C# 7.0 new features

Bill Chung

tw MVC

關於我

- Bill Chung
- 海角點部落
- 專長：說故事

大綱

- 新增的數字表示法
- pattern matching
- expression-bodied
- throw expression
- local functions
- out 引數的新用法
- ValueTuple
- Deconstruct
- ref local and return
- ValueTask

新增的數字表示法

- 二進位表示法
- 在數字常值中使用底線 _ 當作數字分隔符號

```
//二進位數字表示法
int i = 0b0010;
Console.WriteLine(i);

// 使用底線讓數字更清晰
int j = 100_000;
Console.WriteLine(j);

int k = 1_00;
Console.WriteLine(k);

int l = 0b0010_0000;
Console.WriteLine(l);

double d = 3.141_592_653;
Console.WriteLine(d);
```

pattern matching

- 加強 `is` 運算子
- 加強 `switch` 敘述 `(when)`

以前你得要這樣寫

```
static void Main(string[] args)
{
    int i = 10;
    Execute(i);
    Console.ReadLine();
}

private static void Execute(object value)
{
    if (value is int)
    { Console.WriteLine((int)value); }
    else
    { Console.WriteLine("not int"); }
}
```



現在你可以這麼寫

```
static void Main(string[] args)
{
    int i = 10;
    Execute(i);
    Console.ReadLine();
}

private static void Execute(object value)
{
    if (value is int x)
    { Console.WriteLine(x); }
    else
    { Console.WriteLine("not int"); }
}
```



在 switch 上的運用

```
private static  void InitialShape(IShape shape, double x, double y)
{
    switch (shape )
    {
        case MyRectAngle s:
            s.Height = x;
            s.Width = y;
            break;
        case MyCircle s:
            s.Radius = x;
            break;
        case null:
            break;
    }
}
```

在 switch 上加入 when

```
private static void Execute(object value)
{
    switch (value)
    {
        case int x when x > 0 && x < 100:
            Console.WriteLine($"x 是小整數 : {x}");
            break;
        case int x when x > 99 && x < 1000:
            Console.WriteLine($"x 是大整數 : {x}");
            break;
        case int x:
            Console.WriteLine($"x 超出範圍");
            break;
        case string x:
            Console.WriteLine($"x 是字串 : {x}");
            break;
        default:
            Console.WriteLine("不在 case 內");
            break;
    }
}
```

編譯器玩了甚麼魔術

- is 其實是 as
- 結構型別會換成 Nullable<T>
- switch 和 when 會被 if 取代
- 容易造成方法複雜度增加

過去的 is 和 as 有甚麼缺點

- 在 C# 6.0(包含) 以前的
- is
- 會對變數只少做兩次存取
- as
- 無法處理不可為 null 的型別
- 當變數值為 null 的時候，無法分辨究竟是轉型失敗還是傳進來的值就是 null

更狂的 expression-bodied members

- 在 C# 6.0 開始出現此形式的寫法。本來只用在 method 和 read only property。
- C# 7.0 將這個形式擴張到
  - constructor
  - finalizer
  - getter and setter on property
  - indexer

```
public class MyCircle
{
    private double _radius;
    private string _name;
    public double Radius
    {
        get => _radius;
        set => this._radius = value;
    }

    public string Name
    {
        get => _name;
        set => this._name = value ?? "就是這個圓";
    }

    public MyCircle() => _radius =2;

    public double GetArea() =>
                (_radius > 0) ? Math.PI * (Math.Pow(_radius, 2)) : 0;
}
```

throw expression

- 以前 throw 是個單純的敘述(statement) 所以沒有辦法直接放在運算式中使用
- 在 C# 7.0 中新增了 throw 運算式(expression) 解決了這個問題。

```
public class MyCircle
{
    private double _radius;
    private string _name;
    public double Radius
    {
        get => _radius;
        set => _radius = value > 0 ? value : throw new ArgumentException();
    }

    public string Name
    {
        get => _name;
        set => this._name = value ?? throw new ArgumentException();
    }
}
```

local functions

- 改善只會被某個特定 method 呼叫的 method 到處躲來躲去的問題。
- 強化封裝性
- async / await 也能用。

沒有 local function 之前

```
static void Main(string[] args)
{
    List<string> list1 = new List<string>() { "1", "2", "3", "4", "5" };
    Display(list1);
    List<string> list2 = new List<string>() { "A", "B", "C", "D", "E" };
    Display(list2);
    Console.ReadLine();
}

private static void  Display(List<string> list)
{
    foreach (var item in list)
    {
        Console.WriteLine(list);
    }
}
```

使用 local function

```
static void Main(string[] args)
{
    List<string> list1 = new List<string>() { "1", "2", "3", "4", "5" };
    Display(list1);
    List<string> list2 = new List<string>() { "A", "B", "C", "D", "E" };
    Display(list2);
    Console.ReadLine();


    void Display(List<string> list)
    {
        foreach (var item in list)
        {
            Console.WriteLine(item);
        }
    }
}
```

還有一個超神奇的用法

```
static void Main(string[] args)
{
    List<string> list1 = new List<string>() { "1", "2", "3", "4", "5" };
    var list = list1;
    Display();
    List<string> list2 = new List<string>() { "A", "B", "C", "D", "E" };
    list = list2;
    Display();
    Console.ReadLine();


    void Display()
    {
        foreach (var item in list)
        {
            Console.WriteLine(item);
        }
    }
}
```

剛剛的 list 會變成甚麼？

```
[CompilerGenerated]
private struct <>c__DisplayClass0_0
{
    public List<string> list;
}
```

```
[CompilerGenerated] internal static void <Main>g__Display0_0
                                    (ref <>c__DisplayClass0_0 class_Ref1)
 {
     foreach (string str in class_Ref1.list)
     {
                Console.WriteLine(str);
      }
 }
```

out 引數的新用法

- 只有一點小改變 -- 可以直接在呼叫時宣告 out 引數

```
string s="123";

// 以前你要這樣寫
int i;
int.TryParse(s,out i);
Console.WriteLine(i);
// 現在你可以直接這樣寫
int.TryParse(s, out int j);
Console.WriteLine(j);

// ref 不能比照 out 使用
//Test(ref int y = 10);
```

ValueTuple

- ValueTuple 以泛型結構形式定義
- 可以大幅簡化過去使用 Tuple<> 的麻煩
- 以 nuget 套件的形式加入參考



那一年，我們使用的 Tuple

```
static void Main(string[] args)
{
    var data = GetSomthing();
    Console.WriteLine($" {data.Item1} : {data.Item2}");
    Console.ReadLine();
}

private static Tuple<int, string> GetSomthing()
{
    int i = 100;
    string s = "ABC";
    return Tuple.Create(i, s);
}
```

ValueTuple 的各種宣告方式

```
ValueTuple<int, string> x1 = ValueTuple.Create<int, string>(8, "ABC");
Console.WriteLine($"(1) {x1.Item1} : {x1.Item2}");

var x2 = (8, "ABC");
Console.WriteLine($"(2) {x2.Item1} : {x2.Item2}");

var x3 = (length: 8, letters: "ABC");
Console.WriteLine($"(3) {x3.length} : {x3.letters}");

(int length, string letters) x4 = (8, "ABC");
Console.WriteLine($"(4) {x4.length} : {x4.letters}");

(int length, string letters) x5 = (first: 8, second: "ABC");
Console.WriteLine($"(5) {x5.length} : {x5.letters}");
```

八個泛型的多載

```
public struct ValueTuple<T1, T2, T3, T4, T5, T6, T7, TRest>
    where TRest : struct
```

Deconstruct

- 讓型別具有使用指派運算子，也就是 = ，把特定內容的值指派給 ValueTuple 型別的變數。

建立 Deconstruct 執行個體方法

```
class Program
{
    static void Main(string[] args)
    {
        MyRectangle rect = new MyRectangle() { Width = 10, Height = 30 };
        (int x, int y)= rect;
        Console.WriteLine($"{x} -- {y}");
        Console.ReadLine();
    }
}

public class MyRectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
    public void Deconstruct(out int width, out int height)
    {
        width = this.Width;
        height = this.Height;
    }
}
```

Deconstruct 也可以用擴充方法形式

```
public class MyRectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}

public static class MyExtension
{
    public static void Deconstruct
            (this MyRectangle rect, out int width, out int height)
    {
        width = rect.Width;
        height = rect.Height;
    }
}
```

Deconstruct 一個有趣的小地方

```
MyRectangle rect = new MyRectangle() { Width = 5, Height = 60 };
 (int x, int y) = rect;
 Console.WriteLine($"{x} -- {y}");

 Console.ReadLine();
```



```
(new MyRectangle() { Width = 5, Height = 60 })
                           .Deconstruct(out int x, out int  y);
Console.WriteLine($"{x} -- {y}");
Console.ReadLine();
```

ref local and return

- ref 的新變革
  - 在區域變數內直接取得變數指標
  - 在方法回傳值直接回傳變數指標

以前要操作變數指標是一件麻煩事

```
static void Main(string[] args)
{
    int number = 100;
    unsafe
    {
        int* p = &number;
        Console.WriteLine(*p);
        *p = 999;
        Console.WriteLine(number);
    }
    Console.ReadLine();
}
```

現在你只要這麼做就行了

```
static void Main(string[] args)
{
    int number = 100;
    ref int p = ref number;
    Console.WriteLine(p);
    p = 999;
    Console.WriteLine(number);
    Console.ReadLine();
}
```

陣列處理 – 過去

```
static void Main(string[] args)
{
    string[] data = new string[] { "鼠", "牛", "虎", "兔" };
    int index = GetTigerIndex(data);
    data[index] = "老虎";
    Display(data);
    Console.ReadLine();
}

private static int GetTigerIndex(string[] data)
{
    return Array.IndexOf(data, "虎");
}
```

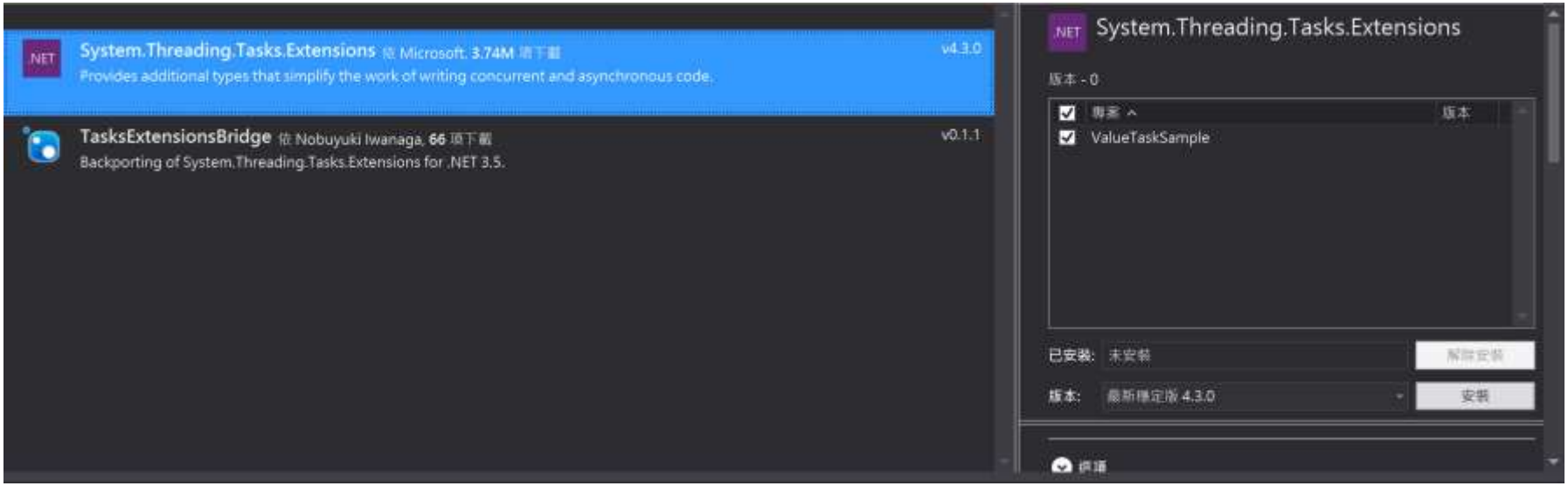
陣列處理 – ref return

```
static void Main(string[] args)
{
    string[] data = new string[] { "鼠", "牛", "虎", "兔" };
    ref string s = ref GetTiger(data);
    s = "大老虎";
    Display(data);
    Console.ReadLine();
}

private static ref string GetTiger(string[] data)
{
    return ref data[Array.IndexOf(data, "虎")];
}
```

ValueTask

- ValueTask 以泛型結構形式定義
- 以 nuget 套件的形式加入參考



```
async private static void Begin()
{
    int x = await Execute();
    Console.WriteLine(x);
}
async private static ValueTask<int> Execute()
{
    await Task.Delay(5000);
    return 300;
}
```

結構再度受到重視

Blog 是記錄知識的最佳平台



https://dotblogs.com.tw

Jetbrains 重構必備工具



ReSharper

https://www.jetbrains.com/resharper/

OzCode 偵錯的魔法師



業界師資、實戰教學



https://skilltree.my

謝謝各位

https://mvc.tw

- 本投影片所包含的商標與文字皆屬原著作者所有。
- 本投影片使用的圖片皆從網路搜尋。
- 本著作係採用姓名標示-非商業性-相同方式分享 3.0 台灣授權。閱讀本授權條款，請到 http://creativecommons.org/licenses/by-nc-sa/3.0/tw/，或寫信至Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.