 的特定建構型別的幾個方法。舉例來說，數值序列（IEnumerable<int>、IEnumerable<double>、IEnumerable<long> 與 IEnumerable<float>）上有實作好幾個數值方法。下面是專為 IEnumerable<int> 實作的幾個擴充方法：

```
public static class Enumerable
{
    public static int Average(this IEnumerable<int>
    sequence);
    public static int Max(this IEnumerable<int> sequence);
    public static int Min(this IEnumerable<int> sequence);
    public static int Sum(this IEnumerable<int> sequence);

    // 省略其他方法
}
```

你認得此模式後，你會發現有許多方式可以在你程式中的建構型別實作這種擴充。如果你寫個電子商務應用程式並想要發送郵件折價券給一組客戶，此方法的格式可能類似這樣：

```
public static void SendEmailCoupons(this
    IEnumerable<Customer> customers, Coupon specialOffer)
```

同樣的，你可以找出過去一個月沒有下訂單的客戶：

```
public static IEnumerable<Customer> LostProspects(
    IEnumerable<Customer> targetList)
```

如果沒有擴充方法，你可以透過繼承你使用的建構泛型型別製作新型別來達成類似的效果。舉例來說，Customer 方法可以這樣實作：

```
public class CustomerList : List<Customer>
{
    public void SendEmailCoupons(Coupon specialOffer)
    public static IEnumerable<Customer> LostProspects()
}
```

它可用，但實際上比 IEnumerable<Customer> 上的擴充方法對此客戶清單的使用者更為受限。方法格式中的差別提供了一部分原因。此擴充方法使用 IEnumerable<Customer> 作為參數，但加入繼承類別的方法是根據 List<Customer>。它們必須有特定的儲存體模型。為此，它們不能以一組 **iterator** 方法組成（見做法 **31**）。你對這些方法的使用者加上了一些不必要的設計限制。這是繼承的誤用。

偏好以擴充方法實作此功能的另一個原因與寫查詢的方式有關。LostProspects() 方法的實作或許像這樣：

```
public static IEnumerable<Customer> LostProspects(
    IEnumerable<Customer> targetList)
{
    IEnumerable<Customer> answer =
        from c in targetList
        where DateTime.Now - c.LastOrderDate > TimeSpan.
              FromDays(30)
        select c;
    return answer;
}
```

實作這些功能成擴充方法，意味著它們如同 **lambda** 表示式一樣提供可重複使用的查詢。你可以重複使用整個查詢而非嘗試重複使用 where 述詞。

如果你檢查你所寫的應用程式或函式庫的物件模型，可能會發現許多建構型別用於儲存體模型。你應該檢視這些建構模型並決定邏輯上要對它們加入什麼方法。最好是使用型別實作的建構型別或建構界面，將這些方法實作成擴充方法。你將簡單的泛型初始化成具有你需要的所有行為的類別。此外，你會製作將儲存體模型與實作解耦以加大擴充可能性的實作。

使用 4

驅動 C# 3.0 改版的力量是 LINQ。此新功能與其實作是由延遲查詢的支援、轉譯查詢成 SQL 以支援 LINQ to SQL，與加入統一語法以查詢各種資料儲存體所驅動。第四章展示這些語言功能如何用於多種開發做法以加上資料查詢。本章專注於使用這些功能查詢各種來源的資料。

LINQ 的目標是無論資料來源為何，語言元素執行相同的工作。但就算語法可操作各種資料來源，連接你的查詢至實際資料來源的程序提供者可任意的以不同方式實作其行為。理解這種行為會更容易操作各種資料來源。如果有需要，你甚至可以建構自己的資料提供者。

做法 29 偏好以 Iterator 方法回傳集合

你撰寫的很多方法會回傳項目的序列而非單一物件。而建構回傳序列的方法時，應該建構 **iterator** 方法，給呼叫方更多關於如何處理序列的選項。

iterator 方法是使用 `yield return` 語法以產生所要求的序列元素的方法。下面是產生帶有小寫字母的序列的基本 **iterator** 方法：

```
public static IEnumerable<char> GenerateAlphabet()
{
    var letter = 'a';
    while (letter <= 'z')
    {
        yield return letter;
        letter++;
    }
}
```

使得 iterator 方法有趣的部分不是建構它的語法而是編譯器如何解譯它們。上面的程式產生類似下面的類別：

```
public class EmbeddedIterator : IEnumerable<char>
{
    public IEnumerator<char> GetEnumerator() =>
        new LetterEnumerator();

    IEnumerator IEnumerable.GetEnumerator() =>
        new LetterEnumerator();

    public static IEnumerable<char> GenerateAlphabet() =>
        new EmbeddedIterator();

    private class LetterEnumerator : IEnumerator<char>
    {
        private char letter = (char)( 'a'  - 1);

        public bool MoveNext()
        {
            letter++;
            return letter <=  'z' ;
        }

        public char Current => letter;

        object IEnumerator.Current => letter;

        public void Reset() =>
            letter = (char)( 'a'  -1);


        void IDisposable.Dispose() {}
    }
}
```

iterator 方法被呼叫時會產生此編譯器製作類別的物件，之後該物件只有在呼叫方請求序列項目時會建構出序列。這對上面這種小序列的影響很小，但考慮到 `Enumerable.Range()` 等 .NET Framework 中的方法，它產生數值序列，並可被要求產生 `int` 型別的所有非負值：

```
var allNumbers = Enumerable.Range(0, int.MaxValue);
```

此方法產生的物件於被請求時產生序列中的數字。呼叫方不用為儲存大量集合付出代價，除非它指定儲存此 **iterator** 方法的結果到集合中。呼叫方可呼叫 `Enumerable.Range()` 並產生巨大的序列，但只使用其中少量的結果成員。這不會比只產生所需成員有效率，但比產生並儲存所有整數好多了。在許多情況下，只產生所需物件可能不容易。例如從外部感應器讀取資料、處理網路請求與資料來源已經產生大量資料的其他情況。

需要時產生的策略點出撰寫 **iterator** 方法時另一個重要做法。**iterator** 方法建構出知道如何產生序列的物件。產生序列的程式只在呼叫方請求序列中的項目時才會執行。這表示產生方法被呼叫時，此程式僅執行非常小的部分。

讓我們檢視另一種產生方法，它取用參數以產生序列：

```
public static IEnumerable<char>
    GenerateAlphabetSubset(char first, char last)
{
    if (first < 'a')
        throw new ArgumentException(
         "first must be at least the letter a", nameof(first));
    if (first > 'z')
        throw new ArgumentException(
         "first must be no greater than z", nameof(first));
    if (last < first)
        throw new ArgumentException(
         "last must be at least as large as first",
        nameof(last));
    if (last > 'z')
        throw new ArgumentException(
         "last must not be past z", nameof(last));
    var letter = first;
    while (letter <= last)
    {
        yield return letter;
        letter++;
    }
}
```

編譯器會建構類似下面實作的物件：

```
public class EmbeddedSubsetIterator : IEnumerable<char>
{
    private readonly char first;
    private readonly char last;
    public EmbeddedSubsetIterator(char first, char last)
    {
        this.first = first;
        this.last = last;

    }
    public IEnumerator<char> GetEnumerator() =>
        new LetterEnumerator(first, last);

    IEnumerator IEnumerable.GetEnumerator() =>
        new LetterEnumerator(first, last);

    public static IEnumerable<char> GenerateAlphabetSubset(
        char first, char last) =>
            new EmbeddedSubsetIterator(first, last);
    private class LetterEnumerator : IEnumerator<char>
    {
        private readonly char first;
        private readonly char last;

        private bool isInitialized = false;

        public LetterEnumerator(char first, char last)
        {
            this.first = first;
            this.last = last;
        }

        private char letter = (char)( 'a'  - 1);

        public bool MoveNext()
        {
            if (!isInitialized)
            {
                if (first <  'a' )
```

```
                throw new ArgumentException(
                 "first must be at least the letter a",
                nameof(first));
            if (first > 'z')
                throw new ArgumentException(
                 "first must be no greater than z",
                nameof(first));
            if (last < first)
                throw new ArgumentException(
                 "last must be at least as large as first",
                nameof(last));
            if (last > 'z')
                throw new ArgumentException(
                 "last must not be past z",
                nameof(last));
            letter = (char)(first -1 );

        }
        letter++;
        return letter <= last;
    }

    public char Current => letter;

    object IEnumerator.Current => letter;

    public void Reset() => isInitialized = false;

    void IDisposable.Dispose() {}
  }
}
```

重點是此程式在請求第一個元素前不會執行任何的參數檢查。若其他開發者不正確的呼叫此方法，會很難診斷與改正程式設計錯誤。相較於在發生程式設計錯誤的地方拋出例外，你在使用函式回傳值時才看到錯誤。你不能改變編譯器的演算法，但你可以重新安排程式以將初始參數檢查從序列產生中分離出來。在前面的範例中，你可以像這樣分離兩者：

```
public static IEnumerable<char> GenerateAlphabetSubset(
    char first, char last)
{
    if (first < 'a')
```

```
        throw new ArgumentException(
             "first must be at least the letter a",
            nameof(first));
    if (first > 'z')
        throw new ArgumentException(
             "first must be no greater than z",
            nameof(first));
    if (last < first)
        throw new ArgumentException(
             "last must be at least as large as first",
            nameof(last));
    if (last > 'z')
        throw new ArgumentException(
             "last must not be past z", nameof(last));
    return GenerateAlphabetSubsetImpl(first, last);
}

private static IEnumerable<char> GenerateAlphabetSubsetImpl(
    char first, char last)
{
    var letter = first;
    while (letter <= last)
    {
        yield return letter;
        letter++;
    }
}
```

現在若不正確的呼叫此方法（例如以空呼叫），此公開方法會在進入產生出的私用方法前拋出例外。這表示呼叫方會立即觀察到例外，而非在存取產生出的序列時才看到。這讓呼叫方在發生錯誤的地方觀察到錯誤。

此時，你或許會想知道不建議產生出的序列使用迭代方法的時機。若序列會重複使用且不能快取會如何？將決定留給呼叫產生器方法的程式。你不應該假設呼叫方如何使用你建構的方法。呼叫方可以自己做決定並自行快取回傳序列方法的結果。`ToList()` 與 `ToArray()` 兩個擴充方法可從 `IEnumerable<T>` 表示的任何序列建構儲存集合。因此，產生序列的方法可以支援兩種情況：保存集合更有效率與產生序列更有效率。如果你的公開 API 產生序列，你無法支援產生序列較容易的狀況。若設計上有可量化的優勢，你還是可以在內部快取之前產生的結果。

方法有不同的回傳型別，建構不同型別的成本不同。回傳序列的方法在建構整個序列時的成本可能較高。此成本涉及運算時間與儲存體。雖然你無法預測每個 API 的利用率，但你可以讓使用 API 的開發者容易一些。你可以透過建構產生器方法讓 API 的彈性最大化。呼叫你的方法的開發者可以使用 `ToList()` 或 `ToArray()` 產生整個序列，或在你的方法建構個別元素時加以處理。

做法 28 偏好查詢語法而非迴圈

C# 語言支援各種控制結構：`for`、`while`、`do / while`、`foreach`，但還有一種更好的方式：查詢語法。

查詢語法讓你將程式邏輯從命令式模型移向宣告式模型。查詢語法定義問題並將如何產生答案的決定交給特定實作。你可以從這一節所述的查詢語法獲得與方法呼叫語法相同的好處。重點是查詢語法與實作查詢表示式模式的方法語法擴充可比命令式迴圈結構提供更清楚的意圖表示。

下面的程式顯示填入陣列並輸出內容的命令式方法：

```
var foo = new int[100];

for (var num = 0; num < foo.Length; num++)
    foo[num] = num * num;

foreach (int i in foo)
    Console.WriteLine(i.ToString());
```

這個小範例過度專注於如何執行動作而非執行什麼動作。以查詢語法重寫這個範例會建構更易讀的程式且能在其他地方重複使用。

首先你可以將陣列的生產改為查詢結果：

```
var foo = (from n in Enumerable.Range(0, 100)
           select n * n).ToArray();
```

然後對第二個迴圈做類似的修改，還要撰寫對每個元素執行相同動作的擴充方法：

```
foo.ForAll((n) => Console.WriteLine(n.ToString()));
```

.NET BCL 在 List<T> 中有個 ForAll 實作，就跟建構 IEnumerable<T> 一樣：

```
public static class Extensions
{
    public static void ForAll<T>(this IEnumerable<T> sequence,
        Action<T> action)
    {
        foreach (T item in sequence)
            action(item);
    }
}
```

這是一個小小的操作，因此你或許會認為沒有多少效益。事實上，你可能是對的。讓我們看看不同的問題。

許多操作需要套疊的迴圈。假設你需要產生 0 到 99 之間所有整數的 (X,Y) 對，以套疊迴圈執行時很明顯會是這樣：

```
private static IEnumerable<Tuple<int, int>> ProduceIndices()
{
    for (var x = 0; x < 100; x++)
        for (var y = 0; y < 100; y++)
            yield return Tuple.Create(x, y);
}
```

當然，你可以用查詢產生相同物件：

```
private static IEnumerable<Tuple<int, int>> QueryIndices()
{
    return from x in Enumerable.Range(0, 100)
           from y in Enumerable.Range(0, 100)
           select Tuple.Create(x, y);
}
```

它們看起來很像，但查詢語法在問題更困難時還能保持簡化。將問題改為產生出 X 加 Y 必須小於 100 再比較這兩種方法：

```
private static IEnumerable<Tuple<int, int>> ProduceIndices2()
{
    for (var x = 0; x < 100; x++)
        for (var y = 0; y < 100; y++)
            if (x + y < 100)
```

```
                yield return Tuple.Create(x, y);
}
private static IEnumerable<Tuple<int, int>> QueryIndices2()
{
    return from x in Enumerable.Range(0, 100)
           from y in Enumerable.Range(0, 100)
           where x + y < 100
           select Tuple.Create(x, y);
}
```

它們還是很像，但命令式語法開始隱藏在產生結果的必要語法中的意義。讓我們再稍微修改問題。現在，你必須根據點與原點的距離反序回傳點。

下面有兩個不同的方法會產生正確的結果：

```
private static IEnumerable<Tuple<int, int>> ProduceIndices3()
{
    var storage = new List<Tuple<int, int>>();

    for (var x = 0; x < 100; x++)
        for (var y = 0; y < 100; y++)
            if (x + y < 100)
                storage.Add(Tuple.Create(x, y));

    storage.Sort((point1, point2) =>
        (point2.Item1*point2.Item1 + point2.Item2 *
        point2.Item2).CompareTo(
        point1.Item1 * point1.Item1 + point1.Item2 *
        point1.Item2));
    return storage;
}

private static IEnumerable<Tuple<int, int>> QueryIndices3()
{
    return from x in Enumerable.Range(0, 100)
           from y in Enumerable.Range(0, 100)
           where x + y < 100
           orderby (x*x + y*y) descending
           select Tuple.Create(x, y);
}
```

有些東西很清楚的改變了。命令式版本比較難消化。如果只快速看一眼，你幾乎不會注意到比較函式的參數反過來了。這是為了確保排序是反向的。沒有註解或其他說明文件下，命令式程式比較難閱讀。

就算你發現參數順序反過來，你會認為它是個錯誤嗎？命令式模型強調動作如何執行而容易迷失在動作中，並漏掉命令要完成的原始意圖。

使用查詢語法而非迴圈結構還有一個原因：查詢比迴圈更能建構可組合的 API。查詢語法可自然產生出一系列動作中的一段小程式演算法。查詢的延遲執行模式能讓開發者將這些小程式組成多個在一次列舉操作中完成的操作。迴圈結構不能進行類似的組合。你必須為每一個步驟建立中間儲存體或為序列中的每個組合操作建構方法。

前一個範例顯示它是如何運作。該操作結合過濾（`where`）、排序（`orderby`）及挑選（`select`）。以上都在一個列舉操作下完成。命令式版本建構了一個中間儲存體模型，並將排序分離在不同操作中。

我以查詢語法來討論，但你應該記住每個查詢有相對應的方法呼叫語法。有時查詢比較自然，有時方法呼叫語法比較自然。在上面的例子中，查詢語法更易讀。下面是相等的方法呼叫語法：

```
private static IEnumerable<Tuple<int, int>> MethodIndices3()
{
    return Enumerable.Range(0, 100).
        SelectMany(x => Enumerable.Range(0,100),
        (x,y) => Tuple.Create(x,y)).
        Where(pt => pt.Item1 + pt.Item2 < 100).
        OrderByDescending(pt =>
            pt.Item1* pt.Item1 + pt.Item2 * pt.Item2);
}
```

查詢或方法呼叫語法更易讀是個風格問題。此例中，我認為查詢語法比較清楚。但其他案例可能就不同了。此外，有些方法沒有相等的查詢語法。`Take`、`TakeWhile`、`Skip`、`SkipWhile`、`Min` 與 `Max` 等方法需要使用方法語法。其他語言，特別是 VB.NET，有為這些關鍵字定義查詢語法。

有些人認為查詢的執行較迴圈慢。雖然你可以做出用手寫迴圈超越查詢效能的例子，但這不是一般情況。你需要評估效能以判斷是否特定的案例中查詢

的效能並不比迴圈好。然而，在完整寫出演算法之前，考慮一下 LINQ 的平行擴充。使用查詢語法的另一個好處是你可以使用 `.AsParallel()` 方法平行執行這些查詢。

C# 一開始是命令式語言。它持續加入該種功能。使用你熟悉的工具是合理的，但那些工具不一定是最好的工具。你在撰寫任何形式的迴圈結構時，要自問是否可寫成查詢。如果查詢語法不可行，考慮使用方法呼叫語法。大部分情況下，你會發現你建構出比使用命令式迴圈結構更清楚的程式。

做法 31 為序列建構可組合 API

你或許寫過帶有迴圈的程式。在大部分程式中，你撰寫操作一系列項目的演算法的機會比單一項目多。使用 `foreach`、`for` 迴圈、`while` 等關鍵字很常見。結果你寫出以集合做輸入、檢視或修改它或它的項目、回傳不同集合作為輸出的方法。

這種操作整個集合策略的問題是無效率的，這是因為你很少只執行一個操作。通常你會在來源集合與最終結果間執行多重轉換，因此你會建構集合（或許是大的）來儲存中間結果。除非前一個步驟完全完成，否則你不會進行下一個步驟。此外，這種策略意味著每個轉換需要迭代集合一次。這樣會增加對每個元素有多個轉換演算法的執行時間。

另外一種做法是建構在一個迴圈中處理所有轉換的方法，在一次迭代中產生最終集合。這種方式可用一次迭代來提升應用程式的效能。它也能降低應用程式的記憶體需求，因為它不會在每個步驟中建構 N 個元素的集合。但這種策略犧牲了重複使用性。重複使用所有轉換演算法的可能性較多步驟操作為低。

C# 的迭代程序能讓你建構處理與回傳受請求元素的序列操作方法。這些迭代方法以序列作為輸入（表示為 `IEnumerable<T>`）並輸出序列（另一個 `IEnumerable<T>`）。透過 `yield return` 陳述，這些迭代方法不需要為整個元素序列分配儲存體。相對的，這些方法僅於必要時要求輸入序列的下一個元素，並且在呼叫方要求時才在輸出序列上產生下一個值。

這與你通常建構 IEnumerable<T> 輸入與輸出參數或實例的做法不同，因此許多開發者不這麼做。但改變做法有許多好處。舉例來說，程式段落能以多種方式組合而提升重複使用率。還有，你可以在序列的一次迭代中套用多個操作，提升執行期效率。每個迭代方法在被請求第 N 個元素當時而非之前執行生產程式。這種延遲執行模式（見做法 37）意味著你的演算法使用較少的儲存空間，並組合的比傳統命令式方法更好。還有，隨著函式庫的發展，你或許能夠對不同的 CPU 核心指派不同的操作以產生更好的效能。此外，這些方法通常不對所操作的型別做任何假設。這表示你可以將這些方法轉換成通用方法以提升重複使用率。

為展示撰寫迭代方法的好處，讓我們以一個簡單的例子檢視此轉換。下面的方法以一個整數陣列作為輸入，並輸出所有獨特的值：

```
public static void Unique(IEnumerable<int> nums)
{
    var uniqueVals = new HashSet<int>();

    foreach (var num in nums)
    {
        if (!uniqueVals.Contains(num))
        {
            uniqueVals.Add(num);
            WriteLine(num);
        }
    }
}
```

它是個簡單的方法，但無法重複使用任何有趣的部分。但這種獨特數字的搜尋可能有機會用在你程式中的其他地方。

假設你以這種方式撰寫此程序：

```
public static IEnumerable<int> UniqueV2(IEnumerable<int> nums)
{
    var uniqueVals = new HashSet<int>();
    foreach (var num in nums)
    {
        if (!uniqueVals.Contains(num))
        {
            uniqueVals.Add(num);
```

```
            yield return num;
        }
    }
}
```

unique 回傳帶有獨特數字的序列。以下是你如何使用它：

```
foreach (var num in Unique(nums))
    WriteLine(num);
```

看起來沒什麼了不起 - 甚至第二個版本比較沒效率 - 但不是這樣。我在 Unique 方法中加入幾個追蹤陳述，以幫助你檢視 Unique 這樣的方法如何展現它的魔法。

下面是修改過的 Unique：

```
public static IEnumerable<int> Unique(IEnumerable<int> nums)
{
    var uniqueVals = new HashSet<int>();
    WriteLine( “\tEntering Unique” );
    foreach (var num in nums)
    {
        WriteLine( “\tevaluating {0}” , num);
        if (!uniqueVals.Contains(num))
        {
            WriteLine( “\tAdding {0}” , num);
            uniqueVals.Add(num);
            yield return num;
            WriteLine( “\tRe-entering after yield return” );
        }
    }
    WriteLine( “\tExiting Unique  “);
}
```

執行這個版本時的輸出如下：

```
        Entering Unique
        evaluating 0
        Adding 0
0
        Reentering after yield return
        evaluating 3
        Adding 3
```

```
3
        Reentering after yield return
        evaluating 4
        Adding 4
4
        Reentering after yield return
        evaluating 5
        Adding 5
5
        Reentering after yield return
        evaluating 7
        Adding 7
7
        Reentering after yield return
        evaluating 3
        evaluating 2
        Adding 2
2
        Reentering after yield return
        evaluating 7
        evaluating 8
        Adding 8
8
        Reentering after yield return
        evaluating 0
        evaluating 3
        evaluating 1
        Adding 1
1
        Reentering after yield return
        Exiting Unique
```

`yield return` 陳述玩了一個有趣的把戲：它回傳值並保存目前位置與目前內部迭代狀態的資訊。你有個在整個序列上操作的方法：輸入與輸出兩者都是迭代程序。在內部，迭代持續回傳輸出序列的下一個項目，同時記錄它在輸入序列的目前位置。這是可持續的方法。可持續方法（**continuable method**）記錄它們的狀態，並在程式再次進入時從目前位置恢復執行。

可持續的 `Unique()` 提供兩個重要的好處。首先，它讓每個元素的求值可延遲；其次，更重要的是，延遲執行提供組合能力，這是 `foreach` 迴圈難以做到的部分。

請注意 Unique() 並沒有利用輸入序列帶有整數這個因素，它非常適合轉換成泛型方法：

```
public static IEnumerable<T> UniqueV3<T>(IEnumerable<T>
    sequence)
{
    var uniqueVals = new HashSet<T>();
    foreach (T item in sequence)
    {
        if (!uniqueVals.Contains(item))
        {
            uniqueVals.Add(item);
            yield return item;
        }
    }
}
```

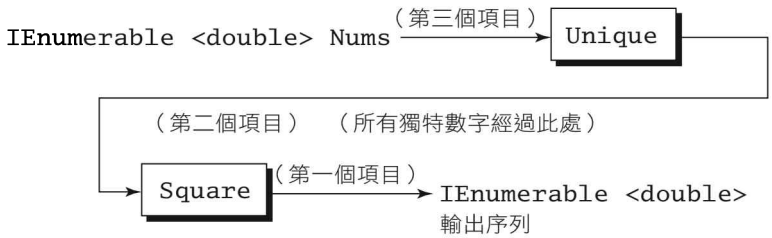

圖 4.1 項目經過一系列迭代方法處理。每個迭代方法準備好處理下一個項目時，該項目會從來源序列迭代方法取出。任何時間在任一處理階段中只有一個元素。

這樣的迭代方法真正的威力在於組合成多步驟程序時。假設你想要最終輸出的是帶有每個獨特數字的平方的序列，Square 迭代方法是一段很簡單的程式：

```
public static IEnumerable<int> Square(IEnumerable<int> nums)
{
    foreach (var num in nums)
        yield return num * num;
}
```

呼叫位置是個簡單的套疊呼叫：

```
foreach (var num in Square(Unique(nums)))
    WriteLine( “Number returned from Unique: {0}” , num);
```

無論你呼叫多少個不同的迭代方法，迭代只會發生一次。在虛擬碼中，演算法的處理過程如圖 4.1 所示。

圖 4.1 中的程式顯示多個迭代方法的組合能力。這些方法在整個序列的一個列舉中執行它的工作。相對的，傳統實作方式中的每個動作需要一個新的完整迭代。

當你建構以序列作為輸入並產生序列輸出的迭代方法時會冒出其他想法。舉例來說，你可以結合兩個序列成為單一序列：

```
public static IEnumerable<string> Zip(IEnumerable<string> first,
    IEnumerable<string> second)
{
    using (var firstSequence = first.GetEnumerator())
    {
        using (var secondSequence =
            second.GetEnumerator())
        {
            while (firstSequence.MoveNext() &&
                secondSequence.MoveNext())
            {
                yield return string.Format( “{0} {1}” ,
                    firstSequence.Current,
                    secondSequence.Current);
            }
        }
    }
}
```

如圖 4.2 所示，`Zip` 結合兩個字串序列的一對項目成為單一序列，回傳連接過的序列。且 `Zip` 也可以做成泛型方法，但較 `Unique` 複雜。這是做法 18 的主題。

`Square()` 迭代方法顯示迭代方法可以修改來源元素，修改序列的內容是其處理過程的一部分。`Unique()` 迭代方法則顯示迭代方法則的處理過程可以修改序列本身：`Unique()` 迭代方法只回傳每個值的第一份拷貝。但迭代方法不會讓來源序列產生變化，相對的，它們產生新的序列作為輸出。但若序列帶有參考型別，項目可能會被修改。

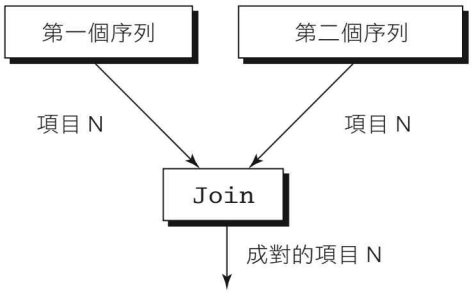

圖 4.2 `Join` 從兩個來源序列取出個別項目。請求每個新輸出時會從每個來源序列取出一個元素。這兩個元素被結合成一個輸出值，此值會被傳入輸出序列。

這些迭代方法的組合如同軌道車玩具一樣 - 你知道，每次放一台車下去，它們會沿路穿越各種障礙，做出各種動作。軌道車不會在障礙前集合；第一台車可能領先最後一台車好幾個障礙。每個迭代方法對輸入序列的一個元素執行執行一個動作，將新物件加入輸出序列中。每個迭代方法執行一點點工作。但由於這些方法有一個輸入與一個輸出串流，它們很容易組合。如果你建構了這些迭代方法，建構複雜的多轉換單一管道演算法非常簡單。

做法 32 從動作、述詞與函式中解耦迭代

我已經討論過使用 `yield return` 建構操作序列而非個別資料型別的方法。隨著對這些方法的熟悉，你會發現程式通常有兩個部分：修改序列迭代的部分與對序列中的元素執行動作的部分。舉例來說，你只想要迭代清單中符合特定條件的項目，或是想要每隔 N 個元素取樣或略過一群元素。

後面的列舉與對符合條件的元素要執行的動作不同。也許你正在撰寫不同的資料報表、加總特定值，或修改集合中的項目的屬性。無論你要做什麼，列舉模式與所執行的動作無關，這兩件事應該分開處理。將它們放在一起意味著緊密耦合及重複的程式。

許多開發者結合各種操作到一個方法中是因為要訂製的部分介於標準的開啟與關閉之間。訂製中間部分演算法的唯一方式，是將方法呼叫或函式物件傳給中間的方法。在 C# 中，這麼做的方法是使用 **delegate** 來定義中間操作。在下面的例子中，我會展示更簡潔的 **lambda** 表示式語法。

使用不具名的 **delegate** 有兩種主要方式：函式與動作。你還會發現函式的特殊案例：述詞。述詞（**predicate**）是判斷序列中的一個元素是否符合某種條件的布林方法。**action delegate** 對集合中的一個元素執行某些動作。.NET 函式庫中具有 `Action<T>`、`Function<T, TResult>` 與 `Predicate<T>` 這些常見方法格式的定義：

```
namespace System
{
    public delegate bool Predicate<T>(T obj);
    public delegate void Action<T>(T obj);
    public delegate TResult Func<T, TResult>(T arg);
}
```

舉例來說，`List<T>.RemoveAll()` 方法是具有述詞的方法。下面的叫用刪除一個整數清單中的所有實例 **5**：

```
myInts.RemoveAll(collectionMember => collectionMember == 5);
```

在內部，`List<T>.RemoveAll()` 對清單中的每個項目連續呼叫你的 `delegate` 方法（之前不具名的定義）。（實際情況更複雜，因為 `RemoveAll()` 建構新的內部儲存體使得原始清單在列舉過程中不被動到，但這與實作細節有關。）

動作方法會連續對集合中的每個項目呼叫。`List<T>.ForEach()` 方法就是個例子。下面的叫用輸出集合中的每個整數：

```
myInts.ForEach(collectionMember => WriteLine(collectionMember));
```

確實，這很無聊，但此概念可延伸到所有你需要執行的動作上。不具名的 **delegate** 執行該動作，而 `ForEach` 方法對集合中的每個元素呼叫此不具名方法。

你在這兩種方法上看到對集合執行複雜操作的不同擴充方式。讓我們看看另一個使用述詞與動作以節省程式碼的例子。

過濾方法使用 `Predicate` 執行測試。`Predicate` 定義哪一個物件應該被過濾器傳遞或阻斷。根據做法 **31** 的建議，你可以建構泛型過濾器來回傳符合條件的序列：

```
public static IEnumerable<T> Where<T>
    (IEnumerable<T> sequence,
    Predicate<T> filterFunc)
{
    if (sequence == null)
        throw new ArgumentNullException(nameof(sequence),
             “sequence must not be null”);
    if (filterFunc == null)
        throw new ArgumentNullException(
         “Predicate must not be null”);
    foreach (T item in sequence)
        if (filterFunc(item))
            yield return item;
}
```

輸入序列中的每個元素以 `Predicate` 方法求值。若 `Predicate` 回傳 `true`，該元素回傳成輸出序列的一部分。任何開發者可以對一個型別撰寫測試條件的方法，而該方法會與此過濾方法相容。

你也可以對序列取樣並回傳相隔 N 的元素：

```
public static IEnumerable<T> EveryNthItem<T>(
    IEnumerable<T> sequence, int period)
{
    var count = 0;
    foreach (T item in sequence)
        if (++count % period == 0)
            yield return item;
}
```

你可以對任何序列套用過濾器，只取出所選的項目。

`func` 可與任何列舉模式組合。下面我們建構一個轉換方法，它透過呼叫一個方法從現有序列建構一個新序列：

```
public static IEnumerable<T> Select<T>(
    IEnumerable<T> sequence, Func<T, T> method)
{
    // 省略 null 檢查
    foreach (T element in sequence)
        yield return method(element);
}
```

下面是如何呼叫 Select 將整數序列轉換成帶有這些整數平方的序列：

```
foreach (int i in Select(myInts, value => value * value))
    WriteLine(i);
```

Transform 方法無需回傳相同型別的元素。你可以修改 Transform 方法以支援型別的轉換：

```
public static IEnumerable<Tout> Select<Tin, Tout>(
    IEnumerable<Tin> sequence, Func<Tin, Tout> method)
{
    // 省略 null 檢查
    foreach (Tin element in sequence)
        yield return method(element);
}
```

以下面的方式呼叫這個版本：

```
foreach (string s in Select(myInts, value => value.ToString()))
    WriteLine(s);
```

如做法 31 所見，撰寫或使用這些方法不難。關鍵是將兩種操作分開：（1）序列的迭代；（2）對序列中的個別元素的操作。你以各種方式套用不具名的 **delegate** 或 **lambda** 表示式以建構程式。這些程序可作為你的應用程式的建構組件。你可以將序列的修改實作成函式（包括述詞的特例），且可使用 **action delegate**（或類似的定義）以於列舉部分元素時操作集合中的項目。

做法 31　被請求時產生序列項目

迭代方法不一定要以序列作為輸入參數。使用 yield return 的迭代方法可建構新的序列，基本上可變成元素序列的產生器。相較於在任何操作前建構整個集合，你可以在被請求時產生值。這表示你避免了建構未被序列消耗方使用的元素。

讓我們看個產生整數序列的例子。你或許會這樣寫：

```
static IList<int> CreateSequence(int numberOfElements,
    int startAt, int stepBy)
{
```

```
    var collection =
        new List<int>(numberOfElements);
    for (int i = 0; i < numberOfElements; i++)
        collection.Add(startAt + i * stepBy);

    return collection;
}
```

它是可行，但比使用 `yield return` 來建構序列較無效率。首先，此方式假設你要將結果放在 `List<double>` 中。若用戶想要將結果儲存在 `BindingList<double>` 等其他結構中，它們必須將其轉換：

```
var data = new
    BindingList<int>(CreateSequence(100,0,5).ToList());
```

這種結構或許會有微妙的 **bug**。`BindingList<T>` 的建構元並不複製清單元素而是使用清單建構元的儲存體位置。若用於初始化 `BindingList<T>` 的儲存體位置能被其他程式存取，你可能會遇到資料正確性錯誤。多個參考指向同一個儲存體位置。

還有，建構整個清單並不會給用戶程式根據特定條件停止產生函式的機會。`CreateSequence` 方法會產生請求數量的元素。依此程式的寫法，使用者無法停止此程序 - 無論是要換頁或其他原因。

還有，此方法可以作為資料序列多次轉換的第一階段（見做法 **31**）。在這種情況下，它會是管道中的瓶頸：在進行下一個步驟前，每個元素必須被建構並加入到內部集合中。

你可以製作迭代方法來排除這些限制：

```
static IEnumerable<int> CreateSequence(int numberOfElements,
    int startAt, int stepBy)
{
    for (var i = 0; i < numberOfElements; i++)
        yield return startAt + i * stepBy;
}
```

邏輯還是一樣：產生一系列的數字。

要注意這個版本的執行方式有項改變。程式每次列舉序列時會重新產生數字序列。由於程式總是產生相同的數字序列，此改變並不影響其行為。這個版本並未假設用戶程式會對儲存位置做什麼。若用戶程式想要 List<double> 值，有個建構元以 IEnumerable<double> 作為初始集合：

```
var listStorage = new List<int>(CreateSequence(100, 0, 5));
```

這是確保只產生一個數字序列的必要措施。你會以這種方式建構 BindingList<double> 集合：

```
var data = new
    BindingList<int>(CreateSequence(100,0,5).ToList());
```

這個程式看起來有點無效率。BindingList<T> 類別不支援取用 IEnumerable<T> 的建構元。但這並非無效率，因為 BindingList 保存對現有清單的參考；它沒有建構另一份拷貝。ToList() 建構保存由 CreateSequence 產生序列的所有元素的一個 List 物件。BindingList<int> 也持有該 List 物件。

使用下列方法能夠停止列舉，只要不請求下一個元素就好。此程式在兩種版本的 CreateSequence() 上都可以運作。但若你使用第一種 CreateSequence() 實作，無論呼叫方是否希望停止列舉清單，所有 1,000 個元素都會被產生。使用列舉程序版本則在找到第一個不相符值時生產就會斷開，如此可產生重大的效能提升。

```
// 使用不具名 delegate
var sequence = CreateSequence(10000, 0, 7).
    TakeWhile(delegate (int num) { return num < 1000; });

// 使用 lambda 記號法
var sequence = CreateSequence(10000, 0, 7).
    TakeWhile((num) => num < 1000);
```

當然，任何條件都可用來決定何時應該停止列舉。你可以檢查使用者是否希望繼續、從其他執行緒取得輸入，或執行你的應用程式必須做的事。列舉方法提供簡單的方式從序列的任何位置中斷列舉。此延遲執行意味只會產生被請求的元素。基本上，用戶程式只在確實使用到演算法中的元素時，演算法才會建構新的元素。

最好只在序列的消費者請求項目時才產生序列項目。若消費者只需要部分演算法執行其工作時，可以避免做額外的工作。這也許節省不多，但當建構元素的成本很大時，就可能節省很多。無論是什麼情況，建構序列的程式在你只產生必要的序列項目時會比較清楚。

做法 34 使用函式參數解耦

開發者通常以其最熟悉的語言功能來描述元件間的約定。對大部分開發者來說，這表示定義基底類別或界面以宣告新類別所需的方法，並根據界面定義撰寫程式。通常這是正確答案，但使用函式參數可讓其他開發者更容易運用你的元件與函式庫來建構程式。使用函式參數表示你的元件並不負責建構所需類別的具體型別。相對的，你的元件透過抽象定義使用任何相依關係。

你應該知道界面與類別的差別，但有時定義與實作界面對特定使用方式太麻煩。或許你會使用傳統的物件導向技巧，但其他技巧可做出更簡單的 API，可以使用 **delegate** 建構約定以減少對用戶程式的需求。

你的挑戰是隔離你的工作與相依性，以及對使用你程式的開發者的隱含假設。有各種原因讓你的程式陷入兩難。你的程式越依賴其他程式，則進行單元測試或在其他環境使用你的程式就越困難。另一方面，越是要求使用你程式的開發者實作特定模式，則對他們加上的限制就越多。

你可以使用函式參數將使用你的元件的程式與你的元件解耦。但這些方法都有代價。若以解耦技巧解開必須一起運作的程式間的耦合，工作會增加且會降低對使用者的清晰度。你必須平衡開發者的需求與解耦技巧能提供的可理解性。此外，實作鬆散耦合 - 使用 **delegate** 或其他通訊機制 - 也意味著你必須處理編譯器所提供的檢查。

在光譜的一端，你可能會為你的用戶類別指定基底類別。這麼做是讓用戶開發運用你的元件的程式最簡單的方式。約定很清楚：繼承基底類別，實作已知的抽象（或其他虛擬）方法就可以。此外，你可以在抽象基底類別實作任何共通的功能。你的元件使用者不需要重新實作該程式。

從你的元件的這一端來看，這種方式的工作也較少。你可以假設特定行為已經實作了，編譯器不會讓他人繼承類別而不提供所有抽象方法的實作。沒有辦法可確保實作的正確，但你知道存在合適的方法。

然而，強迫用戶程式繼承你定義的基底類別，是要求用戶程式特定行為最受限的方式。建構要求基底類別的元件會對使用者產生非常多的限制，你要求特定的類別階層，而沒有其他的使用方式。

建構界面並根據它寫程式可產生比依靠基底類別更鬆散的耦合。你可能會建構界面並強迫用戶程式開發者實作該界面。這種方式產生的關係類似使用基底類別所建構的關係，其中只有兩種重要差別：首先，使用界面不會強制要求使用者的類別階層；其次，你無法簡單的為用戶程式提供必要的預設實作行為。

通常，這些機制對你的目的來說都需要太多的工作。你真的需要定義界面嗎？或定義 **delegate** 格式等更鬆散的耦合會更好？

你已經在做法 **32** 看過一個例子。`List.RemoveAll()` 方法的格式取用型別為 `Predicate<T>` 的 **delegate**：

```
void List<T>.RemoveAll(Predicate<T> match);
```

.NET Framework 的設計者可以用界面的定義來實作此方法：

```
// 不當的多餘耦合
public interface IPredicate<T>
{
    bool Match(T soughtObject);
}
public class List<T>
{
    public void RemoveAll(IPredicate<T> match)
    {
        // 省略
    }
    // 省略其他 API
}
// 使用這個版本需要更多的工作
public class MyPredicate : IPredicate<int>
{
```

```
    public bool Match(int target) =>
        target < 100;
}
```

回頭看看做法 31 就會發現使用為 `List<T>` 定義的版本有多容易。通常，使用 **delegate** 或其他鬆散耦合機制定義你的界面，能讓使用你的類別的開發者更輕鬆。

使用 **delegate** 替代界面的原因是 **delegate** 並非型別的基本屬性。它不算是方法。

.NET Framework 有幾個界面只有一個方法。`IComparable<T>` 與 `IEquatable<T>` 是非常好的界面定義。實作這些界面為你的型別表達一些事情：它支援比較或相等性。實作假設性的 `IPredicate<T>` 並未表達型別的任何事情。一個 **API** 只需要一個方法定義。

你覺得需要定義界面或建構基底類別時通常可以使用函式程式與泛型方法。做法 31 合併兩個序列的 `Zip` 方法：

```
public static IEnumerable<string> Zip(
    IEnumerable<string> first,
    IEnumerable<string> second)
{
    using (var firstSequence = first.GetEnumerator())
    {
        using (var secondSequence = second.GetEnumerator())
        {
            while (firstSequence.MoveNext() &&
                secondSequence.MoveNext())
            {
                yield return string.Format( “{0} {1}” ,
                    firstSequence.Current,
                    secondSequence.Current);
            }
        }
    }
}
```

你可以製作一個泛型方法，並使用函式參數來建構輸出序列：

```
public static IEnumerable<TResult> Zip<T1, T2, TResult>(
    IEnumerable<T1> first,
    IEnumerable<T2> second, Func<T1, T2, TResult> zipper)
{
    using (var firstSequence = first.GetEnumerator())
    {
        using (var secondSequence =
            second.GetEnumerator())
        {
            while (firstSequence.MoveNext() &&
                secondSequence.MoveNext())
            {
                yield return zipper(firstSequence.Current,
                    secondSequence.Current);
            }
        }
    }
}
```

呼叫方現在必須定義 `zipper` 的內容：

```
var result = Zip(first, second, (one, two) =>
    string.Format("{0} {1}", one, two));
```

這在 `Zip` 方法與呼叫方產生較鬆散的耦合。

做法 33（見本章）的 `CreateSequence` 方法可因一些改變而受惠。此版本的做法 33 建構一系列的整數。你可以製作泛型方法並使用函式參數來指定序列應該如何產生：

```
public static IEnumerable<T> CreateSequence<T>(
    int numberOfElements,
    Func<T> generator)
{
    for (var i = 0; i < numberOfElements; i++)
        yield return generator();
}
```

呼叫方如此定義原始行為：

```
var startAt = 0;
var nextValue = 5;
var sequence = CreateSequence(1000,
    () => startAt += nextValue);
```

其他時間，你會想要對一個序列上的所有項目執行一個演算法並回傳單一值。舉例來說，下面的方法產生整數序列的加總值：

```
public static int Sum(IEnumerable<int> nums)
{
    var total = 0;
    foreach (int num in nums)
    {
        total += num;
    }
    return total;
}
```

你可以將此方法做成通用的加總程序，製作 Sum 演算法並以 **delegate** 定義取代：

```
public static T Sum<T>(IEnumerable<T> sequence, T total,
    Func<T, T, T> accumulator)
{
    foreach (T item in sequence)
    {
        total = accumulator(total, item);
    }
    return total;
}
```

以下列方式呼叫：

```
var total = 0;
total = Sum(sequence, total, (sum, num) => sum + num);
```

Sum 方法還是很受限，它必須使用相同型別的序列、回傳值與初始值。你會希望使用不同的型別：

```
var peeps = new List<Employee>();
// 從其他地方加入員工
// 計算薪資加總：
var totalSalary = Sum(peeps, 0M, (person, sum) =>
    sum + person.Salary);
```

只需對 Sum 方法的定義稍加修改，容許不同的序列元素與加總參數型別。讓我們將名稱一併修改為 BCL 中更通用的 Fold：

```
public static TResult Fold<T, TResult>(
    IEnumerable<T> sequence,
    TResult total,
    Func<T, TResult, TResult> accumulator)
{

    foreach (T item in sequence)
    {
        total = accumulator(item, total);
    }
    return total;
}
```

以函式作為參數可將演算法與其所操作的資料型別分離得很好，但在解耦時卻增加解耦元件間通訊所需確保適當處理錯誤的工作。舉例來說，假設你寫了定義事件的程式。你知道必須在發出事件時檢查事件成員是否為 null，用戶程式可能沒有建構事件處理程序，使用 **delegate** 建構界面時需要相同的工作。你的用戶傳遞空的 **delegate** 時的正確行為是什麼？它是個例外或有正確的預設行為？若用戶程式拋出例外會怎樣？你能夠復原嗎？如何復原？

最後，當你從繼承轉換成以 **delegate** 定義時，你必須認識到會有與保存物件或界面參考一樣的執行期耦合。如果你的物件儲存可稍後呼叫 **delegate** 的拷貝，你的物件現在會控制 **delegate** 所參考的物件生命週期。你可能會延長這些物件的生命週期，這與你的物件持有稍後會叫用的其他物件的參考（儲存界面或基底類別的參考）無異，但這比較難從閱讀程式就看出來。

預設的選擇仍是建構表示你的元件如何與用戶程式溝通的界面約定。抽象基底類別給你額外的能力來提供預設實作，否則得要用戶程式自行處理。定義你預期的 **delegate** 給你最大的彈性，但也意味著工具支援較少。更多的工作帶來更大的彈性。

做法 35 不要過載擴充方法

前面（做法 27 與 28）討論過建構界面或型別擴充方法的三種原因：對界面加入預設實作、對封閉的泛型型別建構行為、建構可組合的界面。但擴充方法並非一定是表達你的設計的好方式。在這些狀況中，你對現有型別定義作一些加強，但這些加強不會從基本改變型別的行為。

做法 27 解釋你可以用擴充方法提供一種能以最小界面定義通用動作的預設實作。你可能會想要使用相同的技巧來加強類別型別，甚至會想要建構透過改變所使用的命名空間來替換的多個版本的類別擴充。請不要這麼做！擴充方法提供實作界面型別的預設實作，但有其他更好的方式來擴充類別型別。過度使用與誤用擴充方法，很快就會導致方法衝突並增加維護的成本。

讓我們從誤用擴充方法的例子開始。假設有個 Person 類別是由其他函式庫所產生：

```
public sealed class Person
{
    public string FirstName
    {
        get;
        set;
    }
    public string LastName
    {
        get;
        set;
    }
}
```

你或許會考慮撰寫擴充方法來輸出人名：

```
// 錯誤的開始
// 使用擴充方法來擴充類別
namespace ConsoleExtensions
{
    public static class ConsoleReport
    {
        public static string Format(this Person target) =>
            $"{target.LastName,20}, {target.FirstName,15}";
```

```
    }
}
```

產生輸出很簡單：

```
static void Main(string[] args)
{
    List<Person> somePresidents =
        new List<Person>{
    new Person{
        FirstName = “George”,
        LastName = “Washington” },
    new Person{
        FirstName = “Thomas”,
        LastName = “Jefferson” },
    new Person{
        FirstName = “Abe”,
        LastName = “Lincoln” }
        };

    foreach (Person p in somePresidents)
        Console.WriteLine(p.Format());
}
```

這看起來毫無問題。但後來需求改變了，你發現必須以 XML 格式輸出。有些人可能會撰寫如下的方法：

```
// 更糟糕
// 不同命名空間下的
// 模糊的擴充方法
namespace XmlExtensions
{
    public static class XmlReport
    {
        public static string Format(this Person target) =>
            new XElement(“Person”,
                new XElement(“LastName”, target.LastName),
                new XElement(“FirstName”, target.FirstName)
                ).ToString();
    }
}
```

在原始檔使用 using 陳述以變換輸出的格式，是擴充方法的誤用。這是很脆弱的型別擴充方式，若開發者使用錯誤的命名空間，程式的行為會改變。若忘記使用任何擴充命名空間，程式將無法編譯。若同時需要兩個命名空間中不同的方法，開發者必須根據所需的擴充方法，將類別定義分割成不同的檔案。同時使用不同的命名空間會導致編譯器的模糊參考錯誤。

很明顯你需要以不同方式實作此功能。擴充方法根據物件的編譯期型別呼叫，而變換根據命名空間決定呼叫的方法，使得此策略更為脆弱。

這種功能並非根據你擴充的型別而定。將 Person 物件輸出格式化為 XML 或純文字並非是 Person 的一部分，反而更靠近使用 Person 物件的外部環境。

擴充方法應該用於加強型別的功能，你只應該在增加邏輯上屬於該型別的功能時才製作擴充方法。做法 27 與 28 解釋了界面參數與封閉型別的兩項技巧，你若檢視它們的範例，會發現那些擴充方法建構出對該型別的使用者來說是屬於該型別的方法。

與這一節的範例比較，Format 方法應該是使用 Person 型別的方法而不屬於 Person 型別。事實上，若有可能，它們應該以不同的方法名稱放在同一個類別中：

```
public static class PersonReports
{
    public static string FormatAsText(Person target)=>
        $"{target.LastName,20}, {target.FirstName,15}";
    public static string FormatAsXML(Person target) =>
        new XElement("Person",
            new XElement("LastName", target.LastName),
            new XElement("FirstName", target.FirstName)
            ).ToString();
}
```

此類別帶有這兩種方法的靜態方法，而不同的名稱清楚的反映個別方法的目的。你提供兩個方法給類別的使用者，且不會在公開界面或使用感覺上產生模糊性。開發者在有需要時可以使用它們。你沒有因為不同命名空間下同一格式的方法而引發任何模糊性。這很重要，因為少數開發者會假設改變 using 陳述會改變程式的執行期行為。它們會假設這樣會引起編譯期錯誤而非執行期錯誤。

當然，改變方法名稱使它們不再衝突後，你可以再度讓它們變成擴充方法。這種方法沒什麼好處，看起來沒有擴充該型別而是使用該型別。但由於名稱不會衝突，你可以將兩種方法放在同一個命名空間與類別下。這樣可以避免前面範例中的陷阱。

你應該視型別的擴充方法為單一個全域方法。擴充方法不應該在命名空間下過載，若你發現你必須建構多個相同格式的擴充方法，請停下來。相反的，應該要改變方法格式並考慮建構普通的靜態方法。這種做法可避免編譯器根據 `using` 陳述選擇過載而導致的模糊性。

做法 36　認識查詢表示式如何對應方法呼叫

LINQ 根據兩個概念創建：查詢語言與轉譯查詢語言成一組方法。C# 編譯器將使用查詢語言寫出的查詢表示式轉換成方法呼叫。

每個查詢表示式會對應到一或多個方法呼叫。你應該從兩個角度認識這種對應。從類別使用者的角度，你必須理解你的查詢表示式只是方法呼叫。`where` 句子會轉換成 `Where()` 方法的呼叫，加上適當的參數。作為一個類別設計者，你應該評估平台提供的方法實作，並判斷能否為你的型別建構更好的實作。若不能，則應該託付給基底函式庫的版本。但當你建構更好的版本時，必須確保能完全理解查詢表示式之方法呼叫的轉換。確保你的方法格式能正確處理每一種轉換是你的責任。對某些查詢表示式來說，正確的路徑是很明顯的，但有些較複雜的表示式會比較難以消化。

完整的查詢表示式模式有 11 個方法。下面列出的定義（經 Microsoft Corporation 授權）來自 Anders Hejlsberg、Mads Torgersen、Scott Wiltamuth、與 Peter Golde 所著的 The C# Programming Language Third Edition（Microsoft Corporation，2009）§ 7.15.3：

```
delegate R Func<T1, R>(T1 arg1);
delegate R Func<T1, T2, R>(T1 arg1, T2 arg2);
class C
{
    public C<T> Cast<T>();
}
```

```
class C<T> : C
{
    public C<T> Where(Func<T, bool> predicate);
    public C<U> Select<U>(Func<T, U> selector);
    public C<V> SelectMany<U, V>(Func<T, C<U>> selector,
        Func<T, U, V> resultSelector);
    public C<V> Join<U, K, V>(C<U> inner,
        Func<T, K> outerKeySelector,
        Func<U, K> innerKeySelector,
        Func<T, U, V> resultSelector);
    public C<V> GroupJoin<U, K, V>(C<U> inner,
        Func<T, K> outerKeySelector,
        Func<U, K> innerKeySelector,
        Func<T, C<U>, V> resultSelector);
    public O<T> OrderBy<K>(Func<T, K> keySelector);
    public O<T> OrderByDescending<K>(Func<T, K> keySelector);
    public C<G<K, T>> GroupBy<K>(Func<T, K> keySelector);
    public C<G<K, E>> GroupBy<K, E>(Func<T, K> keySelector,
        Func<T, E> elementSelector);
}

class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T, K> keySelector);
    public O<T> ThenByDescending<K>(Func<T, K> keySelector);
}

class G<K, T> : C<T>
{
    public K Key { get; }
}
```

.NET 的基底函式庫提供了這種模式的通用參考實作。`System.Linq.Enumerable` 提供實作 `IEnumerable<T>` 查詢表示式模式的擴充方法。`System.Linq.Queryable` 對 `IQueryable<T>` 提供類似的擴充方法，以支援查詢提供者轉譯查詢成其他執行格式的功能（舉例來說，LINQ to SQL 的實作將查詢表示式轉換成由 SQL 資料庫引擎執行的 SQL 查詢）。作為類別使用者，你或許會在大部分的查詢中使用這兩個參考實作。

其次，作為類別作者，你可以建構實作 `IEnumerable<T>` 或 `IQueryable<T>`（或衍生自 `IEnumerable<T>` 或 `IQueryable<T>` 的封閉泛型型別）的資料來源，此時

你的型別已經實作了查詢表示式模式。你的型別已經有此實作是因為你使用了定義於基底函式庫的擴充方法。

繼續進行之前，你應該要知道 C# 語言並沒有在查詢表示式模式上強制規定任何的執行語意。你可以建構符合查詢方法格式的方法，並在內部執行任何動作。編譯器無法驗證你的 where 方法是否滿足查詢表示式模式的預期，它只能確保滿足語意約定。這種行為與界面方法沒有差別。舉例來說，你可以建構做任何動作的界面方法，無論它是否符合使用者的預期。

當然，這並不表示應該考慮這種計劃。若你實作了查詢表示式模式方法，你應該要確保它的行為與參考實作在語法與語意上是一致的。除了效能差異外，呼叫方應該無法判斷出是使用了你的方法或參考實作的方法。

從查詢表示式轉譯成方法叫用是個複雜的迭代過程，編譯器重複將表示式轉譯成方法，直到所有表示式都被轉譯為止。此外，編譯器對轉譯的執行有指定的順序，但我不是以這種順序解釋。編譯器順序對編譯器是簡單的，且在 C# 規格中有說明，我是根據容易向人類解釋的順序做選擇。下方以較小、較簡單的例子來討論轉譯。

讓我們檢視下面的查詢中的 where、select、range 變數：

```
var numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var smallNumbers = from n in numbers
                   where n < 5
                   select n;
```

from n in numbers 這個表示式將範圍變數 n 綁定給 numbers 的每個值。where 句子定義會被轉譯成 where 方法的過濾器。where n < 5 這個表示式會轉譯成：

```
numbers.Where(n => n < 5);
```

Where 只是個過濾器。Where 的輸出是輸入序列中滿足述詞元素的子集。輸入與輸出序列必須為相同的型別，正確的 Where 方法不能修改輸入序列中的項目（使用者自定的述詞可以修改項目，但這不是查詢表示式模式的責任）。

where 方法可實作成 numbers 可存取的實例方法或符合 numbers 型別的擴充方法。此例中，numbers 是 int 的陣列。因此方法呼叫中的 n 必須是個整數。

Where 是查詢表示式轉譯成方法呼叫中最簡單的轉譯。在我們繼續下去之前，讓我們更深入探索它是如何運作與它對轉譯的意義，編譯器在過載解析或型別綁定之前完成查詢表示式至方法呼叫的轉譯。編譯器在將查詢表示式轉譯成方法呼叫時不知道是否有任何的可用方法，它不檢視型別且不檢查是否有可用的擴充方法。它只是將查詢表示式轉譯成方法呼叫。在所有查詢被轉譯成方法呼叫語法後，編譯器開始搜尋可用的方法並判斷何者是最佳方法。

接下來，你可以將此簡單的範例擴展以在查詢中加入 select 表示式。select 句子會轉譯成 Select 方法。但在特殊情況下，Select 方法可以用最佳化去掉，例如退化選取（**degenerate select**）以 range 變數選取。退化選取可以去掉是因為輸出序列不等於輸入序列。此範例有個 where 句子，它打破輸入序列與輸出序列間的相等關係，因此此查詢的最終方法呼叫版本如下：

```
var smallNumbers = numbers.Where(n => n < 5);
```

select 句子被刪除是因為它變成多餘的。這是安全的，因為 select 操作的是其他查詢表示式的計算結果（此例中的 where）。

當 select 並非操作其他表示式的計算結果時，它不能用最佳化去掉。以下面的查詢為例：

```
var allNumbers = from n in numbers select n;
```

它會被轉譯成這個方法呼叫：

```
var allNumbers = numbers.Select(n => n);
```

趁我們討論這個主題時，請注意，select 通常用於將輸入元素轉換或投射成不同的元素或不同的型別。下面的查詢修改計算結果值：

```
var numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var smallNumbers = from n in numbers
                   where n < 5
                   select n * n;
```

或者你可以將輸入序列轉換成不同的型別：

```
var numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var squares = from n in numbers
              select new { Number = n, Square = n * n };
```

select 句子對應與查詢表示式模式中格式相符的 Select 方法：

```
var squares = numbers.Select(n =>
    new { Number = n, Square = n * n });
```

select 將輸入型別轉換成輸出型別。select 方法必須對每個輸入元素產生一個輸出元素。還有，正確的 Select 實作必定不會修改輸入序列中的項目。

以上是簡單的查詢表示式。現在我們討論一些較不明顯的轉換。

排序關係對應 OrderBy 與 ThenBy 方法或 OrderByDescending 與 ThenByDescending。以這個查詢來說：

```
var people = from e in employees
             where e.Age > 30
             orderby e.LastName, e.FirstName, e.Age
             select e;
```

它被轉譯成：

```
var people = employees.Where(e => e.Age > 30).
    OrderBy(e => e.LastName).
    ThenBy(e => e.FirstName).
    ThenBy(e => e.Age);
```

請注意，查詢表示式模式定義 ThenBy 操作 OrderBy 或 ThenBy 回傳的序列。這些序列能攜帶於排序鍵相等時讓 ThenBy 操作排序過的子集的標記。

這種轉譯與 orderby 句子以不同句子表示時不同。下面的查詢以 LastName 對序列完整排序，然後再以 FirstName 完整排序，然後再以 Age 完整排序：

```
// 錯。序列排序三次。
var people = from e in employees
             where e.Age > 30
             orderby e.LastName
             orderby e.FirstName
             orderby e.Age
             select e;
```

如另一個查詢所示，你可以指定反向的 orderby 句子：

```
var people = from e in employees
             where e.Age > 30
             orderby e.LastName descending, e.FirstName, e.Age
             select e;
```

OrderBy 方法建構不同的輸出序列型別，因此 thenby 句子能更有效率且型別對整體查詢是正確的。ThenBy 不能操作未排序的序列，只能操作排序過的序列（此範例中的型別為 O<T>）。次範圍已經排序且標記過。若你為型別建構了自定的 OrderBy 與 ThenBy 方法，你必須依循這個規則。你必須對每個排序過的次範圍加上標示符號，以讓後續的 ThenBy 句子可以正確的運作。ThenBy 方法必須加上型別，以正確的排序 OrderBy 或 ThenBy 方法的輸出的次範圍。

以上所說關於 OrderBy 與 ThenBy 的部分同樣適用於 OrderByDescending 與 ThenByDescending。事實上，若你的型別自定這些方法，你應該要四個全部實作。

其餘的表示式轉譯涉及多重步驟。這些查詢涉及群組或引發延續性（continuation）的多個 from 句子。具有延續性的查詢表示式會轉譯成套疊的查詢，然後這些套疊的查詢被轉譯成方法。以下是具有延續性的簡單查詢：

```
var results = from e in employees
              group e by e.Department into d
              select new
              {
                  Department = d.Key,
                  Size = d.Count()
              };
```

在任何其他轉譯執行前，延續性先被轉譯成套疊的查詢：

```
var results = from d in
              from e in employees group e by e.Department
              select new { Department = d.Key,
              Size = d.Count()};
```

建構套疊的查詢後，方法轉譯成這樣：

```
var results = employees.GroupBy(e => e.Department).
    Select(d => new { Department = d.Key,
    Size = d.Count() });
```

前面的查詢顯示一個回傳單一序列的 GroupBy。此查詢表示式模式中其他的 GroupBy 方法回傳群組序列，其中每個群組帶有一個鍵與一個值清單：

```
var results = from e in employees
              group e by e.Department into d
              select new
              {
                  Department = d.Key,
                  Employees = d.AsEnumerable()
              };
```

此查詢對應下列的方法呼叫：

```
var results2 = employees.GroupBy(e => e.Department).
    Select(d => new {
        Department = d.Key,
        Employees = d.AsEnumerable()
    });
```

GroupBy 方法產生一個鍵 / 值清單對的序列；鍵是群組選擇器，值是群組中的項目序列。此查詢的 select 句子會為每個群組的值建構新的物件。但輸出應該要是鍵 / 值對的序列，其值帶有輸入序列中屬於特定群組的每個項目建構出的一些元素。

最後要認識的方法是 SelectMany、Join 與 GroupJoin。這三個方法很複雜，因為它們操作多個輸入序列。實作執行轉譯的方法執行多個序列間的列舉，然後將計算結果攤平成單一輸出序列。SelectMany 產生兩個來源序列的笛卡兒積（**Cartesian product**）。以下面的查詢為例：

```
int[] odds = { 1, 3, 5, 7 };
int[] evens = { 2, 4, 6, 8 };
var pairs = from oddNumber in odds
            from evenNumber in evens
            select new
            {
```

```
            oddNumber,
            evenNumber,
            Sum = oddNumber + evenNumber
        };
```

它產生一個具有 16 個元素的序列：

```
1,2, 3
1,4, 5
1,6, 7
1,8, 9
3,2, 5
3,4, 7
3,6, 9
3,8, 11
5,2, 7
5,4, 9
5,6, 11
5,8, 13
7,2, 9
7,4, 11
7,6, 13
7,8, 15
```

帶有多個 `select` 句子的查詢表示式被轉譯成 `SelectMany` 方法呼叫。此查詢會被轉譯成下列的 `SelectMany` 呼叫：

```
int[] odds = { 1, 3, 5, 7 };
int[] evens = { 2, 4, 6, 8 };
var values = odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new {
        oddNumber,
        evenNumber,
        Sum = oddNumber + evenNumber
    });
```

`SelectMany` 的第一個參數是將第一個來源序列的每個元素對應到第二個來源序列的元素序列，第二個參數（輸出選擇器）從兩個序列的成對項目中建構出投射。

SelectMany() 迭代第一個序列。對第一個序列中的每一個值，它迭代第二個序列，從成對的輸入值產生結果值。被選取的輸出，由兩個序列每個組合值的攤平序列中的每個元素呼叫。一種可能的 SelectMany 實作如下：

```
static IEnumerable<TOutput> SelectMany<T1, T2, TOutput>(
    this IEnumerable<T1> src,
    Func<T1, IEnumerable<T2>> inputSelector,
    Func<T1, T2, TOutput> resultSelector)
{
    foreach (T1 first in src)
    {
        foreach (T2 second in inputSelector(first))
            yield return resultSelector(first, second);
    }
}
```

第一個輸入序列會被迭代，然後使用輸入序列目前的值迭代第二個輸入序列。這很重要，因為第二個序列的輸入選擇器會依靠第一個序列目前的值，然後隨著每一對元素的產生來呼叫結果選擇器。

若你的查詢有更多的表示式且 SelectMany 並沒有建構最終結果，則 SelectMany 建構帶有輸入序列每個下面的資料組。該資料組的序列是後續表示式的輸入序列。以下面修改過的查詢版本為例：

```
int[] odds = { 1, 3, 5, 7 };
int[] evens = { 2, 4, 6, 8 };
var values = from oddNumber in odds
             from evenNumber in evens
             where oddNumber > evenNumber
             select new
             {
                 oddNumber,
                 evenNumber,
                 Sum = oddNumber + evenNumber
             };
```

它產生下列的 SelectMany 方法呼叫：

```
odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new { oddNumber, evenNumber });
```

然後整個查詢轉譯成下面這個陳述：

```
var values = odds.SelectMany(oddNumber => evens,
    (oddNumber, evenNumber) =>
    new { oddNumber, evenNumber }).
    Where(pair => pair.oddNumber > pair.evenNumber).
    Select(pair => new {
        pair.oddNumber,
        pair.evenNumber,
        Sum = pair.oddNumber + pair.evenNumber
    });
```

你在編譯器將多個 from 句子轉譯成 SelectMany 方法呼叫時，處理 SelectMany 的過程中可以發現一個有趣的現象，SelectMany 組合的很好。兩個以上的 from 句子會產生一個以上的 SelectMany() 方法呼叫。從第一個 SelectMany() 呼叫產生的結果對會餵給第二個 SelectMany()，它會產生三重資料組。三重資料組帶有三個序列的各種排列組合。以下面的查詢來說：

```
var triples = from n in new int[] { 1, 2, 3 }
              from s in new string[] { "one", "two",
               "three" }
              from r in new string[] { "I", "II", "III" }
              select new { Arabic = n, Word = s, Roman = r };
```

它會被轉譯成下面的方法呼叫：

```
var numbers = new int[] { 1, 2, 3 };
var words = new string[] { "one", "two", "three" };
var romanNumerals = new string[] { "I", "II", "III" };
var triples = numbers.SelectMany(n => words,
    (n, s) => new { n, s }).
    SelectMany(pair => romanNumerals,
    (pair, n) =>
        new { Arabic = pair.n, Word = pair.s, Roman = n });
```

如你所見，你可以透過套用多個 SelectMany() 呼叫，以從三個擴展到任意數量的輸入序列。後面的範例還展示 SelectMany 如何對你的查詢引入不具名型別，從 SelectMany() 回傳的序列是某種不具名型別的序列。

現在讓我們檢視另外兩種你必須認識的轉譯：Join 與 GroupJoin，這兩者都應用在聯合表示式上。當聯合表示式帶有 into 句子時會使用 GroupJoin，而 Join 用於沒有 into 句子的聯合表示式。

沒有 into 句子的聯合表示式像是這樣：

```
var numbers = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var labels = new string[] { "0", "1", "2", "3", "4", "5" };
var query = from num in numbers
            join label in labels on num.ToString() equals label
            select new { num, label };
```

它會轉譯成：

```
var query = numbers.Join(labels, num => num.ToString(),
    label => label, (num, label) => new { num, label });
```

into 句子建構次區分清單：

```
var groups = from p in projects
             join t in tasks on p equals t.Parent
             into projTasks
             select new { Project = p, projTasks };
```

它會轉譯成 GroupJoin：

```
var groups = projects.GroupJoin(tasks,
    p => p, t => t.Parent, (p, projTasks) =>
        new { Project = p, TaskList = projTasks });
```

轉換所有表示式成方法呼叫的整個過程很複雜，且通常需要多個步驟。

好消息是大部分情況下編譯器會執行正確的轉譯。由於你的型別實作了 IEnumerable<T>，你的型別的使用者會得到正確的行為。

但你或許會想要自行實作查詢表示式模式的方法。或許你的集合型別總是以特定的鍵排序，所以你可以在 OrderBy 方法走捷徑。或許你的型別顯露清單的清單，這表示你的 GroupBy 與 GroupJoin 可以更有效率的實作。

或許你的野心更大，想要建構提供者並實作整個模式。此時你必須認識每個查詢方法的行為並知道要實作什麼。參考範例並在開始進行你的實作之前確定你知道每個查詢方法的預期行為。

你定義的型別建構某種集合模型。使用你的型別的開發者會預期你的集合與其他集合型別的使用方式相同並內建查詢語法。只要是集合模型的型別有支援 `IEnumerable<T>` 就會符合預期。但你的型別可以使用其內部規格來改善預設實作。當你選擇如此進行時，要確保你的型別符合各種查詢模式的約定。

做法 37 在查詢中偏好惰性求值而非積極求值

定義查詢時，你其實沒有取得資料也沒有產生序列，實際上你只是定義你真的迭代查詢時要執行的步驟。這表示每次執行查詢時，你從第一原理執行整個配方，這通常是正確的行為。每個新的列舉產生新的結果，這稱為惰性求值（**lazy evaluation**），但通常這不是你要的。抓取一組變數時，你想要一次抓完並立即讀取，這稱為積極求值（**eager evaluation**）。

撰寫要列舉一次以上的查詢時，你必須考慮你想要的行為。你想要資料的快照？或是想要建構程式以於執行時產生值的序列？

這個概念會大幅改變你習慣的工作方式。你或許視程式為立即執行的東西，但在 **LINQ** 中，程式視為資料。加入其中的 **lambda** 表示式會在之後叫用。不只如此，若提供者以表示式樹替代 **delegate**，這些表示式樹可經由結合新的表示式到同一個表示式樹而於之後組合。

讓我們從解釋惰性與積極求值之間的差異的範例開始。下面的程式產生一個序列然後迭代此序列三次，在迭代間會有個暫停：

```
private static IEnumerable<TResult>
    Generate<TResult>(int number, Func<TResult> generator)
{
    for (var i = 0; i < number; i++)
        yield return generator();
}

private static void LazyEvaluation()
{
```

```
    WriteLine($" Start time for Test One: {DateTime.Now:T}" );
    var sequence = Generate(10, () => DateTime.Now);

    WriteLine( "Waiting....\tPress Return" );
    ReadLine();

    WriteLine( "Iterating..." );
    foreach (var value in sequence)
        WriteLine($" {value:T}" );

    WriteLine( "Waiting....\tPress Return" );
    ReadLine();
    WriteLine( "Iterating..." );
    foreach (var value in sequence)
        WriteLine($" {value:T}" );
}
```

下面是範例的輸出：

```
Start time for Test One: 6:43:23 PM
Waiting....    Press Return

Iterating...
6:43:31 PM
...
6:43:31 PM
Waiting....    Press Return

Iterating...
6:43:42 PM
...
6:43:42 PM
```

注意這個惰性求值範例中的序列是在被迭代時產生的，不同的時間戳記就是證據。序列變數並沒有保存建構出的元素，相反的，它保存建構序列的表示式樹。你應該自己逐步執行這個程式一次以檢視表示式是在什麼時候求值。這是最有啟發性的學習 LINQ 的查詢如何求值的方式。

你可以使用這個功能來從現有查詢組合查詢。相較於從第一個查詢讀取結果並以另一個步驟來處理，你可以在不同的步驟組合查詢然後只需執行組合查詢一次。舉例來說，若修改查詢以標準格式回傳時間：

```
var sequence1 = Generate(10, () => DateTime.Now);
var sequence2 = from value in sequence1
                select value.ToUniversalTime();
```

序列 1 與序列 2 共用功能組合而非資料。序列 2 並非以列舉序列 1 的值並修改每個值來產生，相對的，它是經由執行產生序列 1 的程式並接著產生序列 2 的程式來建構。如果你在不同時間迭代兩個序列，你會看到不相關的序列。序列 2 不會帶有從序列 1 轉換的值，相對的，它會帶有全新的值。它並不產生資料序列然後轉換整個序列成標準時間。相對的，每一行程式使用標準時間產生一組值。

理論上，查詢表示式可以操作無限序列。能夠這樣做的原因是因為它們懶。若正確的撰寫，它們檢視序列的第一個部分然後在找到答案時終止。另一方面，某些查詢表示式在可以產生答案前必須讀取整個序列。認識何時會產生這種瓶頸可以幫助你建構不會有效能影響的自然查詢。成為，這種認識會幫助你避免需要完整序列且會產生瓶頸的次數。

以下面的小程式來說：

```
static void Main(string[] args)
{
    var answers = from number in AllNumbers()
                  select number;

    var smallNumbers = answers.Take(10);
    foreach (var num in smallNumbers)
        Console.WriteLine(num);
}

static IEnumerable<int> AllNumbers()
{
    var number = 0;
    while (number < int.MaxValue)
    {
        yield return number++;
    }
}
```

此範例展示我所說的不需完整序列的方法。此方法輸出的序列為數字的 0,1,2,3,4,5,6,7,8,9。AllNumbers() 也可以產生無限序列（沒錯，它最終會溢流，但在發生前你早已失去耐性）。

它運作的很快的原因是它不需要整個序列。Take() 方法回傳序列的前 N 個物件，因此其他都不重要。

但若你撰寫序列如下，你的程式會跑不完：

```
class Program
{
    static void Main(string[] args)
    {
        var answers = from number in AllNumbers()
                      where number < 10
                      select number;

        foreach (var num in answers)
            Console.WriteLine(num);
    }
}
```

它會不停的執行（或遇到 int.MaxValue），因為此查詢必須檢視每個數字以判斷哪些符合條件。這個版本的邏輯需要整個序列。

有些查詢運算子必須要整個序列才能正確操作。Where 使用整個序列，它逐個檢視每個元素且可能會產生另一個無限序列。OrderBy 需要整個序列來執行排序。Max 與 Min 類似 Where 需要整個序列，沒有辦法不檢視序列的每個元素來執行這些操作。當你需要這些功能時，就會使用這些方法。

你必須思考使用需要整個序列的方法的後果。如你所見，若序列有可能是無限的，你必須避免任何需要整個序列的方法。其次，就算序列不是無限的，任何過濾序列的查詢方法應該在序列的前面載入。若你的查詢中的第一個步驟會從集合中刪除一些元素，這對剩下的查詢工作在效能上有正面的影響。

舉例來說，下面兩個查詢產生相同的結果，但第二個查詢的執行較快。成熟的提供者會將查詢最佳化，兩個查詢的效能一樣。但在 LINQ to Object 的實作中（由 System.Linq.Enumerable 提供），所有的產品都會讀取與排序，然後過濾產品序列。

```
// 過濾前排序
var sortedProductsSlow =
    from p in products
    orderby p.UnitsInStock descending
    where p.UnitsInStock > 100
    select p;

// 排序前過濾
var sortedProductsFast =
    from p in products
    where p.UnitsInStock > 100
    orderby p.UnitsInStock descending
    select p;
```

注意第一個查詢將整個序列排序然後拋出庫存小於 100 的產品。第二個查詢先過濾序列，再排序比較小的序列。有時候，知道一個方法是否需要整個序列的差別在於不會結束的演算法與非常快結束的演算法。你必須理解哪個方法需要完整的序列，並在查詢表示式的最後再執行。

目前我只給了你幾個在查詢中使用惰性求值的理由。大部分情況下，這是最好的辦法。但有時候，你需要某個時間點的值快照。有兩個方法可用來立即產生序列並將結果儲存在容器中：`ToList()` 與 `ToArray()`。兩個方法都執行查詢並將結果儲存在 `List<T>` 或陣列中。

這些方法對幾個目的很有用。藉由強制查詢立即執行，這些方法捕捉資料即時的快照。你可以強制查詢立即執行，而非稍後再列舉序列；還有，可以使用 `ToList()` 或 `ToArray()` 來產生不太可能會變化的查詢結果快照，以供後續使用；也可以快取結果並於稍後使用儲存下來的版本。

在絕大部分情況下，惰性求值較積極求值更省事且更有彈性。在需要積極求值的極少情況下，你可以使用 `ToList()` 或 `ToArray()` 執行查詢並儲存結果，但除非使用積極求值有明確的必要性，不然最好還是使用惰性求值。

做法 38　偏好 lambda 表示式而非方法

這個建議可能與直覺相反。以 **lambda** 表示式寫程式可能會在 **lambda** 的內容中產生重複的程式。你會發現你經常在重複一小段邏輯。下面的程式有重複數次的相同邏輯：

```
var allEmployees = FindAllEmployees();

// 找出第一個員工：
var earlyFolks = from e in allEmployees
                    where e.Classification ==
                    EmployeeType.Salary
                    where e.YearsOfService > 20
                    where e.MonthlySalary < 4000
                    select e;

// 找出最菜的人：
var newest = from e in allEmployees
                where e.Classification == EmployeeType.Salary
                where e.YearsOfService < 20
                where e.MonthlySalary < 4000
                select e;
```

你可以使用具有兩個條件的單一 `Where` 句子來取代多個 `Where`。兩者間沒有顯著的差別。由於查詢組合（見做法 **31**）及簡單的 `where` 述詞可能在行內，其效能是相同的。

你或許會想要將重複的 **lambda** 表示式製作成可重複使用的方法。最後程式看起來像是這樣：

```
// 做成方法：
private static bool LowPaidSalaried(Employee e) =>
    e.MonthlySalary < 4000 && e.Classification ==
    EmployeeType.Salary;

// 在其他地方
var allEmployees = FindAllEmployees();
var earlyFolks = from e in allEmployees
                 where LowPaidSalaried(e) &&
                 e.YearsOfService > 20
                 select e;
```

```
// 找出最菜的人：
var newest = from e in allEmployees
             where LowPaidSalaried(e) && e.YearsOfService < 2
             select e;
```

這是個小程式，因此不會改變很多。但感覺好多了。現在，若員工分級改變或下限改變，你只需要修改一處的邏輯。

不幸的是，這種重構程式的方法讓它較難重複使用。第一個版本實際上比第二個版本更能重複使用。這是因為 lambda 表示式求值、解析與最終執行的方式。如果你跟大部分開發者一樣，則你會視複製為罪惡，且極欲除之而後快。單一方法的版本比較簡單。必要時它只有一份程式拷貝需要修改。這是好的軟體工程。

不幸的是它也是錯的。某些程式會將 lambda 表示式轉換成 delegate，以在你的查詢表示式中執行程式；其他類別會從 lambda 表示式建構表示式樹、解析表示式、在其他環境中執行。LINQ to Object 是前者，LINQ to SQL 是後者。

LINQ to Object 在本機資料儲存體執行查詢，通常儲存於泛型集合中。它的實作建構帶有 lambda 表示式中的邏輯，並執行該程式的不具名 delegate。LINQ to Object 擴充方法使用 `IEnumerable<T>` 作為輸入序列。

另一方面，LINQ to SQL 使用查詢中的表示式樹。表示式樹帶有查詢的邏輯表示。LINQ to SQL 解析此樹並使用表示式樹來建構可以直接在資料庫執行的 T-SQL 查詢。然後，查詢字串（T-SQL 格式）被送到資料庫引擎並執行。

這種處理程序需要 LINQ to SQL 引擎解析表示式樹，並將每一個邏輯操作替換成相對應的 SQL。所有方法呼叫被替換成 `Expression.MethodCall` 節點。LINQ to SQL 引擎無法將模糊的方法呼叫轉譯成 SQL 表示式。相對的，它拋出例外。LINQ to SQL 引擎直接失敗而非嘗試執行多個查詢，將資料帶回應用程式邊界的用戶端並在那邊處理。

若你正在建構任何資料來源的可重複使用函式庫，就必須考慮這種狀況，必須將程式設計成可正確操作任何資料來源。這表示必須分離 lambda 表示式並作為行內程式，以讓你的函式庫正確運作。

當然，這並不表示你應該在函式庫中四處複製程式。它只是表示涉及查詢表示式與 lambda 時，你必須為你的應用程式建構不同的基本單元。從我們的簡單範例來看，你可以這樣建構較大的可重複使用區塊：

```
private static IQueryable<Employee> LowPaidSalariedFilter
    (this IQueryable<Employee> sequence) =>
        from s in sequence
        where s.Classification == EmployeeType.Salary &&
        s.MonthlySalary < 4000
        select s;

// 其他地方：
var allEmployees = FindAllEmployees();

// 找出第一個員工：
var salaried = allEmployees.LowPaidSalariedFilter();

var earlyFolks = salaried.Where(e => e.YearsOfService > 20);

// 找出最菜的人：
var newest = salaried.Where(e => e.YearsOfService < 2);
```

當然，並非每個查詢都很容易修改。你必須在呼叫鏈中向上一點點，以找出可重複使用的清單處理邏輯，來讓相同的 lambda 表示式只需表達一次。記得做法 31 的列舉程序方法，在開始遍歷集合中的項目前不會執行。記得這件事，就可以建構組成程序的一部分並帶有常用 lambda 表示式的小方法，這種方法必須以序列作為輸入並使用 `yield return` 關鍵字回傳序列。

依據相同的模式，你能透過建構可從遠端執行的新表示式樹來組建 `IQueryable` 列舉程序。此處用來找尋一組員工的表示式樹可在執行前組合成程序，`IQueryProvider` 物件（例如 LINQ to SQL 資料來源）處理完整的查詢，而非取出必須在本機執行的一部分。

然後將這些小方法組合，以建立在你的應用程式中使用的較大查詢。這種技巧的好處是可以避免這一節第一個範例的複製程式問題。你還寫出了程式，以便在組合完整的查詢並加以執行時建構表示式樹以供執行。

重複使用複雜查詢中 lambda 表示式最有效率的方式之一，是在封閉泛型型別上建構這些查詢的擴充方法。你可以發現尋找低薪員工的方法就是這種

方法。在實際程式中，你應該建構使用 `IEnumerable<Employee>` 作為參數型別的第二個過載。如此可同時支援 LINQ to SQL 方式的實作與 LINQ to Object 實作。

你可以取用 lambda 表示式與序列方法等較小的基本單元，以組合成所需要的查詢。你可以使用 `IEnumerable<T>` 與 `IQueryable<T>` 建立程式來獲得好處。此外，你沒有破壞可查詢表示式樹的可能求值。

做法 39 避免在函式與動作中拋出例外

當你建構對序列值執行的程式且程式會在序列處理過程拋出例外時，狀態的復原會有問題。你不知道有多少元素被處理過，不知道有什麼東西必須還原，你完全無法回復程式的狀態。

以下面的程式為例，它幫每個人加薪 5 趴：

```
var allEmployees = FindAllEmployees();
allEmployees.ForEach(e => e.MonthlySalary *= 1.05M);
```

有一天此程序在執行時拋出例外。這個例外可能不是在第一個或最後一個員工上面。某些員工加薪了，但其他人沒有。你的程式很難復原到之前的狀態。你能讓資料回到一致的狀態嗎？一旦你不知道程式的狀態，就無法在沒有人工檢視所有資料的狀況下回復狀態。

發生這種問題是因為程式在原處修改序列的元素。它並不遵守強式的例外保證。遇到錯誤時，你不知道發生了什麼、什麼沒有發生。

你藉由保證當方法沒有完成時程式的狀態不會改變來解決這個問題。你可以用各種方式來實作，每個方式有好有壞。

在講缺點前，讓我們更深入的檢視考量的原因。並非每個方法都有此問題。許多方法檢視序列但並沒有修改它。下面的方法檢視每個人的薪水並回傳結果：

```
var total = allEmployees.Aggregate(0M,
    (sum, emp) => sum + emp.MonthlySalary);
```

你無需小心的修改這種不修改序列資料的方法。在許多應用程式中，你會發現大部分的方法並不修改序列。讓我們回到第一個方法，幫每個員工加薪 5 趴。要怎樣修改此方法以確保滿足強式例外保證？

第一個且最簡單的方法是修改動作，以確保前面用 **lambda** 表示式表示的動作方法不會拋出例外。在許多狀況下，修改序列的每個元素前是可以先進行條件檢查。你必須定義函式與述詞以讓方法的約定可以滿足所有的狀況，甚至包括錯誤。這種策略適用於元素會引發例外時。前面的例子中，所有例外是因為老舊的員工記錄與還在儲存體中的離職員工記錄所導致的，略過它們是正確的行為。下面的修改可行：

```
allEmployees.FindAll(
    e => e.Classification == EmployeeType.Active).
    ForEach(e => e.MonthlySalary *= 1.05M);
```

以這種方式改正問題是避免演算法一致性問題的最簡單方式。只要你可以將動作方法撰寫成確保 **lambda** 表示式或動作方法不會因例外結束，它就是最有效率的技巧。

但有時候你無法保證表示式不會拋出例外，現在必須採取成本更高的措施，你必須修改演算法以應付可能的例外。這表示要在拷貝上執行所有工作，然後在操作成功完成後替換原始序列。若覺得無法避免可能的例外，你可以修改前面的演算法：

```
var updates = (from e in allEmployees
               select new Employee
               {
                   EmployeeID = e.EmployeeID,
                   Classification = e.Classification,
                   YearsOfService = e.YearsOfService,
                   MonthlySalary = e.MonthlySalary *= 1.05M
               }).ToList();
allEmployees = updates;
```

你可以看到這些修改的成本。首先，程式比前面的版本要長，這是更多的工作 - 更多的程式要維護與理解，但你也改變了應用程式的效能。新版本建構出所有員工記錄的拷貝，然後交換新舊清單的參考。若員工記錄很大，這可能會導致效能瓶頸。在交換參考前你複製了所有員工記錄。現在，此動作的

約定可能在 Employee 物件無效時拋出例外。查詢外部的程式必須處理這個狀況。

但這種修改還有另一個問題：它是否合理端看如何使用。這個版本限制你以它組合多個函式的操作。這個程式快取完整的清單，這表示它的修改不能在一次列舉中與其他轉換組合，每個轉換變成命令式操作。實務上，你可以建構執行所有轉換的查詢陳述來解決這個問題。你將快取列表並交換整個序列作為整個轉換的最後一個步驟，這種技巧可以保持組合能力並提供強式例外保證。

實務上，這表示撰寫查詢表示式以回傳新的序列，而非原處修改序列的元素。每個組合查詢應該要能夠交換清單，除非在處理序列的過程中產生了例外。

組合查詢改變了你撰寫例外安全程式的方式。若你的動作或函式拋出例外，你沒有辦法保證資料處於一致的狀態。你不知道有多少元素被處理過、有什麼東西必須還原，完全無法回復程式的狀態。但回傳新元素（而非在原處修改元素）給你更好的機會，以確保操作完成或不會動到程式的狀態。

此建議適用於所有會拋出例外的可變方法，它也適用於多執行緒的環境，這個問題在使用 lambda 表示式且其中的程式可能會拋出例外時更難發現。作為最後一個操作，你應該在確定沒有操作產生例外後再交換整個序列。

做法 40 區分提前與延遲執行

宣告式程式是說明性的：它定義要做什麼。命令式程式逐步列出執行指令。兩者都可以有效的建構程式。但混合兩者會導致應用程式不可預期的行為。

你現在執行的命令式程式會計算所需的參數然後呼叫方法。下面這一行程式描述產生答案的一組命令式步驟：

```
var answer = DoStuff(Method1(),
    Method2(),
    Method3());
```

在執行期，它會執行下列動作：

1. 呼叫 Method1 以產生 DoStuff() 的第一個參數。
2. 呼叫 Method2 以產生 DoStuff() 的第二個參數。
3. 呼叫 Method3 以產生 DoStuff() 的第三個參數。
4. 以三個計算所求出的參數呼叫 DoStuff。

你應該很熟悉這種程式。所有參數都是計算出來的，而資料會傳給任一方法。你寫出的演算法是一組產生結果的描述性步驟。

使用 **lambda** 與查詢表示式的延遲執行徹底改變這種程序。下面的程式似乎與前一個範例執行相同的工作，但你很快就會發現有重要的不同：

```
var answer = DoStuff(() => Method1(),
    () => Method2(),
    () => Method3());
```

在執行期，它會執行下列動作：

1. 呼叫 DoStuff()，傳入呼叫 Method1、Method2 與 Method3 的 **lambda** 表示式。
2. 在 DoStuff 中，若且唯若需要 Method1 的結果則呼叫 Method1。
3. 在 DoStuff 中，若且唯若需要 Method2 的結果則呼叫 Method2。
4. 在 DoStuff 中，若且唯若需要 Method3 的結果則呼叫 Method3。
5. Method1、Method2、與 Method3 會以任意的順序呼叫任意次數（包括零次）。

除非需要它的結果，否則這些方法不會被呼叫。這種差別非常大，如果你混合這兩種做法會產生很大的問題。

從外部看，任何方法可用它的回傳值取代，反之亦然，只要方法沒有產生任何副作用。在我們的例子中，DoStuff() 方法在兩種策略間沒有什麼不同。回傳的值相同，兩個策略都正確。若方法對相同的輸入回傳相同的值，則方法回傳值可以由方法呼叫取代，反之亦然。

但將程式視為一個整體來看，這兩行程式之間有重大的差別。命令式模型總是呼叫三個方法，這些方法的副作用總是發生一次而已。相較之下，宣告式模型可能會也可能不會呼叫這些方法。宣告式版本可能會執行任何一個方法

一次以上。兩者間的差別是（1）呼叫方法並傳遞結果給一個方法，與（2）將 delegate 傳給方法並讓方法呼叫 delegate。每次執行此應用程式可能會得到不同的結果，視這些方法執行了什麼動作而定。

加入 lambda 表示式、型別推論與列舉程序，使你的類別能更易使用函式性程式設計概念。你可以建構以函式作為參數，或回傳函式給呼叫方的高階函式。某種程度上這不是很大的改變：純函式與其回傳值可以交換。實務上，函式可能會有副作用，這表示可能會套用不同的規則。

若資料與方法可交換，你要如何選擇？更重要的是，什麼時候選擇哪一個？最重要的差別是資料在使用之前必須求值，而方法可以惰性求值。當你必須提早對資料求值時，你必須先對方法求值並使用其結果作為資料，而非採取函式性方式並替換方法。

決定使用何者，最重要的標準是函式本身與回傳值的可變性中產生副作用的可能性。做法 37（這一章）顯示了一個根據目前時間產生結果的查詢，它回傳值的改變，是根據執行它並快取結果或將此查詢作為函式參數。若函式本身有副作用，程式的行為會依據何時執行此函式而有所不同。

你可以使用技巧來減少提早與延後求值間的差異。純不可變型別不能改變且不改變其他程式狀態；因此它們不會有副作用。在前面的範例中，若 `Method1`、`Method2`、`Method3` 是不可變型別的成員，則提早與延後求值陳述的可觀測行為應該完全一樣。

我的範例沒有任何參數，但若任何延後求值方法有參數，這些參數必須不可變，以確保提早與延後綁定的結果相同。

因此，決定提早或延後求值，最重要一點是你想要達成的語意。若（且唯若）物件與方法不可變，則程式的正確性與以函式取代值相同，反之亦然（此處的“不可變方法”，表示方法不會修改任何全域狀態，像是執行 I/O 操作、修改全域變數或與其他程序溝通）。若物件與方法非不可變，改變提早或延後求值可能會改變程式的行為。這一節之後的內容假設可觀測的行為在提早或延後求值間不會改變。我們會檢視偏好其中一個策略的其他原因。

其中一個判斷是輸入與輸出空間的大小與計算輸出的成本。舉例來說，有個程式以 `Math.PI` 計算 pi 值。從外部看，此值與計算可交換，但計算 pi 需要時間。另一方面，`CalculatePrimeFactors(int)` 方法可用帶有所有整數元素的查詢表取代，此時記憶體中資料表的成本比在有需要時計算值的時間高很多。

實際的問題通常落在這兩種極端的中間。正確的答案不會很明顯，也不會很清楚。

除了分析計算成本與儲存成本外，你還必須考慮方法要如何使用其結果。你會發現在某些情況下，對某些查詢提早求值比較合理。其他情況下，你不常使用臨時的結果。若你確定程式不產生副作用且提早或延遲求值產生正確的檔案，則你可以根據兩種方案的效能做決定。你可以嘗試兩種方式，評估差別，使用比較好的結果。

最後，在某種情況下，你可能會發現混合兩種策略最好。你會發現有時快取的效率最高。在這種情況下，可以建構回傳快取值的 delegate：

```
var cache = Method1();
var answer = DoStuff(() => cache,
    () => Method2(),
    () => Method3());
```

最終決定是根據方法是否在遠端資料儲存體執行。這個因素對 LINQ to SQL 如何處理查詢有很大的影響。每個 LINQ to SQL 查詢以延遲查詢開始：以方法而非資料作為參數。有些方法涉及可在資料庫引擎中完成的工作，有些工作表示必須在部分查詢傳送給資料庫引擎之前處理的本地方法。LINQ to SQL 解析表示式樹。在提交查詢給資料庫引擎之前，它以方法呼叫的結果替換任何本地方法呼叫。只有在方法呼叫不依靠任何被處理的輸入序列中，個別的項目它才可以進行這個處理（見做法 37 與 38）。

一旦 LINQ to SQL 以相等的回傳值替換本地方法呼叫，它將查詢從表示式轉譯成 SQL 陳述，發送給資料庫引擎並在它上面執行。結果是，通過將查詢建構成一組表示式或程式，LINQ to SQL 函式庫可以用等效的 SQL 替換這些方法。如此可改善效能並降低頻寬使用。它也表示 C# 開發者可以花較少的時間學習 T-SQL。其他的提供者也可以這麼做。

但這種做法的前提是在正確的狀況下，你可以將資料當做程式處理，反之亦然。使用 LINQ to SQL 時，本地方法可用回傳值取代，若方法的參數是不依靠輸入序列的常數。還有，LINQ to SQL 函式庫中，相當多的功能將表示式樹轉譯成之後可轉譯成 T-SQL 的邏輯結構。

你現在從 C# 建構演算法，你可以判斷是否使用資料作為參數或使用函式作為參數導致行為不相同。一旦你判斷出兩者都正確，你必須判斷哪一個是比較好的策略。輸入空間較小時，傳遞資料可能比較好。但在其他狀況下，當輸入或輸出空間非常大，且你不一定要使用整個輸入資料空間時，你會發現使用演算法本身作為參數比較好。若不能確定，則傾向使用演算法作為參數，因為實作函式的開發者，可將函式建構為積極對輸出空間求值並操作這些資料。

做法 41 避免捕捉昂貴的資源

閉包建構帶有限界變數的物件。限界變數的生命期可能會讓你意外，且不一定是好的意外。開發者習慣以非常簡單的方式看待區域變數的生命期：變數於宣告時進入範圍，於相對應的區塊結束時離開範圍。區域變數在離開範圍後可被垃圾回收。我們以這種假設來管理資源與物件生命期。

閉包與被捕捉的變數改變了以上的規則。當你在閉包捕捉一個變數，該變數參考物件的生命期會擴展，直到最後一個捕捉變數的 delegate 參考變成垃圾它才會變成垃圾。在某些情況下它還會撐更久。在閉包與被捕捉的變數逸出方法後，它們可以被用戶程式的閉包與 delegate 存取。這些 delegate 與閉包可被其他程式存取，如此類推。最終存取你的 delegate 的程式變成沒完沒了的方法集，不知道你的閉包與 delegate 何時才不再可觸及。這表示若你回傳使用被捕捉變數的 delegate 表示的東西時，你無法得知區域變數離開範圍。

好消息是通常你無需考慮這種行為。managed 型別且未持有昂貴資源的區域變數之後會被垃圾回收，如同一般變數。若區域變數只有用到記憶體則完全不需擔心。

但有些變數持有昂貴的資源。它們的型別實作 `IDisposable` 並需要明確的清理。你可以在實際列舉集合之前提早清理這些資源。你可能會發現檔案或連線關閉得不夠快，且檔案還是開啟中，所以你無法存取檔案。

做法 44 顯示 C# 編譯器如何產生 delegate 與變數如何被閉包捕捉。這一節會檢視如何辨識你已經捕捉到帶有其他資源的變數。我們會檢視如何管理這些資源，以及如何避免捕捉生命期比你所需更長的變數。

以下面的程式為例：

```
var counter = 0;
var numbers = Extensions.Generate(30, () => counter++);
```

它產生類似這樣的程式：

```
private class Closure
{
    public int generatedCounter;
    public int generatorFunc() =>
        generatedCounter++;
}

// 使用
var c = new Closure();
c.generatedCounter = 0;
var sequence = Extensions.Generate(30, new Func<int>(
    c.generatorFunc));
```

這很有趣。隱藏的套疊類別成員限制於 Extensions.Generate 使用的 delegate，這會影響隱藏物件的生命期，並因此影響可被垃圾回收的成員。以下面的程式為例：

```
public IEnumerable<int> MakeSequence()
{
    var counter = 0;
    var numbers = Extensions.Generate(30, () => counter++);
    return numbers;
}
```

此程式回傳的物件使用閉包限制的 delegate。由於回傳值需要 delegate，此 delegate 的生命期延長超出方法的作用期間，此物件的生命期代表變數的限界延長。物件可存取是因為 delegate 實例可存取，而 delegate 還可以存取是因為它是回傳物件的一部分。而物件的所有成員可以存取，是因為物件可存取。

C# 編譯器產生類似下面的程式：

```
public static IEnumerable<int> MakeSequence()
{
    var c = new Closure();
    c.generatedCounter = 0;
    var sequence = Extensions.Generate(30,
        new Func<int>(c.generatorFunc));
    return sequence;
}
```

請注意，此序列帶有限界在初始化該閉包的區域物件 c 中的 delegate 參考。區域變數 c 的生命期超過方法的結束。

通常，這種情況不會有太大的問題，但有兩種狀況會引發問題。首先與 IDisposable 有關。以下面的程式為例，它從 CSV 輸入串流讀取數字，並以數字序列的序列回傳值。每個內層序列帶有該行的數字。它使用做法 27 所示的一些擴充方法。

```
public static IEnumerable<string> ReadLines(
    this TextReader reader)
{
    var txt = reader.ReadLine();
    while (txt != null)
    {
        yield return txt;
        txt = reader.ReadLine();
    }
}

public static int DefaultParse(this string input,
    int defaultValue)
{
    int answer;
    return (int.TryParse(input, out answer))
        ? answer : defaultValue;
}

public static IEnumerable<IEnumerable<int>>
    ReadNumbersFromStream(TextReader t)
{
    var allLines = from line in t.ReadLines()
```

```
                    select line.Split( ',' );
    var matrixOfValues = from line in allLines
                         select from item in line
                                select item.DefaultParse(0);
    return matrixOfValues;
}
```

你這樣使用它：

```
var t = new StreamReader(File.OpenRead( "TestFile.txt" ));
var rowsOfNumbers = ReadNumbersFromStream(t);
```

請記得，查詢只於要存取下一個值時才會產生該值。ReadNumbersFromStream() 方法並沒有將所有資料取出放在記憶體中，而是在有需要時從串流載入值。之後的兩個陳述並未讀取檔案，只會在你開始列舉 rowsOfNumbers 中的值才開啟檔案並開始讀取值。

之後的程式審查中，有人指責你沒有明確的關閉測試檔案，或許他是因為有資源洩漏或嘗試開啟檔案時檔案已經開啟而發現。你改正了這個問題。不幸的是，根源沒有處理到。

```
IEnumerable<IEnumerable<int>> rowOfNumbers;
using (TextReader t = new
    StreamReader(File.OpenRead( "TestFile.txt" )))
    rowOfNumbers = ReadNumbersFromStream(t);
```

你開始進行測試，預期會通過，但你的程式拋出例外與幾行程式：

```
IEnumerable<IEnumerable<int>> rowOfNumbers;
using (TextReader t = new StreamReader(File.OpenRead(
     "TestFile.txt" )))
    rowOfNumbers = ReadNumbersFromStream(t);

foreach (var line in rowOfNumbers)
{
    foreach (int num in line)
        Write( "{0}, ", num);
    WriteLine();
}
```

發生什麼事？你嘗試在關閉檔案後讀取它。迭代拋出 ObjectDisposedException；C# 編譯器將 TextReader 綁在讀取與解析檔案的 delegate 中；這一組程式以 arrayOfNums 變數表示；還沒有真正發生什麼事；串流還沒有讀取，沒有東西被解析。這是在你將資料管理交給呼叫方時會出現的問題之一。若呼叫方誤解資源的生命期，它們會引發資源洩漏與有問題的程式。

改正方式很直接。移動程式，以在關閉檔案前使用數字陣列：

```
using (TextReader t = new
    StreamReader(File.OpenRead( “TestFile.txt” )))
{
    var arrayOfNums = ReadNumbersFromStream(t);

    foreach (var line in arrayOfNums)
    {
        foreach (var num in line)
            Write( “{0}, “, num);
        WriteLine();
    }
}
```

很好，但你的問題並非都這麼簡單。這種策略會產生很多重複的程式，而我們一直都在避免這種情況。因此讓我們看看如何做出更好的答案。前面的程式可行是因為它在檔案關閉前使用數字陣列。

這種寫程式的方式幾乎找不到正確的位置來關閉檔案。你的 API 必須開啟檔案，但必須在後面某一點才能關閉檔案。假設原始的使用模式像這樣：

```
using (TextReader t = new
    StreamReader(File.OpenRead( “TestFile.txt” )))
    return ReadNumbersFromFile(t);
```

現在你找不出關閉檔案的辦法。它在一個程序中開啟，但在呼叫鏈中的上層得關閉檔案。哪裡？你無法確定，因為那不是你的程式。它在呼叫鏈的上層，非你能控制，你甚至不知道檔案名稱，且沒有串流的 handle 可供檢視要關閉什麼。

有個明顯的解決方案是建構一個開啟檔案的方法、讀取序列、回傳序列。以下是可能的實作：

```
public static IEnumerable<string> ParseFile(string path)
{
    using (var r = new StreamReader(File.OpenRead(path)))
    {
        var line = r.ReadLine();
        while (line != null)
        {
            yield return line;
            line = r.ReadLine();
        }
    }
}
```

此方法使用做法 31 所示的延遲執行。重點是 StreamReader 物件只會在所有元素被讀取後才會被處置，無論發生的早晚。檔案物件會被關閉，但只在序列被列舉後。有個較小的例子可以表達我的意思：

```
class Generator : IDisposable
{
    private int count;
    public int GetNextNumber() => count++;

    public void Dispose()
    {
        WriteLine( “Disposing now  “);
    }
}
```

Generator 類別實作了 IDisposable，但只會在你捕捉實作 IDisposable 型別的變數時才會顯示發生了什麼事。以下是使用例子：

```
var query = (from n in SomeFunction()
             select n).Take(5);


foreach (var s in query)

    Console.WriteLine(s);




WriteLine( “Again” );
```

```
foreach (var s in query)
    WriteLine(s);
```

以下是此程式的輸出：

```
0
1
2
3
4
Disposing now
Again
0
1
2
3
4
Disposing now
```

Generator 在你希望時被處置：第一次迭代完成後。Generator 在完成序列的迭代或如同這個查詢提前停止迭代時被處置。

然而這有個問題。請注意！“Disposing now”被輸出了兩次。由於程式迭代序列兩次，導致 Generator 被處置兩次。這在 Generator 類別中不是問題，因為它只是做記號，但檔案範例在第二次列舉序列時拋出例外。第一次列舉完成，而 StreamReader 受到處置，然後第二次列舉嘗試存取已經被處置的串流讀取程序。它無法進行。

若你的應用程式可能會對可處置的資源進行多次列舉，你必須找出其他解決方案。你可能會發現你的應用程式讀取多個值，在演算法中以不同的方式進行處理。使用 **delegate** 來傳遞一或多個演算法給讀取，並處理檔案中記錄的程序會比較好。

你需要這個方法的泛型版本好讓你捕捉這些值的使用，然後在最終處置檔案前在表示式中使用這些值。同樣的動作類似如下：

```
// 使用模式：參數是檔案
// 以及對檔案中的每一行要採取的動作
ProcessFile(“testFile.txt”,
    (arrayOfNums) =>
```

```
    {
        foreach (var line in arrayOfNums)
        {
            foreach (int num in line)
                Write( "{0},  ", num);
            WriteLine();
        }
        // 回傳一些東西以滿足編譯器
        return 0;
    }
);

// 宣告 delegate 型別
public delegate TResult ProcessElementsFromFile<TResult>(
    IEnumerable<IEnumerable<int>> values);

// 讀取檔案、使用 delegate 處理
// 每一行的方法
public static TResult ProcessFile<TResult>(string filePath,
    ProcessElementsFromFile<TResult> action)
{
    using (TextReader t = new StreamReader(File.Open(filePath)))
    {
        var allLines = from line in t.ReadLines()
                       select line.Split( ',' );

        var matrixOfValues = from line in allLines
                             select from item in line
                                    select item.
                                    DefaultParse(0);
        return action(matrixOfValues);
    }
}
```

這看起來有點複雜，但若你以多種方式使用資料時會很有幫助。假設你需要檔案中的最大值：

```
var maximum = ProcessFile( "testFile.txt" ,
    (arrayOfNums) =>
        (from line in arrayOfNums
         select line.Max()).Max());
```

此時檔案串流的使用是完全封裝在 ProcessFile 中。你要的答案是個值，它會從 lambda 表示式回傳。透過改變程式以讓昂貴的資源（此例中的檔案串流）在函式中分配與釋放，你沒有讓昂貴的成員被加入你的閉包中。

在閉包中捕捉昂貴資源的另一個問題比較輕微，但會影響你的應用程式的效能。以下列方法為例：

```
IEnumerable<int> ExpensiveSequence()
{
    int counter = 0;
    var numbers = Extensions.Generate(30,
        () => counter++);

    Console.WriteLine(“counter: {0}”, counter);

    var hog = new ResourceHog();
    numbers = numbers.Union(
        hog.SequenceGeneratedFromResourceHog(
        (val) => val < counter));
    return numbers;
}
```

如同上面其他閉包，此演算法產生之後執行的程式，使用延遲執行模型。這表示 ResourceHog 超越此方法的結束直到用戶程式列舉該序列。此外，若 ResourceHog 不可處置，它會存活直至完全不能存取並被垃圾回收釋放為止。

若它成為瓶頸，你可以重新安排查詢，以讓 ResourceHog 產生的數字被積極求值，使得 ResourceHog 可以立即被清理：

```
IEnumerable<int> ExpensiveSequence()
{
    var counter = 0;
    var numbers = Extensions.Generate(30,
        () => counter++);

    WriteLine(“counter: {0}”, counter);

    var hog = new ResourceHog();
    var mergeSequence = hog.SequenceGeneratedFromResourceHog(
        (val) => val < counter).ToList();
```

```
    numbers = numbers.Union(mergeSequence);
    return numbers;
}
```

這個例子相當清楚，因為程式沒有很複雜。若你的演算法更為複雜，分離不昂貴的資源與昂貴的資源會比較困難。視建構閉包方法的演算法的複雜程度，放鬆被閉包綁定變數中捕捉到的不同資源可能會比較困難。下面的方法使用被閉包捕捉的三個不同區域變數：

```
private static IEnumerable<int> LeakingClosure(int mod)
{
    var filter = new ResourceHogFilter();
    var source = new CheapNumberGenerator();
    var results = new CheapNumberGenerator();

    var importantStatistic = (from num in
                              source.GetNumbers(50)
                              where filter.PassesFilter(num)
                              select num).Average();

    return from num in results.GetNumbers(100)
           where num > importantStatistic
           select num;
}
```

乍看之下沒問題。`ResourceHog` 產生重要的統計數據。它的範圍在方法內，且在方法結束時很快的就變成垃圾。

不幸的是，此方法不像看起來的那麼好。

以下是原因。C# 編譯器對每個閉包實作範圍建構一個套疊的類別。最後的查詢陳述（回傳大於統計數據的數字）需要綁定帶有統計數據的變數閉包。在這個方法的前面必須使用過濾器來建構統計數據。這表示過濾器會被複製到實作閉包的套疊類別中。`return` 陳述回傳使用套疊類別實作 `where` 句子實例的型別。實作閉包套疊類別的實例從方法中洩漏，通常你不會在意。但若 `ResourceHogFilter` 使用了高度耗費的資源，它會耗盡你的應用程式。

為修改這個問題，你必須將方法分割成兩個部分，並讓編譯器建構兩個閉包類別：

```
private static IEnumerable<int> NotLeakingClosure(int mod)
{
    var importantStatistic = GenerateImportantStatistic();

    var results = new CheapNumberGenerator();
    return from num in results.GetNumbers(100)
           where num > importantStatistic
           select num;
}

private static double GenerateImportantStatistic()
{
    var filter = new ResourceHogFilter();
    var source = new CheapNumberGenerator();

    return (from num in source.GetNumbers(50)
            where filter.PassesFilter(num)
            select num).Average();
}
```

你說：“等一下，GenerateImportantStatistic 中的 return 陳述帶有產生統計的查詢，這個閉包還是會洩漏”。不，並沒有。Average 方法要求整個序列（見做法 40）。列舉發生在 GenerateImportantStatistic 範圍內並回傳平均值。帶有 ResourceHogFilter 物件的閉包可以在方法回傳時被垃圾回收。

我選擇以這種方式重寫此方法，是因為將方法寫成具有多個邏輯閉包時會產生更多的問題。雖然你認為編譯器應該會建構多個閉包，它基本上只產生一個閉包來處理方法中所有的 lambda。你在乎表示式之一會從方法回傳且認為其他表示式不重要，但其實很重要。由於編譯器建構一個類別來處理單一範圍所建構的所有閉包，任何閉包中使用的所有成員會插入到該類別中。以下面的方法為例：

```
public IEnumerable<int> MakeAnotherSequence()
{
    var counter = 0;

    var interim = Extensions.Generate(30,
        () => counter++);
    var gen = new Random();
```

```
    var numbers = from n in interim
                  select gen.Next() - n;
    return numbers;
}
```

MakeAnotherSequence() 帶有兩個查詢。第一個查詢產生從 0 到 29 的整數序列。第二個查詢使用亂數產生器修改該序列。C# 編譯器產生一個私用類別來實作帶有 counter 與 gen 的閉包。呼叫 MakeAnotherSequence() 的程式會存取帶有這兩個區域變數類別實例。編譯器不會建構兩個套疊的類別，只有建構一個。此套疊類別的實例會傳給呼叫方。

還有最後一個與閉包中的操作有關的問題。例如：

```
private static void SomeMethod(ref int i)
{
    //...
}
private static void DoSomethingInBackground()
{
    var i = 0;
    var thread = new Thread(delegate ()
    { SomeMethod(ref i); });
    thread.Start();
}
```

此例中，你捕捉一個變數並在兩個執行緒中檢視。此外，你將程式寫成兩個執行緒，透過參考存取它。我打算解釋執行此範例時 i 的值會發生什麼事，但事實上是不可能得知會發生什麼事。兩個執行緒可讀取或修改 i 的值，但根據何者較快，執行緒可隨時改變其值。

當你在你的演算法中使用查詢表示式時，編譯器會為整個方法的所有表示式產生單一閉包。該型別的物件會從你的方法回傳，或許是實作列舉的型別成員。此物件會在系統中存活直到所有使用者被輸出為止。這會產生許多問題。若有任何欄位被複製到實作 IDisposable 的閉包，它會導致正確性的問題。若有欄位的處理是昂貴的，它會導致效能問題。無論是哪一種方式，你必須理解閉包建構的物件何時會被方法回傳。你必須確定你需要這些變數，或在無法確定時要確保閉包能夠清理它們。

做法 42 區分 IEnumerable 與 IQueryable 資料來源

IQueryable<T> 與 IEnumerable<T> 有非常相似的 API 格式。IQueryable<T> 繼承自 IEnumerable<T>。你或許會認為這兩個界面可以交換。許多情況下是可以，且是這樣設計的。相對的，序列就是序列，但序列不一定可以交換。它們的行為不同，且效能可以差別很大。下面兩個查詢陳述相當的不同：

```
var q =
        from c in dbContext.Customers
        where c.City == "London"
        select c;
var finalAnswer = from c in q
                  orderby c.Name
                  select c;
// 省略迭代最終答案序列的程式

var q =
    (from c in dbContext.Customers
     where c.City == "London"
     select c).AsEnumerable();
var finalAnswer = from c in q
                  orderby c.Name
                  select c;

// 省略迭代最終答案的程式
```

這些查詢回傳相同的結果，但以非常不同的方式執行工作。第一個查詢使用基於 IQuieryable 功能的普通 LINQ to SQL 版本；第二個版本強制資料庫物件轉換成 IEnumerable 序列，並在本機執行大部分的工作。這是惰性求值與 LINQ to SQL 的 IQueryable<T> 支援的組合。

執行查詢結果時，LINQ to SQL 函式庫從所有的查詢陳述產生結果。此例中，這表示有對資料庫做一個呼叫。這也表示有一個 SQL 查詢執行了 where 句子與 orderby 句子。

在第二個例子中，以 IEnumerable<T> 序列回傳第一個查詢，表示後續的操作使用 LINQ to Object 實作並以 delegate 執行。第一個陳述引發對資料庫的

呼叫以取得來自 London 的客戶，第二個陳述將第一個陳述的回傳依名稱排序。此排序在本機執行。

你應該注意其中的差別，因為許多查詢在使用 IQueryable 功能時比使用 IEnumerable 功能更有效率。此外，由於 IQueryable 與 IEnumerable 處理查詢表示式的方式差異，你會發現在某個環境可用的查詢在其他環境無法使用。

處理的方式在每個步驟都不同。這是因為使用的型別不同。只要出現 lambda 表示式及函式參數，Enumerable<T> 擴充方法會對它們使用 delegate。另一方面，Queryable<T> 使用表示式樹來處理這些相同的函式元素。表示式樹是保存查詢中組成動作邏輯的資料結構。Enumerable<T> 版本必須在本機執行。lambda 表示式被編譯成方法，它們必須在本機執行。這表示你必須將資料從儲存位置拉回本機的應用程式空間。你會轉換更多的資料並拋棄不必要的資料。

相對的，Queryable 版本解析表示式樹。檢視表示式樹後，此版本將邏輯轉換成適合提供者的格式並在最接近資料位置的地方執行該邏輯。結果是資料轉換較少與整體效能更佳。但使用 IQueryable 界面的查詢表示式程式有些限制，且要依靠序列的 Queryable<T> 的實作。

如同做法 37 所示，IQueryable 提供者並不解析模糊方法。它的邏輯有太多的可能。相對的，它們能理解一組實作於 .NET Framework 的運算子與可能被定義的方法。若你的查詢帶有其他方法呼叫，你可能需要強制查詢使用 Enumerable 實作。

```
private bool isValidProduct(Product p) =>
    p.ProductName.LastIndexOf( 'C' ) == 0;

// 可行
var q1 =
    from p in dbContext.Products.AsEnumerable()
    where isValidProduct(p)
    select p;
// 列舉集合時會拋出例外
var q2 =
    from p in dbContext.Products
    where isValidProduct(p)
    select p;
```

第一個查詢可行，因為 LINQ to Objects 使用 delegate 來實作查詢與方法呼叫。`AsEnumerable()` 呼叫強制查詢進入本機用戶空間，而 `where` 句子的執行使用 LINQ to Objects。第二個查詢拋出例外，原因是 LINQ to SQL 使用 `IQueryable<T>` 實作。LINQ to SQL 帶有將查詢轉譯成 T-SQL 的 `IQueryProvider`。然後 T-SQL 被送到資料庫引擎，資料庫引擎在此背景下執行此 SQL 陳述（見做法 38）。這種方式有好處，因為跨越層次傳輸的資料較少。

在傳統的效能與穩健的取捨中，你能明確的將查詢結果轉譯成 `IEnumerable<T>` 來避免例外。這種方案的缺點是 LINQ to SQL 引擎會從資料庫回傳完整的 `dbContext.Products`。此外，其餘的查詢是在本機執行。由於 `IQueryable<T>` 繼承自 `IEnumerable<T>`，這個方法可以在這兩種來源呼叫。

這聽起來不錯，且是個簡單的方式。但它強迫使用你的方法的程式回到 `IEnumerable<T>` 序列。若開發者使用支援 `IQueryable<T>` 的來源，你必須強迫他將所有來源元素拉回此程序的位址空間，然後在此處理所有元素，最終並回傳結果。

就算你通常會正確的撰寫此程式一次，並寫成通用的類別或界面，但對 `IEnumerable<T>` 與 `IQueryable<T>` 來說不是這樣。就算它們有幾乎相同的內部功能，但它們實作方式的差異表示你應該使用符合你資料來源的實作。實務上，你會知道資料來源是否實作 `IQueryable<T>` 或只有 `IEnumerable<T>`。你的來源實作 `IQueryable` 時，你應該確保你的程式使用該型別。

但你可能偶爾發現有個類別支援同一個 `T` 的 `IEnumerable<T>` 與 `IQueryable<T>`：

```
public static IEnumerable<Product>
    ValidProducts(this IEnumerable<Product> products) =>
    from p in products
    where p.ProductName.LastIndexOf( 'C' ) == 0
    select p;

// OK，因為 LINQ to SQL 提供者
// 支援 string.LastIndexOf()
public static IQueryable<Product>
    ValidProducts(this IQueryable<Product> products) =>
    from p in products
    where p.ProductName.LastIndexOf( 'C' ) == 0
    select p;
```

當然，此程式有重複的問題。你可以使用 AsQueryable() 將 IEnumerable<T> 轉換成 IQueryable<T>，以避免重複：

```
public static IEnumerable<Product>
    ValidProducts(this IEnumerable<Product> products) =>
    from p in products.AsQueryable()
    where p.ProductName.LastIndexOf( 'C' ) == 0
    select p;
```

AsQueryable() 會檢視序列的執行期型別。若序列是 IQueryable，它以 IQueryable 回傳序列；相對的，若序列的執行期型別為 IEnumerable，則 AsQueryable() 使用 LINQ to Objects 建構實作 IQueryable 的包裝程序，並回傳此包裝程序。你收到 Enumerable 實作，但它被包裝在 IQueryable 的參考中。

使用 AsQueryable() 讓你取得最大效益。已經實作 IQueryable 的序列會使用該實作，而只支援 IEnumerable 的序列還是能運作。用戶程式將 IQueryable 序列傳給你時，你的程式會正確的使用 Queryable<T> 方法並支援表示式樹與外部執行。若你操作的是只支援 IEnumerable<T> 的序列，則執行期實作會使用該 IEnumerable 實作。

注意這個版本還是使用一個方法呼叫：string.LastIndexOf()。這是 LINQ to SQL 函式庫正確解析的方法之一，因此你可以在 LINQ to SQL 查詢中使用它。但每個提供者有獨特的功能，因此你不應該認為此方法在每個 IQueryProvider 都有實作。

IQueryable<T> 與 IEnumerable<T> 看起來像是提供相同的功能，差別在如何實作查詢模式。要確保查詢結果的宣告使用符合資料來源的型別。查詢方法是靜態綁定的，宣告正確的查詢變數型別意味著你會獲得正確的行為。

做法 42　使用 Single() 與 First() 以強制查詢的語意結果

快速掃過 LINQ 函式庫會讓你認為它們只為序列設計。但有些方法會回傳單一元素，這種方法的行為與其他方法不同，而這些差別能幫助你表示你對查詢回傳單值結果的意圖與預期。

Single() 只回傳一個元素。若元素不存在，或有多個元素，則 Single() 拋出例外。這是相當強烈的意圖陳述。但若你的假設被證明是錯的，你或許想要立即發現。撰寫應該只回傳一個元素的查詢時，你應該使用 Single()。此方法最清楚的表達你的假設：你預期查詢只回傳一個元素。是的，若你的假設錯誤則它會失敗，它以很快且不會導致資料損毀的方式失敗。立即失敗幫助你做出快速的診斷並改正錯誤。此外，你的應用程式資料不會因為使用錯誤資料進行處理而損毀。查詢立即失敗，因為假設是錯的。

```
var somePeople = new List<Person>{
    new Person { FirstName = “Bill”, LastName = “Gates” },
    new Person { FirstName = “Bill”, LastName = “Wagner” },
    new Person { FirstName = “Bill”, LastName = “Johnson” }};

// 拋出例外是因為序列
// 有一個以上的元素
var answer = (from p in somePeople
              where p.FirstName == “Bill”
              select p).Single();
```

此外，不像我列出的其他查詢，這一個查詢在還沒開始檢視結果時就拋出例外。Single() 立即對查詢求值並回傳單一元素。下面的查詢會以相同的例外失敗（但訊息不同）：

```
var answer = (from p in somePeople
              where p.FirstName == “Larry”
              select p).Single();
```

同樣的，你的程式假設只有一個結果。假設錯誤時，Single() 總是拋出 InvalidOperationException。

若你的查詢會回傳零或一個元素，你可以使用 SingleOrDefault()。但要記得回傳一個以上的值時 SingleOrDefault() 還是會拋出例外。你還是預期你的查詢表示式不會回傳一個以上的值。

```
var answer = (from p in somePeople
              where p.FirstName == “Larry”
              select p).SingleOrDefault();
```

此查詢回傳 null（參考型別的預設值）以表示沒有值符合查詢。

當然，有時候你預期會收到一個以上的值，但你只想要一個值。最好的選擇是 `First()` 或 `FirstOrDefault()`。兩個方法都回傳序列的第一個元素。若序列是空的則回傳預設值。下面的查詢找出得分最高的前鋒，若前鋒都沒有得分則回傳 `null`：

```
// 可行。回傳 null
var answer = (from p in Forwards
              where p.GoalsScored > 0
              orderby p.GoalsScored
              select p).FirstOrDefault();
// 若序列無值則拋出例外
var answer2 = (from p in Forwards
               where p.GoalsScored > 0
               orderby p.GoalsScored
               select p).First();
```

當然，有時候你不想要第一個值。有幾個辦法可以解決這個問題。你可以將元素重新排序以取得對的第一個元素（你可以用其他排序並取得最後一個元素，但這樣會花比較多的時間）。

若你確定要找序列中的什麼位置，你可以使用 `Skip` 與 `First` 取得該元素。下面的查詢找出得分第三高的前鋒：

```
var answer = (from p in Forwards
              where p.GoalsScored > 0
              orderby p.GoalsScored
              select p).Skip(2).First();
```

我選擇 `First()` 而非 `Take()`，以強調只要一個元素而不是只有一個元素的序列。請注意，我使用 `First()` 而非 `FirstOrDefault()`，編譯器會假設至少有三個前鋒有得分。

但你開始查詢特定位置的元素時，此查詢可能有更好的寫法。是否應該檢查不同的屬性？是否應該檢視你的序列有沒有支援 `IList<T>` 並支援索引操作？是否應該重寫演算法以找出單一元素？你會發現其他方式會產生比較清楚的程式。

許多查詢用來回傳單值。只要是單值查詢，最好將查詢寫成回傳單值而非只有一個元素的序列。使用 `Single()` 表示你預期只有一個項目，

SingleOrDefault() 表示零或一個項目，First 與 Last 表示你要從序列取出一個項目。使用其他方法找出一個項目可能表示你沒有將查詢寫成應有的樣子。這對使用或維護你的程式的開發者不夠清楚。

做法 44 避免修改限界變數

下面的程式展示在閉包捕捉變數，然後修改這些變數時會發生什麼事：

```
var index = 0;
Func<IEnumerable<int>> sequence =
   () => Utilities.Generate(30, () => index++);

index = 20;
foreach (int n in sequence())
    WriteLine(n);
WriteLine( “Done” );
index = 100;
foreach (var n in sequence())
    WriteLine(n);
```

此程式輸出 20 到 50 間的數字，後面跟著 100 到 130 間的數字，結果可能會讓你驚訝。這一節接下來討論編譯器所建構產生此結果的程式。此行為合理，你會學到如何利用它。

C# 編譯器將你的查詢表示式轉譯成可執行的程式。雖然 C# 語言有許多新功能，但新的程式還是編譯成與 .NET CLR 的 2.0 版相容的 IL。查詢語法依靠新的組件但不依靠新的 CLR 功能，C# 編譯器將你的查詢與 lambda 表示式轉譯成靜態 delegate、實例 delegate 或閉包，根據 lambda 中的程式選擇要建構哪一個。編譯器採取的路徑視 lambda 的內容而定。這聽起來像是語言的瑣事，但對你的程式有重要的影響。編譯器使用哪一種結構會微妙的改變程式的行為。

並非所有的 lambda 表示式產生相同的程式。最簡單的編譯器工作是對這種程式產生 delegate：

```
int[] someNumbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var answers = from n in someNumbers
              select n * n;
```

編譯器使用靜態 **delegate** 定義實作 `select n * n` 這個 **lambda** 表示式。編譯器輸出的程式如同你寫的一樣：

```
private static int HiddenFunc(int n) => (n * n);

private static Func<int, int> HiddenDelegateDefinition;

// 使用方式：
int[] someNumbers = new int[] { 0, 1, 2, 3, 4, 5,
    6, 7, 8, 9, 10 };
if (HiddenDelegateDefinition == null)
{
    HiddenDelegateDefinition  = new
        Func<int, int>(HiddenFunc);
}
var answers = someNumbers
    .Select<int, int>(HiddenDelegateDefinition);
```

此 **lambda** 表示式的內容並沒有存取任何實例變數或區域變數。此 **lambda** 表示式只存取它的參數。因此，**C#** 編譯器對 **delegate** 的目標建構靜態方法。這是編譯器所能採取的最簡單的路徑。只要表示式能以私用靜態方法實作，編譯器就會產生私用靜態方法與相對應的 **delegate** 定義。這包括如範例的簡單表示式，或存取任何靜態類別變數的方法。

範例 **lambda** 表示式只是包裝在 **delegate** 中的方法呼叫的簡明語法，就是這麼簡單。次簡單的版本是需要存取實例變數，但不存取任何區域變數的 **lambda** 表示式：

```
public class ModFilter
{
    private readonly int modulus;

    public ModFilter(int mod)
    {
        modulus = mod;
    }

    public IEnumerable<int> FindValues(
        IEnumerable<int> sequence)
    {
```

```
        return from n in sequence
               where n % modulus == 0 // 新表示式
               select n * n; // 前面的範例
    }
}
```

編譯器建構實例方法來包裝新表示式的 **delegate**。它與前面的基本概念相同，但使用實例方法使 **delegate** 可以讀取與修改物件的狀態。如同靜態 **delegate** 範例，編譯器將 **lambda** 表示式轉譯成你熟悉的程式。這是 **delegate** 定義與方法呼叫的組合：

```
// 等同 LINQ 前的版本
public class ModFilter
{
    private readonly int modulus;

    // 新方法
    private bool WhereClause(int n) =>
       ((n % this.modulus) == 0);

    // 原來的方法
    private static int SelectClause(int n) =>
        (n * n);

    // 原來的 delegate
    private static Func<int, int> SelectDelegate;

    public IEnumerable<int> FindValues(
        IEnumerable<int> sequence)
    {
        if (SelectDelegate == null)
        {
            SelectDelegate = new Func<int, int>(SelectClause);
        }
        return sequence.Where<int>(
            new Func<int, bool>(this.WhereClause)).
            Select<int, int>(SelectClause);
    }
    // 省略其他方法
}
```

只要你的 lambda 表示式中的程式存取你物件實例的成員變數，編譯器就會產生代表你的 lambda 表示式中的程式的實例方法。這裡沒有什麼魔法。編譯器讓你少打一些字，就是這樣而已，它還是一般的方法呼叫。

但若 lambda 表示式中的程式存取了區域變數或方法參數，編譯器要多做一些工作。此時需要一個閉包。編譯器產生私用套疊類別以實作區域變數的閉包，區域變數必須傳給實作 lambda 表示式的內容的 delegate。此外，任何由 lambda 表示式執行的區域變數修改必須在外層範圍可見。Anders Hejlsberg、Mads Torgersen、Scott Wiltamuth 與 Peter Golde 等人合著的 C# Programming Language，第三版（與後續版本）的§ 7.14.4.1（Microsoft Corporation，2009）描述了這個行為。當然，你可以有一個以上的變數同時在內層與外層範圍，也可有一個以上的查詢表示式。

讓我們對範例方法做個小修改，以讓它存取區域變數：

```
public class ModFilter
{
    private readonly int modulus;

    public ModFilter(int mod)
    {
        modulus = mod;
    }

    public IEnumerable<int> FindValues(
        IEnumerable<int> sequence)
    {
        int numValues = 0;
        return from n in sequence
               where n % modulus == 0 // 新表示式
               // select 句子存取區域變數：
               select n * n / ++numValues;
    }
    // 省略其他方法
}
```

注意 select 句子需要存取 numValues 這個區域變數。為建構閉包，編譯器建構套疊的類別來實作你需要的行為。下面的程式與編譯器產生的相同：

```
// LINQ 之前的簡單閉包版本
public class ModFilter
{
    private sealed class Closure
    {
        public ModFilter outer;
        public int numValues;

        public int SelectClause(int n) =>
             ((n * n) / ++this.numValues);
    }

    private readonly int modulus;

    public ModFilter(int mod)
    {
        this.modulus = mod;
    }

    private bool WhereClause(int n) =>
         ((n % this.modulus) == 0);

    public IEnumerable<int> FindValues
        (IEnumerable<int> sequence)
    {
        var c = new Closure();
        c.outer = this;
        c.numValues = 0;
        return sequence.Where<int>
            (new Func<int, bool>(this.WhereClause))
            .Select<int, int>(
                new Func<int, int>(c.SelectClause));
    }
}
```

在這個版本中，編譯器建構套疊的類別以保存在 **lambda** 表示式中被存取或修改的所有變數。事實上，這些區域變數完整的由套疊類別的欄位取代。**lambda** 表示式中的程式與 **lambda** 外（但在區域方法中）的程式都存取相同的欄位。**lambda** 表示式中的邏輯被編譯成內層類別中的方法。

編譯器處理 **lambda** 表示式中使用的方法參數與處理區域變數的方式完全相同：它將參數複製到表示閉包的套疊類別中。

讓我們重新檢視一開始的例子。如此行為的原因現在就很清楚了。`incrementBy` 這個變數，在被放在閉包中之後但在查詢執行之前修改。你修改內部結構，然後預期它會即時還原並使用之前的版本。

在查詢之間修改限界變數，會因延遲執行的互動與編譯器實作閉包的方式而引發錯誤。因此，你應該避免修改被閉包捕捉的限界變數。

例外的最佳做法

錯誤就是會發生。無論多麼努力，程式還是會遇到預期外的狀況。.NET Framework 的方法或是成功或是拋出例外以表示失敗。若你依循本書的最佳做法，你寫的函式庫與應用程式會更容易使用與擴充。讓程式在拋出例外時更穩健是 C# 開發者的重要技巧。

你的程式會呼叫可拋出例外的方法。呼叫可拋出例外的方法時，程式的行為必須能夠預期。

你的程式也可能直接拋出例外。.NET Framework Design Guidelines 建議在你的方法不能執行被要求的動作時拋出例外。你必須提供診斷所需的各種資訊、可能的改正方式、失敗的根源。你還得確保應用程式處於可知的狀態以供復原。

這一章的方法說明以清楚與精確的方式透過例外溝通失敗的方式。你還會學到如何管理程式狀態以提升復原的可能性。

做法 42 以例外回報方法約定失敗

任何不能執行設計動作的方法應該以例外回報錯誤。有錯的程式很容易會被忽略，而檢查錯誤或傳遞錯誤的程式會影響執行的正常流程、妨礙程式的邏輯。但例外不應該用來作為流程控制機制。這表示你必須以其他公開方法來減少函式庫的使用者在正常操作條件下收到例外的機率。例外在執行期的成本很高，而撰寫無例外的程式很困難。提供 API 給開發者測試條件而無需到處撰寫 `try/catch` 區塊。

例外是比較好的失敗回報機制，因為它們比回傳錯誤代碼的機制有更多的優勢。回傳代碼是方法的格式的一部分，它們通常用來傳達錯誤以外的資訊。回傳的代碼通常是運算的結果，例外只有一個目的：回報錯誤。由於例外是類別型別且你可以繼承自定的例外型別，你可以使用例外來傳達失敗的各種資訊。

錯誤回傳代碼必須由呼叫方的方法處理。相對的，例外在呼叫堆疊向上傳遞直到遇到合適的 `catch` 句子。這讓開發者可以自由的在多個呼叫堆疊層次間隔離錯誤處理程序與錯誤的產生。由於例外類別的豐富性，此隔離不會遺失錯誤資訊。

最後，例外較不容易忽略。若程式沒有合適的 `catch` 句子，則例外會終止應用程式。發生未處理的失敗時可能會導致資料損毀，所以不可以繼續執行。

使用例外回報約定失敗並不表示無法完成你要的工作的方法必須以拋出例外結束。這並不表示每個失敗都是個例外。`File.Exists()` 於檔案存在時回傳 `true` 且在不存在時回傳 `false`。`File.Open()` 在檔案不存在時拋出例外。差別很簡單：`File.Exists()` 以報告檔案是否存在來滿足它的約定。此方法於檔案不存在時還是成功。相對的，`File.Open()` 只有在檔案存在、使用者可以讀取檔案、且行程可開啟檔案供讀取才成功。在第一種情況下，方法在告訴你不想聽的答案時一樣成功。在第二種情況下，方法失敗且你的程式不能繼續。不想要的答案跟失敗是不同的。此方法成功；它給出你要求的資訊。

這種差異對你如何命名方法有重要的影響。執行動作的方法的名稱應該清楚的表示要執行的動作。相對的，測試特定動作的方法的命名應該要指出測試動作。此外，你應該提供測試方法以減少使用例外作為流程控制機制的需求。處理例外花的時間比正常方法呼叫要多。你應該在你的類別中盡量建構方法以讓使用者在執行工作前測試可能的失敗條件。這種做法讓他們能夠更防衛性的設計程式，同時若開發者選擇不在呼叫方法前測試條件時你還是可以拋出例外。

撰寫會拋出例外的方法時，你還應該提供測試導致例外的條件的方法。在內部，你可以使用這些測試方法在繼續前檢查前置條件，在失敗時拋出例外。

假設你有個類別在特定位置沒有特定小工具時會失敗。若你的 API 只有該方法但沒有在程式中提供替代路徑時，你等於鼓勵開發者這樣寫程式：

```
// 不要學：
DoesWorkThatMightFail worker = new DoesWorkThatMightFail();
try
{
    worker.DoWork();
}
catch (WorkerException e)
{
    ReportErrorToUser(
         “Test Conditions Failed. Please check widgets”);
}
```

相對的，你應該加上公開方法讓開發者在執行工作前明確的檢查條件：

```
public class DoesWorkThatMightFail
{
    public bool TryDoWork()
    {
        if (!TestConditions())
            return false;
        Work(); // 可能拋出例外，但可能性很低
        return true;
    }

    // 只在失敗時呼叫，
    // 表示大災難
    public void DoWork()
    {
        Work(); // 失敗時會拋出
    }

    private bool TestConditions()
    {
        // 省略內容
        // 在這裡測試條件
        return true;
    }

    private void Work()
    {
        // 省略
        // 在這裡進行工作
    }
}
```

此模式要求你寫出四個方法：兩個公開方法與兩個私用方法。`TryDoWork()` 方法檢查輸入參數與執行工作的內部物件狀態。然後它呼叫 `Work()` 方法以執行工作。`DoWork()` 呼叫 `Work()` 方法並讓任何失敗產生例外。.NET 使用這種做法是因為拋出例外會影響效能，開發者希望在方法失敗前測試條件以避免這些成本。

現在，加上前面額外的程式後，希望在方法失敗前測試條件的開發者能以更乾淨的方式執行：

```
if (!worker.TryDoWork())
{
    ReportErrorToUser
        ( “Test Conditions Failed. Please check widgets” );
}
```

實務上，測試前置條件可做更多的檢查，像是參數檢查與內部狀態檢查。通常你會在你的類別處理不可信任的來源的輸入，例如使用者輸入、檔案輸入、或未知程式給的參數時使用這種做法。這種失敗有應用程式特定的復原方式且相當常發生。你以不涉及例外的控制機制支援它們。注意我沒有宣稱 `Work()` 不會拋出例外。其他更在意料之外的失敗會在一般的參數檢查之後發生，就算使用者呼叫 `TryDoWork()`，這些失敗還是會以例外回報。

你的方法無法完成約定時拋出例外是你的責任。約定失敗總是以例外回報。由於例外不應該作為流程控制機制，你應該提供其他方法讓開發者在呼叫可能會拋出例外的方法前檢查無效的條件。

做法 46　以 using 與 try/finally 清理資源

使用 **unmanaged** 系統資源的型別應該以 `IDisposable` 界面的 `Dispose()` 方法明確的釋放。.NET 環境的規則讓責任落在使用該型別的程式而非該型別或系統上。因此，使用具有 `Dispose()` 方法的型別時，呼叫 `Dispose()` 來釋放資源是你的責任。確保 `Dispose()` 一定會被呼叫的最佳方式是利用 `using` 陳述或 `try/finally` 區塊。

所有持有 **unmanaged** 資源的型別都實作了 `IDisposable` 界面。此外，它們會為你忘記適當處置的狀況建構終結程序。若你忘記處置這些資源，非記憶體

的資源會在之後終結程序得到機會執行時釋放。這些物件在記憶體停留很久，你的應用程式變成吃掉許多資源的怪獸。

幸好，C# 語言的設計者知道明確的釋放資源是常見的工作。它們在語言中加入關鍵字以讓工作變簡單。

假設你寫了這樣的程式：

```
public void ExecuteCommand(string connString,
  string commandString)
{
    SqlConnection myConnection = new SqlConnection(
      connString);
    var mySqlCommand = new SqlCommand(commandString,
      myConnection);

    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();
}
```

此例中的兩個可處置物件沒有適當的清理：`SqlConnection` 與 `SqlCommand`。這兩個物件會待在記憶體中直到呼叫終結程序為止（這兩個類別都從 `System.ComponentModel.Component` 繼承終結程序）。

你在結束命令與連線時以呼叫 `Dispose` 來解決這個問題：

```
public void ExecuteCommand(string connString,
  string commandString)
{
    var myConnection = new SqlConnection(
      connString);
    var mySqlCommand = new SqlCommand(commandString,
      myConnection);

    myConnection.Open();
    mySqlCommand.ExecuteNonQuery();

    mySqlCommand.Dispose();
    myConnection.Dispose();
}
```

如此是可以的，除非執行 SQL 命令時有拋出例外，這種情況下 Dispose() 的呼叫不會發生。using 陳述會確保呼叫 Dispose()。你在 using 陳述中分配物件，而 C# 編譯器會在每個物件的前後產生 try/finally 區塊：

```
public void ExecuteCommand(string connString,
  string commandString)
{
    using (SqlConnection myConnection = new
      SqlConnection(connString))
    {
        using (SqlCommand mySqlCommand = new
          SqlCommand(commandString,
          myConnection))
        {
            myConnection.Open();
            mySqlCommand.ExecuteNonQuery();
        }
    }
}
```

在函式內使用 Disposable 物件時，using 句子是確保物件被適當處置的最簡單做法。using 陳述在每個分配物件的前後產生 try/finally 區塊。這兩個區塊產生相同的 IL：

```
SqlConnection myConnection = null;

// using 句子範例：
using (myConnection = new SqlConnection(connString))
{
    myConnection.Open();
}

// try/catch 區塊範例：
try
{
    myConnection = new SqlConnection(connString);
    myConnection.Open();
}
finally
{
    myConnection.Dispose();
}
```

若你使用 using 陳述與沒有支援 IDisposable 界面的型別的變數，C# 編譯器會產生錯誤。例如：

```
// 不能編譯：
// string 是封閉的且不支援 IDisposable
using (string msg = “This is a message”)
    Console.WriteLine(msg);
```

using 陳述只有在編譯期型別有支援 IDisposable 時才可行。你不能對模糊物件使用它：

```
// 不能編譯
// 物件不支援 IDisposable
using (object obj = Factory.CreateResource())
    Console.WriteLine(obj.ToString());
```

你需要 as 句子安全的處置可能有或沒有實作 IDisposable 的物件：

```
// 改正錯誤
// 物件可能有或沒有實作 IDisposable
object obj = Factory.CreateResource();
using (obj as IDisposable)
    Console.WriteLine(obj.ToString());
```

若 obj 實作了 IDisposable，using 陳述會產生清理程式。若沒有，using 陳述變成 using(null)，安全但不做任何事。如果你不確定是否應該以 using 區塊包裝物件，以安全考量：假設它有並包裝在前述的 using 句子中。

以上涵蓋了簡單的情況：使用方法區域內的可處置物件時，將物件包裝在 using 陳述中。現在可以進一步檢視更複雜的運用。第一個範例中有兩個不同的物件需要處置：連線與命令。我的範例建構兩個不同的 using 陳述，各包裝一個需要處置的物件。每個 using 陳述產生不同的 try/finally 區塊。如此則程式寫成這樣：

```
public void ExecuteCommand(string connString,
  string commandString)
{
    SqlConnection myConnection = null;
    SqlCommand mySqlCommand = null;
    try
    {
        myConnection = new SqlConnection(connString);
```

```
        try
        {
            mySqlCommand = new SqlCommand
                (commandString, myConnection);

            myConnection.Open();
            mySqlCommand.ExecuteNonQuery();
        }
        finally
        {
            if (mySqlCommand != null)
                mySqlCommand.Dispose();
        }
    }
    finally
    {
        if (myConnection != null)
            myConnection.Dispose();
    }
}
```

每個 `using` 陳述產生一個新的套疊的 `try/finally` 區塊。幸好在一個方法中分配兩個不同的物件且都有實作 `IDisposable` 的情況很少見。若遇到，就這樣寫，因為它還是可行。然而，我發現這樣的程式很醜，因此我在分配多個實作 `IDisposable` 的物件時，我偏好自行撰寫 `try/finally` 區塊：

```
public void ExecuteCommand(string connString,
  string commandString)
{
    SqlConnection myConnection = null;
    SqlCommand mySqlCommand = null;
    try
    {
        myConnection = new SqlConnection(connString);
        mySqlCommand = new SqlCommand(commandString,
          myConnection);

        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
    }
    finally
    {
```

```
        if (mySqlCommand != null)
            mySqlCommand.Dispose();
        if (myConnection != null)
            myConnection.Dispose();
    }
}
```

這樣寫的一個原因是你很容易會寫出一個有 as 陳述的 using 句子：

```
public void ExecuteCommand(string connString,
  string commandString)
{
    // 壞主意。可能會產生資源洩露！
    SqlConnection myConnection =
      new SqlConnection(connString);
    SqlCommand mySqlCommand = new SqlCommand
       (commandString, myConnection);
    using (myConnection as IDisposable)
    using (mySqlCommand as IDisposable)
    {
        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
    }
}
```

這看起來比較乾淨，但潛藏微妙的 bug。SqlConnection 物件在 SqlCommand() 建構元拋出例外時不會被處置。myConnection 參考的物件已經建構，但 SqlCommand 建構元執行時程式不會進入 using 區塊。using 區塊內沒有建構元時，Dispose 的呼叫會被略過。你必須確保實作 IDisposable 的物件在 using 區塊或 try 區塊的範圍內分配，不然資源洩漏可能會發生。

目前為止你已經處理了兩種最明顯的情況。在方法中分配一個可處置物件時，using 陳述是確保你分配的資源在所有情況下都會被釋放的最佳方式。在同一個方法中分配多個物件時，建構多個 using 區塊或自行撰寫單一 try/finally 區塊。

釋放可處置物件還有一個細微差別。某些型別同時支援以 Dispose 方法與 Close 方法釋放資源。SqlConnection 就是這種類別。你可以如下關閉 SqlConnection：

```
public void ExecuteCommand(string connString,
  string commandString)
{
    SqlConnection myConnection = null;
    try
    {
        myConnection = new SqlConnection(connString);
        SqlCommand mySqlCommand = new SqlCommand
          (commandString, myConnection);

        myConnection.Open();
        mySqlCommand.ExecuteNonQuery();
    }
    finally
    {
        if (myConnection != null)
            myConnection.Close();
    }
}
```

這個版本會關閉連線，但與處置它不同。`Dispose` 方法不只是釋放資源：它還通知垃圾回收此物件不再需要被終結。`Dispose` 會呼叫 `GC.SuppressFinalize()`，`Close` 通常不會。就算不需要終結，物件還待在終結佇列中。若可選擇，`Dispose()` 較 `Close()` 更好。細節見做法 17。

`Dispose()` 並不將物件從記憶體移除。它是個讓物件釋放 **unmanaged** 資源的掛鉤。這表示處置還在使用中的物件時會有問題。上面的範例使用 `SQLConnection`。`SQLConnection` 的 `Dispose()` 方法關閉資料庫連線。處置完連線後，`SQLConnection` 物件還在記憶體中，但不再與資料庫連線。它在記憶體中，但沒有用處。不要處置程式還在參考的物件。

C# 的資源管理有時候比 C++ 的資源管理更困難。你無法依靠決定性的終結程序清理所有使用到的資源。但垃圾回收環境讓你比較輕鬆。大部分你會使用到的型別沒有有實作 `IDisposable`。少部分的 .NET Framework 類別實作了 `IDisposable`，在各種情況下都要記得處置它們。你應該將這些物件包裝在 `using` 句子或 `try/finally` 區塊中。無論用哪一個，每次都要確保物件正確的處置掉。

做法 47 建構完整的應用程式專屬例外類別

例外是回報可能在發生錯誤位置的遠方處理的錯誤的機制。所有關於錯誤成因的資訊必須放在例外物件中。因此，你或許會想要將低階錯誤轉譯成應用程式專屬的錯誤而不損失任何原始錯誤的資訊。在你的 C# 應用程式中建構專屬的例外類別時必須考慮的很周到。

第一步是認識何時與為何要建構新的例外類別與如何建構資訊性的例外階層。開發者使用你的函式庫撰寫 `catch` 句子時會根據例外的執行期型別區分要採取的動作。不同的例外類別可採取不同的動作：

```
try
{
    Foo();
    Bar();
}
catch (MyFirstApplicationException e1)
{
    FixProblem(e1);
}
catch (AnotherApplicationException e2)
{
    ReportErrorAndContinue(e2);
}
catch (YetAnotherApplicationException e3)
{
    ReportErrorAndShutdown(e3);
}
catch (Exception e)
{
    ReportGenericError(e);
    throw;
}
finally
{
    CleanupResources();
}
```

不同的執行期例外型別可有不同的 catch 句子。身為應用程式的作者，catch 句子需採取不同的動作時，你必須建構或使用不同的例外類別。注意上面每個例外均以不同方式處理。開發者只在處理方式不同時才會想要為不同的例外類別提供不同的 catch 句子，不然就只是多餘的工作。因此，你只應該在認為開發者會對導致例外的問題採取不同的動作時才要考慮建構不同的例外類別。若非如此，你的使用者將沒得選擇。你可以在遇到例外時結束應用程式，這樣當然會減少工作量，但不會讓使用者滿意。或者他們會研究例外以判斷錯誤是否可改正：

```
private static void SampleTwo()
{
    try
    {
        Foo();
        Bar();
    }
    catch (Exception e)
    {
        switch (e.TargetSite.Name)
        {
            case “Foo”:
                FixProblem(e);
                break;
            case “Bar”:
                ReportErrorAndContinue(e);
                break;
            // Foo 或 Bar 呼叫的一些程序
            default:
                ReportErrorAndShutdown(e);
                throw;
        }
    }
    finally
    {
        CleanupResources();
    }
}
```

這比使用多個 catch 句子差多了。它是非常脆弱的程式：若你改變程序的名稱，它就會出問題。若你將產生錯誤的呼叫移動到共用的工具函式中，它就會出問題。例外的呼叫堆疊越深，這種程式就越脆弱。

在更深入這個主題之前，讓我加上兩個前提。首先，例外不是用來處理所有的錯誤狀況。沒有絕對的規則，但我偏好對若沒有立即處理或回報則會產生持續性問題的錯誤狀況拋出例外。舉例來說，資料庫的資料完整性錯誤應該產生例外。若忽略它只會讓問題惡化。使用者視窗偏好位置的問題比較不會產生嚴重的後果，回傳表示失敗的代碼就夠了。

其次，撰寫 throw 陳述並不表示就要建構新的例外類別。我的建議是依正常的人性建構更多而非更少的例外類別：人們在拋出例外時似乎傾向濫用 System.Exception。這對呼叫方的程式提供很少的資訊。相對的，思考並建構必要的例外類別以讓呼叫方理解成因並提供更好的復原機會。

我再說一次：製作不同例外類別的理由 - 事實上，唯一的理由 - 是讓使用你的 API 的開發者在撰寫 catch 處理程序時比較容易採取不同的動作。檢查可能採取復原動作的錯誤狀況並建構特定的例外類別來處理這些動作。你的應用程式能否從不見的檔案與目錄中復原？它能否從不對的安全權限復原？不見的網路資源又如何？遇到可能會需要不同動作與復原機制的錯誤時建構新的例外類別。

所以現在你要建構自定例外類別。建構新的例外類別時有非常明確的責任。你的例外類別必須在 Exception 結束。你必須從 System.Exception 類別繼承你的例外類別。你很少會對此基底類別增加功能。不同的例外類別的目的是要能夠在 catch 句子中區分錯誤的成因。Visual Studio 或其他編輯器等工具有模板可供建構新的例外類別。

但不要讓你建構的例外類別少了任何東西。Exception 類別有四個建構元：

```
// 預設建構元
public Exception();

// 加上訊息
public Exception(string);

// 加上訊息與內部例外
public Exception(string, Exception);

// 從輸入串流產生
protected Exception(
    SerializationInfo, StreamingContext);
```

建構新的例外類別時要建構以上四個建構元。注意最後一個建構元意味著你的例外類別必須能夠序列化。不同的狀況呼叫不同例外建構方法。（若你選擇繼承自其他例外類別，你應該加上其基底類別的建構元）。將工作委派給基底類別的實作：

```
[Serializable]
public class MyAssemblyException :
  Exception
{
    public MyAssemblyException() :
        base()
    {
    }

    public MyAssemblyException(string s) :
        base(s)
    {
    }

    public MyAssemblyException(string s,
      Exception e) :
        base(s, e)
    {
    }

    // 不一定在所有平台上的 .NET Core 都有支援
    protected MyAssemblyException(
      SerializationInfo info, StreamingContext cxt) :
        base(info, cxt)
    {
    }
}
```

這個新版本提供更多問題發生點的資訊。只要你有建構合適的 `ToString()` 方法，你就會有完整描述產生問題的物件的狀態的例外。不止如此，內部的例外顯示問題的根源：第三方函式庫。

此技巧稱為例外轉譯，將低階例外翻譯成提供更多問題背景的高階例外。在錯誤發生時產生的資訊越多，使用者就越容易診斷與改正問題。透過建構自定的例外型別，你可以轉譯低階問題成為帶有應用程式專屬資訊的例外以供診斷與改正問題。

你的應用程式會拋出例外 - 最好是不常，但還是會發生。若你沒有特別設定，呼叫架構核心的方法發生問題時你的應用程式會產生預設的 .NET Framework 例外。提供更多的細節資訊能讓你與你的使用者診斷與改正問題。可採取不同改正步驟時建構不同的例外類別。提供基底例外類別支援的所有建構元以建構完整的例外類別。使用 `InnerException` 屬性攜帶低階錯誤狀況產生的所有錯誤資訊。

做法 48 偏好強例外保證

拋出例外時，你對應用程式產生了破壞性的事件。流程控制被破壞，預期中的動作不會執行。更糟糕的是你將清理操作留給撰寫最終捕捉到例外的程式的程式設計師。捕捉到例外時可採取的動作與拋出例外時你如何管理程式狀態有直接的關係。幸好 C# 社群無需建構自定的例外安全策略；C++ 社群已經幫我們把困難的部分做好了。從 Tom Cargill 的 "Exception Handling：A False Sense of Security" 一文開始，以及 Dave Abrahams，Herb Sutter，Scott Meyers，Matt Austern，與 Greg Colvin 的貢獻，C++ 社群已經發展出 C# 應用程式可採用的最佳實踐。關於例外處理的討論發生在 1994 年至 2000 年這六年間。他們討論、爭辯、與研究各種困難的問題。我們應該在 C# 中利用這些成果。

Dave Abrahams 定義了三種例外安全保證：基本保證、強保證、與無拋出保證。Herb Sutter 在它的 Exceptional C++（Addision-Wesley，2000）一書討論了這些保證。基本保證表示沒有資源洩漏且在例外離開拋出它的函式後物件還是處於有效狀態。這表示拋出例外的方法的 `finally` 句子會被執行。強例外保證在基本保證之外加上例外發生時程式的狀態不會改變。無拋出保證表示操作不會失敗，方法不會拋出例外。強例外保證在例外的復原與簡化例外處理間提供最佳的平衡。

.NET CLR 對基本保證提供了一些幫助。此環境處理了記憶體管理。唯一能夠因例外而洩漏資源的方式是在持有實作 `IDisposable` 的資源時拋出例外。做法 17 解釋過如何在遇到例外時避免資源洩漏，但那只是故事的一部分。你還要負責確保物件的狀態有效。假設你的型別快取了集合的大小與集合，在 `Add()` 操作拋出例外後你必須確保大小符合實際的儲存體。你的應用程式中有無數的動作在部分完成時會讓應用程式處於無效的狀態。這些狀況很難

處理，因為沒有幾個自動化支援的標準做法。這些問題有很多能以強保證解決。

強保證表示若操作因例外終止，程式的狀態還是不變。操作不是完成就是沒有改變程式的狀態；沒有中間地帶。強保證的好處是你可以在依循強保證的情況下於捕捉到例外時更容易的繼續執行。捕捉到例外時，操作不會發生。它不會啟動且不會做出改變。程式的狀態與沒有開始動作時相同。

我的許多建議可幫助你確保符合強例外保證。你的程式用到的資料元素應該儲存在不可變的值型別中。你也可以使用函式性程式設計方式，像是 LINQ 查詢。這種程式設計方式自動的依循強例外保證。

有時候，你不能使用函式性程式設計方式。若你結合防衛性拷貝與交換技巧，所有對程式狀態的修改可在執行任何會拋出例外的操作後發生。基本原則是以下列方式執行任何資料修改：

1. 製作會被修改的資料的防衛性拷貝。
2. 對防衛性拷貝資料進行修改。
3. 交換暫時的拷貝與原始資料。此操作不能拋出例外。

為你的儲存體型別建構不可變資料結構時這些原則會很容易依循。

以下面使用防衛性拷貝修改員工職稱與薪資的程式為例：

```
public void PhysicalMove(string title, decimal newPay)
{
    // 薪資資料是個結構：
    // 若例外無效則拋出錯誤
    PayrollData d = new PayrollData(title, newPay,
      this.payrollData.DateOfHire);

    // 若正確產生 d 則交換：
    this.payrollData = d;
}
```

有時支援強保證太沒有效率，有時你無法在不產生微妙的 bug 下支援強保證。第一個與最簡單的例子是迴圈結構。程式於迴圈中修改程式狀態並可能會拋出例外時，你將面對困難的抉擇：為迴圈中用到的所有物件建構防衛

性拷貝，或降低標準只支援基本例外保證。沒有絕對的規則，但在 managed 環境中複製分配於 heap 的物件並不像在原生環境中的代價一樣高。.NET 花了很多時間在記憶體管理的最佳化。我偏好盡可能的支援強例外保證，就算要複製大容器也一樣：從錯誤中復原的能力比避免複製的效能提升更重要。若例外無論如何都會導致程式終止，則考慮強例外保證是無意義的。最大的問題是交換參考型別會產生程式錯誤。以下面的程式為例：

```
private List<PayrollData> data;
public IList<PayrollData> MyCollection
{
    get { return data; }
}

public void UpdateData()
{
    // 可能會失敗的不可靠操作：
    var temp = UnreliableOperation();

    // 此操作僅於
    // UnreliableOperation 不拋出
    // 例外時才會發生
    data = temp;
}
```

它只有一個問題：不會成功。`MyCollection` 屬性回傳資料物件的參考。此類別的所有用戶在你呼叫 `UpdateData` 之後還是持有原始 `List<>` 的參考。它們看到的還是舊資料。交換技巧對參考型別無效 - 只對值型別有效。為改正，你必須替換目前參考物件中的資料並確保以不會拋出例外的方式進行。這很困難，因為它是兩個不同的不可分割操作：移除集合中的所有現存物件並加入新物件。你可能會認為移除與加入新項目的風險很小：

```
private List<PayrollData> data;
public IList<PayrollData> MyCollection
{
    get
    {
        return data;
    }
}
```

```
public void UpdateData()
{
    // 可能會失敗的不可靠操作：
    var temp = UnreliableOperation();

    // 此操作僅於
    // UnreliableOperation 不拋出
    // 例外時才會發生
    data.Clear();
    foreach (var item in temp)
        data.Add(item);
}
```

這是合理但並不絕對安全的解決方案。我會提出來是因為“合理”通常是你所需要的。但你需要絕對安全時必須做更多的工作。信封 - 信紙模式能隱藏物件內部的交換以讓你安全的進行交換。

信封 - 信紙模式在公開給你的程式用戶的包裝（信封）中隱藏實作（信紙）。此例中，你建構包裝集合與實作 `IList<PayrollData>` 的類別。該類別帶有 `IList<PayrollData>` 並顯露它的所有方法給類別的用戶。

現在你的類別與 `Envelope` 類別一起處理它的內部資料：

```
private Envelope data;
public IList<PayrollData> MyCollection
{
    get
    {
        return data;
    }
}

public void UpdateData()
{
    data.SafeUpdate(UnreliableOperation());
}
```

`Envelope` 類別將每個請求傳遞給 `IList<PayrollData>` 以實作 `IList`：

```
public class Envelope : IList<PayrollData>
{
    private List<PayrollData> data = new List<PayrollData>();
```

```
public void SafeUpdate(IEnumerable<PayrollData> sourceList)
{
    // 製作拷貝：
    List<PayrollData> updates =
        new List<PayrollData>(sourceList.ToList());

    // 交換：
    data = updates;
}

public PayrollData this[int index]
{
    get { return data[index]; }
    set { data[index] = value; }
}

public int Count => data.Count;

public bool IsReadOnly =>
    ((IList<PayrollData>)data).IsReadOnly;

public void Add(PayrollData item) => data.Add(item);

public void Clear() => data.Clear();

public bool Contains(PayrollData item) =>
    data.Contains(item);

public void CopyTo(PayrollData[] array, int arrayIndex) =>
    data.CopyTo(array, arrayIndex);

public IEnumerator<PayrollData> GetEnumerator() =>
    data.GetEnumerator();

public int IndexOf(PayrollData item) =>
    data.IndexOf(item);

public void Insert(int index, PayrollData item) =>
    data.Insert(index, item);
```

```
    public bool Remove(PayrollData item)
    {
        return ((IList<PayrollData>)data).Remove(item);
    }

    public void RemoveAt(int index)
    {
        ((IList<PayrollData>)data).RemoveAt(index);
    }

    IEnumerator IEnumerable.GetEnumerator() =>
        data.GetEnumerator();
}
```

有許多模板程式要檢視，大部分都很直白。但有幾個重要的部分應該要仔細檢視。首先，注意 `IList` 界面的許多成員由 `List <T>` 類別明確的實作。這是許多方法需要型別轉換的原因。還有，我是根據 `PayrollData` 型別為值型別來寫的程式。若 `PayrollData` 是參考型別，此程式會比較簡單。我將 `PayrollData` 作為值型別以展示差異。型別檢查是根據 `PayrollData` 型別為值型別來寫。

當然，這個練習的目的是要建構與實作 `SafeUpdate` 方法。注意它基本上執行與之前相同的工作。這保證此程式是安全的，就算在多執行緒應用程式中也一樣。此交換不能被中斷。

在一般情況下，你無法改正交換參考型別的問題同時又確保所有用戶取得物件目前的拷貝。交換只能用在值型別。這應該足夠了。

最後，且最嚴格的，是無拋出保證。無拋出保證如同其名稱所示：無拋出保證的方法保證完整執行且絕對不會讓例外離開方法。這對大規模程式的所有程序是不切實際的，但在有些地方，方法必須是無拋出保證。終結程序與 `Dispose` 方法必須不會拋出例外，它們若拋出例外則會比其他方式產生更大的問題。以終結程序來說，拋出例外會終止程式而沒有進行清理。將方法包裝在 `try/catch` 區塊並吃掉所有例外是達成無拋出保證的做法。`Dispose()` 與 `Finalize()` 等大部分必須滿足無拋出保證的方法的責任有限制。因此，你應該能夠以防衛性程式寫出這些滿足無拋出保證的方法。

Dispose 方法拋出例外時，系統會有兩個例外。.NET 環境釋放第一個例外並拋出新的例外。你在程式中無法捕捉第一個例外；它被系統吃掉。這讓錯誤處理變得更複雜。你要如何從看不到的錯誤中復原？

你不應該從例外過濾的 when 句子中拋出。新的例外成為作用中的例外，任何關於原始例外的資訊都無法讀取。

最後一個適用無拋出保證的地方是 **delegate** 的目標。**delegate** 的目標拋出例外時，沒有其他的 **delegate** 目標會被同一個多重發送的 **delegate** 呼叫。唯一的解決方式是確保不會從 **delegate** 的目標拋出任何例外。再重複一次：**delegate** 的目標（包括事件處理程序）不應該拋出例外。這樣做表示發出事件的程式不能參加強例外保證。但我要修改這個建議。做法 **7** 顯示如何叫用 **delegate** 以讓你可以從例外中復原。但並非所有人都這麼做，因此你應該避免在 **delegate** 處理程序中拋出例外。你不在 **delegate** 中拋出例外並不表示其他人會這麼做；不要讓你自己的 **delegate** 叫用依靠無拋出保證。它是防衛性的程式設計：你盡可能做好，因為其他程式設計師可能會寫出最差的程式。

例外會改變應用程式的流程控制。在最糟糕的情況下，什麼事都可能發生 - 或不發生。唯一能知道拋出例外時什麼有改變什麼沒有改變的辦法是實行強例外保證，然後操作不是完成就是不作任何改變。終結程序、Dispose()、when 句子、與 **delegate** 的目標是特殊狀況且應該完成而不會讓例外在任何情況下逸出。最後一句話，注意交換參考型別；它會產生很多微妙的 **bug**。

做法 49 偏好例外過濾而非 catch 與重新拋出

在標準的 catch 句子中，你根據例外型別捕捉例外。其他都不重要。任何套用在程式狀態、物件狀態、或例外的屬性的邏輯都必須在 catch 句子中處理。這些限制可能導致程式捕捉例外，經過分析後重新拋出相同例外。

這種習慣使得之後的分析更困難。它也增加應用程式的執行期成本。你可以使用例外過濾作為管理捕捉與處理例外的辦法以更好的支援之後的分析並避免那些成本。為此，你應該養成使用例外過濾而非 catch 句子中的條件邏輯的習慣。

編譯器產生的程式的差異是養成使用例外過濾而捕捉與重新拋出的習慣最好的理由。例外過濾是 `catch` 句子上的表示式，跟在 `when` 關鍵字後面，限制 `catch` 句子處理特定型別的例外的時機：

```
var retryCount = 0;
var dataString = default(String);

while (dataString == null)
{
    try
    {
        dataString = MakeWebRequest();
    }
    catch (TimeoutException e) when(retryCount++ < 3)
    {
        WriteLine( “Operation timed out. Trying again” );
        // 重新嘗試前先暫停
        Task.Delay(1000 * retryCount);
    }
}
```

編譯器產生在堆疊展開之前對例外過濾求值的程式。原始例外的位置已知，所有呼叫堆疊上的資訊，包括區域變數的值，都還是知道。若表示式求值為 `false` 則執行期在呼叫堆疊繼續向上搜尋相符的 `catch` 表示式以處理該例外。隨著執行期在呼叫堆疊繼續向上搜尋，程式的狀態不會被擾亂。

相較於捕捉到例外時的處理然後判斷你無法處理這個問題並重新拋出例外：

```
var retryCount = 0;
var dataString = default(String);

while (dataString == null)
{
    try
    {
        dataString = MakeWebRequest();
    }
    catch (TimeoutException e)
    {
        if (retryCount++ < 3)
        {
```

```
            WriteLine(“Timed out. Trying again”);
            // 重新嘗試前先暫停
            Task.Delay(1000 * retryCount);
        }
        else
            throw;
    }
}
```

此程式的執行期發現有個 catch 句子可以處理該例外。只要執行期一發現 catch 句子，它就開始展開堆疊。在方法中宣告的大部分區域變數不再有可存取的路徑（若閉包可存取，則已經關閉的變數可能可以存取）。可供診斷成因的重要值可能不再有效。狀態已經遺失。

在 catch 句子中，你的程式判斷出無法復原此錯誤。因此你重新拋出原始例外，上面的程式使用正確的語法重新拋出。不要直接拋出例外；它會建構新的例外物件與新的拋出原點。

這兩種執行路徑的差異表現於不同的診斷與除錯過程。第二種過程失去所有區域變數的值。它還失去問題發生時執行內容的資訊。使用例外過濾時所有診斷資料帶有可幫助你診斷例外的原始成因的所有程式狀態資訊。使用第一種方法時有些資訊會在展開堆疊時失去。檢查下面這三個方法，我已經在呼叫 TreeOfErrors 產生的例外的呼叫堆疊報告上加了註解：

```
static void TreeOfErrors()
{
    try
    {
        SingleBadThing();
    }
    catch (RecoverableException e)
    {
        throw; // 在呼叫堆疊上報告
    }
}
static void TreeOfErrorsTwo()
{
    try
    {
        SingleBadThing(); // 在呼叫堆疊上報告
```

```
    }
    catch (RecoverableException e) when (false)
    {
        WriteLine( “Can’t happen” );
    }
}
```

當你使用 throw 語法時，呼叫堆疊以呼叫堆疊中的位置回報 throw 的位置。這是因為執行進入了 catch 句子，而例外被重新拋出。try 句子前面的行已經遺失。但在你使用例外過濾時，呼叫堆疊回報導致拋出例外的方法呼叫的位置。我已經盡可能將範例程式修改成最小，因此很容易判斷產生例外的位置。在大應用程式中，判斷產生例外的位置比較困難。

使用例外過濾對程式的效能也有正面的影響。.NET CLR 被最佳化以讓沒有進入 catch 句子的 try/catch 區塊對執行期效能的影響最小。但堆疊展開與進入 catch 句子對執行期有重大的影響。能避免堆疊展開與進入 catch 句子而無法處理例外的例外過濾可改善效能。它不會讓效能更糟。

有幾個你可能會採用的做法應該要改變。在某些情況下，你會發現例外的一些屬性影響你是否能處理例外。最常見的狀況涉及以任務為基礎的非同步程式設計。任務在其執行的程式拋出例外時進入失敗狀態。一個任務可能會啟動一個以上的子任務，因此一個任務可能會因多個例外而失敗。為一致的處理這些狀況，Task 類別的 Exception 屬性是個 AggregateException。你必須檢視儲存在 InnerExceptions 屬性的例外（多個）以判斷 Task 程式是否可以處理例外。

例外一個例子是 COMException 類別。它的 HResult 屬性帶有 interop 呼叫所產生的 COM 的 HRESULT。有些 HRESULT 值你可以處理，有些不行。例外過濾可以管理邏輯而無需進入 catch 句子。

最後一個例子，HTTPException 類別有個 GetHttpCode() 方法可回傳 HTTP 回應碼。你能夠對某些錯誤碼（例如 301 重新導向）採取修正動作，但其他錯誤（例如 404 找不到）不行。例外過濾可確保你只處理可以改正的錯誤。

加上例外過濾意味著你可以將例外處理程式寫成 catch 句子只在你可以完全管理例外時執行。你會保存關於錯誤的更多資訊。你的程式在某些狀況跑的更快。當你需要比例外型別更多的錯誤資訊時，加上例外過濾以在進入 catch 句子前判斷是否可以從例外復原。

做法 50 利用例外過濾的副作用

建構總是回傳 false 的例外過濾似乎有矛盾，但有理由要在所有例外產生時加以檢視。如做法 49 所述，例外過濾的執行是執行期搜尋應用程式的 catch 句子的過程的一部分。這表示它們會在展開堆疊前執行。

讓我們從一個範例開始。正式的程式通常會在某處記錄所有未處理的例外，它們可能會將訊息發送到一個集中處，它們可能在本機記錄資訊。無論如何，好程式會記錄所有的問題。

以下面的方法為例：

```
public static bool ConsoleLogException(Exception e)
{
    var oldColor = Console.ForegroundColor;
    Console.ForegroundColor = ConsoleColor.Red;
    WriteLine(“Error: {0}”, e);
    Console.ForegroundColor = oldColor;
    return false;
}
```

它輸出例外資訊到控制台。它可以放在程式中你想要產生日誌記錄的地方。加上 try/catch 句子並套用回傳 false 的例外過濾：

```
try
{
    data = MakeWebRequest();
}
catch (Exception e) when(ConsoleLogException(e)) { }
catch (TimeoutException e) when(failures++ < 10)
{
    WriteLine(“Timeout error: trying again”);
}
```

上面的例子中有幾個重要的做法。此例外過濾總是回傳 false。它不會回傳 true，不然記錄方法會阻斷例外的傳播。其次，注意 catch 句子以捕捉基底 Exception 類別來捕捉所有例外。雖然它捕捉 Exception 類別，此例外過濾列在 catch 句子列的第一個。此 catch 句子不會處理例外（記得過此濾總是回

傳 false），因此執行期會繼續搜尋應用程式的 catch 句子。這種做法是少數被容許以 catch 捕捉所有例外的做法。

依循這些做法表示你可以在程式中任意加上 catch 句子而不會影響執行。你可以在任何位置加入新的 try/catch 句子。你可以在任何現有的 catch 句子的開頭或 try/finally 區塊之前加上 catch (Exception e) when log(e) {}。

你可以在程式中的不同位置將記錄特殊化。範例記錄所有的例外，但你可以對任何特定的型別套用相同的技巧。由於記錄用的例外過濾總是回傳 false，你可以修改被記錄的例外型別。

另一個選項是將記錄用的 catch 句子作為最後一個 catch 句子。如此只會記錄沒有被 catch 句子處理的例外。

```
try
{
    data = MakeWebRequest();
}
catch (TimeoutException e) when(failures++ < 10)
{
    WriteLine( “Timeout error: trying again” );
}
catch (Exception e) when(ConsoleLogException(e)) { }
```

另外一個以此方法記錄的好處是你可以將它擴充到函式庫與套件而不用修改執行流程。在傳統的情境中，記錄只在應用程式層級加入，且只在頂層方法中。這樣會捕捉到沒有於執行期被任何程式處理的例外。另一個選項是在例外即將在應用程式中拋出的位置建構記錄訊息，這樣可在程式產生例外時加以記錄但不會讓底層程式產生的例外被記錄。這些策略都不會讓函式庫與套件的訊息記錄容易產生。捕捉與重新拋出例外讓你的使用者的除錯循環更加困難（見做法 49）。函式庫的作者可使用這種做法來記錄函式庫的每個公開 API。以此做法，堆疊資訊不會被函式庫的記錄機制修改。

日誌記錄不是不處理例外的例外過濾唯一的用途。你也可以使用例外過濾來確保 catch 句子在除錯過程中執行你的應用程式時不會處理任何例外：

```
try
{
    data = MakeWebRequest();
```

```
}
catch (Exception e) when(ConsoleLogException(e)) { }
catch (TimeoutException e) when((failures++ < 10) &&
    (!System.Diagnostics.Debugger.IsAttached))
{
    WriteLine( “Timeout error: trying again” );
```

此例外過濾在除錯工具進行時可防止 `catch` 句子處理任何例外。不在除錯工具下執行時，此例外過濾會通過且例外句子會執行。

它受執行期環境而非建構設定影響。`Debugger.IsAttached` 屬性只在除錯工具依附與此行程時回傳 `true`。它不受除錯或建構設定影響。這種設定的好處是在除錯工具下執行時不會進入你的 `catch` 句子。如果完全套用在你的程式中，你的應用程式在除錯工具底下執行時不會捕捉任何例外。如果你設定除錯工具在遇到未處理的例外時停止，拋出例外時程式會停止。這種做法不會在其他環境影響你的應用程式。它是大型程式中尋找例外的成因很好的工具。

有副作用的例外過濾是觀察例外拋出的好方式。在大程式中，此工具可改變應用程式的行為以幫助找尋拋出例外的原因。找到來源之後，改正就簡單多了。

索引

※ 提醒您：由於翻譯書排版的關係，部分索引名詞的對應頁碼會和實際頁碼有一頁之差。

符號

數字

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

W

Y

Z

Effective C#中文版 | 寫出良好 C# 程式的 50 個具體做法第三版

作　　者：Bill Wagner
譯　　者：楊尊一
企劃編輯：蔡彤孟
文字編輯：詹祐甯
設計裝幀：張寶莉
發 行 人：廖文良

發 行 所：碁峰資訊股份有限公司
地　　址：台北市南港區三重路 66 號 7 樓之 6
電　　話：(02)2788-2408
傳　　真：(02)8192-4433
網　　站：www.gotop.com.tw
書　　號：ICL049100
版　　次：2017 年 09 月
建議售價：NT$315

讀者服務

- 感謝您購買碁峰圖書，如果您對本書的內容或表達上有不清楚的地方或其他建議，請至碁峰網站：「聯絡我們」\「圖書問題」留下您所購買之書籍及問題。（請註明購買書籍之書號及書名，以及問題頁數，以便能儘快為您處理）
  http://www.gotop.com.tw
- 售後服務僅限書籍本身內容，若是軟、硬體問題，請您直接與軟體廠商聯絡。
- 若於購買書籍後發現有破損、缺頁、裝訂錯誤之問題，請直接將書寄回更換，並註明您的姓名、連絡電話及地址，將有專人與您連絡補寄商品。
- 歡迎至碁峰購物網
  http://shopping.gotop.com.tw

C# Programming

Effective C# 中文版

涵蓋 C# 6.0

因應越來越複雜的 C# 語言與日漸龐大的開發社群，作者歸納了 50 種撰寫更好程式的方法，協助讀者撰寫出堅實、高效率、高性能 C# 6.0 程式。本書新版所提供的新方案包括善用泛型與 LINQ，以及專門一章討論例外處理的最佳做法。

清楚、務實、專家秘訣與實用的範例程式，讓眾多開發者對 Effective C# 愛不釋手。憑藉著豐富的 C# 經驗，作者提供了 C# 語言與 .NET 環境中，從資源管理到多核支援的各種問題解決方案，以及如何避開常見的陷阱。讀者可學習到在多種選項下如何選擇最有效的解決方法，與如何撰寫容易維護及改善的程式碼。Wagner 解釋了以下條目的原因與做法：

- 偏好隱含型別的區域變數 (見方法 1)
- 以內插字串取代 string.Format()(見方法 4)
- 以 delegate 表達 callback(見方法 7)
- 善用 .NET 的資源管理 (見方法 11)
- 定義最少且足夠的泛型限制 (見方法 18)
- 使用執行期型別檢查特化通用演算法 (見方法 19)
- 以 delegate 定義方法的型別參數限制 (見方法 23)
- 以擴充方法規範最小界面合約 (見方法 27)
- 建構 sequence 的可組合 API (見方法 31)
- action、predicate、以及函式與迭代的解耦 (見方法 32)
- 偏好 lambda 表示式的方法 (見方法 38)
- 區別提前與延後執行 (見方法 40)
- 避免抓取高成本的資源 (見方法 41)
- 使用例外回報方法合約失敗 (見方法 45)
- 利用例外慣例的副作用 (見方法 50)

如果你已經是成功的 C# 程式設計師，本書將讓你晉身為頂尖高手。

Bill Wagner 曾經為 Microsoft 設計 .NET 學習教材，是最重要的 C# 專家之一，也是 C# Standards Committee 的成員。他是 Humanitarian Toolbox 的總裁，同時任職於 .NET Foundation Advisory Council 與 Technical Steering Committees。曾經在新創與大企業工作過，負責改善開發程序與團隊。同時是位國際知名作者，著作包括本書的前兩個版本以及 More Effective C#。具有 University of Illinois 的資工學士學位。

Addison-Wesley

碁峰資訊股份有限公司
GOTOP INFORMATION INC.
http://www.gotop.com.tw

PEARSON