C#历来语法特性总结

C# 11已与.NET 7一起发布，本文按照.NET的发布顺序，根据微软官方文档整理C#中一些有趣的语法特性。

**注：**基于不同.NET平台创建的项目，默认支持的C#版本是不一样的。下面介绍的语法特性，会说明引入C#的版本，在使用过程中，需要注意使用C#的版本是否支持对应的特性。C#语言版本控制，可参考官方文档。

**匿名函数**

匿名函数是C# 2推出的功能，顾名思义，匿名函数只有方法体，没有名称。匿名函数使用delegate创建，可转换为委托。匿名函数不需要指定返回值类型，它会根据return语句自动判断返回值类型。

**注：**C# 3后推出了lambda表达式，使用lambda可以以更简洁的方式创建匿名函数，应尽量使用lambda来创建匿名函数。与lambda不同的是，使用delegate创建匿名函数可以省略参数列表，可将其转换为具有任何参数列表的委托类型。

// 使用delegate关键字创建，无需指定返回值，可转换为委托，可省略参数列表（与lambda不同）

Func<int, bool> func = delegate { return true; };

**自动属性**

从C# 3开始，当属性访问器中不需要其它逻辑时，可以使用自动属性，以更简洁的方式声明属性。编译时，编译器会为其创建一个仅可以通过get、set访问器访问的私有、匿名字段。使用VS开发时，可以通过snippet代码片段prop+2次tab快速生成自动属性。

// 属性老写法

private string \_name;

public string Name

{

get { return \_name; }

set { \_name = value; }

}

// 自动属性

public string Name { get; set; }

另外，在C# 6以后，可以初始化自动属性：

public string Name { get; set; } = "Louzi";

**匿名类型**

匿名类型是C# 3后推出的功能，它无需显示定义类型，将一组只读属性封装到单个对象中。编译器会自动推断匿名类型的每个属性的类型，并生成类型名称。从CLR的角度看，匿名类型与其它引用类型没什么区别，匿名类型直接派生自object。如果两个或多个匿名对象指定了顺序、名称、类型相同的属性，编译器会把它们视为相同类型的实例。在创建匿名类型时，如果不指定成员名称，编译器会把用于初始化属性的名称作为属性名称。

匿名类型多用于LINQ查询的select查询表达式。匿名类型使用new与初始化列表创建：

// 使用new与初始化列表创建匿名类型

var person = new { Name = "Louzi", Age = 18 };

Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");

// 用于LINQ

var productQuery =

from prod in products

select new { prod.Color, prod.Price };

foreach (var v in productQuery)

{

Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);

}

**LINQ**

C# 3推出了杀手锏功能，查询表达式，即语言集成查询（LINQ）。查询表达式以查询语法表示查询，由一组类似SQL的语法编写的子句组成。

查询表达式必须以from子句开头，必须以select或group子句结尾。在第一个from子句与最后一个select或group子句之间，可以包含：where、orderby、join、let、其它from子句等。

可以为SQL[数据库](https://cloud.tencent.com/solution/database?from_column=20065&from=20065)、XML文档、ADO.NET数据集及实现了IEnumerable或IEnumerable接口的集合对象进行LINQ查询。

完整的查询包括创建数据源、定义查询表达式、执行查询。查询表达式变量是存储查询而不是查询结果，只有在循环访问查询变量后，才会执行查询。

可使用查询语法表示的任何查询都可以使用方法表示，建议使用更易读的查询语法。有些查询操作（如 Count 或 Max）没有等效的查询表达式子句，必须使用方法调用。可以结合使用方法调用和查询语法。

关于LINQ的详细文档，参见微软官方文档

// Data source.

int[] scores = { 90, 71, 82, 93, 75, 82 };

// Query Expression.

IEnumerable<int> scoreQuery = //query variable

from score in scores //required

where score > 80 // optional

orderby score descending // optional

select score; //must end with select or group

// Execute the query to produce the results

foreach (int testScore in scoreQuery)

{

Console.WriteLine(testScore);

}

**Lambda**

C# 3推出了很多强大的功能，如自动属性、扩展方法、隐式类型、LINQ，以及Lambda表达式。

创建Lambda表达式，需要在 => 左侧指定输入参数（空括号指定零个参数，一个参数可以省略括号），右侧指定表达式或语句块（通常两三条语句）。任何Lambda表达式都可以转换为委托类型，表达式Lambda语句还可以转换为表达式树（语句Lambda不可以）。

匿名函数可以省略参数列表，Lambda中不使用的参数可以使用弃元指定（C# 9）。

使用async和await，可以创建包含异步处理的Lambda表达式和语句（C# 5）。

从C# 10开始，当编译器无法推断返回类型时，可以在参数前面指定Lambda表达式的返回类型，此时参数必须加括号。

// Lambda转换为委托

Func<int, int> square = x => x \* x;

// Lambda转换为表达式树

System.Linq.Expressions.Expression<Func<int, int>> e = x => x \* x;

// 使用弃元指定不使用的参数

Func<int, int, int> constant = (\_, \_) => 42;

// 异步Lambda

var lambdaAsync = async () => await JustDelayAsync();

Console.WriteLine($"main thread id: {Thread.CurrentThread.ManagedThreadId}");

lambdaAsync();

static async Task JustDelayAsync()

{

await Task.Delay(1000);

Console.WriteLine($"JustDelayAsync thread id: {Thread.CurrentThread.ManagedThreadId}");

}

// 指定返回类型，不指定返回类型会报错

var choose = object (bool b) => b ? 1 : "two";

**扩展方法**

扩展方法也是C# 3推出的功能，它能够向现有类型添加方法，且无需修改原始类型。扩展方法是一种静态方法，不过是通过实例对象语法进行调用，它的第一个参数指定方法操作的类型，用this修饰。编译器在编译为IL时会转换为静态方法的调用。

如果类型中具有与扩展方法相同名称和签名的方法，则编译器会选择类型中的方法。编译器进行方法调用时，会先在该类型的的实例方法中寻找，找不到再去搜索该类型的扩展方法。

最常见的扩展方法是LINQ，它将查询功能添加到现有的System.Collections.IEnumerable和System.Collections.Generic.IEnumerable类型中。

为struct添加扩展方法时，由于是值传递，只能对struct对象的副本进行更改。从C# 7.2开始，可以为第一个参数添加ref修饰以进行引用传递，这样就可以对struct对象本身进行修改了。

static class MyExtensions

{

public static void OutputStringExtension(this string s) => Console.WriteLine($"output: {s}");

public static void OutputPointExtension(this Point p)

{

p.X = 10;

p.Y = 10;

Console.WriteLine($"output: ({p.X}, {p.Y})");

}

public static void OutputPointWithRefExtension(ref this Point p)

{

p.X = 20;

p.Y = 20;

Console.WriteLine($"output: ({p.X}, {p.Y})");

}

}

// class扩展方法

"Louzi".OutputStringExtension();

// struct扩展方法

Point p = new Point(5, 5);

p.OutputPointExtension(); // output: (10, 10)

Console.WriteLine($"original point: ({p.X}, {p.Y})"); // output: (5, 5)

p.OutputPointWithRefExtension(); // output: (20, 20)

Console.WriteLine($"original point: ({p.X}, {p.Y})"); // output: (20, 20)

**隐式类型（var）**

从C# 3开始，在方法范围内可以声明隐式类型变量（var）。隐式类型为强类型，由编译器决定类型。

var常用于调用构造函数创建对象实例时，从C# 9开始，这种场景也可以使用确定类型的new表达式：

// 隐式类型

var s = new List<int>();

// new表达式

List<int> ss = new();

**注：**当返回匿名类型时，只能使用var。

**对象、集合初始化列表**

从C# 3开始，可以在单条语句中实例化对象或集合并执行成员分配。

使用对象初始化列表，可以在创建对象时向对象的任何可访问字段或属性分配值，可以指定构造函数参数或忽略参数以及括号。

public class Person

{

// 自动属性

public int Age { get; set; }

public string Name { get; set; }

public Person() { }

public Person(string name)

{

Name = name;

}

}

var p1 = new Person { Age = 18, Name = "Louzi" };

var p2 = new Person("Sherilyn") { Age = 18 };

从C# 6开始，对象初始化列表不仅可以初始化可访问字段和属性，还可以设置索引器。

public class MyIntArray

{

public int CurrentIndex { get; set; }

public int[] data = new int[3];

public int this[int index]

{

get => data[index];

set => data[index] = value;

}

}

var myArray = new MyIntArray { [0] = 1, [1] = 3, [2] = 5, CurrentIndex = 0 };

集合初始化列表可以指定一个或多个初始值：

var persons = new List<Person>

{

new Person { Age = 18, Name = "Louzi" },

new Person { Age = 18, Name = "Sherilyn" }

};

**内置泛型委托**

.NET Framework 3.5/4.0，分别提供了内置的Action和Func泛型委托类型。void返回类型的委托可以使用Action类型，Action的变体最多有16个参数。有返回值类型的委托可以使用Func类型，Func类型的变体最多同样16个参数，返回类型为Func声明中的最后一个类型参数。

Action<int> actionInstance = ActionInstance;

Func<int, string> funcInstance = FuncInstance;

static void ActionInstance(int n) => Console.WriteLine($"input: {n}");

static string FuncInstance(int n) => $"param: {n}";

**dynamic**

C# 4主要的功能就是引入了dynamic关键字。dynamic类型在变量使用及其成员引用时会绕过编译时类型检查，在运行时再进行解析。这便实现了与动态类型语言（如JavaScript）类似的构造。

dynamic dyn = 1;

Console.WriteLine(dyn.GetType()); // output: System.Int32

dyn = dyn + 3; // 如果dyn是object类型，此句则会报错

**命名参数与可选参数**

C# 4引入了命名参数和可选参数。命名参数可为形参指定实参，方式是指定匹配的实参与形参，这时无需匹配参数列表中的位置。可选参数通过指定参数默认值，可以省略实参。可选参数需位于参数列表末尾，如果为一系列可选参数中的任意一个提供了实参，则必须为该参数前面的所有可选参数提供实参。

也可以使用OptionalAttribute特性声明可选参数，此时无需为形参提供默认值。

// 命名参数与可选参数

PrintPerson(age: 18, name: "Louzi");

// static void PrintPerson(string name, int age, [Optional, DefaultParameterValue("男")] string sex)

static void PrintPerson(string name, int age, string sex = "男") =>

Console.WriteLine($"name: {name}, age: {age}, sex: {sex}");

**静态导入**

C# 6中推出了静态导入功能，使用using static指令导入类型，可以无需指定类型名称即可访问其静态成员和嵌套类型，这样避免了重复输入类型名称导致的晦涩代码。

using static System.Console;

WriteLine("Hello CSharp");

**异常筛选器(when)**

从C# 6开始，when可用于catch语句中，用来指定为执行特定异常处理程序必须为true的条件表达式，当表达式为false时，则不会执行异常处理。

public static async Task<string> MakeRequest()

{

var client = new HttpClient();

var streamTask = client.GetStringAsync("https://localHost:10000");

try

{

var responseText = await streamTask;

return responseText;

}

catch (HttpRequestException e) when (e.Message.Contains("301"))

{

return "Site Moved";

}

catch (HttpRequestException e) when (e.Message.Contains("404"))

{

return "Page Not Found";

}

catch (HttpRequestException e)

{

return e.Message;

}

}

**自动属性初始化表达式**

C# 6开始，可以为自动属性指定初始化值以使用类型默认值以外的值：

public class DefaultValueOfProperty

{

public string MyProperty { get; set; } = "Property";

}

**表达式体**

从C# 6起，支持方法、运算符和只读属性的表达式体定义，自C# 7.0起，支持构造函数、终结器、属性、索引器的表达式体定义。

static void NewLine() => Console.WriteLine();

**null条件运算符**

C# 6起，推出了null条件运算符，仅当操作数的计算结果为非null时，null条件运算符才会将成员访问?.或元素访问?[]运算应用于其操作数；否则，将返回null。

// null条件表达式

public class ConditionalNull

{

event EventHandler AEvent;

public void RaiseAEvent() => AEvent?.Invoke(this, EventArgs.Empty);

}

**内插字符串**

从C# 6开始，可以使用You can't use 'macro parameter character #' in math mode"..."形式的内插逐字字符串，在此之前的版本，必须使用

Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");

**nameof**

C# 6提供了nameof表达式，nameof可生成变量、类型或成员名称（非完全限定）作为字符串常量。

public string Name

{

get => name;

set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");

}

**out改进**

C# 7.0中对out语法进行了改进，可以直接在方法调用的参数列表中声明out变量，无需再单独编写一条声明语句：代码语言：javascript

复制

void Function(out int arg) { ... }

// 未改进前

int n;

Function(out n);

// 改进后

Function(out int n);

**元组**

C# 7.0中引入了对元组的语言支持（之前版本也有元组但效率低下），可以使用元组表示包含多个数据的简单结构，无需再专门写一个class或struct。元组是值类型的，是包含多个公共字段以表示数据成员的轻量级数据结构，无法为其定义方法。C# 7.3后元组支持==与!=。

// 方式一，使用元组字段的默认名称：Item1、Item2、Item3等

(string, string) unnamedLetters = ("a", "b");

Console.WriteLine($"{unnamedLetters.Item1}, {unnamedLetters.Item2}");

// 方式二

(string Alpha, string Beta) namedLetters = ("a", "b");

Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");

// 方式三

var alphabetStart = (Alpha: "a", Beta: "b");

Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");

// 方式四，C# 7.1开始支持自动推断变量名称

int count = 5;

string label = "Colors used in the map";

var pair = (count, label); // 元组元素名为"count"和"label"

当某方法返回元组时，如需提取元组成员，可通过为元组的每个值声明单独的变量来实现，称为解构元组。使用元组作为方法返回类型，可以替代定义out方法参数。

// 解构元组

var (first, last) = Range(numbers);

Console.WriteLine($"{first} to {last}");

(int max, int min) = Range(numbers);

Console.WriteLine($"{min} to {max}");

**弃元**

从C# 7.0开始支持弃元，弃元是占位符变量，相当于未赋值的变量，表示不想使用该变量，使用下划线\_表示弃元变量。如下列举了一些弃元的使用场景：

// 场景一：丢弃元组值

(\_, \_, area) = city.GetCityInformation(cityName);

// 场景二：从C# 9开始，可以丢弃Lambda表达式中的参数

Func<int, int, int> constant = (\_, \_) => 42;

// 场景三，丢弃out参数

DiscardsOut(out \_);

static void DiscardsOut(out string s)

{

s = "nothing";

Console.WriteLine($"input is {s}");

}

**模式匹配**

C# 7.0添加了模式匹配功能，之后每个主要C#版本都扩展了模式匹配功能。模式匹配用来测试表达式是否具有某些特征，is表达式、switch语句和switch表达式均支持模式匹配，可使用when关键字来指定模式的其他规则。

模式匹配目前包含这些类型：声明模式、类型模式、常量模式、关系模式、逻辑模式、属性模式、位置模式、var模式、弃元模式，详细内容可参考官方文档。

is模式表达式改进了is运算符功能，可在一条指令分配结果：

// is模式匹配

if (input is int count) do somthing... ;

// 老写法

if (input is int)

{

int count = (int)input;

do somthing... ;

}

// is模式进行空检查

string? message = "This is not the null string";

if (message is not null) Console.WriteLine(message);

**default文本表达式**

默认值表达式生成类型的默认值，之前版本仅支持default运算符，C# 7.1后增强了default表达式的功能，当编译器可以推断表达式类型时，可以使用default生成类型的默认值。

// 新写法

Func<string, bool> whereClause = default;

// 老写法

Func<string, bool> whereClause = default(Func<string, bool>);

**switch表达式**

从C# 8开始，可以使用switch表达式。switch表达式相较于switch语句的改进之处在于：

* 变量在switch关键字之前；
* 使用=>替换case :结构；
* 使用弃元\_替换default运算符；
* 使用表达式替换语句。

public enum Level

{

One,

Two,

Three

}

public static int LevelToScore(Level level) => level switch

{

Level.One => 1,

Level.Two => 5,

Level.Three => 10,

\_ => throw new ArgumentOutOfRangeException(nameof(level), $"Not expected level value: {level}"),

};

**using声明**

C# 8添加了using声明功能，它指示编译器声明的变量应在代码块的末尾进行处理。using声明相比传统的using语句代码更简洁，这两种写法都会使编译器在代码块末尾调用Dispose()。

static void WriteLinesToFile(IEnumerable<string> lines)

{

using var file = new System.IO.StreamWriter("WriteLines.txt");

do somthing... ;

return;

// file is disposed here

}

**索引和范围**

C# 8中添加了索引和范围功能，为访问序列中的单个元素或范围提供了简洁的语法。该语法依赖两个新类型与两个新运算符：

* System.Index表示一个序列索引；
* System.Range表示序列的子范围；
* 末尾运算符^，使用该运算符加数字，指定倒数第几个；
* 范围运算符..，指定范围的开始和末尾。

范围运算符包括此范围的开始，但不包括此范围的末尾。

var words = new string[]

{ // 正常索引 索引对应的末尾运算符

"The", // 0 ^9

"quick", // 1 ^8

"brown", // 2 ^7

"fox", // 3 ^6

"jumped", // 4 ^5

"over", // 5 ^4

"the", // 6 ^3

"lazy", // 7 ^2

"dog" // 8 ^1

}; // 9 (words.Length) ^0

Console.WriteLine($"The last word is {words[^1]}"); // dog

var allWords = words[..]; // 包含所有值，等同于words[0..^0].

var firstPhrase = words[..4]; // 开始到words[4]，不包含words[4]

var lastPhrase = words[6..]; // words[6]到末尾

// 声明范围变量

Range phrase = 1..4;

var text = words[phrase];

**??与??=**

??合并运算符：C# 6后可用，如果左操作数的值不为null，则??返回该值；否则，它会计算右操作数并返回其结果。如果左操作数的计算结果为非null，则不会计算其右操作数。

??=合并赋值运算符：C# 8后可用，仅在左侧操作数的求值结果为null时，才将右操作数的值赋值给左操作数。否则，不会计算其右操作数。??=运算符的左操作数必须是变量、属性或索引器元素。

// ??合并运算符

Console.WriteLine($"name is {OutputName(null)}");

static string OutputName(string name) => name ?? "some one";

// 使用??=赋值运算符

variable ??= expression;

// 老写法

if (variable is null)

{

variable = expression;

}

**顶级语句**

C# 9推出了顶级语句，它从应用程序中删除了不必要的流程，应用程序中只有一个文件可使用顶级语句。顶级语句使主程序更易读，减少了不必要的模式：命名空间、class Program和static void Main()。

使用VS创建命令行项目，选择.NET 5及以上版本，就会使用顶级语句。

// 使用VS2022创建.NET 6.0平台的命令行程序默认生成的内容

// See https://aka.ms/new-console-template for more information

Console.WriteLine("Hello, World!");

**global using**

C# 10添加了global using指令，当关键字global出现在using指令之前时，该using适用于整个项目，这样可以减少每个文件using指令的行数。global using 指令可以出现在任何源代码文件的开头，但需添加在非全局using之前。

global修饰符可以与static修饰符一起使用，也可以应用于using别名指令。在这两种情况下，指令的作用域都是当前编译中的所有文件。

global using System;

global using static System.Console; // 全局静态导入

global using Env = System.Environment; // 全局别名

**文件范围的命名空间**

C# 10引入了文件范围的命名空间，可将命名空间包含为语句，后加分号且无需添加大括号。一个代码文件通常只包含一个命名空间，这样简化了代码且消除了一层嵌套。文件范围的命名空间不能声明嵌套的命名空间或第二个文件范围的命名空间，且它必须在声明任何类型之前，该文件内的所有类型都属于该命名空间。这个特性不推荐.容易混淆.

using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);

**with表达式**

C# 9开始引入了with表达式，它使用修改的特定属性和字段生成其操作对象的副本，未修改的值将保留与原对象相同的值。对于引用类型成员，在复制操作数时仅复制对该成员实例的引用，with表达式生成的副本和原对象都具有对同一引用类型实例的访问权限。

在C# 9中，with表达式的左操作数必须为record类型，C# 10进行了改进，with表达式的左操作数也可以是struct类型。

public record NamedPoint(string Name, int X, int Y);

var p1 = new NamedPoint("A", 0, 0);

var p2 = p1 with { Name = "B", X = 5 };

**推荐阅读：**

[.NET多线程下解决资源竞争的7种方法](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fmp.weixin.qq.com%2Fs%3F__biz%3DMzU4MzIyNDA0MQ%3D%3D%26mid%3D2247515704%26idx%3D1%26sn%3D4772fd8bd44678732fe7906311c80277%26chksm%3Dfdaea1becad928a80d451625750dff60f098060d5d1f75fe88da320e5ae93454d6dcdd4e12ec%26scene%3D21%23wechat_redirect&objectId=2330753&objectType=1)

[摸鱼神器，使用 C# 和 WPF 开发的统计小工具 Tai](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fmp.weixin.qq.com%2Fs%3F__biz%3DMzU4MzIyNDA0MQ%3D%3D%26mid%3D2247515668%26idx%3D1%26sn%3D2bf77b405533551f4fff81d48040b2dd%26chksm%3Dfdaea192cad928841319c1f6a03dc368852bdf0f26a4c0dee73d400c2b0a0d43d43ec9c852dc%26scene%3D21%23wechat_redirect&objectId=2330753&objectType=1)

[.NET开源最全的第三方登录整合库 - CollectiveOAuth](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fmp.weixin.qq.com%2Fs%3F__biz%3DMzU4MzIyNDA0MQ%3D%3D%26mid%3D2247515634%26idx%3D1%26sn%3D3fd6e4f4ff1b922b4dcc5c6a3123bdbb%26chksm%3Dfdaea274cad92b6275003a7feca7851dd1fee72a3a9ebd045d1dbfcd4d8a7d00a380503d2adc%26scene%3D21%23wechat_redirect&objectId=2330753&objectType=1)

[【对.NET系统架构改造的一点经验和教训】的技术要点的看法](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fmp.weixin.qq.com%2Fs%3F__biz%3DMzU4MzIyNDA0MQ%3D%3D%26mid%3D2247515612%26idx%3D1%26sn%3Df28c3326c89af1fdf61d916e0c473a8e%26chksm%3Dfdaea25acad92b4cc11deb92480f35b4a72ec2aeb56829b9c2e761f52953cd3694bbf72ebdc0%26scene%3D21%23wechat_redirect&objectId=2330753&objectType=1)

[对.NET系统架构改造的一点经验和教训](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fmp.weixin.qq.com%2Fs%3F__biz%3DMzU4MzIyNDA0MQ%3D%3D%26mid%3D2247515537%26idx%3D1%26sn%3D73130970dde9c19a0673320fa9d644bc%26chksm%3Dfdaea217cad92b0173346b447e261c9b3db7ec54ceeb53941632966465846224f20b2a53a292%26scene%3D21%23wechat_redirect&objectId=2330753&objectType=1)

[一个.NET 7 + DDD + CQRS +React+Vite的实战项目](https://cloud.tencent.com/developer/tools/blog-entry?target=http%3A%2F%2Fmp.weixin.qq.com%2Fs%3F__biz%3DMzU4MzIyNDA0MQ%3D%3D%26mid%3D2247515513%26idx%3D1%26sn%3D9053b2b6bd5719db8741ac0ce775cf64%26chksm%3Dfdaea2ffcad92be94d52aeb2b43aadc22b868f790427ce19b4a06c4cc075e385997304689e4c%26scene%3D21%23wechat_redirect&objectId=2330753&objectType=1)