

Contents

F# 文件

什麼是 F#

開始使用

安裝 F#

Visual Studio 中的 F#

Visual Studio Code 中的 F#

使用 .NET Core CLI 的 F#

Visual Studio for Mac 中的 F#

F# 的教學課程

教學課程

函式程式設計簡介

一級函式

非同步和並行程式設計

非同步程式設計

型別提供者

建立型別提供者

型別提供者的安全性

型別提供者疑難排解

F# 的新功能

F# 5.0

F# 4.7

F# 4.6

F# 4.5

F# 語言參考

關鍵字參考

符號和運算子參考

算術運算子

布林運算子

位元運算子

[可為 Null 的運算子](#)

[函數](#)

[let 繫結](#)

[do 繫結](#)

[Lambda 運算式: fun 關鍵字](#)

[遞迴函式: rec 關鍵字](#)

[進入點](#)

[外部函式](#)

[內嵌函式](#)

[值](#)

[Null 值](#)

[常值](#)

[F# 類型](#)

[類型推斷](#)

[基本類型](#)

[單位類型](#)

[字串](#)

[插入字串](#)

[Tuple](#)

[F# 集合類型](#)

[清單](#)

[陣列](#)

[序列](#)

[配量](#)

[選項。](#)

[值選項](#)

[結果](#)

[泛型](#)

[自動一般化](#)

[條件約束](#)

[以統計方式解析的型別參數](#)

[記錄](#)

匿名記錄

複製和更新記錄運算式

已區分的聯集

列舉

類型縮寫

類別

結構

可為 Null 的實值型別

繼承

介面

抽象類別

成員

類別中的 let 繫結

類別中的 do 繫結

屬性

索引屬性

方法

建構函式

事件

明確欄位: `val` 關鍵字

類型擴充

參數和引數

運算子多載

彈性類型

委派

物件運算式

轉型和轉換

存取控制

條件運算式: if...then...else

比對運算式

模式比對

現用模式

[迴圈:for...to 運算式](#)

[迴圈:for...in 運算式](#)

[迴圈:while...do 運算式](#)

[判斷提示](#)

[例外狀況處理](#)

[例外狀況類型](#)

[Try...with 運算式](#)

[Try...try...finally 運算式](#)

[Raise 函式](#)

[Failwith 函式](#)

[InvalidArg 函式](#)

[屬性](#)

[資源管理:use 關鍵字](#)

[命名空間](#)

[模組](#)

[匯入宣告:open 關鍵字](#)

[簽名檔](#)

[測量單位](#)

[純文字格式](#)

[XML 文件](#)

[延遲運算式](#)

[計算運算式](#)

[非同步工作流程](#)

[查詢運算式](#)

[程式碼引號](#)

[Fixed 關鍵字](#)

[Byrefs](#)

[參考儲存格](#)

[nameof](#)

[編譯器指示詞](#)

[編譯器選項](#)

[F# Interactive 選項](#)

[原始碼程式行、檔案與路徑識別項](#)

[呼叫端資訊](#)

[詳細語法](#)

[編譯器錯誤與警告](#)

[F# 工具](#)

[F# 互動](#)

[F# 樣式指南](#)

[F# 程式碼格式方針](#)

[F# 編碼慣例](#)

[F# 元件設計指導](#)

[在 Azure 上使用 F#](#)

[以 F 開始使用 Azure Blob 儲存體#](#)

[以 F 開始使用 Azure 檔案儲存體#](#)

[以 F 開始使用 Azure 佇列儲存體#](#)

[以 F 開始使用 Azure 資料表儲存體#](#)

[F# Azure 相依性的套件管理](#)

什麼是 F#

2021/3/5 • [Edit Online](#)

F# 是一種功能性程式設計語言，可讓您輕鬆地撰寫正確且可維護的程式碼。

F# 程式設計主要牽涉到定義型別和函式，這些型別和函式會自動加以推斷。這可讓您的焦點保持在問題領域並操作其資料，而不是程式設計的詳細資料。

```
open System // Gets access to functionality in System namespace.

// Defines a function that takes a name and produces a greeting.
let getGreeting name = $"Hello, {name}! Isn't F# great?"

[<EntryPoint>]
let main args =
    // Defines a list of names
    let names = [ "Don"; "Julia"; "Xi" ]

    // Prints a greeting for each name!
    names
    |> List.map getGreeting
    |> List.iter (fun greeting -> printfn $"{greeting}")

0
```

F# 有許多功能，包括：

- 輕量語法
- 預設為不可變
- 型別推斷和自動一般化
- 一級函式
- 強大的資料類型
- 模式比對
- 非同步程式設計

[F# 語言參考](#)中記載了一組完整的功能。

豐富的資料類型

資料類型(例如 [記錄](#) 和 [差異等位](#))可讓您代表複雜的資料和網域。

```
// Group data with Records
type SuccessfulWithdrawal =
    { Amount: decimal
      Balance: decimal }

type FailedWithdrawal =
    { Amount: decimal
      Balance: decimal
      IsOverdraft: bool }

// Use discriminated unions to represent data of 1 or more forms
type WithdrawalResult =
    | Success of SuccessfulWithdrawal
    | InsufficientFunds of FailedWithdrawal
    | CardExpired of System.DateTime
    | UndisclosedFailure
```

F # 記錄和差異聯集預設為非 null、不變且可比較，因此很容易使用。

搭配函式和模式比對來強制執行正確性

F # 函式在實務上很容易宣告和強大。結合 [模式](#) 比對時，它們可讓您定義編譯器強制執行其正確性的行為。

```
// Returns a WithdrawalResult
let withdrawMoney amount = // Implementation elided

let handleWithdrawal amount =
    let w = withdrawMoney amount

    // The F# compiler enforces accounting for each case!
    match w with
    | Success s -> printfn "Successfully withdrew %f{s.Amount}"
    | InsufficientFunds f -> printfn "Failed: balance is %f{f.Balance}"
    | CardExpired d -> printfn "Failed: card expired on {d}"
    | UndisclosedFailure -> printfn "Failed: unknown :("
```

F # 函式也是第一個類別，這表示它們可以作為參數傳遞，並從其他函式傳回。

定義物件作業的函數

F # 對物件有完整的支援，當您需要 blend 資料和功能時，這些物件是很有用的資料類型。F # 函式是用來操作物件。

```
type Set<'T when 'T: comparison>(elements: seq<'T>) =
    member s.IsEmpty = // Implementation elided
    member s.Contains (value) = // Implementation elided
    member s.Add (value) = // Implementation elided
    // ...
    // Further Implementation elided
    // ...
    interface IEnumerable<'T>
    interface IReadOnlyCollection<'T>

module Set =
    let isEmpty (set: Set<'T>) = set.IsEmpty

    let contains element (set: Set<'T>) = set.Contains(element)

    let add value (set: Set<'T>) = set.Add(value)
```

與其撰寫以物件為導向的程式碼，在 F # 中，您通常會撰寫程式碼，將物件視為另一種資料類型，以便處理函式。

[泛型介面](#)、[物件運算式](#)和合理的[成員](#)用法等功能，在較大的 F # 程式中很常見。

後續步驟

若要深入瞭解一組較大的 F # 功能，請參閱 [F # 導覽](#)。

使用 F 開始

2020/11/2 • [Edit Online](#)

您可以在您的電腦上或在線上開始使用 F #。

開始使用您的電腦

在您的電腦上第一次安裝和使用 F # 有多個指南。您可以使用下表來協助做出決定：

OS	🔗 VISUAL STUDIO	🔗 VISUAL STUDIO CODE	🔗
Windows	開始使用 Visual Studio	開始使用 Visual Studio Code	開始使用 .NET Core CLI
macOS	開始使用 Mac 的 VS	開始使用 Visual Studio Code	開始使用 .NET Core CLI
Linux	N/A	開始使用 Visual Studio Code	開始使用 .NET Core CLI

一般情況下，沒有比其他更好的特定功能。建議您在電腦上嘗試使用 F # 的所有方法，以查看您最喜歡的內容！

線上開始使用

如果您不想在電腦上安裝 F # 和 .NET，您也可以在瀏覽器中開始使用 F #：

- F # 的系結程式簡介是 [透過免費系結器服務託管的 Jupyter 筆記本](#)。不需要註冊！
- [Ncave](#) 複寫是互動式的瀏覽器內複寫，使用 [ncave](#) 將 F # 程式碼轉譯成 JavaScript。查看從 F # 基本概念到完整的影片遊戲的許多範例，全都在您的瀏覽器中執行！

安裝 F#

2020/11/2 • [Edit Online](#)

您可以用多種方式安裝 F #，視您的環境而定。

使用 Visual Studio 安裝 F

1. 如果您是第一次下載 [Visual Studio](#)，它會先安裝 Visual Studio 安裝程式。從安裝程式安裝適當的 Visual Studio 版本。

如果您已經安裝 Visual Studio，請選擇您想要新增 F # 的版本旁的 [修改]。

2. 在 [工作負載] 頁面上，選取 [ASP.NET] 和 [網頁程式開發] 工作負載，其中包含 ASP.NET Core 專案的 F # 和 .net Core 支援。
3. 選擇右下角的 [修改]，以安裝您所選取的所有專案。

然後，您可以在 Visual Studio 安裝程式中選擇 [啟動]，以 F # 開啟 Visual Studio。

使用 Visual Studio Code 安裝 F

1. 確定您已在路徑上安裝並提供 [git](#)。您可以 `git --version` 在命令提示字元中輸入，然後按 **enter**，確認它是否已正確安裝。
2. 安裝 [.NET Core SDK](#) 並 [Visual Studio Code](#)。
3. 選取擴充功能圖示，並搜尋 "Ionide"：

Visual Studio Code 中 F # 支援所需的唯一外掛程式是 [Ionide-fsharp.core](#)。不過，您也可以安裝 [Ionide-假](#) 以取得 [假](#) 的支援和 [Ionide Paket](#)，以取得 [Paket](#) 支援。假和 Paket 是其他 F # 的工具，可分別用來建立專案和管理相依性。

使用 Visual Studio for Mac 安裝 F

依預設，F # 會安裝在 [Visual Studio for Mac](#) 中，不論您選擇哪一種設定。

安裝完成之後，請選擇 [開始 Visual Studio]。您也可以透過 macOS 上的搜尋工具開啟 Visual Studio。

在組建伺服器上安裝 F

如果您是透過 .NET SDK 使用 .NET Core 或 .NET Framework，則只需要在您的組建伺服器上安裝 .NET SDK。它具有您所需的一切。

如果您使用 .NET Framework 而 不是使用 .net SDK，則您必須在 Windows Server 上安裝 [Visual Studio Build Tools SKU](#)。在安裝程式中，選取 [.net 桌面 build tools]，然後選取安裝程式功能表右邊的 F # 編譯器 元件。

開始使用F# Visual Studio

2020/1/8 • [Edit Online](#)

F#Visual Studio 的集成F#開發環境(IDE)支援和視覺化檢視。

若要開始，請確定您已[安裝具有F#支援的 Visual Studio](#)。

建立主控台應用程式

Visual Studio 中最基本的專案之一，就是主控台應用程式。以下是建立伺服器的方式：

1. 開啟 Visual Studio 2019。
2. 在開始視窗中，選擇 [建立新專案]。
3. 在 [建立新專案] 頁面上，F# 從 [語言] 清單中選擇。
4. 選擇 [主控台應用程式 (.Net Core)] 範本，然後選擇 [下一步]。
5. 在 [設定您的新專案] 頁面的 [專案名稱] 方塊中，輸入名稱。接著，選擇 [建立]。

Visual Studio 會建立新F#的專案。您可以在 [方案總管] 視窗中看到它。

撰寫程式碼

讓我們開始撰寫一些程式碼。請確定 `Program.fs` 檔案已開啟，然後以下列內容取代其內容：

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

先前的程式碼範例會定義名為 `square` 的函式，以接受名為 `x` 的輸入，並將其與本身相乘。因為F#使用[型別推斷](#)，所以不需要指定 `x` 的類型。F#編譯器會瞭解乘法的有效類型，並根據呼叫 `square` 的方式，將類型指派給 `x`。如果您將滑鼠停留在 `square` 上，您應該會看到下列內容：

```
val square: x:int -> int
```

這就是所謂的函式型別簽章。它可以讀取如下：「方形是一個函式，它會採用名為 `x` 的整數，並產生一個整數」。編譯器現在提供 `square` 的 `int` 類型；這是因為乘法不是所有類型的泛型，而是一組封閉的類型。如果F#您使用不同的輸入類型(例如 `float`)來呼叫 `square`，編譯器會調整類型簽章。

已定義另一個函式 `main`，該函式會以 `EntryPoint` 屬性裝飾。這個屬性會告知F#編譯器應該在該處啟動程式執行。其遵循與其他C樣式程式語言相同的慣例，其中命令列引數可傳遞給此函式，並傳回整數代碼(通常 `0`)。

它位於進入點函式 `main` 中，您可以使用 `12` 的引數呼叫 `square` 函數。然後F#編譯器會指派要 `int -> int` 的 `square` 類型(也就是接受 `int` 並產生 `int` 的函式)。`printfn` 的呼叫是格式化的列印函式，它會使用格式字串並列印結果(和新的一行)。格式字串與C樣式的程式設計語言類似，其參數(`%d`)會對應至傳遞給它的引數，在此案例中為 `12` 和 `(square 12)`。

執行程式碼

您可以按Ctrl+F5來執行程式碼並查看結果。或者，您可以從最上層功能表列選擇 [Debug] > [啟動但不進行調試]。這會執行程式而不進行偵測。

下列輸出會列印到 Visual Studio 開啟的主控制台視窗：

```
12 squared is: 144!
```

恭喜您！您已在 Visual Studio 中F#建立第一個專案，並F#撰寫一個可計算和列印值的函式，然後執行專案來查看結果。

後續步驟

如果您還沒有這麼做，請參閱的[導覽F#](#)，其中涵蓋了F#語言的一些核心功能。其中提供一些功能的F#總覽，以及您可以複製到 Visual Studio 和執行的大量程式碼範例。

[F# 的教學課程](#)

請參閱

- [F#語言參考](#)
- [型別推斷](#)
- [符號和運算子參考](#)

開始在 Visual Studio Code 中使用 F#

2021/3/5 • [Edit Online](#)

您可以使用 [Ionide 外掛程式](#) 來撰寫 Visual Studio Code 的 F#，以取得絕佳的跨平臺、輕量的整合式開發環境，(使用 IntelliSense 和程式碼重構的 IDE) 體驗。造訪 ionide.io 以深入瞭解外掛程式。

若要開始，請確定您已 [正確安裝 F# 和 Ionide 外掛程式](#)。

使用 Ionide 建立您的第一個專案

若要建立新的 F# 專案，請開啟命令列，並使用 .NET Core CLI 建立新的專案：

```
dotnet new console -lang "F#" -o FirstIonideProject
```

完成之後，請將目錄變更為專案，並開啟 Visual Studio Code：

```
cd FirstIonideProject
code .
```

專案載入 Visual Studio Code 之後，您應該會在視窗的左側看到 F# 方案總管窗格開啟。這表示 Ionide 已成功載入您剛才建立的專案。您可以在此時時間點之前在編輯器中撰寫程式碼，但是一旦發生這種情況，所有專案都已完成載入。

設定 F# interactive

首先，請確定 .NET Core 腳本是您的預設腳本環境：

1. 開啟 [Visual Studio Code 設定] ([程式代碼 > 偏好 > 設定])。
2. 搜尋「F# 腳本」一詞。
3. 按一下顯示 [fsharp.core: 使用 SDK 腳本] 的核取方塊。

這目前是必要的，因為 .NET Framework 型腳本中的某些舊版行為不適用於 .NET Core 腳本，而 Ionide 目前致力於提供回溯相容性。未來，.NET Core 腳本會成為預設值。

撰寫您的第一個指令碼

當您將 Visual Studio Code 設定為使用 .NET Core 腳本之後，請流覽至 Visual Studio Code 中的 Explorer 視圖，然後建立新的檔案。將它命名為 *MyFirstScript.fsx*。

現在，在其中新增下列程式碼：

```
let toPigLatin (word: string) =
    let isVowel (c: char) =
        match c with
        | 'a' | 'e' | 'i' | 'o' | 'u'
        | 'A' | 'E' | 'I' | 'O' | 'U' -> true
        | _ -> false

    if isVowel word.[0] then
        word + "yay"
    else
        word.[1..] + string(word.[0]) + "ay"
```

此函式會將單字轉換成 **Pig 拉丁** 的形式。下一步是使用 F# 互動 (FSI) 來進行評估。

醒目提示整個函數 (應為 11 行長)。反白顯示之後，請按住 **Alt** 鍵，然後按 **Enter** 鍵。您會注意到畫面底部會出現一個終端機視窗，看起來應該像這樣：

```
TERMINAL 1: F# Interactive
-
-
-
- let toPigLatin (word: string) =
-     let isVowel (c: char) =
-         match c with
-         | 'a' | 'e' | 'i' | 'o' | 'u'
-         | 'A' | 'E' | 'I' | 'O' | 'U' -> true
-         | _ -> false
-
-         if isVowel word.[0] then
-             word + "yay"
-         else
-             word.[1..] + string(word.[0]) + "ay";;
-
- val toPigLatin : word:string -> string
-
> |
```

這有三個專案：

1. 它已開始 FSI 程式。
2. 它會將您在 FSI 程式上反白顯示的程式碼傳送給您。
3. FSI 進程會評估您傳送的程式碼。

因為您送出的是函式，所以您現在可以使用 FSI 來呼叫該函式！在互動式視窗中，輸入下列內容：

```
toPigLatin "banana";;
```

您應該會看到下列結果：

```
val it : string = "ananabay"
```

現在，讓我們以母音做為第一個字母來嘗試。輸入以下資訊：

```
toPigLatin "apple";;
```

您應該會看到下列結果：

```
val it : string = "appleay"
```

函數看起來像預期般運作。恭喜，您只是在 Visual Studio Code 撰寫了第一個 F# 函式，並使用 FSI 進行評估！

NOTE

您可能已經注意到，FSI 中的行會以終止 `;;`。這是因為 FSI 可讓您輸入多行。`;;` 最後讓 FSI 知道程式碼完成的時間。

說明程式碼

如果您不確定程式碼實際執行的工作，以下是逐步解說。

如您所見，它 `toPigLatin` 是接受單字作為其輸入的函式，並將它轉換成該單字的 Pig-Latin 標記法。其規則如下所示：

如果單字中的第一個字元開頭為母音，請將 "好耶" 新增至單字的結尾。如果它不是以母音開頭，請將第一個字元移至單字的結尾，並將 "ay" 新增至其中。

您可能已注意到 FSI 中的下列事項：

```
val toPigLatin : word:string -> string
```

這是一種函式 `toPigLatin`，它會採用 `string` 做為輸入的 (`word`，稱為)，並傳回另一個 `string`。這就是所謂的函式 **類型** 簽章，這是 f# 的基本部分，這是瞭解 F# 程式碼的關鍵。如果您將滑鼠停留在 Visual Studio Code 的函式上，也會注意到這一點。

在函式主體中，您會看到兩個不同的部分：

1. 稱為的內部函式，`isVowel` 它 `c` 會藉由檢查是否符合透過 **模式** 比對所提供的其中一個模式，來判斷指定的字元 () 是否為母音：

```
let isVowel (c: char) =  
    match c with  
    | 'a' | 'e' | 'i' | 'o' | 'u'  
    | 'A' | 'E' | 'I' | 'O' | 'U' -> true  
    | _ -> false
```

2. `if..then..else` 檢查第一個字元是否為母音的運算式，並根據第一個字元是否為母音來根據輸入字元來檢查傳回值：

```
if isVowel word.[0] then  
    word + "yay"  
else  
    word.[1..] + string(word.[0]) + "ay"
```

因此的流程 `toPigLatin` 如下：

檢查輸入單字的第一個字元是否為母音。如果是，請將 "好耶" 附加至單字的結尾。否則，請將第一個字元移至單字的結尾，並將 "ay" 新增至該字元。

有一項最後要注意的事項：在 F# 中，沒有從函式傳回的明確指示。這是因為 F# 是以運算式為基礎，而且在函式主體中評估的最後一個運算式會決定該函數的傳回值。因為 `if..then..else` 本身是運算式，所以區塊主體或區塊主體的評估會 `then` `else` 決定函數所傳回的值 `toPigLatin`。

將主控台應用程式轉換成 Pig 的拉丁產生器

本文的前幾節示範了撰寫 F# 程式碼的常見第一個步驟：撰寫初始函式，並以互動方式使用 FSI 來執行它。這就是所謂的複寫導向開發，其中的 **複寫代表**「讀取-評估-列印迴圈」。這是實驗功能的絕佳方法，直到您有一些工作。

複寫導向開發的下一步，是將工作程式碼移到 F# 實作為檔案。然後，F# 編譯器會將它編譯成可執行檔元件。

若要開始，請開啟您稍早使用 .NET Core CLI 建立的 `程式.fs` 檔案。您會發現其中已有一些程式碼。

接著，建立名為新的，`module` `PigLatin` 並將您稍早建立的函式複製 `toPigLatin` 到其中，如下所示：


```

module PigLatin =
    let toPigLatin (word: string) =
        let isVowel (c: char) =
            match c with
            | 'a' | 'e' | 'i' | 'o' | 'u'
            | 'A' | 'E' | 'I' | 'O' | 'U' -> true
            | _ -> false

        if isVowel word.[0] then
            word + "yay"
        else
            word.[1..] + string(word.[0]) + "ay"

```

此模組應該在函式 `main` 和宣告下方 `open System`。在 F# 中，宣告的順序很重要，因此您必須先定義函式，才能在檔案中呼叫該函式。

現在，在函式中 `main`，呼叫引數上的 Pig 拉丁產生器函式：

```

[<EntryPoint>]
let main argv =
    for name in argv do
        let newName = PigLatin.toPigLatin name
        printfn "%{name} in Pig Latin is: {newName}"

    0

```

現在您可以從命令列執行您的主控台應用程式：

```
dotnet run apple banana
```

您會看到它與腳本檔案輸出的結果相同，但是這次是執行中的程式！

針對 Ionide 進行疑難排解

以下是一些您可以針對一些可能遇到的問題進行疑難排解的方法：

1. 若要取得 Ionide 的程式碼編輯功能，您的 F# 檔案必須儲存到磁片，並放在 Visual Studio Code 工作區中開啟的資料夾內。
2. 如果您已對系統進行變更，或已安裝 Visual Studio Code 開啟的 Ionide 必要條件，請重新開機 Visual Studio Code。
3. 如果您的專案目錄中有不正確字元，Ionide 可能無法運作。如果發生這種情況，請重新命名您的專案目錄。
4. 如果沒有任何 Ionide 命令可運作，請檢查您的 [Visual Studio Code 機碼](#) 系結，以查看是否意外覆寫這些命令。
5. 如果您的電腦上的 Ionide 已中斷，且上述各項都未修正您的問題，請嘗試移除 `ionide-fsharp` 電腦上的目錄，然後重新安裝外掛程式套件。
6. 如果專案無法載入 (F# 方案總管將會顯示此)，請在該專案上按一下滑鼠右鍵，然後按一下 [查看詳細資料] 以取得更多診斷資訊。

Ionide 是一種開放原始碼專案，由 F# 群體的成員所建立和維護。請在 [ionide-vscode-Fsharp.core GitHub 存放庫](#) 中回報問題，並歡迎您參與。

您也可以 [在 Ionide Gitter 頻道](#) 中向 Ionide 開發人員和 F# 團體要求進一步的協助。

後續步驟

若要深入瞭解 F# 和語言的功能，請參閱 [f# 導覽](#)。

使用 .NET Core CLI 開始使用 F

2021/3/5 • • [Edit Online](#)

本文涵蓋如何在任何作業系統上開始使用 F #, (Windows、macOS 或 Linux) 與 .NET Core CLI。它會使用由主控台應用程式呼叫的類別庫, 來建立多專案的方案。

先決條件

若要開始, 您必須安裝最新的 [.NET Core SDK](#)。

本文假設您知道如何使用命令列, 並擁有慣用的文字編輯器。如果您還沒有使用它, [Visual Studio Code](#) 是適用於 F # 文字編輯器的絕佳選項。

建立簡單的多專案方案

開啟命令提示字元/終端機, 並使用 `dotnet new` 命令建立名為的新方案檔 `FSNetCore` :

```
dotnet new sln -o FSNetCore
```

執行上述命令之後, 會產生下列目錄結構:

```
FSNetCore
├── FSNetCore.sln
```

撰寫類別庫

將目錄變更為 *FSNetCore*。

使用 `dotnet new` 命令, 在 `src` 資料夾中建立名為 `library` 的類別庫專案。

```
dotnet new classlib -lang "F#" -o src/Library
```

執行上述命令之後, 會產生下列目錄結構:

```
├── FSNetCore
│   ├── FSNetCore.sln
│   └── src
│       ├── Library
│       │   ├── Library.fs
│       │   └── Library.fsproj
```

以下列程式碼取代的內容 `Library.fs` :

```
module Library

open Newtonsoft.Json

let getJsonNetJson value =
    let json = JsonConvert.SerializeObject(value)
    $"I used to be {value} but now I'm {json} thanks to JSON.NET!"
```

將 NuGet 套件上的 Newtonsoft.Json 新增至程式庫專案。

```
dotnet add src/Library/Library.fsproj package Newtonsoft.Json
```

Library FSNetCore 使用 `dotnet sln add` 命令將專案新增至方案：

```
dotnet sln add src/Library/Library.fsproj
```

執行 `dotnet build` 以建立專案。建立時，將會還原未解析的相依性。

撰寫使用類別庫的主控台應用程式

使用 `dotnet new` 命令，在名為 App 的 src 資料夾中建立主控台應用程式。

```
dotnet new console -lang "F#" -o src/App
```

執行上述命令之後，會產生下列目錄結構：

```
├─ FSNetCore
│   └─ FSNetCore.sln
│       └─ src
│           ├── App
│           │   ├── App.fsproj
│           │   ├── Program.fs
│           └─ Library
│               ├── Library.fs
│               └─ Library.fsproj
```

將 `Program.fs` 檔案的內容取代為下列程式碼：

```
open System
open Library

[<EntryPoint>]
let main argv =
    printfn "Nice command-line arguments! Here's what JSON.NET has to say about them:"

    for arg in argv do
        let value = getJsonNetJson arg
        printfn $"{value}"

    0 // return an integer exit code
```

Library 使用 `dotnet 加入參考` 加入專案的參考。

```
dotnet add src/App/App.fsproj reference src/Library/Library.fsproj
```

App 使用命令將專案新增至 FSNetCore 方案 `dotnet sln add`：

```
dotnet sln add src/App/App.fsproj
```

還原 NuGet 相依性，`dotnet restore` 並執行 `dotnet build` 以建立專案。

將目錄變更為 `src/App` 主控台專案並執行專案傳遞 `Hello World` 做為引數：

```
cd src/App
dotnet run Hello World
```

您應該會看見下列結果：

```
Nice command-line arguments! Here's what JSON.NET has to say about them:

I used to be Hello but now I'm ""Hello"" thanks to JSON.NET!
I used to be World but now I'm ""World"" thanks to JSON.NET!
```

後續步驟

接下來，請查看 [F #](#) 的教學課程，以深入瞭解不同的 f # 功能。

開始使用 Visual Studio for Mac 中的 F

2021/3/5 • [Edit Online](#)

Visual Studio for Mac IDE 支援 F # 和 Visual F# 工具。確定您已 [安裝 Visual Studio for Mac](#)。

建立主控台應用程式

Visual Studio for Mac 的其中一個最基本的專案是主控台應用程式。以下說明如何執行這項作業。Visual Studio for Mac 開啟之後：

1. 在 [檔案] 功能表上，指向 [新增方案]。
2. 在 [新增專案] 對話方塊中，主控台應用程式有2個不同的範本。在以 .NET Framework 為目標的其他 > .NET 下會有一個。另一個範本是以 .net core 為目標的 > 應用程式，以 .NET Core 為目標。這兩個範本都應該適用於本文的目的。
3. 在 [主控台應用程式] 下，視需要將 c # 變更為 F #。選擇 [下一步] 按鈕繼續進行！
4. 為您的專案命名，並選擇您想要用於應用程式的選項。請注意，畫面側邊的 [預覽] 窗格會顯示將根據選取的選項建立的目錄結構。
5. 按一下 [建立]。您現在應該會在方案總管中看到 F # 專案。

撰寫程式碼

先撰寫一些程式碼，讓我們開始吧。請確定檔案 `Program.fs` 已開啟，然後將其內容取代為下列內容：

```
module HelloSquare

let square x = x * x

[<EntryPoint>]
let main argv =
    printfn "%d squared is: %d!" 12 (square 12)
    0 // Return an integer exit code
```

在之前的程式碼範例中，已 `square` 定義函式，該函式會接受名為 `x` 的輸入，並將其本身相乘。因為 F # 使用 [型別推斷](#)，所以 `x` 不需要指定的型別。F # 編譯器瞭解乘法有效的型別，而且會根據呼叫方式將型別指派給 `x` `square`。如果您將滑鼠停留在上方 `square`，您應該會看到下列內容：

```
val square: x:int -> int
```

這就是所謂的函式類型簽章。它可以像這樣讀取：「方形是一個函式，它會採用名為 `x` 的整數，並產生整數」。請注意，編譯器 `square` `int` 會提供目前的型別，這是因為在 *所有* 類型中的乘法不是泛型，而是在一組封閉的型別中是泛型。此時會挑選 F # 編譯器 `int`，但如果您 `square` 使用不同的輸入類型（例如）來呼叫，則會調整型別簽章 `float`。

另外也定義了另一個函式，`main` 它是以 `EntryPoint` 屬性裝飾，告訴 F # 編譯器，程式執行應該在此開始。它會遵循與其他 [C 樣式程式設計語言](#) 相同的慣例，其中命令列引數可傳遞給此函式，而整數程式碼則會傳回（通常 `0`）。

在這個函式中，我們使用的 `square` 引數來呼叫函式 `12`。然後，F # 編譯器會將的型別指派為 `square`

`int -> int` (也就是採用 `int` 並產生) 的函式 `int`。的呼叫 `printfn` 是格式化的列印函式，它會使用格式字串，類似於 C 樣式的程式設計語言、與格式字串中指定的參數對應的參數，然後列印結果和新的行。

執行您的程式碼

您可以從最上層功能表中按一下 [執行]，然後在不進行偵錯工具的情況下啟動，來執行程式碼並查看結果。這會執行程式而不進行任何偵測，並可讓您查看結果。

您現在應該會在主控台視窗中看到下列輸出 Visual Studio for Mac 快顯視窗：

```
12 squared is 144!
```

恭喜！您已在 Visual Studio for Mac 中建立您的第一個 F# 專案、撰寫 F# 函式以列印呼叫該函式的結果，並執行專案以查看某些結果。

使用 F# 互動

在 Visual Studio for Mac 中，Visual F# 工具的其中一個最佳功能是 F# 互動視窗。它可讓您將程式碼傳送至進程，您可以在其中呼叫該程式碼，並以互動方式查看結果。

若要開始使用，請反白顯示 `square` 您程式碼中所定義的函式。接著，按一下最上層功能表中的 [編輯]。接下來選取 [傳送選取專案至 F# 互動]。這會在 F# 互動視窗中執行程式碼。或者，您也可以在選取專案上按一下滑鼠右鍵，然後選擇 [傳送選取專案] F# 互動。您應該會看到 F# 互動視窗顯示如下：

```
>  
  
val square : x:int -> int  
  
>
```

這會顯示函式的相同函式簽章，您稍早在此函式上方看到該簽章 `square`。因為 `square` 現在已在 F# 互動進程中定義，所以您可以使用不同的值來呼叫它：

```
> square 12;;  
val it : int = 144  
>square 13;;  
val it : int = 169
```

這會執行函數、將結果系結至新的名稱 `it`，並顯示的類型和值 `it`。請注意，您必須使用來終止每一行 `;;`。這是在函式呼叫完成時 F# 互動知道的方式。您也可以在 F# 互動中定義新函數：

```
> let isOdd x = x % 2 <> 0;;  
  
val isOdd : x:int -> bool  
  
> isOdd 12;;  
val it : bool = false
```

上述定義了新的函 `isOdd` 式，它會取得 `int` 並檢查是否為奇數！您可以呼叫此函式，以查看傳回的內容有不同的輸入。您可以呼叫函式呼叫內的函數：

```
> isOdd (square 15);;  
val it : bool = true
```

您也可以使用 [管道轉寄運算子](#)，將值管線至兩個函數：

```
> 15 |> square |> isOdd;;  
val it : bool = true
```

順向運算子和其他的教學課程將在稍後的教學課程中討論。

這只是您可以運用 F# 互動的內容。若要深入瞭解，請參閱 [使用 F# 的互動式程式設計](#)。

後續步驟

如果您尚未這麼做，請查看 [f#](#) 的教學課程，其中包含 f# 語言的一些核心功能。它將概述 F# 的一些功能，並提供可複製到 Visual Studio for Mac 並執行的豐富程式碼範例。另外還有一些絕佳的外部資源可供您使用，請展示在 [F# 指南](#) 中。

另請參閱

- [F# 指南](#)
- [F# 的教學課程](#)
- [F# 語言參考](#)
- [型別推斷](#)
- [符號和運算子參考](#)

F 教學課程

2020/11/2 • [Edit Online](#)

瞭解 F # 的最佳方式是讀取和寫入 F # 程式碼。本文將逐步解說 F # 語言的一些重要功能，並提供您一些可在電腦上執行的程式碼片段。若要瞭解如何設定開發環境，請參閱 [消費者入門](#)。

F # 中有兩個主要概念：函數和類型。本教學課程將著重在這兩個概念中的語言功能。

線上執行程式碼

如果您的電腦上沒有安裝 F #，您可以在 [WebAssembly 上使用 Try F #](#)，在您的瀏覽器中執行所有範例。Ncave 是 F # 的方言，可直接在您的瀏覽器中執行。若要查看複寫中接下來的範例，請參閱 Ncave 複寫的左側功能表列中的範例，> 瞭解 F # > 導覽。

函數和模組

任何 F # 程式最基本的 **部分都是** 組織成 **模組** 的函式。函式會對輸入執行工作以產生輸出，並在 **模組** 下組織，這些是您在 F # 中將專案分組的主要方式。它們是使用系結定義 `let` 的，它會為函式命名並定義其引數。

```
module BasicFunctions =

    /// You use 'let' to define a function. This one accepts an integer argument and returns an integer.
    /// Parentheses are optional for function arguments, except for when you use an explicit type
    annotation.
    let sampleFunction1 x = x*x + 3

    /// Apply the function, naming the function return result using 'let'.
    /// The variable type is inferred from the function return type.
    let result1 = sampleFunction1 4573

    // This line uses '%d' to print the result as an integer. This is type-safe.
    // If 'result1' were not of type 'int', then the line would fail to compile.
    printfn $"The result of squaring the integer 4573 and adding 3 is %d{result1}"

    /// When needed, annotate the type of a parameter name using '(argument:type)'. Parentheses are
    required.
    let sampleFunction2 (x:int) = 2*x*x - x/5 + 3

    let result2 = sampleFunction2 (7 + 4)
    printfn $"The result of applying the 2nd sample function to (7 + 4) is %d{result2}"

    /// Conditionals use if/then/elif/else.
    ///
    /// Note that F# uses white space indentation-aware syntax, similar to languages like Python.
    let sampleFunction3 x =
        if x < 100.0 then
            2.0*x*x - x/5.0 + 3.0
        else
            2.0*x*x + x/5.0 - 37.0

    let result3 = sampleFunction3 (6.5 + 4.5)

    // This line uses '%f' to print the result as a float. As with '%d' above, this is type-safe.
    printfn $"The result of applying the 3rd sample function to (6.5 + 4.5) is %f{result3}"
```

`let` 系結也是您將值系結至名稱的方式，類似于其他語言的變數。`let` 系結預設為 **不可變**，這表示一旦值或函數系結至名稱之後，就無法就地變更。這與其他語言中的變數不同，**這是可變動的**，這表示它們的值可以在任

何時時間點變更。如果您需要可變的系結，您可以使用 `let mutable ...` 語法。

```
module Immutability =

    /// Binding a value to a name via 'let' makes it immutable.
    ///
    /// The second line of code compiles, but 'number' from that point onward will shadow the previous
    definition.
    /// There is no way to access the previous definition of 'number' due to shadowing.
    let number = 2
    // let number = 3

    /// A mutable binding. This is required to be able to mutate the value of 'otherNumber'.
    let mutable otherNumber = 2

    printfn $" 'otherNumber' is {otherNumber}"

    // When mutating a value, use '<-' to assign a new value.
    //
    // Note that '=' is not the same as this. '=' is used to test equality.
    otherNumber <- otherNumber + 1

    printfn $" 'otherNumber' changed to be {otherNumber}"
```

數位、布林值和字串

作為 .NET 語言, F # 支援存在於 .NET 中的相同基礎基本 [類型](#)。

以下是如何以 F # 表示不同的數位類型：

```
module IntegersAndNumbers =

    /// This is a sample integer.
    let sampleInteger = 176

    /// This is a sample floating point number.
    let sampleDouble = 4.1

    /// This computed a new number by some arithmetic. Numeric types are converted using
    /// functions 'int', 'double' and so on.
    let sampleInteger2 = (sampleInteger/4 + 5 - 7) * 4 + int sampleDouble

    /// This is a list of the numbers from 0 to 99.
    let sampleNumbers = [ 0 .. 99 ]

    /// This is a list of all tuples containing all the numbers from 0 to 99 and their squares.
    let sampleTableOfSquares = [ for i in 0 .. 99 -> (i, i*i) ]

    // The next line prints a list that includes tuples, using an interpolated string.
    printfn $"The table of squares from 0 to 99 is:\n{sampleTableOfSquares}"
```

以下是布林值和執行基本條件式邏輯的樣子：

```

module Booleans =

    /// Booleans values are 'true' and 'false'.
    let boolean1 = true
    let boolean2 = false

    /// Operators on booleans are 'not', '&&' and '||'.
    let boolean3 = not boolean1 && (boolean2 || false)

    // This line uses '%b' to print a boolean value. This is type-safe.
    printfn $"The expression 'not boolean1 && (boolean2 || false)' is %b{boolean3}"

```

以下是基本 [字串](#) 操作的樣子：

```

module StringManipulation =

    /// Strings use double quotes.
    let string1 = "Hello"
    let string2 = "world"

    /// Strings can also use @ to create a verbatim string literal.
    /// This will ignore escape characters such as '\', '\n', '\t', etc.
    let string3 = @"C:\Program Files\"

    /// String literals can also use triple-quotes.
    let string4 = """The computer said "hello world" when I told it to!"""

    /// String concatenation is normally done with the '+' operator.
    let helloWorld = string1 + " " + string2

    // This line uses '%s' to print a string value. This is type-safe.
    printfn "%s" helloWorld

    /// Substrings use the indexer notation. This line extracts the first 7 characters as a substring.
    /// Note that like many languages, Strings are zero-indexed in F#.
    let substring = helloWorld.[0..6]
    printfn $"{substring}"

```

Tuple

[元組](#) 在 F# 中是很大的交易。它們是未命名但排序值的群組，可視為值本身。將它們視為從其他值匯總的值。它們有許多用途，例如方便從函式傳回多個值，或將值分組以進行某些特定的便利性。

```

module Tuples =

    /// A simple tuple of integers.
    let tuple1 = (1, 2, 3)

    /// A function that swaps the order of two values in a tuple.
    ///
    /// F# Type Inference will automatically generalize the function to have a generic type,
    /// meaning that it will work with any type.
    let swapElems (a, b) = (b, a)

    printfn $"The result of swapping (1, 2) is {(swapElems (1,2))}"

    /// A tuple consisting of an integer, a string,
    /// and a double-precision floating point number.
    let tuple2 = (1, "fred", 3.1415)

    printfn $"tuple1: {tuple1}\ttuple2: {tuple2}"

```

您也可以建立 `struct` 元組。這些也會與 c # 7/Visual Basic 15 個元組(也就是元組)完全交互操作 `struct` :

```
/// Tuples are normally objects, but they can also be represented as structs.
///
/// These interoperate completely with structs in C# and Visual Basic.NET; however,
/// struct tuples are not implicitly convertible with object tuples (often called reference tuples).
///
/// The second line below will fail to compile because of this. Uncomment it to see what happens.
let sampleStructTuple = struct (1, 2)
//let thisWillNotCompile: (int*int) = struct (1, 2)

// Although you can
let convertFromStructTuple (struct(a, b)) = (a, b)
let convertToStructTuple (a, b) = struct(a, b)

printfn $"Struct Tuple: {sampleStructTuple}\nReference tuple made from the Struct Tuple: {(sampleStructTuple
|> convertFromStructTuple)}"
```

請務必注意，因為 `struct` 元組是實值型別，所以無法隱含地轉換成參考元組，反之亦然。您必須在參考和結構元組之間明確地轉換。

管線和組合

`|>` 在 F # 中處理資料時，會廣泛使用管道運算子(例如)。這些運算子是功能，可讓您以彈性的方式建立函式的「管線」。下列範例會逐步解說如何利用這些運算子來建立簡單的功能管線：

```

module PipelinesAndComposition =

    /// Squares a value.
    let square x = x * x

    /// Adds 1 to a value.
    let addOne x = x + 1

    /// Tests if an integer value is odd via modulo.
    let isOdd x = x % 2 <> 0

    /// A list of 5 numbers. More on lists later.
    let numbers = [ 1; 2; 3; 4; 5 ]

    /// Given a list of integers, it filters out the even numbers,
    /// squares the resulting odds, and adds 1 to the squared odds.
    let squareOddValuesAndAddOne values =
        let odds = List.filter isOdd values
        let squares = List.map square odds
        let result = List.map addOne squares
        result

    printfn $"processing {numbers} through 'squareOddValuesAndAddOne' produces: {squareOddValuesAndAddOne
numbers}"

    /// A shorter way to write 'squareOddValuesAndAddOne' is to nest each
    /// sub-result into the function calls themselves.
    ///
    /// This makes the function much shorter, but it's difficult to see the
    /// order in which the data is processed.
    let squareOddValuesAndAddOneNested values =
        List.map addOne (List.map square (List.filter isOdd values))

    printfn $"processing {numbers} through 'squareOddValuesAndAddOneNested' produces:
{squareOddValuesAndAddOneNested numbers}"

    /// A preferred way to write 'squareOddValuesAndAddOne' is to use F# pipe operators.
    /// This allows you to avoid creating intermediate results, but is much more readable
    /// than nesting function calls like 'squareOddValuesAndAddOneNested'
    let squareOddValuesAndAddOnePipeline values =
        values
        |> List.filter isOdd
        |> List.map square
        |> List.map addOne

    printfn $"processing {numbers} through 'squareOddValuesAndAddOnePipeline' produces:
{squareOddValuesAndAddOnePipeline numbers}"

    /// You can shorten 'squareOddValuesAndAddOnePipeline' by moving the second `List.map` call
    /// into the first, using a Lambda Function.
    ///
    /// Note that pipelines are also being used inside the lambda function. F# pipe operators
    /// can be used for single values as well. This makes them very powerful for processing data.
    let squareOddValuesAndAddOneShorterPipeline values =
        values
        |> List.filter isOdd
        |> List.map(fun x -> x |> square |> addOne)

    printfn $"processing {numbers} through 'squareOddValuesAndAddOneShorterPipeline' produces:
{squareOddValuesAndAddOneShorterPipeline numbers}"

```

先前的範例使用 F# 的許多功能，包括清單處理函式、第一級函式和 [部分應用程式](#)。雖然對每個概念的深入瞭解都可能更簡單，但是在建立管線時，可以清楚地使用函式來處理資料。

清單、陣列和序列

清單、陣列和序列是 F # 核心程式庫中的三個主要集合類型。

清單 是相同類型之專案的已排序、不可變的集合。它們是單一連結的清單，這表示它們是用來進行列舉，但如果很大，則會有不佳的隨機存取和串連選項。這與其他熱門語言中的清單相較之下，通常不會使用單一連結的清單來表示清單。

```
module Lists =

    /// Lists are defined using [ ... ]. This is an empty list.
    let list1 = [ ]

    /// This is a list with 3 elements. ';' is used to separate elements on the same line.
    let list2 = [ 1; 2; 3 ]

    /// You can also separate elements by placing them on their own lines.
    let list3 = [
        1
        2
        3
    ]

    /// This is a list of integers from 1 to 1000
    let numberList = [ 1 .. 1000 ]

    /// Lists can also be generated by computations. This is a list containing
    /// all the days of the year.
    let daysList =
        [ for month in 1 .. 12 do
            for day in 1 .. System.DateTime.DaysInMonth(2017, month) do
                yield System.DateTime(2017, month, day) ]

    // Print the first 5 elements of 'daysList' using 'List.take'.
    printfn $"The first 5 days of 2017 are: {daysList |> List.take 5}"

    /// Computations can include conditionals. This is a list containing the tuples
    /// which are the coordinates of the black squares on a chess board.
    let blackSquares =
        [ for i in 0 .. 7 do
            for j in 0 .. 7 do
                if (i+j) % 2 = 1 then
                    yield (i, j) ]

    /// Lists can be transformed using 'List.map' and other functional programming combinators.
    /// This definition produces a new list by squaring the numbers in numberList, using the pipeline
    /// operator to pass an argument to List.map.
    let squares =
        numberList
        |> List.map (fun x -> x*x)

    /// There are many other list combinations. The following computes the sum of the squares of the
    /// numbers divisible by 3.
    let sumOfSquares =
        numberList
        |> List.filter (fun x -> x % 3 = 0)
        |> List.sumBy (fun x -> x * x)

    printfn $"The sum of the squares of numbers up to 1000 that are divisible by 3 is: %d{sumOfSquares}"
```

陣列 是具有相同類型之專案的固定大小、可變動集合。它們支援快速隨機存取專案，而且比 F # 清單更快，因為它們只是連續的記憶體區塊。

```

module Arrays =

    /// This is The empty array. Note that the syntax is similar to that of Lists, but uses `[| ... |]`
    instead.
    let array1 = [| |]

    /// Arrays are specified using the same range of constructs as lists.
    let array2 = [| "hello"; "world"; "and"; "hello"; "world"; "again" |]

    /// This is an array of numbers from 1 to 1000.
    let array3 = [| 1 .. 1000 |]

    /// This is an array containing only the words "hello" and "world".
    let array4 =
        [| for word in array2 do
            if word.Contains("l") then
                yield word |]

    /// This is an array initialized by index and containing the even numbers from 0 to 2000.
    let evenNumbers = Array.init 1001 (fun n -> n * 2)

    /// Sub-arrays are extracted using slicing notation.
    let evenNumbersSlice = evenNumbers.[0..500]

    /// You can loop over arrays and lists using 'for' loops.
    for word in array4 do
        printfn $"word: {word}"

    // You can modify the contents of an array element by using the left arrow assignment operator.
    //
    // To learn more about this operator, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/values/index#mutable-variables
    array2.[1] <- "WORLD!"

    /// You can transform arrays using 'Array.map' and other functional programming operations.
    /// The following calculates the sum of the lengths of the words that start with 'h'.
    let sumOfLengthsOfWords =
        array2
        |> Array.filter (fun x -> x.StartsWith "h")
        |> Array.sumBy (fun x -> x.Length)

    printfn $"The sum of the lengths of the words in Array 2 is: %d{sumOfLengthsOfWords}"

```

序列 是元素的邏輯系列，全都是相同的類型。這些是比清單和陣列更一般的型別，可將您的「view」成任何邏輯系列元素。它們也有可能是 *延遲* 的，這表示只有在需要時，才可以計算元素。

```

module Sequences =

    /// This is the empty sequence.
    let seq1 = Seq.empty

    /// This a sequence of values.
    let seq2 = seq { yield "hello"; yield "world"; yield "and"; yield "hello"; yield "world"; yield "again"
    }

    /// This is an on-demand sequence from 1 to 1000.
    let numbersSeq = seq { 1 .. 1000 }

    /// This is a sequence producing the words "hello" and "world"
    let seq3 =
        seq { for word in seq2 do
                if word.Contains("l") then
                    yield word }

    /// This sequence producing the even numbers up to 2000.
    let evenNumbers = Seq.init 1001 (fun n -> n * 2)

    let rnd = System.Random()

    /// This is an infinite sequence which is a random walk.
    /// This example uses yield! to return each element of a subsequence.
    let rec randomWalk x =
        seq { yield x
              yield! randomWalk (x + rnd.NextDouble() - 0.5) }

    /// This example shows the first 100 elements of the random walk.
    let first100ValuesOfRandomWalk =
        randomWalk 5.0
        |> Seq.truncate 100
        |> Seq.toList

    printfn $"First 100 elements of a random walk: {first100ValuesOfRandomWalk}"

```

遞迴函式

處理集合或元素的順序通常是以 F # 中的 [遞迴](#) 來完成。雖然 F # 支援迴圈和命令式程式設計，但最好是遞迴，因為這樣可以更輕鬆地確保正確性。

NOTE

下列範例會利用運算式的模式比對 `match`。本文稍後將說明此基礎結構。

```

module RecursiveFunctions =

    /// This example shows a recursive function that computes the factorial of an
    /// integer. It uses 'let rec' to define a recursive function.
    let rec factorial n =
        if n = 0 then 1 else n * factorial (n-1)

    printfn $"Factorial of 6 is: %{factorial 6}"

    /// Computes the greatest common factor of two integers.
    ///
    /// Since all of the recursive calls are tail calls,
    /// the compiler will turn the function into a loop,
    /// which improves performance and reduces memory consumption.
    let rec greatestCommonFactor a b =
        if a = 0 then b
        elif a < b then greatestCommonFactor a (b - a)
        else greatestCommonFactor (a - b) b

    printfn $"The Greatest Common Factor of 300 and 620 is %{greatestCommonFactor 300 620}"

    /// This example computes the sum of a list of integers using recursion.
    let rec sumList xs =
        match xs with
        | [] -> 0
        | y::ys -> y + sumList ys

    /// This makes 'sumList' tail recursive, using a helper function with a result accumulator.
    let rec private sumListTailRecHelper accumulator xs =
        match xs with
        | [] -> accumulator
        | y::ys -> sumListTailRecHelper (accumulator+y) ys

    /// This invokes the tail recursive helper function, providing '0' as a seed accumulator.
    /// An approach like this is common in F#.
    let sumListTailRecursive xs = sumListTailRecHelper 0 xs

    let oneThroughTen = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

    printfn $"The sum 1-10 is %{sumListTailRecursive oneThroughTen}"

```

F # 也有 Tail 呼叫優化的完整支援，這是將遞迴呼叫優化，使其與迴圈結構一樣快的方式。

記錄和區分聯集類型

記錄和等位類型是 F # 程式碼中使用的兩個基本資料類型，通常是在 F # 程式中表示資料的最佳方式。雖然這會讓它們類似於其他語言的類別，但其中一個主要差異在於它們具有結構化相等的語法。這表示它們是「原生」可比較且相等的相等，只要檢查其中一個是否等於另一個。

[記錄](#) 是已命名值的匯總，具有選擇性成員 (例如) 方法。如果您熟悉 c # 或 JAVA，則這些應該類似於 Poco 或 Pojo，只是結構相等且較不會發生。


```

module RecordTypes =

    /// This example shows how to define a new record type.
    type ContactCard =
        { Name      : string
          Phone     : string
          Verified  : bool }

    /// This example shows how to instantiate a record type.
    let contact1 =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false }

    /// You can also do this on the same line with ';' separators.
    let contactOnSameLine = { Name = "Alf"; Phone = "(206) 555-0157"; Verified = false }

    /// This example shows how to use "copy-and-update" on record values. It creates
    /// a new record value that is a copy of contact1, but has different values for
    /// the 'Phone' and 'Verified' fields.
    ///
    /// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/copy-and-update-record-expressions
    let contact2 =
        { contact1 with
          Phone = "(206) 555-0112"
          Verified = true }

    /// This example shows how to write a function that processes a record value.
    /// It converts a 'ContactCard' object to a string.
    let showContactCard (c: ContactCard) =
        c.Name + " Phone: " + c.Phone + (if not c.Verified then " (unverified)" else "")

    printfn $"Alf's Contact Card: {showContactCard contact1}"

    /// This is an example of a Record with a member.
    type ContactCardAlternate =
        { Name      : string
          Phone     : string
          Address   : string
          Verified  : bool }

    /// Members can implement object-oriented members.
    member this.PrintedContactCard =
        this.Name + " Phone: " + this.Phone + (if not this.Verified then " (unverified)" else "") +
        this.Address

    let contactAlternate =
        { Name = "Alf"
          Phone = "(206) 555-0157"
          Verified = false
          Address = "111 Alf Street" }

    // Members are accessed via the '.' operator on an instantiated type.
    printfn $"Alf's alternate contact card is {contactAlternate.PrintedContactCard}"

```

您也可以將記錄表示為結構。這是使用屬性來完成的 `[<Struct>]` :

```
/// Records can also be represented as structs via the 'Struct' attribute.  
/// This is helpful in situations where the performance of structs outweighs  
/// the flexibility of reference types.  
[<Struct>]  
type ContactCardStruct =  
    { Name      : string  
      Phone     : string  
      Verified  : bool }
```

差異聯集 (du) 是可能是數個名稱的表單或案例的值。儲存在類型中的資料可以是數個相異值的其中一個。

```

module DiscriminatedUnions =

    /// The following represents the suit of a playing card.
    type Suit =
        | Hearts
        | Clubs
        | Diamonds
        | Spades

    /// A Discriminated Union can also be used to represent the rank of a playing card.
    type Rank =
        /// Represents the rank of cards 2 .. 10
        | Value of int
        | Ace
        | King
        | Queen
        | Jack

    /// Discriminated Unions can also implement object-oriented members.
    static member GetAllRanks() =
        [ yield Ace
          for i in 2 .. 10 do yield Value i
          yield Jack
          yield Queen
          yield King ]

    /// This is a record type that combines a Suit and a Rank.
    /// It's common to use both Records and Discriminated Unions when representing data.
    type Card = { Suit: Suit; Rank: Rank }

    /// This computes a list representing all the cards in the deck.
    let fullDeck =
        [ for suit in [ Hearts; Diamonds; Clubs; Spades] do
          for rank in Rank.GetAllRanks() do
            yield { Suit=suit; Rank=rank } ]

    /// This example converts a 'Card' object to a string.
    let showPlayingCard (c: Card) =
        let rankString =
            match c.Rank with
            | Ace -> "Ace"
            | King -> "King"
            | Queen -> "Queen"
            | Jack -> "Jack"
            | Value n -> string n
        let suitString =
            match c.Suit with
            | Clubs -> "clubs"
            | Diamonds -> "diamonds"
            | Spades -> "spades"
            | Hearts -> "hearts"
        rankString + " of " + suitString

    /// This example prints all the cards in a playing deck.
    let printAllCards() =
        for card in fullDeck do
            printfn "${showPlayingCard card}"

```

您也可以使用 Du 作為 **單一案例的差異聯集**，以協助您在基本類型上進行領域模型化。通常會使用時間、字串和其他基本型別來代表某個東西，因此會提供特定意義。不過，僅使用資料的基本標記法可能會導致錯誤地指派不正確的值！將每一種類型的資訊表示為不同的單一案例聯集，可以在此案例中強制執行正確性。

```
// Single-case DUs are often used for domain modeling. This can buy you extra type safety
// over primitive types such as strings and ints.
//
// Single-case DUs cannot be implicitly converted to or from the type they wrap.
// For example, a function which takes in an Address cannot accept a string as that input,
// or vice versa.
type Address = Address of string
type Name = Name of string
type SSN = SSN of int

// You can easily instantiate a single-case DU as follows.
let address = Address "111 Alf Way"
let name = Name "Alf"
let ssn = SSN 1234567890

/// When you need the value, you can unwrap the underlying value with a simple function.
let unwrapAddress (Address a) = a
let unwrapName (Name n) = n
let unwrapSSN (SSN s) = s

// Printing single-case DUs is simple with unwrapping functions.
printfn $"Address: {address |> unwrapAddress}, Name: {name |> unwrapName}, and SSN: {ssn |> unwrapSSN}"
```

如上一個範例所示，若要取得單一案例差異聯集的基礎值，您必須明確地將它解除包裝。

此外，Du 也支援遞迴定義，可讓您輕鬆地表示樹狀結構和原本遞迴的資料。例如，以下是您可以使用和函式來表示二進位搜尋樹狀結構的方式 `exists` `insert` 。

```
/// Discriminated Unions also support recursive definitions.
///
/// This represents a Binary Search Tree, with one case being the Empty tree,
/// and the other being a Node with a value and two subtrees.
type BST<'T> =
    | Empty
    | Node of value:'T * left: BST<'T> * right: BST<'T>

/// Check if an item exists in the binary search tree.
/// Searches recursively using Pattern Matching. Returns true if it exists; otherwise, false.
let rec exists item bst =
    match bst with
    | Empty -> false
    | Node (x, left, right) ->
        if item = x then true
        elif item < x then (exists item left) // Check the left subtree.
        else (exists item right) // Check the right subtree.

/// Inserts an item in the Binary Search Tree.
/// Finds the place to insert recursively using Pattern Matching, then inserts a new node.
/// If the item is already present, it does not insert anything.
let rec insert item bst =
    match bst with
    | Empty -> Node(item, Empty, Empty)
    | Node(x, left, right) as node ->
        if item = x then node // No need to insert, it already exists; return the node.
        elif item < x then Node(x, insert item left, right) // Call into left subtree.
        else Node(x, left, insert item right) // Call into right subtree.
```

由於 Du 可讓您以資料類型代表樹狀結構的遞迴結構，因此在此遞迴結構上運作相當簡單，而且保證正確性。在模式比對中也支援，如下所示。

此外，您可以 `struct` 使用屬性將 du 表示為 `s [<Struct>]`：

```
/// Discriminated Unions can also be represented as structs via the 'Struct' attribute.  
/// This is helpful in situations where the performance of structs outweighs  
/// the flexibility of reference types.  
///  
/// However, there are two important things to know when doing this:  
///     1. A struct DU cannot be recursively-defined.  
///     2. A struct DU must have unique names for each of its cases.  
[<Struct>]  
type Shape =  
    | Circle of radius: float  
    | Square of side: float  
    | Triangle of height: float * width: float
```

不過，在這種情況下，有兩個重要事項要牢記在心：

1. 無法遞迴定義結構 DU。
2. 結構 DU 的每個案例都必須有唯一的名稱。

無法遵循上述動作將會導致編譯錯誤。

模式比對

模式比對是 f# 語言功能，可讓您在 f# 類型上操作的正確性。在上述範例中，您可能已注意到很多

`match x with ...` 語法。此結構可讓編譯器瞭解資料類型的「圖形」，以強制您在使用資料類型時，透過所謂的完整模式比對來考慮所有可能的情況。這在正確性方面相當強大，而且可以用來「增益」，這通常是執行時間在編譯時間方面的顧慮。

```

module PatternMatching =

  /// A record for a person's first and last name
  type Person = {
    First : string
    Last  : string
  }

  /// A Discriminated Union of 3 different kinds of employees
  type Employee =
    | Engineer of engineer: Person
    | Manager of manager: Person * reports: List<Employee>
    | Executive of executive: Person * reports: List<Employee> * assistant: Employee

  /// Count everyone underneath the employee in the management hierarchy,
  /// including the employee. The matches bind names to the properties
  /// of the cases so that those names can be used inside the match branches.
  /// Note that the names used for binding do not need to be the same as the
  /// names given in the DU definition above.
  let rec countReports(emp : Employee) =
    1 + match emp with
    | Engineer(person) ->
      0
    | Manager(person, reports) ->
      reports |> List.sumBy countReports
    | Executive(person, reports, assistant) ->
      (reports |> List.sumBy countReports) + countReports assistant

  /// Find all managers/executives named "Dave" who do not have any reports.
  /// This uses the 'function' shorthand to as a lambda expression.
  let findDaveWithOpenPosition(emps : List<Employee>) =
    emps
    |> List.filter(function
      | Manager({First = "Dave"}, []) -> true // [] matches an empty list.
      | Executive({First = "Dave"}, [], _) -> true
      | _ -> false) // '_' is a wildcard pattern that matches anything.
      // This handles the "or else" case.

```

您可能已經注意到，這是使用 `_` 模式。這就是所謂的 **萬用字元模式**，也就是說「我不在意什麼東西」的方法。雖然很方便，但您可以不小心略過詳盡的模式比對，如果您不小心使用，則無法再從編譯時期大規模地規範獲益 `_`。當您在模式比對時不在意某些分解型別的片段，或是當您在模式比對運算式中列舉所有有意義的案例時，最好使用它。

在下列範例中，`_` 當剖析作業失敗時，會使用案例。

```

open System

/// You can also use the shorthand function construct for pattern matching,
/// which is useful when you're writing functions which make use of Partial Application.
let private parseHelper (f: string -> bool * 'T) = f >> function
    | (true, item) -> Some item
    | (false, _) -> None

let parseDateTimeOffset = parseHelper DateTimeOffset.TryParse

let result = parseDateTimeOffset "1970-01-01"
match result with
| Some dto -> printfn "It parsed!"
| None -> printfn "It didn't parse!"

// Define some more functions which parse with the helper function.
let parseInt = parseHelper Int32.TryParse
let parseDouble = parseHelper Double.TryParse
let parseTimeSpan = parseHelper TimeSpan.TryParse

```

現用模式 是另一種功能強大的結構，可搭配模式比對使用。它們可讓您將輸入資料分割成自訂表單，並在模式比對呼叫位置分解它們。它們也可以參數化，因此可讓您將資料分割定義為函數。擴充先前的範例以支援現用模式看起來像這樣：

```

// Active Patterns are another powerful construct to use with pattern matching.
// They allow you to partition input data into custom forms, decomposing them at the pattern match call
// site.
//
// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/active-patterns
let (|Int|_|) = parseInt
let (|Double|_|) = parseDouble
let (|Date|_|) = parseDateTimeOffset
let (|TimeSpan|_|) = parseTimeSpan

/// Pattern Matching via 'function' keyword and Active Patterns often looks like this.
let printParseResult = function
    | Int x -> printfn $"%d{x}"
    | Double x -> printfn $"%f{x}"
    | Date d -> printfn $"%O{d}"
    | TimeSpan t -> printfn $"%O{t}"
    | _ -> printfn "Nothing was parse-able!"

// Call the printer with some different values to parse.
printParseResult "12"
printParseResult "12.045"
printParseResult "12/28/2016"
printParseResult "9:01PM"
printParseResult "banana!"

```

選用類型

差異聯集類型的一個特殊案例是選項類型，因此它是 F# 核心程式庫的一部分。

選項類型 是代表兩種情況之一的類型：值，或完全沒有。在任何情況下，其值可能會或可能不會由特定作業產生。然後，這會強制您考慮這兩種情況，使其成為編譯時期的考慮，而不是執行時間的考慮。這些通常用於用來代表「nothing」的 Api `null`，因此 `NullReferenceException` 在許多情況下都不需要擔心。

```

/// Option values are any kind of value tagged with either 'Some' or 'None'.
/// They are used extensively in F# code to represent the cases where many other
/// languages would use null references.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/options
module OptionValues =

    /// First, define a zip code defined via Single-case Discriminated Union.
    type ZipCode = ZipCode of string

    /// Next, define a type where the ZipCode is optional.
    type Customer = { ZipCode: ZipCode option }

    /// Next, define an interface type that represents an object to compute the shipping zone for the
    customer's zip code,
    /// given implementations for the 'getState' and 'getShippingZone' abstract methods.
    type IShippingCalculator =
        abstract GetState : ZipCode -> string option
        abstract GetShippingZone : string -> int

    /// Next, calculate a shipping zone for a customer using a calculator instance.
    /// This uses combinators in the Option module to allow a functional pipeline for
    /// transforming data with Optionals.
    let CustomerShippingZone (calculator: IShippingCalculator, customer: Customer) =
        customer.ZipCode
        |> Option.bind calculator.GetState
        |> Option.map calculator.GetShippingZone

```

測量單位

F# 類型系統的一項獨特功能，就是能夠透過測量單位提供數值常值的內容。

測量單位 可讓您將數值型別與某個單位（例如計量）建立關聯，並讓函式對單位（而非數值常值）執行工作。如此一來，編譯器就可以驗證在特定內容下傳遞的數值常數值型別是否合理，從而消除與該工作類型相關聯的執行階段錯誤。


```

/// Units of measure are a way to annotate primitive numeric types in a type-safe way.
/// You can then perform type-safe arithmetic on these values.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/units-of-measure
module UnitsOfMeasure =

    /// First, open a collection of common unit names
    open Microsoft.FSharp.Data.UnitSystems.SI.UnitNames

    /// Define a unitized constant
    let sampleValue1 = 1600.0<meter>

    /// Next, define a new unit type
    [<Measure>]
    type mile =
        /// Conversion factor mile to meter.
        static member asMeter = 1609.34<meter/mile>

    /// Define a unitized constant
    let sampleValue2 = 500.0<mile>

    /// Compute metric-system constant
    let sampleValue3 = sampleValue2 * mile.asMeter

    // Values using Units of Measure can be used just like the primitive numeric type for things like
    printing.
    printfn $"After a %{sampleValue1} race I would walk %{sampleValue2} miles which would be
    %{sampleValue3} meters"

```

F# 核心程式庫會定義許多 SI 單位類型和單位轉換。若要深入瞭解，請參閱 [fsharp.core.UnitSystems.Unitsymbols.s](#) 命名空間。

類別和介面

F# 也有 .NET 類別、[介面](#)、[抽象類別](#)、[繼承](#)等的完整支援。

[類別](#) 是代表 .net 物件的類型，其 [成員](#) 可以有屬性、方法和事件。

```

/// Classes are a way of defining new object types in F#, and support standard Object-oriented constructs.
/// They can have a variety of members (methods, properties, events, etc.)
///
/// To learn more about Classes, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/classes
///
/// To learn more about Members, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/members
module DefiningClasses =

    /// A simple two-dimensional Vector class.
    ///
    /// The class's constructor is on the first line,
    /// and takes two arguments: dx and dy, both of type 'double'.
    type Vector2D(dx : double, dy : double) =

        /// This internal field stores the length of the vector, computed when the
        /// object is constructed
        let length = sqrt (dx*dx + dy*dy)

        /// 'this' specifies a name for the object's self-identifier.
        // In instance methods, it must appear before the member name.
        member this.DX = dx

        member this.DY = dy

        member this.Length = length

        /// This member is a method. The previous members were properties.
        member this.Scale(k) = Vector2D(k * this.DX, k * this.DY)

    /// This is how you instantiate the Vector2D class.
    let vector1 = Vector2D(3.0, 4.0)

    /// Get a new scaled vector object, without modifying the original object.
    let vector2 = vector1.Scale(10.0)

    printfn $"Length of vector1: {vector1.Length}\nLength of vector2: {vector2.Length}"

```

定義泛型類別也非常簡單。

```

/// Generic classes allow types to be defined with respect to a set of type parameters.
/// In the following, 'T' is the type parameter for the class.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/generics/
module DefiningGenericClasses =

    type StateTracker<'T>(initialElement: 'T) =

        /// This internal field store the states in a list.
        let mutable states = [ initialElement ]

        /// Add a new element to the list of states.
        member this.UpdateState newState =
            states <- newState :: states // use the '<-' operator to mutate the value.

        /// Get the entire list of historical states.
        member this.History = states

        /// Get the latest state.
        member this.Current = states.Head

    /// An 'int' instance of the state tracker class. Note that the type parameter is inferred.
    let tracker = StateTracker 10

    // Add a state
    tracker.UpdateState 17

```

若要執行介面，您可以使用 `interface ... with` 語法或 [物件運算式](#)。

```

/// Interfaces are object types with only 'abstract' members.
/// Object types and object expressions can implement interfaces.
///
/// To learn more, see: https://docs.microsoft.com/dotnet/fsharp/language-reference/interfaces
module ImplementingInterfaces =

    /// This is a type that implements IDisposable.
    type ReadFile() =

        let file = new System.IO.StreamReader("readme.txt")

        member this.ReadLine() = file.ReadLine()

        // This is the implementation of IDisposable members.
        interface System.IDisposable with
            member this.Dispose() = file.Close()

    /// This is an object that implements IDisposable via an Object Expression
    /// Unlike other languages such as C# or Java, a new type definition is not needed
    /// to implement an interface.
    let interfaceImplementation =
        { new System.IDisposable with
            member this.Dispose() = printfn "disposed" }

```

要使用的類型

類別、記錄、差別聯集和元組的存在會導致重要問題：您應該使用哪一項？就像大部分的生活一樣，答案都取決於您的情況。

元組很適合用來傳回函式的多個值，並使用特定值的隨選匯總作為值本身。

記錄是來自元組的「逐步執行」，具有命名標籤和選擇性成員的支援。它們很適合用於透過您的程式傳輸中資料的低人員表示。因為它們的結構相等，所以很容易與比較使用。

差別聯集有許多用途，但核心優點是能夠搭配模式比對使用，以考慮資料可擁有的所有可能「圖形」。

類別很適合許多原因，例如當您需要表示資訊，同時也該資訊系統至功能時。根據經驗法則，當您在概念上系結至某些資料的功能時，使用類別和面向物件程式設計的原則是一項很大的好處。與 c # 和 Visual Basic 交互操作時，類別也是慣用的資料類型，因為這些語言幾乎都是使用類別。

後續步驟

現在您已看過語言的一些主要功能，接下來您應該準備好撰寫您的第一個 F # 程式！查看 [消費者入門](#)，瞭解如何設定您的開發環境並撰寫一些程式碼。

後續學習的步驟可以是您喜歡的任何內容，但我們建議使用 [F # 中的功能程式設計](#)，以熟悉核心功能程式設計概念。在 F # 中建立強大的程式時，這些都是不可或缺的。

此外，請參閱 [f # 語言參考](#)，以查看 f # 的完整概念內容集合。

F 中的功能程式設計簡介

2021/3/5 • [Edit Online](#)

功能性程式設計是一種程式設計樣式，強調函式和不可變數據的使用。具型別函式程式設計是指將函式程式設計與靜態類型(例如 F #)結合。一般情況下，功能程式設計會強調下列概念：

- 作為您使用之主要結構的函式
- 運算式而非語句
- 變數上的不可變值
- 命令式程式設計的宣告式程式設計

在本系列中，您將探索使用 F # 的功能性程式設計概念和模式。在過程中，您也將學習一些 F #。

詞彙

功能性程式設計(如同其他程式設計架構)隨附的詞彙，您最終將需要學習。以下是您將會看到的一些常見詞彙：

- **函數** -函式是一種結構，會在指定輸入時產生輸出。更正式來說，它會將某個專案從一個集合 對應 到另一個集合。這項形式在許多方面都是以實體方式來完成，特別是在使用資料集合的函式時。這是功能性程式設計中最基本的 (和重要的) 概念。
- **運算式** -運算式是在產生值的程式碼中的結構。在 F # 中，必須系結或明確忽略此值。運算式可以完整由函式呼叫取代。
- **純度** -純度是函數的屬性，因此相同的引數的傳回值一律相同，而且其評估沒有任何副作用。純虛擬函式完全取決於其引數。
- **參考透明度** -參考透明度是運算式的屬性，可將它們取代為其輸出，而不會影響程式的行為。
- **永久性** -永久性表示無法就地變更值。這與可以就地變更的變數相比較。

範例

下列範例示範這些核心概念。

函式

函式程式設計中最常見的基本結構是函數。以下是將1加到整數的簡單函式：

```
let addOne x = x + 1
```

其類型簽章如下所示：

```
val addOne: x:int -> int
```

簽章可以讀取為「`addOne` 接受 `int` 已命名 `x` 且將會產生」`int`。更正式 `addOne` 來說，是將一組整數的值 對應 至整數列。`->` Token 表示此對應。在 F # 中，您通常可以查看函式簽章，以瞭解其用途。

那麼，為什麼簽章很重要？在具型別函式程式設計中，函數的執行通常比實際的型別簽章更不重要！`addOne` 將值1加入至整數的事實執行時間很有趣，但是當您在建立程式時，它接受並傳回的事實 `int` 就是通知您實際使用此函式的方式。此外，一旦您正確地使用此函式(相對於其類型簽章)，則診斷任何問題只能在函式主體內進行 `addOne`。這是具類型的功能性程式設計背後的促成。

運算式

運算式是評估為值的結構。相較於執行動作的語句，運算式可以考慮執行可傳回值的動作。運算式幾乎一律用於功能性程式設計，而不是語句。

請考慮先前的函數 `addOne` 的主體 `addOne` 為運算式：

```
// 'x + 1' is an expression!  
let addOne x = x + 1
```

這是定義函數結果型別的運算式結果 `addOne`。例如，組成此函式的運算式可能會變更為不同的類型，例如 `string`：

```
let addOne x = x.ToString() + "1"
```

函數的簽章現在是：

```
val addOne: x:'a -> string
```

由於 F# 中的任何型別都可以 `ToString()` 呼叫它，因此的型別已 `x` 成為泛型 (稱為 [自動一般化](#))，而結果型別為 `string`。

運算式不只是函數的主體。您可以有運算式來產生您在其他地方使用的值。其中一個常見的情況是 `if`：

```
// Checks if 'x' is odd by using the mod operator  
let isOdd x = x % 2 <> 0  
  
let addOneIfOdd input =  
    let result =  
        if isOdd input then  
            input + 1  
        else  
            input  
  
    result
```

`if` 運算式會產生名為的 `result` 值。請注意，您可以 `result` 完全省略，讓 `if` 運算式成為 `addOneIfOdd` 函數主體。要記住運算式的重點在於它們會產生一個值。

有一種特殊的型 `unit` 別，會在沒有要傳回的專案時使用。例如，請考慮這個簡單的函式：

```
let printString (str: string) =  
    printfn $"String is: {str}"
```

簽章看起來像這樣：

```
val printString: str:string -> unit
```

`unit` 型別指出沒有傳回實際值。當您的常式必須「執行工作」，但因為該工作不會傳回任何值時，這非常有用。

這與命令式程式設計相反，其中對等的 `if` 結構是語句，而產生值通常是透過變動變數來完成。例如，在 C# 中，程式碼的撰寫方式可能如下所示：

```
bool IsOdd(int x) => x % 2 != 0;

int AddOneIfOdd(int input)
{
    var result = input;

    if (IsOdd(input))
    {
        result = input + 1;
    }

    return result;
}
```

值得注意的是，c# 和其他 C 樣式的語言都支援 [三元運算式](#)，這種運算式允許以運算式為基礎的條件式程式設計。

在功能性程式設計中，使用語句來改變值是很罕見的。雖然有些功能性語言支援語句和變化，但是在功能性程式設計中使用這些概念並不常見。

純虛擬函式

如先前所述，純虛擬函式是函式，其功能如下：

- 針對相同的輸入，一律評估為相同的值。
- 沒有副作用。

在此內容中考慮數學函數很有說明。在數學中，函式只會依賴其引數，而且沒有任何副作用。在數學函數中 $f(x) = x + 1$ ，的值 $f(x)$ 只取決於的值 x 。函式程式設計中的純虛擬函式是一樣的。

撰寫純虛擬函式時，函式只能相依赖于其引數，而不會執行任何會產生副作用的動作。

以下是非純虛擬函式的範例，因為它相依赖于全域、可變動的狀態：

```
let mutable value = 1

let addOneToValue x = x + value
```

此函式 `addOneToValue` 很清楚地 impure，因為 `value` 可以隨時變更為具有不同于1的值。這種以全域值為依據的模式，可避免在功能性程式設計中使用。

以下是非純虛擬函式的另一個範例，因為它會執行副作用：

```
let addOneToValue x =
    printfn $"x is %d{x}"
    x + 1
```

雖然此函式不會相依赖于全域值，但它會將的值寫入 `x` 程式的輸出。雖然這樣做並不會發生任何錯誤，但這表示此函式不是單純的。如果程式的另一個部分相依赖于程式外部的某個專案，例如輸出緩衝區，則呼叫此函式會影響程式的其他部分。

移除 `printfn` 語句會讓函數成為單純的：

```
let addOneToValue x = x + 1
```

雖然此函式在本質上並不 優於使用語句的舊版 `printfn`，但它確實保證此函式確實會傳回值。每次呼叫此函式會產生相同的結果：它只會產生值。由純度提供的可預測性，是許多功能程式設計人員致力於的一些功能。

不變性

最後，具型別函式程式設計的其中一個最基本概念是永久性。在 F # 中，所有值預設都是不可變的。這表示，除非您將它們明確地標示為可變動，否則它們無法就地變動。

在實務上，使用不可變的值表示您將程式設計的方法從「我需要變更一些東西」變更為「我需要產生新的值」。

例如，將 1 加 1 值表示產生新的值，而不會變更現有的值：

```
let value = 1
let secondValue = value + 1
```

在 F # 中，下列程式碼 **不會** 改變 `value` 函數; 相反地，它會執行相等檢查：

```
let value = 1
value = value + 1 // Produces a 'bool' value!
```

有些功能性程式設計語言並不支援變動。在 F # 中，它是支援的，但它不是值的預設行為。

此概念甚至進一步延伸至資料結構。在功能性程式設計中，不可變的資料結構，例如 `set` (以及許多) 有不同于一開始預期的不同執行。就概念而言，將專案加入至集合中的專案並不會變更集合，它會產生具有新增值的 **新** 集合。在幕後，這通常是透過不同的資料結構來完成，讓您能夠有效率地追蹤值，以便將資料的適當標記法提供給結果。

這種使用值和資料結構的樣式是很重要的，因為它會強制您將修改某些事物的任何作業視為建立新的版本。這可讓您的程式中的等號和可比較性等專案保持一致。

後續步驟

下一節將徹底涵蓋函式，探索可在功能性程式設計中使用這些功能的不同方式。

第 [一級函式](#) 探索更深層的函式，並示範如何在各種內容中使用這些函數。

進一步閱讀

[思考功能](#) 系列是另一項絕佳的資源，可瞭解如何使用 F # 進行功能性程式設計。它涵蓋功能性程式設計的基本概念，並使用 F # 功能來說明概念，以提供實用且容易閱讀的方式。

一級函式

2020/11/2 • [Edit Online](#)

函式程式設計語言的定義特性是將函式提升為第一級狀態的功能。您應該能夠使用函式來處理其他內建類型的值，而且能夠以相當程度的投入量來進行。

第一個類別狀態的典型量值包括下列各項：

- 您可以將函式系結至識別碼嗎？也就是說，您可以為它們命名嗎？
- 您可以將函數儲存在資料結構中，例如清單中？
- 您可以在函式呼叫中傳遞函式作為引數嗎？
- 您可以從函式呼叫傳回函數嗎？

最後兩個量值會定義所謂的 *高階作業* 或 *更高順序* 的函式。高階函式接受函數做為引數，並傳回函式作為函式呼叫的值。這些作業支援將函數設計支柱為對應函式和函式組合。

為值命名

如果函式是第一級的值，您必須能夠命名它，就像您可以命名整數、字串和其他內建類型一樣。這在功能程式設計文獻中稱為將識別碼系結至值。F# 使用系結將名稱系 `let` [結至值](#)：`let <identifier> = <value>`。下列程式碼顯示兩個範例。

```
// Integer and string.
let num = 10
let str = "F#"
```

您可以輕鬆地將函數命名。下列範例會定義名為的函式，該函式會將識別碼系結 `squareIt` `squareIt` 至 [lambda 運算式](#) `fun n -> n * n`。函式 `squareIt` 有一個參數，`n` 而它會傳回該參數的平方。

```
let squareIt = fun n -> n * n
```

F# 提供下列更簡潔的語法，以較少的輸入來達成相同的結果。

```
let squareIt2 n = n * n
```

接下來的範例會使用第一個樣式，`let <function-name> = <lambda-expression>` 以強調函式宣告和其他類型值的宣告之間的相似之處。不過，所有命名的函式也可以使用精簡的語法來撰寫。其中一些範例是以這兩種方式撰寫的。

將值儲存在資料結構中

第一個類別的值可以儲存在資料結構中。下列程式碼顯示在清單和元組中儲存值的範例。

```
// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]

// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )
```

為了確認儲存在元組中的函式名稱確實是評估為函式，下列範例會使用 `fst` 和 `snd` 運算子來解壓縮元組中的第一個和第二個元素 `funAndArgTuple`。元組中的第一個專案是 `squareIt`，而第二個元素是 `num`。 `num` 在先前的範例中，識別碼會系結至整數10，此為函數的有效引數 `squareIt`。第二個運算式會將元組中的第一個專案套用至元組中的第二個元素：`squareIt num`。

```
// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))
```

同樣地，如同識別碼 `num` 和整數10可以交替使用，因此可以使用識別碼 `squareIt` 和 `lambda` 運算式 `fun n -> n * n`。

```
// Make a tuple of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))
```

傳遞值做為引數

如果某個值在語言中具有第一級的狀態，您可以將它當作引數傳遞給函式。例如，通常會將整數和字串當作引數傳遞。下列程式碼顯示在 F# 中做為引數傳遞的整數和字串。

```
// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)
```

如果函式具有頂級狀態，您必須能夠以同樣的方式將它們當作引數傳遞。請記住，這是較高順序函式的第一個特性。

在下列範例中，函數 `applyIt` 有兩個參數：`op` 和 `arg`。如果您在有一個參數的函式中傳送，並將函 `op` 式的適當引數傳送至，函數會傳回套用 `arg` 至的結果 `op arg`。在下列範例中，函式引數和整數引數都會以相同的方式傳送，方法是使用它們的名稱。

```
// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)
```

將函式作為引數傳送至另一個函式的能力，是以函式程式設計語言（例如地圖或篩選作業）的常見抽象概念為基礎。例如，對應作業是較高階的函式，它會捕捉逐步執行清單的函式所共用的計算、對每個元素執行某些動作，然後傳回結果清單。您可能會想要遞增整數清單中的每個專案，或每個元素的平方，或是將字串清單中的每個專案變更為大寫。計算容易出錯的部分是遞迴程式，它會逐步執行此清單，並建立要傳回的結果清單。該元件是在對應函式中所捕捉。您必須為特定應用程式撰寫的所有專案，都是您想要個別套用至每個清單專案的函式，（新增、求大小寫）。該函式會以引數的形式傳送至對應函式，就像 `squareIt` `applyIt` 在先前的範例中所傳送的一樣。

F# 為大部分的集合類型（包括 [清單](#)、[陣列](#) 和 [序列](#)）提供對應方法。下列範例使用清單。語法是

```
List.map <the function> <the list>。
```

```
// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot
```

如需詳細資訊，請參閱 [清單](#)。

從函式呼叫傳回值

最後，如果函式在語言中具有第一級的狀態，您就必須能夠將它傳回做為函式呼叫的值，就像您傳回其他類型（例如整數和字串）一樣。

下列函式呼叫會傳回整數並加以顯示。

```
// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)
```

下列函式呼叫會傳回字串。

```
// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()
```

下列函式呼叫(宣告為 inline)會傳回布林值。顯示的值為 `True`。

```
System.Console.WriteLine((fun n -> n % 2 = 1) 15)
```

以函式呼叫的值傳回函式的能力，是較高順序函數的第二個特性。在下列範例中，`checkFor` 定義為採用一個引數的函式，`item` 並傳回新的函式作為其值。傳回的函式會接受清單作為其引數，`lst` 並 `item` 在中搜尋 `lst`。如果 `item` 存在，則函數會傳回 `true`。如果 `item` 不存在，則函數會傳回 `false`。如上一節所示，下列程式碼會使用提供的清單函式 `list.exists` 來搜尋清單。

```
let checkFor item =
    let functionToReturn = fun lst ->
        List.exists (fun a -> a = item) lst
    functionToReturn
```

下列程式碼會使用來建立新的函式，該函式 `checkFor` 會接受一個引數、一個清單，並在清單中搜尋7。

```
// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)
```

下列範例會使用 F# 中函式的第一個類別狀態來宣告函式，`compose` 此函式會傳回兩個函式引數的組合。

```
// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expression syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
            op1 (op2 n)

        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:
let compose3 op1 op2 =
    let funToReturn = fun n ->
        op1 (op2 n)

    funToReturn
```

NOTE

如需更短的版本，請參閱下一節「擴充函數」。

下列程式碼會將兩個函式作為引數傳送至，這兩個函式 `compose` 都會採用相同類型的單一引數。傳回值是新的函式，這是兩個函式引數的組合。

```
// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)
```

NOTE

F # 提供兩個運算子, `<<` 以及 `>>` 撰寫函數。例如, `let squareAndDouble2 = doubleIt << squareIt`
`let squareAndDouble = compose doubleIt squareIt` 在上一個範例中相當於。

下列範例會傳回函式作為函式呼叫的值, 以建立簡單的猜測遊戲。若要建立遊戲, 請 `makeGame` 使用您希望他人猜測的值來呼叫 `target`。函式的傳回值 `makeGame` 是接受一個引數 (猜測) 的函式, 並報告猜測是否正確。

```
let makeGame target =  
    // Build a lambda expression that is the function that plays the game.  
    let game = fun guess ->  
        if guess = target then  
            System.Console.WriteLine("You win!")  
        else  
            System.Console.WriteLine("Wrong. Try again.")  
    // Now just return it.  
    game
```

下列程式碼會呼叫, 並傳送 `makeGame` 的值 `7` `target`。識別碼 `playGame` 會系結至傳回的 lambda 運算式。因此, 是一種函式, 它會將的 `playGame` 值作為其一個引數 `guess`。

```
let playGame = makeGame 7  
// Send in some guesses.  
playGame 2  
playGame 9  
playGame 7  
  
// Output:  
// Wrong. Try again.  
// Wrong. Try again.  
// You win!  
  
// The following game specifies a character instead of an integer for target.  
let alphaGame = makeGame 'q'  
alphaGame 'c'  
alphaGame 'r'  
alphaGame 'j'  
alphaGame 'q'  
  
// Output:  
// Wrong. Try again.  
// Wrong. Try again.  
// Wrong. Try again.  
// You win!
```

擴充函數

您可以利用 F # 函式宣告中的隱含 *currying*, 更簡潔地撰寫上一節中的許多範例。Currying 是一種程式, 可將具有多個參數的函式轉換為一系列的內嵌函式, 其中每個函式都有單一參數。在 F # 中, 有多個參數的函式在本質上是局部的。例如, 在 `compose` 上一節中, 您可以使用三個參數, 以下列簡潔的樣式來撰寫。

```
let compose4 op1 op2 n = op1 (op2 n)
```

不過, 結果是一個參數的函式, 它會傳回一個參數的函式, 而後者接著會傳回一個參數的另一個函式, 如下所示 `compose4curried`。

```
let compose4curried =  
    fun op1 ->  
        fun op2 ->  
            fun n -> op1 (op2 n)
```

您可以透過數種方式來存取此函式。下列每個範例都會傳回並顯示18。您可以 `compose4` 使用 `compose4curried` 中的任何範例取代。

```
// Access one layer at a time.  
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)  
  
// Access as in the original compose examples, sending arguments for  
// op1 and op2, then applying the resulting function to a value.  
System.Console.WriteLine((compose4 doubleIt squareIt) 3)  
  
// Access by sending all three arguments at the same time.  
System.Console.WriteLine(compose4 doubleIt squareIt 3)
```

若要確認函式的運作方式仍與之前相同，請再次嘗試原來的測試案例。

```
let doubleAndSquare4 = compose4 squareIt doubleIt  
// The following expression returns and displays 36.  
System.Console.WriteLine(doubleAndSquare4 3)  
  
let squareAndDouble4 = compose4 doubleIt squareIt  
// The following expression returns and displays 18.  
System.Console.WriteLine(squareAndDouble4 3)
```

NOTE

您可以藉由將元組中的參數括住來限制 currying。如需詳細資訊，請參閱 [參數和引數](#) 中的「參數模式」。

下列範例會使用隱含 currying 來寫入較短的版本 `makeGame`。結構和傳回函式的詳細資料 `makeGame` `game`，在此格式中較不明確，但您可以使用原始測試案例來確認結果是相同的。

```
let makeGame2 target guess =  
    if guess = target then  
        System.Console.WriteLine("You win!")  
    else  
        System.Console.WriteLine("Wrong. Try again.")  
  
let playGame2 = makeGame2 7  
playGame2 2  
playGame2 9  
playGame2 7  
  
let alphaGame2 = makeGame2 'q'  
alphaGame2 'c'  
alphaGame2 'r'  
alphaGame2 'j'  
alphaGame2 'q'
```

如需 currying 的詳細資訊，請參閱 [函式](#) 中的「部分引數應用程式」。

識別碼和函式定義是可互換的

`num` 先前範例中的變數名稱會評估為整數10，而且如果 `num` 有效，則10也是有效的。函式識別碼和其值也是如

此: 可以使用函式名稱的任何位置, 都可以使用它所系結的 lambda 運算式。

下列範例會定義名為的函式 `Boolean` `isNegative`, 然後使用函數的名稱和函式的定義。接下來的三個範例都會傳回並顯示 `False`。

```
let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)
```

若要進一步進行, 請將系結的值取代為 `applyIt` `applyIt`。

```
System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0) 10)
```

函數是 F 中的第一個類別值#

上述各節中的範例會示範 F # 中的函式符合 F # 中第一個類別值的準則:

- 您可以將識別碼系結至函式定義。

```
let squareIt = fun n -> n * n
```

- 您可以將函數儲存在資料結構中。

```
let funTuple2 = ( BMI Calculator, fun n -> n * n )
```

- 您可以將函數當作引數傳遞。

```
let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]
```

- 您可以傳回函數作為函式呼叫的值。

```
let checkFor item =
    let functionToReturn = fun lst ->
        List.exists (fun a -> a = item) lst
    functionToReturn
```

如需 F # 的詳細資訊, 請參閱 [f # 語言參考](#)。

範例

描述

下列程式碼包含本主題中的所有範例。

程式碼

```
// ** GIVE THE VALUE A NAME **
```



```

// Integer and string.
let num = 10
let str = "F#"

let squareIt = fun n -> n * n

let squareIt2 n = n * n

// ** STORE THE VALUE IN A DATA STRUCTURE **

// Lists.

// Storing integers and strings.
let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
let stringList = [ "one"; "two"; "three" ]

// You cannot mix types in a list. The following declaration causes a
// type-mismatch compiler error.
//let failedList = [ 5; "six" ]

// In F#, functions can be stored in a list, as long as the functions
// have the same signature.

// Function doubleIt has the same signature as squareIt, declared previously.
//let squareIt = fun n -> n * n
let doubleIt = fun n -> 2 * n

// Functions squareIt and doubleIt can be stored together in a list.
let funList = [ squareIt; doubleIt ]

// Function squareIt cannot be stored in a list together with a function
// that has a different signature, such as the following body mass
// index (BMI) calculator.
let BMICalculator = fun ht wt ->
    (float wt / float (squareIt ht)) * 703.0

// The following expression causes a type-mismatch compiler error.
//let failedFunList = [ squareIt; BMICalculator ]

// Tuples.

// Integers and strings.
let integerTuple = ( 1, -7 )
let stringTuple = ( "one", "two", "three" )

// A tuple does not require its elements to be of the same type.
let mixedTuple = ( 1, "two", 3.3 )

// Similarly, function elements in tuples can have different signatures.
let funTuple = ( squareIt, BMICalculator )

// Functions can be mixed with integers, strings, and other types in
// a tuple. Identifier num was declared previously.
//let num = 10
let moreMixedTuple = ( num, "two", 3.3, squareIt )

// You can pull a function out of a tuple and apply it. Both squareIt and num
// were defined previously.

```

```

// were defined previously.
let funAndArgTuple = (squareIt, num)

// The following expression applies squareIt to num, returns 100, and
// then displays 100.
System.Console.WriteLine((fst funAndArgTuple)(snd funAndArgTuple))

// Make a list of values instead of identifiers.
let funAndArgTuple2 = ((fun n -> n * n), 10)

// The following expression applies a squaring function to 10, returns
// 100, and then displays 100.
System.Console.WriteLine((fst funAndArgTuple2)(snd funAndArgTuple2))

// ** PASS THE VALUE AS AN ARGUMENT **

// An integer is passed to squareIt. Both squareIt and num are defined in
// previous examples.
//let num = 10
//let squareIt = fun n -> n * n
System.Console.WriteLine(squareIt num)

// String.
// Function repeatString concatenates a string with itself.
let repeatString = fun s -> s + s

// A string is passed to repeatString. HelloHello is returned and displayed.
let greeting = "Hello"
System.Console.WriteLine(repeatString greeting)

// Define the function, again using lambda expression syntax.
let applyIt = fun op arg -> op arg

// Send squareIt for the function, op, and num for the argument you want to
// apply squareIt to, arg. Both squareIt and num are defined in previous
// examples. The result returned and displayed is 100.
System.Console.WriteLine(applyIt squareIt num)

// The following expression shows the concise syntax for the previous function
// definition.
let applyIt2 op arg = op arg
// The following line also displays 100.
System.Console.WriteLine(applyIt2 squareIt num)

// List integerList was defined previously:
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

// You can send the function argument by name, if an appropriate function
// is available. The following expression uses squareIt.
let squareAll = List.map squareIt integerList

// The following line displays [1; 4; 9; 16; 25; 36; 49]
printfn "%A" squareAll

// Or you can define the action to apply to each list element inline.
// For example, no function that tests for even integers has been defined,
// so the following expression defines the appropriate function inline.
// The function returns true if n is even; otherwise it returns false.
let evenOrNot = List.map (fun n -> n % 2 = 0) integerList

```

```

// The following line displays [false; true; false; true; false; true; false]
printfn "%A" evenOrNot

// ** RETURN THE VALUE FROM A FUNCTION CALL **

// Function doubleIt is defined in a previous example.
//let doubleIt = fun n -> 2 * n
System.Console.WriteLine(doubleIt 3)
System.Console.WriteLine(squareIt 4)

// The following function call returns a string:

// str is defined in a previous section.
//let str = "F#"
let lowercase = str.ToLower()

System.Console.WriteLine((fun n -> n % 2 = 1) 15)

let checkFor item =
    let functionToReturn = fun lst ->
        List.exists (fun a -> a = item) lst
    functionToReturn

// integerList and stringList were defined earlier.
//let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]
//let stringList = [ "one"; "two"; "three" ]

// The returned function is given the name checkFor7.
let checkFor7 = checkFor 7

// The result displayed when checkFor7 is applied to integerList is True.
System.Console.WriteLine(checkFor7 integerList)

// The following code repeats the process for "seven" in stringList.
let checkForSeven = checkFor "seven"

// The result displayed is False.
System.Console.WriteLine(checkForSeven stringList)

// Function compose takes two arguments. Each argument is a function
// that takes one argument of the same type. The following declaration
// uses lambda expression syntax.
let compose =
    fun op1 op2 ->
        fun n ->
            op1 (op2 n)

// To clarify what you are returning, use a nested let expression:
let compose2 =
    fun op1 op2 ->
        // Use a let expression to build the function that will be returned.
        let funToReturn = fun n ->
            op1 (op2 n)
        // Then just return it.
        funToReturn

// Or, integrating the more concise syntax:

```

```

let compose3 op1 op2 =
    let funToReturn = fun n ->
        op1 (op2 n)
    funToReturn

// Functions squareIt and doubleIt were defined in a previous example.
let doubleAndSquare = compose squareIt doubleIt
// The following expression doubles 3, squares 6, and returns and
// displays 36.
System.Console.WriteLine(doubleAndSquare 3)

let squareAndDouble = compose doubleIt squareIt
// The following expression squares 3, doubles 9, returns 18, and
// then displays 18.
System.Console.WriteLine(squareAndDouble 3)

let makeGame target =
    // Build a lambda expression that is the function that plays the game.
    let game = fun guess ->
        if guess = target then
            System.Console.WriteLine("You win!")
        else
            System.Console.WriteLine("Wrong. Try again.")
    // Now just return it.
    game

let playGame = makeGame 7
// Send in some guesses.
playGame 2
playGame 9
playGame 7

// Output:
// Wrong. Try again.
// Wrong. Try again.
// You win!

// The following game specifies a character instead of an integer for target.
let alphaGame = makeGame 'q'
alphaGame 'c'
alphaGame 'r'
alphaGame 'j'
alphaGame 'q'

// Output:
// Wrong. Try again.
// Wrong. Try again.
// Wrong. Try again.
// You win!

// ** CURRIED FUNCTIONS **

let compose4 op1 op2 n = op1 (op2 n)

let compose4curried =
    fun op1 ->
        fun op2 ->
            fun n -> op1 (op2 n)

```

```

// Access one layer at a time.
System.Console.WriteLine(((compose4 doubleIt) squareIt) 3)

// Access as in the original compose examples, sending arguments for
// op1 and op2, then applying the resulting function to a value.
System.Console.WriteLine((compose4 doubleIt squareIt) 3)

// Access by sending all three arguments at the same time.
System.Console.WriteLine(compose4 doubleIt squareIt 3)


let doubleAndSquare4 = compose4 squareIt doubleIt
// The following expression returns and displays 36.
System.Console.WriteLine(doubleAndSquare4 3)

let squareAndDouble4 = compose4 doubleIt squareIt
// The following expression returns and displays 18.
System.Console.WriteLine(squareAndDouble4 3)


let makeGame2 target guess =
    if guess = target then
        System.Console.WriteLine("You win!")
    else
        System.Console.WriteLine("Wrong. Try again.")

let playGame2 = makeGame2 7
playGame2 2
playGame2 9
playGame2 7

let alphaGame2 = makeGame2 'q'
alphaGame2 'c'
alphaGame2 'r'
alphaGame2 'j'
alphaGame2 'q'


// ** IDENTIFIER AND FUNCTION DEFINITION ARE INTERCHANGEABLE **


let isNegative = fun n -> n < 0

// This example uses the names of the function argument and the integer
// argument. Identifier num is defined in a previous example.
//let num = 10
System.Console.WriteLine(applyIt isNegative num)

// This example substitutes the value that num is bound to for num, and the
// value that isNegative is bound to for isNegative.
System.Console.WriteLine(applyIt (fun n -> n < 0) 10)


System.Console.WriteLine((fun op arg -> op arg) (fun n -> n < 0) 10)


// ** FUNCTIONS ARE FIRST-CLASS VALUES IN F# **

//let squareIt = fun n -> n * n

```

```
let funTuple2 = ( BMICalculator, fun n -> n * n )

let increments = List.map (fun n -> n + 1) [ 1; 2; 3; 4; 5; 6; 7 ]

//let checkFor item =
//    let functionToReturn = fun lst ->
//        List.exists (fun a -> a = item) lst
//    functionToReturn
```

另請參閱

- [清單](#)
- [元組](#)
- [函式](#)
- `let` [綁定](#)
- [Lambda 運算式](#): `fun` [關鍵字](#)

F 中的非同步程式設計

2021/3/5 • [Edit Online](#)

非同步程式設計是基於各種原因而對新式應用程式來說很重要的一種機制。大部分的開發人員會遇到兩個主要的使用案例：

- 呈現可提供大量並行連入要求的伺服器程式，同時將要求處理期間所佔用的系統資源降至最低，同時等待來自該進程外部系統或服務的輸入
- 維護回應迅速的 UI 或主執行緒，同時進行背景工作

雖然背景工作通常牽涉到多個執行緒的使用率，但請務必考慮非同步和多執行緒的概念。事實上，這些都是分開的考慮，而不是另一種。本文將更詳細地說明不同的概念。

非同步已定義

上一個重點是，非同步與多個執行緒的使用無關，值得進一步說明。有時候相關的概念有三個，但彼此之間完全無關：

- Concurrency 在重迭的時間週期內執行多個計算時。
- 並行當多個計算或單一計算的數個部分在相同時間執行時。
- 非同步當一或多個計算可以與主要程式流程分開執行時。

這三個都是互相關聯的概念，但很容易混為一談，尤其是在一起使用時。例如，您可能需要以平行方式執行多個非同步計算。此關聯性並不表示平行處理原則或非同步暗示另一個。

如果您考慮「非同步」這個字的 etymology，則牽涉到兩個部分：

- "a"，表示「不」。
- 「同步」，表示「同時」。

當您將這兩個詞彙結合在一起時，您會看到「非同步」表示「不是同時」。這樣就完成了！這項定義中沒有平行存取或平行處理原則的含意。在實務上也是如此。

在實際的情況下，F# 中的非同步計算排程為獨立于主要程式流程的執行。此獨立執行不表示並行或平行處理，也不表示計算一律會在背景中發生。事實上，根據計算的本質和計算執行所在的環境而定，非同步計算甚至可以同步執行。

您應該具備的主要重點是，非同步計算與主要程式流程無關。雖然非同步計算的執行時間或方式有一些保證，但是有一些方法可協調和排程。本文的其餘部分會探索 F# 非同步核心概念，以及如何使用 F# 內建的類型、函式和運算式。

核心概念

在 F# 中，非同步程式設計是以三個核心概念為中心：

- `Async<'T>` 型別，表示可組合的非同步計算。
- `Async` 模組函數可讓您排程非同步工作、撰寫非同步計算，以及轉換非同步結果。
- `async { }` [計算運算式](#)，提供建立和控制非同步計算的便利語法。

您可以在下列範例中看到這三個概念：

```

open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has {bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    printTotalFileBytes "path-to-file.txt"
    |> Async.RunSynchronously

    Console.Read() |> ignore
    0

```

在此範例中，`printTotalFileBytes` 函數的型別為 `string -> Async<unit>`。呼叫函數並不會實際執行非同步計算。相反地，它會傳回 `Async<unit>`，作為要以非同步方式執行的工作規格。它會 `Async.AwaitTask` 在其主體中呼叫，以將的結果轉換 `ReadAllBytesAsync` 為適當的類型。

另一個重要的程式列是的呼叫 `Async.RunSynchronously`。如果您想要實際執行 F# 非同步計算，這是啟動函式的其中一個非同步模組，您必須呼叫這些函式。

這是 C#/Visual 基本程式設計風格的基本差異 `async`。在 F# 中，可以將非同步計算視為冷工作。它們必須明確地啟動，才能實際執行。這有一些優點，因為它可讓您比使用 C# 或 Visual Basic 更輕鬆地結合和排序非同步工作。

合併非同步計算

以下是透過合併計算來建立上一個範例的範例：

```

open System
open System.IO

let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has {bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    argv
    |> Seq.map printTotalFileBytes
    |> Async.Parallel
    |> Async.Ignore
    |> Async.RunSynchronously

    0

```

如您所見，`main` 函數有更多元素。就概念而言，它會執行下列動作：

1. 使用將命令列引數轉換成一系列 `Async<unit>` 計算 `Seq.map`。
2. 建立 `Async<'T[]>`，以 `printTotalFileBytes` 在執行時平行排程和執行計算。
3. 建立 `Async<unit>` 會執行平行計算的，並忽略其結果 (也就是 `unit[]`)。
4. 明確執行整體組成的計算 `Async.RunSynchronously`，並封鎖直到完成為止。

當此程式執行時，會 `printTotalFileBytes` 針對每個命令列引數平行執行。由於非同步計算獨立執行程式流程，因此沒有任何定義的順序可列印其資訊並完成執行。計算會以平行方式排程，但不保證其執行順序。

順序非同步計算

由於 `Async<'T>` 是工作的規格，而不是已執行的工作，因此您可以輕鬆地執行更複雜的轉換。以下範例會針對一組非同步計算進行順序，讓它們在另一個之後執行。

```
let printTotalFileBytes path =
    async {
        let! bytes = File.ReadAllBytesAsync(path) |> Async.AwaitTask
        let fileName = Path.GetFileName(path)
        printfn $"File {fileName} has {bytes.Length} bytes"
    }

[<EntryPoint>]
let main argv =
    argv
    |> Seq.map printTotalFileBytes
    |> Async.Sequential
    |> Async.Ignore
    |> Async.RunSynchronously
    |> ignore
```

這會排程依 `printTotalFileBytes` 元素循序執行，`argv` 而非以平行方式進行排程。因為後續的計算完成執行之後，每個後續的作業都不會排程，所以計算會排序，使其執行不會有任何重迭。

重要的非同步模組函數

當您在 F# 中撰寫非同步程式碼時，通常會與處理計算排程的架構互動。不過，這不一定是如此，因此最好瞭解可用來排程非同步工作的各種功能。

因為 F# 非同步計算是工作的 *規格*，而不是表示已在執行中的工作，所以必須使用啟動函式來明確地啟動它們。有許多 [非同步啟動方法](#) 在不同的內容中很有用。下一節將說明一些較常見的啟動函式。

Async. Async.startchild

在非同步計算中啟動子計算。這可讓您同時執行多個非同步計算。子系計算會與父計算共用解除標記。如果取消父代計算，子計算也會取消。

簽章：

```
computation: Async<'T> * timeout: ?int -> Async<Async<'T>>
```

使用時機：

- 當您想要同時執行多個非同步計算(而不是一次一個)，但未以平行方式排程它們時。
- 當您想要將子計算的存留期與父計算的存留期系結時。

要注意的事項：

- 啟動多個計算和 `Async.StartChild` 平行排程不同。如果您想要以平行方式排程計算，請使用 `Async.Parallel`。
- 取消父計算將會觸發其啟動的所有子計算的取消。

Async. StartImmediate

執行非同步計算，並在目前的作業系統執行緒上立即開始。如果您需要在計算期間更新呼叫執行緒上的某個內容，這會很有說明。例如，如果非同步計算必須更新 UI (例如更新進度列)，則 `Async.StartImmediate` 應該使用。

簽章：

```
computation: Async<unit> * cancellationToken: ?CancellationToken -> unit
```

使用時機：

- 當您需要在非同步計算的過程中，更新呼叫執行緒上的某些內容。

要注意的事項：

- 非同步計算中的程式碼將會在發生排程的任何執行緒上執行。如果執行緒是以某種方式區分，例如 UI 執行緒，這可能會造成問題。在這種情況下，`Async.StartImmediate` 可能不適合使用。

Async. Async.starttask

線上程集區中執行計算。傳回 `Task<TResult>`，當計算終止 (產生結果、擲回例外狀況或取消) 時，會在對應的狀態下完成。如果未提供取消權杖，則會使用預設解除標記。

簽章：

```
computation: Async<'T> * taskCreationOptions: ?TaskCreationOptions * cancellationToken: ?CancellationToken -> Task<'T>
```

使用時機：

- 當您需要呼叫 .NET API 時，它會產生 `Task<TResult>` 以代表非同步計算的結果。

要注意的事項：

- 此呼叫會配置額外的 `Task` 物件，如果經常使用，則會增加額外負荷。

Async. Parallel

排定要平行執行的非同步計算順序，以提供的順序產生結果陣列。藉由指定參數，可以選擇性地調整/節流處理平行處理原則的程度 `maxDegreeOfParallelism`。

簽章：

```
computations: seq<Async<'T>> * ?maxDegreeOfParallelism: int -> Async<'T[]>
```

使用時機：

- 如果您需要同時執行一組計算，而不依賴它們的執行順序。
- 如果您不需要以平行方式排程的結果，直到它們全部完成為止。

要注意的事項：

- 一旦所有計算完成之後，您就只能存取產生的值陣列。
- 計算會在最後一次排程時執行。此行為表示您無法依賴它們的執行順序。

Async. 連續

排定以傳遞的循序執行非同步計算的順序。第一個計算將會執行，然後再執行一次，依此類推。不會以平行方式執行計算。

簽章：

```
computations: seq<Async<'T>> -> Async<'T[]>
```

使用時機：

- 如果您需要依序執行多個計算。

要注意的事項：

- 一旦所有計算完成之後，您就只能存取產生的值陣列。
- 計算將依傳遞給此函式的循序執行，這可能表示在傳回結果之前，會經歷更多時間。

Async. Async.awaittask

傳回非同步計算，等候指定的 `Task<TResult>` 完成，並將其結果傳回為 `Async<'T>`

簽章：

```
task: Task<'T> -> Async<'T>
```

使用時機：

- 當您使用的 .NET API 會 `Task<TResult>` 在 F# 非同步計算中傳回。

要注意的事項：

- 例外狀況會依照 `AggregateException` 工作平行程式庫的慣例包裝，而這種行為與 F# `async` 通常會如何呈現例外狀況不同。

Async. Catch

建立異步計算，以執行指定的 `Async<'T>`，傳回 `Async<Choice<'T, exn>>`。如果指定的 `Async<'T>` 成功完成，則 `Choice1Of2` 會傳回具有結果值的。如果例外狀況在完成之前擲回，則 `Choice2Of2` 會傳回，並產生例外狀況。如果它是在由許多計算所組成的非同步計算上使用，且其中一個計算擲回例外狀況，則會完全停止包含的計算。

簽章：

```
computation: Async<'T> -> Async<Choice<'T, exn>>
```

使用時機：

- 當您執行的非同步工作可能因例外狀況而失敗，而您想要在呼叫端中處理該例外狀況時。

要注意的事項：

- 當使用結合或排序的非同步計算時，如果其中一個「內部」計算擲回例外狀況，則會完全停止包含的計算。

Async. Ignore

建立會執行指定計算但捨棄其結果的非同步計算。

簽章：

```
computation: Async<'T> -> Async<unit>
```

使用時機：

- 當您有不需結果的非同步計算時。這類似于 `ignore` 非非同步程式碼的函式。

要注意的事項：

- 如果您必須使用 `Async.Ignore`，因為您想要使用 `Async.Start` 或其他需要的函式 `Async<unit>`，請考慮是否要捨棄結果。避免只為了符合型別簽章而捨棄結果。

Async. Async.RunSynchronously

執行非同步計算，並等候其在呼叫執行緒上的結果。如果計算產生一個例外狀況，則會傳播例外狀況。此呼叫正在封鎖中。

簽章：

```
computation: Async<'T> * timeout: ?int * cancellationToken: ?CancellationTok... -> 'T
```

使用時機：

- 如果需要，請在應用程式中只使用一次(在可執行檔的進入點)。
- 當您不在意效能，並且想要一次執行一組其他非同步作業時。

要注意的事項：

- 呼叫會 `Async.RunSynchronously` 封鎖呼叫的執行緒，直到執行完成為止。

Async. 開始

啟動會線上程集區中傳回的非同步計算 `unit`。不會等待其完成，並(或)觀察到例外狀況結果。開頭的嵌套計算 `Async.Start` 會與呼叫它們的父代計算分開啟動；其存留期不會系結至任何父代計算。如果取消父代計算，則不會取消任何子計算。

簽章：

```
computation: Async<unit> * cancellationToken: ?CancellationTok... -> unit
```

僅用於：

- 您有不產生結果及/或需要處理一項的非同步計算。
- 非同步計算完成時，您不需要知道。
- 您不在意非同步計算執行所在的執行緒。
- 您不需要留意或報告執行所產生的例外狀況。

要注意的事項：

- 從開始計算所引發的例外狀況 `Async.Start` 不會傳播至呼叫端。呼叫堆疊將會完全展開。
- 任何工作(例如 `printfn`) 啟動的呼叫都 `Async.Start` 不會導致程式執行的主要執行緒發生效果。

與 .NET 交互操作

您可以使用 .NET 程式庫或使用 `async/await` 樣式非同步程式設計的 c# 程式碼基底。因為 c# 和大部分的 .NET 程式庫都使用 `Task<TResult>` 和 `Task` 類型作為其核心抽象概念，而不是 `Async<'T>`，所以您必須在這兩種方法之間的界限之間進行非同步。

如何使用 .NET async 和 `Task<T>`

使用使用(的 .NET 非同步程式庫和程式碼基底 `Task<TResult>`，) 具有傳回值的非同步計算相當簡單，而且具備 F# 的內建支援。

您可以使用 `Async.AwaitTask` 函數來等候 .net 非同步計算：

```
let getValueFromLibrary param =
    async {
        let! value = DotNetLibrary.GetValueAsync param |> Async.AwaitTask
        return value
    }
```

您可以使用 `Async.StartAsTask` 函數，將非同步計算行程給 .net 呼叫端：

```
let computationForCaller param =
    async {
        let! result = getAsyncResult param
        return result
    } |> Async.StartAsTask
```

如何使用 .NET async 和 `Task`

若要使用的 Api 使用 `Task` (也就是不會傳回值) 的 .net 非同步計算，您可能需要加入其他函式，以將轉換成 `Async<'T>` `Task`：

```
module Async =
    // Async<unit> -> Task
    let startTaskFromAsyncUnit (comp: Async<unit>) =
        Async.StartAsTask comp :> Task
```

已經有 `Async.AwaitTask` 接受 `Task` as 輸入的。使用這個與先前定義的函 `startTaskFromAsyncUnit` 式，您可以 `Task` 從 F # 非同步計算啟動和等候類型。

多執行緒的關聯性

雖然這篇文章中提到執行緒，但是有兩個重要事項要記住：

1. 非同步計算和執行緒之間沒有任何關聯性，除非在目前的執行緒上明確啟動。
2. F # 中的非同步程式設計不是多重執行緒的抽象概念。

例如，根據工作的本質而定，計算實際上可能會在其呼叫端的執行緒上執行。計算也可以線上程之間「跳躍」，以在一小段時間內進行處理，以在「等候」(期間(例如，當網路通話處於傳輸)時執行有用的工作)。

雖然 F # 提供了一些功能，可讓您在目前的執行緒上啟動非同步計算 (或明確地不) 目前的執行緒上，非同步通常不會與特定的執行緒策略相關聯。

請參閱

- [F # 非同步程式設計模型](#)
- [Jet.com 的 F # 非同步指南](#)
- [適用於有趣和收益之非同步程式設計指南的 F #](#)
- [C # 中的非同步和 F #: C 中的非同步陷阱](#)

型別提供者

2021/3/5 • [Edit Online](#)

F# 型別提供者是一個元件，該元件會提供類型、屬性和方法讓您在程式中使用。型別提供者會產生所謂的 **提供型別**，這些型別是由 F# 編譯器所產生，並且是以外部資料源為基礎。

例如，SQL 的 F# 型別提供者可以產生代表關係資料庫中資料表和資料行的類型。事實上，這就是 [SQLProvider](#) 型別提供者所做的。

提供的類型取決於型別提供者的輸入參數。這類輸入可以是範例資料來源 (例如 JSON 架構檔案)、直接指向外部服務的 URL，或資料來源的連接字串。型別提供者也可以確保只有在需要時才會擴充類型的群組;也就是說，如果您的程式實際參考型別，就會展開它們。這樣就能以強類型的方式視需要直接整合大型的資訊空間，例如線上資料市場。

有生產力和清除的型別提供者

型別提供者有兩種形式：有生產力和清除。

有生產力型別提供者所產生的型別，可以做為 .NET 型別寫到產生的元件中。這可讓它們從其他元件中的程式碼使用。這表示，資料來源的具型別標記法通常必須是可使用 .NET 型別來表示的類型標記法。

清除型別提供者所產生的型別只能在其產生來源的元件或專案中使用。這些類型是暫時的;也就是說，它們不會寫入元件中，而且不能由其他元件中的程式碼使用。它們可能包含 *延遲* 的成員，可讓您從可能的無限資訊空間使用提供的類型。它們適用於使用大型和互連資料來源的一小部分。

一般使用的類型提供者

下列廣泛使用的程式庫包含不同用途的型別提供者：

- Fsharp.core 包括 JSON、XML、CSV 以及 HTML 檔案格式和資源的類型提供者。
- [SQLProvider](#) 可透過物件對應以及對這些資料來源的 F# LINQ 查詢，提供對關聯資料庫的強型別存取。
- [Fsharp.core](#) 有一組型別提供者，可用於在 F# 中，以編譯時間檢查的 t-sql。
- [Azure 儲存體型別提供者](#) 提供適用於 Azure Blob、資料表和佇列的類型，可讓您存取這些資源，而不需要在整個程式中指定資源名稱做為字串。
- [Fsharp.core](#) 包含 [GraphQLProvider](#)，它會根據 URL 所指定的 GraphQL 伺服器來提供類型。

您可以視需要 [建立自己的自訂型別提供者](#)，或由其他人建立的參考型別提供者。例如，假設您的組織的資料服務提供大量且不斷增加的具名資料集，各資料集都有自己的穩定資料結構描述。您可能選擇建立一個類型提供者，以強類型方式讀取結構描述並對程式設計人員呈現最新的可用資料集。

另請參閱

- [教學課程：建立型別提供者](#)
- [F# 語言參考](#)

教學課程：建立型別提供者

2021/3/5 • [Edit Online](#)

F # 中的型別提供者機制是其對資訊豐富程式設計支援的重要部分。本教學課程說明如何建立您自己的型別提供者，方法是逐步解說幾個簡單的型別提供者，以說明基本概念。如需 F # 中型別提供者機制的詳細資訊，請參閱 [類型提供者](#)。

F # 生態系統包含一系列常用的網際網路和企業資料服務的類型提供者。例如：

- Fsharp.core 包括 JSON、XML、CSV 和 HTML 檔案格式的類型提供者。
- [SQLProvider](#) 可透過物件對應以及對這些資料來源的 F # LINQ 查詢，提供 SQL 資料庫的強型別存取。
- [Fsharp.core](#) 有一組型別提供者，可用於在 F # 中，以編譯時間檢查的 t-sql。
- [Fsharp.core](#) 是一組較舊的型別提供者，僅供用來存取 SQL、Entity Framework、ODATA 和 WSDL Data services .NET Framework 程式設計。

您可以在必要時建立自訂型別提供者，也可以參考其他人所建立的型別提供者。例如，您的組織可能會有一個資料服務，可提供大量且不斷增加的命名資料集，每個資料集都有自己的穩定資料架構。您可以建立可讀取架構的型別提供者，並以強型別方式向程式設計師呈現目前的資料集。

開始之前

型別提供者機制主要是設計用來將穩定的資料和服務資訊空間插入 F # 程式設計體驗中。

這項機制並非設計用來插入資訊空間，其架構在程式執行期間會以與程式邏輯相關的方式進行變更。此外，此機制不是針對語言中繼程式設計所設計，即使該網域包含一些有效的用途也一樣。您應該只在必要時才使用此機制，而在開發型別提供者時會產生非常高的價值。

您應該避免撰寫無法使用架構的類型提供者。同樣地，您應該避免撰寫一般 (或甚至是現有) .NET 程式庫都已足夠的類型提供者。

開始之前，您可能會詢問下列問題：

- 您有資料來源的架構嗎？如果是，F # 和 .NET 類型系統的對應是什麼？
- 您可以使用現有的 (動態型別) API 做為您的實作為起點嗎？
- 您和您的組織是否有足夠的型別提供者使用，讓您有價值的撰寫？一般的 .NET 程式庫是否符合您的需求？
- 您的架構會變更多少？
- 在編碼期間是否會變更？
- 程式碼撰寫會話之間會變更嗎？
- 它會在程式執行期間變更嗎？

型別提供者最適合的情況是，架構在執行時間和編譯器代碼存留期間都是穩定的情況。

簡單類型提供者

此範例為 HelloWorldTypeProvider，類似于 [examples](#) [F # 型別提供者 SDK](#) 目錄中的範例。提供者可以使用包含 100 清除類型的「類型空間」，如下列程式碼所示，使用 F # 簽章語法，並省略所有的詳細資料 `Type1`。如需已清

除之類型的詳細資訊，請參閱本主題稍後的已 [清除提供類型的詳細資料](#)。

```
namespace Samples.HelloWorldTypeProvider

type Type1 =
    /// This is a static property.
    static member StaticProperty : string

    /// This constructor takes no arguments.
    new : unit -> Type1

    /// This constructor takes one argument.
    new : data:string -> Type1

    /// This is an instance property.
    member InstanceProperty : int

    /// This is an instance method.
    member InstanceMethod : x:int -> char

    nested type NestedType =
        /// This is StaticProperty1 on NestedType.
        static member StaticProperty1 : string
        ...
        /// This is StaticProperty100 on NestedType.
        static member StaticProperty100 : string

type Type2 =
...

type Type100 =
...
```

請注意，所提供的類型和成員集合是靜態已知的。此範例不會利用提供者的能力來提供相依赖于架構的類型。下列程式碼概述型別提供者的實作為，詳細資料將在本主題的後續章節中討論。

WARNING

這段程式碼和線上範例之間可能有差異。


```

namespace Samples.FSharp.HelloWorldTypeProvider

open System
open System.Reflection
open ProviderImplementation.ProvidedTypes
open FSharp.Core.CompilerServices
open FSharp.Quotations

// This type defines the type provider. When compiled to a DLL, it can be added
// as a reference to an F# command-line compilation, script, or project.
[<TypeProvider>]
type SampleTypeProvider(config: TypeProviderConfig) as this =

    // Inheriting from this type provides implementations of ITypeProvider
    // in terms of the provided types below.
    inherit TypeProviderForNamespaces(config)

    let namespaceName = "Samples.HelloWorldTypeProvider"
    let thisAssembly = Assembly.GetExecutingAssembly()

    // Make one provided type, called TypeN.
    let makeOneProvidedType (n:int) =
        ...
    // Now generate 100 types
    let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]

    // And add them to the namespace
    do this.AddNamespace(namespaceName, types)

[<assembly:TypeProviderAssembly>]
do()

```

若要使用此提供者，請開啟 Visual Studio 的個別實例，建立 F# 腳本，然後使用 `#r`，從您的腳本加入提供者的參考，如下列程式碼所示：

```

#r @".\bin\Debug\Samples.HelloWorldTypeProvider.dll"

let obj1 = Samples.HelloWorldTypeProvider.Type1("some data")

let obj2 = Samples.HelloWorldTypeProvider.Type1("some other data")

obj1.InstanceProperty
obj2.InstanceProperty

[ for index in 0 .. obj1.InstanceProperty-1 -> obj1.InstanceMethod(index) ]
[ for index in 0 .. obj2.InstanceProperty-1 -> obj2.InstanceMethod(index) ]

let data1 = Samples.HelloWorldTypeProvider.Type1.NestedType.StaticProperty35

```

然後，在 `Samples.HelloWorldTypeProvider` 型別提供者產生的命名空間底下尋找類型。

在重新編譯提供者之前，請確定您已關閉所有使用 provider DLL 之 Visual Studio 和 F# 互動的實例。否則，將會發生組建錯誤，因為輸出 DLL 將會被鎖定。

若要使用 `print` 語句來偵測這個提供者，請建立一個腳本來公開提供者的問題，然後使用下列程式碼：

```
fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

若要使用 Visual Studio 來偵測此提供者，請以系統管理認證開啟 Visual Studio 的開發人員命令提示字元，然後執行下列命令：

```
devenv.exe /debugexe fsc.exe -r:bin\Debug\HelloWorldTypeProvider.dll script.fsx
```

另一種方式是開啟 Visual Studio, 開啟 [偵錯工具] 功能表, 選擇 `Debug/Attach to process...`, 然後附加到 `devenv` 您正在編輯腳本的另一個進程。藉由使用這個方法, 您可以使用完整的 IntelliSense 和其他功能), 以互動方式將運算式輸入至第二個 (實例, 以便更輕鬆地以型別提供者的特定邏輯為目標。

您可以停用 Just My Code 的偵錯工具, 以更妥善識別產生的程式碼中的錯誤。如需如何啟用或停用這項功能的詳細資訊, 請參閱 [使用偵錯工具流覽程式碼](#)。此外, 您也可以開啟 `Debug` 功能表, 然後選擇 `Exceptions` 或選擇 `Ctrl + Alt + E` 鍵開啟對話方塊, 來設定第一個可能發生的例外狀況攔截 `Exceptions`。在該對話方塊中, `Common Language Runtime Exceptions` 選取下的 `Thrown` 核取方塊。

實作為型別提供者

本節將逐步引導您完成型別提供者執行的主要區段。首先, 您要定義自訂型別提供者本身的型別:

```
[<TypeProvider>]  
type SampleTypeProvider(config: TypeProviderConfig) as this =
```

這個型別必須是公用的, 而且您必須使用 `TypeProvider` 屬性來標記它, 如此一來, 當另一個 F# 專案參考包含型別的元件時, 編譯器就會辨識型別提供者。`Config` 參數是選擇性的, 如果有的話, 則會包含 F# 編譯器所建立之型別提供者實例的內容設定資訊。

接下來, 您會執行 `ITypeProvider` 介面。在此情況下, 您會使用 `TypeProviderForNamespaces` API 中的型別 `ProvidedTypes` 做為基底類型。此協助程式類型可以提供有限的立即提供命名空間集合, 其中每個命名空間都會直接包含有限數目的固定、立即提供類型。在此內容中, 提供者 *立即* 會產生型別, 即使它們不需要或使用也一樣。

```
inherit TypeProviderForNamespaces(config)
```

接下來, 定義指定所提供類型之命名空間的本機私用值, 並尋找型別提供者元件本身。此元件稍後會用來做為所提供之已清除類型的邏輯父系型別。

```
let namespaceName = "Samples.HelloWorldTypeProvider"  
let thisAssembly = Assembly.GetExecutingAssembly()
```

接著, 建立一個函式來提供每個類型的 `Type1 .. Type100`。本主題稍後將更詳細地說明此函式。

```
let makeOneProvidedType (n:int) = ...
```

接下來, 產生 100 提供的類型:

```
let types = [ for i in 1 .. 100 -> makeOneProvidedType i ]
```

接下來, 將類型新增為所提供的命名空間:

```
do this.AddNamespace(namespaceName, types)
```

最後, 加入元件屬性, 表示您正在建立型別提供者 DLL:

```
[<assembly:TypeProviderAssembly>]  
do()
```

提供一種類型和其成員

`makeOneProvidedType` 函數會執行提供其中一種類型的實際工作。

```
let makeOneProvidedType (n:int) =  
...
```

此步驟說明此函數的執行。首先，建立提供的型別 (例如 `Type1`、when `n = 1` 或 `Type57` (當 `n = 57`) 時)。

```
// This is the provided type. It is an erased provided type and, in compiled code,  
// will appear as type 'obj'.  
let t = ProvidedTypeDefinition(thisAssembly, namespaceName,  
                                "Type" + string n,  
                                baseType = Some typeof<obj>)
```

您應該注意下列幾點：

- 這會清除提供的類型。因為您指出基底類型為 `obj`，所以實例會在已編譯的程式碼中顯示為 `obj` 類型的值。
- 當您指定非巢狀型別時，必須指定元件和命名空間。針對已清除的類型，元件應該是型別提供者元件本身。

接下來，將 XML 檔加入至類型。這是延遲的檔，也就是當主機編譯器需要時，視需要計算。

```
t.AddXmlDocDelayed (fun () -> $""This provided type {"Type" + string n}"" )
```

接下來，您要將提供的靜態屬性加入至類型：

```
let staticProp = ProvidedProperty(propertyName = "StaticProperty",  
                                   propertyType = typeof<string>,  
                                   isStatic = true,  
                                   getterCode = (fun args -> <@@ "Hello!" @@>))
```

取得此屬性一律會評估為字串 "Hello !"。 `GetterCode` 屬性(property)使用 F# 引號，代表主機編譯器為了取得屬性而產生的程式碼。如需有關引號的詳細資訊，請參閱 [Code 引號 \(F#\)](#)。

將 XML 檔集加入至屬性。

```
staticProp.AddXmlDocDelayed(fun () -> "This is a static property")
```

現在將提供的屬性附加至提供的型別。您必須將提供的成員附加至一種類型。否則，成員將永遠無法存取。

```
t.AddMember staticProp
```

現在，建立未採用任何參數的提供的函式。

```
let ctor = ProvidedConstructor(parameters = [ ],  
                                invokeCode = (fun args -> <@@ "The object data" :> obj @@>))
```

函式的會傳回 `InvokeCode` F # 引號，代表呼叫函式時，主機編譯器所產生的程式碼。例如，您可以使用下列的函式：

```
new Type10()
```

將會使用基礎資料「物件資料」來建立所提供類型的實例。加上引號的程式碼包含轉換成 `obj`，因為該型別會將這個提供的型別抹除 (當您宣告提供的型別) 時所指定的型別。

將 XML 檔新增至函式，並將提供的函式新增至提供的類型：

```
ctor.AddXmlDocDelayed(fun () -> "This is a constructor")

t.AddMember ctor
```

建立第二個提供的函式，採用一個參數：

```
let ctor2 =
    ProvidedConstructor(parameters = [ ProvidedParameter("data",typeof<string>) ],
        invokeCode = (fun args -> <@@ (%(args.[0]) : string) :> obj @@>))
```

函 `InvokeCode` 式的會再次傳回 F # 引號，代表主機編譯器針對方法呼叫所產生的程式碼。例如，您可以使用下列的函式：

```
new Type10("ten")
```

所提供型別的實例會以基礎資料 "十" 建立。您可能已經注意到函數會傳回 `InvokeCode` 一個引號。此函數的輸入是運算式的清單，每個函式參數都有一個。在此情況下，會在中提供代表單一參數值的運算式 `args.[0]`。呼叫函式的程式碼會將傳回值強制轉型為已清除的型別 `obj`。當您將第二個提供的函式加入至類型之後，您可以建立提供的實例屬性：

```
let instanceProp =
    ProvidedProperty(propertyName = "InstanceProperty",
        propertyType = typeof<int>,
        getterCode= (fun args ->
            <@@ ((%(args.[0]) : obj) :?> string).Length @@>))
instanceProp.AddXmlDocDelayed(fun () -> "This is an instance property")
t.AddMember instanceProp
```

取得此屬性會傳回字串的長度，也就是代表物件。`GetterCode` 屬性會傳回 F # 引號，以指定主機編譯器產生的程式碼以取得屬性。如同 `InvokeCode`，函數會傳回 `GetterCode` 引號。主機編譯器會使用引數清單來呼叫這個函數。在此情況下，引數只會包含代表要呼叫 `getter` 之實例的單一運算式，您可以使用來存取此實例 `args.[0]`。接著會將的實 `GetterCode` 接合至已清除之類型的結果引號 `obj` 中，並使用 `cast` 來滿足編譯器用來檢查物件是否為字串之類型的機制。下一個部分會 `makeOneProvidedType` 提供具有一個參數的實例方法。

```
let instanceMeth =
    ProvidedMethod(methodName = "InstanceMethod",
        parameters = [ProvidedParameter("x",typeof<int>)],
        returnType = typeof<char>,
        invokeCode = (fun args ->
            <@@ ((%(args.[0]) : obj) :?> string).Chars(%(args.[1]) : int) @@>))

instanceMeth.AddXmlDocDelayed(fun () -> "This is an instance method")
// Add the instance method to the type.
t.AddMember instanceMeth
```

最後，建立包含100嵌套屬性的巢狀型別。建立此巢狀型別和其屬性會延遲，也就是依需求計算。

```
t.AddMembersDelayed(fun () ->
    let nestedType = ProvidedTypeDefinition("NestedType", Some typeof<obj>)

    nestedType.AddMembersDelayed (fun () ->
        let staticPropsInNestedType =
            [
                for i in 1 .. 100 ->
                    let valueOfTheProperty = "I am string " + string i

                    let p =
                        ProvidedProperty(propertyName = "StaticProperty" + string i,
                            propertyType = typeof<string>,
                            isStatic = true,
                            getterCode= (fun args -> <@@ valueOfTheProperty @@>))

                    p.AddXmlDocDelayed(fun () ->
                        $"This is StaticProperty{i} on NestedType")

                    p
            ]

        staticPropsInNestedType)

    [nestedType])
```

已清除提供類型的詳細資料

本節中的範例只會提供已 *清除的提供類型*，在下列情況下特別有用：

- 當您針對只包含資料和方法的資訊空間寫入提供者時。
- 當您撰寫的提供者，正確的執行時間型別語義對資訊空間的實際使用並不重要。
- 當您撰寫的提供者的資訊空間太大且互相連接時，為資訊空間產生真正的 .NET 型別，並不是技術上可行的。

在此範例中，會將每個提供的型別清除為類型 `obj`，而類型的所有用法在編譯的程式碼中會顯示為類型 `obj`。事實上，這些範例中的基礎物件是字串，但類型會顯示為 `System.Object` .net 編譯器代碼中的。如同抹除型別的所有使用方式，您可以使用明確的裝箱、取消清除和轉換來破壞清除的類型。在此情況下，使用物件時，可能會產生不正確 cast 例外狀況。提供者執行時間可以定義自己的私用表示型別，以協助防範 false 標記法。您無法在 F# 中定義清除的類型。可能只會清除提供的類型。您必須瞭解使用您的類型提供者的已清除型別，或提供已清除之類型的提供者，才能瞭解其實際和語義的後果。清除的類型沒有真正的 .NET 類型。因此，您無法對型別進行精確的反映，而且如果您使用執行時間轉換和依賴確切執行時間型別語義的其他技術，您可能會破壞清除的型別。清除的型別 Subversion 通常會在執行時間產生類型轉換例外狀況。

選擇已清除提供類型的標記法

針對已清除之提供類型的某些用途，不需要標記法。例如，已清除的提供類型可能只包含靜態屬性和成員，而不包含任何函式，而且沒有方法或屬性會傳回類型的實例。如果您可以觸達已清除之提供類型的實例，您必須考慮下列問題：

所提供類型的抹除為何？

- 抹除所提供的類型是類型在已編譯的 .NET 程式碼中的顯示方式。
- 在類型的繼承鏈中，抹除所提供的已清除類別類型一律是第一個未清除的基底類型。
- 所提供已清除介面類別型的抹除一律為 `System.Object`。

所提供類型的標記法為何？

- 已清除之提供類型的可能物件集合稱為其標記法。在本檔的範例中，所有已清除的所提供類型的表示

`Type1..Type100` 一律為字串物件。

提供之類型的所有表示都必須與所提供類型的抹除相容。(否則，F# 編譯器會在使用型別提供者時提供錯誤，否則會產生不正確無法驗證 .NET 程式碼。如果型別提供者傳回的程式碼提供了無效的表示方式，則該型別提供者無效。

您可以使用下列其中一種方法來選擇所提供物件的標記法，兩者都很常見：

- 如果您只是在現有的 .NET 型別上提供強型別包裝函式，則您的型別通常很適合用來清除該型別、使用該型別的實例做為標記法，或兩者都使用。當使用強型別版本時，該類型上的大部分現有方法仍有意義時，這個方法就很適合。
- 如果您想要建立與任何現有 .NET API 截然不同的 API，請建立執行時間類型，以做為所提供類型的抹除和標記法。

本檔中的範例會使用字串作為所提供物件的表示。通常可能適合使用其他物件作為標記法。例如，您可以使用字典作為屬性包：

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new Dictionary<string,obj>()) :> obj @@>))
```

或者，您可以在型別提供者中定義要在執行時間用來形成標記法的型別，以及一或多個執行時間作業：

```
type DataObject() =
    let data = Dictionary<string,obj>()
    member x.RuntimeOperation() = data.Count
```

接著，提供的成員可以建立此物件類型的實例：

```
ProvidedConstructor(parameters = [],
    invokeCode= (fun args -> <@@ (new DataObject()) :> obj @@>))
```

在這種情況下，您可以在建立時將此類型指定為，以 (選擇性地) 使用此類型做為類型抹除 `baseType`

`ProvidedTypeDefinition`：

```
ProvidedTypeDefinition(..., baseType = Some typeof<DataObject> )
...
ProvidedConstructor(..., InvokeCode = (fun args -> <@@ new DataObject() @@>), ...)
```

重要課程

上一節說明如何建立簡單的清除型別提供者，以提供一系列的類型、屬性和方法。本節也說明了抹除型別的概念，包括從型別提供者提供已清除類型的一些優點和缺點，以及已清除之類型的標記法。

使用靜態參數的型別提供者

即使提供者不需要存取任何本機或遠端資料，透過靜態資料將型別提供者參數化的功能也可提供許多有趣的案例。在本節中，您將瞭解一些將這類提供者結合在一起的基本技巧。

輸入已核取的 **Regex** 提供者

假設您想要在 [Regex](#) 提供下列編譯時期保證的介面中，針對包裝 .net 程式庫的正則運算式，執行型別提供者：

- 確認正則運算式是否有效。
- 根據正則運算式中的任何組名，提供相符專案的命名屬性。

本節說明如何使用型別提供者來建立 `RegexTyped` 正則運算式模式所參數化的型別，以提供這些優點。如果提供的模式無效，則編譯器會回報錯誤，而且型別提供者可以從模式中解壓縮群組，以便您可以使用相符的命名屬性來存取它們。當您設計型別提供者時，您應該考慮其公開的 API 對終端使用者的外觀，以及這種設計如何轉譯為 .NET 程式碼。下列範例顯示如何使用這類 API 來取得區碼的元件：

```
type T = RegexTyped< @"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)">
let reg = T()
let result = T.IsMatch("425-555-2345")
let r = reg.Match("425-555-2345").Group_AreaCode.Value //r equals "425"
```

下列範例顯示型別提供者如何轉譯這些呼叫：

```
let reg = new Regex(@"(?<AreaCode>^\d{3})-(?<PhoneNumber>\d{3}-\d{4}$)")
let result = reg.IsMatch("425-123-2345")
let r = reg.Match("425-123-2345").Groups["AreaCode"].Value //r equals "425"
```

請注意下列幾點：

- 標準 `Regex` 類型代表參數化 `RegexTyped` 型別。
- 此函式會 `RegexTyped` 導致呼叫 `Regex` 的函式，並傳入模式的靜態型別引數。
- 方法的結果 `Match` 會以標準 `Match` 類型表示。
- 每個命名群組都會產生提供的屬性，而且存取屬性會導致在相符的集合上使用索引子 `Groups`。

下列程式碼是執行這類提供者的邏輯核心，此範例會省略將所有成員新增至提供的類型。如需每個已加入成員的詳細資訊，請參閱本主題稍後的適當章節。如需完整的程式碼，請從 CodePlex 網站上的 [F # 3.0 範例套件](#) 下載範例。

```

namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

            match parameterValues with
            | [| :? string as pattern|] ->

                // Create an instance of the regular expression.
                //
                // This will fail with System.ArgumentException if the regular expression is not valid.
                // The exception will escape the type provider and be reported in client code.
                let r = System.Text.RegularExpressions.Regex(pattern)

                // Declare the typed regex provided type.
                // The type erasure of this type is 'obj', even though the representation will always be a Regex
                // This, combined with hiding the object methods, makes the IntelliSense experience simpler.
                let ty =
                    ProvidedTypeDefinition(
                        thisAssembly,
                        rootNamespace,
                        typeName,
                        baseType = Some baseTy)

                ...

                ty
            | _ -> failwith "unexpected parameter values"))

    do this.AddNamespace(rootNamespace, [regexTy])

[<TypeProviderAssembly>]
do ()

```

請注意下列幾點：

- 型別提供者採用兩個靜態參數： `pattern`，這是必要的， `options` 而且是選擇性的 (因為) 會提供預設值。
- 提供靜態引數之後，您可以建立正則運算式的實例。如果 `Regex` 的格式不正確，這個實例將會擲回例外狀況，並將此錯誤回報給使用者。
- 在 `DefineStaticParameters` 回调中，您要定義在提供引數之後將傳回的型別。
- 這段程式碼會將設定 `HideObjectMethods` 為 `true`，讓 `IntelliSense` 體驗維持簡化。這個屬性會讓 `Equals`、`GetHashCode`、`Finalize` 和 `GetType` 成員從提供的物件的 `IntelliSense` 清單中隱藏。
- 您可以使用 `obj` 做為方法的基底類型，但您會使用 `Regex` 物件做為此類型的運行時程表示法，如以下範

例所示。

- `Regex` `ArgumentException` 當正則運算式無效時，對此函數的呼叫會擲回。編譯器會攔截這個例外狀況，並在編譯時期或 Visual Studio 編輯器中，將錯誤訊息報告給使用者。這個例外狀況可讓您驗證正則運算式，而不需要執行應用程式。

以上定義的類型並不實用，因為它不包含任何有意義的方法或屬性。首先，新增靜態 `IsMatch` 方法：

```
let isMatch =
    ProvidedMethod(
        methodName = "IsMatch",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = typeof<bool>,
        isStatic = true,
        invokeCode = fun args -> <@@ Regex.IsMatch(%args.[0], pattern) @@>)

isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified input string."
ty.AddMember isMatch
```

先前的程式碼會定義方法 `IsMatch`，此方法會接受字串做為輸入並傳回 `bool`。唯一棘手的部分是在 `args` 定義內使用引數 `InvokeCode`。在此範例中，`args` 是代表這個方法之引數的引號清單。如果方法是實例方法，則第一個引數代表 `this` 引數。不過，針對靜態方法，引數全都只是方法的明確引數。請注意，引號值的類型應該符合指定的傳回類型（在此案例中為 `bool`）。另請注意，此程式碼 `AddXmlDoc` 會使用方法來確保提供的方法也有實用的檔，您可以透過 IntelliSense 提供這些檔。

接下來，加入實例 `Match` 方法。不過，這個方法應該會傳回所提供類型的值，以便以強型別 `Match` 方式來存取群組。因此，您會先宣告 `Match` 類型。因為這個型別取決於提供為靜態引數的模式，所以這個型別必須嵌套在參數化型別定義中：

```
let matchTy =
    ProvidedTypeDefinition(
        "MatchType",
        baseType = Some baseTy,
        hideObjectMethods = true)

ty.AddMember matchTy
```

然後，您可以將一個屬性新增至每個群組的比對類型。在執行時間，會以值表示比對 `Match`，因此定義屬性的引號必須使用 `Groups` 索引屬性來取得相關的群組。

```
for group in r.GetGroupNames() do
    // Ignore the group named 0, which represents all input.
    if group <> "0" then
        let prop =
            ProvidedProperty(
                propertyName = group,
                propertyType = typeof<Group>,
                getterCode = fun args -> <@@ ((%args.[0]:obj) :?)> Match.Groups[group] @@>)
            prop.AddXmlDoc($"Gets the "{group}" group from this match")
        matchTy.AddMember prop
```

同樣地，請注意，您要將 XML 檔新增至提供的屬性。另請注意，如果有提供函式，就可以讀取屬性 `GetterCode`，而且如果有提供函式，就可以撰寫屬性 `SetterCode`，因此產生的屬性是唯讀的。

現在您可以建立傳回此類型值的實例方法 `Match`：

```

let matchMethod =
    ProvidedMethod(
        methodName = "Match",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = matchTy,
        invokeCode = fun args -> <@@ ((%args.[0]:obj) :?> Regex).Match(%args.[1]) :> obj @@>)

matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this regular
expression"

ty.AddMember matchMeth

```

因為您要建立實例方法，所以 `args.[0]` 代表 `RegexTyped` 呼叫方法的實例，而 `args.[1]` 是輸入引數。

最後，提供一個函式，以便能夠建立所提供型別的實例。

```

let ctor =
    ProvidedConstructor(
        parameters = [],
        invokeCode = fun args -> <@@ Regex(pattern, options) :> obj @@>)

ctor.AddXmlDoc("Initializes a regular expression instance.")

ty.AddMember ctor

```

此函式只會清除建立標準的 .NET Regex 實例，這會再次封裝至物件，因為 `obj` 是所提供類型的抹除。有了這項變更，本主題稍早指定的範例 API 使用方式就會如預期般運作。以下是完整的程式碼和最後的程式碼：

```

namespace Samples.FSharp.RegexTypeProvider

open System.Reflection
open Microsoft.FSharp.Core.CompilerServices
open Samples.FSharp.ProvidedTypes
open System.Text.RegularExpressions

[<TypeProvider>]
type public CheckedRegexProvider() as this =
    inherit TypeProviderForNamespaces()

    // Get the assembly and namespace used to house the provided types.
    let thisAssembly = Assembly.GetExecutingAssembly()
    let rootNamespace = "Samples.FSharp.RegexTypeProvider"
    let baseTy = typeof<obj>
    let staticParams = [ProvidedStaticParameter("pattern", typeof<string>)]

    let regexTy = ProvidedTypeDefinition(thisAssembly, rootNamespace, "RegexTyped", Some baseTy)

    do regexTy.DefineStaticParameters(
        parameters=staticParams,
        instantiationFunction=(fun typeName parameterValues ->

            match parameterValues with
            | [| :? string as pattern|] ->

                // Create an instance of the regular expression.

                let r = System.Text.RegularExpressions.Regex(pattern)

                // Declare the typed regex provided type.

                let ty =
                    ProvidedTypeDefinition(
                        thisAssembly,
                        rootNamespace,

```

```

        typeName,
        baseType = Some baseTy)

ty.AddXmlDoc "A strongly typed interface to the regular expression '%s'"

// Provide strongly typed version of Regex.IsMatch static method.
let isMatch =
    ProvidedMethod(
        methodName = "IsMatch",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = typeof<bool>,
        isStatic = true,
        invokeCode = fun args -> <@@ Regex.IsMatch(%args.[0], pattern) @@>)

isMatch.AddXmlDoc "Indicates whether the regular expression finds a match in the specified
input string"

ty.AddMember isMatch

// Provided type for matches
// Again, erase to obj even though the representation will always be a Match
let matchTy =
    ProvidedTypeDefinition(
        "MatchType",
        baseType = Some baseTy,
        hideObjectMethods = true)

// Nest the match type within parameterized Regex type.
ty.AddMember matchTy

// Add group properties to match type
for group in r.GetGroupNames() do
    // Ignore the group named 0, which represents all input.
    if group <> "0" then
        let prop =
            ProvidedProperty(
                propertyName = group,
                propertyType = typeof<Group>,
                getterCode = fun args -> <@@ ((%args.[0]:obj) :?> Match).Groups.[group] @@>)
        prop.AddXmlDoc(sprintf @"Gets the ""%s"" group from this match" group)
        matchTy.AddMember(prop)

// Provide strongly typed version of Regex.Match instance method.
let matchMeth =
    ProvidedMethod(
        methodName = "Match",
        parameters = [ProvidedParameter("input", typeof<string>)],
        returnType = matchTy,
        invokeCode = fun args -> <@@ ((%args.[0]:obj) :?> Regex).Match(%args.[1]) :> obj @@>)
matchMeth.AddXmlDoc "Searches the specified input string for the first occurrence of this
regular expression"

ty.AddMember matchMeth

// Declare a constructor.
let ctor =
    ProvidedConstructor(
        parameters = [],
        invokeCode = fun args -> <@@ Regex(pattern) :> obj @@>)

// Add documentation to the constructor.
ctor.AddXmlDoc "Initializes a regular expression instance"

ty.AddMember ctor

ty
| _ -> failwith "unexpected parameter values"))

do this.AddNamespace(rootNamespace [regexTy])

```

```
do this.AddNamespace(typeof(T).Assembly)
[<TypeProviderAssembly>]
do ()
```

重要課程

本節說明如何建立可在其靜態參數上運作的型別提供者。提供者會檢查靜態參數，並根據其值提供作業。

由本機資料支援的型別提供者

通常，您可能會想要型別提供者根據靜態參數，以及本機或遠端系統的資訊來呈現 Api。本節將討論以本機資料（例如本機資料檔案）為基礎的型別提供者。

Simple CSV File Provider

簡單的範例，請考慮使用以逗號分隔值 (CSV) 格式來存取科學資料的型別提供者。本節假設 CSV 檔案包含標頭資料列，後面接著浮點數資料，如下表所示：

II (II)	II (I)
50.0	3.7
100.0	5.2
150.0	6.4

本節說明如何提供型別，讓您用來取得具有 `Distance` 型別屬性 `float<meter>` 和型別屬性的資料列 `Time` `float<second>`。為了簡單起見，會進行下列假設：

- 標頭名稱不是單位，或格式為 "Name (unit) " 且不包含逗號。
- 單位是所有系統國際 (SI) 單位，與 [fsharp.core.UnitNames 模組 \(F #\)](#) 模組定義。
- 單位都是簡單的 (例如，計量)，而不是複合 (例如，計量/秒)。
- 所有資料行都包含浮點數據。

更完整的提供者會放寬這些限制。

同樣地，第一步是考慮 API 的外觀。假設有一個包含先前資料表內容的 `info.csv` 檔案 (採用逗號分隔的格式)，則提供者的使用者應該可以編寫類似下列範例的程式碼：

```
let info = new MiniCsv<"info.csv">()
for row in info.Data do
let time = row.Time
printfn $"{float time}"
```

在此情況下，編譯器應該將這些呼叫轉換成如下列範例所示的內容：

```
let info = new CsvFile("info.csv")
for row in info.Data do
let (time:float) = row.[1]
printfn $"%f{float time}"
```

最佳轉譯需要型別提供者 `CsvFile` 在型別提供者的元件中定義實數型別。型別提供者通常依賴一些協助程式類型和方法來包裝重要的邏輯。因為量值會在執行時間清除，所以您可以使用做為資料 `float[]` 列的清除型別。編譯器會將不同的資料行視為具有不同的量數值型別。例如，在我們的範例中，第一個資料行的類型為

`float<meter>`，而第二個數據行有 `float<second>`。不過，已清除的表示可以維持相當簡單。

下列程式碼顯示執行的核心。

```
// Simple type wrapping CSV data
type CsvFile(filename) =
    // Cache the sequence of all data lines (all lines but the first)
    let data =
        seq {
            for line in File.ReadAllLines(filename) |> Seq.skip 1 ->
                line.Split(',') |> Array.map float
        }
    |> Seq.cache
    member _.Data = data

[<TypeProvider>]
type public MiniCsvProvider(cfg:TypeProviderConfig) as this =
    inherit TypeProviderForNamespaces(cfg)

    // Get the assembly and namespace used to house the provided types.
    let asm = System.Reflection.Assembly.GetExecutingAssembly()
    let ns = "Samples.FSharp.MiniCsvProvider"

    // Create the main provided type.
    let csvTy = ProvidedTypeDefinition(asm, ns, "MiniCsv", Some(typeof<obj>))

    // Parameterize the type by the file to use as a template.
    let filename = ProvidedStaticParameter("filename", typeof<string>)
    do csvTy.DefineStaticParameters([filename], fun tyName [| :? string as filename |] ->

        // Resolve the filename relative to the resolution folder.
        let resolvedFilename = Path.Combine(cfg.ResolutionFolder, filename)

        // Get the first line from the file.
        let headerLine = File.ReadLines(resolvedFilename) |> Seq.head

        // Define a provided type for each row, erasing to a float[].
        let rowTy = ProvidedTypeDefinition("Row", Some(typeof<float[]>))

        // Extract header names from the file, splitting on commas.
        // use Regex matching to get the position in the row at which the field occurs
        let headers = Regex.Matches(headerLine, "[^,]+")

        // Add one property per CSV field.
        for i in 0 .. headers.Count - 1 do
            let headerText = headers.[i].Value

            // Try to decompose this header into a name and unit.
            let fieldName, fieldTy =
                let m = Regex.Match(headerText, @"(?<field>.+)\s((?<unit>.+)\s)")
                if m.Success then

                    let unitName = m.Groups["unit"].Value
                    let units = ProvidedMeasureBuilder.Default.SI unitName
                    m.Groups["field"].Value, ProvidedMeasureBuilder.Default.AnnotateType(typeof<float>,
[units])

                else
                    // no units, just treat it as a normal float
                    headerText, typeof<float>

            let prop =
                ProvidedProperty(fieldName, fieldTy,
                    getterCode = fun [row] -> <@@ (%row:float[]).[i] @@>)

            // Add metadata that defines the property's location in the referenced file.
            prop.AddDefinitionLocation(1, headers.[i].Index + 1, filename)
            rowTy.AddMember(prop)
```

```

// Define the provided type, erasing to CsvFile.
let ty = ProvidedTypeDefinition(asm, ns, tyName, Some(typeof<CsvFile>))

// Add a parameterless constructor that loads the file that was used to define the schema.
let ctor0 =
    ProvidedConstructor([],
        invokeCode = fun [] -> <@@ CsvFile(resolvedFilename) @@>)
ty.AddMember ctor0

// Add a constructor that takes the file name to load.
let ctor1 = ProvidedConstructor([ProvidedParameter("filename", typeof<string>)],
    invokeCode = fun [filename] -> <@@ CsvFile(%%filename) @@>)
ty.AddMember ctor1

// Add a more strongly typed Data property, which uses the existing property at runtime.
let prop =
    ProvidedProperty("Data", typedefof<seq<_>>.MakeGenericType(rowTy),
        getterCode = fun [csvFile] -> <@@ (%%csvFile:CsvFile).Data @@>)
ty.AddMember prop

// Add the row type as a nested type.
ty.AddMember rowTy
ty)

// Add the type to the namespace.
do this.AddNamespace(ns, [csvTy])

```

請注意下列有關執行的重點：

- 多載的函式允許讀取具有相同架構的原始檔案或。當您針對本機或遠端資料源撰寫型別提供者時，此模式很常見，而此模式允許使用本機檔案做為遠端資料的範本。
- 您可以使用傳遞至型別提供者函式的 [TypeProviderConfig](#) 值來解析相對檔案名。
- 您可以使用 `AddDefinitionLocation` 方法來定義所提供屬性的位置。因此，如果您在 `Go To Definition` 提供的屬性上使用，則會在 Visual Studio 中開啟 CSV 檔案。
- 您可以使用 `ProvidedMeasureBuilder` 類型來查閱 SI 單位，並產生相關的 `float<_>` 類型。

重要課程

本節說明如何使用包含在資料來源本身的簡單架構來建立本機資料來源的類型提供者。

更進一步

下列各節包含進一步研究的建議。

查看已清除類型的已編譯器代碼

為了讓您瞭解如何使用型別提供者對應至發出的程式碼，請使用本主題稍早所使用的來查看下列函式

`HelloWorldTypeProvider` 。

```

let function1 () =
    let obj1 = Samples>HelloWorldTypeProvider.Type1("some data")
    obj1.InstanceProperty

```

以下是使用 `ildasm.exe` 所反向組譯的產生程式碼的影像：

```

.class public abstract auto ansi sealed Module1
extends [mscorlib]System.Object
{
    .custom instance void [FSharp.Core]Microsoft.FSharp.Core.CompilationMappingAttribute::ctor(valuetype [FSharp.Core]Microsoft.FSharp.Core.SourceConstructFlags)
    = ( 01 00 07 00 00 00 00 00 )
    .method public static int32 function1() cil managed
    {
        // Code size          24 (0x18)
        .maxstack 3
        .locals init ([0] object obj1)
        IL_0000: nop
        IL_0001: ldstr      "some data"
        IL_0006: unbox.any [mscorlib]System.Object
        IL_000b: stloc.0
        IL_000c: ldloc.0
        IL_000d: call      !!0 [FSharp.Core_2]Microsoft.FSharp.Core.LanguagePrimitives/IntrinsicFunctions::UnboxGeneric<string>(object)
        IL_0012: callvirt instance int32 [mscorlib_3]System.String::get_Length()
        IL_0017: ret
    } // end of method Module1::function1

} // end of class Module1

```

如範例所示，所有提及的型別 `Type1` 和 `InstanceProperty` 屬性都已清除，只留下相關執行時間類型的作業。

型別提供者的設計和命名慣例

撰寫型別提供者時，請觀察下列慣例。

連接通訊協定的提供者 一般而言，資料和服務連接通訊協定（例如 OData 或 SQL 連接）的最大提供者 Dll 名稱應該以 `TypeProvider` 或結尾 `TypeProviders`。例如，使用類似下列字串的 DLL 名稱：

```
Fabrikam.Management.BasicTypeProviders.dll
```

確定您提供的類型是對應命名空間的成員，並指出您所執行的連接通訊協定：

```

Fabrikam.Management.BasicTypeProviders.WmiConnection<...>
Fabrikam.Management.BasicTypeProviders.DataProtocolConnection<...>

```

一般程式碼撰寫的公用程式提供者。 針對正則運算式之類的公用程式類型提供者，型別提供者可能是基底程式庫的一部分，如下列範例所示：

```
#r "Fabrikam.Core.Text.Utilities.dll"
```

在此情況下，所提供的類型會根據一般的 .NET 設計慣例出現在適當的時間點：

```

open Fabrikam.Core.Text.RegexTyped

let regex = new RegexTyped<"a+b+a+b+">()

```

單一資料來源。 某些型別提供者會連接到單一專用資料來源，並只提供資料。在此情況下，您應該卸載

`TypeProvider` 尾碼並使用一般的 .net 命名慣例：

```

#r "Fabrikam.Data.Freebase.dll"

let data = Fabrikam.Data.Freebase.Astronomy.Asteroids

```

如需詳細資訊，請參閱 `GetConnection` 本主題稍後所述的設計慣例。

型別提供者的設計模式

下列各節描述撰寫型別提供者時，您可以使用的設計模式。

`GetConnection` 設計模式

大部分的类型別提供者都應該撰寫成使用 `GetConnection` `FSharp.Data.TypeProviders.dll` 中的型別提供者所使用的模式，如下列範例所示：

```
#r "Fabrikam.Data.WebDataStore.dll"

type Service = Fabrikam.Data.WebDataStore<...static connection parameters...>

let connection = Service.GetConnection(...dynamic connection parameters...)

let data = connection.Astronomy.Asteroids
```

遠端資料和服務支援的型別提供者

在您建立由遠端資料和服務支援的型別提供者之前，您必須考慮連線程式設計中固有的一系列問題。這些問題包括下列考慮：

- 架構對應
- 架構變更存在時的活動和失效
- 架構快取
- 資料存取作業的非同步執行
- 支援查詢，包括 LINQ 查詢
- 認證和驗證

本主題不會進一步探索這些問題。

其他撰寫技巧

當您撰寫自己的型別提供者時，您可能會想要使用下列其他技巧。

視需要建立類型和成員

`ProvidedType` API 有延遲的 `AddMember` 版本。

```
type ProvidedType =
    member AddMemberDelayed : (unit -> MemberInfo) -> unit
    member AddMembersDelayed : (unit -> MemberInfo list) -> unit
```

這些版本是用來建立隨選的類型空間。

提供陣列類型和泛型型別具現化

您會將提供的成員（其簽章包含陣列類型、`byref` 型別和泛型型別的具現化）使用 `normal` `MakeArrayType`、`MakePointerType` 和的 `MakeGenericType` 任何實例 `Type`（包括）`ProvidedTypeDefinitions`。

NOTE

在某些情況下，您可能必須在中使用 helper `ProvidedTypeBuilder.MakeGenericType`。如需詳細資訊，請參閱 [類型提供者 SDK 檔](#)。

提供量值注釋的單位

ProvidedTypes API 提供提供量值附注的協助程式。例如，若要提供型別 `float<kg>`，請使用下列程式碼：

```
let measures = ProvidedMeasureBuilder.Default
let kg = measures.SI "kilogram"
let m = measures.SI "meter"
let float_kg = measures.AnnotateType(typeof<float>,[kg])
```

若要提供類型 `Nullable<decimal<kg/m^2>>`，請使用下列程式碼：

```
let kgpm2 = measures.Ratio(kg, measures.Square m)
let dkgpm2 = measures.AnnotateType(typeof<decimal>,[kgpm2])
let nullableDecimal_kgpm2 = typedefof<System.Nullable<_>>.MakeGenericType [|dkgpm2 |]
```

存取 Project-Local 或 Script-Local 資源

型別提供者的每個實例都可以在 `TypeProviderConfig` 結構中指定一個值。此值包含提供者的「解析資料夾」(也就是編譯的專案資料夾或包含腳本)的目錄、參考元件的清單，以及其他資訊。

失效

提供者可以引發失效信號，通知 F# 語言服務架構假設可能已變更。當發生失效時，如果 Visual Studio 中裝載提供者，則會重做 typecheck。當提供者裝載于 F# 互動或由 F# 編譯器 (fsc.exe) 時，就會忽略此信號。

快取架構資訊

提供者通常必須快取架構資訊的存取權。快取的資料應該使用以靜態參數形式提供的檔案名或使用者資料來儲存。架構快取的範例是 `LocalSchemaFile` 元件中型別提供者的參數 `FSharp.Data.TypeProviders`。在這些提供者的執行中，這個靜態參數會指示型別提供者使用指定的本機檔案中的架構資訊，而不是透過網路來存取資料來源。若要使用快取的架構資訊，您也必須將靜態參數設定 `ForceUpdate` 為 `false`。您可以使用類似的技術來啟用線上和離線資料存取。

支援元件

當您編譯或檔案時 `.dll` `.exe`，產生之類型的支援 `.dll` 檔會以靜態方式連結到產生的元件中。建立此連結的方法是將中繼語言 (IL) 型別定義，以及從支援元件到最終元件的任何 managed 資源複製。當您使用 F# 互動時，不會複製支援 `.dll` 檔案，而會改為直接載入 F# 互動的進程。

來自型別提供者的例外狀況和診斷

來自所提供類型之所有成員的所有使用，可能會擲回例外狀況。在所有情況下，如果型別提供者擲回例外狀況，則主機編譯器會將錯誤屬性指定給特定的類型提供者。

- 型別提供者例外狀況絕不會導致編譯器內部錯誤。
- 類型提供者無法報告警告。
- 當型別提供者裝載于 F# 編譯器、F# 開發環境或 F# 互動時，會攔截到該提供者的所有例外狀況。
Message 屬性一律是錯誤文字，且不會顯示任何堆疊追蹤。如果您即將擲回例外狀況，您可以擲回下列範例：`System.NotSupportedException`、`System.IO.IOException`、`System.Exception`。

提供產生的類型

到目前為止，本檔已說明如何提供清除的類型。您也可以使用 F# 中的型別提供者機制來提供產生的型別，這些型別會以真實的 .NET 型別定義新增至使用者的程式。您必須使用類型定義來參考所產生的所提供類型。

```
open Microsoft.FSharp.TypeProviders

type Service = ODataService<"http://services.odata.org/Northwind/Northwind.svc/">
```

屬於 F# 3.0 版本的 ProvidedTypes-0.2 協助程式程式碼，只有有限的支援提供產生的類型。下列語句對於產生的型別定義而言必須是 true：

- `isErased` 必須設為 `false`。
- 產生的類型必須加入至新建立的 `ProvidedAssembly()`，這表示產生的程式碼片段的容器。
- 提供者的元件必須具有在磁片上具有相符 .dll 檔案的實際支援 .NET .dll 檔案。

規則和限制

當您撰寫型別提供者時，請記住下列規則和限制。

提供的類型必須可以連線

所有提供的類型都應該可從非巢狀型別連接。呼叫函式或的呼叫時，會提供非巢狀型別

`TypeProviderForNamespaces` `AddNamespace`。例如，如果提供者提供型別 `StaticClass.P : T`，您必須確定 `T` 是非嵌套型別或在其中嵌套。

例如，某些提供者有一個靜態類別，例如 `DataTypes` 包含這些 `T1, T2, T3, ...` 類型。否則，此錯誤表示找到元件 `A` 中的類型 `T` 的參考，但在該元件中找不到該類型。如果出現此錯誤，請確認您的所有子類型都可以從提供者類型中取得。注意：這些型別 `T1, T2, T3...` 稱為「動態」類型。請記得將它們放在可存取的命名空間或父系型別中。

型別提供者機制的限制

F# 中的型別提供者機制有下列限制：

- F# 中型別提供者的基礎結構不支援提供的泛型型別或提供的泛型方法。
- 機制不支援具有靜態參數的巢狀型別。

開發秘訣

在開發過程中，您可能會發現下列秘訣很有說明：

執行 Visual Studio 的兩個實例

您可以在一個實例中開發型別提供者，並在另一個實例中測試該提供者，因為測試 IDE 會對 .dll 檔案進行鎖定，防止重建型別提供者。因此，您必須在第一個實例內建提供者時，關閉 Visual Studio 的第二個實例，然後您必須在建立提供者之後重新開啟第二個實例。

使用 fsc.exe 調用的偵錯工具型別提供者

您可以使用下列工具來叫用型別提供者：

- `fsc.exe` (F# 命令列編譯器)
- `fsi.exe` (F# 互動編譯器)
- `devenv.exe` (Visual Studio)

您通常可以在測試腳本檔案上使用 `fsc.exe`，以最輕鬆的方式來偵測型別提供者 (例如 .fsx)。您可以從命令提示字元啟動偵錯工具。

```
devenv /debugexe fsc.exe script.fsx
```

您可以使用列印到 `stdout` 記錄。

另請參閱

- [型別提供者](#)
- [型別提供者 SDK](#)

型別提供者安全性

2019/10/23 • [Edit Online](#)

型別提供者是所參考組件 (DII) 您 F# 專案或指令碼包含連接至外部資料來源，並呈現此型別資訊的程式碼 F# 類型的環境。一般而言，參考組件中的程式碼才會執行當您編譯和接著執行程式碼 (或在指令碼的情況下傳送的程式碼 F# 互動式)。不過，型別提供者組件會執行在 Visual Studio 中，當程式碼只是已在編輯器中瀏覽。這是因為型別提供者需要執行額外的資訊，加入編輯器，例如 快速諮詢工具提示，IntelliSense 完成等等。如此一來，有一些額外的安全性考量的型別提供者組件，因為它們會自動在 Visual Studio 處理序內執行。

安全性警告對話方塊

當第一次使用特定的型別提供者組件，Visual Studio 會顯示警告型別提供者是執行 [安全性] 對話方塊。Visual Studio 會載入型別提供者之前，它可讓您決定您是否信任此特定的提供者的機會。如果您信任來源的型別提供者，然後選取「我信任此型別提供者」。如果您不信任來源的型別提供者，然後選取 我不信任此型別提供者。」信任的提供者，讓它能夠在 Visual Studio 內執行，並提供 IntelliSense 及建置功能。但是，如果惡意型別提供者本身，執行其程式碼可能會危害您的電腦。

如果您的專案會包含參考型別提供者，您選擇在對話方塊中不信任的程式碼，然後在編譯時期，編譯器會報告錯誤，指出型別提供者不受信任。相依於不受信任的型別提供者的任何型別會以紅色波浪線表示。它可以安全地瀏覽程式碼編輯器中。

如果您決定要變更直接在 Visual Studio 中的信任設定，請執行下列步驟。

若要變更型別提供者信任設定

1. 在上 **Tools** 功能表上，選取 **Options**，然後展開 **F# Tools** 節點。
2. 選取 **Type Providers**，在型別提供者清單中，型別提供者信任時，請選取此核取方塊並清除那些您不信任的核取方塊。

另請參閱

- [類型提供者](#)

型別提供者疑難排解

2019/10/23 • [Edit Online](#)

本主題將告訴您，並提供您最有可能使用型別提供者時遇到的問題可能的解決方案。

可能發生問題的类型別提供者

如果您遇到問題，當您使用型別提供者時，您可以檢閱下表中的最常見的解決方案。

❗	💡
❗。型別提供者最適合的資料來源結構描述時很穩定。如果您新增資料表或資料行，或對該結構描述中的另一項變更，型別提供者無法自動辨識這些變更。	清除或重建專案。若要清除專案，選擇❗，■ <i>ProjectName</i> 功能表列上。若要重建專案，選擇❗，■ <i>ProjectName</i> 功能表列上。這些動作會重設所有的型別提供者狀態，並強制重新連線至資料來源，並取得更新的結構描述資訊的提供者。
❗。URL 或連接字串不正確、網路已關閉，或在資料來源「或」服務無法使用。	Web 服務或 OData 服務，您可以嘗試驗證 URL 是否正確，且該服務的 Internet Explorer 中的 URL。資料庫連接字串，您可以使用中的資料連接工具❗來確認連接字串是否有效，且資料庫可供使用。還原您的連線之後，您應該也要清除或重建專案，讓型別提供者會重新連線到網路。
❗。您必須針對資料來源] 或 [web 服務的有效權限。	SQL 連線，使用者名稱和連接字串或組態檔中指定的密碼必須是有效的資料庫。如果您使用 Windows 驗證，您必須具有資料庫的存取權。資料庫管理員可以識別所需的權限存取每個資料庫和資料庫內的每個項目。 針對 web 服務或資料服務，您必須使用適當的認證。大部分的型別提供者提供的 DataContext 物件，其中包含您可以使用適當的使用者名稱和存取金鑰設定的認證屬性。
❗。檔案的路徑不是有效的。	請確認路徑是否正確，且檔案存在。此外，您必須加上引號的路徑中的任何 backslashes 適當，或使用逐字字串或三重括住的字串。

另請參閱

- [類型提供者](#)

F # 5.0 的新功能

2021/3/5 • [Edit Online](#)

F # 5.0 在 F # 語言和 F# 互動中新增了幾項改進。它是以 .net 5 發行。

您可以從 [.net 下載頁面](#) 下載最新的 .net SDK。

開始使用

F # 5.0 適用於所有 .NET Core 散發套件和 Visual Studio 工具。如需詳細資訊，請參閱 [F # 入門\(英文\)](#) 以深入瞭解。

F # 腳本中的套件參考

F # 5 以語法提供 F # 腳本中封裝參考的支援 `#r "nuget:..."`。例如，請考慮下列套件參考：

```
#r "nuget: Newtonsoft.Json"

open Newtonsoft.Json

let o = { | X = 2; Y = "Hello" | }

printfn $"{JsonConvert.SerializeObject o}"
```

您也可以套件名稱之後提供明確的版本，如下所示：

```
#r "nuget: Newtonsoft.Json,11.0.1"
```

封裝參考支援具有原生相依性的套件，例如 ML.NET。

封裝參考也支援封裝參考相依的特殊需求的封裝 `.dll`。例如，[FParsec](#) 套件用來要求使用者在 `FParsecCS.dll` F# 互動中參考之前，先手動確定其相依的參考 `FParsec.dll`。這已不再需要，您可以參考封裝，如下所示：

```
#r "nuget: FParsec"

open FParsec

let test p str =
    match run p str with
    | Success(result, _, _) -> printfn $"Success: {result}"
    | Failure(msg, _, _) -> printfn $"Failure: {msg}"

test pfloat "1.234"
```

這項功能會實行 [F # 工具 RFC FST-1027](#)。如需封裝參考的詳細資訊，請參閱 [F# 互動](#) 教學課程。

字串插補

F # 插入字串與 c # 或 JavaScript 插入字串相當類似，因為它們可讓您在字串常值中的「漏洞」內撰寫程式碼。以下是基本範例：

```
let name = "Phillip"
let age = 29
printfn $"Name: {name}, Age: {age}"

printfn $"I think {3.0 + 0.14} is close to {System.Math.PI}!"
```

不過，F # 插入字串也允許型別插補（就像函式一樣 `sprintf` ），以強制插入內容內的運算式符合特定的型別。它會使用相同的格式規範。

```
let name = "Phillip"
let age = 29

printfn $"Name: %s{name}, Age: %d{age}"

// Error: type mismatch
printfn $"Name: %s{age}, Age: %d{name}"
```

在上述輸入的插補範例中，`%s` 需要插補是型別 `string`，而 `%d` 需要插補是 `integer`。

此外，任何任意的 F # 運算式（或運算式）都可以放在插補內容的側邊。您甚至可以撰寫更複雜的運算式，如下所示：

```
let str =
    $""The result of squaring each odd item in {[1..10]} is:
{
    let square x = x * x
    let isOdd x = x % 2 <> 0
    let oddSquares xs =
        xs
        |> List.filter isOdd
        |> List.map square
    oddSquares [1..10]
}
""
```

雖然我們不建議您這麼做，但我們不建議這麼做。

這項功能會實行 [F # RFC FS-1001](#)。

對 nameof 的支援

F # 5 支援 `nameof` 運算子，可解析其使用的符號，並在 F # 來源中產生其名稱。這在各種情況下都很有用，例如記錄，並可保護您的記錄免于原始程式碼中的變更。

```

let months =
    [
        "January"; "February"; "March"; "April";
        "May"; "June"; "July"; "August"; "September";
        "October"; "November"; "December"
    ]

let lookupMonth month =
    if (month > 12 || month < 1) then
        invalidArg (nameof month) (sprintf "Value passed in was %d." month)

    months.[month-1]

printfn $"{lookupMonth 12}"
printfn $"{lookupMonth 1}"
printfn $"{lookupMonth 13}"

```

最後一行將會擲回例外狀況，且會在錯誤訊息中顯示「月份」。

您可以採用幾乎每個 F# 結構的名稱：

```

module M =
    let f x = nameof x

printfn $"{M.f 12}"
printfn $"{nameof M}"
printfn $"{nameof M.f}"

```

有三個最後的新增專案是運算子運作方式的變更：加入 `nameof<'type-parameter>` 泛型型別參數的表單，以及在模式比對運算式中做為模式使用的功能 `nameof`。

採用運算子名稱可提供其來源字串。如果您需要編譯的表單，請使用運算子的編譯名稱：

```

nameof(+) // "+"
nameof op_Addition // "op_Addition"

```

採用型別參數的名稱需要稍微不同的語法：

```

type C<'TType> =
    member _.TypeName = nameof<'TType>

```

這與 `typeof<'T>` 和 `typedefof<'T>` 運算子類似。

F# 5 也加入了 `nameof` 可在運算式中使用的模式支援 `match`：

```

[<Struct; IsByRefLike>]
type RecordedEvent = { EventType: string; Data: ReadOnlySpan<byte> }

type MyEvent =
    | AData of int
    | BData of string

let deserialize (e: RecordedEvent) : MyEvent =
    match e.EventType with
    | nameof AData -> AData (JsonSerializer.Deserialize<int> e.Data)
    | nameof BData -> BData (JsonSerializer.Deserialize<string> e.Data)
    | t -> failwithf "Invalid EventType: %s" t

```

上述程式碼使用 'nameof'，而不是比對運算式中的字串常值。

這項功能會實行 [F # RFC FS-1003](#)。

開放式型別宣告

F # 5 也加入了對開放式型別宣告的支援。開啟的型別宣告就像在 c # 中開啟靜態類別一樣，除了有些不同的語法和一些稍微不同的行為，以配合 F # 語義。

使用開放式型別宣告，您可以使用 `open` 任何類型來公開其內部的靜態內容。此外，您可以 `open` F # 定義的等位和記錄來公開其內容。例如，如果您的模組中有定義聯集，而且想要存取其案例，但不想要開啟整個模組，這會很有用。

```
open type System.Math

let x = Min(1.0, 2.0)

module M =
    type DU = A | B | C

    let someOtherFunction x = x + 1

// Open only the type inside the module
open type M.DU

printfn $"{A}"
```

與 c # 不同的是，當您 `open type` 在兩種類型上公開具有相同名稱的成員時，最後一個型別中的成員會 `open` 遮蔽另一個名稱。這與已存在之遮蔽的 F # 語義一致。

這項功能會實行 [F # RFC FS-1068](#)。

內建資料類型的一致切割行為

配量內建 `FSharp.Core` 資料類型的行為 (陣列、清單、字串、2d 陣列、3d 陣列、4d 陣列) 在 F # 5 之前用來不一致。某些邊緣案例行為擲回例外狀況，而有些則不會。在 F # 5 中，所有內建類型現在都會針對無法產生的配量傳回空白配量：

```
let l = [ 1..10 ]
let a = [| 1..10 |]
let s = "hello!"

// Before: would return empty list
// F# 5: same
let emptyList = l.[-2..(-1)]

// Before: would throw exception
// F# 5: returns empty array
let emptyArray = a.[-2..(-1)]

// Before: would throw exception
// F# 5: returns empty string
let emptyString = s.[-2..(-1)]
```

這項功能會實行 [F # RFC FS-1077](#)。

固定-Fsharp.core 中3D 和4D 陣列的索引配量

F # 5.0 使用內建3D 和4D 陣列類型中具有固定索引的配量支援。

為了說明這一點，請考慮下列3D 陣列：

$z = 0 \mid x \setminus y \mid 0 \mid 1 \mid \mid \text{-----} \mid \text{---} \mid \text{---} \mid \mid 0 \mid 0 \mid 1 \mid \mid 1 \mid 2 \mid 3 \mid$

$z = 7 \mid x \setminus y \mid 0 \mid 1 \mid \mid \text{-----} \mid \text{---} \mid \text{---} \mid \mid 0 \mid 4 \mid 5 \mid \mid 1 \mid 6 \mid 7 \mid$

如果您想要從陣列解壓縮配量，該怎麼辦 `[| 4; 5 |]` ？這現在很簡單！

```
// First, create a 3D array to slice

let dim = 2
let m = Array3D.zeroCreate<int> dim dim dim

let mutable count = 0

for z in 0..dim-1 do
    for y in 0..dim-1 do
        for x in 0..dim-1 do
            m.[x,y,z] <- count
            count <- count + 1

// Now let's get the [4;5] slice!
m.[*, 0, 1]
```

這項功能會實行 [F # RFC FS-1077b](#)。

F # 引號的增強功能

F # 程式 [代碼引號](#) 現在能夠保留類型條件約束資訊。請考慮下列範例：

```
open FSharp.Linq.RuntimeHelpers

let eval q = LeafExpressionConverter.EvaluateQuotation q

let inline negate x = -x
// val inline negate: x: ^a -> ^a when ^a : (static member ( ~- ) : ^a -> ^a)

<@ negate 1.0 @> |> eval
```

函數所產生的條件約束 `inline` 會保留在程式碼引號中。`negate` 現在可以評估函數的 quoted 表單。

這項功能會實行 [F # RFC FS-1071](#)。

Applicative 計算運算式

目前使用的[計算運算式 \(CEs\)](#) 來建立「內容計算」的模型，或以更多功能的程式設計易懂術語 monadic 計算。

F # 5 引進了 applicative CEs，可提供不同的計算模型。Applicative CEs 允許更有效率的計算，前提是每個計算都是獨立的，且其結果會在結尾累積。當計算彼此獨立時，也會完整可並行，讓 CE 作者可以撰寫更有效率的程式庫。這項優點有一項限制，但不允許相依赖于先前計算值的計算。

下列範例顯示類型的基本 applicative CE `Result`。

```
// First, define a 'zip' function
module Result =
    let zip x1 x2 =
        match x1,x2 with
        | Ok x1res, Ok x2res -> Ok (x1res, x2res)
        | Error e, _ -> Error e
        | _, Error e -> Error e

// Next, define a builder with 'MergeSources' and 'BindReturn'
type ResultBuilder() =
    member _.MergeSources(t1: Result<'T,'U>, t2: Result<'T1,'U>) = Result.zip t1 t2
    member _.BindReturn(x: Result<'T,'U>, f) = Result.map f x

let result = ResultBuilder()

let run r1 r2 r3 =
    // And here is our applicative!
    let res1: Result<int, string> =
        result {
            let! a = r1
            and! b = r2
            and! c = r3
            return a + b - c
        }

    match res1 with
    | Ok x -> printfn $"{nameof res1} is: {x}"
    | Error e -> printfn $"{nameof res1} is: {e}"

let printApplicatives () =
    let r1 = Ok 2
    let r2 = Ok 3 // Error "fail!"
    let r3 = Ok 4

    run r1 r2 r3
    run r1 (Error "failure!") r3
```

如果您是目前在其程式庫中公開 CEs 的程式庫作者，還有一些需要注意的其他考慮事項。

這項功能會實行 [F # RFC FS-1063](#)。

介面可以在不同的泛型具現化中執行

您現在可以在不同的泛型具現化中執行相同的介面：

```
type IA<'T> =
    abstract member Get : unit -> 'T

type MyClass() =
    interface IA<int> with
        member x.Get() = 1
    interface IA<string> with
        member x.Get() = "hello"

let mc = MyClass()
let iaInt = mc :> IA<int>
let iaString = mc :> IA<string>

iaInt.Get() // 1
iaString.Get() // "hello"
```

這項功能會實行 [F # RFC FS-1031](#)。

預設介面成員耗用量

F # 5 可讓您使用預設實作為 [介面](#)。

請考慮以 c # 定義的介面，如下所示：

```
using System;

namespace CSharp
{
    public interface MyDimasd
    {
        public int Z => 0;
    }
}
```

您可以使用 F # 來使用它來執行介面的任何標準方法：

```
open CSharp

// You can implement the interface via a class
type MyType() =
    member _.M() = ()

    interface MyDim

let md = MyType() :> MyDim
printfn $"DIM from C#: %d{md.Z}"

// You can also implement it via an object expression
let md' = { new MyDim }
printfn $"DIM from C# but via Object Expression: %d{md'.Z}"
```

這可讓您安全地利用以新式 c # 撰寫的 c # 程式碼和 .NET 元件(當它們希望使用者能夠使用預設的實值)時。

這項功能會實行 [F # RFC FS-1074](#)。

使用可為 null 的實數值型別簡化 interop

[可為 null 的 \(值\) 類型](#) (稱為可為 null 的型別，在過去) 已經由 F # 所支援，但是與它們進行互動是因為您在

`Nullable` `Nullable<SomeType>` 每次想要傳遞值時都必須建立或包裝函式。現在，`Nullable<ThatValueType>` 如果目標型別相符，則編譯器會將實值型別隱含轉換成。現在可以執行下列程式碼：

```
#r "nuget: Microsoft.Data.Analysis"

open Microsoft.Data.Analysis

let dateTimes = PrimitiveDataFrameColumn<DateTime>("DateTimes")

// The following line used to fail to compile
dateTimes.Append(DateTime.Parse("2019/01/01"))

// The previous line is now equivalent to this line
dateTimes.Append(Nullable<DateTime>(DateTime.Parse("2019/01/01")))
```

這項功能會實行 [F # RFC FS-1075](#)。

預覽：反向索引

F # 5 也引進了允許反向索引的預覽。語法是 `^idx`。以下是您可以如何從清單結尾的元素1值：

```
let xs = [1..10]

// Get element 1 from the end:
xs.[^1]

// From the end slices

let lastTwoOldStyle = xs.[(xs.Length-2)..]

let lastTwoNewStyle = xs.[^1..]

lastTwoOldStyle = lastTwoNewStyle // true
```

您也可以為自己的類型定義反向索引。若要這樣做，您必須執行下列方法：

```
GetReverseIndex: dimension: int -> offset: int
```

以下是此類型的範例 `Span<'T>`：

```
open System

type Span<'T> with
    member sp.Slice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

    member sp.GetReverseIndex(_, offset: int) =
        sp.Length - offset

let printSpan (sp: Span<int>) =
    let arr = sp.ToArray()
    printfn $"{arr}"

let run () =
    let sp = [| 1; 2; 3; 4; 5 |].AsSpan()

    // Pre-# 5.0 slicing on a Span<'T>
    printSpan sp.[0..] // [|1; 2; 3; 4; 5|]
    printSpan sp[..3] // [|1; 2; 3|]
    printSpan sp.[1..3] // [|2; 3|]

    // Same slices, but only using from-the-end index
    printSpan sp.[..^0] // [|1; 2; 3; 4; 5|]
    printSpan sp[..^2] // [|1; 2; 3|]
    printSpan sp.[^4..^2] // [|2; 3|]

run() // Prints the same thing twice
```

這項功能會實行 [F # RFC FS-1076](#)。

預覽：計算運算式中的自訂關鍵字多載

計算運算式是程式庫和架構作者的強大功能。它們可讓您定義知名的成員，並形成您正在使用之網域的 DSL，以大幅提升元件的表達。

F # 5 新增了在計算運算式中多載自訂作業的預覽支援。它允許撰寫和使用下列程式碼：

```

open System

type InputKind =
    | Text of placeholder:string option
    | Password of placeholder: string option

type InputOptions =
    { Label: string option
      Kind : InputKind
      Validators : (string -> bool) array }

type InputBuilder() =
    member t.Yield(_) =
        { Label = None
          Kind = Text None
          Validators = [||] }

    [<CustomOperation("text")>]
    member this.Text(io, ?placeholder) =
        { io with Kind = Text placeholder }

    [<CustomOperation("password")>]
    member this.Password(io, ?placeholder) =
        { io with Kind = Password placeholder }

    [<CustomOperation("label")>]
    member this.Label(io, label) =
        { io with Label = Some label }

    [<CustomOperation("with_validators")>]
    member this.Validators(io, [<ParamArray>] validators) =
        { io with Validators = validators }

let input = InputBuilder()

let name =
    input {
        label "Name"
        text
        with_validators
            (String.IsNullOrWhiteSpace >> not)
    }

let email =
    input {
        label "Email"
        text "Your email"
        with_validators
            (String.IsNullOrWhiteSpace >> not)
            (fun s -> s.Contains "@")
    }

let password =
    input {
        label "Password"
        password "Must contains at least 6 characters, one number and one uppercase"
        with_validators
            (String.exists Char.IsUpper)
            (String.exists Char.IsDigit)
            (fun s -> s.Length >= 6)
    }

```

在這項變更之前，您可以撰寫 `InputBuilder` 型別，但是您無法使用它在範例中使用的方式。因為可以使用多載、選擇性參數和現在的型別，所以一切都如同 `System.ParamArray` 預期般運作。

這項功能會實行 [F # RFC FS-1056](#)。

F# 4.7 中的新增功能

2020/3/19 • [Edit Online](#)

F# 4.7 為 F# 語言添加了多項改進。

開始使用

F# 4.7 在所有 .NET 核心發行版本和視覺化工作室工具中均可用。[開始使用 F#](#)瞭解更多資訊。

語言版本

F# 4.7 編譯器引入了通過專案檔案中的屬性設置有效語言版本的能力：

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

可以將其 `4.6` 設置為值、`4.7`latest` 和 `preview`。預設值為 `latest`。

如果將其設置為 `preview`，編譯器將啟動編譯器中實現的所有 F# 預覽功能。

隱式收益

不再需要在 `yield` 可以推斷類型的陣列、清單、序列或計算運算式中應用關鍵字。在下面的示例中，兩個運算式都需要 F# 4.7 之前每個條目的 `yield` 語句：

```
let s = seq { 1; 2; 3; 4; 5 }

let daysOfWeek includeWeekend =
[
    "Monday"
    "Tuesday"
    "Wednesday"
    "Thursday"
    "Friday"
    if includeWeekend then
        "Saturday"
        "Sunday"
]
```

如果引入單個 `yield` 關鍵字，則所有其他項也必須已 `yield` 應用於該關鍵字。

在運算式中使用時，隱式收益不會啟動，該運算式 `yield!` 還用於執行諸如扁平序列之類的操作。在這些情況下，今天必須繼續 `yield` 使用。

萬用字元識別碼

在涉及類的 F# 代碼中，自識別碼必須始終在成員聲明中顯式。但是，在從未使用自識別碼的情況下，傳統上使用雙底線來指示無名的自識別碼是慣例。現在可以使用單個底線：

```
type C() =
    member _ .M() = ()
```

這也適用於 `for` 迴圈：

```
for _ in 1..10 do printfn "Hello!"
```

縮進鬆弛

在 F# 4.7 之前，主建構函式和靜態成員參數的縮進要求要求過高的縮進。它們現在只需要一個縮進範圍：

```
type OffsideCheck(a:int,
  b:int, c:int,
  d:int) = class end

type C() =
  static member M(a:int,
    b:int, c:int,
    d:int) = 1
```

4.6 中F#的新功能

2020/2/5 • [Edit Online](#)

F#4.6 新增多項語言的F#增強功能。

開始使用

F#4.6 適用於所有 .NET Core 發行版本和 Visual Studio 工具。[開始使用F#](#) 以深入瞭解。

匿名記錄

[匿名記錄](#)是4.6 中F#引進的F#一種新類型。它們是命名值的簡單匯總，不需要在使用之前宣告。您可以將它們宣告為結構或參考型別。它們預設是參考型別。

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

當您想要將實值型別分組，而且在效能相關的案例中運作時，它們也可以宣告為結構：

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    struct {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

這些功能非常強大，可用於許多案例。深入瞭解[匿名記錄](#)。

ValueOption 函式

4.5 中F#新增的 ValueOption 類型現在具有選項類型的「模組系統函式同位」。一些較常使用的範例如下：


```
// Multiply a value option by 2 if it has value
let xOpt = ValueSome 99
let result = xOpt |> ValueOption.map (fun v -> v * 2)

// Reverse a string if it exists
let strOpt = ValueSome "Mirror image"
let reverse (str: string) =
    match str with
    | null
    | "" -> ValueNone
    | s ->
        str.ToCharArray()
        |> Array.rev
        |> string
        |> ValueSome

let reversedString = strOpt |> ValueOption.bind reverse
```

這可讓 ValueOption 在實數值型別改善效能的案例中使用，就像選項一樣。

F # 4.5 的新功能

2020/11/2 • [Edit Online](#)

F # 4.5 加入了多項 F # 語言的增強功能。其中許多功能都已新增在一起，可讓您以 F # 撰寫有效率的程式碼，同時確保此程式碼是安全的。這麼做表示在使用這些結構時，將幾個概念新增至語言，以及大量編譯器分析。

開始使用

F # 4.5 適用於所有 .NET Core 發行版本和 Visual Studio 工具。[開始使用 F #](#)以深入瞭解。

Span 和類似 byref 的結構

[Span<T>](#).Net Core 中引進的類型可讓您以強型別方式表示記憶體中的緩衝區，而 f # 中現在允許使用 f # 4.5。下列範例會示範如何在 [Span<T>](#) 具有不同緩衝區標記法的上重複使用函數：

```
let safeSum (bytes: Span<byte>) =
    let mutable sum = 0
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes.[i]
    sum

// managed memory
let arrayMemory = Array.zeroCreate<byte>(100)
let arraySpan = new Span<byte>(arrayMemory)

safeSum(arraySpan) |> printfn "res = %d"

// native memory
let nativeMemory = Marshal.AllocHGlobal(100);
let nativeSpan = new Span<byte>(nativeMemory.ToPointer(), 100)

safeSum(nativeSpan) |> printfn "res = %d"
Marshal.FreeHGlobal(nativeMemory)

// stack memory
let mem = NativePtr.stackalloc<byte>(100)
let mem2 = mem |> NativePtr.toVoidPtr
let stackSpan = Span<byte>(mem2, 100)

safeSum(stackSpan) |> printfn "res = %d"
```

其中一個重要的層面是，Span 和其他 [byref 類似的結構](#) 都有非常嚴格的靜態分析，由編譯器執行，以限制其使用方式，因為您可能會發現非預期的情況。這是在 F # 4.5 中引進的效能、表達和安全性之間的基本取捨。

改頭換面 byref

在 F # 4.5 之前，F # 中的 [byref](#) 對許多應用程式而言是不安全的，而是無效的。Byref 的 Soundness 問題已在 F # 4.5 中解決，同時也套用了對 span 和類似 byref 的結構所做的相同靜態分析。

inref<不> 和 outref<不>

為了表示唯讀、僅限寫入和讀取/寫入受控指標的概念，F # 4.5 引進了 `inref<'T>`，`outref<'T>` 分別代表唯讀和僅限寫入指標的類型。每個都有不同的語義。例如，您無法寫入 `inref<'T>`：

```
let f (dt: inref<DateTime>) =  
    dt <- DateTime.Now // ERROR - cannot write to an inref!
```

根據預設，型別推斷會將 managed 指標推斷為與 `inref<'T>` F # 程式碼的不可變性質相同，除非已經將某些專案宣告為可變動。若要使其成為可寫入的專案，您必須在將 `mutable` 其位址傳遞給操作它的函式或成員之前，宣告類型。若要深入瞭解，請參閱[byref](#)。

ReadOnly 結構

從 F # 4.5 開始，您可以使用來標注結構， `IsReadOnlyAttribute` 如下所示：

```
[<IsReadOnly; Struct>]  
type S(count1: int, count2: int) =  
    member x.Count1 = count1  
    member x.Count2 = count2
```

這不允許您在結構中宣告可變動的成員，並且會發出中繼資料，讓 F # 和 c # 從元件取用時，將它視為唯讀。若要深入瞭解，請參閱[ReadOnly 結構](#)。

Void 指標

`voidptr` 類型會加入 F # 4.5，如下列函數所示：

- `NativePtr.ofVoidPtr` 將 void 指標轉換成原生 int 指標
- `NativePtr.toVoidPtr` 若要將原生 int 指標轉換成 void 指標

當與使用 void 指標的原生元件互通時，這會很有說明。

match! 關鍵字

關鍵字會在 `match!` 計算運算式內時增強模式比對：

```
// Code that returns an asynchronous option  
let checkBananaAsync (s: string) =  
    async {  
        if s = "banana" then  
            return Some s  
        else  
            return None  
    }  
  
// Now you can use 'match!'  
let funcWithString (s: string) =  
    async {  
        match! checkBananaAsync s with  
        | Some bananaString -> printfn "It's banana!"  
        | None -> printfn "%s" s  
    }
```

這可讓您縮短程式碼，這通常牽涉到混合選項(或其他類型)與計算運算式(例如 `async`)。若要深入瞭解，請參閱[match!](#)。

陣列、清單和順序運算式中的寬鬆 upcasting 需求

混合型別，其中一個可能繼承自陣列、清單和序列運算式內的另一個類型，傳統上會要求您使用或，將任何衍生的型別向上轉換成其父型別 `>` `upcast`。這現在是寬鬆的，如下所示：

```
let x0 : obj list = [ "a" ] // ok pre-F# 4.5
let x1 : obj list = [ "a"; "b" ] // ok pre-F# 4.5
let x2 : obj list = [ yield "a" :> obj ] // ok pre-F# 4.5

let x3 : obj list = [ yield "a" ] // Now ok for F# 4.5, and can replace x2
```

陣列和清單運算式的縮排放寬

在 F # 4.5 之前，當陣列和清單運算式當做引數傳遞給方法呼叫時，您需要過度將它縮排。已不再需要此動作：

```
module NoExcessiveIndenting =
    System.Console.WriteLine(format="{0}", arg = [|
        "hello"
    |])
    System.Console.WriteLine([|
        "hello"
    |])
```

F# 語言參考

2020/11/2 • [Edit Online](#)

本節是 F # 語言的參考，這是以 .NET 為目標的多重架構程式設計語言。F # 語言支援功能性、物件導向和命令式程式設計模型。

F # 核心程式庫 API 參考

[F # 核心程式庫 \(fsharp.core\) API 參考](#) 是所有 F # 核心程式庫命名空間、模組、類型和函式的參考。

F# 語彙基元

下表所顯示的參考文章會提供在 F # 中當做標記使用的關鍵字、符號和常值資料表。

TITLE	II
關鍵字參考	包含所有 F# 語言關鍵字相關資訊的連結。
符號和運算子參考	包含 F# 語言使用之符號和運算子的表格。
常值	描述 F# 常值的語法以及如何指定 F# 常值的類型資訊。

F# 語言概念

下表顯示描述語言概念的可用參考主題。

TITLE	II
函式	函式是所有程式設計語言的基礎程式執行單位。如同其他語言, F# 函式有名稱、可以有參數並且接受引數, 而且也有主體。F# 也支援函式程式設計建構, 例如將函式視為值、在運算式中使用不具名函式、組合函式以形成新函式、局部調用函式, 以及透過部分套用函式引數的隱含定義函式。
F# 類型	描述 F# 中所使用的類型以及 F# 類型的命名及描述方式。
類型推斷	描述 F # 編譯器如何推斷值、變數、參數和傳回值的類型。
自動一般化	描述 F# 的泛型建構。
繼承	描述繼承, 這是用來在物件導向程式設計中建立 "is-a" 關聯性 (或稱子類型) 的模型。
成員	描述 F# 物件類型的成員。
參數和引數	描述對定義參數以及將引數傳遞至函式、方法和屬性的語言支援, 其中包含如何以參考方式傳遞的詳細資訊。
運算子多載	描述如何在類別或記錄類型以及共用層多載算術運算子。

TITLE	說明
轉型和轉換	描述 F# 中對類型轉換的支援。
存取控制	描述 F# 的存取控制。存取控制表示宣告哪些用戶端能夠使用特定的程式元素，例如類型、方法、函式等等。
模式比對	描述模式，這是轉換輸入資料的規則，而且會在 F# 語言中使用。您可以比較資料與模式、將資料分解為構成部分，或從資料以各種方式解壓縮資訊。
現用模式	描述作用中的模式。作用中的模式可讓您定義可細分輸入資料的具名部分。您可以使用作用中的模式，以自訂方式分解每個部分的資料。
判斷提示	描述 <code>assert</code> 運算式，這個偵錯功能可用來測試運算式。當運算式在偵測模式中發生錯誤時，判斷提示會顯示系統錯誤對話方塊。
例外狀況處理	包含 F# 語言例外狀況處理支援的資訊。
attributes	描述可讓中繼資料套用至程式設計建構的屬性。
資源管理 : <code>use</code> 關鍵字	描述可控制資源初始設定及解除的關鍵字 <code>use</code> 和 <code>using</code> 。
namespaces	描述 F# 的命名空間支援。命名空間可讓您將名稱附加至程式項目群組，將程式碼依相關功能分類。
單元	描述模組。F# 模組是 F# 程式碼群組，例如 F# 程式中的值、類型和函式值。程式碼分組成不同模組有助於將相關程式碼整理到同一處，以及避免程式中發生名稱衝突。
匯入宣告 : <code>open</code> 關鍵字	描述 <code>open</code> 的運作方式。使用匯入宣告指定某個模組或命名空間後，就可以直接參考該模組或命名空間內的項目，而無須使用完整名稱。
簽章	描述簽章和簽章檔。簽章檔案包含一組 F# 程式項目的公開金鑰相關資訊，例如類型、命名空間和模組。它可用來指定這些程式項目的協助工具。
XML 文件	描述針對 XML 文件註解 (也稱為三斜線註解) 產生文件檔案的支援。您可以從 F# 中的程式碼批註產生檔，如同其他 .NET 語言一樣。
詳細語法	描述未啟用輕量型語法時的 F# 建構語法。詳細語法是透過程式碼頂端的 <code>#light "off"</code> 指示詞所表示。
純文字格式	瞭解如何在 F# 應用程式和腳本中使用 <code>sprintf</code> 和其他純文字格式。

F# 類型

下表顯示描述 F# 語言所支援類型的可用參考主題。

TITLE	說明
值	描述值，這是具有特定類型且不可變的數量；值可以是整數或浮點數、字元或文字、清單、序列、陣列、元組、差別聯集、記錄、類別類型或函式值。
基本類型	描述 F# 語言中使用的基本基本類型。它也會提供對應的 .NET 類型以及每個類型的最小值和最大值。
單位類型	描述 <code>unit</code> 類型，這個類型表示缺少特定值； <code>unit</code> 類型只有單一值，作為沒有或不需要其他值時的預留位置。
字串	描述 F# 中的字串。 <code>string</code> 類型以一連串的 Unicode 字元表示不可變文字。 <code>string</code> 是 <code>System.String</code> 在 .NET Framework 中的別名。
Tuple	描述元組，這是不具名但有序之值 (可能是不同的類型) 的群組。
F# 集合類型	F# 函式集合類型的概觀，包括陣列、清單、序列 (seq)、對應和集的類型。
清單	描述清單。F# 的清單是一連串有序且不可變的項目，而所有項目的類型都相同。
選項	描述選項類型。F# 中的選項於不一定有值時使用。選項有基礎類型，而且可能擁有該類型的值或沒有值。
序列	描述序列。序列是一連串的邏輯項目，而所有項目的類型都相同。只有在必要時才會計算個別順序元素，因此標記法可能會比常值元素計數還小。
陣列	描述陣列。陣列是一系列固定大小、以零起始、可變的連續資料項目，而所有項目的類型都相同。
記錄	描述記錄。記錄表示具名值的簡單彙總，並選擇性地搭配成員。
已區分的聯集	描述相異聯集，其支援可能是各種命名案例之一的值，每個都有可能不同的值和類型。
列舉	描述列舉，這是一組已定義之具名值的類型。列舉可用來取代常值，讓程式碼更容易閱讀及維護。
參考儲存格	描述參考儲存格，這是可讓您以參考語意建立可變變數的儲存位置。
類型縮寫	描述類型縮寫，這是類型的替代名稱。
類別	描述類別，這是表示可以有屬性、方法和事件之物件的類型。
結構	描述結構，這是一種比較精簡的物件類型，適用於資料量較少且行為簡單的類型，效率比類別更好。
介面	描述介面，介面可指定其他類別應提供實作的一組相關成員。

TITLE	🔖
抽象類別	描述抽象類別，這種類別不見得會實作出所有或部分成員，而是留待衍生類別各自提供實作。
類型擴充	描述類型擴充，可讓您將成員新增至先前定義的物件類型。
彈性類型	描述彈性類型。彈性類型注釋指出參數、變數或值的類型與指定的類型相容，而相容性是由類別或介面的物件導向階層中的位置所決定。
委派	描述將函式呼叫表示為物件的委派。
測量單位	描述測量單位。F# 中的浮點值可以有通常用來表示長度、容量、質量等的關聯測量單位。
型別提供者	描述型別提供者，並且提供逐步解說如何使用內建型別提供者來存取資料庫和 Web 服務的連結。

F# 運算式

下表列出描述 F# 運算式的主題。

TITLE	🔖
條件運算式 : <code>if...then...else</code>	描述 <code>if...then...else</code> 運算式，這個運算式會根據指定的布林運算式，執行不同的程式碼分支，也會運算出不同的值。
比對運算式	描述 <code>match</code> 運算式，此種運算式提供分支控制，可根據運算式與一組模式的比較結果，決定程式應沿著哪個分支繼續執行。
迴圈 : <code>for...to</code> 運算式	描述 <code>for...to</code> 運算式，這個運算式會重複執行某段迴圈，重複次數等於迴圈變數的範圍值。
迴圈 : <code>for...in</code> 運算式	描述 <code>for...in</code> 運算式，這個迴圈建構會使用可列舉集合 (例如範圍運算式、序列、清單、陣列或其他支援列舉的建構) 中符合模式的所有項目，重複執行一段程式碼。
迴圈 : <code>while...do</code> 運算式	描述 <code>while...do</code> 運算式，當指定的測試條件為 true 時，用來重複執行一次 (迴圈)。
物件運算式	描述物件運算式，這些運算式會根據現有的基底類型、一個介面或一組介面來建立動態建立、匿名物件類型的新執行個體。
延遲運算式	描述延遲運算式，這些運算式是不會立即評估的計算，而是在實際需要結果時進行評估。
計算運算式	描述 F# 中的計算運算式提供便利的語法，用於撰寫可以使用控制流程建構和繫結進行排序和合併的計算。它們可用來為 <i>monads</i> 提供一個方便的語法，這是一項功能性程式設計功能，可用來管理功能程式中的資料、控制和副作用。非同步工作流程是一種計算運算式，支援非同步和平行計算。如需詳細資訊，請參閱 非同步工作流程 。

TITLE	[[
非同步工作流程	描述非同步工作流程，這是一種語言功能，可讓您使用與原本撰寫同步程式碼極為相似的方式，來撰寫非同步程式碼。
程式碼引號	描述程式碼引號，此語言功能可讓您以程式設計方式產生及使用 F# 程式碼運算式。
查詢運算式	描述查詢運算式，這種語言功能可為 F# 實作 LINQ，並且可讓您針對資料來源或可列舉集合撰寫查詢。

編譯器支援的建構

下表列出描述編譯器支援之特殊建構的主題。

[[[[
編譯器選項	描述 F# 編譯器的命令列選項。
編譯器指示詞	描述處理器指示詞和編譯器指示詞。
原始碼程式行、檔案與路徑識別項	描述識別碼 <code>__LINE__</code> 、 <code>__SOURCE_DIRECTORY__</code> 和 <code>__SOURCE_FILE__</code> ，這是內建的值，可讓您存取程式碼中的原始程式列號、目錄和檔案名。

關鍵字參考

2020/11/2 • [Edit Online](#)

本主題包含所有 F # 語言關鍵字的相關資訊連結。

F # 關鍵字表

下表依字母順序顯示所有 F # 關鍵字，以及包含詳細資訊的簡短描述和相關主題的連結。

'''	''	''
<code>abstract</code>	成員 抽象類別	表示方法，該方法在宣告或為虛擬的類型中沒有任何實值，而且具有預設實值。
<code>and</code>	<code>let</code> 綁定 記錄 成員 約束	用於相互遞迴系結和記錄、屬性宣告，以及泛型參數的多個條件約束。
<code>as</code>	類別 模式比對	用來為目前的類別物件提供物件名稱。也可用來為模式比對中的整個模式提供名稱。
<code>assert</code>	判斷提示	用於在偵錯工具期間驗證程式代碼。
<code>base</code>	類別 繼承	用作基類物件的名稱。
<code>begin</code>	詳細語法	在詳細資訊語法中，表示程式碼區塊的開頭。
<code>class</code>	類別	在詳細資訊語法中，指出類別定義的開頭。
<code>default</code>	成員	表示抽象方法的執行。與抽象方法宣告一起使用，以建立虛擬方法。
<code>delegate</code>	委派	用來宣告委派。
<code>do</code>	do 繫結 迴圈 : <code>for...to</code> 運算式 迴圈 : <code>for...in</code> 運算式 迴圈 : <code>while...do</code> 運算式	用於迴圈結構或執行命令式程式碼。

'''	''	''
done	詳細語法	在詳細資訊語法中, 指出迴圈運算式中程式碼區塊的結尾。
downcast	轉型和轉換	用來轉換為繼承鏈中較低的類型。
downto	迴圈: for...to 運算式	在 for 運算式中, 用於反向計算時使用。
elif	條件運算式: if...then...else	用於條件式分支。的簡短形式 else if 。
else	條件運算式: if...then...else	用於條件式分支。
end	結構 已區分的聯集 記錄 類型擴充 詳細語法	在 [型別定義和類型延伸] 中, 指出成員定義區段的結尾。 在詳細資訊語法中, 用來指定開頭為關鍵字的程式碼區塊結尾 begin 。
exception	例外狀況處理 例外狀況類型	用來宣告例外狀況類型。
extern	外部函式	指出宣告的程式元素是在另一個二進位或元件中定義。
false	基本類型	當做布林常值使用。
finally	例外狀況: try...finally 運算式	與一起使用 try , 以引入無論是否發生例外狀況都會執行的程式碼區塊。
fixed	固定	用來「釘選」堆疊上的指標, 以防止它被垃圾收集。
for	迴圈: for...to 運算式 迴圈: for..in 運算式	用於迴圈結構。
fun	Lambda 運算式: fun 關鍵字	用於 lambda 運算式, 也稱為匿名函數。
function	比對運算式 Lambda 運算式: 有趣的關鍵字	用來作為關鍵字的較短替代, fun 以及 match 在單一引數上具有模式比對之 lambda 運算式中的運算式。
global	命名空間	用來參考最上層的 .NET 命名空間。
if	條件運算式: if...then...else	在條件式分支結構中使用。

关键字	关键字	关键字
<code>in</code>	迴圈:for...in 運算式 詳細語法	用於順序運算式和(在詳細語法中), 以分隔運算式與系結。
<code>inherit</code>	繼承	用來指定基類或基底介面。
<code>inline</code>	函式 內嵌函式	用來表示應該直接整合至呼叫端程式碼的函式。
<code>interface</code>	介面	用來宣告和執行介面。
<code>internal</code>	存取控制	用來指定成員可顯示在元件內, 但不在其外部。
<code>lazy</code>	延遲運算式	用來指定只有在需要結果時才會執行的運算式。
<code>let</code>	<code>let</code> 綁定	用來將名稱與值或函數系結或系結。
<code>let!</code>	非同步工作流程 計算運算式	用於非同步工作流程中, 以將名稱系結至非同步計算的結果, 或在其他計算運算式中用來將名稱系結至計算類型的結果。
<code>match</code>	比對運算式	用來將值與模式比較, 以進行分支。
<code>match!</code>	計算運算式	用來內嵌對計算運算式的呼叫, 以及其他結果的模式比對。
<code>member</code>	成員	用來宣告物件型別中的屬性或方法。
<code>module</code>	單元	用來將名稱與相關類型、值和函數的群組建立關聯, 以邏輯方式將它與其他程式碼分開。
<code>mutable</code>	<code>let</code> 繫結	用來宣告變數, 也就是可以變更的值。
<code>namespace</code>	命名空間	用來將名稱與一組相關類型和模組產生關聯, 以邏輯方式將它與其他程式碼分開。
<code>new</code>	建構函式 約束	用來宣告、定義或叫用建立或可建立物件的函式。 也在泛型參數條件約束中用來指出型別必須有特定的函式。
<code>not</code>	符號和運算子參考 約束	其實不是關鍵字。不過, <code>not struct</code> 組合會用來做為泛型參數條件約束。

关键字	关键字	关键字
<code>null</code>	Null 值 約束	<p>指出物件是否存在。</p> <p>也在泛型參數條件約束中使用。</p>
<code>of</code>	已區分的聯集 委派 例外狀況類型	<p>用在區分等位中, 表示值的分類類型, 以及委派和例外狀況宣告中的型別。</p>
<code>open</code>	匯入宣告: <code>open</code> 關鍵字	<p>用來使命名空間或模組的內容可供使用, 而不需限定。</p>
<code>or</code>	符號和運算子參考 約束	<p>使用布林值條件做為布林值 <code>or</code> 運算子。相當於 <code> </code>。</p> <p>也用於成員條件約束中。</p>
<code>override</code>	成員	<p>用來執行與基底版本不同之抽象或虛擬方法的版本。</p>
<code>private</code>	存取控制	<p>將成員的存取限制為相同類型或模組中的程式碼。</p>
<code>public</code>	存取控制	<p>允許從類型外部存取成員。</p>
<code>rec</code>	函式	<p>用來表示函式是遞迴的。</p>
<code>return</code>	非同步工作流程 計算運算式	<p>用來指出要提供作為計算運算式結果的值。</p>
<code>return!</code>	計算運算式 非同步工作流程	<p>用來指出計算運算式, 在評估時, 會提供包含計算運算式的結果。</p>
<code>select</code>	查詢運算式	<p>用於查詢運算式中, 以指定要解壓縮的欄位或資料行。請注意, 這是內容關鍵字, 這表示它實際上不是保留字, 而且只適用於適當內容中的關鍵字。</p>
<code>static</code>	成員	<p>用來表示方法或屬性, 該方法或屬性可在沒有類型實例的情況下呼叫, 或在類型的所有實例之間共用的值成員。</p>
<code>struct</code>	結構 Tuple 約束	<p>用來宣告結構類型。</p> <p>用來指定結構元組。</p> <p>也在泛型參數條件約束中使用。</p> <p>用於模組定義中的 OCaml 相容性。</p>

'''	''	'
then	條件運算式: <code>if...then...else</code> 建構函式	在條件運算式中使用。 也用來在物件結構之後執行副作用。
to	迴圈: <code>for...to</code> 運算式	在迴圈中用 <code>for</code> 來指出範圍。
true	基本類型	當做布林常值使用。
try	例外狀況: <code>try...with</code> 運算式 例外狀況: <code>try...finally</code> 運算式	用來引進可能產生例外狀況的程式碼區塊。與或一起 <code>with</code> 使用 <code>finally</code> 。
type	F# 類型 類別 記錄 結構 列舉 已區分的聯集 類型縮寫 測量單位	用來宣告類別、記錄、結構、區分聯集、列舉類型、測量單位或類型縮寫。
upcast	轉型和轉換	用來轉換為繼承鏈中較高的類型。
use	資源管理: <code>use</code> 關鍵字	用來取代 <code>let</code> 需要 <code>Dispose</code> 呼叫以釋放資源的值。
use!	計算運算式 非同步工作流程	<code>let!</code> 針對需要呼叫以釋出資源的值使用, 而不是在非同步工作流程和其他計算運算式中使用 <code>Dispose</code> 。
val	明確欄位: <code>val</code> 關鍵字 簽章 成員	在有限的情況下, 在簽章中用來表示值, 或用於宣告成員的類型。
void	基本類型	指出 .NET <code>void</code> 型別。與其他 .NET 語言交互操作時使用。
when	約束	用於布林值條件 (當符合模式的), 以及引入泛型型別參數的條件約束子句時。
while	迴圈: <code>while...do</code> 運算式	引進迴圈結構。

'''	''	'
with	<p>比對運算式</p> <p>物件運算式</p> <p>複製和更新記錄運算式</p> <p>類型擴充</p> <p>例外狀況: try...with 運算式</p>	與模式比對 match 運算式中的關鍵字一起使用。也可用於物件運算式、記錄複製運算式和類型延伸模組以引入成員定義, 以及引入例外狀況處理常式。
yield	清單、陣列、序列	用於清單、陣列或序列運算式中, 以產生序列的值。通常可以省略, 因為在大部分情況下都是隱含的。
yield!	<p>計算運算式</p> <p>非同步工作流程</p>	用於計算運算式中, 以將給定計算運算式的結果附加至包含計算運算式的結果集合。
const	型別提供者	型別提供者可以使用 const 做為關鍵字, 將常數常值指定為型別參數引數。

下列權杖會保留在 F # 中, 因為它們是 OCaml 語言中的關鍵字:

- asr
- land
- lor
- lsl
- lsr
- lxor
- mod
- sig

如果您使用 --mlcompatibility 編譯器選項, 則可以使用上述關鍵字做為識別碼。

下列權杖會保留為關鍵字, 以供未來 F # 語言的擴充之用:

- atomic
- break
- checked
- component
- const
- constraint
- constructor
- continue
- eager
- event
- external
- functor
- include
- method

- `mixin`
- `object`
- `parallel`
- `process`
- `protected`
- `pure`
- `sealed`
- `tailcall`
- `trait`
- `virtual`
- `volatile`

另請參閱

- [F # 語言參考](#)
- [符號和運算子參考](#)
- [編譯器選項](#)

符號和運算子參考

2020/11/2 • [Edit Online](#)

本文包含 F # 語言中使用的符號和運算子的表格。

符號和運算子的資料表

下表描述 F # 語言中使用的符號，並提供符號和連結的一些用法的簡短描述，以取得詳細資訊。符號會依據 ASCII 字元集的順序加以排序。

符號	說明	用法
!	參考儲存格 計算運算式	<ul style="list-style-type: none">取值參考儲存格。在關鍵字後面，代表由工作流程所控制的關鍵字行為之修改版本。
!=		<ul style="list-style-type: none">F# 中不使用。使用 <> 進行不等比較運算。
"	常值 字串	<ul style="list-style-type: none">分隔文字字串。
"""	字串	分隔逐字文字字串。有別於 @"...", 因為您可以在字串中使用單引號來表示引號字元。
#	編譯器指示詞 彈性類型	<ul style="list-style-type: none">開頭前置處理器或編譯器指示詞，例如，#light。與類型一起使用時，表示「彈性類型」**，這指的是類型或任何一種其衍生的類型。
\$		<ul style="list-style-type: none">使用於內部，用於特定編譯器產生的變數與函式名稱。
%	算術運算子 程式碼引號	<ul style="list-style-type: none">計算整數餘數。用於將運算式接合成具類型的程式碼引號。
%%	程式碼引號	<ul style="list-style-type: none">用於將運算式接合成不具類型的程式碼引號。
%?	可為 Null 的運算子	<ul style="list-style-type: none">當右側是可為 null 的型別時，計算整數餘數。

語法符號	說明	用途
<code>&</code>	比對運算式	<ul style="list-style-type: none"> 計算可變動值的位址，以在與其他語言相互操作時使用。 用於 AND 模式中。
<code>&&</code>	布林運算子	<ul style="list-style-type: none"> 計算布林值 AND 運算。
<code>&&&</code>	位元運算子	<ul style="list-style-type: none"> 計算位元 AND 運算。
<code>'</code>	常值 自動一般化	<ul style="list-style-type: none"> 分隔單一字元常值。 表示泛型類型參數。
<code>``...``</code>		<ul style="list-style-type: none"> 分隔若在其他狀況下不會是合法識別項的識別項，例如語言關鍵字。
<code>()</code>	單位類型	<ul style="list-style-type: none"> 代表單位類型的單一值。
<code>(...)</code>	元組 運算子多載	<ul style="list-style-type: none"> 表示運算式的評估順序。 分隔 Tuple。 用於運算子定義中。
<code>(*...*)</code>		<ul style="list-style-type: none"> 分隔無法跨越多行的註解。
<code>(...)</code>	現用模式	<ul style="list-style-type: none"> 分隔使用中的模式。也稱為「香蕉夾」<code>**</code>。
<code>*</code>	算術運算子 元組 測量單位	<ul style="list-style-type: none"> 當做二元運算子使用時，將左側與右側相乘。 在類型中，表示在 Tuple 中配對。 用於測量單位類型中。
<code>*?</code>	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 類型時，將左側與右側相乘。
<code>**</code>	算術運算子	<ul style="list-style-type: none"> 計算乘幂運算 (<code>x ** y</code> 表示 <code>x</code> 的 <code>y</code> 次方)。
<code>+</code>	算術運算子	<ul style="list-style-type: none"> 當做二元運算子使用時，將左側與右側相加。 當做一元運算子使用時，表示正數量。(正式地說，它會產生正負號維持不變的相同值。)

語法	說明	備註
<code>+?</code>	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時, 將左側與右側相加。
<code>,</code>	元組	<ul style="list-style-type: none"> 分隔 Tuple 的項目或型別參數。
<code>-</code>	算術運算子	<ul style="list-style-type: none"> 當做二元運算子使用時, 從左側減去右側。 當做一元運算子使用時, 執行負運算。
<code>-?</code>	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時, 從左側減去右側。
<code>-></code>	函式 比對運算式	<ul style="list-style-type: none"> 在函式類型中, 分隔引數並傳回值。 產生運算式 (在循序項運算式中); 相當於 <code>yield</code> 關鍵字。 用於比對運算式中
<code>.</code>	成員 基本類型	<ul style="list-style-type: none"> 存取成員, 並分隔完整名稱中的個別名稱。 以浮點數指定小數點。
<code>..</code>	迴圈: <code>for...in</code> 運算式	<ul style="list-style-type: none"> 指定範圍。
<code>.. ..</code>	迴圈: <code>for...in</code> 運算式	<ul style="list-style-type: none"> 指定範圍與遞增值。
<code>.[...]</code>	陣列	<ul style="list-style-type: none"> 存取陣列元素。
<code>/</code>	算術運算子 測量單位	<ul style="list-style-type: none"> 左側 (分子) 除以右側 (分母)。 用於測量單位類型中。
<code>/?</code>	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時, 左側除以右側。
<code>//</code>		<ul style="list-style-type: none"> 表示單行註解的開頭。
<code>///</code>	XML 文件	<ul style="list-style-type: none"> 表示 XML 註解。
<code>:</code>	函式	<ul style="list-style-type: none"> 在類型註釋中, 將參數或成員名稱與其類型分隔。

=====	==	==
::	清單 比對運算式	<ul style="list-style-type: none"> 建立清單。左側的項目會附加到右側清單的前面。 用於模式比對中以分隔清單的組件。
:=	參考儲存格	<ul style="list-style-type: none"> 將值指派給參考儲存格。
:	轉型和轉換	<ul style="list-style-type: none"> 將類型轉換成階層中較高階的類型。
:?	比對運算式	<ul style="list-style-type: none"> <code>true</code> 如果值符合指定的型別 ((如果它是) 的子類型, 則傳回, 否則傳回 <code>false</code> (型別測試運算子)。
:?>	轉型和轉換	<ul style="list-style-type: none"> 將類型轉換成階層中較低階的類型。
;	詳細語法 清單 記錄	<ul style="list-style-type: none"> 分隔運算式 (大部分用於詳細語法中)。 分隔清單的元素。 分隔記錄的欄位。
<	算術運算子	<ul style="list-style-type: none"> 計算小於運算。
<?	可為 Null 的運算子	當右側是可為 null 的類型時, 計算小於運算。
<<	函式	<ul style="list-style-type: none"> 以相反順序撰寫兩個函式; 第二個函式會先執行 (反向撰寫運算子)。
<<<	位元運算子	<ul style="list-style-type: none"> 將左側中數量中的位元, 向左移位右側所指定的位元數。
<-	值	<ul style="list-style-type: none"> 將值指派給變數。
<...>	自動一般化	<ul style="list-style-type: none"> 分隔型別參數。
<>	算術運算子	<ul style="list-style-type: none"> 如果左側不等於右側, 即傳回 <code>true</code>; 否則, 傳回 <code>false</code>。
<>?	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時, 計算「不等於」運算。

符號	說明	說明
<code><=</code>	算術運算子	<ul style="list-style-type: none"> 如果左側小於或等於右側，即傳回 <code>true</code>；否則傳回 <code>false</code>。
<code><=?</code>	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時，計算「小於或等於」運算。
<code> ></code>	函式	<ul style="list-style-type: none"> 將左側的結果傳遞至右側的函式 (正向管道運算子)。
<code> ></code>	<code>(>)< t 1, t 2, 'U></code> 函數	<ul style="list-style-type: none"> 將左側兩個引數的 Tuple 傳遞至右側的函式。
<code> ></code>	<code>(>)< t 1, t 2, t 3, 'U></code> 函數	<ul style="list-style-type: none"> 將左側三個引數的 Tuple 傳遞至右側的函式。
<code>< </code>	函式	<ul style="list-style-type: none"> 將右側的運算結果傳遞至左側的函式 (反向管道運算子)。
<code>< </code>	<code>(<)< t 1, t 2, 'U></code> 函數	<ul style="list-style-type: none"> 將右側兩個引數的 Tuple 傳遞至左側的函式。
<code>< </code>	<code>(<)< t 1, t 2, t 3, 'U></code> 函數	<ul style="list-style-type: none"> 將右側三個引數的 Tuple 傳遞至左側的函式。
<code><@...@></code>	程式碼引號	<ul style="list-style-type: none"> 分隔具類型的程式碼引號。
<code><@@...@@></code>	程式碼引號	<ul style="list-style-type: none"> 分隔不具類型的程式碼引號。
<code>=</code>	算術運算子	<ul style="list-style-type: none"> 如果左側等於右側，即傳回 <code>true</code>；否則傳回 <code>false</code>。
<code>=?</code>	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時，計算「等於」運算。
<code>==</code>		<ul style="list-style-type: none"> F# 中不使用。使用 <code>=</code> 進行等號比較運算。
<code>></code>	算術運算子	<ul style="list-style-type: none"> 如果左側大於右側，即傳回 <code>true</code>；否則傳回 <code>false</code>。
<code>>?</code>	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時，計算「大於」運算。

#####	[[[[
>>	函式	<ul style="list-style-type: none"> 撰寫兩個函式 (正向撰寫運算子)。
>>>	位元運算子	<ul style="list-style-type: none"> 將左側數量中的位元, 向右移位右側指定的位數。
>=	算術運算子	<ul style="list-style-type: none"> <code>true</code> 如果左邊大於或等於右邊, 則傳回, 否則傳回 <code>false</code>。
>=?	可為 Null 的運算子	<ul style="list-style-type: none"> 當右側是可為 null 的類型時, 計算「大於或等於」運算。
?	參數和引數	<ul style="list-style-type: none"> 指定選擇性引數。 用做為動態方法及屬性呼叫的運算子。您必須提供自己的實作。
? ... <- ...		<ul style="list-style-type: none"> 用來做為設定動態屬性的運算子。您必須提供自己的實作。
?>=, ?>, ?<=, ?<, ?=, ?<>, ?+, ?-, ?*, ?/	可為 Null 的運算子	<ul style="list-style-type: none"> 相當於沒有 ? 前置詞的對應運算子, 其中可為 null 的類型在左側。
>=? , >? , <=? , <? , =? , <>? , +? , -? , *? , /?	可為 Null 的運算子	<ul style="list-style-type: none"> 相當於沒有 ? 前置詞的對應運算子, 其中可為 null 的類型在右側。
?>=? , ?>? , ?<=? , ?<? , ?=? , ?<>? , ?+? , ?-? , ?*? , ?/?	可為 Null 的運算子	<ul style="list-style-type: none"> 相當於沒有問號括住的對應運算子, 其兩側都是可為 null 的類型。
@	清單 字串	<ul style="list-style-type: none"> 串連兩個清單。 置於字串常值前面時, 表示要逐字解譯字串, 但是不解譯逸出字元。
[...]	清單	<ul style="list-style-type: none"> 分隔清單的元素。
[...]	陣列	<ul style="list-style-type: none"> 分隔陣列的元素。
[<...>]	屬性	<ul style="list-style-type: none"> 分隔屬性。

符號	說明	用法
<code>\</code>	字串	<ul style="list-style-type: none">逸出下一個字元;用於字元和字串常值中。
<code>^</code>	以統計方式解析的型別參數 字串	<ul style="list-style-type: none">指定必須在編譯階段 (而非執行階段) 解析的型別參數。串連字串。
<code>^^^</code>	位元運算子	<ul style="list-style-type: none">計算位元互斥 OR 運算。
<code>_</code>	比對運算式 泛型	<ul style="list-style-type: none">表示萬用字元模式。指定匿名泛型參數。
<code>`</code>	自動一般化	<ul style="list-style-type: none">使用於內部, 用於表示泛型型別參數。
<code>{...}</code>	序列 記錄	<ul style="list-style-type: none">分隔循序項運算式與計算運算式。用於記錄定義中。
<code> </code>	比對運算式	<ul style="list-style-type: none">分隔各個相符的情況、個別差別聯集宣告和列舉值。
<code> </code>	布林運算子	<ul style="list-style-type: none">計算布林值 OR 運算。
<code> </code>	位元運算子	<ul style="list-style-type: none">計算位元 OR 運算。
<code>~~</code>	運算子多載	<ul style="list-style-type: none">用以宣告一元負運算子的多載。
<code>~~~</code>	位元運算子	<ul style="list-style-type: none">計算位元 NOT 運算。
<code>~-</code>	運算子多載	<ul style="list-style-type: none">用以宣告一元相減運算子的多載。
<code>~+</code>	運算子多載	<ul style="list-style-type: none">用以宣告一元相加運算子的多載。

運算子優先順序

下表顯示 F# 語言中運算子和其他運算式關鍵字之優先順序，順序從最低的優先順序到最高的優先順序。同時列出關聯性 (如果適用的話)。

'''	'''
as	Right
when	Right
(管道)	Left
;	Right
let	Nonassociative
function, fun, match, try	Nonassociative
if	Nonassociative
not	Right
->	Right
:=	Right
,	Nonassociative
or,	Left
&, &&	Left
:>, :?>	Right
< op、 > op、 = 、 op、 & op、 & (包括 <<<、>>>、 、&&&)	Left
^ 主管 (包括 ^^^)	Right
::	Right
:?	未關聯
- op、 + op	適用於這些符號的中置用法
* op、 / op、 % op	Left
** 主管	Right
f x (函式應用程式)	Left
(模式比對)	Right

'''	'''
前置運算子 (<code>+</code> <i>op</i> 、 <code>-</code> <i>op</i> 、 <code>%</code> 、 <code>...</code> 、 <code>%%</code> 、 <code>&</code> 、 <code>&&</code> 、 <code>!</code> <i>op</i> 、 <code>~</code> <i>op</i>)	Left
<code>.</code>	Left
<code>f(x)</code>	Left
<code>f< 類型 ></code>	Left

F# 支援自訂運算子多載。這表示您可以定義自己的運算子。在上表中，*op* 可以是任何有效的 (可能是空的) 運算子字元序列，即內建或使用者定義的序列。因此，您可以使用此表格來判斷要對自訂運算子使用什麼字元序列，以達到您想要的優先順序等級。前置 `.` 字元在編譯器判斷優先順序時，會予以忽略。

另請參閱

- [F # 語言參考](#)
- [運算子多載](#)

算術運算子

2019/10/23 • [Edit Online](#)

本主題描述F#語言中可用的算術運算子。

二進位算術運算子的摘要

下表摘要說明可用於取消裝箱整數和浮點類型的二元算術運算子。

符號	說明
<code>+</code> (加法, plus)	選定. 當數位同時加上, 且總和超出類型所支援的最大絕對值時, 可能發生溢位條件。
<code>-</code> (減法, 減號)	選定. 當減去不帶正負號的類型時, 或是浮點值太小而無法以類型表示時, 可能出現的下溢條件。
<code>*</code> (乘法, times)	選定. 當數位相乘且產品超出類型所支援的最大絕對值時, 可能發生溢位條件。
<code>/</code> (相除, 除以)	零除會導致 <code>DivideByZeroException</code> 整數類資料類型。對於浮點類型, 零除會提供您特殊的浮點值 <code>+Infinity</code> 或 <code>-Infinity</code> 。當浮點數太小而無法以類型表示時, 還有可能出現的下溢條件。
<code>%</code> (餘數, rem)	傳回除法運算的餘數。結果的正負號與第一個運算元的正負號相同。
<code>**</code> (乘幂, 表示的乘幂)	當結果超出類型的最大絕對值時, 可能發生溢位條件。 乘幂運算子僅適用於浮點類型。

一元算術運算子的摘要

下表摘要說明可用於整數和浮點類型的一元算術運算子。

符號	說明
<code>+</code> 3.402823e38	可以套用至任何算術運算式。不會變更值的正負號。
<code>-</code> (負, 負值)	可以套用至任何算術運算式。變更值的正負號。

整數類資料類型溢位或下溢的行為是要換行。這會導致不正確的結果。整數溢位是可能嚴重的問題, 可能會導致未撰寫軟體的安全性問題。如果這是您應用程式的顧慮, 請考慮在中 `Microsoft.FSharp.Core.Operators.Checked` 使用檢查的運算子。

二進位比較運算子的摘要

下表顯示整數和浮點類型可用的二進位比較運算子。這些運算子會傳回類型 `bool` 的值。

因為 IEEE 浮點標記法不支援完全相等的作業, 所以不應直接將浮點數與相等進行比較。藉由檢查程式碼, 您可以

輕鬆驗證的兩個數字, 實際上可能會有不同的位標記法。

'''	''
<code>=</code> (相等、等於)	這不是指派運算子。僅用於比較。這是泛型運算子。
<code>></code> (大於)	這是泛型運算子。
<code><</code> (小於)	這是泛型運算子。
<code>>=</code> (大於或等於)	這是泛型運算子。
<code><=</code> (小於或等於)	這是泛型運算子。
<code><></code> (不等於)	這是泛型運算子。

多載和泛型運算子

本主題中討論的所有運算子都定義于 `fsharp.core` 命名空間中。某些運算子是使用靜態解析的類型參數來定義。這表示與該運算子搭配使用的每個特定型別都有個別的定義。所有的一元和二元算術和位運算子都是在此類別中。比較運算子是泛型的, 因此可與任何類型搭配使用, 而不只是基本的算術類型。區分聯集和記錄類型有自己的自訂實作為 F# 編譯器所產生。類別類型使用方法 [Equals](#)。

泛型運算子是可自訂的。若要自訂比較函數, [Equals](#) 請覆寫以提供您自己的自訂相等 [IComparable](#) 比較, 然後執行。介面具有單一方法, 也就是 [CompareTo](#) 方法。 [System.IComparable](#)

運算子和型別推斷

在運算式中使用運算子, 會限制該運算子上的型別推斷。此外, 使用運算子會防止自動一般化, 因為使用運算子意指算術類型。如果沒有任何其他資訊, F# 編譯器會推斷 `int` 為算術運算式的類型。您可以藉由指定其他類型來覆寫此行為。因此 `function1 int`, 會將 `function2` 下列 `float` 程式碼中的引數類型和傳回類型推斷為, 但是的類型會推斷為。

```
// x, y and return value inferred to be int
// function1: int -> int -> int
let function1 x y = x + y

// x, y and return value inferred to be float
// function2: float -> float -> float
let function2 (x: float) y = x + y
```

另請參閱

- [符號和運算子參考](#)
- [運算子多載](#)
- [位元運算子](#)
- [布林運算子](#)

Boolean 運算子

2019/10/23 • [Edit Online](#)

本主題說明中的布林運算子的支援F#語言。

布林運算子的摘要

下表摘要說明可用於布林運算子F#語言。唯一支援這些運算子的類型是 `bool` 型別。

<code>!!!</code>	<code>!!</code>
<code>not</code>	布林值的否定運算
<code> </code>	布林值 OR
<code>&&</code>	布林值 AND

布林值 AND 和 OR 運算子執行 *最少運算評估*，也就是它們評估運算式運算子的右側時才需要判斷整體運算式的結果。第二個運算式 `&&` 運算子會評估第一個運算式評估為時，才 `true`；的第二個運算式 `||` 運算子會評估，只有第一個運算式評估為 `false`。

另請參閱

- [位元運算子](#)
- [算術運算子](#)
- [符號和運算子參考](#)

位元運算子

2019/10/23 • [Edit Online](#)

本主題說明中所提供的位元運算子F#語言。

位元運算子的摘要

下表描述中的 `unboxed` 整數類資料類型都支援的位元運算子F#語言。

'''	''
<code>&&&</code>	位元 AND 運算子。如果且只有在兩個來源運算元的對應位元為 1, 結果中會有值為 1。
<code> </code>	位元的 OR 運算子。在結果中的位元具有值 1, 如果有任一個來源中的對應位元的運算元都是 1。
<code>^^^</code>	位元互斥 OR 運算子。在結果中的位元具有值 1, 如果且只有在來源運算元的位元具有不相等的值。
<code>~~~</code>	位元否定運算子。這是一元運算子, 而且產生的結果, 在其中所有的 0 位元, 在來源運算元會轉換成 1 的位元和所有的 1 位元轉換成 0 位元。
<code><<<</code>	位元左移運算子。結果會是第一個運算元與位元左移第二個運算元的位元數目。移出的最大顯著性的位置的位元不會旋轉的最小顯著性的位置。最小顯著性的位元是以零填補。第二個引數的型別是 <code>int32</code> 。
<code>>>></code>	位元右移位運算子。結果會是第一個運算元與位元向右移位由第二個運算元的位元數目。移出的最小顯著性的位置的位元不會旋轉的最大顯著性的位置。不帶正負號的類型, 最小顯著性的位元是以零填補。帶正負號的型別具有負數值, 最小顯著性的位元會填補的。第二個引數的型別是 <code>int32</code> 。

與位元運算子, 可以使用下列類型: `byte`, `sbyte`, `int16`, `uint16`, `int32 (int)`, `uint32`, `int64`, `uint64`, `nativeint`, 以及 `unativeint`。

另請參閱

- [符號和運算子參考](#)
- [算術運算子](#)
- [布林運算子](#)

可為 Null 的運算子

2021/3/5 • [Edit Online](#)

可為 null 的運算子是二元算術或比較運算子，可在一或兩個邊使用可為 null 的算術類型。當您使用來自來源的資料(例如允許 null 來取代實際值的資料庫)時，就會經常發生可為 null 的類型。可為 null 的運算子經常用於查詢運算式中。除了算術和比較可為 null 的運算子之外，轉換運算子也可以用來在可為 null 的類型之間轉換。某些查詢運算子也有可為 null 的版本。

可為 Null 運算子的資料表

下表列出 F# 語言支援的可為 null 運算子。

可為 NULL	可為 NULL	可為 NULL
? >=	>= ?	? >= ?
? >	> ?	? > 嗎?
? <=	<= ?	? <= ?
? <	< ?	? < 嗎?
?=	=?	?=?
? <>	<> ?	? <> 嗎?
?+	+?	?+?
?-	-?	?-?
?*	*?	?*?
?/	/?	?/?
?%	%?	?%?

備註

可為 null 的運算子會包含在命名空間 `fsharp.core` 的 `NullableOperators` 模組中。可為 null 的資料類型為

`System.Nullable<'T>`。

在查詢運算式中，從允許 null 而非值的資料來源中選取資料時，就會產生可為 null 的類型。在 SQL Server 資料庫中，資料表中的每個資料行都有一個屬性，指出是否允許 null。如果允許 null，則從資料庫傳回的資料可以包含無法以基本資料類型(例如、等)表示的 null `int` `float`。因此，資料會以 `System.Nullable<int>` 取代，而不是傳回 `int` `System.Nullable<float>` `float`。您可以使用屬性從物件取得實際值 `System.Nullable<'T>.Value`，而且您可以藉 `System.Nullable<'T>.HasValue` 由呼叫方法來判斷物件是否具有值 `HasValue`。另一個實用的方法是 `System.Nullable<'T>.GetValueOrDefault` 方法，可讓您取得適當類型的值或預設值。預設值是某種形式的「零」值，例如 0、0.0 或 `false`。

可為 null 的型別可以使用一般的轉換運算子(例如或)轉換成不可為 null 的基本類型 `int` `float`。您也可以使用可為 null 型別的轉換運算子，從一個可為 null 的型別轉換為另一個可為 null 的型別。適當的轉換運算子具有與標準名稱相同的名稱，但它們位於不同的模組中，也就是 `fsharp.core` 命名空間中的可為 null 模組。一般而言，您會在使用查詢運算式時開啟此命名空間。在這種情況下，您可以將前置詞加入至適當的轉換運算子，以使用可為 null 的轉換運算子 `Nullable.`，如下列程式碼所示。

```
open Microsoft.FSharp.Linq

let nullableInt = new System.Nullable<int>(10)

// Use the Nullable.float conversion operator to convert from one nullable type to another nullable type.
let nullableFloat = Nullable.float nullableInt

// Use the regular non-nullable float operator to convert to a non-nullable float.
printfn $"%f{float nullableFloat}"
```

輸出為 `10.000000`。

可為 null 的資料欄位(例如)的查詢運算子 `sumByNullable` 也存在於查詢運算式中。非 nullable 型別的查詢運算子與可為 null 的型別不相容，因此當您使用可為 null 的資料值時，您必須使用適當查詢運算子的可為 null 版本。如需詳細資訊，請參閱 [查詢運算式](#)。

下列範例示範如何在 F# 查詢運算式中使用可為 null 的運算子。第一個查詢顯示如何撰寫沒有可為 null 運算子的查詢;第二個查詢顯示使用可為 null 運算子的對等查詢。如需完整的內容，包括如何設定資料庫以使用此範例程式碼，請參閱 [逐步解說:使用類型提供者存取 SQL Database](#)。

```
open System
open System.Data
open System.Data.Linq
open Microsoft.FSharp.Data.TypeProviders
open Microsoft.FSharp.Linq

[<Generate>]
type dbSchema = SqlConnection<"Data Source=MYSERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated Security=SSPI;">

let db = dbSchema.GetDataContext()

query {
    for row in db.Table2 do
        where (row.TestData1.HasValue && row.TestData1.Value > 2)
        select row
} |> Seq.iter (fun row -> printfn "%d{row.TestData1.Value} %s{row.Name}")

query {
    for row in db.Table2 do
        // Use a nullable operator ?>
        where (row.TestData1 ?> 2)
        select row
} |> Seq.iter (fun row -> printfn "%d{row.TestData1.GetValueOrDefault()} %s{row.Name}")
```

另請參閱

- [型別提供者](#)
- [查詢運算式](#)

函式

2020/11/2 • [Edit Online](#)

函式是所有程式設計語言的基礎程式執行單位。如同其他語言，F# 函式有名稱、可以有參數並且接受引數，而且也有主體。F# 也支援函式程式設計建構，例如將函式視為值、在運算式中使用不具名函式、組合函式以形成新函式、局部調用函式，以及透過部分套用函式引數的隱含定義函式。

您可以使用 `let` 關鍵字定義函式，若是遞迴函式，則可以使用 `let rec` 關鍵字組合來定義函式。

語法

```
// Non-recursive function definition.  
let [inline] function-name parameter-list [ : return-type ] = function-body  
// Recursive function definition.  
let rec function-name parameter-list = recursive-function-body
```

備註

function-name 表示函式的識別項。*parameter-list* 由以空格分隔的連續參數所組成。您可以指定每個參數的明確類型，如〈參數〉一節中所述。若未指定特定的引數類型，編譯器會嘗試從函式主體推斷類型。*function-body* 由運算式組成。函式主體通常是由組合運算式組成，其中包含數個會產生最終運算式作為傳回值的運算式。*return-type* 是選擇性項目，包含後面接著類型的冒號。若您沒有明確指定傳回值的類型，則編譯器會從最終運算式判斷傳回型別。

簡單的函式定義如下所示：

```
let f x = x + 1
```

在上述範例中，函式名稱為 `f`，引數是類型為 `int` 的 `x`，函式主體為 `x + 1`，而傳回值的類型為 `int`。

函式可以標記為 `inline`。如需 `inline` 的資訊，請參閱[內嵌函式](#)。

影響範圍

在模組範圍以外的任何範圍層級，重複使用值或函式名稱並非錯誤。若重複使用名稱，後面宣告的名稱會遮蔽前面宣告的名稱。不過，在模組中的最上層範圍，名稱必須是唯一名稱。例如，下列程式碼出現在模組範圍時會產生錯誤，但在函式內時則不會：

```
let list1 = [ 1; 2; 3]  
// Error: duplicate definition.  
let list1 = []  
let function1 =  
    let list1 = [1; 2; 3]  
    let list1 = []  
    list1
```

但是，下列程式碼在任何範圍層級都可以使用：


```
let list1 = [ 1; 2; 3]
let sumPlus x =
// OK: inner list1 hides the outer list1.
  let list1 = [1; 5; 10]
  x + List.sum list1
```

參數

參數名稱必須列於函式名稱後面。您可以指定參數的類型，如下列範例所示：

```
let f (x : int) = x + 1
```

若您指定類型，請將它放在參數名稱之後，並且以冒號來分隔類型與名稱。若您省略此參數的類型，編譯器便會推斷參數類型。例如，在下列函式定義中，引數 `x` 經推斷為 `int` 類型，因為 `1` 是 `int` 類型。

```
let f x = x + 1
```

不過，編譯器會嘗試將函式盡可能推斷成為泛型。例如，請注意下列程式碼：

```
let f x = (x, x)
```

函式會從任何類型的一個引數建立元組。因為沒有指定類型，所以函式可以與任何引數類型搭配使用。如需詳細資訊，請參閱[自動產生](#)。

函式主體

函式主體可以包含區域變數和函式的定義。這類變數和函式是在目前函式主體的範圍內，而不是在主體以外。啟用輕量型語法選項時，必須使用縮排來表示定義是在函式主體中，如下列範例所示：

```
let cylinderVolume radius length =
  // Define a local value pi.
  let pi = 3.14159
  length * pi * radius * radius
```

如需詳細資訊，請參閱[程式碼格式化方針](#)和[詳細語法](#)。

傳回值

編譯器會使用函式主體中的最終運算式，來判斷傳回值和類型。編譯器可能會從先前的運算式推斷最終運算式的類型。在上節示範的 `cylinderVolume` 函式中，`pi` 的類型是從常值 `3.14159` 的類型經判斷為 `float`。編譯器會使用 `pi` 的類型，判斷運算式 `h * pi * r * r` 的類型為 `float`。因此，函式的整體傳回型別為 `float`。

若要明確指定傳回值，請撰寫如下的程式碼：

```
let cylinderVolume radius length : float =
  // Define a local value pi.
  let pi = 3.14159
  length * pi * radius * radius
```

撰寫如上所示的程式碼時，編譯器會將 `float` 套用至整個函式。若您也要將它套用至參數類型，請使用下列程式碼：

```
let cylinderVolume (radius : float) (length : float) : float
```

呼叫函式

您可以指定函式名稱，後面接著空格，再接著以空格分隔的任何引數來呼叫函式。例如，若要呼叫函式 `cylinderVolume` 並將結果指派給值 `vol`，您可以撰寫下列程式碼：

```
let vol = cylinderVolume 2.0 3.0
```

部分套用引數

若您提供的引數數目比指定的數目更少，則必須建立需要其餘引數的新函式。這個處理引數的方法稱為「局部調用」**，是 F# 這類函式程式設計語言的特色。例如，假設您要使用兩種大小的管子，其中一個半徑為 2.0，另一個半徑為 3.0。您可以建立會判斷管子容量的函式，如下所示：

```
let smallPipeRadius = 2.0
let bigPipeRadius = 3.0

// These define functions that take the length as a remaining
// argument:

let smallPipeVolume = cylinderVolume smallPipeRadius
let bigPipeVolume = cylinderVolume bigPipeRadius
```

接著，視需要為兩個不同大小的各種管子長度提供其他引數：

```
let length1 = 30.0
let length2 = 40.0
let smallPipeVol1 = smallPipeVolume length1
let smallPipeVol2 = smallPipeVolume length2
let bigPipeVol1 = bigPipeVolume length1
let bigPipeVol2 = bigPipeVolume length2
```

遞迴函式

「遞迴函式」** 是會自我呼叫的函式。您必須在 `let` 關鍵字後面指定 `rec` 關鍵字來使用遞迴函式。請從函式主體中叫用遞迴函式，就像叫用任何函式呼叫一樣。下列遞迴函式會計算第 n 個的斐波上數位。Fibonacci 數字序列自古聞名，此序列中的每個連續數字都是前兩個數字的總和。

```
let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)
```

若不謹慎撰寫遞迴函式，而且也不了解特殊技巧 (例如累計值和接續符號的用法)，某些遞迴函式便可能會使程式堆疊溢位，或導致執行時沒有效率。

函式值

在 F# 中，所有函式都視為值，而實際上它們稱為「函式值」**。由於函式都是值，因此可以作為其他函式的引數，或在其他會用到這些值的內容中使用。以下是接受函式值作為引數的函式範例：

```
let apply1 (transform : int -> int) y = transform y
```

您可以使用 `->` 語彙基元來指定函式值的類型。在此語彙基元左邊是引數的類型，右邊則是傳回值。在上述範例中，`apply1` 函式會接受 `transform` 函式作為引數，其中 `transform` 是接受整數並傳回另一個整數的函式。在下列範例程式碼中，會示範 `apply1` 的用法：

```
let increment x = x + 1

let result1 = apply1 increment 100
```

在上述程式碼執行之後，`result` 的值會是 101。

多個引數是以連續 `->` 語彙基元分隔，如下列範例所示：

```
let apply2 ( f: int -> int -> int) x y = f x y

let mul x y = x * y

let result2 = apply2 mul 10 20
```

結果為 200。

Lambda 運算式

「Lambda 運算式」** 是不具名函式。在上述範例中，您可以使用 Lambda 運算式，而不定義具名函式 `increment` 和 `mul`，如下所示：

```
let result3 = apply1 (fun x -> x + 1) 100

let result4 = apply2 (fun x y -> x * y ) 10 20
```

您可以透過 `fun` 關鍵字定義 Lambda 運算式。Lambda 運算式類似函式定義，但是它使用 `->` 語彙基元來分隔引數清單與函式主體，而不是 `=` 語彙基元。如同一般函式定義，您可以推斷或明確指定引數類型，而 Lambda 運算式的傳回型別是從主體中最後一個運算式的類型來推斷。如需詳細資訊，請參閱 [Lambda 運算式](#)：`fun` 關鍵字。

函式組合和管線

F# 中的函式可以從其他函式組合。兩個函式 `function1` 和 `function2` 會組合成另一個函式，表示先套用 `function1`，接著套用 `function2`：

```
let function1 x = x + 1
let function2 x = x * 2
let h = function1 >> function2
let result5 = h 100
```

結果為 202。

管線可讓函式呼叫鏈結在一起成為連續作業。管線運作方式如下：

```
let result = 100 |> function1 |> function2
```

結果同樣是 202。

組合運算子會採用兩個函式，並傳回一個函式。相較之下，管線運算子則採用一個函式與一個引數，並傳回一個值。下列程式碼範例可透過示範函式簽章和使用方式的不同，顯示管線和組合運算子之間的差異。

```
// Function composition and pipeline operators compared.

let addOne x = x + 1
let timesTwo x = 2 * x

// Composition operator
// ( >> ) : ('T1 -> 'T2) -> ('T2 -> 'T3) -> 'T1 -> 'T3
let Compose2 = addOne >> timesTwo

// Backward composition operator
// ( << ) : ('T2 -> 'T3) -> ('T1 -> 'T2) -> 'T1 -> 'T3
let Compose1 = addOne << timesTwo

// Result is 5
let result1 = Compose1 2

// Result is 6
let result2 = Compose2 2

// Pipelining
// Pipeline operator
// ( |> ) : 'T1 -> ('T1 -> 'U) -> 'U
let Pipeline2 x = addOne x |> timesTwo

// Backward pipeline operator
// ( <| ) : ('T -> 'U) -> 'T -> 'U
let Pipeline1 x = addOne <| timesTwo x

// Result is 5
let result3 = Pipeline1 2

// Result is 6
let result4 = Pipeline2 2
```

多載函式

您可以多載類型的方法，但不是函式的方法。如需詳細資訊，請參閱[方法](#)。

請參閱

- [值](#)
- [F# 語言參考](#)

let 繫結

2020/11/2 • [Edit Online](#)

系統會將識別碼與值或函數產生關聯。您可以使用 `let` 關鍵字將名稱系結至值或函數。

語法

```
// Binding a value:
let identifier-or-pattern [: type] =expressionbody-expression
// Binding a function value:
let identifier parameter-list [: return-type ] =expressionbody-expression
```

備註

`let` 關鍵字是在系結運算式中用來定義一個或多個名稱的值或函數值。運算式最簡單的形式會將名稱系結 `let` 至簡單的值，如下所示。

```
let i = 1
```

如果您使用新的行將運算式與識別碼分開，則必須縮排運算式的每一行，如下列程式碼所示。

```
let someVeryLongIdentifier =
  // Note indentation below.
  3 * 4 + 5 * 6
```

如下列程式碼所示，您可以指定包含名稱的模式，而不只是名稱。

```
let i, j, k = (1, 2, 3)
```

主體運算式是使用名稱的運算式。本文運算式會出現在自己的行上，並以關鍵字中的第一個字元縮排對齊 `let`：

```
let result =

  let i, j, k = (1, 2, 3)

  // Body expression:
  i + 2*j + 3*k
```

系統 `let` 可以出現在模組層級、類別類型的定義或區域範圍中，例如在函式定義中。`let` 模組或類別類型中最上層的系結不需要有內文運算式，但在其他範圍層級，則需要主體運算式。系結名稱可在定義點之後使用，但不能在系結出現之前的任何時間點使用 `let`，如下列程式碼所示。

```
// Error:
printfn "%d" x
let x = 100
// OK:
printfn "%d" x
```

函數系結

函數系結會遵循值系結的規則，但函數系結會包含函數名稱和參數，如下列程式碼所示。

```
let function1 a =  
  a + 1
```

一般情況下，參數是模式，例如元組模式：

```
let function2 (a, b) = a + b
```

系結 `let` 運算式會評估為最後一個運算式的值。因此，在下列程式碼範例中，的值 `result` 是從 `100 * function3 (1, 2)` 評估為的計算 `300`。

```
let result =  
  let function3 (a, b) = a + b  
  100 * function3 (1, 2)
```

如需詳細資訊，請參閱[函式](#)。

類型注釋

您可以指定參數的類型，方法是包含冒號 (:) 後面接著類型名稱，全都以括弧括住。您也可以在最後一個參數之後附加冒號和類型，以指定傳回值的類型。的完整類型注釋 `function1` (使用整數做為參數類型) 如下所示。

```
let function1 (a: int) : int = a + 1
```

如果沒有明確的型別參數，則會使用型別推斷來判斷函式的參數類型。這可能包括自動將參數的類型一般化為泛型。

如需詳細資訊，請參閱 [自動一般化](#) 和 [型別推斷](#)。

類別中的 let 繫結

系結 `let` 可以出現在類別型別中，但不能出現在結構或記錄型別中。若要在類別型別中使用 `let` 系結，類別必須有主要的函式。函式參數必須出現在類別定義中的類型名稱之後。類別型別中的系結 `let` 會定義該類別型別的私用欄位和成員，並且與型別中的系結形成型別的系結程式 `do` 代碼。下列程式碼範例會顯示 `MyClass` 具有私用欄位 `field1` 和的類別 `field2`。

```
type MyClass(a) =  
  let field1 = a  
  let field2 = "text"  
  do printfn "%d %s" field1 field2  
  member this.F input =  
    printfn "Field1 %d Field2 %s Input %A" field1 field2 input
```

和的範圍 `field1` `field2` 僅限於宣告它們的型別。如需詳細資訊，請參閱類別和[類別 let](#) 中的系結。

Let 系結中的型別參數

在 `let` 模組層級、型別或計算運算式中的系結可以有明確的型別參數。運算式中的 `let` 系結 (例如在函式定義內) 不能有型別參數。如需詳細資訊，請參閱[泛型](#)。

Let 系結上的屬性

屬性可以套用至模組中的最上層系結 `let`，如下列程式碼所示。

```
[<Obsolete>]  
let function1 x y = x + y
```

Let 系結的範圍和協助工具

以 `let` 系結宣告的實體範圍僅限於包含範圍 (的部分，例如函式、模組、檔案或類別在系結出現之後)。因此，您可以說 `let` 系結將名稱引入範圍中。在模組中，只要模組可供存取，模組的用戶端就可以存取 `let` 系結值或函數，因為模組中的 `let` 系結會編譯成模組的公用函式。相較之下，「讓類別中的系結」是類別的私用。

一般來說，當用戶端程式代碼使用模組時，模組中的函式必須以模組的名稱來限定。例如，如果模組 `Module1` 有一個函式 `function1`，則使用者會指定 `Module1.function1` 參考該函數。

模組的使用者可能會使用匯入宣告，讓該模組內的函式可供使用，而不會受到模組名稱的限定。在剛才提及的範例中，模組的使用者可以在該案例中使用匯入宣告開啟模組，`Module1` 之後再參考 `function1` 直接。

```
module Module1 =  
    let function1 x = x + 1.0  
  
module Module2 =  
    let function2 x =  
        Module1.function1 x  
  
open Module1  
  
let function3 x =  
    function1 x
```

某些模組具有 `RequireQualifiedAccess` 屬性，這表示它們所公開的函式必須以模組的名稱來限定。例如，F# 清單模組有這個屬性。

如需模組和存取控制的詳細資訊，請參閱 [模組](#) 和 [存取控制](#)。

請參閱

- [函式](#)
- `let` [類別中的系結](#)

do 繫結

2019/10/23 • [Edit Online](#)

`do` 系結是用來執行程式碼, 而不需要定義函式或值。此外, 您可以在類別中使用系結, 請參閱 `do` [類別中的系結](#)。

語法

```
[ attributes ]  
[ do ]expression
```

備註

`do` 當您想要獨立執行函式或值定義以外的程式碼時, 請使用系結。系結中 `do` 的運算式 `unit` 必須傳回。當模組初始化時, 會 `do` 執行最上層系結中的程式碼。關鍵字 `do` 是選擇性的。

屬性可以套用至最上層 `do` 系結。例如, 如果您的程式使用 COM Interop, 您可能會想要將 `STAThread` 屬性套用至您的程式。您可以使用系結上 `do` 的屬性來執行此動作, 如下列程式碼所示。

```
open System  
open System.Windows.Forms  
  
let form1 = new Form()  
form1.Text <- "XYZ"  
  
[<STAThread>]  
do  
    Application.Run(form1)
```

另請參閱

- [F# 語言參考](#)
- [函式](#)

Lambda 運算式:有趣的關鍵字 (F#)

2019/10/23 • [Edit Online](#)

`fun` 關鍵字是用來定義 lambda 運算式, 也就是匿名函式。

語法

```
fun parameter-list -> expression
```

備註

參數清單通常包含名稱和選擇性的參數類型。更常見的情況是,參數清單可以由任何F#模式組成。如需可能模式的完整清單,請參閱[模式](#)比對。有效參數的清單包含下列範例。

```
// Lambda expressions with parameter lists.  
fun a b c -> ...  
fun (a: int) b c -> ...  
fun (a : int) (b : string) (c:float) -> ...  
  
// A lambda expression with a tuple pattern.  
fun (a, b) -> ...  
  
// A lambda expression with a list pattern.  
fun head :: tail -> ...
```

運算式是函式的主體, 最後一個運算式會產生傳回值。有效 lambda 運算式的範例包括下列各項:

```
fun x -> x + 1  
fun a b c -> printfn "%A %A %A" a b c  
fun (a: int) (b: int) (c: int) -> a + b * c  
fun x y -> let swap (a, b) = (b, a) in swap (x, y)
```

使用 Lambda 運算式

當您想要對清單或其他集合執行作業, 而且想要避免定義函數的額外工作時, Lambda 運算式特別有用。許多F#程式庫函式會採用函式值做為引數, 在這些情況下使用 lambda 運算式會特別方便。下列程式碼會將 lambda 運算式套用至清單的元素。在此情況下, 匿名函式會將1新增至清單的每個元素。

```
let list = List.map (fun i -> i + 1) [1;2;3]  
printfn "%A" list
```

另請參閱

- [函式](#)

遞迴函式：rec 關鍵字

2021/3/5 • [Edit Online](#)

`rec` 關鍵字會與關鍵字一起使用 `let`，以定義遞迴函數。

語法

```
// Recursive function:
let rec function-nameparameter-list =
    function-body

// Mutually recursive functions:
let rec function1-nameparameter-list =
    function1-body

and function2-nameparameter-list =
    function2-body

...
```

備註

遞迴函式-呼叫本身的函式會在 F# 語言中使用關鍵字明確識別 `rec`。關鍵字讓系結的 `rec` 名稱 `let` 可在其主體中使用。

下列範例顯示的遞迴函式，會使用數學定義來計算第 n 個斐的量值。

```
let rec fib n =
    match n with
    | 0 | 1 -> n
    | n -> fib (n-1) + fib (n-2)
```

NOTE

在實務上，上述範例之類的程式碼並不理想，因為它 unnecessarily 已計算的旁邊值。這是因為它不是 tail 遞迴，這會在本文中進一步說明。

方法會在其定義的類型內隱含遞迴，這表示不需要加入 `rec` 關鍵字。例如：

```
type MyClass() =
    member this.Fib(n) =
        match n with
        | 0 | 1 -> n
        | n -> this.Fib(n-1) + this.Fib(n-2)
```

不過，在類別中的系結並非隱含遞迴。所有系結 `let` 函數都需要 `rec` 關鍵字。

尾遞迴

針對某些遞迴函式，必須將更多「單純」定義重構為 [結尾遞迴](#) 的定義。這可避免不必要的 recomputations。例如，您可以如下所示重新撰寫先前的量值產生器：

```
let fib n =
  let rec loop acc1 acc2 n =
    match n with
    | 0 -> acc1
    | 1 -> acc2
    | _ ->
      loop acc2 (acc1 + acc2) (n - 1)
  loop 0 1 n
```

這是更複雜的執行。產生數量詞是一種非常簡單的「一般」演算法的絕佳範例，但實際上並沒有效率。在 F # 中，有幾個層面會讓它有效率，但仍會以遞迴方式定義：

- 名為的遞迴內建函式 `loop`，它是慣用 F # 模式。
- 將累積值傳遞給遞迴呼叫的兩個累積參數。
- 檢查的值 `n`，以傳回特定的累積。

如果此範例是使用迴圈反復寫入，則程式碼看起來會像是兩個不同的值累積數目，直到符合特定條件為止。

這是結尾遞迴的原因，是因為遞迴呼叫不需要在呼叫堆疊上儲存任何值。所有計算的中繼值都會透過輸入至內建函式來累積。這也可讓 F # 編譯器將程式碼優化，就像您撰寫迴圈一樣快 `while`。

通常會撰寫 F # 程式碼，以遞迴方式處理具有內部和外部函式的某個內容，如先前範例所示。內部函式會使用尾遞迴，而外部函式則是更好的呼叫端介面。

相互遞迴函數

有時候函式是 *相互遞迴* 的，這表示呼叫會形成圓形，其中一個函式會呼叫另一個函式，而後者會呼叫另一個函式，並在其間有任意數目的呼叫。您必須在一個系結中一起定義這類函數 `let`，`and` 並使用關鍵字將它們連結在一起。

下列範例顯示兩個互相遞迴的函式。

```
let rec Even x =
  if x = 0 then true
  else Odd (x-1)
and Odd x =
  if x = 0 then false
  else Even (x-1)
```

遞迴值

您也可以將系結的 `let` 值定義為遞迴。這有時候是為了記錄而進行的。使用 F # 5 和函式 `nameof`，您可以執行下列動作：

```
let rec nameDoubles = nameof nameDoubles + nameof nameDoubles
```

請參閱

- [函式](#)

進入點

2019/10/23 • [Edit Online](#)

本主題說明用來設定F#程式進入點的方法。

語法

```
[<EntryPoint>]  
let-function-binding
```

備註

在先前的語法中, *let* 函數系結是系結中 `let` 函式的定義。

編譯為可執行檔之程式的進入點是執行正式開始的位置。您可以藉由F# `EntryPoint` 將屬性套用至程式的 `main` 函式, 來指定應用程式的進入點。這個函式 (使用 `let` 系結所建立) 必須是上次編譯檔案中的最後一個函式。上次編譯的檔案是專案中的最後一個檔案, 或傳遞至命令列的最後一個檔案。

進入點函式具有類型 `string array -> int`。在命令列上提供的引數會傳遞至 `main` 字串陣列中的函式。陣列的第一個元素是第一個引數;可執行檔的名稱不會包含在陣列中, 因為它是在某些其他語言中。傳回值會當做進程的結束代碼使用。零通常表示成功;非零值表示發生錯誤。非零傳回碼的特定意義沒有任何慣例;傳回碼的意義是應用程式特有的。

下列範例說明簡單 `main` 的函式。

```
[<EntryPoint>]  
let main args =  
    printfn "Arguments passed to function : %A" args  
    // Return 0. This indicates success.  
    0
```

當使用命令列 `EntryPoint.exe 1 2 3` 來執行此程式碼時, 輸出會如下所示。

```
Arguments passed to function : [|"1"; "2"; "3"|]
```

隱含進入點

當程式沒有明確指出進入點的`EntryPoint`屬性時, 最後一個要編譯的檔案中的最上層系結會當做進入點使用。

另請參閱

- [函式](#)
- [let 繫結](#)

外部函式

2019/10/23 • [Edit Online](#)

本主題描述 F# 在機器碼中呼叫函數的語言支援。

語法

```
[<DllImport( arguments )>]  
extern declaration
```

備註

在先前的語法中, *引數* 代表提供給 `System.Runtime.InteropServices.DllImportAttribute` 屬性的引數。第一個引數是字串, 代表包含此函式之 DLL 的名稱, 但不含 .dll 副檔名。可以為

`System.Runtime.InteropServices.DllImportAttribute` 類別的任何公用屬性提供其他引數, 例如呼叫慣例。

假設您有一個包含 C++ 下列匯出函式的原生 DLL。

```
#include <stdio.h>  
extern "C" void __declspec(dllexport) HelloWorld()  
{  
    printf("Hello world, invoked by F#\n");  
}
```

您可以使用下列程式碼 F#, 從呼叫這個函式。

```
open System.Runtime.InteropServices  
  
module InteropWithNative =  
    [DllImport(@"C:\bin\nativedll", CallingConvention = CallingConvention.Cdecl)]  
    extern void HelloWorld()  
  
InteropWithNative.HelloWorld()
```

與機器碼的互通性稱為「*平台叫用*」, 它是 CLR 的一項功能。如需詳細資訊, 請參閱 [與 Unmanaged 程式碼互通](#)。該區段中的資訊適用於 F#。

另請參閱

- [函式](#)

內嵌函式

2019/10/23 • [Edit Online](#)

內嵌函式是直接整合到呼叫程式碼中的函式。

使用內嵌函數

當您使用靜態類型參數時, 以類型參數參數化的任何函式都必須是內嵌的。這可保證編譯器可以解析這些型別參數。當您使用一般泛型型別參數時, 沒有這類限制。

除了啟用成員條件約束以外, 內嵌函數也有助於優化程式碼。不過, 過度利用內嵌函式可能會導致您的程式碼較不能抵抗編譯器優化和程式庫函式的變更。基於這個理由, 您應該避免使用內嵌函式進行優化, 除非您已經嘗試過所有其他的優化技術。讓函式或方法內嵌的作業有時可以改善效能, 但不一定都是如此。因此, 您也應該使用效能測量來確認讓任何指定的函式內嵌確實具有正面的效果。

`inline` 修飾詞可以套用至最上層的函式、模組層級, 或類別中的方法層級。

下列程式碼範例說明最上層的內嵌函式、內嵌實例方法, 以及內嵌的靜態方法。

```
let inline increment x = x + 1
type WrapInt32() =
    member inline this.incrementByOne(x) = x + 1
    static member inline Increment(x) = x + 1
```

內嵌函式和型別推斷

的存在 `inline` 會影響型別推斷。這是因為內嵌函式可以有靜態解析的類型參數, 而非內嵌函數則不能。下列

`inline` 程式碼範例顯示的案例很有用 `float`, 因為您使用的函式具有靜態解析的類型參數, 也就是轉換運算子。

```
let inline printAsFloatingPoint number =
    printfn "%f" (float number)
```

如果沒有 `int` 修飾詞, 型別推斷會強制函式採用特定的型別, 在此案例中為 `inline`。但是使用 `inline` 修飾詞時, 函式也會推斷為具有靜態解析的類型參數。 `inline` 使用修飾詞時, 會推斷型別, 如下所示:

```
^a -> unit when ^a : (static member op_Explicit : ^a -> float)
```

這表示函式會接受任何支援轉換成 `float` 的類型。

另請參閱

- [函式](#)
- [條件約束](#)
- [以統計方式解析的類型參數](#)

值

2019/10/23 • [Edit Online](#)

F# 中的值是具有特定類型的數量；值可以是整數或浮點數、字元或文字、清單、序列、陣列、元組、差別聯集、記錄、類別類型或函式值。

繫結值

「繫結」一詞表示建立名稱與定義的關聯。`let` 關鍵字會繫結值，如下列範例所示：

```
let a = 1
let b = 100u
let str = "text"

// A function value binding.

let f x = x + 1
```

值的類型是從定義推斷的。若為基本類型 (例如整數或浮點數)，則是從常值的類型來決定類型。因此，在上述範例中，編譯器推斷 `b` 的類型是 `unsigned int`，而推斷 `a` 的類型是 `int`。函式值的類型是從函式主體中的傳回值決定。如需函式值類型的詳細資訊，請參閱[函式](#)。如需常值型別的詳細資訊，請參閱[常值](#)。

根據預設，編譯器不會針對未使用的系結髮出診斷。若要接收這些訊息，請在您的專案檔中或叫用編譯器時啟用 `--warnon` 警告 1182 (請參閱[編譯器選項](#)底下)。

使用不可變的原因

不可變的值是在程式執行過程中無法變更的值。若您習慣使用 C++、Visual Basic 或 C# 等語言，F# 首重不可變的值，而不是程式執行期間可指派新值的變數，這可能讓您訝異。不可變的資料是函式程式設計的重要項目。在多執行緒環境中，難以管理可由許多不同的執行緒變更的共用可變變數。此外，對於可變變數，有時候難以判斷變數是否會在傳遞至另一個函式時變更。

在純函式語言中，沒有變數，而且函式只作為數學函式。程序語言中的程式碼使用變數指派來變更值，而在函式語言中的對等程式碼則有作為輸入的不可變值、不可變的函式，以及作為輸出的不同不可變值。如此在數學方面嚴謹的程度，可以讓程式的行為更符合預期。如此一來，編譯器就能更嚴格地檢查程式碼，並且在最佳化時更有效率，也有助於程式開發人員更容易了解及撰寫正確程式碼。所以，比起一般程序程式碼，偵錯函式程式碼也更為容易。

F# 不是純函式語言，但完全支援函式程式設計。使用不可變的值是良好做法，因為這麼做可讓程式碼從函式程式設計的重要層面獲益。

可變變數

您可以使用 `mutable` 關鍵字指定可變更的變數。F# 的可變變數通常應該有限制範圍，可以是類型欄位或區域值。有限制範圍的可變變數更容易控制，同時也比較不會遭到誤改。

您可以透過與定義值相同的方式使用 `let` 關鍵字，將初始值指派給可變變數。不過，差異在於後續可以透過 `<-` 運算子將新值指派給可變變數，如下列範例所示。

```
let mutable x = 1
x <- x + 1
```

標示 `mutable` 的值可能會自動升 `'a ref` 階為 (如果被關閉所捕捉), 包括建立像 `seq` 是產生器等的表單。如果您想要在發生這種情況時收到通知, 請在您的專案檔中或叫用編譯器時啟用警告3180。

相關主題

“	“
let 繫結	提供使用關鍵字將名稱 <code>let</code> 系結至值和函數的相關資訊。
函式	提供 F# 函式的概觀。

另請參閱

- [Null 值](#)
- [F# 語言參考](#)

Null 值

2020/11/2 • [Edit Online](#)

本主題說明如何在 F # 中使用 null 值。

Null 值

Null 值通常不會用於 F # 中的值或變數。不過，在某些情況下，null 會顯示為異常值。如果類型是在 F # 中定義，除非將 [AllowNullLiteral](#) 屬性套用至類型，否則不允許使用 null 作為一般值。如果類型是以其他 .NET 語言定義，null 是可能的值，而且當您與這類型別互通時，F # 程式碼可能會遇到 null 值。

針對 F # 中定義且嚴格地使用 f # 的型別，直接使用 F # 程式庫建立 null 值的唯一方法是使用 [Unchecked.defaultof](#) 或 [array2d.zeroscreate](#)。不過，針對從其他 .NET 語言使用的 F # 類型，或如果您使用的是不是以 F # 撰寫的 API (例如 .NET Framework)，則可能會發生 null 值。

您可以使用 `option` F # 中的型別，因為您可能會在另一個 .net 語言中使用具有可能 null 值的參考變數。

`option` 如果沒有任何物件，則您可以使用選項值(如果沒有物件)，而不是 null `None`。 `Some(obj)` 當有物件時，您可以使用選項值與物件 `obj`。如需詳細資訊，請參閱[選項](#)。請注意，如果是，則您仍然可以 `null` 將值封裝至選項中 `Some x` `x` `null`。因此，當值為時，請務必使用此 `None` 值 `null`。

`null` 關鍵字是 F # 語言中的有效關鍵字，當您使用以其他 .net 語言撰寫的 .NET Framework api 或其他 api 時，您必須使用此關鍵字。在兩種情況下，您可能需要 null 值，就是當您呼叫 .NET API 並將 null 值當作引數傳遞時，以及當您從 .NET 方法呼叫中解讀傳回值或輸出參數時。

若要將 null 值傳遞至 .NET 方法，只需在 `null` 呼叫程式碼中使用關鍵字。下列程式碼範例會說明這點。

```
open System

// Pass a null value to a .NET method.
let ParseDateTime (str: string) =
    let (success, res) = DateTime.TryParse(str, null, System.Globalization.DateTimeStyles.AssumeUniversal)
    if success then
        Some(res)
    else
        None
```

若要解讀從 .NET 方法取得的 null 值，請使用模式比對(如果可以的話)。下列程式碼範例示範如何使用模式比對來解讀 `ReadLine` 當它嘗試讀取超過輸入資料流程結尾時所傳回的 null 值。

```
// Open a file and create a stream reader.
let fileStream1 =
    try
        System.IO.File.OpenRead("TextFile1.txt")
    with
        | :? System.IO.FileNotFoundException -> printfn "Error: TextFile1.txt not found."; exit(1)

let streamReader = new System.IO.StreamReader(fileStream1)

// ProcessNextLine returns false when there is no more input;
// it returns true when there is more input.
let ProcessNextLine nextLine =
    match nextLine with
    | null -> false
    | inputString ->
        match ParseDateTime inputString with
        | Some(date) -> printfn "%s" (date.ToLocalTime().ToString())
        | None -> printfn "Failed to parse the input."
        true

// A null value returned from .NET method ReadLine when there is
// no more input.
while ProcessNextLine (streamReader.ReadLine()) do ()
```

F # 類型的 Null 值也可以使用其他方式產生，例如，當您使用時，會 `Array.zeroCreate` 呼叫 `Unchecked.defaultof`。您必須小心使用這類程式碼，以將 null 值保持封裝。在僅供 F # 使用的程式庫中，您不需要在每個函式中檢查是否有 null 值。如果您要撰寫與其他 .NET 語言交互操作的程式庫，您可能必須加入 null 輸入參數的檢查並擲回 `ArgumentNullException`，就像在 c # 或 Visual Basic 程式碼中所做的一樣。

您可以使用下列程式碼來檢查任意值是否為 null。

```
match box value with
| null -> printf "The value is null."
| _ -> printf "The value is not null."
```

另請參閱

- [值](#)
- [比對運算式](#)

常值

2020/11/2 • [Edit Online](#)

本文提供的表格顯示如何以 F # 指定常值的類型。

常值類型

下表顯示 F # 中的常數值型別。以十六進位標記法表示數位的字元不區分大小寫;識別類型的字元會區分大小寫。

「	「	「	「
sbyte	帶正負號的8位整數	y	<code>86y</code> <code>0b00000101y</code>
byte	不帶正負號的8位自然數位	uy	<code>86uy</code> <code>0b00000101uy</code>
int16	帶正負號的16位整數	s	<code>86s</code>
uint16	不帶正負號的16位自然數位	us	<code>86us</code>
int	帶正負號的32位整數	l 或 none	<code>86</code>
int32			<code>86l</code>
uint	不帶正負號的32位自然數位	u 或 ul	<code>86u</code>
uint32			<code>86ul</code>
nativeint	帶正負號之自然數的原生指標	n	<code>123n</code>
unativeint	原生指標做為不帶正負號的自然數位	聯合國	<code>0x00002D3Fun</code>
int64	帶正負號的64位整數	L	<code>86L</code>
uint64	不帶正負號的64位自然數位	UL	<code>86UL</code>
single、float32	32位浮點數	F 或 f	<code>4.14F</code> 或 <code>4.14f</code>
		如果	<code>0x00000001f</code>
浮點雙	64位浮點數	無	<code>4.14</code> 、 <code>2.3E+32</code> 或 <code>2.3e+32</code>
		如果	<code>0x000000000000000LF</code>

備註

Unicode 字串可以包含您可以使用來指定的明確編碼方式，`\u` 後面接著16位的十六進位程式碼 (0000-FFFF) 或您可以使用指定的32編碼，`\u` 並在後面接著32位的十六進位程式碼，表示任何 Unicode 程式碼點 (00000000 0010FFFF)。

不允許使用以外的其他位運算子 `|||`。

其他基底中的整數

您也可以 `0x` `0o` 分別使用或前置詞，以十六進位、八進位或二進位檔來指定已簽署的32位整數 `0b`。

```
let numbers = (0x9F, 0o77, 0b1010)
// Result: numbers : int * int * int = (159, 63, 10)
```

數值常值中的底線

您可以使用底線字元 `()` 來分隔數位 `_`。

```
let value = 0xDEAD_BEEF

let valueAsBits = 0b1101_1110_1010_1101_1011_1110_1110_1111

let exampleSSN = 123_456_7890
```

F# 類型

2020/11/2 • [Edit Online](#)

本主題描述 F# 中使用的類型，以及 F# 類型的命名和描述方式。

F# 類型摘要

某些類型會被視為 *基本類型*，例如各種大小的布林型別 `bool` 和整數和浮點數類型，包括位元組和字元的類型。這些類型會在 [基本類型](#) 中描述。

語言內建的其他類型包括元組、清單、陣列、序列、記錄和區分等位。如果您有使用其他 .NET 語言的經驗，且正在學習 F#，則應該閱讀每一種類型的相關主題。這些 F# 特定類型支援函式程式設計語言常見的程式設計樣式。其中許多型別在 F# 程式庫中都有相關聯的模組，可支援這些類型上的一般作業。

函數的類型包含參數類型和傳回類型的相關資訊。

.NET Framework 是物件類型、介面類別型、委派類型和其他專案的來源。您可以定義自己的物件類型，就像在任何其他 .NET 語言中一樣。

此外，F# 程式碼可以定義別名（名稱為「*類型縮寫*」），這是類型的替代名稱。當類型未來可能會變更，而且您想要避免變更相依赖于型別的程式碼時，您可以使用類型縮寫。或者，您可以使用類型縮寫做為類型的易記名稱，使程式碼更容易閱讀和瞭解。

F# 提供實用的集合型別，這些型別是以功能程式設計為考慮而設計的。使用這些集合類型，可協助您撰寫更能以樣式運作的程式碼。如需詳細資訊，請參閱 [F# 集合類型](#)。

類型的語法

在 F# 程式碼中，您通常必須寫出類型的名稱。每個類型都有一個語法形式，而您可以在類型批註、抽象方法宣告、委派宣告、簽章和其他結構中使用這些語法形式。每當您在解譯器中宣告新的程式結構時，解譯器就會列印結構的名稱和其類型的語法。這個語法可能只是使用者自訂類型的識別碼，也可能是內建的識別碼（例如 `int` 或 `string`），但對於更複雜的型別來說，語法較複雜。

下表顯示 F# 型別語法的各個層面。

II	IIII	II
基本類型	類型名稱	<code>int</code> <code>float</code> <code>string</code>
匯總類型 (類別、結構、等位、記錄、列舉等等)	類型名稱	<code>System.DateTime</code> <code>Color</code>
類型縮寫	類型縮寫-名稱	<code>bigint</code>

II	IIII	II
完整型別	命名空間。類型名稱 或 模組。類型名稱 或 命名空間。模組。類型名稱	<code>System.IO.StreamWriter</code>
array	類型名稱[] 或 類型名稱 陣列	<code>int[]</code> <code>array<int></code> <code>int array</code>
二維陣列	類型名稱[,]	<code>int[,]</code> <code>float[,]</code>
三維陣列	類型名稱[, ,]	<code>float[, ,]</code>
Tuple	類型-name1 * 類型-name2 ..。	例如, <code>(1, 'b', 3)</code> 具有類型 <code>int * char * int</code>
Generic Type - 泛型類型	類型參數**泛型型別名稱 或 泛型型別名稱 <類型參數-清單>	<code>'a list</code> <code>list<'a></code> <code>Dictionary<'key, 'value></code>
結構化類型 (提供特定型別引數的泛型型別)	類型-引數**泛型型別名稱 或 泛型型別名稱 <類型-引數-清單>	<code>int option</code> <code>string list</code> <code>int ref</code> <code>option<int></code> <code>list<string></code> <code>ref<int></code> <code>Dictionary<int, string></code>
具有單一參數的函數類型	參數-type1 - >傳回類型	接受 <code>int</code> 並傳回具有類型的函式 <code>string`int -> string</code>
具有多個參數的函數類型	參數-type1 - > 參數-type2 - > ...-傳回 > 類型	接受和的函式, <code>int</code> <code>float</code> 並傳回 <code>string</code> 具有類型的 <code>int -> float -> string</code>
較高的 order 函數作為參數	(函數類型)	<code>List.map</code> 具有類型 <code>('a -> 'b) -> 'a list -> 'b list</code>

☐☐	☐☐☐	☐☐
Delegate - 委派	函式類型的委派	<code>delegate of unit -> int</code>
彈性類型	<i>#類型名稱</i>	<code>#System.Windows.Forms.Control</code> <code>#seq<int></code>

[相關主題]

☐☐	☐☐
基本類型	描述內建的簡單類型，例如整數類資料類型、布林值類型和字元類型。
單位類型	描述 <code>unit</code> 類型、具有一個值的型別，以及由 (<code># A1;</code> 表示的類型，相當於 <code>void</code> c# 中和 <code>Nothing</code> Visual Basic。
元組	描述元組類型，這個類型是由任何類型的關聯值（以配對、三合一、時等分組）所組成。
選項	描述選項類型，此類型可能具有值或為空白。
清單	描述清單，也就是已排序的專案，這是所有相同類型的不可變元素系列。
陣列	描述陣列，這些陣列是相同類型之可變動專案的已排序集合，且這些元素佔用連續的記憶體區塊，且大小為固定大小。
序列	描述代表邏輯序列值的序列類型。個別的值只會在必要時計算。
記錄	描述記錄類型，這是一個小型的命名值匯總。
已區分的聯集	描述差異聯集類型，此類型的值可以是一組可能類型的任一個。
函式	描述函數值。
類別	描述類別類型，這是對應至 .NET 參考型別的物件類型。類別類型可以包含成員、屬性、實介面和基底類型。
結構	描述 <code>struct</code> 型別，這是對應至 .net 值型別的物件類型。此 <code>struct</code> 類型通常代表較小的資料匯總。
介面	描述介面型別，這些類型代表一組提供特定功能但未包含任何資料的成員。介面型別必須由物件型別實作為有用。
委派	描述表示函式做為物件的委派類型。
列舉	描述列舉型別，其值屬於一組命名值。
屬性	描述用來指定另一種類型之中繼資料的屬性。

❏	❏
例外狀況類型	描述指定錯誤資訊的例外狀況。

類型推斷

2019/10/23 • [Edit Online](#)

本主題描述F#編譯器如何推斷值、變數、參數和傳回值的類型。

一般型別推斷

類型推斷的概念是, 您不需要指定F#結構的類型, 除非編譯器無法最終推算類型。省略明確類型資訊並不表示是F#動態類型的語言, 或中F#的值是弱式類型。F#是靜態類型語言, 這表示編譯器會在編譯期間會推算每個結構的確切類型。如果沒有足夠的資訊可讓編譯器推算出每個結構的類型, 您就必須提供額外的類型資訊, 通常是在程式碼中的某處新增明確的類型註釋。

參數和傳回類型的推斷

在參數清單中, 您不需要指定每個參數的類型。此外, F#是靜態類型的語言, 因此每個值和運算式在編譯時期都有一個明確的類型。針對您未明確指定的類型, 編譯器會根據內容來推斷類型。如果未另行指定類型, 則會將它推斷為泛型。如果程式碼以不一致的方式使用值, 在這種情況下, 如果沒有任何單一推斷類型滿足值的所有使用, 編譯器就會報告錯誤。

函式的傳回型別是由函式中最後一個運算式的型別所決定。

例如, 在下列程式碼中, 參數類型 `a` 和 `b` 和傳回類型 `int` 都會推斷為, 因為常 `100` 值的類型 `int` 為。

```
let f a b = a + b + 100
```

您可以藉由變更常值來影響型別推斷。如果您藉 `uint32` 由 `100` `u` 附加 `b` 尾碼來 `uint32` 建立, 則會將、和傳回值的類型推斷為。 `a`

您也可以使用其他表示類型限制的結構 (例如僅適用於特定類型的函數和方法), 來影響型別推斷。

此外, 您可以將明確類型註釋套用至函式或方法參數或運算式中的變數, 如下列範例所示。如果在不同的條件約束之間發生衝突, 則會產生錯誤。

```
// Type annotations on a parameter.
let addu1 (x : uint32) y =
    x + y

// Type annotations on an expression.
let addu2 x y =
    (x : uint32) + y
```

您也可以在所有參數之後提供類型註釋, 以明確指定函式的傳回值。

```
let addu1 x y : uint32 =
    x + y
```

類型註釋對參數很有用的常見情況是, 當參數是物件類型, 而您想要使用成員時。

```
let replace(str: string) =
    str.Replace("A", "a")
```

自動一般化

如果函式程式碼不相依赖于參數的類型, 則編譯器會將參數視為泛型。這稱為「*自動一般化*」, 它可以是在不增加複雜度的情況下撰寫一般程式碼的強大輔助工具。

例如, 下列函式會將任何類型的兩個參數結合成一個元組。

```
let makeTuple a b = (a, b)
```

類型會推斷為

```
'a -> 'b -> 'a * 'b
```

其他資訊

類型推斷會在F#語言規格中以更詳細的方式加以描述。

另請參閱

- [自動一般化](#)

基本類型

2021/3/5 • [Edit Online](#)

本主題列出在 F# 語言中定義的基本類型。這些類型在 F# 中是最基本的，形成幾乎每一個 F# 程式的基礎。它們是 .NET 基本型別的超集合。

F#	.NET	說明	字面量
<code>bool</code>	<code>Boolean</code>	可能的值是 <code>true</code> 和 <code>false</code> 。	<code>true</code> / <code>false</code>
<code>byte</code>	<code>Byte</code>	從0到255的值。	<code>1uy</code>
<code>sbyte</code>	<code>SByte</code>	介於-128 到127之間的值。	<code>1y</code>
<code>int16</code>	<code>Int16</code>	介於-32768 到32767之間的值。	<code>1s</code>
<code>uint16</code>	<code>UInt16</code>	從0到65535的值。	<code>1us</code>
<code>int</code>	<code>Int32</code>	介於-2147483648 到 2147483647之間的值。	<code>1</code>
<code>uint</code>	<code>UInt32</code>	從0到4294967295的值。	<code>1u</code>
<code>int64</code>	<code>Int64</code>	從-9223372036854775808 到9223372036854775807 的值。	<code>1L</code>
<code>uint64</code>	<code>UInt64</code>	0到 18446744073709551615 之間的值。	<code>1UL</code>
<code>nativeint</code>	<code>IntPtr</code>	作為帶正負號整數的原生指標。	<code>nativeint 1</code>
<code>unativeint</code>	<code>UIntPtr</code>	原生指標，作為不帶正負號的整數。	<code>unativeint 1</code>
<code>decimal</code>	<code>Decimal</code>	浮點數資料類型至少有28個有效位數。	<code>1.0</code>
<code>float</code> , <code>double</code>	<code>Double</code>	64位浮點數類型。	<code>1.0</code>
<code>float32</code> , <code>single</code>	<code>Single</code>	32位浮點數類型。	<code>1.0f</code>
<code>char</code>	<code>Char</code>	Unicode 字元值。	<code>'c'</code>
<code>string</code>	<code>String</code>	Unicode 文字。	<code>"str"</code>

「	.NET 「	「	「
<code>unit</code>	不適用	指出沒有實際值。型別只有一個正式值，即表示 <code>()</code> 。單位值(<code>()</code>)通常用來做為需要值的預留位置，但沒有可用的實際值或意義。	<code>()</code>

NOTE

您可以使用類型，對64位整數類型的整數執行太大的計算 `bigint`。 `bigint` 不會被視為基本類型;這是的縮寫 `System.Numerics.BigInteger`。

另請參閱

- [F # 語言參考](#)

單位類型

2019/11/4 • [Edit Online](#)

`unit` 類型是表示沒有特定值的類型。`unit` 類型只有單一值，當沒有其他值存在或需要時，它會作為預留位置。

語法

```
// The value of the unit type.  
()
```

備註

每F#個運算式都必須評估為值。若為不會產生相關值的運算式，則會使用 `unit` 類型的值。`unit` 類型類似于 C#和C++等語言中的 `void` 類型。

`unit` 類型具有單一值，且該值是由權杖 `()` 所表示。

`unit` 類型的值通常用來進程式F#設計，以保存語言語法需要值的位置，但不需要任何值或需要時。範例可能是 `printf` 函數的傳回值。因為 `printf` 作業的重要動作會在函式中發生，所以函式不需要傳回實際的值。因此，傳回值的類型為 `unit`。

某些結構預期會有 `unit` 值。例如，`do` 系結或模組最上層的任何程式碼，都應該評估為 `unit` 值。當模組最上層的 `do` 系結或程式碼產生的結果不是未使用的 `unit` 值時，編譯器會報告警告，如下列範例所示。

```
let function1 x y = x + y  
// The next line results in a compiler warning.  
function1 10 20  
// Changing the code to one of the following eliminates the warning.  
// Use this when you do want the return value.  
let result = function1 10 20  
// Use this if you are only calling the function for its side effects,  
// and do not want the return value.  
function1 10 20 |> ignore
```

此警告是功能性程式設計的特性;它不會出現在其他 .NET 程式設計語言中。在純粹功能的程式中，函式沒有任何副作用，最後傳回的值是函式呼叫的唯一結果。因此，忽略結果時，可能是程式設計錯誤。雖然F#不是純粹的功能性程式設計語言，但最好是盡可能遵循功能性程式設計風格。

請參閱

- [橢圓](#)
- [F# 語言參考](#)

字串

2020/11/2 • [Edit Online](#)

型別會將 `string` 不可變的文字表示為 Unicode 字元序列。 `string` 是 `System.String` 在 .NET 中的別名。

備註

字串常值是以引號 (") 字元分隔。 () 的反斜線字元 \ 用來編碼某些特殊字元。反斜線和下一個字元統稱為「escape」序列。下表顯示 F# 字串常值中支援的 Escape 序列。

說明	Escape 序列
警示	<code>\a</code>
退格鍵	<code>\b</code>
換頁字元	<code>\f</code>
新行字元	<code>\n</code>
歸位字元	<code>\r</code>
索引標籤	<code>\t</code>
垂直 Tab 鍵	<code>\v</code>
反斜線	<code>\\</code>
引號	<code>\"</code>
單引號	<code>\'</code>
Unicode 字元	<code>\DDD</code> (, 其中 <code>D</code> 指出十進位數; 範圍為 000-255; 例如 <code>\231</code> = "ζ")
Unicode 字元	<code>\xHH</code> (, 其中 <code>H</code> 指出十六進位數位; 範圍為 00-FF, 例如 <code>\xE7</code> = "ζ")
Unicode 字元	<code>\uHHHH</code> (UTF-16) (其中 <code>H</code> 指出十六進位數位; 範圍為 0000-FFFF; 例如, <code>\u00E7</code> = "ζ")
Unicode 字元	<code>\U00HHHHHHH</code> (32 UTF-8) (, 其中 <code>H</code> 指出十六進位數位; 範圍為 000000-10FFFF; 例如, <code>\U0001F47D</code> = "🍷")

IMPORTANT

`\DDD` Escape 序列是十進位標記法, 而非八進位標記法, 就像大多數其他語言一樣。因此, 數位 `8` 和 `9` 是有效的, 而序列 `\032` 表示 (U + 0020) 的空格, 而八進位標記法中的相同程式碼點則為 `\040`。

NOTE

受限於 0-255 (0xFF) 的範圍，`\DDD` 和 `\x` escape 序列實際上是 [ISO-8859-1](#) 字元集，因為該字元集符合前 256 Unicode 程式碼點。

逐字字串

如果前面加上 `@` 符號，常值為逐字字串。這表示會忽略任何 escape 序列，但會將兩個引號字元視為一個引號字元。

三個引號的字串

此外，字串可以用三個引號括住。在此情況下，會忽略所有的 escape 序列，包括雙引號字元。若要指定包含內嵌引號字串的字串，您可以使用逐字字串或以三括弧括住的字串。如果您使用逐字字串，您必須指定兩個引號字元，以表示單引號字元。如果您使用以三個引號括住的字串，則可以使用單引號字元，而不將它們剖析為字串的結尾。當您使用 XML 或其他包含內嵌引號的結構時，這項技術會很有用。

```
// Using a verbatim string
let xmlFragment1 = @"<book author=""Milton, John"" title=""Paradise Lost"">"

// Using a triple-quoted string
let xmlFragment2 = """<book author="Milton, John" title="Paradise Lost">"""
```

在程式碼中，接受分行符號的字串會被接受，而且分行符號會以逐字的形式解釋為分行符號，除非反斜線字元是分行符號之前的最後一個字元。使用反斜線字元時，會忽略下一行的開頭空白字元。下列程式碼會產生具有值的字串，`str1` `"abc\ndef"` 以及 `str2` 具有值的字串 `"abcdef"`。

```
let str1 = "abc
def"
let str2 = "abc\
def"
```

字串編制索引和切割

您可以使用類似陣列的語法來存取字串中的個別字元，如下所示。

```
printfn "%c" str1.[1]
```

輸出為 `b`。

或者，您可以使用陣列配量語法來解壓縮子字串，如下列程式碼所示。

```
printfn "%s" (str1.[0..2])
printfn "%s" (str2.[3..5])
```

輸出如下。

```
abc
def
```

您可以使用不帶正負號的位元組陣列來代表 ASCII 字串，請輸入 `byte[]`。您將尾碼新增 `B` 至字串常值，以表示它是 ASCII 字串。搭配位元組陣列使用的 ASCII 字串常值支援與 Unicode 字串相同的 escape 序列，但

Unicode escape 序列除外。

```
// "abc" interpreted as a Unicode string.  
let str1 : string = "abc"  
// "abc" interpreted as an ASCII byte array.  
let bytearray : byte[] = "abc"b
```

字串運算子

+ 操作員可以用來串連字號串，維持與 .NET Framework 字串處理功能的相容性。下列範例說明字串串連。

```
let string1 = "Hello, " + "world"
```

String 類別

因為 F # 中的字串類型實際上是 .NET Framework `System.String` 類型，所以所有 `System.String` 成員都可以使用。這包括 **+** 用來串連字號串的運算子、`Length` 屬性和 `Chars` 屬性 (property)，這會以 Unicode 字元陣列的形式傳回字串。如需字串的詳細資訊，請參閱 `System.String`。

藉由使用的 `Chars` 屬性 `System.String`，您可以藉由指定索引來存取字串中的個別字元，如下列程式碼所示。

```
let printChar (str : string) (index : int) =  
    printfn "First character: %c" (str.Chars(index))
```

字串模組

字串處理的其他功能會包含在 `String` 命名空間的模組中 `FSharp.Core`。如需詳細資訊，請參閱 [字串模組](#)。

另請參閱

- [F # 語言參考](#)

插入字串

2021/3/5 • [Edit Online](#)

內插字串是可讓您在其中內嵌 F # 運算式的 **字串**。它們在各種案例中很有用，因為字串值可能會根據值或運算式的結果而變更。

語法

```
$"string-text {expr}"  
$"string-text %format-specifier{expr}"  
$""string-text {"embedded string literal"}""
```

備註

插補字串可讓您在字串常值內的「漏洞」內撰寫程式碼。以下是基本範例：

```
let name = "Phillip"  
let age = 30  
printfn $"Name: {name}, Age: {age}"  
  
printfn $"I think {3.0 + 0.14} is close to {System.Math.PI}!"
```

每個 `{ }` 大括弧配對之間的內容可以是任何 F # 運算式。

若要對大括弧組進行換用 `{ }`，請撰寫兩個括弧，如下所示：

```
let str = $"A pair of braces: {{}}"  
// "A pair of braces: {}"
```

具類型的插入字串

插入字串也可以有 F # 格式規範來強制執行型別安全。

```
let name = "Phillip"  
let age = 30  
  
printfn $"Name: %s{name}, Age: %d{age}"  
  
// Error: type mismatch  
printfn $"Name: %s{age}, Age: %d{name}"
```

在上述範例中，程式碼會錯誤 `age` 地傳遞值 `name`，其中應該是，反之亦然/反之亦然。因為插入字串使用格式規範，所以這是編譯錯誤，而不是微妙的執行時間 bug。

純文字格式中涵蓋的所有格式規範都是在插入字串內有效。

逐字插補字串

F # 支援以三個引號括住的逐字字串，讓您可以內嵌字串常值。

```
let age = 30

printfn $"" "Name: {\"Phillip\"}, Age: %d{age}""
```

對齊內插字串中的運算式

您可以在插入字串內將運算式靠左對齊或靠右對齊 `|`，以及指定多少空間。下列插入字串會將左邊和右邊的運算式分別靠左和向右對齊七個空格。

```
printfn $""|{\"Left\",-7}|{\"Right\",7}|""
// |Left   |   Right|
```

字串插值和 `FormattableString` 格式

您也可以套用遵守下列規則的格式 [FormattableString](#)：

```
let speedOfLight = 299792.458
printfn $"The speed of light is {speedOfLight:N3} km/s."
// "The speed of light is 299,792.458 km/s."
```

此外，您也可以透過類型注釋，將內插字串的類型檢查成 a [FormattableString](#)：

```
let frmtStr = $"The speed of light is {speedOfLight:N3} km/s." : FormattableString
// Type: FormattableString
// The speed of light is 299,792.458 km/s.
```

請注意，類型注釋必須在插入字串運算式本身。F# 不會以隱含方式將內插字串轉換成 [FormattableString](#)。

另請參閱

- [字串](#)

Tuple

2020/11/2 • [Edit Online](#)

元組是未命名但排序值的群組，可能是不同的類型。元組可以是參考型別或結構。

語法

```
(element, ... , element)
struct(element, ... ,element )
```

備註

上述語法中的每個 元素 都可以是任何有效的 F# 運算式。

範例

元組的範例包括相同或不同類型的配對、三合一等等。下列程式碼說明一些範例。

```
(1, 2)

// Triple of strings.
("one", "two", "three")

// Tuple of generic types.
(a, b)

// Tuple that has mixed types.
("one", 1, 2.0)

// Tuple of integer expressions.
(a + 1, b + 1)

// Struct Tuple of floats
struct (1.025f, 1.5f)
```

取得個別值

您可以使用模式比對來存取和指派元組元素的名稱，如下列程式碼所示。

```
let print tuple1 =
    match tuple1 with
    | (a, b) -> printfn "Pair %A %A" a b
```

您也可以透過系統，透過運算式外部的模式比對來解構元組 `match` `let`：

```
let (a, b) = (1, 2)

// Or as a struct
let struct (c, d) = struct (1, 2)
```

或者，您可以對元組進行模式比對，作為函式的輸入：

```
let getDistance ((x1,y1): float*float) ((x2,y2): float*float) =  
    // Note the ability to work on individual elements  
    (x1*x2 - y1*y2)  
    |> abs  
    |> sqrt
```

如果您只需要一個元組的元素，則可以使用 (底線) 的萬用字元，以避免為不需要的值建立新的名稱。

```
let (a, _) = (1, 2)
```

從參考元組將元素複製到結構元組也很簡單：

```
// Create a reference tuple  
let (a, b) = (1, 2)  
  
// Construct a struct tuple from it  
let struct (c, d) = struct (a, b)
```

`fst` `snd` 只有) 傳回元組的第一個和第二個元素時，才會有函數和 (參考元組)。

```
let c = fst (1, 2)  
let d = snd (1, 2)
```

沒有可傳回三個三個元素的內建函數，但您可以輕鬆地撰寫一個，如下所示。

```
let third (_, _, c) = c
```

一般而言，最好使用模式比對來存取個別的元組元素。

使用元組

元組提供一個便利的方式，從函式傳回多個值，如下列範例所示。此範例會執行整數除法運算，並將作業的舍入結果傳回為元組配對的第一個成員，並傳回餘數做為配對的第二個成員。

```
let divRem a b =  
    let x = a / b  
    let y = a % b  
    (x, y)
```

當您想要避免一般函數語法所隱含的函式引數隱含 currying 時，元組也可以當做函式引數使用。

```
let sumNoCurry (a, b) = a + b
```

定義函式的一般語法可 `let sum a b = a + b` 讓您定義函式，該函式是函式之第一個引數的部分應用，如下列程式碼所示。

```
let sum a b = a + b  
  
let addTen = sum 10  
let result = addTen 95  
// Result is 105.
```

使用元組作為參數會停用 currying。如需詳細資訊，請參閱[函式中的「部分引數應用程式式」](#)。

元組類型的名稱

當您寫出屬於元組的型別名稱時，會使用 `*` 符號來分隔專案。若為包含 `int`、`float` 和 `string` 的元組 (例如 `(10, 10.0, "ten")`)，則會以下列方式寫入類型。

```
int * float * string
```

與 c # 元組交互操作

C # 7.0 引進了元組至語言。C # 中的元組是結構，相當於 F # 中的結構元組。如果您需要與 c # 交互操作，您必須使用結構元組。

這麼做很簡單。例如，假設您必須將元組傳遞給 c # 類別，然後使用它的結果，也就是元組：

```
namespace CSharpTupleInterop
{
    public static class Example
    {
        public static (int, int) AddOneToXAndY((int x, int y) a) =>
            (a.x + 1, a.y + 1);
    }
}
```

在您的 F # 程式碼中，您可以傳遞結構元組作為參數，並將結果作為結構元組使用。

```
open TupleInterop

let struct (newX, newY) = Example.AddOneToXAndY(struct (1, 2))
// newX is now 2, and newY is now 3
```

在參考元組和結構元組之間轉換

由於參考元組和結構元組具有完全不同的基礎標記法，因此它們無法隱含轉換。也就是說，如下所示的程式碼將不會編譯：

```
// Will not compile!
let (a, b) = struct (1, 2)

// Will not compile!
let struct (c, d) = (1, 2)

// Won't compile!
let f(t: struct(int*int)): int*int = t
```

您必須在一個元組上進行模式比對，並使用組成零件來建立另一個。例如：

```
// Pattern match on the result.
let (a, b) = (1, 2)

// Construct a new tuple from the parts you pattern matched on.
let struct (c, d) = struct (a, b)
```

參考元組的編譯形式

本節說明編譯元組的形式。除非您的目標是 .NET Framework 3.5 或更低版本，否則不需要讀取此處的資訊。

元組會編譯成數個泛型型別之一的物件，這些泛型型別全都命名為 arity 上的多載，`System.Tuple` 或類型參數的數目。當您從另一種語言（例如 c # 或 Visual Basic，或使用不知道 F # 結構的工具）來查看元組類型時，元組類型會以此形式出現。這些 `Tuple` 類型是在 .NET Framework 4 中引進。如果您的目標是舊版 .NET Framework，編譯器會使用 `System.Tuple` 來自 2.0 版 F # 核心程式庫的版本。此程式庫中的類型僅適用於以 .NET Framework 2.0、3.0 和 3.5 版本為目標的應用程式。類型轉送是用來確保 .NET Framework 2.0 和 .NET Framework 4 F # 元件之間的二進位相容性。

結構元組的編譯形式

結構元組（例如 `struct (x, y)`），基本上與參考元組不同。它們會編譯成 `ValueTuple` 類型、以 arity 來多載，或是類型參數的數目。它們相當於 [c # 7.0 元組](#) 和 [Visual Basic 2017 元組](#)，而且可交互操作雙向。

另請參閱

- [F # 語言參考](#)
- [F# 類型](#)

F # 集合類型

2020/11/2 • [Edit Online](#)

藉由查看本主題，您可以判斷哪一個 F # 集合類型最適合特定的需求。這些集合類型與 .NET 中的集合類型 (例如命名空間中的型別不同) 不同之處在於，`System.Collections.Generic` F # 集合類型是以功能性程式設計的觀點來設計，而不是以物件導向的觀點來設計。更具體來說，只有陣列集合具有可變動的元素。因此，當您修改集合時，會建立已修改之集合的實例，而不是改變原創組合。

集合類型也會隨著儲存物件的資料結構類型而有所不同。雜湊資料表、連結清單和陣列等資料結構具有不同的效能特性，以及一組不同的可用作業。

集合類型的資料表

下表顯示 F # 集合類型。

「	「	「「
清單	相同類型的已排序且不可變的元素系列。實作為連結清單。	清單 List 模組
陣列	以零為基底的固定資料元素集合，這些專案全都屬於相同類型。	陣列 Array 模組 Array2D 模組 Array3D 模組
序列	全部為一種類型之元素的邏輯序列。當您有大量的資料收集，但不一定會預期使用所有專案時，序列特別有用。個別順序元素只會在必要時計算，因此如果未使用所有元素，序列的執行效能會優於清單。順序是以類型表示 <code>seq<'T></code> ，這是的別名 <code>IEnumerable<T></code> 。因此，任何執行的 .NET Framework 型別都 <code>System.Collections.Generic.IEnumerable<'T></code> 可以當做序列使用。	序列 Seq 模組
地圖	元素的不可變字典。專案是由索引鍵存取。	Map 模組
設定	以二進位樹狀結構為基礎的不可變集合，其中的比較是 F # 結構比較函式，其可能會 <code>System.IComparable</code> 在索引鍵值上使用介面的實值。	設定模組

函數的資料表

本節將比較 F # 集合類型上可用的函式。會提供函數的計算複雜度，其中 N 是第一個集合的大小，而 M 是第二個集合的大小 (如果有的話)。虛線 (-) 表示此函式無法在集合上使用。因為序列會延遲評估，所以函式 (例如) `Seq.distinct` 可能會是 O (1) 因為它會立即傳回，但它仍會在列舉時影響順序的效能。

「	ARRAY	「	「	「	「	「
附加	O (N)	O (N)	O (N)	-	-	傳回新的集合，其中包含第一個集合的元素，後面接著第二個集合的元素。
add	-	-	-	O (記錄 (N) # A3	O (記錄 (N) # A3	傳回已加入專案的新集合。

⌈	ARRAY	⌈	⌈	⌈	⌈	⌈
average	O (N)	O (N)	O (N)	-	-	傳回集合中元素的平均值。
averageBy	O (N)	O (N)	O (N)	-	-	傳回已套用至每個元素之所提供函式的結果平均值。
array.blit	O (N)	-	-	-	-	複製陣列的區段。
快取	-	-	O (N)	-	-	計算和儲存序列的元素。
cast	-	-	O (N)	-	-	將元素轉換成指定的類型。
choose	O (N)	O (N)	O (N)	-	-	將指定的函式套用 f 至清單的每個元素 x。傳回清單, 其中包含函式傳回之每個元素的結果 Some (f (x))。
收集	O (N)	O (N)	O (N)	-	-	將指定的函式套用至集合中的每個元素、串連所有結果, 並傳回合並的清單。
Seq.compare with	-	-	O (N)	-	-	使用指定的比較函式、element by 專案來比較兩個序列。
concat	O (N)	O (N)	O (N)	-	-	將指定列舉列舉型別合併為單一串連列舉。
contains	-	-	-	-	O (記錄 (N) # A3	如果集合包含指定的元素, 則傳回 true。
containsKey	-	-	-	O (記錄 (N) # A3	-	測試專案是否在對應的定義域中。
count	-	-	-	-	O (N)	傳回集合中項目的數目。
Seq.countby	-	-	O (N)	-	-	將索引鍵產生函式套用至序列的每個專案, 並傳回會產生唯一索引鍵的序列, 以及其在原始序列中的發生次數。
copy	O (N)	-	O (N)	-	-	複製集合。

⌈	ARRAY	⌈	⌈	⌈	⌈	⌈
建立	O (N)	-	-	-	-	建立整個專案的陣列, 這些元素最初都是指定的值。
delay	-	-	O (1)	-	-	傳回從指定之序列的延遲規格建立的序列。
差異	-	-	-	-	O (M * log (N) # A3	傳回新的集合, 其中包含從第一個集合中移除之第二個集合的元素。
distinct			O (1) *			根據專案的泛型雜湊和相等比較, 傳回不含重複專案的序列。如果專案在順序中多次出現, 則會捨棄之後的發生次數。
Seq.distinctby			O (1) *			根據給定的索引鍵產生函數所傳回之索引鍵的泛型雜湊和相等比較, 傳回不含重複專案的序列。如果專案在順序中多次出現, 則會捨棄之後的發生次數。
empty	O (1)	O (1)	O (1)	O (1)	O (1)	建立空集合。
exists	O (N)	O (N)	O (N)	O (記錄 (N) # A3	O (記錄 (N) # A3	測試序列中的任何元素是否符合指定的述詞。
array.exists2	O (min (N, M) # A3	-	O (min (N, M) # A3			測試輸入序列中是否有任何一對對應的元素符合指定的述詞。
fill	O (N)					將陣列的元素範圍設定為指定的值。
filter	O (N)	O (N)	O (N)	O (N)	O (N)	傳回新的集合, 其中只包含指定述詞所傳回之集合的元素 <code>true</code> 。
尋找	O (N)	O (N)	O (N)	O (記錄 (N) # A3	-	傳回指定的函式傳回的第一個元素 <code>true</code> 。 <code>System.Collections.Generic.KeyNotF</code> 如果沒有這類元素, 則傳回。

⌈	ARRAY	⌈	⌈	⌈	⌈	⌈
findIndex	O (N)	O (N)	O (N)	-	-	傳回陣列中滿足給定述詞的第一個元素的索引。如果沒有任何專案滿足述詞, 則會引發 <code>System.Collections.Generic.KeyNotFoundException</code> 。
Map.findkey	-	-	-	O (記錄 (N) # A3	-	評估集合中每個對應的函式, 並傳回函式傳回之第一個對應的索引鍵 <code>true</code> 。如果沒有這類專案存在, 則會引發此函式 <code>System.Collections.Generic.KeyNotFoundException</code> 。
摺疊	O (N)	O (N)	O (N)	O (N)	O (N)	將函式套用至集合中的每個元素, 並透過計算將累計引數串接。如果輸入函式為 f, 且元素為 i0。在中, 此函式會計算 f (... (f s i0) ...)。
list.fold2	O (N)	O (N)	-	-	-	將函數套用至兩個集合的對應元素, 並透過計算將累計引數串接。集合的大小必須相同。如果輸入函式為 f, 且元素為 i0。iN 和 j0。jN, 此函式會計算 f (... (f s i0 j0) ...) jN 中。
List.foldback	O (N)	O (N)	-	O (N)	O (N)	將函式套用至集合中的每個元素, 並透過計算將累計引數串接。如果輸入函式為 f, 且元素為 i0。在中, 此函式會在 s) # A3 中計算 f i0 (... (f。

⌊	ARRAY	⌊	⌊	⌊	⌊	⌊
List.foldback2	O (N)	O (N)	-	-	-	將函數套用至兩個集合的對應元素, 並透過計算將累計引數串接。集合的大小必須相同。如果輸入函式為 f, 且元素為 i0 .。 . iN 和 j0 .。 . jN, 此函式會計算 jN s) # A3 中的 f i0 j0 (... (f。
forall	O (N)	O (N)	O (N)	O (N)	O (N)	測試集合的所有元素是否滿足指定的述詞。
array.forall2	O (N)	O (N)	O (N)	-	-	測試集合中所有對應的元素是否符合給定的述詞成對。
get/n	O (1)	O (N)	O (N)	-	-	根據指定的索引, 從集合傳回專案。
head	-	O (1)	O (1)	-	-	傳回集合的第一個元素。
init	O (N)	O (N)	O (1)	-	-	建立集合, 並指定維度和產生器函式來計算元素。
Seq.initinfinite	-	-	O (1)	-	-	產生順序, 在反復查看時, 會藉由呼叫指定的函式傳回連續的元素。
intersect	-	-	-	-	O (記錄 (N) * 記錄 (M) # A5	計算兩個集合的交集。
Set.intersectmany	-	-	-	-	O (N1 * N2 ...)	計算一連串集合的交集。順序不得為空白。
isEmpty	O (1)	O (1)	O (1)	O (1)	-	<code>true</code> 如果集合是空的, 則傳回。
Set.isprosubset	-	-	-	-	O (M * log (N) # A3	<code>true</code> 如果第一個集合的所有專案都在第二個集合中, 而且第二個集合中至少有一個元素不在第一個集合中, 則傳回。

⌘	ARRAY	⌘	⌘	⌘	⌘	⌘
Set.isproper superset	-	-	-	-	$O(M * \log(N) \cdot A^3)$	<code>true</code> 如果第二個集合的所有元素都在第一個集合中，而且第二個集合中至少有一個元素不在第二個集合中，則傳回。
Set.issubset	-	-	-	-	$O(M * \log(N) \cdot A^3)$	<code>true</code> 如果第一個集合的所有元素都在第二個集合中，則傳回。
Set.issuperset	-	-	-	-	$O(M * \log(N) \cdot A^3)$	<code>true</code> 如果第二個集合的所有元素都在第一個集合中，則傳回。
Iter	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	將指定的函式套用至集合的每個元素。
list.iteri	$O(N)$	$O(N)$	$O(N)$	-	-	將指定的函式套用至集合的每個元素。傳遞至函式的整數指出元素的索引。
array.iteri2	$O(N)$	$O(N)$	-	-	-	將指定的函式套用至一對從兩個數組中的相符索引所繪製的元素。傳遞至函式的整數指出元素的索引。這兩個數組的長度必須相同。
list.iter2	$O(N)$	$O(N)$	$O(N)$	-	-	將指定的函式套用至一對從兩個數組中的相符索引所繪製的元素。這兩個數組的長度必須相同。
last	$O(1)$	$O(N)$	$O(N)$	-	-	傳回適用集合中的最後一個專案。
長度	$O(1)$	$O(N)$	$O(N)$	-	-	傳回集合中的元素數目。
map	$O(N)$	$O(N)$	$O(1)$	-	-	建立集合，其專案為將指定函式套用至陣列的每個元素的結果。

⌈	ARRAY	⌈	⌈	⌈	⌈	⌈
list.map2	O (N)	O (N)	O (1)	-	-	建立集合, 其專案是將指定的函式套用至兩個集合之對應元素的結果。這兩個輸入陣列的長度必須相同。
list.map3	-	O (N)	-	-	-	建立集合, 其專案是將指定函式同時套用至三個集合之對應元素的結果。
Mapi	O (N)	O (N)	O (N)	-	-	建立陣列, 其專案為將指定函式套用至陣列的每個元素的結果。傳遞至函式的整數索引, 表示要轉換之元素的索引。
list.mapi2	O (N)	O (N)	-	-	-	建立集合, 其專案為將指定函式套用至兩個集合的對應元素的結果, 也會傳遞元素的索引。這兩個輸入陣列的長度必須相同。
max	O (N)	O (N)	O (N)	-	-	傳回集合中最大的專案, 並使用 <code>max</code> 運算子進行比較。
maxBy	O (N)	O (N)	O (N)	-	-	傳回集合中最大的專案, 相較于在函數結果上使用 <code>max</code> 。
Set.maxelement	-	-	-	-	O (記錄 (N) # A3)	根據用於集合的順序, 傳回集合中最大的元素。
分鐘	O (N)	O (N)	O (N)	-	-	傳回集合中最小的專案, 使用 <code>min</code> 運算子進行比較。
minBy	O (N)	O (N)	O (N)	-	-	傳回集合中最小的專案, 並使用函數結果上的 <code>min</code> 運算子來比較。
Set.minelement	-	-	-	-	O (記錄 (N) # A3)	根據用於集合的順序, 傳回集合中的最小元素。

☐	ARRAY	☐	☐	☐	☐	☐
List.ofarray	-	O (N)	O (1)	O (N)	O (N)	建立集合, 其中包含與指定陣列相同的元素。
ofList	O (N)	-	O (1)	O (N)	O (N)	建立集合, 其包含與指定清單相同的元素。
List.ofseq	O (N)	O (N)	-	O (N)	O (N)	建立集合, 其包含與指定序列相同的元素。
派	-	-	O (N)	-	-	傳回輸入序列中每個專案及其前置專案的序列, 但第一個專案除外, 後者只會傳回做為第二個元素的前置專案。
partition	O (N)	O (N)	-	O (N)	O (N)	將集合分割成兩個集合。第一個集合包含指定述詞傳回的專案 <code>true</code> , 而第二個集合則包含指定述詞傳回的元素 <code>false</code> 。
permute	O (N)	O (N)	-	-	-	傳回陣列, 其中包含根據指定排列排列的所有元素。
選擇	O (N)	O (N)	O (N)	O (記錄 (N) # A3	-	將指定的函式套用到後續的元素, 並傳回函式傳回部分的第一個結果。如果函式永遠不會傳回部分, <code>System.Collections.Generic.KeyNotFoundException</code> 就會引發。
readonly	-	-	O (N)	-	-	建立可委派給指定順序物件的順序物件。這種作業可確保類型轉換無法重新探索並改變原始序列。例如, 如果指定了陣列, 傳回的序列將會傳回陣列的元素, 但您無法將傳回的序列物件轉換成陣列。

⌘	ARRAY	⌘	⌘	⌘	⌘	⌘
reduce	$O(N)$	$O(N)$	$O(N)$	-	-	將函式套用至集合中的每個元素, 並透過計算將累計引數串接。此函式一開始會將函式套用至前兩個元素, 將此結果連同第三個元素一起傳遞至函式, 依此類推。函數會傳回最終結果。
Array.reduceback	$O(N)$	$O(N)$	-	-	-	將函式套用至集合中的每個元素, 並透過計算將累計引數串接。如果輸入函式為 f , 且元素為 $i0$ 。在中, 此函式會計算) # $A3$ 中-1 的 f $i0$ (... (f 。
remove	-	-	-	$O(\text{記錄}(N) \# A3)$	$O(\text{記錄}(N) \# A3)$	從對應的定義域中移除專案。如果元素不存在, 則不會引發任何例外狀況。
複製	-	$O(N)$	-	-	-	建立指定長度的清單, 並將每個元素設定為指定的值。
轉速	$O(N)$	$O(N)$	-	-	-	以反向順序傳回包含元素的新清單。
掃描	$O(N)$	$O(N)$	$O(N)$	-	-	將函式套用至集合中的每個元素, 並透過計算將累計引數串接。這項作業會將函式套用至第二個引數和清單的第一個元素。接著, 作業會將此結果連同第二個元素一起傳遞到函式中, 依此類推。最後, 此作業會傳回中繼結果清單和最終結果。
Array.scanback	$O(N)$	$O(N)$	-	-	-	類似于 List.foldback 作業, 但會傳回中繼和最終結果。
singleton	-	-	$O(1)$	-	$O(1)$	傳回只產生一個專案的序列。

⌈	ARRAY	⌈	⌈	⌈	⌈	⌈
set	O (1)	-	-	-	-	將陣列的元素設定為指定的值。
skip	-	-	O (N)	-	-	傳回序列, 這個序列會略過基礎序列的 N 個元素, 然後產生序列的其餘專案。
skipWhile	-	-	O (N)	-	-	傳回序列, 此序列會在指定的述詞傳回時略過基礎序列的元素, <code>true</code> 然後產生序列的其餘專案。
sort	O (N * 記錄 (n) # A3 平均) O (N ^ 2) 最糟的情況	O (N * 記錄 (n) # A3)	O (N * 記錄 (n) # A3)	-	-	依元素值排序集合。使用 compare 來比較元素。
sortBy	O (N * 記錄 (n) # A3 平均) O (N ^ 2) 最糟的情況	O (N * 記錄 (n) # A3)	O (N * 記錄 (n) # A3)	-	-	使用指定投射提供的索引鍵來排序指定的清單。使用 compare 來比較索引鍵。
Array.sortinplace	O (N * 記錄 (n) # A3 平均) O (N ^ 2) 最糟的情況	-	-	-	-	藉由就地變更並使用指定的比較函式, 來排序陣列的元素。使用 compare 來比較元素。
Array.sortinplaceby	O (N * 記錄 (n) # A3 平均) O (N ^ 2) 最糟的情況	-	-	-	-	藉由就地變更並使用指定的索引鍵投射, 來排序陣列的元素。使用 compare 來比較元素。
Array.sortinplacewith	O (N * 記錄 (n) # A3 平均) O (N ^ 2) 最糟的情況	-	-	-	-	使用指定的比較函式來排序陣列的元素, 並使用指定的比較函數作為順序。
List.sortwith	O (N * 記錄 (n) # A3 平均) O (N ^ 2) 最糟的情況	O (N * 記錄 (n) # A3)	-	-	-	使用指定的比較函式作為順序, 並傳回新的集合, 以排序集合的元素。
sub	O (N)	-	-	-	-	建立陣列, 其中包含由起始索引和長度指定的指定子範圍。
Sum	O (N)	O (N)	O (N)	-	-	傳回集合中元素的總和。

⌈	ARRAY	⌈	⌈	⌈	⌈	⌈
sumBy	O (N)	O (N)	O (N)	-	-	傳回將函數套用至集合的每個專案所產生的結果總和。
尾巴	-	O (1)	-	-	-	傳回不含第一個元素的清單。
take	-	-	O (N)	-	-	傳回序列的元素, 直到指定的計數為止。
takeWhile	-	-	O (1)	-	-	傳回序列, 此序列會在指定的述詞傳回時產生基礎序列的元素, <code>true</code> 然後不會傳回更多元素。
toArray	-	O (N)	O (N)	O (N)	O (N)	從指定的集合建立陣列。
toList	O (N)	-	O (N)	O (N)	O (N)	從指定的集合建立清單。
List.toSeq	O (1)	O (1)	-	O (1)	O (1)	從指定的集合建立序列。
truncate	-	-	O (1)	-	-	傳回序列, 這個序列會在列舉時傳回不超過 N 個元素。
tryFind	O (N)	O (N)	O (N)	O (記錄 (N) # A3	-	搜尋符合指定之述詞的元素。
Array.tryFindIndex	O (N)	O (N)	O (N)	-	-	搜尋符合指定述詞的第一個元素, 並傳回相符專案的索引, <code>None</code> 如果沒有這類專案, 則為。
Map.tryFindKey	-	-	-	O (記錄 (N) # A3	-	傳回集合中符合指定述詞之第一個對應的索引鍵, <code>None</code> 如果沒有這類專案存在, 則傳回。
Array.tryPick	O (N)	O (N)	O (N)	O (記錄 (N) # A3	-	將指定的函式套用到連續的元素, 並傳回函式針對某個值傳回的第一個結果 <code>Some</code> 。如果沒有這類元素存在, 則作業會傳回 <code>None</code> 。
展開	-	-	O (N)	-	-	傳回包含指定計算所產生之元素的序列。

「	ARRAY	「	「	「	「	「
union	-	-	-	-	$O(M * \log(N) \# A3)$	計算兩個集合的聯集。
Set.unionmany	-	-	-	-	$O(N1 * N2 \dots)$	計算一系列集合的聯集。
unzip	$O(N)$	$O(N)$	$O(N)$	-	-	將配對清單分割成兩個清單。
list.unzip3	$O(N)$	$O(N)$	$O(N)$	-	-	將三合一清單分割成三個清單。
視窗	-	-	$O(N)$	-	-	傳回序列, 這個序列會產生包含從輸入序列中繪製之元素的滑動視窗。每個視窗都會以全新的陣列傳回。
zip	$O(N)$	$O(N)$	$O(N)$	-	-	將兩個集合合併成成對的清單。這兩個清單必須有相等的長度。
array.zip3	$O(N)$	$O(N)$	$O(N)$	-	-	將三個集合合併成三個清單。清單必須有相等的長度。

另請參閱

- [F# 類型](#)
- [F # 語言參考](#)

清單

2020/11/2 • [Edit Online](#)

在 F# 中, list 是包含一系列經過排序且類型相同的固定元素。若要對清單執行基本作業, 請使用[清單模組](#)中的函數。

NOTE

F# 的 docs.microsoft.com API 參考不完整。如果您遇到任何中斷的連結, 請改為參考[F# 核心程式庫檔](#)。

建立及初始化 list

您可以藉由明確列出元素 (以逗號分隔並括以方括號) 來定義 list, 如下列程式碼所示。

```
let list123 = [ 1; 2; 3 ]
```

您也可以元素之間加入分行符號; 若您選擇加入分行符號, 就不一定需要使用分號。當元素初始化運算式比較長, 或當您想要為每個元素加入逗號時, 以後一種語法編寫的程式碼會比較容易閱讀。

```
let list123 = [  
    1  
    2  
    3 ]
```

通常 list 的所有元素都應必須是同一類型。但當 list 指定的元素具有衍生類型的基底類型時, 就不在此限。下列的 `Button` 與 `CheckBox` 因為都是從 `Control` 衍生而來, 所以可接受。

```
let myControlList : Control list = [ new Button(); new CheckBox() ]
```

如下列程式碼所示, 您也可以使用以整數指定的範圍 (以範圍運算子 `..` 分隔) 來定義 list 元素。

```
let list1 = [ 1 .. 10 ]
```

空 list 由不含任何內容的一對方括號指定。

```
// An empty list.  
let listEmpty = []
```

您也可以使用順序運算式建立 list。如需詳細資訊, 請參閱[序列運算式](#)。例如下列程式碼會建立從 1 到 10 之整數平方的 list。

```
let listOfSquares = [ for i in 1 .. 10 -> i*i ]
```

使用 list 的運算子

您可以使用 `::` (cons) 運算子。若 `list1` 為 `[2; 3; 4]`, 下列程式碼將 `list2` 建立為 `[100; 2; 3; 4]`。

```
let list2 = 100 :: list1
```

您可以使用 `@` 運算子串流具有相容類型的 list，如下列程式碼所示。當 `list1` 為 `[2; 3; 4]` 及 `list2` 為 `[100; 2; 3; 4]` 時，此程式碼會將 `list3` 建立為 `[2; 3; 4; 100; 2; 3; 4]`。

```
let list3 = list1 @ list2
```

[清單模組](#)中提供了在清單上執行作業的功能。

由於 F# 中的 list 為固定，因此任何修改作業都會產生新的 list，而不會修改現有的 list。

F# 中的清單會實作為單向連結清單，這表示只存取清單標頭的作業是 $O(1)$ ，而元素存取是 $O(n)$ 。

屬性

list 類型支援下列屬性：

名稱	類型	說明
前端	<code>'T</code>	第一個元素。
空白	<code>'T list</code>	此為靜態屬性，會傳回適當類型的空 list。
IsEmpty	<code>bool</code>	<code>true</code> 表示 list 不含任何元素。
Item	<code>'T</code>	使用位於指定索引的元素 (以零為基底)。
長度	<code>int</code>	項目的數目。
Tail	<code>'T list</code>	list 沒有第一個元素。

下列是使用這些屬性的一些範例。

```
let list1 = [ 1; 2; 3 ]

// Properties
printfn "list1.IsEmpty is %b" (list1.IsEmpty)
printfn "list1.Length is %d" (list1.Length)
printfn "list1.Head is %d" (list1.Head)
printfn "list1.Tail.Head is %d" (list1.Tail.Head)
printfn "list1.Tail.Tail.Head is %d" (list1.Tail.Tail.Head)
printfn "list1.Item(1) is %d" (list1.Item(1))
```

使用 list

使用 list 編寫程式讓您只需要編寫小量的程式碼，就可以執行複雜的運算。本節說明一些使用函式編寫程式時不可或缺的常用 list 運算。

list 的遞迴功能

list 是唯一適合遞迴程式設計技巧的函式。假設有一項作業必須對 list 的每個元素執行。您可以利用遞迴方式執行此作業，方法是先對 list 的 head 執行運算，再傳遞 list 的 tail (此 list 只含原始 list，而不含第一個元素，所以比較小)，然後再執行下一個遞迴層級的程式碼。

若要編寫這類遞迴函式，可以在模式比對中使用 `cons` 運算子 (`::`)，以區分 `list` 的 `head` 與 `tail`。

下列程式碼範例示範如何使用模式比對，實作執行 `list` 運算的遞迴函式。

```
let rec sum list =
  match list with
  | head :: tail -> head + sum tail
  | [] -> 0
```

前述程式碼對於小型 `list` 的效果很好，但對於大型的 `list`，可能會發生堆疊溢位。下列程式碼是前述程式碼的改良版，運用了遞迴函式標準技巧中的 `accumulator` 引數。使用 `accumulator` 引數可以遞迴函式的 `tail`，進而達到節省空間的目的。

```
let sum list =
  let rec loop list acc =
    match list with
    | head :: tail -> loop tail (acc + head)
    | [] -> acc
  loop list 0
```

函式 `RemoveAllMultiples` 為遞迴函式，可接受兩個 `list`。第一個 `list` 包含倍數要移除的數字；第二個 `list` 是數字要移除的 `list`。下列範例會使用此遞迴函式消除 `list` 中的所有非質數，而只保留質數。

```
let IsPrimeMultipleTest n x =
  x = n || x % n <> 0

let rec RemoveAllMultiples listn listx =
  match listn with
  | head :: tail -> RemoveAllMultiples tail (List.filter (IsPrimeMultipleTest head) listx)
  | [] -> listx

let GetPrimesUpTo n =
  let max = int (sqrt (float n))
  RemoveAllMultiples [ 2 .. max ] [ 1 .. n ]

printfn "Primes Up To %d:\n %A" 100 (GetPrimesUpTo 100)
```

輸出如下所示：

```
Primes Up To 100:
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71; 73; 79; 83; 89; 97]
```

模組函式

`List` 模組提供可存取清單元素的函式。`head` 元素是最方便存取的元素。使用屬性 `head` 或模組函數 `head`。您可以使用 `tail` 屬性或 `list.tail` 函數來存取清單的結尾。若要依索引尋找元素，請使用 `List.nth` 函數。`List.nth` 會周遊 `list`。因此，它是 $O(n)$ 。若您的程式碼需要頻繁地使用 `List.nth`，您或可改用 `array`，而不要使用 `list`。`array` 中的元素存取為 $O(1)$ 。

List 的布林運算

`IsEmpty` 函數會判斷清單是否有任何元素。

清單 `exists` 函式會將布林測試套用至清單的元素，並 `true` 在任何專案符合測試時傳回。`List.exists2` 類似，但會在兩個清單中的後續元素配對上運作。

下列程式碼示範 `List.exists` 的用法。

```
// Use List.exists to determine whether there is an element of a list satisfies a given Boolean expression.
// containsNumber returns true if any of the elements of the supplied list match
// the supplied number.
let containsNumber number list = List.exists (fun elem -> elem = number) list
let list0to3 = [0 .. 3]
printfn "For list %A, contains zero is %b" list0to3 (containsNumber 0 list0to3)
```

輸出如下所示：

```
For list [0; 1; 2; 3], contains zero is true
```

下列範例示範 `List.exists2` 的用法。

```
// Use List.exists2 to compare elements in two lists.
// isEqualElement returns true if any elements at the same position in two supplied
// lists match.
let isEqualElement list1 list2 = List.exists2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
let list1to5 = [ 1 .. 5 ]
let list5to1 = [ 5 .. -1 .. 1 ]
if (isEqualElement list1to5 list5to1) then
    printfn "Lists %A and %A have at least one equal element at the same position." list1to5 list5to1
else
    printfn "Lists %A and %A do not have an equal element at the same position." list1to5 list5to1
```

輸出如下所示：

```
Lists [1; 2; 3; 4; 5] and [5; 4; 3; 2; 1] have at least one equal element at the same position.
```

如果您想要測試清單中的所有元素是否符合條件，您可以使用 `forall`。

```
let isAllZeroes list = List.forall (fun elem -> elem = 0.0) list
printfn "%b" (isAllZeroes [0.0; 0.0])
printfn "%b" (isAllZeroes [0.0; 1.0])
```

輸出如下所示：

```
true
false
```

同樣地，`array.forall2` 會判斷兩個清單中對應位置的所有元素是否都符合包含每一對元素的布林運算式。

```
let listEqual list1 list2 = List.forall2 (fun elem1 elem2 -> elem1 = elem2) list1 list2
printfn "%b" (listEqual [0; 1; 2] [0; 1; 2])
printfn "%b" (listEqual [0; 0; 0] [0; 1; 0])
```

輸出如下所示：

```
true
false
```

List 的排序運算

`List.sort`、`sortBy` 和 `list.sortwith` 函數排序清單。排序函式可指定要使用這三個函式中的哪一個函式。`List.sort` 會使用預設的泛型比較。泛型比較會依不同的泛型比較函式而使用不同的全域運算子來比較值。其可與多種元

素類型搭配使用，例如簡單的數值類型、tuple、記錄、差別聯集、list、array 及所有實作 `System.IComparable` 的類型。對於實作 `System.IComparable` 的類型，泛型比較會使用 `System.IComparable.CompareTo()` 函式。泛型比較也會搭配字串，但會使用不涉及文化的排序順序。請勿對不支援的類型 (例如函式類型) 使用泛型比較。此外，預設泛型比較對小型結構類型的效能最好；若需要經常比較及排序比較大的結構類型，可考慮實作 `System.IComparable` 及佈建成效更好的 `System.IComparable.CompareTo()` 方法實作。

`List.sortBy` 可接受函式傳回的值做為排序準則；而 `List.sortWith` 可接受比較函式做為引數。當您要使用的類型不支援比較時，或比較需要更複雜的比較語意 (例如使用涉及文化的字串) 時，即可使用後兩個函式。

下列範例示範 `List.sort` 的用法。

```
let sortedList1 = List.sort [1; 4; 8; -2; 5]
printfn "%A" sortedList1
```

輸出如下所示：

```
[-2; 1; 4; 5; 8]
```

下列範例示範 `List.sortBy` 的用法。

```
let sortedList2 = List.sortBy (fun elem -> abs elem) [1; 4; 8; -2; 5]
printfn "%A" sortedList2
```

輸出如下所示：

```
[1; -2; 4; 5; 8]
```

下列範例示範 `List.sortWith` 的用法。此範例會使用自訂比較函式 `compareWidgets` 先比較自訂類型的第一個欄位，當第一個欄位的值相等時，再比較其他欄位。

```
type Widget = { ID: int; Rev: int }

let compareWidgets widget1 widget2 =
    if widget1.ID < widget2.ID then -1 else
    if widget1.ID > widget2.ID then 1 else
    if widget1.Rev < widget2.Rev then -1 else
    if widget1.Rev > widget2.Rev then 1 else
    0

let listToCompare = [
    { ID = 92; Rev = 1 }
    { ID = 110; Rev = 1 }
    { ID = 100; Rev = 5 }
    { ID = 100; Rev = 2 }
    { ID = 92; Rev = 1 }
]

let sortedWidgetList = List.sortWith compareWidgets listToCompare
printfn "%A" sortedWidgetList
```

輸出如下所示：


```
[{ID = 92;
  Rev = 1;}; {ID = 92;
  Rev = 1;}; {ID = 100;
  Rev = 2;}; {ID = 100;
  Rev = 5;}; {ID = 110;
  Rev = 1;}]
```

List 的搜尋運算

list 支援多種搜尋運算。最簡單的[清單](#)，可讓您尋找符合指定條件的第一個元素。

下列程式碼範例示範如何使用 `List.find` 從 list 中尋找第一個可以被 5 整除的數字。

```
let isDivisibleBy number elem = elem % number = 0
let result = List.find (isDivisibleBy 5) [ 1 .. 100 ]
printfn "%d " result
```

The result is 5.

如果必須先轉換專案，請呼叫[List](#)，它會採用會傳回選項的函式，並尋找第一個選項值，也就是 `Some(x)`。

`List.pick` 不會傳回元素，而會傳回結果 `x`。若找不到相符的元素，`List.pick` 會擲出 `System.Collections.Generic.KeyNotFoundException`。下列程式碼示範 `List.pick` 的用法。

```
let valuesList = [ ("a", 1); ("b", 2); ("c", 3) ]

let resultPick = List.pick (fun elem ->
    match elem with
    | (value, 2) -> Some value
    | _ -> None) valuesList

printfn "%A" resultPick
```

輸出如下所示：

```
"b"
```

另一個搜尋作業群組[tryFind](#)和相關函式會傳回選項值。`List.tryFind` 函式會傳回 list 中第一個滿足條件的元素(如其存在);若找不到，即會傳回選項值 `None`。[Array.tryFindIndex](#)會傳回專案的索引(如果找到的話)，而不是元素本身。下列程式碼是這些函式的示範。

```
let list1d = [1; 3; 7; 9; 11; 13; 15; 19; 22; 29; 36]
let isEven x = x % 2 = 0
match List.tryFind isEven list1d with
| Some value -> printfn "The first even value is %d." value
| None -> printfn "There is no even value in the list."

match List.tryFindIndex isEven list1d with
| Some value -> printfn "The first even value is at position %d." value
| None -> printfn "There is no even value in the list."
```

輸出如下所示：

```
The first even value is 22.
The first even value is at position 8.
```

List 的算術運算

清單模組內建常見的算數運算，例如 `sum` 和 `average`。若要使用 `list.sum`，`list` 元素類型必須支援 `+` 運算子，且值為零。所有內建算術類型皆滿足這些條件。若要使用 `List.average`，元素類型必須支援不含餘數的除法，這會排除整數類型，但允許浮點類型。`SumBy`和`averageBy`函式會採用函式做為參數，並使用此函數的結果來計算總和或平均值的值。

下列程式碼示範 `List.sum`、`List.sumBy` 及 `List.average` 的用法。

```
// Compute the sum of the first 10 integers by using List.sum.
let sum1 = List.sum [1 .. 10]

// Compute the sum of the squares of the elements of a list by using List.sumBy.
let sum2 = List.sumBy (fun elem -> elem*elem) [1 .. 10]

// Compute the average of the elements of a list by using List.average.
let avg1 = List.average [0.0; 1.0; 1.0; 2.0]

printfn "%f" avg1
```

輸出為 `1.000000`。

下列程式碼示範 `List.averageBy` 的用法。

```
let avg2 = List.averageBy (fun elem -> float elem) [1 .. 10]
printfn "%f" avg2
```

輸出為 `5.5`。

List 與 Tuple

`zip` 及 `unzip` 函式可以操作包含 tuple 的 `list`。這些函式會將各自包含一個值的兩個 `list` 合併成一份元組清單，或將一份元組清單分割成兩個只含單一值的 `list`。最簡單的 `List.zip` 函式會採用兩個單一專案清單，並產生一組成對的元組。另一個版本 (`List.zip3`) 接受單一專案的三個清單，並產生具有三個元素的單一元組清單。下列程式碼範例示範 `List.zip` 的用法。

```
let list1 = [ 1; 2; 3 ]
let list2 = [ -1; -2; -3 ]
let listZip = List.zip list1 list2
printfn "%A" listZip
```

輸出如下所示：

```
[(1, -1); (2, -2); (3, -3)]
```

下列程式碼範例示範 `List.zip3` 的用法。

```
let list3 = [ 0; 0; 0 ]
let listZip3 = List.zip3 list1 list2 list3
printfn "%A" listZip3
```

輸出如下所示：

```
[(1, -1, 0); (2, -2, 0); (3, -3, 0)]
```

對應的解壓縮版本 `list.unzip3` 會接受元組的清單和傳回清單，其中第一個清單會包含每個元組中第一個專案的所有專案，而第二個清單則包含每個元組的第二個元素，依此類推。

下列程式碼範例示範如何使用[List. 解壓縮](#)。

```
let lists = List.unzip [(1,2); (3,4)]
printfn "%A" lists
printfn "%A %A" (fst lists) (snd lists)
```

輸出如下所示：

```
([1; 3], [2; 4])
[1; 3] [2; 4]
```

下列程式碼範例示範如何使用[list.unzip3](#)。

```
let listsUnzip3 = List.unzip3 [(1,2,3); (4,5,6)]
printfn "%A" listsUnzip3
```

輸出如下所示：

```
([1; 4], [2; 5], [3; 6])
```

List 元素的運算

F# 支援多種 list 元素運算。最簡單的是[iter](#)，可讓您在清單的每個元素上呼叫函式。變異數包含[array.iter2](#)，可讓您在兩個清單的專案上執行作業，[list.iteri](#)，這類似，[List.iter](#) 不同之處在於每個專案的索引都會當做引數傳遞給每個元素所呼叫的函式，而[array.iteri2](#)則是和的功能組合。[List.iter2](#) [List.iteri](#) 下列程式碼範例會示範這些函數。

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
List.iter (fun x -> printfn "List.iter: element is %d" x) list1
List.iteri (fun i x -> printfn "List.iteri: element %d is %d" i x) list1
List.iter2 (fun x y -> printfn "List.iter2: elements are %d %d" x y) list1 list2
List.iteri2 (fun i x y ->
    printfn "List.iteri2: element %d of list1 is %d element %d of list2 is %d"
        i x y)
    list1 list2
```

輸出如下所示：

```
List.iter: element is 1
List.iter: element is 2
List.iter: element is 3
List.iteri: element 0 is 1
List.iteri: element 1 is 2
List.iteri: element 2 is 3
List.iter2: elements are 1 4
List.iter2: elements are 2 5
List.iter2: elements are 3 6
List.iteri2: element 0 of list1 is 1; element 0 of list2 is 4
List.iteri2: element 1 of list1 is 2; element 1 of list2 is 5
List.iteri2: element 2 of list1 is 3; element 2 of list2 is 6
```

轉換清單元素的另一個常用函式是 [\[清單\]](#)，可讓您在清單的每個元素上套函式，並將所有的結果放入新的清單中。[List.map2](#)和[list.map3](#)是採用多個清單的變化。您也可以使用[list.mapi2](#)，如果除了元素之外，還必須傳遞每個專案的索引給函式。[List.mapi2](#) 與 [List.mapi](#) 的唯一差別在於 [List.mapi2](#) 可與兩個 list 搭配使用。下列範例說明了清單。

```
let list1 = [1; 2; 3]
let newList = List.map (fun x -> x + 1) list1
printfn "%A" newList
```

輸出如下所示：

```
[2; 3; 4]
```

下列範例示範 `List.map2` 的用法。

```
let list1 = [1; 2; 3]
let list2 = [4; 5; 6]
let sumList = List.map2 (fun x y -> x + y) list1 list2
printfn "%A" sumList
```

輸出如下所示：

```
[5; 7; 9]
```

下列範例示範 `List.map3` 的用法。

```
let newList2 = List.map3 (fun x y z -> x + y + z) list1 list2 [2; 3; 4]
printfn "%A" newList2
```

輸出如下所示：

```
[7; 10; 13]
```

下列範例示範 `List.mapi` 的用法。

```
let newListAddIndex = List.mapi (fun i x -> x + i) list1
printfn "%A" newListAddIndex
```

輸出如下所示：

```
[1; 3; 5]
```

下列範例示範 `List.mapi2` 的用法。

```
let listAddTimesIndex = List.mapi2 (fun i x y -> (x + y) * i) list1 list2
printfn "%A" listAddTimesIndex
```

輸出如下所示：

```
[0; 7; 18]
```

`List.collect`就像 `List.map`，不同的是，每個元素都會產生一個清單，而所有這些清單都會串連成最後一個清單。在下列程式碼中，list 的每個元素都會產生三個數字。所有數字都會收集到一個 list 中。

```
let collectList = List.collect (fun x -> [for i in 1..3 -> x * i]) list1
printfn "%A" collectList
```

輸出如下所示：

```
[1; 2; 3; 2; 4; 6; 3; 6; 9]
```

您也可以使用 [\[清單\] 篩選](#)，它會採用布林值條件，並產生只包含符合指定條件之元素的新清單。

```
let evenOnlyList = List.filter (fun x -> x % 2 = 0) [1; 2; 3; 4; 5; 6]
```

所得 list 為 `[2; 4; 6]`。

[\[對應\]](#) 和 [\[篩選\]](#)、[\[清單\]](#) 的組合，可讓您同時轉換及選取元素。`List.choose` 會套用可以傳回選項給每個 list 元素的函式，並會在函式傳回選項值 `Some` 時，為元素傳回新的結果清單。

下列程式碼示範如何使用 `List.choose` 選取不在單字清單內的大寫單字。

```
let listWords = [ "and"; "Rome"; "Bob"; "apple"; "zebra" ]
let isCapitalized (string:string) = System.Char.IsUpper string.[0]
let results = List.choose (fun elem ->
    match elem with
    | elem when isCapitalized elem -> Some(elem + "'s")
    | _ -> None) listWords
printfn "%A" results
```

輸出如下所示：

```
["Rome's"; "Bob's"]
```

多個 List 的運算

多個 list 可以相互結合。若要將兩個清單聯結成一個，請使用 [List.append](#)。若要加入兩個以上的清單，請使用 [List.concat](#)。

```
let list1to10 = List.append [1; 2; 3] [4; 5; 6; 7; 8; 9; 10]
let listResult = List.concat [ [1; 2; 3]; [4; 5; 6]; [7; 8; 9] ]
List.iter (fun elem -> printf "%d " elem) list1to10
printfn ""
List.iter (fun elem -> printf "%d " elem) listResult
```

Fold 與 Scan 運算

有些 list 運算涉及所有 list 元素間彼此的相依性。折迭和掃描工作就像 `List.iter` 和 `List.map` 在中，您會在每個專案上叫用函式，但這些作業會提供一個額外的參數，稱為透過計算來攜帶資訊的 *累計*。

使用 `List.fold` 對 list 執行運算。

下列程式碼範例示範如何使用 [List.折頁](#) 來執行各種作業。

清單會經過周遊；accumulator `acc` 一值會隨著運算進行而傳遞。第一個引數在接受 accumulator 及 list 元素之後，會傳回 list 元素的暫時結果。第二個引數是 accumulator 的初始值。

```

let sumList list = List.fold (fun acc elem -> acc + elem) 0 list
printfn "Sum of the elements of list %A is %d." [ 1 .. 3 ] (sumList [ 1 .. 3 ])

// The following example computes the average of a list.
let averageList list = (List.fold (fun acc elem -> acc + float elem) 0.0 list / float list.Length)

// The following example computes the standard deviation of a list.
// The standard deviation is computed by taking the square root of the
// sum of the variances, which are the differences between each value
// and the average.
let stdDevList list =
    let avg = averageList list
    sqrt (List.fold (fun acc elem -> acc + (float elem - avg) ** 2.0 ) 0.0 list / float list.Length)

let testList listTest =
    printfn "List %A average: %f stddev: %f" listTest (averageList listTest) (stdDevList listTest)

testList [1; 1; 1]
testList [1; 2; 1]
testList [1; 2; 3]

// List.fold is the same as to List.iter when the accumulator is not used.
let printList list = List.fold (fun acc elem -> printfn "%A" elem) () list
printList [0.0; 1.0; 2.5; 5.1 ]

// The following example uses List.fold to reverse a list.
// The accumulator starts out as the empty list, and the function uses the cons operator
// to add each successive element to the head of the accumulator list, resulting in a
// reversed form of the list.
let reverseList list = List.fold (fun acc elem -> elem::acc) [] list
printfn "%A" (reverseList [1 .. 10])

```

這些函式若對多個 list 執行，函式名稱中會有一位數字記錄其版本。例如，[list.fold2](#)會在兩個清單上執行計算。

下列範例示範 `List.fold2` 的用法。

```

// Use List.fold2 to perform computations over two lists (of equal size) at the same time.
// Example: Sum the greater element at each list position.
let sumGreatest list1 list2 = List.fold2 (fun acc elem1 elem2 ->
    acc + max elem1 elem2) 0 list1 list2

let sum = sumGreatest [1; 2; 3] [3; 2; 1]
printfn "The sum of the greater of each pair of elements in the two lists is %d." sum

```

`List.fold` 和 [\[清單\]](#) 中的不同，它會傳回 `List.fold` 額外參數的最後一個值，但會傳回 `List.scan` (的中繼值清單，以及額外參數的最終值)。

其中每個函式都包含反向變化(例如[list.foldback](#))，這會依清單的進行順序和引數的順序而有所不同。此外，`List.fold` 和 `List.foldBack` 具有[list.fold2](#)和[list.foldback2](#)這兩個長度相同的清單。對每個元素執行的函式，皆可使用兩個 list 的對應元素執行特定動作。如下列範例所示，兩個 list 的元素類型可以不同，其中一個 list 包含銀行帳戶的異動金額，另一個 list 則包含異動的類型 (存款或提款)。

```
// Discriminated union type that encodes the transaction type.
type Transaction =
  | Deposit
  | Withdrawal

let transactionTypes = [Deposit; Deposit; Withdrawal]
let transactionAmounts = [100.00; 1000.00; 95.00 ]
let initialBalance = 200.00

// Use fold2 to perform a calculation on the list to update the account balance.
let endingBalance = List.fold2 (fun acc elem1 elem2 ->
    match elem1 with
    | Deposit -> acc + elem2
    | Withdrawal -> acc - elem2)
    initialBalance
    transactionTypes
    transactionAmounts

printfn "%f" endingBalance
```

對於加總這類運算，因為 `List.fold` 與 `List.foldBack` 的結果與周遊順序無關，所以兩者的效果相同。在下列範例中，`List.foldBack` 會用於新增 list 中的元素。

```
let sumListBack list = List.foldBack (fun elem acc -> acc + elem) list 0
printfn "%d" (sumListBack [1; 2; 3])

// For a calculation in which the order of traversal is important, fold and foldBack have different
// results. For example, replacing fold with foldBack in the listReverse function
// produces a function that copies the list, rather than reversing it.
let copyList list = List.foldBack (fun elem acc -> elem::acc) list []
printfn "%A" (copyList [1 .. 10])
```

下列範例會回到銀行帳戶範例。這次會新增「利息運算」異動類型。最終結餘取決於異動的順序。

```

type Transaction2 =
    | Deposit
    | Withdrawal
    | Interest

let transactionTypes2 = [Deposit; Deposit; Withdrawal; Interest]
let transactionAmounts2 = [100.00; 1000.00; 95.00; 0.05 / 12.0 ]
let initialBalance2 = 200.00

// Because fold2 processes the lists by starting at the head element,
// the interest is calculated last, on the balance of 1205.00.
let endingBalance2 = List.fold2 (fun acc elem1 elem2 ->
    match elem1 with
    | Deposit -> acc + elem2
    | Withdrawal -> acc - elem2
    | Interest -> acc * (1.0 + elem2))
    initialBalance2
    transactionTypes2
    transactionAmounts2

printfn "%f" endingBalance2

// Because foldBack2 processes the lists by starting at end of the list,
// the interest is calculated first, on the balance of only 200.00.
let endingBalance3 = List.foldBack2 (fun elem1 elem2 acc ->
    match elem1 with
    | Deposit -> acc + elem2
    | Withdrawal -> acc - elem2
    | Interest -> acc * (1.0 + elem2))
    transactionTypes2
    transactionAmounts2
    initialBalance2

printfn "%f" endingBalance3

```

函式[清單。[縮小](#)]有點類似 `List.fold` 和 `List.scan`，不同之處在於 `List.reduce` 會採用接受元素類型之兩個引數的函式，而不是只有一個，而其中一個引數會作為累計結果，這表示它會儲存計算的中繼結果。

`List.reduce` 會從運算頭兩個 list 元素開始，然後再並用運算結果與下一個元素。因為沒有任何具有專屬類型的 accumulator，所以可以使用 `List.reduce` 取代 `List.fold`，但前提是 accumulator 與元素類型必須屬於相同類型。下列程式碼示範 `List.reduce` 的用法。若提供的 list 不含任何元素，`List.reduce` 會擲出例外狀況。

在下列程式碼中，第一個 Lambda 運算式呼叫的引數設定為 2 與 4，並傳回 6；下一個呼叫的引數設定為 6 與 10，所以結果為 16。

```

let sumAList list =
    try
        List.reduce (fun acc elem -> acc + elem) list
    with
        | :? System.ArgumentException as exc -> 0

let resultSum = sumAList [2; 4; 10]
printfn "%d " resultSum

```

List 與其他收集類型的相互轉換

`List` 模組提供可以讓 sequence 與 array 相互轉換的函式。若要在序列之間轉換，請使用 [list.toSeq](#) 或 [list.ofSeq](#)。若要在陣列之間進行轉換，請使用 [toArray](#) 或 [list.ofArray](#)。

其他運算

如需有關清單上其他作業的詳細資訊，請參閱程式庫參考主題 [集合](#)。[清單模組](#)。

另請參閱

- [F# 語言參考](#)
- [F# 類型](#)
- [序列](#)
- [陣列](#)
- [選項](#)

陣列

2021/3/5 • [Edit Online](#)

陣列是固定大小、以零為基礎且可變動的連續資料元素集合，這些專案全都屬於相同類型。

建立陣列

您可以用數種方式建立陣列。您可以藉由列出介於和之間的連續值 `[| |]`，並以分號分隔來建立小型陣列，如下列範例所示。

```
let array1 = [| 1; 2; 3 |]
```

您也可以將每個專案放在個別的行上，在這種情況下，分號分隔符號是選擇性的。

```
let array1 =  
    [  
        1  
        2  
        3  
    |]
```

陣列元素的型別是從使用的常值推斷而來，而且必須是一致的。下列程式碼會造成錯誤，因為1.0 是 float，而2和3是整數。

```
// Causes an error.  
// let array2 = [| 1.0; 2; 3 |]
```

您也可以使用順序運算式來建立陣列。以下範例會建立從1到10的整數平方陣列。

```
let array3 = [| for i in 1 .. 10 -> i * i |]
```

若要建立陣列，其中所有元素都會初始化為零，請使用 `Array.zeroCreate`。

```
let arrayOfTenZeroes : int array = Array.zeroCreate 10
```

存取元素

您可以使用點運算子來存取陣列元素 (`.`) 和方括弧 (`[` 和 `]`)。

```
array1.[0]
```

陣列索引從0開始。

您也可以使用配量標記法來存取陣列元素，這可讓您指定陣列的子範圍。配量標記法的範例如下。

```
// Accesses elements from 0 to 2.

array1.[0..2]

// Accesses elements from the beginning of the array to 2.

array1[..2]

// Accesses elements from 2 to the end of the array.

array1.[2..]
```

使用配量標記法時，會建立陣列的新複本。

陣列類型和模組

所有 F # 陣列的型別都是 .NET Framework 型別 [System.Array](#) 。因此, F # 陣列支援中所有可用的功能 [System.Array](#) 。

此 [Array](#) 模組支援一維陣列上的作業。模組 [Array2D](#) 、 [Array3D](#) 和包含的函式 [Array4D](#) 分別支援兩個、三個和四個維度的陣列作業。您可以使用來建立順位大於四的陣列 [System.Array](#) 。

簡單函數

[Array.get](#) 取得專案。 [Array.length](#) 提供陣列的長度。 [Array.set](#) 將元素設定為指定的值。下列程式碼範例說明這些函數的用法。

```
let array1 = Array.create 10 ""
for i in 0 .. array1.Length - 1 do
    Array.set array1 i (i.ToString())
for i in 0 .. array1.Length - 1 do
    printf "%s " (Array.get array1 i)
```

輸出如下。

```
0 1 2 3 4 5 6 7 8 9
```

建立陣列的函式

有幾個函式會建立陣列，而不需要現有的陣列。 [Array.empty](#) 建立不包含任何專案的新陣列。 [Array.create](#) 建立指定大小的陣列，並將所有元素設定為提供的值。 [Array.init](#) 建立陣列，並指定維度和函式來產生元素。 [Array.zeroCreate](#) 建立陣列，其中所有元素都會初始化為數組類型的零值。下列程式碼示範這些函數。

```
let myEmptyArray = Array.empty
printfn "Length of empty array: %d" myEmptyArray.Length

printfn "Array of floats set to 5.0: %A" (Array.create 10 5.0)

printfn "Array of squares: %A" (Array.init 10 (fun index -> index * index))

let (myZeroArray : float array) = Array.zeroCreate 10
```

輸出如下。

```
Length of empty array: 0
Area of floats set to 5.0: [|5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0; 5.0|]
Array of squares: [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.copy` 建立新的陣列，其中包含從現有陣列複製的元素。請注意，複製是淺層複製，這表示如果元素類型是參考型別，則只會複製參考，而不會複製基礎物件。下列程式碼範例會說明這點。

```
open System.Text

let firstArray : StringBuilder array = Array.init 3 (fun index -> new StringBuilder(""))
let secondArray = Array.copy firstArray
// Reset an element of the first array to a new value.
firstArray.[0] <- new StringBuilder("Test1")
// Change an element of the first array.
firstArray.[1].Insert(0, "Test2") |> ignore
printfn "%A" firstArray
printfn "%A" secondArray
```

上述程式碼的輸出如下所示：

```
[|Test1; Test2; |]
[|; Test2; |]
```

字串 `Test1` 只會出現在第一個陣列中，因為建立新專案的作業會覆寫中的參考，`firstArray` 但不會影響仍然存在於中之空字串的原始參考 `secondArray`。字串 `Test2` 會出現在這兩個數組中 `Insert`，因為類型上的作業 `System.Text.StringBuilder` 會影響在 `System.Text.StringBuilder` 這兩個數組中所參考的基礎物件。

`Array.sub` 從陣列的子範圍產生新的陣列。您可以藉由提供開始索引和長度來指定子範圍。下列程式碼示範 `Array.sub` 的用法。

```
let a1 = [| 0 .. 99 |]
let a2 = Array.sub a1 5 10
printfn "%A" a2
```

輸出顯示子陣列在元素5開始，且包含10個元素。

```
[|5; 6; 7; 8; 9; 10; 11; 12; 13; 14|]
```

`Array.append` 藉由結合兩個現有的陣列，建立新的陣列。

下列程式碼示範 陣列. `append`。

```
printfn "%A" (Array.append [| 1; 2; 3|] [| 4; 5; 6|])
```

上述程式碼的輸出如下所示。

```
[|1; 2; 3; 4; 5; 6|]
```

`Array.choose` 選取要包含在新陣列中的陣列元素。下列程式碼示範 `Array.choose`。請注意，陣列的元素類型不需要符合選項類型中所傳回值的類型。在此範例中，元素類型為，`int` 而選項是多項式函式的結果，`elem*elem - 1` 做為浮點數。

```
printfn "%A" (Array.choose (fun elem -> if elem % 2 = 0 then
    Some(float (elem*elem - 1))
    else
    None) [| 1 .. 10 |])
```

上述程式碼的輸出如下所示。

```
[|3.0; 15.0; 35.0; 63.0; 99.0|]
```

`Array.collect` 在現有陣列的每個陣列元素上執行指定的函式，然後收集函式所產生的元素，並將它們合併成新的陣列。下列程式碼示範 `Array.collect` 。

```
printfn "%A" (Array.collect (fun elem -> [| 0 .. elem |]) [| 1; 5; 10|])
```

上述程式碼的輸出如下所示。

```
[|0; 1; 0; 1; 2; 3; 4; 5; 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

`Array.concat` 接受一連串的陣列，並將它們結合成單一陣列。下列程式碼示範 `Array.concat` 。

```
Array.concat [ [|0..3|] ; [|4|] ]
//output [|0; 1; 2; 3; 4|]

Array.concat [| [|0..3|] ; [|4|] |]
//output [|0; 1; 2; 3; 4|]
```

上述程式碼的輸出如下所示。

```
[|(1, 1, 1); (1, 2, 2); (1, 3, 3); (2, 1, 2); (2, 2, 4); (2, 3, 6); (3, 1, 3);
(3, 2, 6); (3, 3, 9)|]
```

`Array.filter` 採用布林條件函式，並產生新的陣列，其中只包含輸入陣列中條件為 `true` 的元素。下列程式碼示範 `Array.filter` 。

```
printfn "%A" (Array.filter (fun elem -> elem % 2 = 0) [| 1 .. 10|])
```

上述程式碼的輸出如下所示。

```
[|2; 4; 6; 8; 10|]
```

`Array.rev` 藉由反轉現有陣列的順序來產生新的陣列。下列程式碼示範 `Array.rev` 。

```
let stringReverse (s: string) =
    System.String(Array.rev (s.ToCharArray()))

printfn "%A" (stringReverse("!dlrow olleH"))
```

上述程式碼的輸出如下所示。

```
"Hello world!"
```

您可以輕鬆地結合陣列模組中的函式，使用管線運算子 `()` 來轉換陣列 `|>`，如下列範例所示。

```
[| 1 .. 10 |]  
|> Array.filter (fun elem -> elem % 2 = 0)  
|> Array.choose (fun elem -> if (elem <> 8) then Some(elem*elem) else None)  
|> Array.rev  
|> printfn "%A"
```

輸出為

```
[|100; 36; 16; 4|]
```

多維陣列

您可以建立多維度陣列，但是沒有寫入多維陣列常值的語法。使用運算子 `array2D`，從陣列元素序列的序列中建立陣列。序列可以是陣列或清單常值。例如，下列程式碼會建立二維陣列。

```
let my2DArray = array2D [ [ 1; 0 ]; [0; 1] ]
```

您也可以使用函式 `Array2D.init` 來初始化兩個維度的陣列，而類似的函式可用於三和四個維度的陣列。這些函式會取得用來建立元素的函式。若要建立包含設定為初始值之專案的二維陣列，而不是指定函式，請使用函式 `Array2D.create`，此函式也適用於最多四個維度的陣列。下列程式碼範例會先示範如何建立包含所需元素的陣列陣列，然後使用 `Array2D.init` 來產生所需的二維陣列。

```
let arrayOfArrays = [ [| 1.0; 0.0 |]; [|0.0; 1.0 |] ]  
let twoDimensionalArray = Array2D.init 2 2 (fun i j -> arrayOfArrays.[i].[j])
```

陣列索引和切割語法支援最高排名4的陣列。當您指定多個維度中的索引時，您可以使用逗號來分隔索引，如下列程式碼範例所示。

```
twoDimensionalArray.[0, 1] <- 1.0
```

二維陣列的型別會寫出做為 (例如，`<type>[,]` `int[,]` `double[,]`)，而三維陣列的型別則是撰寫為 `<type>[,,,]`，依此類推，針對較高維度的陣列。

只有一維陣列的可用函式子集也適用於多維度陣列。

陣列切割和多維陣列

在二維陣列 (矩陣) 中，您可以指定範圍，並使用萬用字元 `(*)` 字元來指定整個資料列或資料行，藉以解壓縮子矩陣。

```
// Get rows 1 to N from an NxM matrix (returns a matrix):  
matrix[1.., *]  
  
// Get rows 1 to 3 from a matrix (returns a matrix):  
matrix[1..3, *]  
  
// Get columns 1 to 3 from a matrix (returns a matrix):  
matrix[:, 1..3]  
  
// Get a 3x3 submatrix:  
matrix[1..3, 1..3]
```

您可以將多維陣列分解成相同或較低維度的 subarrays。例如，您可以藉由指定單一資料列或資料行，從矩陣取得向量。

```
// Get row 3 from a matrix as a vector:  
matrix[3, *]  
  
// Get column 3 from a matrix as a vector:  
matrix[:, 3]
```

您可以針對實際引數存取運算子和多載方法的型別使用此切割語法 `GetSlice`。例如，下列程式碼會建立包裝 F# 2D 陣列的矩陣類型、執行專案屬性以提供陣列索引編制的支援，以及執行三個版本的 `GetSlice`。如果您可以使用此程式碼作為矩陣類型的範本，您可以使用本節描述的所有配量作業。

```

type Matrix<'T>(N: int, M: int) =
  let internalArray = Array2D.zeroCreate<'T> N M

  member this.Item
    with get(a: int, b: int) = internalArray.[a, b]
    and set(a: int, b: int) (value:'T) = internalArray.[a, b] <- value

  member this.GetSlice(rowStart: int option, rowFinish : int option, colStart: int option, colFinish : int option) =
    let rowStart =
      match rowStart with
      | Some(v) -> v
      | None -> 0
    let rowFinish =
      match rowFinish with
      | Some(v) -> v
      | None -> internalArray.GetLength(0) - 1
    let colStart =
      match colStart with
      | Some(v) -> v
      | None -> 0
    let colFinish =
      match colFinish with
      | Some(v) -> v
      | None -> internalArray.GetLength(1) - 1
    internalArray.[rowStart..rowFinish, colStart..colFinish]

  member this.GetSlice(row: int, colStart: int option, colFinish: int option) =
    let colStart =
      match colStart with
      | Some(v) -> v
      | None -> 0
    let colFinish =
      match colFinish with
      | Some(v) -> v
      | None -> internalArray.GetLength(1) - 1
    internalArray.[row, colStart..colFinish]

  member this.GetSlice(rowStart: int option, rowFinish: int option, col: int) =
    let rowStart =
      match rowStart with
      | Some(v) -> v
      | None -> 0
    let rowFinish =
      match rowFinish with
      | Some(v) -> v
      | None -> internalArray.GetLength(0) - 1
    internalArray.[rowStart..rowFinish, col]

module test =
  let generateTestMatrix x y =
    let matrix = new Matrix<float>(3, 3)
    for i in 0..2 do
      for j in 0..2 do
        matrix.[i, j] <- float(i) * x - float(j) * y
    matrix

  let test1 = generateTestMatrix 2.3 1.1
  let submatrix = test1.[0..1, 0..1]
  printfn $"{submatrix}"

  let firstRow = test1.[0,*]
  let secondRow = test1.[1,*]
  let firstCol = test1.[*,0]
  printfn $"{firstCol}"

```


陣列上的布耳函數

`Array.exists` `Array.exists2` 一或兩個數組中的函式和測試專案，分別為。 `true` 如果符合條件的) 有元素 (或元素組，則這些函式會採用測試函式，並傳回 `Array.exists2` 。

下列程式碼示範如何使用 `Array.exists` 和 `Array.exists2` 。在這些範例中，會藉由只套用其中一個引數 (在這些情況下為函式引數) 來建立新的函式。

```
let allNegative = Array.exists (fun elem -> abs (elem) = elem) >> not
printfn "%A" (allNegative [| -1; -2; -3 |])
printfn "%A" (allNegative [| -10; -1; 5 |])
printfn "%A" (allNegative [| 0 |])

let haveEqualElement = Array.exists2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (haveEqualElement [| 1; 2; 3 |] [| 3; 2; 1 |])
```

上述程式碼的輸出如下所示。

```
true
false
false
true
```

同樣地，函 `Array.forall` 式會測試陣列，以判斷每個專案是否符合布林值條件。變異 `Array.forall2` 會使用布耳函數來執行相同的動作，此函式牽涉到兩個相等長度陣列的元素。下列程式碼說明如何使用這些函數。

```
let allPositive = Array.forall (fun elem -> elem > 0)
printfn "%A" (allPositive [| 0; 1; 2; 3 |])
printfn "%A" (allPositive [| 1; 2; 3 |])

let allEqual = Array.forall2 (fun elem1 elem2 -> elem1 = elem2)
printfn "%A" (allEqual [| 1; 2 |] [| 1; 2 |])
printfn "%A" (allEqual [| 1; 2 |] [| 2; 1 |])
```

這些範例的輸出如下所示。

```
false
true
true
false
```

搜尋陣列

`Array.find` 採用布林函式，並傳回函式傳回的第一個專案 `true` ，`System.Collections.Generic.KeyNotFoundException` 如果找不到符合條件的專案，則會引發。 `Array.findIndex` 類似，不同的是它會傳回專案的 `Array.find` 索引，而不是元素本身。

下列 `Array.find` 程式碼會使用和 `Array.findIndex` 來找出同時為完美方形和完美 cube 的數位。

```

let arrayA = [| 2 .. 100 |]
let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let element = Array.find (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
let index = Array.findIndex (fun elem -> isPerfectSquare elem && isPerfectCube elem) arrayA
printfn "The first element that is both a square and a cube is %d and its index is %d." element index

```

輸出如下。

```
The first element that is both a square and a cube is 64 and its index is 62.
```

`Array.tryFind` 類似 `Array.find`，不同之處在於它的結果是選項類型，`None` 如果找不到任何元素，則會傳回。`Array.tryFind`Array.find` 如果您不知道相符的專案是否在陣列中，則應該使用而不是。同樣地，它 `Array.tryFindIndex` 也是一樣，`Array.findIndex` 但選項類型是傳回值。如果找不到任何元素，則選項為 `None`。

下列程式碼示範 `Array.tryFind` 的用法。此程式碼相依於先前的程式碼。

```

let delta = 1.0e-10
let isPerfectSquare (x:int) =
    let y = sqrt (float x)
    abs(y - round y) < delta
let isPerfectCube (x:int) =
    let y = System.Math.Pow(float x, 1.0/3.0)
    abs(y - round y) < delta
let lookForCubeAndSquare array1 =
    let result = Array.tryFind (fun elem -> isPerfectSquare elem && isPerfectCube elem) array1
    match result with
    | Some x -> printfn "Found an element: %d" x
    | None -> printfn "Failed to find a matching element."

lookForCubeAndSquare [| 1 .. 10 |]
lookForCubeAndSquare [| 100 .. 1000 |]
lookForCubeAndSquare [| 2 .. 50 |]

```

輸出如下。

```

Found an element: 1
Found an element: 729
Failed to find a matching element.

```

`Array.tryPick` 當您需要轉換元素，並加以尋找時，請使用。結果為函式傳回已轉換元素做為選項值的第一個專案，`None` 如果找不到這類元素，則為。

下列程式碼示範 `Array.tryPick` 的用法。在此情況下，不會使用 lambda 運算式，而是定義數個本機 helper 函數來簡化程式碼。

```

let findPerfectSquareAndCube array1 =
    let delta = 1.0e-10
    let isPerfectSquare (x:int) =
        let y = sqrt (float x)
        abs(y - round y) < delta
    let isPerfectCube (x:int) =
        let y = System.Math.Pow(float x, 1.0/3.0)
        abs(y - round y) < delta
    // intFunction : (float -> float) -> int -> int
    // Allows the use of a floating point function with integers.
    let intFunction function1 number = int (round (function1 (float number)))
    let cubeRoot x = System.Math.Pow(x, 1.0/3.0)
    // testElement: int -> (int * int * int) option
    // Test an element to see whether it is a perfect square and a perfect
    // cube, and, if so, return the element, square root, and cube root
    // as an option value. Otherwise, return None.
    let testElement elem =
        if isPerfectSquare elem && isPerfectCube elem then
            Some(elem, intFunction sqrt elem, intFunction cubeRoot elem)
        else None
    match Array.tryPick testElement array1 with
    | Some (n, sqrt, cuberoot) -> printfn "Found an element %d with square root %d and cube root %d." n sqrt
    cuberoot
    | None -> printfn "Did not find an element that is both a perfect square and a perfect cube."

findPerfectSquareAndCube [| 1 .. 10 |]
findPerfectSquareAndCube [| 2 .. 100 |]
findPerfectSquareAndCube [| 100 .. 1000 |]
findPerfectSquareAndCube [| 1000 .. 10000 |]
findPerfectSquareAndCube [| 2 .. 50 |]

```

輸出如下。

```

Found an element 1 with square root 1 and cube root 1.
Found an element 64 with square root 8 and cube root 4.
Found an element 729 with square root 27 and cube root 9.
Found an element 4096 with square root 64 and cube root 16.
Did not find an element that is both a perfect square and a perfect cube.

```

在陣列上執行計算

函數會傳回 `Array.average` 陣列中每個元素的平均值。它受限於支援整數除以整數的專案類型，其中包含浮點數類型，但不包括整數類資料類型。函數會傳回在 `Array.averageBy` 每個專案上呼叫函式的結果平均值。若為整數類資料類型的陣列，您可以使用 `Array.averageBy`，並讓函式將每個元素轉換成浮點數類型以進行計算。

`Array.max` 如果專案 `Array.min` 類型支援，請使用或來取得最大或最小元素。同樣地，也可讓函式 `Array.maxBy` `Array.minBy` 先執行，也許可以轉換成支援比較的类型。

`Array.sum` 加入陣列的元素，並 `Array.sumBy` 在每個專案上呼叫函式，並將結果加在一起。

若要在陣列中的每個元素上執行函式，而不儲存傳回值，請使用 `Array.iter`。若為涉及兩個相等長度陣列的函式，請使用 `Array.iter2`。如果您也需要保留函式結果的陣列，請使用 `Array.map` 或 `Array.map2`，它會一次在兩個數組上運作。

變數 `Array.iteri` 和可 `Array.iteri2` 讓元素的索引參與計算；和相同的情況也是如此 `Array.mapi` `Array.mapi2`。

`Array.fold` 包含陣列所有元素的函式、、、`Array.foldBack` `Array.reduce` `Array.reduceBack` `Array.scan` 和 `Array.scanBack` 執行演算法。同樣地，變化 `Array.fold2` 和 `Array.foldBack2` 在兩個數組上執行計算。

這些執行計算的函式會對應到 [清單模組](#) 中相同名稱的函式。如需使用範例，請參閱 [清單](#)。

修改陣列

`Array.set` 將元素設定為指定的值。`Array.fill` 將陣列中的元素範圍設定為指定的值。下列程式碼提供的範例 `Array.fill` 。

```
let arrayFill1 = [| 1 .. 25 |]  
Array.fill arrayFill1 2 20 0  
printfn "%A" arrayFill1
```

輸出如下。

```
[|1; 2; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 23; 24; 25|]
```

您可以使用將 `Array.blit` 某個陣列的子區段複製到另一個陣列。

從其他類型轉換

`Array.ofList` 從清單建立陣列。`Array.ofSeq` 從序列建立陣列。`Array.toList` 並 `Array.toSeq` 從陣列型別轉換為這些其他集合類型。

排序陣列

使用 `Array.sort` 來排序陣列，方法是使用泛型比較函數。使用指定一個函式，此函式 `Array.sortBy` 會產生值（稱為索引 鍵），以在索引鍵上使用泛型比較函數進行排序。`Array.sortWith` 如果您想要提供自訂比較函數，請使用。`Array.sort`、`Array.sortBy` 和 `Array.sortWith` 全都以新陣列的形式傳回已排序的陣列。變化 `Array.sortInPlace`、`Array.sortInPlaceBy` 和會 `Array.sortInPlaceWith` 修改現有的陣列，而不是傳回新的陣列。

陣列和元組

函數 `Array.zip` 和會 `Array.unzip` 將元組的陣列轉換成陣列的元組，反之亦然。`Array.zip3` 和 `Array.unzip3` 很類似，不同之處在於它們會使用三個元素的元組或三個數組的元組。

針對陣列進行平行計算

模組 `Array.Parallel` 包含在陣列上執行平行計算的函數。在版本4之前，以 .NET Framework 版本為目標的應用程式無法使用此模組。

另請參閱

- [F # 語言參考](#)
- [F# 類型](#)

序列

2020/11/2 • [Edit Online](#)

「*序列*」(sequence)是一種元素的邏輯系列，全都是一種類型。當您有大量的資料收集，但不一定會預期使用所有元素時，序列特別有用。個別順序元素只會在必要時計算，因此在不使用所有元素的情況下，序列可以提供比清單更佳的效能。順序是以類型表示 `seq<'T>`，這是的別名 `IEnumerable<T>`。因此，任何實介面的 .NET 型別都 `IEnumerable<T>` 可以用來做為序列。`Seq` 模組可支援涉及序列的操作。

序列運算式

*順序運算式*是評估為序列的運算式。順序運算式可採用許多形式。最簡單的形式會指定範圍。例如，

`seq { 1 .. 5 }` 建立包含五個元素的序列，包括端點1和5。您也可以指定遞增 (，或在兩個雙句點之間遞減)。例如，下列程式碼會建立10的倍數序列。

```
// Sequence that has an increment.  
seq { 0 .. 10 .. 100 }
```

順序運算式是由產生序列值的 F # 運算式所組成。您也可以透過程式設計的方式產生值：

```
seq { for i in 1 .. 10 -> i * i }
```

先前的範例使用 `->` 運算子，可讓您指定其值將成為序列一部分的運算式。您只能使用 `->` (如果其後的程式碼的每個部分都傳回值)。

或者，您也可以指定 `do` 關鍵字，並選擇性 `yield` 的方法如下：

```
seq { for i in 1 .. 10 do yield i * i }  
  
// The 'yield' is implicit and doesn't need to be specified in most cases.  
seq { for i in 1 .. 10 do i * i }
```

下列程式碼會產生一份座標組清單，以及代表方格的陣列中的索引。請注意，第一個 `for` 運算式需要 `do` 指定。

```
let (height, width) = (10, 10)  
  
seq {  
    for row in 0 .. width - 1 do  
        for col in 0 .. height - 1 ->  
            (row, col, row*width + col)  
}
```

`if` 序列中使用的運算式是篩選準則。例如，若要產生只有一個質數的序列，假設您有一個 `isprime` 類型的函式 `int -> bool`，請依照下列方式來建立順序。

```
seq { for n in 1 .. 100 do if isprime n then n }
```

如先前所述，在 `do` 這裡是必要的，因為沒有 `else` 與搭配使用的分支 `if`。如果您嘗試使用 `->`，您會收到錯誤，指出並非所有分支都會傳回值。

yield! 關鍵字

有時，您可能會想要在另一個序列中包含一系列的元素。若要在另一個序列內包含序列，您必須使用 `yield!` 關鍵字：

```
// Repeats '1 2 3 4 5' ten times
seq {
  for _ in 1..10 do
    yield! seq { 1; 2; 3; 4; 5}
}
```

另一種思考方式 `yield!` 是將內部序列壓平合併，然後將它包含在包含順序中。

`yield!` 在運算式中使用時，所有其他的單一值都必須使用 `yield` 關鍵字：

```
// Combine repeated values with their values
seq {
  for x in 1..10 do
    yield x
    yield! seq { for i in 1..x -> i}
}
```

上述範例會產生的值，`x` 以及每個的所有值 `1` `x` `x` 。

範例

第一個範例會使用包含反復專案、篩選和 `yield` 的序列運算式來產生陣列。此程式碼會將1和100之間的質數序列列印到主控台。

```
// Recursive isprime function.
let isprime n =
  let rec check i =
    i > n/2 || (n % i <> 0 && check (i + 1))
  check 2

let aSequence =
  seq {
    for n in 1..100 do
      if isprime n then
        n
  }

for x in aSequence do
  printfn "%d" x
```

下列範例會建立一個由三個元素的元組組成的乘法資料表，其中每個專案都包含兩個因素和產品：

```
let multiplicationTable =
  seq {
    for i in 1..9 do
      for j in 1..9 ->
        (i, j, i*j)
  }
```

下列範例示範如何使用，將 `yield!` 個別序列合併成單一的最終順序。在此情況下，二進位樹狀結構中的每個子樹的序列會串連在遞迴函式中，以產生最終的順序。

```
// Yield the values of a binary tree in a sequence.
type Tree<'a> =
    | Tree of 'a * Tree<'a> * Tree<'a>
    | Leaf of 'a

// inorder : Tree<'a> -> seq<'a>
let rec inorder tree =
    seq {
        match tree with
        | Tree(x, left, right) ->
            yield! inorder left
            yield x
            yield! inorder right
        | Leaf x -> yield x
    }

let mytree = Tree(6, Tree(2, Leaf(1), Leaf(3)), Leaf(9))
let seq1 = inorder mytree
printfn "%A" seq1
```

使用順序

序列支援許多與 [清單](#) 相同的功能。序列也支援使用按鍵產生函數進行分組和計算等作業。序列也支援更多樣化的函式來解壓縮個子序列。

許多資料類型(例如清單、陣列、集合和地圖)都是可列舉的集合，因此是隱含的序列。接受序列做為引數的函式可搭配任何一般 F# 資料型別使用，除了任何執行的 .NET 資料類型之外

`System.Collections.Generic.IEnumerable<'T>`。將其與接受清單作為引數的函式相比較，該函式只能接受清單。

此類型 `seq<'T>` 是的類型縮寫 `IEnumerable<'T>`。這表示任何實作為泛型的型別(

`System.Collections.Generic.IEnumerable<'T>` 包括 F# 中的陣列、清單、集合和對應，以及大部分的 .net 集合型別)都與型別相容，`seq` 而且可以在預期順序的任何地方使用。

模組函式

[Fsharp.core](#) 命名空間中的 [Seq 模組](#) 包含使用順序的函式。這些函式也適用於清單、陣列、對應和集合，因為所有這些類型都是可列舉的，因此可視為序列。

建立序列

您可以使用順序運算式(如先前所述)，或使用特定函數來建立順序。

您可以使用 [seq](#) [空白](#) 來建立空的序列，也可以使用 [seq](#) 來建立只有一個指定元素的序列。

```
let seqEmpty = Seq.empty
let seqOne = Seq.singleton 10
```

您可以使用 [Seq.init](#) 來建立使用您提供的函式來建立元素的序列。您也會提供序列的大小。此函式就像 [List.init](#) 一樣，不同的是在您逐一查看序列之前，不會建立元素。下列程式碼說明如何使用 `Seq.init`。

```
let seqFirst5MultiplesOf10 = Seq.init 5 (fun n -> n * 10)
Seq.iter (fun elem -> printf "%d " elem) seqFirst5MultiplesOf10
```

輸出為

```
0 10 20 30 40
```

您可以使用 `list.ofarray` 和 `seq.ofList< t>` 函數，從陣列和清單建立序列。不過，您也可以使用轉換運算子，將陣列和清單轉換成序列。下列程式碼中會顯示這兩種技術。

```
// Convert an array to a sequence by using a cast.
let seqFromArray1 = [| 1 .. 10 |] :> seq<int>

// Convert an array to a sequence by using Seq.ofArray.
let seqFromArray2 = [| 1 .. 10 |] |> Seq.ofArray
```

藉由使用 `Seq` 轉換，您可以從弱型別集合中建立序列，例如中定義的序列 `System.Collections`。這類弱型別集合具有元素類型 `System.Object`，而且使用非泛型型別來列舉 `System.Collections.Generic.IEnumerable<T>`。下列程式碼說明如何使用將 `Seq.cast` 轉換 `System.Collections.ArrayList` 成序列。

```
open System

let arr = ResizeArray<int>(10)

for i in 1 .. 10 do
    arr.Add(10)

let seqCast = Seq.cast arr
```

您可以使用 `Seq.initInfinite` 函數來定義無限序列。針對這類序列，您會提供一個函式，以從專案的索引產生每個元素。由於延遲評估，因此可能會有無限的順序；您可以藉由呼叫您指定的函式，視需要建立元素。下列程式碼範例會產生無限的浮點數序列，在此案例中為連續整數平方的 reciprocals 序列。

```
let seqInfinite =
    Seq.initInfinite (fun index ->
        let n = float (index + 1)
        1.0 / (n * n * (if ((index + 1) % 2 = 0) then 1.0 else -1.0)))

printfn "%A" seqInfinite
```

`Seq` 會從計算函式產生順序，此函式會接受狀態並將其轉換為產生序列中的每個後續元素。此狀態只是用來計算每個專案的值，而且可在每個元素計算時變更。的第二個引數 `Seq.unfold` 是用來啟動序列的起始值。

`Seq.unfold` 使用狀態的選項類型，可讓您藉由傳回值來終止序列 `None`。下列程式碼顯示兩個序列範例，`seq1` 以及 `fib` 由作業產生的 `unfold`。第一個，`seq1` 是最多20個數字的簡單序列。第二個會 `fib` 使用 `unfold` 來計算斐波里的序列。因為量值序列中的每個元素都是前兩個斐的兩個量值的總和，所以狀態值是由序列中前兩個數字組成的元組。初始值是 `(1,1)` 序列中的前兩個數字。


```

let seq1 =
  0 // Initial state
  |> Seq.unfold (fun state ->
    if (state > 20) then
      None
    else
      Some(state, state + 1))

printfn "The sequence seq1 contains numbers from 0 to 20."

for x in seq1 do
  printf "%d " x

let fib =
  (1, 1) // Initial state
  |> Seq.unfold (fun state ->
    if (snd state > 1000) then
      None
    else
      Some(fst state + snd state, (snd state, fst state + snd state)))

printfn "\nThe sequence fib contains Fibonacci numbers."
for x in fib do printf "%d " x

```

輸出如下所示：

```

The sequence seq1 contains numbers from 0 to 20.

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The sequence fib contains Fibonacci numbers.

2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

```

下列程式碼範例會使用此處所述的許多序列模組函式來產生和計算無限序列的值。程式碼可能需要幾分鐘的時間來執行。

```

// generateInfiniteSequence generates sequences of floating point
// numbers. The sequences generated are computed from the fDenominator
// function, which has the type (int -> float) and computes the
// denominator of each term in the sequence from the index of that
// term. The isAlternating parameter is true if the sequence has
// alternating signs.
let generateInfiniteSequence fDenominator isAlternating =
    if (isAlternating) then
        Seq.initInfinite (fun index ->
            1.0 / (fDenominator index) * (if (index % 2 = 0) then -1.0 else 1.0))
    else
        Seq.initInfinite (fun index -> 1.0 / (fDenominator index))

// The harmonic alternating series is like the harmonic series
// except that it has alternating signs.
let harmonicAlternatingSeries = generateInfiniteSequence (fun index -> float index) true

// This is the series of reciprocals of the odd numbers.
let oddNumberSeries = generateInfiniteSequence (fun index -> float (2 * index - 1)) true

// This is the series of reciprocals of the squares.
let squaresSeries = generateInfiniteSequence (fun index -> float (index * index)) false

// This function sums a sequence, up to the specified number of terms.
let sumSeq length sequence =
    (0, 0.0)
    |>
    Seq.unfold (fun state ->
        let subtotal = snd state + Seq.item (fst state + 1) sequence
        if (fst state >= length) then
            None
        else
            Some(subtotal, (fst state + 1, subtotal)))

// This function sums an infinite sequence up to a given value
// for the difference (epsilon) between subsequent terms,
// up to a maximum number of terms, whichever is reached first.
let infiniteSum infiniteSeq epsilon maxIteration =
    infiniteSeq
    |> sumSeq maxIteration
    |> Seq.pairwise
    |> Seq.takeWhile (fun elem -> abs (snd elem - fst elem) > epsilon)
    |> List.ofSeq
    |> List.rev
    |> List.head
    |> snd

// Compute the sums for three sequences that converge, and compare
// the sums to the expected theoretical values.
let result1 = infiniteSum harmonicAlternatingSeries 0.00001 100000
printfn "Result: %f ln2: %f" result1 (log 2.0)

let pi = Math.PI
let result2 = infiniteSum oddNumberSeries 0.00001 10000
printfn "Result: %f pi/4: %f" result2 (pi/4.0)

// Because this is not an alternating series, a much smaller epsilon
// value and more terms are needed to obtain an accurate result.
let result3 = infiniteSum squaresSeries 0.0000001 1000000
printfn "Result: %f pi*pi/6: %f" result3 (pi*pi/6.0)

```

搜尋和尋找元素

序列支援清單的可用功能: [Seq.exists](#)、[array.exists2](#)、[seq.find](#)、[findIndex](#)、[Seq.pick](#)、[tryFind](#)和 [array.tryfindindex](#)。這些可用於序列的函式版本, 只會評估序列到所搜尋的專案。如需範例, 請參閱 [清單](#)。

取得個子序列

`Seq.filter` 和 `seq.choose` 如同可用於清單的對應函式，不同的是，在評估 sequence 元素之前，不會發生篩選和選擇。

`Seq` 會從另一個序列建立序列，但會將序列限制為指定的元素數目。`Seq.take` 會建立新的序列，其中只包含序列開頭的指定元素數目。如果序列中的專案數少於您所指定的時間，則會擲回 `Seq.take` `System.InvalidOperationException`。和之間的差異在於，`Seq.take` `Seq.truncate` `Seq.truncate` 如果元素數目少於您指定的數目，則不會產生錯誤。

下列程式碼顯示和之間的行為和差異 `Seq.truncate` `Seq.take`。

```
let mySeq = seq { for i in 1 .. 10 -> i*i }
let truncatedSeq = Seq.truncate 5 mySeq
let takenSeq = Seq.take 5 mySeq

let truncatedSeq2 = Seq.truncate 20 mySeq
let takenSeq2 = Seq.take 20 mySeq

let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""

// Up to this point, the sequences are not evaluated.
// The following code causes the sequences to be evaluated.
truncatedSeq |> printSeq
truncatedSeq2 |> printSeq
takenSeq |> printSeq
// The following line produces a run-time error (in printSeq):
takenSeq2 |> printSeq
```

發生錯誤之前的輸出會如下所示。

```
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
1 4 9 16 25
1 4 9 16 25 36 49 64 81 100
```

藉由使用 `takeWhile`，您可以將述詞函式指定（布耳函數），然後根據述詞為原始序列的這些元素所組成的另一個序列建立序列，但在述詞傳回 `true` 的第一個專案之前停止 `false`。`Seq.skip` 會傳回序列，以略過另一個序列的第一個專案，並傳回其餘專案的指定數目。`SkipWhile` 會傳回一個序列，只要述詞傳回，就會略過另一個序列的第一個元素 `true`，然後傳回其餘的元素，從述詞傳回的第一個專案開始 `false`。

下列程式碼範例說明、和之間的行為和 `Seq.takeWhile` 差異 `Seq.skip` `Seq.skipWhile`。

```
// takeWhile
let mySeqLessThan10 = Seq.takeWhile (fun elem -> elem < 10) mySeq
mySeqLessThan10 |> printSeq

// skip
let mySeqSkipFirst5 = Seq.skip 5 mySeq
mySeqSkipFirst5 |> printSeq

// skipWhile
let mySeqSkipWhileLessThan10 = Seq.skipWhile (fun elem -> elem < 10) mySeq
mySeqSkipWhileLessThan10 |> printSeq
```

輸出如下。

```
1 4 9
36 49 64 81 100
16 25 36 49 64 81 100
```

轉換序列

`Seq` 會建立新的序列，以將輸入序列的後續元素分組到元組中。

```
let printSeq seq1 = Seq.iter (printf "%A ") seq1; printfn ""
let seqPairwise = Seq.pairwise (seq { for i in 1 .. 10 -> i*i })
printSeq seqPairwise

printfn ""
let seqDelta = Seq.map (fun elem -> snd elem - fst elem) seqPairwise
printSeq seqDelta
```

`Seq` 就像這樣，不同的是，`Seq.pairwise` 除了產生一系列的元組之外，它也會產生一連串陣列，其中包含從序列（視窗）的相鄰元素複本。您可以指定每個陣列中所需的相鄰元素數目。

下列程式碼範例示範 `Seq.windowed` 的用法。在此情況下，視窗中的元素數目為3。此範例會使用在 `printSeq` 上的一個程式碼範例中定義的。

```
let seqNumbers = [ 1.0; 1.5; 2.0; 1.5; 1.0; 1.5 ] :> seq<float>
let seqWindows = Seq.windowed 3 seqNumbers
let seqMovingAverage = Seq.map Array.average seqWindows
printfn "Initial sequence: "
printSeq seqNumbers
printfn "\nWindows of length 3: "
printSeq seqWindows
printfn "\nMoving average: "
printSeq seqMovingAverage
```

輸出如下。

初始順序：

```
1.0 1.5 2.0 1.5 1.0 1.5

Windows of length 3:
[|1.0; 1.5; 2.0|] [|1.5; 2.0; 1.5|] [|2.0; 1.5; 1.0|] [|1.5; 1.0; 1.5|]

Moving average:
1.5 1.666666667 1.5 1.333333333
```

具有多個序列的作業

`Seq.zip` 和 `Seq.zip3` 採用兩個或三個序列，並產生一系列的元組。這些函式就像是適用於 [清單](#) 的對應函數。沒有對應的功能可將一個序列分隔為兩個或多個序列。如果您需要順序的此功能，請將序列轉換為清單，並使用 [[解壓縮](#)]。

排序、比較和分組

清單支援的排序函式也適用於序列。這包括 `Seq.sort` 和 `seq.sortBy`。這些函數會逐一查看整個序列。

您可以使用 `seq.comparewith` 函數來比較兩個序列。函式會輪流比較後續的專案，並在遇到第一個不相等的配對時停止。任何額外的元素都不會影響到比較。

下列程式碼示範 `Seq.compareWith` 的用法。

```
let sequence1 = seq { 1 .. 10 }
let sequence2 = seq { 10 .. -1 .. 1 }

// Compare two sequences element by element.
let compareSequences =
    Seq.compareWith (fun elem1 elem2 ->
        if elem1 > elem2 then 1
        elif elem1 < elem2 then -1
        else 0)

let compareResult1 = compareSequences sequence1 sequence2
match compareResult1 with
| 1 -> printfn "Sequence1 is greater than sequence2."
| -1 -> printfn "Sequence1 is less than sequence2."
| 0 -> printfn "Sequence1 is equal to sequence2."
| _ -> failwith("Invalid comparison result.")
```

在先前的程式碼中，只會計算和檢查第一個元素，而結果為-1。

`Seq.countBy` 所採用的函式會為每個專案產生一個稱為索引 鍵 的值。藉由在每個專案上呼叫此函式來產生每個專案的索引鍵。`Seq.countBy` 然後傳回包含索引鍵值的序列，以及產生每個索引鍵值的元素數目計數。

```
let mySeq1 = seq { 1.. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let seqResult =
    mySeq1
    |> Seq.countBy (fun elem ->
        if elem % 3 = 0 then 0
        elif elem % 3 = 1 then 1
        else 2)

printSeq seqResult
```

輸出如下。

```
(1, 34) (2, 33) (0, 33)
```

先前的輸出顯示，產生金鑰1、33值產生金鑰2的原始序列有34個元素，以及產生金鑰0的33值。

您可以藉由呼叫 `Seq`，來群組序列的元素。`Seq.groupBy` 接受從專案產生索引鍵的序列和函式。函數會在序列的每個專案上執行。`Seq.groupBy` 傳回一系列的元組，其中每個元組的第一個元素是索引鍵，而第二個專案是產生該索引鍵的元素序列。

下列程式碼範例示範 `Seq.groupBy` 如何使用將數位序列從1到100分割成具有相異索引鍵值0、1和2的三個群組。

```
let sequence = seq { 1 .. 100 }

let printSeq seq1 = Seq.iter (printf "%A ") seq1

let sequences3 =
    sequences
    |> Seq.groupBy (fun index ->
        if (index % 3 = 0) then 0
        elif (index % 3 = 1) then 1
        else 2)

sequences3 |> printSeq
```

輸出如下。

```
(1, seq [1; 4; 7; 10; ...]) (2, seq [2; 5; 8; 11; ...]) (0, seq [3; 6; 9; 12; ...])
```

您可以藉由呼叫 [Seq.distinct](#) 來建立可消除重複專案的序列。或者，您可以使用 [Seq.seq.distinctby](#)，它會接受在每個專案上呼叫按鍵產生函數。產生的序列包含原始序列中具有唯一索引鍵的元素；稍後會捨棄產生重複索引鍵的元素給先前的元素。

下列程式碼範例說明如何使用 `Seq.distinct`。 `Seq.distinct` 是藉由產生代表二進位數的序列來示範，然後顯示唯一的相異元素為0和1。

```
let binary n =
    let rec generateBinary n =
        if (n / 2 = 0) then [n]
        else (n % 2) :: generateBinary (n / 2)

    generateBinary n
    |> List.rev
    |> Seq.ofList

printfn "%A" (binary 1024)

let resultSequence = Seq.distinct (binary 1024)
printfn "%A" resultSequence
```

下列程式碼示範如何 `Seq.distinctBy` 從包含負數和正數的序列開始，並使用絕對值函式作為索引鍵產生函數。產生的序列遺漏了對應到序列中之負數的所有正數，因為負數會出現在序列中，因此會被選取，而不是具有相同絕對值或索引鍵的正數。

```
let inputSequence = { -5 .. 10 }
let printSeq seq1 = Seq.iter (printf "%A ") seq1

printfn "Original sequence: "
printSeq inputSequence

printfn "\nSequence with distinct absolute values: "
let seqDistinctAbsoluteValue = Seq.distinctBy (fun elem -> abs elem) inputSequence
printSeq seqDistinctAbsoluteValue
```

Readonly 和快取順序

[Seq.readonly](#) 會建立序列的唯讀複本。 `Seq.readonly` 當您有讀寫集合（例如陣列），而且您不想要修改原創組合時，會很有用。此函數可以用來保留資料封裝。在下列程式碼範例中，會建立包含陣列的型別。屬性會公開陣列，而不是傳回陣列，而是使用來傳回從陣列建立的序列 `Seq.readonly`。

```

type ArrayContainer(start, finish) =
    let internalArray = [| start .. finish |]
    member this.RangeSeq = Seq.readonly internalArray
    member this.RangeArray = internalArray

let newArray = new ArrayContainer(1, 10)
let rangeSeq = newArray.RangeSeq
let rangeArray = newArray.RangeArray
// These lines produce an error:
//let myArray = rangeSeq :> int array
//myArray.[0] <- 0
// The following line does not produce an error.
// It does not preserve encapsulation.
rangeArray.[0] <- 0

```

[Seq.cache](#) 會建立序列的儲存版本。使用 `Seq.cache` 可避免重新評估序列，或當您有多個使用序列的執行緒時，但您必須確定每個專案只會在一次時採取動作。當您有多個執行緒正在使用的序列時，您可以有一個執行緒來列舉和計算原始序列的值，而其餘的執行緒可以使用快取的序列。

在序列上執行計算

簡單的算數運算和清單類似，例如 [seq.average](#)、[seq.sum](#)、[averageBy](#)、[seq.sumBy](#) 等等。

[Seq](#)、[seq.減低](#) 和 [seq.scan](#) 就像是可用於清單的對應函式。序列支援這些函式的完整變化子集，這些功能會列出支援。如需詳細資訊和範例，請參閱 [清單](#)。

另請參閱

- [F# 語言參考](#)
- [F# 類型](#)

配量

2021/3/5 • [Edit Online](#)

本文說明如何從現有的 F # 類型取得配量，以及如何定義您自己的配量。

在 F # 中，配量是任何資料類型的子集。配量類似于 [索引子](#)，但會產生多個值，而不是從基礎資料結構產生單一值。配量使用 `..` 運算子語法來選取資料類型中指定之索引的範圍。如需詳細資訊，請參閱 [迴圈運算式參考文章](#)。

F # 目前有配量字串、清單、陣列和多維度 (2D、3D、4D) 陣列的內建支援。切割最常用於 F # 陣列和清單。您可以使用類型定義中的方法，或在範圍內的 `GetSlice` [類型延伸](#)中，將配量新增至自訂資料類型。

切割 F # 清單和陣列

最常見的資料類型為 F # 清單和陣列。下列範例示範如何配量清單：

```
// Generate a list of 100 integers
let fullList = [ 1 .. 100 ]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullList.[1..5]
printfn $"Small slice: {smallSlice}"

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullList[..5]
printfn $"Unbounded beginning slice: {unboundedBeginning}"

// Create a slice from an index to the end of the list
let unboundedEnd = fullList.[94..]
printfn $"Unbounded end slice: {unboundedEnd}"
```

配量陣列就像切割清單一樣：

```
// Generate an array of 100 integers
let fullArray = [| 1 .. 100 |]

// Create a slice from indices 1-5 (inclusive)
let smallSlice = fullArray.[1..5]
printfn $"Small slice: {smallSlice}"

// Create a slice from the beginning to index 5 (inclusive)
let unboundedBeginning = fullArray[..5]
printfn $"Unbounded beginning slice: {unboundedBeginning}"

// Create a slice from an index to the end of the list
let unboundedEnd = fullArray.[94..]
printfn $"Unbounded end slice: {unboundedEnd}"
```

配量多維陣列

F # 支援 F # 核心程式庫中的多維度陣列。如同一維陣列，多維陣列的配量也很有用。不過，引入額外的維度會規定稍微不同的語法，讓您可以取得特定資料列和資料行的片段。

下列範例示範如何配量 2D 陣列：


```
// Generate a 3x3 2D matrix
let A = array2D [[1;2;3];[4;5;6];[7;8;9]]
printfn $"Full matrix:\n {A}"

// Take the first row
let row0 = A.[0,*]
printfn $"{row0}"

// Take the first column
let col0 = A.[*,0]
printfn $"{col0}"

// Take all rows but only two columns
let subA = A.[*,0..1]
printfn $"{subA}"

// Take two rows and all columns
let subA' = A.[0..1,*]
printfn $"{subA}"

// Slice a 2x2 matrix out of the full 3x3 matrix
let twoByTwo = A.[0..1,0..1]
printfn $"{twoByTwo}"
```

定義其他資料結構的配量

F# 核心程式庫會定義一組有限類型的配量。如果您想要定義更多資料類型的配量，可以在類型定義本身或類型延伸中進行。

例如，以下是您可能為類別定義配量 `ArraySegment<T>` 以允許方便資料操作的方式：

```
open System

type ArraySegment<'TItem> with
    member segment.GetSlice(start, finish) =
        let start = defaultArg start 0
        let finish = defaultArg finish segment.Count
        ArraySegment(segment.Array, segment.Offset + start, finish - start)

let arr = ArraySegment [| 1 .. 10 |]
let slice = arr.[2..5] //[ 3; 4; 5]
```

使用和類型的另一個範例 `Span<T>` `ReadOnlySpan<T>`：

```
open System

type ReadOnlySpan<'T> with
    member sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

type Span<'T> with
    member sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

let printSpan (sp: Span<int>) =
    let arr = sp.ToArray()
    printfn $"{arr}"

let sp = [| 1; 2; 3; 4; 5 |].AsSpan()
printSpan sp.[0..] // [|1; 2; 3; 4; 5|]
printSpan sp[..5] // [|1; 2; 3; 4; 5|]
printSpan sp.[0..3] // [|1; 2; 3|]
printSpan sp.[1..3] // |2; 3|]
```

內建 F # 配量為結尾(含)

F # 中的所有內建配量都是結尾(含);亦即, 上限會包含在配量中。若為具有起始索引和結束索引的指定配量 `x` `y`, 則產生的配量會包含 *y*th 值。

```
// Define a new list
let xs = [1 .. 10]

printfn $"{xs.[2..5]}" // Includes the 5th index
```

內建 F # 空白配量

如果語法可能會產生不存在的配量, 則 F # 清單、陣列、序列、字串、多維度 (2D、3D、4D) 陣列都將會產生空的磁區。

請考慮下列範例:

```
let l = [ 1..10 ]
let a = [| 1..10 |]
let s = "hello!"

let emptyList = l.[-2..(-1)]
let emptyArray = a.[-2..(-1)]
let emptyString = s.[-2..(-1)]
```

IMPORTANT

C # 開發人員可能會預期這些會擲回例外狀況, 而不是產生空白的配量。這是以 F # 撰寫空白集合的事實設計決策。空白的 F # 清單可以使用另一個 F # 清單來撰寫, 空字串可以加入至現有的字串等等。使用以參數形式傳遞的值來取得配量是很常見的, 並藉由產生空的集合來滿足 F # 程式碼的複合本質, 進而容忍超出範圍 > 的能力。

固定-3D 和4D 陣列的索引配量

針對 F # 3D 和4D 陣列，您可以「修正」特定的索引，並將已修正該索引的其他維度進行配量。

為了說明這一點，請考慮下列3D 陣列：

z = 0

X\Y	0	1
0	0	1
1	2	3

z = 1

X\Y	0	1
0	4	5
1	6	7

如果您想要從陣列中將配量解壓縮 `[| 4; 5 |]`，請使用固定索引配量。

```
let dim = 2
let m = Array3D.zeroCreate<int> dim dim dim

let mutable count = 0

for z in 0..dim-1 do
    for y in 0..dim-1 do
        for x in 0..dim-1 do
            m.[x,y,z] <- count
            count <- count + 1

// Now let's get the [4;5] slice!
m.[*, 0, 1]
```

最後一行 `y` `z` 會修正3d 陣列的和索引，並採用 `x` 對應至矩陣的其餘值。

另請參閱

- [索引屬性](#)

選項。

2020/11/2 • [Edit Online](#)

當命名值或變數的實際值可能不存在時，就會使用 F # 中的選項類型。選項具有基礎類型，而且可以保存該類型的值，或可能沒有值。

備註

下列程式碼說明產生選項類型的函式。

```
let keepIfPositive (a : int) = if a > 0 then Some(a) else None
```

您可以看到，如果輸入 `a` 大於0，`Some(a)` 就會產生。否則，`None` 就會產生。

`None` 當某個選項沒有實際值時，就會使用此值。否則，運算式會 `Some(...)` 為選項提供值。值 `Some` 和在 `None` 模式比對中很有用，如下列函式所示 `exists`，`true` 如果選項有值，則傳回，`false` 如果沒有，則會傳回。

```
let exists (x : int option) =  
    match x with  
    | Some(x) -> true  
    | None -> false
```

使用選項

當搜尋不會傳回相符的結果時，通常會使用選項，如下列程式碼所示。

```
let rec tryFindMatch pred list =  
    match list with  
    | head :: tail -> if pred(head)  
                        then Some(head)  
                        else tryFindMatch pred tail  
    | [] -> None  
  
// result1 is Some 100 and its type is int option.  
let result1 = tryFindMatch (fun elem -> elem = 100) [ 200; 100; 50; 25 ]  
  
// result2 is None and its type is int option.  
let result2 = tryFindMatch (fun elem -> elem = 26) [ 200; 100; 50; 25 ]
```

在先前的程式碼中，會以遞迴方式搜尋清單。函式 `tryFindMatch` 會採用會傳回布林值的述詞函式 `pred`，以及要搜尋的清單。如果找到滿足述詞的元素，遞迴就會結束，而且函數會傳回值做為運算式中的選項 `Some(head)`。當空白清單相符時，遞迴就會結束。在該時間點 `head`，找不到值，而且 `None` 會傳回。

許多 F # 程式庫函式會在集合中搜尋可能存在或可能不存在的值，以傳回 `option` 型別。依照慣例，這些函數會以前置詞開頭 `try`，例如 [Seq.tryFindIndex](#)。

當值可能不存在時，選項可能也很有用，例如，當您嘗試建立值時，可能會擲回例外狀況。下列程式碼範例會說明這點。

```
open System.IO
let openFile filename =
    try
        let file = File.Open (filename, FileMode.Create)
        Some(file)
    with
        | ex -> eprintf "An exception occurred with message %s" ex.Message
        None
```

`openFile` 上一個範例中的函式有型別，`string -> File option` 因為 `File` 如果成功開啟檔案，且發生例外狀況，則會傳回物件 `None`。根據情況而定，它可能不是攔截例外狀況的適當設計選項，而不是允許它傳播。

此外，您仍然可以傳遞 `null` 或值為 `null` 的值，以作為 `Some` 選項的大小寫。這通常是要避免的，通常是在一般 F# 程式設計中，但由於 .NET 中的參考型別本質的緣故，因此可能會發生這種情況。

選項屬性和方法

選項類型支援下列屬性和方法。

名稱	型別	說明
<code>None</code>	<code>'T option</code>	靜態屬性，可讓您建立具有值的選項值 <code>None</code> 。
<code>IsNone</code>	<code>bool</code>	<code>true</code> 如果選項具有值，則傳回 <code>None</code> 。
<code>IsSome</code>	<code>bool</code>	<code>true</code> 如果選項的值不是，則會傳回 <code>None</code> 。
一些	<code>'T option</code>	靜態成員，它會建立一個值不是的選項 <code>None</code> 。
<code>ReplTest1</code>	<code>'T</code>	傳回基礎值， <code>System.NullReferenceException</code> 如果值為，則會擲回 <code>None</code> 。

選項模組

有一個模組 [選項](#)，它包含對選項執行作業的實用函式。某些函式會重複屬性的功能，但在需要函式的內容中很有用。`IsSome` 和 `isNone` 都是可測試選項是否保留值的模組函數。[選項.get](#) 會取得值(如果有的話)。如果沒有任何值，則會擲回 `System.ArgumentException`。

如果有值，則 `bind` 函式會在值上執行函數。函數必須正好採用一個引數，而且其參數類型必須是選項類型。函數的傳回值是另一個選項類型。

選項模組也包含對應至可供清單、陣列、序列和其他集合類型使用之函數的函式。這些函數包括 `Option.map`、`Option.iter`、`Option.forall`、`Option.exists`、`Option.foldBack`、`Option.fold` 和 `Option.count`。這些函數可讓您使用選項，就像是零或一個專案的集合。如需詳細資訊和範例，請參閱 [清單](#) 中的集合函數討論。

轉換成其他類型

選項可以轉換成清單或陣列。當某個選項轉換成其中一個資料結構時，產生的資料結構會有零個或一個元素。若要將選項轉換成陣列，請使用 `Option.toArray`。若要將選項轉換成清單，請使用 `Option.toList`。

另請參閱

- [F # 語言參考](#)
- [F# 類型](#)

值的選項

2019/12/6 • [Edit Online](#)

當下列兩種情況F#成立時，會使用值選項輸入：

1. 案例適用於 [F#選項](#)。
2. 使用結構可在您的案例中提供效能優勢。

並非所有的效能相關案例都是使用結構來「解決」。使用時，您必須考慮複製的額外成本，而不是參考型別。不過，大型F#程式通常會具現化許多可流經最忙碌路徑的選擇性類型，在這種情況下，結構通常可以在程式的存留期內產生較佳的整體效能。

定義

Value 選項會定義為[結構的區分](#)等位，與參考選項類型類似。其定義可以透過這種方式來考慮：

```
[<StructuralEquality; StructuralComparison>]
[<Struct>]
type ValueOption<'T> =
    | ValueNone
    | ValueSome of 'T
```

Value 選項符合結構相等和比較。主要差異在於編譯的名稱、型別名稱和大小寫名稱全都表示它是實值型別。

使用值選項

值選項的使用方式就像[選項](#)一樣。`ValueSome` 可用來表示值存在，而且當值不存在時，會使用 `ValueNone`：

```
let tryParseDateTime (s: string) =
    match System.DateTime.TryParse(s) with
    | (true, dt) -> ValueSome dt
    | (false, _) -> ValueNone

let possibleDateString1 = "1990-12-25"
let possibleDateString2 = "This is not a date"

let result1 = tryParseDateTime possibleDateString1
let result2 = tryParseDateTime possibleDateString2

match (result1, result2) with
| ValueSome d1, ValueSome d2 -> printfn "Both are dates!"
| ValueSome d1, ValueNone -> printfn "Only the first is a date!"
| ValueNone, ValueSome d2 -> printfn "Only the second is a date!"
| ValueNone, ValueNone -> printfn "None of them are dates!"
```

如同[選項](#)，傳回 `ValueOption` 之函式的命名慣例是在其前面加上 `try`。

值選項屬性和方法

目前有一個屬性用於值選項：`Value`。叫用此屬性時，如果沒有任何值存在，就會引發 [InvalidOperationException](#)。

值選項函數

Fsharp.core 中的 `ValueOption` 模組包含 `Option` 模組的對等功能。名稱有一些差異, 例如 `defaultValueArg` :

```
val defaultValueArg : arg:'T voption -> defaultValue:'T -> 'T
```

其作用就像 `Option` 模組中的 `defaultArg` , 但會改為操作值選項。

請參閱

- [選項](#)

結果

2021/3/5 • [Edit Online](#)

此 `Result<'T, 'TFailure>` 類型可讓您撰寫可組合的容錯程式碼。

語法

```
// The definition of Result in FSharp.Core
[<StructuralEquality; StructuralComparison>]
[<CompiledName("FSharpResult`2")>]
[<Struct>]
type Result<'T, 'TError> =
    | Ok of ResultValue:'T
    | Error of ErrorValue:'TError
```

備註

請參閱的 `Result` 內建組合器模組 `Result` 類型。

請注意，結果型別是 [結構](#) 差異聯集。結構相等語義適用於此處。

此 `Result` 類型通常用於 monadic 錯誤處理，在 F # 社區中通常稱為 [鐵路導向程式設計](#)。下列簡單的範例示範這種方法。

```
// Define a simple type which has fields that can be validated
type Request =
    { Name: string
      Email: string }

// Define some logic for what defines a valid name.
//
// Generates a Result which is an Ok if the name validates;
// otherwise, it generates a Result which is an Error.
let validateName req =
    match req.Name with
    | null -> Error "No name found."
    | "" -> Error "Name is empty."
    | "bananas" -> Error "Bananas is not a name."
    | _ -> Ok req

// Similarly, define some email validation logic.
let validateEmail req =
    match req.Email with
    | null -> Error "No email found."
    | "" -> Error "Email is empty."
    | s when s.EndsWith("bananas.com") -> Error "No email from bananas.com is allowed."
    | _ -> Ok req

let validateRequest reqResult =
    reqResult
    |> Result.bind validateName
    |> Result.bind validateEmail

let test() =
    // Now, create a Request and pattern match on the result.
    let req1 = { Name = "Phillip"; Email = "phillip@contoso.biz" }
    let res1 = validateRequest (Ok req1)
    match res1 with
    | Ok req -> printfn $"My request was valid! Name: {req.Name} Email {req.Email}"
    | Error e -> printfn $"Error: {e}"
    // Prints: "My request was valid! Name: Phillip Email: phillip@contoso.biz"

    let req2 = { Name = "Phillip"; Email = "phillip@bananas.com" }
    let res2 = validateRequest (Ok req2)
    match res2 with
    | Ok req -> printfn $"My request was valid! Name: {req.Name} Email {req.Email}"
    | Error e -> printfn $"Error: {e}"
    // Prints: "Error: No email from bananas.com is allowed."

test()
```

如您所見，如果您強制將各種驗證函式全部傳回，就很容易將它們連結在一起 `Result`。這可讓您將像這樣的功能分解成可組合的小部分，就像您需要的一樣。這也具有在往返驗證的結尾 **強制** 使用 **模式** 比對的額外值，進而強制執行較高程度的程式正確性。

另請參閱

- [已區分的聯集](#)
- [模式比對](#)

泛型

2019/10/23 • [Edit Online](#)

F# 函式值、方法、屬性和彙總類型 (例如類別、記錄和差別聯集) 都可以是「泛型」。泛型建構至少包含一個類型參數，此參數通常是由泛型建構的使用者所提供。泛型函式和類型可讓您撰寫適用於各種類型的程式碼，而不需對每一種類型重複輸入程式碼。由於編譯器的型別推斷和自動一般化機制通常會將程式碼隱含推斷為泛型，所以要讓程式碼在 F# 中成為泛型很簡單。

語法

```
// Explicitly generic function.
let function-name<type-parameters> parameter-list =
    function-body

// Explicitly generic method.
[ static ] member object-identifier.method-name<type-parameters> parameter-list [ return-type ] =
    method-body

// Explicitly generic class, record, interface, structure,
// or discriminated union.
type type-name<type-parameters> type-definition
```

備註

在函式或類型名稱之後的角括弧中，明確泛型函式或類型的宣告非常類似於非泛型函式或類型的宣告，但類型參數的規格 (和使用方式) 除外。

宣告通常為隱含泛型。若您並未完全指定每一個用於組成函式或類型之參數的類型，則編譯器會嘗試根據您撰寫的程式碼來推斷每一個參數、值和變數的類型。如需詳細資訊，請參閱[型別推斷](#)。若類型或函式的程式碼並未限制參數的類型，則函式或類型即為隱含泛型。此程序稱為「自動一般化」。自動一般化有一些限制。例如，若 F# 編譯器無法推斷泛型建構的類型，則該編譯器所回報的錯誤會參考一種稱為「值限制」的限制。在此情況下，您可能必須新增一些類型註解。如需自動一般化和值限制的詳細資訊，以及如何變更程式碼來解決問題的詳細資訊，請參閱[自動產生](#)。

在先前的語法中，*type-parameters* 是一份代表未知類型的參數清單 (以逗號分隔)，而每一個參數都是以單引號開頭，並選擇性加入條件約束子句，以進一步限制哪些類型可用於該類型參數。如需各種條件約束子句的語法以及條件約束的其他資訊，請參閱[條件約束](#)。

語法中的 *type-definition* 與非泛型型別的類型定義相同。這包含類別類型的建構函式參數、選擇性 `as` 子句、等號、記錄欄位、`inherit` 子句、差別聯集的選項、`let` 和 `do` 繫結、成員定義，以及非泛型型別定義中允許的任何其他項目。

其他語法項目與非泛型函式和類型的語法項目相同。例如，*object-identifier* 是一個代表包含物件本身的識別項。

屬性、欄位和建構函式不能比封入類型更加泛型。此外，模組中的值不可以為泛型。

隱含泛型建構

當 F# 編譯器推斷程式碼中的類型時，它會自動將任何可以成為泛型的函式自動視為泛型。若您明確指定類型 (例如參數類型)，則會防止自動一般化。

在下列程式碼範例中，`makeList` 是泛型的，即使它或它的參數皆未明確地宣告為泛型。

```
let makeList a b =  
    [a; b]
```

此函式的簽章被推斷為 `'a -> 'a -> 'a list`。請注意，這個範例中的 `a` 和 `b` 被推斷為具有相同的類型。這是因為它們一起包含在清單中，而且清單的所有項目都必須是相同的類型。

您也可以使用單引號語法，表示參數類型是泛型型別參數，而讓函式成為泛型。在下列程式碼中，`function1` 為泛型，因為它的參數已用這種方式宣告為類型參數。

```
let function1 (x: 'a) (y: 'a) =  
    printfn "%A %A" x y
```

明確泛型建構

以角括弧 (`<type-parameter>`) 明確宣告函式的類型參數，也可以讓函式變成泛型函式。以下的程式碼可說明這點。

```
let function2<'T> x y =  
    printfn "%A, %A" x y
```

使用泛型建構

當您使用泛型函式或方法時，您不一定要指定類型引數。編譯器會使用型別推斷來推斷適當的類型引數。若仍然模稜兩可，您可以在角括弧中提供類型引數，並以逗號分隔多個類型引數。

下列程式碼顯示如何使用先前章節中定義的函式。

```
// In this case, the type argument is inferred to be int.  
function1 10 20  
// In this case, the type argument is float.  
function1 10.0 20.0  
// Type arguments can be specified, but should only be specified  
// if the type parameters are declared explicitly. If specified,  
// they have an effect on type inference, so in this example,  
// a and b are inferred to have type int.  
let function3 a b =  
    // The compiler reports a warning:  
    function1<int> a b  
    // No warning.  
    function2<int> a b
```

NOTE

依照名稱參考泛型型別的方法有兩種。例如，`list<int>` 和 `int list` 就是兩種用於參考具有單一類型引數 `int` 之泛型型別 `list` 的方法。後面的形式照慣例僅適用於內建的 F# 類型，例如 `list` 和 `option`。若有多個類型引數，您通常會使用語法 `Dictionary<int, string>`，但是也可以使用語法 `(int, string) Dictionary`。

使用萬用字元作為類型引數

若要指定應該由編譯器推斷的類型引數，您可以使用底線或萬用字元符號 (`_`)，而非使用具名的類型引數。如下列程式碼所示。

```
let printSequence (sequence1: Collections.seq<_>) =  
    Seq.iter (fun elem -> printf "%s " (elem.ToString())) sequence1
```

泛型型別和函式的條件約束

在泛型型別或函式定義中，您只能使用已知可用於泛型型別參數的建構。如此一來，才能在編譯階段啟用函式和方法呼叫的驗證。若您明確宣告類型參數，則可以將明確條件約束套用至泛型型別參數，以通知編譯器可以使用某些方法和函式。不過，若您允許 F# 編譯器推斷您的泛型參數類型，則它會為您判斷適當的條件約束。如需詳細資訊，請參閱[條件約束](#)。

以統計方式解析的型別參數

F# 程式中可以使用的類型參數有兩種。第一種是前幾節中所描述的該類泛型型別參數。這一種類型參數等同於在 Visual Basic 和 C# 等語言中使用的泛型型別參數。另一種類型參數則專屬於 F#，稱為「以統計方式解析的類型參數」。如需這類建構的資訊，請參閱[以統計方式解析的類型參數](#)。

範例

```
// A generic function.  
// In this example, the generic type parameter 'a' makes function3 generic.  
let function3 (x : 'a) (y : 'a) =  
    printf "%A %A" x y  
  
// A generic record, with the type parameter in angle brackets.  
type GR<'a> =  
    {  
        Field1: 'a;  
        Field2: 'a;  
    }  
  
// A generic class.  
type C<'a>(a : 'a, b : 'a) =  
    let z = a  
    let y = b  
    member this.GenericMethod(x : 'a) =  
        printfn "%A %A %A" x y z  
  
// A generic discriminated union.  
type U<'a> =  
    | Choice1 of 'a  
    | Choice2 of 'a * 'a  
  
type Test() =  
    // A generic member  
    member this.Function1<'a>(x, y) =  
        printfn "%A, %A" x y  
  
    // A generic abstract method.  
    abstract abstractMethod<'a, 'b> : 'a * 'b -> unit  
    override this.abstractMethod<'a, 'b>(x:'a, y:'b) =  
        printfn "%A, %A" x y
```

另請參閱

- [語言參考](#)
- [型別](#)
- [以統計方式解析的類型參數](#)
- [泛型](#)

- 自動一般化
- 條件約束

自動一般化

2019/10/23 • [Edit Online](#)

F#使用型別推斷來評估函數和運算式的類型。本主題描述如何F#自動一般化引數和類型的函式, 以便在可能的情況下使用多個類型。

自動一般化

當F#編譯器在函式上執行型別推斷時, 會判斷指定的參數是否可以為泛型。編譯器會檢查每個參數, 並判斷函數是否相依於該參數的特定類型。如果不是, 則會將型別推斷為泛型。

下列程式碼範例說明編譯器推斷為泛型的函式。

```
let max a b = if a > b then a else b
```

類型推斷為 `'a -> 'a -> 'a`。

類型指出這是一個函式, 該函式接受相同未知類型的兩個引數, 並傳回該相同類型的值。先前的函式可以是泛型的其中一個原因是, 大於運算子 (`>`) 本身就是泛型。大於運算子具有 `'a -> 'a -> bool` 簽章。並非所有運算子都是泛型的, 而且如果函式中的程式碼使用參數類型搭配非泛型函數或運算子, 則該參數類型無法一般化。

因為 `max` 是泛型, 所以可以搭配 `int`、`float` 等類型使用, 如下列範例所示。

```
let biggestFloat = max 2.0 3.0
let biggestInt = max 2 3
```

不過, 這兩個引數必須屬於相同的類型。簽章為 `'a -> 'a -> 'a`, 而 `'a -> 'b -> 'a` 非。因此, 下列程式碼會產生錯誤, 因為類型不相符。

```
// Error: type mismatch.
let biggestIntFloat = max 2.0 3
```

`max` 函數也適用於任何支援大於運算子的類型。因此, 您也可以字串上使用它, 如下列程式碼所示。

```
let testString = max "cab" "cat"
```

值限制

編譯器只會在具有明確引數的完整函式定義和簡單的不可變值上執行自動一般化。

這表示編譯器會在您嘗試編譯的程式碼未充分限制為特定類型, 但也無法歸納時發出錯誤。此問題的錯誤訊息指的是對值進行自動一般化的限制, 做為 *值限制*。

一般而言, 當您想要將結構設為泛型, 但編譯器沒有足夠的資訊來將它一般化, 或當您不小心在非泛型結構中省略足夠的類型資訊時, 就會發生值限制錯誤。「值限制」錯誤的解決方案, 是以下列其中一種方式提供更明確的資訊, 以更完整地限制型別推斷問題:

- 將明確的類型注釋新增至值或參數, 以將類型限制為非泛型。
- 如果問題是使用 `nongeneralizable` 結構來定義泛型函式, 例如函式組合或未完全套用的擴充函式引數, 請

嘗試重寫函數做為一般函式定義。

- 如果問題是太複雜而無法一般化的運算式, 請新增額外的未使用參數, 使其進入函式。
- 加入明確的泛型型別參數。此選項很少使用。
- 下列程式碼範例將說明每個案例。

案例 1:運算式太複雜。在此範例中, 清單 `counter` 的目的 `int option ref` 是, 但未定義為簡單的不可變值。

```
let counter = ref None
// Adding a type annotation fixes the problem:
let counter : int option ref = ref None
```

案例 2:使用 `nongeneralizable` 結構來定義泛型函數。在此範例中, 結構是 `nongeneralizable` 的, 因為它牽涉到部分應用函式引數。

```
let maxhash = max << hash
// The following is acceptable because the argument for maxhash is explicit:
let maxhash obj = (max << hash) obj
```

案例 3:加入額外、未使用的參數。因為此運算式不夠簡單, 無法進行一般化, 所以編譯器會發出值限制錯誤。

```
let emptyList10 = Array.create 10 []
// Adding an extra (unused) parameter makes it a function, which is generalizable.
let emptyList10 () = Array.create 10 []
```

案例 4:加入型別參數。

```
let arrayOf10Lists = Array.create 10 []
// Adding a type parameter and type annotation lets you write a generic value.
let arrayOf10Lists<'T> = Array.create 10 ([]:'T list)
```

在最後一個案例中, 值會變成類型函式, 可用來建立許多不同類型的值, 例如, 如下所示:

```
let intLists = arrayOf10Lists<int>
let floatLists = arrayOf10Lists<float>
```

另請參閱

- [類型推斷](#)
- [泛型](#)
- [以統計方式解析的類型參數](#)
- [條件約束](#)

條件約束

2019/11/4 • [Edit Online](#)

本主題描述可套用至泛型型別參數的條件約束，以指定泛型型別或函式中的型別引數需求。

語法

```
type-parameter-list when constraint1 [ and constraint2]
```

備註

有幾個不同的條件約束可供您套用，以限制可用於泛型型別的類型。下表列出並描述這些條件約束。

條件約束	語法	說明
類型條件約束	<i>類型參數</i> : > <i>類型</i>	提供的類型必須等於或衍生自指定的類型，或者，如果類型是介面，則提供的類型必須執行介面。
Null 條件約束	<i>類型參數</i> : null	提供的類型必須支援 null 常值。這包括所有 .NET 物件類型，但 F# 不包含清單、元組、函式、類別、記錄或聯集類型。
明確成員條件約束	[(! <i>類型參數</i> 或 <i>。。</i> 或 <i>類型參數</i>): (<i>成員簽章</i>)	提供的型別引數中至少必須有一個具有指定簽章的成員；不適用於一般用途。成員必須在類型或隱含類型延伸的一部分上明確定義，才能成為明確成員條件約束的有效目標。
構造函式條件約束	<i>類型參數</i> : (新的: unit-> 'a)	提供的類型必須具有無參數的函式。
實值型別條件約束	: 結構	提供的類型必須是 .NET 實數值型別。
參考型別條件約束	: not 結構	提供的型別必須是 .NET 引用型別。
列舉類型條件約束	: 列舉 < <i>基礎類型</i> >	提供的類型必須是具有指定基礎類型的列舉類型。不適用於一般用途。
委派條件約束	: 委派 < <i>元組-參數類型</i> 、 <i>傳回類型</i> >	提供的類型必須是具有指定的引數和傳回值的委派類型。不適用於一般用途。
比較準則約束	: 比較	提供的類型必須支援比較。
相等條件約束	: 相等	提供的類型必須支援相等。

'''	''	''
非受控條件約束	:非受控	提供的型別必須是非受控型別。非受控類型是特定的基本型別(<code>sbyte</code> 、 <code>byte</code> 、 <code>char</code> 、 <code>nativeint</code> 、 <code>unativeint</code> 、 <code>float32</code> 、 <code>float</code> 、 <code>int16</code> 、 <code>uint16</code> 、 <code>int32</code> 、 <code>uint32</code> 、 <code>int64</code> 、 <code>uint64</code> 或 <code>decimal</code>)、列舉類型、 <code>nativeptr<_></code> ，或其欄位皆為非受控類型的非泛型結構。

當您的程式碼必須使用條件約束類型(而非一般類型)上可用的功能時，您必須加入條件約束。例如，如果您使用類型條件約束來指定類別類型，您可以在泛型函數或類型中使用該類別的任何一個方法。

明確寫入型別參數時，有時需要指定條件約束，因為沒有條件約束，編譯器無法驗證您所使用的功能是否可用於該型別的執行時間可能會提供的任何型別。實參。

您在程式碼中F#使用的最常見條件約束是指定基類或介面的類型條件約束。連結F#庫會使用其他條件約束來執行特定功能，例如明確成員條件約束(用來執行算術運算子的運算子多載)，主要是因為F#支援 common language runtime 所支援的一組完整條件約束。

在型別推斷程式期間，編譯器會自動推斷某些條件約束。例如，如果您在函數中使用 `+` 運算子，則編譯器會在運算式中使用的變數類型上推斷明確成員條件約束。

下列程式碼說明一些條件約束宣告：

```

// Base Type Constraint
type Class1<'T when 'T :> System.Exception> =
class end

// Interface Type Constraint
type Class2<'T when 'T :> System.IComparable> =
class end

// Null constraint
type Class3<'T when 'T : null> =
class end

// Member constraint with instance member
type Class5<'T when 'T : (member Method1 : 'T -> int)> =
class end

// Member constraint with property
type Class6<'T when 'T : (member Property1 : int)> =
class end

// Constructor constraint
type Class7<'T when 'T : (new : unit -> 'T)>() =
member val Field = new 'T()

// Reference type constraint
type Class8<'T when 'T : not struct> =
class end

// Enumeration constraint with underlying value specified
type Class9<'T when 'T : enum<uint32>> =
class end

// 'T must implement IComparable, or be an array type with comparable
// elements, or be System.IntPtr or System.UIntPtr. Also, 'T must not have
// the NoComparison attribute.
type Class10<'T when 'T : comparison> =
class end

// 'T must support equality. This is true for any type that does not
// have the NoEquality attribute.
type Class11<'T when 'T : equality> =
class end

type Class12<'T when 'T : delegate<obj * System.EventArgs, unit>> =
class end

type Class13<'T when 'T : unmanaged> =
class end

// Member constraints with two type parameters
// Most often used with static type parameters in inline functions
let inline add(value1 : ^T when ^T : (static member (+) : ^T * ^T -> ^T), value2 : ^T) =
value1 + value2

// ^T and ^U must support operator +
let inline heterogeneousAdd(value1 : ^T when (^T or ^U) : (static member (+) : ^T * ^U -> ^T), value2 : ^U) =
value1 + value2

// If there are multiple constraints, use the and keyword to separate them.
type Class14<'T,'U when 'T : equality and 'U : equality> =
class end

```

請參閱

- [泛型](#)

- 條件約束

以統計方式解析的型別參數

2019/11/4 • [Edit Online](#)

靜態解析的類型參數是在編譯時期（而不是在執行時間）取代為實際類型的類型參數。它們前面會加上插入號（^）符號。

語法

```
^type-parameter
```

備註

在 F# 語言中，有兩種不同類型的型別參數。第一種是標準的泛型型別參數。這些會以單引號（'）表示，如同 `'T` 和 `'U`。它們相當於其他 .NET Framework 語言中的泛型型別參數。另一種方法是以靜態方式解析，並以插入號符號表示，如 `^T` 和 `^U`。

靜態解析的類型參數主要適用於結合成員條件約束，這是條件約束，可讓您指定類型引數必須有特定成員或成員，才能使用。您無法使用一般泛型型別參數來建立這種條件約束。

下表摘要說明兩種類型參數之間的相似性和差異。

語法	執行階段	編譯時間
語法	<code>'T</code> 、 <code>'U</code>	<code>^T</code> 、 <code>^U</code>
解決時間	執行階段	編譯時間
成員條件約束	不能與成員條件約束搭配使用。	可以與成員條件約束搭配使用。
程式碼產生	具有標準泛型型別參數的類型（或方法）會產生單一泛型型別或方法。	會產生類型和方法的多個具現化，每個所需的類型各一個。
搭配類型使用	可以用於類型。	不能用在類型上。
搭配內嵌函數使用	否。內嵌函式不能以標準泛型型別參數參數化。	可以。靜態解析的類型參數不能用於非內嵌的函式或方法。

許多 F# 核心程式庫函式（尤其是運算子）都有靜態解析的類型參數。這些函式和運算子是內嵌的，可讓您以有效率的方式產生數值計算的程式碼。

使用運算子的內嵌方法和函式，或使用其他具有靜態解析類型參數的函式，也可以使用靜態解析的類型參數本身。通常，型別推斷會推斷這類內嵌函式，使其具有靜態解析的型別參數。下列範例說明的運算子定義，會推斷為具有靜態解析的類型參數。

```
let inline (+@) x y = x + x * y
// Call that uses int.
printfn "%d" (1 +@ 1)
// Call that uses float.
printfn "%f" (1.0 +@ 0.5)
```

已解決的 `(+@)` 類型是以 `(+)` 和 `(*)` 的使用為基礎，這兩者都會導致型別推斷在靜態解析的型別參數上推斷成員條件約束。如F#解釋器所示，已解析的類型如下。

```
^a -> ^c -> ^d
when (^a or ^b) : (static member ( + ) : ^a * ^b -> ^d) and
(^a or ^c) : (static member ( * ) : ^a * ^c -> ^b)
```

輸出如下。

```
2
1.500000
```

從F# 4.1 開始，您也可以在以靜態方式解析的類型參數簽章中指定具體的類型名稱。在舊版的語言中，編譯器實際上可以推斷類型名稱，但實際上無法在簽章中指定。從 4.1 F#，您也可以在以靜態方式解析的類型參數簽章中指定具體的類型名稱。以下為範例：

```
let inline konst x _ = x

type CFuncutor() =
    static member inline fmap (f: ^a -> ^b, a: ^a list) = List.map f a
    static member inline fmap (f: ^a -> ^b, a: ^a option) =
        match a with
        | None -> None
        | Some x -> Some (f x)

    // default implementation of replace
    static member inline replace< ^a, ^b, ^c, ^d, ^e when ^a :> CFuncutor and (^a or ^d): (static member
fmap: (^b -> ^c) * ^d -> ^e) > (a, f) =
        ((^a or ^d) : (static member fmap : (^b -> ^c) * ^d -> ^e) (konst a, f))

    // call overridden replace if present
    static member inline replace< ^a, ^b, ^c when ^b: (static member replace: ^a * ^b -> ^c)>(a: ^a, f: ^b)
=
        (^b : (static member replace: ^a * ^b -> ^c) (a, f))

let inline replace_instance< ^a, ^b, ^c, ^d when (^a or ^c): (static member replace: ^b * ^c -> ^d)> (a: ^b,
f: ^c) =
    ((^a or ^c): (static member replace: ^b * ^c -> ^d) (a, f))

// Note the concrete type 'CFuncutor' specified in the signature
let inline replace (a: ^a) (f: ^b): ^a0 when (CFuncutor or ^b): (static member replace: ^a * ^b -> ^a0) =
    replace_instance<CFuncutor, _, _, _> (a, f)
```

請參閱

- [泛型](#)
- [類型推斷](#)
- [自動一般化](#)
- [條件約束](#)
- [內嵌函式](#)

記錄

2021/3/5 • [Edit Online](#)

記錄表示具名值的簡單彙總，並選擇性地搭配成員。可以是結構或參考型別。它們預設為參考類型。

語法

```
[ attributes ]
type [accessibility-modifier] typename =
    { [ mutable ] label1 : type1;
      [ mutable ] label2 : type2;
      ... }
[ member-list ]
```

備註

在先前的語法中，*typename* 是記錄類型的名稱，*label1* 和 *label2* 是值的名稱（稱為 **標籤**），而 *type1* 和 *type2* 是這些值的類型。*成員清單* 是類型的選擇性成員清單。您可以使用 `[<Struct>]` 屬性來建立結構記錄，而不是參考型別的記錄。

以下有一些範例。

```
// Labels are separated by semicolons when defined on the same line.
type Point = { X: float; Y: float; Z: float; }

// You can define labels on their own line with or without a semicolon.
type Customer =
    { First: string
      Last: string;
      SSN: uint32
      AccountNumber: uint32; }

// A struct record.
[<Struct>]
type StructPoint =
    { X: float
      Y: float
      Z: float }
```

當每個標籤都在不同的行上時，分號是選擇性的。

您可以在稱為 **記錄運算式** 的運算式中設定值。如果標籤與其他記錄類型) 的不同，則編譯器會從所使用的標籤推斷類型 (。括弧 ()) 括住記錄運算式。下列程式碼會顯示記錄運算式，該運算式會使用具有標籤和的三個 float 元素來初始化記錄 `x` `y` `z` 。

```
let mypoint = { X = 1.0; Y = 1.0; Z = -1.0; }
```

如果有另一種類型也有相同的標籤，請勿使用縮寫的表單。

```
type Point = { X: float; Y: float; Z: float; }
type Point3D = { X: float; Y: float; Z: float }
// Ambiguity: Point or Point3D?
let mypoint3D = { X = 1.0; Y = 1.0; Z = 0.0; }
```

最近宣告類型的標籤優先於先前宣告的型別，因此在上述範例中，`mypoint3D` 會推斷為 `Point3D`。您可以明確地指定記錄類型，如下列程式碼所示。

```
let myPoint1 = { Point.X = 1.0; Y = 1.0; Z = 0.0; }
```

您可以針對記錄類型定義方法，就像類別類型一樣。

使用記錄運算式建立記錄

您可以使用記錄中定義的標籤來初始化記錄。執行這項工作的運算式稱為「*記錄運算式*」。使用大括弧來括住記錄運算式，並使用分號做為分隔符號。

下列範例顯示如何建立記錄。

```
type MyRecord =
{ X: int
  Y: int
  Z: int }

let myRecord1 = { X = 1; Y = 2; Z = 3; }
```

在記錄運算式和類型定義中，最後一個欄位後面的分號是選擇性的，不論這些欄位是否全都在同一行。

當您建立記錄時，必須提供每個欄位的值。您無法在初始化運算式中參考任何欄位的其他欄位值。

在下列程式碼中，的型別 `myRecord2` 是從欄位名稱推斷而來。（選擇性）您可以明確地指定型別名稱。

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

當您必須複製現有的記錄，而且可能變更部分域值時，另一種形式的記錄結構可能很有用。下面這行程式碼將說明這一點。

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

這種形式的記錄運算式稱為「*複製和更新記錄運算式*」。

依預設，記錄是不可變的；不過，您可以使用複製和更新運算式，輕鬆地建立修改過的記錄。您也可以明確地指定可變欄位。

```
type Car =
{ Make : string
  Model : string
  mutable Odometer : int }

let myCar = { Make = "Fabrikam"; Model = "Coupe"; Odometer = 108112 }
myCar.Odometer <- myCar.Odometer + 21
```

請勿使用 `DefaultValue` 屬性搭配記錄欄位。更好的方法是使用已初始化為預設值的欄位來定義記錄的預設實例，然後使用複製和更新記錄運算式來設定與預設值不同的任何欄位。


```
// Rather than use [<DefaultValue>], define a default record.
type MyRecord =
    { Field1 : int
      Field2 : int }

let defaultRecord1 = { Field1 = 0; Field2 = 0 }
let defaultRecord2 = { Field1 = 1; Field2 = 25 }

// Use the with keyword to populate only a few chosen fields
// and leave the rest with default values.
let rr3 = { defaultRecord1 with Field2 = 42 }
```

建立相互遞迴記錄

建立記錄時，您可能會想要讓它相依于您稍後要定義的另一種類型。這是編譯錯誤，除非您將記錄類型定義為相互遞迴。

使用關鍵字來定義相互遞迴記錄 `and`。這可讓您將2個以上的記錄類型連結在一起。

例如，下列程式碼會將 `Person` 和 `Address` 類型定義為相互遞迴：

```
// Create a Person type and use the Address type that is not defined
type Person =
    { Name: string
      Age: int
      Address: Address }
// Define the Address type which is used in the Person record
and Address =
    { Line1: string
      Line2: string
      PostCode: string
      Occupant: Person }
```

如果您要定義先前的範例而不指定 `and` 關鍵字，則不會進行編譯。`and` 相互遞迴定義必須有關鍵詞。

使用記錄進行模式比對

記錄可以搭配模式比對使用。您可以明確指定某些欄位，並提供變數給將在相符時指派的其他欄位。下列程式碼範例會說明這點。

```
type Point3D = { X: float; Y: float; Z: float }
let evaluatePoint (point: Point3D) =
    match point with
    | { X = 0.0; Y = 0.0; Z = 0.0 } -> printfn "Point is at the origin."
    | { X = xVal; Y = 0.0; Z = 0.0 } -> printfn "Point is on the x-axis. Value is %f." xVal
    | { X = 0.0; Y = yVal; Z = 0.0 } -> printfn "Point is on the y-axis. Value is %f." yVal
    | { X = 0.0; Y = 0.0; Z = zVal } -> printfn "Point is on the z-axis. Value is %f." zVal
    | { X = xVal; Y = yVal; Z = zVal } -> printfn "Point is at (%f, %f, %f)." xVal yVal zVal

evaluatePoint { X = 0.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 100.0; Y = 0.0; Z = 0.0 }
evaluatePoint { X = 10.0; Y = 0.0; Z = -1.0 }
```

此程式碼的輸出如下所示。

```
Point is at the origin.
Point is on the x-axis. Value is 100.000000.
Point is at (10.000000, 0.000000, -1.000000).
```

記錄和成員

您可以在記錄上指定成員，就像您可以使用類別一樣。不支援欄位。常見的方法是定義 `Default` 靜態成員，以便輕鬆地記錄結構：

```
type Person =
    { Name: string
      Age: int
      Address: string }

    static member Default =
        { Name = "Phillip"
          Age = 12
          Address = "123 happy fun street" }

let defaultPerson = Person.Default
```

如果您使用自我識別碼，該識別碼會參考其成員所呼叫之記錄的實例：

```
type Person =
    { Name: string
      Age: int
      Address: string }

    member this.WeirdToString() =
        this.Name + this.Address + string this.Age

let p = { Name = "a"; Age = 12; Address = "abc123" }
let weirdString = p.WeirdToString()
```

記錄與類別之間的差異

[記錄欄位] 與 [類別] 欄位不同之處在於它們會自動公開為屬性，並用於建立和複製記錄。記錄結構與類別結構也不同。在記錄類型中，您無法定義函數。相反地，本主題所述的結構語法也適用。類別在兩個函式參數、欄位和屬性之間沒有直接關聯性。

如同 `union` 和 `structure` 型別，記錄具有結構化相等的語法。類別具有參考相等的語法。下列程式碼範例示範此工作。

```
type RecordTest = { X: int; Y: int }

let record1 = { X = 1; Y = 2 }
let record2 = { X = 1; Y = 2 }

if (record1 = record2) then
    printfn "The records are equal."
else
    printfn "The records are unequal."
```

此程式碼的輸出如下所示：

```
The records are equal.
```

如果您以類別撰寫相同的程式碼，這兩個類別物件將會不相等，因為這兩個值會在堆積上代表兩個物件，而只有位址會 (比較，除非類別類型會覆寫 `System.Object.Equals`) 的方法。

如果您需要記錄的參考相等，請在 `[<ReferenceEquality>]` 記錄上方加入屬性。

請參閱

- [F# 類型](#)
- [類別](#)
- [F # 語言參考](#)
- [參考相等](#)
- [模式比對](#)

匿名記錄

2021/3/5 • [Edit Online](#)

匿名記錄是命名值的簡單匯總，不需要在使用之前宣告。您可以將它們宣告為結構或參考型別。它們預設是參考型別。

語法

下列範例示範匿名記錄語法。分隔為 `[item]` 的專案是選擇性的。

```
// Construct an anonymous record
let value-name = [struct] {| Label1: Type1; Label2: Type2; ...|}

// Use an anonymous record as a type parameter
let value-name = Type-Name<[struct] {| Label1: Type1; Label2: Type2; ...|}>

// Define a parameter with an anonymous record as input
let function-name (arg-name: [struct] {| Label1: Type1; Label2: Type2; ...|}) ...
```

基本使用方式

匿名記錄最好被視為不需要在具現化之前宣告的 F# 記錄類型。

例如，在這裡，您可以如何與產生匿名記錄的函式互動：

```
open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let r = 2.0
let stats = getCircleStats r
printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
    r stats.Diameter stats.Area stats.Circumference
```

下列範例 `printCircleStats` 會使用接受匿名記錄做為輸入的函式來展開上一個範例：

```

open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    {| Diameter = d; Area = a; Circumference = c |}

let printCircleStats r (stats: {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats

```

`printCircleStats` 使用任何匿名記錄類型呼叫，而該類型與輸入類型沒有相同的「圖形」，將無法編譯：

```

printCircleStats r {| Diameter = 2.0; Area = 4.0; MyCircumference = 12.566371 |}
// Two anonymous record types have mismatched sets of field names
// '["Area"; "Circumference"; "Diameter"]' and '["Area"; "Diameter"; "MyCircumference"]'

```

結構匿名記錄

您也可以使用選擇性關鍵字將匿名記錄定義為結構 `struct`。下列範例會藉由產生和取用結構匿名記錄來擴大前一個範例：

```

open System

let getCircleStats radius =
    let d = radius * 2.0
    let a = Math.PI * (radius ** 2.0)
    let c = 2.0 * Math.PI * radius

    // Note that the keyword comes before the '{| |}' brace pair
    struct {| Area = a; Circumference = c; Diameter = d |}

// the 'struct' keyword also comes before the '{| |}' brace pair when declaring the parameter type
let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

let r = 2.0
let stats = getCircleStats r
printCircleStats r stats

```

Structness 推斷

結構匿名記錄也允許「structness 推斷」，您不需要在呼叫位置指定 `struct` 關鍵字。在此範例中，您會在 `struct` 呼叫時省略關鍵字 `printCircleStats`：

```

let printCircleStats r (stats: struct {| Area: float; Circumference: float; Diameter: float |}) =
    printfn "Circle with radius: %f has diameter %f, area %f, and circumference %f"
        r stats.Diameter stats.Area stats.Circumference

printCircleStats r {| Area = 4.0; Circumference = 12.6; Diameter = 12.6 |}

```

反向模式-指定 `struct` 輸入類型不是結構匿名記錄時，將無法編譯。

在其他類型中內嵌匿名記錄

宣告案例為記錄的 [區分](#) 等位是很有用的。但是，如果記錄中的資料與不同聯集的類型相同，您就必須將所有類型定義為相互遞迴。使用匿名記錄可避免這種限制。以下是模式比對的範例型別和函數：

```
type FullName = { FirstName: string; LastName: string }

// Note that using a named record for Manager and Executive would require mutually recursive definitions.
type Employee =
  | Engineer of FullName
  | Manager of { | Name: FullName; Reports: Employee list | }
  | Executive of { | Name: FullName; Reports: Employee list; Assistant: Employee | }

let getFirstName e =
  match e with
  | Engineer fullName -> fullName.FirstName
  | Manager m -> m.Name.FirstName
  | Executive ex -> ex.Name.FirstName
```

複製和更新運算式

匿名記錄支援具有 [複製和更新運算式](#) 的結構。例如，以下是您可以如何建立匿名記錄的新實例，以複製現有的資料：

```
let data = { | X = 1; Y = 2 | }
let data' = { | data with Y = 3 | }
```

不過，不同于命名記錄，匿名記錄可讓您使用複製和更新運算式來建立完全不同的表單。下列範例會使用先前範例中的相同匿名記錄，並將它擴充至新的匿名記錄：

```
let data = { | X = 1; Y = 2 | }
let expandedData = { | data with Z = 3 | } // Gives { | X=1; Y=2; Z=3 | }
```

您也可以從命名記錄的實例中，建立匿名記錄：

```
type R = { X: int }
let data = { X = 1 }
let data' = { | data with Y = 2 | } // Gives { | X=1; Y=2 | }
```

您也可以將資料複製到參考和結構匿名記錄：

```
// Copy data from a reference record into a struct anonymous record
type R1 = { X: int }
let r1 = { X = 1 }

let data1 = struct { | r1 with Y = 1 | }

// Copy data from a struct record into a reference anonymous record
[<Struct>]
type R2 = { X: int }
let r2 = { X = 1 }

let data2 = { | r1 with Y = 1 | }

// Copy the reference anonymous record data into a struct anonymous record
let data3 = struct { | data2 with Z = r2.X | }
```

匿名記錄的屬性

匿名記錄有許多特性，對於完全瞭解其使用方式而言是不可或缺的。

匿名記錄為名義

匿名記錄是 [名義類型](#)。最好將它們視為命名 [記錄](#) 類型 (這也是不需要前置宣告的名義)。

請考慮下列兩個匿名記錄宣告的範例：

```
let x = { | X = 1 | }
let y = { | Y = 1 | }
```

`x` 和 `y` 值有不同的類型，且彼此不相容。它們不會 equatable，而且也不能比較。為了說明這一點，請考慮對等的命名記錄：

```
type X = { X: int }
type Y = { Y: int }

let x = { X = 1 }
let y = { Y = 1 }
```

與類型相等或比較有關的匿名記錄與其命名記錄相等時，並沒有任何固有的差異。

匿名記錄會使用結構相等和比較

就像記錄類型一樣，匿名記錄是結構 equatable 且可比較的。只有在所有組成類型都支援相等和比較 (例如記錄類型) 的情況下，才會出現此情況。若要支援相等或比較，兩個匿名記錄必須具有相同的「圖形」。

```
{ | a = 1+1 | } = { | a = 2 | } // true
{ | a = 1+1 | } > { | a = 1 | } // true

// error FS0001: Two anonymous record types have mismatched sets of field names '["a"]' and '["a"; "b"]'
{ | a = 1 + 1 | } = { | a = 2; b = 1 | }
```

匿名記錄可序列化

您可以將匿名記錄序列化，就像您可以使用命名記錄一樣。以下是使用 [Newtonsoft.Json](#) 的範例：

```
open Newtonsoft.Json

let phillip' = {| name="Phillip"; age=28 |}
let philStr = JsonConvert.SerializeObject(phillip')

let phillip = JsonConvert.DeserializeObject<{|name: string; age: int|}>(philStr)
printfn $"Name: {phillip.name} Age: %d{phillip.age}"
```

匿名記錄適用於透過網路傳送輕量資料，而不需要事先為您的序列化/還原序列化類型定義網域。

匿名記錄與 c# 匿名型別交互操作

您可以使用需要使用 [c# 匿名型別](#) 的 .net API。C# 匿名型別很容易使用匿名記錄來與相交交互操作。下列範例示範如何使用匿名記錄來呼叫需要匿名型別的 [LINQ](#) 多載：

```
open System.Linq

let names = [ "Ana"; "Felipe"; "Emilia" ]
let nameGrouping = names.Select(fun n -> {| Name = n; FirstLetter = n.[0] |})
for ng in nameGrouping do
    printfn $"{ng.Name} has first letter {ng.FirstLetter}"
```

在整個 .NET 中，有許多其他的 Api 都需要使用匿名型別來傳遞。匿名記錄是您使用它們的工具。

限制

匿名記錄對於其使用方式有一些限制。有些是其設計固有的，但有些則適合變更。

模式比對的限制

匿名記錄不支援模式比對，不同於命名記錄。有三個原因：

1. 模式必須考慮匿名記錄的每個欄位，與命名的記錄類型不同。這是因為匿名記錄不支援結構化 subtyping – 它們是名義類型。
2. 由於 (1)，因此無法在模式比對運算式中使用其他模式，因為每個不同的模式會代表不同的匿名記錄類型。
3. 由於 (3)，任何匿名記錄模式都比使用 "dot" 標記法更詳細。

有開放語言的建議，可 [讓您在有限的內容中進行模式比對](#)。

可變動性的限制

目前無法以資料定義匿名記錄 `mutable`。有開放的 [語言建議](#) 可允許可變數據。

結構匿名記錄的限制

您無法將結構匿名記錄宣告為 `IsByRefLike` 或 `IsReadOnly`。和匿名記錄有 [開放的語言建議](#) `IsByRefLike` `IsReadOnly`。

複製和更新記錄運算式

2020/4/21 • [Edit Online](#)

*複製和更新記錄表達式*是複製現有記錄、更新指定欄位並返回更新的記錄的運算式。

語法

```
{ record-name with  
  updated-labels }  
  
{| anonymous-record-name with  
  updated-labels |}
```

備註

預設情況下,記錄和匿名記錄是不可變的,因此無法更新現有記錄。要創建更新的記錄,必須再次指定記錄的所有欄位。為了簡化此工作,可以使用*複製與更新表示式*。此表示式採用現有記錄,使用運算式中的指定欄位和表示式指定的缺失欄位創建新的相同類型。

當您必須複製現有記錄並可能更改某些欄位值時,這非常有用。

例如,以新創建的記錄為例。

```
let myRecord2 = { MyRecord.X = 1; MyRecord.Y = 2; MyRecord.Z = 3 }
```

要僅更新該紀錄中的兩個字段,可以使用*複製和更新記錄表示式*。

```
let myRecord3 = { myRecord2 with Y = 100; Z = 2 }
```

另請參閱

- [記錄](#)
- [匿名記錄](#)
- [F# 語言參考](#)

已區分的聯集

2021/3/5 • [Edit Online](#)

差異聯集可支援可能是數個命名案例之一的值，可能是每個都有不同的值和類型。相異聯集適用於異質資料；可能具有特殊案例的資料，包括有效和錯誤案例；從某個實例到另一個實例的類型不同的資料；以及做為小型物件階層的替代方案。此外，遞迴差異聯集用來表示樹狀結構資料結構。

語法

```
[ attributes ]
type [accessibility-modifier] type-name =
  | case-identifier1 [of [ fieldname1 : ] type1 [ * [ fieldname2 : ] type2 ...]
  | case-identifier2 [of [fieldname3 : ]type3 [ * [ fieldname4 : ]type4 ...]

[ member-list ]
```

備註

相異聯集與其他語言的等位類型類似，但有一些差異。如同 c + + 中的等位類型或 Visual Basic 中的 variant 類型，儲存在值中的資料不是固定的；它可以是數個不同選項的其中一個。不過，不同于這些其他語言的等位，每個可能的選項都會獲得一個 **案例識別碼**。案例識別碼是適用於各種可能類型之值的名稱，這些類型的物件可能是：這些值是選擇性的。如果值不存在，則案例相當於列舉案例。如果有值，每個值都可以是指定類型的單一值，或是匯總多個相同或不同類型之欄位的元組。您可以提供個別欄位的名稱，但名稱是選擇性的，即使相同案例中的其他欄位已命名也一樣。

區分等位的協助工具預設為 `public`。

例如，請考慮下列圖形類型的宣告。

```
type Shape =
  | Rectangle of width : float * length : float
  | Circle of radius : float
  | Prism of width : float * float * height : float
```

上述程式碼會宣告差異聯集圖形，其值可以是下列三種情況之一：矩形、圓形和 Prism。每個案例都有一組不同的欄位。矩形大小寫有兩個名稱的欄位，兩者都 `float` 有名稱寬度和長度的型別。圓形案例只有一個命名欄位，半徑。Prism 案例有三個欄位，其中兩個（寬度和高度）命名為 [欄位]。未命名的欄位稱為匿名欄位。

您可以根據下列範例提供已命名和匿名欄位的值來建立物件。

```
let rect = Rectangle(length = 1.3, width = 10.0)
let circ = Circle (1.0)
let prism = Prism(5., 2.0, height = 3.0)
```

這段程式碼顯示您可以在初始化中使用已命名的欄位，也可以依賴宣告中的欄位順序，然後再依序提供每個欄位的值。先前程式碼中的函式呼叫會 `rect` 使用命名欄位，但的函式呼叫會 `circ` 使用排序。您可以混用已排序的欄位和命名欄位，就像在的結構中一樣 `prism`。

此 `option` 類型是 F # 核心程式庫中的簡單差異聯集。型別宣告 `option` 如下。

```
// The option type is a discriminated union.
type Option<'a> =
    | Some of 'a
    | None
```

先前的程式碼指定類型 `Option` 是具有兩個案例的差異聯集，`Some` 以及 `None`。 `Some` 案例具有相關聯的值，其中包含一個匿名欄位，而該欄位的型別是由型別參數表示 `'a`。 `None` 案例沒有相關聯的值。因此， `option` 型別會指定具有某個類型值或沒有任何值的泛型型別。此類型 `Option` 也有較常用的小寫類型別名 `option`。

案例識別碼可用來當做差異聯集類型的函式。例如，下列程式碼會用來建立類型的值 `option`。

```
let myOption1 = Some(10.0)
let myOption2 = Some("string")
let myOption3 = None
```

案例識別碼也會在模式比對運算式中使用。在模式比對運算式中，會提供與個別案例相關聯之值的識別碼。例如，在下列程式碼中， `x` 是提供與型別案例相關聯之值的識別碼 `Some option`。

```
let printValue opt =
    match opt with
    | Some x -> printfn "%A" x
    | None -> printfn "No value."
```

在模式比對運算式中，您可以使用指定的欄位來指定差異聯集相符專案。針對先前宣告的圖形類型，您可以使用命名欄位，如下列程式碼所示，用來將欄位的值解壓縮。

```
let getShapeWidth shape =
    match shape with
    | Rectangle(width = w) -> w
    | Circle(radius = r) -> 2. * r
    | Prism(width = w) -> w
```

一般情況下，您可以使用案例識別碼，而不使用聯集的名稱來限定。如果您想要使用聯集的名稱來限定名稱，您可以將 `RequireQualifiedAccess` 屬性套用至聯集類型定義。

解除包裝區分等位

在 F# 差異等位中，通常用於包裝單一型別的網域模型。您也可以透過模式比對，輕鬆地將基礎值解壓縮。單一案例不需要使用 `match` 運算式：

```
let ([UnionCaseIdentifier] [values]) = [UnionValue]
```

下列範例為其示範：

```
type ShaderProgram = | ShaderProgram of id:int

let someFunctionUsingShaderProgram shaderProgram =
    let (ShaderProgram id) = shaderProgram
    // Use the unwrapped value
    ...
```

也可以直接在函式參數中使用模式比對，因此您可以在該處解除包裝單一案例：

```
let someFunctionUsingShaderProgram (ShaderProgram id) =  
    // Use the unwrapped value  
    ...
```

結構區分等位

您也可以將區分等位表示為結構。這是透過屬性來完成 `[<Struct>]` 。

```
[<Struct>]  
type SingleCase = Case of string  
  
[<Struct>]  
type Multicase =  
    | Case1 of Case1 : string  
    | Case2 of Case2 : int  
    | Case3 of Case3 : double
```

因為這些是實值型別，而不是參考型別，所以相較于參考差異聯集，還有其他考慮：

1. 它們會複製為實數值型別，並具有實數值型別的語義。
2. 您無法使用具有 multicase 結構差異聯集的遞迴型別定義。
3. 您必須為 multicase 結構差異聯集提供唯一的案例名稱。

使用區分聯集而非物件階層

您通常可以使用差異聯集做為小型物件階層的較簡單替代項。例如，您可以使用下列差異聯集，而不是 `Shape` 具有圓形、正方形等衍生類型的基類。

```
type Shape =  
    // The value here is the radius.  
    | Circle of float  
    // The value here is the side length.  
    | EquilateralTriangle of double  
    // The value here is the side length.  
    | Square of double  
    // The values here are the height and width.  
    | Rectangle of double * double
```

您可以使用模式比對來分支至適當的公式來計算這些數量，而不是使用虛擬方法來計算區域或周邊。在下列範例中，根據圖形，使用不同的公式來計算區域。

```

let pi = 3.141592654

let area myShape =
  match myShape with
  | Circle radius -> pi * radius * radius
  | EquilateralTriangle s -> (sqrt 3.0) / 4.0 * s * s
  | Square s -> s * s
  | Rectangle (h, w) -> h * w

let radius = 15.0
let myCircle = Circle(radius)
printfn "Area of circle that has radius %f: %f" radius (area myCircle)

let squareSide = 10.0
let mySquare = Square(squareSide)
printfn "Area of square that has side %f: %f" squareSide (area mySquare)

let height, width = 5.0, 10.0
let myRectangle = Rectangle(height, width)
printfn "Area of rectangle that has height %f and width %f is %f" height width (area myRectangle)

```

輸出如下所示：

```

Area of circle that has radius 15.000000: 706.858347
Area of square that has side 10.000000: 100.000000
Area of rectangle that has height 5.000000 and width 10.000000 is 50.000000

```

針對樹狀結構資料結構使用區分聯集

差異聯集可以是遞迴的，也就是說，等位本身可以包含在一或多個案例的型別中。遞迴差異聯集可以用來建立樹狀結構，用來建立程式設計語言中的運算式模型。在下列程式碼中，會使用遞迴差異聯集來建立二進位樹狀結構資料結構。Union 包含兩個案例，也 `Node` 就是具有整數值、左和右樹系的節點，以及用 `Tip` 來終止樹狀結構的節點。

```

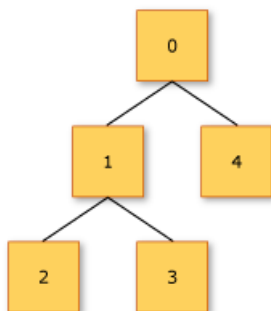
type Tree =
  | Tip
  | Node of int * Tree * Tree

let rec sumTree tree =
  match tree with
  | Tip -> 0
  | Node(value, left, right) ->
    value + sumTree(left) + sumTree(right)

let myTree = Node(0, Node(1, Node(2, Tip, Tip), Node(3, Tip, Tip)), Node(4, Tip, Tip))
let resultSumTree = sumTree myTree

```

在先前的程式碼中，`resultSumTree` 的值為10。下圖顯示的樹狀結構 `myTree`。



如果樹狀結構中的節點是異類的，則區分聯集的運作效果很好。在下列程式碼中，類型 `Expression` 以簡單的程

式設計語言表示運算式的抽象語法樹狀結構，支援數位和變數的加法和乘法。某些聯集案例不是遞迴的，且代表 `Number` () 的 () 或變數 `Variable` 。其他案例是遞迴的，表示作業 (`Add` 和 `Multiply`)，其中運算元也是運算式。`Evaluate` 函數會使用 `match` 運算式以遞迴方式處理語法樹狀結構。

```
type Expression =
  | Number of int
  | Add of Expression * Expression
  | Multiply of Expression * Expression
  | Variable of string

let rec Evaluate (env:Map<string,int>) exp =
  match exp with
  | Number n -> n
  | Add (x, y) -> Evaluate env x + Evaluate env y
  | Multiply (x, y) -> Evaluate env x * Evaluate env y
  | Variable id -> env.[id]

let environment = Map.ofList [ "a", 1 ;
                               "b", 2 ;
                               "c", 3 ]

// Create an expression tree that represents
// the expression: a + 2 * b.
let expressionTree1 = Add(Variable "a", Multiply(Number 2, Variable "b"))

// Evaluate the expression a + 2 * b, given the
// table of values for the variables.
let result = Evaluate environment expressionTree1
```

執行此程式碼時，的值 `result` 為5。

成員

您可以定義區分等位的成員。下列範例顯示如何定義屬性並執行介面：

```
open System

type IPrintable =
    abstract Print: unit -> unit

type Shape =
  | Circle of float
  | EquilateralTriangle of float
  | Square of float
  | Rectangle of float * float

member this.Area =
  match this with
  | Circle r -> 2.0 * Math.PI * r
  | EquilateralTriangle s -> s * s * sqrt 3.0 / 4.0
  | Square s -> s * s
  | Rectangle(l, w) -> l * w

interface IPrintable with
  member this.Print () =
    match this with
    | Circle r -> printfn $"Circle with radius %{r}"
    | EquilateralTriangle s -> printfn $"Equilateral Triangle of side %{s}"
    | Square s -> printfn $"Square with side %{s}"
    | Rectangle(l, w) -> printfn $"Rectangle with length %{l} and width %{w}"
```

通用屬性

下列屬性通常會出現在區分等位中：

- [`<RequireQualifiedAccess>`]
- [`<NoEquality>`]
- [`<NoComparison>`]
- [`<Struct>`]

另請參閱

- [F # 語言參考](#)

列舉

2020/11/2 • [Edit Online](#)

列舉也稱為*列舉*，這是將標籤指派給值子集的整數類資料類型。*enums* 列舉可用來取代常值，讓程式碼更容易閱讀及維護。

語法

```
type enum-name =  
| value1 = integer-literal1  
| value2 = integer-literal2  
...
```

備註

列舉看起來很像是具有簡單值的差異聯集，不同之處在於可以指定值。這些值通常是從0或1開始的整數，或是代表位位置的整數。如果列舉的目的是要代表位位置，您也應該使用 [旗標](#) 屬性。

列舉的基礎類型取決於所使用的常值，因此，例如，您可以針對不帶正負號的 `1u` `2u` 整數 () 類型，使用具有尾碼的常值(例如、等) `uint32` 。

當您參考命名值時，您必須使用列舉型別本身的名稱做為辨識符號，也就是，而 `enum-name.value1` 不只是 `value1` 。這種行為與不同聯集的行為不同。這是因為列舉一律具有 [RequireQualifiedAccess](#) 屬性。

下列程式碼顯示列舉和使用列舉。

```
// Declaration of an enumeration.  
type Color =  
| Red = 0  
| Green = 1  
| Blue = 2  
// Use of an enumeration.  
let col1 : Color = Color.Red
```

您可以使用適當的運算子輕鬆地將列舉轉換成基礎型別，如下列程式碼所示。

```
// Conversion to an integral type.  
let n = int col1
```

列舉類型可以具有下列其中一個基礎類型：`sbyte`、`byte`、`int16`、`uint16`、`int32`、`uint32`、`int64`、`uint64` 和 `char` 。列舉型別會在 .NET Framework 中表示為繼承自的型別，而繼承自的類型 `System.Enum` `System.ValueType` 。因此，它們是位於堆疊上或內嵌于包含物件中的實值型別，而基礎類型的任何值都是列舉的有效值。這在對列舉值進行模式比對時很重要，因為您必須提供可攔截未命名值的模式。

`enum` F# 程式庫中的函式可用來產生列舉值，甚至是除了其中一個預先定義的命名值以外的值。您可以使用 `enum` 如下所示的函數。

```
let col2 = enum<Color>(3)
```

預設 `enum` 函數適用於類型 `int32` 。因此，它不能搭配具有其他基礎類型的列舉類型使用。請改用下列各項。


```
type uColor =  
    | Red = 0u  
    | Green = 1u  
    | Blue = 2u  
let col3 = Microsoft.FSharp.Core.LanguagePrimitives.EnumOfValue<uint32, uColor>(2u)
```

此外，列舉的案例一律會發出為 `public`。這是為了使其符合 c # 和 .NET 平臺的其餘部分。

另請參閱

- [F # 語言參考](#)
- [轉型和轉換](#)

類型縮寫

2019/10/23 • [Edit Online](#)

*類型縮寫*是類型的別名或替代名稱。

語法

```
type [accessibility-modifier] type-abbreviation = type-name
```

備註

您可以使用類型縮寫來為類型提供更有意義的名稱, 以便讓程式碼更容易閱讀。您也可以使用它們來建立容易使用的名稱, 以用於不需要寫出的其他類型。此外, 您可以使用類型縮寫, 讓變更基礎類型變得更容易, 而不需要變更使用該類型的所有程式碼。以下是簡單的類型縮寫。

縮寫類型的存取範圍預設 `public` 為。

```
type SizeType = uint32
```

類型縮寫可以包含泛型參數, 如下列程式碼所示。

```
type Transform<'a> = 'a -> 'a
```

在先前的程式碼 `Transform` 中, 是類型縮寫, 代表接受任何類型的單一引數, 並傳回該相同類型之單一值的函式。

類型縮寫不會保留在 .NET Framework 的 MSIL 程式碼中。因此, 當您使用另F#一個 .NET Framework 語言的元件時, 您必須使用基礎類型名稱做為類型縮寫。

類型縮寫也可以用在測量單位上。如需詳細資訊, 請參閱[測量單位](#)。

另請參閱

- [F# 語言參考](#)

類別

2021/3/5 • [Edit Online](#)

類別是代表可以有屬性、方法和事件之物件的類型。

語法

```
// Class definition:
type [access-modifier] type-name [type-params] [access-modifier] ( parameter-list ) [ as identifier ] =
[ class ]
[ inherit base-type-name(base-constructor-args) ]
[ let-bindings ]
[ do-bindings ]
member-list
...
[ end ]
// Mutually recursive class definitions:
type [access-modifier] type-name1 ...
and [access-modifier] type-name2 ...
...
```

備註

類別代表 .NET 物件類型的基本描述;類別是一種主要類型概念, 支援 F # 中的物件導向程式設計。

在上述語法中, `type-name` 是任何有效的識別碼。 `type-params` 描述選擇性的泛型型別參數。它包含以角括弧括住的型別參數名稱和條件約束 (`<` 和 `>`)。如需詳細資訊, 請參閱 [泛型](#) 和 [條件約束](#)。描述函式 `parameter-list` 參數。第一個存取修飾詞與類型有關。第二個則是主要的函式。在這兩種情況下, 預設值為 `public`。

您可以使用關鍵字來指定類別的基類 `inherit`。您必須為基類的函式提供引數(以括弧括住)。

您可以使用系結來宣告類別本機的欄位或函式值 `let`, 而且必須遵循系結的一般規則 `let`。 `do-bindings` 區段包含要在物件結構上執行的程式碼。

`member-list` 包含其他的函式、實例和靜態方法宣告、介面宣告、抽象系結, 以及屬性和事件宣告。這些是 [成員](#)中的描述。

搭配 `identifier` 選擇性關鍵字使用的會 `as` 提供執行個體變數的名稱或本身的識別碼, 可在類型定義中用來參考該類型的實例。如需詳細資訊, 請參閱本主題稍後的「自我識別碼」一節。

`class` `end` 標示定義開始和結束的關鍵字是選擇性的。

相互遞迴的型別(也就是彼此參考的型別)會與關鍵字聯結在一起, `and` 就像互相遞迴函數一樣。如需範例, 請參閱「相互遞迴類型」一節。

建構函式

此函式是建立類別型別實例的程式碼。類別的函式在 F # 中的運作方式與其他 .NET 語言中的運作方式稍有不同。在 F # 類別中, 一律會有一個主要的函式, 其中的引數會在 `parameter-list` 型別名稱後面, 且主體是由 `let` 類別宣告開頭的 (和) 系結 `let rec`, 以及 `do` 接下來的系結所組成。主要函式的引數位於整個類別宣告的範圍內。

您可以使用關鍵字加入其他的函式 `new`, 以新增成員, 如下所示:

```
new ( argument-list ) = constructor-body
```

新的函式主體必須叫用在類別宣告頂端指定的主要函式。

下列範例將說明這個概念。在下列程式碼中，`MyClass` 有兩個函式，這是採用兩個引數的主要函式，以及不接受引數的另一個函式。

```
type MyClass1(x: int, y: int) =  
    do printfn "%d %d" x y  
    new() = MyClass1(0, 0)
```

let 和 do 系結

`let` 類別定義中的和系結 `do` 會形成主要類別的函式內文，因此每當建立類別實例時，就會執行這些系結。如果系結 `let` 是函數，則會將它編譯成成員。如果系結 `let` 是未在任何函式或成員中使用的值，則會將它編譯成函式的本機變數。否則，它會編譯成類別的欄位。`do` 接下來的運算式會編譯成主要的函式，並對每個實例執行初始化程式碼。因為任何其他的函式一律會呼叫主要的函式，所以不論呼叫哪一個函式，系結和系結 `let` `do` 一律都會執行。

系結所建立的欄位 `let` 可以在類別的方法和屬性中存取，不過，即使靜態方法採用執行個體變數作為參數，也無法從靜態方法存取這些欄位。如果有的話，就無法使用自我識別碼來存取它們。

自我識別碼

自我識別碼 是表示目前實例的名稱。自我識別碼與 `this` C# 或 C++ 或 `Me` Visual Basic 中的關鍵字類似。您可以使用兩種不同的方式來定義自我識別碼，視您是否要讓自我識別碼位於整個類別定義的範圍中，或僅針對個別的方法而定。

若要定義整個類別的自我識別碼，請在 [函式 `as` 參數清單] 的右括弧後面使用關鍵字，並指定識別碼名稱。

若只要定義一個方法的自我識別碼，請在方法名稱和句點 (之前，在成員宣告中提供自我識別碼。) 為分隔符號。

下列程式碼範例說明建立自我識別碼的兩種方式。在第一行中，`as` 關鍵字是用來定義自我識別碼。在第五行中，識別碼 `this` 是用來定義其範圍限制為方法的自我識別碼 `PrintMessage` 。

```
type MyClass2(dataIn) as self =  
    let data = dataIn  
    do  
        self.PrintMessage()  
    member this.PrintMessage() =  
        printf "Creating MyClass2 with Data %d" data
```

不同于其他 .NET 語言，您可以在想要的情況下命名自我識別碼;您不受限於、或之類的 `self` 名稱 `Me` `this` 。

使用關鍵字宣告的自我識別碼，在 `as` 基底的函式之前都不會初始化。因此，在基底函式之前或內部使用時，

```
System.InvalidOperationException: The initialization of an object or value resulted in an object or value being accessed recursively before it was fully initialized.
```

會在執行時間引發。您可以在基底的函式之後自由使用自我識別碼，例如在系結或系結中 `let` `do` 。

泛型類型參數

泛型型別參數是以角括弧 (`<` 和 `>`) 來指定，並以單引號的形式加上識別碼。多個泛型型別參數會以逗號分隔。泛型型別參數在整個宣告的範圍中。下列程式碼範例示範如何指定泛型型別參數。

```
type MyGenericClass<'a> (x: 'a) =  
    do printfn "%A" x
```

使用型別時，會推斷型別引數。在下列程式碼中，推斷的型別是一系列的元組。

```
let g1 = MyGenericClass( seq { for i in 1 .. 10 -> (i, i*i) } )
```

指定繼承

`inherit` 子句會識別直接基類(如果有的話)。在 F # 中，只允許一個直接基類。類別所執行的介面不會被視為基類。[介面主題](#)中會討論介面。

您可以從衍生類別存取基類的方法和屬性，方法是使用 `language` 關鍵字 `base` 做為識別碼，後面接著一個句點 (。) 和成員的名稱。

如需詳細資訊，請參閱[繼承](#)。

成員區段

您可以在本節中定義靜態或實例方法、屬性、介面執行、抽象成員、事件宣告和其他的函式。此區段中不能出現 `Let` 和 `do` 系結。因為成員可以新增至類別以外的各種 F # 型別，所以會在個別的主題中討論 [成員](#)。

相互遞迴類型

當您定義以迴圈方式參考彼此的型別時，您可以使用關鍵字將型別定義組成一串 `and`。 `and` 關鍵字會取代 `type` 第一個定義以外的所有關鍵字，如下所示。

```
open System.IO  
  
type Folder(pathIn: string) =  
    let path = pathIn  
    let filenameArray : string array = Directory.GetFiles(path)  
    member this.FileArray = Array.map (fun elem -> new File(elem, this)) filenameArray  
  
and File(filename: string, containingFolder: Folder) =  
    member this.Name = filename  
    member this.ContainingFolder = containingFolder  
  
let folder1 = new Folder(".")  
for file in folder1.FileArray do  
    printfn "%s" file.Name
```

輸出是目前目錄中所有檔案的清單。

使用類別、等位、記錄和結構的時機

由於有各種類型可供選擇，因此您需要充分瞭解每種類型的設計目的，以針對特定情況選取適當的類型。類別是設計用來在物件導向程式設計環境中使用。物件導向程式設計是針對 .NET Framework 所撰寫的應用程式中所使用的主要範例。如果您的 F # 程式碼必須與 .NET Framework 或另一個物件導向的程式庫密切合作，特別是當您必須從物件導向的型別系統(例如 UI 程式庫)延伸時，類別可能很適合。

如果您未與物件導向程式碼緊密地互通，或您正在撰寫獨立的程式碼，因而受到保護而無法經常與物件導向程式碼互動，您應該考慮使用記錄和區分等位。您可以使用一種簡單的方法，也就是使用適當的模式比對程式碼，將其視為更簡單的物件階層替代方法。如需差異聯集的詳細資訊，請參閱 [區分聯集](#)。

記錄的優點是比類別更簡單，但是當類型的需求超過其簡單的需求時，記錄就不適用。記錄基本上是值的簡單匯

總，沒有可執行自訂動作之個別函式，而不需要隱藏的欄位，也不需要繼承或介面。雖然可以將屬性和方法等成員新增至記錄，以使其行為更複雜，但儲存在記錄中的欄位仍是值的簡單匯總。如需記錄的詳細資訊，請參閱 [記錄](#)。

結構也適用於小型資料匯總，但它們與類別和記錄不同，因為它們是 .NET 實數值型別。類別和記錄是 .NET 參考型別。實值型別和參考型別的語義在實值型別是以傳值方式傳遞的。這表示當其以參數形式傳遞或從函式傳回時，會為位複製位。它們也會儲存在堆疊上，或者，如果當做欄位使用，則內嵌于父物件中，而不是儲存在堆積上的個別位置。因此，當存取堆積的額外負荷發生問題時，結構適用於經常存取的資料。如需結構的詳細資訊，請參閱 [結構](#)。

另請參閱

- [F # 語言參考](#)
- [成員](#)
- [繼承](#)
- [介面](#)

結構

2020/11/2 • [Edit Online](#)

結構是一種精簡的物件類型，相較于具有少量資料和簡單行為之類型的類別，其效率可能更高。

語法

```
[ attributes ]
type [accessibility-modifier] type-name =
    struct
        type-definition-elements-and-members
    end
// or
[ attributes ]
[<StructAttribute>]
type [accessibility-modifier] type-name =
    type-definition-elements-and-members
```

備註

結構是實 *值* 型別，這表示這些型別會直接儲存在堆疊上，或做為欄位或陣列元素（內嵌于父系型別）使用。有別於類別與記錄，結構具有以值傳遞的語意。這表示它們主要對於經常存取及複製的小型資料彙總相當實用。

在先前的語法中，顯示了兩個表單。第一個表單的語法略為複雜，但使用卻很頻繁，因為當您使用 `struct` 和 `end` 關鍵字時，可以省略出現在第二個表單中的 `StructAttribute` 屬性。您可以將 `StructAttribute` 縮寫為只有 `Struct`。

上述語法中的 *類型定義-元素和成員* 代表成員宣告和定義。結構可以具有建構函式及可變動和不可變的欄位，同時它們可以宣告成員及介面實作。如需詳細資訊，請參閱 [成員](#)。

結構無法參與繼承、不能包含 `let` 或 `do` 繫結，且不得以遞迴方式包含其本身類型的欄位（不過它們可以包含參考其本身類型的參考儲存格）。

因為結構不允許 `let` 繫結，因此您必須使用 `val` 關鍵字來宣告結構中的欄位。`val` 關鍵字會定義欄位及其類型，但不允許進行初始化。相反地，`val` 宣告會初始化為零或 `null`。基於這個理由，具有隱含建構函式（也就是緊接在宣告中結構名稱後的指定參數）的結構，需要 `val` 宣告標註 `DefaultValue` 屬性。具有已定義的建構函式之結構，仍然支援零初始化。因此，`DefaultValue` 屬性是一種零值對於欄位有效的宣告。結構的隱含建構函式不會執行任何動作，因為類型上不允許 `let` 和 `do` 繫結，但傳入的隱含建構函式參數值均可用作為私用欄位。

明確建構函式可能會牽涉到欄位值的初始化。當您的結構有明確的建構函式時，它仍然可以支援零初始化。不過，請勿在 `DefaultValue` 宣告上使用 `val` 屬性，因為它與明確建構函式相衝突。如需宣告的詳細資訊 `val`，請參閱 [明確欄位](#)：`val` 關鍵字。

結構上允許屬性和存取範圍修飾詞，並遵循與其他類型相同的規則。如需詳細資訊，請參閱 [屬性](#) 和 [存取控制](#)。

下列程式碼範例說明結構定義。

```
// In Point3D, three immutable values are defined.
// x, y, and z will be initialized to 0.0.
type Point3D =
    struct
        val x: float
        val y: float
        val z: float
    end

// In Point2D, two immutable values are defined.
// It also has a member which computes a distance between itself and another Point2D.
// Point2D has an explicit constructor.
// You can create zero-initialized instances of Point2D, or you can
// pass in arguments to initialize the values.
type Point2D =
    struct
        val X: float
        val Y: float
        new(x: float, y: float) = { X = x; Y = y }

        member this.GetDistanceFrom(p: Point2D) =
            let dX = (p.X - this.X) ** 2.0
            let dY = (p.Y - this.Y) ** 2.0

            dX + dY
            |> sqrt
    end
```

ByRefLike 結構

您可以定義您自己的結構，以遵循 `byref` 類似的語義：如需詳細資訊，請參閱 [byref](#)。這是使用屬性來完成的 [IsByRefLikeAttribute](#)：

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` 不表示 `Struct`。兩者都必須存在於類型上。

`byref` F# 中的「-like」結構是堆疊系結的實值型別。它永遠不會在 managed 堆積上進行配置。`byref` 類似的結構適用於高效能程式設計，因為它會利用存留期和非捕捉的一組強大檢查來強制執行。這些規則包括：

- 它們可用來作為函式參數、方法參數、區域變數、方法傳回。
- 它們不得為類別或一般結構的靜態或實例成員。
- `async` (方法或 lambda 運算式時，無法由任何關閉結構來捕捉它們。
- 它們不能用來做為泛型參數。

雖然這些規則非常嚴格地限制使用方式，但它們會以安全的方式滿足高效能運算的承諾。

ReadOnly 結構

您可以使用屬性來標注結構 [IsReadOnlyAttribute](#)。例如：


```
[<IsReadOnly; Struct>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsReadOnly` 不表示 `Struct` 。您必須加入兩者以具有 `IsReadOnly` 結構。

使用這個屬性會發出中繼資料，讓 F # 和 c # 知道分別將它視為 `inref<'T>` 和 `in ref` 。

在 `readonly` 結構內定義可變值會產生錯誤。

結構記錄和區分等位

您可以將 [記錄](#) 和 [差異](#) 聯集表示為具有 `[<Struct>]` 屬性的結構。請參閱每篇文章以深入瞭解。

另請參閱

- [F # 語言參考](#)
- [類別](#)
- [記錄](#)
- [成員](#)

可為 Null 的實值型別

2021/3/5 • [Edit Online](#)

可為 *null* 的實值型別 `Nullable<'T>` 代表也可以是任何 [結構](#) 類型 `null`。當您與可選擇用來代表這些類型類型的程式庫和元件 (例如整數) 進行互動時, 這項功能會很有說明 `null`。支援此結構的基礎類型為 `System.Nullable<T>`。

Syntax

```
Nullable<'T>  
Nullable value
```

使用值宣告並指派

宣告可為 *null* 的實值型別, 就像在 F# 中宣告任何類似包裝函式的型別一樣:

```
open System  
  
let x = 12  
let nullableX = Nullable<int> x
```

您也可以省略泛型型別參數, 並允許型別推斷來解析它:

```
open System  
  
let x = 12  
let nullableX = Nullable x
```

若要指派給可為 *null* 的實值型別, 您也必須是明確的。F# 定義可為 *null* 的實數值型別沒有隱含轉換:

```
open System  
  
let mutable x = Nullable 12  
x <- Nullable 13
```

指派 null

您無法直接指派 `null` 給可為 *null* 的實值型別。請 `Nullable()` 改用:

```
let mutable a = Nullable 42  
a <- Nullable()
```

這是因為沒有 `Nullable<'T>` `null` 適當的值。

傳遞並指派給成員

使用成員和 F# 值兩者之間的主要差異在於, 當您使用成員時, 可以隱含推斷可為 *null* 的實數值型別。請考慮採用可為 *null* 實值型別做為輸入的下列方法:

```

type C() =
    member _.M(x: Nullable<int>) = x.HasValue
    member val NVT = Nullable 12 with get, set

let c = C()
c.M(12)
c.NVT <- 12

```

在上述範例中，您可以傳遞 `12` 給方法 `M`。您也可以指派 `12` 給 auto 屬性 `NVT`。如果輸入可視為可為 null 的實值型別，且符合目標型別，則 F# 編譯器會隱含地轉換這類呼叫或指派。

檢查可為 null 的實值型別實例

不同于代表可能值的一般化結構，可為 null 的實值型別不 [會與模式](#) 比對搭配使用。相反地，您必須使用 `if` 運算式並檢查 `HasValue` 屬性。

若要取得基礎值，請在 `Value` 檢查之後使用屬性 `HasValue`，如下所示：

```

open System

let a = Nullable 42

if a.HasValue then
    printfn $"{a} is {a.Value}"
else
    printfn $"{a} has no value."

```

可為 null 的運算子

針對可為 null 的實數值型別 (例如算術或比較) 作業，可以使用 [可為 null 的運算子](#)。

您可以使用命名空間中的轉換運算子，從一個可為 null 的實值型別轉換為另一個 `FSharp.Linq`：

```

open System
open FSharp.Linq

let nullableInt = Nullable 10
let nullableFloat = Nullable.float nullableInt

```

您也可以使用適當的不可為 null 運算子來轉換為基本型別，如果沒有任何值，則會對例外狀況產生風險：

```

open System
open FSharp.Linq

let nullableInt = Nullable 10
let nullableFloat = Nullable.float nullableInt

printfn $"value is %f{float nullableFloat}"

```

您也可以使用可為 null 的運算子作為檢查和的短期運算子 `HasValue` `Value`：

```
open System
open FSharp.Linq

let nullableInt = Nullable 10
let nullableFloat = Nullable.float nullableInt

let isBigger = nullableFloat ?> 1.0
let isBiggerLongForm = nullableFloat.HasValue && nullableFloat.Value > 1.0
```

`?>` 比較會檢查左邊是否可為 null, 而且只有在有值時才會成功。它相當於其後的行。

另請參閱

- [結構](#)
- [F # 選項](#)

在物件導向程式設計中，會使用繼承來建立「是」關聯性或 subtyping 的模型。

指定繼承關聯性

您可以在類別宣告中使用關鍵字來指定繼承關聯性 `inherit`。下列範例會顯示基本的語法形式。

```
type MyDerived(...) =  
    inherit MyBase(...)
```

一個類別最多隻能有一個直接基類。如果您沒有使用關鍵字來指定基類 `inherit`，則類別會隱含地繼承自 `System.Object`。

繼承的成員

如果類別繼承自另一個類別，則衍生類別的使用者可以使用基類的方法和成員，如同它們是衍生類別的直接成員。

任何 `let` 系結和函式參數都是類別私用的，因此無法從衍生類別存取。

關鍵字 `base` 會在衍生類別中提供，並參考基類實例。它會像自我識別碼一樣使用。

虛擬方法和覆寫

相較于其他 .NET 語言，(和屬性的虛擬方法) 在 F# 中的運作方式稍有不同。若要宣告新的虛擬成員，請使用 `abstract` 關鍵字。無論您是否提供該方法的預設執行，您都可以這麼做。因此，基底類別中虛擬方法的完整定義會遵循此模式：

```
abstract member [method-name] : [type]  
  
default [self-identifier].[method-name] [argument-list] = [method-body]
```

在衍生類別中，此虛擬方法的覆寫會遵循此模式：

```
override [self-identifier].[method-name] [argument-list] = [method-body]
```

如果您省略基類中的預設實值，基類就會變成抽象類別。

下列程式碼範例說明如何在基類中宣告新的虛擬方法 `function1`，以及如何在衍生類別中覆寫它。

```
type MyClassBase1() =  
    let mutable z = 0  
    abstract member function1 : int -> int  
    default u.function1(a : int) = z <- z + a; z  
  
type MyClassDerived1() =  
    inherit MyClassBase1()  
    override u.function1(a: int) = a + 1
```

建構函式和繼承

基類的函式必須在衍生類別中呼叫。基類 (base class) 的引數會出現在子句的引數清單中 `inherit`。使用的值必須取決於提供給衍生類別的引數。

下列程式碼顯示基類和衍生類別，其中衍生類別會在繼承子句中呼叫基類的函式：

```
type MyClassBase2(x: int) =  
    let mutable z = x * x  
    do for i in 1..z do printf "%d " i  
  
type MyClassDerived2(y: int) =  
    inherit MyClassBase2(y * 2)  
    do for i in 1..y do printf "%d " i
```

在多個函式的情況下，可以使用下列程式碼。衍生類別的第一行是 `inherit` 子句，而這些欄位會顯示為使用關鍵字宣告的明確欄位 `val`。如需詳細資訊，請參閱 [明確欄位：val 關鍵字](#)。

```
type BaseClass =  
    val string1 : string  
    new (str) = { string1 = str }  
    new () = { string1 = "" }  
  
type DerivedClass =  
    inherit BaseClass  
  
    val string2 : string  
    new (str1, str2) = { inherit BaseClass(str1); string2 = str2 }  
    new (str2) = { inherit BaseClass(); string2 = str2 }  
  
let obj1 = DerivedClass("A", "B")  
let obj2 = DerivedClass("A")
```

繼承的替代方案

在需要稍微修改類型的情況下，請考慮使用物件運算式作為繼承的替代方法。下列範例說明如何使用物件運算式做為建立新衍生型別的替代方法：

```
open System  
  
let object1 = { new Object() with  
    override this.ToString() = "This overrides object.ToString()"  
}  
  
printfn "%s" (object1.ToString())
```

如需物件運算式的詳細資訊，請參閱 [物件運算式](#)。

當您建立物件階層時，請考慮使用差異聯集，而不是繼承。差異聯集也可以針對共用通用整體類型的不同物件，建立各種不同的行為模型。單一差異聯集通常可以免除一些不同的衍生類別需求。如需差異聯集的詳細資訊，請參閱 [區分聯集](#)。

另請參閱

- [物件運算式](#)
- [F# 語言參考](#)

介面

2021/3/5 • [Edit Online](#)

介面會指定其他類別所執行的相關成員集合。

語法

```
// Interface declaration:
[ attributes ]
type [accessibility-modifier] interface-name =
    [ interface ]      [ inherit base-interface-name ...]
    abstract member1 : [ argument-types1 -> ] return-type1
    abstract member2 : [ argument-types2 -> ] return-type2
    ...
[ end ]

// Implementing, inside a class type definition:
interface interface-name with
    member self-identifier.member1argument-list = method-body1
    member self-identifier.member2argument-list = method-body2

// Implementing, by using an object expression:
[ attributes ]
let class-name (argument-list) =
    { new interface-name with
        member self-identifier.member1argument-list = method-body1
        member self-identifier.member2argument-list = method-body2
        [ base-interface-definitions ]
    }
    member-list
```

備註

介面宣告類似類別宣告，但不會執行任何成員。相反地，所有成員都是抽象的，如關鍵字所指示 `abstract`。您未提供抽象方法的方法主體。不過，您也可以提供預設的執行方式，也就是將個別的成員定義做為方法與 `default` 關鍵字一起使用。這麼做相當於在其他 .NET 語言的基類中建立虛擬方法。這類虛擬方法可以在執行介面的類別中覆寫。

介面的預設存取範圍是 `public`。

您可以使用一般 F# 語法，選擇性地為每個方法參數指定名稱：

```
type ISprintable =
    abstract member Print : format:string -> unit
```

在上述 `ISprintable` 範例中，`Print` 方法具有類型的單一參數，其名稱為 `string` `format`。

有兩種方式可以執行介面：使用物件運算式，以及使用類別類型。在一種情況下，類別類型或物件運算式都會提供介面之抽象方法的方法主體。實作為每個實作為介面的型別。因此，不同類型的介面方法可能會彼此不同。

`interface` `end` 當您使用輕量語法時，關鍵字和（標示定義的開頭和結尾）是選擇性的。如果您未使用這些關鍵字，編譯器會藉由分析您所使用的結構，嘗試推斷型別為類別或介面。如果您定義成員或使用其他類別語法，則會將型別解釋為類別。

.NET 程式碼撰寫樣式是以大寫來開始所有的介面 `I` 。

您可以兩種方式指定多個參數：F # 樣式和 .NET 樣式。這兩種方法都會針對 .NET 取用者進行編譯，但 F # 樣式會強制 F # 呼叫端使用 F # 樣式參數應用程式和 .NET 樣式會強制 F # 呼叫端使用元組引數應用程式。

```
type INumeric1 =  
    abstract Add: x: int -> y: int -> int  
  
type INumeric2 =  
    abstract Add: x: int * y: int -> int
```

使用類別類型來執行介面

您可以使用 `interface` 關鍵字、介面名稱和關鍵字，然後使用介面成員定義，在類別型別中執行一個或多個介面，`with` 如下列程式碼所示。

```
type IPrintable =  
    abstract member Print : unit -> unit  
  
type SomeClass1(x: int, y: float) =  
    interface IPrintable with  
        member this.Print() = printfn "%d %f" x y
```

介面會被繼承，因此任何衍生的類別都不需要重新執行它們。

呼叫介面方法

介面方法只能透過介面呼叫，而不能透過任何實介面型別的物件來呼叫。因此，您可能必須使用運算子或運算子來向上轉型為介面型別，才能 `:>` `upcast` 呼叫這些方法。

當您擁有類型的物件時，若要呼叫介面方法 `SomeClass`，您必須將物件向上轉型為介面類別型，如下列程式碼所示。

```
let x1 = new SomeClass1(1, 2.0)  
(x1 :> IPrintable).Print()
```

替代方法是在將和呼叫介面方法的物件上宣告方法，如下列範例所示。

```
type SomeClass2(x: int, y: float) =  
    member this.Print() = (this :> IPrintable).Print()  
    interface IPrintable with  
        member this.Print() = printfn "%d %f" x y  
  
let x2 = new SomeClass2(1, 2.0)  
x2.Print()
```

使用物件運算式來執行介面

物件運算式提供簡單的方法來執行介面。當您不需要建立已命名的型別，而且您只想要支援介面方法的物件，而不需要任何其他方法時，它們就很有用。下列程式碼說明物件運算式。


```
let makePrintable(x: int, y: float) =
    { new IPrintable with
        member this.Print() = printfn "%d %f" x y }
let x3 = makePrintable(1, 2.0)
x3.Print()
```

介面繼承

介面可以繼承自一或多個基底介面。

```
type Interface1 =
    abstract member Method1 : int -> int

type Interface2 =
    abstract member Method2 : int -> int

type Interface3 =
    inherit Interface1
    inherit Interface2
    abstract member Method3 : int -> int

type MyClass() =
    interface Interface3 with
        member this.Method1(n) = 2 * n
        member this.Method2(n) = n + 100
        member this.Method3(n) = n / 10
```

使用預設實執行介面

C# 支援使用預設實作為定義介面，如下所示：

```
using System;

namespace CSharp
{
    public interface MyDim
    {
        public int Z => 0;
    }
}
```

這些可直接從 F# 取用：

```
open CSharp

// You can implement the interface via a class
type MyType() =
    member _.M() = ()

    interface MyDim

let md = MyType() :> MyDim
printfn $"DIM from C#: {md.Z}"

// You can also implement it via an object expression
let md' = { new MyDim }
printfn $"DIM from C# but via Object Expression: {md'.Z}"
```

您可以使用覆寫的預設實值 `override`，就像覆寫任何虛擬成員一樣。

沒有預設實值之介面中的任何成員仍必須明確地實作為。

在不同的泛型具現化中執行相同的介面

F # 支援在不同的泛型具現化上執行相同的介面，如下所示：

```
type IA<'T> =  
    abstract member Get : unit -> 'T  
  
type MyClass() =  
    interface IA<int> with  
        member x.Get() = 1  
    interface IA<string> with  
        member x.Get() = "hello"  
  
let mc = MyClass()  
let iaInt = mc :> IA<int>  
let iaString = mc :> IA<string>  
  
iaInt.Get() // 1  
iaString.Get() // "hello"
```

另請參閱

- [F # 語言參考](#)
- [物件運算式](#)
- [類別](#)

抽象類別

2019/10/23 • [Edit Online](#)

抽象類別是讓部分或所有成員不會執行的類別，以便讓衍生類別能夠提供執行。

語法

```
// Abstract class syntax.  
[<AbstractClass>]  
type [ accessibility-modifier ] abstract-class-name =  
[ inherit base-class-or-interface-name ]  
[ abstract-member-declarations-and-member-definitions ]  
  
// Abstract member syntax.  
abstract member member-name : type-signature
```

備註

在物件導向程式設計中，抽象類別會當做階層的基類使用，並代表一組不同物件類型的一般功能。顧名思義，抽象類別通常不會直接對應到問題網域中的具體實體。不過，它們確實代表有多少不同的具體實體具有共同的功能。

抽象類別必須具有 `AbstractClass` 屬性。它們可以有已實和未執行的成員。套用至類別時，使用「抽象」一詞，與其他 .net 語言相同。不過，套用至方法(和屬性)時，使用「抽象」(*abstract*)一詞與 F# 其他 .net 語言中的使用方式稍有不同。在 F# 中，當方法以 `abstract` 關鍵字標記時，這表示成員在該類型的虛擬函式的內部資料表中有一個專案，稱為「虛擬分派插槽」。換句話說，方法是虛擬的，但是 `virtual` 在 F# 語言中不使用關鍵字。無論方法 `abstract` 是否已實作為，都會在虛擬方法上使用關鍵字。虛擬分派位置的宣告與該分派插槽的方法定義不同。因此，在 F# 另一個 .net 語言中，虛擬方法宣告和定義的對等，是抽象方法宣告和個別定義的組合，其中包含 `default` 關鍵字或 `override` 關鍵字。如需詳細資訊和範例，請參閱 [方法](#)。

只有在有已宣告但未定義的抽象方法時，才會將類別視為抽象。因此，具有抽象方法的類別不一定是抽象類別。除非類別有未定義的抽象方法，否則請不要使用 `AbstractClass` 屬性。

在先前的語法中，[協助工具修飾詞](#) `public` 可以 `private` 是 `internal`、或。如需詳細資訊，請參閱 [存取控制](#)。

如同其他類型，抽象類別可以有基類和一個或多個基底介面。每個基類或介面都會與 `inherit` 關鍵字一起出現在不同的行上。

抽象類別的類型定義可以包含完整定義的成員，但也可以包含抽象成員。Abstract 成員的語法會分別顯示在先前的語法中。在此語法中，成員的型別簽章是一份清單，其中包含依 `->` 序排列的參數類型和傳回型別，並以標記和/或 `*` token 分隔，以適用於擴充和元組參數。抽象成員類型簽章的語法與簽章檔案中所使用的語法相同，以及 IntelliSense 在 Visual Studio Code 編輯器中所顯示的語法。

下列程式碼說明抽象類別圖形，其中包含兩個非抽象衍生類別：方形和 Circle。此範例示範如何使用抽象類別、方法和屬性。在此範例中，「抽象類別」圖形代表具體實體 circle 和正方形的通用元素。所有圖形(在二維座標系統中)的一般功能會被抽象化成 Shape 類別：格線上的位置、旋轉角度，以及區域和周邊屬性。這些可以覆寫，但位置除外，個別圖形無法變更的行為。

可以覆寫旋轉方法，如同 Circle 類別，這是因為其對稱而旋轉不變。因此，在 Circle 類別中，旋轉方法會由不執行任何動作的方法所取代。

```
// An abstract class that has some methods and properties defined
```

```

// and some left abstract.
[<AbstractClass>]
type Shape2D(x0 : float, y0 : float) =
    let mutable x, y = x0, y0
    let mutable rotAngle = 0.0

    // These properties are not declared abstract. They
    // cannot be overridden.
    member this.CenterX with get() = x and set xval = x <- xval
    member this.CenterY with get() = y and set yval = y <- yval

    // These properties are abstract, and no default implementation
    // is provided. Non-abstract derived classes must implement these.
    abstract Area : float with get
    abstract Perimeter : float with get
    abstract Name : string with get

    // This method is not declared abstract. It cannot be
    // overridden.
    member this.Move dx dy =
        x <- x + dx
        y <- y + dy

    // An abstract method that is given a default implementation
    // is equivalent to a virtual method in other .NET languages.
    // Rotate changes the internal angle of rotation of the square.
    // Angle is assumed to be in degrees.
    abstract member Rotate: float -> unit
    default this.Rotate(angle) = rotAngle <- rotAngle + angle

type Square(x, y, sideLengthIn) =
    inherit Shape2D(x, y)
    member this.SideLength = sideLengthIn
    override this.Area = this.SideLength * this.SideLength
    override this.Perimeter = this.SideLength * 4.
    override this.Name = "Square"

type Circle(x, y, radius) =
    inherit Shape2D(x, y)
    let PI = 3.141592654
    member this.Radius = radius
    override this.Area = PI * this.Radius * this.Radius
    override this.Perimeter = 2. * PI * this.Radius
    // Rotating a circle does nothing, so use the wildcard
    // character to discard the unused argument and
    // evaluate to unit.
    override this.Rotate(_) = ()
    override this.Name = "Circle"

let square1 = new Square(0.0, 0.0, 10.0)
let circle1 = new Circle(0.0, 0.0, 5.0)
circle1.CenterX <- 1.0
circle1.CenterY <- -2.0
square1.Move -1.0 2.0
square1.Rotate 45.0
circle1.Rotate 45.0
printfn "Perimeter of square with side length %f is %f, %f"
    (square1.SideLength) (square1.Area) (square1.Perimeter)
printfn "Circumference of circle with radius %f is %f, %f"
    (circle1.Radius) (circle1.Area) (circle1.Perimeter)

let shapeList : list<Shape2D> = [ (square1 :> Shape2D);
    (circle1 :> Shape2D) ]
List.iter (fun (elem : Shape2D) ->
    printfn "Area of %s: %f" (elem.Name) (elem.Area))
    shapeList

```

輸出：

```
Perimeter of square with side length 10.000000 is 40.000000
Circumference of circle with radius 5.000000 is 31.415927
Area of Square: 100.000000
Area of Circle: 78.539816
```

另請參閱

- [類別](#)
- [成員](#)
- [方法](#)
- [屬性](#)

Members

2019/11/25 • [Edit Online](#)

本節描述 F# 物件類型的成員。

備註

「成員」是作為類型定義的一部分且以 `member` 關鍵字宣告的功能。F# 物件類型 (例如記錄、類別、差別聯集、介面和結構) 支援成員。如需詳細資訊, 請參閱[記錄](#)、[類別](#)、[差別聯集](#)、[介面](#)和[結構](#)。

成員一般構成類型的公用介面, 因此除非另外指定, 否則成員為公用。成員也可以宣告為私用或內用。如需詳細資訊, 請參閱[存取控制](#)。類型的簽章也可以用來公開或不公開類型的特定成員。如需詳細資訊, 請參閱[簽章](#)。

只與類別搭配使用的私用欄位和 `do` 繫結, 並不是真正的成員, 因為它們永遠不是類型之公用介面的一部分, 也不是以 `member` 關鍵字宣告, 但本節中也會加以描述。

相關主題

「	」
類別中的 <code>let</code> 繫結	描述類別中私用欄位和函式的定義。
類別中的 <code>do</code> 繫結	描述物件初始設定程式碼的規格。
內容	描述類別和其他類型中的屬性成員。
索引屬性	描述類別和其他類型中的類似陣列屬性。
方法	描述作為類型成員的函式。
建構函式	描述初始化型別物件的特殊函式。
運算子多載	描述型別之自訂運算子的定義。
事件	描述 F# 中的事件定義和事件處理支援。
明確欄位: <code>val</code> 關鍵字	描述類型中未初始化欄位的定義。

類別中的 let 繫結

2019/10/23 • [Edit Online](#)

您可以使用 `let` 類別定義中的系結F#，定義類別的私用欄位和私用函式。

語法

```
// Field.  
[static] let [ mutable ] binding1 [ and ... binding-n ]  
  
// Function.  
[static] let [ rec ] binding1 [ and ... binding-n ]
```

備註

先前的語法會出現在類別標題和繼承宣告之後，但在任何成員定義之前。語法類似于 `let` 類別外的系結，但是在類別中定義的名稱具有限制為類別的範圍。`let` 系結會建立私用欄位或函式；若要公開資料或函數，請宣告屬性或成員方法。

不是靜態的系結稱為「實例系 `let` 結」(instance binding)。`let` 建立 `let` 物件時，會執行實例系結。靜態 `let` 系結是類別之靜態初始化運算式的一部分，保證會在第一次使用型別之前執行。

實例 `let` 系結中的程式碼可以使用主要的函式的參數。

類別中的系結不允許 `let` 屬性和存取範圍修飾詞。

下列程式碼範例說明類別中的 `let` 數種系結類型。

```
type PointWithCounter(a: int, b: int) =  
    // A variable i.  
    let mutable i = 0  
  
    // A let binding that uses a pattern.  
    let (x, y) = (a, b)  
  
    // A private function binding.  
    let privateFunction x y = x * x + 2*y  
  
    // A static let binding.  
    static let mutable count = 0  
  
    // A do binding.  
    do  
        count <- count + 1  
  
    member this.Prop1 = x  
    member this.Prop2 = y  
    member this.CreatedCount = count  
    member this.FunctionValue = privateFunction x y  
  
let point1 = PointWithCounter(10, 52)  
  
printfn "%d %d %d %d" (point1.Prop1) (point1.Prop2) (point1.CreatedCount) (point1.FunctionValue)
```

輸出如下。

建立欄位的替代方式

您也可以使用 `val` 關鍵字來建立私用欄位。使用 `val` 關鍵字時，不會在建立物件時指定欄位的值，而是使用預設值來初始化。如需詳細資訊，[請參閱明確欄位:Val 關鍵字](#)。

您也可以使用成員定義來定義類別中的私用欄位，並將關鍵字 `private` 新增至定義。如果您想要變更成員的存取範圍，而不需要重寫程式碼，這會很有用。如需詳細資訊，請參閱[存取控制](#)。

另請參閱

- [成員](#)
- [類別中的 `do` 繫結](#)
- [let 關係](#)

類別中的 do 繫結

2019/10/23 • [Edit Online](#)

類別定義中的系結會在物件建立時執行動作, 或在第一 `do` 次使用該類型時, 針對靜態系結。 `do`

語法

```
[static] do expression
```

備註

系結會與系結一起 `let` 出現, 但在類別定義中的成員定義之前。 `do` 雖然關鍵字對於 `do` 模組層級的系結是選擇性的, 但對於 `do` 類別定義中的系結而言, 並不是選擇性的。 `do`

針對任何給定類型的每個物件的結構, 非靜態 `do` 系結和非靜態 `let` 系結會依照它們出現在類別定義中的順序來執行。一個 `do` 類型中可能會發生多個系結。非靜態 `let` 系結和非靜態 `do` 系結會成為主要函式的主體。[非靜態 `do` 系結] 區段中的程式碼可以參考主要的函式參數, 以及系結區段 `let` 中所定義的任何值或函數。

只要類別具有 `do` 類別標題中 `as` 關鍵字所定義的自我識別碼, 而且這些成員的所有用法都是以類別的自我識別碼限定, 非靜態系結就可以存取類別的成員。

由於 `let` 系結會初始化類別的私用欄位, 因此通常必須確保成員如預期般運作, 而且 `do` 系結通常會在 `let` 系結之後放置, 讓 `do` 系結中的程式碼可以使用完全初始化的物件來執行。如果您的程式碼在初始化完成之前嘗試使用成員, 則會引發 `InvalidOperationException`。

靜態 `do` 系結可以參考封閉式類別的靜態成員或欄位, 但不能參考實例成員或欄位。靜態 `do` 系結會成為類別靜態初始化運算式的一部分, 保證會在第一次使用類別之前執行。

類型中的 `do` 系結會忽略屬性。如果在系結中 `do` 執行的程式碼需要屬性, 則必須將它套用至主要的函式。

在下列程式碼中, 類別具有靜態 `do` 系結和非靜態 `do` 系結。物件有一個具有兩個參數 (`a` 和 `b`) 的函式, 以及兩個私用欄位 `let` 定義于類別的系結中。也會定義兩個屬性。這些全都位於 [非靜態 `do` 系結] 區段的 [範圍] 中, 如列印所有這些值的程式碼所示。

```
open System

type MyType(a:int, b:int) as this =
    inherit Object()
    let x = 2*a
    let y = 2*b
    do printfn "Initializing object %d %d %d %d %d %d"
        a b x y (this.Prop1) (this.Prop2)
    static do printfn "Initializing MyType."
    member this.Prop1 = 4*x
    member this.Prop2 = 4*y
    override this.ToString() = System.String.Format("{0} {1}", this.Prop1, this.Prop2)

let obj1 = new MyType(1, 2)
```

輸出如下。

```
Initializing MyType.  
Initializing object 1 2 2 4 8 16
```

另請參閱

- [成員](#)
- [類別](#)
- [建構函式](#)
- [類別中的](#) `let` [繫結](#)
- `do` [關係](#)

屬性

2021/3/5 • [Edit Online](#)

屬性 是代表與物件相關聯之值的成員。

語法

```
// Property that has both get and set defined.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with [accessibility-modifier] get() =
    get-function-body
and [accessibility-modifier] set parameter =
    set-function-body

// Alternative syntax for a property that has get and set.
[ attributes-for-get ]
[ static ] member [accessibility-modifier-for-get] [self-identifier.]PropertyName =
    get-function-body
[ attributes-for-set ]
[ static ] member [accessibility-modifier-for-set] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName =
    get-function-body

// Alternative syntax for property that has get only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with get() =
    get-function-body

// Property that has set only.
[ attributes ]
[ static ] member [accessibility-modifier] [self-identifier.]PropertyName
with set parameter =
    set-function-body

// Automatically implemented properties.
[ attributes ]
[ static ] member val [accessibility-modifier] PropertyName = initialization-expression [ with get, set ]
```

備註

屬性代表物件導向程式設計中的「具有」關聯性，表示與物件實例相關聯的資料，或具有類型的靜態屬性。

您可以用兩種方式宣告屬性，取決於您是否要明確指定基礎值（也稱為屬性的備份存放區），或者，如果您想要讓編譯器自動為您產生備份存放區。一般而言，如果屬性具有非一般的實值，以及當屬性只是值或變數的簡單包裝函式時，您應該使用更明確的方式。若要明確宣告屬性，請使用 `member` 關鍵字。此宣告式語法後面接著指定 `get` 和 `set` 方法（也稱為 存取子）的語法。語法區段中顯示的各種明確語法形式適用於讀/寫、唯讀和僅限寫入屬性。若為唯讀屬性，您只需定義 `get` 方法；對於僅限寫入屬性，只定義 `set` 方法。請注意，當屬性同時具有 `get` 和 `set` 存取子時，替代語法可讓您針對每個存取子指定不同的屬性和存取範圍修飾詞，如下列程式碼所示。

```
// A read-only property.
member this.MyReadOnlyProperty = myInternalValue
// A write-only property.
member this.MyWriteOnlyProperty with set (value) = myInternalValue <- value
// A read-write property.
member this.MyReadWriteProperty
    with get () = myInternalValue
    and set (value) = myInternalValue <- value
```

若為具有和方法的讀取/寫入屬性 `get`，`set` 則和的順序 `get` `set` 可以反轉。或者，您可以只提供顯示的語法 `get`，以及顯示的語法，`set` 而不是使用結合的語法。這麼做可讓您更輕鬆地將個人 `get` 或 `set` 方法批註起來，如果這是您可能需要做的事情。下列程式碼顯示使用合併語法的替代方法。

```
member this.MyReadWriteProperty with get () = myInternalValue
member this.MyReadWriteProperty with set (value) = myInternalValue <- value
```

保存屬性資料的私用值稱為「**備份存放區**」。若要讓編譯器自動建立備份存放區，請使用關鍵字 `member val`，省略自我識別碼，然後提供運算式來初始化屬性。如果屬性是可變動的，請包含 `with get, set`。例如，下列類別型別包含兩個自動執行的屬性。`Property1` 是唯讀的，而且會初始化為提供給主要函式的引數，而且 `Property2` 是可設定的屬性，初始化為空字串：

```
type MyClass(property1 : int) =
    member val Property1 = property1
    member val Property2 = "" with get, set
```

自動執行的屬性是類型初始化的一部分，因此必須在任何其他成員定義之前包含它們，就像型別 `let` 定義中的系結和系結一樣 `do`。請注意，初始化自動執行之屬性的運算式只會在初始化時進行評估，而不是在每次存取屬性時進行評估。這種行為與明確執行之屬性的行為不同。這實際上是指將這些屬性初始化的程式碼會加入至類別的函式。請考慮下列顯示這項差異的程式碼：

```
type MyClass() =
    let random = new System.Random()
    member val AutoProperty = random.Next() with get, set
    member this.ExplicitProperty = random.Next()

let class1 = new MyClass()

printfn $"class1.AutoProperty = {class1.AutoProperty}"
printfn $"class1.ExplicitProperty = {class1.ExplicitProperty}"
```

輸出

```
class1.AutoProperty = 1853799794
class1.AutoProperty = 1853799794
class1.ExplicitProperty = 978922705
class1.ExplicitProperty = 1131210765
```

上述程式碼的輸出顯示，`AutoProperty` 的值會在重複呼叫時保持不變，而 `ExplicitProperty` 會在每次呼叫時變更。這會示範每次不會評估自動執行之屬性的運算式，如同明確屬性的 `getter` 方法。

WARNING

有一些程式庫(例如 `Entity Framework ()`)會在基類的函式 `System.Data.Entity` 中執行自訂作業，而這些函式在初始化自動執行的屬性時無法正常運作。在這些情況下，請嘗試使用明確的屬性。

屬性可以是類別、結構、區分等位、記錄、介面和類型延伸的成員，也可以在物件運算式中定義。

屬性可以套用至屬性。若要將屬性 (attribute) 套用至屬性 (attribute)，請將屬性 (attribute) 上的屬性 (property)。如需詳細資訊，請參閱[屬性](#)。

依預設，屬性為 public。存取範圍修飾詞也可以套用至屬性。若要套用協助工具修飾詞，請在屬性名稱前面加入它 (如果它要同時套用到 `get` 和方法) `set` ; `get` `set` 如果每個存取子需要不同的協助工具，請在和關鍵字之前加入。存取範圍修飾詞可以是下列其中一項：`public`、`private`、`internal`。如需詳細資訊，請參閱[存取控制](#)。

每次存取屬性時，就會執行屬性執行。

靜態和實例屬性

屬性可以是靜態或實例屬性。靜態屬性可以在沒有實例的情況下叫用，並用於與類型相關聯的值，而不是個別的物件。若為靜態屬性，請省略自我識別碼。實例屬性需要自我識別碼。

下列靜態屬性定義是以您有靜態欄位的案例為基礎，而該欄位 `myStaticValue` 是屬性的備份存放區。

```
static member MyStaticProperty
    with get() = myStaticValue
    and set(value) = myStaticValue <- value
```

屬性也可以是類似陣列的，在這種情況下，它們稱為 [索引/屬性](#)。如需詳細資訊，請參閱[索引屬性](#)。

屬性的類型注釋

在許多情況下，編譯器會有足夠的資訊從支援存放區的型別推斷屬性型別，但是您可以藉由加入型別注釋來明確設定型別。

```
// To apply a type annotation to a property that does not have an explicit
// get or set, apply the type annotation directly to the property.
member this.MyProperty1 : int = myInternalValue
// If there is a get or set, apply the type annotation to the get or set method.
member this.MyProperty2 with get() : int = myInternalValue
```

使用屬性集存取子

您可以使用運算子來設定提供存取子的屬性 `set` `<-`。

```
// Assume that the constructor argument sets the initial value of the
// internal backing store.
let mutable myObject = new MyType(10)
myObject.MyProperty <- 20
printfn "%d" (myObject.MyProperty)
```

輸出為 20。

抽象屬性

屬性可以是抽象的。如同方法一樣，這 `abstract` 表示有與屬性相關聯的虛擬分派。抽象屬性可以是真正的抽象概念，也就是沒有相同類別中的定義。因此，包含這類屬性的類別就是抽象類別。或者，`abstract` 也可以表示屬性是虛擬的，而且在這種情況下，定義必須存在於相同的類別中。請注意，抽象屬性不得為私用，而且如果一個存取子是抽象的，另一個存取子也必須是抽象的。如需抽象類別的詳細資訊，請參閱[抽象類別](#)。

```
// Abstract property in abstract class.
// The property is an int type that has a get and
// set method
[<AbstractClass>]
type AbstractBase() =
  abstract Property1 : int with get, set

// Implementation of the abstract property
type Derived1() =
  inherit AbstractBase()
  let mutable value = 10
  override this.Property1 with get() = value and set(v : int) = value <- v

// A type with a "virtual" property.
type Base1() =
  let mutable value = 10
  abstract Property1 : int with get, set
  default this.Property1 with get() = value and set(v : int) = value <- v

// A derived type that overrides the virtual property
type Derived2() =
  inherit Base1()
  let mutable value2 = 11
  override this.Property1 with get() = value2 and set(v) = value2 <- v
```

另請參閱

- [成員](#)
- [方法](#)

索引屬性

2019/11/25 • [Edit Online](#)

在定義可抽象化已排序資料的類別時，有時會很有說明，提供該資料的索引存取權，而不會公開基礎的執行。這是使用 `Item` 成員來完成。

語法

```
// Indexed property that can be read and written to
member self-identifier.Item
  with get(index-values) =
    get-member-body
  and set index-values values-to-set =
    set-member-body

// Indexed property can only be read
member self-identifier.Item
  with get(index-values) =
    get-member-body

// Indexed property that can only be set
member self-identifier.Item
  with set index-values values-to-set =
    set-member-body
```

備註

上述語法的形式顯示如何定義具有 `get` 和 `set` 方法的索引屬性、僅具有 `get` 方法，或僅具有 `set` 方法。您也可以結合僅供 `get` 使用的語法和針對 `set` 所顯示的語法，並產生同時具有 `get` 和 `set` 的屬性。第二個表單可讓您將不同的存取範圍修飾詞和屬性放在 `get` 和 `set` 方法上。

藉由使用 `Item` 的名稱，編譯器會將屬性視為預設的索引屬性。*預設的索引屬性*是您可以在物件實例上使用類似陣列的語法來存取的屬性。例如，如果 `o` 是定義此屬性之類型的物件，則會使用語法 `o.[index]` 來存取屬性。

存取非預設索引屬性的語法是在括弧中提供屬性名稱和索引，就像一般成員一樣。例如，如果 `o` 上的屬性呼叫 `Ordinal`，您會撰寫 `o.Ordinal(index)` 來存取它。

無論您使用哪一種形式，都應該一律在索引屬性上使用 `set` 方法的擴充形式。如需擴充函數的詳細資訊，[請參閱函式](#)。

範例

下列程式碼範例說明如何定義和使用具有 `get` 和 `set` 方法的預設和非預設索引屬性。

```

type NumberStrings() =
  let mutable ordinals = [| "one"; "two"; "three"; "four"; "five";
                           "six"; "seven"; "eight"; "nine"; "ten" |]
  let mutable cardinals = [| "first"; "second"; "third"; "fourth";
                             "fifth"; "sixth"; "seventh"; "eighth";
                             "ninth"; "tenth" |]

  member this.Item
    with get(index) = ordinals.[index]
    and set index value = ordinals.[index] <- value
  member this.Ordinal
    with get(index) = ordinals.[index]
    and set index value = ordinals.[index] <- value
  member this.Cardinal
    with get(index) = cardinals.[index]
    and set index value = cardinals.[index] <- value

let nstrs = new NumberStrings()
nstrs.[0] <- "ONE"
for i in 0 .. 9 do
  printf "%s " (nstrs.[i])
printfn ""

nstrs.Cardinal(5) <- "6th"

for i in 0 .. 9 do
  printf "%s " (nstrs.Ordinal(i))
  printf "%s " (nstrs.Cardinal(i))
printfn ""

```

Output

```

ONE two three four five six seven eight nine ten
ONE first two second three third four fourth five fifth six 6th
seven seventh eight eighth nine ninth ten tenth

```

具有多個索引值的索引屬性

已編制索引的屬性可以有一個以上的索引值。在此情況下，使用屬性時，值會以逗號分隔。這類屬性中的 `set` 方法必須有兩個擴充引數，第一個是包含索引鍵的元組，而第二個是要設定的值。

下列程式碼示範如何使用具有多個索引值的索引屬性。

```

open System.Collections.Generic

/// Basic implementation of a sparse matrix based on a dictionary
type SparseMatrix() =
  let table = new Dictionary<(int * int), float>()
  member _.Item
    // Because the key is comprised of two values, 'get' has two index values
    with get(key1, key2) = table.[(key1, key2)]

    // 'set' has two index values and a new value to place in the key's position
    and set (key1, key2) value = table.[(key1, key2)] <- value

let sm = new SparseMatrix()
for i in 1..1000 do
  sm.[i, i] <- float i * float i

```

請參閱

- 成員

方法

2020/11/2 • [Edit Online](#)

方法是與型別相關聯的函式。在物件導向程式設計中，會使用方法來公開和執行物件和類型的功能和行為。

語法

```
// Instance method definition.
[ attributes ]
member [inline] self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Static method definition.
[ attributes ]
static member [inline] method-name parameter-list [ : return-type ] =
    method-body

// Abstract method declaration or virtual dispatch slot.
[ attributes ]
abstract member method-name : type-signature

// Virtual method declaration and default implementation.
[ attributes ]
abstract member method-name : type-signature
[ attributes ]
default self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Override of inherited virtual method.
[ attributes ]
override self-identifier.method-name parameter-list [ : return-type ] =
    method-body

// Optional and DefaultParameterValue attributes on input parameters
[ attributes ]
[ modifier ] member [inline] self-identifier.method-name ([<Optional; DefaultParameterValue( default-value
)>] input) [ : return-type ]
```

備註

在先前的語法中，您可以看到各種形式的方法宣告和定義。在較長的方法主體中，換行字元會在等號(=)之後，而整個方法主體會縮排。

屬性可以套用至任何方法宣告。它們優先於方法定義的語法，且通常會列在個別行上。如需詳細資訊，請參閱[屬性](#)。

可以標示方法 `inline`。如需 `inline` 的資訊，請參閱[內嵌函式](#)。

非內嵌方法可以在型別內以遞迴方式使用;不需要明確使用 `rec` 關鍵字。

實例方法

實例方法是以 `member` 關鍵字和 *自我識別碼* 宣告，後面接著一個句點(.)和方法名稱和參數。如同系結的情況 `let`，參數清單可以是模式。通常，您會在元組形式的括弧中括住方法參數，這是方法在以其他 .NET Framework 語言建立時，在 F# 中的顯示方式。不過，局部分隔形式(參數以空格分隔)也很常見，而且也支援其他模式。

下列範例說明非抽象實例方法的定義和使用方式。

```
type SomeType(factor0: int) =  
    let factor = factor0  
    member this.SomeMethod(a, b, c) =  
        (a + b + c) * factor  
  
    member this.SomeOtherMethod(a, b, c) =  
        this.SomeMethod(a, b, c) * factor
```

在實例方法中，請勿使用自我識別碼來存取使用 `let` 系結所定義的欄位。存取其他成員和屬性時，請使用自我識別碼。

靜態方法

關鍵字 `static` 是用來指定在沒有實例的情況下呼叫方法，而且不會與物件實例相關聯。否則，方法是實例方法。

下一節中的範例會顯示使用關鍵字宣告的欄位 `let`、使用關鍵字宣告的屬性成員 `member`，以及使用關鍵字宣告的靜態方法 `static`。

下列範例說明靜態方法的定義和使用方式。假設這些方法定義是在 `SomeType` 上一節的類別中。

```
static member SomeStaticMethod(a, b, c) =  
    (a + b + c)  
  
static member SomeOtherStaticMethod(a, b, c) =  
    SomeType.SomeStaticMethod(a, b, c) * 100
```

抽象和虛擬方法

關鍵字 `abstract` 指出方法具有虛擬分派位置，而且在類別中可能沒有定義。*虛擬分派位置*是在內部維護的函式資料表中的專案，可在執行時間用來查詢物件導向型別中的虛擬函式呼叫。虛擬分派機制是實架構的 *機制*，這是物件導向程式設計的一項重要功能。至少有一個沒有定義之抽象方法的類別是 *抽象類別*，這表示不可以建立該類別的實例。如需抽象類別的詳細資訊，請參閱 [抽象類別](#)。

抽象方法宣告不包含方法主體。相反地，方法的名稱後面會接著冒號 (:) 和方法的類型簽章。當您將滑鼠指標暫停 Visual Studio Code 編輯器中的方法名稱上方 (但不含參數名稱) 時，方法的類型簽章與 IntelliSense 所顯示的類型簽章相同。當您以互動方式工作時，解譯器也會顯示型別簽章 (fsi.exe)。方法的類型簽章是藉由列出參數的類型，後面接著傳回型別，並以適當的分隔符號號來形成。局部擴充參數會以分隔 `->`，而且元組參數會以分隔 `*`。傳回值一律與引數以 `->` 符號分隔。括弧可以用來將複雜參數分組，例如當函式類型是參數時，或指出元組視為單一參數，而不是兩個參數。

您也可以將定義新增至類別，並使用 `default` 關鍵字 (如本主題中的語法區塊所示)，以提供抽象方法的預設定義。在相同類別中具有定義的抽象方法，相當於其他 .NET Framework 語言中的虛擬方法。無論定義是否存在，`abstract` 關鍵字都會在類別的虛擬函式資料表中建立新的分派位置。

無論基類是否實作為抽象方法，衍生類別都可以提供抽象方法的執行。若要在衍生類別中執行抽象方法，請在衍生類別中定義具有相同名稱和簽章的方法 (使用 `override` 或關鍵字除外 `default`)，並提供方法主體。關鍵字 `override` 和 `default` 表示完全相同的內容。`override` 如果新的方法會覆寫基類執行，請使用，`default` 當您在與原始抽象宣告相同的類別中建立實值時，請使用。請勿 `abstract` 在執行基類中宣告為抽象方法的方法上使用關鍵字。

下列範例說明 `Rotate` 具有預設實值的抽象方法，相當於 .NET Framework 的虛擬方法。

```
type Ellipse(a0 : float, b0 : float, theta0 : float) =  
    let mutable axis1 = a0  
    let mutable axis2 = b0  
    let mutable rotAngle = theta0  
    abstract member Rotate: float -> unit  
    default this.Rotate(delta : float) = rotAngle <- rotAngle + delta
```

下列範例說明會覆寫基類方法的衍生類別。在此情況下，覆寫會變更行為，使該方法不會執行任何動作。

```
type Circle(radius : float) =  
    inherit Ellipse(radius, radius, 0.0)  
    // Circles are invariant to rotation, so do nothing.  
    override this.Rotate(_) = ()
```

多載方法

多載方法是在指定類型中具有相同名稱但有不同引數的方法。在 F # 中，通常會使用選擇性引數，而不是多載的方法。但是，如果引數是採用元組格式，而非局部引數，則允許在語言中使用多載的方法。

選擇性引數

從 F # 4.1 開始，您也可以方法中有選擇性的引數搭配預設參數值。這是為了協助加速與 c # 程式碼的交互操作。下列範例示範語法：

```
// A class with a method M, which takes in an optional integer argument.  
type C() =  
    _ .M([<Optional; DefaultValue(12)>] i) = i + 1
```

請注意，傳入的值 `DefaultValue` 必須符合輸入類型。在上述範例中，它是 `int`。嘗試傳遞非整數值 `DefaultValue` 會導致編譯錯誤。

範例：屬性和方法

下列範例包含型別，其中包含欄位、私用函式、屬性和靜態方法的範例。

```

type RectangleXY(x1 : float, y1: float, x2: float, y2: float) =
  // Field definitions.
  let height = y2 - y1
  let width = x2 - x1
  let area = height * width
  // Private functions.
  static let maxFloat (x: float) (y: float) =
    if x >= y then x else y
  static let minFloat (x: float) (y: float) =
    if x <= y then x else y
  // Properties.
  // Here, "this" is used as the self identifier,
  // but it can be any identifier.
  member this.X1 = x1
  member this.Y1 = y1
  member this.X2 = x2
  member this.Y2 = y2
  // A static method.
  static member intersection(rect1 : RectangleXY, rect2 : RectangleXY) =
    let x1 = maxFloat rect1.X1 rect2.X1
    let y1 = maxFloat rect1.Y1 rect2.Y1
    let x2 = minFloat rect1.X2 rect2.X2
    let y2 = minFloat rect1.Y2 rect2.Y2
    let result : RectangleXY option =
      if ( x2 > x1 && y2 > y1) then
        Some (RectangleXY(x1, y1, x2, y2))
      else
        None
    result

// Test code.
let testIntersection =
  let r1 = RectangleXY(10.0, 10.0, 20.0, 20.0)
  let r2 = RectangleXY(15.0, 15.0, 25.0, 25.0)
  let r3 : RectangleXY option = RectangleXY.intersection(r1, r2)
  match r3 with
  | Some(r3) -> printfn "Intersection rectangle: %f %f %f %f" r3.X1 r3.Y1 r3.X2 r3.Y2
  | None -> printfn "No intersection found."

testIntersection

```

另請參閱

- [成員](#)

建構函式

2020/11/2 • [Edit Online](#)

本文說明如何定義和使用函式來建立和初始化類別和結構物件。

類別物件的結構

類別類型的物件具有函數。有兩種類型的函數。其中一個是主要的函式，其參數會出現在型別名稱後面的括弧中。您可以使用關鍵字來指定其他選擇性的其他函式 `new`。任何這類額外的函式都必須呼叫主要的函式。

主要的函式包含和系結 `let` `do`，它會出現在類別定義的開頭。系結會宣告 `let` 類別的私用欄位和方法，系結會 `do` 執行程式碼。如需類別函式中系結的詳細資訊 `let`，請參閱 [let 類別中的系結](#)。如需有關在函式中系結的詳細資訊 `do`，請參閱 [do 類別中的系結](#)。

不論您想要呼叫的是主要的函式或其他的函數，您都可以使用運算式來建立物件 `new`，並搭配或不搭配選擇性 `new` 關鍵字。您可以使用函式引數來初始化您的物件，方法是依序列出引數，並以逗號分隔並括在括弧中，或使用括弧中的具名引數和值。您也可以在物件的結構中，使用屬性名稱和指派值來設定物件的屬性，就如同使用命名的函式引數一樣。

下列程式碼說明具有函式和各種建立物件方法的類別：

```
// This class has a primary constructor that takes three arguments
// and an additional constructor that calls the primary constructor.
type MyClass(x0, y0, z0) =
  let mutable x = x0
  let mutable y = y0
  let mutable z = z0
  do
    printfn "Initialized object that has coordinates (%d, %d, %d)" x y z
  member this.X with get() = x and set(value) = x <- value
  member this.Y with get() = y and set(value) = y <- value
  member this.Z with get() = z and set(value) = z <- value
  new() = MyClass(0, 0, 0)

// Create by using the new keyword.
let myObject1 = new MyClass(1, 2, 3)
// Create without using the new keyword.
let myObject2 = MyClass(4, 5, 6)
// Create by using named arguments.
let myObject3 = MyClass(x0 = 7, y0 = 8, z0 = 9)
// Create by using the additional constructor.
let myObject4 = MyClass()
```

輸出如下所示：

```
Initialized object that has coordinates (1, 2, 3)
Initialized object that has coordinates (4, 5, 6)
Initialized object that has coordinates (7, 8, 9)
Initialized object that has coordinates (0, 0, 0)
```

結構結構

結構會遵循類別的所有規則。因此，您可以擁有主要的函式，也可以使用來提供其他的函式 `new`。不過，結構和類別之間有一個重要的差異：結構可以有無參數的函式（也就是沒有引數的函式），即使未定義主要的函式。無

參數的函式會將所有欄位初始化為該類型的預設值，通常為零或其相等。您為結構定義的任何函式都必須至少有一個引數，才不會與無參數的處理常式衝突。

此外，結構通常會有使用關鍵字建立的欄位 `val`；類別也可以有這些欄位。具有使用關鍵字定義之欄位的結構和類別 `val`，也可以使用記錄運算式在其他的函式中進行初始化，如下列程式碼所示。

```
type MyStruct =  
  struct  
    val X : int  
    val Y : int  
    val Z : int  
    new(x, y, z) = { X = x; Y = y; Z = z }  
  end  
  
let myStructure1 = new MyStruct(1, 2, 3)
```

如需詳細資訊，請參閱[明確欄位](#)：`val` 關鍵字。

在函式中執行副作用

類別中的主要函式可以在系結中執行程式碼 `do`。不過，如果您必須在不使用系結的其他函式中執行程式碼，該怎麼辦 `do`？若要這麼做，請使用 `then` 關鍵字。

```
// Executing side effects in the primary constructor and  
// additional constructors.  
type Person(nameIn : string, idIn : int) =  
  let mutable name = nameIn  
  let mutable id = idIn  
  do printfn "Created a person object."  
  member this.Name with get() = name and set(v) = name <- v  
  member this.ID with get() = id and set(v) = id <- v  
  new() =  
    Person("Invalid Name", -1)  
  then  
    printfn "Created an invalid person object."  
  
let person1 = new Person("Humberto Acevedo", 123458734)  
let person2 = new Person()
```

主要的函式的副作用仍會執行。因此，輸出如下所示：

```
Created a person object.  
Created a person object.  
Created an invalid person object.
```

為什麼 `then` 需要而不是另一個的原因 `do` 是，`do` 當關鍵字存在於其他的函式 `unit` 主體中時，關鍵字會有其標準意義來分隔傳回的運算式。它在主要的函式內容中只具有特殊意義。

函式中的自我識別碼

在其他成員中，您會在每個成員的定義中提供目前物件的名稱。您也可以使用緊接在函式參數後面的關鍵字，將自我識別碼放在類別定義的第一行 `as`。下列範例說明此語法。

```

type MyClass1(x) as this =
  // This use of the self identifier produces a warning - avoid.
  let x1 = this.X
  // This use of the self identifier is acceptable.
  do printfn "Initializing object with X =%d" this.X
  member this.X = x

```

在其他的函式中，您也可以將子句放在函式參數後面，以定義自我識別碼 `as`。下列範例說明這個語法：

```

type MyClass2(x : int) =
  member this.X = x
  new() as this = MyClass2(0) then printfn "Initializing with X = %d" this.X

```

當您嘗試在完全定義物件之前使用它，可能會發生問題。因此，使用自我識別碼可能會導致編譯器發出警告，並插入額外的檢查，以確保物件的成員在初始化之前不會被存取。您應該只在主要函式的系結中 `do`，或在其他函式中的關鍵字之後，使用自我識別碼 `then`。

本身識別碼的名稱不一定要是 `this`。它可以是任何有效的識別碼。

在初始化時將值指派給屬性

您可以將表單指派清單附加至函式 `property = value` 的引數清單，以將值指派給初始化程式碼中的類別物件屬性。作法如下列的程式碼範例所示：

```

type Account() =
  let mutable balance = 0.0
  let mutable number = 0
  let mutable firstName = ""
  let mutable lastName = ""
  member this.AccountNumber
    with get() = number
    and set(value) = number <- value
  member this.FirstName
    with get() = firstName
    and set(value) = firstName <- value
  member this.LastName
    with get() = lastName
    and set(value) = lastName <- value
  member this.Balance
    with get() = balance
    and set(value) = balance <- value
  member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
  member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount

let account1 = new Account(AccountNumber=8782108,
                           FirstName="Darren", LastName="Parker",
                           Balance=1543.33)

```

下列版本的舊版程式碼說明了一般引數、選擇性引數和屬性設定在一個函式呼叫中的組合：


```
type Account(accountNumber : int, ?first: string, ?last: string, ?bal : float) =  
  let mutable balance = defaultArg bal 0.0  
  let mutable number = accountNumber  
  let mutable firstName = defaultArg first ""  
  let mutable lastName = defaultArg last ""  
  member this.AccountNumber  
    with get() = number  
    and set(value) = number <- value  
  member this.FirstName  
    with get() = firstName  
    and set(value) = firstName <- value  
  member this.LastName  
    with get() = lastName  
    and set(value) = lastName <- value  
  member this.Balance  
    with get() = balance  
    and set(value) = balance <- value  
  member this.Deposit(amount: float) = this.Balance <- this.Balance + amount  
  member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount  
  
let account1 = new Account(8782108, bal = 543.33,  
                           FirstName="Raman", LastName="Iyer")
```

繼承類別中的函式

從具有函數的基類繼承時，您必須在繼承子句中指定其引數。如需詳細資訊，請參閱[函數和繼承](#)。

靜態的函式或類型的構造函式

除了指定建立物件的程式碼之外，`let` `do` 也可以在第一次使用類型來執行類型層級的初始化之前，在執行的類別類型中撰寫靜態和系結。如需詳細資訊，請參閱類別中 `let` [類別](#)和系結 `do` [中的系結](#)。

另請參閱

- [成員](#)

事件

2021/3/5 • [Edit Online](#)

事件可讓您產生函式呼叫與使用者動作的關聯，而且對 GUI 程式設計而言十分重要。事件也可以由應用程式或作業系統觸發。

處理事件

當您使用 GUI 程式庫如 Windows Forms 或 Windows Presentation Foundation (WPF) 時，應用程式中的大部分程式碼都會執行以回應程式庫預先定義的事件。這些預先定義的事件是 GUI 類別的成員，例如表單和控制項。您可以參考特定重要具名事件 (例如 `Click` 類別的 `Form` 事件)，以及叫用 `Add` 方法，將自訂行為加入至預先存在的事件 (例如按一下按鈕事件)，如下列程式碼所示。如果您從 F# Interactive 執行此程式碼，請省略

`System.Windows.Forms.Application.Run(System.Windows.Forms.Form)` 的呼叫。

```
open System.Windows.Forms

let form = new Form(Text="F# Windows Form",
                    Visible = true,
                    TopMost = true)

form.Click.Add(fun evArgs -> System.Console.Beep())
Application.Run(form)
```

`Add` 方法的類型為 `('a -> unit) -> unit`。因此，事件處理常式方法會接受一個參數 (通常是事件引數)，並傳回 `unit`。上述範例示範事件處理常式做為 Lambda 運算式。事件處理常式也可以是函式值，如下列程式碼範例所示。範例中也會示範事件處理常式參數如何用來提供事件類型專屬資訊。對於 `MouseMove` 事件，系統會傳遞 `System.Windows.Forms.MouseEventArgs` 物件，其中包含指標的 `X` 和 `Y` 位置。

```
open System.Windows.Forms

let Beep evArgs =
    System.Console.Beep( )

let form = new Form(Text = "F# Windows Form",
                    Visible = true,
                    TopMost = true)

let MouseMoveEventHandler (evArgs : System.Windows.Forms.MouseEventArgs) =
    form.Text <- System.String.Format("{0},{1}", evArgs.X, evArgs.Y)

form.Click.Add(Beep)
form.MouseMove.Add(MouseMoveEventHandler)
Application.Run(form)
```

建立自訂事件

F# 事件是以 F# `事件` 類型表示，它會實 `IEvent` 介面。`IEvent` 本身是結合兩個其他介面的功能和 `IDelegateEvent` 的介面 `System.IObservable<'T>`。`IDelegateEvent` 因此，`Event` 具有相當於其他語言中委派的功能，加上來自 `IObservable` 的額外功能，這表示 F# 事件支援事件篩選，以及使用 F# 第一級函式和 Lambda 運算式做為事件處理常式。這項功能是在 [事件模組](#) 中提供。

若要在類別上建立其運作方式與任何其他 .NET Framework 事件類似的事件，請將 `let` 繫結 (將 `Event` 定義為

類別中的欄位) 加入至類別。您可以指定所需的事件引數類型做為型別引數, 或是空過, 讓編譯器推斷適當類型。您也必須定義將事件公開為 CLI 事件的事件成員。這個成員應具有 `CLIEvent` 屬性。它的宣告方式與屬性類似, 而且其實作只是呼叫事件的 `Publish` 屬性。您類別的使用者可以使用已發行事件的 `Add` 方法加入處理常式。`Add` 方法的引數可以是 Lambda 運算式。您可以使用事件的 `Trigger` 屬性引發事件, 將引數傳遞至處理函式。下列程式碼範例會說明這點。在此範例中, 推斷的事件類型引數是 `Tuple`, 代表 Lambda 運算式的引數。

```
open System.Collections.Generic

type MyClassWithCLIEvent() =

    let event1 = new Event<_>()

    [<CLIEvent>]
    member this.Event1 = event1.Publish

    member this.TestEvent(arg) =
        event1.Trigger(this, arg)

let classWithEvent = new MyClassWithCLIEvent()
classWithEvent.Event1.Add(fun (sender, arg) ->
    printfn "Event1 occurred! Object data: %s" arg)

classWithEvent.TestEvent("Hello World!")

System.Console.ReadLine() |> ignore
```

輸出如下。

```
Event1 occurred! Object data: Hello World!
```

這裡會說明 `Event` 模組所提供的額外功能。下列程式碼範例說明如何使用 `Event.create` 建立事件和觸發程序方法, 並以 Lambda 運算式形式加入兩個事件處理常式, 然後觸發事件以執行這兩個 Lambda 運算式。

```
type MyType() =
    let myEvent = new Event<_>()

    member this.AddHandlers() =
        Event.add (fun string1 -> printfn "%s" string1) myEvent.Publish
        Event.add (fun string1 -> printfn "Given a value: %s" string1) myEvent.Publish

    member this.Trigger(message) =
        myEvent.Trigger(message)

let myMyType = MyType()
myMyType.AddHandlers()
myMyType.Trigger("Event occurred.")
```

上述程式碼的輸出如下。

```
Event occurred.
Given a value: Event occurred.
```

處理事件資料流

您可以使用模組中的函式, `Event.add` `Event` 以高度自訂的方式處理事件的資料流程, 而不只是使用事件加入事件的事件處理常式。若要這麼做, 請使用正向管道 (`|>`) 與事件做為一連串函式呼叫中的第一個值, 並且使用 `Event` 模組函式做為後續函式呼叫。

下列程式碼範例顯示如何設定只在特定條件下才會呼叫其處理常式的事件。

```
let form = new Form(Text = "F# Windows Form",  
                    Visible = true,  
                    TopMost = true)  
form.MouseMove  
|> Event.filter ( fun evArgs -> evArgs.X > 100 && evArgs.Y > 100)  
|> Event.add ( fun evArgs ->  
    form.BackColor <- System.Drawing.Color.FromArgb(  
        evArgs.X, evArgs.Y, evArgs.X ^^ evArgs.Y ) )
```

可 [觀察模組](#) 包含可在可觀察物件上運作的類似函式。可預見物件與事件類似，但是只有在已訂閱可預見物件時，才會主動訂閱事件。

實作介面事件

當您開發 UI 元件時，通常會從建立新表單或繼承自現有表單或控制項的新控制項開始進行。事件經常是在介面上定義，在這種情況下，您必須實作介面才能實作事件。`System.ComponentModel.INotifyPropertyChanged` 介面會定義單一 `System.ComponentModel.INotifyPropertyChanged.PropertyChanged` 事件。下列程式碼將示範如何實作這個繼承的介面所定義的事件：

```

module CustomForm

open System.Windows.Forms
open System.ComponentModel

type AppForm() as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs ->
            this.Property1 <- "text2"
            this.Property2 <- "text3")

    // This property does not have the property-changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property-changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

    // Expose the PropertyChanged event as a first class .NET event.
    [<CLIEvent>]
    member this.PropertyChanged = propertyChanged.Publish

    // Define the add and remove methods to implement this interface.
    interface INotifyPropertyChanged with
        member this.add_PropertyChanged(handler) = propertyChanged.Publish.AddHandler(handler)
        member this.remove_PropertyChanged(handler) = propertyChanged.Publish.RemoveHandler(handler)

    // This is the event-handler method.
    member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
        let newProperty = this.GetType().GetProperty(args.PropertyName)
        let newValue = newProperty.GetValue(this :> obj) :?> string
        printfn "Property {args.PropertyName} changed its value to {newValue}"

    // Create a form, hook up the event handler, and start the application.
    let appForm = new AppForm()
    let inpc = appForm :> INotifyPropertyChanged
    inpc.PropertyChanged.Add(appForm.OnPropertyChanged)
    Application.Run(appForm)

```

如果您要在建構函式中連接事件，程式碼會稍微複雜，因為事件連結必須位於另一個建構函式的 `then` 區塊內，如下面範例所示：

```

module CustomForm

open System.Windows.Forms
open System.ComponentModel

// Create a private constructor with a dummy argument so that the public
// constructor can have no arguments.
type AppForm private (dummy) as this =
    inherit Form()

    // Define the propertyChanged event.
    let propertyChanged = Event<PropertyChangedEventHandler, PropertyChangedEventArgs>()
    let mutable underlyingValue = "text0"

    // Set up a click event to change the properties.
    do
        this.Click |> Event.add(fun evArgs ->
            this.Property1 <- "text2"
            this.Property2 <- "text3")

    // This property does not have the property changed event set.
    member val Property1 : string = "text" with get, set

    // This property has the property changed event set.
    member this.Property2
        with get() = underlyingValue
        and set(newValue) =
            underlyingValue <- newValue
            propertyChanged.Trigger(this, new PropertyChangedEventArgs("Property2"))

[<CLIEvent>]
member this.PropertyChanged = propertyChanged.Publish

// Define the add and remove methods to implement this interface.
interface INotifyPropertyChanged with
    member this.add_PropertyChanged(handler) = this.PropertyChanged.AddHandler(handler)
    member this.remove_PropertyChanged(handler) = this.PropertyChanged.RemoveHandler(handler)

// This is the event handler method.
member this.OnPropertyChanged(args : PropertyChangedEventArgs) =
    let newProperty = this.GetType().GetProperty(args.PropertyName)
    let newValue = newProperty.GetValue(this :> obj) :?> string
    printfn "Property {args.PropertyName} changed its value to {newValue}"

new() as this =
    new AppForm(0)
    then
        let inpc = this :> INotifyPropertyChanged
        inpc.PropertyChanged.Add(this.OnPropertyChanged)

// Create a form, hook up the event handler, and start the application.
let appForm = new AppForm()
Application.Run(appForm)

```

另請參閱

- [成員](#)
- [處理和引發事件](#)
- [Lambda 運算式](#): `fun` 關鍵字

明確欄位：val 關鍵字

2020/11/2 • [Edit Online](#)

`val` 關鍵字用來宣告位置以儲存類別中或結構類型中的值，但不初始化。以這種方式宣告的儲存位置稱為 *明確欄位*。`val` 關鍵字的另一種用法是搭配 `member` 關鍵字來宣告自動實作的屬性。如需自動執行之屬性的詳細資訊，請參閱 [屬性](#)。

語法

```
val [ mutable ] [ access-modifier ] field-name : type-name
```

備註

在類別或結構類型中定義欄位通常是使用 `let` 繫結。不過，`let` 繫結必須在類別建構函式中初始化，但不一定總是可行、必要或適合。當您想要未初始化的欄位時，您可以使用 `val` 關鍵字。

明確欄位可以是靜態或非靜態。*存取修飾詞*可以是 `public`、`private` 或 `internal`。根據預設，明確欄位是 `public`。這不同於類別中永遠是 `private` 的 `let` 繫結。

具有主要的函式之類別類型中的明確欄位必須有 `DefaultValue` 屬性。這個屬性指定欄位初始化為零。欄位的類型必須支援零初始化。下列類型支援零初始化：

- 具有零值的基本類型。
- 支援 `null` 值的類型，可能為正常值、異常值或值的一種表示法。這包括類別、`Tuple`、記錄、函式、介面、.NET 參考類型、`unit` 類型，以及差別聯集類型。
- .NET 實值類型。
- 所有欄位都支援預設零值的一種結構。

例如，稱為 `someField` 的不可變動欄位在 .NET 編譯表示法中有一個名為 `someField@` 的支援欄位，您可以使用名為 `someField` 的屬性來存取儲存的值。

對於可變動欄位，.NET 編譯表示法是 .NET 欄位。

WARNING

.NET Framework 命名空間 `System.ComponentModel` 包含具有相同名稱的屬性。如需此屬性的詳細資訊，請參閱 [DefaultValueAttribute](#)。

下列程式碼示範在具有主要建構函式的類別中使用明確欄位，也示範 `let` 繫結，方便對照。請注意，`let` 繫結的欄位 `myInt1` 是 `private`。從成員方法參考 `let` 繫結欄位 `myInt1` 時，不需要自我識別項 `this`。但是，當您參考明確欄位 `myInt2` 和 `myString` 時，需要自我識別項。

```

type MyType() =
  let mutable myInt1 = 10
  [<DefaultValue>] val mutable myInt2 : int
  [<DefaultValue>] val mutable myString : string
  member this.SetValsAndPrint( i: int, str: string) =
    myInt1 <- i
    this.myInt2 <- i + 1
    this.myString <- str
    printfn "%d %d %s" myInt1 (this.myInt2) (this.myString)

let myObject = new MyType()
myObject.SetValsAndPrint(11, "abc")
// The following line is not allowed because let bindings are private.
// myObject.myInt1 <- 20
myObject.myInt2 <- 30
myObject.myString <- "def"

printfn "%d %s" (myObject.myInt2) (myObject.myString)

```

輸出如下所示：

```

11 12 abc
30 def

```

下列程式碼示範在沒有主要建構函式的類別中使用明確欄位。在此例子中，`DefaultValue` 屬性不是必要的，但在類型所定義的建構函式中必須初始化所有欄位。

```

type MyClass =
  val a : int
  val b : int
  // The following version of the constructor is an error
  // because b is not initialized.
  // new (a0, b0) = { a = a0; }
  // The following version is acceptable because all fields are initialized.
  new(a0, b0) = { a = a0; b = b0; }

let myClassObj = new MyClass(35, 22)
printfn "%d %d" (myClassObj.a) (myClassObj.b)

```

輸出為 `35 22`。

下列程式碼示範在結構中使用明確欄位。因為結構是實值型別，所以它會自動具有無參數的函式，將其欄位的值設定為零。因此，不需要 `DefaultValue` 屬性。

```

type MyStruct =
  struct
    val mutable myInt : int
    val mutable myString : string
  end

let mutable myStructObj = new MyStruct()
myStructObj.myInt <- 11
myStructObj.myString <- "xyz"

printfn "%d %s" (myStructObj.myInt) (myStructObj.myString)

```

輸出為 `11 xyz`。

請注意，如果您要使用不含關鍵字欄位的欄位來初始化您的結構 `mutable`，您的指派將會處理在指派之後將會捨棄的結構複本。因此，您的結構不會變更。


```
[<Struct>]
type Foo =
    val mutable bar: string
    member self.ChangeBar bar = self.bar <- bar
    new (bar) = {bar = bar}

let foo = Foo "1"
foo.ChangeBar "2" //make implicit copy of Foo, changes the copy, discards the copy, foo remains unchanged
printfn "%s" foo.bar //prints 1

let mutable foo' = Foo "1"
foo'.ChangeBar "2" //changes foo'
printfn "%s" foo'.bar //prints 2
```

平常不適合使用明確欄位。一般而言，可能的話，應該在類別中使用 `let` 繫結，而不是明確欄位。在某些互通性案例中，例如您需要定義結構供平台叫用呼叫原生 API，或在 COM interop 案例中，明確欄位很有用。如需詳細資訊，請參閱 [外部函數](#)。另一種情況是您使用 F# 程式碼產生器來產生沒有主要建構函式的類別，這時也可能需要明確欄位。對於執行緒靜態變數或類似的建構，明確欄位也很有用。如需詳細資訊，請參閱 `System.ThreadStaticAttribute`。

當關鍵字 `member val` 一起出現在類型定義中時，這是自動實作屬性的定義。如需詳細資訊，請參閱 [屬性](#) 中定義的介面的私用 C++ 專屬實作。

另請參閱

- [屬性](#)
- [成員](#)
- `let` [類別中的系結](#)

類型延伸模組

2021/3/5 • [Edit Online](#)

類型延伸 (也稱為 *增強指定*) 是一系列的功能，可讓您將新成員加入至先前定義的物件類型。這三項功能包括：

- 內建類型延伸模組
- 選擇性類型延伸模組
- 擴充方法

每個都可以在不同的案例中使用，而且有不同的取捨。

語法

```
// Intrinsic and optional extensions
type typename with
  member self-identifier.member-name =
    body
  ...

// Extension methods
open System.Runtime.CompilerServices

[<Extension>]
type Extensions() =
  [<Extension>]
  static member extension-name (ty: typename, [args]) =
    body
  ...
```

內建類型延伸模組

內建類型延伸模組是延伸使用者定義型別的延伸模組。

內建類型延伸模組必須定義在相同的檔案中，以及 在其所擴充之類型的相同命名空間或模組中。任何其他定義都會導致它們成為 [選用的類型延伸模組](#)。

內建類型延伸模組有時可讓您以更清楚的方式來分隔類型宣告中的功能。下列範例示範如何定義內建類型延伸：

```
namespace Example

type Variant =
  | Num of int
  | Str of string

module Variant =
  let print v =
    match v with
    | Num n -> printf "Num %d" n
    | Str s -> printf "Str %s" s

// Add a member to Variant as an extension
type Variant with
  member x.Print() = Variant.print x
```

使用類型擴充功能可讓您分隔下列各項：

- 型別的宣告 `Variant`
- 根據其「圖形」來列印類別的功能 `Variant`
- 使用物件樣式表示法存取列印功能的方式 `.`

這是將所有專案定義為成員的替代方案 `Variant`。雖然它並非原本就更好的方法，但是在某些情況下，它可以是更清楚的功能表示。

內建類型延伸模組會編譯為它們所擴充之類型的成員，並且在反映檢查型別時出現在類型上。

選擇性類型延伸模組

選擇性類型延伸模組是在要擴充之類型的原始模組、命名空間或元件之外出現的延伸模組。

選擇性類型延伸模組適用於擴充您自己未定義的類型。例如：

```
module Extensions

type IEnumerable<'T> with
    /// Repeat each element of the sequence n times
    member xs.RepeatElements(n: int) =
        seq {
            for x in xs do
                for _ in 1 .. n -> x
        }
}
```

您現在可以存取 `RepeatElements` 相同的成員，只要 `IEnumerable<T>` `Extensions` 模組是在您所使用的範圍中開啟即可。

由反映檢查時，選擇性的延伸模組不會出現在擴充類型上。選用的延伸模組必須在模組中，而且只有在包含擴充功能的模組是開啟或在範圍內時，才會在範圍內。

選擇性的擴充成員會編譯成靜態成員，而靜態成員會以隱含方式將物件實例傳遞為第一個參數。不過，它們的作用就像是實例成員或靜態成員（根據其宣告方式）。

C # 或 Visual Basic 取用者也看不到選用的擴充成員。它們只能在其他 F # 程式碼中使用。

內建和選擇性類型延伸的一般限制

您可以在類型變數受到條件約束的泛型型別上宣告類型延伸。需求是延伸宣告的條件約束符合宣告型別的條件約束。

但是，即使在宣告的型別和型別延伸之間符合條件約束，也有可能是因為在類型參數上強加不同需求的擴充成員主體所推斷的條件約束，而不是宣告的型別。例如：

```
open System.Collections.Generic

// NOT POSSIBLE AND FAILS TO COMPILE!
//
// The member 'Sum' has a different requirement on 'T' than the type IEnumerable<'T>
type IEnumerable<'T> with
    member this.Sum() = Seq.sum this
```

沒有任何方法可讓此程式碼使用選擇性的類型延伸模組：

- 同樣地，`Sum` 成員在 (上具有不同的條件約束 `'T` `static member get_Zero`，且) 與類型延伸模組所定義的不同 `static member (+)`。
- 將類型延伸修改為具有相同的條件約束，`Sum` 將不再符合中的定義條件約束 `IEnumerable<'T>`。

- `member this.Sum` 將變更為 `member inline this.Sum` 會提供類型條件約束不相符的錯誤。

您需要的是「float in space」的靜態方法，而且可以像是延伸型別一樣呈現。這就是需要擴充方法的地方。

擴充方法

最後，擴充方法 (有時稱為「C# 樣式延伸成員」) 可以在 F# 中宣告為類別上的靜態成員方法。

當您想要在會限制型別變數的泛型型別上定義延伸時，擴充方法會很有用。例如：

```
namespace Extensions

open System.Collections.Generic
open System.Runtime.CompilerServices

[<Extension>]
type IEnumerableExtensions =
    [<Extension>]
    static member inline Sum(xs: IEnumerable<'T>) = Seq.sum xs
```

使用時，此程式碼會使其顯示為 `Sum` 在上定義的 `IEnumerable<T>`，只要已 `Extensions` 開啟或在範圍內即可。

為了讓擴充功能可供 VB.NET 程式碼使用，`ExtensionAttribute` 元件層級需要額外的程式碼：

```
module AssemblyInfo
open System.Runtime.CompilerServices
[<assembly:Extension>]
do ()
```

其他備註

類型延伸也有下列屬性：

- 您可以擴充任何可以存取的類型。
- 內建和選擇性類型延伸模組可以定義 *任何* 成員類型，而不只是方法。例如，擴充屬性也是可行的。
- `self-identifier` [語法](#) 中的標記代表所叫用之型別的實例，就像一般成員一樣。
- 擴充成員可以是靜態或實例成員。
- 類型延伸的型別變數必須符合宣告型別的條件約束。

類型延伸也有下列限制：

- 類型延伸模組不支援虛擬或抽象方法。
- 類型延伸模組不支援以增強指定覆寫方法。
- 類型延伸模組不支援 [靜態解析的型別參數](#)。
- 選擇性類型延伸模組不支援做為增強指定的函式。
- 類型縮寫不能定義為類型 [縮寫](#)。
- 類型延伸模組對 (而言是不正確，`byref<'T>` 不過它們可以) 宣告。
- 類型延伸模組對屬性而言無效 (但可以) 宣告。
- 您可以定義多載相同名稱之其他方法的擴充功能，但 F# 編譯器會在有不明確的呼叫時，將喜好設定提供給非擴充方法。

最後，如果有多個類型的內建類型延伸，則所有成員都必須是唯一的。對於選擇性的類型延伸，相同類型之不同類型延伸中的成員可以有相同的名稱。只有當用戶端程式代碼開啟兩個定義相同成員名稱的不同範圍時，才會發生不明確的錯誤。

請參閱

- [F # 語言參考](#)
- [成員](#)

參數和引數

2021/3/5 • [Edit Online](#)

本主題說明定義參數以及將引數傳遞至函式、方法和屬性的語言支援。其中包含如何以傳址方式傳遞的資訊，以及如何定義和使用可採用可變動數目之引數的方法。

參數和引數

詞彙 **參數** 是用來描述預期要提供之值的名稱。詞彙 **引數** 用於每個參數所提供的值。

您可以在元組或局部合併形式中指定參數，或以兩者的某種組合來指定參數。您可以使用明確的參數名稱傳遞引數。方法的參數可以指定為選擇性，並指定預設值。

參數模式

提供給函式和方法的參數通常是以空格分隔的模式。這表示，在主體中，比對 [運算式](#) 中所述的任何模式都可以用於函數或成員的參數清單中。

方法通常會使用傳遞引數的元組形式。因為元組格式符合在 .NET 方法中傳遞引數的方式，所以這可讓您從其他 .NET 語言的觀點來看得到更清楚的結果。

局部加表單最常用於使用系結所建立的函式 `let` 。

下列虛擬程式碼會顯示元組和引數的範例。

```
// Tuple form.  
member this.SomeMethod(param1, param2) = ...  
// Curried form.  
let function1 param1 param2 = ...
```

當某些引數位於元組中，而有些引數不是時，可能會合並形成。

```
let function2 param1 (param2a, param2b) param3 = ...
```

其他模式也可以在參數清單中使用，但如果參數模式不符合所有可能的輸入，則執行時間可能會有不完整的相符項。`MatchFailureException` 當引數的值與參數清單中指定的模式不相符時，就會產生例外狀況。當參數模式允許不完整的相符專案時，編譯器會發出警告。至少有一個其他模式通常適用於參數清單，也就是萬用字元模式。當您只想要忽略任何提供的引數時，請在參數清單中使用萬用字元模式。下列程式碼說明如何在引數清單中使用萬用字元模式。

```
let makeList _ = [ for i in 1 .. 100 -> i * i ]  
// The arguments 100 and 200 are ignored.  
let list1 = makeList 100  
let list2 = makeList 200
```

當您不想要使用通常以字串陣列形式提供的命令列引數時（如下列程式碼所示），當您不需要傳入的引數（例如在程式的主要進入點中）時，萬用字元模式會很有用。

```
[<EntryPoint>]
let main _ =
    printfn "Entry point!"
    0
```

有時在引數中使用的其他模式是 `as` 模式，以及與差異聯集和作用中模式相關聯的識別碼模式。您可以使用單一案例的差異聯集模式，如下所示。

```
type Slice = Slice of int * int * string

let GetSubstring1 (Slice(p0, p1, text)) =
    printfn "Data begins at %d and ends at %d in string %s" p0 p1 text
    text.[p0..p1]

let substring = GetSubstring1 (Slice(0, 4, "Et tu, Brute?"))
printfn "Substring: %s" substring
```

輸出如下。

```
Data begins at 0 and ends at 4 in string Et tu, Brute?
Et tu
```

使用中的模式可作為參數，例如，將引數轉換為所需的格式時，如下列範例所示：

```
type Point = { x : float; y : float }

let (| Polar |) { x = x; y = y } =
    ( sqrt (x*x + y*y), System.Math.Atan (y/ x) )

let radius (Polar(r, _)) = r
let angle (Polar(_, theta)) = theta
```

您可以使用 `as` 模式來儲存相符的值作為區域值，如下列程式碼所示。

```
let GetSubstring2 (Slice(p0, p1, text) as s) = s
```

偶爾使用的另一個模式是一個函式，此函式會將最後一個引數保留為未命名的函式，該函式是在隱含引數上立即執行模式比對之 lambda 運算式的功能。以下是這行程式碼的範例。

```
let isNil = function [] -> true | _::_ -> false
```

這段程式碼會定義一個函式，此函式會採用泛型清單，並 `true` 在清單空白時傳回，否則會傳回 `false`。使用這類技術可能使程式碼更難以閱讀。

有時候，牽涉到不完整相符專案的模式會很有用，例如，如果您知道程式中的清單只有三個元素，您可能會在參數清單中使用如下的模式。

```
let sum [a; b; c;] = a + b + c
```

使用具有不完整相符專案的模式最適合用於快速原型設計和其他暫存用途。編譯器會發出這類程式碼的警告。這類模式無法涵蓋所有可能輸入的一般案例，因此不適合元件 Api。

具名引數

方法的引數可以使用以逗號分隔的引數清單中的位置來指定，也可以藉由提供名稱，並在後面加上等號和要傳入的值，明確地傳遞給方法。如果是藉由提供名稱所指定，則其顯示方式可能會與宣告中使用的順序不同。

具名引數可以讓程式碼更容易閱讀，而且可更適應 API 中特定類型的變更，例如重新排列方法參數的順序。

只有方法可以使用具名引數，而不是針對系結函 `let` 式、函數值或 lambda 運算式。

下列程式碼範例示範如何使用具名引數。

```
type SpeedingTicket() =
    member this.GetMPHOver(speed: int, limit: int) = speed - limit

let CalculateFine (ticket : SpeedingTicket) =
    let delta = ticket.GetMPHOver(limit = 55, speed = 70)
    if delta < 20 then 50.0 else 100.0

let ticket1 : SpeedingTicket = SpeedingTicket()
printfn "%f" (CalculateFine ticket1)
```

在呼叫類別的函式時，您可以使用類似具名引數的語法來設定類別的屬性值。下列範例顯示此語法。

```
type Account() =
    let mutable balance = 0.0
    let mutable number = 0
    let mutable firstName = ""
    let mutable lastName = ""
    member this.AccountNumber
        with get() = number
        and set(value) = number <- value
    member this.FirstName
        with get() = firstName
        and set(value) = firstName <- value
    member this.LastName
        with get() = lastName
        and set(value) = lastName <- value
    member this.Balance
        with get() = balance
        and set(value) = balance <- value
    member this.Deposit(amount: float) = this.Balance <- this.Balance + amount
    member this.Withdraw(amount: float) = this.Balance <- this.Balance - amount

let account1 = new Account(AccountNumber=8782108,
                             FirstName="Darren", LastName="Parker",
                             Balance=1543.33)
```

如需詳細資訊，請參閱 (F #) 的函式。

選擇性參數

您可以使用參數名稱前面的問號來指定方法的選擇性參數。選擇性參數會被解釋為 F # 選項類型，因此您可以使用具有和的運算式，以定期查詢選項類型的方式來查詢它們 `match` `Some` `None`。只有成員可以使用選擇性參數，而不允許在使用系結所建立的函式上使用 `let`。

您可以透過參數名稱將現有的選擇性值傳遞給方法，例如 `?arg=None` 或 `?arg=Some(3)` 或 `?arg=arg`。當您建立的方法會將選擇性引數傳遞給另一個方法時，這會很有用。

您也可以使用函 `defaultArg` 式，此函式會設定選擇性引數的預設值。`defaultArg` 函數會採用選擇性參數做為第一個引數，並使用預設值做為第二個引數。

下列範例說明選用參數的用法。


```

type DuplexType =
    | Full
    | Half

type Connection(?rate0 : int, ?duplex0 : DuplexType, ?parity0 : bool) =
    let duplex = defaultArg duplex0 Full
    let parity = defaultArg parity0 false
    let mutable rate = match rate0 with
        | Some rate1 -> rate1
        | None -> match duplex with
            | Full -> 9600
            | Half -> 4800
    do printfn "Baud Rate: %d Duplex: %A Parity: %b" rate duplex parity

let conn1 = Connection(duplex0 = Full)
let conn2 = Connection(duplex0 = Half)
let conn3 = Connection(300, Half, true)
let conn4 = Connection(?duplex0 = None)
let conn5 = Connection(?duplex0 = Some(Full))

let optionalDuplexValue : option<DuplexType> = Some(Half)
let conn6 = Connection(?duplex0 = optionalDuplexValue)

```

輸出如下。

```

Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false
Baud Rate: 300 Duplex: Half Parity: true
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 9600 Duplex: Full Parity: false
Baud Rate: 4800 Duplex: Half Parity: false

```

基於 c # 和 Visual Basic interop 的用途, 您可以使用 `[<Optional; DefaultParameterValue(...)>]` F # 中的屬性, 讓呼叫端將引數視為選擇性。這相當於在 c # 中將引數定義為選擇性, 如中所示 `MyMethod(int i = 3)` 。

```

open System
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue("Hello world")>] message) =
        printfn $"{message}"

```

您也可以將新的物件指定為預設參數值。例如, `Foo` 成員可以改為使用選擇性的 `CancellationToken` 作為輸入:

```

open System.Threading
open System.Runtime.InteropServices
type C =
    static member Foo([<Optional; DefaultParameterValue(CancellationToken())>] ct: CancellationToken) =
        printfn $"{ct}"

```

提供做為引數的值 `DefaultParameterValue` 必須符合參數的類型。例如, 不允許下列動作:

```

type C =
    static member Wrong([<Optional; DefaultParameterValue("string")>] i:int) = ()

```

在這種情況下, 編譯器會產生警告, 而且會完全忽略這兩個屬性。請注意, 預設值 `null` 必須以類型標注, 否則編譯器會推斷錯誤的型別, 亦即 `[<Optional; DefaultParameterValue(null:obj)>] o:obj` 。

以傳址方式傳遞

以傳址方式傳遞 F # 值牽涉到 [byref](#), 也就是 managed 指標類型。應使用何種類型的指引如下：

- `inref<'T>` 如果您只需要讀取指標, 請使用。
- `outref<'T>` 如果您只需要寫入指標, 請使用。
- `byref<'T>` 如果您需要讀取和寫入指標, 請使用。

```
let example1 (x: inref<int>) = printfn $"It's %d{x}"

let example2 (x: outref<int>) = x <- x + 1

let example3 (x: byref<int>) =
    printfn $"It's %d{x}"
    x <- x + 1

let test () =
    // No need to make it mutable, since it's read-only
    let x = 1
    example1 &x

    // Needs to be mutable, since we write to it
    let mutable y = 2
    example2 &y
    example3 &y // Now 'y' is 3
```

因為參數是指標, 且值是可變動的, 所以在執行函式之後, 會保留值的任何變更。

您可以使用元組作為傳返回值, 以 `out` 在 .net 程式庫方法中儲存任何參數。或者, 您可以將 `out` 參數視為 `byref` 參數。下列程式碼範例說明這兩種方式。

```
// TryParse has a second parameter that is an out parameter
// of type System.DateTime.
let (b, dt) = System.DateTime.TryParse("12-20-04 12:21:00")

printfn "%b %A" b dt

// The same call, using an address of operator.
let mutable dt2 = System.DateTime.Now
let b2 = System.DateTime.TryParse("12-20-04 12:21:00", &dt2)

printfn "%b %A" b2 dt2
```

參數陣列

有時候, 您必須定義一個函式, 該函式會接受任意數量的異類類型參數。建立所有可能的多載方法, 以考慮可使用的所有類型, 並不是很實用的做法。 .NET 執行可透過參數陣列功能提供這類方法的支援。在其簽章中接受參數陣列的方法可以提供任意數目的參數。參數會放入陣列中。陣列元素的類型會決定可以傳遞給函式的參數類型。如果您使用 `System.Object` 做為元素類型來定義參數陣列, 則用戶端程式代碼可以傳遞任何類型的值。

在 F # 中, 只能在方法中定義參數陣列。它們不能用在模組中定義的獨立函式或函數中。

您可以使用屬性定義參數陣列 `ParamArray` 。 `ParamArray` 屬性只能套用至最後一個參數。

下列程式碼說明如何呼叫採用參數陣列的 .NET 方法, 以及 F # 中具有接受參數陣列之方法的型別定義。

```

open System

type X() =
    member this.F([<ParamArray>] args: Object[]) =
        for arg in args do
            printfn "%A" arg

[<EntryPoint>]
let main _ =
    // call a .NET method that takes a parameter array, passing values of various types
    Console.WriteLine("a {0} {1} {2} {3} {4}", 1, 10.0, "Hello world", 1u, true)

    let xobj = new X()
    // call an F# method that takes a parameter array, passing values of various types
    xobj.F("a", 1, 10.0, "Hello world", 1u, true)
    0

```

在專案中執行時，上述程式碼的輸出如下所示：

```

a 1 10 Hello world 1 True
"a"
1
10.0
"Hello world"
1u
true

```

另請參閱

- [成員](#)

運算子多載

2020/11/2 • [Edit Online](#)

本主題描述如何在類別或記錄類型，以及在全域層級上多載算術運算子。

語法

```
// Overloading an operator as a class or record member.  
static member (operator-symbols) (parameter-list) =  
    method-body  
// Overloading an operator at the global level  
let [inline] (operator-symbols) parameter-list = function-body
```

備註

在先前的語法中，**運算子-符號**是`+`、`-`、`*`、`/`、`=`等等之一。參數清單會以該運算子的一般語法來指定運算元。方法主體會構造產生的值。

運算子的運算子多載必須是靜態的。一元運算子的運算子多載(例如 `+` 和 `-`)必須在 **運算子-符號** 中使用波狀符號 `()`，以指出運算子是一元運算子而非二元運算子，如下列宣告所示。

```
static member (~-) (v : Vector)
```

下列程式碼說明只有兩個運算子的向量類別，一個運算子用於一元減號運算，一個運算子用於純量乘法。在此範例中，需要兩個純量運算的多載，因為不論向量和純量的出現順序為何，運算子都必須運作。

```
type Vector(x: float, y : float) =  
    member this.x = x  
    member this.y = y  
    static member (~-) (v : Vector) =  
        Vector(-1.0 * v.x, -1.0 * v.y)  
    static member (*) (v : Vector, a) =  
        Vector(a * v.x, a * v.y)  
    static member (*) (a, v: Vector) =  
        Vector(a * v.x, a * v.y)  
    override this.ToString() =  
        this.x.ToString() + " " + this.y.ToString()  
  
let v1 = Vector(1.0, 2.0)  
  
let v2 = v1 * 2.0  
let v3 = 2.0 * v1  
  
let v4 = - v2  
  
printfn "%s" (v1.ToString())  
printfn "%s" (v2.ToString())  
printfn "%s" (v3.ToString())  
printfn "%s" (v4.ToString())
```

建立新的運算子

您可以多載所有標準運算子，但您也可以在特定字元的序列中建立新的運算子。允許的運算子字元為 `! . , , ,`

'''	''''''
<code>~-</code>	<code>op_UnaryNegation</code>
<code>=</code>	<code>op_Equality</code>
<code><=</code>	<code>op_LessThanOrEqual</code>
<code>>=</code>	<code>op_GreaterThanOrEqual</code>
<code><</code>	<code>op_LessThan</code>
<code>></code>	<code>op_GreaterThan</code>
<code>?</code>	<code>op_Dynamic</code>
<code>?<-</code>	<code>op_DynamicAssignment</code>
<code> ></code>	<code>op_PipeRight</code>
<code>< </code>	<code>op_PipeLeft</code>
<code>!</code>	<code>op_Dereference</code>
<code>>></code>	<code>op_ComposeRight</code>
<code><<</code>	<code>op_ComposeLeft</code>
<code><@ @></code>	<code>op_Quotation</code>
<code><@@ @@></code>	<code>op_QuotationUntyped</code>
<code>+=</code>	<code>op_AdditionAssignment</code>
<code>-=</code>	<code>op_SubtractionAssignment</code>
<code>*=</code>	<code>op_MultiplyAssignment</code>
<code>/=</code>	<code>op_DivisionAssignment</code>
<code>..</code>	<code>op_Range</code>
<code>.. ..</code>	<code>op_RangeStep</code>

請注意，`not` F # 中的運算子不會發出，`op_Inequality` 因為它不是符號運算子。它是發出 IL 以否定布林運算式的函數。

此處未列出的其他運算子字元組合可以用來做為運算子，而且具有藉由串連下表中的個別字元名稱所組成的名稱。例如，`+ !` 成為 `op_PlusBang` 。

符號	名稱
>	Greater
<	Less
+	Plus
-	Minus
*	Multiply
/	Divide
=	Equals
~	Twiddle
%	Percent
.	Dot
&	Amp
	Bar
@	At
^	Hat
!	Bang
?	Qmark
(LParen
,	Comma
)	RParen
[LBrack
]	RBrack

前置和中置運算子

前置 運算子必須放在運算元或運算元前面，與函式非常類似。中置運算子應放在兩個運算元之間。

只有特定運算子可以當做前置運算子使用。某些運算子一律為前置運算子，其他運算子可以是中置或前置詞，而其餘的一律是中置運算子。開頭為 `!` 的運算子 (不包括 `!=`) 以及 `~` 運算子或 `~` 的重複序列一律為前置運算子。運算子 `+`、`-`、`++`、`--`、`&`、`&&`、`%` 和 `%%` 可以是前置運算子或中置運算子。您可以藉由在前置運

算子的開頭新增，來區別這些運算子的前置詞版本與中置版本 `~`。`~` 只有在定義時，才會使用運算子。

範例

下列程式碼說明如何使用運算子多載來執行分數類型。分數是以分子和分母表示。函數 `hcf` 是用來決定最高的常見因數，這會用來減少分數。

```
// Determine the highest common factor between
// two positive integers, a helper for reducing
// fractions.
let rec hcf a b =
    if a = 0u then b
    elif a < b then hcf a (b - a)
    else hcf (a - b) b

// type Fraction: represents a positive fraction
// (positive rational number).
type Fraction =
{
    // n: Numerator of fraction.
    n : uint32
    // d: Denominator of fraction.
    d : uint32
}

// Produce a string representation. If the
// denominator is "1", do not display it.
override this.ToString() =
    if (this.d = 1u)
    then this.n.ToString()
    else this.n.ToString() + "/" + this.d.ToString()

// Add two fractions.
static member (+) (f1 : Fraction, f2 : Fraction) =
    let nTemp = f1.n * f2.d + f2.n * f1.d
    let dTemp = f1.d * f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Adds a fraction and a positive integer.
static member (+) (f1: Fraction, i : uint32) =
    let nTemp = f1.n + i * f1.d
    let dTemp = f1.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Adds a positive integer and a fraction.
static member (+) (i : uint32, f2: Fraction) =
    let nTemp = f2.n + i * f2.d
    let dTemp = f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Subtract one fraction from another.
static member (-) (f1 : Fraction, f2 : Fraction) =
    if (f2.n * f1.d > f1.n * f2.d)
    then failwith "This operation results in a negative number, which is not supported."
    let nTemp = f1.n * f2.d - f2.n * f1.d
    let dTemp = f1.d * f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Multiply two fractions.
static member (*) (f1 : Fraction, f2 : Fraction) =
    let nTemp = f1.n * f2.n
    let dTemp = f1.d * f2.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }
```



```

let nctTemp = nct + niTemp dTemp
{ n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// Divide two fractions.
static member (/) (f1 : Fraction, f2 : Fraction) =
    let nTemp = f1.n * f2.d
    let dTemp = f2.n * f1.d
    let hcfTemp = hcf nTemp dTemp
    { n = nTemp / hcfTemp; d = dTemp / hcfTemp }

// A full set of operators can be quite lengthy. For example,
// consider operators that support other integral data types,
// with fractions, on the left side and the right side for each.
// Also consider implementing unary operators.

let fraction1 = { n = 3u; d = 4u }
let fraction2 = { n = 1u; d = 2u }
let result1 = fraction1 + fraction2
let result2 = fraction1 - fraction2
let result3 = fraction1 * fraction2
let result4 = fraction1 / fraction2
let result5 = fraction1 + 1u
printfn "%s + %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result1.ToString())
printfn "%s - %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result2.ToString())
printfn "%s * %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result3.ToString())
printfn "%s / %s = %s" (fraction1.ToString()) (fraction2.ToString()) (result4.ToString())
printfn "%s + 1 = %s" (fraction1.ToString()) (result5.ToString())

```

輸出：

```

3/4 + 1/2 = 5/4
3/4 - 1/2 = 1/4
3/4 * 1/2 = 3/8
3/4 / 1/2 = 3/2
3/4 + 1 = 7/4

```

全域層級的運算子

您也可以在全域層級定義運算子。下列程式碼會定義操作員 `+`。

```

let inline (+?) (x: int) (y: int) = x + 2*y
printf "%d" (10 +? 1)

```

上述程式碼的輸出為 `12`。

您可以用這種方式重新定義一般算術運算子，因為 F# 的範圍規則會規定新定義的運算子優先於內建運算子。

關鍵字 `inline` 通常會與全域運算子搭配使用，這通常是最適合整合至呼叫程式碼的小型函式。使運算子函式內嵌也可讓它們使用靜態解析的型別參數，以產生靜態解析的一般程式碼。如需詳細資訊，請參閱 [內嵌](#) 函式和 [靜態解析的型別參數](#)。

另請參閱

- [成員](#)

彈性類型

2020/11/2 • [Edit Online](#)

彈性類型 注釋指出參數、變數或值的類型與指定的類型相容，其中的相容性是由類別或介面的物件導向階層中的位置所決定。彈性型別特別適用於自動轉換類型階層中較高的類型時，但您仍然想要讓您的功能使用階層中的任何類型，或任何實介面的型別。

語法

```
#type
```

備註

在先前的語法中，*type* 代表基底類型或介面。

彈性型別相當於具有條件約束的泛型型別，其條件約束會將允許的類型限制為與基底或介面類別型相容的類型。也就是說，下列兩行程式碼是相等的。

```
#SomeType

'T when 'T :> SomeType
```

彈性類型適用於數種類型的情況。例如，當您有較高階的函式（使用函式做為引數）的函式時，讓函式傳回彈性型別通常會很有用。在下列範例中，在中使用具有 `sequence` 引數的彈性型別，可 `iterate2` 讓更高的 `order` 函數使用可產生序列、陣列、清單和任何其他可列舉型別的函數。

請考慮下列兩個函式，其中一個會傳回序列，另一個則會傳回彈性型別。

```
let iterate1 (f : unit -> seq<int>) =
    for e in f() do printfn "%d" e
let iterate2 (f : unit -> #seq<int>) =
    for e in f() do printfn "%d" e

// Passing a function that takes a list requires a cast.
iterate1 (fun () -> [1] :> seq<int>)

// Passing a function that takes a list to the version that specifies a
// flexible type as the return value is OK as is.
iterate2 (fun () -> [1])
```

另舉一個範例，請考慮採用 `Seq` 程式庫函數：

```
val concat: sequences:seq<#seq<'T>> -> seq<'T>
```

您可以將下列任何可列舉的序列傳遞給此函式：

- 清單清單
- 陣列清單
- 清單的陣列
- 序列的陣列

- 可列舉序列的任何其他組合

下列 `Seq.concat` 程式碼會使用來示範您可以使用彈性類型來支援的案例。

```
let list1 = [1;2;3]
let list2 = [4;5;6]
let list3 = [7;8;9]

let concat1 = Seq.concat [ list1; list2; list3 ]
printfn "%A" concat1

let array1 = [|1;2;3|]
let array2 = [|4;5;6|]
let array3 = [|7;8;9|]

let concat2 = Seq.concat [ array1; array2; array3 ]
printfn "%A" concat2

let concat3 = Seq.concat [| list1; list2; list3 |]
printfn "%A" concat3

let concat4 = Seq.concat [| array1; array2; array3 |]
printfn "%A" concat4

let seq1 = { 1 .. 3 }
let seq2 = { 4 .. 6 }
let seq3 = { 7 .. 9 }

let concat5 = Seq.concat [| seq1; seq2; seq3 |]

printfn "%A" concat5
```

輸出如下。

```
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
seq [1; 2; 3; 4; ...]
```

在 F# 中，如同其他物件導向的語言，有一些內容會將實介面的衍生類型或型別自動轉換成基底類型或介面型別。這些自動轉換會在直接引數中進行，但不會在型別位於從屬位置、成為更複雜型別的一部分（例如函式類型的傳回型別），或做為型別引數。因此，當您要套用它的型別是更複雜型別的一部分時，彈性型別標記法主要很有用。

另請參閱

- [F# 語言參考](#)
- [泛型](#)

委派

2019/10/23 • [Edit Online](#)

委派代表當做物件的函式呼叫。在 F# 中, 您通常應該使用函式值來將函式表示為第一類的值;不過, 委派會在 .NET Framework 中使用, 因此當您與預期它們的 Api 互通時, 需要用到。撰寫專供其他 .NET Framework 語言使用的程式庫時, 也可能會使用它們。

語法

```
type delegate-typename = delegate of type1 -> type2
```

備註

在先前的語法中 `type1`, 代表引數類型或類型 `type2`, 而表示傳回類型。所表示 `type1` 的引數類型會自動進行擴充。這表示針對此類型, 如果目標函式的引數是已擴充的, 而且已經在元組表單中的引數有括弧的元組, 您就可以使用元組表單。自動 currying 會移除一組括弧, 並保留符合目標方法的元組引數。請參閱程式碼範例, 以瞭解您在每個案例中應該使用的語法。

委派可以附加至 F# 函數值, 以及靜態或實例方法。F# 函數值可以直接當做引數傳遞至委派的函式。若為靜態方法, 您可以使用類別的名稱和方法來建立委派。若為實例方法, 您可以在一個引數中提供物件實例和方法。在這兩種情況下, 都會使用 `.` 成員存取運算子 ()。

委派 `Invoke` 類型上的方法會呼叫封裝的函式。此外, 委派可以藉由參考不含括弧的叫用方法名稱, 以函式值的方式傳遞。

下列程式碼顯示用來建立委派的語法, 代表類別中的各種方法。根據該方法是靜態方法或實例方法, 以及它在元組形式或擴充形式中是否有引數, 宣告和指派委派的語法稍有不同。

```

type Test1() =
    static member add(a : int, b : int) =
        a + b
    static member add2 (a : int) (b : int) =
        a + b

    member x.Add(a : int, b : int) =
        a + b
    member x.Add2 (a : int) (b : int) =
        a + b

// Delegate1 works with tuple arguments.
type Delegate1 = delegate of (int * int) -> int
// Delegate2 works with curried arguments.
type Delegate2 = delegate of int * int -> int

let InvokeDelegate1 (dlg: Delegate1) (a: int) (b: int) =
    dlg.Invoke(a, b)
let InvokeDelegate2 (dlg: Delegate2) (a: int) (b: int) =
    dlg.Invoke(a, b)

// For static methods, use the class name, the dot operator, and the
// name of the static method.
let del1 = Delegate1(Test1.add)
let del2 = Delegate2(Test1.add2)

let testObject = Test1()

// For instance methods, use the instance value name, the dot operator, and the instance method name.
let del3 = Delegate1(testObject.Add)
let del4 = Delegate2(testObject.Add2)

for (a, b) in [ (100, 200); (10, 20) ] do
    printfn "%d + %d = %d" a b (InvokeDelegate1 del1 a b)
    printfn "%d + %d = %d" a b (InvokeDelegate2 del2 a b)
    printfn "%d + %d = %d" a b (InvokeDelegate1 del3 a b)
    printfn "%d + %d = %d" a b (InvokeDelegate2 del4 a b)

```

下列程式碼顯示一些您可以使用委派的的不同方式。

```

type Delegate1 = delegate of int * char -> string

let replicate n c = String.replicate n (string c)

// An F# function value constructed from an unapplied let-bound function
let function1 = replicate

// A delegate object constructed from an F# function value
let delObject = Delegate1(function1)

// An F# function value constructed from an unapplied .NET member
let functionValue = delObject.Invoke

List.map (fun c -> functionValue(5,c)) ['a'; 'b'; 'c']
|> List.iter (printfn "%s")

// Or if you want to get back the same curried signature
let replicate' n c = delObject.Invoke(n,c)

// You can pass a lambda expression as an argument to a function expecting a compatible delegate type
// System.Array.ConvertAll takes an array and a converter delegate that transforms an element from
// one type to another according to a specified function.
let stringArray = System.Array.ConvertAll(['a';'b'], fun c -> replicate' 3 c)
printfn "%A" stringArray

```

先前程式碼範例的輸出如下所示。

```

aaaaa
bbbbbb
ccccc
[|"aaa"; "bbb"|]

```

另請參閱

- [F# 語言參考](#)
- [參數和引數](#)
- [事件](#)

物件運算式

2021/3/5 • [Edit Online](#)

物件運算式 是一個運算式，它會根據現有的基底類型、介面或介面集，建立動態建立之匿名物件類型的新實例。

語法

```
// When typename is a class:
{ new typename [type-params]arguments with
  member-definitions
  [ additional-interface-definitions ]
}
// When typename is not a class:
{ new typename [generic-type-args] with
  member-definitions
  [ additional-interface-definitions ]
}
```

備註

在先前的語法中，*typename* 代表現有的類別型別或介面型別。*類型* 參數描述選擇性的泛型型別參數。*引數* 只適用於需要使用函式參數的類別類型。*成員定義* 是基類方法的覆寫，或是基類或介面的抽象方法的執行。

下列範例說明數種不同類型的物件運算式。

```

// This object expression specifies a System.Object but overrides the
// ToString method.
let obj1 = { new System.Object() with member x.ToString() = "F#" }
printfn $"{obj1}"

// This object expression implements the IFormattable interface.
let delimiter(delim1: string, delim2: string, value: string) =
    { new System.IFormattable with
        member x.ToString(format: string, provider: System.IFormatProvider) =
            if format = "D" then
                delim1 + value + delim2
            else
                value }

let obj2 = delimiter("{","}", "Bananas!");

printfn "%A" (System.String.Format("{0:D}", obj2))

// Define two interfaces
type IFirst =
    abstract F : unit -> unit
    abstract G : unit -> unit

type ISecond =
    inherit IFirst
    abstract H : unit -> unit
    abstract J : unit -> unit

// This object expression implements both interfaces.
let implementer() =
    { new ISecond with
        member this.H() = ()
        member this.J() = ()
        interface IFirst with
            member this.F() = ()
            member this.G() = () }

```

使用物件運算式

當您想要避免建立新的、命名的型別所需的額外程式碼和額外負荷時，請使用物件運算式。如果您使用物件運算式將程式中所建立的類型數目降至最低，您可以減少程式碼的行數，並避免不必要的類型激增。您可以使用物件運算式來自訂現有的型別，或為手邊的特定案例提供適當的介面執行，而不需建立許多型別來處理特定情況。

另請參閱

- [F # 語言參考](#)

轉型和轉換 (F#)

2020/2/22 • [Edit Online](#)

本主題描述中F#類型轉換的支援。

算術類型

F#提供各種基本類型(例如整數和浮點類型之間)之算術轉換的轉換運算子。整數和字元轉換運算子具有 checked 和 unchecked 形式;浮點運算子和 `enum` 轉換運算子不會。未檢查的表單定義于

`Microsoft.FSharp.Core.Operators` 中, 而檢查的表單則定義于 `Microsoft.FSharp.Core.Operators.Checked` 中。如果產生的值超出目標型別的限制, 檢查的表單會檢查溢位並產生執行時間例外狀況。

每個運算子的名稱都與目的地類型的名稱相同。例如, 在下列程式碼中, 已明確標注類型, `byte` 會以兩個不同的意義出現。第一次出現的是類型, 而第二個是轉換運算子。

```
let x : int = 5

let b : byte = byte x
```

下表顯示在中定義的F#轉換運算子。

'''	''
<code>byte</code>	轉換成 <code>byte</code> , 這是8位不帶正負號的類型。
<code>sbyte</code>	轉換成帶正負號的位元組。
<code>int16</code>	轉換為16位帶正負號的整數。
<code>uint16</code>	轉換成16位不帶正負號的整數。
<code>int32, int</code>	轉換為32位帶正負號的整數。
<code>uint32</code>	轉換為32位不帶正負號的整數。
<code>int64</code>	轉換為64位帶正負號的整數。
<code>uint64</code>	轉換為64位不帶正負號的整數。
<code>nativeint</code>	轉換成原生整數。
<code>unativeint</code>	轉換成不帶正負號的原生整數。
<code>float, double</code>	轉換成64位雙精確度 IEEE 浮點數字。
<code>float32, single</code>	轉換成32位單精確度 IEEE 浮點數字。
<code>decimal</code>	轉換成 <code>System.Decimal</code> 。

'''	''
<code>char</code>	轉換成 <code>System.Char</code> , Unicode 字元。
<code>enum</code>	轉換為列舉類型。

除了內建的基本型別之外，您還可以使用這些運算子搭配以適當簽章來執行 `op_Explicit` 或 `op_Implicit` 方法的類型。例如，`int` 轉換運算子適用於任何提供靜態方法的類型，`op_Explicit` 會採用類型做為參數，並傳回 `int`。如需方法無法以傳回型別多載之一般規則的特殊例外狀況，您可以為 `op_Explicit` 和 `op_Implicit` 執行此動作。

列舉類型

`enum` 運算子是泛型運算子，接受一個代表要轉換之 `enum` 類型的類型參數。當它轉換成列舉型別時，型別推斷會嘗試判斷您想要轉換的 `enum` 型別。在下列範例中，變數 `col1` 不會明確地加上批註，但它的型別是從較新的等號比較測試推斷而來。因此，編譯器可以推算您轉換成 `Color` 列舉。或者，您也可以提供類型注釋，如同下列範例中的 `col2`。

```
type Color =
    | Red = 1
    | Green = 2
    | Blue = 3

// The target type of the conversion cannot be determined by type inference, so the type parameter must be explicit.
let col1 = enum<Color> 1

// The target type is supplied by a type annotation.
let col2 : Color = enum 2
```

您也可以將目標列舉型別明確指定為型別參數，如下列程式碼所示：

```
let col3 = enum<Color> 3
```

請注意，只有在列舉的基礎類型與所轉換的類型相容時，列舉轉換才會生效。在下列程式碼中，因為 `int32` 與 `uint32` 不相符，所以轉換無法編譯。

```
// Error: types are incompatible
let col4 : Color = enum 2u
```

如需詳細資訊，[請參閱列舉](#)。

轉換物件類型

物件階層中的類型之間的轉換是物件導向程式設計的基礎。轉換有兩種基本類型：[向上轉換] (upcasting) 和 [向下轉換] (向下檢視)。向上轉換階層表示從衍生的物件參考轉換成基底物件參考。只要基類位於衍生類別的繼承階層架構中，這類轉換就一定會正常執行。將階層從基底物件參考向下轉換至衍生的物件參考時，只有在物件實際上是正確目的地(衍生)類型的實例，或衍生自目的地類型的類型時，才會成功。

F#提供這些轉換類型的運算子。`:>` 運算子會向上轉換階層，而 `:?>` 運算子會在階層中往下轉換。

Upcasting

在許多物件導向語言中，upcasting 都是隱含的;在F#中，規則會稍有不同。當您將引數傳遞至物件類型上的方法時，會自動套用 Upcasting。不過，針對模組中的 let 系結函式，除非將參數類型宣告為彈性類型，否則 upcasting

不會是自動的。如需詳細資訊，請參閱[彈性類型](#)。

`:>` 運算子會執行靜態轉換，這表示轉換成功是在編譯時期決定的。如果使用 `:>` 的轉換成功編譯，它是有效的轉換，而且在執行時間沒有任何可能發生的失敗。

您也可以使用 `upcast` 運算子來執行這類轉換。下列運算式會指定階層的向上轉換：

```
upcast expression
```

當您使用向上轉型運算子時，編譯器會嘗試從內容推斷您要轉換成的類型。如果編譯器無法判斷目標型別，則編譯器會報告錯誤。可能需要類型注釋。

向下檢視

`:?>` 運算子會執行動態轉換，這表示在執行時間會決定轉換成功與否。使用 `:?>` 運算子的轉換不會在編譯時期檢查；但是在執行時間，會嘗試轉換成指定的型別。如果物件與目標型別相容，轉換就會成功。如果物件與目標型別不相容，則執行時間會引發 `InvalidCastException`。

您也可以使用 `downcast` 運算子來執行動態類型轉換。下列運算式會指定將階層向下轉換為從程式內容推斷的類型：

```
downcast expression
```

就 `upcast` 運算子而言，如果編譯器無法從內容推斷特定目標型別，就會報告錯誤。可能需要類型注釋。

下列程式碼說明如何使用 `:>` 和 `:?>` 運算子。此程式碼說明當您知道轉換會成功時，最適合使用 `:?>` 運算子，因為它會在轉換失敗時擲回 `InvalidCastException`。如果您不知道轉換將會成功，使用 `match` 運算式的類型測試會較好，因為它可避免產生例外狀況的額外負荷。

```
type Base1() =
    abstract member F : unit -> unit
    default u.F() =
        printfn "F Base1"

type Derived1() =
    inherit Base1()
    override u.F() =
        printfn "F Derived1"

let d1 : Derived1 = Derived1()

// Upcast to Base1.
let base1 = d1 :> Base1

// This might throw an exception, unless
// you are sure that base1 is really a Derived1 object, as
// is the case here.
let derived1 = base1 :?> Derived1

// If you cannot be sure that b1 is a Derived1 object,
// use a type test, as follows:
let downcastBase1 (b1 : Base1) =
    match b1 with
    | :? Derived1 as derived1 -> derived1.F()
    | _ -> ()

downcastBase1 base1
```

因為泛型運算子 `downcast` 和 `upcast` 依賴型別推斷來判斷引數和傳回型別，所以在上述程式碼中，您可以取代

```
let base1 = d1 :> Base1
```

取代為

```
let base1: Base1 = upcast d1
```

請注意，類型注釋是必要的，因為 `upcast` 本身無法判斷基類。

另請參閱

- [F# 語言參考](#)

存取控制

2019/11/4 • [Edit Online](#)

存取控制是指宣告哪些用戶端可以使用特定的程式元素，例如類型、方法和函式。

存取控制的基本概念

在F#中，存取控制規範 `public`、`internal` 和 `private` 可以套用至模組、類型、方法、值定義、函數、屬性和明確欄位。

- `public` 表示實體可由所有呼叫者存取。
- `internal` 表示實體只能從相同的元件存取。
- `private` 表示實體只能從封閉式類型或模組存取。

NOTE

存取規範 `protected` 不會用於中F#，不過，如果您使用的類型是以支援 `protected` 存取的語言撰寫，則此為可接受的。因此，如果您覆寫受保護的方法，您的方法只會在類別和其子代內保持可存取狀態。

一般來說，規範會放在機構名稱的前方，但使用 `mutable` 或 `inline` 規範時，會出現在存取控制規範後面。

如果未使用存取規範，則預設值為 `public`，但類型中的 `let` 系結一律會 `private` 為類型。

中F#的簽章提供另一個機制來F#控制程式元素的存取。存取控制不需要簽章。如需詳細資訊，請參閱[簽章](#)。

存取控制的規則

存取控制受限於下列規則：

- 繼承宣告（也就是使用 `inherit` 來指定類別的基類）、介面宣告（也就是指定類別實作為介面），而抽象成員一律具有與封入類型相同的存取範圍。因此，您無法在這些結構上使用存取控制規範。
- 區分等位中個別案例的協助工具是由區分聯集本身的存取範圍所決定。也就是說，特定聯集的大小寫不會比等位本身更容易存取。
- 記錄類型的個別欄位可存取性是由記錄本身的存取範圍所決定。也就是說，特定記錄標籤的存取權不如記錄本身。

範例

下列程式碼說明如何使用存取控制規範。專案中有兩個檔案，`Module1.fs` 和 `Module2.fs`。每個檔案都是隱含的模組。因此，`Module1` 和 `Module2` 都有兩個模組。私用型別和內部型別都是在 `Module1` 中定義。無法從 `Module2` 存取私用型別，但內部型別可以。

```
// Module1.fs

module Module1

// This type is not usable outside of this file
type private MyPrivateType() =
    // x is private since this is an internal let binding
    let x = 5
    // X is private and does not appear in the QuickInfo window
    // when viewing this type in the Visual Studio editor
    member private this.X() = 10
    member this.Z() = x * 100

type internal MyInternalType() =
    let x = 5
    member private this.X() = 10
    member this.Z() = x * 100

// Top-level let bindings are public by default,
// so "private" and "internal" are needed here since a
// value cannot be more accessible than its type.
let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

// let bindings at the top level are public by default,
// so result1 and result2 are public.
let result1 = myPrivateObj.Z
let result2 = myInternalObj.Z
```

下列程式碼會測試 `Module1.fs` 中建立之類型的存取範圍。

```
// Module2.fs
module Module2

open Module1

// The following line is an error because private means
// that it cannot be accessed from another file or module
// let private myPrivateObj = new MyPrivateType()
let internal myInternalObj = new MyInternalType()

let result = myInternalObj.Z
```

請參閱

- [F# 語言參考](#)
- [簽章](#)

條件運算式：`if...then...else`

2019/10/23 • [Edit Online](#)

`if...then...else` 運算式會執行不同的程式碼分支，也會根據指定的布林運算式，評估為不同的值。

語法

```
if boolean-expression then expression1 [ else expression2 ]
```

備註

在先前的語法中，*運算式*會在布林運算式評估為 `true` 時執行，否則會執行 *運算式2*。

不同于其他語言，`if...then...else` 結構是運算式，而不是語句。這表示它會產生值，這是執行之分支中最後一個運算式的值。每個分支中產生的數值型別必須相符。如果沒有明確 `else` 分支，其類型為 `unit`。因此，如果 `then` 分支的型別是 `unit` 以外的任何型別 `else`，則必須有一個具有相同傳回型別的分支。將運算式 `if...then...else` 連結在一起時，您可以使用 `elif` 關鍵字，`else if` 而不是，它們是相等的。

範例

下列範例說明如何使用 `if...then...else` 運算式。

```
let test x y =
    if x = y then "equals"
    elif x < y then "is less than"
    else "is greater than"

printfn "%d %s %d." 10 (test 10 20) 20

printfn "What is your name? "
let nameString = System.Console.ReadLine()

printfn "What is your age? "
let ageString = System.Console.ReadLine()
let age = System.Int32.Parse(ageString)

if age < 10
then printfn "You are only %d years old and already learning F#? Wow!" age
```

```
10 is less than 20
What is your name? John
How old are you? 9
You are only 9 years old and already learning F#? Wow!
```

另請參閱

- [F# 語言參考](#)

Match 運算式

2019/10/23 • [Edit Online](#)

`match` 運算式提供的分支控制是以運算式與一組模式的比較為基礎。

語法

```
// Match expression.
match test-expression with
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...

// Pattern matching function.
function
| pattern1 [ when condition ] -> result-expression1
| pattern2 [ when condition ] -> result-expression2
| ...
```

備註

模式比對運算式可根據測試運算式與一組模式的比較, 來進行複雜的分支。在運算式中, 會依序將*測試運算式*與每個模式進行比較, 當找到相符的結果時, 就會評估對應的*結果運算式*, 並傳回產生的值做為 `match` 運算式的值。

`match`

前面的語法中所顯示的模式比對函式, 是在引數上立即執行模式比對的 `lambda` 運算式。先前語法中所顯示的模式比對函式相當於下列。

```
fun arg ->
  match arg with
  | pattern1 [ when condition ] -> result-expression1
  | pattern2 [ when condition ] -> result-expression2
  | ...
```

如需 `lambda` 運算式的詳細資訊, [請參閱 lambda 運算式](#): `fun` 關鍵字。

一整組模式應涵蓋輸入變數的所有可能相符專案。通常, 您會使用萬用字元模式 (`_`) 做為最後一個模式, 以比對任何先前不相符的輸入值。

下列程式碼說明使用 `match` 運算式的一些方式。如需可使用之所有可能模式的參考和範例, [請參閱模式比對](#)。


```

let list1 = [ 1; 5; 100; 450; 788 ]

// Pattern matching by using the cons pattern and a list
// pattern that tests for an empty list.
let rec printList listx =
  match listx with
  | head :: tail -> printf "%d " head; printList tail
  | [] -> printfn ""

printList list1

// Pattern matching with multiple alternatives on the same line.
let filter123 x =
  match x with
  | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
  | a -> printfn "%d" a

// The same function written with the pattern matching
// function syntax.
let filterNumbers =
  function | 1 | 2 | 3 -> printfn "Found 1, 2, or 3!"
           | a -> printfn "%d" a

```

保護模式

您可以使用 `when` 子句來指定其他條件, 讓變數必須滿足以符合模式。這類子句稱為「*防護*」。除非對與該 `when` 防護相關聯的模式進行比對, 否則不會評估緊接在關鍵字後面的運算式。

下列範例說明如何使用 `guard` 來指定變數模式的數值範圍。請注意, 您可以使用布林運算子來結合多個條件。

```

let rangeTest testValue mid size =
  match testValue with
  | var1 when var1 >= mid - size/2 && var1 <= mid + size/2 -> printfn "The test value is in range."
  | _ -> printfn "The test value is out of range."

rangeTest 10 20 5
rangeTest 10 20 10
rangeTest 10 20 40

```

請注意, 因為不能在模式中使用常值以外的值, 所以如果 `when` 您必須將輸入的某些部分與值進行比較, 則必須使用子句。這會以下列程式碼顯示:

```

// This example uses patterns that have when guards.
let detectValue point target =
  match point with
  | (a, b) when a = target && b = target -> printfn "Both values match target %d." target
  | (a, b) when a = target -> printfn "First value matched target in (%d, %d)" target b
  | (a, b) when b = target -> printfn "Second value matched target in (%d, %d)" a target
  | _ -> printfn "Neither value matches target."

detectValue (0, 0) 0
detectValue (1, 0) 0
detectValue (0, 10) 0
detectValue (10, 15) 0

```

請注意, 當「聯集」模式受到防護的涵蓋時, 此防護會套用至所有模式, 而不只是最後一個。例如, 假設有下列程式碼, 則此 `when a > 12` 防護適用 `A a` 和 `B a`:

```
type Union =  
  | A of int  
  | B of int  
  
let foo() =  
  let test = A 42  
  match test with  
  | A a  
  | B a when a > 41 -> a // the guard applies to both patterns  
  | _ -> 1  
  
foo() // returns 42
```

另請參閱

- [F# 語言參考](#)
- [使用中模式](#)
- [模式比對](#)

模式比對

2021/3/5 • [Edit Online](#)

模式是轉換輸入資料的規則。在 F# 語言中會使用它們來比較資料與邏輯結構或結構、將資料分解為構成部分，或從資料以各種方式將資訊解壓縮。

備註

模式用於許多語言結構，例如 `match` 運算式。當您要處理系統中函式的引數 `let`、`lambda` 運算式，以及與運算式相關聯的例外狀況處理常式時，就會使用這些函數 `try...with`。如需詳細資訊，請參閱 [Match 運算式](#)、[let Bindings](#)、[Lambda 運算式](#)：`fun` 關鍵字和 [例外狀況](#)：`try...with` 運算式。

例如，在運算式中 `match`，`模式` 會跟在管道符號之後。

```
match expression with
| pattern [ when condition ] -> result-expression
...
```

每個模式都是以某種方式轉換輸入的規則。在 `match` 運算式中，會依序檢查每個模式，以查看輸入資料是否與模式相容。如果找到相符的結果運算式，就會執行結果運算式。如果找不到相符項，則會測試下一個模式規則。在比對 [運算式](#) 中說明選擇性的 `when` 條件部分。

下表顯示支援的模式。在執行時間，輸入會根據表格中所列的順序，針對下列每個模式進行測試，而模式會以遞迴方式套用（從第一次到最後，當它們出現在您的程式碼中），並從左至右套用至每一行的模式。

“	“	“
常數模式	任何數值、字元或字串常值、列舉常數或已定義的常值識別碼	<code>1.0</code> , <code>"test"</code> , <code>30</code> , <code>Color.Red</code>
識別碼模式	差異聯集的 <code>case</code> 值、例外狀況標籤或作用中模式案例	<code>Some(x)</code> <code>Failure(msg)</code>
變數模式	<i>identifier</i>	<code>a</code>
<code>as</code> 模式	<i>模式 as 識別碼</i>	<code>(a, b) as tuple1</code>
或模式	<i>pattern1 pattern2</i>	<code>([h] [h; _])</code>
AND 模式	<i>pattern1 & pattern2</i>	<code>(a, b) & (_, "test")</code>
缺點模式	<i>identifier :: list-identifier</i>	<code>h :: t</code>
清單模式	<i>[pattern_1; ...; pattern_n]</i>	<code>[a; b; c]</code>
陣列模式	<i>[pattern_1; ...; pattern_n]</i>	<code>[a; b; c]</code>
括弧模式	<i>(模式)</i>	<code>(a)</code>

II	II	II
元組模式	(<i>pattern_1</i> , <i>pattern_n</i>)	(a, b)
記錄模式	{ <i>identifier1</i> = <i>pattern_1</i> ;...; <i>identifier_n</i> = <i>pattern_n</i> }	{ Name = name; }
萬用字元模式		_
搭配類型注釋的模式	<i>模式</i> : <i>類型</i>	a : int
類型測試模式	:? <i>類型</i> [as <i>識別碼</i>]	:? System.DateTime as dt
Null 模式	null	null
Nameof 模式	<i>nameof</i> 運算式	nameof str

常數模式

常數模式包括數值、字元和字串常值、列舉常數 (包含) 包含的列舉型別名稱。 `match` 只有常數模式的運算式可以與其他語言的 `case` 語句進行比較。輸入會與常值進行比較, 如果值相等, 則模式會相符。常值的型別必須與輸入的型別相容。

下列範例示範如何使用常值模式, 並同時使用變數模式和或模式。

```
[<Literal>]
let Three = 3

let filter123 x =
    match x with
    // The following line contains literal patterns combined with an OR pattern.
    | 1 | 2 | Three -> printfn "Found 1, 2, or 3!"
    // The following line contains a variable pattern.
    | var1 -> printfn "%d" var1

for x in 1..10 do filter123 x
```

常值模式的另一個範例是以列舉常數為基礎的模式。當您使用列舉常數時, 必須指定列舉型別名稱。

```
type Color =
    | Red = 0
    | Green = 1
    | Blue = 2

let printColorName (color:Color) =
    match color with
    | Color.Red -> printfn "Red"
    | Color.Green -> printfn "Green"
    | Color.Blue -> printfn "Blue"
    | _ -> ()

printColorName Color.Red
printColorName Color.Green
printColorName Color.Blue
```

識別碼模式

如果模式是形成有效識別碼的字元字串，則識別碼的格式會決定模式的比對方式。如果識別碼的長度超過單一字元，並以大寫字元開頭，則編譯器會嘗試使其符合識別碼模式。此模式的識別碼可能是以常值屬性標記的值、差異聯集大小寫、例外狀況識別碼或使用中的模式案例。如果找不到相符的識別碼，則比對會失敗，而下一個模式規則(變數模式)會與輸入進行比較。

差異聯集模式可以是簡單名稱的案例，也可以具有值或包含多個值的元組。如果有值，您必須指定值的識別碼。在元組的案例中，您必須為元組的每個專案提供具有識別碼的元組模式，或使用一個或多個命名聯集欄位的功能變數名稱來提供識別碼。如需範例，請參閱本節中的程式碼範例。

此 `option` 類型是具有兩個案例的差異聯集，`Some` 以及 `None`。其中一個案例 (`Some`) 有一個值，但另一個 (`None`) 只是一個命名的案例。因此，`Some` 必須有與案例相關聯之值的變數 `Some`，但 `None` 必須單獨出現。在下列程式碼中，會將變數 `var1` 指定為符合大小寫所取得的值 `Some`。

```
let printOption (data : int option) =
    match data with
    | Some var1 -> printfn "%d" var1
    | None -> ()
```

在下列範例中，差異聯集 `PersonName` 集包含混合的字串和字元，表示可能的名稱形式。差異聯集的案例為 `FirstOnly`、`LastOnly` 和 `FirstLast`。

```
type PersonName =
    | FirstOnly of string
    | LastOnly of string
    | FirstLast of string * string

let constructQuery personName =
    match personName with
    | FirstOnly(firstName) -> printf "May I call you %s?" firstName
    | LastOnly(lastName) -> printf "Are you Mr. or Ms. %s?" lastName
    | FirstLast(firstName, lastName) -> printf "Are you %s %s?" firstName lastName
```

針對具有命名欄位的差異聯集，您可以使用等號(=)來將命名欄位的值解壓縮。例如，請考慮使用類似如下的宣告的差異聯集。

```
type Shape =
    | Rectangle of height : float * width : float
    | Circle of radius : float
```

您可以使用模式比對運算式中的命名欄位，如下所示。

```
let matchShape shape =
    match shape with
    | Rectangle(height = h) -> printfn $"Rectangle with length %f{h}"
    | Circle(r) -> printfn $"Circle with radius %f{r}"
```

使用命名欄位是選擇性的，因此在上述範例中，`Circle(r)` 和都 `Circle(radius = r)` 有相同的效果。

當您指定多個欄位時，請使用分號(;)作為分隔符號。

```
match shape with
| Rectangle(height = h; width = w) -> printfn $"Rectangle with height %f{h} and width %f{w}"
| _ -> ()
```

現用模式可讓您定義更複雜的自訂模式比對。如需使用中模式的詳細資訊，請參閱使用中的 [模式](#)。

在例外狀況處理常式內容中的模式比對中，會使用識別碼為例外狀況的情況。如需例外狀況處理中的模式比對的詳細資訊，請參閱 [例外狀況](#)：`try...with` [運算式](#)。

變數模式

變數模式會將符合的值指派給變數名稱，然後在符號右邊的執行運算式中使用 `->`。變數模式只會比對任何輸入，但變數模式通常會出現在其他模式內，因此可讓您將更複雜的結構（例如元組和陣列）分解為變數。

下列範例示範在元組模式內的變數模式。

```
let function1 x =  
    match x with  
    | (var1, var2) when var1 > var2 -> printfn "%d is greater than %d" var1 var2  
    | (var1, var2) when var1 < var2 -> printfn "%d is less than %d" var1 var2  
    | (var1, var2) -> printfn "%d equals %d" var1 var2  
  
function1 (1,2)  
function1 (2, 1)  
function1 (0, 0)
```

as 模式

`as` 模式是一種已 `as` 附加子句的模式。子句會將 `as` 相符的值系結至運算式的執行運算式中可使用的名稱 `match`，或者，如果在系結中使用此模式，則會將 `let` 名稱加入至區域範圍的系結。

下列範例會使用 `as` 模式。

```
let (var1, var2) as tuple1 = (1, 2)  
printfn "%d %d %A" var1 var2 tuple1
```

或模式

當輸入資料可以比對多個模式，而您想要執行與結果相同的程式碼時，就會使用或模式。或模式兩側的類型都必須相容。

下列範例示範或模式。

```
let detectZeroOR point =  
    match point with  
    | (0, 0) | (0, _) | (_, 0) -> printfn "Zero found."  
    | _ -> printfn "Both nonzero."  
detectZeroOR (0, 0)  
detectZeroOR (1, 0)  
detectZeroOR (0, 10)  
detectZeroOR (10, 15)
```

AND 模式

和模式要求輸入必須符合兩個模式。和模式兩邊的類型都必須相容。

下列範例類似于 `detectZeroTuple` 本主題稍後的「元組模式」一節中所示，但在這裡，`var1` 和 `var2` 都是使用和模式來取得值。

```

let detectZeroAND point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (var1, var2) & (0, _) -> printfn "First value is 0 in (%d, %d)" var1 var2
    | (var1, var2) & (_, 0) -> printfn "Second value is 0 in (%d, %d)" var1 var2
    | _ -> printfn "Both nonzero."
detectZeroAND (0, 0)
detectZeroAND (1, 0)
detectZeroAND (0, 10)
detectZeroAND (10, 15)

```

缺點模式

缺點模式可用來將清單分解為第一個元素、*head*，以及包含其餘元素（*結尾*）的清單。

```

let list1 = [ 1; 2; 3; 4 ]

// This example uses a cons pattern and a list pattern.
let rec printList l =
    match l with
    | head :: tail -> printf "%d " head; printList tail
    | [] -> printfn ""

printList list1

```

清單模式

清單模式可將清單分解成多個元素。清單模式本身只能比對特定元素數目的清單。

```

// This example uses a list pattern.
let listLength list =
    match list with
    | [] -> 0
    | [ _ ] -> 1
    | [ _; _ ] -> 2
    | [ _; _; _ ] -> 3
    | _ -> List.length list

printfn "%d" (listLength [ 1 ])
printfn "%d" (listLength [ 1; 1 ])
printfn "%d" (listLength [ 1; 1; 1; ])
printfn "%d" (listLength [ ] )

```

陣列模式

陣列模式與清單模式類似，可用於分解特定長度的陣列。

```
// This example uses array patterns.
let vectorLength vec =
    match vec with
    | [] var1 [] -> var1
    | [] var1; var2 [] -> sqrt (var1*var1 + var2*var2)
    | [] var1; var2; var3 [] -> sqrt (var1*var1 + var2*var2 + var3*var3)
    | _ -> failwith (sprintf "vectorLength called with an unsupported array size of %d." (vec.Length))

printfn "%f" (vectorLength [| 1. |])
printfn "%f" (vectorLength [| 1.; 1. |])
printfn "%f" (vectorLength [| 1.; 1.; 1.; |])
printfn "%f" (vectorLength [| |])
```

括弧模式

括弧可在模式周圍分組，以達成所需的關聯性。在下列範例中，括弧是用來控制和模式之間的關聯性，以及缺點模式之間的關聯性。

```
let countValues list value =
    let rec checkList list acc =
        match list with
        | (elem1 & head) :: tail when elem1 = value -> checkList tail (acc + 1)
        | head :: tail -> checkList tail acc
        | [] -> acc
    checkList list 0

let result = countValues [ for x in -10..10 -> x*x - 4 ] 0
printfn "%d" result
```

元組模式

元組模式會比對元組形式的輸入，並使用元組中的每個位置的模式比對變數，將元組分解為其組成元素。

下列範例將示範元組模式，也會使用常值模式、變數模式和萬用字元模式。

```
let detectZeroTuple point =
    match point with
    | (0, 0) -> printfn "Both values zero."
    | (0, var2) -> printfn "First value is 0 in (0, %d)" var2
    | (var1, 0) -> printfn "Second value is 0 in (%d, 0)" var1
    | _ -> printfn "Both nonzero."
detectZeroTuple (0, 0)
detectZeroTuple (1, 0)
detectZeroTuple (0, 10)
detectZeroTuple (10, 15)
```

記錄模式

記錄模式用於分解記錄，以將欄位值解壓縮。模式不需要參考記錄的所有欄位;任何省略的欄位都不會參與比對，也不會解壓縮。


```
// This example uses a record pattern.

type MyRecord = { Name: string; ID: int }

let IsMatchByName record1 (name: string) =
    match record1 with
    | { MyRecord.Name = nameFound; MyRecord.ID = _; } when nameFound = name -> true
    | _ -> false

let recordX = { Name = "Parker"; ID = 10 }
let isMatched1 = IsMatchByName recordX "Parker"
let isMatched2 = IsMatchByName recordX "Hartono"
```

萬用字元模式

萬用字元模式是以底線 `()` 字元表示，並比對 `_` 任何輸入（如同變數模式），但會捨棄輸入而不是指派給變數。萬用字元模式通常用於其他模式中，作為符號右邊運算式中不需要的值預留位置 `->`。萬用字元模式也經常用於模式清單的結尾，以符合任何不相符的輸入。本主題的許多程式碼範例會示範萬用字元模式。請參閱上述程式碼中的一個範例。

具有類型注釋的模式

模式可以有類型注釋。這些行為類似其他類型注釋和參考參考，例如其他類型注釋。模式中的類型注釋周圍需要括弧。下列程式碼顯示具有類型注釋的模式。

```
let detect1 x =
    match x with
    | 1 -> printfn "Found a 1!"
    | (var1 : int) -> printfn "%d" var1
detect1 0
detect1 1
```

類型測試模式

型別測試模式是用來比對輸入與型別。如果輸入類型與 (的相符項或) 類型中所指定之類型的衍生型別相符，則比對成功。

下列範例示範型別測試模式。

```
open System.Windows.Forms

let RegisterControl(control:Control) =
    match control with
    | :? Button as button -> button.Text <- "Registered."
    | :? CheckBox as checkbox -> checkbox.Text <- "Registered."
    | _ -> ()
```

如果您只是要檢查識別碼是否屬於特定的衍生類型，則不需要模式的 `as identifier` 部分，如下列範例所示：

```
type A() = class end
type B() = inherit A()
type C() = inherit A()

let m (a: A) =
    match a with
    | :? B -> printfn "It's a B"
    | :? C -> printfn "It's a C"
    | _ -> ()
```

Null 模式

Null 模式符合當您使用允許 null 值的類型時，可能出現的 null 值。當與 .NET Framework 程式碼交互操作時，經常會使用 Null 模式。例如，.NET API 的傳回值可能是運算式的輸入 `match`。您可以根據傳回值是否為 null，以及傳回值的其他特性來控制程式流程。您可以使用 null 模式來防止 null 值傳播至程式的其餘部分。

下列範例會使用 null 模式和變數模式。

```
let ReadFromFile (reader : System.IO.StreamReader) =
    match reader.ReadLine() with
    | null -> printfn "\n"; false
    | line -> printfn "%s" line; true

let fs = System.IO.File.Open("../..\\Program.fs", System.IO.FileMode.Open)
let sr = new System.IO.StreamReader(fs)
while ReadFromFile(sr) = true do ()
sr.Close()
```

Nameof 模式

`nameof` 當字符串值等於關鍵字後面的運算式時，模式會與字符串相符 `nameof`。例如：

```
let f (str: string) =
    match str with
    | nameof str -> "It's 'str'!"
    | _ -> "It is not 'str'!"

f "str" // matches
f "asdf" // does not match
```

`nameof` 如需您可以使用之名稱的相關資訊，請參閱運算子。

另請參閱

- [比對運算式](#)
- [現用模式](#)
- [F # 語言參考](#)

現用模式

2020/11/2 • [Edit Online](#)

*現用模式*可讓您定義用來細分輸入資料的命名分割，如此一來，您就可以在模式比對運算式中使用這些名稱，就像您針對區分聯集所做的一樣。您可以使用作用中的模式，以自訂方式分解每個部分的資料。

語法

```
// Active pattern of one choice.
let (|identifier|) [arguments] valueToMatch = expression

// Active Pattern with multiple choices.
// Uses a FSharp.Core.Choice<_,...,> based on the number of case names. In F#, the limitation n <= 7
applies.
let (|identifer1|identifier2|...|) valueToMatch = expression

// Partial active pattern definition.
// Uses a FSharp.Core.option<_> to represent if the type is satisfied at the call site.
let (|identifier|_|) [arguments ] valueToMatch = expression
```

備註

在先前的語法中，識別碼是以 *引數* 表示之輸入資料的資料分割名稱，換句話說，是引數的所有值集合的子集名稱。現用模式定義中最多可以有七個數據分割。*運算式* 描述要將資料分解成的表單。您可以使用現用模式定義來定義規則，以判斷指定為引數的值屬於哪些已命名的分割區。(| 和 |) 符號稱為 *香蕉剪輯*，而這種類型的 let 系結所建立的函式稱為 *作用中辨識器*。

例如，請考慮下列具有引數的現用模式。

```
let (|Even|Odd|) input = if input % 2 = 0 then Even else Odd
```

您可以使用模式比對運算式中的現用模式，如下列範例所示。

```
let TestNumber input =
    match input with
    | Even -> printfn "%d is even" input
    | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

此程式的輸出如下所示：

```
7 is odd
11 is odd
32 is even
```

使用中模式的另一種用法是以多種方式分解資料類型，例如，當相同的基礎資料有各種可能的標記法時。例如，`Color` 物件可能會分解成 RGB 標記法或 HSB 標記法。

```

open System.Drawing

let (|RGB|) (col : System.Drawing.Color) =
    ( col.R, col.G, col.B )

let (|HSB|) (col : System.Drawing.Color) =
    ( col.GetHue(), col.GetSaturation(), col.GetBrightness() )

let printRGB (col: System.Drawing.Color) =
    match col with
    | RGB(r, g, b) -> printfn " Red: %d Green: %d Blue: %d" r g b

let printHSB (col: System.Drawing.Color) =
    match col with
    | HSB(h, s, b) -> printfn " Hue: %f Saturation: %f Brightness: %f" h s b

let printAll col colorString =
    printfn "%s" colorString
    printRGB col
    printHSB col

printAll Color.Red "Red"
printAll Color.Black "Black"
printAll Color.White "White"
printAll Color.Gray "Gray"
printAll Color.BlanchedAlmond "BlanchedAlmond"

```

上述程式的輸出如下所示：

```

Red
Red: 255 Green: 0 Blue: 0
Hue: 360.000000 Saturation: 1.000000 Brightness: 0.500000
Black
Red: 0 Green: 0 Blue: 0
Hue: 0.000000 Saturation: 0.000000 Brightness: 0.000000
White
Red: 255 Green: 255 Blue: 255
Hue: 0.000000 Saturation: 0.000000 Brightness: 1.000000
Gray
Red: 128 Green: 128 Blue: 128
Hue: 0.000000 Saturation: 0.000000 Brightness: 0.501961
BlanchedAlmond
Red: 255 Green: 235 Blue: 205
Hue: 36.000000 Saturation: 1.000000 Brightness: 0.901961

```

結合使用現用模式時，這兩種方式可讓您將資料分割和分解成適當的形式，並針對計算最方便的形式，在適當的資料上執行適當的計算。

產生的模式比對運算式可讓您以方便閱讀的方式寫入資料，大幅簡化可能複雜的分支和資料分析程式碼。

部分現用模式

有時候，您只需要分割部分的輸入空間。在這種情況下，您會撰寫一組符合某些輸入但無法符合其他輸入的部分模式。不一定會產生值的現用模式稱為「*部分現用模式*」；它們具有屬於選項類型的傳回值。若要定義部分現用模式，請在 `_` 香蕉剪輯內的模式清單結尾使用萬用字元 `()`。下列程式碼說明如何使用部分現用模式。

```

let (|Integer|_|) (str: string) =
    let mutable intvalue = 0
    if System.Int32.TryParse(str, &intvalue) then Some(intvalue)
    else None

let (|Float|_|) (str: string) =
    let mutable floatvalue = 0.0
    if System.Double.TryParse(str, &floatvalue) then Some(floatvalue)
    else None

let parseNumeric str =
    match str with
    | Integer i -> printfn "%d : Integer" i
    | Float f -> printfn "%f : Floating point" f
    | _ -> printfn "%s : Not matched." str

parseNumeric "1.1"
parseNumeric "0"
parseNumeric "0.0"
parseNumeric "10"
parseNumeric "Something else"

```

上一個範例的輸出如下所示：

```

1.100000 : Floating point
0 : Integer
0.000000 : Floating point
10 : Integer
Something else : Not matched.

```

使用部分現用模式時，有時個別的選擇可能不相鄰或互斥，但它們不需要。在下列範例中，模式正方形和模式 Cube 不是連續的，因為有些數位同時為正方形和 cube，例如64。下列程式會使用和模式來結合正方形和 Cube 模式。它會列印出最多1000的整數，其中同時為正方形和 cube，以及僅為 cube 的所有整數。

```

let err = 1.e-10

let isNearlyIntegral (x:float) = abs (x - round(x)) < err

let (|Square|_|) (x : int) =
    if isNearlyIntegral (sqrt (float x)) then Some(x)
    else None

let (|Cube|_|) (x : int) =
    if isNearlyIntegral ((float x) ** ( 1.0 / 3.0)) then Some(x)
    else None

let findSquareCubes x =
    match x with
    | Cube x & Square _ -> printfn "%d is a cube and a square" x
    | Cube x -> printfn "%d is a cube" x
    | _ -> ()

[ 1 .. 1000 ] |> List.iter (fun elem -> findSquareCubes elem)

```

輸出如下所示：

```
1 is a cube and a square
8 is a cube
27 is a cube
64 is a cube and a square
125 is a cube
216 is a cube
343 is a cube
512 is a cube
729 is a cube and a square
1000 is a cube
```

參數化現用模式

現用模式一律會針對要比對的專案使用至少一個引數，但它們也可能會採用其他引數，在此情況下，會套用參數化現用模式的名稱。其他引數可讓一般模式特製化。例如，使用正則運算式來剖析字串的現用模式通常會包含正則運算式做為額外的參數，如下列程式碼所示，這也會使用 `Integer` 先前的程式碼範例中定義的部分現用模式。在此範例中，會提供使用適用於各種日期格式之正則運算式的字串，以自訂一般 `ParseRegex` 現用模式。整數現用模式是用來將相符的字串轉換成可傳遞至 `DateTime` 函數的整數。

```
open System.Text.RegularExpressions

// ParseRegex parses a regular expression and returns a list of the strings that match each group in
// the regular expression.
// List.tail is called to eliminate the first element in the list, which is the full matched expression,
// since only the matches for each group are wanted.
let (|ParseRegex|_|) regex str =
    let m = Regex(regex).Match(str)
    if m.Success
    then Some (List.tail [ for x in m.Groups -> x.Value ])
    else None

// Three different date formats are demonstrated here. The first matches two-
// digit dates and the second matches full dates. This code assumes that if a two-digit
// date is provided, it is an abbreviation, not a year in the first century.
let parseDate str =
    match str with
    | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{1,2})$" [Integer m; Integer d; Integer y]
      -> new System.DateTime(y + 2000, m, d)
    | ParseRegex "(\d{1,2})/(\d{1,2})/(\d{3,4})" [Integer m; Integer d; Integer y]
      -> new System.DateTime(y, m, d)
    | ParseRegex "(\d{1,4})-(\d{1,2})-(\d{1,2})" [Integer y; Integer m; Integer d]
      -> new System.DateTime(y, m, d)
    | _ -> new System.DateTime()

let dt1 = parseDate "12/22/08"
let dt2 = parseDate "1/1/2009"
let dt3 = parseDate "2008-1-15"
let dt4 = parseDate "1995-12-28"

printfn "%s %s %s %s" (dt1.ToString()) (dt2.ToString()) (dt3.ToString()) (dt4.ToString())
```

先前的程式碼的輸出如下所示：

```
12/22/2008 12:00:00 AM 1/1/2009 12:00:00 AM 1/15/2008 12:00:00 AM 12/28/1995 12:00:00 AM
```

現用模式不限於模式比對運算式，您也可以在任何 `let`-系結上使用它們。

```
let (|Default|) onNone value =  
    match value with  
    | None -> onNone  
    | Some e -> e  
  
let greet (Default "random citizen" name) =  
    printfn "Hello, %s!" name  
  
greet None  
greet (Some "George")
```

先程式碼的輸出如下所示：

```
Hello, random citizen!  
Hello, George!
```

另請參閱

- [F # 語言參考](#)
- [比對運算式](#)

迴圈：for..to 運算式

2019/11/27 • [Edit Online](#)

`for...to` 運算式是用來在迴圈變數的值範圍內反復執行迴圈。

語法

```
for identifier = start [ to | downto ] finish do
    body-expression
```

備註

識別碼的型別是從開始和完成運算式的型別推斷而來。這些運算式的類型必須是32位整數。

雖然在技術上是運算式，但 `for...to` 更像命令式程式設計語言中的傳統語句。主體運算式的傳回類型必須是 `unit`。下列範例示範 `for...to` 運算式的各種用法。

```
// A simple for...to loop.
let function1() =
    for i = 1 to 10 do
        printf "%d " i
    printfn ""

// A for...to loop that counts in reverse.
let function2() =
    for i = 10 downto 1 do
        printf "%d " i
    printfn ""

function1()
function2()

// A for...to loop that uses functions as the start and finish expressions.
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function3 x y =
    for i = (beginning x y) to (ending x y) do
        printf "%d " i
    printfn ""

function3 10 4
```

上述程式碼的輸出如下。

```
1 2 3 4 5 6 7 8 9 10
10 9 8 7 6 5 4 3 2 1
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

另請參閱

- [F# 語言參考](#)
- [迴圈：for...in 運算式](#)

- 迴圈: `while...do` 運算式

迴圈：for...in 運算式

2019/10/23 • [Edit Online](#)

這個迴圈結構是用來反復查看可列舉集合中的模式相符專案，例如範圍運算式、序列、清單、陣列或其他支援列舉的結構。

語法

```
for pattern in enumerable-expression do
    body-expression
```

備註

運算式可以與其他 .net 語言中 `for each` 的語句比較，因為它是用來對可列舉集合中的值進行迴圈處理。

`for...in` 不過，`for...in` 也支援在集合上進行模式比對，而不只是在整個集合上反覆運算。

可列舉的運算式可以指定為可列舉集合，或使用 `..` 運算子做為整數類型的範圍。可列舉集合包含清單、序列、陣列、集合、對應等等。`System.Collections.IEnumerable` 可以使用任何可執行檔型別。

當您使用 `..` 運算子來表示範圍時，您可以使用下列語法。

啟動。 *finish*

您也可以使用包含名為 *skip* 之遞增值的版本，如下列程式碼所示。

啟動。 略過。 *finish*

當您使用整數範圍和簡單計數器變數做為模式時，一般的行為是在每個反復專案上將計數器變數遞增1，但如果範圍包含 *skip* 值，則會改由 *skip* 值來遞增計數器。

在此模式中符合的值也可以在主體運算式中使用。

下列程式碼範例說明 `for...in` 運算式的用法。

```
// Looping over a list.
let list1 = [ 1; 5; 100; 450; 788 ]
for i in list1 do
    printfn "%d" i
```

輸出如下。

```
1
5
100
450
788
```

下列範例顯示如何在序列上執行迴圈，以及如何使用元組模式，而不是簡單變數。

```
let seq1 = seq { for i in 1 .. 10 -> (i, i*i) }
for (a, asqr) in seq1 do
    printfn "%d squared is %d" a asqr
```

輸出如下。

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

下列範例示範如何在簡單的整數範圍內執行迴圈。

```
let function1() =
    for i in 1 .. 10 do
        printf "%d " i
    printfn ""
function1()
```

Function1 的輸出如下所示。

```
1 2 3 4 5 6 7 8 9 10
```

下列範例示範如何使用略過2的範圍來迴圈，其中包括範圍的每個其他元素。

```
let function2() =
    for i in 1 .. 2 .. 10 do
        printf "%d " i
    printfn ""
function2()
```

的輸出 `function2` 如下所示。

```
1 3 5 7 9
```

下列範例顯示如何使用字元範圍。

```
let function3() =
    for c in 'a' .. 'z' do
        printf "%c " c
    printfn ""
function3()
```

的輸出 `function3` 如下所示。

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

下列範例顯示如何使用反向反覆運算的負略過值。

```
let function4() =
    for i in 10 .. -1 .. 1 do
        printf "%d " i
    printfn " ... Lift off!"
function4()
```

的輸出 `function4` 如下所示。

```
10 9 8 7 6 5 4 3 2 1 ... Lift off!
```

範圍的開始和結束也可以是運算式，例如函式，如下列程式碼所示。

```
let beginning x y = x - 2*y
let ending x y = x + 2*y

let function5 x y =
    for i in (beginning x y) .. (ending x y) do
        printf "%d " i
    printfn ""

function5 10 4
```

`function5` 具有此輸入的輸出如下所示。

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

下一個範例顯示當迴圈中不需要元素時，使用萬用字元（`_`）。

```
let mutable count = 0
for _ in list1 do
    count <- count + 1
printfn "Number of elements in list1: %d" count
```

輸出如下。

```
Number of elements in list1: 5
```

Note 您可以在 `for...in` 序列運算式和其他計算運算式中使用，在此情況下會使用自 `for...in` 定義版本的運算式。如需詳細資訊，請參閱[序列](#)、[非同步工作流程](#)和[計算運算式](#)。

另請參閱

- [F# 語言參考](#)
- [環回 `for...to` 運算式](#)
- [環回 `while...do` 運算式](#)

迴圈：while...do 運算式

2019/10/23 • [Edit Online](#)

當 `while...do` 指定的測試條件為 `true` 時，會使用運算式來執行反復執行(迴圈)。

語法

```
while test-expression do
  body-expression
```

備註

測試運算式會進行評估;如果為 `true`，則會執行本文運算式，並再次評估測試運算式。內文運算式的類型 `unit` 必須是。如果測試運算式為 `false` 則反復專案會結束。

下列範例說明 `while...do` 運算式的用法。

```
open System

let lookForValue value maxValue =
    let mutable continueLooping = true
    let randomNumberGenerator = new Random()
    while continueLooping do
        // Generate a random number between 1 and maxValue.
        let rand = randomNumberGenerator.Next(maxValue)
        printf "%d " rand
        if rand = value then
            printfn "\nFound a %d!" value
            continueLooping <- false

lookForValue 10 20
```

先前程式碼的輸出是介於1到20之間的亂數字資料流程，最後一個是10。

```
13 19 8 18 16 2 10
Found a 10!
```

NOTE

您可以在 `while...do` 序列運算式和其他計算運算式中使用，在此情況下會使用自 `while...do` 定義版本的運算式。如需詳細資訊，請參閱[序列](#)、[非同步工作流程](#)和[計算運算式](#)。

另請參閱

- [F# 語言參考](#)
- [環回 `for...in` 運算式](#)
- [環回 `for...to` 運算式](#)

判斷提示

2019/10/25 • [Edit Online](#)

`assert` 運算式是一個可供您用來測試運算式的偵錯工具。當運算式在偵測模式中發生錯誤時，判斷提示會顯示系統錯誤對話方塊。

語法

```
assert condition
```

備註

`assert` 運算式的類型 `bool -> unit`。

`assert` 函式會解析為 `Debug.Assert`。這表示其行為等同於直接呼叫 `Debug.Assert`。

只有當您在 Debug 模式下編譯時，才會啟用判斷提示檢查；也就是，如果已定義常數 `DEBUG`。在專案系統中，預設會在 [偵錯工具] 設定中定義 `DEBUG` 常數，而不是在 [發行] 設定中。

無法使用 F# 例外狀況處理來攔截判斷提示失敗錯誤。

範例

下列程式碼範例說明如何使用 `assert` 運算式。

```
let subtractUnsigned (x : uint32) (y : uint32) =  
    assert (x > y)  
    let z = x - y  
    z  
  
// This code does not generate an assertion failure.  
let result1 = subtractUnsigned 2u 1u  
// This code generates an assertion failure.  
let result2 = subtractUnsigned 1u 2u
```

請參閱

- [F# 語言參考](#)

例外狀況處理

2019/11/4 • [Edit Online](#)

本節包含 F# 語言例外狀況處理支援的相關資訊。

例外狀況處理基本概念

例外狀況處理是 .NET Framework 中處理錯誤狀況的標準方式。因此，任何 .NET 語言都必須支援此機制，包括 F#。「例外狀況」是封裝錯誤之相關資訊的物件。發生錯誤時，即會引發例外狀況並停止一般執行。而執行階段會針對例外狀況搜尋適合的處理常式。搜尋會在目前的函式中開始執行，繼續向上搜尋堆疊，遍及呼叫端層級，直到找到相符的處理常式為止。接著，就會執行這個處理常式。

此外，當堆疊回溯時，執行階段會執行 `finally` 區塊中的所有程式碼，確保在回溯程序期間正確清除物件。

相關主題

「	」
例外狀況類型	描述如何宣告例外狀況類型。
例外狀況: <code>try...with</code> 運算式	描述支援例外狀況處理的語言建構。
例外狀況: <code>try...finally</code> 運算式	描述當例外狀況擲回時，可讓您於堆疊回溯時執行清除程式碼的語言建構。
例外狀況: <code>raise</code> 函式	描述如何擲回例外狀況物件。
例外狀況: <code>failwith</code> 函式	描述如何產生一般 F# 例外狀況。
例外狀況: <code>invalidArg</code> 函式	描述如何產生無效引數例外狀況。

例外狀況類型

2020/11/2 • [Edit Online](#)

F # 中有兩種例外狀況類別：.NET 例外狀況類型和 F # 例外狀況類型。本主題說明如何定義和使用 F # 例外狀況類型。

語法

```
exception exception-type of argument-type
```

備註

在先前的語法中，*例外狀況類型* 是新 F # 例外狀況類型的名稱，而 *引數類型* 代表當您引發此類型的例外狀況時，可以提供的引數類型。您可以使用 *引數類型* 的元組類型來指定多個引數。

F # 例外狀況的一般定義如下所示。

```
exception MyError of string
```

您可以使用函數來產生此類型的例外狀況 `raise`，如下所示。

```
raise (MyError("Error message"))
```

您可以直接在運算式中的篩選準則中使用 F # 例外狀況類型 `try...with`，如下列範例所示。

```
exception Error1 of string
// Using a tuple type as the argument type.
exception Error2 of string * int

let function1 x y =
    try
        if x = y then raise (Error1("x"))
        else raise (Error2("x", 10))
    with
        | Error1(str) -> printfn "Error1 %s" str
        | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

您使用 F # 中的關鍵字定義的例外狀況類型 `exception` 是繼承自的新類型 `System.Exception`。

另請參閱

- [例外狀況處理](#)
- [例外狀況](#): `raise` 函式
- [例外狀況階層](#)

例外狀況：Try...with 運算式

2019/10/23 • [Edit Online](#)

本主題描述 `try...with` 運算式, 這是在F#語言中用於例外狀況處理的運算式。

語法

```
try
    expression1
with
| pattern1 -> expression2
| pattern2 -> expression3
...
```

備註

運算式是用來處理中F#的例外狀況。`try...with` 這類似于中 `try...catch` C#的語句。在上述語法中, *運算式* 中的程式碼可能會產生例外狀況。`try...with` 運算式會傳回值。如果未擲回任何例外狀況, 則整個運算式會傳回值為 *運算式*。如果擲回例外狀況, 每個 *模式* 會依序與例外狀況進行比較, 而針對第一個比對模式, 會執行對應的 *運算式* (稱為 *例外狀況處理常式*), 並使用整體運算式傳回該例外狀況處理常式中運算式的值。如果沒有符合的模式, 例外狀況會傳播呼叫堆疊, 直到找到相符的處理常式為止。在例外狀況處理常式中, 從每個運算式傳回的數值型別, 必須符合從 `try` 區塊中的運算式傳回的類型。

通常, 發生錯誤的事實也表示沒有可從每個例外狀況處理常式中的運算式傳回的有效值。常見的模式是讓運算式的類型是選項類型。下列程式碼範例說明此模式。

```
let divide1 x y =
    try
        Some (x / y)
    with
        | :? System.DivideByZeroException -> printfn "Division by zero!"; None

let result1 = divide1 100 0
```

例外狀況可以是 .NET 例外狀況, 也可以F#是例外狀況。您可以使用F# `exception` 關鍵字來定義例外狀況。

您可以使用各種不同的模式來篩選例外狀況類型和其他條件; 下表摘要說明這些選項。

"	"
<code>:? exception-type</code>	符合指定的 .NET 例外狀況類型。
<code>:? 例外狀況-類型做為識別碼</code>	符合指定的 .NET 例外狀況類型, 但提供例外狀況的已命名值。
<code>exception-name(arguments)</code>	符合F#例外狀況類型, 並系結引數。
<code>identifier</code>	符合任何例外狀況, 並將名稱系結至例外狀況物件。相當於 <code>:? Exception ■識別碼</code>
<code>條件的識別碼</code>	如果條件為 true, 則符合任何例外狀況。

範例

下列程式碼範例說明如何使用各種例外狀況處理常式模式。

```
// This example shows the use of the as keyword to assign a name to a
// .NET exception.
let divide2 x y =
    try
        Some( x / y )
    with
        | :? System.DivideByZeroException as ex -> printfn "Exception! %s " (ex.Message); None

// This version shows the use of a condition to branch to multiple paths
// with the same exception.
let divide3 x y flag =
    try
        x / y
    with
        | ex when flag -> printfn "TRUE: %s" (ex.ToString()); 0
        | ex when not flag -> printfn "FALSE: %s" (ex.ToString()); 1

let result2 = divide3 100 0 true

// This version shows the use of F# exceptions.
exception Error1 of string
exception Error2 of string * int

let function1 x y =
    try
        if x = y then raise (Error1("x"))
        else raise (Error2("x", 10))
    with
        | Error1(str) -> printfn "Error1 %s" str
        | Error2(str, i) -> printfn "Error2 %s %d" str i

function1 10 10
function1 9 2
```

NOTE

結構是 `try...finally` 運算式中的另一個運算式。 `try...with` 因此, 如果您的 `with` 程式碼同時需要區塊 `finally` 和區塊, 您就必須將這兩個運算式加以嵌套。

NOTE

您可以在 `try...with` 非同步工作流程和其他計算運算式中使用, 在此情況下會使用 `try...with` 自訂版本的運算式。如需詳細資訊, 請參閱 [非同步工作流程](#) 和 [計算運算式](#)。

另請參閱

- [例外狀況處理](#)
- [例外狀況類型](#)
- 例外狀況: `try...finally` 運算式

例外狀況：Try...try...finally 運算式

2019/10/23 • [Edit Online](#)

`try...finally` 運算式可讓您執行清理程式碼，即使程式碼區塊擲回例外狀況也一樣。

語法

```
try
    expression1
finally
    expression2
```

備註

運算式可用來在上述語法中，以運算式2執行程式碼，而不論是否在執行運算式的過程中產生例外狀況。

`try...finally`

運算式類型不會對整個運算式的值造成影響；未發生例外狀況時傳回的型別，是運算式類型中的最後一個值。發生例外狀況時，不會傳回任何值，而且控制權的流程會傳送至呼叫堆疊的下一個相符的例外狀況處理常式。如果找不到任何例外狀況處理常式，則程式會終止。在執行比對處理常式中的程式碼或程式終止之前，會執行

`finally` 分支中的程式碼。

下列程式碼示範 `try...finally` 運算式的用法。

```
let divide x y =
    let stream : System.IO.FileStream = System.IO.File.Create("test.txt")
    let writer : System.IO.StreamWriter = new System.IO.StreamWriter(stream)
    try
        writer.WriteLine("test1");
        Some( x / y )
    finally
        writer.Flush()
        printfn "Closing stream"
        stream.Close()

let result =
    try
        divide 100 0
    with
        | :? System.DivideByZeroException -> printfn "Exception handled."; None
```

主控台的輸出如下所示。

```
Closing stream
Exception handled.
```

如您在輸出中所見，在處理外部例外狀況之前，資料流程已關閉，而且 `test.txt` 檔案包含文字 `test1`，這表示緩衝區已排清並寫入磁片，即使已傳送例外狀況也是一樣。控制外部例外狀況處理常式。

請注意，`try...with` 結構是 `try...finally` 來自結構的個別結構。因此，如果您的 `with` 程式碼同時需要區塊 `finally` 和區塊，您就必須將這兩個結構加以嵌套，如下列程式碼範例所示。

```
exception InnerError of string
exception OuterError of string

let function1 x y =
    try
        try
            if x = y then raise (InnerError("inner"))
            else raise (OuterError("outer"))
        with
            | InnerError(str) -> printfn "Error1 %s" str
    finally
        printfn "Always print this."

let function2 x y =
    try
        function1 x y
    with
        | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

在計算運算式(包括順序運算式和非同步工作流程)的內容中, 嘗試 ...最後, 運算式可以有自訂的實作為。如需詳細資訊, 請參閱[計算運算式](#)。

另請參閱

- [例外狀況處理](#)
- 例外狀況: `try...with` 運算式

例外狀況：raise 函式

2019/10/23 • [Edit Online](#)

`raise` 函數是用來表示發生錯誤或例外狀況。錯誤的相關資訊會在例外狀況物件中捕捉。

語法

```
raise (expression)
```

備註

`raise` 函式會產生例外狀況物件, 並起始堆疊回溯進程。堆疊回溯程式是由 common language runtime (CLR) 所管理, 因此此進程的行為與任何其他 .NET 語言相同。堆疊回溯程式是搜尋符合所產生之例外狀況的例外狀況處理常式。搜尋會從目前 `try...with` 的運算式開始 (如果有的話)。區塊中的 `with` 每個模式會依序進行檢查。找到相符的例外狀況處理常式時, 會將例外狀況視為已處理; 否則, 會先展開堆疊並 `with` 封鎖呼叫鏈, 直到找到相符的處理常式為止。當 `finally` 堆疊回溯時, 在呼叫鏈中遇到的任何區塊也會依序執行。

函數等同於或 C++ 中的 C# `throw` `raise` 在 `reraise` `catch` 處理常式中使用, 將相同的例外狀況向上傳播至呼叫鏈。

下列程式碼範例說明如何使用 `raise` 函數來產生例外狀況。

```
exception InnerError of string
exception OuterError of string

let function1 x y =
    try
        try
            if x = y then raise (InnerError("inner"))
            else raise (OuterError("outer"))
        with
            | InnerError(str) -> printfn "Error1 %s" str
    finally
        printfn "Always print this."

let function2 x y =
    try
        function1 x y
    with
        | OuterError(str) -> printfn "Error2 %s" str

function2 100 100
function2 100 10
```

`raise` 函數也可以用來引發 .net 例外狀況, 如下列範例所示。

```
let divide x y =
    if (y = 0) then raise (System.ArgumentException("Divisor cannot be zero!"))
    else
        x / y
```

另請參閱

- [例外狀況處理](#)
- [例外狀況類型](#)
- [例外狀況](#): `try...with` [運算式](#)
- [例外狀況](#): `try...finally` [運算式](#)
- [例外狀況](#): `failwith` [函式](#)
- [例外狀況](#): `invalidArg` [函式](#)

例外狀況：Failwith 函式

2019/10/23 • [Edit Online](#)

函式F#會產生例外狀況。 `failwith`

語法

```
failwith error-message-string
```

備註

前述語法中的**錯誤訊息字串**為常值字串或類型 `string` 的值。它會成為 `Message` 例外狀況的屬性。

所產生 `failwith` 的例外狀況 `System.Exception` 是例外狀況, 這是在程式碼中 `Failure` F#具有名稱的參考。下列程式碼說明 `failwith` 如何使用來擲回例外狀況。

```
let divideFailwith x y =
    if (y = 0) then failwith "Divisor cannot be zero."
    else
        x / y

let testDivideFailwith x y =
    try
        divideFailwith x y
    with
        | Failure(msg) -> printfn "%s" msg; 0

let result1 = testDivideFailwith 100 0
```

另請參閱

- [例外狀況處理](#)
- [例外狀況類型](#)
- 例外狀況: `try...with` 運算式
- 例外狀況: `try...finally` 運算式
- 例外狀況: `raise` 函式

例外狀況：InvalidArg 函式

2019/10/23 • [Edit Online](#)

`invalidArg` 函式會產生引數例外狀況。

語法

```
invalidArg parameter-name error-message-string
```

備註

先前語法中的參數名稱是字串，其引數名稱無效。*錯誤訊息字串*是文字字串或類型 `string` 的值。它會成為 `Message` exception 物件的屬性。

所產生 `invalidArg` 的例外狀況 `System.ArgumentException` 是例外狀況。下列程式碼說明 `invalidArg` 如何使用來擲回例外狀況。

```
let months = [| "January"; "February"; "March"; "April";  
               "May"; "June"; "July"; "August"; "September";  
               "October"; "November"; "December" |]  
  
let lookupMonth month =  
    if (month > 12 || month < 1)  
        then invalidArg "month" (sprintf "Value passed in was %d." month)  
           months.[month - 1]  
  
printfn "%s" (lookupMonth 12)  
printfn "%s" (lookupMonth 1)  
printfn "%s" (lookupMonth 13)
```

輸出如下所示，後面接著堆疊追蹤(未顯示)。

```
December  
January  
System.ArgumentException: Month parameter out of range.
```

另請參閱

- [例外狀況處理](#)
- [例外狀況類型](#)
- 例外狀況: `try...with` 運算式
- 例外狀況: `try...finally` 運算式
- 例外狀況: `raise` 函式
- 例外狀況: `failwith` 函式

屬性

2020/2/22 • [Edit Online](#)

屬性可讓中繼資料套用至程式設計結構。

語法

```
[<target:attribute-name(arguments)>]
```

備註

在先前的語法中，*目標*是選擇性的，如果有的話，則會指定要套用屬性的程式實體類型。*Target*的有效值會顯示在本檔稍後所顯示的表格中。

*屬性名稱*是指有效屬性類型的名稱(可能是以命名空間限定)，而不含後置字元 `Attribute` 通常用於屬性類型名稱。例如，您可以將類型 `ObsoleteAttribute` 縮短為在此內容中 `Obsolete`。

*引數*是屬性類型之函式的引數。如果屬性具有無參數的函式，則可以省略引數清單和括弧。屬性同時支援位置引數和具名引數。*位置引數*是以它們出現的順序來使用的引數。如果屬性具有公用屬性，則可以使用具名引數。您可以使用引數清單中的下列語法來設定這些參數。

```
property-name = property-value
```

這類屬性初始化可以依任何順序排列，但必須遵循任何位置引數。以下是使用位置引數和屬性初始化的屬性範例：

```
open System.Runtime.InteropServices

[<DllImport("kernel32", SetLastError=true)>]
extern bool CloseHandle(nativeint handle)
```

在此範例中，屬性是 `DllImportAttribute`，這裡以縮短的形式使用。第一個引數是位置參數，而第二個是屬性。

屬性是一種 .NET 程式設計結構，可讓稱為*屬性的物件*與型別或其他程式元素產生關聯。套用屬性的程式元素稱為*屬性目標*。屬性通常包含其目標的相關中繼資料。在此內容中，中繼資料可能是除了其欄位和成員以外類型的任何資料。

中F#的屬性可以套用至下列程式設計結構：函式、方法、元件、模組、類型(類別、記錄、結構、介面、委派、列舉、等位等等)、程式(attribute)、屬性(property)、欄位、參數、型別參數和傳回值。屬性不允許用於類別、運算式或工作流程運算式中的 `let` 系結。

一般而言，屬性宣告會直接出現在屬性目標的宣告前面。可以同時使用多個屬性宣告，如下所示：

```
[<Owner("Jason Carlson")>]
[<Company("Microsoft")>]
type SomeType1 =
```

您可以在執行時間使用 .NET 反映來查詢屬性。

如先前的程式碼範例所示，您可以個別宣告多個屬性，或者，如果您使用分號來分隔個別屬性和構造函式，則可

以在一組方括弧中宣告它們，如下所示：

```
[<Owner("Darren Parker"); Company("Microsoft")>]
type SomeType2 =
```

通常遇到的屬性包括 `Obsolete` 屬性、安全性考慮的屬性、COM 支援的屬性、與程式碼擁有權相關的屬性，以及指出類型是否可以序列化的屬性。下列範例示範 `Obsolete` 屬性的用法。

```
open System

[<Obsolete("Do not use. Use newFunction instead.")>]
let obsoleteFunction x y =
    x + y

let newFunction x y =
    x + 2 * y

// The use of the obsolete function produces a warning.
let result1 = obsoleteFunction 10 100
let result2 = newFunction 10 100
```

針對 `assembly` 和 `module` 的屬性目標，您可以將屬性套用至元件中的最上層 `do` 系結。您可以在屬性宣告中包含 `assembly` 或 `module` 這一字，如下所示：

```
open System.Reflection
[<assembly:AssemblyVersionAttribute("1.0.0.0")>]
do
    printfn "Executing..."
```

如果您省略套用至 `do` 系結之屬性的屬性目標，則F#編譯器會嘗試判斷對該屬性有意義的屬性目標。許多屬性類別都具有類型 `System.AttributeUsageAttribute` 的屬性，其中包含該屬性所支援之可能目標的相關資訊。如果 `System.AttributeUsageAttribute` 指出屬性支援以函式作為目標，則會採用屬性以套用至程式的主要進入點。如果 `System.AttributeUsageAttribute` 指出屬性支援將元件當做目標，則編譯器會採用屬性以套用至元件。大部分的屬性都不會同時套用至函式和元件，但在這些情況下，會採用屬性以套用至程式的 `main` 函式。如果明確指定屬性目標，則會將屬性套用至指定的目標。

雖然您通常不需要明確指定屬性目標，但在屬性中，*target*的有效值和使用方式範例如下表所示：

目標	範例
組件 (assembly)	<pre>[<assembly: AssemblyVersion("1.0.0.0")>]</pre>
return	<pre>let function1 x : [<return: MyCustomAttributeThatWorksOnReturns>] int = x + 1</pre>
field	<pre>[<DefaultValue>] val mutable x: int</pre>

屬性	<pre>[<Obsolete>] this.MyProperty = x</pre>
參數	<pre>member this.MyMethod([<Out>] x : ref<int>) = x := 10</pre>
type	<pre>[<type: StructLayout(LayoutKind.Sequential)>] type MyStruct = struct val x : byte val y : int end</pre>

另請參閱

- [F# 語言參考](#)

資源管理:Use 關鍵字

2019/10/23 • [Edit Online](#)

本主題描述關鍵字 `use` `using` 和函式, 此函式可以控制資源的初始化和釋放。

資源

「資源」一詞會以多種方式使用。是的, 資源可以是應用程式所使用的資料, 例如字串、圖形等等, 但是在此內容中, 資源指的是軟體或作業系統資源, 例如圖形裝置內容、檔案控制代碼、網路和資料庫。連接、並行物件 (例如等候控制碼等等)。應用程式使用這些資源牽涉到從作業系統或其他資源提供者取得資源, 然後將較新的資源發行至集區, 以便提供給另一個應用程式。當應用程式未將資源釋放回通用集區時, 就會發生問題。

管理資源

若有效率地管理應用程式中的資源, 您必須以可預測的方式立即發行資源。 .NET Framework 藉由提供 `System.IDisposable` 介面來協助您執行這項操作。執行的類型 `System.IDisposable` 具有方法, `System.IDisposable.Dispose` 可正確地釋出資源。妥善撰寫的應用程式保證 `System.IDisposable.Dispose` 當您不再需要持有有限資源的任何物件時, 會立即呼叫。幸運的是, 大部分的 .NET 語言都提供支援, 讓 F# 這種情況變得更容易, 而且也不例外。有兩個實用的語言結構支援 dispose 模式: `use` 系結 `using` 和函式。

使用系結

關鍵字的形式類似 `let` 于系結: `use`

使用 $值 = 運算式$

它會提供與系結相同 `let` 的功能, 但會在 `Dispose` 值超出範圍時, 將對值的呼叫加入。請注意, 編譯器會在值上插入 `null` 檢查, 因此如果值為 `null`, 則不 `Dispose` 會嘗試呼叫。

下列範例顯示如何使用 `use` 關鍵字自動關閉檔案。

```
open System.IO

let writetofile filename obj =
    use file1 = File.CreateText(filename)
    file1.WriteLine("{0}", obj.ToString() )
    // file1.Dispose() is called implicitly here.

writetofile "abc.txt" "Humpty Dumpty sat on a wall."
```

NOTE

您可以在 `use` 計算運算式中使用, 在此情況下會使用自 `use` 定義版本的運算式。如需詳細資訊, 請參閱 [序列](#)、[非同步工作流程](#) 和 [計算運算式](#)。

使用函數

`using` 函數的格式如下:

`using (運算式=) 函數或 lambda`

在運算式中, 會建立必須處置的物件。 `using` 運算式類型的結果 (必須處置的物件) 會變成函式或 *lambda* 的引數、值, 這是需要符合所產生之值的另一個型別引數的函數。運算式型別, 或需要該類型之引數的 `lambda` 運算式。在函式執行結束時, 執行時間會呼叫 `Dispose` 並釋出資源 (除非值為 `null`, 在此情況下不會嘗試對 `Dispose` 進行呼叫)。

下列範例示範具有 `lambda` `using` 運算式的運算式。

```
open System.IO

let writetofile2 filename obj =
    using (System.IO.File.CreateText(filename)) ( fun file1 ->
        file1.WriteLine("{0}", obj.ToString() )
    )

writetofile2 "abc2.txt" "The quick sly fox jumps over the lazy brown dog."
```

下一個範例會顯示 `using` 具有函數的運算式。

```
let printToFile (file1 : System.IO.StreamWriter) =
    file1.WriteLine("Test output");

using (System.IO.File.CreateText("test.txt")) printToFile
```

請注意, 函式可能是已套用一些引數的函式。下列程式碼範例示範此工作。它會建立包含字串 `XYZ` 的檔案。

```
let printToFile2 obj (file1 : System.IO.StreamWriter) =
    file1.WriteLine(obj.ToString())

using (System.IO.File.CreateText("test.txt")) (printToFile2 "XYZ")
```

`using` 函式 `use` 和系結幾乎等同于完成相同工作的方式。關鍵字可讓您更充分掌控呼叫的時間 `Dispose`。
`using` 當您使用 `using` 時 `Dispose`, 會在函式或 `lambda` 運算式的結尾呼叫, 而當您使用 `use` 關鍵字時 `Dispose`, 會在包含程式碼區塊的結尾呼叫。一般來說, 您應該偏好使用 `use`, 而不是 `using` 函式。

另請參閱

- [F# 語言參考](#)

命名空間

2020/5/6 • [Edit Online](#)

命名空間可讓您將名稱附加至 F# 程式元素的群組，以將程式碼組織成相關功能的區域。命名空間通常是 F# 檔案中的最上層元素。

語法

```
namespace [rec] [parent-namespaces.]identifier
```

備註

如果您想要將程式碼放在命名空間中，則檔案中的第一個宣告必須宣告命名空間。整個檔案的內容接著會成為命名空間的一部分，前提是檔案中不會有其他命名空間宣告。如果是這種情況，則所有程式碼必須等到下一個命名空間宣告，才會被視為在第一個命名空間內。

命名空間不能直接包含值和函數。相反地，值和函式必須包含在模組中，而模組則包含在命名空間中。命名空間可以包含類型(模組)。

XML 檔批註可以在命名空間的上方宣告，但會被忽略。編譯器指示詞也可以在命名空間的上方宣告。

命名空間可以使用 namespace 關鍵字明確宣告，或在宣告模組時以隱含方式宣告。若要明確宣告命名空間，請使用 namespace 關鍵字並在後面加上命名空間名稱。下列範例顯示的程式碼檔案會宣告具有類型 `Widgets` 的命名空間，以及該命名空間中包含的模組。

```
namespace Widgets

type MyWidget1 =
    member this.WidgetName = "Widget1"

module WidgetsModule =
    let widgetName = "Widget2"
```

如果檔案的整個內容都在一個模組中，您也可以使用 `module` 關鍵字，並在完整的模組名稱中提供新的命名空間名稱，以隱含方式宣告命名空間。下列範例顯示的程式碼檔案會宣告命名空間 `Widgets` 和包含函 `WidgetsModule` 式的模組。

```
module Widgets.WidgetModule

let widgetFunction x y =
    printfn "%A %A" x y
```

下列程式碼相當於上述程式碼，但模組是本機模組宣告。在此情況下，命名空間必須出現在自己的行上。

```
namespace Widgets

module WidgetModule =

    let widgetFunction x y =
        printfn "%A %A" x y
```

如果一個或多個命名空間中的同一個檔案需要一個以上的模組，您就必須使用本機模組宣告。當您使用本機模組宣告時，不能在模組宣告中使用限定的命名空間。下列程式碼顯示具有命名空間宣告和兩個本機模組宣告的檔案。在此情況下，模組會直接包含在命名空間中;沒有與檔案同名的隱含建立模組。檔案中的任何其他程式碼（例如 `do` 系結）都是在命名空間中，而不是在內部模組中，因此您必須使用 `widgetFunction` 模組名稱來限定模組成員。

```
namespace Widgets

module WidgetModule1 =
    let widgetFunction x y =
        printfn "Module1 %A %A" x y
module WidgetModule2 =
    let widgetFunction x y =
        printfn "Module2 %A %A" x y

module useWidgets =

    do
        WidgetModule1.widgetFunction 10 20
        WidgetModule2.widgetFunction 5 6
```

此範例的輸出如下所示。

```
Module1 10 20
Module2 5 6
```

如需詳細資訊，請參閱[模組](#)。

嵌套命名空間

當您建立嵌套命名空間時，您必須完整限定它。否則，您會建立新的最上層命名空間。命名空間宣告中會忽略縮排。

下列範例顯示如何宣告嵌套命名空間。

```
namespace Outer

    // Full name: Outer.MyClass
    type MyClass() =
        member this.X(x) = x + 1

// Fully qualify any nested namespaces.
namespace Outer.Inner

    // Full name: Outer.Inner.MyClass
    type MyClass() =
        member this.Prop1 = "X"
```

檔案和元件中的命名空間

命名空間可以在單一專案或編譯中跨越多個檔案。「命名空間片段」一詞描述包含在一個檔案中的命名空間部分。命名空間也可以跨越多個元件。例如，`System` 命名空間包含整個 .NET Framework，這會跨越許多元件，並包含許多嵌套的命名空間。

全域命名空間

您可以使用預先定義 `global` 的命名空間，將名稱放在 `.net` 最上層命名空間中。

```
namespace global

type SomeType() =
    member this.SomeMember = 0
```

您也可以使用全域來參考最上層的 .NET 命名空間，例如，解決與其他命名空間的名稱衝突。

```
global.System.Console.WriteLine("Hello World!")
```

遞迴命名空間

命名空間也可以宣告為遞迴，讓所有包含的程式碼都可以互相遞迴。這是透過來 `namespace rec` 完成。使用 `namespace rec` 可減輕無法在類型和模組之間撰寫相互引用程式碼的一些難題。以下是這種情況的範例：

```
namespace rec MutualReferences

type Orientation = Up | Down
type PeelState = Peeled | Unpeeled

// This exception depends on the type below.
exception DontSqueezeTheBananaException of Banana

type Banana(orientation : Orientation) =
    member val IsPeeled = false with get, set
    member val Orientation = orientation with get, set
    member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled ] with get, set

    member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
    member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on the
    exception above.

module BananaHelpers =
    let peel (b: Banana) =
        let flip (banana: Banana) =
            match banana.Orientation with
            | Up ->
                banana.Orientation <- Down
                banana
            | Down -> banana

        let peelSides (banana: Banana) =
            banana.Sides
            |> List.map (function
                | Unpeeled -> Peeled
                | Peeled -> Peeled)

        match b.Orientation with
        | Up -> b |> flip |> peelSides
        | Down -> b |> peelSides
```

請注意，例外 `DontSqueezeTheBananaException` 狀況和類別 `Banana` 都會彼此參考。此外，模組 `BananaHelpers` 和類別 `Banana` 也會彼此參考。如果您已從 `rec` `MutualReferences` 命名空間移除關鍵字，這將無法以 F # 表示。

這項功能也適用於最上層 [模組](#)。

請參閱

- [F # 語言參考](#)
- [單元](#)

- F # RFC FS-1009-允許檔案內的相互參考型別和模組超過更大的範圍

單元

2020/5/6 • [Edit Online](#)

在 F# 語言的內容中，*模組*是 f# 程式碼的群組，例如 f# 程式中的值、類型和函數值。程式碼分組成不同模組有助於將相關程式碼整理到同一處，以及避免程式中發生名稱衝突。

語法

```
// Top-level module declaration.  
module [accessibility-modifier] [qualified-namespace.]module-name  
declarations  
// Local module declaration.  
module [accessibility-modifier] module-name =  
    declarations
```

備註

F# 模組是 F# 程式碼結構的群組，例如類型、值、函數值和系結中 `do` 的程式碼。它會實作為僅具有靜態成員的 common language runtime (CLR) 類別。有兩種類型的模組宣告，視模組中是否包含整個檔案而定：最上層模組宣告和本機模組宣告。最上層模組宣告包含模組中的整個檔案。最上層模組宣告只能出現為檔案中的第一個宣告。

在最上層模組宣告的語法中，選擇性的*限定命名空間*是包含模組之嵌套命名空間名稱的序列。限定的命名空間不一定要預先宣告。

您不需要在最上層模組中縮排宣告。您必須將本機模組中的所有宣告縮排。在本機模組宣告中，只有在該模組宣告下縮排的宣告是模組的一部分。

如果程式碼檔案不是以最上層模組宣告或命名空間宣告開頭，則檔案的整個內容(包括任何本機模組)會成為隱含建立之最上層模組的一部分，該模組具有與檔案相同的名稱(不含副檔名)，而第一個字母轉換成大寫。例如，請考慮下列檔案。

```
// In the file program.fs.  
let x = 40
```

這個檔案會編譯成以這種方式撰寫的：

```
module Program  
    let x = 40
```

如果您在檔案中有多個模組，您必須針對每個模組使用本機模組宣告。如果宣告了封閉式命名空間，這些模組就是封閉命名空間的一部分。如果未宣告封入命名空間，則模組會成為隱含建立之最上層模組的一部分。下列程式碼範例顯示包含多個模組的程式碼檔案。編譯器會隱含地建立名為 `MultipleModules` 的最上層模組，`MyModule1` 而 `MyModule2` 和會嵌套在該最上層的模組中。

```
// In the file multiplemodules.fs.
// MyModule1
module MyModule1 =
    // Indent all program elements within modules that are declared with an equal sign.
    let module1Value = 100

    let module1Function x =
        x + 10

// MyModule2
module MyModule2 =

    let module2Value = 121

    // Use a qualified name to access the function.
    // from MyModule1.
    let module2Function x =
        x * (MyModule1.module1Function module2Value)
```

如果您在專案或單一編譯中有多個檔案，或者如果您要建立程式庫，則必須在檔案頂端包含命名空間宣告或模組宣告。F# 編譯器只會在專案或編譯命令列中只有一個檔案，而且您正在建立應用程式時，隱含地判斷模組名稱。

協助工具修飾詞可以是下列其中一項：`public`、`private`、`internal`。如需詳細資訊，請參閱[存取控制](#)。預設值是 `public`。

參考模組中的程式碼

當您從另一個模組參考函數、類型和值時，您必須使用限定名稱或開啟模組。如果您使用限定名稱，則必須指定命名空間、模組，以及所需之程式元素的識別碼。您可以使用點(.)來分隔限定路徑的每個部分，如下所示。

```
Namespace1.Namespace2.ModuleName.Identifier
```

您可以開啟模組或一或多個命名空間，以簡化程式碼。如需有關開啟命名空間和模組的詳細資訊，請參閱[匯入宣告](#)：`open` 關鍵字。

下列程式碼範例顯示最上層模組，其中包含直到檔案結尾為止的所有程式碼。

```
module Arithmetic

let add x y =
    x + y

let sub x y =
    x - y
```

若要從相同專案中的另一個檔案使用此程式碼，您可以使用限定名稱，或在使用函式之前開啟模組，如下列範例所示。

```
// Fully qualify the function name.
let result1 = Arithmetic.add 5 9
// Open the module.
open Arithmetic
let result2 = add 5 9
```

嵌套模組

模組可以嵌套。內部模組必須縮排為外部模組宣告，以表示它們是內部模組，而不是新模組。例如，比較下列兩

個範例。模組 `Z` 是下列程式碼中的內部模組。

```
module Y =  
  let x = 1  
  
  module Z =  
    let z = 5
```

但是模組 `Z` 在下列程式碼中 `Y` 是模組的同級。

```
module Y =  
  let x = 1  
  
module Z =  
  let z = 5
```

模組 `Z` 也是下列程式碼中的同輩模組，因為它不會縮排至模組 `Y` 中的其他宣告。

```
module Y =  
  let x = 1  
  
  module Z =  
    let z = 5
```

最後，如果外部模組沒有宣告，後面緊接著另一個模組宣告，則會假設新的模組宣告為內部模組，但如果第二個模組定義的縮排比第一個更遠，編譯器會警告您。

```
// This code produces a warning, but treats Z as a inner module.  
module Y =  
  module Z =  
    let z = 5
```

若要排除警告，請將內部模組縮排。

```
module Y =  
  module Z =  
    let z = 5
```

如果您想要將檔案中的所有程式碼都放在單一外部模組中，而您想要使用內部模組，則外部模組不需要等號，而且外部模組中的宣告（包括任何內部模組宣告）都不必縮排。內部模組宣告內的宣告必須縮排。下列程式碼會顯示這種情況。

```
// The top-level module declaration can be omitted if the file is named  
// TopLevel.fs or topLevel.fs, and the file is the only file in an  
// application.  
module TopLevel  
  
  let topLevelX = 5  
  
  module Inner1 =  
    let inner1X = 1  
  module Inner2 =  
    let inner2X = 5
```

遞迴模組

F # 4.1 引進模組的概念，可讓所有內含的程式碼互相遞迴。這是透過來 `module rec` 完成。使用 `module rec` 可減輕無法在類型和模組之間撰寫相互引用程式碼的一些難題。以下是這種情況的範例：

```
module rec RecursiveModule =
  type Orientation = Up | Down
  type PeelState = Peeled | Unpeeled

  // This exception depends on the type below.
  exception DontSqueezeTheBananaException of Banana

  type Banana(orientation : Orientation) =
    member val IsPeeled = false with get, set
    member val Orientation = orientation with get, set
    member val Sides: PeelState list = [ Unpeeled; Unpeeled; Unpeeled; Unpeeled] with get, set

    member self.Peel() = BananaHelpers.peel self // Note the dependency on the BananaHelpers module.
    member self.SqueezeJuiceOut() = raise (DontSqueezeTheBananaException self) // This member depends on
the exception above.

  module BananaHelpers =
    let peel (b: Banana) =
      let flip (banana: Banana) =
        match banana.Orientation with
        | Up ->
          banana.Orientation <- Down
          banana
        | Down -> banana

      let peelSides (banana: Banana) =
        banana.Sides
        |> List.map (function
          | Unpeeled -> Peeled
          | Peeled -> Peeled)

      match b.Orientation with
      | Up -> b |> flip |> peelSides
      | Down -> b |> peelSides
```

請注意，例外 `DontSqueezeTheBananaException` 狀況和類別 `Banana` 都會彼此參考。此外，模組 `BananaHelpers` 和類別 `Banana` 也會彼此參考。如果您從 `rec RecursiveModule` 模組中移除關鍵字，這將無法以 F # 表示。

這項功能也可以在具有 F # 4.1 的命名空間中使用。

請參閱

- [F # 語言參考](#)
- [命名空間](#)
- [F # RFC FS-1009-允許檔案內的相互參考型別和模組超過更大的範圍](#)

匯入宣告：`open` 關鍵字

2021/3/5 • [Edit Online](#)

匯入宣告會指定模組或命名空間，而您可以參考其元素，而不需使用完整名稱。

語法

```
open module-or-namespace-name
open type type-name
```

備註

每次使用完整命名空間或模組路徑來參考程式碼，可以建立難以撰寫、讀取和維護的程式碼。相反地，您可以將 `open` 關鍵字用於常用的模組和命名空間，如此一來，當您參考該模組或命名空間的成員時，就可以使用名稱的簡短形式，而不是完整名稱。這個關鍵字類似于 `using` C# 中的關鍵字、`using namespace` Visual C++ 中和 `Imports` Visual Basic。

提供的模組或命名空間必須位於相同的專案或參考的專案或元件中。如果不是，您可以加入專案的參考，或使用 `-reference` 命令列選項 (或其縮寫 `-r`)。如需詳細資訊，請參閱[編譯器選項](#)。

匯入宣告會讓宣告後面的程式碼中的名稱可供使用，最多可包含在封入命名空間、模組或檔案的結尾。

當您使用多個匯入宣告時，它們應該會出現在不同的行上。

下列程式碼顯示 `open` 如何使用關鍵字來簡化程式碼。

```
// Without the import declaration, you must include the full
// path to .NET Framework namespaces such as System.IO.
let writeToFile1 filename (text: string) =
    let stream1 = new System.IO.FileStream(filename, System.IO.FileMode.Create)
    let writer = new System.IO.StreamWriter(stream1)
    writer.WriteLine(text)

// Open a .NET Framework namespace.
open System.IO

// Now you do not have to include the full paths.
let writeToFile2 filename (text: string) =
    let stream1 = new FileStream(filename, FileMode.Create)
    let writer = new StreamWriter(stream1)
    writer.WriteLine(text)

writeToFile2 "file1.txt" "Testing..."
```

當有多個開啟的模組或命名空間中發生不明確的名稱時，F# 編譯器不會發出錯誤或警告。發生多義性時，F# 會優先偏好最近開啟的模組或命名空間。例如，在下列程式碼中，`empty` `Seq.empty` 即使 `empty` 位於和模組中，也是一樣 `List` `Seq`。

```
open List
open Seq
printfn "%{empty}"
```

因此，當您開啟包含具有相同名稱之成員的模組或命名空間時，請小心，`List` `Seq` 改為考慮使用限定名稱。您應該避免程式碼相依赖于匯入宣告順序的任何情況。

開放式型別宣告

F# 支援 `open` 型別，如下所示：

```
open type System.Math
PI
```

這會公開所有可存取的靜態欄位和類型的成員。

您也可以使用 `open` F# 定義 [記錄](#) 和差異聯集 類型來公開靜態成員。在區分等位的情況下，您也可以公開聯集案例。這有助於存取在您可能不想要開啟的模組內宣告的型別中的聯集案例，如下所示：

```
module M =
    type DU = A | B | C

    let someOtherFunction x = x + 1

// Open only the type inside the module
open type M.DU

printfn "%A" A
```

預設會開啟的命名空間

有些命名空間在 F# 程式碼中經常會用到，因此會隱含地開啟，而不需要明確的匯入宣告。下表顯示預設開啟的命名空間。

命名空間	說明
<code>FSharp.Core</code>	包含內建類型 (例如和) 的基本 F# 型別定義 <code>int</code> <code>float</code> 。
<code>FSharp.Core.Operators</code>	包含基本的算數運算 <code>+</code> ，例如和 <code>*</code> 。
<code>FSharp.Collections</code>	包含不可變的集合類別 <code>List</code> ，例如和 <code>Array</code> 。
<code>FSharp.Control</code>	包含控制項結構的類型，例如延遲評估和非同步工作流程。
<code>FSharp.Text</code>	包含格式化 IO 的函式，例如 <code>printf</code> 函數。

AutoOpen 屬性

`AutoOpen` 如果您想要在參考元件時自動開啟命名空間或模組，您可以將屬性套用至元件。您也可以將 `AutoOpen` 屬性套用至模組，以在父模組或命名空間開啟時自動開啟該模組。如需詳細資訊，請參閱 [AutoOpenAttribute](#)。

RequireQualifiedAccess 屬性

某些模組、記錄或等位類型可能會指定 `RequireQualifiedAccess` 屬性。當您參考這些模組、記錄或等位的專案時，您必須使用限定名稱，不論是否包含匯入宣告。如果您策略性地在定義常用名稱的類型上使用此屬性，則有助於避免名稱衝突，進而使程式碼在程式庫中的變更時更具彈性。如需詳細資訊，請參閱 [RequireQualifiedAccessAttribute](#)。

另請參閱

- [F # 語言參考](#)
- [命名空間](#)
- [單元](#)

簽章

2019/10/23 • [Edit Online](#)

簽章檔案包含一組 F# 程式項目的公開金鑰相關資訊，例如類型、命名空間和模組。它可用來指定這些程式項目的協助工具。

備註

每個 F# 程式碼檔案都可以有一個 *簽章檔案*，它是和程式碼檔案同名的檔案，但副檔名是 .fsi，不是 .fs。如果直接使用命令列，也可以在命令列編譯中加入簽章檔案。為區別程式碼檔案和簽章檔案，程式碼檔案有時稱為 *實作檔案*。在專案中，簽章檔案應該位在相關聯的程式碼檔案之前。

簽章檔案描述對應實作檔案中的命名空間、模組、類型和成員。您使用簽章檔案中的資訊，指定在對應的實作檔案中，哪些程式碼部分可以從實作檔案外部的程式碼存取，哪些部分從實作檔案內部的程式碼存取。簽章檔案包含的命名空間、模組和類型，必須是實作檔案所包含之命名空間、模組和類型的子集。加上本主題稍後列出的某些例外狀況，簽章檔案中沒有列出的那些語言項目會被實作檔案視為私用的。如果專案或命令列中找不到簽章檔，會使用預設的協助工具。

如需預設協助工具的詳細資訊，請參閱[存取控制](#)。

在簽章檔案中，不用重複類型定義以及每個方法或函式的實作。而要使用每個方法和函式的簽章，做為模組或命名空間片段實作功能的完整規格。類型簽章的語法與介面和抽象類別的抽象方法宣告中使用的語法一樣，當它正確顯示編譯的輸入時，也是由 IntelliSense 和 F# 解譯器 fsi.exe 顯示。

如果類型簽章的資訊不足以指出類型是否已密封，或是否為介面類型，您就必須在編譯器中加入表示類型本質的屬性。下表說明這種用途的屬性。

“	“
<code>[<Sealed>]</code>	用於沒有任何抽象成員的類型，或不應該擴充的類型。
<code>[<Interface>]</code>	用於介面類型。

如果實作檔案之簽章和宣告間的屬性不一致，編譯器就會發生錯誤。

使用關鍵字 `val` 建立值或函式值的簽章。關鍵字 `type` 引進類型簽章。

您可以使用 `--sig` 編譯器選項產生簽章檔案。一般而言，不用手動撰寫 .fsi 檔案。而是使用編譯器產生 .fsi 檔案，將它們加入您的專案 (如有)，然後移除您不想存取的方法和函式來編輯這些檔案。

類型簽章有數個規則：

- 實作檔案中的類型縮寫絕不能符合簽章檔案中沒有縮寫的類型。
- 記錄和差別聯集必須全部公開或不公開欄位和建構函式，而簽章中的順序必須符合實作檔案中的順序。類別會顯示簽章的部分、全部欄位和方法，或完全不顯示。
- 具有建構函式的類別和結構必須公開其基底類別的宣告 (`inherits` 宣告)。此外，具有建構函式的類別和結構必須公開所有的抽象方法和介面宣告。
- 介面類型必須顯示所有的方法和介面。

值簽章的規則如下：

- 協助工具修飾詞 (`public`、`internal` 等等) 和簽章中的 `inline` 及 `mutable` 修飾詞必須符合實作中的修飾詞。
- 泛型類型參數 (無論是推斷隱含或明確宣告) 的數目必須相符, 而且泛型類型參數中的類型和類型條件約束必須相符。
- 如果使用了 `Literal` 屬性, 它必須同時出現在簽章和實作中, 而且兩者都必須使用相同的常值。
- 簽章及實作的參數模式 (也稱為 *Arity*) 必須一致。
- 如果簽章檔案中的參數名稱與對應的執行檔案不同, 則會改為使用簽章檔案中的名稱, 這可能會在進行偵錯工具或分析時造成問題。如果您想要收到這類不符的通知, 請在您的專案檔中或叫用編譯器時啟用 `--warnon` 警告 3218 (請參閱[編譯器選項](#)底下)。

下列程式碼範例顯示的簽章檔案範例, 具有命名空間、模組、函式值和有適當屬性的類型簽章。它也顯示對應的實作檔案。

```
// Module1.fsi

namespace Library1
module Module1 =
    val function1 : int -> int
    type Type1 =
        new : unit -> Type1
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Sealed>]
    type Type2 =
        new : unit -> Type2
        member method1 : unit -> unit
        member method2 : unit -> unit

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

下列程式碼顯示實作檔案。

```
namespace Library1

module Module1 =

    let function1 x = x + 1

    type Type1() =
        member type1.method1() =
            printfn "type1.method1"
        member type1.method2() =
            printfn "type1.method2"

    [<Sealed>]
    type Type2() =
        member type2.method1() =
            printfn "type2.method1"
        member type2.method2() =
            printfn "type2.method2"

    [<Interface>]
    type InterfaceType1 =
        abstract member method1 : int -> int
        abstract member method2 : string -> unit
```

另請參閱

- [F# 語言參考](#)
- [存取控制](#)
- [編譯器選項](#)

測量單位

2020/11/2 • [Edit Online](#)

F# 中的浮點數和帶正負號的整數值可以有相關聯的測量單位，通常用來表示長度、數量、品質等等。藉由使用具有單位的數量，您可以讓編譯器確認算術關聯性具有正確的單位，這有助於防止程式設計錯誤。

語法

```
[<Measure>] type unit-name [ = measure ]
```

備註

先前的語法會將 *單位名稱* 定義為量值的單位。選擇性元件是用來根據先前定義的單位來定義新的量值。例如，下列程式碼定義 `cm` (釐米) 的量值。

```
[<Measure>] type cm
```

下列程式碼定義 (milliliter) 的量值，`ml` () 的三釐米 `cm^3` 。

```
[<Measure>] type ml = cm^3
```

在先前的語法中，*measure* 是包含單位的公式。在牽涉到單元的公式中，(正面和負面) 支援整數次幂，單位之間的空格表示兩個單位的產品，`*` 也指出單位的乘積，並 `/` 指出單位的商。若為交互單位，您可以使用負整數的乘幂或 `/`，表示單位公式的分子和分母之間的分隔。分母中的多個單位應以括弧括住。以空格分隔的單位會被視為 `/` 分母的一部分，但是之後的任何單位 `*` 會被解釋為分子的一部分。

您可以單獨使用單元運算式中的1來表示維度數量，或與其他單位(例如，在分子中)一起使用。例如，速率的單位會寫入 `1/s`，其中 `s` 表示秒數。單元公式中不會使用括弧。您未在單元公式中指定數值轉換常數;不過，您可以個別定義具有單位的轉換常數，並在單元檢查計算中使用它們。

表示相同專案的單位公式可以用各種相等的方式撰寫。因此，編譯器會將單元公式轉換成一致的表單，這會將負幂轉換成 reciprocals、將單位群組為單一分子和分母，以及 Alphabetizes 分子和分母中的單位。

例如，單位公式 `kg m s^-2` 和 `m / s s * kg` 都轉換成 `kg m/s^2` 。

您可以使用浮點數運算式中的測量單位。使用浮點數搭配相關聯的量值，可以增加另一層級的安全性，並協助避免當您使用弱式具型別浮點數時，公式中可能發生的單元不相符錯誤。如果您撰寫使用單位的浮點運算式，運算式中的單位必須相符。

您可以使用角括弧的單位公式來標注常值，如下列範例所示。

```
1.0<cm>  
55.0<miles/hour>
```

您不會在數位和角括弧之間加上空格;不過，您可以包含常值尾碼(例如 `f`)，如下列範例所示。

```
// The f indicates single-precision floating point.  
55.0f<miles/hour>
```

這類批註會將常值的類型從其基本類型 (例如 `float`) 變更為維度類型, 例如 `float<cm>` 或, 在此案例中為 `float<miles/hour>`。的單位注釋 `<1>` 表示量維度數量, 而其類型相當於不含 `unit` 參數的基本型別。

度量單位的類型是浮點數或帶正負號的整數類型, 以及額外的單位注釋 (以方括弧表示)。因此, 當您將轉換的類型從 (的字母 `g`) 轉換為 `kg` (公斤) 時, 您會描述這些類型, 如下所示。

```
let convertg2kg (x : float<g>) = x / 1000.0<g/kg>
```

測量單位會用於編譯時間單元檢查, 但不會保存在執行時間環境中。因此, 它們不會影響效能。

測量單位可以套用至任何類型, 而不只是浮點數類型;不過, 只有浮點數類型、帶正負號的整數類資料類型和 `decimal` 類型支援維度數量。因此, 在基本類型上使用測量單位, 以及包含這些基本型別的匯總, 則只合理。

下列範例說明如何使用測量單位。

```
// Mass, grams.
[<Measure>] type g
// Mass, kilograms.
[<Measure>] type kg
// Weight, pounds.
[<Measure>] type lb

// Distance, meters.
[<Measure>] type m
// Distance, cm
[<Measure>] type cm

// Distance, inches.
[<Measure>] type inch
// Distance, feet
[<Measure>] type ft

// Time, seconds.
[<Measure>] type s

// Force, Newtons.
[<Measure>] type N = kg m / s^2

// Pressure, bar.
[<Measure>] type bar
// Pressure, Pascals
[<Measure>] type Pa = N / m^2

// Volume, milliliters.
[<Measure>] type ml
// Volume, liters.
[<Measure>] type L

// Define conversion constants.
let gramsPerKilogram : float<g kg^-1> = 1000.0<g/kg>
let cmPerMeter : float<cm/m> = 100.0<cm/m>
let cmPerInch : float<cm/inch> = 2.54<cm/inch>

let mlPerCubicCentimeter : float<ml/cm^3> = 1.0<ml/cm^3>
let mlPerLiter : float<ml/L> = 1000.0<ml/L>

// Define conversion functions.
let convertGramsToKilograms (x : float<g>) = x / gramsPerKilogram
let convertCentimetersToInches (x : float<cm>) = x / cmPerInch
```

下列程式碼範例說明如何從量值浮點數轉換為維度浮點數。您只需乘以1.0, 將維度套用至1.0。您可以將其抽象化為類似的函式 `degreesFahrenheit`。

此外，當您將維度值傳遞至預期可量值之浮點數的函式時，您必須使用運算子取消單位或轉型為 `float` `float`。在此範例中，您會將的 `1.0<degC>` 引數除以， `printf` 因為預期會有 `printf` 量維度數量。

```
[<Measure>] type degC // temperature, Celsius/Centigrade
[<Measure>] type degF // temperature, Fahrenheit

let convertCtoF ( temp : float<degC> ) = 9.0<degF> / 5.0<degC> * temp + 32.0<degF>
let convertFtoC ( temp: float<degF> ) = 5.0<degC> / 9.0<degF> * ( temp - 32.0<degF>)

// Define conversion functions from dimensionless floating point values.
let degreesFahrenheit temp = temp * 1.0<degF>
let degreesCelsius temp = temp * 1.0<degC>

printfn "Enter a temperature in degrees Fahrenheit."
let input = System.Console.ReadLine()
let parsedOk, floatValue = System.Double.TryParse(input)
if parsedOk
then
    printfn "That temperature in Celsius is %8.2f degrees C." ((convertFtoC (degreesFahrenheit
floatValue))/(1.0<degC>))
else
    printfn "Error parsing input."
```

下列範例會話顯示此程式碼的輸出和輸入。

```
Enter a temperature in degrees Fahrenheit.
90
That temperature in degrees Celsius is    32.22.
```

使用一般單位

您可以撰寫泛型函式，以便在具有相關測量單位的資料上運作。若要這麼做，您可以指定一種型別搭配泛型單位做為型別參數，如下列程式碼範例所示。

```
// Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

let genericSumUnits ( x : float<'u>) (y: float<'u>) = x + y

let v1 = 3.1<m/s>
let v2 = 2.7<m/s>
let x1 = 1.2<m>
let t1 = 1.0<s>

// OK: a function that has unit consistency checking.
let result1 = genericSumUnits v1 v2
// Error reported: mismatched units.
// Uncomment to see error.
// let result2 = genericSumUnits v1 x1
```

使用泛型單位建立匯總類型

下列程式碼示範如何建立匯總型別，其中包含具有泛型單位之個別浮點數的值。這可讓您建立可搭配各種單位使用的單一類型。此外，泛型單位也會藉由確保具有一組單位的泛型型別，與具有不同單位集合的相同泛型型別不同，來保留型別安全。這項技術的基礎是將 `Measure` 屬性套用至型別參數。

```
// Distance, meters.
[<Measure>] type m
// Time, seconds.
[<Measure>] type s

// Define a vector together with a measure type parameter.
// Note the attribute applied to the type parameter.
type vector3D[<Measure>] 'u' = { x : float<'u>; y : float<'u>; z : float<'u>}

// Create instances that have two different measures.
// Create a position vector.
let xvec : vector3D<m> = { x = 0.0<m>; y = 0.0<m>; z = 0.0<m> }
// Create a velocity vector.
let v1vec : vector3D<m/s> = { x = 1.0<m/s>; y = -1.0<m/s>; z = 0.0<m/s> }
```

執行時間的單位

測量單位用於靜態類型檢查。當您編譯浮點值時，會排除測量單位，因此在執行時間會遺失單位。因此，任何嘗試執行相依赖于在執行時間檢查單位的功能都是不可行的。例如，執行函式 `ToString` 以列印出單位是不可行的。

轉換

若要轉換具有單位 (的型別 (例如, `float<'u>`) 為沒有單位的型別)，您可以使用標準轉換函數。例如，您可以使用 `float` 來轉換成沒有 `float` 單位的值，如下列程式碼所示。

```
[<Measure>]
type cm
let length = 12.0<cm>
let x = float length
```

若要將無值的值轉換成具有單位的值，您可以乘以以適當單位標注的1或1.0 值。不過，為了撰寫互通性層，還有一些明確的函式，可讓您用來將未使用的值轉換成具有單位的值。這些都位於 [Fsharp.core languageprimitives.physicalequality](https://fsharpcore.github.io/languageprimitives.physicalequality) 模組中。例如，若要從未使用的轉換成 `float` `float<cm>`，請使用 `FloatWithMeasure`，如下列程式碼所示。

```
open Microsoft.FSharp.Core
let height:float<cm> = LanguagePrimitives.FloatWithMeasure x
```

F # 核心程式庫中的測量單位

在命名空間中有提供單位程式庫 `FSharp.Data.UnitSystems.SI`。它會在其符號形式中包含 SI 單位 (例如 `m` 子命名空間中的計量) `UnitSymbols`，以及其完整名稱 (例如 `meter` `UnitNames` 子命名空間中的計量)。

另請參閱

- [F # 語言參考](#)

純文字格式

2020/11/2 • [Edit Online](#)

F # 使用 `printf`、`printfn`、和相關函式，支援純文字的類型檢查格式 `sprintf`。例如

```
dotnet fsi

> printfn "Hello %s, %d + %d is %d" "world" 2 2 (2+2);;
```

提供輸出

```
Hello world, 2 + 2 is 4
```

F # 也允許將結構化值格式化為純文字。例如，請考慮下列範例，其會將輸出格式化為類似矩陣的元組顯示。

```
dotnet fsi

> printfn "%A" [ for i in 1 .. 5 -> [ for j in 1 .. 5 -> (i, j) ] ];;

[(1, 1); (1, 2); (1, 3); (1, 4); (1, 5)];
[(2, 1); (2, 2); (2, 3); (2, 4); (2, 5)];
[(3, 1); (3, 2); (3, 3); (3, 4); (3, 5)];
[(4, 1); (4, 2); (4, 3); (4, 4); (4, 5)];
[(5, 1); (5, 2); (5, 3); (5, 4); (5, 5)]
```

當您 `%A` 在格式化字串中使用格式時，會啟用結構化純文字格式 `printf`。它也會在 F # interactive 中格式化值的輸出時啟用，其中輸出會包含額外的資訊，而且還可自訂。純文字格式也可透過任何對 `x.ToString()` F # `union` 和記錄值的呼叫來觀察，包括在偵錯工具、記錄和其他工具中隱含發生的。

檢查 `printf` 格式字串

如果 `printf` 使用格式函數搭配的引數與格式字串中的 `printf` 格式規範不相符，就會回報編譯時期錯誤。例如

```
sprintf "Hello %s" (2+2)
```

提供輸出

```
sprintf "Hello %s" (2+2)
-----^

stdin(3,25): error FS0001: The type 'string' does not match the type 'int'
```

技術上而言，當使用 `printf` 和其他相關函式時，F # 編譯器中的特殊規則會檢查當做格式字串傳遞的字串常值，確保後續套用的引數是正確的類型，以符合所使用的格式規範。

的格式規範 `printf`

格式規格 `printf` 是具有 `%` 標記的字串，表示格式。格式預留位置是由類型的解讀方式所組成

`%[flags][width][.precision][type]`，如下所示：

格式字元	資料類型	說明
<code>%b</code>	bool	格式化為 <code>true</code> 或 <code>false</code>
<code>%s</code>	字串	格式化為其非的非格式內容
<code>%c</code>	char	格式化為字元常值
<code>%d</code> , <code>%i</code>	基本整數類型	格式化為十進位整數, 如果基本整數類型已簽署, 則為帶正負號
<code>%u</code>	基本整數類型	格式化為不帶正負號的十進位整數
<code>%x</code> , <code>%X</code>	基本整數類型	已格式化為不帶正負號的十六進位數位 (-f 或 -F 分別用於十六進位數位)
<code>%o</code>	基本整數類型	格式化為不帶正負號的八進位數位
<code>%e</code> , <code>%E</code>	基本浮點類型	格式化為具有格式的帶正負號值 <code>[-]d.dddde[sign]ddd</code> , 其中 d 是單一十進位數, dddd 是一或多個十進位數, ddd 只是三個小數位數, 而 sign 是 <code>+</code> 或 <code>-</code>
<code>%f</code>	基本浮點類型	格式化為具有格式的帶正負號值 <code>[-]dddd.dddd</code> , 其中 <code>dddd</code> 是一或多個十進位數。小數點前面的位數取決於數字的大小, 小數點後面的位數則取決於要求的精確度。
<code>%g</code> , <code>%G</code>	基本浮點類型	使用當做以或格式列印的帶正負號值進行格式化, 以較 <code>%f</code> <code>%e</code> 精簡的指定值和有效位數為準。
<code>%M</code>	<code>System.Decimal</code> 值	使用的 <code>"G"</code> 格式標準格式化 <code>System.Decimal.ToString(format)</code>
<code>%O</code>	任何值	藉由將物件裝箱並呼叫其方法來進行格式化 <code>System.Object.ToString()</code>
<code>%A</code>	任何值	使用預設版面配置設定, 以結構化純文字格式格式化
<code>%a</code>	任何值	需要兩個引數: 格式函數接受內容參數和值, 以及要列印的特定值
<code>%t</code>	任何值	需要一個引數: 格式化函數接受內容參數, 以輸出或傳回適當的文字
<code>%%</code>	(無)	不需要任何引數, 而且會列印純量的字元: <code>%</code>

基本整數類型 `byte` (`System.Byte`)、 `sbyte` (`System.SByte`)、 `int16` (`System.Int16`)、 `uint16` (`System.UInt16`)、 `int32` (`System.Int32`)、 `uint32` (`System.UInt32`)、 `int64` (`System.Int64`)、 `uint64` (`System.UInt64`)、 `nativeint` (`System.IntPtr`) 和 `unativeint` `System.UIntPtr` 。**基本浮點類型** `float` (

`System.Double`) 和 `float32` (`System.Single`) 。

選擇性寬度是一個整數，表示結果的最小寬度。例如，`%6d` 會列印一個整數，並在前面加上空格，以填滿至少六個字元。如果 width 為 `*`，則會採用額外的整數引數來指定對應的寬度。

有效的旗標如下：

旗標	說明	說明
<code>0</code>	新增零(而不是空格)來組成所需的寬度	
<code>-</code>	靠左對齊指定寬度內的結果	
<code>+</code>	<code>+</code> 如果數位是正數 (以符合否定的正負號，請新增一個字元 <code>-</code>)	
空白字元	如果數位是正數 (以符合否定的 '-' 正負號，請新增額外的空間)	

`Printf` `#` 旗標無效，如果使用，將會回報編譯時期錯誤。

值是不因文化特性而異的格式。文化特性設定與格式無關，`printf` 不同之處在於它們會影響 `%O` 和格式化的結果 `%A` 。如需詳細資訊，請參閱[結構化純文字格式](#)。

%A 樣式

`%A` 格式規範是用來以人類看得懂的方式來格式化值，也適用於報告診斷資訊。

基本值

使用規範來格式化純文字時 `%A`，F# 數值會使用其後綴和不因文化特性而異。使用浮點精確度的10個位置來格式化浮點值。例如

```
printfn "%A" (1L, 3n, 5u, 7, 4.03f, 5.000000001, 5.000000001)
```

塊

```
(1L, 3n, 5u, 7, 4.03000021f, 5.000000001, 5.0)
```

使用規範時 `%A`，字串會使用引號來格式化。不會新增轉義碼，而是會列印原始字元。例如

```
printfn "%A" ("abc", "a\tb\nc\d")
```

塊

```
("abc", "a\tb\nc\d")
```

.NET 值

使用規範來格式化純文字時 `%A`，非 F# .net 物件是使用 `x.ToString()` 和所指定的 .net 預設設定來格式化 `System.Globalization.CultureInfo.CurrentCulture` `System.Globalization.CultureInfo.CurrentUICulture` 。例如

```
open System.Globalization

let date = System.DateTime(1999, 12, 31)

CultureInfo.CurrentCulture <- CultureInfo.GetCultureInfo("de-DE")
printfn "Culture 1: %A" date

CultureInfo.CurrentCulture <- CultureInfo.GetCultureInfo("en-US")
printfn "Culture 2: %A" date
```

塊

```
Culture 1: 31.12.1999 00:00:00
Culture 2: 12/31/1999 12:00:00 AM
```

結構化值

使用規範來格式化純文字時 `%A`，區塊縮排會用於 F# 清單和元組。如先前範例所示。陣列的結構也會使用，包括多維度陣列。一維陣列會以語法顯示 `[| ... |]`。例如

```
printfn "%A" [| for i in 1 .. 20 -> (i, i*i) |]
```

塊

```
[|(1, 1); (2, 4); (3, 9); (4, 16); (5, 25); (6, 36); (7, 49); (8, 64); (9, 81);
(10, 100); (11, 121); (12, 144); (13, 169); (14, 196); (15, 225); (16, 256);
(17, 289); (18, 324); (19, 361); (20, 400)|]
```

預設列印寬度為80。您可以使用格式規範中的列印寬度來自訂此寬度。例如

```
printfn "%10A" [| for i in 1 .. 5 -> (i, i*i) |]

printfn "%20A" [| for i in 1 .. 5 -> (i, i*i) |]

printfn "%50A" [| for i in 1 .. 5 -> (i, i*i) |]
```

塊

```
[|(1, 1);
(2, 4);
(3, 9);
(4, 16);
(5, 25)|]
[|(1, 1); (2, 4);
(3, 9); (4, 16);
(5, 25)|]
[|(1, 1); (2, 4); (3, 9); (4, 16); (5, 25)|]
```

將列印寬度指定為0會導致不使用列印寬度。除了輸出中的內嵌字串包含分行符號以外，會產生一行文字。例如：

```
printfn "%0A" [| for i in 1 .. 5 -> (i, i*i) |]

printfn "%0A" [| for i in 1 .. 5 -> "abc\ndef" |]
```

塊

```
[|(1, 1); (2, 4); (3, 9); (4, 16); (5, 25)|]
[|"abc
def"; "abc
def"; "abc
def"; "abc
def"|]
```

深度限制4用於順序 (`IEnumerable`) 值，這會顯示為 `seq { ... }` 。[深度限制] 100 用於清單和陣列值。例如

```
printfn "%A" (seq { for i in 1 .. 10 -> (i, i*i) })
```

塊

```
seq [(1, 1); (2, 4); (3, 9); (4, 16); ...]
```

區塊縮排也會用於公用記錄和聯集值的結構。例如

```
type R = { X : int list; Y : string list }

printfn "%A" { X = [ 1;2;3 ]; Y = ["one"; "two"; "three"] }
```

塊

```
{ X = [1; 2; 3]
  Y = ["one"; "two"; "three"] }
```

如果 `%+A` 使用了，則也會使用反映來顯示記錄和等位的私用結構。例如：

```
type internal R =
    { X : int list; Y : string list }
    override _.ToString() = "R"

let internal data = { X = [ 1;2;3 ]; Y = ["one"; "two"; "three"] }

printfn "external view:\n%A" data

printfn "internal view:\n%+A" data
```

塊

```
external view:
R

internal view:
{ X = [1; 2; 3]
  Y = ["one"; "two"; "three"] }
```

大型、迴圈或深層的嵌套值

大型結構化的值會格式化為最大整體物件節點計數10000。深層的嵌套值會格式化為深度100。在這兩種情況下 `...`，都是用來省略部分輸出。例如

```

type Tree =
  | Tip
  | Node of Tree * Tree

let rec make n =
  if n = 0 then
    Tip
  else
    Node(Tip, make (n-1))

printfn "%A" (make 1000)

```

會產生具有部分省略的大型輸出：

```
Node(Tip, Node(Tip, ....Node (... , ...)...))
```

迴圈會在物件圖形中偵測到，並在偵測 `...` 到迴圈的位置使用。例如：

```

type R = { mutable Links: R list }
let r = { Links = [] }
r.Links <- [r]
printfn "%A" r

```

塊

```
{ Links = [...] }
```

Lazy、null 和 function 值

當尚未評估值時，延遲值會列印成 `Value is not created` 或對等的文字。

Null 值會列印成，`null` 除非將值的靜態類型判斷為 `null` 允許標記法的聯集類型。

F # 函式值會列印為其內部產生的關閉名稱，例如 `<fun:it@43-7>`。

使用自訂純文字格式 `StructuredFormatDisplay`

使用規範時 `%A`，`StructuredFormatDisplay` 會遵守類型宣告上的屬性是否存在。這可用來指定用來顯示值的代理文字和屬性。例如：

```

[<StructuredFormatDisplay("Counts({Clicks})")>]
type Counts = { Clicks:int list}

printfn "%20A" {Clicks=[0..20]}

```

塊

```

Counts([0; 1; 2; 3;
        4; 5; 6; 7;
        8; 9; 10; 11;
        12; 13; 14;
        15; 16; 17;
        18; 19; 20])

```

藉由覆寫自訂純文字格式 `ToString`

的預設執行 `ToString` 在 F # 程式設計中是可觀察的。通常，預設結果不適合在程式設計人員面向的資訊顯示或

使用者輸出中使用，因此，通常會覆寫預設的執行。

根據預設，F # 記錄和聯集類型會以使用的執行來覆寫的執行 `ToString` `sprintf "%+A"`。例如

```
type Counts = { Clicks:int list }

printfn "%s" ({Clicks=[0..10]}.ToString())
```

塊

```
{ Clicks = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10] }
```

針對類別類型，不會提供的預設實作為， `ToString` 而且會使用 .net 預設值，這會報告類型的名稱。例如

```
type MyClassType(clicks: int list) =
    member _.Clicks = clicks

let data = [ MyClassType([1..5]); MyClassType([1..5]) ]
printfn "Default structured print gives this:\n%A" data
printfn "Default ToString gives:\n%s" (data.ToString())
```

塊

```
Default structured print gives this:
[MyClassType; MyClassType]
Default ToString gives:
[MyClassType; MyClassType]
```

新增的覆寫 `ToString` 可以提供更好的格式。

```
type MyClassType(clicks: int list) =
    member _.Clicks = clicks
    override _.ToString() = sprintf "MyClassType(%0A)" clicks

let data = [ MyClassType([1..5]); MyClassType([1..5]) ]
printfn "Now structured print gives this:\n%A" data
printfn "Now ToString gives:\n%s" (data.ToString())
```

塊

```
Now structured print gives this:
[MyClassType([1; 2; 3; 4; 5]); MyClassType([1; 2; 3; 4; 5])]
Now ToString gives:
[MyClassType([1; 2; 3; 4; 5]); MyClassType([1; 2; 3; 4; 5])]
```

F # 互動式結構化列印

F # Interactive (`dotnet fsi`) 會使用延伸版本的結構化純文字格式來報告值，並允許其他自訂能力。如需詳細資訊，請參閱 [F # Interactive](#)。

自訂 debug 顯示

適用於 .NET 的偵錯工具會使用和之類的屬性 `DebuggerDisplay` `DebuggerTypeProxy`，而且這些屬性會影響偵錯工具檢查視窗中物件的結構化顯示。F # 編譯器會針對區分聯集和記錄類型自動產生這些屬性，但不會針對類別、

介面或結構類型來產生。

F # 純文字格式會忽略這些屬性，但在進行 F # 型別的調試時，執行這些方法以改善顯示會很有用。

另請參閱

- [字串](#)
- [記錄](#)
- [已區分的聯集](#)
- [F# 互動](#)

使用 XML 批註記錄您的程式碼

2021/3/5 • [Edit Online](#)

您可以從三斜線產生檔 (//) F# 中的程式碼批註。XML 批註可以在程式碼檔中的宣告之前 (.fs) 或簽章 (.fsi) 檔。

XML 文件註解是一種特殊類型的註解，新增於任何使用者定義型別或成員的定義上方。XML 文件註解具特殊性，因為編譯器可以處理它們以在編譯時期產生 XML 文件檔案。編譯器產生的 XML 檔案可以與您的 .NET 元件一起散發，讓 Ide 可以使用工具提示來顯示類型或成員的快速資訊。此外，XML 檔案可以透過 [fsdocs](#) 之類的工具來執行，以產生 API 參考網站。

XML 檔批註（如同所有其他批註）將會被編譯器忽略，除非已啟用以下所述的選項，以便在編譯時期檢查批註的有效性和完整性。

您可以執行下列其中一項動作，以在編譯時期產生 XML 檔案：

- 您可以在 `GenerateDocumentationFile` 專案檔的 `<PropertyGroup>` 區段中加入 `.fsproj` 專案，這會在專案目錄中產生 XML 檔案，該檔案的根檔案名與元件相同。例如：

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
```

- 如果您正在使用 Visual Studio 開發應用程式，請以滑鼠右鍵按一下專案，然後選取 [屬性]。在屬性對話方塊中，選取 [建置] 索引標籤，然後檢查 [XML 文件檔案]。您也可以變更編譯器寫入檔案的位置。

有兩種方式可以撰寫 XML 檔批註：with 和不含 XML 標記。兩者都使用三斜線批註。

沒有 XML 標記的批註

如果 `///` 批註的開頭不是，則 `<` 會將整個註解文字視為程式碼結構的摘要檔（緊接在後面）。當您只想撰寫每個結構的簡短摘要時，請使用這個方法。

批註會在檔準備期間編碼為 XML，因此 `<` `>` 不需要將字元（如、和）編碼 `&`。如果您未明確指定摘要標記，則不應指定其他標記，例如 `param` 或傳回標記。

下列範例顯示不含 XML 標記的替代方法。在此範例中，批註中的整個文字都會被視為摘要。

```
/// Creates a new string whose characters are the result of applying
/// the function mapping to each of the characters of the input string
/// and concatenating the resulting strings.
val collect : (char -> string) -> string -> string
```

具有 XML 標記的批註

如果批註主體的開頭是 `<`（一般 `<summary>`），則會使用 xml 標記將它視為 xml 格式的批註主體。第二個可讓您指定簡短摘要的個別附注、其他備註、每個參數的檔、類型參數和所擲回的例外狀況，以及傳回值的描述。

以下是簽章檔案中的一般 XML 檔批註：


```
/// <summary>Builds a new string whose characters are the results of applying the function <c>mapping</c>
/// to each of the characters of the input string and concatenating the resulting
/// strings.</summary>
/// <param name="mapping">The function to produce a string from each character of the input string.</param>
///<param name="str">The input string.</param>
///<returns>The concatenated string.</returns>
///<exception cref="System.ArgumentNullException">Thrown when the input string is null.</exception>
val collect : (char -> string) -> string -> string
```

建議的標記

如果您使用 XML 標記，下表描述 F # XML 程式碼批註中所識別的外部標記。

標記	說明
<code><summary></code>  <code></summary></code>	指定 文字 是程式元素的簡短描述。描述通常是一或兩個句子。
<code><remarks></code>  <code></remarks></code>	指定 文字 包含程式元素的補充資訊。
<code><param name="</code>  <code>"></code>  <code></param></code>	指定函數或方法參數的名稱和描述。
<code><typeparam name="</code>  <code>"></code>  <code></typeparam></code>	指定型別參數的名稱和描述。
<code><returns></code>  <code></returns></code>	指定 文字 描述函數或方法的傳回值。
<code><exception cref="</code>  <code>"></code>  <code></exception></code>	指定可以產生的例外狀況類型，以及擲回它的情況。
<code><seealso cref="</code>  <code>"</code>  <code></></code>	指定另一種類型之檔的 [另一個] 連結。參考是 XML 檔檔案中顯示的名稱。另請參閱檔頁面底部的連結。

下表描述在 [描述] 區段內使用的標記：

標記	說明
<code><para></code>  <code></para></code>	指定文欄位落。這是用來分隔  標記內的文字。
<code><code></code>  <code></code></code>	指定 文字 是多行程式碼。檔產生器可以使用這個標記，以適用於程式碼的字型來顯示文字。
<code><paramref name="</code>  <code></></code>	指定相同檔批註中的參數參考。
<code><typeparamref name="</code>  <code></></code>	在相同的檔批註中指定類型參數的參考。
<code><c></code>  <code></c></code>	指定 文字 為內嵌程式碼。檔產生器可以使用這個標記，以適用於程式碼的字型來顯示文字。
<code><see cref="</code>  <code>"></code>  <code></see></code>	指定另一個程式元素的內嵌連結。參考是 XML 檔檔案中顯示的名稱。文字 是連結中所顯示的文字。

使用者定義標記

先前的標記代表 F # 編譯器和一般 F # 編輯器工具所能辨識的標記。不過，使用者可以定義自己的標記。Fsdocs 之類的工具可為您提供額外標記的支援 `<namespace doc>`。自訂或內部文件產生工具也可以與標準標記搭配使

用，而且可以支援從 HTML 到 PDF 的多個輸出格式。

編譯時間檢查

當 `--warnon:3390` 啟用時，編譯器會驗證 XML 的語法和和標記中參考的參數 `<param>` `<paramref>` 。

記載 F # 結構

F # 結構 (例如模組、成員、聯集案例和記錄欄位) 會在 `///` 其宣告之前立即以批註記載。如有需要，會在 `///` 引數清單之前提供批註，以記載類別的隱含函式。例如：

```
/// This is the type
type SomeType
    /// This is the implicit constructor
    (a: int, b: int) =

    /// This is the member
    member _.Sum() = a + b
```

限制

F # 不支援 C # 和其他 .NET 語言中的 XML 檔的某些功能。

- 例如，在 F # 中，交互參考必須使用對應符號的完整 XML 簽章 `cref="T:System.Console"`。簡單的 C # 樣式交互參考 (例如) `cref="Console"` 不會針對完整的 XML 簽章進行詳細檢查，而且 F # 編譯器不會檢查這些元素。某些檔工具可能會允許後續處理使用這些交叉參考，但是應該使用完整簽章。
- `<include>` `<inheritdoc>` F # 編譯器不支援這些標記。如果使用它們，則不會提供任何錯誤，但會直接複製到產生的檔檔，而不會影響所產生的檔。
- F # 編譯器不會檢查交互參考，即使使用時也一樣 `-warnon:3390` 。
- `<typeparam>` `<typeparamref>` F # 編譯器不會檢查標記中使用的名稱，即使使用時也不會檢查 `--warnon:3390` 。
- 如果遺漏檔，則不會提供任何警告，即使使用時也不會出現 `--warnon:3390` 。

建議

有許多原因，建議您記錄程式碼。以下是一些最佳作法、一般使用案例，以及在 F # 程式碼中使用 XML 檔標記時應該知道的事項。

- `--warnon:3390` 在您的程式碼中啟用選項，以協助確保您的 xml 檔是有效的 xml。
- 請考慮新增簽章檔案，以便將大型 XML 檔批註與您的實作分開。
- 為保持一致性，應該記錄所有公開可見的類型和其成員。如果您必須執行它，則請執行。
- 最小的模組、類型和其成員應具有純文字 `///` 批註或 `<summary>` 標記。這會顯示在 F # 編輯工具的自動完成工具提示視窗中。
- 應該使用結尾為句號的完整句子來撰寫文件文字。

另請參閱

- [C # XML 檔批註 \(C# 程式設計指南\)](#)。
- [F # 語言參考](#)

- 編譯器選項

延遲運算式

2021/3/5 • [Edit Online](#)

延遲運算式 是不會立即評估的運算式，而是在需要結果時進行評估。這有助於提升程式碼的效能。

語法

```
let identifier = lazy ( expression )
```

備註

在先前的語法中，*expression* 是只有在需要結果時才會評估的程式碼，而 *identifier* 是儲存結果的值。值的型別為 `Lazy<'T>`，其中使用的實際型別取決於 `'T` 運算式的結果。

延遲運算式可讓您藉由將運算式的執行限制在需要結果的情況下，來改善效能。

若要強制執行運算式，您可以呼叫方法 `Force`。`Force` 只會執行一次執行。後續呼叫 `Force` 會傳回相同的結果，但不會執行任何程式碼。

下列程式碼說明如何使用 lazy 運算式和使用 `Force`。在此程式碼中，的型別 `result` 為 `Lazy<int>`，而且方法會傳回 `Force` `int`。

```
let x = 10
let result = lazy (x + 10)
printfn "%d" (result.Force())
```

延遲評估(但不 `Lazy` 是類型)也用於序列。如需詳細資訊，請參閱 [序列](#)。

另請參閱

- [F # 語言參考](#)
- [LazyExtensions 模組](#)

計算運算式

2021/3/5 • [Edit Online](#)

F# 中的計算運算式提供一個方便的語法，可用來撰寫可使用控制流程結構和系結來排序和結合的計算。視計算運算式的種類而定，您可以將它們視為表示 monads、monoids、monad 轉換器和 applicative 函子的方式。不過，不同于其他語言 (例如 Haskell) 中的標記法，它們並不會系結至單一抽象概念，也不會依賴宏或其他形式的元程式來完成方便且上下文相關的語法。

概觀

計算可以採用許多形式。最常見的計算形式是單一執行緒執行，這很容易理解和修改。不過，並非所有形式的計算都像單一執行緒執行一樣簡單。部分範例包括：

- 不具決定性的計算
- 非同步計算
- Effectful 計算
- 有生產力計算

一般來說，您必須在應用程式的某些部分中執行 *內容相關* 的計算。撰寫內容相關的程式碼可能是一項挑戰，因為在沒有抽象的情況下，可以輕易地在指定的內容之外「流失」計算，以防止您這樣做。這些抽象概念通常很難自行撰寫，這也是為什麼 F# 有一般化的方法，稱為 **計算運算式**。

計算運算式提供統一的語法和抽象模型，用於編碼內容相關計算。

每個計算運算式 *都是由產生器型別* 所支援。產生器類型會定義可供計算運算式使用的作業。請參閱 [建立新的計算運算式類型](#)，其會顯示如何建立自訂計算運算式。

語法概觀

所有計算運算式的格式如下：

```
builder-expr { cexper }
```

其中 `builder-expr` 是定義計算運算式之產生器類型的名稱，而 `cexper` 則是計算運算式的運算式主體。例如，`async` 計算運算式程式碼看起來會像這樣：

```
let fetchAndDownload url =  
    async {  
        let! data = downloadData url  
  
        let processedData = processData data  
  
        return processedData  
    }
```

在計算運算式中有特殊的額外語法可用，如先前的範例所示。下列運算式形式可以搭配計算運算式使用：

```
expr { let! ... }
expr { do! ... }
expr { yield ... }
expr { yield! ... }
expr { return ... }
expr { return! ... }
expr { match! ... }
```

這些關鍵字和其他標準 F# 關鍵字只有在支援產生器型別中已定義時，才可在計算運算式中使用。唯一的例外是 `match!`，它本身就是可供使用的語法，`let!` 後面接著結果的模式比對。

產生器型別是一個物件，它會定義特殊方法來管理計算運算式片段的組合方式;也就是說，其方法會控制計算運算式的行為。另一種描述 builder 類別的方式，就是假設它可讓您自訂許多 F# 結構的作業，例如迴圈和系結。

`let!`

關鍵字會將呼叫的結果系結至 `let!` 另一個計算運算式的名稱：

```
let doThingsAsync url =
    async {
        let! data = getDataAsync url
        ...
    }
```

如果您使用來系結計算運算式的呼叫 `let`，將不會取得計算運算式的結果。相反地，您會將未產生的呼叫值系結 至該計算 運算式。使用系結 `let!` 至結果。

`let!` 由產生器 `Bind(x, f)` 類型上的成員定義。

`do!`

`do!` 關鍵字是用來呼叫計算運算式，該運算式會傳回類似型別的 `unit` (由產生器 `Zero`) 的成員定義：

```
let doThingsAsync data url =
    async {
        do! submitData data url
        ...
    }
```

針對 [非同步工作流程](#)，此類型為 `Async<unit>`。針對其他計算運算式，類型可能是 `CExpType<unit>`。

`do!` 是由產生者 `Bind(x, f)` 類型上的成員所定義 `f unit`。

`yield`

`yield` 關鍵字是用來從計算運算式傳回值，讓它可以作為 [IEnumerable<T>](#)：

```
let squares =
    seq {
        for i in 1..10 do
            yield i * i
    }

for sq in squares do
    printfn $"{sq}"
```

在大部分的情況下，呼叫端可以省略它。最常見的省略方法 `yield` 是使用 `->` 運算子：

```
let squares =
    seq {
        for i in 1..10 -> i * i
    }

for sq in squares do
    printfn $"{sq}"
```

針對可能會產生許多不同值的更複雜運算式，而且可能有條件地省略關鍵字可以執行下列作業：

```
let weekdays includeWeekend =
    seq {
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    }
```

如同 `c#` 中的 `yield` 關鍵字，計算運算式中的每個專案都會在反覆運算時產生回來。

`yield` 由 `Yield(x)` 產生器型別的成員定義，其中 `x` 是要傳回的專案。

`yield!`

`yield!` 關鍵字是用來壓平合併計算運算式中的值集合：

```
let squares =
    seq {
        for i in 1..3 -> i * i
    }

let cubes =
    seq {
        for i in 1..3 -> i * i * i
    }

let squaresAndCubes =
    seq {
        yield! squares
        yield! cubes
    }

printfn $"{squaresAndCubes}" // Prints - 1; 4; 9; 1; 8; 27
```

進行評估時，所呼叫的計算運算式 `yield!` 會將其專案逐一產生，並將結果壓平合併。

`yield!` 由產生器 `YieldFrom(x)` 型別上的成員定義，其中 `x` 是值的集合。

不同于 `yield`，`yield!` 必須明確指定。其行為在計算運算式中不是隱含的。

`return`

`return` 關鍵字會在對應至計算運算式的型別中包裝一個值。除了使用的計算運算式之外 `yield`，它也用來「完成」計算運算式：

```
let req = // 'req' is of type 'Async<data>'
    async {
        let! data = fetch url
        return data
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return` 是由產生器 `Return(x)` 型別上的成員所定義，其中 `x` 是要換行的專案。

`return!`

`return!` 關鍵字會知道計算運算式的值，並將結果包裝為對應于計算運算式的型別：

```
let req = // 'req' is of type 'Async<data>'
    async {
        return! fetch url
    }

// 'result' is of type 'data'
let result = Async.RunSynchronously req
```

`return!` 由產生器 `ReturnFrom(x)` 型別上的成員定義，其中 `x` 是另一個計算運算式。

`match!`

`match!` 關鍵字可讓您內嵌對另一個計算運算式的呼叫，以及其結果的模式比對：

```
let doThingsAsync url =
    async {
        match! callService url with
        | Some data -> ...
        | None -> ...
    }
```

使用呼叫計算運算式時 `match!`，它會瞭解呼叫的結果(例如) `let!`。這通常用於呼叫計算運算式，其中的結果是 [選擇性](#)的。

內建計算運算式

F# 核心程式庫會定義三個內建的計算運算式：[順序運算式](#)、[非同步工作流程](#)和 [查詢運算式](#)。

建立新的計算運算式類型

您可以藉由建立產生器類別，並在類別上定義某些特殊方法，來定義您自己的計算運算式的特性。Builder 類別可以選擇性地定義下表所列的方法。

下表說明可在工作流程產生器類別中使用的方法。

⌞	⌞ (S)	⌞
<code>Bind</code>	<code>M<'T> * ('T -> M<'U>) -> M<'U></code>	<code>let!</code> <code>do!</code> 在計算運算式中呼叫和。
<code>Delay</code>	<code>(unit -> M<'T>) -> M<'T></code>	將計算運算式包裝為函數。
<code>Return</code>	<code>'T -> M<'T></code>	<code>return</code> 在計算運算式中呼叫。

⌈	⌈ (S)	⌈
<code>ReturnFrom</code>	<code>M<'T> -> M<'T></code>	<code>return!</code> 在計算運算式中呼叫。
<code>Run</code>	<code>M<'T> -> M<'T></code> 或 <code>M<'T> -> 'T</code>	執行計算運算式。
<code>Combine</code>	<code>M<'T> * M<'T> -> M<'T></code> 或 <code>M<unit> * M<'T> -> M<'T></code>	在計算運算式中呼叫以進行排序。
<code>For</code>	<code>seq<'T> * ('T -> M<'U>) -> M<'U></code> 或 <code>seq<'T> * ('T -> M<'U>) -> seq<M<'U>></code>	針對 <code>for...do</code> 計算運算式中的運算式呼叫。
<code>TryFinally</code>	<code>M<'T> * (unit -> unit) -> M<'T></code>	針對 <code>try...finally</code> 計算運算式中的運算式呼叫。
<code>TryWith</code>	<code>M<'T> * (exn -> M<'T>) -> M<'T></code>	針對 <code>try...with</code> 計算運算式中的運算式呼叫。
<code>Using</code>	<code>'T * ('T -> M<'U>) -> M<'U> when 'T :> IDisposable</code>	在計算運算式中呼叫以進行系統 <code>use</code> 。
<code>While</code>	<code>(unit -> bool) * M<'T> -> M<'T></code>	針對 <code>while...do</code> 計算運算式中的運算式呼叫。
<code>Yield</code>	<code>'T -> M<'T></code>	針對 <code>yield</code> 計算運算式中的運算式呼叫。
<code>YieldFrom</code>	<code>M<'T> -> M<'T></code>	針對 <code>yield!</code> 計算運算式中的運算式呼叫。
<code>Zero</code>	<code>unit -> M<'T></code>	<code>else</code> <code>if...then</code> 在計算運算式中針對運算式的空分支呼叫。
<code>Quote</code>	<code>Quotations.Expr<'T> -> Quotations.Expr<'T></code>	指出計算運算式以 <code>Run</code> 引號形式傳遞至成員。它會將計算的所有實例轉譯成引號。

產生器類別中的許多方法都會使用並傳回 `M<'T>` 結構，而這通常是個別定義型別，可區分要合併的計算類型，例如，`Async<'T>` 針對非同步工作流程和 `Seq<'T>` 序列工作流程。這些方法的簽章可讓它們彼此合併和互相合併，以便將從一個結構傳回的工作流程物件傳遞給下一個結構。編譯器在剖析計算運算式時，會使用上表中的方法和計算運算式中的程式碼，將運算式轉換成一連串的嵌套函式呼叫。

嵌套運算式的格式如下：

```
builder.Run(builder.Delay(fun () -> {| cexpr |}))
```

在上述程式碼中，`Run` `Delay` 如果未在計算運算式產生器類別中定義，則會省略和的呼叫。計算運算式的主體（在此表示為 `{| cexpr |}`）會轉譯成包含產生器類別之方法的呼叫，並遵循下表所述的翻譯。計算運算式

`{| cexpr |}` 會根據這些轉譯以遞迴方式定義，其中 `expr` 是 F# 運算式且 `cexpr` 為計算運算式。

'''	''
<code>{ let binding in cexpr }</code>	<code>let binding in { cexpr }</code>
<code>{ let! pattern = expr in cexpr }</code>	<code>builder.Bind(expr, (fun pattern -> { cexpr }))</code>
<code>{ do! expr in cexpr }</code>	<code>builder.Bind(expr, (fun () -> { cexpr }))</code>
<code>{ yield expr }</code>	<code>builder.Yield(expr)</code>
<code>{ yield! expr }</code>	<code>builder.YieldFrom(expr)</code>
<code>{ return expr }</code>	<code>builder.Return(expr)</code>
<code>{ return! expr }</code>	<code>builder.ReturnFrom(expr)</code>
<code>{ use pattern = expr in cexpr }</code>	<code>builder.Using(expr, (fun pattern -> { cexpr }))</code>
<code>{ use! value = expr in cexpr }</code>	<code>builder.Bind(expr, (fun value -> builder.Using(value, (fun value -> { cexpr }))))</code>
<code>{ if expr then cexpr0 }</code>	<code>if expr then { cexpr0 } else builder.Zero()</code>
<code>{ if expr then cexpr0 else cexpr1 }</code>	<code>if expr then { cexpr0 } else { cexpr1 }</code>
<code>{ match expr with pattern_i -> cexpr_i }</code>	<code>match expr with pattern_i -> { cexpr_i }</code>
<code>{ for pattern in expr do cexpr }</code>	<code>builder.For(enumeration, (fun pattern -> { cexpr }))</code>
<code>{ for identifier = expr1 to expr2 do cexpr }</code>	<code>builder.For(enumeration, (fun identifier -> { cexpr }))</code>
<code>{ while expr do cexpr }</code>	<code>builder.While(fun () -> expr, builder.Delay({ cexpr }))</code>
<code>{ try cexpr with pattern_i -> expr_i }</code>	<code>builder.TryWith(builder.Delay({ cexpr })), (fun value -> match value with pattern_i -> expr_i exn -> reraise exn)))</code>
<code>{ try cexpr finally expr }</code>	<code>builder.TryFinally(builder.Delay({ cexpr })), (fun () -> expr))</code>
<code>{ cexpr1; cexpr2 }</code>	<code>builder.Combine({ cexpr1 }, { cexpr2 })</code>
<code>{ other-expr; cexpr }</code>	<code>expr; { cexpr }</code>
<code>{ other-expr }</code>	<code>expr; builder.Zero()</code>

在上表中，`other-expr` 描述資料表中未列出的運算式。產生器類別不需要執行所有方法，並支援上表所列的所有翻譯。在該類型的計算運算式中，未執行的這些結構無法使用。例如，如果您不想要 `use` 在計算運算式中支援關鍵字，可以省略 `use` `builder` 類別中的定義。

下列程式碼範例會示範將計算封裝成一系列步驟的計算運算式，這些步驟可以一次一個步驟來評估。差異聯集型別，會將 `OkOrException` 目前所評估的運算式的錯誤狀態編碼。這段程式碼示範一些您可以在計算運算式中使用的一般模式，例如一些產生器方法的未定案實作為。

```
// Computations that can be run step by step
type Eventually<'T> =
    | Done of 'T
    | NotYetDone of (unit -> Eventually<'T>)

module Eventually =
    // The bind for the computations. Append 'func' to the
    // computation.
    let rec bind func expr =
        match expr with
        | Done value -> func value
        | NotYetDone work -> NotYetDone (fun () -> bind func (work()))

    // Return the final value wrapped in the Eventually type.
    let result value = Done value

    type OkOrException<'T> =
        | Ok of 'T
        | Exception of System.Exception

    // The catch for the computations. Stitch try/with throughout
    // the computation, and return the overall result as an OkOrException.
    let rec catch expr =
        match expr with
        | Done value -> result (Ok value)
        | NotYetDone work ->
            NotYetDone (fun () ->
                let res = try Ok(work()) with | exn -> Exception exn
                match res with
                | Ok cont -> catch cont // note, a tailcall
                | Exception exn -> result (Exception exn))

    // The delay operator.
    let delay func = NotYetDone (fun () -> func())

    // The stepping action for the computations.
    let step expr =
        match expr with
        | Done _ -> expr
        | NotYetDone func -> func ()

    // The rest of the operations are boilerplate.
    // The tryFinally operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryFinally expr compensation =
        catch (expr)
        |> bind (fun res ->
            compensation();
            match res with
            | Ok value -> result value
            | Exception exn -> raise exn)

    // The tryWith operator.
    // This is boilerplate in terms of "result", "catch", and "bind".
    let tryWith exn handler =
        catch exn
        |> bind (function Ok value -> result value | Exception exn -> handler exn)

    // The whileLoop operator.
    // This is boilerplate in terms of "result" and "bind".
    let rec whileLoop pred body =
        if pred() then body |> bind (fun _ -> whileLoop pred body)
        else result ()
```

```

// The sequential composition operator.
// This is boilerplate in terms of "result" and "bind".
let combine expr1 expr2 =
    expr1 |> bind (fun () -> expr2)

// The using operator.
let using (resource: #System.IDisposable) func =
    tryFinally (func resource) (fun () -> resource.Dispose())

// The forLoop operator.
// This is boilerplate in terms of "catch", "result", and "bind".
let forLoop (collection:seq<_>) func =
    let ie = collection.GetEnumerator()
    tryFinally
        (whileLoop
            (fun () -> ie.MoveNext())
            (delay (fun () -> let value = ie.Current in func value)))
        (fun () -> ie.Dispose())

// The builder class.
type EventuallyBuilder() =
    member x.Bind(comp, func) = Eventually.bind func comp
    member x.Return(value) = Eventually.result value
    member x.ReturnFrom(value) = value
    member x.Combine(expr1, expr2) = Eventually.combine expr1 expr2
    member x.Delay(func) = Eventually.delay func
    member x.Zero() = Eventually.result ()
    member x.TryWith(expr, handler) = Eventually.tryWith expr handler
    member x.TryFinally(expr, compensation) = Eventually.tryFinally expr compensation
    member x.For(coll:seq<_>, func) = Eventually.forLoop coll func
    member x.Using(resource, expr) = Eventually.using resource expr

let eventually = new EventuallyBuilder()

let comp = eventually {
    for x in 1..2 do
        printfn $" x = %d{x}"
    return 3 + 4 }

// Try the remaining lines in F# interactive to see how this
// computation expression works in practice.
let step x = Eventually.step x

// returns "NotYetDone <closure>"
comp |> step

// prints "x = 1"
// returns "NotYetDone <closure>"
comp |> step |> step

// prints "x = 1"
// prints "x = 2"
// returns "Done 7"
comp |> step |> step |> step |> step

```

計算運算式具有運算式傳回的基礎類型。基礎類型可能代表可執行檔計算結果或延遲計算，或可能提供逐一查看某些類型集合的方法。在上述範例中，最後是基礎類型。若為序列運算式，基礎類型為 [System.Collections.Generic.IEnumerable<T>](#)。若為查詢運算式，基礎類型為 [System.Linq.IQueryable](#)。如果是非同步工作流程，基礎類型為 [Async](#)。Async 物件代表要執行以計算結果的工作。例如，您可以呼叫 [Async.RunSynchronously](#) 來執行計算，並傳回結果。

自訂作業

您可以在計算運算式上定義自訂作業，並使用自訂作業做為運算運算式中的運算子。例如，您可以在查詢運算式

中包含查詢運算子。當您定義自訂作業時，您必須在計算運算式中定義 Yield 和方法。若要定義自訂作業，請將它放在計算運算式的 builder 類別中，然後套用 `CustomOperationAttribute`。這個屬性會採用字串做為引數，這是要在自訂作業中使用的名稱。這個名稱會在計算運算式的左大括弧開頭的範圍內。因此，您不應該使用與此區塊中的自訂作業同名的識別碼。例如，請避免在 `all` 查詢運算式中使用識別碼，例如或 `last`。

以新的自訂作業擴充現有的產生器

如果您已經有 builder 類別，則可以從這個產生器類別外部延伸其自訂作業。擴充功能必須在模組中宣告。命名空間不能包含延伸成員，除了在相同檔案和定義類型的相同命名空間宣告群組中。

下列範例顯示現有類別的擴充 `FSharp.Linq.QueryBuilder`。

```
type FSharp.Linq.QueryBuilder with

    [
```

另請參閱

- [F # 語言參考](#)
- [非同步工作流程](#)
- [序列](#)
- [查詢運算式](#)

非同步工作流程

2020/11/2 • [Edit Online](#)

本文描述 F# 中的支援以非同步方式執行計算，也就是不會封鎖其他工作的執行。例如，您可以使用非同步計算來撰寫應用程式，這些應用程式的 UI 會在應用程式執行其他工作時，仍能回應使用者。

語法

```
async { expression }
```

備註

在先前的語法中，的計算 `expression` 是設定為以非同步方式執行，也就是在執行非同步睡眠作業、i/o 和其他非同步作業時，不會封鎖目前的計算執行緒。非同步計算通常會在背景執行緒上啟動，而在目前的執行緒上繼續執行。運算式的型別是 `Async<'T>`，其中 `'T` 是使用關鍵字時，運算式所傳回的型別 `return`。這類運算式中的程式碼稱為 *非同步區塊*或 *非同步區塊*。

非同步程式設計的方式有許多種，而類別則 `Async` 提供支援數個案例的方法。一般的方法是建立 `Async` 物件，以代表您想要以非同步方式執行的計算或計算，然後使用其中一個觸發函數來啟動這些計算。不同的觸發函式提供不同的方式來執行非同步計算，而您要使用哪一個方法取決於您要使用目前的執行緒、背景執行緒或 .NET Framework 工作物件，以及在計算完成時是否應該執行接續函數。例如，若要在目前的執行緒上啟動非同步計算，您可以使用 `Async.StartImmediate`。當您從 UI 執行緒啟動非同步計算時，不會封鎖處理使用者動作（例如按鍵和滑鼠活動）的主要事件迴圈，讓應用程式保持回應。

使用 let ! 的非同步綁定

在非同步工作流程中，某些運算式和作業是同步的，有些則是較長的計算，其設計目的是要以非同步方式傳回結果。當您以非同步方式呼叫方法，而不是使用一般系結時 `let`，就會使用 `let!`。的作用 `let!` 是在執行計算時，讓執行在其他計算或執行緒上繼續執行。在系結的右側傳回之後 `let!`，其餘的非同步工作流程會繼續執行。

下列程式碼顯示與之間的 `let` 差異 `let!`。使用的程式碼 `let` 只會建立異步計算做為物件，您稍後可以使用來執行，例如 `Async.StartImmediate` 或 `Async.RunSynchronously`。使用的程式碼 `let!` 會開始計算，然後執行緒會暫停，直到結果可用為止，此時會繼續執行。

```
// let just stores the result as an asynchronous operation.
let (result1 : Async<byte[]>) = stream.AsyncRead(bufferSize)
// let! completes the asynchronous operation and returns the data.
let! (result2 : byte[]) = stream.AsyncRead(bufferSize)
```

除了之外 `let!`，您還可以使用 `use!` 來執行非同步系結。和之間的差異與 `let!` `use!` 和之間的差異相同 `let` `use`。若為 `use!`，則會在目前的範圍結束時處置物件。請注意，在目前的 F# 語言版本中，不 `use!` 允許將值初始化為 null，即使這樣做也一樣 `use`。

非同步基本類型

執行單一非同步工作並傳回結果的方法，稱為 *非同步基本類型*，而這些是專門設計來搭配使用 `let!`。F# 核心程式庫中定義了數個非同步基本類型。這兩種 Web 應用程式的方法都是在模組中定義

`FSharp.Control.WebExtensions`：`WebRequest.AsyncGetResponse` 和 `WebClient.AsyncDownloadString`。這兩個基本專案都會從網頁下載資料(假設有一個 URL)。`AsyncGetResponse` 產生 `System.Net.WebResponse` 物件，並 `AsyncDownloadString` 產生表示網頁 HTML 的字串。

課程模組中包含數個非同步 i/o 作業的基本專案 `FSharp.Control.CommonExtensions`。類別的這些擴充方法 `System.IO.Stream` 為 `Stream.AsyncRead` 和 `Stream.AsyncWrite`。

您也可以藉由定義一個函式，其完整主體會以非同步區塊括住，以撰寫您自己的非同步基本專案。

若要在使用 F# 非同步程式設計模型的其他非同步模型所設計的 .NET Framework 中使用非同步方法，您可以建立會傳回 F# 物件的函式 `Async`。F# 程式庫具有可讓您輕鬆執行此作業的函式。

這裡包含使用非同步工作流程的其中一個範例;檔中有許多其他的方法，適用於 [非同步類別](#) 的方法。

此範例示範如何使用非同步工作流程，以平行方式執行計算。

在下列程式碼範例中，函式會 `fetchAsync` 取得 Web 要求所傳回的 HTML 文字。`fetchAsync` 函數包含非同步程式碼區塊。當您對非同步基本的結果進行系結時，在此情況下 `AsyncDownloadString`，`let!` 會使用，而不是 `let`。

您可以使用函數 `Async.RunSynchronously` 來執行非同步作業，並等候其結果。例如，您可以使用函式搭配函數來平行執行多個非同步作業 `Async.Parallel` `Async.RunSynchronously`。此函式 `Async.Parallel` 會取得物件的清單 `Async`、將每個工作物件的程式碼設定 `Async` 為平行執行，並傳回 `Async` 代表平行計算的物件。就像單一作業一樣，您可以呼叫 `Async.RunSynchronously` 來開始執行。

函式 `runAll` 會以平行方式啟動三個非同步工作流程，並等候直到全部完成為止。

```
open System.Net
open Microsoft.FSharp.Control.WebExtensions

let urlList = [ "Microsoft.com", "http://www.microsoft.com/"
                "MSDN", "http://msdn.microsoft.com/"
                "Bing", "http://www.bing.com"
              ]

let fetchAsync(name, url:string) =
    async {
        try
            let uri = new System.Uri(url)
            let webClient = new WebClient()
            let! html = webClient.AsyncDownloadString(uri)
            printfn "Read %d characters for %s" html.Length name
        with
            | ex -> printfn "%s" (ex.Message);
    }

let runAll() =
    urlList
    |> Seq.map fetchAsync
    |> Async.Parallel
    |> Async.RunSynchronously
    |> ignore

runAll()
```

另請參閱

- [F# 語言參考](#)
- [計算運算式](#)
- [Control.Async 類別](#)

查詢運算式

2020/11/2 • [Edit Online](#)

查詢運算式可讓您查詢資料來源，並將資料放入所需的表單。查詢運算式提供 F# 中 LINQ 的支援。

語法

```
query { expression }
```

備註

查詢運算式是一種類似於順序運算式的計算運算式。就像您在序列運算式中提供程式碼來指定序列一樣，您也可以查詢運算式中提供程式碼來指定一組資料。在序列運算式中，`yield` 關鍵字會識別要傳回做為結果序列一部分的資料。在查詢運算式中，`select` 關鍵字會執行相同的函式。除了 `select` 關鍵字之外，F# 也支援許多查詢運算子，這些運算子與 SQL SELECT 語句的部分很類似。以下是簡單查詢運算式的範例，以及連接到 Northwind OData 來源的程式碼。

```
// Use the OData type provider to create types that can be used to access the Northwind database.
// Add References to FSharp.Data.TypeProviders and System.Data.Services.Client
open Microsoft.FSharp.Data.TypeProviders

type Northwind = ODataService<"http://services.odata.org/Northwind/Northwind.svc">
let db = Northwind.GetDataContext()

// A query expression.
let query1 =
    query {
        for customer in db.Customers do
            select customer
    }

// Print results
query1
|> Seq.iter (fun customer -> printfn "Company: %s Contact: %s" customer.CompanyName customer.ContactName)
```

在上述程式碼範例中，查詢運算式是以大括弧括住。運算式中程式碼的意義是，會傳回查詢結果中資料庫之 Customers 資料表中的每個客戶。查詢運算式會傳回實和的型別 `IQueryable<T>` `IEnumerable<T>`，因此可以使用 [Seq 模組](#) 進行反覆運算，如範例所示。

每個計算運算式類型都是從產生器類別建立的。查詢計算運算式的 builder 類別為 `QueryBuilder`。如需詳細資訊，請參閱 [計算運算式](#) 和 [QueryBuilder 類別](#)。

查詢運算子

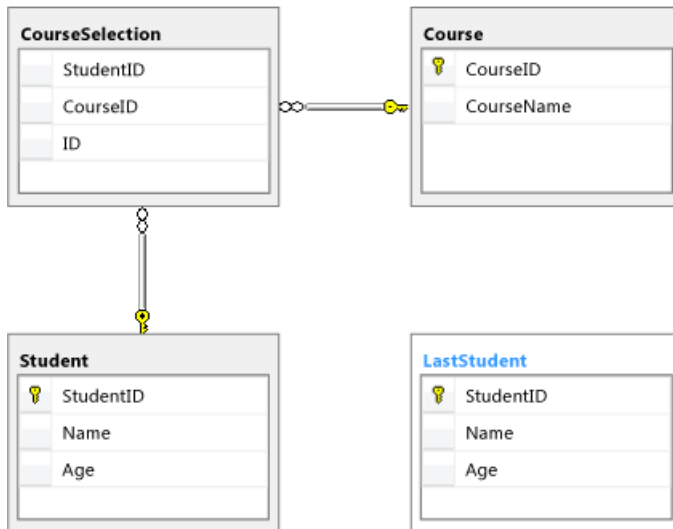
查詢運算子可讓您指定查詢的詳細資料，例如，將條件放在要傳回的記錄，或指定結果的排序次序。查詢來源必須支援查詢運算子。如果您嘗試使用不支援的查詢運算子，將會擲回 `System.NotSupportedException`。

查詢運算式中只允許可轉譯為 SQL 的運算式。例如，當您使用查詢運算子時，運算式中不允許函式呼叫 `where`。

表1顯示可用的查詢運算子。此外，請參閱本主題稍後的 Table2，它會比較 SQL 查詢和對等的 F# 查詢運算式。某些型別提供者不支援某些查詢運算子。尤其是，OData 型別提供者在支援的查詢運算子中會受到限制，因為

OData 的限制。

此資料表採用下列格式的資料庫：



接下來的表格中的程式碼也會假設下列資料庫連接程式碼。專案應加入 Fsharp.core 的參考，以及 Fsharp.data.typeproviders 元件的參考資料。本主題結尾會包含建立此資料庫的程式碼。

```
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq
open Microsoft.FSharp.Linq

type schema = SqlConnection< @"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated Security=SSPI;" >

let db = schema.GetDataContext()

// Needed for some query operator examples:
let data = [ 1; 5; 7; 11; 18; 21]
```

表 1. 查詢運算子

<div>contains</div>	<p>判斷選取的專案是否包含指定的元素。</p> <pre>query { for student in db.Student do select student.Age.Value contains 11 }</pre>
<div>count</div>	<p>傳回選取的元素數目。</p> <pre>query { for student in db.Student do select student count }</pre>

last	<p>選取到目前為止所選取的最後一個元素。</p> <pre> query { for number in data do last } </pre>
lastOrDefault	<p>選取到目前為止選取的最後一個專案, 如果找不到任何專案, 則為預設值。</p> <pre> query { for number in data do where (number < 0) lastOrDefault } </pre>
exactlyOne	<p>選取到目前為止選取的單一特定元素。如果有多個元素, 則會擲回例外狀況。</p> <pre> query { for student in db.Student do where (student.StudentID = 1) select student exactlyOne } </pre>
exactlyOneOrDefault	<p>選取到目前為止選取的單一特定專案, 如果找不到該元素, 則為預設值。</p> <pre> query { for student in db.Student do where (student.StudentID = 1) select student exactlyOneOrDefault } </pre>
headOrDefault	<p>選取到目前為止選取的第一個專案, 如果序列中沒有包含任何元素, 則為預設值。</p> <pre> query { for student in db.Student do select student headOrDefault } </pre>

<div data-bbox="193 103 276 132">select</div>	<p>投射到目前為止所選取的每個元素。</p> <div data-bbox="834 183 1433 347"> <pre> query { for student in db.Student do select student } </pre> </div>
<div data-bbox="193 414 266 443">where</div>	<p>根據指定的述詞選取元素。</p> <div data-bbox="834 495 1433 683"> <pre> query { for student in db.Student do where (student.StudentID > 4) select student } </pre> </div>
<div data-bbox="193 754 266 784">minBy</div>	<p>為目前為止選取的每個專案選取一個值，並傳回最小產生的值。</p> <div data-bbox="834 864 1433 1028"> <pre> query { for student in db.Student do minBy student.StudentID } </pre> </div>
<div data-bbox="193 1097 266 1126">maxBy</div>	<p>為目前為止選取的每個專案選取一個值，並傳回產生的最大值。</p> <div data-bbox="834 1207 1433 1370"> <pre> query { for student in db.Student do maxBy student.StudentID } </pre> </div>
<div data-bbox="193 1440 288 1469">groupBy</div>	<p>根據指定的索引鍵選取器，將所選的元素分組。</p> <div data-bbox="834 1518 1433 1706"> <pre> query { for student in db.Student do groupBy student.Age into g select (g.Key, g.Count()) } </pre> </div>
<div data-bbox="193 1778 276 1807">sortBy</div>	<p>依據指定的排序索引鍵，以遞增順序排序所選取的元素。</p> <div data-bbox="834 1856 1433 2045"> <pre> query { for student in db.Student do sortBy student.Name select student } </pre> </div>

<div>sortByDescending</div>	<p>依據指定的排序索引鍵，以遞減順序排序所選取的元素。</p> <pre> query { for student in db.Student do sortByDescending student.Name select student }</pre>
<div>thenBy</div>	<p>依指定的排序索引鍵，以遞增循序執行所選取專案的後續排序。只有在 <code>sortBy</code> 、或之後，才能使用這個運算子 <code>sortByDescending</code> <code>thenBy</code> <code>thenByDescending</code> 。</p> <pre> query { for student in db.Student do where student.Age.HasValue sortBy student.Age.Value thenBy student.Name select student }</pre>
<div>thenByDescending</div>	<p>依指定的排序索引鍵，以遞減循序執行所選取專案的後續排序。只有在 <code>sortBy</code> 、或之後，才能使用這個運算子 <code>sortByDescending</code> <code>thenBy</code> <code>thenByDescending</code> 。</p> <pre> query { for student in db.Student do where student.Age.HasValue sortBy student.Age.Value thenByDescending student.Name select student }</pre>
<div>groupValBy</div>	<p>為目前為止選取的每個專案選取一個值，並依指定的索引鍵將元素分組。</p> <pre> query { for student in db.Student do groupValBy student.Name student.Age into g select (g, g.Key, g.Count()) }</pre>

<div>join</div>	<p>根據相符的索引鍵, 相互關聯兩組選取的值。請注意, 聯接運算式的 = sign 周圍的索引鍵順序很重要。在 [所有聯結] 中, 如果行在符號之後分割 <code>-></code>, 縮排必須至少縮排至關鍵字 <code>for</code> 。</p> <pre> query { for student in db.Student do join selection in db.CourseSelection on (student.StudentID = selection.StudentID) select (student, selection) }</pre>
<div>groupJoin</div>	<p>根據相符的索引鍵將兩組選取的值相互關聯, 並將結果分組。請注意, 聯接運算式的 = sign 周圍的索引鍵順序很重要。</p> <pre> query { for student in db.Student do groupJoin courseSelection in db.CourseSelection on (student.StudentID = courseSelection.StudentID) into g for courseSelection in g do join course in db.Course on (courseSelection.CourseID = course.CourseID) select (student.Name, course.CourseName) }</pre>
<div>leftOuterJoin</div>	<p>根據相符的索引鍵將兩組選取的值相互關聯, 並將結果分組。如果任何群組都是空的, 則會改用具有單一預設值的群組。請注意, 聯接運算式的 = sign 周圍的索引鍵順序很重要。</p> <pre> query { for student in db.Student do leftOuterJoin selection in db.CourseSelection on (student.StudentID = selection.StudentID) into result for selection in result.DefaultIfEmpty() do select (student, selection) }</pre>
<div>sumByNullable</div>	<p>為目前為止選取的每個專案選取可為 null 的值, 並傳回這些值的總和。如果任何可為 null 的沒有值, 則會予以忽略。</p> <pre> query { for student in db.Student do sumByNullable student.Age }</pre>

minByNullable	<p>為目前為止選取的每個專案選取可為 null 的值，並傳回這些值的最小值。如果任何可為 null 的沒有值，則會予以忽略。</p> <pre> query { for student in db.Student do minByNullable student.Age } </pre>
maxByNullable	<p>為目前為止選取的每個專案選取可為 null 的值，並傳回這些值的最大值。如果任何可為 null 的沒有值，則會予以忽略。</p> <pre> query { for student in db.Student do maxByNullable student.Age } </pre>
averageByNullable	<p>為目前為止選取的每個專案選取可為 null 的值，並傳回這些值的平均值。如果任何可為 null 的沒有值，則會予以忽略。</p> <pre> query { for student in db.Student do averageByNullable (Nullable.float student.Age) } </pre>
averageBy	<p>為目前為止選取的每個專案選取一個值，並傳回這些值的平均值。</p> <pre> query { for student in db.Student do averageBy (float student.StudentID) } </pre>
distinct	<p>從目前為止選取的元素中選取相異元素。</p> <pre> query { for student in db.Student do join selection in db.CourseSelection on (student.StudentID = selection.StudentID) distinct } </pre>

exists	<p>判斷任何選取到目前為止的元素是否符合條件。</p> <pre> query { for student in db.Student do where (query { for courseSelection in db.CourseSelection do exists (courseSelection.StudentID = student.StudentID) }) select student } } </pre>
find	<p>選取到目前為止所選取的第一個元素, 以滿足指定的條件。</p> <pre> query { for student in db.Student do find (student.Name = "Abercrombie, Kim") } </pre>
all	<p>判斷目前選取的所有元素是否都符合條件。</p> <pre> query { for student in db.Student do all (SqlMethods.Like(student.Name, "%,%")) } </pre>
head	<p>選取到目前為止所選取的第一個元素。</p> <pre> query { for student in db.Student do head } </pre>
nth	<p>在指定的索引中選取到目前為止所選取索引處的元素。</p> <pre> query { for numbers in data do nth 3 } </pre>

<div>skip</div>	<p>略過目前為止選取的專案數目，然後選取其餘的元素。</p> <pre> query { for student in db.Student do skip 1 } </pre>
<div>skipWhile</div>	<p>只要指定的條件為 true, 就會略過序列中的專案, 然後選取其餘的元素。</p> <pre> query { for number in data do skipWhile (number < 3) select student } </pre>
<div>sumBy</div>	<p>為目前為止選取的每個專案選取一個值, 並傳回這些值的總和。</p> <pre> query { for student in db.Student do sumBy student.StudentID } </pre>
<div>take</div>	<p>選取到目前為止所選取的連續元素數目。</p> <pre> query { for student in db.Student do select student take 2 } </pre>
<div>takeWhile</div>	<p>只要指定的條件為 true, 就會從序列中選取專案, 然後略過其餘的元素。</p> <pre> query { for number in data do takeWhile (number < 10) } </pre>

<code>sortByNullable</code>	<p>依給定可為 null 的排序關鍵字，以遞增順序排序所選取的元素。</p> <pre>query { for student in db.Student do sortByNullable student.Age select student }</pre>
<code>sortByNullableDescending</code>	<p>依給定可為 null 的排序關鍵字，以遞減順序排序所選取的元素。</p> <pre>query { for student in db.Student do sortByNullableDescending student.Age select student }</pre>
<code>thenByNullable</code>	<p>依給定可為 null 的排序關鍵字，以遞增循序執行所選取專案的後續排序。這個運算子只能在 <code>sortBy</code>、<code>sortByDescending</code>、<code>thenBy</code>、或 <code>thenByDescending</code> 其可為 null 的 variant 之後立即使用。</p> <pre>query { for student in db.Student do sortBy student.Name thenByNullable student.Age select student }</pre>
<code>thenByNullableDescending</code>	<p>依給定可為 null 的排序關鍵字，以遞減循序執行所選取專案的後續排序。這個運算子只能在 <code>sortBy</code>、<code>sortByDescending</code>、<code>thenBy</code>、或 <code>thenByDescending</code> 其可為 null 的 variant 之後立即使用。</p> <pre>query { for student in db.Student do sortBy student.Name thenByNullableDescending student.Age select student }</pre>

Transact-SQL 和 F# 查詢運算式的比較

下表顯示一些常見的 Transact-SQL 查詢及其在 F # 中的對應專案。此資料表中的程式碼也會假設與上一個資料表相同的資料庫，以及用來設定類型提供者的相同初始程式碼。

表 2. Transact-SQL 和 F# 查詢運算式

TRANSACT-SQL (T-SQL)	F # (F#)
----------------------	----------

<p>從資料表中選取所有欄位。</p> <pre>SELECT * FROM Student</pre>	<pre>// All students. query { for student in db.Student do select student }</pre>
<p>計算資料表中的記錄。</p> <pre>SELECT COUNT(*) FROM Student</pre>	<pre>// Count of students. query { for student in db.Student do count }</pre>
<p>EXISTS</p> <pre>SELECT * FROM Student WHERE EXISTS (SELECT * FROM CourseSelection WHERE CourseSelection.StudentID = Student.StudentID)</pre>	<pre>// Find students who have signed up at least one course. query { for student in db.Student do where (query { for courseSelection in db.CourseSelection do exists (courseSelection.StudentID = student.StudentID) }) select student }</pre>
<p>群組</p> <pre>SELECT Student.Age, COUNT(*) FROM Student GROUP BY Student.Age</pre>	<pre>// Group by age and count. query { for n in db.Student do groupBy n.Age into g select (g.Key, g.Count()) } // OR query { for n in db.Student do groupValBy n.Age n.Age into g select (g.Key, g.Count()) }</pre>
<p>使用條件分組。</p> <pre>SELECT Student.Age, COUNT(*) FROM Student GROUP BY Student.Age HAVING student.Age > 10</pre>	<pre>// Group students by age where age > 10. query { for student in db.Student do groupBy student.Age into g where (g.Key.HasValue && g.Key.Value > 10) select (g.Key, g.Count()) }</pre>

使用計數條件分組。

```
SELECT Student.Age, COUNT( * )
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
```

```
// Group students by age and count number of
students
// at each age with more than 1 student.
query {
  for student in db.Student do
    groupBy student.Age into group
    where (group.Count() > 1)
    select (group.Key, group.Count())
}
```

群組、計數和總和。

```
SELECT Student.Age, COUNT( * ),
SUM(Student.Age) as total
FROM Student
GROUP BY Student.Age
```

```
// Group students by age and sum ages.
query {
  for student in db.Student do
    groupBy student.Age into g
    let total =
      query {
        for student in g do
          sumByNullable student.Age
      }
    select (g.Key, g.Count(), total)
}
```

依計數分組、計算和排序。

```
SELECT Student.Age, COUNT( * ) as myCount
FROM Student
GROUP BY Student.Age
HAVING COUNT( * ) > 1
ORDER BY COUNT( * ) DESC
```

```
// Group students by age, count number of
students
// at each age, and display all with count > 1
// in descending order of count.
query {
  for student in db.Student do
    groupBy student.Age into g
    where (g.Count() > 1)
    sortByDescending (g.Count())
    select (g.Key, g.Count())
}
```

IN 一組指定的值

```
SELECT *
FROM Student
WHERE Student.StudentID IN (1, 2, 5, 10)
```

```
// Select students where studentID is one of a
given list.
let idQuery =
  query {
    for id in [1; 2; 5; 10] do
      select id
  }
query {
  for student in db.Student do
    where (idQuery.Contains(student.StudentID))
    select student
}
```

LIKE 和 TOP。

```
-- '_e%' matches strings where the second
character is 'e'
SELECT TOP 2 * FROM Student
WHERE Student.Name LIKE '_e%'
```

```
// Look for students with Name match _e%
pattern and take first two.
query {
  for student in db.Student do
    where (SqlMethods.Like( student.Name,
"_e%") )
    select student
    take 2
}
```

LIKE 已設定模式相符。

```
-- '[abc]%' matches strings where the first
character is
-- 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[abc]%'
```

```
query {
  for student in db.Student do
    where (SqlMethods.Like( student.Name, "[
abc]%" )
    select student
}
```

LIKE 使用設定排除模式。

```
-- '[^abc]%' matches strings where the first
character is
-- not 'a', 'b', 'c', 'A', 'B', or 'C'
SELECT * FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
// Look for students with name matching
[^abc]%% pattern.
query {
  for student in db.Student do
    where (SqlMethods.Like( student.Name, "[
^abc]%" )
    select student
}
```

LIKE 在一個欄位上, 選取不同的欄位。

```
SELECT StudentID AS ID FROM Student
WHERE Student.Name LIKE '[^abc]%'
```

```
query {
  for n in db.Student do
    where (SqlMethods.Like( n.Name, "[^abc]%" )
  )
  select n.StudentID
}
```

LIKE , 包含子字串搜尋。

```
SELECT * FROM Student
WHERE Student.Name like '%A%'
```

```
// Using Contains as a query filter.
query {
  for student in db.Student do
    where (student.Name.Contains("a"))
    select student
}
```

JOIN 有兩個數據表很簡單。

```
SELECT * FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join Student and CourseSelection tables.
query {
  for student in db.Student do
    join selection in db.CourseSelection
    on (student.StudentID =
selection.StudentID)
    select (student, selection)
}
```

LEFT JOIN 有兩個數據表。

```
SELECT * FROM Student
LEFT JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
//Left Join Student and CourseSelection tables.
query {
  for student in db.Student do
    leftOuterJoin selection in
      db.CourseSelection
      on (student.StudentID =
selection.StudentID) into result
    for selection in result.DefaultIfEmpty() do
      select (student, selection)
}
```

COUNT 的 JOIN

```
SELECT COUNT( * ) FROM Student
JOIN CourseSelection
ON Student.StudentID =
CourseSelection.StudentID
```

```
// Join with count.
query {
  for n in db.Student do
    join e in db.CourseSelection
      on (n.StudentID = e.StudentID)
    count
}
```

DISTINCT

```
SELECT DISTINCT StudentID FROM CourseSelection
```

```
// Join with distinct.
query {
  for student in db.Student do
    join selection in db.CourseSelection
      on (student.StudentID =
selection.StudentID)
    distinct
}
```

相異計數。

```
SELECT DISTINCT COUNT(StudentID) FROM
CourseSelection
```

```
// Join with distinct and count.
query {
  for n in db.Student do
    join e in db.CourseSelection
      on (n.StudentID = e.StudentID)
    distinct
    count
}
```

BETWEEN

```
SELECT * FROM Student
WHERE Student.Age BETWEEN 10 AND 15
```

```
// Selecting students with ages between 10 and
15.
query {
  for student in db.Student do
    where (student.Age ?>= 10 && student.Age ?<
15)
    select student
}
```

OR

```
SELECT * FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
```

```
// Selecting students with age that's either 11
or 12.
query {
  for student in db.Student do
    where (student.Age.Value = 11 ||
student.Age.Value = 12)
    select student
}
```

OR 使用順序

```
SELECT * FROM Student
WHERE Student.Age = 12 OR Student.Age = 13
ORDER BY Student.Age DESC
```

```
// Selecting students in a certain age range
and sorting.
query {
  for n in db.Student do
    where (n.Age.Value = 12 || n.Age.Value =
13)
    sortByNullableDescending n.Age
    select n
}
```

TOP、OR 和順序。

```
SELECT TOP 2 student.Name FROM Student
WHERE Student.Age = 11 OR Student.Age = 12
ORDER BY Student.Name DESC
```

```
// Selecting students with certain ages,
// taking account of the possibility of nulls.
query {
  for student in db.Student do
    where
      ((student.Age.HasValue &&
student.Age.Value = 11) ||
      (student.Age.HasValue &&
student.Age.Value = 12))
    sortByDescending student.Name
    select student.Name
    take 2
}
```

UNION 兩個查詢。

```
SELECT * FROM Student
UNION
SELECT * FROM lastStudent
```

```
let query1 =
  query {
    for n in db.Student do
      select (n.Name, n.Age)
  }

let query2 =
  query {
    for n in db.LastStudent do
      select (n.Name, n.Age)
  }

query2.Union (query1)
```

兩個查詢的交集。

```
SELECT * FROM Student
INTERSECT
SELECT * FROM LastStudent
```

```
let query1 =
    query {
        for n in db.Student do
            select (n.Name, n.Age)
        }

let query2 =
    query {
        for n in db.LastStudent do
            select (n.Name, n.Age)
        }

query1.Intersect(query2)
```

CASE 條件。

```
SELECT student.StudentID,
CASE Student.Age
    WHEN -1 THEN 100
    ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special value.
query {
    for student in db.Student do
        select
            (if student.Age.HasValue &&
student.Age.Value = -1 then
                (student.StudentID,
System.Nullable<int>(100), student.Age)
            else (student.StudentID, student.Age,
student.Age))
        }
}
```

多個案例。

```
SELECT Student.StudentID,
CASE Student.Age
    WHEN -1 THEN 100
    WHEN 0 THEN 1000
    ELSE Student.Age
END,
Student.Age
FROM Student
```

```
// Using if statement to alter results for
special values.
query {
    for student in db.Student do
        select
            (if student.Age.HasValue &&
student.Age.Value = -1 then
                (student.StudentID,
System.Nullable<int>(100), student.Age)
            elif student.Age.HasValue &&
student.Age.Value = 0 then
                (student.StudentID,
System.Nullable<int>(1000), student.Age)
            else (student.StudentID, student.Age,
student.Age))
        }
}
```

多個資料表。

```
SELECT * FROM Student, Course
```

```
// Multiple table select.
query {
  for student in db.Student do
    for course in db.Course do
      select (student, course)
    }
}
```

多個聯結。

```
SELECT Student.Name, Course.CourseName
FROM Student
JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Multiple joins.
query {
  for student in db.Student do
    join courseSelection in db.CourseSelection
      on (student.StudentID =
courseSelection.StudentID)
    join course in db.Course
      on (courseSelection.CourseID =
course.CourseID)
    select (student.Name, course.CourseName)
  }
}
```

多個左方外部聯結。

```
SELECT Student.Name, Course.CourseName
FROM Student
LEFT OUTER JOIN CourseSelection
ON CourseSelection.StudentID =
Student.StudentID
LEFT OUTER JOIN Course
ON Course.CourseID = CourseSelection.CourseID
```

```
// Using leftOuterJoin with multiple joins.
query {
  for student in db.Student do
    leftOuterJoin courseSelection in
db.CourseSelection
      on (student.StudentID =
courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty()
do
      leftOuterJoin course in db.Course
        on (courseSelection.CourseID =
course.CourseID) into g2
      for course in g2.DefaultIfEmpty() do
        select (student.Name, course.CourseName)
      }
    }
}
```

下列程式碼可以用來建立這些範例的範例資料庫。

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

USE [master];
GO

IF EXISTS (SELECT * FROM sys.databases WHERE name = 'MyDatabase')
DROP DATABASE MyDatabase;
GO

-- Create the MyDatabase database.
CREATE DATABASE MyDatabase COLLATE SQL_Latin1_General_CP1_CI_AS;
GO

-- Specify a simple recovery model
-- to keep the log growth to a minimum.
ALTER DATABASE MyDatabase
SET RECOVERY SIMPLE;
GO

USE MyDatabase;
```


GO

```
CREATE TABLE [dbo].[Course] (  
    [CourseID] INT NOT NULL,  
    [CourseName] NVARCHAR (50) NOT NULL,  
    PRIMARY KEY CLUSTERED ([CourseID] ASC)  
);
```

```
CREATE TABLE [dbo].[Student] (  
    [StudentID] INT NOT NULL,  
    [Name] NVARCHAR (50) NOT NULL,  
    [Age] INT NULL,  
    PRIMARY KEY CLUSTERED ([StudentID] ASC)  
);
```

```
CREATE TABLE [dbo].[CourseSelection] (  
    [ID] INT NOT NULL,  
    [StudentID] INT NOT NULL,  
    [CourseID] INT NOT NULL,  
    PRIMARY KEY CLUSTERED ([ID] ASC),  
    CONSTRAINT [FK_CourseSelection_ToTable] FOREIGN KEY ([StudentID]) REFERENCES [dbo].[Student] ([StudentID])  
    ON DELETE NO ACTION ON UPDATE NO ACTION,  
    CONSTRAINT [FK_CourseSelection_Course_1] FOREIGN KEY ([CourseID]) REFERENCES [dbo].[Course] ([CourseID]) ON  
    DELETE NO ACTION ON UPDATE NO ACTION  
);
```

```
CREATE TABLE [dbo].[LastStudent] (  
    [StudentID] INT NOT NULL,  
    [Name] NVARCHAR (50) NOT NULL,  
    [Age] INT NULL,  
    PRIMARY KEY CLUSTERED ([StudentID] ASC)  
);
```

-- Insert data into the tables.

USE MyDatabase

```
INSERT INTO Course (CourseID, CourseName)  
VALUES(1, 'Algebra I');  
INSERT INTO Course (CourseID, CourseName)  
VALUES(2, 'Trigonometry');  
INSERT INTO Course (CourseID, CourseName)  
VALUES(3, 'Algebra II');  
INSERT INTO Course (CourseID, CourseName)  
VALUES(4, 'History');  
INSERT INTO Course (CourseID, CourseName)  
VALUES(5, 'English');  
INSERT INTO Course (CourseID, CourseName)  
VALUES(6, 'French');  
INSERT INTO Course (CourseID, CourseName)  
VALUES(7, 'Chinese');
```

```
INSERT INTO Student (StudentID, Name, Age)  
VALUES(1, 'Abercrombie, Kim', 10);  
INSERT INTO Student (StudentID, Name, Age)  
VALUES(2, 'Abolrous, Hazen', 14);  
INSERT INTO Student (StudentID, Name, Age)  
VALUES(3, 'Hance, Jim', 12);  
INSERT INTO Student (StudentID, Name, Age)  
VALUES(4, 'Adams, Terry', 12);  
INSERT INTO Student (StudentID, Name, Age)  
VALUES(5, 'Hansen, Claus', 11);  
INSERT INTO Student (StudentID, Name, Age)  
VALUES(6, 'Penor, Lori', 13);  
INSERT INTO Student (StudentID, Name, Age)  
VALUES(7, 'Perham, Tom', 12);  
INSERT INTO Student (StudentID, Name, Age)  
VALUES(8, 'Peng, Yun-Feng', NULL);
```

```
INSERT INTO CourseSelection (ID, StudentID, CourseID)  
VALUES(1, 1, 2);
```

```
VALUES(1, 1, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(2, 1, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(3, 1, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(4, 2, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(5, 2, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(6, 2, 6);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(7, 2, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(8, 3, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(9, 3, 1);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(10, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(11, 4, 5);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(12, 4, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(13, 5, 3);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(14, 5, 2);
INSERT INTO CourseSelection (ID, StudentID, CourseID)
VALUES(15, 7, 3);
```

下列程式碼包含本主題中所顯示的範例程式碼。

```
#if INTERACTIVE
#r "FSharp.Data.TypeProviders.dll"
#r "System.Data.dll"
#r "System.Data.Linq.dll"
#endif
open System
open Microsoft.FSharp.Data.TypeProviders
open System.Data.Linq.SqlClient
open System.Linq

type schema = SqlConnection<"Data Source=SERVER\INSTANCE;Initial Catalog=MyDatabase;Integrated
Security=SSPI;">

let db = schema.GetDataContext()

let data = [1; 5; 7; 11; 18; 21]

type Nullable<'T when 'T : ( new : unit -> 'T) and 'T : struct and 'T :> ValueType > with
    member this.Print() =
        if this.HasValue then this.Value.ToString()
        else "NULL"

printfn "\ncontains query operator"
query {
    for student in db.Student do
        select student.Age.Value
        contains 11
}
|> printfn "Is at least one student age 11? %b"

printfn "\ncount query operator"
query {
    for student in db.Student do
        select student
        count
}
```

```

|> printfn "Number of students: %d"

printfn "\nlast query operator."
let num =
    query {
        for number in data do
            sortBy number
            last
    }
printfn "Last number: %d" num

open Microsoft.FSharp.Linq

printfn "\nlastOrDefault query operator."
query {
    for number in data do
        sortBy number
        lastOrDefault
}
|> printfn "lastOrDefault: %d"

printfn "\nexactlyOne query operator."
let student2 =
    query {
        for student in db.Student do
            where (student.StudentID = 1)
            select student
            exactlyOne
    }
printfn "Student with StudentID = 1 is %s" student2.Name

printfn "\nexactlyOneOrDefault query operator."
let student3 =
    query {
        for student in db.Student do
            where (student.StudentID = 1)
            select student
            exactlyOneOrDefault
    }
printfn "Student with StudentID = 1 is %s" student3.Name

printfn "\nheadOrDefault query operator."
let student4 =
    query {
        for student in db.Student do
            select student
            headOrDefault
    }
printfn "head student is %s" student4.Name

printfn "\nselect query operator."
query {
    for student in db.Student do
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nwhere query operator."
query {
    for student in db.Student do
        where (student.StudentID > 4)
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nminBy query operator."
let student5 =
    query {
        for student in db.Student do

```

```

        for student in db.Student do
            minBy student.StudentID
        }

printfn "\nmaxBy query operator."
let student6 =
    query {
        for student in db.Student do
            maxBy student.StudentID
        }

printfn "\ngroupBy query operator."
query {
    for student in db.Student do
        groupBy student.Age into g
        select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "Age: %s Count at that age: %d" (age.Print()) count)

printfn "\nsortBy query operator."
query {
    for student in db.Student do
        sortBy student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nsortByDescending query operator."
query {
    for student in db.Student do
        sortByDescending student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.StudentID student.Name)

printfn "\nthenBy query operator."
query {
    for student in db.Student do
        where student.Age.HasValue
        sortBy student.Age.Value
        thenBy student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\nthenByDescending query operator."
query {
    for student in db.Student do
        where student.Age.HasValue
        sortBy student.Age.Value
        thenByDescending student.Name
        select student
}
|> Seq.iter (fun student -> printfn "StudentID, Name: %d %s" student.Age.Value student.Name)

printfn "\ngroupValBy query operator."
query {
    for student in db.Student do
        groupValBy student.Name student.Age into g
        select (g, g.Key, g.Count())
}
|> Seq.iter (fun (group, age, count) ->
    printfn "Age: %s Count at that age: %d" (age.Print()) count
    group |> Seq.iter (fun name -> printfn "Name: %s" name))

printfn "\n sumByNullable query operator"
query {
    for student in db.Student do
        sumByNullable student.Age

```

```

}
|> (fun sum -> printfn "Sum of ages: %s" (sum.Print()))

printfn "\n minByNullable"
query {
    for student in db.Student do
        minByNullable student.Age
}
|> (fun age -> printfn "Minimum age: %s" (age.Print()))

printfn "\n maxByNullable"
query {
    for student in db.Student do
        maxByNullable student.Age
}
|> (fun age -> printfn "Maximum age: %s" (age.Print()))

printfn "\n averageBy"
query {
    for student in db.Student do
        averageBy (float student.StudentID)
}
|> printfn "Average student ID: %f"

printfn "\n averageByNullable"
query {
    for student in db.Student do
        averageByNullable (Nullable.float student.Age)
}
|> (fun avg -> printfn "Average age: %s" (avg.Print()))

printfn "\n find query operator"
query {
    for student in db.Student do
        find (student.Name = "Abercrombie, Kim")
}
|> (fun student -> printfn "Found a match with StudentID = %d" student.StudentID)

printfn "\n all query operator"
query {
    for student in db.Student do
        all (SqlMethods.Like(student.Name, "%,%"))
}
|> printfn "Do all students have a comma in the name? %b"

printfn "\n head query operator"
query {
    for student in db.Student do
        head
}
|> (fun student -> printfn "Found the head student with StudentID = %d" student.StudentID)

printfn "\n nth query operator"
query {
    for numbers in data do
        nth 3
}
|> printfn "Third number is %d"

printfn "\n skip query operator"
query {
    for student in db.Student do
        skip 1
}
|> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

printfn "\n skipWhile query operator"
query {
    for number in data do

```

```

        skipWhile (number < 3)
        select number
    }
    |> Seq.iter (fun number -> printfn "Number = %d" number)

    printfn "\n sumBy query operator"
    query {
        for student in db.Student do
            sumBy student.StudentID
    }
    |> printfn "Sum of student IDs: %d"

    printfn "\n take query operator"
    query {
        for student in db.Student do
            select student
            take 2
    }
    |> Seq.iter (fun student -> printfn "StudentID = %d" student.StudentID)

    printfn "\n takeWhile query operator"
    query {
        for number in data do
            takeWhile (number < 10)
    }
    |> Seq.iter (fun number -> printfn "Number = %d" number)

    printfn "\n sortByNullable query operator"
    query {
        for student in db.Student do
            sortByNullable student.Age
            select student
    }
    |> Seq.iter (fun student ->
        printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

    printfn "\n sortByNullableDescending query operator"
    query {
        for student in db.Student do
            sortByNullableDescending student.Age
            select student
    }
    |> Seq.iter (fun student ->
        printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

    printfn "\n thenByNullable query operator"
    query {
        for student in db.Student do
            sortBy student.Name
            thenByNullable student.Age
            select student
    }
    |> Seq.iter (fun student ->
        printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

    printfn "\n thenByNullableDescending query operator"
    query {
        for student in db.Student do
            sortBy student.Name
            thenByNullableDescending student.Age
            select student
    }
    |> Seq.iter (fun student ->
        printfn "StudentID, Name, Age: %d %s %s" student.StudentID student.Name (student.Age.Print()))

    printfn "All students: "
    query {
        for student in db.Student do
            select student
    }

```

```

}
|> Seq.iter (fun student -> printfn "%s %d %s" student.Name student.StudentID (student.Age.Print()))

printfn "\nCount of students: "
query {
    for student in db.Student do
        count
}
|> (fun count -> printfn "Student count: %d" count)

printfn "\nExists."
query {
    for student in db.Student do
        where
            (query {
                for courseSelection in db.CourseSelection do
                    exists (courseSelection.StudentID = student.StudentID) })
        select student
}
|> Seq.iter (fun student -> printfn "%A" student.Name)

printfn "\n Group by age and count"
query {
    for n in db.Student do
        groupBy n.Age into g
        select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\n Group value by age."
query {
    for n in db.Student do
        groupValBy n.Age n.Age into g
        select (g.Key, g.Count())
}
|> Seq.iter (fun (age, count) -> printfn "%s %d" (age.Print()) count)

printfn "\nGroup students by age where age > 10."
query {
    for student in db.Student do
        groupBy student.Age into g
        where (g.Key.HasValue && g.Key.Value > 10)
        select (g, g.Key)
}
|> Seq.iter (fun (students, age) ->
    printfn "Age: %s" (age.Value.ToString())
    students
    |> Seq.iter (fun student -> printfn "%s" student.Name))

printfn "\nGroup students by age and print counts of number of students at each age with more than 1 student."
query {
    for student in db.Student do
        groupBy student.Age into group
        where (group.Count() > 1)
        select (group.Key, group.Count())
}
|> Seq.iter (fun (age, ageCount) ->
    printfn "Age: %s Count: %d" (age.Print()) ageCount)

printfn "\nGroup students by age and sum ages."
query {
    for student in db.Student do
        groupBy student.Age into g
        let total = query { for student in g do sumByNullable student.Age }
        select (g.Key, g.Count(), total)
}
|> Seq.iter (fun (age, count, total) ->
    printfn "Age: %d" (age.GetValueOrDefault()))

```

```

    printfn "Count: %d" count
    printfn "Total years: %s" (total.ToString()))

printfn "\nGroup students by age and count number of students at each age, and display all with count > 1 in
descending order of count."
query {
    for student in db.Student do
        groupBy student.Age into g
        where (g.Count() > 1)
        sortByDescending (g.Count())
        select (g.Key, g.Count())
}
|> Seq.iter (fun (age, myCount) ->
    printfn "Age: %s" (age.Print())
    printfn "Count: %d" myCount)

printfn "\n Select students from a set of IDs"
let idList = [1; 2; 5; 10]
let idQuery =
    query { for id in idList do select id }
query {
    for student in db.Student do
        where (idQuery.Contains(student.StudentID))
        select student
}
|> Seq.iter (fun student ->
    printfn "Name: %s" student.Name)

printfn "\nLook for students with Name match _e%% pattern and take first two."
query {
    for student in db.Student do
        where (SqlMethods.Like( student.Name, "_e%" ) )
        select student
        take 2
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with Name matching [abc]%% pattern."
query {
    for student in db.Student do
        where (SqlMethods.Like( student.Name, "[abc]%" ) )
        select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern."
query {
    for student in db.Student do
        where (SqlMethods.Like( student.Name, "[^abc]%" ) )
        select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nLook for students with name matching [^abc]%% pattern and select ID."
query {
    for n in db.Student do
        where (SqlMethods.Like( n.Name, "[^abc]%" ) )
        select n.StudentID
}
|> Seq.iter (fun id -> printfn "%d" id)

printfn "\n Using Contains as a query filter."
query {
    for student in db.Student do
        where (student.Name.Contains("a"))
        select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

```



```

printfn "\nSearching for names from a list."
let names = [|"a";"b";"c"|]
query {
    for student in db.Student do
        if names.Contains (student.Name) then select student
}
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\nJoin Student and CourseSelection tables."
query {
    for student in db.Student do
        join selection in db.CourseSelection
            on (student.StudentID = selection.StudentID)
        select (student, selection)
}
|> Seq.iter (fun (student, selection) -> printfn "%d %s %d" student.StudentID student.Name
selection.CourseID)

printfn "\nLeft Join Student and CourseSelection tables."
query {
    for student in db.Student do
        leftOuterJoin selection in db.CourseSelection
            on (student.StudentID = selection.StudentID) into result
        for selection in result.DefaultIfEmpty() do
            select (student, selection)
}
|> Seq.iter (fun (student, selection) ->
    let selectionID, studentID, courseID =
        match selection with
        | null -> "NULL", "NULL", "NULL"
        | sel -> (sel.ID.ToString(), sel.StudentID.ToString(), sel.CourseID.ToString())
    printfn "%d %s %d %s %s %s" student.StudentID student.Name (student.Age.GetValueOrDefault()) selectionID
studentID courseID)

printfn "\nJoin with count"
query {
    for n in db.Student do
        join e in db.CourseSelection
            on (n.StudentID = e.StudentID)
        count
}
|> printfn "%d"

printfn "\n Join with distinct."
query {
    for student in db.Student do
        join selection in db.CourseSelection
            on (student.StudentID = selection.StudentID)
        distinct
}
|> Seq.iter (fun (student, selection) -> printfn "%s %d" student.Name selection.CourseID)

printfn "\n Join with distinct and count."
query {
    for n in db.Student do
        join e in db.CourseSelection
            on (n.StudentID = e.StudentID)
        distinct
        count
}
|> printfn "%d"

printfn "\n Selecting students with age between 10 and 15."
query {
    for student in db.Student do
        where (student.Age.Value >= 10 && student.Age.Value < 15)
        select student
}
|> Seq.iter (fun student -> printfn "%e" student.Name)

```

```

|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students with age either 11 or 12."
query {
    for student in db.Student do
        where (student.Age.Value = 11 || student.Age.Value = 12)
        select student
    }
|> Seq.iter (fun student -> printfn "%s" student.Name)

printfn "\n Selecting students in a certain age range and sorting."
query {
    for n in db.Student do
        where (n.Age.Value = 12 || n.Age.Value = 13)
        sortByNullableDescending n.Age
        select n
    }
|> Seq.iter (fun student -> printfn "%s %s" student.Name (student.Age.Print()))

printfn "\n Selecting students with certain ages, taking account of possibility of nulls."
query {
    for student in db.Student do
        where
            ((student.Age.HasValue && student.Age.Value = 11) ||
             (student.Age.HasValue && student.Age.Value = 12))
        sortByDescending student.Name
        select student.Name
        take 2
    }
|> Seq.iter (fun name -> printfn "%s" name)

printfn "\n Union of two queries."
module Queries =
    let query1 = query {
        for n in db.Student do
            select (n.Name, n.Age)
        }

    let query2 = query {
        for n in db.LastStudent do
            select (n.Name, n.Age)
        }

    query2.Union (query1)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Intersect of two queries."
module Queries2 =
    let query1 = query {
        for n in db.Student do
            select (n.Name, n.Age)
        }

    let query2 = query {
        for n in db.LastStudent do
            select (n.Name, n.Age)
        }

    query1.Intersect(query2)
    |> Seq.iter (fun (name, age) -> printfn "%s %s" name (age.Print()))

printfn "\n Using if statement to alter results for special value."
query {
    for student in db.Student do
        select
            (if student.Age.HasValue && student.Age.Value = -1 then
                (student.StudentID, System.Nullable<int>(100), student.Age)
            else (student.StudentID, student.Age, student.Age))
    }

```

```

|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Using if statement to alter results special values."
query {
    for student in db.Student do
    select
        (if student.Age.HasValue && student.Age.Value = -1 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
        elif student.Age.HasValue && student.Age.Value = 0 then
            (student.StudentID, System.Nullable<int>(100), student.Age)
        else (student.StudentID, student.Age, student.Age))
}
|> Seq.iter (fun (id, value, age) -> printfn "%d %s %s" id (value.Print()) (age.Print()))

printfn "\n Multiple table select."
query {
    for student in db.Student do
    for course in db.Course do
    select (student, course)
}
|> Seq.iteri (fun index (student, course) ->
    if index = 0 then
        printfn "StudentID Name Age CourseID CourseName"
    printfn "%d %s %s %d %s" student.StudentID student.Name (student.Age.Print()) course.CourseID
course.CourseName)

printfn "\nMultiple Joins"
query {
    for student in db.Student do
    join courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID)
    join course in db.Course
        on (courseSelection.CourseID = course.CourseID)
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)

printfn "\nMultiple Left Outer Joins"
query {
    for student in db.Student do
    leftOuterJoin courseSelection in db.CourseSelection
        on (student.StudentID = courseSelection.StudentID) into g1
    for courseSelection in g1.DefaultIfEmpty() do
    leftOuterJoin course in db.Course
        on (courseSelection.CourseID = course.CourseID) into g2
    for course in g2.DefaultIfEmpty() do
    select (student.Name, course.CourseName)
}
|> Seq.iter (fun (studentName, courseName) -> printfn "%s %s" studentName courseName)

```

以下是在 F# 互動中執行此程式碼時的完整輸出。

```

--> Referenced 'C:\Program Files (x86)\Reference Assemblies\Microsoft\FSharp\3.0\Runtime\v4.0\Type
Providers\FSharp.Data.TypeProviders.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.dll'

--> Referenced 'C:\Windows\Microsoft.NET\Framework\v4.0.30319\System.Data.Linq.dll'

contains query operator
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp5E3C.dll'...
Binding session to 'C:\Users\ghogen\AppData\Local\Temp\tmp611A.dll'...
Is at least one student age 11? true

count query operator
Number of students: 8

1-st query operator

```

last query operator.

Last number: 21

lastOrDefault query operator.

lastOrDefault: 21

exactlyOne query operator.

Student with StudentID = 1 is Abercrombie, Kim

exactlyOneOrDefault query operator.

Student with StudentID = 1 is Abercrombie, Kim

headOrDefault query operator.

head student is Abercrombie, Kim

select query operator.

StudentID, Name: 1 Abercrombie, Kim

StudentID, Name: 2 Abolrous, Hazen

StudentID, Name: 3 Hance, Jim

StudentID, Name: 4 Adams, Terry

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 6 Penor, Lori

StudentID, Name: 7 Perham, Tom

StudentID, Name: 8 Peng, Yun-Feng

where query operator.

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 6 Penor, Lori

StudentID, Name: 7 Perham, Tom

StudentID, Name: 8 Peng, Yun-Feng

minBy query operator.

maxBy query operator.

groupBy query operator.

Age: NULL Count at that age: 1

Age: 10 Count at that age: 1

Age: 11 Count at that age: 1

Age: 12 Count at that age: 3

Age: 13 Count at that age: 1

Age: 14 Count at that age: 1

sortBy query operator.

StudentID, Name: 1 Abercrombie, Kim

StudentID, Name: 2 Abolrous, Hazen

StudentID, Name: 4 Adams, Terry

StudentID, Name: 3 Hance, Jim

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 8 Peng, Yun-Feng

StudentID, Name: 6 Penor, Lori

StudentID, Name: 7 Perham, Tom

sortByDescending query operator.

StudentID, Name: 7 Perham, Tom

StudentID, Name: 6 Penor, Lori

StudentID, Name: 8 Peng, Yun-Feng

StudentID, Name: 5 Hansen, Claus

StudentID, Name: 3 Hance, Jim

StudentID, Name: 4 Adams, Terry

StudentID, Name: 2 Abolrous, Hazen

StudentID, Name: 1 Abercrombie, Kim

thenBy query operator.

StudentID, Name: 10 Abercrombie, Kim

StudentID, Name: 11 Hansen, Claus

StudentID, Name: 12 Adams, Terry

StudentID, Name: 12 Hance, Jim

StudentID, Name: 12 Perham, Tom

StudentID, Name: 13 Penor, Lori
StudentID, Name: 14 Abolrous, Hazen

thenByDescending query operator.
StudentID, Name: 10 Abercrombie, Kim
StudentID, Name: 11 Hansen, Claus
StudentID, Name: 12 Perham, Tom
StudentID, Name: 12 Hance, Jim
StudentID, Name: 12 Adams, Terry
StudentID, Name: 13 Penor, Lori
StudentID, Name: 14 Abolrous, Hazen

groupValBy query operator.
Age: NULL Count at that age: 1
Name: Peng, Yun-Feng
Age: 10 Count at that age: 1
Name: Abercrombie, Kim
Age: 11 Count at that age: 1
Name: Hansen, Claus
Age: 12 Count at that age: 3
Name: Hance, Jim
Name: Adams, Terry
Name: Perham, Tom
Age: 13 Count at that age: 1
Name: Penor, Lori
Age: 14 Count at that age: 1
Name: Abolrous, Hazen

sumByNullable query operator
Sum of ages: 84

minByNullable
Minimum age: 10

maxByNullable
Maximum age: 14

averageBy
Average student ID: 4.500000

averageByNullable
Average age: 12

find query operator
Found a match with StudentID = 1

all query operator
Do all students have a comma in the name? true

head query operator
Found the head student with StudentID = 1

nth query operator
Third number is 11

skip query operator
StudentID = 2
StudentID = 3
StudentID = 4
StudentID = 5
StudentID = 6
StudentID = 7
StudentID = 8

skipWhile query operator
Number = 5
Number = 7
Number = 11
Number = 18

Number = 21

sumBy query operator
Sum of student IDs: 36

take query operator
StudentID = 1
StudentID = 2

takeWhile query operator
Number = 1
Number = 5
Number = 7

sortByNullable query operator
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 2 Abolrous, Hazen 14

sortByNullableDescending query operator
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 8 Peng, Yun-Feng NULL

thenByNullable query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12

thenByNullableDescending query operator
StudentID, Name, Age: 1 Abercrombie, Kim 10
StudentID, Name, Age: 2 Abolrous, Hazen 14
StudentID, Name, Age: 4 Adams, Terry 12
StudentID, Name, Age: 3 Hance, Jim 12
StudentID, Name, Age: 5 Hansen, Claus 11
StudentID, Name, Age: 8 Peng, Yun-Feng NULL
StudentID, Name, Age: 6 Penor, Lori 13
StudentID, Name, Age: 7 Perham, Tom 12

All students:
Abercrombie, Kim 1 10
Abolrous, Hazen 2 14
Hance, Jim 3 12
Adams, Terry 4 12
Hansen, Claus 5 11
Penor, Lori 6 13
Perham, Tom 7 12
Peng, Yun-Feng 8 NULL

Count of students:
Student count: 8

Exists.
"Abercrombie, Kim"
"Abolrous, Hazen"

"Hance, Jim"
"Adams, Terry"
"Hansen, Claus"
"Perham, Tom"

Group by age and count

NULL 1
10 1
11 1
12 3
13 1
14 1

Group value by age.

NULL 1
10 1
11 1
12 3
13 1
14 1

Group students by age where age > 10.

Age: 11
Hansen, Claus
Age: 12
Hance, Jim
Adams, Terry
Perham, Tom
Age: 13
Penor, Lori
Age: 14
Abolrous, Hazen

Group students by age and print counts of number of students at each age with more than 1 student.

Age: 12 Count: 3

Group students by age and sum ages.

Age: 0
Count: 1
Total years:
Age: 10
Count: 1
Total years: 10
Age: 11
Count: 1
Total years: 11
Age: 12
Count: 3
Total years: 36
Age: 13
Count: 1
Total years: 13
Age: 14
Count: 1
Total years: 14

Group students by age and count number of students at each age, and display all with count > 1 in descending order of count.

Age: 12
Count: 3

Select students from a set of IDs

Name: Abercrombie, Kim
Name: Abolrous, Hazen
Name: Hansen, Claus

Look for students with Name match _e% pattern and take first two.

Penor, Lori
Perham, Tom

Look for students with Name matching [abc]% pattern.

Abercrombie, Kim
Abolrous, Hazen
Adams, Terry

Look for students with name matching [^abc]% pattern.

Hance, Jim
Hansen, Claus
Penor, Lori
Perham, Tom
Peng, Yun-Feng

Look for students with name matching [^abc]% pattern and select ID.

3
5
6
7
8

Using Contains as a query filter.

Abercrombie, Kim
Abolrous, Hazen
Hance, Jim
Adams, Terry
Hansen, Claus
Perham, Tom

Searching for names from a list.

Join Student and CourseSelection tables.

2 Abolrous, Hazen 2
3 Hance, Jim 3
5 Hansen, Claus 5
2 Abolrous, Hazen 2
5 Hansen, Claus 5
6 Penor, Lori 6
3 Hance, Jim 3
2 Abolrous, Hazen 2
1 Abercrombie, Kim 1
2 Abolrous, Hazen 2
5 Hansen, Claus 5
2 Abolrous, Hazen 2
3 Hance, Jim 3
2 Abolrous, Hazen 2
3 Hance, Jim 3

Left Join Student and CourseSelection tables.

1 Abercrombie, Kim 10 9 3 1
2 Abolrous, Hazen 14 1 1 2
2 Abolrous, Hazen 14 4 2 2
2 Abolrous, Hazen 14 8 3 2
2 Abolrous, Hazen 14 10 4 2
2 Abolrous, Hazen 14 12 4 2
2 Abolrous, Hazen 14 14 5 2
3 Hance, Jim 12 2 1 3
3 Hance, Jim 12 7 2 3
3 Hance, Jim 12 13 5 3
3 Hance, Jim 12 15 7 3
4 Adams, Terry 12 NULL NULL NULL
5 Hansen, Claus 11 3 1 5
5 Hansen, Claus 11 5 2 5
5 Hansen, Claus 11 11 4 5
6 Penor, Lori 13 6 2 6
7 Perham, Tom 12 NULL NULL NULL
8 Peng, Yun-Feng 0 NULL NULL NULL

Join with count

15

Join with distinct.

Abercrombie, Kim 2
Abercrombie, Kim 3
Abercrombie, Kim 5
Abolrous, Hazen 2
Abolrous, Hazen 5
Abolrous, Hazen 6
Abolrous, Hazen 3
Hance, Jim 2
Hance, Jim 1
Adams, Terry 2
Adams, Terry 5
Adams, Terry 2
Hansen, Claus 3
Hansen, Claus 2
Perham, Tom 3

Join with distinct and count.

15

Selecting students with age between 10 and 15.

Abercrombie, Kim
Abolrous, Hazen
Hance, Jim
Adams, Terry
Hansen, Claus
Penor, Lori
Perham, Tom

Selecting students with age either 11 or 12.

Hance, Jim
Adams, Terry
Hansen, Claus
Perham, Tom

Selecting students in a certain age range and sorting.

Penor, Lori 13
Perham, Tom 12
Hance, Jim 12
Adams, Terry 12

Selecting students with certain ages, taking account of possibility of nulls.

Hance, Jim
Adams, Terry

Union of two queries.

Abercrombie, Kim 10
Abolrous, Hazen 14
Hance, Jim 12
Adams, Terry 12
Hansen, Claus 11
Penor, Lori 13
Perham, Tom 12
Peng, Yun-Feng NULL

Intersect of two queries.

Using if statement to alter results for special value.

1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL

Using if statement to alter results special values

using IF statement to alter results special values.

```
1 10 10
2 14 14
3 12 12
4 12 12
5 11 11
6 13 13
7 12 12
8 NULL NULL
```

Multiple table select.

StudentID Name Age CourseID CourseName

```
1 Abercrombie, Kim 10 1 Algebra I
2 Abolrous, Hazen 14 1 Algebra I
3 Hance, Jim 12 1 Algebra I
4 Adams, Terry 12 1 Algebra I
5 Hansen, Claus 11 1 Algebra I
6 Penor, Lori 13 1 Algebra I
7 Perham, Tom 12 1 Algebra I
8 Peng, Yun-Feng NULL 1 Algebra I
1 Abercrombie, Kim 10 2 Trigonometry
2 Abolrous, Hazen 14 2 Trigonometry
3 Hance, Jim 12 2 Trigonometry
4 Adams, Terry 12 2 Trigonometry
5 Hansen, Claus 11 2 Trigonometry
6 Penor, Lori 13 2 Trigonometry
7 Perham, Tom 12 2 Trigonometry
8 Peng, Yun-Feng NULL 2 Trigonometry
1 Abercrombie, Kim 10 3 Algebra II
2 Abolrous, Hazen 14 3 Algebra II
3 Hance, Jim 12 3 Algebra II
4 Adams, Terry 12 3 Algebra II
5 Hansen, Claus 11 3 Algebra II
6 Penor, Lori 13 3 Algebra II
7 Perham, Tom 12 3 Algebra II
8 Peng, Yun-Feng NULL 3 Algebra II
1 Abercrombie, Kim 10 4 History
2 Abolrous, Hazen 14 4 History
3 Hance, Jim 12 4 History
4 Adams, Terry 12 4 History
5 Hansen, Claus 11 4 History
6 Penor, Lori 13 4 History
7 Perham, Tom 12 4 History
8 Peng, Yun-Feng NULL 4 History
1 Abercrombie, Kim 10 5 English
2 Abolrous, Hazen 14 5 English
3 Hance, Jim 12 5 English
4 Adams, Terry 12 5 English
5 Hansen, Claus 11 5 English
6 Penor, Lori 13 5 English
7 Perham, Tom 12 5 English
8 Peng, Yun-Feng NULL 5 English
1 Abercrombie, Kim 10 6 French
2 Abolrous, Hazen 14 6 French
3 Hance, Jim 12 6 French
4 Adams, Terry 12 6 French
5 Hansen, Claus 11 6 French
6 Penor, Lori 13 6 French
7 Perham, Tom 12 6 French
8 Peng, Yun-Feng NULL 6 French
1 Abercrombie, Kim 10 7 Chinese
2 Abolrous, Hazen 14 7 Chinese
3 Hance, Jim 12 7 Chinese
4 Adams, Terry 12 7 Chinese
5 Hansen, Claus 11 7 Chinese
6 Penor, Lori 13 7 Chinese
7 Perham, Tom 12 7 Chinese
8 Peng, Yun-Feng NULL 7 Chinese
```

```

Multiple Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
Hance, Jim Trigonometry
Hance, Jim Algebra I
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Perham, Tom Algebra II

```

```

Multiple Left Outer Joins
Abercrombie, Kim Trigonometry
Abercrombie, Kim Algebra II
Abercrombie, Kim English
Abolrous, Hazen Trigonometry
Abolrous, Hazen English
Abolrous, Hazen French
Abolrous, Hazen Algebra II
Hance, Jim Trigonometry
Hance, Jim Algebra I
Adams, Terry Trigonometry
Adams, Terry English
Adams, Terry Trigonometry
Hansen, Claus Algebra II
Hansen, Claus Trigonometry
Penor, Lori
Perham, Tom Algebra II
Peng, Yun-Feng

```

```

type schema
val db : schema.ServiceTypes.SimpleDataContextTypes.MyDatabase1
val student : System.Data.Linq.Table<schema.ServiceTypes.Student>
val data : int list = [1; 5; 7; 11; 18; 21]
type Nullable<'T
    when 'T : (new : unit -> 'T) and 'T : struct and
    'T :> System.ValueType> with
    member Print : unit -> string
val num : int = 21
val student2 : schema.ServiceTypes.Student
val student3 : schema.ServiceTypes.Student
val student4 : schema.ServiceTypes.Student
val student5 : int = 1
val student6 : int = 8
val idList : int list = [1; 2; 5; 10]
val idQuery : seq<int>
val names : string [] = [|"a"; "b"; "c"|]
module Queries = begin
    val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
    val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end
module Queries2 = begin
    val query1 : System.Linq.IQueryable<string * System.Nullable<int>>
    val query2 : System.Linq.IQueryable<string * System.Nullable<int>>
end

```

另請參閱

- [F # 語言參考](#)
- [QueryBuilder 類別](#)

- 計算運算式

程式碼引號

2021/3/5 • [Edit Online](#)

本文描述程式 *代碼引號*，這是一種語言功能，可讓您以程式設計方式產生和使用 F# 程式碼運算式。這項功能可讓您產生表示 F# 程式碼的抽象語法樹狀目錄。然後，您可以根據應用程式的需求來進行和處理抽象語法樹狀結構。例如，您可以使用樹狀結構來產生 F# 程式碼，或以某些其他語言產生程式碼。

引號運算式

以 *引號括住* 的運算式是程式碼中的 F# 運算式，以不是編譯為程式一部分的方式來分隔，而是編譯成代表 F# 運算式的物件。您可以使用下列兩種方式的其中一種來標記引號運算式：使用類型資訊或不使用類型資訊。如果您想要包含型別資訊，請使用符號 `<@` 和 `@>` 來分隔加上引號的運算式。如果您不需要類型資訊，您可以使用符號 `<@@` 和 `@@>`。下列程式碼顯示型別和不具類型的引號。

```
open Microsoft.FSharp.Quotations
// A typed code quotation.
let expr : Expr<int> = <@ 1 + 1 @>
// An untyped code quotation.
let expr2 : Expr = <@@ 1 + 1 @@>
```

如果您不包含型別資訊，則遍歷大型運算式樹狀結構會比較快速。以具型別符號括住的運算式產生型別是 `Expr<'T>`，其中型別參數的運算式類型是由 F# 編譯器的型別推斷演算法所決定。當您使用不含類型資訊的程式碼引號時，引號運算式 *的類型為* 非泛型型別的運算式。您可以呼叫具類型類別上的 `Raw` 屬性 `Expr` 來取得不具類型的 `Expr` 物件。

有各種靜態方法，可讓您以程式設計方式在類別中產生 F# 運算式物件，`Expr` 而不需要使用加上引號的運算式。

請注意，程式碼引號必須包含完整的運算式。例如，針對系結 `let`，您需要系結名稱的定義，以及使用此系結的其他運算式。在詳細資訊語法中，這是緊接在關鍵字後面的運算式 `in`。在模組的最上層，這只是模組中的下一個運算式，但在引號中，它是明確需要的。

因此，下列運算式無效。

```
// Not valid:
// <@ let f x = x + 1 @>
```

但下列運算式是有效的。

```
// Valid:
<@ let f x = x + 10 in f 20 @>
// Valid:
<@
    let f x = x + 10
    f 20
@>
```

若要評估 F# 引號，您必須使用 *f# 引號評估* 工具。它可支援評估和執行 F# 運算式物件。

F# 引號也會保留類型條件約束資訊。請考慮下列範例：

```
open FSharp.Linq.RuntimeHelpers

let eval q = LeafExpressionConverter.EvaluateQuotation q

let inline negate x = -x
// val inline negate: x: ^a -> ^a when ^a : (static member ( ~- ) : ^a -> ^a)

<@ negate 1.0 @> |> eval
```

函數所產生的條件約束 `inline` 會保留在程式碼 quotation 中。`negate` 現在可以評估函數的 quoted 表單。

Expr 類型

型別的實例 `Expr` 代表 F # 運算式。`Expr` F # 程式庫檔中記載了泛型和非泛型型別。如需詳細資訊，請參閱 [fsharp.core](#)。引號命名空間 和 引號. `Expr` 類別。

接合運算子

接合可讓您將常值程式碼引號與您以程式設計方式或另一個程式碼引號建立的運算式結合。`% And` `%%` 運算子可讓您將 F # 運算式物件加入至程式碼引號中。您可以使用 `%` 運算子將具類型的運算式物件插入類型的引號中; 您可以使用 `%%` 運算子將不具類型的運算式物件插入不具類型的引號中。這兩個運算子都是一元前置運算子。因此，如果是類型的 `expr` 運算式 `Expr`，則下列程式碼是有效的。

```
<@@ 1 + %%expr @@>
```

如果 `expr` 是類型的類型引號 `Expr<int>`，則下列程式碼是有效的。

```
<@ 1 + %expr @>
```

範例

描述

下列範例說明如何使用程式碼引號將 F # 程式碼放入 expression 物件中，然後列印代表運算式的 F # 程式碼。定義的函 `println` 式包含遞迴函 `print` 式，該函式會 `Expr` 以易記格式顯示) 類型的 F # 運算式物件 (。

[Fsharp.core](#)中有數個使用中的模式，可以用來分析運算式的物件。這個範例不包含可能出現在 F # 運算式中的所有可能模式。任何無法辨識的模式都會觸發與萬用字元模式的相符 (`_`) 並且會使用方法來呈現，而此 `ToString` 方法會在型別上 `Expr` 讓您知道要加入至比對運算式的現用模式。

程式碼

```
module Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.Patterns
open Microsoft.FSharp.Quotations.DerivedPatterns

let println expr =
    let rec print expr =
        match expr with
        | Application(expr1, expr2) ->
            // Function application.
            print expr1
            printf " "
            print expr2
        | SpecificCall <@@ (+) @@> (_, _, exprList) ->
            // Matches a call to (+). Must appear before Call pattern.
            print exprList.Head
```

```

        printf " + "
        print exprList.Tail.Head
    | Call(exprOpt, methodInfo, exprList) ->
        // Method or module function call.
        match exprOpt with
        | Some expr -> print expr
        | None -> printf "%s" methodInfo.DeclaringType.Name
        printf ".%s(" methodInfo.Name
        if (exprList.IsEmpty) then printf ")" else
        print exprList.Head
        for expr in exprList.Tail do
            printf ","
            print expr
        printf ")"
    | Int32(n) ->
        printf "%d" n
    | Lambda(param, body) ->
        // Lambda expression.
        printf "fun (%s:%s) -> " param.Name (param.Type.ToString())
        print body
    | Let(var, expr1, expr2) ->
        // Let binding.
        if (var.IsMutable) then
            printf "let mutable %s = " var.Name
        else
            printf "let %s = " var.Name
        print expr1
        printf " in "
        print expr2
    | PropertyGet(_, propOrValInfo, _) ->
        printf "%s" propOrValInfo.Name
    | String(str) ->
        printf "%s" str
    | Value(value, typ) ->
        printf "%s" (value.ToString())
    | Var(var) ->
        printf "%s" var.Name
    | _ -> printf "%s" (expr.ToString())
print expr
printfn ""

let a = 2

// exprLambda has type "(int -> int)".
let exprLambda = <@ fun x -> x + 1 @>
// exprCall has type unit.
let exprCall = <@ a + 1 @>

println exprLambda
println exprCall
println <@@ let f x = x + 10 in f 10 @@>

```

輸出

```

fun (x:System.Int32) -> x + 1
a + 1
let f = fun (x:System.Int32) -> x + 10 in f 10

```

範例

描述

您也可以使用 [exprshape.rebuildshapecombination 模組](#) 中的三個作用中模式，以較少的現用模式來流覽運算式樹狀架構。當您想要遍歷樹狀結構，但在大部分的節點中都不需要所有資訊時，這些作用中的模式會很有用。當

您使用這些模式時，任何 F# 運算式都會符合下列三種模式的其中一種：`ShapeVar` 如果運算式為變數，則為，如果運算式是 `ShapeLambda` lambda 運算式，或 `ShapeCombination` 運算式為任何其他專案，則為。如果您使用先前程式碼範例中的作用中模式來遍歷運算式樹狀架構，則必須使用更多的模式來處理所有可能的 F# 運算式類型，而且您的程式碼會比較複雜。如需詳細資訊，請參閱 [exprshape.rebuildshapecombination](#)。
`ShapeVar|ShapeLambda|Shapecombination` 現用現用模式。

下列程式碼範例可作為更複雜周遊的基礎。在此程式碼中，會針對涉及函式呼叫的運算式建立運算式樹狀架構 `add`。 `SpecificCall`現用模式可用來偵測運算式樹狀架構中的任何呼叫 `add`。此現用模式會將呼叫的引數指派給 `exprList` 值。在此情況下，只有兩個，因此會提取這些函式，並在引數上以遞迴方式呼叫函式。結果會插入至程式碼引號，表示 `mul` 使用拼接運算子 `()` 的呼叫 `%%`。 `println` 上一個範例中的函式是用來顯示結果。

其他現用模式分支中的程式碼只會重新產生相同的運算式樹狀架構，所以產生的運算式中唯一的變更就是從變更 `add` 為 `mul`。

程式碼

```
module Module1
open Print
open Microsoft.FSharp.Quotations
open Microsoft.FSharp.Quotations.DerivedPatterns
open Microsoft.FSharp.Quotations.ExprShape

let add x y = x + y
let mul x y = x * y

let rec substituteExpr expression =
    match expression with
    | SpecificCall <@@ add @@> (_, _, exprList) ->
        let lhs = substituteExpr exprList.Head
        let rhs = substituteExpr exprList.Tail.Head
        <@@ mul %%lhs %%rhs @@>
    | ShapeVar var -> Expr.Var var
    | ShapeLambda (var, expr) -> Expr.Lambda (var, substituteExpr expr)
    | ShapeCombination(shapeComboObject, exprList) ->
        RebuildShapeCombination(shapeComboObject, List.map substituteExpr exprList)

let expr1 = <@@ 1 + (add 2 (add 3 4)) @@>
println expr1
let expr2 = substituteExpr expr1
println expr2
```

輸出

```
1 + Module1.add(2,Module1.add(3,4))
1 + Module1.mul(2,Module1.mul(3,4))
```

另請參閱

- [F# 語言參考](#)

Fixed 關鍵字

2021/3/5 • [Edit Online](#)

`fixed` 關鍵字可讓您將本機的「釘選」至堆疊，以防止在垃圾收集期間收集或移動它。它是用於低層級的程式設計案例。

語法

```
use ptr = fixed expression
```

備註

這會擴充運算式的語法，以允許將指標解壓縮，並將其系結至無法在垃圾收集期間收集或移動的名稱。

運算式的指標是透過關鍵詞系結至識別碼，藉此修正 `fixed` `use`。這種方式的語法類似於透過關鍵詞的資源管理 `use`。指標在範圍內是固定的，而且一旦超出範圍，就不會再修正。`fixed` 無法在系結的內容之外使用 `use`。您必須使用來系結名稱的指標 `use`。

使用 `fixed` 必須發生在函式或方法的運算式內。它不能用在腳本層級或模組層級的範圍。

就像所有指標程式碼一樣，這是一項不安全的功能，會在使用時發出警告。

範例

```
open Microsoft.FSharp.NativeInterop

type Point = { mutable X: int; mutable Y: int}

let squareWithPointer (p: nativeptr<int>) =
    // Dereference the pointer at the 0th address.
    let mutable value = NativePtr.get p 0

    // Perform some work
    value <- value * value

    // Set the value in the pointer at the 0th address.
    NativePtr.set p 0 value

let pnt = { X = 1; Y = 2 }
printfn $"pnt before - X: {pnt.X} Y: {pnt.Y}" // prints 1 and 2

// Note that the use of 'fixed' is inside a function.
// You cannot fix a pointer at a script-level or module-level scope.
let doPointerWork() =
    use ptr = fixed &pnt.Y

    // Square the Y value
    squareWithPointer ptr
    printfn $"pnt after - X: {pnt.X} Y: {pnt.Y}" // prints 1 and 4

doPointerWork()
```

另請參閱

- [NativePtr 模組](#)

Byrefs

2021/3/5 • [Edit Online](#)

F# 有兩個主要的功能區，可應付低層級程式設計的空間：

- `byref` / `inref` / `outref` 類型，也就是 managed 指標。它們具有使用方式的限制，因此您無法編譯在執行時間不正確程式。
- 類似的結構 `byref`，也就是具有類似語義和相同編譯時間限制的 [結構](#) `byref<'T>`。其中一個範例是 `Span<T>`。

語法

```
// Byref types as parameters
let f (x: byref<'T>) = ()
let g (x: inref<'T>) = ()
let h (x: outref<'T>) = ()

// Calling a function with a byref parameter
let mutable x = 3
f &x

// Declaring a byref-like struct
open System.Runtime.CompilerServices

[<Struct; IsByRefLike>]
type S(count1: int, count2: int) =
    member x.Count1 = count1
    member x.Count2 = count2
```

Byref、inref 和 outref

有三種形式 `byref`：

- `inref<'T>`，用來讀取基礎值的 managed 指標。
- `outref<'T>`，用於寫入基礎值的 managed 指標。
- `byref<'T>`，用來讀取和寫入基礎值的 managed 指標。

`byref<'T>` 可以傳遞至 `inref<'T>` 預期的。同樣地，您 `byref<'T>` 可以在預期的位置傳遞 `outref<'T>`。

使用 byref

若要使用 `inref<'T>`，您必須使用下列內容來取得指標值 `&`：

```
open System

let f (dt: inref<DateTime>) =
    printfn $"Now: {dt}"

let usage =
    let dt = DateTime.Now
    f &dt // Pass a pointer to 'dt'
```

若要使用或來寫入指標 `outref<'T>` `byref<'T>`，您也必須將抓取指標的值設為 `mutable`。

```
open System

let f (dt: byref<DateTime>) =
    printfn $"Now: %O{dt}"
    dt <- DateTime.Now

// Make 'dt' mutable
let mutable dt = DateTime.Now

// Now you can pass the pointer to 'dt'
f &dt
```

如果您只是要寫入指標，而不是讀取指標，請考慮使用 `outref<'T>` 而不是 `byref<'T>`。

Inref 語義

請考慮下列程式碼：

```
let f (x: inref<SomeStruct>) = x.SomeField
```

就語義而言，這表示下列各項：

- 指標的預留位置 `x` 只能用來讀取值。
- `struct` 針對內嵌于中之欄位所取得的任何指標 `SomeStruct` 都會提供型別 `inref<_>`。

以下也成立：

- 沒有任何暗示其他執行緒或別名沒有的寫入權限 `x`。
- 沒有任何隱含的 `SomeStruct` 改變，因為這是的 `x` `inref`。

但是，如果是不可變的 F# 實值型別，則會將 `this` 指標推斷為 `inref`。

所有這些規則都表示指標的持有者 `inref` 可能不會修改所指向之記憶體之立即內容。

Outref 語義

的目的 `outref<'T>` 是要指出指標只能寫入至。非預期的情況下，`outref<'T>` 允許讀取基礎值（儘管其名稱）。這是基於相容性的目的。語義上，`outref<'T>` 與不同 `byref<'T>`。

Interop 與 C#

`in ref` 除了傳回之外，C# 也支援和 `out ref` 關鍵字 `ref`。下表顯示 F# 如何解讀 C# 發出的內容：

C # 註	F # 註
<code>ref</code> 傳回值	<code>outref<'T></code>
<code>ref readonly</code> 傳回值	<code>inref<'T></code>
<code>in ref</code> 參數	<code>inref<'T></code>
<code>out ref</code> 參數	<code>outref<'T></code>

下表顯示 F# 發出的內容：

F # 註	註
<code>inref<'T></code> 引數	<code>[In]</code> 引數上的屬性

F # 文法	說明
<code>inref<'T></code> 返回	<code>modreq</code> 值上的屬性
<code>inref<'T></code> 在抽象插槽或執行中	<code>modreq</code> on 引數或 return
<code>outref<'T></code> 引數	<code>[Out]</code> 引數上的屬性

型別推斷和多載規則

`inref<'T>` 在下列情況下，F # 編譯器會推斷型別：

1. 具有屬性的 .NET 參數或傳回型別 `IsReadOnly` 。
2. `this` 結構類型上沒有可變動欄位的指標。
3. 衍生自另一個指標之記憶體位置的位址 `inref<_>` 。

當採用的隱含位址時 `inref`，具有類型之引數的多載 `SomeType` 優先於具有類型之引數的多載 `inref<SomeType>`。例如：

```
type C() =
    static member M(x: System.DateTime) = x.AddDays(1.0)
    static member M(x: inref<System.DateTime>) = x.AddDays(2.0)
    static member M2(x: System.DateTime, y: int) = x.AddDays(1.0)
    static member M2(x: inref<System.DateTime>, y: int) = x.AddDays(2.0)

let res = System.DateTime.Now
let v = C.M(res)
let v2 = C.M2(res, 4)
```

在這兩種情況下，`System.DateTime` 會解析接受的多載，而不是採用多載 `inref<System.DateTime>`。

類似 Byref 的結構

除了 `byref` / `inref` / `outref` 三個之外，您還可以定義自己的結構，這些結構可以遵循 `byref` 類似的語法。這是使用屬性來完成的 [IsByRefLikeAttribute](#)：

```
open System
open System.Runtime.CompilerServices

[<IsByRefLike; Struct>]
type S(count1: Span<int>, count2: Span<int>) =
    member x.Count1 = count1
    member x.Count2 = count2
```

`IsByRefLike` 不表示 `Struct`。兩者都必須存在於類型上。

`byref` F # 中的「-like」結構是堆疊系結的實值型別。它永遠不會在 managed 堆積上進行配置。`byref` 類似的結構適用於高效能程式設計，因為它會利用存留期和非捕捉的一組強大檢查來強制執行。這些規則包括：

- 它們可用來作為函式參數、方法參數、區域變數、方法傳回。
- 它們不得為類別或一般結構的靜態或實例成員。
- `async` (方法或 lambda 運算式時，無法由任何關閉結構來捕捉它們。
- 它們不能用來做為泛型參數。

最後一點對於 F # 管線樣式程式設計很重要，因為它 `|>` 是參數化其輸入類型的泛型函式。未來可能會放寬這 `|>` 項限制，因為它是內嵌的，而且不會對其主體中的非內嵌泛型函數進行任何呼叫。

雖然這些規則會嚴格限制使用方式，但它們會以安全的方式滿足高效能運算的承諾。

Byref 傳回

從 F# 函數或成員可以產生和取用的 Byref 傳回。使用傳回的 `byref` 方法時，會隱含地取值。例如：

```
let squareAndPrint (data : byref<int>) =
    let squared = data*data    // data is implicitly dereferenced
    printfn $"{d{squared}}"
```

若要傳回 byref，包含值的變數必須存留于目前範圍的時間。此外，若要傳回 byref，請使用 `&value` (，其中 value 是長度超過目前範圍) 的變數。

```
let mutable sum = 0
let safeSum (bytes: Span<byte>) =
    for i in 0 .. bytes.Length - 1 do
        sum <- sum + int bytes.[i]
    &sum // sum lives longer than the scope of this function.
```

若要避免隱含的取值，例如透過多個連結呼叫傳遞參考，請使用 `&x` (，其中 `x` 是) 的值。

您也可以直接指派給 return `byref`。請考慮下列 (高度命令式) 程式：

```
type C() =
    let mutable nums = [| 1; 3; 7; 15; 31; 63; 127; 255; 511; 1023 |]

    override _.ToString() = String.Join(' ', nums)

    member _.FindLargestSmallerThan(target: int) =
        let mutable ctr = nums.Length - 1

        while ctr > 0 && nums.[ctr] >= target do ctr <- ctr - 1

        if ctr > 0 then &nums.[ctr] else &nums.[0]

[<EntryPoint>]
let main argv =
    let c = C()
    printfn $"Original sequence: %O{c}"

    let v = &c.FindLargestSmallerThan 16

    v <- v*2 // Directly assign to the byref return

    printfn $"New sequence:      %O{c}"

    0 // return an integer exit code
```

此為輸出：

```
Original sequence: 1 3 7 15 31 63 127 255 511 1023
New sequence:      1 3 7 30 31 63 127 255 511 1023
```

Byref 的範圍

系結 `let` 值不能有其參考超出定義的範圍。例如，不允許下列情況：

```
let test2 () =  
  let x = 12  
  &x // Error: 'x' exceeds its defined scope!  
  
let test () =  
  let x =  
    let y = 1  
    &y // Error: `y` exceeds its defined scope!  
  ()
```

這可防止您取得不同的結果，這取決於您是否使用優化進行編譯。

參考儲存格

2019/10/23 • [Edit Online](#)

參考儲存格是儲存位置，可讓您使用參考語義建立可變值。

語法

```
ref expression
```

備註

您可以在值的前面使用 `ref` 運算子，建立可封裝值的新參考儲存格。由於基礎值是可變的，因此您接著可以變更這個基礎值。

參考儲存格不只是地址，還會保存實際值。當您使用 `ref` 運算子建立參考儲存格時，會建立基礎值的複本做為已封裝的可變值。

您可以使用 `!` (驚嘆號) 運算子來取值 (Dereference) 參考儲存格。

下列程式碼範例將示範參考儲存格的宣告和用法。

```
// Declare a reference.  
let refVar = ref 6  
  
// Change the value referred to by the reference.  
refVar := 50  
  
// Dereference by using the ! operator.  
printfn "%d" !refVar
```

輸出為 `50`。

參考儲存格為 `Ref` 泛型記錄型別的執行個體，其宣告如下。

```
type Ref<'a> =  
{ mutable contents: 'a }
```

`'a ref` 型別是 `Ref<'a>` 的同義字。IDE 中的編譯器和 IntelliSense 會對此型別顯示前者，但基礎定義則為後者。

`ref` 運算子會建立新的參考儲存格。下列程式碼為 `ref` 運算子的宣告。

```
let ref x = { contents = x }
```

下表顯示參考儲存格上的可用功能。

=====	“	“	“
<code>!</code> (取值運算子)	傳回基礎值。	<code>'a ref -> 'a</code>	<code>let (!) r = r.contents</code>

符號	說明	類型	範例
<code>:=</code> (指派運算子)	變更基礎值。	<code>'a ref -> 'a -> unit</code>	<pre>let (:=) r x = r.contents <- x</pre>
<code>ref</code> (運算子)	將值封裝至新的參考儲存格。	<code>'a -> 'a ref</code>	<pre>let ref x = { contents = x }</pre>
<code>Value</code> property	取得或設定基礎值。	<code>unit -> 'a</code>	<pre>member x.Value = x.contents</pre>
<code>contents</code> (記錄欄位)	取得或設定基礎值。	<code>'a</code>	<pre>let ref x = { contents = x }</pre>

有數個方式可以存取基礎值。取值運算子 (`!`) 傳回的值不是可指派的值。因此如果您要修改基礎值，則必須改用指派運算子 (`:=`)。

`Value` 屬性和 `contents` 欄位都是可指派的值。因此，您可以使用它們來存取或變更基礎值，如下列程式碼所示。

```
let xRef : int ref = ref 10

printfn "%d" (xRef.Value)
printfn "%d" (xRef.contents)

xRef.Value <- 11
printfn "%d" (xRef.Value)
xRef.contents <- 12
printfn "%d" (xRef.contents)
```

輸出如下。

```
10
10
11
12
```

`contents` 欄位是針對與其他 ML 版本相容而提供，而且會在編譯期間產生警告。若要停用這個警告，請使用 `--mlcompatibility` 編譯器選項。如需詳細資訊，請參閱[編譯器選項](#)。

C#程式設計人員應該 `ref` 知道C#在中的不是相同 `ref` 的F#東西。中F#的對等結構是[byref](#)，這是與參考儲存格不同的概念。

如果已關閉 `mutable`，則標示為的 `'a ref` 值可能會自動升級為，請參閱[values](#)。

另請參閱

- [F# 語言參考](#)
- [參數和引數](#)
- [符號和運算子參考](#)
- [值](#)

Nameof

2021/3/5 • [Edit Online](#)

運算式會產生比對來源中 `nameof` 幾乎任何 F # 結構之來源名稱的字串常數。

語法

```
nameof symbol
nameof<'TGeneric>
```

備註

`nameof` 的運作方式是解析傳遞給它的符號，並產生在原始程式碼中宣告的符號名稱。這在各種情況下都很有用，例如記錄，並可保護您的記錄免于原始程式碼中的變更。

```
let months =
    [
        "January"; "February"; "March"; "April";
        "May"; "June"; "July"; "August"; "September";
        "October"; "November"; "December"
    ]

let lookupMonth month =
    if (month > 12 || month < 1) then
        invalidArg (nameof month) ("Value passed in was %d{month}.")

    months.[month-1]

printfn "%s" (lookupMonth 12)
printfn "%s" (lookupMonth 1)
printfn "%s" (lookupMonth 13)
```

最後一行將擲回例外狀況，並 `"month"` 將在錯誤訊息中顯示。

您可以採用幾乎每個 F # 結構的名稱：

```
module M =
    let f x = nameof x

printfn $"{(M.f 12)}"
printfn $"{(nameof M)}"
printfn $"{(nameof M.f)}"
```

`nameof` 不是第一個類別的函式，因此無法使用。這表示它無法部分套用，而且無法透過 F # 管線運算子將值輸送至其中。

Nameof on 運算子

F # 中的運算子可以用兩種方式來使用，例如運算子文字本身或代表已編譯表單的符號。`nameof` 在運算子上，將會產生在來源中宣告之運算子的名稱。若要取得編譯的名稱，請在來源中使用編譯的名稱：

```
nameof(+) // "+"
nameof op_Addition // "op_Addition"
```

泛型上的 Nameof

您也可以採用泛型型別參數的名稱，但語法不同：

```
let f<'a> () = nameof<'a>
f() // "a"
```

`nameof<'TGeneric>` 將採用來源中所定義的符號名稱，而不是取代在呼叫位置的型別名稱。

語法不同的原因是要與其他 F # 內建運算子(例如和) `typeof<>` 對齊 `typedefof<>`。這使得 F # 與對泛型型別和來源中任何其他動作的運算子相對應。

模式比對中的 Nameof

此 `nameof` 模式可讓您 `nameof` 在模式比對運算式中使用，如下所示：

```
let f (str: string) =
    match str with
    | nameof str -> "It's 'str'!"
    | _ -> "It is not 'str'!"

f "str" // matches
f "asdf" // does not match
```

編譯器指示詞

2021/3/5 • [Edit Online](#)

本主題描述處理器指示詞和編譯器指示詞。

前置處理器指示詞

前置處理器指示詞的字首會加上 # 符號，且自身會出現在一行中。它是由前置處理器解譯，會在編譯器本身之前執行。

下表列出 F# 中可用的前置處理器指示詞。

'''	''
<code>#if</code> 符號	支援條件式編譯。如果已定義符號，則 <code>#if</code> 會包含後面區段 <i>symbol</i> 中的程式碼。符號也可以使用否定 <code>!</code> 。
<code>#else</code>	支援條件式編譯。若未定義與先前的 <code>#if</code> 搭配使用的符號，則標記要包含的程式碼區段。
<code>#endif</code>	支援條件式編譯。標示程式碼的條件式區段結尾。
<code>#</code> 行 <i>int</i> 、 <code>#</code> 行 <i>int</i> 字串、 <code>#</code> 行 <i>int</i> 逐字字串	表示原始的原始程式碼行和檔案名稱 (適用於偵錯)。這項功能提供用於產生 F# 原始程式碼的工具。
<code>#nowarn</code> <i>warningcode</i>	停用編譯器警告。若要停用警告，請從編譯器輸出中找出其號碼並包含在引號中。略過 "FS" 前置詞。若要停用同一行的多個警告號碼，請以引號括住每個號碼，並以一個空格分隔每個字串。例如：

```
#nowarn "9" "40"
```

停用警告的效果會套用到整個檔案，包括指示詞前面的檔案部分。|

條件式編譯指示詞

其中一個指示詞停用的程式碼在 Visual Studio Code 編輯器中會呈現暗灰色。

NOTE

條件式編譯指示詞的行為與其他語言的行為不同。例如，您無法使用包含符號的布林運算式，且 `true` 和 `false` 沒有特殊意義。在 `if` 指示詞中使用的符號必須透過命令列定義，或在專案設定中定義；沒有任何 `define` 前置處理器指示詞。

下列程式碼說明如何使用 `#if`、`#else` 和 `#endif` 指示詞。在此範例中，程式碼包含兩個版本的 `function1` 定義。當 `VERSION1` 使用 `-define` 編譯器選項定義時，會啟動指示詞與指示詞之間的程式碼 `#if` `#else`。否則會啟動 `#else` 與 `#endif` 之間的程式碼。

```

#if VERSION1
let function1 x y =
    printfn "x: %d y: %d" x y
    x + 2 * y
#else
let function1 x y =
    printfn "x: %d y: %d" x y
    x - 2*y
#endif

let result = function1 10 20

```

F# 中沒有 `#define` 前置處理器指示詞。您必須使用編譯器選項或專案設定來定義 `#if` 指示詞所使用的符號。

條件式編譯指示詞可以巢狀化。縮排對於前置處理器指示詞而言不重要。

您也可以使用來對符號進行否定 `!`。在此範例中，只有在 不進行偵錯工具時，才会有字串的值：

```

#if !DEBUG
let str = "Not debugging!"
#else
let str = "Debugging!"
#endif

```

行指示詞

建置時，編譯器會參考發生的每個錯誤所在的行號，以報告 F# 程式碼中的錯誤。這些行號從 1 開始，從檔案的第一行起算。不過，如果您正在從另一個工具產生 F# 原始程式碼，產生的程式碼中的行號通常不重要，因為產生的 F# 程式碼中的錯誤很可能來自其他來源。`#line` 指示詞為工具的作者提供一種方式，即產生 F# 原始程式碼，將有關原始行號和來源檔案的相關資訊傳遞給產生的 F# 程式碼。

使用 `#line` 指示詞時，檔案名稱必須括在引號內。除非逐字語彙基元 (`@`) 會出現在字串前面，否則必須逸出反斜線字元 (使用兩個反斜線字元而非一個)，才能在路徑中使用它們。下面是有效的行語彙基元。在這些範例中，假設透過工具執行原始檔案 `Script1` 時，會自動產生 F# 程式碼檔案，並會在 `Script1` 檔的第 25 行，從某些語彙基元中產生這些指示詞位置處的程式碼。

```

# 25
#line 25
#line 25 "C:\\Projects\\MyProject\\MyProject\\Script1"
#line 25 @"C:\Projects\MyProject\MyProject\Script1"
# 25 @"C:\Projects\MyProject\MyProject\Script1"

```

這些語彙基元指出在這個位置產生的 F# 程式碼，是衍生自 `Script1` 的 25 行或附近的某些建構。

編譯器指示詞

編譯器指示詞類似前置處理器指示詞，因為它們會加上 `#` 符號前置詞，但不是由前置處理器解譯，而是留給編譯器解譯及處理。

下表列出可在 F# 中使用的編譯器指示詞。

'''

''

'''	''
<code>#light [on "]" off "]</code>	啟用或停用輕量型語法，與其他 ML 版本相容。根據預設，會啟用輕量型語法。一律會啟用詳細語法。因此，您可以使用輕量型語法和詳細語法。指示詞 <code>#light</code> 本身就相當於 <code>#light "on"</code> 。如果您指定 <code>#light "off"</code> ，您必須針對所有語言建構使用詳細語法。會假設您使用輕量型語法，在文件中顯示 F# 語法。如需詳細資訊，請參閱 詳細資訊語法 。

如 (# A0) 指示詞的解譯器，請參閱 [使用 F # 的互動式程式設計](#)。

另請參閱

- [F # 語言參考](#)
- [編譯器選項](#)

編譯器選項

2021/3/17 • [Edit Online](#)

本主題說明 F# 編譯器 fsc.exe 的編譯器命令列選項。

您也可以藉由設定專案屬性來控制編譯環境。針對以 .NET Core 為目標的專案，中的「其他旗標」屬性

`<OtherFlags>...</OtherFlags>` `.fsproj` 會用來指定額外的命令列選項。

依字母順序排列的編譯器選項

下表顯示依字母順序列出的編譯器選項。某些 F# 編譯器選項類似於 c# 編譯器選項。如果是這種情況，則會提供 c# 編譯器選項主題的連結。

選項	說明
<code>-a filename.fs</code>	從指定的檔案產生文件庫。此選項是 <code>--target:library filename.fs</code> 的簡短形式。
<code>--baseaddress:address</code>	指定載入 DLL 時慣用的基底位址。 這個編譯器選項相當於相同名稱的 c# 編譯器選項。如需詳細資訊，請參閱 /baseaddress (C# 編譯器選項) 。
<code>--codepage:id</code>	如果必要的頁面不是系統目前的預設字碼頁，請指定在編譯期間要使用的字碼頁。 這個編譯器選項相當於相同名稱的 c# 編譯器選項。如需詳細資訊，請參閱 /程式字碼頁 (C# 編譯器選項) 。
<code>--consolecolors</code>	指定錯誤和警告在主控台上使用以色彩標示的文字。
<code>--crossoptimize[+ -]</code>	啟用或停用跨模組優化。
<code>--delaysign[+ -]</code>	只使用強式名稱金鑰的公開部分來延遲簽署元件。 這個編譯器選項相當於相同名稱的 c# 編譯器選項。如需詳細資訊，請參閱 /delaysign (C# 編譯器選項) 。
<code>--checked[+ -]</code>	啟用或停用產生溢位檢查。 這個編譯器選項相當於相同名稱的 c# 編譯器選項。如需詳細資訊，請參閱 (C# 編譯器選項)/核取 。
<code>--debug[+ -]</code> <code>-g[+ -]</code> <code>--debug:[full pdbonly]</code> <code>-g: [full pdbonly]</code>	啟用或停用產生的 debug 資訊，或指定要產生的 debug 資訊類型。預設值為 <code>full</code> ，可讓您附加至執行中的程式。選擇 <code>pdbonly</code> 取得儲存在 pdb (程式資料庫) 檔中的有限偵錯工具資訊。 相當於相同名稱的 c# 編譯器選項。如需相關資訊，請參閱 /debug (C# 編譯器選項) 。

選項	說明
<pre>--define:symbol</pre> <pre>-d:symbol</pre>	定義條件式編譯中使用的符號。
<pre>--deterministic[+ -]</pre>	產生具決定性的元件 (包括模組版本 GUID 和時間戳記)。此選項不能與萬用字元版本號碼搭配使用, 而且只支援內嵌和可移植的調試型別
<pre>--doc:xml doc-filename</pre>	<p>指示編譯器針對指定的檔案產生 XML 檔批註。如需詳細資訊, 請參閱 XML Documentation。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊, 請參閱 /doc (C# 編譯器選項)。</p>
<pre>--fullpaths</pre>	<p>指示編譯器產生完整路徑。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊, 請參閱 /fullpaths (C# 編譯器選項)。</p>
<pre>--help</pre> <pre>-?</pre>	顯示使用方式資訊, 包括所有編譯器選項的簡短描述。
<pre>--highentropyva[+ -]</pre>	啟用或停用高熵位址空間配置隨機載入 (ASLR) (增強的安全性功能)。OS 會隨機化記憶體中的位置, 其中會載入應用程式 (的基礎結構, 例如堆疊和堆積)。如果您啟用此選項, 作業系統可以使用此隨機載入, 在 64 位電腦上使用完整的 64 位位址空間。
<pre>--keycontainer:key-container-name</pre>	指定強式名稱金鑰容器。
<pre>--keyfile:filename</pre>	指定用於簽署所產生元件的公開金鑰檔案名。
<pre>--lib:folder-name</pre> <pre>-I:folder-name</pre>	<p>指定要搜尋參考之元件的目錄。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊, 請參閱 /lib (C# 編譯器選項)。</p>
<pre>--linkresource:resource-info</pre>	<p>將指定的資源連結至元件。資源資訊的格式為 <code>filename[name[public private]]</code></p> <p>使用這個選項連結單一資源, 是使用選項內嵌整個資源檔的替代方案 <code>--resource</code>。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊, 請參閱 /linkresource (C# 編譯器選項)。</p>
<pre>--mlcompatibility</pre>	當您使用設計來與其他 ML 版本相容的功能時, 會忽略出現的警告。
<pre>--noframework</pre>	停用 .NET Framework 元件的預設參考。
<pre>--nointerfacedata</pre>	指示編譯器省略它通常會加入至包含 F # 專屬中繼資料之元件的資源。

名稱	說明
<code>--nologo</code>	啟動編譯器時，不會顯示橫幅文字。
<code>--nooptimizationdata</code>	指示編譯器只包含執行內嵌結構的優化。禁止跨模組內嵌，但改善二進位檔相容性。
<code>--nowin32manifest</code>	指示編譯器省略預設的 Win32 資訊清單。
<code>--nowarn:warning-number-list</code>	<p>停用依數位列出的特定警告。以逗號分隔每個警告編號。您可以從編譯輸出中探索任何警告的警告編號。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊，請參閱 /nowarn (C# 編譯器選項)。</p>
<code>--optimize[+ -][optimization-option-list]</code> <code>-O[+ -] [optimization-option-list]</code>	<p>啟用或停用優化。您可以藉由列出某些優化選項，以選擇性地加以停用或啟用。這些是：<code>nojitoptimize</code>、<code>nojittracking</code>、<code>nolocaloptimize</code>、<code>nocrossoptimize</code>、<code>notailcalls</code>。</p>
<code>--out:output-filename</code> <code>-o:output-filename</code>	<p>指定已編譯的元件或模組的名稱。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊，請參閱 /(C# 編譯器選項)。</p>
<code>--pathmap:path=sourcePath,...</code>	<p>指定如何將實體路徑對應到編譯器所輸出的來源路徑名稱。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊，請參閱 /pathmap (C# 編譯器選項)。</p>
<code>--pdb:pdb-filename</code>	<p>將輸出偵錯工具的 (命名為程式資料庫) 檔。此選項只適用於 <code>--debug</code> 也啟用時。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊，請參閱 /pdb (C# 編譯器選項)。</p>
<code>--platform:platform-name</code>	<p>指定產生的程式碼只會在指定的平臺上執行 (<code>x86</code>、<code>Itanium</code> 或 <code>x64</code>)，或者，如果選擇了平臺名稱 <code>anycpu</code>，則會指定產生的程式碼可以在任何平臺上執行。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊，請參閱 /平臺 (C# 編譯器選項)。</p>
<code>--preferreduilang:lang</code>	指定慣用的輸出語言文化特性名稱 (例如， <code>es-ES</code> <code>ja-JP</code>)。
<code>--quotations-debug</code>	指定應針對衍生自 F # 引號常值和反映定義的運算式發出額外的偵錯工具資訊。偵錯工具資訊會加入至 F # 運算式樹狀結構節點的自訂屬性。請參閱程式 代碼引號 和 CustomAttributes 。
<code>--reference:assembly-filename</code> <code>-r:assembly-filename</code>	<p>讓 F # 或 .NET Framework 元件中的程式碼可供所要編譯的程式碼使用。</p> <p>這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊，請參閱 /參考 (C# 編譯器選項)。</p>

名稱	說明
<code>--resource:resource-filename</code>	<p>將 managed 資源檔內嵌到產生的元件中。</p> <p>這個編譯器選項相當於相同名稱的 c# 編譯器選項。如需詳細資訊, 請參閱 /資源 (C# 編譯器選項)。</p>
<code>--sig:signature-filename</code>	<p>根據產生的元件產生簽名檔。如需有關簽名檔的詳細資訊, 請參閱簽章。</p>
<code>--simpleresolution</code>	<p>指定應該使用以目錄為基礎的 Mono 規則來解析元件參考, 而不是使用 MSBuild 解析。預設值是使用 MSBuild 解析 (除了在 Mono 下執行時)。</p>
<code>--standalone</code>	<p>指定要產生包含其所有相依性的元件, 使其本身執行, 而不需要其他元件, 例如 F# 程式庫。</p>
<code>--staticlink:assembly-name</code>	<p>以靜態方式連結指定的元件, 以及相依于此元件的所有參考 Dll。使用元件名稱, 而不是 DLL 名稱。</p>
<code>--subsystemversion</code>	<p>指定所產生之可執行檔所要使用的 OS 子系統版本。使用 6.02 適用於 Windows 8.1, 6.01 適用於 windows 7, 6.00 適用於 Windows Vista。此選項只適用於可執行檔, 而不是 Dll, 而且只有在您的應用程式相依于特定作業系統版本上可用的特定安全性功能時, 才需要使用。如果使用此選項, 且使用者嘗試在較低版本的作業系統上執行您的應用程式, 它將會失敗並出現錯誤訊息。</p>
<code>--tailcalls[+ -]</code>	<p>啟用或停用 tail IL 指令, 這會導致堆疊框架重複用於 tail 遞迴函式。這個選項預設為啟用。</p>
<code>--target:[exe winexe library module] filename</code>	<p>指定所產生之已編譯器代碼的類型和檔案名。</p> <ul style="list-style-type: none"> <code>exe</code> 表示主控台應用程式。 <code>winexe</code> 表示 Windows 應用程式與主控台應用程式不同, 因為它沒有標準的輸入/輸出資料流程 (stdin、stdout 和 stderr) 已定義。 <code>library</code> 是沒有進入點的元件。 <code>module</code> 是 .NET Framework 模組 (.netmodule), 稍後可以與其他模組結合為元件。 <p>這個編譯器選項相當於相同名稱的 c# 編譯器選項。如需詳細資訊, 請參閱 /目標 (C# 編譯器選項)。</p>
<code>--times</code>	<p>顯示編譯的計時資訊。</p>
<code>--utf8output</code>	<p>啟用 UTF-8 編碼的列印編譯器輸出。</p>
<code>--warn:warning-level</code>	<p>設定警告層級 (0 到 5)。預設層級為3。每個警告都會根據其嚴重性獲得等級。層級5可提供比層級1更多但較不嚴重的警告。</p> <p>層級5警告為: 21 (在執行時間檢查遞迴使用)、22 (依 <code>let rec</code> 順序評估)、45 (完整抽象) 和 52 (防禦性複製)。所有其他警告都是層級2。</p> <p>這個編譯器選項相當於相同名稱的 c# 編譯器選項。如需詳細資訊, 請參閱 /警告 (C# 編譯器選項)。</p>

語法	說明
<code>--warnon:warning-number-list</code>	啟用可能預設關閉或由另一個命令列選項停用的特定警告。 1182 (未使用的變數) 警告預設為關閉。
<code>--warnaserror[+ -] [warning-number-list]</code>	啟用或停用將警告報告為錯誤的選項。您可以提供要停用或啟用的特定警告編號。稍後在命令列中的選項會覆寫先前命令列中的選項。例如, 若要指定您不想回報為錯誤的警告, 請指定 <code>--warnaserror+</code> <code>--warnaserror-:warning-number-list</code> 。 這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊, 請參閱 /warnaserror (C# 編譯器選項) 。
<code>--win32manifest:manifest-filename</code>	將 Win32 資訊清單檔加入至編譯。這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊, 請參閱 /win32manifest (C# 編譯器選項) 。
<code>--win32res:resource-filename</code>	將 Win32 資源檔新增至編譯。 這個編譯器選項相當於相同名稱的 c # 編譯器選項。如需詳細資訊, 請參閱 /win32res ((C#) 編譯器選項) 。

相關文章

主題	說明
F# Interactive 選項	描述 F # 解釋器支援的命令列選項 fsi.exe。
專案屬性參考	描述專案的 UI, 包括提供組建選項的專案屬性頁。

F# 互動選項

2020/11/2 • [Edit Online](#)

本文說明 F# 互動所支援的命令列選項 `fsi.exe`。F# 互動接受許多與 F# 編譯器相同的命令列選項，但也接受一些額外的選項。

使用 F# 互動編寫腳本

F# 互動，`dotnet fsi` 可以透過互動方式啟動，也可以從命令列啟動來執行腳本。命令列語法為




```
dotnet fsi [options] [ script-file [arguments] ]
```

F# 腳本檔案的副檔名為 `.fsx`。

F# 互動選項的表格

下表摘要說明 F# 互動所支援的選項。您可以在命令列上或透過 Visual Studio IDE 來設定這些選項。若要在 Visual Studio IDE 中設定這些選項，請開啟 [工具] 功能表，選取 [選項]，展開 [F# 工具] 節點，然後選取 [F# 互動]。

清單元素會以分號分隔 F# 互動選項引數，(;)。

--	用來指示 F# 互動將其餘引數視為 F# 程式或腳本的命令列引數。您可以使用 <code>Fsi > my.application.commandlineargs</code> ，在程式碼中存取這些引數。
--checked[+ -]	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。
--  : < int>	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。
--consolecolors[+ -]	以色彩輸出警告和錯誤訊息。
--crossoptimize[+ -]	啟用或停用跨模組優化。
--debug[+ -] --debug: [full pdbonly  embedded] -g[+ -] -g: [full pdbonly  embedded]	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。
--define: <  >	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。
--  [+ -]	產生具決定性的元件 (包括模組版本 GUID 和時間戳記)。

ff	ff
--exec	指示 F # interactive 在載入檔案或執行命令列上指定的腳本檔案後結束。
--fullpaths	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--gui[+ -]	啟用或停用 Windows Forms 事件迴圈。預設值為 [已啟用]。
--■ -?	用來顯示命令列語法及每個選項的簡短描述。
--lib: < folder ■> -l: < ■■■>	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--load: < filename>	在啟動時編譯指定的原始程式碼, 並將編譯的 F # 結構載入至會話。
--mlcompatibility	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--noframework	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--nologo	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--nowarn: < ■-■>	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--optimize[+ -]	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--preferreduilang: < lang>	指定慣用的輸出語言文化特性名稱 (例如, es、ja-jp) 。
--quiet	隱藏 F# 互動的輸出至 stdout 資料流程。
--■-debug	指定應針對衍生自 F # 引號常值和反映定義的運算式發出額外的偵錯工具資訊。偵錯工具資訊會加入至 F # 運算式樹狀結構節點的自訂屬性。請參閱程式 代碼引號 和 CustomAttributes 。
--readline[+ -]	啟用或停用互動式模式的 tab 鍵自動完成。
--reference: < filename> -r: < filename>	與 fsc.exe 編譯器選項相同。如需詳細資訊, 請參閱 編譯器選項 。
--tailcalls[+ -]	啟用或停用 tail IL 指令, 這會導致堆疊框架重複用於 tail 遞迴函式。這個選項預設為啟用。

“	“
<code>--targetprofile: < ■></code>	指定此元件的目標 framework 設定檔。有效值為 <code>mscorlib</code> 、 <code>netcore</code> 或 <code>netstandard</code> 。預設為 <code>mscorlib</code> 。
<code>--use: < filename></code>	告知解譯器在啟動時使用指定的檔案做為初始輸入。
<code>--utf8output</code>	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。
<code>--■: < ■■></code>	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。
<code>--warnaserror[+ -]</code>	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。
<code>--warnaserror[+ -]:** < int-list > **</code>	與 <code>fsc.exe</code> 編譯器選項相同。如需詳細資訊，請參閱 編譯器選項 。

F# 互動結構化列印

F# 互動 (`dotnet fsi`) 會使用延伸版本的 [結構化純文字格式](#) 來報告值。

1. `%A` 支援純文字格式的所有功能，而有些則是可自訂的。
2. 如果輸出主控台支援色彩，則會以色彩標示列印。
3. 除非您明確評估該字串，否則會將限制放在顯示的字串長度。
4. 您可以透過物件取得一組使用者可定義的設定 `fsi`。

針對報告值自訂純文字列印的可用設定如下：

```
open System.Globalization

fsi.FormatProvider <- CultureInfo("de-DE") // control the default culture for primitives

fsi.PrintWidth <- 120           // Control the width used for structured printing

fsi.PrintDepth <- 10           // Control the maximum depth of nested printing

fsi.PrintLength <- 10          // Control the length of lists and arrays

fsi.PrintSize <- 100           // Control the maximum overall object count

fsi.ShowProperties <- false    // Control whether properties of .NET objects are shown by default

fsi.ShowIEnumerable <- false  // Control whether sequence values are expanded by default

fsi.ShowDeclarationValues <- false // Control whether values are shown for declaration outputs
```

使用和自訂 `AddPrinter` ``AddPrintTransformer`

您可以使用和自訂 F# 互動輸出中的列印 `fsi.AddPrinter` `fsi.AddPrintTransformer`。第一個函數會提供文字來取代物件的列印。第二個函式會傳回要改為顯示的代理物件。例如，請考慮下列 F# 程式碼：

```
open System

fsi.AddPrinter<DateTime>(fun dt -> dt.ToString("s"))

type DateAndLabel =
    { Date: DateTime
      Label: string }

let newYearsDay1999 =
    { Date = DateTime(1999, 1, 1)
      Label = "New Year" }
```

如果您在 F# 互動中執行此範例，它會根據格式化選項組進行輸出。在此情況下，它會影響日期和時間的格式：

```
type DateAndLabel =
    { Date: DateTime
      Label: string }
val newYearsDay1999 : DateAndLabel = { Date = 1999-01-01T00:00:00
                                         Label = "New Year" }
```

`fsi.AddPrintTransformer` 可以用來提供要列印的代理物件：

```
type MyList(values: int list) =
    member _.Values = values

fsi.AddPrintTransformer(fun (x:MyList) -> box x.Values)

let x = MyList([1..10])
```

輸出如下：

```
val x : MyList = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

如果傳遞的轉換器函式 `fsi.AddPrintTransformer` 傳回 `null`，則會忽略列印轉換器。這可以用來以型別開頭，以篩選任何輸入值 `obj`。例如：

```
fsi.AddPrintTransformer(fun (x:obj) ->
    match x with
    | :? string as s when s = "beep" -> box ["quack"; "quack"; "quack"]
    | _ -> null)

let y = "beep"
```

輸出如下：

```
val y : string = ["quack"; "quack"; "quack"]
```

相關主題

“	“
編譯器選項	描述 F# 編譯器可用的命令列選項， <code>fsc.exe</code> 。

原始碼程式行、檔案與路徑識別項

2019/10/23 • [Edit Online](#)

識別碼 `__LINE__` `__SOURCE_DIRECTORY__` 和是內建值，可讓您存取程式碼中的原始程式列號、目錄和檔案名。
`__SOURCE_FILE__`

語法

```
__LINE__  
__SOURCE_DIRECTORY__  
__SOURCE_FILE__
```

備註

這些值的每一個都 `string` 具有類型。

下表摘要說明中F#可用的原始程式列、檔案和路徑識別碼。這些識別碼不是預處理器宏;這些是編譯器可辨識的內建值。

識別碼	說明
<code>__LINE__</code>	評估為目前的行號，並 <code>#line</code> 考慮指示詞。
<code>__SOURCE_DIRECTORY__</code>	評估為來原始目錄的目前完整路徑，並考慮 <code>#line</code> 指示詞。
<code>__SOURCE_FILE__</code>	評估為目前的來原始檔案名，不含其路徑，考慮 <code>#line</code> 指示詞。

如需指示詞的 `#line` 詳細資訊，請參閱[編譯器指示詞](#)。

範例

下列程式碼範例示範如何使用這些值。

```
let printSourceLocation() =  
    printfn "Line: %s" __LINE__  
    printfn "Source Directory: %s" __SOURCE_DIRECTORY__  
    printfn "Source File: %s" __SOURCE_FILE__  
printSourceLocation()
```

輸出：

```
Line: 4  
Source Directory: C:\Users\username\Documents\Visual Studio 2017\Projects\SourceInfo\SourceInfo  
Source File: Program.fs
```

另請參閱

- [編譯器指示詞](#)

- [F# 語言參考](#)

呼叫者資訊

2021/3/5 • [Edit Online](#)

使用 Caller Info 屬性，您就可以取得有關方法之呼叫端的資訊。您可以取得原始程式碼的檔案路徑、原始程式碼中的行號，以及呼叫端的成員名稱。這項資訊有助於追蹤、偵錯及建立診斷工具。

若要取得這項資訊，可使用套用至選擇性參數的屬性，每個屬性都有預設值。下表列出 `CompilerServices` 命名空間中定義的呼叫端資訊屬性：

名稱	說明	預設值
<code>CallerFilePath</code>	包含呼叫端的原始程式檔完整路徑。這是編譯時間的檔案路徑。	<code>String</code>
<code>CallerLineNumber</code>	原始程式檔中呼叫方法所在的行號。	<code>Integer</code>
<code>CallerMemberName</code>	呼叫端的方法或屬性名稱。請參閱本主題稍後的「成員名稱」一節。	<code>String</code>

範例

下列範例顯示如何使用這些屬性來追蹤呼叫者。

```
open System.Diagnostics
open System.Runtime.CompilerServices
open System.Runtime.InteropServices

type Tracer() =
    member _.DoTrace(message: string,
        [CallerMemberName; Optional; DefaultParameterValue("")] memberName: string,
        [CallerFilePath; Optional; DefaultParameterValue("")] path: string,
        [CallerLineNumber; Optional; DefaultParameterValue(0)] line: int) =
        Trace.WriteLine(sprintf $"Message: {message}")
        Trace.WriteLine(sprintf $"Member name: {memberName}")
        Trace.WriteLine(sprintf $"Source file path: {path}")
        Trace.WriteLine(sprintf $"Source line number: {line}")
```

備註

呼叫端資訊屬性只能套用至選擇性參數。呼叫端資訊屬性會讓編譯器針對每個以呼叫端資訊屬性裝飾的選擇性參數，寫入適當的值。

在編譯時期，Caller Info 的值會做為常值發出至中繼語言 (IL)。不同于例外狀況之 `StackTrace` 屬性的結果，結果不會受到模糊化的影響。

您可以明確提供選擇性引數來控制呼叫端資訊，或是隱藏呼叫端資訊。

成員名稱

您可以使用 `CallerMemberName` 屬性來避免將成員名稱指定為 `String` 所呼叫方法的引數。藉由使用這項技術，您可以避免重新命名重構不會變更值的問題 `String`。這項優點對於下列工作特別有用：

- 使用追蹤和診斷常式。

- 在系結資料時執行 `INotifyPropertyChanged` 介面。這個介面可讓物件的屬性告知繫結的控制項屬性已變更，所以控制項可以顯示更新的資訊。如果沒有 `CallerMemberName` 屬性，您必須將屬性名稱指定為常值。

下圖顯示當您使用 `CallerMemberName` 屬性時所傳回的成員名稱。

「'''」	「''''''」
方法、屬性或事件	產生呼叫的方法、屬性或事件的名稱。
建構函式	字串 ".ctor"
靜態建構函式	字串 ".cctor"
解構函式	字串 "Finalize"
使用者定義的運算子或轉換	產生的成員名稱，例如 "op_Addition"。
屬性建構函式	套用屬性的成員名稱。如果屬性為成員內的任何項目 (例如參數、傳回值或泛型類型參數)，這個結果會是與該項目相關聯的成員名稱。
無包含的成員 (例如，組件層級或套用至類型的屬性)。	選擇性參數的預設值。

另請參閱

- [屬性](#)
- [具名引數](#)
- [選擇性參數](#)

詳細語法

2021/3/5 • [Edit Online](#)

有兩種形式的語法可用於 F # 語言：*詳細語法* 和 *輕量語法* 中的許多結構。詳細資訊語法並不常用，但其優點在於縮排較不敏感。輕量語法較短，而且會使用縮排來表示結構的開頭和結尾，而不是如、`、` 等等的其他關鍵字

`begin` `end` `in`。預設語法是輕量語法。本主題說明未啟用輕量語法時 F # 結構的語法。詳細語法一律為啟用狀態，因此即使您啟用輕量語法，還是可以針對某些結構使用詳細語法。您可以使用指示詞來停用輕量語法

`#light "off"`。

結構表

下表顯示在這兩種形式之間有差異之 F # 語言結構的輕量和詳細語法。在此表格中，角括弧 (< >) 括住使用者提供的語法元素。如需有關這些結構中所使用之語法的詳細資訊，請參閱每個語言結構的檔。

輕量語法	詳細語法	輕量語法
複合運算式	<pre><expression1> <expression2></pre>	<pre><expression1>; <expression2></pre>
嵌套 <code>let</code> 系統	<pre>let f x = let a = 1 let b = 2 x + a + b</pre>	<pre>let f x = let a = 1 in let b = 2 in x + a + b</pre>
程式碼區塊	<pre>(<expression1> <expression2>)</pre>	<pre>begin <expression1>; <expression2>; end</pre>
<code>`for...do`</code>	<pre>for counter = start to finish do ...</pre>	<pre>for counter = start to finish do ... done</pre>
<code>`while...do`</code>	<pre>while <condition> do ...</pre>	<pre>while <condition> do ... done</pre>

`for...in`	<pre> for var in start .. finish do ... </pre>	<pre> for var in start .. finish do ... done </pre>
`do`	<pre> do ... </pre>	<pre> do ... in </pre>
記錄 (record)	<pre> type <record-name> = { <field-declarations> } <value-or-member- definitions> </pre>	<pre> type <record-name> = { <field-declarations> } with <value-or-member- definitions> end </pre>
Class - 類別	<pre> type <class-name>(<params>) = ... </pre>	<pre> type <class-name>(<params>) = class ... end </pre>
structure	<pre> [<StructAttribute>] type <structure-name> = ... </pre>	<pre> type <structure-name> = struct ... end </pre>
區分聯集	<pre> type <union-name> = <value-or-member definitions> </pre>	<pre> type <union-name> = with <value-or-member- definitions> end </pre>

interface	<pre>type <interface-name> = ...</pre>	<pre>type <interface-name> = interface ... end</pre>
物件運算式	<pre>{ new <type-name> with <value-or-member- definitions> <interface- implementations> }</pre>	<pre>{ new <type-name> with <value-or-member- definitions> end <interface- implementations> }</pre>
介面實作	<pre>interface <interface-name> with <value-or-member- definitions></pre>	<pre>interface <interface-name> with <value-or-member- definitions> end</pre>
類型延伸模組	<pre>type <type-name> with <value-or-member- definitions></pre>	<pre>type <type-name> with <value-or-member- definitions> end</pre>
name	<pre>module <module-name> = ...</pre>	<pre>module <module-name> = begin ... end</pre>

另請參閱

- [F # 語言參考](#)
- [編譯器指示詞](#)
- [程式碼格式化方針](#)

F # 編譯器訊息

2021/3/12 • [Edit Online](#)

本節將詳細說明 F # 編譯器會針對某些結構發出的編譯器錯誤和警告。預設的錯誤集可透過下列方式變更：

- 使用編譯器選項將特定警告視為錯誤 `-warnaserror+` ,
- 使用編譯器選項忽略特定警告 `-nowarn`

如果此區段中尚未記錄特定的警告或錯誤：

- 移至此頁面的結尾，並傳送包含錯誤數目或文字的意見反應，或
- 遵循 [create-new-fsharp-compiler-message](#) 中的指示，並開啟此存放庫的提取要求，以自行新增。

另請參閱

- [F # 編譯器選項](#)

使用 F 的互動式程式設計#

2021/3/12 • [Edit Online](#)

F # Interactive (dotnet fsi) 可用來以互動方式在主控台執行 F # 程式碼, 或執行 F # 腳本。換句話說, F# Interactive 會執行 F# 語言的 REPL (讀取、評估、列印迴圈)。

若要從主控台執行 F # Interactive, 請執行 `dotnet fsi`。您將可以 `dotnet fsi` 在任何 .NET SDK 中找到。

如需可用命令列選項的詳細資訊, 請參閱 [F # 互動式選項](#)。

直接在 F # Interactive 中執行程式碼

因為 F # Interactive 是複寫 (讀取-eval-列印迴圈), 所以您可以在其中以互動方式執行程式碼。以下是從命令列執行之後的互動式會話範例 `dotnet fsi` :

```
Microsoft (R) F# Interactive version 11.0.0.0 for F# 5.0
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> let square x = x * x;;
val square : x:int -> int

> square 12;;
val it : int = 144

> printfn "Hello, FSI!"
- ;;
Hello, FSI!
val it : unit = ()
```

您會看到兩個主要專案:

1. 所有程式碼都必須以雙分號結束, (`;;` 要評估)
2. 程式碼會進行評估並儲存在 `it` 值中。您可以 `it` 互動方式參考。

F # Interactive 也支援多行輸入。您只需要以雙分號 (的) 來終止提交 `;;`。請考慮下列在 F # Interactive 中貼上和評估的程式碼片段:

```
> let getOddSquares xs =
-   xs
-   |> List.filter (fun x -> x % 2 <> 0)
-   |> List.map (fun x -> x * x)
-
-   printfn "%A" (getOddSquares [1..10]);;
[1; 9; 25; 49; 81]
val getOddSquares : xs:int list -> int list
val it : unit = ()

>
```

程式碼的格式會保留下來, 且 (`;;`) 終止輸入的雙重分號。F # Interactive 接著會評估程式碼並印出結果!

使用 F 編寫腳本#

以互動方式在 F # Interactive 中評估程式碼可能是很棒的學習工具，但是您很快就會發現，在一般編輯器中撰寫程式碼並不是那麼有效率。若要支援一般程式碼編輯，您可以撰寫 F # 腳本。

腳本會使用 .fsx 副檔名。您可以直接執行 `dotnet fsi` 並指定 f # 原始程式碼的指令檔名，而 f # interactive 會即時讀取程式碼並執行它，而不是編譯原始程式碼，然後稍後再執行已編譯的元件。例如，請考慮下列稱為的腳本

`Script.fsx`：

```
let getOddSquares xs =
    xs
    |> List.filter (fun x -> x % 2 <> 0)
    |> List.map (fun x -> x * x)

printfn "%A" (getOddSquares [1..10])
```

當您在電腦上建立這個檔案時，您可以使用來執行它，`dotnet fsi` 並直接在終端機視窗中查看輸出：

```
dotnet fsi Script.fsx
[1; 9; 25; 49; 81]
```

[Visual studio](#)、[visual Studio Code](#)和[visual studio for Mac](#)原生支援 F # 腳本處理。

以 F # Interactive 參考封裝

NOTE

封裝管理系統可延伸，請進一步閱讀 [其他擴充功能](#)。

F # Interactive 支援使用 `#r "nuget:"` 語法和選擇性版本參考 NuGet 套件：

```
#r "nuget: Newtonsoft.Json"
open Newtonsoft.Json

let data = { | Name = "Don Syme"; Occupation = "F# Creator" | }
JsonConvert.SerializeObject(data)
```

如果未指定版本，則會採用最高可用的非預覽套件。若要參考特定版本，請透過逗點引入版本。在參考套件的預覽版本時，這會很方便。例如，請考慮使用 [DiffSharp](#)預覽版本的這個腳本：

```
#r "nuget: DiffSharp-lite, 1.0.0-preview-328097867"
open DiffSharp

// A 1D tensor
let t1 = dsharp.tensor [ 0.0 .. 0.2 .. 1.0 ]

// A 2x2 tensor
let t2 = dsharp.tensor [ [ 0; 1 ]; [ 2; 2 ] ]

// Define a scalar-to-scalar function
let f (x: Tensor) = sin (sqrt x)

printfn $"{f (dsharp.tensor 1.2)}"
```

指定套件來源

您也可以使用命令來指定套件來源 `#i`。下列範例會指定遠端和本機來源：

```
#i "nuget:https://my-remote-package-source/index.json"
#i @"path-to-my-local-source"
```

這會告訴解析引擎也要考慮將遠端和/或本機來源新增至腳本。

您可以在腳本中指定任意數量的封裝參考。

NOTE

(使用架構參考的腳本目前有一項限制, 例如 `Microsoft.NET.Sdk.Web` 或 `Microsoft.NET.Sdk.WindowsDesktop`)。無法使用 Saturn、Giraffe、WinForms 等套件。這是在問題 [#9417](#) 中追蹤。

如需詳細資訊, 請參閱 [封裝管理擴充性和其他擴充功能](#)。

使用 F # interactive 參考磁片上的元件

或者, 如果您在磁片上有一個元件, 而且想要在腳本中參考該元件, 則可以使用 `#r` 語法來指定元件。在編譯為的專案中, 請考慮下列程式碼 `MyAssembly.dll` :

```
// MyAssembly.fs
module MyAssembly
let myFunction x y = x + 2 * y
```

其中一個已編譯, 您可以在名為的檔案中參考它, 如下所示 `Script.fsx` :

```
#r "path/to/MyAssembly.dll"

printfn $"{MyAssembly.myFunction 10 40}"
```

輸出如下所示:

```
dotnet fsi Script.fsx
90
```

您可以在腳本中指定任意數量的元件參考。

載入其他腳本

編寫腳本時, 針對不同工作使用不同的腳本通常會很有說明。有時候您可能會想要在另一個腳本中重複使用程式碼。您可以直接使用載入和評估其內容, 而不是將其內容複寫到您的檔案中 `#load` 。

請考慮下列 `Script1.fsx` 事項:

```
let square x = x * x
```

以及使用中的檔案 `Script2.fsx` :

```
#load "Script1.fsx"
open Script1

printfn $"{square 12}"
```

請注意，宣告 `open Script1` 是必要的。這是因為 F # 腳本中的結構會編譯成最上層模組，也就是它所在的指令檔名。

您可以評估 `Script2.fsx` 如下：

```
dotnet fsi Script2.fsx
144
```

您可以 `#load` 在腳本中指定所需的指示詞數目。

使用 `fsi` F # 程式碼中的物件

F # 腳本可存取 `fsi` 代表 F # 互動式會話的自訂物件。它可讓您自訂輸出格式等專案。它也是您可以存取命令列引數的方式。

下列範例顯示如何取得和使用命令列引數：

```
let args = fsi.CommandLineArgs

for arg in args do
    printfn $"{arg}"
```

進行評估時，會列印所有引數。第一個引數一律是所評估腳本的名稱：

```
dotnet fsi Script1.fsx hello world from fsi
Script1.fsx
hello
world
from
fsi
```

您也可以使用 `System.Environment.GetCommandLineArgs()` 來存取相同的引數。

F # Interactive 指示詞參考

`#r` 先前看到的和指示詞 `#load` 只適用於 F # Interactive。F # Interactive 中僅提供數個指示詞：

<code>!!!</code>	DESCRIPTION
<code>#r "nuget:..."</code>	從 NuGet 參考封裝
<code>#r "assembly-name.dll"</code>	參考磁片上的元件
<code>#load "file-name.fsx"</code>	讀取原始程式檔、進行編譯，並加以執行。
<code>#help</code>	顯示可用指示詞的詳細資訊。
<code>#I</code>	指定加上引號的組件搜尋路徑。
<code>#quit</code>	終止 F# Interactive 工作階段。

III	DESCRIPTION
#time "on" 或 #time "off"	它本身會 #time 切換是否顯示效能資訊。當它是時 "on" , F # Interactive 會針對所解讀和執行的每個程式碼區段, 測量即時、CPU 時間和垃圾收集資訊。

當您在 F# Interactive 中指定檔案或路徑時, 應該要有一個字串常值。因此, 檔案和路徑必須以引號括住, 並遵循一般的逸出字元。您可以使用 @ 字元讓 F # Interactive 將包含路徑的字串解讀為逐字字串。這會導致 F# Interactive 會忽略所有逸出字元。

互動式和編譯的預處理器指示詞

當您編譯 F # Interactive 中的程式碼時, 無論是以互動方式執行或是執行腳本, 都會定義符號 Interactive 。當您在編譯器中編譯器代碼時, 會定義已 編譯 的符號。因此, 如果程式碼在編譯和互動模式中必須不同, 您可以使用這些預處理器指示詞進行條件式編譯, 以決定要使用的是哪一個。例如:

```
#if INTERACTIVE
// Some code that executes only in FSI
// ...
#endif
```

在 Visual Studio 中使用 F # Interactive

若要透過 Visual Studio 執行 F# Interactive, 您可以按一下標示為 **F# Interactive** 的合適工具列按鈕, 或使用按鍵 **Ctrl+Alt+F**。如此會開啟互動式視窗, 也就是執行 F# Interactive 工作階段的工具視窗。您也可以選取要在互動式視窗中執行的一些程式碼, 然後按按鍵組合 **Alt + Enter**。F# Interactive 會隨即在標示為 **F# Interactive** 的工具視窗中啟動。當您使用這個按鍵組合時, 請確定編輯器視窗具有焦點。

不論是使用主控台還是否 Visual Studio, 命令提示字元都會出現, 表示解譯器在等待您輸入。您可以如同在程式碼檔案中一樣輸入程式碼。若要編譯並執行程式碼, 請輸入兩個分號 (;) 以終止一或數行的輸入。

F# Interactive 會嘗試編譯程式碼, 如果成功的話, 它會執行程式碼, 並列印它所編譯的類型與值的簽章。如果發生錯誤, 解譯器就會列印錯誤訊息。

在同一個工作階段中輸入的程式碼, 可以存取先前輸入的所有建構, 因此您可以建置程式。若有需要, 可利用 [工具] 視窗中的延伸緩衝區, 視需要將程式碼複製到檔案。

在 Visual Studio 中執行 F# Interactive 時, 會與專案分開執行, 因此, 舉例來說, 除非您將函式的程式碼複製到 [互動] 視窗, 否則無法使用在 F# Interactive 中專案內所定義的建構。

您可以調整設定, (選項) 來控制 F # Interactive 命令列引數。在 [工具] 功能表上, 選取 [選項...], 然後展開 [F# 工具]。您只能變更 F# Interactive 選項和 [64 位元 F# Interactive] 這兩項設定, 而且只有在 64 位元電腦上執行 F# Interactive 時才相關。這項設定會決定您是否要執行專用的64位版本的 fsi.exe 或 fsianycpu.exe, 其使用電腦架構來判斷是否以32位或64位進程的形式執行。

相關文章

II	II
F# Interactive 選項	描述 F # Interactive、fsi.exe 的命令列語法和選項。

F# 樣式指南

2020/11/2 • [Edit Online](#)

下列文章說明格式化 F# 程式碼的指導方針，以及語言功能和其使用方式的主題指引。

本指南的設計是根據在大型程式碼基底中使用 F# 搭配不同的程式設計人員群組。本指引通常會在程式的需求隨著時間而變更時，導致成功使用 F# 並將挫折降到最低。

優質 F# 程式碼的五個準則

當您撰寫 F# 程式碼時，請記住下列原則，特別是在將隨著時間變更的系統中。進一步文章中的每一項指導方針都是來自這五個重點。

1. 良好的 F# 程式碼簡潔、易懂且可組合

F# 有許多功能可讓您以較少的程式碼來表達動作，並重複使用一般功能。F# 核心程式庫也包含許多實用的型別和函式，可用來處理常見的資料集合。撰寫您自己的函式，以及 F# 核心程式庫中的函式 (或其他程式庫) 屬於常式慣用 F# 程式設計的一部分。一般來說，如果您可以用較少的程式碼來表達問題的解決方案，其他開發人員 (或未來的自我) 將會令人激賞。此外，強烈建議您在需要執行重要工作時使用程式庫 (例如 Fsharp.core)、執行 F# 的 [大型 .net 程式庫](#)，或 [NuGet](#) 上的協力廠商套件。

2. 良好的 F# 程式碼可互通

交互操作可採用多種形式，包括使用不同語言的程式碼。與其他呼叫端交互操作的程式碼界限是很重要的部分，即使呼叫端也在 F# 中也是一樣。在撰寫 F# 時，您應該一律考慮其他程式碼如何呼叫您所撰寫的程式碼，包括從另一種語言 (例如 c#)。 [F# 元件設計指導方針](#) 會詳細描述互通性。

3. 良好的 F# 程式碼會使用物件程式設計，而不是物件方向

F# 具有在 .NET 中使用物件進行程式設計的完整支援，包括 [類別](#)、[介面](#)、[存取修飾詞](#)、[抽象類別](#) 等等。針對更複雜的功能程式碼 (例如必須是內容感知的函式)，物件可以輕鬆地以函式不能的方式封裝內容資訊。[選擇性參數](#) 和 [選擇性](#) 地使用 [多載的功能](#)，可讓呼叫者更容易使用這項功能。

4. 良好的 F# 程式碼順利執行而不會公開變化

這不是撰寫高效能程式碼的秘密，您必須使用變化。這就是電腦的運作方式。這類程式碼通常很容易出錯，因此很難正確取得。避免對呼叫端公開變化。相反地，請 [建立一個功能介面](#)，以在效能很重要時隱藏以變化為基礎的實作為基礎。

5. Toolable 良好的 F# 程式碼

工具很適合用於大型程式碼基底，而且您可以撰寫 F# 程式碼，以便使用 F# 語言工具來更有效地使用它。其中一個範例是確保您不會以無點的程式設計方式來濫用這個方法它，如此就可以使用偵錯工具來檢查中繼值。另一個範例是使用 [XML 檔批註](#) 來描述結構，讓編輯器中的工具提示可以在呼叫位置顯示這些批註。請務必考慮其他程式設計人員如何使用其工具來讀取、流覽、調試和操作您的程式碼。

後續步驟

[F# 程式碼格式設定指導方針](#) 提供有關如何格式化程式碼以方便閱讀的指引。

[F# 編碼慣例](#) 提供 f# 程式設計慣用語的指引，可協助長期維護較大的 f# 程式碼基底。

[F# 元件設計指導方針](#) 提供撰寫 F# 元件的指引，例如程式庫。

F# 程式碼格式方針

2021/3/12 • [Edit Online](#)

本文提供如何格式化程式碼的指導方針，讓您的 F# 程式碼為：

- 更清晰
- 根據 Visual Studio 和其他編輯器中格式化工具所套用的慣例
- 類似于線上的其他程式碼

這些指導方針是根據 [Anh->ahn-dung \(英文 phan\)](#) 的 [F# 格式慣例的完整指南](#)。

縮排的一般規則

F# 預設會使用顯著的空白字元。下列指導方針旨在提供有關如何操控這項可能強加的一些挑戰的指引。

使用空格

需要縮排時，您必須使用空格，而不是索引標籤。至少需要一個空格。您的組織可以建立編碼標準，以指定要用於縮排的空間數目；一般情況下，會在每個層級上有兩個、三個或四個空格的縮排。

我們建議每個縮排有四個空格。

也就是說，程式的縮排是一項主觀的考慮。變化是正常的，但您應該遵循的第一個規則是 *縮排的一致性*。選擇一般接受的縮排樣式，並在整個程式碼基底中有系統地使用。

將空白字元格式化

F# 是區分泛空白字元。雖然泛空白字元的大部分語法都是由適當的縮排所涵蓋，但還有其他一些需要考慮的事項。

算術運算式中的格式化運算子

一律在二元算術運算式周圍使用空白字元：

```
let subtractThenAdd x = x - 1 + 3
```

一元 `-` 運算子應該一律緊接在其所否定的值之後：

```
// OK
let negate x = -x

// Bad
let negateBad x = - x
```

在操作員之後加入空白字元 `-` 可能會讓其他人混淆。

總而言之，一定要：

- 以空白字元括住二元運算子
- 一元運算子之後永遠沒有尾端空白字元

二元算術運算子的指導方針特別重要。若無法圍住二元 `-` 運算子，與某些格式化選項結合時，可能會導致將它解釋為一元 `-`。

以空白字元括住自訂運算子定義

一律使用空白字元來圍繞運算子定義：

```
// OK
let ( !> ) x f = f x

// Bad
let (!>) x f = f x
```

若為任何以和為開頭 `*` 且包含多個字元的自訂運算子，您需要在定義的開頭加上空白字元，以避免編譯器的混淆。基於這個原因，我們建議您只使用單一空白字元來括住所有運算子的定義。

以空白字元括住函式參數箭號

定義函式的簽章時，請使用空格括住 `->` 符號：

```
// OK
type MyFun = int -> int -> string

// Bad
type MyFunBad = int->int->string
```

以空白字元括住函式引數

定義函式時，請在每個引數前後使用空白字元。

```
// OK
let myFun (a: decimal) b c = a + b + c

// Bad
let myFunBad (a:decimal)(b)c = a + b + c
```

避免區分名稱的對齊

一般來說，請搜尋以避免對命名敏感的縮排和對齊：

```
// OK
let myLongValueName =
    someExpression
    |> anotherExpression

// Bad
let myLongValueName = someExpression
                        |> anotherExpression
```

這有時稱為「虛名對齊」或「虛名縮排」。避免這種情況的主要原因如下：

- 重要的程式碼會移到右邊
- 實際程式碼的寬度較低
- 重新命名可能會中斷對齊

請為提供相同的，以便 `do` / `do!` 讓縮排與保持一致 `let` / `let!`。以下是 `do` 在類別中使用的範例：

```
// OK
type Foo () =
  let foo =
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
  do
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog

// Bad - notice the "do" expression is indented one space less than the `let` expression
type Foo () =
  let foo =
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
  do fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
```

以下是 `do!` 使用2個空格進行縮排 (的範例, 因為在 `do!` 縮排) 使用4個空格時, 方法之間沒有剛好差異:

```
// OK
async {
  let! foo =
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
  do!
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
}

// Bad - notice the "do!" expression is indented two spaces more than the `let!` expression
async {
  let! foo =
    fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
  do! fooBarBaz
    |> loremIpsumDolorSitAmet
    |> theQuickBrownFoxJumpedOverTheLazyDog
}
```

將參數放置在新的一行上以進行長定義

如果您有較長的函式定義, 請將參數放在新行上, 然後將它們縮排, 以符合後續參數的縮排層級。


```

module M =
  let longFunctionWithLotsOfParameters
    (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
    (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
    (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
    =
    // ... the body of the method follows

  let longFunctionWithLotsOfParametersAndReturnType
    (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
    (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
    (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
    : Return Type =
    // ... the body of the method follows

  let longFunctionWithLongTupleParameter
    (
      aVeryLongParam: AVeryLongTypeThatYouNeedToUse,
      aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse,
      aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse
    ) =
    // ... the body of the method follows

  let longFunctionWithLongTupleParameterAndReturnType
    (
      aVeryLongParam: AVeryLongTypeThatYouNeedToUse,
      aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse,
      aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse
    ) : Return Type =
    // ... the body of the method follows

```

這也適用於使用元組的成員、函式和參數：

```

type TM() =
  member _.LongMethodWithLotsOfParameters
    (
      aVeryLongParam: AVeryLongTypeThatYouNeedToUse,
      aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse,
      aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse
    ) =
    // ... the body of the method

type TC
  (
    aVeryLongCtorParam: AVeryLongTypeThatYouNeedToUse,
    aSecondVeryLongCtorParam: AVeryLongTypeThatYouNeedToUse,
    aThirdVeryLongCtorParam: AVeryLongTypeThatYouNeedToUse
  ) =
  // ... the body of the class follows

```

如果參數是 curried，請將 `=` 字元連同任何傳回型別放在新行上：

```

type C() =
    member _.LongMethodWithLotsOfCurriedParamsAndReturnType
        (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        : ReturnType =
        // ... the body of the method
    member _.LongMethodWithLotsOfCurriedParams
        (aVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aSecondVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        (aThirdVeryLongParam: AVeryLongTypeThatYouNeedToUse)
        =
        // ... the body of the method

```

這是避免太長的行 (的方法。如果傳回類型可能具有長名稱)，而且在新增參數時有較少的行損毀。

類型注釋

右邊的函式引數類型注釋

在定義具有類型注釋的引數時，請使用符號後面的空白字元 `:`：

```

// OK
let complexFunction (a: int) (b: int) c = a + b + c

// Bad
let complexFunctionBad (a :int) (b :int) (c:int) = a + b + c

```

以空白字元括住傳回型別注釋

在 `let` 系結函式或實數值型別注釋中，如果函式) 的情況下 (傳回型別，請使用符號前後的空白字元 `:`：

```

// OK
let expensiveToCompute : int = 0 // Type annotation for let-bound value
let myFun (a: decimal) b c : decimal = a + b + c // Type annotation for the return type of a function
// Bad
let expensiveToComputeBad1:int = 1
let expensiveToComputeBad2 :int = 2
let myFunBad (a: decimal) b c:decimal = a + b + c

```

格式化空白行

- 以兩個空白行分隔最上層函數和類別定義。
- 類別內的方法定義會以單一空白行分隔。
- 額外的空白行可 (謹慎地使用) 來分隔相關函式的群組。您可以在一堆相關的囉 (之間省略空白行，例如，一組虛擬實)。
- 請謹慎使用函式中的空白行，以表示邏輯區段。

格式化批註

通常會優先使用多個雙斜線批註，而不是 ML 樣式的區塊批註。

```

// Prefer this style of comments when you want
// to express written ideas on multiple lines.

(*
    ML-style comments are fine, but not a .NET-ism.
    They are useful when needing to modify multi-line comments, though.
*)

```

內嵌批註應為第一個字母大寫。

```
let f x = x + 1 // Increment by one.
```

命名規範

針對類別系結、運算式系結和模式系結值和函數使用 camelCase

這是常見且接受的 F # 樣式，可針對系結為區域變數或模式比對和函式定義中的所有名稱使用 camelCase。

```
// OK
let addIAndJ i j = i + j

// Bad
let addIAndJ I J = I+J

// Bad
let AddIAndJ i j = i + j
```

類別中的本機系結函式也應使用 camelCase。

```
type MyClass() =

    let doSomething () =

        let firstResult = ...

        let secondResult = ...

    member x.Result = doSomething()
```

針對模組系結公用函式使用 camelCase

當模組系結函式是公用 API 的一部分時，它應該使用 camelCase：

```
module MyAPI =
    let publicFunctionOne param1 param2 param2 = ...

    let publicFunctionTwo param1 param2 param3 = ...
```

針對內部和私用模組系結值和函式使用 camelCase

將 camelCase 用於私用模組系結的值，包括下列各項：

- 腳本中的特定函數
- 組成模組或型別內部執行的值

```
let emailMyBossTheLatestResults =
    ...
```

針對參數使用 camelCase

所有參數都應該根據 .NET 命名慣例使用 camelCase。

```
module MyModule =  
    let myFunction paramOne paramTwo = ...  
  
type MyClass() =  
    member this.MyMethod(paramOne, paramTwo) = ...
```

針對模組使用 PascalCase

(頂層、內部、私用、嵌套) 的所有模組都應該使用 PascalCase。

```
module MyTopLevelModule  
  
module Helpers =  
    module private SuperHelpers =  
        ...  
  
    ...
```

針對類型宣告、成員和標籤使用 PascalCase

類別、介面、結構、列舉、委派、記錄和區分等位都應該以 PascalCase 命名。記錄和差異聯集的類型與標籤內的成員也應使用 PascalCase。

```
type IMyInterface =  
    abstract Something: int  
  
type MyClass() =  
    member this.MyMethod(x, y) = x + y  
  
type MyRecord = { IntVal: int; StringVal: string }  
  
type SchoolPerson =  
    | Professor  
    | Student  
    | Advisor  
    | Administrator
```

針對 .NET 內建函式使用 PascalCase

命名空間、例外狀況、事件和專案/`.dll` 名稱也應使用 PascalCase。這不僅會讓取用者更自然地使用其他 .NET 語言，它也與您可能遇到的 .NET 命名慣例一致。

避免名稱中有底線

在過去，某些 F# 程式庫在名稱中使用底線。不過，這不會再被廣泛接受，部分原因是它與 .NET 命名慣例相衝突。話雖如此，某些 F# 程式設計人員會大量使用底線，部分基於歷史原因，而且容錯和尊重很重要。不過，此樣式通常是由有關於是否使用它的其他人不喜歡。

其中一個例外狀況包含與原生元件的互通性，其中的底線是通用的。

使用標準 F# 運算子

下列運算子是在 F# 標準程式庫中定義，而且應該使用，而不是定義對等專案。建議使用這些運算子，因為它通常會讓程式碼更容易閱讀及慣用。背景 OCaml 或其他功能性程式設計語言的開發人員，可能習慣于不同的慣用語。下列清單摘要說明建議的 F# 運算子。

```
x |> f // Forward pipeline
f >> g // Forward composition
x |> ignore // Discard away a value
x + y // Overloaded addition (including string concatenation)
x - y // Overloaded subtraction
x * y // Overloaded multiplication
x / y // Overloaded division
x % y // Overloaded modulus
x && y // Lazy/short-cut "and"
x || y // Lazy/short-cut "or"
x <<< y // Bitwise left shift
x >>> y // Bitwise right shift
x ||| y // Bitwise or, also for working with “flags” enumeration
x &&& y // Bitwise and, also for working with “flags” enumeration
x ^^ y // Bitwise xor, also for working with “flags” enumeration
```

使用泛型的前置詞語法 (`Foo<T>`) 喜好設定為後置語法 (`T Foo`)

F# 會同時繼承命名泛型型別的后置 ML 樣式 (例如, `int list`) 和前置詞 .net 樣式 (例如 `list<int>`)。偏好使用 .NET 樣式, 除了五個特定的類型:

1. 針對 F# 清單, 請使用後置格式: `int list` 而不是 `list<int>`。
2. 針對 F# 選項, 請使用後置格式: `int option` 而不是 `option<int>`。
3. 針對 F# 值選項, 請使用後置格式: `int voption` 而不是 `voption<int>`。
4. 針對 F# 陣列, 請使用語法名稱, `int[]` 而不是 `int array` 或 `array<int>`。
5. 針對參考資料格, 請使用 `int ref` 而不是 `ref<int>` 或 `Ref<int>`。

若為所有其他類型, 請使用前置詞形式。

格式化元組

元組具現化應該是以括弧括住, 而且其中的分隔逗點後面應該接著一個空格, 例如: `(1, 2)`, `(x, y, z)`。

通常會接受在元組的模式比對中省略括弧:

```
let (x, y) = z // Destructuring
let x, y = z // OK

// OK
match x, y with
| 1, _ -> 0
| x, 1 -> 0
| x, y -> 1
```

如果元組是函式的傳回值, 通常也會接受省略括弧:

```
// OK
let update model msg =
    match msg with
    | 1 -> model + 1, []
    | _ -> model, [ msg ]
```

總而言之, 建議使用括弧化元組具現化, 但使用元組進行模式比對或傳回值時, 會將其視為可避免括弧。

格式化區分聯集宣告

| 在類型定義中縮排四個空格:

```
// OK
type Volume =
  | Liter of float
  | FluidOunce of float
  | ImperialPint of float

// Not OK
type Volume =
  | Liter of float
  | US Pint of float
  | ImperialPint of float
```

設定區分等位的格式

使用括弧/元集合參數之前的空格來區分聯集的案例：

```
// OK
let opt = Some ("A", 1)

// Not OK
let opt = Some("A", 1)
```

跨多行分割的具現化差異聯集，應該為包含的資料提供具有縮排的新範圍：

```
let tree1 =
    BinaryNode
        (BinaryNode (BinaryValue 1, BinaryValue 2),
         BinaryNode (BinaryValue 3, BinaryValue 4))
```

右括弧也可以在新行上：

```
let tree1 =
    BinaryNode(
        BinaryNode (BinaryValue 1, BinaryValue 2),
        BinaryNode (BinaryValue 3, BinaryValue 4)
    )
```

格式化記錄聲明

{ 在類型定義中縮排四個空格，並在同一行上啟動欄位清單：

```
// OK
type PostalAddress =
    { Address: string
      City: string
      Zip: string }
    member x.ZipAndCity = $"{x.Zip} {x.City}"

// Not OK
type PostalAddress =
    { Address: string
      City: string
      Zip: string }
    member x.ZipAndCity = $"{x.Zip} {x.City}"

// Unusual in F#
type PostalAddress =
    {
        Address: string
        City: string
        Zip: string
    }
```

如果您要在記錄上宣告介面或成員，最好將開頭標記放在新行上，然後將結束記號放在新行上：

```
// Declaring additional members on PostalAddress
type PostalAddress =
    {
        Address: string
        City: string
        Zip: string
    }
    member x.ZipAndCity = $"{x.Zip} {x.City}"

type MyRecord =
    {
        SomeField: int
    }
interface IMyInterface
```

格式化記錄

簡短記錄可以用一行撰寫：

```
let point = { X = 1.0; Y = 0.0 }
```

較長的記錄應該使用新的標籤行：

```
let rainbow =
    { Boss = "Jeffrey"
      Lackeys = ["Zippy"; "George"; "Bungle"] }
```

如果您是下列情況，最好將開啟權杖放在新的一行上，將內容索引標籤為一個範圍，並將結束記號放在新行上：

- 在具有不同縮排範圍的程式碼中移動記錄
- 將它們輸送至函式

```

let rainbow =
{
    Boss1 = "Jeffrey"
    Boss2 = "Jeffrey"
    Boss3 = "Jeffrey"
    Boss4 = "Jeffrey"
    Boss5 = "Jeffrey"
    Boss6 = "Jeffrey"
    Boss7 = "Jeffrey"
    Boss8 = "Jeffrey"
    Lackeys = ["Zippy"; "George"; "Bungle"]
}

type MyRecord =
{
    SomeField: int
}
interface IMyInterface

let foo a =
a
|> Option.map
    (fun x ->
        {
            MyField = x
        })

```

相同的規則適用於清單和陣列元素。

格式化複製和更新記錄運算式

複製和更新記錄運算式仍是一筆記錄，因此適用類似的指導方針。

簡短運算式可以容納在同一行：

```
let point2 = { point with X = 1; Y = 2 }
```

較長的運算式應該使用新的行：

```

let rainbow2 =
{ rainbow with
    Boss = "Jeffrey"
    Lackeys = [ "Zippy"; "George"; "Bungle" ] }

```

如同記錄指引，您可能會想要為大括弧專用不同的行，並使用運算式將一個範圍向右縮排。在某些特殊情況下（例如，使用不含括弧的選擇性包裝值），您可能需要將大括弧放在同一行：

```

type S = { F1: int; F2: string }
type State = { Foo: S option }

let state = { Foo = Some { F1 = 1; F2 = "Hello" } }
let newState =
{
    state with
        Foo =
            Some {
                F1 = 0
                F2 = ""
            }
}

```


格式化清單和陣列

`x :: 1` 以括弧括住的空格撰寫 `::` (`::` 是中置運算子，因此以空格) 括住。

在一行上宣告的清單和陣列，在左括弧之後和右括弧前面都必須有一個空格：

```
let xs = [ 1; 2; 3 ]
let ys = [| 1; 2; 3; |]
```

請一律在兩個不同的大括弧運算子之間使用至少一個空格。例如，在和之間保留一個空格 `[{`。

```
// OK
[ { IngredientName = "Green beans"; Quantity = 250 }
  { IngredientName = "Pine nuts"; Quantity = 250 }
  { IngredientName = "Feta cheese"; Quantity = 250 }
  { IngredientName = "Olive oil"; Quantity = 10 }
  { IngredientName = "Lemon"; Quantity = 1 } ]

// Not OK
[ { IngredientName = "Green beans"; Quantity = 250 }
  { IngredientName = "Pine nuts"; Quantity = 250 }
  { IngredientName = "Feta cheese"; Quantity = 250 }
  { IngredientName = "Olive oil"; Quantity = 10 }
  { IngredientName = "Lemon"; Quantity = 1 }]
```

相同的指導方針適用於元組的清單或陣列。

分割成多行的清單和陣列，會遵循與記錄相同的規則：

```
let pascalsTriangle =
    [|
        [ 1 ]
        [ 1; 1 ]
        [ 1; 2; 1 ]
        [ 1; 3; 3; 1 ]
        [ 1; 4; 6; 4; 1 ]
        [ 1; 5; 10; 10; 5; 1 ]
        [ 1; 6; 15; 20; 15; 6; 1 ]
        [ 1; 7; 21; 35; 35; 21; 7; 1 ]
        [ 1; 8; 28; 56; 70; 56; 28; 8; 1 ]
    |]
```

和記錄一樣，在自己的行上宣告左右括弧將會讓程式碼四處移動和輸送至函式。

以程式設計方式產生陣列和清單時，最好不要在 `->` `do ... yield` 永遠產生值時使用：

```
// Preferred
let squares = [ for x in 1..10 -> x * x ]

// Not preferred
let squares' = [ for x in 1..10 do yield x * x ]
```

舊版的 F# 語言需要 `yield` 在可能有條件地產生資料的情況下指定，或可能有連續的運算式可供評估。`yield` 除非您必須使用較舊的 F# 語言版本進行編譯，否則偏好省略這些關鍵字：

```
// Preferred
let daysOfWeek includeWeekend =
    [
        "Monday"
        "Tuesday"
        "Wednesday"
        "Thursday"
        "Friday"
        if includeWeekend then
            "Saturday"
            "Sunday"
    ]

// Not preferred
let daysOfWeek' includeWeekend =
    [
        yield "Monday"
        yield "Tuesday"
        yield "Wednesday"
        yield "Thursday"
        yield "Friday"
        if includeWeekend then
            yield "Saturday"
            yield "Sunday"
    ]
```

在某些情況下，`do...yield` 可能有助於提高可讀性。不過，您應該將這些情況納入主觀考慮。

格式化 if 運算式

條件的縮排取決於構成它們的運算式大小和複雜度。當下列情況時，請將它們寫入一行：

- `cond`、`e1` 和 `e2` 是簡短的
- `e1` 和 `e2` 本身並非 `if/then/else` 運算式本身。

```
if cond then e1 else e2
```

如果任何運算式為多行或運算式，則為 `if/then/else`。

```
if cond then
    e1
else
    e2
```

使用和的多個條件，`elif` `else` 會在 `if` 遵循一行運算式的規則時，在與相同的範圍中縮排 `if/then/else`。

```
if cond1 then e1
elif cond2 then e2
elif cond3 then e3
else e4
```

如果有任何條件或運算式為多行，則整個 `if/then/else` 運算式為多行：

```
if cond1 then
  e1
elif cond2 then
  e2
elif cond3 then
  e3
else
  e4
```

模式比對結構

| 針對相符的每個子句(沒有縮排)使用。如果運算式很短, 則如果每個子運算式也很簡單, 您可以考慮使用一行。

```
// OK
match l with
| { him = x; her = "Posh" } :: tail -> x
| _ :: tail -> findDavid tail
| [] -> failwith "Couldn't find David"

// Not OK
match l with
  | { him = x; her = "Posh" } :: tail -> x
  | _ :: tail -> findDavid tail
  | [] -> failwith "Couldn't find David"
```

如果模式比對箭號右邊的運算式太大, 請將它移到下一行, 從縮排一個步驟 `match` / | 。

```
match lam with
| Var v -> 1
| Abs(x, body) ->
  1 + sizeLambda body
| App(lam1, lam2) ->
  sizeLambda lam1 + sizeLambda lam2
```

匿名函式的模式比對(從開始 `function`)通常不會縮排太遠。例如, 如下所示將一個範圍縮排, 如下所示:

```
lambdaList
|> List.map
  (function
    | Abs(x, body) -> 1 + sizeLambda 0 body
    | App(lam1, lam2) -> sizeLambda (sizeLambda 0 lam1) lam2
    | Var v -> 1)
```

在開頭的函式中的模式比 `let` `let rec` 對應該在開始之後縮排四個空格 `let`, 即使 `function` 使用關鍵字也是如此:

```
let rec sizeLambda acc = function
  | Abs(x, body) -> sizeLambda (succ acc) body
  | App(lam1, lam2) -> sizeLambda (sizeLambda acc lam1) lam2
  | Var v -> succ acc
```

我們不建議對齊箭號。

格式化 try/with 運算式

例外狀況類型的模式比對應該在與相同的層級上縮排 `with` 。

```

try
  if System.DateTime.Now.Second % 3 = 0 then
    raise (new System.Exception())
  else
    raise (new System.ApplicationException())
with
| :? System.ApplicationException ->
  printfn "A second that was not a multiple of 3"
| _ ->
  printfn "A second that was a multiple of 3"

```

格式化函式參數應用程式

一般而言，大部分引數都是在同一行提供：

```

let x = sprintf "\t%s - %i\n\r" x.IngredientName x.Quantity

let printListWithOffset a list1 =
  List.iter (fun elem -> printfn $"%d{a + elem}") list1

```

當管線很在意時，通常也會有同樣的情況，也就是在同一行上以引數的形式來套用局部函數：

```

let printListWithOffsetPiped a list1 =
  list1
  |> List.iter (fun elem -> printfn $"%d{a + elem}")

```

不過，您可能會想要在新行上將引數傳遞至函式，因為可讀性或引數清單或引數名稱太長。在此情況下，請使用一個範圍縮排：

```

// OK
sprintf "\t%s - %i\n\r"
    x.IngredientName x.Quantity

// OK
sprintf
    "\t%s - %i\n\r"
    x.IngredientName x.Quantity

// OK
let printVolumes x =
  printf "Volume in liters = %f, in us pints = %f, in imperial = %f"
    (convertVolumeToLiter x)
    (convertVolumeUSPint x)
    (convertVolumeImperialPint x)

```

針對 lambda 運算式，您可能也想要考慮將 lambda 運算式的主體放在新行上（以一個範圍縮排，如果夠長）：

```

let printListWithOffset a list1 =
    List.iter
        (fun elem ->
            printfn $"A very long line to format the value: %d{a + elem}")
        list1

let printListWithOffsetPiped a list1 =
    list1
    |> List.iter
        (fun elem ->
            printfn $"A very long line to format the value: %d{a + elem}")

```

如果 lambda 運算式的主體長度為多行，您應該考慮將它重構至本機範圍的函式。

參數通常應該相對於函式或關鍵字進行縮排 `fun` / `function`，而不論函式出現的內容為何：

```

// With 4 spaces indentation
list1
|> List.fold
    someLongParam
    anotherLongParam

list1
|> List.iter
    (fun elem ->
        printfn $"A very long line to format the value: %d{elem}")

// With 2 spaces indentation
list1
|> List.fold
    someLongParam
    anotherLongParam

list1
|> List.iter
    (fun elem ->
        printfn $"A very long line to format the value: %d{elem}")

```

當函式採用單一多行元組引數時，適用於格式化函式、靜態成員和成員調用的相同規則也適用。

```

let myFunction (a: int, b: string, c: int, d: bool) =
    ()

myFunction(
    478815516,
    "A very long string making all of this multi-line",
    1515,
    false
)

```

格式化中置運算子

以空格分隔運算子。這項規則的明顯例外是 `!` 和 `.` 運算子。

中置運算式可在相同資料行上進行組合：

```
acc +
(sprintf "%t%s - %i\n\r"
    x.IngredientName x.Quantity)

let function1 arg1 arg2 arg3 arg4 =
    arg1 + arg2 +
    arg3 + arg4
```

格式化管線運算子或可變指派

管線 `|>` 運算子應該在其運作的運算式底下。

```
// Preferred approach
let methods2 =
    System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun asm -> asm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat

// Not OK
let methods2 = System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun asm -> asm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat

// Not OK either
let methods2 = System.AppDomain.CurrentDomain.GetAssemblies()
    |> List.ofArray
    |> List.map (fun asm -> asm.GetTypes())
    |> Array.concat
    |> List.ofArray
    |> List.map (fun t -> t.GetMethods())
    |> Array.concat
```

這也適用於可變的 setter：

```
// Preferred approach
ctx.Response.Headers.[HeaderNames.ContentType] <-
    Constants.jsonApiMediaType |> StringValues
ctx.Response.Headers.[HeaderNames.ContentLength] <-
    bytes.Length |> string |> StringValues

// Not OK
ctx.Response.Headers.[HeaderNames.ContentType] <- Constants.jsonApiMediaType
    |> StringValues
ctx.Response.Headers.[HeaderNames.ContentLength] <- bytes.Length
    |> string
    |> StringValues
```

格式化模組

本機模組中的程式碼必須相對於模組進行縮排，但是最上層模組中的程式碼不應該縮排。命名空間元素不需要縮排。

```
// A is a top-level module.
module A

let function1 a b = a - b * b
```

```
// A1 and A2 are local modules.
module A1 =
    let function1 a b = a * a + b * b

module A2 =
    let function2 a b = a * a - b * b
```

格式化物件運算式和介面

物件運算式和介面的對齊方式應該與在 `member` 四個空格之後縮排的方式相同。

```
let comparer =
    { new IComparer<string> with
        member x.Compare(s1, s2) =
            let rev (s: String) =
                new String (Array.rev (s.ToCharArray()))
            let reversed = rev s1
            reversed.CompareTo (rev s2) }
```

格式化運算式中的空白字元

避免 F# 運算式中有多餘的空白字元。

```
// OK
spam (ham.[1])

// Not OK
spam ( ham.[ 1 ] )
```

具名引數也不應該有周圍的空格 `=` :

```
// OK
let makeStreamReader x = new System.IO.StreamReader(path=x)

// Not OK
let makeStreamReader x = new System.IO.StreamReader(path = x)
```

格式化函式、靜態成員和成員調用

如果運算式是簡短的，請以空格分隔引數，並將它保留在同一行。

```
let person = new Person(a1, a2)

let myRegexMatch = Regex.Match(input, regex)

let untypedRes = checker.ParseFile(file, source, opts)
```

如果運算式很長，請使用分行符號和縮排一個範圍，而不是縮排到括弧。

```

let person =
    new Person(
        argument1,
        argument2
    )

let myRegexMatch =
    Regex.Match(
        "my longer input string with some interesting content in it",
        "myRegexPattern"
    )

let untypedRes =
    checker.ParseFile(
        fileName,
        sourceText,
        parsingOptionsWithDefines
    )

```

即使只有單一多行引數，也適用相同的規則。

```

let poemBuilder = StringBuilder()
poemBuilder.AppendLine(
    """
    The last train is nearly due
    The Underground is closing soon
    And in the dark, deserted station
    Restless in anticipation
    A man waits in the shadows
    """
)

Option.traverse(
    create
    >> Result.setError [ invalidHeader "Content-Checksum" ]
)

```

格式化屬性

[屬性](#) 位於結構的上方：

```

[<SomeAttribute>]
type MyClass() = ...

[<RequireQualifiedAccess>]
module M =
    let f x = x

[<Struct>]
type MyRecord =
    { Label1: int
      Label2: string }

```

它們應該位於任何 XML 檔之後：

```

/// Module with some things in it.
[<RequireQualifiedAccess>]
module M =
    let f x = x

```


格式化參數上的屬性

屬性也可以放在參數上。在此情況下，請將它放在與參數相同的行上，然後在名稱之前：

```
// Defines a class that takes an optional value as input defaulting to false.
type C() =
    member _.M([<Optional; DefaultParameterValue(false)>] doSomething: bool)
```

格式化多個屬性

當您將多個屬性套用至非參數的結構時，應該將它們放在每行一個屬性中：

```
[<Struct>]
[<IsByRefLike>]
type MyRecord =
    { Label1: int
      Label2: string }
```

套用至參數時，它們必須在同一行，並以 `;` 分隔符號分隔。

格式化常值

使用屬性的 **F# 常值** `Literal` 應該將屬性放在其本身的行，並使用 PascalCase 命名：

```
[<Literal>]
let Path = __SOURCE_DIRECTORY__ + "/" + __SOURCE_FILE__

[<Literal>]
let MyUrl = "www.mywebsitethatiamworkingwith.com"
```

避免將屬性放在與值相同的行上。

格式化計算運算式作業

建立 **計算運算式** 的自訂作業時，建議使用 camelCase 命名：

```

type MathBuilder () =
    member _.Yield _ = 0

    [<CustomOperation("addOne")>]
    member _.AddOne (state: int) =
        state + 1

    [<CustomOperation("subtractOne")>]
    member _.SubtractOne (state: int) =
        state - 1

    [<CustomOperation("divideBy")>]
    member _.DivideBy (state: int, divisor: int) =
        state / divisor

    [<CustomOperation("multiplyBy")>]
    member _.MultiplyBy (state: int, factor: int) =
        state * factor

let math = MathBuilder()

// 10
let myNumber =
    math {
        addOne
        addOne
        addOne

        subtractOne

        divideBy 2

        multiplyBy 10
    }

```

正在建立模型的網域最後應該會驅動命名慣例。如果慣用為使用不同的慣例，則應改用該慣例。

F# 編碼慣例

2021/3/5 • [Edit Online](#)

下列慣例是從使用大型 F# 基底的經驗來制定的。[良好 F# 程式碼的五個準則](#)是每個建議的基礎。它們與 [f# 元件設計指導方針](#) 相關，但適用於任何 f# 程式碼，而不只是程式庫之類的元件。

組織程式碼

F# 提供兩種主要的方式來組織程式碼：模組和命名空間。這些都很類似，但有下列差異：

- 命名空間會編譯為 .NET 命名空間。模組會編譯為靜態類別。
- 命名空間一律為最上層。模組可以是最上層的，也可以嵌套在其他模組內。
- 命名空間可以橫跨多個檔案。模組無法。
- 模組可以使用和裝飾 `[<RequireQualifiedAccess>]` `[<AutoOpen>]`。

下列指導方針可協助您使用這些程式碼來組織程式碼。

偏好最上層的命名空間

對於任何公開取用的程式碼，命名空間都優先於最上層的模組。因為它們會編譯為 .NET 命名空間，所以可以從 c# 取用，而不會有任何問題。

```
// Good!
namespace MyCode

type MyClass() =
    ...
```

使用最上層模組可能不會在從 F# 呼叫時出現不同的情況，但針對 c# 取用者，呼叫者可能會因為必須符合此模組而感到驚訝 `MyClass` `MyCode`。

```
// Bad!
module MyCode

type MyClass() =
    ...
```

謹慎申請 `[<AutoOpen>]`

此 `[<AutoOpen>]` 結構可會污染可供呼叫端使用的範圍，而發生問題的答案是「魔術」。這不是件好事。這項規則的例外狀況是 F# 核心程式庫本身 (但這項事實也有點有爭議的)。

但是，如果您有想要與公用 API 分開組織的公用 API 的協助程式功能，這是一個便利的方式。

```

module MyAPI =
    [<AutoOpen>]
    module private Helpers =
        let helper1 x y z =
            ...

    let myFunction1 x =
        let y = ...
        let z = ...

        helper1 x y z

```

這可讓您將函式公用 API 中的實作詳細資料完全分開，而不需要在每次呼叫時都完整限定協助程式。

此外，您可以在命名空間層級公開擴充方法和運算式產生器，以整齊地表示 `[<AutoOpen>]`。

`[<RequireQualifiedAccess>]` 在名稱可能發生衝突時使用，或者您覺得它有助於提高可讀性

將 `[<RequireQualifiedAccess>]` 屬性新增至模組，表示模組可能無法開啟，而且模組的元素參考需要明確的存取權。例如，`Microsoft.FSharp.Collections.List` 模組有這個屬性。

當模組中的函式和值有可能與其他模組中的名稱衝突的名稱時，這非常有用。需要限定存取權可能會大幅增加資源庫的長期維護和 evolvability。

```

[<RequireQualifiedAccess>]
module StringTokenization =
    let parse s = ...

    ...

    let s = getAString()
    let parsed = StringTokenization.parse s // Must qualify to use 'parse'

```

排序 `open` 語句拓撲

在 F# 中，宣告的順序很重要，包括 `open` 語句。這與 C# 不同，它的效果與檔案 `using` `using static` 中這些語句的順序無關。

在 F# 中，開啟至範圍的專案可以遮蔽其他已經存在的專案。這表示重新排列 `open` 語句可以改變程式碼的意義。因此，所有語句的任意排序 (例如 `open`，不建議使用順序)，以免您產生可能預期的不同行為。

相反地，我們建議您將它們排序**拓撲**；也就是說，依照您的系統層級定義順序來排序您的 `open` 語句。也可以考慮在不同拓撲層級內進行英數位元排序。

例如，以下是 F# 編譯器服務公用 API 檔案的拓撲排序：

```
namespace Microsoft.FSharp.Compiler.SourceCodeServices
```

```
open System
open System.Collections.Generic
open System.Collections.Concurrent
open System.Diagnostics
open System.IO
open System.Reflection
open System.Text

open FSharp.Compiler
open FSharp.Compiler.AbstractIL
open FSharp.Compiler.AbstractIL.Diagnostics
open FSharp.Compiler.AbstractIL.IL
open FSharp.Compiler.AbstractIL.ILBinaryReader
open FSharp.Compiler.AbstractIL.Internal
open FSharp.Compiler.AbstractIL.Internal.Library

open FSharp.Compiler.AccessibilityLogic
open FSharp.Compiler.Ast
open FSharp.Compiler.CompileOps
open FSharp.Compiler.CompileOptions
open FSharp.Compiler.Driver

open Internal.Utilities
open Internal.Utilities.Collections
```

請注意，分行符號會分隔拓撲層，之後每個圖層會以順序的方式排序。這會完全組織程式碼，而不會不小心地遮蔽值。

使用類別包含具有副作用的值

初始化某個值時，有許多次可能會有副作用，例如將內容具現化至資料庫或其他遠端資源。將模組中的這類專案初始化是很吸引人，並在後續的函式中使用：

```
// This is bad!
module MyApi =
    let dep1 = File.ReadAllText "/Users/<name>/connectionstring.txt"
    let dep2 = Environment.GetEnvironmentVariable "DEP_2"

    let private r = Random()
    let dep3() = r.Next() // Problematic if multiple threads use this

    let function1 arg = doStuffWith dep1 dep2 dep3 arg
    let function2 arg = doSutffWith dep1 dep2 dep3 arg
```

這通常不是個好主意，原因如下：

首先，應用程式設定會使用和推送至程式碼基底 `dep1` `dep2` 。這在較大的程式碼基底中很難維護。

第二，如果您的元件本身會使用多個執行緒，則靜態初始化的資料不應該包含不安全線程的值。這顯然是違反的 `dep3` 。

最後，模組初始化會編譯成整個編譯單位的靜態函式。如果該模組中的 `let` 系結值初始化發生任何錯誤，則會將它列為， `TypeInitializationException` 然後針對應用程式的整個存留期進行快取。這可能很難診斷。通常您可以嘗試的內部例外狀況，但如果沒有，則不會告訴根本原因為何。

相反地，只要使用簡單的類別來保存相依性即可：

```
type MyParametricApi(dep1, dep2, dep3) =
    member _.Function1 arg1 = doStuffWith dep1 dep2 dep3 arg1
    member _.Function2 arg2 = doStuffWith dep1 dep2 dep3 arg2
```

這會啟用下列功能：

1. 將任何相依狀態推送至 API 本身之外。
2. 您現在可以在 API 外部進行設定。
3. 相依值的初始化錯誤不太可能會被資訊清單為 `TypeInitializationException`。
4. API 現在更容易測試。

錯誤管理

大型系統中的錯誤管理是一項複雜且差別細微的工作，而且沒有任何銀級的專案可確保您的系統具有容錯能力且運作正常。下列指導方針應提供導覽此困難空間的指引。

代表您的網域內建類型中的錯誤案例和不合法的狀態

使用 [差異](#) 聯集時，F# 讓您能夠在類型系統中代表錯誤的程式狀態。例如：

```
type MoneyWithdrawalResult =
    | Success of amount:decimal
    | InsufficientFunds of balance:decimal
    | CardExpired of DateTime
    | UndisclosedFailure
```

在此情況下，有三種已知方式可從銀行帳戶提款 money。每個錯誤案例都是以類型表示，因此可在整個程式中安全地處理。

```
let handleWithdrawal amount =
    let w = withdrawMoney amount
    match w with
    | Success am -> printfn $"Successfully withdrew %{am}"
    | InsufficientFunds balance -> printfn $"Failed: balance is %{balance}"
    | CardExpired expiredDate -> printfn $"Failed: card expired on {expiredDate}"
    | UndisclosedFailure -> printfn "Failed: unknown"
```

一般情況下，如果您可以針對在網域中可能會失敗的不同方式建立模型，則錯誤處理常式代碼不會再被視為您必須在一般程式流程中處理的某些專案。它只是一般程式流程的一部分，不是例外狀況。這有兩個主要優點：

1. 當您的網域隨著時間變更時，更容易維護。
2. 錯誤案例更容易進行單元測試。

當錯誤無法以類型表示時，請使用例外狀況

並非所有錯誤都可以在問題網域中表示。這類錯誤本質上很例外，因此能夠在 F# 中引發和攔截例外狀況。

首先，建議您閱讀 [例外狀況設計指導方針](#)。這些也適用於 F#。

F# 中可用來引發例外狀況的主要結構，應該依照下列喜好設定順序來考慮：

"	"	"
<code>nullArg</code>	<code>nullArg "argumentName"</code>	<code>System.ArgumentNullException</code> 使用指定的引數名稱引發。

「	「	「
<code>invalidArg</code>	<code>invalidArg "argumentName" "message"</code>	<code>System.ArgumentException</code> 使用指定的引數名稱和訊息來引發。
<code>invalidOp</code>	<code>invalidOp "message"</code>	<code>System.InvalidOperationException</code> 使用指定的訊息引發。
<code>raise</code>	<code>raise (ExceptionType("message"))</code>	擲回例外狀況的一般用途機制。
<code>failwith</code>	<code>failwith "message"</code>	<code>System.Exception</code> 使用指定的訊息引發。
<code>failwithf</code>	<code>failwithf "format string" argForFormatString</code>	<code>System.Exception</code> 使用格式字串及其輸入所決定的訊息來引發。

使用 `nullArg`、`invalidArg` 和 `invalidOp` 做為要擲回的機制 `ArgumentNullException` `ArgumentException` 和 `InvalidOperationException` 適當時機。

和函式 `failwith` `failwithf` 通常應該避免，因為它們會引發基底 `Exception` 類型，而不是特定的例外狀況。根據例外狀況 [設計指導方針](#)，您會想要在可以的情況下引發更明確的例外狀況。

使用例外狀況處理語法

F# 透過語法支援例外狀況模式 `try...with`：

```
try
    tryGetFileContents()
with
| :? System.IO.FileNotFoundException as e -> // Do something with it here
| :? System.Security.SecurityException as e -> // Do something with it here
```

如果您想要讓程式碼保持乾淨，請在發生例外狀況的情況時，協調要執行的功能。處理這種情況的其中一種方法，就是使用 [主動模式](#)，將發生例外狀況本身的功能分組。例如，您可能會使用 API，當它擲回例外狀況時，會將寶貴的資訊包含在例外狀況中繼資料中。在主動模式內的已攔截例外狀況內，解除包裝有用的值，並傳回該值，在某些情況下可能會很有說明。

請勿使用 monadic 錯誤處理來取代例外狀況

例外狀況通常會在功能程式設計中視為 taboo。事實上，例外狀況違反了純度，因此可以放心地將它們視為無法正常運作。不過，這會忽略程式碼必須執行的實際情況，而且可能發生執行階段錯誤。一般來說，撰寫程式碼時假設大部分的專案都不是單純也不是總計，以將不愉快的意外情況降到最低。

請務必考慮下列與 .NET 執行時間和跨語言生態系統中的相關性和「適當性」相關的例外狀況核心優勢/層面：

1. 它們包含詳細的診斷資訊，在對問題進行偵錯工具時很有說明。
2. 執行時間和其他 .NET 語言都能充分瞭解它們。
3. 相較於在特定的程式碼中執行部分部分的部分，它們可以減少重複使用的程式碼。

第三個點很重要。針對重要複雜的作業，無法使用例外狀況可能會導致處理如下的結構：

```
Result<Result<MyType, string>, string list>
```

這可能會導致「stringly 型別」錯誤的模式比對等脆弱的程式碼：

```
let result = doStuff()
match result with
| Ok r -> ...
| Error e ->
    if e.Contains "Error string 1" then ...
    elif e.Contains "Error string 2" then ...
    else ... // Who knows?
```

此外，對於會傳回 "更好" 類型的 "simple" 函式，忍受任何例外狀況可能很吸引人：

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with _ -> None
```

可惜的是，`tryReadAllText` 可能會擲回許多例外狀況，而這些例外狀況可能會發生在檔案系統上，而此程式碼會捨棄您的環境中可能發生錯誤的任何資訊。如果您以結果型別取代此程式碼，則會回到「stringly 型別」的錯誤訊息剖析：

```
// This is bad!
let tryReadAllText (path : string) =
    try System.IO.File.ReadAllText path |> Ok
    with e -> Error e.Message

let r = tryReadAllText "path-to-file"
match r with
| Ok text -> ...
| Error e ->
    if e.Contains "uh oh, here we go again..." then ...
    else ...
```

然後，將例外狀況物件本身放在函式中，`Error` 就會強制您在呼叫位置（而不是在函式中）適當處理例外狀況類型。這麼做可有效地建立已檢查的例外狀況，這些例外狀況是 `unfun` 來處理成為 API 的呼叫者。

上述範例的一個好方法是攔截 特定的例外狀況，並在該例外狀況的內容中傳回有意義的值。如果您依照下列方式修改函式 `tryReadAllText`，`None` 有更多意義：

```
let tryReadAllTextIfPresent (path : string) =
    try System.IO.File.ReadAllText path |> Some
    with :? FileNotFoundException -> None
```

這個函式現在會在找不到檔案時適當地處理案例，並將該意義指派給傳回，而不是全部運作。這個傳回值可以對應至該錯誤案例，而不會捨棄任何內容相關資訊，或強制呼叫端處理在程式碼中該點可能不相關的案例。

等型別 `Result<'Success, 'Error>` 適用於不會進行嵌套的基本作業，而且 F# 選用型別最適合用來表示何時有可能傳回 某事物 或 任何東西。不過，它們不是例外狀況的取代，而且不應該用於嘗試取代例外狀況。相反地，應謹慎套用，以依目標的方式處理例外狀況和錯誤管理原則的特定層面。

部分應用程式和無點程式設計

F# 支援部分的應用程式，因此，您可以使用不同的方式來以無點的樣式進行程式設計。這對於在模組內重複使用程式碼或執行某些工作很有說明，但它並不是公開的內容。一般來說，無點程式設計並不是本身的作用，而且可以為不是可以沉浸在樣式中的人員新增重要的認知屏障。

請勿在公用 **Api** 中使用部分應用程式和 **currying**

只要稍微例外，在公用 **Api** 中使用部分應用程式可能會對取用者造成混淆。`let` F# 程式碼中的系結值通常是

值，而不是函數值。將值與函式值混合在一起可能會導致在 exchange 中儲存少量的程式碼，以達相當多的認知負擔，特別是當結合運算子(例如) `>>` 來撰寫函數時。

考慮無點程式設計的工具含意

擴充函式不會標記其引數。這會影響工具。請考慮下列兩個功能：

```
let func name age =  
    printfn $"My name is {name} and I am %{age} years old!"  
  
let funcWithApplication =  
    printfn "My name is %s and I am %d years old!"
```

兩者都是有效的函式，但 `funcWithApplication` 是一個局部函數。當您將滑鼠停留在編輯器中的類型上時，就會看到：

```
val func : name:string -> age:int -> unit  
  
val funcWithApplication : (string -> int -> unit)
```

在呼叫位置，工具(例如 Visual Studio)中的工具提示會提供您類型簽章，但因為沒有定義任何名稱，所以不會顯示名稱。名稱對於良好的 API 設計很重要，因為它們可協助呼叫端更清楚瞭解 API 背後的意義。在公用 API 中使用無點程式碼，可讓呼叫端更難理解。

如果您遇到可公開取用的無點程式碼 `funcWithApplication`，則建議進行完整展開，讓工具可以針對引數使用有意義的名稱。

此外，如果不可能，則偵錯工具的程式碼可能會很困難。偵錯工具依賴系結至名稱的值(例如，系結 `let`)，以便您可以在執行時檢查中繼值。當您的程式碼沒有要檢查的值時，就不會有任何要進行的偵錯工具。未來，偵錯工具可能會發展，以根據先前執行的路徑來合成這些值，但不建議您在 *可能* 的偵錯工具上建立您的理由。

將部分應用程式視為減少內部重複使用的技術

相較於上一個重點，部分應用程式是一項絕佳的工具，可減少應用程式內的重複使用或更深入的 API 內部。對於執行更複雜 Api 的單元測試而言很有說明，因為未定案通常是很棘手的問題。例如，下列程式碼會示範如何完成大部分的模擬架構，而不需要對這類架構進行外部相依性，以及瞭解相關的定制 API。

例如，請考慮下列解決方案拓撲：

```
MySolution.sln  
|_ImplementationLogic.fsproj  
|_ImplementationLogic.Tests.fsproj  
|_API.fsproj
```

`ImplementationLogic.fsproj` 可能會公開程式碼，例如：

```
module Transactions =  
    let doTransaction txnContext txnType balance =  
        ...  
  
    type Transactor(ctx, currentBalance) =  
        member _.ExecuteTransaction(txnType) =  
            Transactions.doTransaction ctx txnType currentBalance  
        ...
```

中的單元測試 `Transactions.doTransaction` `ImplementationLogic.Tests.fsproj` 很簡單：

```
namespace TransactionsTestingUtil

open Transactions

module TransactionsTestable =
    let getTestableTransactionRoutine mockContext = Transactions.doTransaction mockContext
```

部分套用 `doTransaction` 模擬內容物件可讓您在所有單元測試中呼叫函式，而不需要每次都要建立模擬內容：

```
namespace TransactionTests

open Xunit
open TransactionTypes
open TransactionsTestingUtil
open TransactionsTestingUtil.TransactionsTestable

let testableContext =
    { new ITransactionContext with
        member _.TheFirstMember() = ...
        member _.TheSecondMember() = ... }

let transactionRoutine = getTestableTransactionRoutine testableContext

[<Fact>]
let ``Test withdrawal transaction with 0.0 for balance``() =
    let expected = ...
    let actual = transactionRoutine TransactionType.Withdraw 0.0
    Assert.Equal(expected, actual)
```

這項技術不應該通用地套用至整個程式碼基底，但這是減少複雜內部和單元測試這些內部的好方法。

存取控制

F# 有多個 [存取控制](#) 選項，可從 .net 執行時間中的可用內容繼承。這些都不只適用於型別，您也可以將它們用於函數。

- 偏好非 `public` 類型和成員，直到您需要它們可供公開取用為止。這也可減少取用者的需求。
- 努力保留所有協助程式功能 `private`。
- 請考慮在 helper 函式的 `[<AutoOpen>]` 私用模組上使用，如果它們變成很多。

型別推斷和泛型

型別推斷可以節省您輸入許多未定案的內容。F# 編譯器中的自動一般化可協助您撰寫更多的一般程式碼，幾乎不需要額外投入您的部分。不過，這些功能並不普遍。

- 請考慮在公用 Api 中以明確類型標記引數名稱，而不依賴此的型別推斷。

這是因為您應該控制 API 的圖形，而不是編譯器的形式。雖然編譯器可以在推斷類型時進行精確的工作，但如果它所依賴的本質具有變更類型，則您的 API 圖形可能會變更。這可能是您想要的結果，但它幾乎會產生突破性的 API 變更，下游取用者就必須處理這些變更。相反地，如果您明確控制公用 API 的形狀，就可以控制這些重大變更。在 DDD 方面，可以將這視為反損毀層。

- 請考慮為您的泛型引數提供有意義的名稱。

除非您要撰寫非特定網域專屬的真正泛型程式碼，否則有意義的名稱可協助其他程式設計人員瞭解他們所使用的網域。例如，在與檔資料庫互動的內容中命名的型別參數，可讓您所使用的函式 `'Document` 或成員接受一般檔案類型更加清楚。

- 請考慮使用 PascalCase 來命名泛型型別參數。

這是在 .NET 中執行作業的一般方式，因此建議您使用 PascalCase，而不是 snake_case 或 camelCase。

最後，對於 F# 或大型程式碼基底的人員而言，自動一般化不一定是 boon。使用一般的元件時，有認知負擔。此外，如果自動一般化的函式不會搭配不同的輸入類型使用（請單獨使用這些函式，以作為這類），如此一來，這些函式就不會在該時間點成為泛型。請一律考慮您所撰寫的程式碼是否會真正受益于泛型。

效能

針對具有高配置速率的小型類型考慮結構

使用結構（也稱為實值型別）通常可以針對某些程式碼產生更高的效能，因為它通常可避免設定物件。不過，結構不一定是「更快速」的按鈕：如果結構中的資料大小超過16個位元組，複製資料通常可能會比使用參考型別更多的 CPU 時間。

若要判斷是否應該使用結構，請考慮下列條件：

- 如果您的資料大小是16個位元組或更小。
- 如果您可能有許多這些類型的實例在執行中程式的記憶體中。

如果第一個條件適用，您通常應該使用結構。如果兩者都適用，您幾乎都應該使用結構。在某些情況下，您可能會在某些情況下套用先前的條件，但使用結構不會比使用參考型別更好或更糟，但很可能很罕見。不過，一定要在進行這類變更時進行測量，而不是在假設或直覺上運作。

以高配置速率分組較小數值型別時，請考慮結構元組

請考慮下列兩個功能：

```
let rec runWithTuple t offset times =
    let offsetValues x y z offset =
        (x + offset, y + offset, z + offset)

    if times <= 0 then
        t
    else
        let (x, y, z) = t
        let r = offsetValues x y z offset
        runWithTuple r offset (times - 1)

let rec runWithStructTuple t offset times =
    let offsetValues x y z offset =
        struct(x + offset, y + offset, z + offset)

    if times <= 0 then
        t
    else
        let struct(x, y, z) = t
        let r = offsetValues x y z offset
        runWithStructTuple r offset (times - 1)
```

當您使用 [BenchmarkDotNet](#) 之類的統計效能評定工具來評定這些函式時，您會發現 `runWithStructTuple` 使用結構元組的函式會以更快的速度執行40%，並且不會配置任何記憶體。

不過，在您自己的程式碼中，這些結果不一定是如此。如果您將函式標示為 `inline`，則使用參考元組的程式碼可能會取得一些額外的優化，或可配置的程式碼可能只會經過優化。當效能考慮時，您應該一律測量結果，而且永遠不會根據假設或直覺來操作。

當類型很小且配置速率很高時，請考慮結構記錄

稍早所述的經驗法則也保留 [F# 記錄類型](#)。請考慮下列用來處理這些資料類型的資料類型和函數：

```

type Point = { X: float; Y: float; Z: float }

[<Struct>]
type SPoint = { X: float; Y: float; Z: float }

let rec processPoint (p: Point) offset times =
    let inline offsetValues (p: Point) offset =
        { p with X = p.X + offset; Y = p.Y + offset; Z = p.Z + offset }

    if times <= 0 then
        p
    else
        let r = offsetValues p offset
        processPoint r offset (times - 1)

let rec processStructPoint (p: SPoint) offset times =
    let inline offsetValues (p: SPoint) offset =
        { p with X = p.X + offset; Y = p.Y + offset; Z = p.Z + offset }

    if times <= 0 then
        p
    else
        let r = offsetValues p offset
        processStructPoint r offset (times - 1)

```

這類似于先前的元組程式碼，但這次範例會使用記錄和內嵌的內建函式。

當您使用 [BenchmarkDotNet](#) 之類的統計效能評定工具來對這些函式進行基準測試時，您會發現

`processStructPoint` 執行速度快到60%，而且在 managed 堆積上未配置任何內容。

當資料類型很小且具有高配置速率時，請考慮結構區分等位

先前有關結構元組和記錄效能的觀察也保留給 [F #](#) 差異聯集。請考慮下列程式碼：

```

type Name = Name of string

[<Struct>]
type SName = SName of string

let reverseName (Name s) =
    s.ToCharArray()
    |> Array.rev
    |> string
    |> Name

let structReverseName (SName s) =
    s.ToCharArray()
    |> Array.rev
    |> string
    |> SName

```

通常會針對網域模型定義像這樣的單一案例差異聯集。當您使用 [BenchmarkDotNet](#) 之類的統計效能評定工具來對這些函式進行基準測試時，您會發現其 `structReverseName` 執行速度比小型字串快 25% `reverseName`。針對大型字串，兩者都會執行相同的。因此，在此情況下，最好是使用結構。如先前所述，一律測量並不會在假設或直覺上運作。

雖然先前的範例顯示結構差異聯集產生較佳的效能，但在建立定義域模型時，通常會有較大的區分等位。較大的資料類型（例如，如果它們是結構，視其上的作業而定）可能不會同時執行，因為可能會涉及更多的複製。

功能性程式設計和變化

F # 值預設為固定值，可讓您避免特定類別的 bug（特別是）平行存取和平行處理原則的類別。不過，在某些情況下，若要達到最佳（，或甚至合理）執行時間或記憶體配置的效率，最好是使用狀態的就地變化來實行工作範圍。

您可以使用關鍵字搭配 F # 來加入宣告。 `mutable`

`mutable` 在 F # 中使用可能會覺得功能純度很可能。這是可理解的，但功能的純度可能會有效能目標。折衷的是封裝變化，讓呼叫端不需要在意呼叫函式時會發生什麼事。這可讓您針對效能關鍵程式碼，透過以變化為基礎的執行來撰寫功能介面。

將可變的程式碼包裝在不可變的介面中

使用參考透明度做為目標時，請務必撰寫不會公開效能關鍵函式之可變 `underbelly` 的程式碼。例如，下列程式碼會 `Array.contains` 在 F # 核心程式庫中實作為函式：

```
[<CompiledName("Contains")>]
let inline contains value (array:'T[]) =
    checkNotNull "array" array
    let mutable state = false
    let mutable i = 0
    while not state && i < array.Length do
        state <- value = array.[i]
        i <- i + 1
    state
```

多次呼叫此函式並不會變更基礎陣列，也不會要求您維持任何使用中的可變狀態。它是參考上透明的，雖然幾乎每一行程式碼都會使用變化。

考慮將可變數據封裝在類別中

先前的範例使用單一函式來封裝使用可變動資料的作業。對於更複雜的資料集而言，這不一定足夠。請考慮下列函數集：

```
open System.Collections.Generic

let addToClosureTable (key, value) (t: Dictionary<_,>) =
    if not (t.ContainsKey(key)) then
        t.Add(key, value)
    else
        t.[key] <- value

let closureTableCount (t: Dictionary<_,>) = t.Count

let closureTableContains (key, value) (t: Dictionary<_, HashSet<_>>) =
    match t.TryGetValue(key) with
    | (true, v) -> v.Equals(value)
    | (false, _) -> false
```

這段程式碼的效能很高，但它會公開呼叫端負責維護的變化式資料結構。這可以包裝在類別內，但不能變更任何基礎成員：

```
open System.Collections.Generic

/// The results of computing the LALR(1) closure of an LR(0) kernel
type Closure1Table() =
    let t = Dictionary<Item0, HashSet<TerminalIndex>>()

    member _.Add(key, value) =
        if not (t.ContainsKey(key)) then
            t.Add(key, value)
        else
            t.[key] <- value

    member _.Count = t.Count

    member _.Contains(key, value) =
        match t.TryGetValue(key) with
        | (true, v) -> v.Equals(value)
        | (false, _) -> false
```

`Closure1Table` 封裝基礎以變化為基礎的資料結構，因此不會強制呼叫者維護基礎資料結構。類別是一種功能強大的方法，可封裝以變化為基礎的資料和常式，而不會向呼叫端公開詳細資料。

偏好 `let mutable` 參考資料格

參考資料格是一種表示值參考的方法，而不是值本身。雖然它們可用於效能關鍵的程式碼，但不建議使用。請考慮下列範例：

```
let kernels =
    let acc = ref Set.empty

    processWorkList startKernels (fun kernel ->
        if not ((!acc).Contains(kernel)) then
            acc := (!acc).Add(kernel)
        ...)

!acc |> Seq.toList
```

使用參考儲存格現在會「干擾」所有後續的程式碼，而且必須對基礎資料進行取值和重新參考。請改為考慮：

```
let kernels =
    let mutable acc = Set.empty

    processWorkList startKernels (fun kernel ->
        if not (acc.Contains(kernel)) then
            acc <- acc.Add(kernel)
        ...)

acc |> Seq.toList
```

除了 lambda 運算式中間的單一變化點之外，其他所有接觸的程式碼都 `acc` 可以使用與一般系結不可變的值不同的方式來達成 `let`。這可讓您更輕鬆地隨著時間變更。

物件程式設計

F# 對於物件和麵向物件的 (OO) 概念具有完整的支援。雖然許多 OO 概念都很強大且很有用，但並非全部都適合使用。下列清單提供高階各類 OO 功能的指引。

在許多情況下，請考慮使用這些功能：

- 點標記法 (`x.Length`)
- 執行個體成員
- 隱含的函式
- 靜態成員
- 索引子標記法 (`arr.[x]`)
- 指名的和選擇性引數
- 介面和介面的實現

請先找不到這些功能，但在方便解決問題時，請謹慎套用這些功能：

- 方法多載
- 封裝的可變數據
- 類型上的運算子
- Auto 屬性
- 執行 `IDisposable` 和 `IEnumerable`
- 類型延伸模組
- 事件
- 結構
- 委派
- 列舉

一般來說，除非您必須使用這些功能，否則請避免這些功能：

- 繼承型型別階層和執行繼承
- Null 和 `Unchecked.defaultof<_>`

偏好透過繼承撰寫

[超越繼承的組合](#) 是良好的 F# 程式碼可以遵守的長期用法。基本原則是，您不應該公開基類，並強制呼叫端繼承自該基類以取得功能。

如果您不需要類別，請使用物件運算式來執行介面

[物件運算式](#) 可讓您即時執行介面，將實介面系結至值，而不需要在類別內部執行此動作。這很方便，尤其是當您只需要執行介面，且不需要完整類別時更是如此。

例如，以下是在 [Ionide](#) 中執行的程式碼，如果您已新增沒有語句的符號，則會提供程式碼修正動作 `open`：

```
let private createProvider () =
    { new CodeActionProvider with
        member this.provideCodeActions(doc, range, context, ct) =
            let diagnostics = context.diagnostics
            let diagnostic = diagnostics |> Seq.tryFind (fun d -> d.message.Contains "Unused open statement")

            let res =
                match diagnostic with
                | None -> [[]]
                | Some d ->
                    let line = doc.lineAt d.range.start.line
                    let cmd = createEmpty<Command>
                    cmd.title <- "Remove unused open"
                    cmd.command <- "fsharp.unusedOpenFix"
                    cmd.arguments <- Some ([| doc |> unbox; line.range |> unbox; []] |> ResizeArray)
                    [|cmd|]

            res
            |> ResizeArray
            |> U2.Case1
    }
```

由於與 Visual Studio Code API 互動時，不需要類別，因此物件運算式是理想的工具。當您想要以臨機操作的方式，以測試常式來取代介面時，它們也非常適合單元測試。

請考慮輸入縮寫以縮短簽章

類型縮寫 是將標籤指派給另一種類型的便利方法，例如函式簽章或更複雜的型別。例如，下列別名會將標籤指派給使用 **CNTK**(深度學習程式庫) 定義計算所需的內容：

```
open CNTK

// DeviceDescriptor, Variable, and Function all come from CNTK
type Computation = DeviceDescriptor -> Variable -> Function
```

`Computation` 名稱是一種便利的方式，用來表示任何符合其別名的簽章的函式。使用這類的類型縮寫很方便，而且可讓程式碼更簡潔。

避免使用類型縮寫來表示您的網域

雖然型別縮寫對於為函式簽章提供名稱很方便，但在縮寫其他類型時可能會造成混淆。請考慮以下縮寫：

```
// Does not actually abstract integers.
type BufferSize = int
```

這可能會讓您以多種方式造成混淆：

- `BufferSize` 不是抽象概念;這只是整數的另一個名稱。
- 如果 `BufferSize` 在公用 API 中公開，就可以輕易地將它誤解為非單純的表示 `int`。一般而言，網欄位型別具有多個屬性，而且不是基本類型，例如 `int`。此縮寫違反該假設。
- (的大小寫 `BufferSize`) 表示此型別會保存更多資料。
- 相較于將具名引數提供給函式，此別名不會提供更高的清晰度。
- 縮寫不會在編譯的 IL 中資訊清單;它只是一個整數，而此別名是編譯時期的結構。

```
module Networking =
    ...
    let send data (bufferSize: int) = ...
```

總而言之，類型縮寫的缺點是它們在縮寫的類型上 **不** 是抽象概念。在上述範例中，`BufferSize` 只是 `int` 在幕後，沒有額外的資料，也不會有任何來自型別系統的優點，除了已有的內容之外 `int`。

使用類型縮寫來表示網域的替代方法是使用單一案例差異聯集。先前的範例可以模型化，如下所示：

```
type BufferSize = BufferSize of int
```

如果您撰寫的程式碼會以 `BufferSize` 和其基礎值運作，您必須建立一個，而不是傳入任何任意整數：

```
module Networking =
    ...
    let send data (BufferSize size) =
    ...
```

這樣可以減少錯誤地將任意整數傳遞給函式的可能性 `send`，因為呼叫端必須在 `BufferSize` 呼叫函式之前，先建立型別來包裝值。

F# 元件設計指導

2021/3/5 • [Edit Online](#)

本檔是一組 F# 程式設計的元件設計方針，以 F# 元件設計指導方針、v14、Microsoft Research 以及 F# Software Foundation 原本策劃和維護的版本為基礎。

本檔假設您已熟悉 F# 程式設計。很多人都感謝 F# 團體參與本指南的各種版本的投稿和實用意見反應。

總覽

本檔探討一些與 F# 元件設計和程式碼相關的問題。元件可以表示下列其中一項：

- 在 F# 專案中，具有該專案內外部取用者的圖層。
- 適用於跨元件界限的 F# 程式碼耗用量的程式庫。
- 一種程式庫，可供跨元件界限的任何 .NET 語言取用。
- 適用於透過套件存放庫（例如 [NuGet](#)）散發的程式庫。

本文所述的技術會遵循 [良好 F# 程式碼的五個準則](#)，因此會適當地使用功能和物件程式設計。

無論是哪一種方法，元件和程式庫設計師在嘗試製作最容易使用的 API 時，都面臨許多實際和平凡的問題。Conscientious 應用程式的 [.net 程式庫設計指導方針](#)，將引導您建立一組一致的 api，以供使用。

一般指導方針

F# 程式庫有一些適用的通用指導方針，而不考慮程式庫的目標物件。

瞭解 .NET 程式庫設計指導方針

無論您正在進行的 F# 程式碼撰寫種類為何，都有很大的說明，讓您瞭解 [.net 程式庫設計指導方針](#)。大部分其他 F# 和 .NET 程式設計人員都將熟悉這些指導方針，並預期 .NET 程式碼必須符合這些方針。

.NET 程式庫設計指導方針提供有關命名、設計類別和介面、成員設計（屬性、方法、事件等）等等的一般指引，而且是各種設計指導的實用第一個參考重點。

將 XML 檔批註新增至您的程式碼

適用於公用 Api 的 XML 檔可確保使用者在使用這些類型和成員時，可以取得絕佳的 Intellisense 和 Quickinfo，並為文件庫啟用建立檔檔。請參閱 [Xml 檔](#)，瞭解可用於 xmldoc 批註內其他標記的各種 xml 標記。

```
/// A class for representing (x,y) coordinates
type Point =

    /// Computes the distance between this point and another
    member DistanceTo: otherPoint:Point -> float
```

您可以使用簡短形式的 XML 批註 (`/// comment`)，或 () 的標準 XML 批註 `///<summary>comment</summary>`。

請考慮針對穩定的程式庫和元件 Api 使用明確的簽章檔案 (fsi)。

使用 F# 程式庫中的明確簽章檔案可提供公用 API 的簡潔摘要，這有助於確保您知道媒體櫃的完整公開介面，並提供公開檔和內部實作為詳細資料的清楚分隔。簽章檔案會藉由要求在執行和簽章檔案中進行變更，而增加了變更公用 API 的分歧。如此一來，通常只會在 API 已 solidified，且不再需要大幅變更時，才會引進簽名檔。

一律遵循在 .NET 中使用字串的最佳作法

遵循 [在 .net 中使用字串的最佳作法](#) 指引。尤其是，在適當的) 的情況下，一律明確地在字串轉換和比較 (中陳述

文化意圖。

F # 面向程式庫的指導方針

本節說明開發公用 F # 面向程式庫的建議;也就是說, 程式庫會公開公用 Api, 供 F # 開發人員使用。針對 F #, 有各種適用的程式庫設計建議。如果沒有遵循的特定建議, .NET 程式庫設計指導方針就是回溯指引。

命名規範

使用 .NET 命名和大小寫慣例

下表會遵循 .NET 命名和大小寫慣例。另外還包含 F # 結構的新增功能。

「	「	「	「	「
具體類型	PascalCase	名詞/形容詞	清單、雙重、複雜	具體類型是結構、類別、列舉、委派、記錄和等位。雖然型別名稱在 OCaml 中通常是小寫, 但 F # 已採用類型的 .NET 命名配置。
DLL	PascalCase		Fabrikam.Core.dll	
聯集標記	PascalCase	名詞	部分、新增、成功	請勿在公用 Api 中使用前置詞。(選擇性) 在內部使用前置詞, 例如 <div>type Teams = TAlpha TBeta TDelta.</div>
事件	PascalCase	動詞命令	ValueChanged/Value Changing	
例外狀況	PascalCase		WebException	名稱的結尾應該是「例外狀況」。
欄位	PascalCase	名詞	CurrentName	
介面型別	PascalCase	名詞/形容詞	IDisposable	名稱應該以 "I" 開頭。
方法	PascalCase	動詞命令	ToString	
命名空間	PascalCase		Fsharp.core 核心	一般來說 <div><Organization>. <Technology>[. <Subnamespace>]</div> , 如果該技術與組織無關, 就會捨棄組織。
參數	camelCase	名詞	typeName、轉換、範圍	
let 值 (內部)	camelCase 或 PascalCase	名詞/動詞	getValue、myTable	

⌈	⌈	⌈	⌈	⌈
讓值 (外部)	camelCase 或 PascalCase	名詞/動詞	List. map, 日期, Today	在遵循傳統的功能設計模式時, let 系結值通常是公用的。不過, 當識別碼可以從其他 .NET 語言使用時, 通常會使用 PascalCase。
屬性	PascalCase	名詞/形容詞	IsEndOfFile、背景色彩	布林值屬性通常使用的是, 而且應該是肯定, 如同在 IsEndOfFile 中, 不會 IsNotEndOfFile。

避免縮寫

.NET 指導方針不鼓勵使用縮寫 (例如「使用 `OnClick` 而非 `OnBtnClick`)。常見的縮寫 (例如「 `Async` 非同步」) 是容許的。這項指導方針有時會被忽略, 以進行功能性程式設計; 例如, 會 `List.iter` 使用 "反覆運算" 的縮寫。基於這個理由, 使用縮寫通常可以在 F # 對 F # 程式設計中獲得更高的程度, 但仍應避免在公用元件設計中使用。

避免大小寫名稱衝突

.NET 指導方針表示單獨使用大小寫, 因為某些用戶端語言 (例如 Visual Basic) 不區分大小寫。

適當時使用縮寫

XML 之類的縮寫不是縮寫, 而是廣泛用於 .NET 程式庫中的 uncapitalized 格式 (Xml) 。只應使用知名且廣泛辨識的縮寫。

使用 PascalCase 進行泛型參數名稱

請在公用 Api 中使用泛型參數名稱的 PascalCase, 包括 F # 面向的程式庫。尤其是, 針對 `T` `U` 任意泛型參數使用、`T1` `T2` 等名稱, 而且當特定名稱有意義時, 對於 F # 對應的程式庫, 請使用如 `Key` 、 `Value` 、 `Arg` (但不是 `TKey`) 的名稱。

將 PascalCase 或 camelCase 用於 F # 模組中的公用函式和值

camelCase 適用於設計為使用非限定 (的公用函式, 例如 `invalidArg`), 以及「標準集合函式」 (例如, List. map) 。在這兩種情況下, 函式名稱的作用相當於語言中的關鍵字。

物件、類型和模組設計

使用命名空間或模組來包含您的類型和模組

元件中的每個 F # 檔案都應該以命名空間宣告或模組宣告做為開頭。

```
namespace Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
    ...

module CommonOperations =
    ...
```

或

```

module Fabrikam.BasicOperationsAndTypes

type ObjectType1() =
    ...

type ObjectType2() =
    ...

module CommonOperations =
    ...

```

使用模組與命名空間來組織最上層程式碼的差異如下：

- 命名空間可以橫跨多個檔案
- 命名空間不能包含 F # 函數，除非它們在內部模組內
- 任何指定模組的程式碼都必須包含在單一檔案中
- 最上層模組可以包含 F # 函式，而不需要內部模組

最上層命名空間或模組之間的選擇會影響程式碼的編譯格式，因此，如果您的 API 最終是在 F # 程式碼之外使用，則會影響其他 .NET 語言的視圖。

針對物件類型的作業使用方法和屬性

使用物件時，最好確保取用的功能會實作為該類型上的方法和屬性。

```

type HardwareDevice() =

    member this.ID = ...

    member this.SupportedProtocols = ...

type HashTable<'Key, 'Value>(comparer: IEqualityComparer<'Key>) =

    member this.Add(key, value) = ...

    member this.ContainsKey(key) = ...

    member this.ContainsValue(value) = ...

```

指定成員的大量功能不一定要在該成員中實作為，但該功能的取用部分應該是。

使用類別封裝可變動狀態

在 F # 中，這只需要在尚未由另一種語言結構(例如，關閉、序列運算式或非同步計算)封裝的狀態下完成。

```

type Counter() =
    // let-bound values are private in classes.
    let mutable count = 0

    member this.Next() =
        count <- count + 1
        count

```

使用介面來分組相關的作業

使用介面類別型來代表一組作業。這是其他選項的慣用選項，例如函數的元組或函式的記錄。

```

type Serializer =
    abstract Serialize<'T> : preserveRefEq: bool -> value: 'T -> string
    abstract Deserialize<'T> : preserveRefEq: bool -> pickle: string -> 'T

```

依喜好設定：

```

type Serializer<'T> = {
    Serialize: bool -> 'T -> string
    Deserialize: bool -> string -> 'T
}

```

介面是 .NET 中的第一級概念，您可以用來達成函子一般提供的功能。此外，它們也可以用來將存在類型編碼到您的程式中，而無法使用哪些功能記錄。

使用模組將處理集合的函式分組

當您定義集合類型時，請考慮為新的集合類型提供一組標準的作業，例如 `CollectionType.map` 和 `CollectionType.iter`)。

```

module CollectionType =
    let map f c =
        ...
    let iter f c =
        ...

```

如果您包含這類別模組，請遵循 Fsharp.core 中所找到函式的標準命名慣例。

使用模組來分組一般標準函式的功能，特別是在數學和 DSL 程式庫中

例如，`Microsoft.FSharp.Core.Operators` 會自動開啟最上層函式的集合 (例如 `abs` 和 FSharp.Core.dll 所 `sin` 提供的)。

同樣地，統計資料連結庫可能包含具有函式的模組，`erf` 以及 `erfc` 此模組是設計為明確或自動開啟的。

考慮使用 `RequireQualifiedAccess` 並小心套用 `AutoOpen` 屬性

將 `[<RequireQualifiedAccess>]` 屬性新增至模組，表示模組可能無法開啟，而且模組的元素參考需要明確的存取權。例如，`Microsoft.FSharp.Collections.List` 模組有這個屬性。

當模組中的函式和值有可能與其他模組中的名稱衝突的名稱時，這非常有用。需要限定存取權可能會大幅增加資源庫的長期維護和 evolvability。

將 `[<AutoOpen>]` 屬性加入至模組時，表示當包含的命名空間開啟時，將會開啟模組。`[<AutoOpen>]` 屬性也可以套用至元件，以指出參考元件時自動開啟的模組。

例如，統計資料連結庫 MathsHeaven。統計資料 可能包含 `module MathsHeaven.Statistics.Operators` 包含函數 `erf` 和 `erfc`。將此模組標示為是合理的 `[<AutoOpen>]`。這表示 `open MathsHeaven.Statistics` 也會開啟此模組，並將名稱 `erf` 和移 `erfc` 至範圍中。的另一種用法 `[<AutoOpen>]` 是針對包含擴充方法的模組。

過度 `[<AutoOpen>]` 使用潛在客戶污染命名空間，而屬性應謹慎使用。針對特定網域中的特定程式庫，合理使用 `[<AutoOpen>]` 可能會導致更好的可用性。

請考慮在適當使用已知運算子的類別上定義運算子成員

有時類別會用來建立數學結構 (例如向量) 的模型。當模型化的網域有已知運算子時，將它們定義為類別內建的成員會很有說明。

```

type Vector(x: float) =

    member v.X = x

    static member (*) (vector: Vector, scalar: float) = Vector(vector.X * scalar)

    static member (+) (vector1: Vector, vector2: Vector) = Vector(vector1.X + vector2.X)

let v = Vector(5.0)

let u = v * 10.0

```

本指南對應于這些類型的一般 .NET 指引。不過，這在 F# 程式碼中可能也很重要，因為這可讓這些型別搭配使用 F# 函式和具有成員條件約束的方法，例如 `sumBy`。

請考慮使用 **CompiledName** 來提供。其他 .NET 語言取用者的網路易記名稱

有時您可能會想要為 F# 取用者以一種樣式來命名某個樣式 (例如，以小寫的成員形式出現，使其看起來像是模組系結函式)，但在編譯成元件時，名稱會有不同的樣式。您可以使用 `[<CompiledName>]` 屬性，為使用元件的非 F# 程式碼提供不同的樣式。

```

type Vector(x:float, y:float) =

    member v.X = x
    member v.Y = y

    [<CompiledName("Create")>]
    static member create x y = Vector (x, y)

let v = Vector.create 5.0 3.0

```

藉由使用 `[<CompiledName>]`，您可以針對元件的非 F# 取用者使用 .net 命名慣例。

使用成員函式的方法多載，如果這樣做會提供更簡單的 API

方法多載是一種功能強大的工具，可簡化可能需要執行類似功能的 API，但使用不同的選項或引數。

```

type Logger() =

    member this.Log(message) =
        ...
    member this.Log(message, retryPolicy) =
        ...

```

在 F# 中，在引數數目而非引數類型上多載更為常見。

如果這些類型的設計可能會進化，則隱藏記錄和聯集類型的表示

避免洩漏物件的具體標記法。例如，`DateTime.net` 程式庫設計的外部公用 API 不會顯示值的具體標記法。在執行時間，通用語言執行平臺會知道將在執行期間使用的認可的實作為。不過，已編譯的程式碼不會自行收取具體標記法的相依性。

避免使用擴充性的實作為繼承

在 F# 中，很少使用執行繼承。此外，繼承階層通常很複雜，而且在新的需求抵達時很難變更。在 F# 中，繼承實行仍存在於 F# 中，以提供最適合問題的解決方案，但在設計多型 (例如介面執行) 時，您應該在 F# 程式中尋找替代的技巧。

函式和成員特徵標記

傳回少量的多個不相關值時，使用元組來傳回值

以下是在傳回型別中使用元組的絕佳範例：

```
val divrem: BigInteger -> BigInteger -> BigInteger * BigInteger
```

針對包含許多元件的傳回型別，或元件與單一可識別實體相關的位置，請考慮使用命名的型別，而不是元組。

`Async<T>` 在 F # API 界限上用於非同步程式設計

如果有一個名為的對應同步作業會傳回 `Operation T`，則非同步作業應命名為，`AsyncOperation` 如果它傳回 `Async<T>` 或傳回 `OperationAsync Task<T>`。對於公開 Begin/End 方法的常用 .NET 型別，請考慮使用 `Async.FromBeginEnd` 將擴充方法撰寫為外觀，以提供 F # 非同步程式設計模型給這些 .Net api。

```
type SomeType =
    member this.Compute(x:int): int =
        ...
    member this.AsyncCompute(x:int): Async<int> =
        ...

type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        ...
```

例外狀況

請參閱 [錯誤管理](#)，以瞭解例外狀況、結果和選項的適當用法。

延伸模組成員

謹慎地在 F # 到 F # 元件中套用 F # 擴充功能成員

F # 延伸模組成員通常只適用於在大部分使用模式中，與型別相關聯之內建作業的封閉作業。其中一個常見用途是提供針對各種 .NET 類型更慣用至 F # 的 Api：

```
type System.ServiceModel.Channels.IInputChannel with
    member this.AsyncReceive() =
        Async.FromBeginEnd(this.BeginReceive, this.EndReceive)

type System.Collections.Generic.IDictionary<'Key,'Value> with
    member this.TryGet key =
        let ok, v = this.TryGetValue key
        if ok then Some v else None
```

聯集類型

針對樹狀結構資料使用差異聯集，而非類別階層

以遞迴方式定義類似樹狀結構的結構。這對繼承很麻煩，但是具有差異聯集的優雅。

```
type BST<'T> =
    | Empty
    | Node of 'T * BST<'T> * BST<'T>
```

以差異聯集表示類似樹狀結構的資料也可讓您在模式比對中受益於 exhaustiveness。

使用 `[<RequireQualifiedAccess>]` 案例名稱不是足夠唯一的等位類型

您可能會發現某個網域中的相同名稱是不同專案的最佳名稱，例如差異聯集案例。您可以使用

`[<RequireQualifiedAccess>]` 來區分大小寫名稱，以避免因為因遮蔽而相依赖于語句順序而觸發混淆錯誤 [open](#)

如果這些類型的設計可能會進化，請隱藏二進位相容性聯集的區分等位表示

等位型別依賴 F # 模式比對形式來提供簡潔的程式設計模型。如先前所述，如果這些類型的設計可能會進化，您應該避免暴露具體的資料標記法。

例如，您可以使用私用或內部宣告或使用簽章檔案來隱藏差異聯集的標記法。

```
type Union =  
    private  
    | CaseA of int  
    | CaseB of string
```

如果您不小心地顯示差異聯集，您可能會發現不需要中斷使用者程式碼就能為您的程式庫進行版本。相反地，請考慮公開一或多個使用中的模式，以允許您類型值的模式比對。

現用模式提供了一種替代方式，可提供 F# 取用者與模式比對，同時避免直接公開 F# 等位類型。

內嵌函式和成員條件約束

使用具有隱含成員條件約束和靜態解析泛型型別的內嵌函式來定義泛型數值演算法

算術成員條件約束和 F# 比較準則約束是 F# 程式設計的標準。例如，請參考下列程式碼：

```
let inline highestCommonFactor a b =  
    let rec loop a b =  
        if a = LanguagePrimitives.GenericZero<_> then b  
        elif a < b then loop a (b - a)  
        else loop (a - b) b  
    loop a b
```

此函式的類型如下所示：

```
val inline highestCommonFactor : ^T -> ^T -> ^T  
    when ^T : (static member Zero : ^T)  
    and ^T : (static member ( - ) : ^T * ^T -> ^T)  
    and ^T : equality  
    and ^T : comparison
```

這是適用於數學程式庫中公用 API 的功能。

避免使用成員條件約束來模擬型別類別和打字型別

您可以使用 F# 成員條件約束來模擬「打字打字」。不過，使用此功能的成員通常不應該用於 F# 對 F# 程式庫設計中。這是因為根據不熟悉或非標準隱含條件約束的程式庫設計，通常會導致使用者的程式碼變差，並系結至一個特定的架構模式。

此外，在這種情況下大量使用成員條件約束很有可能會導致編譯時間很長。

運算子定義

避免定義自訂符號運算子

自訂運算子在某些情況下是不可或缺的，而且在大型的實作為程式碼主體內是非常有用的標記裝置。針對程式庫的新使用者，命名函式通常更容易使用。此外，自訂符號運算子可能很難記載，而且使用者發現因為 IDE 和搜尋引擎有現有的限制，所以更難以查閱運算子的說明。

因此，最好以命名函式和成員的形式發行您的功能，並在標記權益超過檔和擁有這些專案的認知成本時，另外公開此功能的運算子。

測量單位

針對 F# 程式碼中新增的型別安全，謹慎使用測量單位

其他 .NET 語言查看時，會清除度量單位的其他輸入資訊。請注意，.NET 元件、工具和反映將會看到類型-san 單位。例如，c# 取用者會看到 `float` 而不是 `float<kg>`。

類型縮寫

謹慎使用類型縮寫以簡化 F# 程式碼

.NET 元件、工具和反映將不會看到類型的縮寫名稱。類型縮寫的顯著用法也可能使定義域比實際的更複雜，這可能會讓取用者混淆。

避免其成員和屬性本質上不同于要縮寫之類型上的公用類型的類型縮寫

在此情況下，要縮寫的型別會顯示太多關於所定義之實際型別的標記法。相反地，請考慮將縮寫包裝在類別類型或單一案例的差異聯集 (或者，當效能為必要時，請考慮使用結構類型將縮寫) 包裝。

例如，將多個地圖定義為 F # 地圖的特殊案例很吸引人，例如：

```
type MultiMap<'Key, 'Value> = Map<'Key, 'Value list>
```

不過，這種類型上的邏輯點標記運算與地圖上的作業不同，例如，lookup 運算子對應是合理的。[key] 如果索引鍵不在字典中，則傳回空的清單，而不是引發例外狀況。

從其他 .NET 語言使用的程式庫指導方針

設計用來從其他 .NET 語言使用的程式庫時，請務必遵守 [.net 程式庫設計指導方針](#)。在本檔中，這些程式庫會標示為香草 .NET 程式庫，而非使用 F # 結構的 F # 對應程式庫，而不受限制。設計香草 .NET 程式庫表示讓熟悉且慣用的 Api 與其余 .NET Framework 保持一致，方法是將在公用 API 中使用 F # 特定的結構降至最低。下列各節將說明這些規則。

適用於程式庫的命名空間和型別設計 (可用於其他 .NET 語言)

將 .NET 命名慣例套用至元件的公用 API

請特別注意縮寫名稱和 .NET 大寫準則的使用。

```
type pCoord = ...
    member this.theta = ...

type PolarCoordinate = ...
    member this.Theta = ...
```

使用命名空間、類型和成員作為元件的主要組織結構

所有包含公用功能的檔案都應該以宣告開頭 `namespace`，且命名空間中的唯一公開實體應該是類型。請勿使用 F # 模組。

使用非公用模組來保存實作為程式碼、公用程式類型和公用程式函式。

靜態類型應優先於模組，因為它們可讓 API 的未來演進使用多載和其他可能不會在 F # 模組中使用的 .NET API 設計概念。

例如，取代下列公用 API：

```
module Fabrikam

module Utilities =
    let Name = "Bob"
    let Add2 x y = x + y
    let Add3 x y z = x + y + z
```

請改為考慮：

```
namespace Fabrikam

[<AbstractClass; Sealed>]
type Utilities =
    static member Name = "Bob"
    static member Add(x,y) = x + y
    static member Add(x,y,z) = x + y + z
```

如果類型的設計不會進化，請在香草 .NET Api 中使用 F # 記錄類型

F# 記錄類型會編譯成簡單的 .NET 類別。這些適用於 Api 中一些簡單、穩定的類型。請考慮使用 `[<NoEquality>]` 和 `[<NoComparison>]` 屬性來抑制自動產生介面。也請避免在香草 .NET Api 中使用可變動的記錄欄位，因為這些欄位會公開公用欄位。請一律考慮類別是否會為未來的 API 演進提供更有彈性的選項。

例如，下列 F# 程式碼會向 c# 取用者公開公用 API：

F#：

```
[<NoEquality; NoComparison>]
type MyRecord =
    { FirstThing: int
      SecondThing: string }
```

C#：

```
public sealed class MyRecord
{
    public MyRecord(int firstThing, string secondThing);
    public int FirstThing { get; }
    public string SecondThing { get; }
}
```

在香草 .NET Api 中隱藏 F# 聯集類型的標記法

F# 等位類型通常不會跨元件界限使用，即使是 F# 對 F# 程式碼。它們是在元件和程式庫內部使用的絕佳實作為裝置。

設計香草 .NET API 時，請考慮使用私用宣告或簽章檔案來隱藏聯集類型的表示。

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True
```

您也可以增強在內部使用聯集表示的類型，以提供所需的成員。 .NET 面向的 API。

```
type PropLogic =
    private
    | And of PropLogic * PropLogic
    | Not of PropLogic
    | True

    /// A public member for use from C#
    member x.Evaluate =
        match x with
        | And(a,b) -> a.Evaluate && b.Evaluate
        | Not a -> not a.Evaluate
        | True -> true

    /// A public member for use from C#
    static member CreateAnd(a,b) = And(a,b)
```

使用架構的設計模式來設計 GUI 和其他元件

.NET 中提供許多不同的架構，例如 WinForms、WPF 和 ASP.NET。如果您要設計要在這些架構中使用的元件，則應該使用每個的命名和設計慣例。例如，針對 WPF 程式設計，採用 WPF 設計模式來設計您所設計的類別。針對使用者介面程式設計中的模型，請使用如中所找到的設計模式，例如事件和以通知為基礎的集合 [System.Collections.ObjectModel](#)。

程式庫的物件和成員設計（以供其他 .NET 語言使用）

使用 `CLIEvent` 屬性來公開 .NET 事件

`DelegateEvent` 使用特定的 .net 委派型別來建立，此型別會採用物件並 `EventArgs` (，而不是 `Event` 使用預設的型別，而是使用 `FSharpHandler` 預設的型別) 讓事件以熟悉的方式發佈到其他 .net 語言。

```
type MyBadType() =
    let myEv = new Event<int>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish

type MyEventArgs(x: int) =
    inherit System.EventArgs()
    member this.X = x

    /// A type in a component designed for use from other .NET languages
type MyGoodType() =
    let myEv = new DelegateEvent<EventHandler<MyEventArgs>>()

    [<CLIEvent>]
    member this.MyEvent = myEv.Publish
```

將非同步作業公開為傳回 .NET 工作的方法

工作會在 .NET 中用來代表使用中的非同步計算。工作通常比 F # 物件的複合少 `Async<T>`，因為它們代表「已在執行」的工作，而且無法以執行平行組合的方式組合在一起，或隱藏取消信號和其他內容參數的傳播。

不過，無論如何，傳回工作的方法都是 .NET 上非同步程式設計的標準標記法。

```
/// A type in a component designed for use from other .NET languages
type MyType() =

    let compute (x: int): Async<int> = async { ... }

    member this.ComputeAsync(x) = compute x |> Async.StartAsTask
```

您通常也會想要接受明確的解除標記：

```
/// A type in a component designed for use from other .NET languages
type MyType() =
    let compute(x: int): Async<int> = async { ... }
    member this.ComputeAsTask(x, cancellation_token) = Async.StartAsTask(compute x, cancellation_token)
```

使用 .NET 委派型別，而非 F # 函式類型

這裡的「F # 函式類型」表示「箭號」類型 `int -> int`，例如。

而不是：

```
member this.Transform(f: int->int) =
    ...
```

執行此動作：

```
member this.Transform(f: Func<int,int>) =
    ...
```

F # 函式類型會顯示為 `class FSharpFunc<T,U>` 其他 .net 語言，而且較不適合瞭解委派類型的語言功能和工具。撰寫以 .NET Framework 3.5 或更高版本為目標的較高順序方法時，`System.Func` 和 `System.Action` 委派是正確發佈的 api，可讓 .net 開發人員以低摩擦的方式取用這些 api。(以 .NET Framework 2.0 為目標時，系統定義的委

派類型會受到更多限制,請考慮使用預先定義的委派類型, 例如 `System.Converter<T,U>` 或定義特定的委派類型。
)

另一方面, .NET 委派不是 F # 面向程式庫的自然 (請參閱下一節的 F # 面向程式庫)。因此, 開發香草 .NET 程式庫的高階方法時, 常見的採用策略是使用 F # 函式類型來撰寫所有的實作者, 然後使用委派作為實際 F # 實作為之上的精簡外觀來建立公用 API。

使用 `TryGetValue` 模式, 而不是傳回 F # 選項值, 而且偏好使用方法多載, 以 F # 選項值做為引數

Api 中 F # 選項類型的常見使用模式, 是使用標準 .NET 設計技術, 在香草 .NET Api 中進行更好的運用。請考慮使用 `bool` 傳回型別加上 `out` 參數, 如同 "TryGetValue" 模式, 而不是傳回 F # 選項值。而不是將 F # 選項值作為參數, 而是考慮使用方法多載或選擇性引數。

```
member this.ReturnOption() = Some 3

member this.ReturnBoolAndOut(outVal: byref<int>) =
    outVal <- 3
    true

member this.ParamOption(x: int, y: int option) =
    match y with
    | Some y2 -> x + y2
    | None -> x

member this.ParamOverload(x: int) = x

member this.ParamOverload(x: int, y: int) = x + y
```

使用 .NET 集合介面類型 `IEnumerable <T>` 和 `IDictionary <Key,Value>` 作為參數和傳回值

避免使用具象的集合類型, 例如 .NET 陣列、F # 型別和 .NET 具象的 `T[]` `list<T>` `Map<Key,Value>` `Set<T>` 集合類型 (例如) `Dictionary<Key,Value>`。 .NET 程式庫設計指導方針對於使用各種集合類型 (例如) 有很大的建議 `IEnumerable<T>`。在某些情況下, 您可以在效能的情況下, 使用陣列 () 的某些用途 `T[]`。請注意 `seq<T>`, 特別是 F # 別名 `IEnumerable<T>`, 因此 `seq` 通常是香草 .net API 的適當類型。

而非 F # 清單:

```
member this.PrintNames(names: string list) =
    ...
```

使用 F # 序列:

```
member this.PrintNames(names: seq<string>) =
    ...
```

您可以使用單位類型作為方法的唯一輸入類型來定義零引數方法, 或使用唯一的傳回型別來定義 `void` 傳回方法。

避免使用單位類型的其他用途。這些都很好:

```
✓ member this.NoArguments() = 3

✓ member this.ReturnVoid(x: int) = ()
```

這是不好的:

```
member this.WrongUnit( x: unit, z: int) = ((), ())
```

在香草 .NET API 界限上檢查是否有 `null` 值

F # 的程式碼通常會有較少的 `null` 值, 因為不可變的設計模式, 以及對 F # 類型使用 `null` 常值的限制。其他 .NET

語言通常會使用 null 做為值更頻繁。因此，公開香草 .NET API 的 F # 程式碼應該在 API 界限上檢查 null 的參數，並防止這些值更深入地流向 F # 程式碼。您 `isNull` 可以使用模式上的函數或模式比對 `null` 。

```
let checkNonNull argName (arg: obj) =
    match arg with
    | null -> nullArg argName
    | _ -> ()

let checkNonNull` argName (arg: obj) =
    if isNull arg then nullArg argName
    else ()
```

避免使用元組作為傳回值

相反地，最好是傳回持有匯總資料的命名型別，或使用 out 參數傳回多個值。雖然元組和結構元組存在於 .NET (包括結構元組) 的 c # 語言支援，但它們最常不提供適用於 .NET 開發人員的理想和預期 API。

避免使用參數的 currying

相反地，請使用 .NET 呼叫慣例 `Method(arg1,arg2,...,argN)` 。

```
member this.TupledArguments(str, num) = String.replicate num str
```

秘訣:如果您要設計可從任何 .NET 語言使用的程式庫，則不會實際執行一些實驗性 c # 和 Visual Basic 程式設計，以確保您的程式庫能夠從這些語言中「感覺正確」。您也可以使用 .NET 反映程式和 Visual Studio 物件瀏覽器之類的工具，以確保程式庫及其檔會如預期般出現給開發人員。

附錄

設計其他 .NET 語言使用之 F # 程式碼的端對端範例

請考慮下列類別：

```
open System

type Point1(angle,radius) =
    new() = Point1(angle=0.0, radius=0.0)
    member x.Angle = angle
    member x.Radius = radius
    member x.Stretch(l) = Point1(angle=x.Angle, radius=x.Radius * l)
    member x.Warp(f) = Point1(angle=f(x.Angle), radius=x.Radius)
    static member Circle(n) =
        [ for i in 1..n -> Point1(angle=2.0*Math.PI/float(n), radius=1.0) ]
```

此類別的推斷 F # 類型如下所示：

```
type Point1 =
    new : unit -> Point1
    new : angle:double * radius:double -> Point1
    static member Circle : n:int -> Point1 list
    member Stretch : l:double -> Point1
    member Warp : f:(double -> double) -> Point1
    member Angle : double
    member Radius : double
```

讓我們來看看如何使用另一個 .NET 語言，讓程式設計人員看到 F # 型別。例如，大約的 c # "signature" 如下所示：

```
// C# signature for the unadjusted Point1 class
public class Point1
{
    public Point1();

    public Point1(double angle, double radius);

    public static Microsoft.FSharp.Collections.List<Point1> Circle(int count);

    public Point1 Stretch(double factor);

    public Point1 Warp(Microsoft.FSharp.Core.FastFunc<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}
```

關於 F # 如何在這裡表示結構，有一些重要的重點需要注意。例如：

- 中繼資料(例如引數名稱)已保留。
- 採用兩個引數的 F # 方法會變成採用兩個引數的 c # 方法。
- 函數和清單會成為 F # 程式庫中對應類型的參考。

下列程式碼示範如何調整此程式碼，以將這些專案納入考慮。

```
namespace SuperDuperFSharpLibrary.Types

type RadialPoint(angle:double, radius:double) =

    /// Return a point at the origin
    new() = RadialPoint(angle=0.0, radius=0.0)

    /// The angle to the point, from the x-axis
    member x.Angle = angle

    /// The distance to the point, from the origin
    member x.Radius = radius

    /// Return a new point, with radius multiplied by the given factor
    member x.Stretch(factor) =
        RadialPoint(angle=angle, radius=radius * factor)

    /// Return a new point, with angle transformed by the function
    member x.Warp(transform:Func<_,_>) =
        RadialPoint(angle=transform.Invoke angle, radius=radius)

    /// Return a sequence of points describing an approximate circle using
    /// the given count of points
    static member Circle(count) =
        seq { for i in 1..count ->
            RadialPoint(angle=2.0*Math.PI/float(count), radius=1.0) }
```

程式碼的推斷 F # 類型如下所示：

```

type RadialPoint =
    new : unit -> RadialPoint
    new : angle:double * radius:double -> RadialPoint
    static member Circle : count:int -> seq<RadialPoint>
    member Stretch : factor:double -> RadialPoint
    member Warp : transform:System.Func<double,double> -> RadialPoint
    member Angle : double
    member Radius : double

```

C # 簽章現在如下所示：

```

public class RadialPoint
{
    public RadialPoint();

    public RadialPoint(double angle, double radius);

    public static System.Collections.Generic.IEnumerable<RadialPoint> Circle(int count);

    public RadialPoint Stretch(double factor);

    public RadialPoint Warp(System.Func<double,double> transform);

    public double Angle { get; }

    public double Radius { get; }
}

```

準備此類型以做為香草 .NET 程式庫一部分的修正，如下所示：

- 已調整數個名稱： `Point1`、`n`、`l` 和會 `f` `RadialPoint` 分別成為、`count` `factor` 和 `transform`。
- 使用的傳回型別， `seq<RadialPoint>` 而不是藉 `RadialPoint list` 由將使用的清單結構變更 `[...]` 為的順序結構 `IEnumerable<RadialPoint>`。
- 使用 .NET 委派型別， `System.Func` 而非 F # 函數類型。

這使得在 c # 程式碼中使用的更好變得更容易。

在 Azure 上使用 F#

2021/3/5 • [Edit Online](#)

F# 是一種優秀的雲端程式設計語言，經常用來撰寫 Web 應用程式、雲端服務、雲端託管微服務，並用於可調整的資料處理。

在下列各節中，您可以找到如何搭配 F# 使用各種 Azure 服務的資源。

NOTE

若本文件集中未包含特定的 Azure 服務，請參閱 Azure Functions 或 .NET 文件以找出該服務。某些 Azure 服務與語言無關，不需要任何特定語言的文件，因此未在此處列出。

使用 Azure 虛擬機器搭配 F##

Azure 支援各種不同的虛擬機器 (VM) 組態，請參閱 [Linux](#) 和 [Azure 虛擬機器](#)。

若要在虛擬機器上安裝 F# 以執行、編譯及/或處理指令碼，請參閱 [Using F# on Linux](#) (在 Linux 上使用 F#) 和 [Using F# on Windows](#) (在 Windows 上使用 F#)。

使用 Azure Functions 搭配 F##

[Azure Functions](#) 是在雲端輕鬆執行一小段程式碼或「函式」的解決方案。您可以只撰寫處理手邊問題所需的程式碼，而不需擔心要執行它的整個應用程式或基礎結構。您的函式會連接到 Azure 儲存體和其他雲端託管資源中的事件。資料則透過函式引數流入您的 F# 函式。您可以使用選擇的開發語言，信任 Azure 以視需要進行調整。

Azure Functions 支援 F# 作為第一級語言，可以在有效率、易於反應且可調整的情況下執行 F# 程式碼。如需如何使用 F# 搭配 Azure Functions 的參考文件，請參閱 [Azure Functions F# 開發人員參考](#)。

使用 Azure Functions 和 F# 的其他資源：

- [Scale Up Azure Functions in F# Using Suave](#) (使用 Suave 擴增 F# Azure Functions)
- [How to create Azure function in F#](#) (如何使用 F# 建立 Azure Function)
- [搭配 Azure Functions 使用 Azure 型別提供者](#)

使用 Azure 儲存體搭配 F##

Azure 儲存體是新式應用程式的儲存體服務基礎層，這些應用程式依賴持久性、可用性和延展性來符合客戶的需求。您可以使用下列文章所述的技術，直接與 Azure 儲存體服務互動 F# 程式。

- [以 F 開始使用 Azure Blob 儲存體#](#)
- [以 F 開始使用 Azure 檔案儲存體#](#)
- [以 F 開始使用 Azure 佇列儲存體#](#)
- [以 F 開始使用 Azure 資料表儲存體#](#)

Azure 儲存體也可以透過宣告式組態 (而不是明確的 API 呼叫) 與 Azure Functions 一起使用。請參閱包含 F# 範例的 [Azure 儲存體的 Azure Functions 觸發程序和繫結](#)。

使用 Azure App Service 搭配 F##

[Azure App Service](#) 是一種雲端平台，用來建置功能強大的 Web 和行動應用程式，這些應用程式可在任何地方

(雲端或內部部署) 連接到資料。

- [F# Azure Web API example](#) (F# Azure Web API 範例)
- [Hosting F# in a web application on Azure](#) (在 Azure 的 Web 應用程式中裝載 F#)

在 Azure HDInsight 或 Azure Databricks 上搭配 F# 使用 Apache Spark

[Apache Spark for Azure HDInsight](#) 是一種開放原始碼處理架構，可執行大規模的資料分析應用程式。[Azure Databricks](#) 是一個針對 Microsoft Azure 雲端服務平台進行最佳化的 Apache Spark 分析平台。Azure 可讓 Apache Spark 部署變得輕鬆又具成本效益。使用 [.net 進行 Apache Spark](#) 的 F# 開發 Spark 應用程式，這是一組適用於 Apache Spark 的 .net 系統。

- 適用於 Apache Spark F# 範例的 .NET
- 在 Azure HDInsight 中安裝 .NET Interactive Jupyter 筆記本
- 將 Apache Spark 作業提交至 Azure HDInsight
- 將 Apache Spark 作業提交至 Azure Databricks

使用 Azure Cosmos DB 搭配 F#

[Azure Cosmos DB](#) 是適用於高可用性、全球散發之應用程式的 NoSQL 服務。

Azure Cosmos DB 可以透過兩種方式與 F# 搭配使用：

1. 透過建立 F# Azure Functions，這會對 Azure Cosmos DB 集合進行回應或變更。請參閱 [Azure Functions 的 Azure Cosmos DB](#) 系統，或
2. 使用 [Azure Cosmos DB .NET SDK FOR SQL API](#)。相關的範例是以 c# 撰寫。

使用 Azure 事件中樞搭配 F##

[Azure 事件中樞](#) 可從網站、應用程式和裝置提供雲端等級的遙測數據。

Azure 事件中樞可以透過下列兩種方式與 F# 搭配使用：

1. 透過建立事件所觸發的 F# Azure Functions。請參閱 [Azure Function 事件中樞的觸發程序](#)，或
2. 透過使用 [.NET SDK for Azure](#)。請注意，這些範例是根據 C#。

使用 Azure 通知中樞搭配 F##

[Azure 通知中樞](#) 是一種多平台、向外延展的推播基礎結構，可讓您將任何後端 (雲端或內部部署) 中的行動推播通知傳送到任何行動平台。

Azure 通知中樞可以透過下列兩種方式與 F# 搭配使用：

1. 透過建立可將結果傳送到通知中樞的 F# Azure Functions。請參閱 [Azure Function 通知中樞的輸出觸發程序](#)，或
2. 透過使用 [.NET SDK for Azure](#)。請注意，這些範例是根據 C#。

使用 F# 在 Azure 上實作 Webhook#

[Webhook](#) 是透過 Web 要求觸發的回呼。GitHub 等網站會使用 Webhook 以訊號通知事件。

您可以使用 F# 實作 Webhook，並透過 [含有 Webhook 繫結的 F# Azure Function](#) 在 Azure 上裝載 Webhook。

使用 Webjobs 搭配 F##

[Webjobs](#) 是您可以使用下列三種方式在 App Service Web 應用程式中執行的程式：依需求、連續或根據排程。

[Example F# Webjob](#) (F# Webjob 範例)

使用 F# 在 Azure 上實作計時器#

計時器觸發程序會根據排程，以單次或週期性方式呼叫函數。

您可以使用 F# 實作計時器，並透過[含有計時器觸發程序的 F# Azure Function](#) 在 Azure 上裝載計時器。

使用 F# 指令碼部署和管理 Azure 資源

Azure VM 可能會使用 Microsoft.Azure.Management 套件和 API，透過 F# 指令碼以程式設計方式部署和管理。

例如，請參閱[開始使用 .NET 的管理程式庫](#)和[使用 Azure 資源管理員](#)。

同樣地，其他 Azure 資源也可以使用相同的元件，透過 F# 指令碼進行部署和管理。例如，您可以建立儲存體帳戶、部署 Azure 雲端服務、建立 Azure Cosmos DB 實例，並從 F# 腳本以程式設計方式管理 Azure 通知中樞。

通常不需要使用 F# 指令碼來部署和管理資源。例如，Azure 資源也可以直接從 JSON 範本描述(可參數化)進行部署。請參閱包含 [Azure 快速入門範本](#)等範例的 [Azure 資源管理員範本](#)。

其他資源

- [所有 Azure 服務的完整文件](#)

以 F 開始使用 Azure Blob 儲存體

2021/3/5 • [Edit Online](#)

Azure Blob 儲存體是將非結構化資料儲存在雲端作為物件/blob 的服務。Blob 儲存體可以儲存任何類型的文字或二進位資料，例如文件、媒體檔案或應用程式安裝程式。Blob 儲存體也稱為物件儲存體。

本文說明如何使用 Blob 儲存體執行一般工作。這些範例是使用適用於 .NET 的 Azure 儲存體用戶端程式庫，以 F # 撰寫。涵蓋的工作包括如何上傳、列出、下載和刪除 blob。

如需 blob 儲存體的概念總覽，請參閱 [適用於 blob 儲存體的 .net 指南](#)。

先決條件

若要使用本指南，您必須先 [建立 Azure 儲存體帳戶](#)。您也需要此帳戶的儲存體存取金鑰。

建立 F # 腳本並開始 F# 互動

本文中的範例可用於 F # 應用程式或 F # 腳本。若要建立 F # 腳本，請 `.fsx` `blobs.fsx` 在您的 f # 開發環境中建立副檔名為的檔案，例如。

接下來，使用 [套件管理員](#)（例如 [Paket](#) 或 [NuGet](#)），`WindowsAzure.Storage`
`Microsoft.WindowsAzure.ConfigurationManager` `WindowsAzure.Storage.dll`
`Microsoft.WindowsAzure.Configuration.dll` 在您的腳本中使用指示詞安裝和封裝和參考 `#r`。

新增命名空間宣告

在 `blobs.fsx` 檔案頂端新增下列 `open` 陳述式：

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Blob // Namespace for Blob storage types
```

取得您的連接字串

在本教學課程中，您需要 Azure 儲存體連接字串。如需連接字串的詳細資訊，請參閱 [設定儲存體連接字串](#)。

在本教學課程中，您會在腳本中輸入連接字串，如下所示：

```
let storageConnString = "..." // fill this in from your storage account
```

不過，這不建議用於實際的專案。儲存體帳戶金鑰很類似儲存體帳戶的根密碼。請務必小心保護您的儲存體帳戶金鑰。請避免轉發給其他使用者、進行硬式編碼，或將它儲存在其他人可以存取的純文字檔案。如果您認為金鑰可能已遭盜用，您可以使用 Azure 入口網站重新產生金鑰。

針對實際的應用程式，維護儲存體連接字串的最佳方式是在設定檔中。若要從設定檔提取連接字串，您可以執行下列動作：

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

是否使用 Azure Configuration Manager 可由您選擇。您也可以使用 API, 例如 .NET Framework 的型別

`ConfigurationManager` 。

解析連接字串

若要剖析連接字串, 請使用:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

這會傳回 `CloudStorageAccount` 。

建立一些本機虛擬資料

開始之前, 請在腳本的目錄中建立一些虛擬本機資料。稍後, 您可以上傳此資料。

```
// Create a dummy file to upload
let localFile = __SOURCE_DIRECTORY__ + "/myfile.txt"
File.WriteAllText(localFile, "some data")
```

建立 Blob 服務用戶端

此 `CloudBlobClient` 類型可讓您取出儲存在 Blob 儲存體中的容器和 blob。以下是建立服務用戶端的其中一種方式:

```
let blobClient = storageAccount.CreateCloudBlobClient()
```

您現在可以開始撰寫程式碼, 以讀取 Blob 儲存體的資料並將資料寫入其中。

建立容器

此範例說明如何建立尚不存在的容器:

```
// Retrieve a reference to a container.
let container = blobClient.GetContainerReference("mydata")

// Create the container if it doesn't already exist.
container.CreateIfNotExists()
```

根據預設, 新容器屬私人性質, 這表示您必須指定儲存體存取金鑰才能從此容器下載 Blob。若要讓所有人都能使用容器中的檔案, 您可以使用下列程式碼將容器設定為公用容器:

```
let permissions = BlobContainerPermissions(PublicAccess=BlobContainerPublicAccessType.Blob)
container.SetPermissions(permissions)
```

網際網路上的任何人都可以看到公用容器中的 Blob, 但要有適當的帳戶存取金鑰或共用存取簽章, 才能修改或刪除這些 Blob。

將 Blob 上傳至容器

Azure Blob 儲存體支援區塊 Blob 和頁面 Blob。在大多數情況下, 區塊 blob 是建議使用的類型。

若要將檔案上傳至區塊 Blob, 請取得容器參照, 並使用該參照來取得區塊 Blob 參照。當您有 blob 參考之後, 就可以藉由呼叫方法, 將任何資料流程上傳至其中 `UploadFromFile`。如果 blob 不存在, 此作業會建立 blob, 如果存在, 則會加以覆寫。

```
// Retrieve reference to a blob named "myblob.txt".
let blockBlob = container.GetBlockBlobReference("myblob.txt")

// Create or overwrite the "myblob.txt" blob with contents from the local file.
do blockBlob.UploadFromFile(localFile)
```

列出容器中的 Blob

若要列出容器中的 Blob，請先取得容器參照。然後，您可以使用容器的 `ListBlobs` 方法來取得 blob 和(或)其中的目錄。若要針對傳回的，存取一組豐富的屬性和方法 `IBlobItem`，您必須將它轉換成 `CloudBlockBlob`、`CloudPageBlob` 或 `CloudBlobDirectory` 物件。如果不清楚類型，可使用類型檢查來決定要將其轉換至何種類型。下列程式碼示範如何擷取和輸出 `mydata` 容器中每個項目的 URI：

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, false) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %0" blob.Properties.Length blob.Uri

    | :? CloudPageBlob as pageBlob ->
        printfn "Page blob of length %d: %0" pageBlob.Properties.Length pageBlob.Uri

    | :? CloudBlobDirectory as directory ->
        printfn "Directory: %0" directory.Uri

    | _ ->
        printfn "Unknown blob type: %0" (item.GetType())
```

您也可以使用名稱的路徑資訊來命名 blob。這會建立虛擬目錄結構，讓您能夠組織及周遊，就像使用傳統檔案系統一樣。目錄結構僅限虛擬，Blob 儲存體中唯一可用的資源是容器和 blob。不過，儲存體用戶端程式庫會提供 `CloudBlobDirectory` 物件來參考虛擬目錄，並簡化使用以這種方式組織之 blob 的處理常式。

例如，假設名為 `photos` 的容器中有下面這一組區塊 blob：

```
photo1.jpg
2015/架構/description.txt
2015/架構/photo3.jpg
2015/架構/photo4.jpg
2016/架構/photo5.jpg
2016/架構/photo6.jpg
2016/架構/description.txt
2016/photo7.jpg\
```

當您在 `ListBlobs` 容器上呼叫 (如上述範例) 所示，會傳回階層式清單。如果它同時包含 `CloudBlobDirectory` 和 `CloudBlockBlob` 物件，分別代表容器中的目錄和 blob，則產生的輸出看起來會像這樣：

```
Directory: https://<accountname>.blob.core.windows.net/photos/2015/
Directory: https://<accountname>.blob.core.windows.net/photos/2016/
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

(選擇性)您可以將 `UseFlatBlobListing` 方法的參數設定 `ListBlobs` 為 `true`。在此情況下，容器中的每個 blob 都會以物件的形式傳回 `CloudBlockBlob`。的呼叫會傳回一般清單，如下所 `ListBlobs` 示：

```
// Loop over items within the container and output the length and URI.
for item in container.ListBlobs(null, true) do
    match item with
    | :? CloudBlockBlob as blob ->
        printfn "Block blob of length %d: %0" blob.Properties.Length blob.Uri

    | _ ->
        printfn "Unexpected blob type: %0" (item.GetType())
```

而且，視容器目前的內容而定，結果看起來像這樣：

```
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2015/architecture/description.txt
Block blob of length 314618: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo3.jpg
Block blob of length 522713: https://<accountname>.blob.core.windows.net/photos/2015/architecture/photo4.jpg
Block blob of length 4: https://<accountname>.blob.core.windows.net/photos/2016/architecture/description.txt
Block blob of length 419048: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo5.jpg
Block blob of length 506388: https://<accountname>.blob.core.windows.net/photos/2016/architecture/photo6.jpg
Block blob of length 399751: https://<accountname>.blob.core.windows.net/photos/2016/photo7.jpg
Block blob of length 505623: https://<accountname>.blob.core.windows.net/photos/photo1.jpg
```

下載 Blob

若要下載 blob，請先取得 blob 參考，然後呼叫 `DownloadToStream` 方法。下列範例 `DownloadToStream` 會使用方法，將 blob 內容傳送給資料流程物件，您接著可以將這些內容保存到本機檔案。

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDownload = container.GetBlockBlobReference("myblob.txt")

// Save blob contents to a file.
do
    use fileStream = File.OpenWrite(__SOURCE_DIRECTORY__ + "/path/download.txt")
    blobToDownload.DownloadToStream(fileStream)
```

您也可以使用 `DownloadToStream` 方法，將 blob 的內容下載為文字字串。

```
let text =
    use memoryStream = new MemoryStream()
    blobToDownload.DownloadToStream(memoryStream)
    Text.Encoding.UTF8.GetString(memoryStream.ToArray())
```

刪除 Blob

若要刪除 blob，請先取得 blob 參考，然後 `Delete` 在其上呼叫方法。

```
// Retrieve reference to a blob named "myblob.txt".
let blobToDelete = container.GetBlockBlobReference("myblob.txt")

// Delete the blob.
blobToDelete.Delete()
```

以非同步方式分頁列出 Blob

如果您要列出大量的 Blob，或是要控制在單一系列出作業中傳回的結果數，您可以在結果頁面中列出 Blob。此範例說明如何以非同步方式分頁傳回結果，使執行不會因為等待大型結果集傳回而中斷。

此範例顯示一般 blob 清單，但您也可以將方法的參數設定為，以執行階層式清單 `useFlatBlobListing`

`ListBlobsSegmentedAsync` `false` 。

此範例會使用區塊來定義非同步方法 `async` 。

`let!` 關鍵字會暫停執行範例方法，直到清單工作完成為止。

```
let ListBlobsSegmentedInFlatListing(container:CloudBlobContainer) =
    async {

        // List blobs to the console window, with paging.
        printfn "List blobs in pages:"

        // Call ListBlobsSegmentedAsync and enumerate the result segment
        // returned, while the continuation token is non-null.
        // When the continuation token is null, the last page has been
        // returned and execution can exit the loop.

        let rec loop continuationToken (i:int) =
            async {
                let! ct = Async.CancellationToken
                // This overload allows control of the page size. You can return
                // all remaining results by passing null for the maxResults
                // parameter, or by calling a different overload.
                let! resultSegment =
                    container.ListBlobsSegmentedAsync(
                        "", true, BlobListingDetails.All, Nullable 10,
                        continuationToken, null, null, ct)
                |> Async.AwaitTask

                if (resultSegment.Results |> Seq.length > 0) then
                    printfn "Page %d:" i

                    for blobItem in resultSegment.Results do
                        printfn "\t%O" blobItem.StorageUri.PrimaryUri

                    printfn ""

                    // Get the continuation token.
                    let continuationToken = resultSegment.ContinuationToken
                    if (continuationToken <> null) then
                        do! loop continuationToken (i+1)
            }

        do! loop null 1
    }
```

我們現在可以使用這個非同步常式，如下所示。首先，使用本教學課程稍早所建立的本機檔案，上傳一些虛擬資料 ()。

```
// Create some dummy data by uploading the same file over and over again
for i in 1 .. 100 do
    let blob = container.GetBlockBlobReference("myblob" + string i + ".txt")
    use fileStream = System.IO.File.OpenRead(localFile)
    blob.UploadFromFile(localFile)
```

現在，呼叫常式。您可以使用 `Async.RunSynchronously` 來強制執行非同步作業。

```
ListBlobsSegmentedInFlatListing container |> Async.RunSynchronously
```

寫入附加 Blob

附加 Blob 已針對附加作業 (例如紀錄) 最佳化。如同區塊 blob，附加 blob 是由區塊所組成，但當您將新區塊新

增至附加 blob 時，它一律會附加至 blob 的結尾。您無法更新或刪除附加 Blob 中的現有區塊。附加 Blob 的區塊識別碼不會公開顯示，因為該識別碼適用於區塊 Blob。

附加 Blob 中的每個區塊大小都不同，最大為 4 MB，而附加 Blob 可包含高達 50,000 個區塊。因此，附加 Blob 的大小上限稍高於 195 GB (4 MB X 50,000 個區塊)。

下列範例會建立新的附加 blob，並在其中附加一些資料，以模擬簡單的記錄作業。

```
// Get a reference to a container.
let appendContainer = blobClient.GetContainerReference("my-append-blobs")

// Create the container if it does not already exist.
appendContainer.CreateIfNotExists() |> ignore

// Get a reference to an append blob.
let appendBlob = appendContainer.GetAppendBlobReference("append-blob.log")

// Create the append blob. Note that if the blob already exists, the
// CreateOrReplace() method will overwrite it. You can check whether the
// blob exists to avoid overwriting it by using CloudAppendBlob.Exists().
appendBlob.CreateOrReplace()

let numBlocks = 10

// Generate an array of random bytes.
let rnd = new Random()
let bytes = Array.zeroCreate<byte>(numBlocks)
rnd.NextBytes(bytes)

// Simulate a logging operation by writing text data and byte data to the
// end of the append blob.
for i in 0 .. numBlocks - 1 do
    let msg = sprintf "Timestamp: %u \tLog Entry: %d\n" DateTime.UtcNow bytes.[i]
    appendBlob.AppendText(msg)

// Read the append blob to the console window.
let downloadedText = appendBlob.DownloadText()
printfn "%s" downloadedText
```

如需了解有關 Blob 的三種類型間差異的資訊，請參閱 [了解區塊 Blob、分頁 Blob 和附加 Blob](#)。

並行存取

若要支援從多個用戶端或多個程序執行個體並行存取 Blob，您可以使用 ETags 或「租用」。

- **Etag** - 提供方法來偵測 Blob 或容器已被另一個程序修改過
- **租用** - 提供一段時間來取得對 blob 的獨佔、可再生、寫入或刪除存取權的方法。

如需詳細資訊，請參閱 [管理 Microsoft Azure 儲存體中的並行存取](#)。

命名容器

儲存體 Blob 中的每個 Blob 必須位於一個容器中。此容器會組成 blob 名稱的一部分。例如，`mydata` 是這些範例 blob URI 中容器的名稱：

- `https://storagesample.blob.core.windows.net/mydata/blob1.txt`
- `https://storagesample.blob.core.windows.net/mydata/photos/myphoto.jpg`

容器名稱必須是有效的 DNS 名稱，且符合下列命名規則：

1. 容器名稱必須以字母或數字開頭，並且只能包含字母、數字和虛線 (-) 字元。

2. 每個虛線 (-) 字元前後必須立即接著字母或數字；容器名稱中不允許連續虛線。
3. 容器名稱的所有字母必須都是小寫。
4. 容器名稱必須介於 3 至 63 個字元長。

容器的名稱必須一律為小寫。如果在容器名稱中含有大寫字母，或其他違反容器命名規則，您可能會收到「400 錯誤 (不正確的要求)」錯誤訊息。

管理 Blob 安全性

根據預設，Azure 儲存體會限制擁有帳戶存取金鑰的帳戶擁有者的存取權來保持資料安全。當您需要共用儲存體帳戶中的 Blob 資料時，請注意不可危及您帳戶存取金鑰的安全性。此外，您可以加密 blob 資料以確保透過網路與 Azure 儲存體中的安全。

控制對 blob 資料的存取

根據預設，您儲存體帳戶中的 blob 資料僅供儲存體帳戶擁有者使用。依預設，驗證對 Blob 儲存體的要求需要帳戶存取金鑰。不過，您可能會想要將某些 blob 資料提供給其他使用者。

加密 blob 資料

Azure 儲存體支援在用戶端和伺服器上加密 blob 資料。

後續步驟

了解 Blob 儲存體的基礎概念之後，請使用下列連結深入了解。

工具

- [F # AzureStorageTypeProvider](#)
F # 型別提供者，可用來流覽 Blob、資料表和佇列 Azure 儲存體資產，並輕鬆地對其套用 CRUD 作業。
- [Fsharp.core](#)
使用 Microsoft Azure 資料表儲存體服務的 F # API
- [Microsoft Azure 儲存體總管 \(MASE\)](#)
Microsoft 提供的免費獨立應用程式，可讓您在 Windows、OS X 和 Linux 上以視覺化方式處理 Azure 儲存體資料。

Blob 儲存體參考

- [適用於 .NET 的 Azure 儲存體 API](#)
- [Azure 儲存體服務 REST API 參考](#)

相關指南

- [適用於 .NET 的 Azure Blob 儲存體範例](#)
- [開始使用 AzCopy](#)
- [設定 Azure 儲存體連接字串](#)
- [Azure 儲存體團隊部落格](#)
- [快速入門:使用 .NET 在物件儲存體中建立 Blob](#)

以 F# 開始使用 Azure 檔案儲存體

2021/3/5 • [Edit Online](#)

Azure 檔案儲存體是一項服務，可使用標準 [伺服器訊息區 \(SMB\) 通訊協定](#)，在雲端中提供檔案共用。SMB 2.1 和 SMB 3.0 皆受到支援。透過 Azure 檔案儲存體，您可以快速地將依賴檔案共用的繼承應用程式遷移到 Azure，而不需要重寫成本。在 Azure 虛擬機器、雲端服務或內部部署中執行的應用程式，可掛接雲端中的檔案共用，就像桌面應用程式掛接一般 SMB 共用一樣。可同時掛接和存取檔案儲存體共用的應用程式元件數量沒有限制。

如需檔案儲存體的概念總覽，請參閱檔案 [儲存體的 .net 指南](#)。

先決條件

若要使用本指南，您必須先 [建立 Azure 儲存體帳戶](#)。您也需要此帳戶的儲存體存取金鑰。

建立 F# 腳本並開始 F# 互動

本文中的範例可用於 F# 應用程式或 F# 腳本。若要建立 F# 腳本，請 `.fsx` `files.fsx` 在您的 f# 開發環境中建立副檔名為的檔案，例如。

接下來，使用 [套件管理員](#)（例如 [Paket](#) 或 [NuGet](#)），`WindowsAzure.Storage` `WindowsAzure.Storage.dll` 在您的腳本中使用指示詞安裝封裝和參考 `#r`。

新增命名空間宣告

在 `files.fsx` 檔案頂端新增下列 `open` 陳述式：

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.File // Namespace for File storage types
```

取得您的連接字串

在本教學課程中，您將需要 Azure 儲存體連接字串。如需連接字串的詳細資訊，請參閱 [設定儲存體連接字串](#)。

在本教學課程中，您將在腳本中輸入連接字串，如下所示：

```
let storageConnString = "..." // fill this in from your storage account
```

不過，這不建議用於實際的專案。儲存體帳戶金鑰很類似儲存體帳戶的根密碼。請務必小心保護您的儲存體帳戶金鑰。請避免轉發給其他使用者、進行硬式編碼，或將它儲存在其他人可以存取的純文字檔案。如果您認為金鑰可能已遭盜用，您可以使用 Azure 入口網站重新產生金鑰。

針對實際的應用程式，維護儲存體連接字串的最佳方式是在設定檔中。若要從設定檔提取連接字串，您可以執行下列動作：

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

是否使用 Azure Configuration Manager 可由您選擇。您也可以使用 API，例如 .NET Framework 的型別

ConfigurationManager 。

解析連接字串

若要剖析連接字串，請使用：

```
// Parse the connection string and return a reference to the storage account.  
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

這會傳回 CloudStorageAccount 。

建立檔案服務用戶端

此 CloudFileClient 類型可讓您以程式設計方式使用儲存在檔案儲存體中的檔案。以下是建立服務用戶端的其中一種方式：

```
let fileClient = storageAccount.CreateCloudFileClient()
```

現在您已準備好撰寫程式碼，以讀取資料，並將資料寫入檔案儲存體。

建立檔案共用

此範例會示範如何建立檔案共用(如果尚未存在)：

```
let share = fileClient.GetShareReference("myfiles")  
share.CreateIfNotExists()
```

建立根目錄和子目錄

在這裡，您會取得根目錄並取得根目錄的子目錄。如果兩者都不存在，您可以建立兩者。

```
let rootDir = share.GetRootDirectoryReference()  
let subDir = rootDir.GetDirectoryReference("myLogs")  
subDir.CreateIfNotExists()
```

將文字上傳為檔案

此範例說明如何將文字上傳為檔案。

```
let file = subDir.GetFileReference("log.txt")  
file.UploadText("This is the content of the log file")
```

將檔案下載到檔案的本機複本

您可以在這裡下載剛才建立的檔案，並將內容附加至本機檔案。

```
file.DownloadToFile("log.txt", FileMode.Append)
```

設定檔案共用的大小上限

下列範例示範如何檢查共用的目前使用狀況，以及如何設定共用的配額。FetchAttributes 必須呼叫以填入共用的 Properties，並 SetProperties 將本機變更傳播至 Azure 檔案儲存體。

```
// stats.Usage is current usage in GB
let stats = share.GetStats()
share.FetchAttributes()

// Set the quota to 10 GB plus current usage
share.Properties.Quota <- stats.Usage + 10 |> Nullable
share.SetProperties()

// Remove the quota
share.Properties.Quota <- Nullable()
share.SetProperties()
```

產生檔案或檔案共用的共用存取簽章

您可以為檔案共用或個別檔案產生共用存取簽章 (SAS)。您也可以檔案共用上建立共用存取原則，以管理共用存取簽章。建議您建立共用存取原則，因為如果必須洩漏 SAS，它提供了一種撤銷 SAS 的方式。

在這裡，您會在共用上建立共用存取原則，然後使用該原則在共用中的檔案上提供 SAS 的限制。

```
// Create a 24-hour read/write policy.
let policy =
    SharedAccessFilePolicy
        (SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable),
         Permissions = (SharedAccessFilePermissions.Read ||| SharedAccessFilePermissions.Write))

// Set the policy on the share.
let permissions = share.GetPermissions()
permissions.SharedAccessPolicies.Add("policyName", policy)
share.SetPermissions(permissions)

let sasToken = file.GetSharedAccessSignature(policy)
let sasUri = Uri(file.StorageUri.PrimaryUri.ToString() + sasToken)

let fileSas = CloudFile(sasUri)
fileSas.UploadText("This write operation is authenticated via SAS")
```

如需建立與使用共用存取簽章的詳細資訊，請參閱[共用存取簽章 \(SAS\)](#)和[透過 Blob 儲存體建立與使用 SAS](#)。

複製檔案

您可以將檔案複製到另一個檔案，或複製到 blob，或是將 blob 複製到檔案。如果您要將 blob 複製到檔案，或將檔案複製到 blob，您 **必須** 使用共用存取簽章 (SAS) 來驗證來源物件，即使您是在相同的儲存體帳戶內進行複製也一樣。

將檔案複製到另一個檔案

在這裡，您會將檔案複製到相同共用中的另一個檔案。由於此複製作業是在相同儲存體帳戶中的檔案間進行複製，所以您可以使用共用金鑰驗證執行複製。

```
let destFile = subDir.GetFileReference("log_copy.txt")
destFile.StartCopy(file)
```

將檔案複製到 Blob

在這裡，您會建立檔案，並將它複製到相同儲存體帳戶內的 blob。您會建立來源檔案的 SAS，供服務用來在複製作業期間驗證來源檔案的存取權。

```
// Get a reference to the blob to which the file will be copied.
let blobClient = storageAccount.CreateCloudBlobClient()
let container = blobClient.GetContainerReference("myContainer")
container.CreateIfNotExists()
let destBlob = container.GetBlockBlobReference("log_blob.txt")

let filePolicy =
    SharedAccessFilePolicy
        (Permissions = SharedAccessFilePermissions.Read,
         SharedAccessExpiryTime = (DateTimeOffset.UtcNow.AddHours(24.) |> Nullable))

let fileSas2 = file.GetSharedAccessSignature(filePolicy)
let sasUri2 = Uri(file.StorageUri.PrimaryUri.ToString() + fileSas2)
destBlob.StartCopy(sasUri2)
```

您可以用相同方式將 Blob 複製到檔案。如果來源物件為 Blob，則請建立 SAS，以便在複製作業期間驗證該 Blob 存取權。

使用度量疑難排解檔案儲存體

Azure 儲存體分析支援檔儲存體的計量。利用度量資料，您可以追蹤要求及診斷問題。

您可以從 [Azure 入口網站](#) 啟用檔案儲存體的計量，也可以從 F # 執行，如下所示：

```
open Microsoft.Azure.Storage.File.Protocol
open Microsoft.Azure.Storage.Shared.Protocol

let props =
    FileServiceProperties(
        (HourMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (14 |> Nullable),
            Version = "1.0"),
         MinuteMetrics = MetricsProperties(
            MetricsLevel = MetricsLevel.ServiceAndApi,
            RetentionDays = (7 |> Nullable),
            Version = "1.0"))

fileClient.SetServiceProperties(props)
```

後續步驟

如需 Azure 檔案儲存體的詳細資訊，請參閱這些連結。

概念性文章和影片

- [Azure 檔案儲存體:適用於 Windows 和 Linux 的無摩擦雲端 SMB 檔案系統](#)
- [如何搭配 Linux 使用 Azure 檔案儲存體](#)

檔案儲存體的工具支援

- [搭配使用 Azure PowerShell 與 Azure 儲存體](#)
- [如何搭配使用 AzCopy 與 Microsoft Azure 儲存體](#)
- [使用 Azure CLI 上傳、下載及列出 Blob](#)

參考

- [Storage Client Library for .NET 參考資料](#)
- [檔案服務 REST API 參考](#)

部落格文章

- [Azure 檔案儲存體現已正式推出](#)
- [Azure 檔案儲存體內部](#)
- [Microsoft Azure 檔案服務簡介](#)
- [保留與 Microsoft Azure 檔案的連線](#)

以 F 開始使用 Azure 佇列儲存體

2021/3/5 • [Edit Online](#)

Azure 佇列儲存體可提供應用程式元件之間的雲端通訊。設計擴充性的應用程式時，會經常分離應用程式元件，以便進行個別擴充。佇列儲存體可針對應用程式元件間的通訊，提供非同步傳訊，無論應用程式元件是在雲端、桌面、內部部署伺服器或行動裝置上執行。佇列儲存體也支援管理非同步工作並建置處理工作流程。

關於本教學課程

本教學課程說明如何使用 Azure 佇列儲存體來撰寫一些常見工作的 F # 程式碼。涵蓋的工作包括建立和刪除佇列，以及新增、讀取和刪除佇列訊息。

如需佇列儲存體的概念總覽，請參閱 [佇列儲存體的 .net 指南](#)。

先決條件

若要使用本指南，您必須先 [建立 Azure 儲存體帳戶](#)。您也需要此帳戶的儲存體存取金鑰。

建立 F # 腳本並開始 F# 互動

本文中的範例可用於 F # 應用程式或 F # 腳本。若要建立 F # 腳本，請 `.fsx` `queues.fsx` 在您的 f # 開發環境中建立副檔名為的檔案，例如。

接下來，使用 [套件管理員](#) (例如 [Paket](#) 或 [NuGet](#))，`WindowsAzure.Storage` `WindowsAzure.Storage.dll` 在您的腳本中使用指示詞安裝封裝和參考 `#r`。

新增命名空間宣告

在 `queues.fsx` 檔案頂端新增下列 `open` 陳述式：

```
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Queue // Namespace for Queue storage types
```

取得您的連接字串

在本教學課程中，您將需要 Azure 儲存體連接字串。如需連接字串的詳細資訊，請參閱 [設定儲存體連接字串](#)。

在本教學課程中，您將在腳本中輸入連接字串，如下所示：

```
let storageConnString = "..." // fill this in from your storage account
```

不過，這不建議用於實際的專案。儲存體帳戶金鑰很類似儲存體帳戶的根密碼。請務必小心保護您的儲存體帳戶金鑰。請避免轉發給其他使用者、進行硬式編碼，或將它儲存在其他人可以存取的純文字檔案。如果您認為金鑰可能已遭盜用，您可以使用 Azure 入口網站重新產生金鑰。

針對實際的應用程式，維護儲存體連接字串的最佳方式是在設定檔中。若要從設定檔提取連接字串，您可以執行下列動作：

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

是否使用 Azure Configuration Manager 可由您選擇。您也可以使用 API, 例如 .NET Framework 的型別 `ConfigurationManager` 。

解析連接字串

若要剖析連接字串, 請使用:

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

這會傳回 `CloudStorageAccount` 。

建立佇列服務用戶端

`CloudQueueClient` 類別可讓您取出儲存在佇列儲存體中的佇列。以下是建立服務用戶端的其中一種方式:

```
let queueClient = storageAccount.CreateCloudQueueClient()
```

您現在可以開始撰寫程式碼, 以讀取佇列儲存體的資料並將資料寫入其中。

建立佇列

這個範例會示範如何建立佇列(如果尚未存在):

```
// Retrieve a reference to a container.
let queue = queueClient.GetQueueReference("myqueue")

// Create the queue if it doesn't already exist
queue.CreateIfNotExists()
```

將訊息插入佇列

若要將訊息插入現有佇列, 請先建立新的 `CloudQueueMessage` 。接下來, 呼叫 `AddMessage` 方法。您 `CloudQueueMessage` 可以從字串 (採用 utf-8 格式) 或陣列來建立 `byte` , 如下所示:

```
// Create a message and add it to the queue.
let message = new CloudQueueMessage("Hello, World")
queue.AddMessage(message)
```

查看下一個訊息

您可以藉由呼叫方法, 在佇列前面查看訊息, 而不需要將它從佇列中移除 `PeekMessage` 。

```
// Peek at the next message.
let peekedMessage = queue.PeekMessage()
let msgAsString = peekedMessage.AsString
```

取得下一個要處理的訊息

您可以藉由呼叫方法, 在佇列前端取得訊息以進行處理 `GetMessage` 。

```
// Get the next message. Successful processing must be indicated via DeleteMessage later.
let retrieved = queue.GetMessage()
```


您稍後會使用來指示訊息的成功處理 `DeleteMessage` 。

變更佇列訊息的內容

您可以在佇列中就地變更已抓取訊息的內容。如果訊息代表工作作業，則您可以使用此功能來更新工作作業的狀態。下列程式碼將使用新的內容更新佇列訊息，並將可見度逾時設定延長 60 秒。這可儲存與訊息相關的工作狀態，並提供用戶端多一分鐘的時間繼續處理訊息。您可以使用此技巧來追蹤佇列訊息上的多步驟工作流程，如果因為硬體或軟體故障而導致某個處理步驟失敗，將無需從頭開始。通常，您也會保留重試計數，如果訊息重試次數超過數次，您就會刪除它。這麼做可防止每次處理時便觸發應用程式錯誤的訊息。

```
// Update the message contents and set a new timeout.
retrieved.SetMessageContent("Updated contents.")
queue.UpdateMessage(retrieved,
    TimeSpan.FromSeconds(60.0),
    MessageUpdateFields.Content ||| MessageUpdateFields.Visibility)
```

將下一個訊息清除佇列

您的程式碼可以使用兩個步驟將訊息自佇列中清除佇列。當您呼叫時 `GetMessage`，會取得佇列中的下一則訊息。從 `GetMessage` 傳回的訊息，對於從此佇列讀取訊息的任何其他程式碼而言，將會是不可見的。依預設，此訊息會維持 30 秒的不可見狀態。若要完成從佇列中移除訊息，您還必須呼叫 `DeleteMessage`。這個移除訊息的兩步驟程序可確保您的程式碼因為硬體或軟體故障而無法處理訊息時，另一個程式碼的執行個體可以取得相同訊息並再試一次。您的程式碼會在 `DeleteMessage` 處理完訊息之後立即呼叫。

```
// Process the message in less than 30 seconds, and then delete the message.
queue.DeleteMessage(retrieved)
```

使用非同步工作流程搭配一般佇列儲存體 Api

此範例示範如何使用非同步工作流程搭配一般佇列儲存體 Api。

```
async {
    let! exists = queue.CreateIfNotExistsAsync() |> Async.AwaitTask

    let! retrieved = queue.GetMessageAsync() |> Async.AwaitTask

    // ... process the message here ...

    // Now indicate successful processing:
    do! queue.DeleteMessageAsync(retrieved) |> Async.AwaitTask
}
```

其他將訊息移出佇列的選項

自訂從佇列中擷取訊息的方法有兩種。首先，您可以取得一批訊息 (最多 32 個)。其次，您可以設定較長或較短的可見度逾時，讓您的程式碼有較長或較短的時間可以完全處理每個訊息。下列程式碼範例會使用 `GetMessages` 在一個呼叫中取得 20 個訊息，然後處理每個訊息。它也會將可見度逾時設定為每個訊息五分鐘。系統會為所有訊息同時開始 5 分鐘，因此在呼叫後經過 5 分鐘後 `GetMessages`，任何尚未刪除的訊息都會重新顯示。

```
for msg in queue.GetMessages(20, Nullable(TimeSpan.FromMinutes(5))) do
    // Process the message here.
    queue.DeleteMessage(msg)
```

取得佇列長度

您可以取得佇列中的估計訊息數目。`FetchAttributes` 方法會要求佇列服務取出佇列屬性，包括訊息計數。`ApproximateMessageCount` 屬性會傳回方法所取出的最後一個值 `FetchAttributes`，而不會呼叫佇列服務。

```
queue.FetchAttributes()  
let count = queue.ApproximateMessageCount.GetValueOrDefault()
```

刪除佇列

若要刪除佇列及其內含的所有訊息，請 `Delete` 在佇列物件上呼叫方法。

```
// Delete the queue.  
queue.Delete()
```

後續步驟

了解佇列儲存體的基礎概念之後，請參考下列連結以了解有關更複雜的儲存工作。

- [適用於 .NET 的 Azure 儲存體 API](#)
- [Azure 儲存體型別提供者](#)
- [Azure 儲存體團隊部落格](#)
- [設定 Azure 儲存體連接字串](#)
- [Azure 儲存體服務 REST API 參考](#)

開始使用 Azure 資料表儲存體和使用 F 的 Azure Cosmos DB 資料表 API#

2021/3/5 • [Edit Online](#)

Azure 資料表儲存體是一項服務，可將結構化的 NoSQL 資料儲存在雲端中。表格儲存體是具有無結構描述設計的索引鍵/屬性存放區。由於表格儲存體並無結構描述，因此可輕易隨著應用程式發展需求改寫資料。所有類型的應用程式都可以用快速且具成本效益的方式存取資料。相較於類似資料量的傳統 SQL，資料表儲存體通常可大幅降低成本。

您可以使用表格儲存體來儲存具彈性的資料集，例如 Web 應用程式的使用者資料、通訊錄、裝置資訊，以及服務所需的任何其他中繼資料類型。您可以在資料表中儲存任意數目的實體，且儲存體帳戶可包含任意數目的資料表，最高可達儲存體帳戶的容量限制。

Azure Cosmos DB 提供針對 Azure 資料表儲存體所撰寫，且需要高階功能的應用程式資料表 API，例如：

- 周全的全域發佈。
- 全球專用的輸送量。
- 99 百分位數的單一數字毫秒延遲。
- 保證高可用性。
- 自動次要索引。

針對 Azure 資料表儲存體所撰寫的應用程式可以使用資料表 API 來遷移至 Azure Cosmos DB，而不需要變更程式碼並利用 premium 功能。資料表 API 具有適用於 .NET、Java、Python 和 Node.js 的用戶端 SDK。

如需詳細資訊，請參閱 [Azure Cosmos DB 資料表 API 簡介](#)。

關於本教學課程

本教學課程說明如何使用 Azure 資料表儲存體或 Azure Cosmos DB 資料表 API 撰寫 F# 程式碼來進行一些常見工作，包括建立和刪除資料表，以及插入、更新、刪除和查詢資料表資料。

先決條件

若要使用本指南，您必須先 [建立 Azure 儲存體帳戶](#) 或 [Azure Cosmos DB 帳戶](#)。

建立 F# 腳本並開始 F# 互動

本文中的範例可用於 F# 應用程式或 F# 腳本。若要建立 F# 腳本，請 `.fsx` `tables.fsx` 在您的 f# 開發環境中建立副檔名為的檔案，例如。

接下來，使用 [套件管理員](#)（例如 [Paket](#) 或 [NuGet](#)），`WindowsAzure.Storage` `WindowsAzure.Storage.dll` 在您的腳本中使用指示詞安裝封裝和參考 `#r`。請再次進行，以 `Microsoft.WindowsAzure.ConfigurationManager` 取得 Microsoft Azure 命名空間。

新增命名空間宣告

在 `tables.fsx` 檔案頂端新增下列 `open` 陳述式：

```
open System
open System.IO
open Microsoft.Azure // Namespace for CloudConfigurationManager
open Microsoft.Azure.Storage // Namespace for CloudStorageAccount
open Microsoft.Azure.Storage.Table // Namespace for Table storage types
```

取得 Azure 儲存體連接字串

如果您要連接到 Azure 儲存體表格服務，您將需要此教學課程的連接字串。您可以從 Azure 入口網站複製您的連接字串。如需連接字串的詳細資訊，請參閱 [設定儲存體連接字串](#)。

取得 Azure Cosmos DB 連接字串

如果您要連接到 Azure Cosmos DB，您將需要此教學課程的連接字串。您可以從 Azure 入口網站複製您的連接字串。在 Azure 入口網站的 Cosmos DB 帳戶中，移至 [設定 > 連接字串]，然後選取 [複製] 按鈕以複製您的主要連接字串。

針對本教學課程，請在腳本中輸入您的連接字串，如下列範例所示：

```
let storageConnString = "..." // fill this in from your storage account
```

不過，這不建議用於實際的專案。儲存體帳戶金鑰很類似儲存體帳戶的根密碼。請務必小心保護您的儲存體帳戶金鑰。請避免轉發給其他使用者、進行硬式編碼，或將它儲存在其他人可以存取的純文字檔案。如果您認為金鑰可能已遭盜用，您可以使用 Azure 入口網站重新產生金鑰。

針對實際的應用程式，維護儲存體連接字串的最佳方式是在設定檔中。若要從設定檔提取連接字串，您可以執行下列動作：

```
// Parse the connection string and return a reference to the storage account.
let storageConnString =
    CloudConfigurationManager.GetSetting("StorageConnectionString")
```

是否使用 Azure Configuration Manager 可由您選擇。您也可以使用 API，例如 .NET Framework 的型別

`ConfigurationManager`。

解析連接字串

若要剖析連接字串，請使用：

```
// Parse the connection string and return a reference to the storage account.
let storageAccount = CloudStorageAccount.Parse(storageConnString)
```

這會傳回 `CloudStorageAccount`。

建立表格服務用戶端

`CloudTableClient` 類別可讓您在資料表儲存體中取出資料表和實體。以下是建立服務用戶端的其中一種方式：

```
// Create the table client.
let tableClient = storageAccount.CreateCloudTableClient()
```

您現在可以開始撰寫程式碼，以讀取表格儲存體的資料並將資料寫入其中。

建立資料表

此範例說明如何建立尚不存在的資料表：

```
// Retrieve a reference to the table.
let table = tableClient.GetTableReference("people")

// Create the table if it doesn't exist.
table.CreateIfNotExists()
```

將實體新增至資料表

實體必須具有繼承自的類型 `TableEntity`。您可以使用任何想要的方式進行擴充 `TableEntity`，但您的類型 必須具有無參數的函式。只有具有 `get` `set` 的屬性 會儲存在您的 Azure 資料表中。

實體的資料分割和資料列索引鍵可唯一識別資料表中的實體。查詢具有相同分割區索引鍵的實體，其速度快於查詢具有不同分割區索引鍵的實體，但使用不同的資料分割索引鍵可提供更佳的延展性或平行作業。

以下範例 `Customer` 會使用 `lastName` 做為資料分割索引鍵，並使用 `firstName` 做為資料列索引鍵。

```
type Customer(firstName, lastName, email: string, phone: string) =
    inherit TableEntity(partitionKey=lastName, rowKey=firstName)
    new() = Customer(null, null, null, null)
    member val Email = email with get, set
    member val PhoneNumber = phone with get, set

let customer =
    Customer("Walter", "Harp", "Walter@contoso.com", "425-555-0101")
```

現在新增 `Customer` 至資料表。若要這樣做，請建立在 `TableOperation` 資料表上執行的。在此情況下，您會建立作業 `Insert`。

```
let insertOp = TableOperation.Insert(customer)
table.Execute(insertOp)
```

插入一批實體

您可以使用單一寫入作業，將一批實體插入資料表中。批次作業可讓您將作業合併成單一執行，但是有一些限制：

- 您可以在相同的批次作業中執行更新、刪除和插入。
- 批次作業最多可包含100個實體。
- 批次作業中的所有實體都必須有相同的分割區索引鍵。
- 雖然您可以在批次作業中執行查詢，但它必須是批次中唯一的作業。

以下是將兩個插入結合到批次作業的程式碼：

```
let customer1 =
    Customer("Jeff", "Smith", "Jeff@contoso.com", "425-555-0102")

let customer2 =
    Customer("Ben", "Smith", "Ben@contoso.com", "425-555-0103")

let batchOp = TableBatchOperation()
batchOp.Insert(customer1)
batchOp.Insert(customer2)
table.ExecuteBatch(batchOp)
```

擷取資料分割中的所有實體

若要向資料表查詢資料分割中的所有實體，請使用 `TableQuery` 物件。在這裡，您會篩選出 "Smith" 為分割區索引鍵的實體。

```
let query =
    TableQuery<Customer>().Where(
        TableQuery.GenerateFilterCondition(
            "PartitionKey", QueryComparisons.Equal, "Smith"))

let result = table.ExecuteQuery(query)
```

您現在會列印結果：

```
for customer in result do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

擷取資料分割中某個範圍的實體

如果您不想要查詢資料分割中的所有實體，可結合資料分割索引鍵篩選器與資料列索引鍵篩選器來指定範圍。在這裡，您會使用兩個篩選器來取得 "Smith" 分割區中的所有實體，其中資料列索引鍵 (名字) 開頭為字母中 "M" 之前的字母。

```
let range =
    TableQuery<Customer>().Where(
        TableQuery.CombineFilters(
            TableQuery.GenerateFilterCondition(
                "PartitionKey", QueryComparisons.Equal, "Smith"),
            TableOperators.And,
            TableQuery.GenerateFilterCondition(
                "RowKey", QueryComparisons.LessThan, "M")))

let rangeResult = table.ExecuteQuery(range)
```

您現在會列印結果：

```
for customer in rangeResult do
    printfn "customer: %A %A" customer.RowKey customer.PartitionKey
```

擷取單一實體

您可以撰寫查詢來擷取單一特定實體。在這裡，您會使用 `TableOperation` 來指定客戶 "Ben Smith"。您可以取回，而不是集合 `Customer`。在查詢中同時指定資料分割索引鍵和資料列索引鍵，是從表格服務中取得單一實體最快的方式。

```
let retrieveOp = TableOperation.Retrieve<Customer>("Smith", "Ben")

let retrieveResult = table.Execute(retrieveOp)
```

您現在會列印結果：

```
// Show the result
let retrieveCustomer = retrieveResult.Result :?> Customer
printfn "customer: %A %A" retrieveCustomer.RowKey retrieveCustomer.PartitionKey
```

取代實體

若要更新實體，請從表格服務中取出它、修改實體物件，然後使用作業將變更儲存回資料表服務 `Replace`。這會導致在伺服器上完全取代實體，除非伺服器上的實體在抓取之後已經變更，在這種情況下，作業會失敗。這項失敗是為了防止您的應用程式不慎覆寫其他來源的變更。

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0103"
    let replaceOp = TableOperation.Replace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeded")
with e ->
    Console.WriteLine("Update failed")
```

插入或取代實體

有時候，您不知道實體是否存在於資料表中。如果有的話，就不再需要儲存在其中的目前值。您可以使用 `InsertOrReplace` 來建立實體，或將它取代(如果存在的話)，而不論其狀態為何。

```
try
    let customer = retrieveResult.Result :?> Customer
    customer.PhoneNumber <- "425-555-0104"
    let replaceOp = TableOperation.InsertOrReplace(customer)
    table.Execute(replaceOp) |> ignore
    Console.WriteLine("Update succeeded")
with e ->
    Console.WriteLine("Update failed")
```

查詢實體屬性的子集

資料表查詢可以只從實體中取出幾個屬性，而不是所有屬性。這項稱為「投射」的技術可以改善查詢效能，尤其是針對大型實體。在這裡，您只會使用和傳回電子郵件地址 `DynamicTableEntity` `EntityResolver`。本機儲存體模擬器不支援投影，因此此程式碼只有在您使用表格服務上的帳戶時才會執行。

```
// Define the query, and select only the Email property.
let projectionQ = TableQuery<DynamicTableEntity>().Select [|"Email"|]

// Define an entity resolver to work with the entity after retrieval.
let resolver = EntityResolver<string>(fun pk rk ts props etag ->
    if props.ContainsKey("Email") then
        props["Email"].StringValue
    else
        null
)

let resolvedResults = table.ExecuteQuery(projectionQ, resolver, null, null)
```

以非同步方式擷取頁面中的實體

如果您要讀取大量的實體，而且想要在抓取這些實體時進行處理，而不是等候它們全部傳回，您可以使用分段的查詢。在這裡，您會使用非同步工作流程在頁面中傳回結果，如此一來，當您在等候一組大型結果傳回時，不會封鎖執行。

```
let tableQ = TableQuery<Customer>()

let asyncQuery =
    let rec loop (cont: TableContinuationToken) = async {
        let! ct = Async.CancellationToken
        let! result = table.ExecuteQuerySegmentedAsync(tableQ, cont, ct) |> Async.AwaitTask

        // ...process the result here...

        // Continue to the next segment
        match result.ContinuationToken with
        | null -> ()
        | cont -> return! loop cont
    }
loop null
```

您現在會同步執行此計算：

```
let asyncResults = asyncQuery |> Async.RunSynchronously
```

刪除實體

您可以在抓取實體之後將其刪除。如同更新實體，如果實體自您抓取之後已經變更，就會失敗。

```
let deleteOp = TableOperation.Delete(customer)
table.Execute(deleteOp)
```

刪除資料表

您可以從儲存體帳戶中刪除資料表。已刪除的資料表在刪除後一段時間內，將無法重新建立。

```
table.DeleteIfExists()
```

後續步驟

既然您已瞭解資料表儲存體的基本概念，請遵循下列連結以瞭解更複雜的儲存體工作和 Azure Cosmos DB 資料表 API。

- [Azure Cosmos DB 資料表 API 簡介](#)
- [Storage Client Library for .NET 參考資料](#)
- [Azure 儲存體型別提供者](#)
- [Azure 儲存體團隊部落格](#)
- [設定連接字串](#)

F# Azure 相依性的套件管理

2020/11/2 • [Edit Online](#)

當您使用套件管理員時，取得 Azure 開發的套件很簡單。這兩個選項為 [Paket](#) 和 [NuGet](#)。

使用 Paket

如果您使用 [Paket](#) 作為相依性管理員，您可以使用此 `paket.exe` 工具來新增 Azure 相依性。例如：

```
> paket add nuget WindowsAzure.Storage
```

或者，如果您要使用 [Mono](#) 進行跨平臺 .net 開發：

```
> mono paket.exe add nuget WindowsAzure.Storage
```

這會將 `WindowsAzure.Storage` 您在目前目錄中的專案套件相依性集合新增至您的套件相依性集合，並修改該檔案 `paket.dependencies`，然後下載封裝。如果您先前已設定相依性，或正在使用另一個開發人員已設定相依性的專案，您可以在本機解析並安裝相依性，如下所示：

```
> paket install
```

或者，對於 Mono 開發：

```
> mono paket.exe install
```

您可以將所有套件相依性更新為最新版本，如下所示：

```
> paket update
```

或者，對於 Mono 開發：

```
> mono paket.exe update
```

使用 NuGet

如果您使用 [NuGet](#) 做為您的相依性管理員，您可以使用此 `nuget.exe` 工具來新增 Azure 相依性。例如：

```
> nuget install WindowsAzure.Storage -ExcludeVersion
```

或者，對於 Mono 開發：

```
> mono nuget.exe install WindowsAzure.Storage -ExcludeVersion
```

這會將 `WindowsAzure.Storage` 您在目前目錄中的專案套件相依性集合新增至集合，並下載套件。如果您先前已設定相依性，或正在使用另一個開發人員已設定相依性的專案，您可以在本機解析並安裝相依性，如下所示：

```
> nuget restore
```

或者，對於 Mono 開發：

```
> mono nuget.exe restore
```

您可以將所有套件相依性更新為最新版本，如下所示：

```
> nuget update
```

或者，對於 Mono 開發：

```
> mono nuget.exe update
```

參考組件

若要在 F# 腳本中使用您的封裝，您需要使用指示詞來參考封裝中包含的元件 `#r`。例如：

```
> #r "packages/WindowsAzure.Storage/lib/net40/Microsoft.WindowsAzure.Storage.dll"
```

如您所見，您必須指定 DLL 的相對路徑和完整的 DLL 名稱，這可能與封裝名稱不完全相同。路徑將包含 framework 版本，也可能包含套件版本號碼。若要尋找所有已安裝的元件，您可以在 Windows 命令列上使用像這樣的內容：

```
> cd packages/WindowsAzure.Storage  
> dir /s/b *.dll
```

或在 Unix shell 中，如下所示：

```
> find packages/WindowsAzure.Storage -name "*.dll"
```

這會提供您已安裝元件的路徑。您可以從該處選取您 framework 版本的正確路徑。