

C# 元组类型

2018/05/15 • 👤 👤 🌐 👤

本文内容

[命名元组和未命名元组](#)

[元组投影初始值设定项](#)

[相等和元组](#)

[赋值和元组](#)

[作为方法返回值的元组](#)

[析构](#)

[元组作为 out 参数](#)

[结束语](#)

C# 元组是使用轻量语法定义的类型。其优点包括：更简单的语法，基于元素数量（称为“基数”）和元素类型的转换规则，以及一致的副本、相等测试和赋值规则。但另一方面，元组不支持一些与继承相关的面向对象的语法。[C# 7.0 中的新增功能](#)文章中的“元组”一节对其进行了概述。

在本文中，你将了解用于控制 C# 7.0 及更高版本中的元组的语言规则、这些规则的各种用法，以及有关如何使用元组的初步指导。

❗ 备注

新的元组功能需要 [ValueTuple](#) 类型。为在不包括该类型的平台上使用它，必须添加 NuGet 包 [System.ValueTuple](#)。

这类似于依赖框架提供的类型的其他语言功能。例如，依赖 `INotifyCompletion` 接口的 `async` 和 `await`，依赖 `IEnumerable<T>` 的 LINQ。但是，随着 .NET 越来越不依赖平台，交付机制也在发生改变。.NET Framework 交付频率可能不会与语言编译器的始终相同。新语言功能依赖于新类型时，这些类型将在交付语言功能时以 NuGet 包的形式提供。这些新类型添加到 .NET 标准 API 并作为框架的一部分交付后，将删除 NuGet 包要求。

我们先解释一下为什么要添加新的元组支持。方法返回单个对象。借助元组，可以更轻松地对该单个对象中的多个值打包。

.NET Framework 已具有泛型 `Tuple` 类。但这些类有两个主要限制。其一，`Tuple` 类将其属性命名为 `Item1`、`Item2` 等。这些名称未承载任何语义信息。使用这些 `Tuple` 类型

无法表达各属性的含义。通过新的语言功能，可对元组中的各元素进行声明并为其赋予有意义的语义名称。

Tuple 类因其引用类型会导致更多性能问题。使用任一 Tuple 类型即意味着分配对象。在热路径中，分配许多小型对象可能会对应用程序性能产生明显的影响。因此，元组的语言支持使用新的 ValueTuple 结构。


为避免这些缺陷，可创建 class 或 struct 来承载多个元素。但这样做不仅加大了工作量，还掩盖了你的设计意图。创建 struct 或 class 意味着定义一个具有数据和行为的类型。很多时候，你其实只是想存储单个对象中的多个值而已。

这些语言功能和 ValueTuple 泛型结构共同实施以下规则：不能向这些元组类型添加任何行为（方法）。所有 ValueTuple 类型都是*可变结构*。每个成员字段都是公共字段。这使它们变得非常轻量。但是，这意味着在要求永久性的场合无法使用元组。

元组是比 class 和 struct 类型更为简单灵活的数据容器。我们来探讨一下它们之间的差异。

命名元组和未命名元组

ValueTuple 结构具有名为 Item1、Item2、Item3 等的字段，与现有 Tuple 类型中定义的属性类似。这些名称是可用于*未命名元组*的唯一名称。如果不为元组提供任何备用字段名称，即表示创建了一个未命名元组：

C#	 复制
<pre>var unnamed = ("one", "two");</pre>	

上例中的元组已使用文本常量进行初始化，并且不会有 C# 7.1 中使用“元组字段名称投影”创建的元素名称。


但是，在初始化元组时，可以使用新语言功能为每个字段提供更好的名称。如此便创建了*命名元组*。命名元组仍将元素命名为 Item1、Item2、Item3 等。不过，它们还会为这些已命名的元素提供同义词。通过为每个元素指定名称即可创建命名元组。其中一种方式是在元组初始化过程中指定名称：

C#	 复制
<pre>var named = (first: "one", second: "two");</pre>	


这些同义词由编译器和语言处理，因此，你可以高效地使用命名元组。IDE 和编辑器可以使用 Roslyn API 读取这些语义名称。可以在同一程序集中的任何位置通过这些语义名

称引用命名元组的元素。 编译器在生成已编译的输出时，会将已定义的名称替换为 `Item*` 等效项。 已编译的 Microsoft 中间语言 (MSIL) 不包括为这些元素赋予的名称。

从 C# 7.1 开始，元组的字段名称可能会通过用于初始化此元组的变量提供。 这称为[元组投影初始值设定项](#)。 以下代码用于创建名为 `accumulation` 的元组，包含元素 `count`（整数）和 `sum`（双精度）。

C#	 复制
<pre>var sum = 12.5; var count = 5; var accumulation = (count, sum);</pre>	

编译器必须传达为从公共方法或属性返回的元组创建的这些名称。 在这种情况下，编译器会在方法上添加 [TupleElementNamesAttribute](#) 特性。 此特性包含一个 [TransformNames](#) 列表属性，该属性包含为元组中的每个元素赋予的名称。


 **备注**

Visual Studio 等开发工具还读取其元数据，并提供 IntelliSense 和其他使用元数据字段名称的功能。

请务必理解新元组和 `ValueTuple` 类型的这些基础知识，这样才能理解将命名元组赋给彼此的规则。

元组投影初始值设定项

一般情况下，元组投影初始值设定项使用元组初始化语句右侧的变量或字段名称。 如果未提供显式名称，上述名称将优先于任何投影的名称。 例如，在以下初始值设定项中，元素为 `explicitFieldOne` 和 `explicitFieldTwo`，而非 `localVariableOne` 和 `localVariableTwo`：

C#	 复制
<pre>var localVariableOne = 5; var localVariableTwo = "some text"; var tuple = (explicitFieldOne: localVariableOne, explicitFieldTwo: localVariableTwo);</pre>	

对于任何未提供显式名称的字段，将投影适用的隐式名称。 不要求提供显式或隐式语义名称。 以下初始化表达式具有字段名称 `Item1` 其值为 42 和 `stringContent`（其值为“The answer to everything”）：

C#	复制
<pre>var stringContent = "The answer to everything"; var mixedTuple = (42, stringContent);</pre>	

在以下两种情况下，不会将候选字段名称投影到元组字段：

1. 候选名称是保留元组名称时。示例包括 `Item3`、`ToString` 或 `Rest`。
2. 候选名称重复了另一元组的显式或隐式字段名称时。

这两个条件可避免多义性。如果这些名称已用作元组中某字段的字段名称，它们将导致多义。这两个条件都不会导致编译时错误。但不会向没有投影名称的元素投影语义名称。以下示例说明了这两个条件：

C#	复制	运行
<pre>var ToString = "This is some text"; var one = 1; var Item1 = 5; var projections = (ToString, one, Item1); // Accessing the first field: Console.WriteLine(projections.Item1); // There is no semantic name 'ToString' // Accessing the second field: Console.WriteLine(projections.one); Console.WriteLine(projections.Item2); // Accessing the third field: Console.WriteLine(projections.Item3); // There is no semantic name 'Item1`. var pt1 = (X: 3, Y: 0); var pt2 = (X: 3, Y: 4); var xCoords = (pt1.X, pt2.X); // There are no semantic names for the fields // of xCoords. // Accessing the first field: Console.WriteLine(xCoords.Item1); // Accessing the second field: Console.WriteLine(xCoords.Item2);</pre>		



这些情况不会导致编译器错误，因为当元组字段名称投影不可用时，它将成为使用 C# 7.0 编写的代码的一项重大改变。

相等和元组



从 C# 7.3 开始，元组类型支持 `==` 和 `!=` 运算符。这些运算符按顺序将左边参数的每个成员与右边参数的每个成员进行比较。这些比较将发生短路。只要有一对不相等，它们即会停止计算成员。以下代码示例使用 `==`，但比较规则均适用于 `!=`。以下代码示例演示两对整数的相等比较：

C#	 复制	 运行
<pre>var left = (a: 5, b: 10); var right = (a: 5, b: 10); Console.WriteLine(left == right); // displays 'true'</pre>		

有几条规则，可使元组相等测试更方便。如果其中一个元组是可以为空值的元组，则元组相等将执行[提升转换](#)，如以下代码中所示：

C#	 复制	 运行
<pre>var left = (a: 5, b: 10); var right = (a: 5, b: 10); (int a, int b)? nullableTuple = right; Console.WriteLine(left == nullableTuple); // Also true</pre>		

元组相等还将对这两个元组的每个成员执行隐式转换。这些转换包括提升转换、扩大转换或其他隐式转换。以下示例演示整数 2 元组可以与较长的 2 元组进行比较，因为进行了从整数元组到较长元组的隐式转换：



C#	 复制	 运行
<pre>// lifted conversions var left = (a: 5, b: 10); (int? a, int? b) nullableMembers = (5, 10); Console.WriteLine(left == nullableMembers); // Also true // converted type of left is (long, long) (long a, long b) longTuple = (5, 10); Console.WriteLine(left == longTuple); // Also true // comparisons performed on (long, long) tuples (long a, int b) longFirst = (5, 10); (int a, long b) longSecond = (5, 10); Console.WriteLine(longFirst == longSecond); // Also true</pre>		

元组成员名称不参与相等测试。但是，如果其中一个操作数是含有显式名称的元组文本，则当这些名称与其他操作数的名称不匹配时，编译器将生成警告 CS8383。在两个操作数都为元组文本的情况下，警告位于右侧操作数，如以下示例中所述：

C#	 复制	 运行
----	--	--

```
(int a, string b) pair = (1, "Hello");
(int z, string y) another = (1, "Hello");
Console.WriteLine(pair == another); // true. Member names don't participate.
Console.WriteLine(pair == (z: 1, y: "Hello")); // warning: literal contains
different member names
```

最后，元组可能包含嵌套元组。元组相等通过嵌套元组比较每个操作数的“形状”，如下示例中所示：


C#	 复制	 运行
<pre>(int, (int, int)) nestedTuple = (1, (2, 3)); Console.WriteLine(nestedTuple == (1, (2, 3)));</pre>		

当两个元组具有不同形状时，比较它们是否相等（或不相等）将出现编译时错误。编译器不会尝试对嵌套元组进行任何析构来比较它们。

赋值和元组

语言支持在具有相同元素数量的元组类型之间赋值，其中每个右侧元素都可被隐式转换为相应的左侧元素。对于其他转换，不考虑进行赋值。当两个元组具有不同形状时，将一个元组分配给另一个元组将出现编译时错误。编译器不会尝试对嵌套元组进行任何析构来分配它们。让我们看一下元组类型之间允许的赋值类型。

注意以下示例中使用的这些变量：

C#	 复制
<pre>// The 'arity' and 'shape' of all these tuples are compatible. // The only difference is the field names being used. var unnamed = (42, "The meaning of life"); var anonymous = (16, "a perfect square"); var named = (Answer: 42, Message: "The meaning of life"); var differentNamed = (SecretConstant: 42, Label: "The meaning of life");</pre>	

前两个变量（unnamed 和 anonymous）没有为元素提供语义名称。字段名称为 Item1 和 Item2。后两个变量（named 和 differentName）为元素提供了语义名称。这两个元组具有不同的元素名称。

这四个元组具有相同数量的元素（称为“基数”），这些元素的类型也完全一样。因此可进行以下赋值：

C#	 复制
----	--

```

unnamed = named;

named = unnamed;
// 'named' still has fields that can be referred to
// as 'answer', and 'message':
Console.WriteLine($"{named.Answer}, {named.Message}");

// unnamed to unnamed:
anonymous = unnamed;

// named tuples.
named = differentNamed;
// The field names are not assigned. 'named' still has
// fields that can be referred to as 'answer' and 'message':
Console.WriteLine($"{named.Answer}, {named.Message}");


// With implicit conversions:
// int can be implicitly converted to long
(long, string) conversion = named;

```

请注意，元组的名称未赋值。元素的赋值顺序遵循元素在元组中的顺序。

元素类型或数量不同的元组不可赋值：

C#

 复制

```


// Does not compile.
// CS0029: Cannot assign Tuple(int,int,int) to Tuple(int, string)
var differentShape = (1, 2, 3);
named = differentShape;

```

作为方法返回值的元组

元组最常见的用途之一是作为方法返回值。我们来看一个示例。以下面的方法为例，该方法计算一个数列的标准差：

C#

 复制

```

public static double StandardDeviation(IEnumerable<double> sequence)
{
    // Step 1: Compute the Mean:
    var mean = sequence.Average();

    // Step 2: Compute the square of the differences between each number
    // and the mean:
    var squaredMeanDifferences = from n in sequence
                                select (n - mean) * (n - mean);

    // Step 3: Find the mean of those squared differences:
    var meanOfSquaredDifferences = squaredMeanDifferences.Average();
}

```

```
// Step 4: Standard Deviation is the square root of that mean:
var standardDeviation = Math.Sqrt(meanOfSquaredDifferences);
return standardDeviation;
}
```


① 备注

这些示例计算得出未修正的样本标准差。与 `Average` 扩展方法一样，修正后的样本标准差公式将与平均数之差的平方的总和除以 $(N-1)$ ，而不是 N 。有关这些标准差公式之间的区别的更多信息，请查看统计信息文本。

前面的代码采用教科书上的标准差公式。它会生成正确的答案，但实施起来非常低效。此方法对数列进行两次计算：一次生成平均数，一次生成与平均数之差的平方的平均数。（请记住，LINQ 查询进行迟缓计算，因此，在计算与平均数的差以及这些差的平均数时只需计算一次。）

有一个计算标准差的备用公式，它只对数列计算一次。此计算公式在计算数列时生成两个值：数列中所有项的总和，以及每个平方值的总和：

C#

 复制

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    double sum = 0;
    double sumOfSquares = 0;
    double count = 0;

    foreach (var item in sequence)
    {
        count++;
        sum += item;
        sumOfSquares += item * item;
    }

    var variance = sumOfSquares - sum * sum / count;
    return Math.Sqrt(variance / count);
}
```

此版本虽然只对序列进行一次枚举。但其代码不可重复使用。到后面，你会发现许多不同的统计计算会用到数列项数、数列总和以及数列平方和。让我们重构此方法，编写一个可生成这三个值的实用方法。所有这三个值都可以作为一个元组返回。

让我们更新此方法，以便将在计算过程中得出的三个值存储在一个元组中。以下是更新后的版本：

 复制

C#

 复制


```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    var computation = (Count: 0, Sum: 0.0, SumOfSquares: 0.0);

    foreach (var item in sequence)
    {
        computation.Count++;
        computation.Sum += item;
        computation.SumOfSquares += item * item;
    }

    var variance = computation.SumOfSquares - computation.Sum *
computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}
```

在 Visual Studio 的重构支持下，可以轻松地将核心统计信息的功能提取到私有方法中。从而得到一个 private static 方法，该方法返回具有 Sum、SumOfSquares 和 Count 这三个值的元组类型：

C#

 复制

```
public static double StandardDeviation(IEnumerable<double> sequence)
{
    (int Count, double Sum, double SumOfSquares) computation =
ComputeSumAndSumOfSquares(sequence);

    var variance = computation.SumOfSquares - computation.Sum *
computation.Sum / computation.Count;
    return Math.Sqrt(variance / computation.Count);
}


private static (int Count, double Sum, double SumOfSquares)
ComputeSumAndSumOfSquares(IEnumerable<double> sequence)
{
    var computation = (count: 0, sum: 0.0, sumOfSquares: 0.0);

    foreach (var item in sequence)
    {
        computation.count++;
        computation.sum += item;
        computation.sumOfSquares += item * item;
    }

    return computation;
}
```

如果你想手动进行一些快速编辑，该语言可提供更多选项供你使用。首先，可以使用 var 声明来初始化 ComputeSumAndSumOfSquares 方法调用的元组结果。此外，还可以在


ComputeSumAndSumOfSquares 方法内创建三个离散变量。 下面的代码演示了最终版本：

C#	
<pre>public static double StandardDeviation(IEnumerable<double> sequence) { var computation = ComputeSumAndSumOfSquares(sequence); var variance = computation.SumOfSquares - computation.Sum * computation.Sum / computation.Count; return Math.Sqrt(variance / computation.Count); } private static (int Count, double Sum, double SumOfSquares) ComputeSumAndSumOfSquares(IEnumerable<double> sequence) { double sum = 0; double sumOfSquares = 0; int count = 0; foreach (var item in sequence) { count++; sum += item; sumOfSquares += item * item; } return (count, sum, sumOfSquares); }</pre>	

这个最终版本可用于任何需要这三个值或其任意子集的方法。

该语言支持其他用于管理这些元组返回方法中的元素名称的选项。

可以删除返回值声明中的字段名称，返回一个未命名元组：

C#	
<pre>private static (double, double, int) ComputeSumAndSumOfSquares(IEnumerable<double> sequence) { double sum = 0; double sumOfSquares = 0; int count = 0; foreach (var item in sequence) { count++; sum += item; sumOfSquares += item * item; } }</pre>	

```
        return (sum, sumOfSquares, count);
    }
```


此元组的字段被命名为 `Item1`、`Item2` 和 `Item3`。建议为从方法返回的元组的元素提供语义名称。

元组另一个常见的用途是在你创作 LINQ 查询时。最终投影的结果通常包含被选中的对象的某些（而不是全部）属性。

传统做法是将查询结果投影成一个匿名类型的对象序列。这种做法存在很多限制，主要是因为匿名类型无法在方法的返回类型中方便地命名。也可以将 `object` 或 `dynamic` 用作结果类型，但这种备用方法会产生高昂的性能成本。

返回一个元组类型序列非常简单，并且不管是在编译时还是通过 IDE 工具，都可以获取其元素的名称和类型。以 `ToDo` 应用程序为例。可以定义一个与下面类似的类，以表示待办事项列表中的某一项：


C#

 复制

```
public class ToDoItem
{
    public int ID { get; set; }
    public bool IsDone { get; set; }
    public DateTime DueDate { get; set; }
    public string Title { get; set; }
    public string Notes { get; set; }
}
```

移动应用程序可能支持当前待办事项的简洁版，即仅显示标题。该 LINQ 查询生成一个仅包含 ID 和标题的投影。返回一个元组序列的方法很好地表达了该设计：

C#

 复制

```
internal IEnumerable<(int ID, string Title)> GetCurrentItemsMobileList()
{
    return from item in AllItems
           where !item.IsDone
           orderby item.DueDate
           select (item.ID, item.Title);
}
```


❗ 备注

在 C# 7.1 中，通过元组投影可使用元素，以类似于在匿名类型中命名属性的方式创建命名元组。在以上代码中，查询投影中的 `select` 语句将创建具有元素 `ID` 和 `Title` 的元组。


命名元组可以是签名的一部分。它让编译器和 IDE 工具提供静态检查，看结果的用法是否正确。命名元组还承载了静态类型信息，因此无需使用高成本的运行时功能（如反射或动态绑定）来处理结果。

析构


通过对方法返回的元组进行析构，可以解封元组中的所有项。有三种元组析构方法。首先，可在括号内显式声明每个字段的类型，为元组中的每个元素创建离散变量：

C#	 复制
<pre>public static double StandardDeviation(IEnumerable<double> sequence) { (int count, double sum, double sumOfSquares) = ComputeSumAndSumOfSquares(sequence); var variance = sumOfSquares - sum * sum / count; return Math.Sqrt(variance / count); }</pre>	

也可以通过在括号外使用 `var` 关键字，隐式声明元组中每个字段的类型化变量：

C#	 复制
<pre>public static double StandardDeviation(IEnumerable<double> sequence) { var (sum, sumOfSquares, count) = ComputeSumAndSumOfSquares(sequence); var variance = sumOfSquares - sum * sum / count; return Math.Sqrt(variance / count); }</pre>	

还可以在括号内将 `var` 关键字与任意或全部变量声明结合使用。

C#	 复制
<pre>(double sum, var sumOfSquares, var count) = ComputeSumAndSumOfSquares(sequence);</pre>	

即使元组中的每个字段都具有相同的类型，也不能在括号外使用特定类型。

也可以使用现有声明析构元组：

C#	 复制
----	--

```
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }

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

⚠ 警告


不能混合现有声明和括号内的声明。例如，不允许以下内容：`(var x, y) = MyMethod();`。这将产生错误 CS8184，因为 `x` 在括号内声明，且 `y` 以前在其他位置声明。

析构用户定义类型

如上所示，可以析构任何元组类型。也可以对任何用户定义的类型（类、结构甚至接口）轻松启用析构。

类型作者可定义一个或多个赋值给任意数量的 `out` 变量的 `Deconstruct` 方法，这类变量表示构成该类型的数据元素。例如，以下 `Person` 类型定义 `Deconstruct` 方法，该方法将 `person` 对象析构成表示名字和姓氏的元素：

C#

 复制


```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}
```

该析构方法支持从 `Person` 赋值给两个表示 `FirstName` 和 `LastName` 属性的字符串：


C#

 复制

```
var p = new Person("Althea", "Goodwin");
var (first, last) = p;
```

即使对未创作的类型，也可以启用析构。Deconstruct 方法可以是一种扩展方法，用于解封对象的可访问数据成员。以下示例显示从 Person 类型派生的 Student 类型，以及将 Student 析构成三个变量（表示 FirstName、LastName 和 GPA）的扩展方法：

C#


 复制

```
public class Student : Person
{
    public double GPA { get; }
    public Student(string first, string last, double gpa) :
        base(first, last)
    {
        GPA = gpa;
    }
}

public static class Extensions
{
    public static void Deconstruct(this Student s, out string first, out
string last, out double gpa)
    {
        first = s.FirstName;
        last = s.LastName;
        gpa = s.GPA;
    }
}
```

Student 对象现在有两个可访问的 Deconstruct 方法：为 Student 类型声明的扩展方法，以及 Person 类型的成员。两者都可用，可将 Student 析构为两个或三个变量。如果为 student 分配三个变量，则返回名字、姓氏和 GPA。如果为 student 分配两个变量，则仅返回名字和姓氏。

C#

 复制

```
var s1 = new Student("Cary", "Totten", 4.5);
var (fName, lName, gpa) = s1;
```

在某个类或类层次结构中定义多个 Deconstruct 方法时应非常小心。具有相同数量的 out 参数的多个 Deconstruct 方法很快就会产生歧义。调用方可能无法轻松调用所需的 Deconstruct 方法。

在此示例中，发生有歧义的调用的几率很小，因为用于 Person 的 Deconstruct 方法有两个输出参数，而用于 Student 的 Deconstruct 方法有三个输出参数。


析构运算符不参与测试相等。下面的示例生成编译器错误 CS0019：

C#	 复制
<pre>Person p = new Person("Althea", "Goodwin"); if (("Althea", "Goodwin") == p) Console.WriteLine(p);</pre>	


Deconstruct 方法无法将 Person 对象 p 转换为包含两个字符串的元组，但它在相等测试上下文中不适用。

元组作为 out 参数

元组自身可用作 out 参数。不要与前面提到的[析构函数](#)部分中的任何多义性混淆。在方法调用中，只需描述元组的形状：

C#	 复制
<pre>Dictionary<int, (int, string)> dict = new Dictionary<int, (int, string)>(); dict.Add(1, (234, "First!")); dict.Add(2, (345, "Second")); dict.Add(3, (456, "Last")); // TryGetValue already demonstrates using out parameters dict.TryGetValue(2, out (int num, string place) pair); Console.WriteLine(\$"{pair.num}: {pair.place}"); /* * Output: * 345: Second */</pre>	

另外，还可以使用 [unnamed](#) 元组，并将其字段作为 Item1 和 Item2 引用：

C#	 复制
<pre>dict.TryGetValue(2, out (int, string) pair); // ... Console.WriteLine(\$"{pair.Item1}: {pair.Item2}");</pre>	

结束语

命名元组的新语言和库支持简化了设计工作：与类和结构一样，使用数据结构存储多个元素，但不定义行为。对这些类型使用元组非常简单明了。既可以获得静态类型检查的

所有好处，又不需要使用更复杂的 `class` 或 `struct` 语法来创作类型。即便如此，元组还是对 `private` 或 `internal` 这样的实用方法最有用。当公共方法返回具有多个元素的值时，请创建用户定义的类型（`class` 或 `struct` 类型）。

此页面有帮助吗？

 是  否
