以下将是 C# 7.0 中所有计划的语言特性的描述。随着 Visual Studio “15” Preview 4 版本的发布，这些特性中的大部分将活跃起来。现在是时候来展示这些特性，你也告诉借此告诉我们你的想法！

C＃7.0 增加了许多新功能，并专注于数据消费，简化代码和性能的改善。或许最大的特性就是元组和模式匹配，元组可以很容易地拥有多个返回结果，而模型匹配可以根据数据的“形”的不同来简化代码。我们希望，将它们结合起来，从而使你的代码更加简洁高效，也可以使你更加快乐并富有成效。

请点击 Visual Studio 窗口顶部的反馈按钮，告诉我们哪些是你不期待的特性或者你关于提升这些特性的思考。还有许多功能没有在 Preview 4 版本中实现。接下来我会描述一些我们发布的最终版本里将会起作用的特性，和一些一旦不起作用机即会删除掉的特性。我也是支持对这些计划作出改变，尤其是作为我们从你那儿得到反馈的结果。当最终版本发布时，这些特性中的一些将会改变或者删除。

如果你好奇这些特性的设计过程，你可以在 Roslyn GitHub site 上找到很多设计笔记和讨论。

希望 C＃7.0 能带给你快乐！

**输出变量**

在当前的 C＃ 中，使用输出参数并不像我们想的那样方便。在你调用一个无输出参数的方法之前，首先必须声明一个变量并传递给它。如果你没有初始化这些变量，你就无法使用 var 来声明它们，除非先指定完整的类型：

[IMG_256](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

public void PrintCoordinates(Point p)

{

int x, y; // have to "predeclare"

p.GetCoordinates(out x, out y);

WriteLine($"({x}, {y})");

}

[IMG_257](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

在 C＃7.0 中，我们正在增加输出变量和声明一个作为能够被传递的输出实参的变量的能力：

[IMG_258](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

public void PrintCoordinates(Point p)

{

p.GetCoordinates(out int x, out int y);

WriteLine($"({x}, {y})");

}

[IMG_259](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

注意，变量是在封闭块的范围内，所以后续也可以使用它们。大多数类型的声明不建立自己的范围，因此在他们中声明的变量通常会被引入到封闭范围。

Note：在 Preview 4 中，适用范围规则更为严格：输出变量的作用域是声明它们的语句，因此直到下个版本发布时，上面的示例才会起作用。

由于输出变量直接被声明为实参传递给输出形参，编译器通常会告诉他们应该是的类型（除非有冲突过载），所以使用 var 来代替声明它们的方式是比较好的：

p.GetCoordinates(out var x, out var y);

输出参数的一种常见用法是Try模式，其中一个布尔返回值表示成功，输出参数就会携带所获的结果：

public void PrintStars(string s)

{

if (int.TryParse(s, out var i)) { WriteLine(new string('\*', i)); }

else { WriteLine("Cloudy - no stars tonight!"); }

}

注意：这里i只用在 if 语句来定义它，所以 Preview 4 可以将这个处理的很好。

我们计划允许以 a\* 为形式的“通配符”作为输出参数，这会让你忽略了你不关心参数：

p.GetCoordinates(out int x, out \*); // I only care about x

Note：在 C#7.0 中是否会包含通配符还不确定。

**模式匹配**

C＃ 7.0 引入了模式概念。抽象地讲，模式是句法元素，能用来测试一个数据是否具有某种“形”，并在被应用时，从值中提取有效信息。

C＃7.0 中的模式示例：

* C 形式的常量模式（C是C#中的常量表达式），可以测试输入是否等于C
* T X 形式的类型模式（T是一种类型、X是一个标识符），可以测试输入是否是T类型，如果是，会将输入值提取成T类型的新变量X
* Var x 形式的 Var 模式（x是一个标识符），它总是匹配的，并简单地将输入值以它原本的类型存入一个新变量X中。

这仅仅是个开始 - 模式是一种新型的 C＃ 中的语言元素。未来，我们希望增加更多的模式到 C# 中。

在 C＃7.0，我们正在加强两个现有的具有模式的语言结构：

* is 表达式现在具有一种右手侧的模式，而不仅仅是一种类型
* switch 语句中的 case 语句现在可以使用匹配模式，不只是常数值

在 C＃的未来版本中，我们可能会增加更多的被用到的模式。

**具有模式的 IS 表达式**

下面是使用 is 表达式的示例，其中利用了常量模式和类型模式：

[IMG_260](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

public void PrintStars(object o)

{

if (o is null) return; // constant pattern "null"

if (!(o is int i)) return; // type pattern "int i"

WriteLine(new string('\*', i));

}

[IMG_261](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

正如你们看到，模式变量（模式引入的变量）和早前描述的输出变量比较类似，它们可以在表达式中间声明，并在最近的范围内使用。就像输出变量一样，模式变量是可变的。

注：就像输出变量一样，严格范围规则适用于Preview 4。

模式和 Try方法可以很好地协同：

if (o is int i || (o is string s && int.TryParse(s, out i)) { /\* use i \*/ }

**具有模式的 Switch 语句**

我们正在归纳 Switch 语句：

* 可以设定任何类型的 Switch 语句（不只是原始类型）
* 模式可以用在 case 语句中
* Case 语句可以有特殊的条件

下面是一个简单的例子：

[IMG_262](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

switch(shape)

{

case Circle c:

WriteLine($"circle with radius {c.Radius}");

break;

case Rectangle s when (s.Length == s.Height):

WriteLine($"{s.Length} x {s.Height} square");

break;

case Rectangle r:

WriteLine($"{r.Length} x {r.Height} rectangle");

break;

default:

WriteLine("<unknown shape>");

break;

case null:

throw new ArgumentNullException(nameof(shape));

}

[IMG_263](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

关于新扩展的 switch 语句，有几点需要注意：

* Case 语句的顺序现在变得重要：就像 catch 语句一样，case 语句的范围现在可以相交，第一个匹配上的会被选中。此外，就像 catch 语句一样，编译器通过去除明显不会进入的 case 来帮助你。在此之前，你甚至不需要告诉判断的顺序，所以这并不是一个使用 case 语句的巨大的改变。
* 默认的语句还是最后被判断：尽管 null 的 case 语句在最后语句之前出现，它也会在默认语句被选中之前被测试。这是与现有 Switch 语义兼容的。然而，好的做法通常会将默认语句放到最后。
* Switch 不会到最后的 null 语句：这是因为当前 IS 表达式的例子具有类型匹配，不会匹配到 null。这保证了空值不会不小心被任何的类型模式匹配上的情况;你必须更明确如何处理它们（或放弃它而使用默认语句）。

通过一个 case 引入模式变量:标签仅在相应的 Switch 范围内。

**元组**

这是一个从方法中返回多个值的常见模式。目前可选用的选项并非是最佳的：

* 输出参数：使用起来比较笨拙（即使有上述的改进），他们在使用异步方法是不起作用的。
* System.Tuple<...> 返回类型：冗余使用和请求一个元组对象的分配。
* 方法的定制传输类型：对于类型，具有大量的代码开销，其目的只是暂时将一些值组合起来。
* 通过动态返回类型返回匿名类型：很高的性能开销，没有静态类型检查。

在这点要做到更好，C＃7.0 增加的元组类型和元组文字：

(string, string, string) LookupName(long id) // tuple return type{

... // retrieve first, middle and last from data storage

return (first, middle, last); // tuple literal

}

这个方法可以有效地返回三个字符串，以元素的形式包含在一个元组值里。

这种方法的调用将会收到一个元组，并且可以单独地访问其中的元素：

var names = LookupName(id);

WriteLine($"found {names.Item1} {names.Item3}.");

Item1 等是元组元素的默认名称，也可以被一直使用。但他们不具有描述性，所以你可以选择添加更好的：

(string first, string middle, string last) LookupName(long id) // tuple elements have names

现在元组的接收者有多个具有描述性的名称可用：

var names = LookupName(id);

WriteLine($"found {names.first} {names.last}.");

你也可以直接在元组文字指定元素名称：

return (first: first, middle: middle, last: last); // named tuple elements in a literal

一般可以给元组类型分配一些彼此无关的名称：只要各个元素是可分配的，元组类型就可以自如地转换为其他的元组类型。也有一些限制，特别是对元组文字，即常见的和告警错误，如不慎交换元素名称的情况下，就会出现错误。

Note：这些限制尚未在 Preview 4 中实现。

元组是值类型的，它们的元素是公开的，可变的。他们有值相等，如果所有的元素都是成对相等的（并且具有相同的哈希值），那么这两个元组也是相等的（并且具有相同的哈希值）。

这使得在需要返回多个值的情况下，元组会非常有用。举例来说，如果你需要多个 key 值的字典，使用元组作为你的 key 值，一切会非常顺利。如果你需要在每个位置都具有多个值的列表，使用元组进行列表搜索，会工作的很好。

Note：元组依赖于一组基本类型，却不包括在 Preview 4 中。为了使该特性工作，你可以通过 NuGet 获取它们：

* 右键单击 Solution Explorer 中的项目，然后选择“管理的NuGet包......”
* 选择“Browse”选项卡，选中“Include prerelease”，选择“nuget.org”作为“Package source”
* 搜索“System.ValueTuple”并安装它。

**解构**

消耗元组的另一种方法是将解构它们。一个解构声明是一个将元组（或其他值）分割成部分并单独分配到新变量的语法：

(string first, string middle, string last) = LookupName(id1); // deconstructing declaration

WriteLine($"found {first} {last}.");

在解构声明中，您可以使用 var 来声明单独的变量：

(var first, var middle, var last) = LookupName(id1); // var inside

或者将一个单独的 var 作为一个缩写放入圆括号外面：

var (first, middle, last) = LookupName(id1); // var outside

你也可以使用解构任务来解构成现有的变量

(first, middle, last) = LookupName(id2); // deconstructing assignment

解构不只是应用于元组。任何的类型都可以被解构，只要它具有（实例或扩展）的解构方法：

public void Deconstruct(out T1 x1, ..., out Tn xn) { ... }

输出参数构成了解构结果中的值。

（为什么它使用了参数，而不是返回一个元组？这是为了让你针对不同的值拥有多个重载）。

[IMG_264](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

class Point

{

public int X { get; }

public int Y { get; }

public Point(int x, int y) { X = x; Y = y; }

public void Deconstruct(out int x, out int y) { x = X; y = Y; }

}

(var myX, var myY) = GetPoint(); // calls Deconstruct(out myX, out myY);

[IMG_265](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

这是一种常见的模式，以一种对称的方式包含了构建和解构。

对于输出变量，我们计划在解构中加入通配符，来化简你不关心的变量：

(var myX, \*) = GetPoint(); // I only care about myX

Note：通配符是否会出现在C＃7.0中，这仍是未知数。

**局部函数**

有时候，一个辅助函数可以在一个独立函数内部起作用。现在，你可以以一个局部函数的方式在其它函数内部声明这样的函数：

[IMG_266](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

public int Fibonacci(int x)

{

if (x < 0) throw new ArgumentException("Less negativity please!", nameof(x));

return Fib(x).current;

(int current, int previous) Fib(int i)

{

if (i == 0) return (1, 0);

var (p, pp) = Fib(i - 1);

return (p + pp, p);

}

}

[IMG_267](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

闭合范围内的参数和局部变量在局部函数的内部是可用的，就如同它们在 lambda 表达式中一样。

举一个例子，迭代的方法实现通常需要一个非迭代的封装方法，以便在调用时检查实参。（迭代器本身不启动运行，直到 MoveNext 被调用）。局部函数非常适合这样的场景：

[IMG_268](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

public IEnumerable<T> Filter<T>(IEnumerable<T> source, Func<T, bool> filter)

{

if (source == null) throw new ArgumentNullException(nameof(source));

if (filter == null) throw new ArgumentNullException(nameof(filter));

return Iterator();

IEnumerable<T> Iterator()

{

foreach (var element in source)

{

if (filter(element)) { yield return element; }

}

}

}

[IMG_269](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

如果迭代器有一个私有方法传递给过滤器，那么当其它成员意外的使用迭代器时，迭代器也变得可用（即使没有参数检查）。此外，还会采取相同的实参作为过滤器，以便替换范围内的参数。

注意：在 Preview 4，局部函数在调用之前，必须被声明。这个限制将会被松开，以便使得局部函数从定义分配中读取时，能够被调用。

**文字改进**

C＃7.0 允许 \_ 出现，作为数字分隔号：

var d = 123\_456;

var x = 0xAB\_CD\_EF;

你可以将 \_ 放入任意的数字之间，以提高可读性，它们对值没有影响。

此外，C＃7.0 引入了二进制文字，这样你就可以指定二进制模式而不用去了解十六进制。

var b = 0b1010\_1011\_1100\_1101\_1110\_1111;

**引用返回和局部引用**

就像在 C# 中通过引用来传递参数（使用引用修改器），你现在也可以通过引用来返回参数，同样也可以以局部变量的方式存储参数。

[IMG_270](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

public ref int Find(int number, int[] numbers)

{

for (int i = 0; i < numbers.Length; i++)

{

if (numbers[i] == number)

{

return ref numbers[i]; // return the storage location, not the value }

}

throw new IndexOutOfRangeException($"{nameof(number)} not found");

}

int[] array = { 1, 15, -39, 0, 7, 14, -12 };ref int place = ref Find(7, array); // aliases 7's place in the array

place = 9; // replaces 7 with 9 in the array

WriteLine(array[4]); // prints 9

[IMG_271](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

这是绕过占位符进入大数据结构的好方法。例如，一个游戏也许会将它的数据保存在大型预分配的阵列结构中（为了避免垃圾回收机制暂停）。方法可以将直接引用返回成一个结构，通过它的调用者可以读取和修改它。

也有一些限制，以确保安全：

* 你只能返回“安全返回”的引用：一个是传递给你的引用，一个是指向对象中的引用。
* 本地引用会被初始化成一个本地存储，并且不能指向另一个存储。

**异步返回类型**

到现在为止，C＃ 的异步方法必须返回 void，Task 或 Task<T>。C＃7.0 允许其它类型以这种能从一个方法中返回的方式被定义，因为它们可以以异步方法被返回的方式来定义其它类型。

例如我们计划建立一个 ValueTask<T> 结构类型的数据。建立它是为了防止异步运行的结果在等待时已可用的情境下，对 Task<T> 进行分配。对于许多实例中设计缓冲的异步场景，这可以大大减少分配的数量并显著地提升性能。

Note：异步返回类型尚未在 Preview 4 中提供。

更多的 expression bodied 成员：

expression bodied 的方法和属性是对 C# 6.0 的巨大提升。C# 7.0 为 expression bodied 事件列表增加了访问器，结构器和终结器。

[IMG_272](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

class Person

{

private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int, string>();

private int id = GetId();

public Person(string name) => names.TryAdd(id, name); // constructors

~Person() => names.TryRemove(id, out \*); // destructors

public string Name

{

get => names[id]; // getters

set => names[id] = value; // setters }

}

[IMG_273](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

Note：这些额外增加的 expression bodied 的成员尚未在 Preview 4 中提供。

这是社区共享的示例，而不是 Microsoft C# 编译团队提供的，还是开源的！

## **Throw 表达式**

在表达式中间抛出一个异常是很容易的：只需为自己的代码调用一个方法！但在 C＃7.0 中，我们允许在任意地方抛出一个表达式：

[IMG_274](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

class Person

{

public string Name { get; }

public Person(string name) => Name = name ?? throw new ArgumentNullException(name);

public string GetFirstName()

{

var parts = Name.Split(" ");

return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");

}

public string GetLastName() => throw new NotImplementedException();

}

[IMG_275](http://www.cnblogs.com/powertoolsteam/p/javascript:void(0);" \o "复制代码)

Note：Throw 表达式尚未在Preview 4中提供。

http://www.cnblogs.com/qianxingdewoniu/p/5827756.html