30分钟掌握 C#7

1. out 变量（out variables）

以前我们使用out变量必须在使用前进行声明，C# 7.0 给我们提供了一种更简洁的语法 “使用时进行内联声明” 。如下所示：

```
var input = ReadLine();
if (int.TryParse(input, out var result))
{
    WriteLine("您输入的数字是：{0}",result);
}
else
{
    WriteLine("无法解析输入...");
}
```

上面代码编译后：

```
int num;
string s = Console.ReadLine();
if (int.TryParse(s, out num))
{
    Console.WriteLine("您输入的数字是：{0}", num);
}
else
{
    Console.WriteLine("无法解析输入...");
}
```

原理解析：所谓的 “内联声明” 编译后就是以前的原始写法，只是现在由编译器来完成。

备注：在进行内联声明时，即可直接写明变量的类型也可以写隐式类型，因为out关键字修饰的一定是局部变量。

2. 元组（Tuples）

元组（Tuple）在 .Net 4.0 的时候就有了，但元组也有些缺点，如：

1）Tuple 会影响代码的可读性，因为它的属性名都是：Item1，Item2.. 。

2）Tuple 还不够轻量级，因为它是引用类型（Class）。

备注：上述所指 Tuple 还不够轻量级，是从某种意义上来说的或者是一种假设，即假设分配操作非常的多。

C# 7 中的元组（ValueTuple）解决了上述两个缺点：

1）ValueTuple 支持语义上的字段命名。

2）ValueTuple 是值类型（Struct）。

1. 如何创建一个元组？

```
var tuple = (1, 2);                           // 使用语法糖创建元组
var tuple2 = ValueTuple.Create(1, 2);         // 使用静态方法【Create】创建元组
var tuple3 = new ValueTuple<int, int>(1, 2);  // 使用 new 运算符创建元组

WriteLine($"first：{tuple.Item1}, second：{tuple.Item2}, 上面三种方式都是等价的。");
```

原理解析：上面三种方式最终都是使用 new 运算符来创建实例。

2. 如何创建给字段命名的元组？

```
// 左边指定字段名称
(int one, int two) tuple = (1, 2);
WriteLine($"first：{tuple.one}, second：{tuple.two}");

// 右边指定字段名称
var tuple2 = (one: 1, two: 2);
WriteLine($"first：{tuple2.one}, second：{tuple2.two}");

// 左右两边同时指定字段名称
(int one, int two) tuple3 = (first: 1, second: 2);    /* 此处会有警告：由于目标类型（xx）已指定了其它名称，因为忽略元组名称xxx */
WriteLine($"first：{tuple3.one}, second：{tuple3.two}");
```

注：左右两边同时指定字段名称，会使用左边的字段名称覆盖右边的字段名称（一一对应）。

原理解析：上述给字段命名的元组在编译后其字段名称还是：Item1, Item2...，即：“命名” 只是语义上的命名。

3. 什么是解构？

解构顾名思义就是将整体分解成部分。

4. 解构元组，如下所示：

```
var (one, two) = GetTuple();

WriteLine($"first：{one}, second：{two}");
```

```
static (int, int) GetTuple()
{
    return (1, 2);
}
```

原理解析：解构元组就是将元组中的字段值赋值给声明的局部变量（编译后可查看）。

备注：在解构时 “=” 左边能提取变量的数据类型（如上所示），元组中字段类型相同时即可提取具体类型也可以是隐式类型，但元组中字段类型不相同时只能提取隐式类型。

5. 解构可以应用于 .Net 的任意类型，但需要编写 Deconstruct 方法成员（实例或扩展）。如下所示：

```
public class Student
{
    public Student(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public string Name { get; set; }

    public int Age { get; set; }

    public void Deconstruct(out string name, out int age)
    {
        name = Name;
        age = Age;
    }
}
```

使用方式如下：

```
var (Name, Age) = new Student("Mike", 30);

WriteLine($"name：{Name}, age：{Age}");
```

原理解析：编译后就是由其实例调用 Deconstruct 方法，然后给局部变量赋值。

Deconstruct 方法签名：

```
// 实例签名
public void Deconstruct(out type variable1, out type variable2...)

// 扩展签名
public static void Deconstruct(this type instance, out type variable1, out type variable2...)
```

总结：1. 元组的原理是利用了成员类型的嵌套或者是说成员类型的递归。2. 编译器很牛B才能提供如此优美的语法。

```
使用 ValueTuple 则需要导入: Install - Package System.ValueTuple
```

3. 模式匹配（Pattern matching）

1. is 表达式（is expressions），如：

```
static int GetSum(IEnumerable<object> values)
{
    var sum = 0;
    if (values == null) return sum;

    foreach (var item in values)
    {
        if (item is short)     // C# 7 之前的 is expressions
        {
            sum += (short)item;
        }
        else if (item is int val)  // C# 7 的 is expressions
        {
            sum += val;
        }
        else if (item is string str && int.TryParse(str, out var result))  // is expressions 和 out variables 结合使用
        {
            sum += result;
        }
        else if (item is IEnumerable<object> subList)
        {
            sum += GetSum(subList);
        }
    }

    return sum;
}
```

使用方法：

```
条件控制语句（obj is type variable）
{
   // Processing...
}
```

原理解析：此 is 非彼 is ，这个扩展的 is 其实是 as 和 if 的组合。即它先进行 as 转换再进行 if 判断，判断其结果是否为 null，不等于 null 则执行语句块逻辑，反之不行。由上可知其实C# 7之前我们也可实现类似的功能，只是写法上比较繁琐。

2. switch语句更新（switch statement updates），如：

```
static int GetSum(IEnumerable<object> values)
{
    var sum = 0;
    if (values == null) return 0;

    foreach (var item in values)
    {
        switch (item)
        {
            case 0:                // 常量模式匹配
                break;
            case short sval:       // 类型模式匹配
                sum += sval;
                break;
            case int ival:
                sum += ival;
                break;
            case string str when int.TryParse(str, out var result):   // 类型模式匹配 + 条件表达式
                sum += result;
                break;
            case IEnumerable<object> subList when subList.Any():
                sum += GetSum(subList);
                break;
            default:
                throw new InvalidOperationException("未知的类型");
        }
    }

    return sum;
}
```

使用方法：

```
switch (item)
{
    case type variable1:
        // processing...
        break;
    case type variable2 when predicate:
        // processing...
        break;
    default:
        // processing...
        break;
}
```

原理解析：此 switch 非彼 switch，编译后你会发现扩展的 switch 就是 as 、if 、goto 语句的组合体。同 is expressions 一样，以前我们也能实现只是写法比较繁琐并且可读性不强。

总结：模式匹配语法是想让我们在简单的情况下实现类似与多态一样的动态调用，即在运行时确定成员类型和调用具体的实现。

4. 局部引用和引用返回 (Ref locals and returns)

我们知道 C# 的 ref 和 out 关键字是对值传递的一个补充，是为了防止值类型大对象在Copy过程中损失更多的性能。现在在C# 7中 ref 关键字得到了加强，它不仅可以获取值类型的引用而且还可以获取某个变量（引用类型）的局部引用。如：

```
static ref int GetLocalRef(int[,] arr, Func<int, bool> func)
{
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            if (func(arr[i, j]))
            {
                return ref arr[i, j];
            }
        }
    }

    throw new InvalidOperationException("Not found");
}
```

Call：

```
int[,] arr = { { 10, 15 }, { 20, 25 } };
ref var num = ref GetLocalRef(arr, c => c == 20);
num = 600;

Console.WriteLine(arr[1, 0]);
```

Print results：

```
600
```

使用方法：

1. 方法的返回值必须是引用返回：

a) 声明方法签名时必须在返回类型前加上 ref 修饰。

b) 在每个 return 关键字后也要加上 ref 修饰，以表明是返回引用。

2. 分配引用（即赋值），必须在声明局部变量前加上 ref 修饰，以及在方法返回引用前加上 ref 修饰。

注：C# 开发的是托管代码，所以一般不希望程序员去操作指针。并由上述可知在使用过程中需要大量的使用 ref 来标明这是引用变量（编译后其实没那么多），当然这也是为了提高代码的可读性。

总结：虽然 C# 7 中提供了局部引用和引用返回，但为了防止滥用所以也有诸多约束，如：

1. 你不能将一个值分配给 ref 变量，如：

```
ref int num = 10;   // error：无法使用值初始化按引用变量
```

2. 你不能返回一个生存期不超过方法作用域的变量引用，如：

```
public ref int GetLocalRef(int num) => ref num;   // error: 无法按引用返回参数，因为它不是 ref 或 out 参数
```

3. ref 不能修饰 “属性” 和 “索引器” 。

```
var list = new List<int>();
ref var n = ref list.Count;  // error: 属性或索引器不能作为 out 或 ref 参数传递
```

原理解析：非常简单就是指针传递，并且个人觉得此语法的使用场景非常有限，都是用来处理大对象的，目的是减少GC提高性能。

5. 局部函数（Local functions）

C# 7 中的一个功能 “局部函数” ，如下所示：

```
 static IEnumerable<char> GetCharList(string str)
 {
     if (IsNullOrWhiteSpace(str))
         throw new ArgumentNullException(nameof(str));

     return GetList();

     IEnumerable<char> GetList()
     {
         for (int i = 0; i < str.Length; i++)
         {
             yield return str[i];
         }
     }
 }
```

使用方法：

```
[数据类型,void] 方法名（[参数]）
{
   // Method body；[] 里面都是可选项
}
```

原理解析：局部函数虽然是在其他函数内部声明，但它编译后就是一个被 internal 修饰的静态函数，它是属于类，至于它为什么能够使用上级函数中的局部变量和参数呢？那是因为编译器会根据其使用的成员生成一个新类型（Class/Struct）然后将其传入函数中。由上可知则局部函数的声明跟位置无关，并可无限嵌套。

总结：个人觉得局部函数是对 C# 异常机制在语义上的一次补充（如上例），以及为代码提供清晰的结构而设置的语法。但局部函数也有其缺点，就是局部函数中的代码无法复用（反射除外）。

6. 更多的表达式体成员（More expression-bodied members）

C# 6 的时候就支持表达式体成员，但当时只支持 “函数成员” 和 “只读属性” ，这一特性在C# 7中得到了扩展，它能支持更多的成员：构造函数、析构函数、带 get，set 访问器的属性、以及索引器。如下所示：

```
public class Student
{
    private string _name;

    // Expression-bodied constructor
    public Student(string name) => _name = name;

    // Expression-bodied finalizer
    ~Student() => Console.WriteLine("Finalized!");

    // Expression-bodied get / set accessors.
    public string Name
    {
        get => _name;
        set => _name = value ?? "Mike";
    }

    // Expression-bodied indexers
    public string this[string name] => Convert.ToBase64String(Encoding.UTF8.GetBytes(name));
}
```

备注：索引器其实在C# 6中就得到了支持，但其它三种在C# 6中未得到支持。

7. Throw 表达式（Throw expressions）

异常机制是C#的重要组成部分，但在以前并不是所有语句都可以抛出异常的，如：条件表达式（ ？ ：）、null合并运算符（??）、一些Lambda表达式。而使用 C# 7 您可在任意地方抛出异常。如：

```
public class Student
{
    private string _name = GetName() ?? throw new ArgumentNullException(nameof(GetName));

    private int _age;

    public int Age
    {
        get => _age;
        set => _age = value <= 0 || value >= 130 ? throw new ArgumentException("参数不合法") : value;
    }

    static string GetName() => null;
}
```

8. 扩展异步返回类型（Generalized async return types）

以前异步的返回类型必须是：Task、Task<T>、void，现在 C# 7 中新增了一种类型：ValueTask<T>，如下所示：

```
public async ValueTask<int> Func()
{
    await Task.Delay(3000);
    return 100;
}
```

总结：ValueTask<T> 与 ValueTuple 非常相似，所以就不列举： ValueTask<T> 与 Task 之间的异同了，但它们都是为了优化特定场景性能而新增的类型。

```
使用 ValueTask<T> 则需要导入： Install - Package System.Threading.Tasks.Extensions
```

9. 数字文本语法的改进（Numeric literal syntax improvements）

C# 7 还包含两个新特性：二进制文字、数字分隔符，如下所示：

```
var one = 0b0001;
var sixteen = 0b0001_0000;

long salary = 1000_000_000;
decimal pi = 3.141_592_653_589m;
```

注：二进制文本是以0b（零b）开头，字母不区分大小写；数字分隔符只有三个地方不能写：开头，结尾，小数点前后。

总结：二进制文本，数字分隔符 可使常量值更具可读性。